Retours sur la conception
UML
Classes et Séquences

Licence Professionnelle DAM
2015-2016

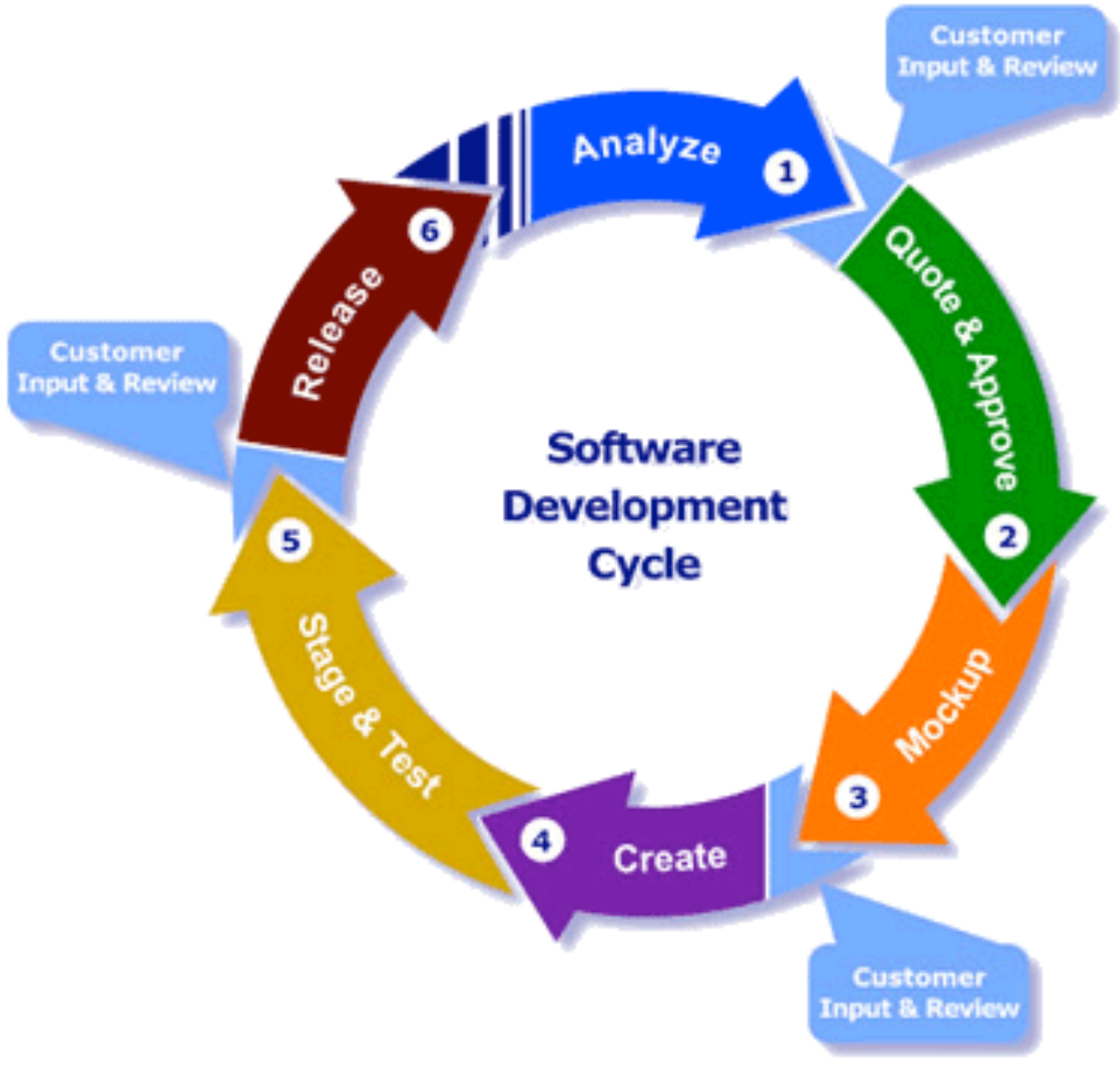
IUT de Nice

Simon Urli
urli@i3s.unice.fr

Inspiré des cours de Mireille Blay-Fornarino

Retour sur la conception logicielle

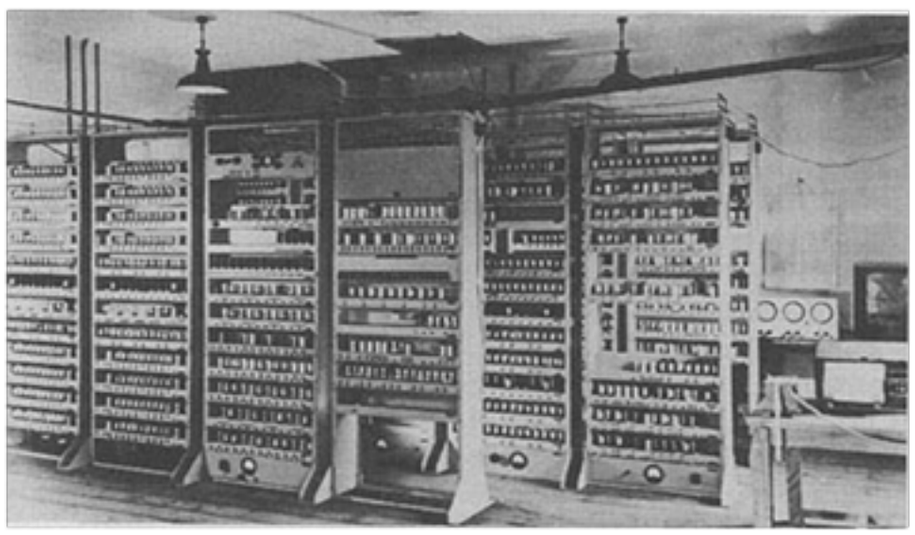
Activités du développement logiciel



Histoire de la gestion de la complexité dans le développement logiciel

Premiers programmes : langage machine

- Dépendants de l'architecture des ordinateurs
- Un jeu d'instruction = un type de machine
- Portage => réécriture du programme



```
sub     $sp, $sp, 4
sw      r,0($sp)
```

Création des premiers langages structurés

- Structures de contrôles (if, while, do, goto...)
- Séparer les concepts du langage de leur représentation en langage machine
- Pouvoir écrire un seul programme pour plusieurs architectures distinctes

```
      PROGRAM DEGRAD
!
! Imprime une table de conversion degrés -> radians
! ================================================
!
! Déclaration des variables
      INTEGER DEG
      REAL RAD, COEFF
!
! En-tête de programme
      WRITE ( *, 10)
   10 FORMAT      (' ',20('*') /              &
     &             ' * Degres * Radians *' /  &
     &             ' ', 20('*') )
!
! Corps de programme
      COEFF = (2.0 * 3.1416) / 360.0
      DO DEG = 0, 90
         RAD = DEG * COEFF
         WRITE ( *, 20) DEG, RAD
   20 FORMAT         (' *  ',I4,'  * ',F7.5,' *')
      END DO
!
! Fin du tableau
      WRITE ( *, 30)
   30 FORMAT      (' ',20('*') )
!
! Fin de programme
      STOP
      END PROGRAM DEGRAD
```

Développement par décomposition

- Travail sur la structure des programmes pour éviter la programmation spaghetti
- Evolution des langages pour pouvoir définir des modules fonctionnelles dans les programmes
- Stratégie de type «diviser pour régner»

Explosion des besoins

- Explosion de la miniaturisation électronique
- Chûte des coûts du matériel informatique
- L'informatique est devenu courant à la fois en entreprise et chez les particuliers
- Explosion des besoins logiciels dans tous les domaines
- Comment créer plus vite et mieux plus de logiciels plus complexes ?

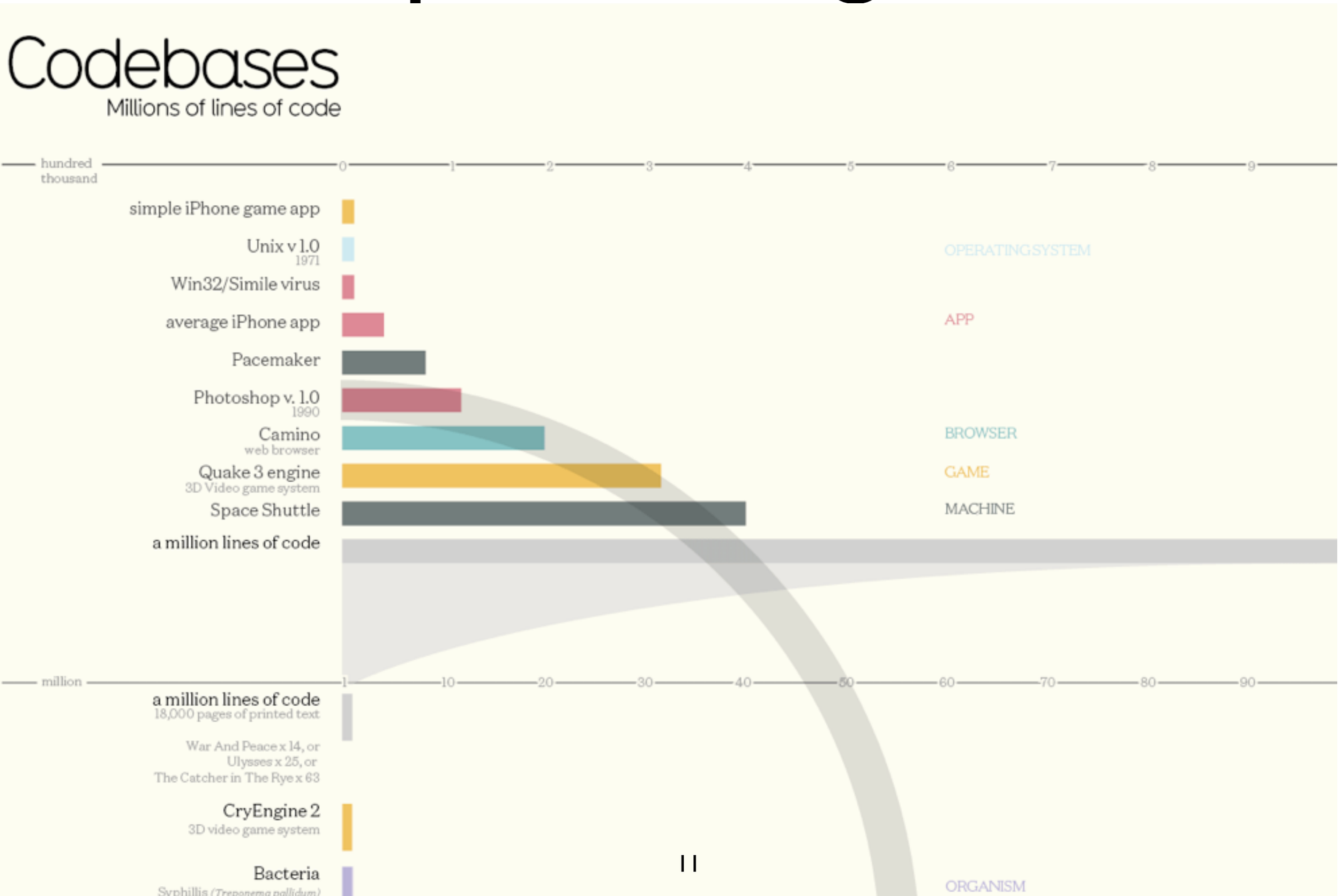
Avènement du paradigme objet

- Montée en abstraction : évolution des langages pour pouvoir définir la notion d'objets
- Définition des notions de contrats, de composants, de services, de frameworks
- Automatisation via la génération de code à partir d'abstractions
- Ingénierie des modèles

Et après ???

- Recherche en génie logiciel
- Vers des usines de logiciels ?
- Travail sur des logiciels qui réutilisent des logiciels ?
- La difficulté de la gestion de l'évolution !
- Abstraire plus ? Concilier abstraction et spécification du domaine ?

Complexité logicielle ?



Echecs logiciels

- 1985-87 : Therac 25 - Appareil de radiothéraphie. Bug dans le logiciel d'adaptation des radiations => Plusieurs morts
- 1990 : Crash du réseau AT&T pendant 9 heures : une seule ligne de code en faute
- 1996 : Interruption de la mission Ariane 5 : Problème d'encodage des entiers.
- Et d'autres !

Diviser pour résoudre

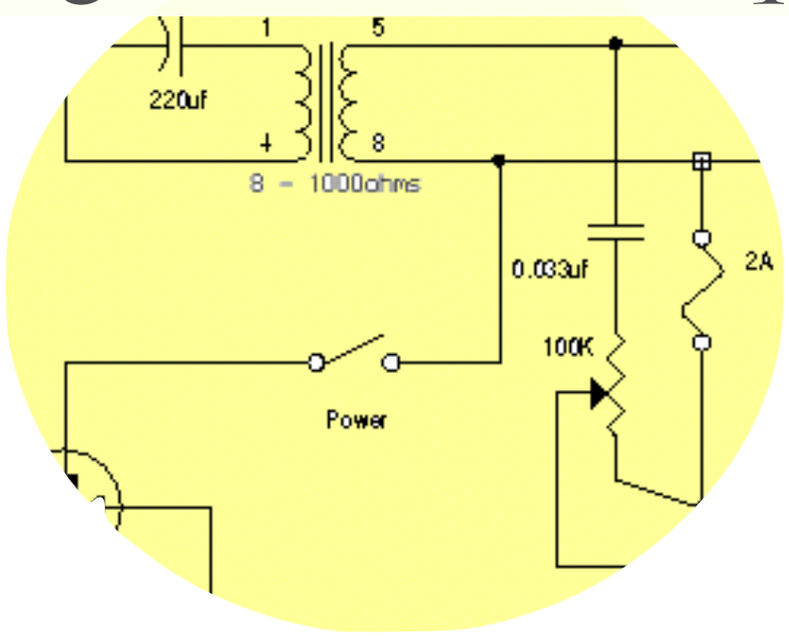
- Techniquement en s'appuyant sur la modularité :
  - Encapsulation, faible couplage
  - Notion d'objets
- En s'organisant au delà du seul développement :
  - Analyse et conception
  - Validation et vérification

Développer dans la durée grâce à la modélisation

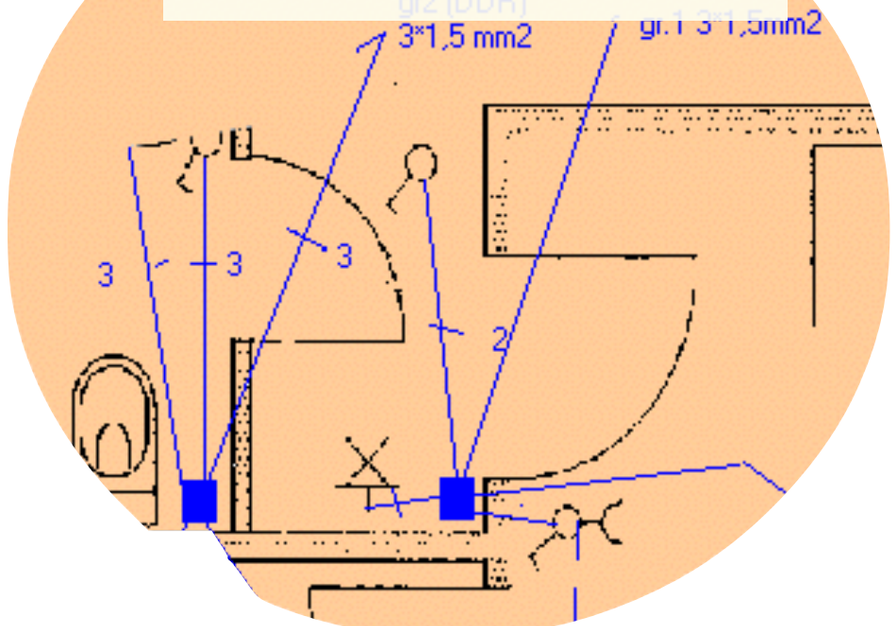
- Techniquement :
  - Assurer une meilleure continuité entre le «domaine du problème» et le «domaine des solutions»
  - Définir les fonctionnalités de manière à prévoir les tests et à considérer tout de suite la maintenance
- Du point de vue de l'organisation :
  - Tracer les changements
  - Gérer l'évolution
  - Utiliser des méthodes agiles

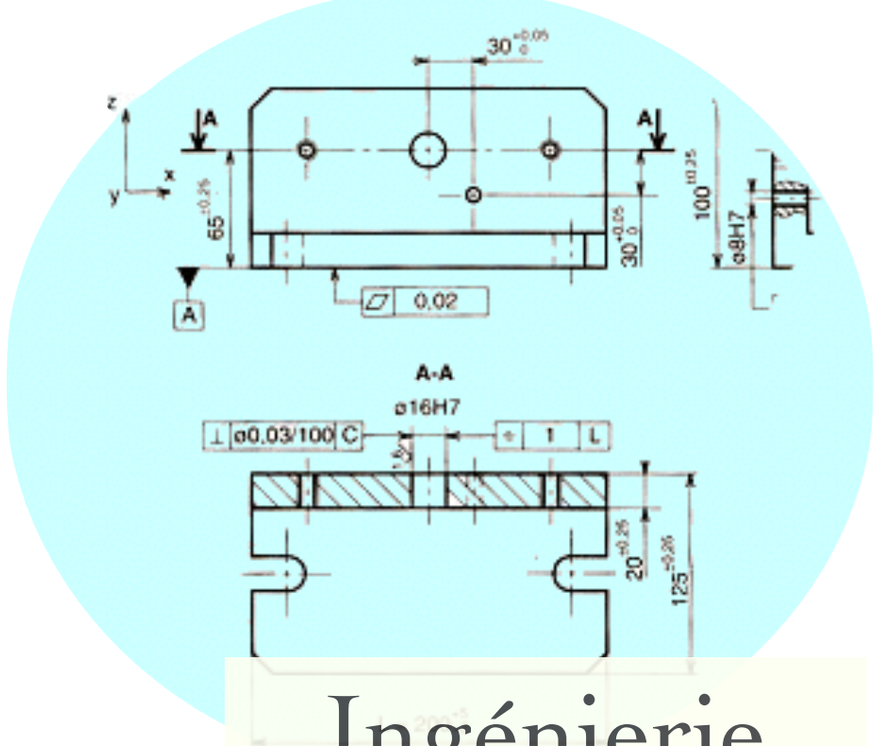
Retours sur UML

Ingénierie Électrique

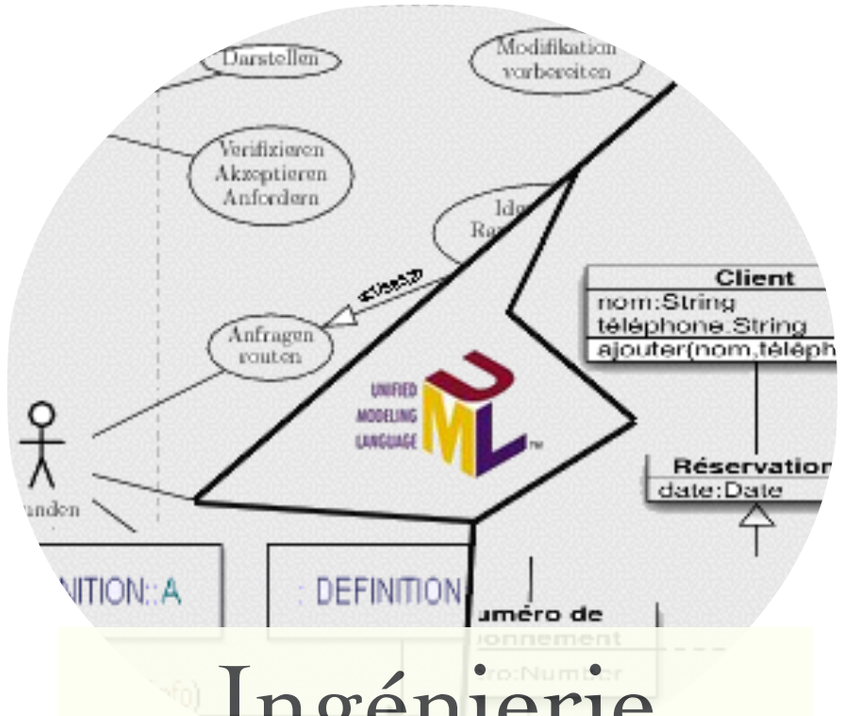


Ingénierie du Bâtiment





Ingénierie Mécanique

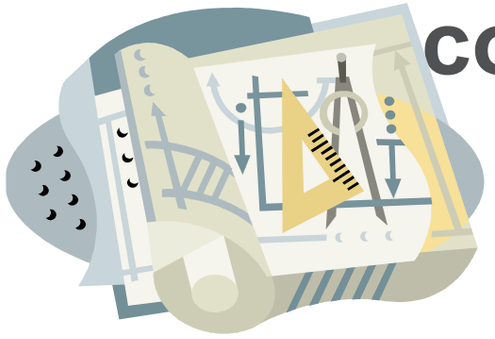


Ingénierie Logicielle

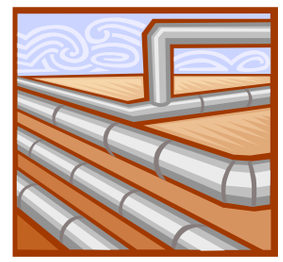
UML : un support à la modélisation

- Modèle : simplification de la réalité dans le but de :

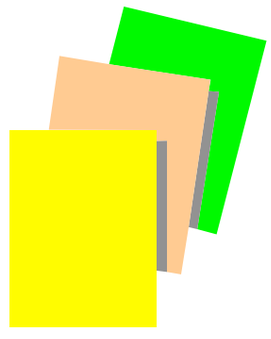
Visualiser le système



Spécifier la **structure et le comportement** du système



Aider à la construction du système



Documenter les décisions

Qu'est-ce qu'UML ?

- UML est un **langage visuel**
- Il supporte :
  - la visualisation
  - la spécification
  - la construction
  - la documentation

- Descriptions graphiques et textuelles

- Syntaxe et sémantique

- Architecture et comportement

- Génération de code

Points forts d'UML

- UML est un langage **normalisé**
  - Gain de précision
  - Gage de stabilité
  - Encourage l'utilisation d'outils
- UML est un support de communication performant
  - Permet de cadrer l'analyse
  - Facilite la compréhension de systèmes complexes
  - Langage universel en ingénierie logiciel

Points faibles d'UML

- Nécessite une période d'apprentissage et d'adaptation
- Beaucoup de concepts dans UML
- Ne suffit pas en soi à réussir un projet : le processus de développement, la qualité du code, etc sont nécessaires
- Toujours pas la panacée universelle !

D'où la création de DSL (langages spécifiques à UN domaine)… mais hors scope de ce cours !

Introduction à UML

survol

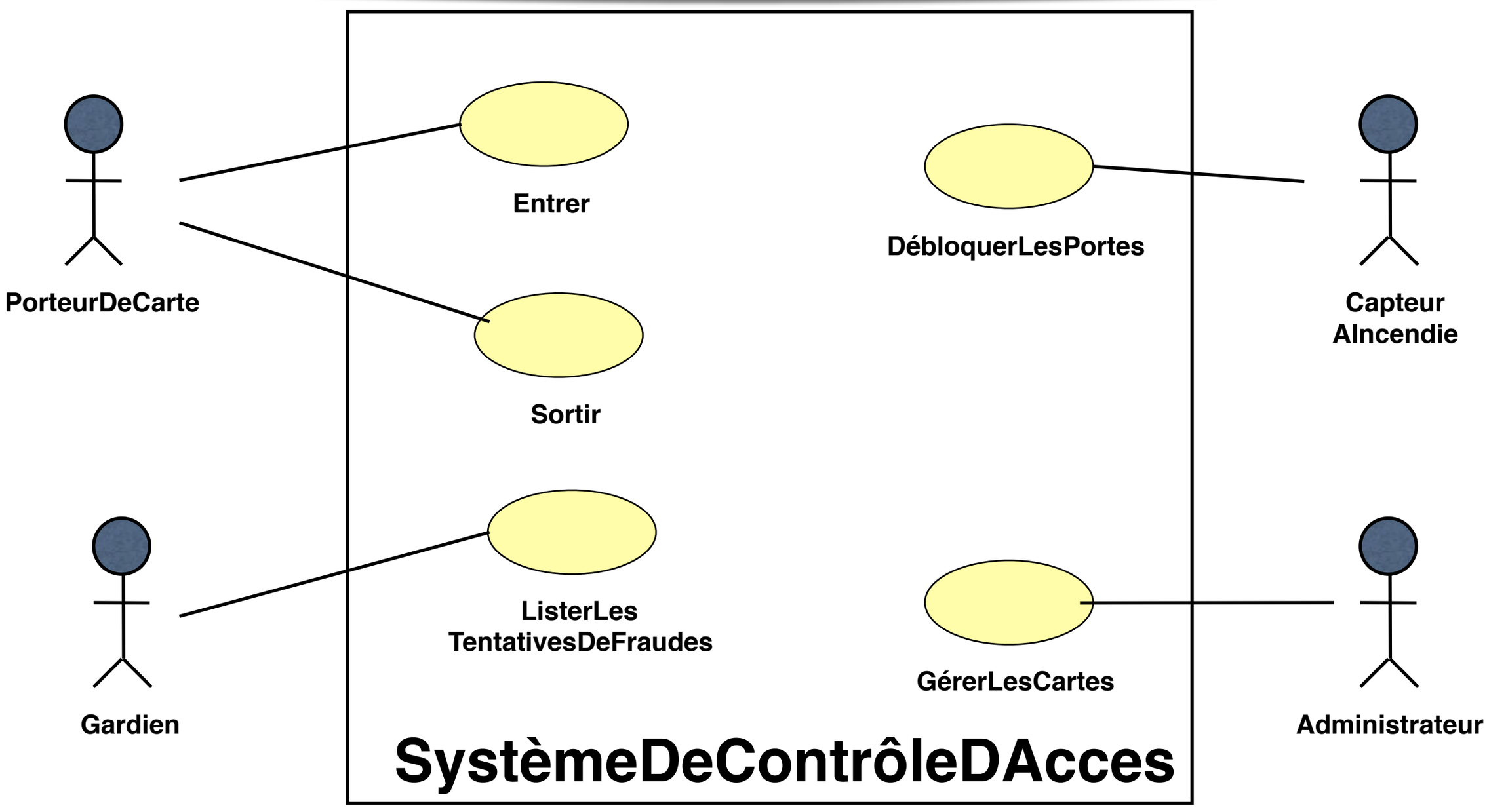
Vue fonctionnelle

la vue fonctionnelle cherche à appréhender
les interactions entre les
acteurs/utilisateurs
et le système,
sous forme d'objectifs à atteindre (**cas d'utilisation**) et sous forme chronologique de scénarios d'interaction typiques (**diagrammes de séquences**)

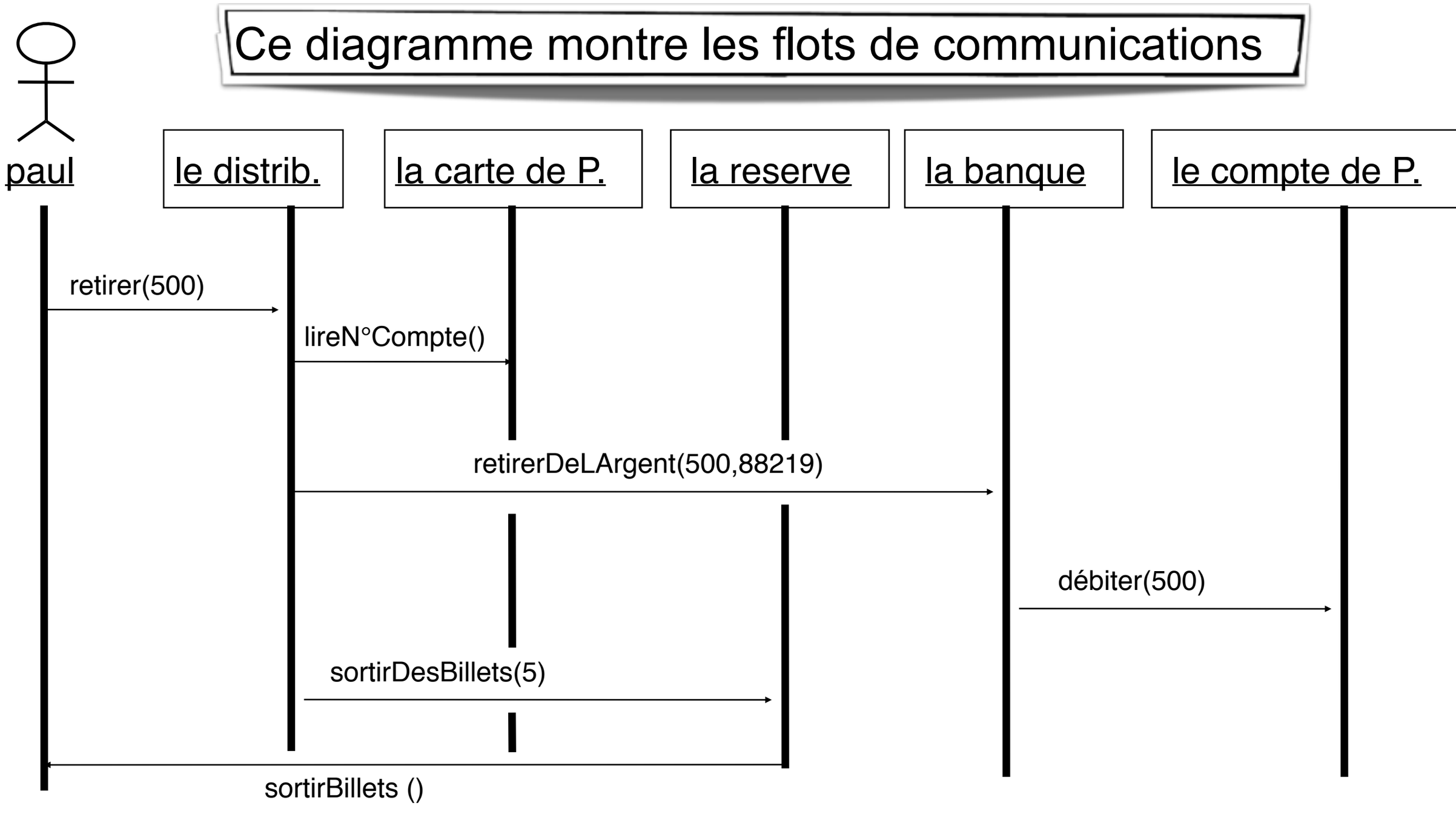
http://wwwdfr.ensta.fr/Cours/docs/IN204/uml.pdf

Diagrammes des cas d'utilisation

Ce diagramme montre ce que que fait le système et qui l'utilise



Diagrammes de séquence

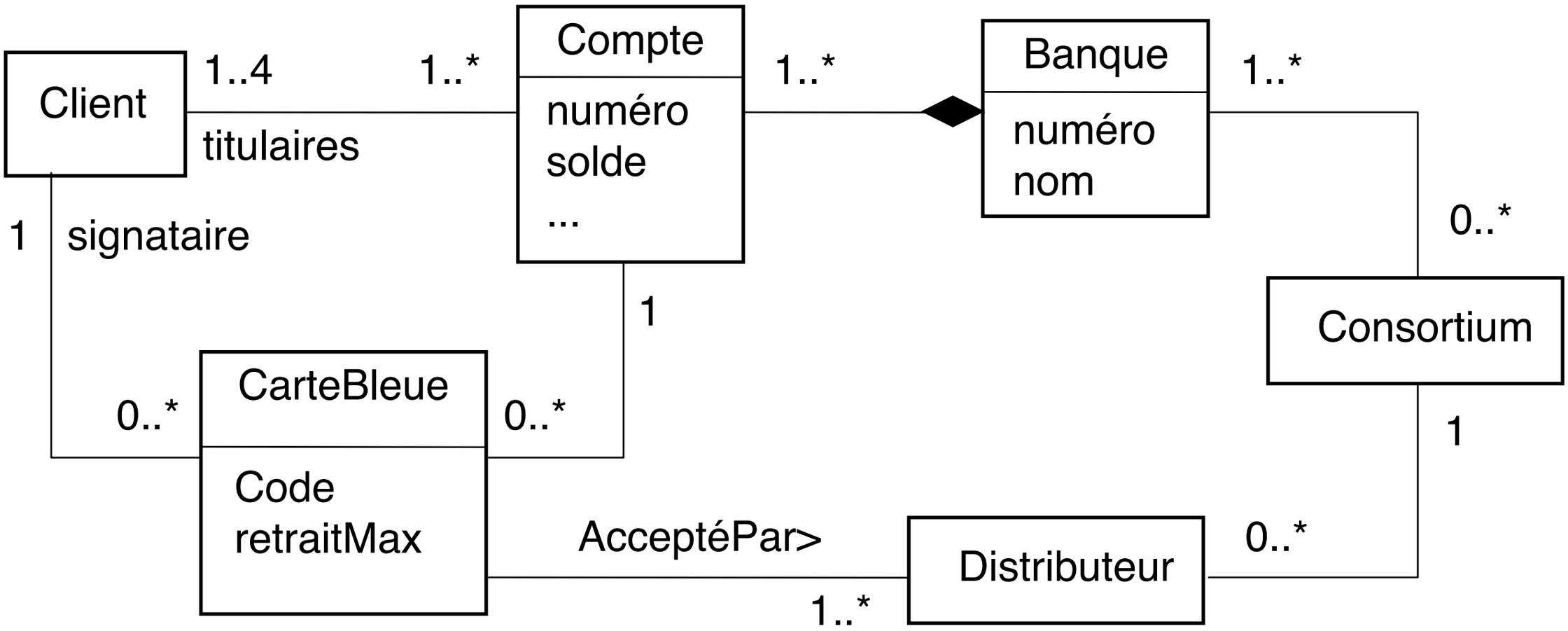


Vue Structurelle

la vue structurelle, ou statique, vise à
identifier les objets/composants
constituant le programme, leurs attributs, opérations et méthodes, ainsi que les liens ou associations qui les unissent (**diagramme de classes**). Elle permet aussi de regrouper les classes fortement liées entre elles en des composants les plus autonomes possibles (**diagramme de packages**).
A l'intérieur de chaque package, on trouve un diagramme de classes.

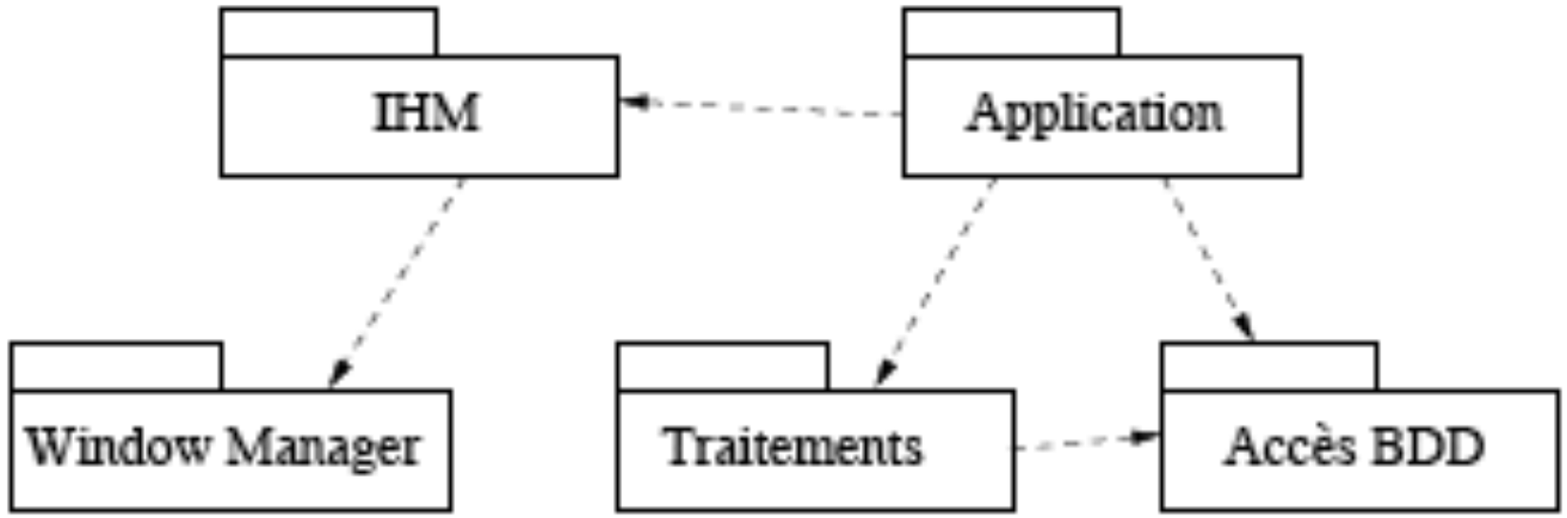
http://wwwdfr.ensta.fr/Cours/docs/IN204/uml.pdf

Diagrammes de classes



Ce diagramme montre les classes et les relations entre elles

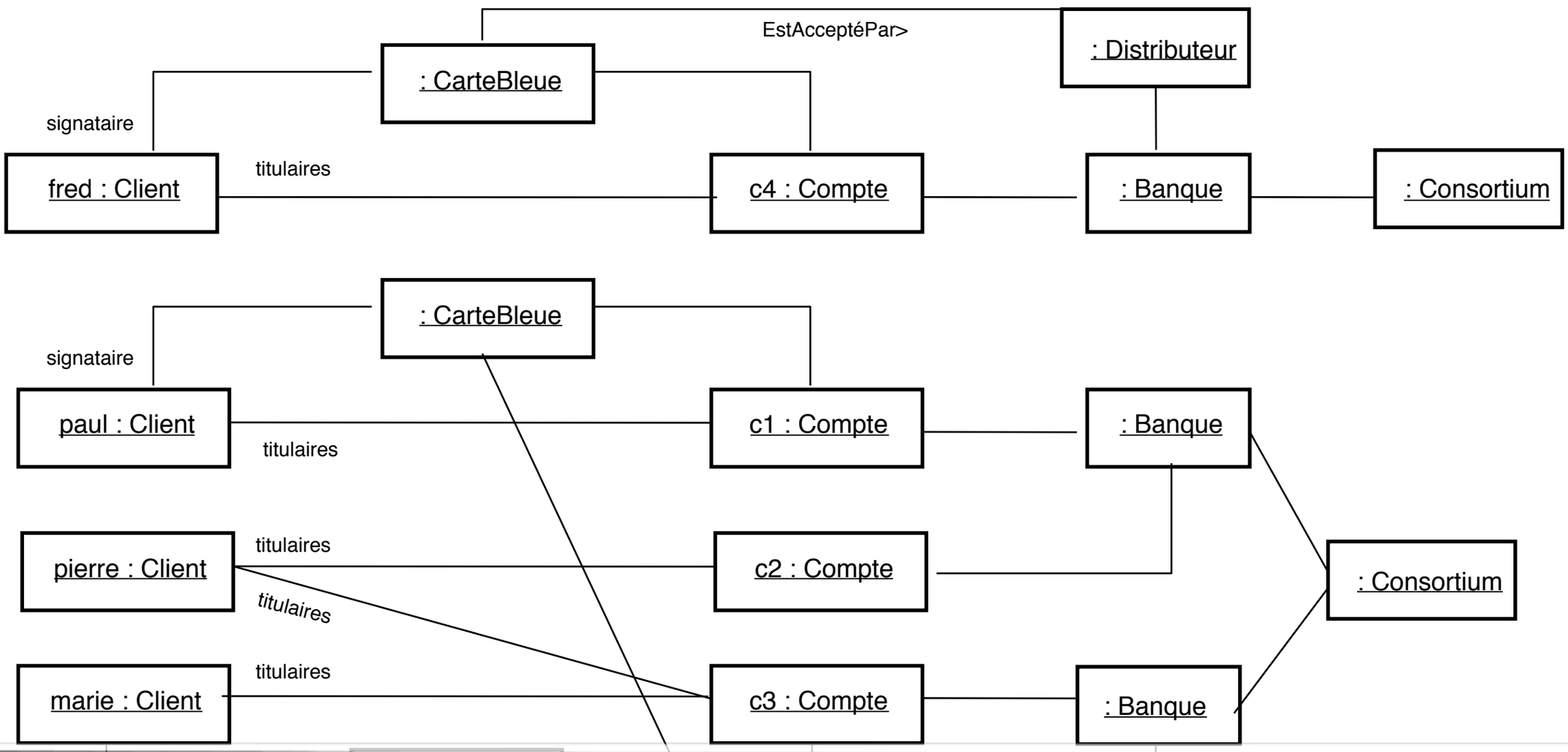
Diagrammes de packages



Regrouper entre elles des classes liées les unes aux autres de manière à faciliter la maintenance ou l'évolution du projet et de rendre aussi indépendantes que possible les différentes parties d'un logiciel.

http://wwwdfr.ensta.fr/Cours/docs/IN204/uml.pdf

Diagrammes d'objets



Ce diagramme montre les instances et les liens entre elles à l'exécution.

Vue Dynamique

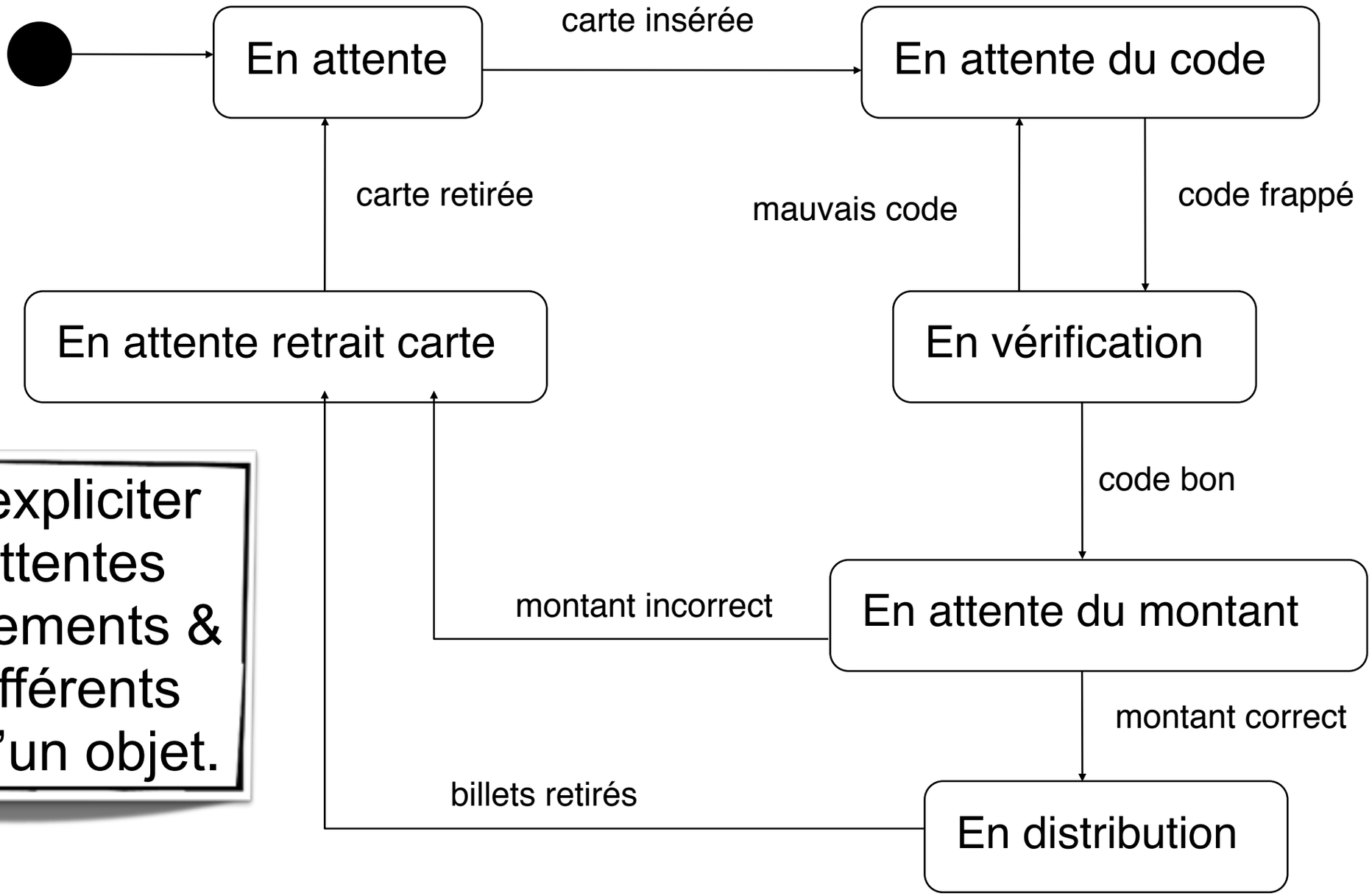
la vue dynamique vise à décrire
l'évolution (la dynamique) des objets complexes du programme tout au long de leur cycle de vie.

De leur naissance à leur mort, les objets voient leurs changement d'états guidés par les interactions avec les autres objets(les **diagrammes d'états**).

Le **diagramme d'activité** est une sorte d'organigramme correspondant à une version simplifiée du diagramme d'états. Il permet de modéliser des activités qui se déroulent en parallèle les unes des autres, quand ce parallélisme peut poser problème.

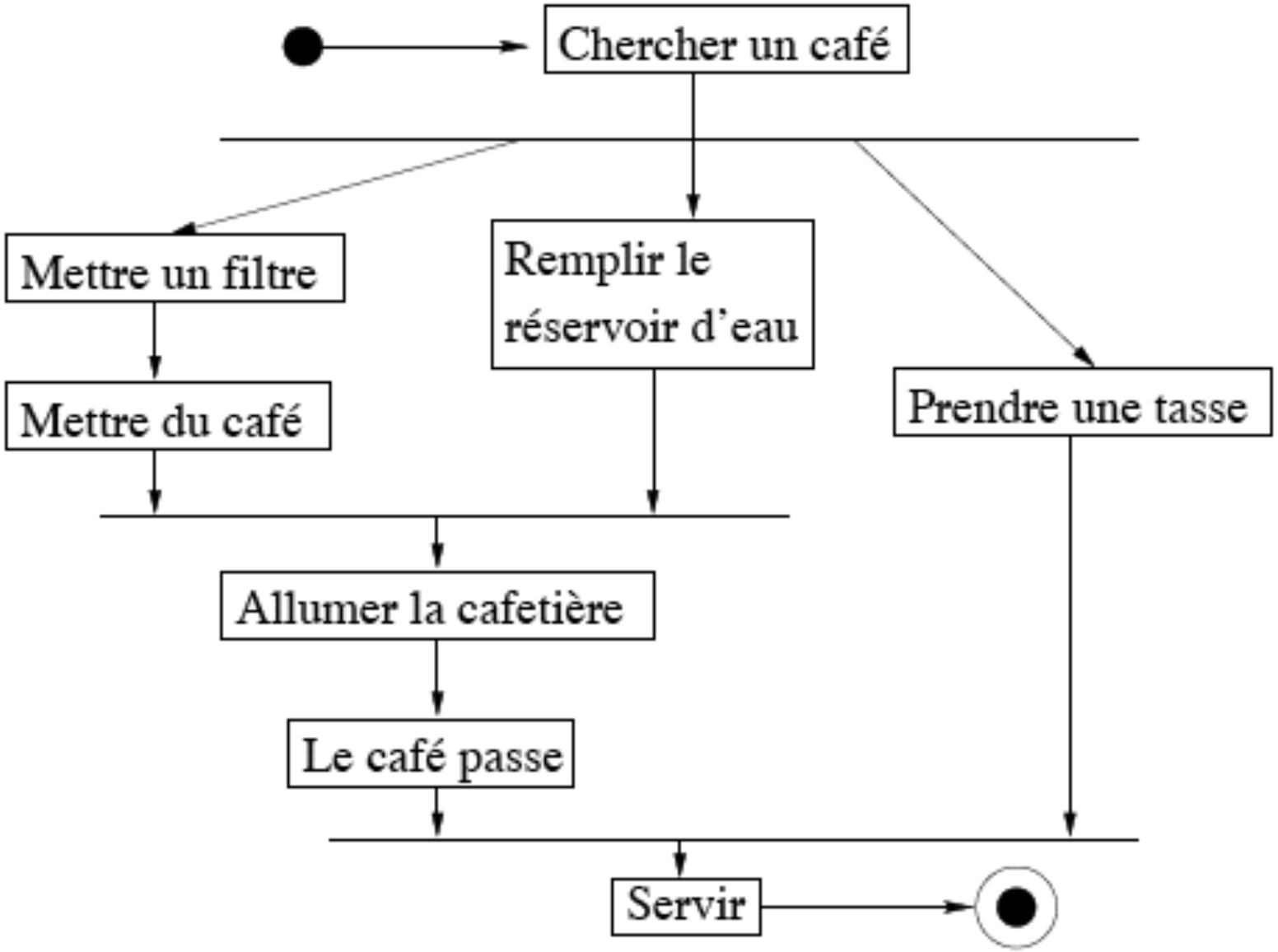
http://wwwdfr.ensta.fr/Cours/docs/IN204/uml.pdf

Diagrammes d'états



Pour expliciter les attentes d'évènements & les différents états d'un objet.

Diagrammes d'activités



Qu'est-ce qu'UML ?
Divers modes d'utilisation selon [Fowler 2003]

➡**Mode esquisse (*sketch*)**

- Informelle, incomplète
- Souvent manuelle (tableau)

➡***Support de communication pour concevoir les parties critiques***

➡**Mode plan (*blue print*)**

- Diagrammes détaillés

➡***Génération d'un squelette de code à partir des diagrammes***

➡***Nécessité de compléter le code pour obtenir un exécutable***

➡**Mode langage de programmation**

- Spécification complète, formelle et **exécutable**

➡***Pas vraiment disponible actuellement !***

Qu'est-ce qu'UML ?
Divers modes d'utilisation selon [Fowler 2003]

➡ Modèles descriptifs vs prescriptifs

- Descriptifs ; Décrire l'existant (domaine, métier)
- Prescriptifs ; Décrire le futur système à réaliser

➡ Modèles destinés à différents acteurs

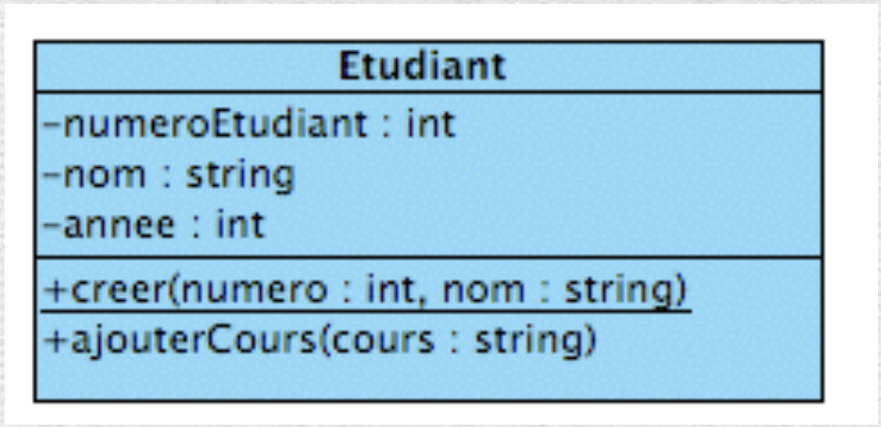
- Pour l'utilisateur ; Décrire le quoi
- Pour les concepteurs/développeurs ; Décrire le comment

http://liris.cnrs.fr/csolnon/coursUML.pdf

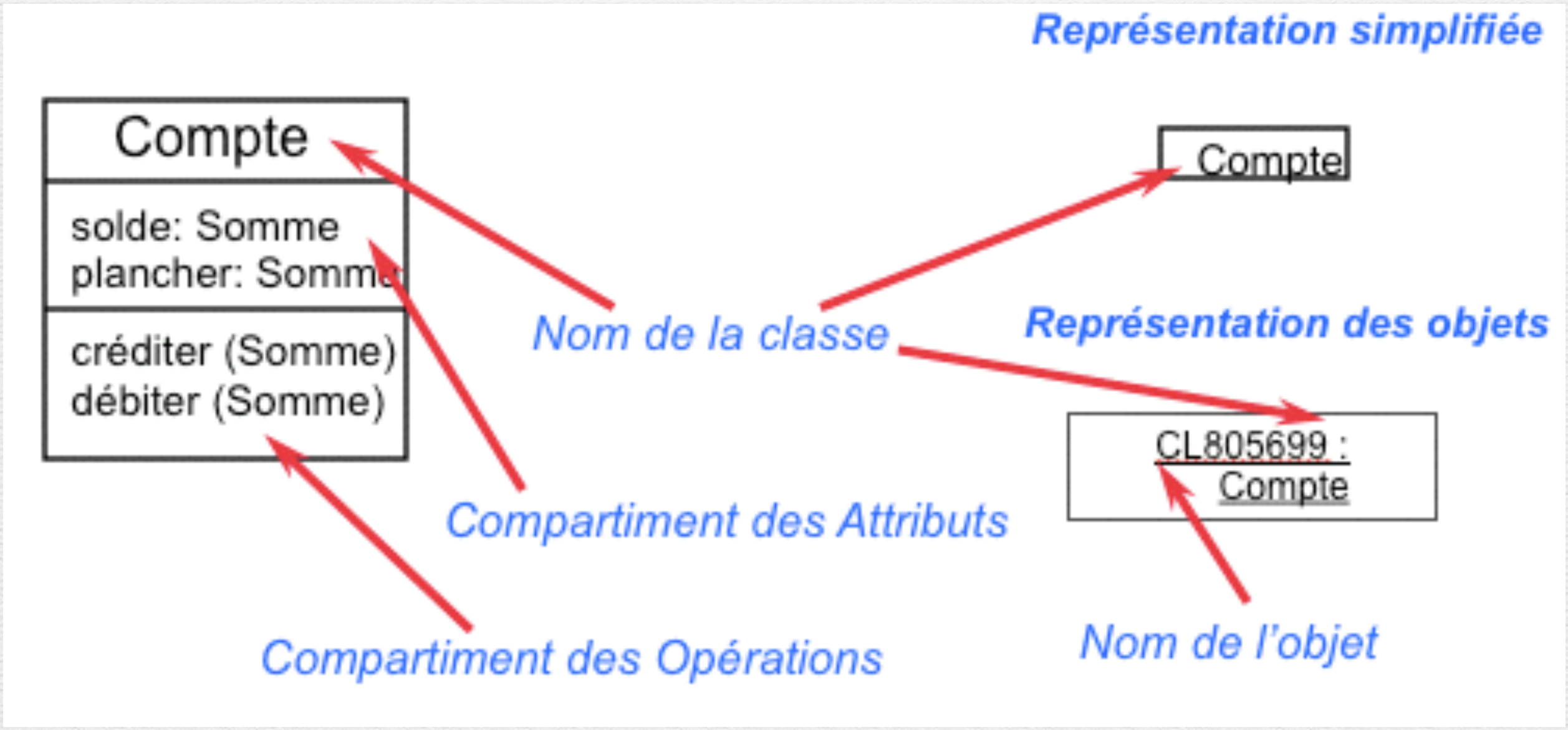
Diagramme de Classes

Classes

- Une classe est une collection *(modèle)* d'objets avec une structure commune, un comportement commun, des relations identiques et une sémantique identique
- On identifie les classes en recherchant les *concepts* du domaine et en examinant les *objets* dans les diagrammes
- La représentation graphique d'une classe consiste en un rectangle avec 3 compartiments
- Les noms des classes devraient être choisis dans le vocabulaire du domaine
  - il est bon d'établir des standards pour les nom
  - i.e., toutes les classes sont des noms communs commençant par une majuscule



Notations UML pour classes et objets



Relations entre classes

Explicitation des notations

«Relations»,
associations,
agrégation,
généralisation,
compléments

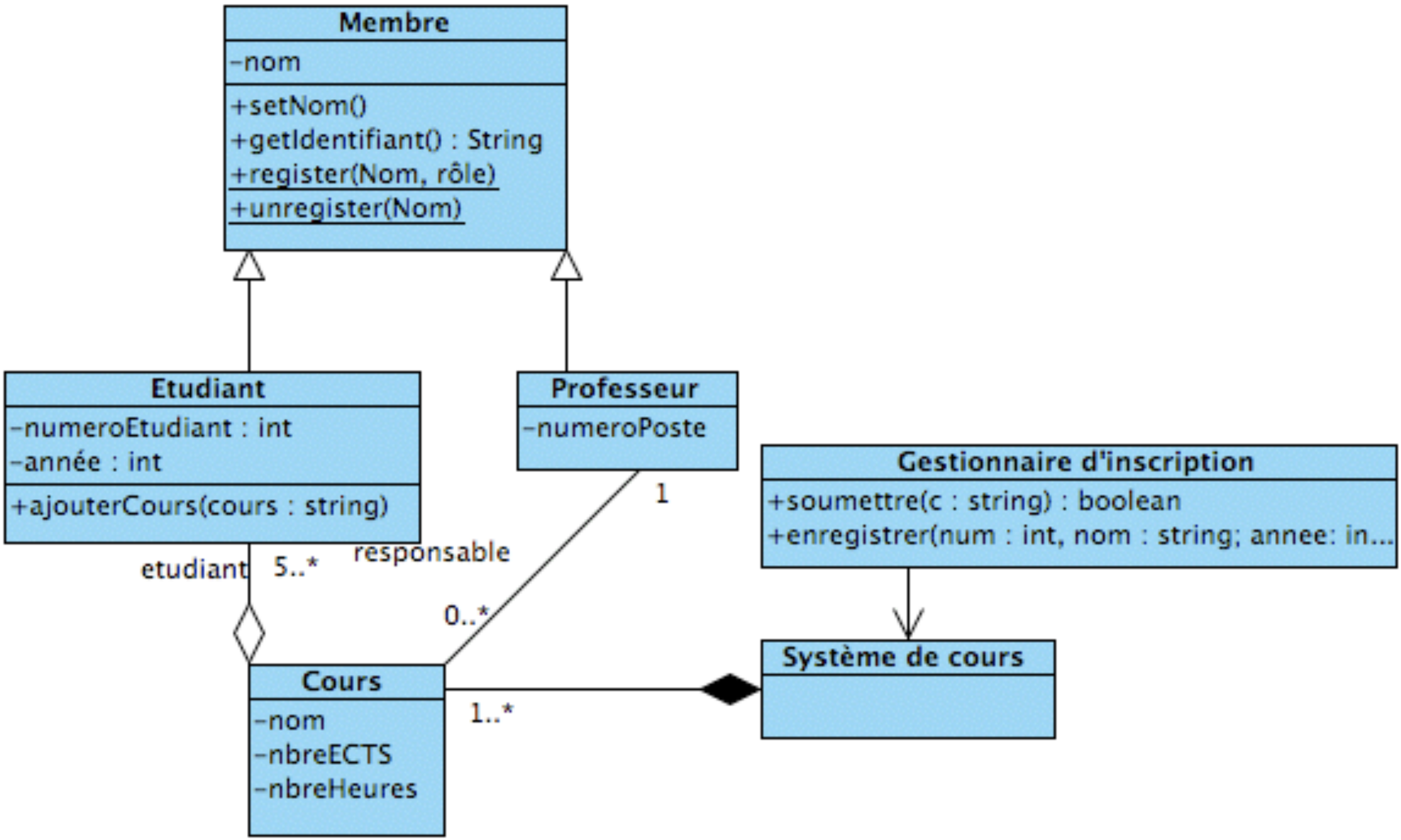
Relations

- Les relations fournissent un chemin de communication entre objets - si deux objets ont besoin de se parler, il doit exister un lien entre eux

- Trois types de relations :
  - Association
  - Agrégation
  - *Dépendance*

Si vous hésitez, utiliser une association !

Relations


Membre
-nom
+setNom()
+getIdentifiant() : String
+register(Nom, rôle)
+unregister(Nom)
Etudiant
-numeroEtudiant : int
-année : int
+ajouterCours(cours : string)
Professeur
-numeroPoste
Gestionnaire d'inscription
+soumettre(c : string) : boolean
+enregistrer(num : int, nom : string; annee: in...
etudiant
5..*
responsable
1
0..*
Cours
-nom
-nbreECTS
-nbreHeures
1..*
Système de cours

Associations

Nommage
Rôle
Multiplicité
Navigation

- Les associations peuvent avoir des étiquettes:
  - Il s'agit du **nom de l'association**.
- Les associations peuvent avoir des noms de **rôle**:
  - un nom de rôle identifie le rôle ou la responsabilité de l'objet dans l'association.
- Les associations peuvent indiquer la **navigation** avec une pointe de flèche ouverte:
  - Pas de flèche => bidirectionnelle
  - La plupart des associations sont unidirectionnelles en fin de conception.
- Les associations peuvent indiquer une **multiplicité**.

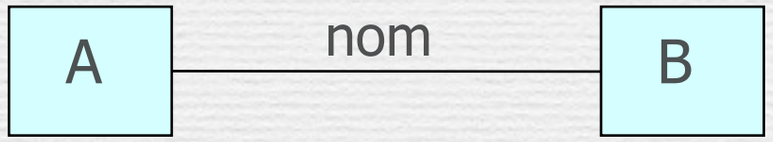
Relations

- Une association est une connexion entre classes
  - Une association est représentée par une ligne connectant les classes
- Une agrégation est une relation plus forte et s'établit entre un tout et ses parties
  - Une agrégation est représentée par une ligne connectant les classes avec un losange du côté de la classe représentant le tout

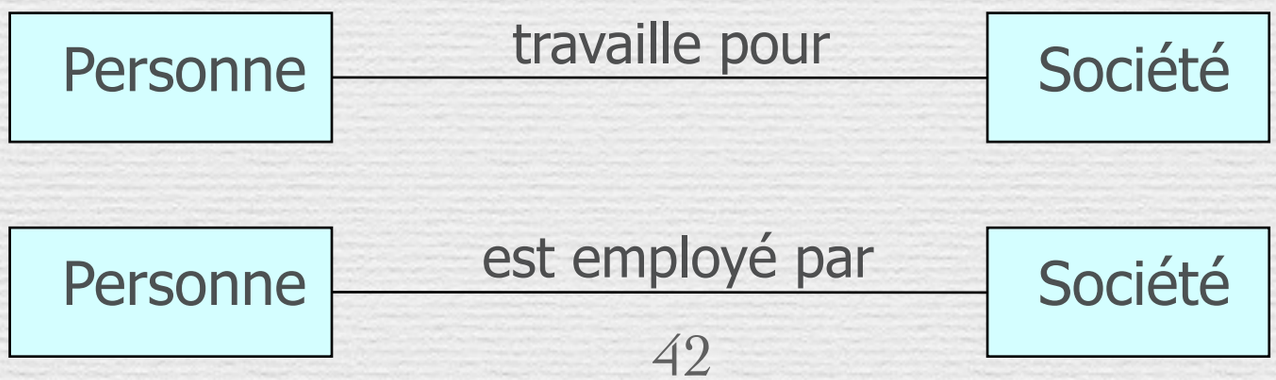
- *Une dépendance est une relation faible établie entre un client et un fournisseur et où le client n'a pas de connaissance sémantique sur le fournisseur*
  - *Une dépendance est représentée par une flèche en pointillés allant du client au fournisseur*

Nommage des associations

Une association peut être nommée afin de faciliter la compréhension des modèles. Dans ce cas le nom est indiqué au milieu du lien symbolisant l'association



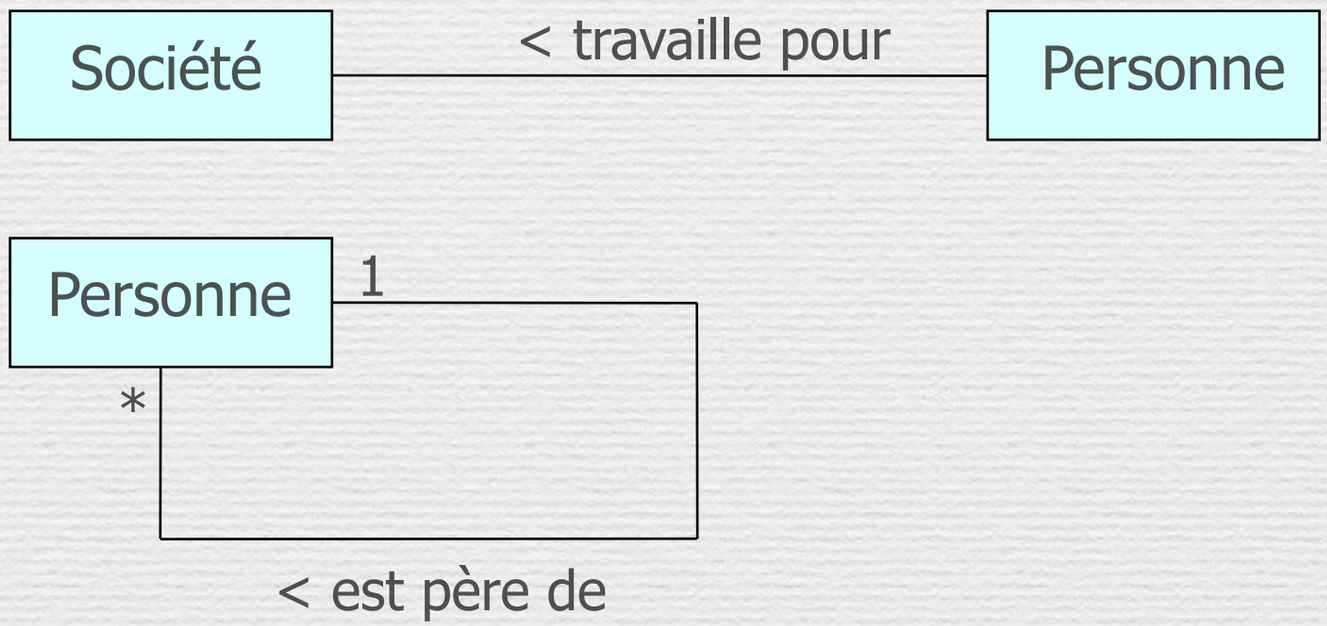
L'usage recommande de choisir comme nom d'une association une forme verbale active (exemple : travaille pour) ou passive (exemple : est employé par)



Nommage des associations…

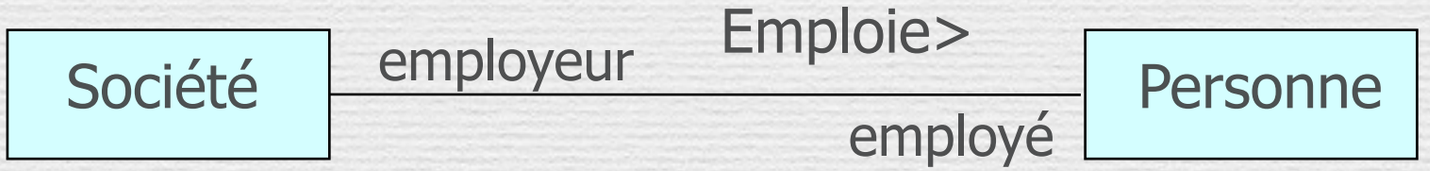
- Par défaut le sens de lecture du nom d'une association est de gauche à droite
- Dans le cas où la lecture du nom est ambiguë, on peut ajouter l'un des signes < ou > pour indiquer le sens de lecture

- **Exemples**

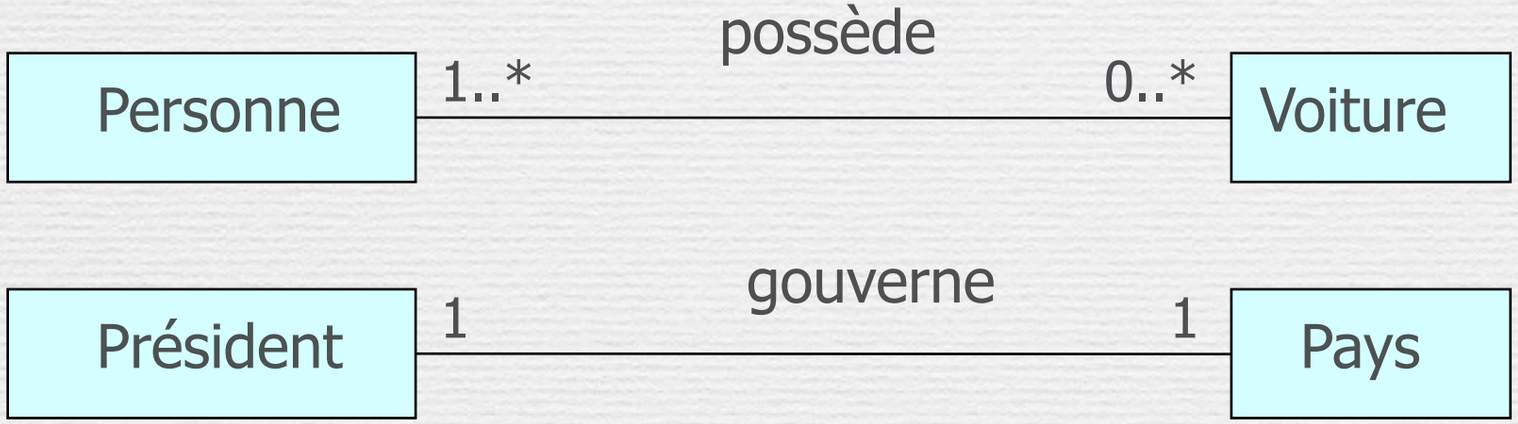


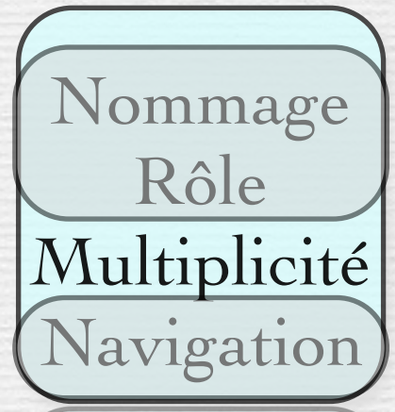
Rôles des extrémités d'association

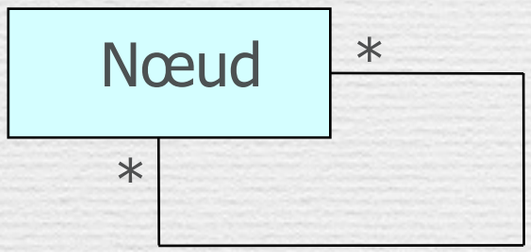
- On peut attribuer à une extrémité d'une association un nom appelé rôle qui décrit comment une classe source voit une classe destination au travers de l'association
- Le rôle est placé près de la fin de l'association et à côté de la classe à laquelle il est appliqué
- L'utilisation des rôles est optionnelle
- Représentation et exemple



Multiplicité : exemple





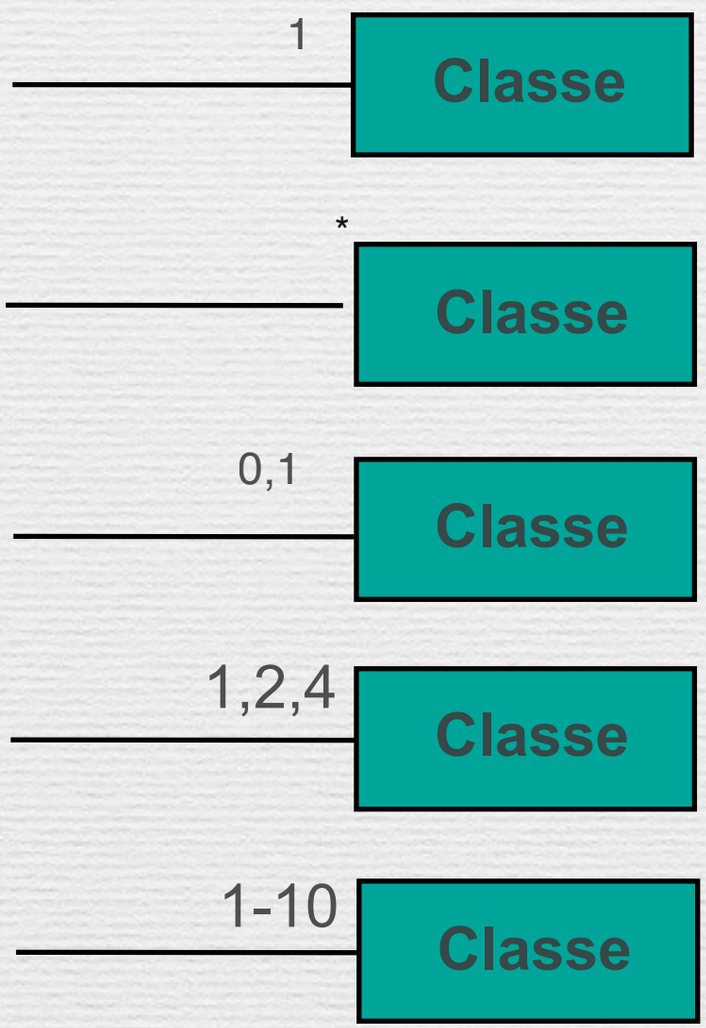


Association réflexive

Un réseau informatique est composé de nœuds inter-connectés

Multiplicité

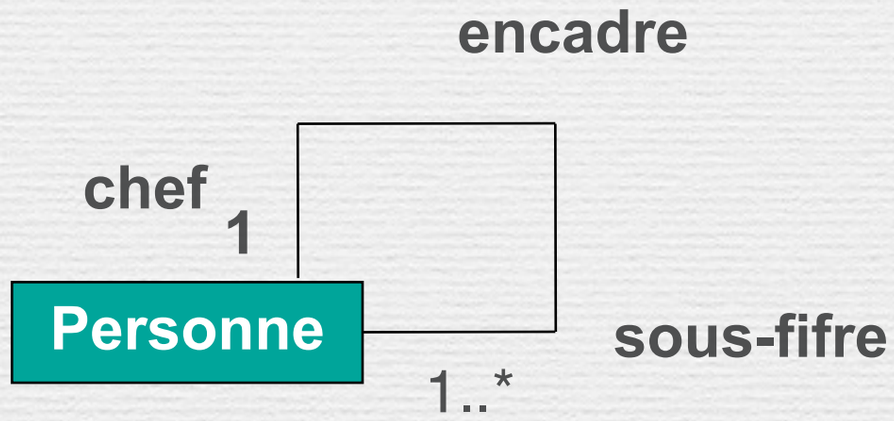
- La multiplicité est définie par le nombre d'objets qui participent à une relation
  - La multiplicité est le nombre d'instances d'une classe reliées à UNE instance d'une autre classe
  - Pour chaque association et agrégation, il y a deux multiplicités : une à chaque bout de la relation



Cas particuliers de relations

- Relations réflexives

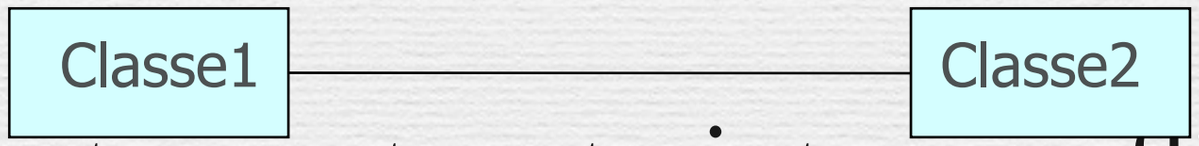
Aie aie... une personne a toujours un chef... elle est le chef d'elle-même?



Une relation réflexive lie des objets de même classe

Navigation

- Bien que les associations soit bi-directionnelles par défaut, il peut être bon de limiter la navigation à un seul sens
  - Les objets de Classe2 sont accessibles à partir de ceux de Classe1 et vice-versa

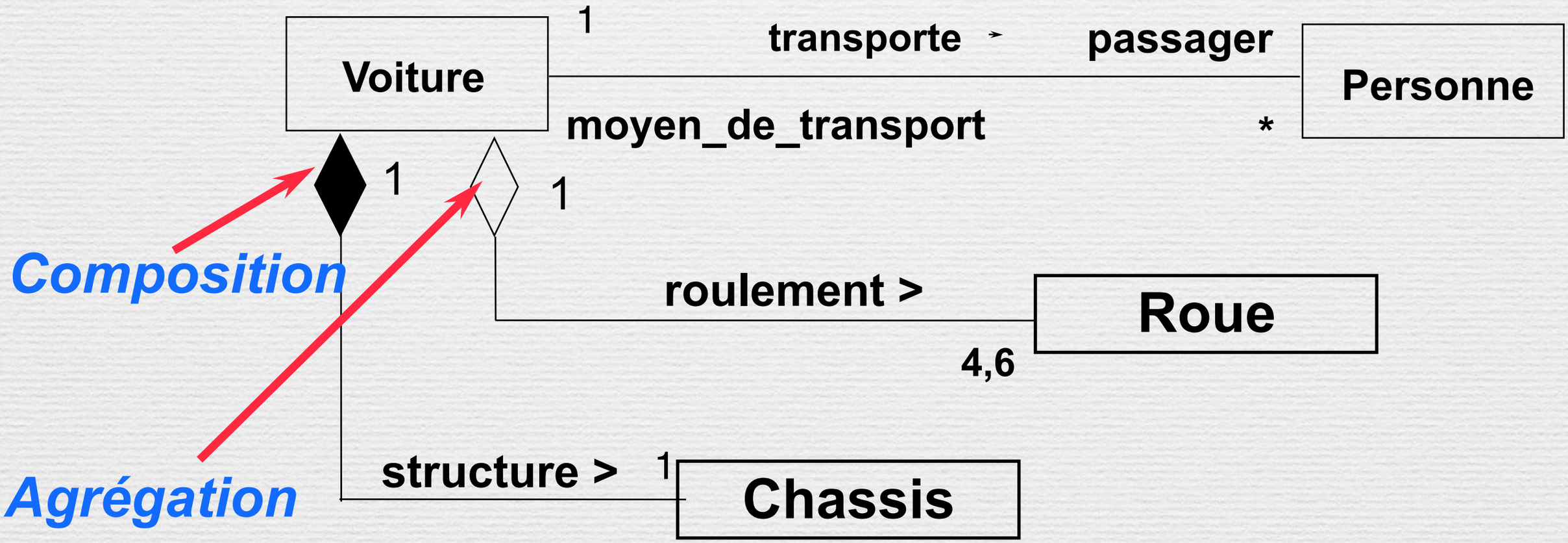


- Si la navigation est restreinte, une flèche indique le sens de navigation
  - Les objets de la Classe 1 sont accessibles à la classe 2

Nommage
Rôle
Multiplicité
Navigation

Composition et Agrégation

- Cas particuliers de relations :
  - Notion de *tout* et *parties*



Agrégation

- L'agrégation représente une association de type ensemble/élément
- L'agrégation ne concerne qu'un seul rôle d'une association
- Représentation

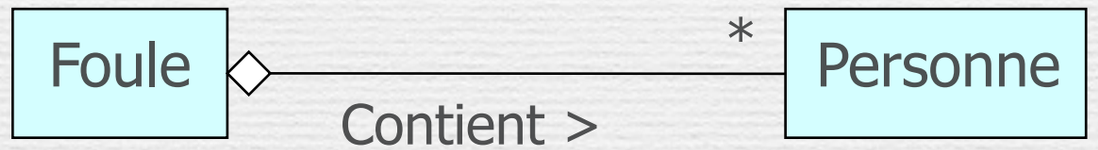
- L'agrégation permet de modéliser une contrainte d'intégrité et de désigner l'agrégat comme gérant de cette contrainte

Agrégation…

- Exemple 1

– Une personne est dans une foule

– Une foule contient plusieurs personnes



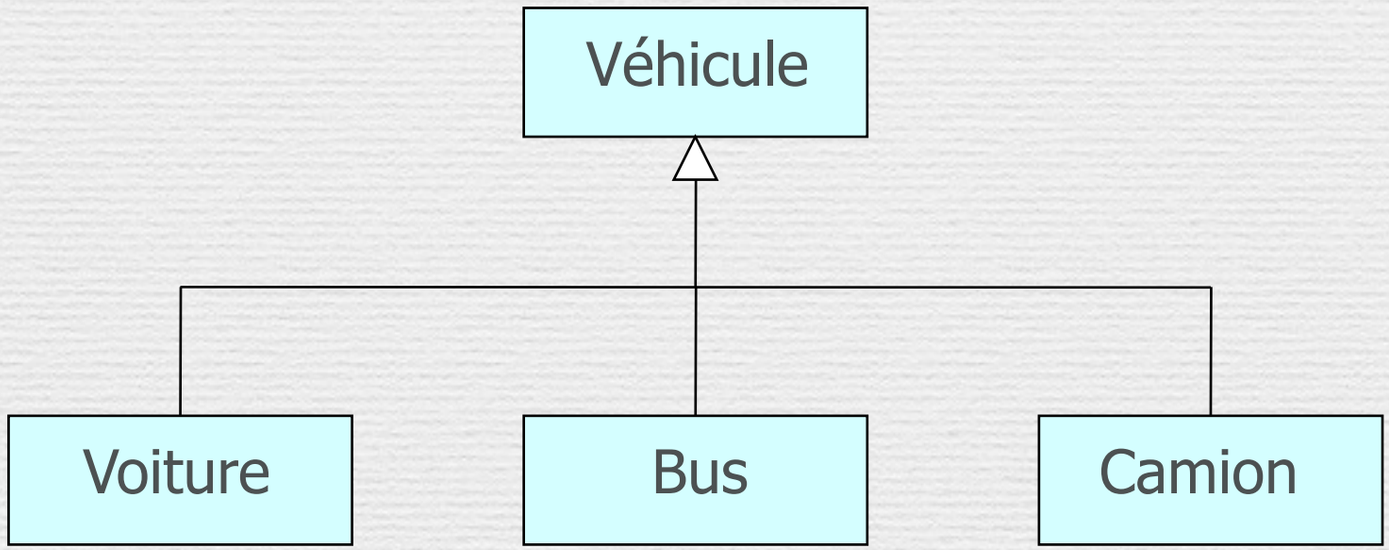
- Exemple 2 (Agrégation partagée)

– Une personne fait partie de plusieurs équipes

– Une équipe contient plusieurs personnes

Généralisation

- C'est une relation de classification entre un élément général et un élément plus spécifique
- L'élément le plus spécifique est cohérent avec l'élément le plus général et contient plus d'informations
- Exemple

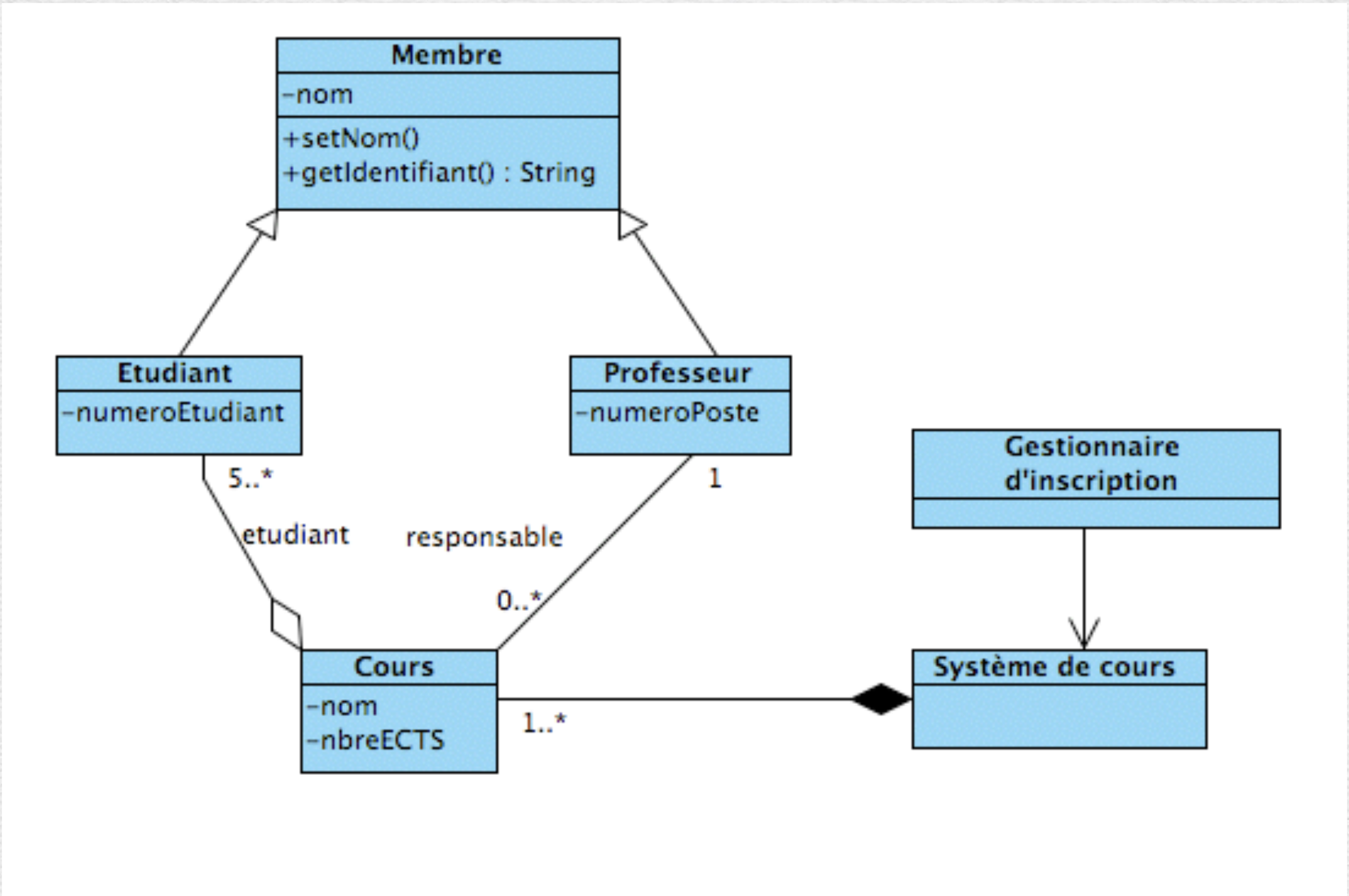


Héritage

- L'héritage est une relation entre une super-classe (classe de base) et ses sous-classes (classes dérivées)
- Deux manières d'identifier une relation d'héritage :
  - Généralisation
  - Spécialisation
- Les éléments communs (attributs, comportements, relations) sont reportés au niveau le plus haut de la hiérarchie

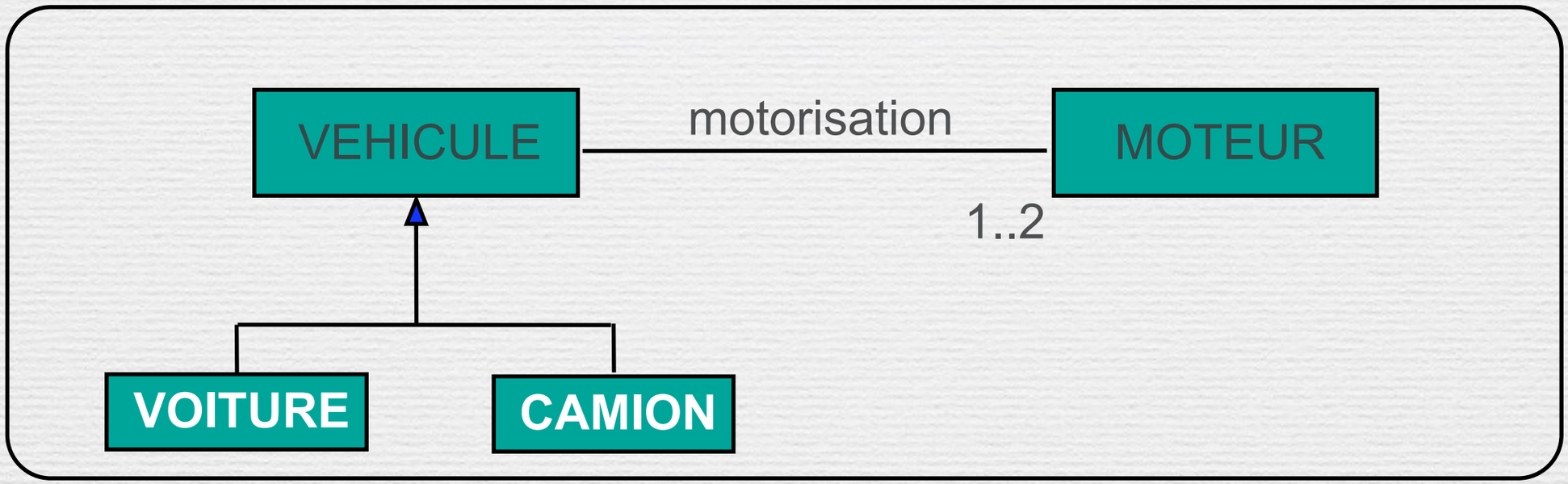
Généralisation

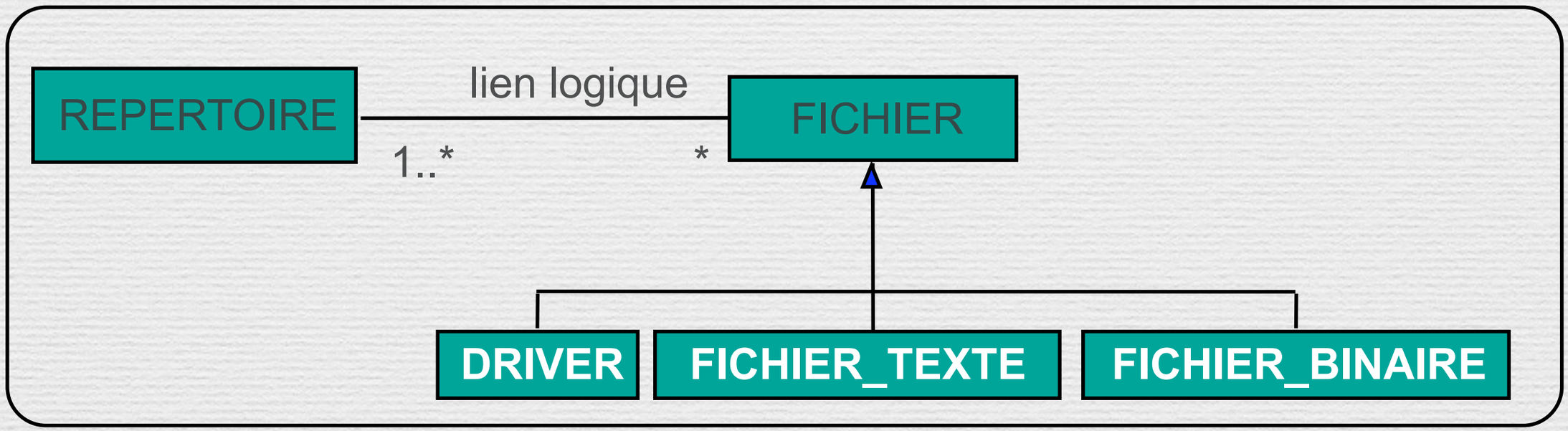
- Extension par l'ajout d'attributs et d'opérations



Héritage des relations

- Les relations sont héritées par les sous classes :





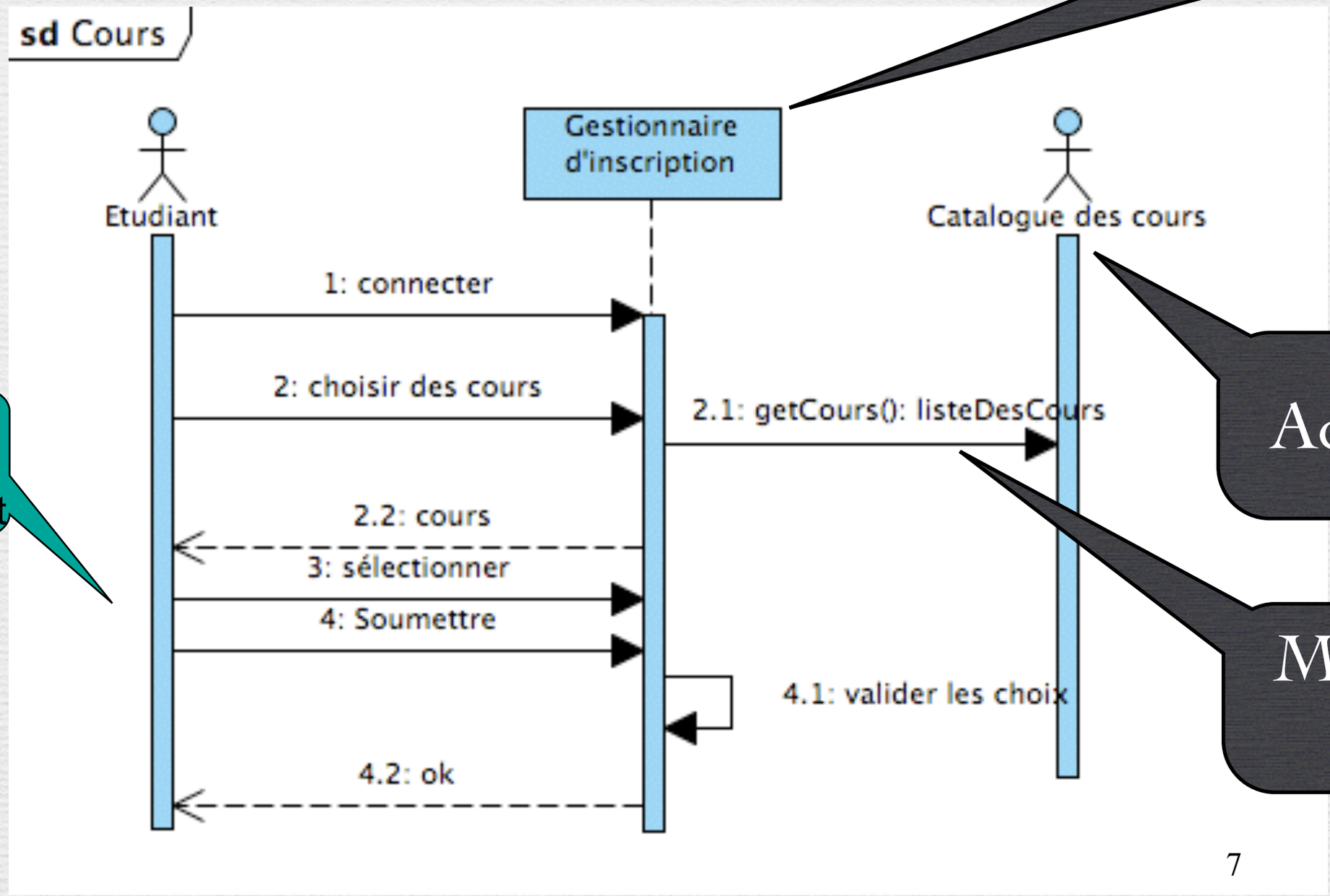
Généralisation signifie :

- Héritage
  - L'enfant acquiert les propriétés du ou des parents (s) : les attributs, les relations et les opérations
- Substituabilité
  - Il est possible de substituer une instance d'une sous-classe à une instance de la classe.

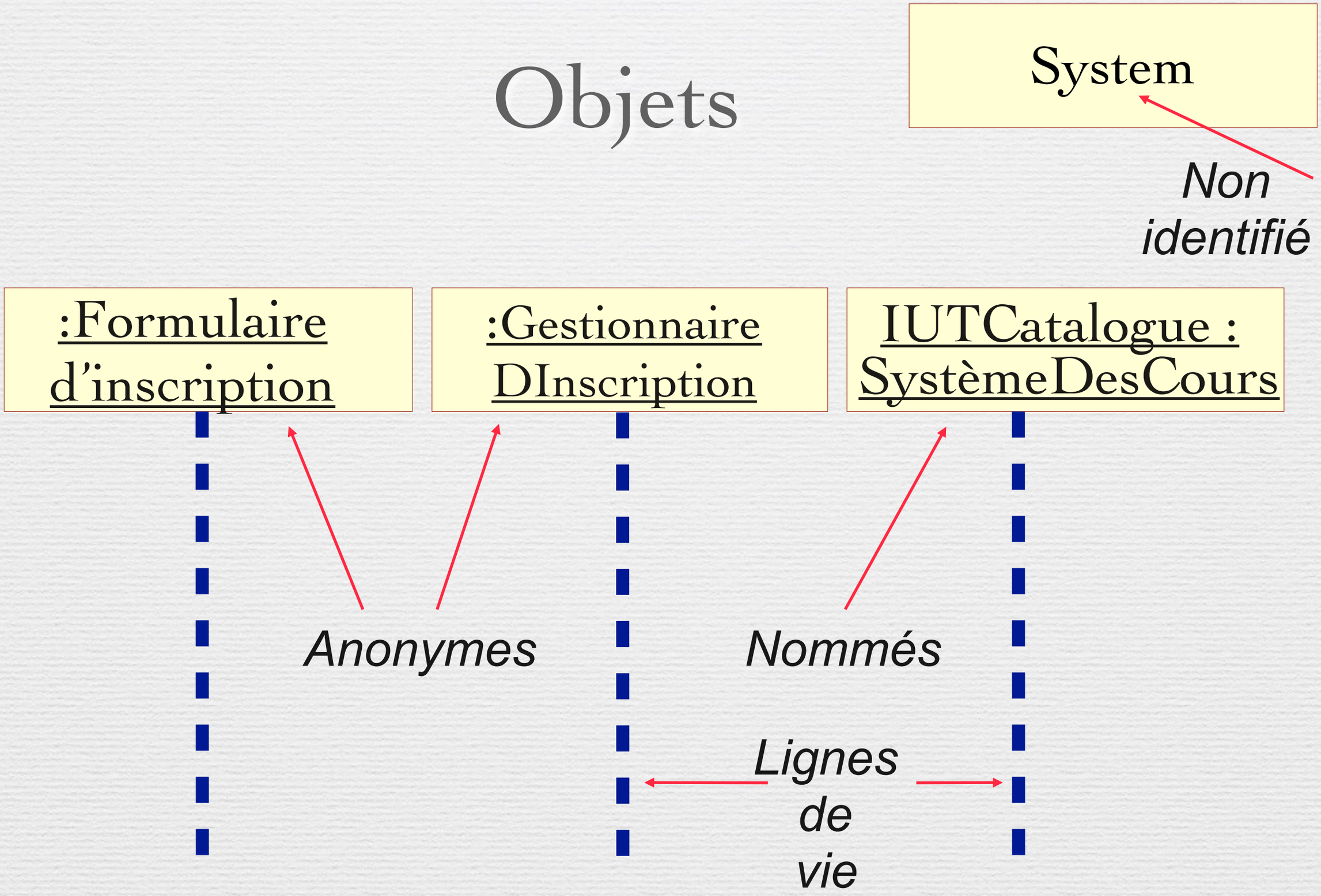
Diagramme de Séquences

Diagramme de Séquence

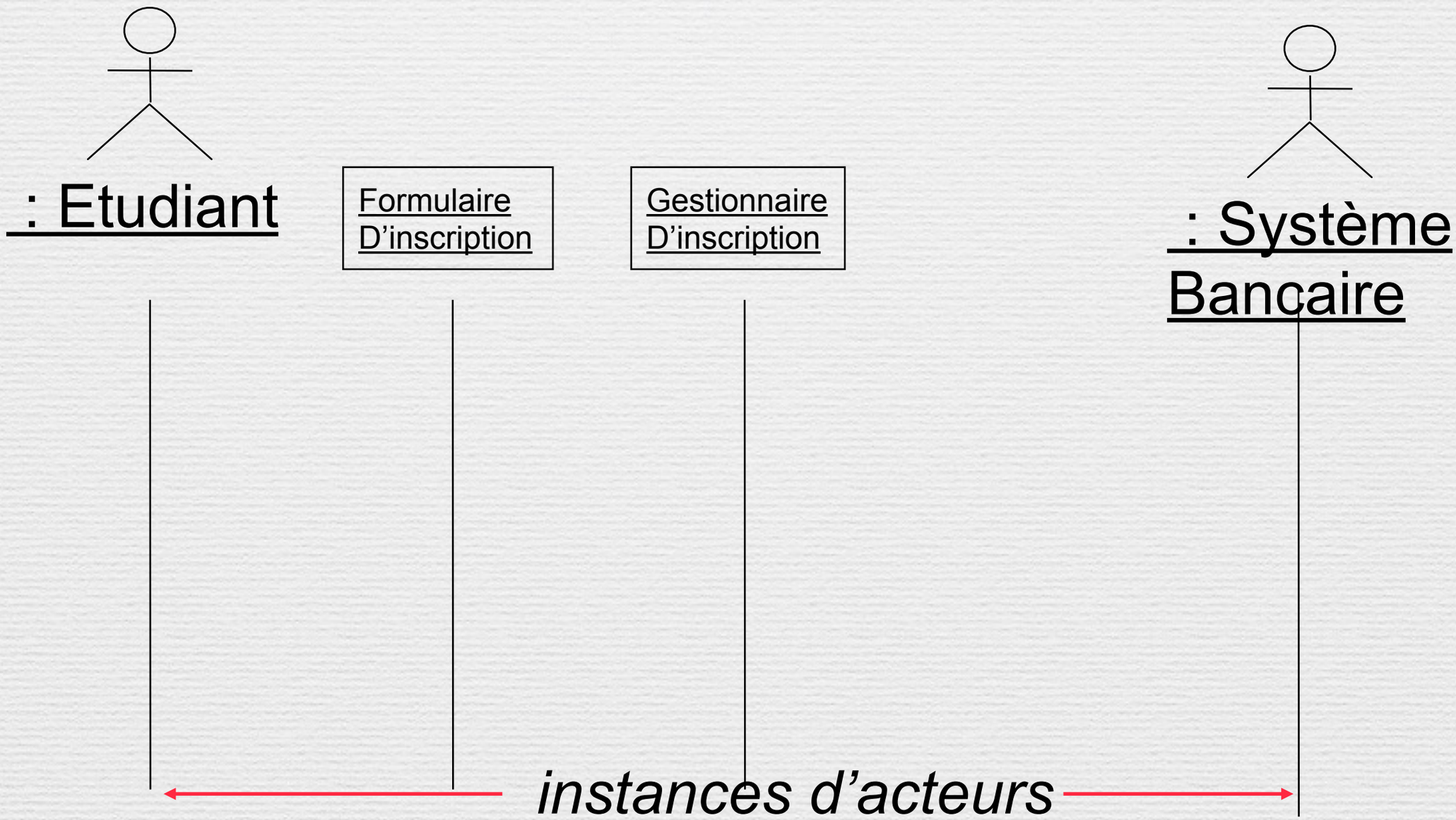
- Vue temporelle de l'interaction entre objets



Objets


System
Non identifié
:Formulaire d'inscription
:Gestionnaire DInscription
IUTCatalogue : SystèmeDesCours
Anonymes
Nommés
Lignes de vie

Objets et Acteurs

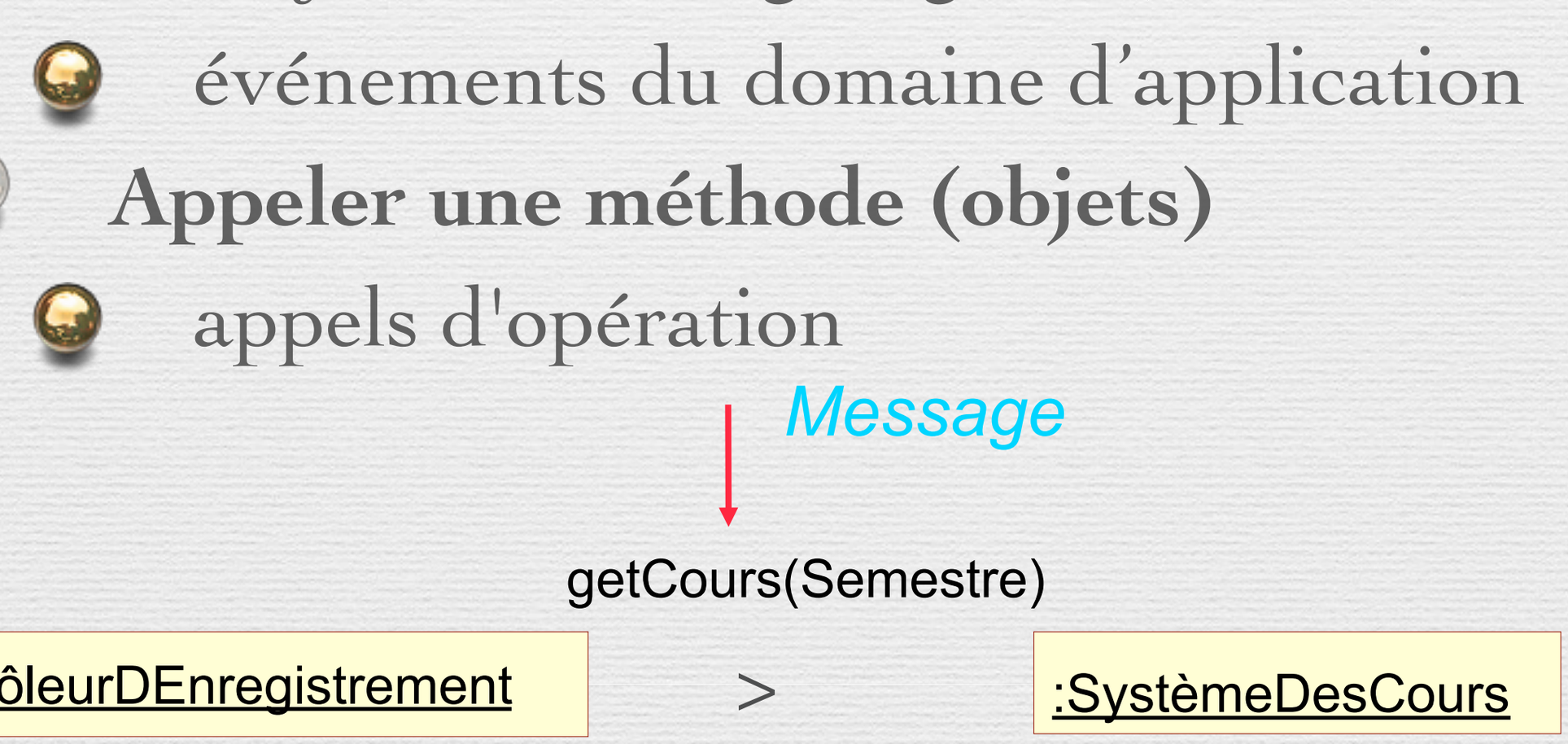


7

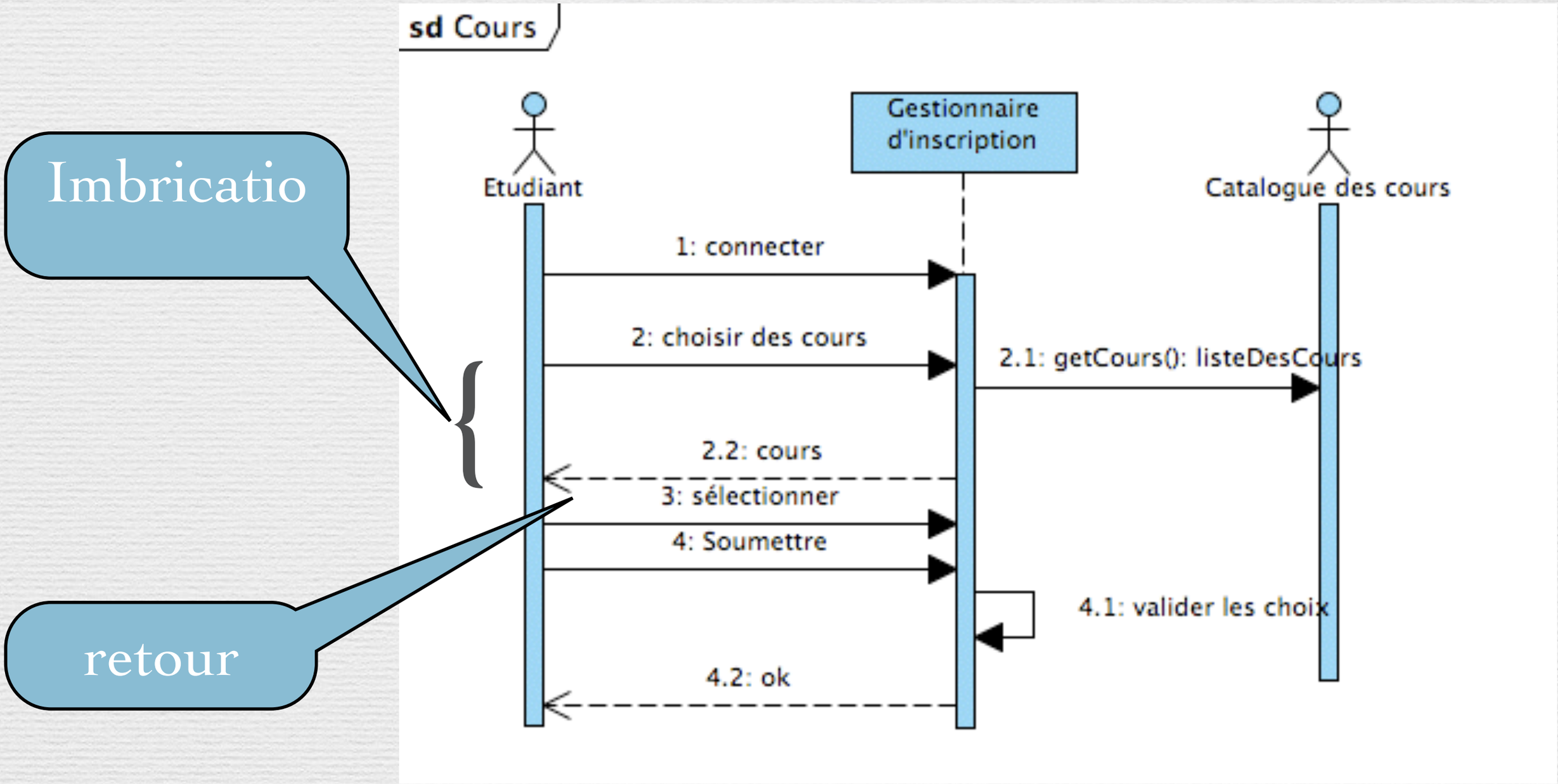
Messages

Un message est la spécification d'une communication entre objets avec les informations nécessaires pour qu'une activité s'ensuive.

- Envoyer un message/signal (acteur)
  - événements du domaine d'application
- **Appeler une méthode (objets)**
  - appels d'opération

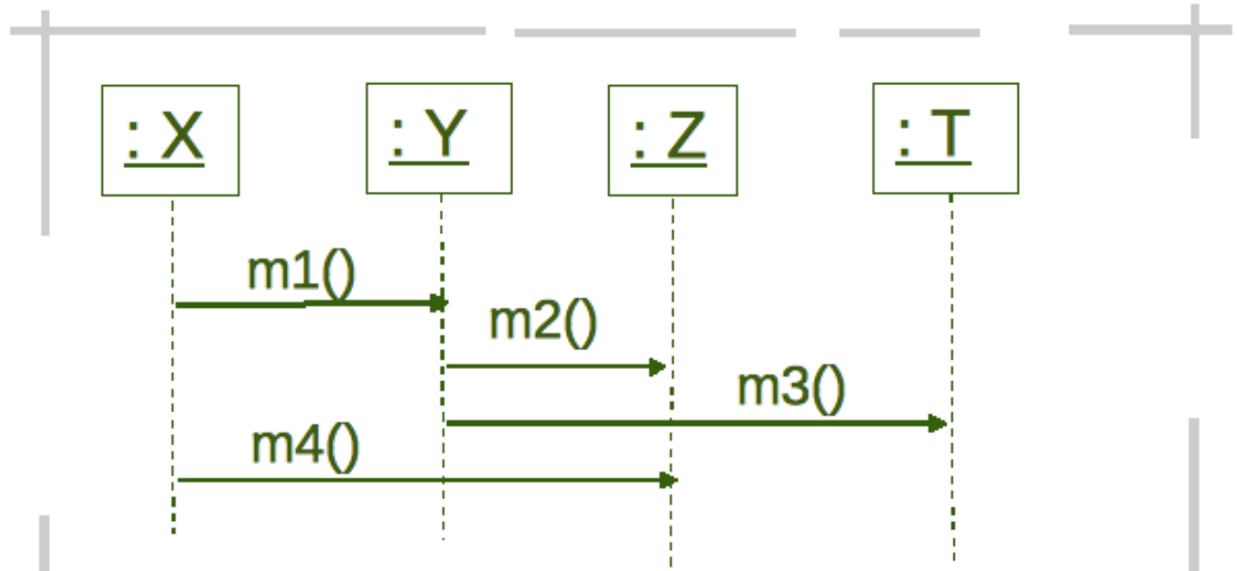


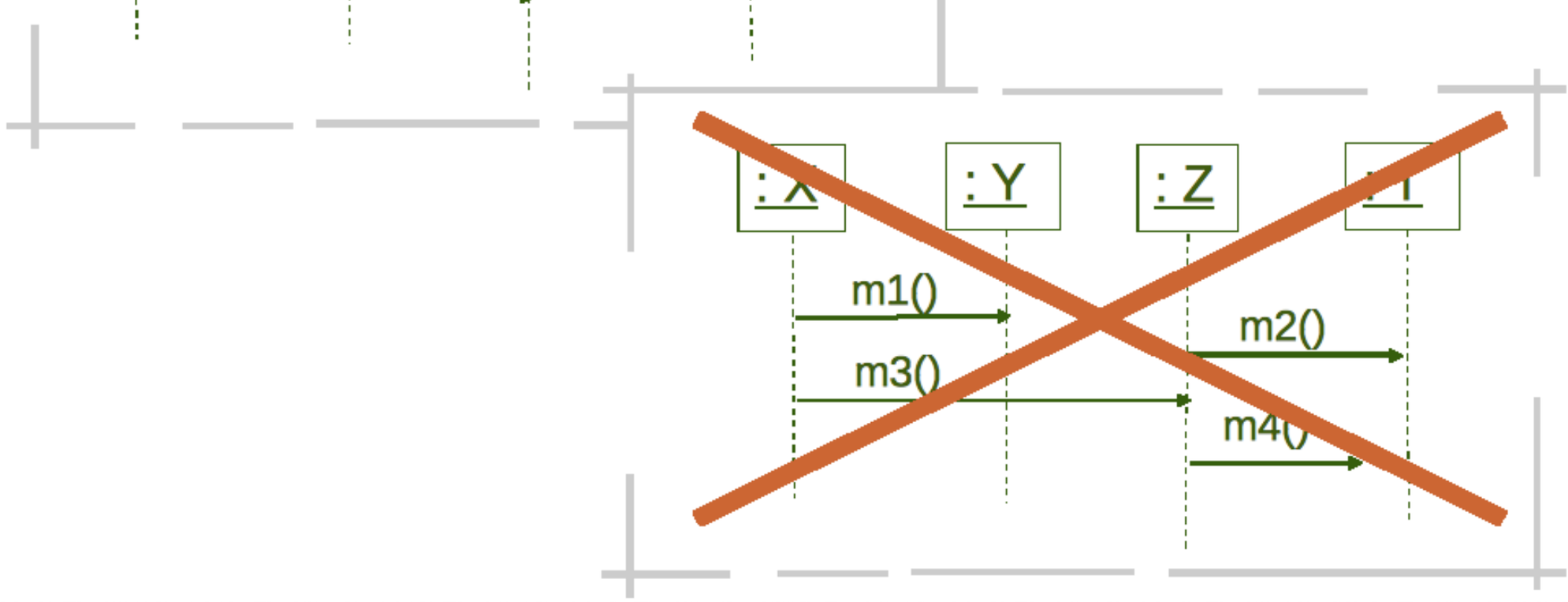
Envoi de message



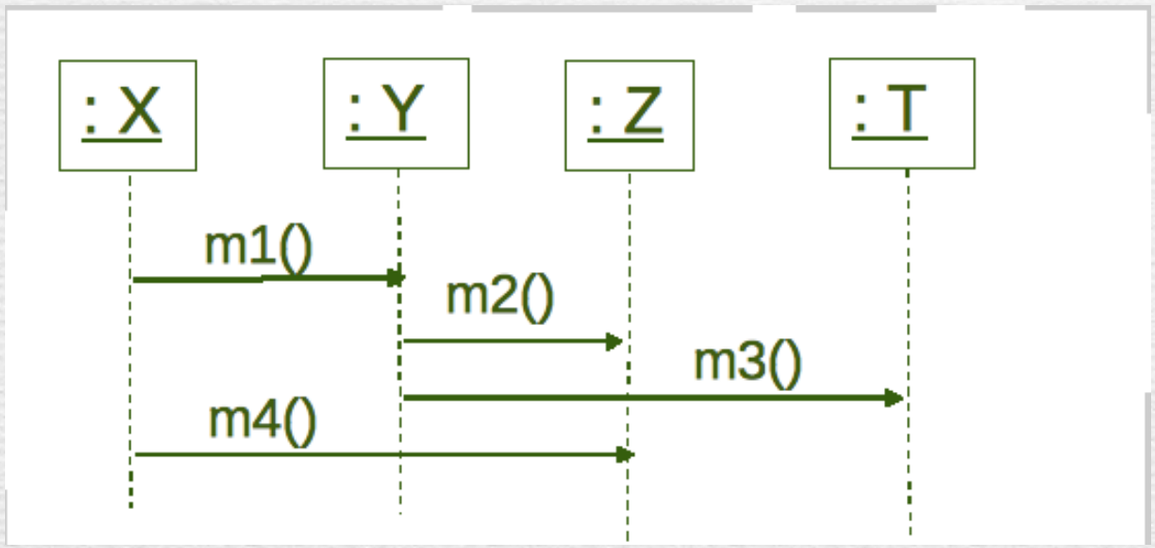
Succession des appels

- MESSAGES (suite)
  - ✔ Soigner la succession des appels





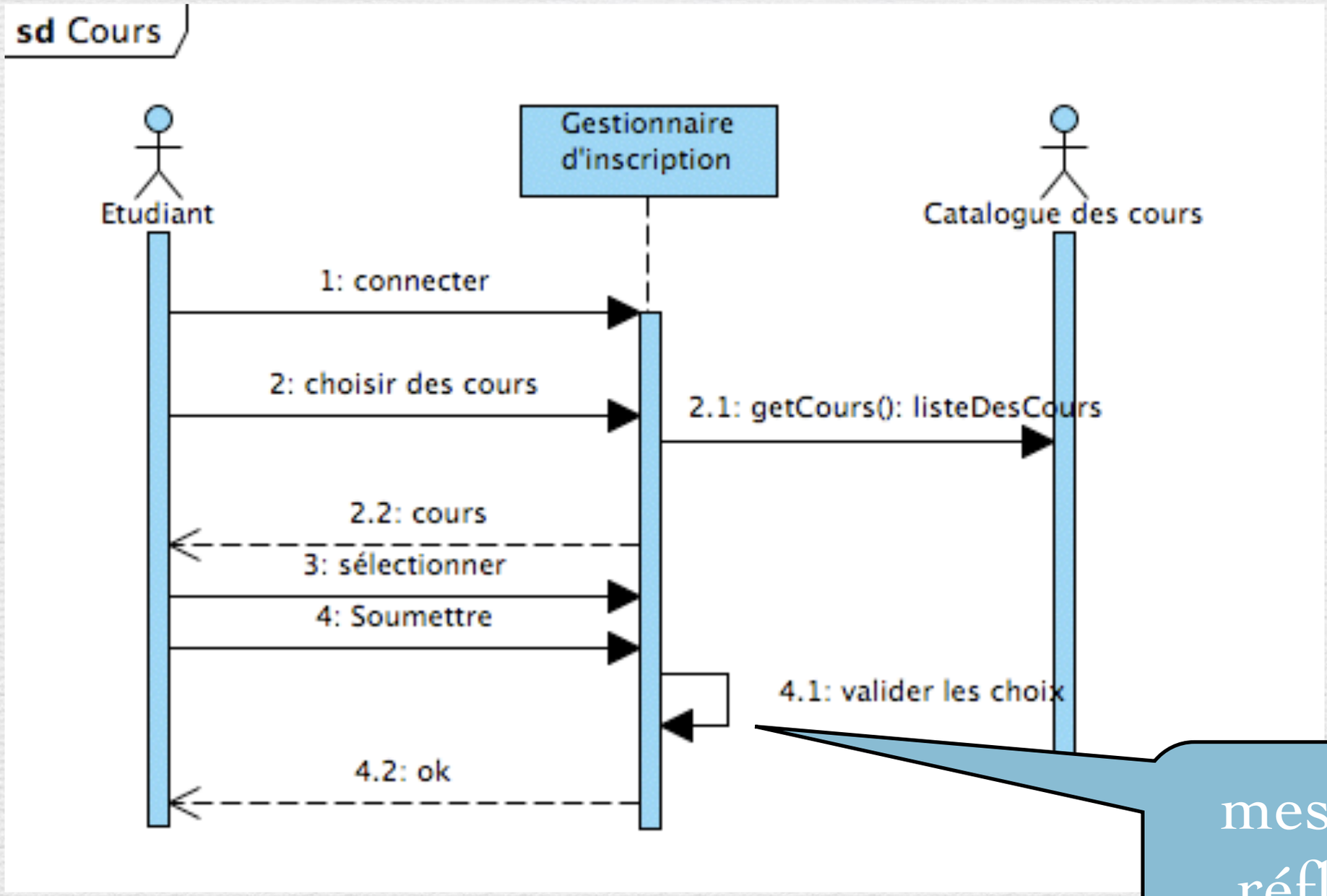
Lien avec le code



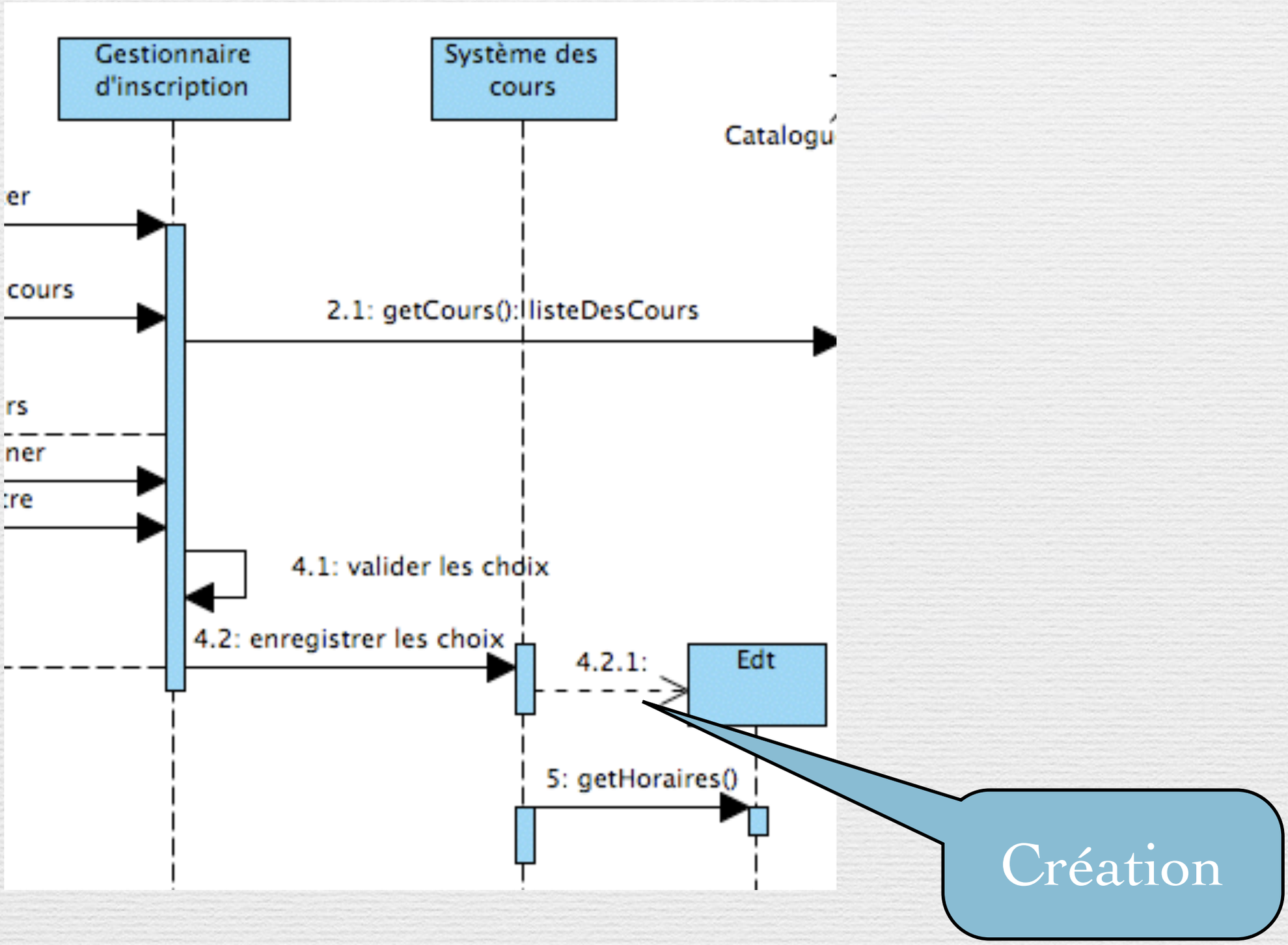
Y	Z	T
+m1()	+m2() +m4()	+m3()

```
public class Y {
    private Z z;
    private T t;

    public void m1() {
        z.m2();
        t.m3();
    }
}
```

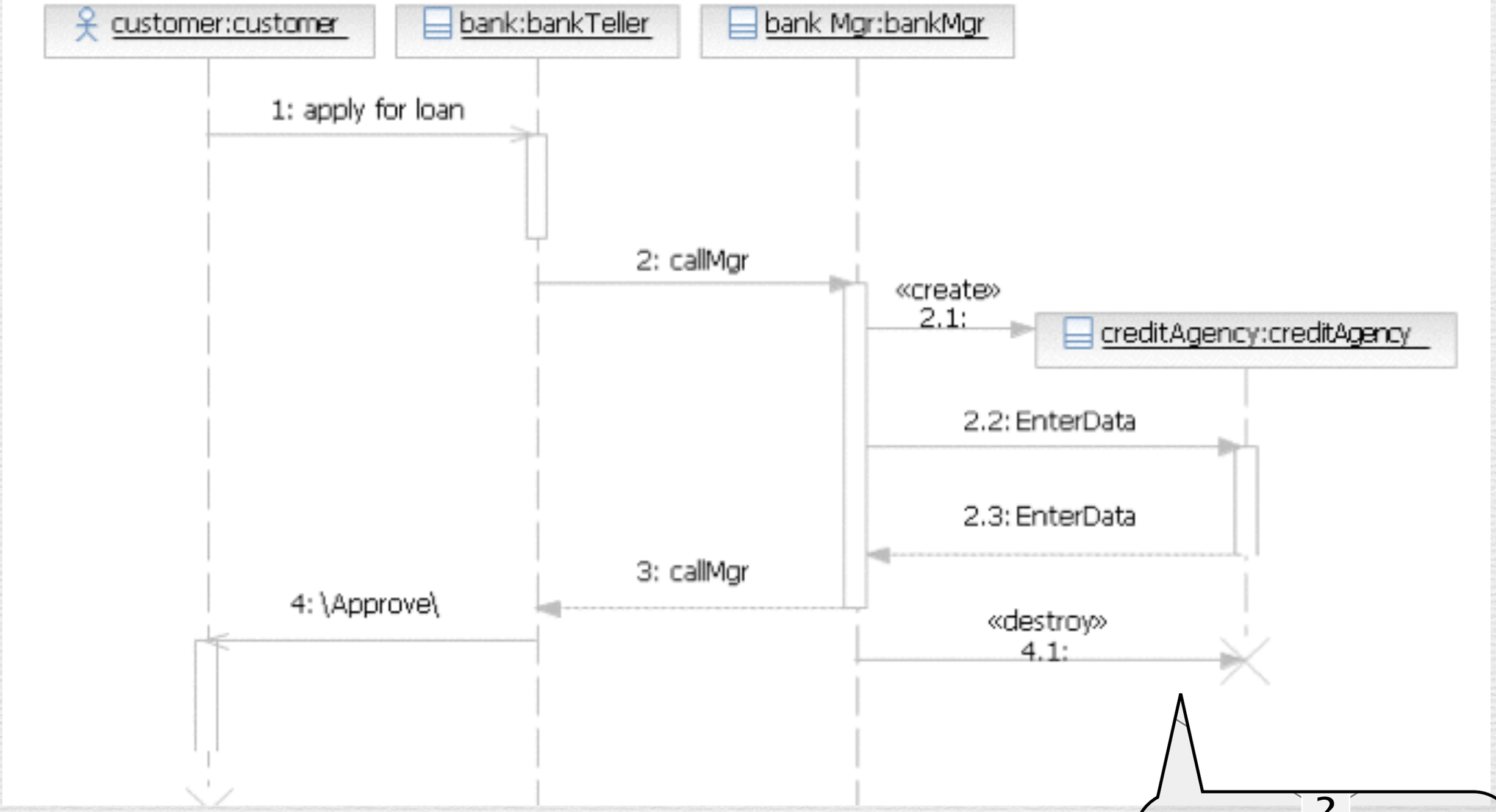
Envoi de message

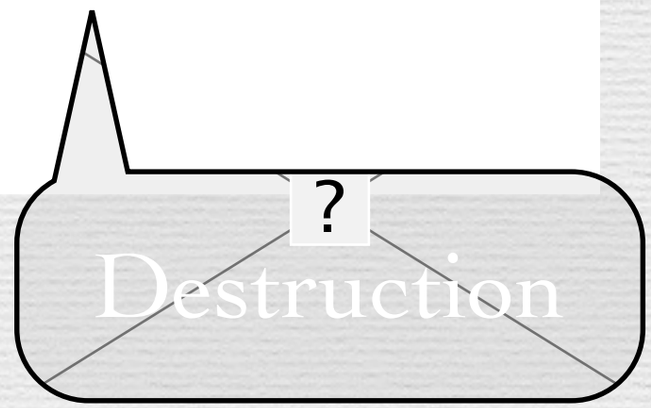


Création

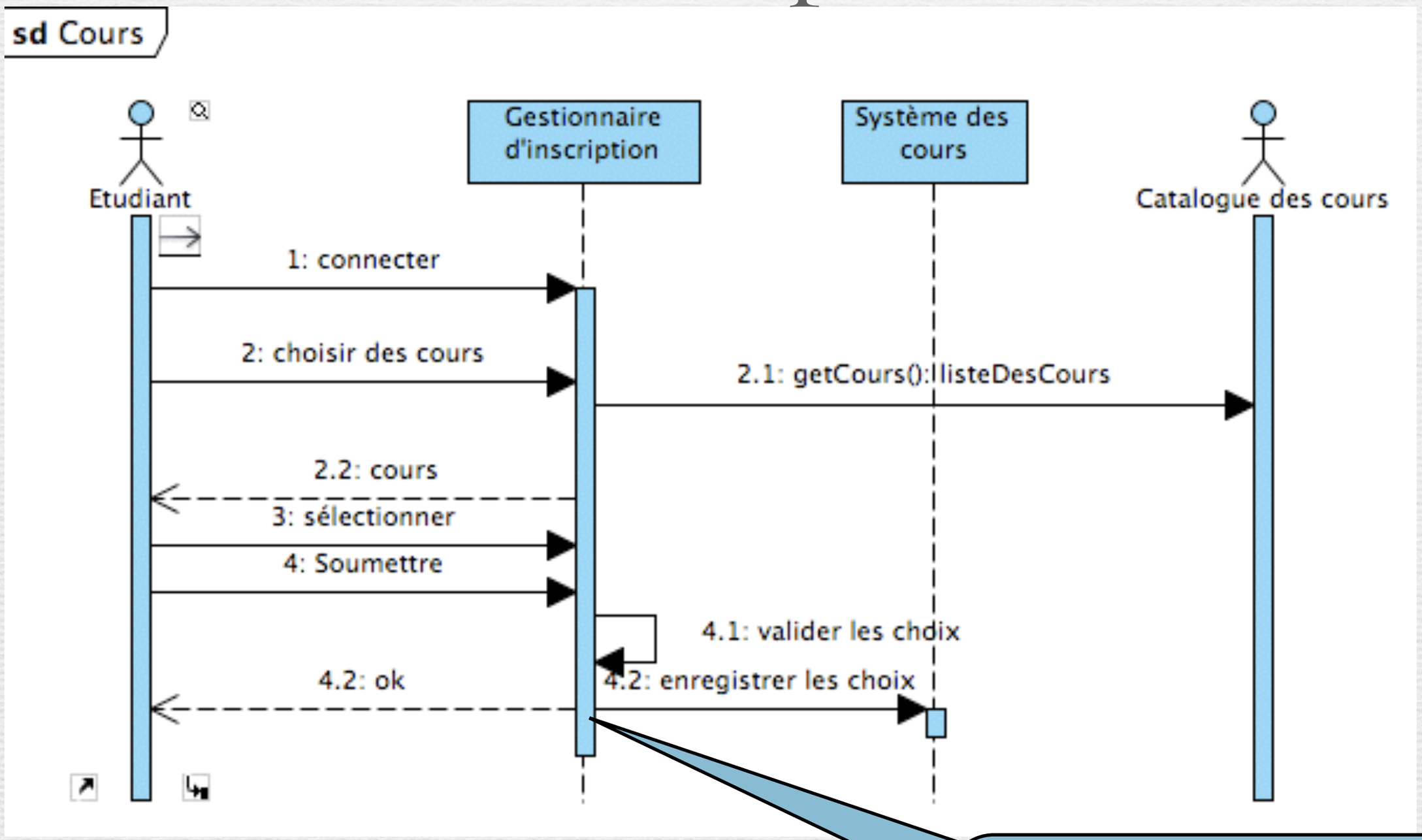

Gestionnaire d'inscription
Système des cours
Catalogu
er
cours
2.1: getCours(): listeDesCours
rs
ner
re
4.1: valider les choix
4.2: enregistrer les choix
4.2.1:
Edt
5: getHoraires()
Création

Destruction

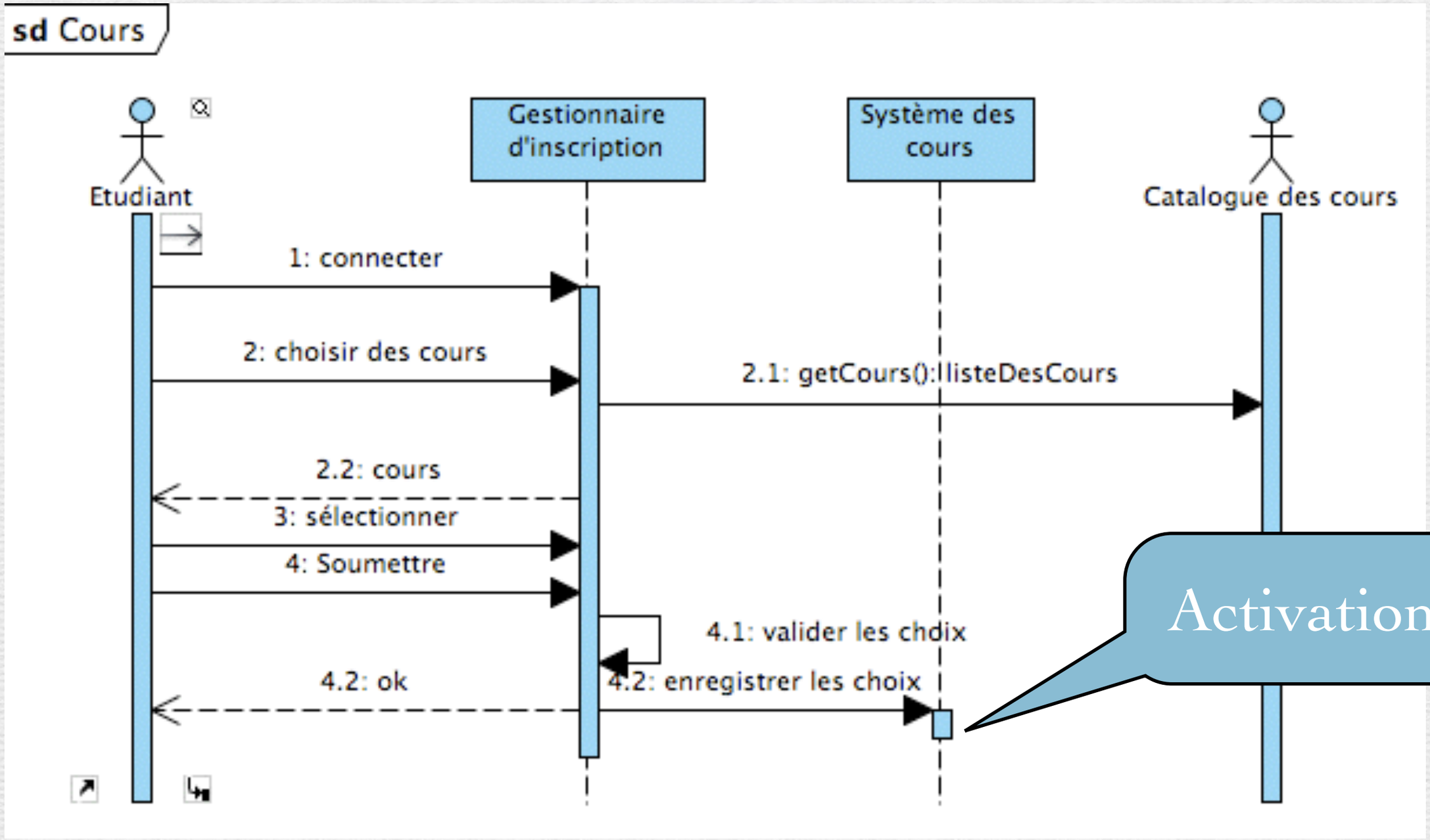




Temps

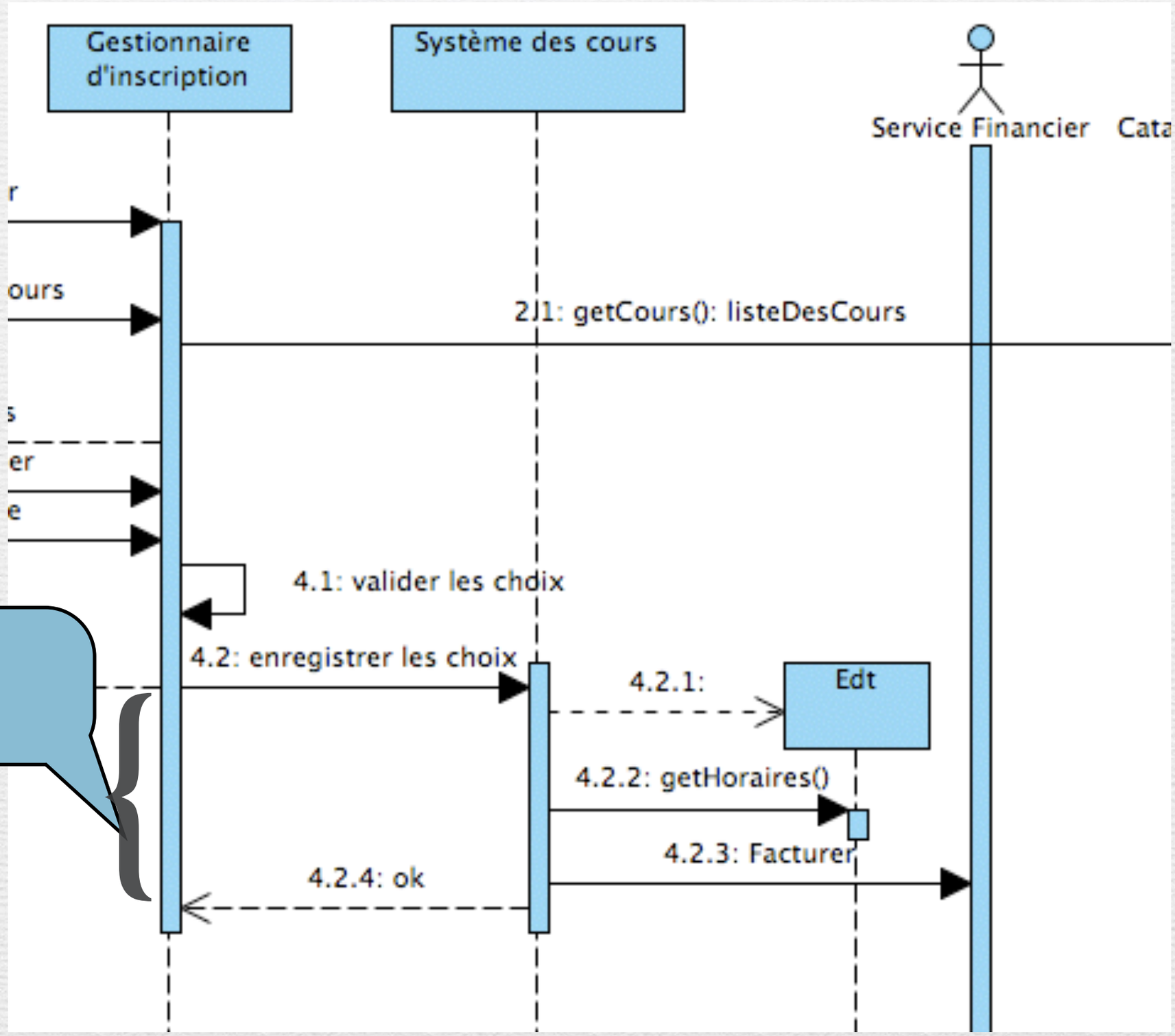

sd Cours
Etudiant
Gestionnaire d'inscription
Système des cours
Catalogue des cours
1: connecter
2: choisir des cours
2.1: getCours(): listeDesCours
2.2: cours
3: sélectionner
4: Soumettre
4.1: valider les choix
4.2: ok
4.2: enregistrer les choix
En même temps

Activation



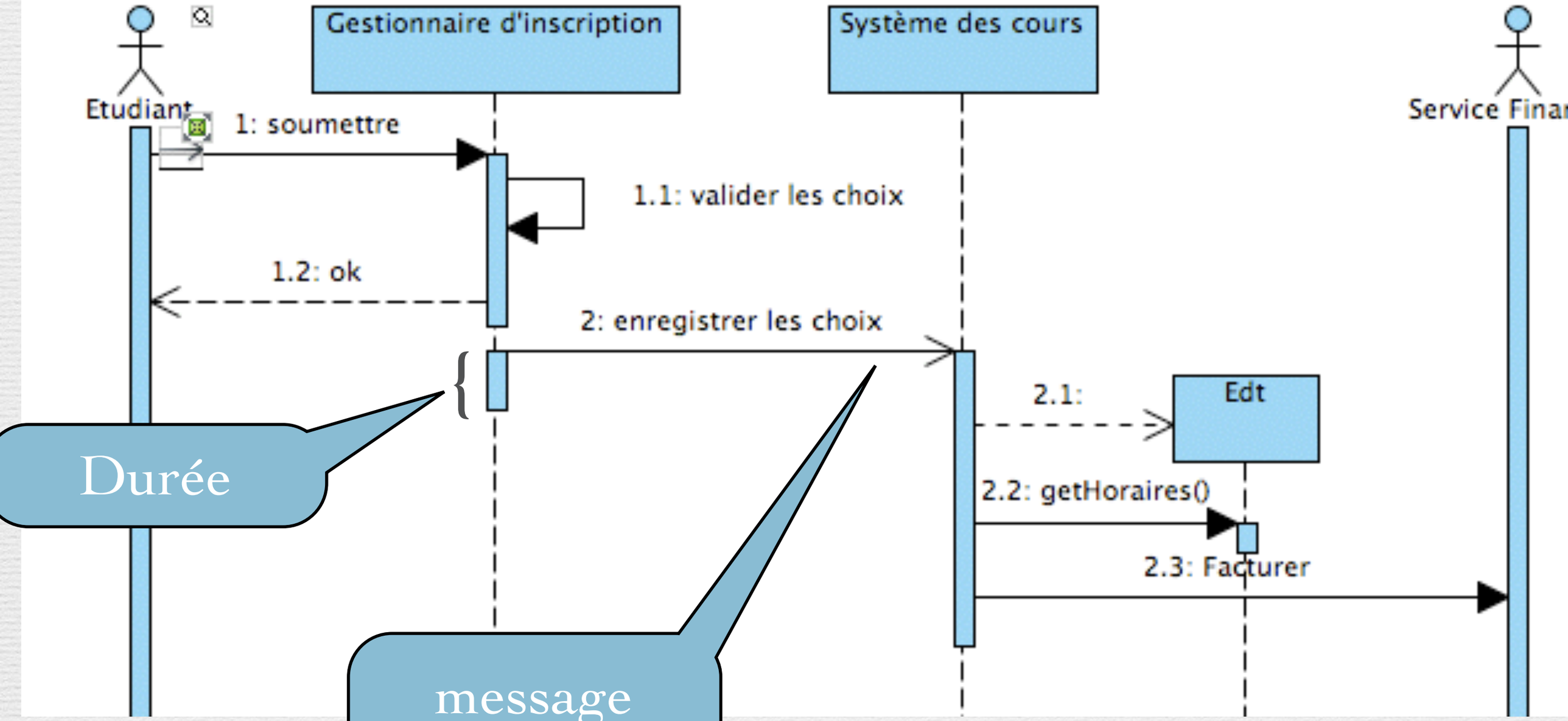
- Une *activation* représente le temps durant lequel un objet est actif, c'est à dire en train d'exécuter une opération

Synchrone

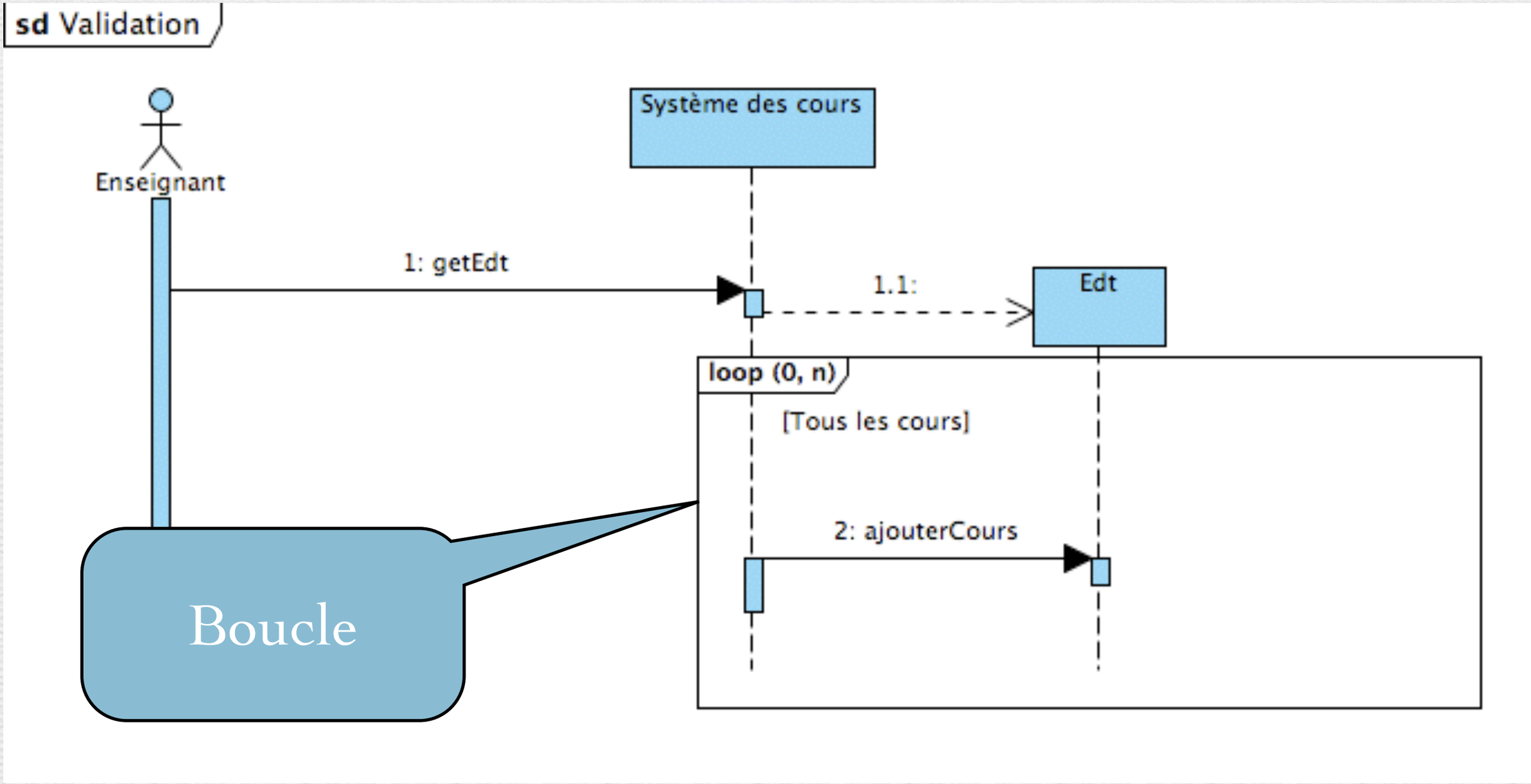


Durée

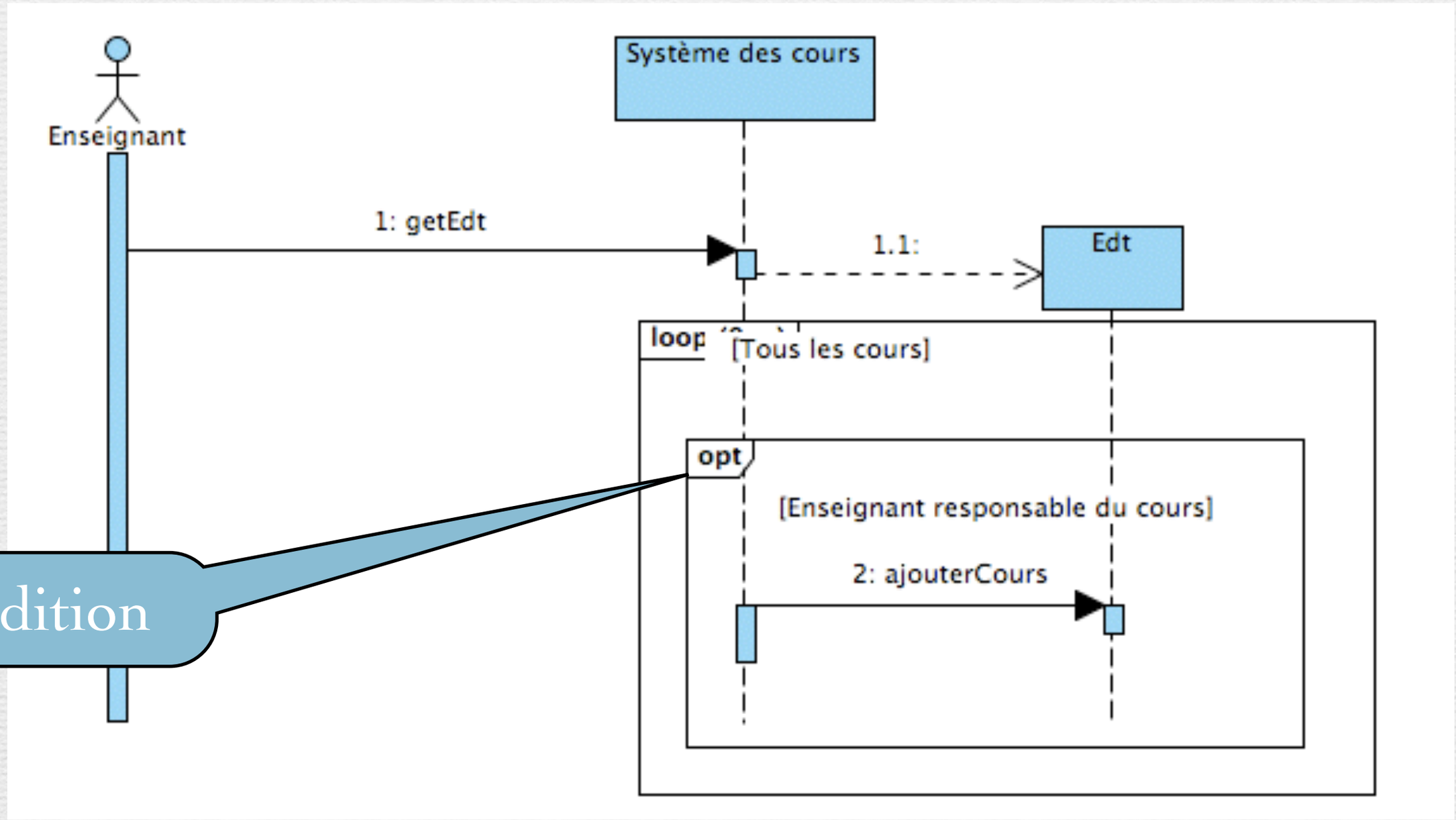
Asynchrone


Gestionnaire d'inscription
Système des cours
Etudiant
Service Finar
1: soumettre
1.1: valider les choix
1.2: ok
2: enregistrer les choix
2.1:
Edt
2.2: getHoraires()
2.3: Facturer
Durée
message
asynchrone

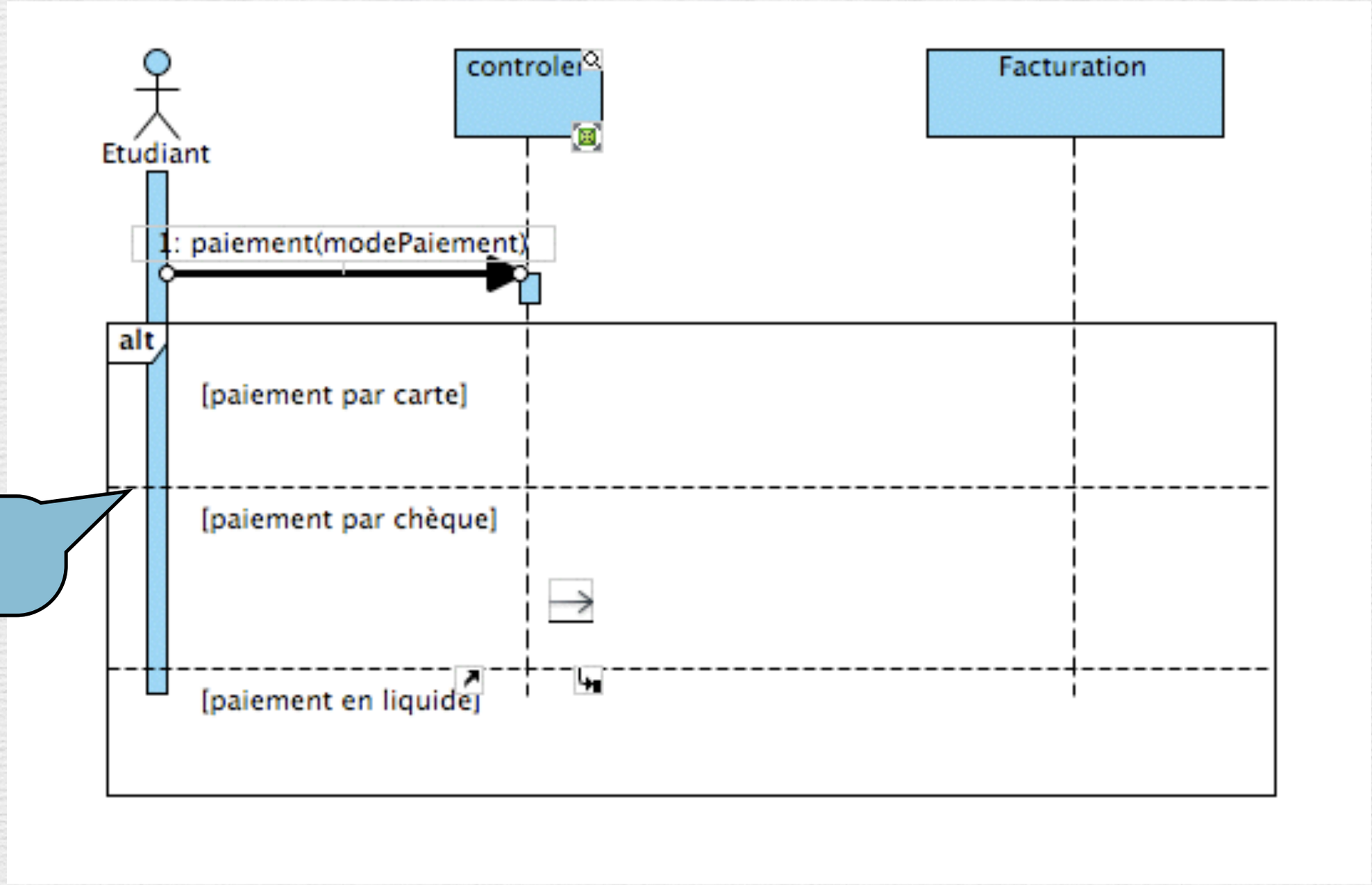
Boucle


sd Validation
Enseignant
Système des cours
1: getEdt
1.1:
Edt
loop (0, n)
[Tous les cours]
2: ajouterCours
Boucle

Conditions

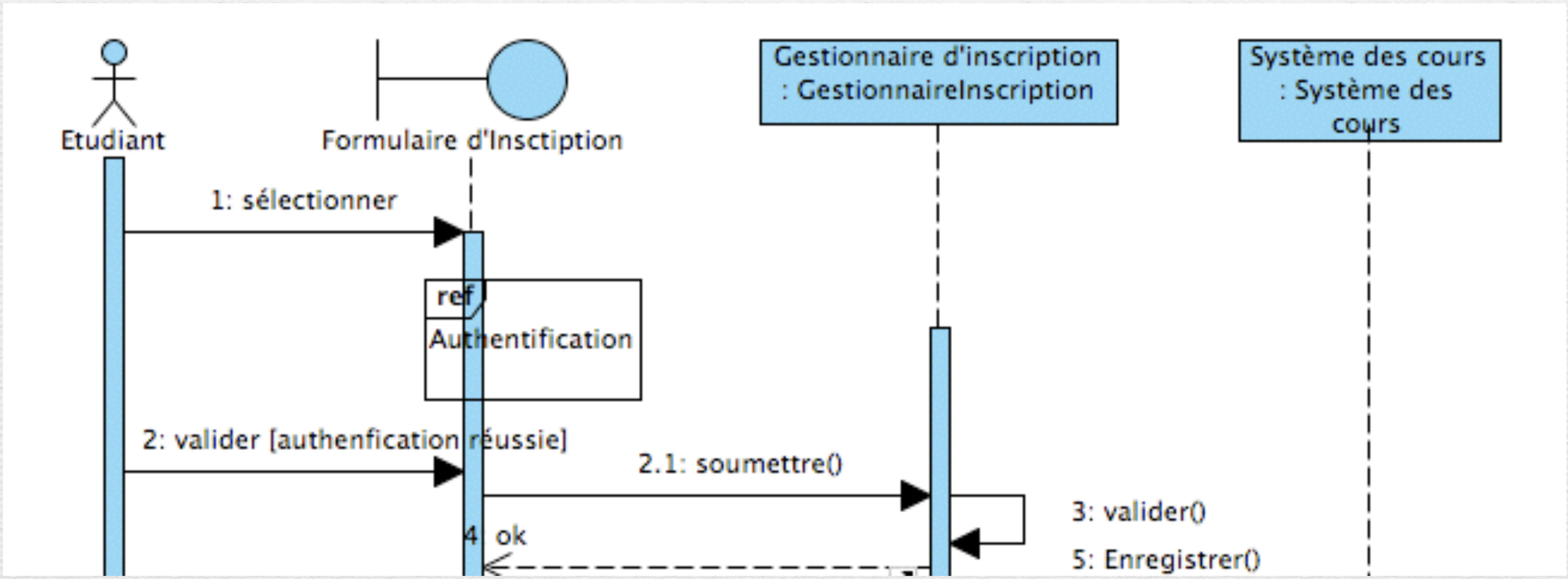

Enseignant
Système des cours
1: getEdt
1.1:
Edt
loop
[Tous les cours]
opt
[Enseignant responsable du cours]
2: ajouterCours
Condition

Conditions

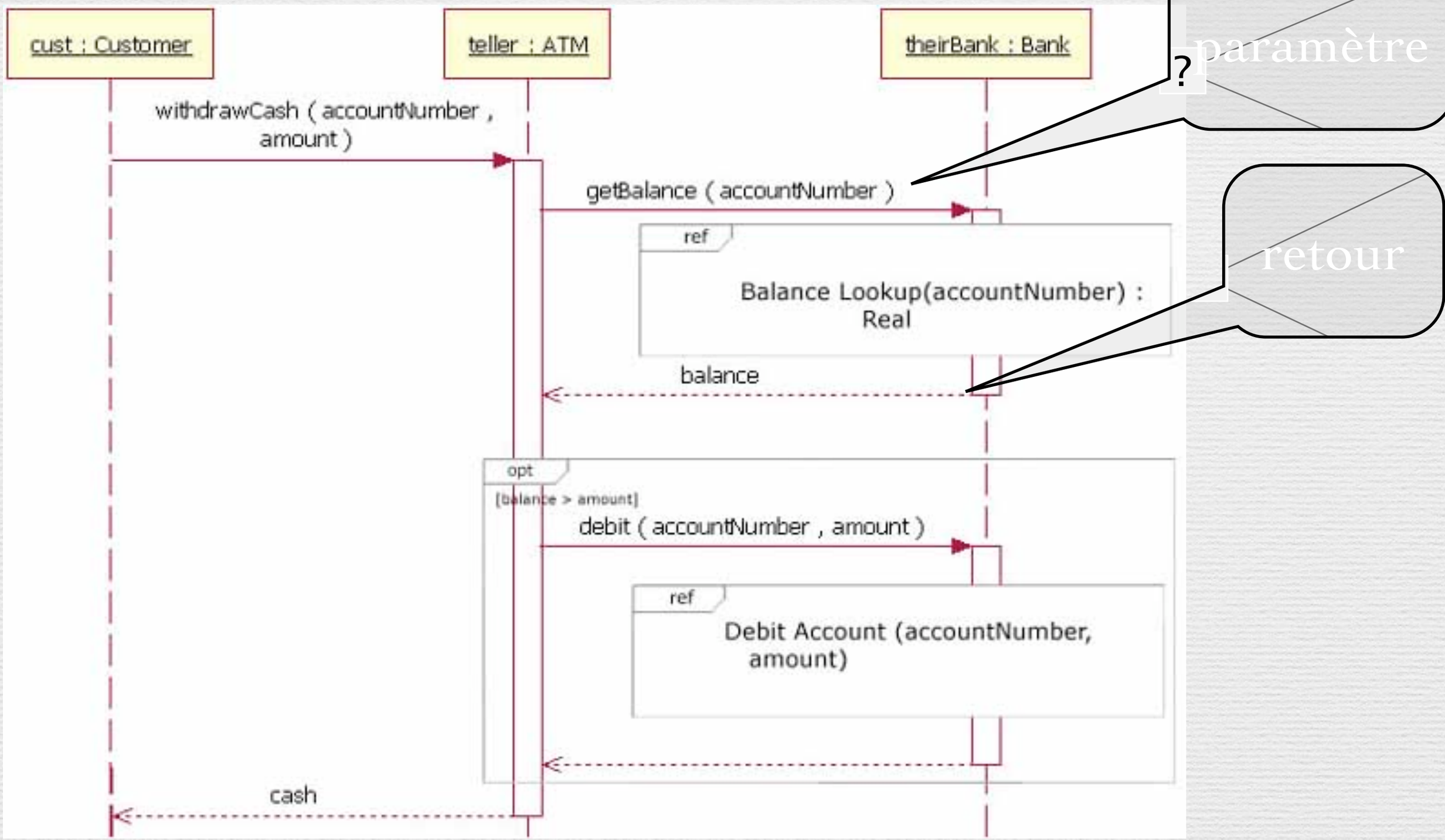


Condition

Référence(1)


Etudiant
Formulaire d'Insctiption
Gestionnaire d'inscription
: GestionnaireInscription
Système des cours
: Système des
cours
1: sélectionner
ref
Authentification
2: valider [authenfication réussie]
2.1: soumettre()
3: valider()
4 ok
5: Enregistrer()

Référence(2)


cust : Customer
teller : ATM
theirBank : Bank
withdrawCash (accountNumber , amount)
getBalance (accountNumber)
ref
Balance Lookup(accountNumber) : Real
balance
opt
[balance > amount]
debit (accountNumber , amount)
ref
Debit Account (accountNumber, amount)
cash
?
paramètre
retour

Bonnes pratiques

Any fool can write code that a computer can understand.
Good programmers write code that humans can understand.
Martin Fowler

L'art du «Codage»

«Conventional wisdom says that once a project is in the coding phase, the work is mostly mechanical, transcribing the design into executable statements. We think that this attitude is the single biggest reason that many programs are ugly, inefficient, poorly structured, unmaintainable, and just plain wrong.
Coding is not mechanical. If it were, all the CASE tools that people pinned their hopes on in the early 1980s would have replaced programmers long ago. There are decisions to be made every minute—decisions that require careful thought and judgment if the resulting program is to enjoy a long, accurate, and productive life.»

Hunt, Thomas «The pragmatic Programmer»

1. Eviter la duplication

2. Composition versus Héritage

3. Optimisation

4. Programmation par coïncidence

5. Estimation des algorithmes

6. Point de vue sur le «refactoring»

http://tck.io/posts/sharing_models.html

Ecrire du bon code : Don’t Repeat Yourself (DRY)

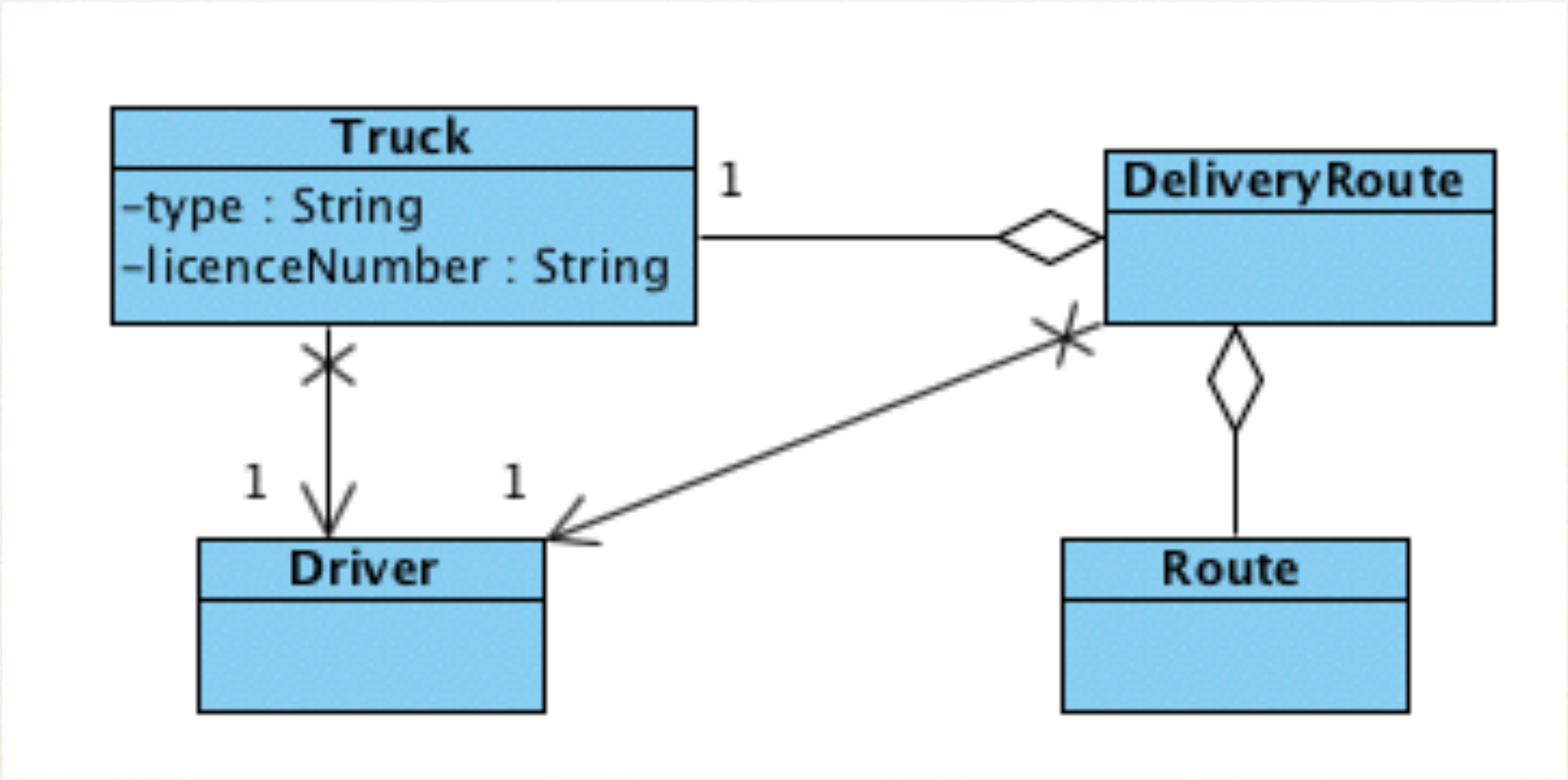
> Every piece of knowledge must have a single, unambiguous, authoritative representation within a system

Causes de duplication des codes :
- Imposition a priori de l’environnement,
- Inattention,
- Facilité,
- Multiplicité des développeurs

DRY (1) : Des exemples de duplications imposées et des solutions

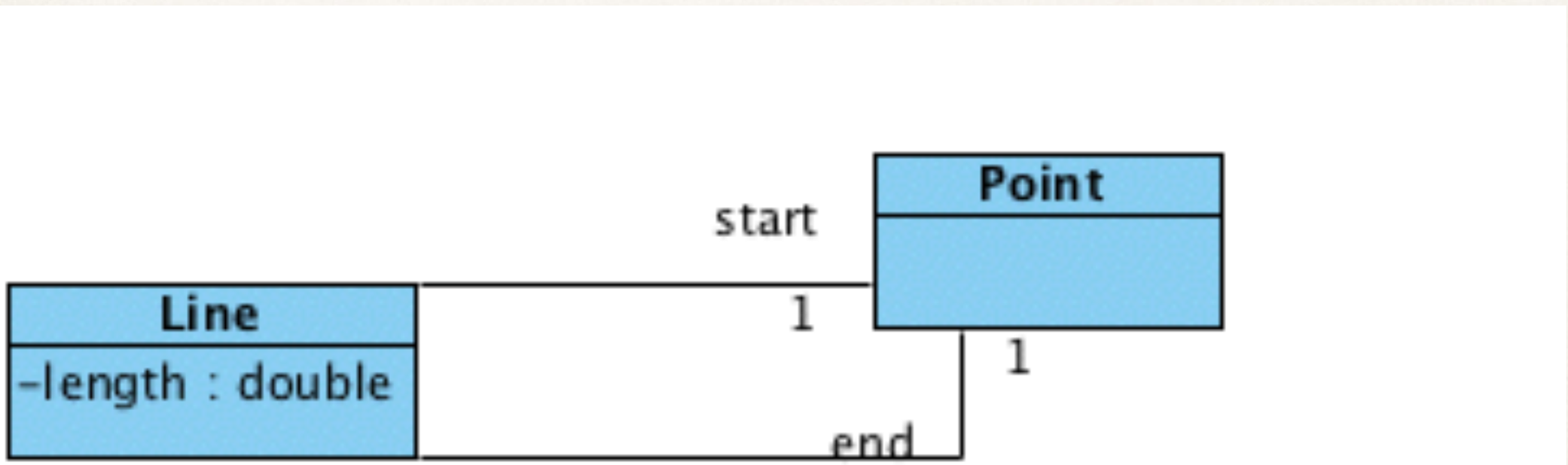
- Documentation du code
  - => ne garder les commentaires détaillés que pour le haut niveau.
- Multiples représentations d'une information (coté client et serveur par exemple, une classe miroir d'une table dans la BD) => des filtres, des générateurs de code, metadata et générateur, génération de la classe à partir de la BDD ou du schéma, ou du modèle,...
- Les langages forcent des duplications : utiliser des outils!

Dry (2) : Des exemples de duplication par inattention et des solutions



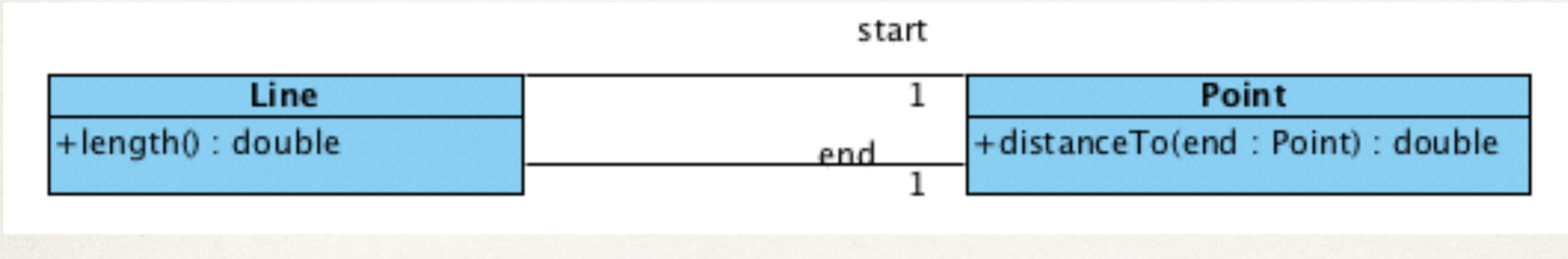
- Que faut-il modifier pour changer un chauffeur ? N'y a t'il pas une connaissance dupliquée?

Dry (2) : Des exemples de duplications par inattention et des solutions



- Qu'est-ce qui est dupliqué?

Dry (2) : Des exemples de duplications par inattention et des solutions



- return start.distanceTo(end);

Ecrire du bon code : Préférer la composition à l'héritage

> Inheritance is a powerful way to achieve code reuse, but it is not always the best tool for the job. Used inappropriately, it leads to fragile software. It is safe to use inheritance within a package, where the subclass and the superclass implementations are under the control of the same programmers. It is also safe to use inheritance when extending classes specifically designed and documented for extension. Inheriting from ordinary concrete classes across package boundaries, however, is dangerous.
>
> Joshua Bloch

http://verraes.net/2014/05/final-classes-in-php/

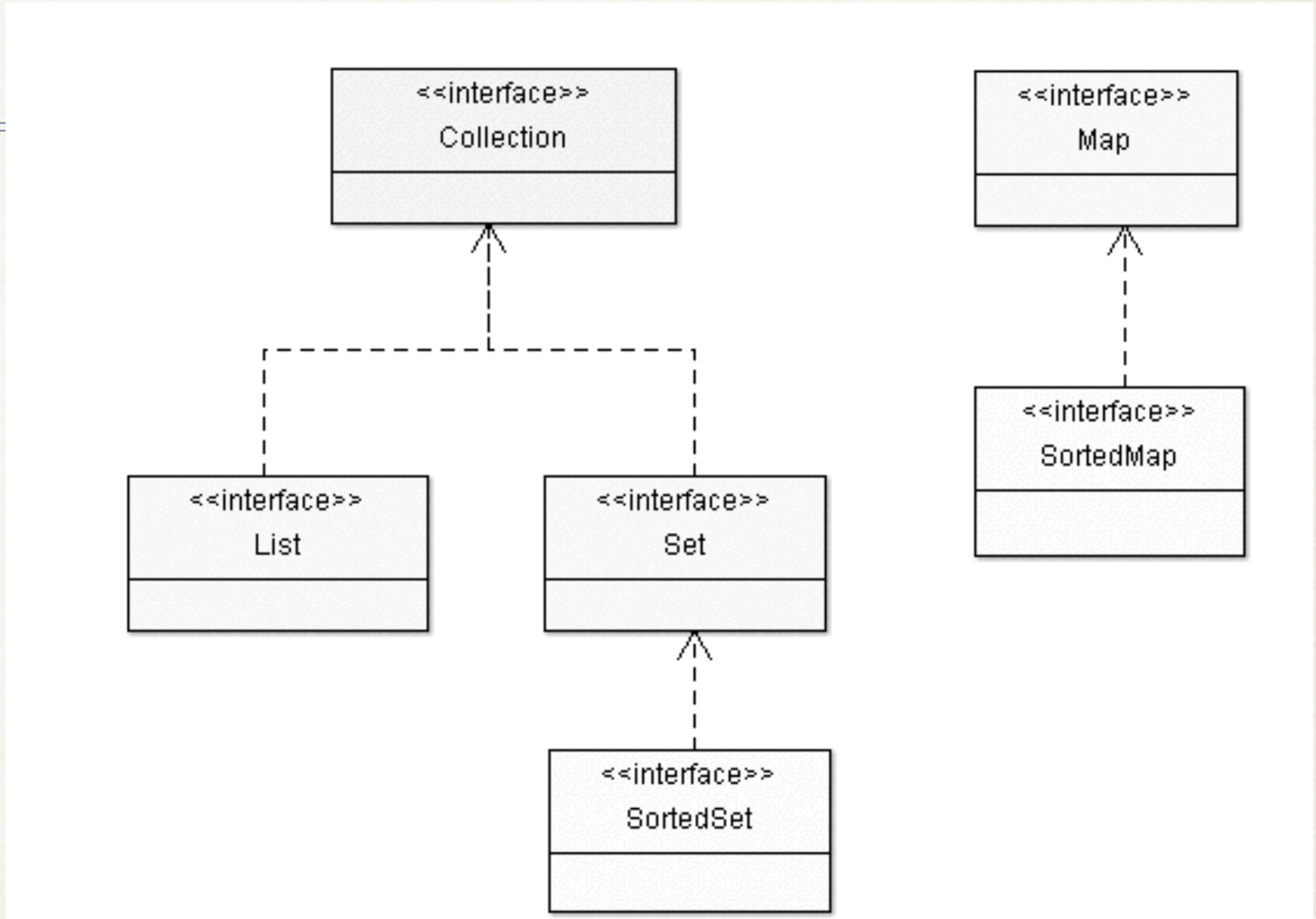
Héritage & Composition

- Préférer la composition à l'héritage
  - L'héritage a été mis en avant pour la réutilisation
    - Trop !
  - Souvent, l'héritage est rigide et la composition est souple

Composition

- Method of reuse in which new functionality is obtained by creating an object *composed of* other objects
- The new functionality is obtained by delegating functionality to one of the objects being composed
- Sometimes called *aggregation* or *containment*, although some authors give special meanings to these terms

Inheritance vs Composition Example




```
public class HashSet<E>
extends AbstractSet<E>
implements Set<E>, Cloneable, Serializable
```

Inheritance vs Composition Example

- Suppose we want a variant of HashSet that keeps track of the number of attempted insertions. So we subclass HashSet as follows:

```
public class InstrumentedHashSet extends HashSet {
// The number of attempted element insertions
private int addCount = 0;

public InstrumentedHashSet() {super();}
public InstrumentedHashSet(Collection c) {super(c);}
public InstrumentedHashSet(int initCap, float loadFactor) {
    super(initCap, loadFactor);
}
```

http://uet.vnu.edu.vn/~chauttm/e-books/java/Effective.Java.2nd.Edition.May.2008.3000th.Release.pdf

Inheritance vs Composition Example (Continued)

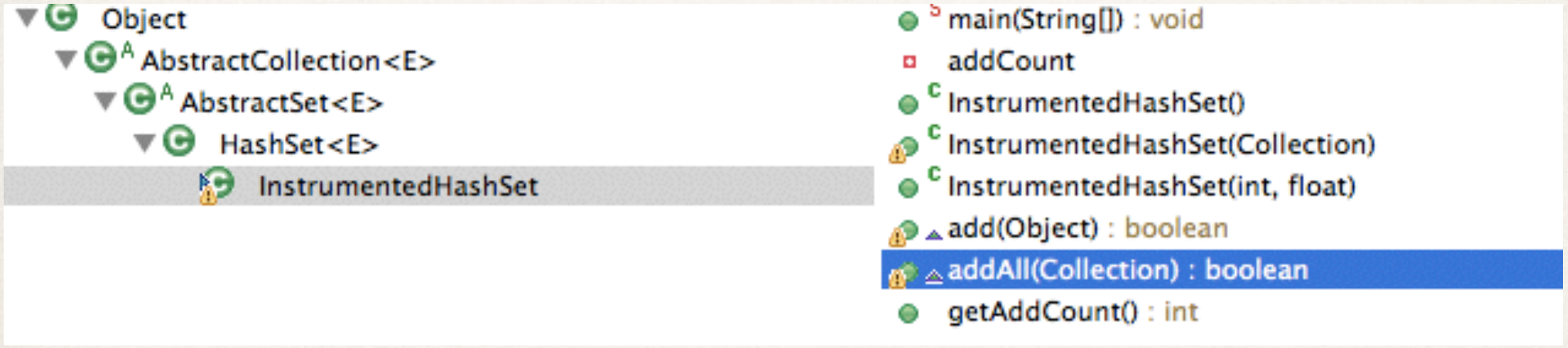
```
public boolean add(Object o) {
    addCount++;
    return super.add(o);
}
    public boolean addAll(Collection c) {
        addCount += c.size();
        return super.addAll(c);
    }

    public int getAddCount() {
        return addCount;
    }
}
```

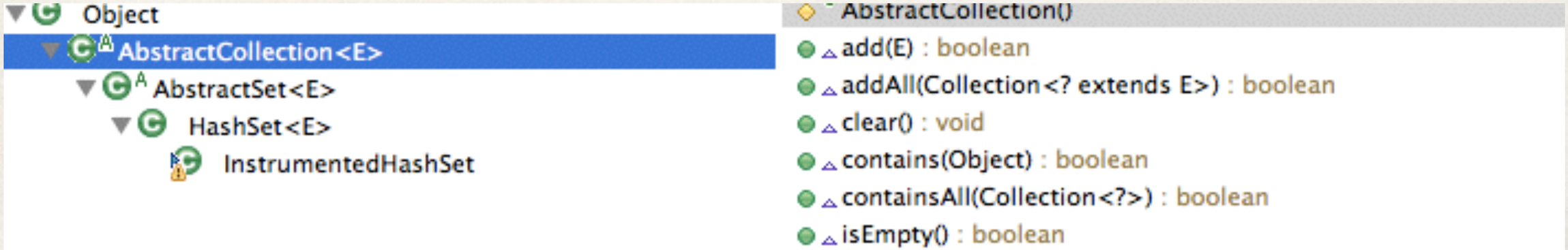
http://uet.vnu.edu.vn/~chauttm/e-books/java/Effective.Java.2nd.Edition.May.2008.3000th.Release.pdf

Inheritance vs Composition Example (Continued)

```
public static void main(String[] args) {
    InstrumentedHashSet s = new InstrumentedHashSet();
    s.addAll(Arrays.asList(new String[] {"Snap","Crackle","Pop"}));
    System.out.println(s.getAddCount());
}
```

- Let's test it! **6, why ??**





Inheritance vs Composition Example (Continued)

```
public static void main(String[] args) {
    InstrumentedHashSet s = new InstrumentedHashSet();
    s.addAll(Arrays.asList(new String[] {"Snap","Crackle","Pop"}));
    System.out.println(s.getAddCount());
}
```

- Let's test it! 6, why ??

```
144:    public boolean addAll(Collection<? extends E> c)
145:    {
146:      Iterator<? extends E> itr = c.iterator();
147:      boolean modified = false;
148:      int pos = c.size();
149:      while (--pos >= 0)
150:        modified |= add(itr.next());
151:      return modified;
152:    }
```

Inheritance vs Composition Example (Continued)

- L'implementation des superclasses a affecté l'opération de la sous-classe.
- Une autre approche est d'utiliser la composition : La nouvelle classe «InstrumentedSet» n'est plus une sorte de «Set» mais est composée d'un ensemble d'objet.
- Cette classe « InstrumentedSet» duplique l'interface «Set», mais toutes les opérations sont déléguées à l'objet ensemble contenu.
- *InstrumentedSet is known as a **wrapper** class, since it wraps an instance of a Set object.*

http://uet.vnu.edu.vn/~chauttm/e-books/java/Effective.Java.2nd.Edition.May.2008.3000th.Release.pdf

Inheritance vs Composition Example (Continued)

```
public class InstrumentedSet implements Set {
    private final Set s;
    private int addCount = 0;

    public InstrumentedSet(Set s) {this.s = s;}

public boolean add(Object o) {
    addCount++;
    return s.add(o);
}

public boolean addAll(Collection c) {
    addCount += c.size();
    return s.addAll(c);
}

public int getAddCount() {return addCount;}
```

```
// Forwarding methods (the rest of the Set interface methods)
public void clear() { s.clear(); }
public boolean contains(Object o) { return s.contains(o); }
public boolean isEmpty() { return s.isEmpty(); }
public int size() { return s.size(); }
public Iterator iterator() { return s.iterator(); }
public boolean remove(Object o) { return s.remove(o); }
public boolean containsAll(Collection c)
    { return s.containsAll(c); }
public boolean removeAll(Collection c)
    { return s.removeAll(c); }
public boolean retainAll(Collection c)
    { return s.retainAll(c); }
public Object[] toArray() { return s.toArray(); }
public Object[] toArray(Object[] a) { return s.toArray(a); }
public boolean equals(Object o) { return s.equals(o); }
public int hashCode() { return s.hashCode(); }
public String toString() { return s.toString(); }
}
```

Inheritance vs Composition Example (Continued)

```
public static void main(String[] args) {
    InstrumentedSet s1 = new InstrumentedSet(new HashSet());
    s1.addAll(Arrays.asList(new String[] {"Snap","Crackle","Pop"}));
    System.out.println(s1 + " " + s1.getAddCount());
}
```

```
[Pop, Snap, Crackle] 3
```

Inheritance vs Composition Example (Continued)

```
public static void main(String[] args) {
    List list = new ArrayList();
    InstrumentedSet s1 = new InstrumentedSet(new TreeSet(list));
    s1.addAll(Arrays.asList(new String[] {"Snap","Crackle","Pop"}));
    System.out.println(s1 + " " + s1.getAddCount());
}
```

```
[Crackle, Pop, Snap] 3
```

Inheritance vs Composition Example (Continued)

- Note several things:
  - This class is a Set
  - It has one constructor whose argument is a Set
  - The contained Set object can be an object of any class that implements the Set interface (and not just a HashSet)
  - This class is very flexible and can wrap any preexisting Set object
- Example:

```
List list = new ArrayList();
Set s1 = new InstrumentedSet(new TreeSet(list));

int capacity = 7;
float loadFactor = .66f;
Set s2 = new InstrumentedSet(new HashSet(capacity, loadFactor));
```

Advantages/Disadvantages of Inheritance

- Advantages:
  - New implementation is easy, since most of it is inherited
  - Easy to modify or extend the implementation being reused

- Disadvantages:
  - Breaks encapsulation, since it exposes a subclass to implementation details of its superclass
  - "White-box" reuse, since internal details of superclasses are often visible to subclasses
  - Subclasses may have to be changed if the implementation of the superclass changes

Advantages/Disadvantages Of Composition

- **Avantages**:
  - Objets contenus sont accessibles par la classe contenant uniquement à travers leurs interfaces
  - Réutilisation «Boîte noire» car les détails internes des objets contenus ne sont pas visibles : Bonne encapsulation
  - Réduit les dépendances de mise en œuvre
  - Chaque classe se concentre sur sa propre tâche
  - La composition peut être définie dynamiquement lors de l'exécution à travers des objets qui acquièrent des références à d'autres objets du même type
- **Inconvénients**:
  - Les systèmes résultant ont tendance à avoir plus d'objets

http://uet.vnu.edu.vn/~chauttm/e-books/java/Effective.Java.2nd.Edition.May.2008.3000th.Release.pdf

Coad's Rules of Using Inheritance

- Use inheritance only when all of the following criteria are satisfied:
  - A subclass expresses "is a special kind of" and not "is a role played by a"
  - An instance of a subclass never needs to become an object of another class
  - A subclass **extends**, rather than **overrides** or **nullifies**, the responsibilities of its superclass
  - A subclass does not extend the capabilities of what is merely an utility class

http://uet.vnu.edu.vn/~chauttm/e-books/java/Effective.Java.2nd.Edition.May.2008.3000th.Release.pdf

Premature Optimization

```php
if (isset($frm['title_german'][strcspn($frm['title_german'], '<>')]))
{
    // ...
}
```

Optimiser les points vraiment utiles !
Ne mettez pas en péril la lisibilité et la maintenance de votre code pour de pseudo micro-optimisations!
Ne gâcher pas votre temps!

http://nikic.github.io/2011/12/27/Dont-be-STUPID-GRASP-SOLID.html

Eviter la programmation par coïncidence

«Developers who don't actively think about their code are programming by coincidence—the code might work, but there's no particular reason why »

Hunt, Thomas «The pragmatic Programmer»

- «Mon code est tombé en marche, enfin !»...
- coïncidence? Accidents d'implémentation ?

```
paint(g);
invalidate();
validate();
revalidate();
repaint();
paintImmediately(r);
```

```
public void reinit(){
    size(650, 550);
    background(255, 255, 255);
    image(loadImage("background.png"), 0, 0);
}

public void setup() {
  frameRate(4);
  reinit();
  memoriseCarts() ;....
  // on veut garder la main sur le jeu c'est mousePressed qui stimule des redraw
  noLoop();
}

public void draw() {
 afficherJoueurs();
 //afficherCartes();
}
```

Eviter la programmation par coïncidence

«Developers who don't actively think about their code are programming by coincidence—the code might work, but there's no particular reason why »

Hunt, Thomas «The pragmatic Programmer»

- Ben, chez moi, ça marche...
- coïncidence? Accidents de contexte ? Hypothèses implicites ?

- «A oui! ça ne peut pas marcher parce que tu n'as pas mis le code sous bazarland»

Penser à «Estimer» vos algorithmes

- Comment le programme se comportera s'il y a 1000 enregistrements? 1 000 000? Quelle partie optimiser?

- Quelles dépendances entre par exemple la taille des données (longueur d'une liste, par exemple) et le temps de calcul? et la mémoire nécessaire?

- S'il faut 1s pour traiter 100 éléments, pour en traiter 1000, en faut-il :
  - 1, (O(1)) : temps constant
  - 10 (O(n)) : temps linéaire
  - 3 (O(log(n)) : temps logarithmique
  - 100 ($O(n^2)$)
  - 10^{263} ($O(e^n)$) : temps exponentiel

Penser à «Estimer» vos algorithmes

- Tester vos estimations

- Optimiser si cela est **utile** et en tenant compte du **contexte**.

«Refactoring» ou l'art du «jardinage logiciel»

« Rather than construction, software is more like gardening—it is more organic than concrete. You plant many things in a garden according to an initial plan and conditions. Some thrive, others are destined to end up as compost. You may move plantings relative to each other to take advantage of the interplay of light and shadow, wind and rain. Overgrown plants get split or pruned, and colors that clash may get moved to more aesthetically pleasing locations. You pull weeds, and you fertilize plantings that are in need of some extra help. You constantly monitor the health of the garden, and make adjustments (to the soil, the plants, the layout) as needed»

Hunt, Thomas «The pragmatic Programmer»

La théorie des fenêtres cassées ou Eviter l'entropie du système

Codez toujours en pensant que celui qui maintiendra votre code est un psychopathe qui connait votre adresse.

Martin Golding

Ne pas laisser de fenêtre cassée :

- Réparer les codes
- Corriger les design dès que les défauts sont détectés.
- Si vous ne pouvez pas régler le problème, le circonscrire : annoter le code, noter «Not Implemented», ...
- Mais ne laisser pas des codes se déteriorer ou c'est l'ensemble de l'application qui en pâtira.

- Etude urbaine en 1982 : http://en.wikipedia.org/wiki/Broken_windows_theory

«Refactoring» : quand ?

- Pour éliminer les «fenêtres cassées»
- Pour améliorer le design : duplication (DRY), couplage, performance, ...
- Pour ajuster en fonction des besoins et des demandes de changements

- **Souvent, dès le début**

- Réfléchissez comme un jardinier pas comme un maçon...

http://www.jhackens.com/2006/10/jardinier/

«Refactoring» : comment ?

- Utilisez des outils pour identifier les changements (cf. cours Métriques)
- Utilisez des outils pour factoriser (Par exemple, Eclipse et les outils d'extractions de méthodes, ...)

- **Organiser le refactoring**
  - Planifiez, Mettez des priorités, Mémorisez les changements à faire
  - Soyez sûr de vos tests avant de refactoriser
  - Progressez pas à pas

http://www.jhackens.com/2006/10/jardinier/

Principes SOLID

S.O.L.I.D : l'essentiel !

Single responsibility principle (SRP) : une classe n'a qu'une seule responsabilité (ou préoccupation).

- **Open/closed principle (OCP)** : une classe doit être ouverte à l'extension (par héritage, par exemple) mais fermé à la modification (attributs privés, par exemple).

- **Liskov subtition principle (LSP)** : les objets d'un programme doivent pouvoir être remplacés par des instances de leurs sous-types sans «casser» le programme.

- **Interface segregation principle (ISP)** : il vaut mieux plusieurs interfaces spécifiques qu'une unique interface générique.

- **Dependency inversion principle (DIP)** : il faut dépendre des abstractions, pas des réalisations concrètes.

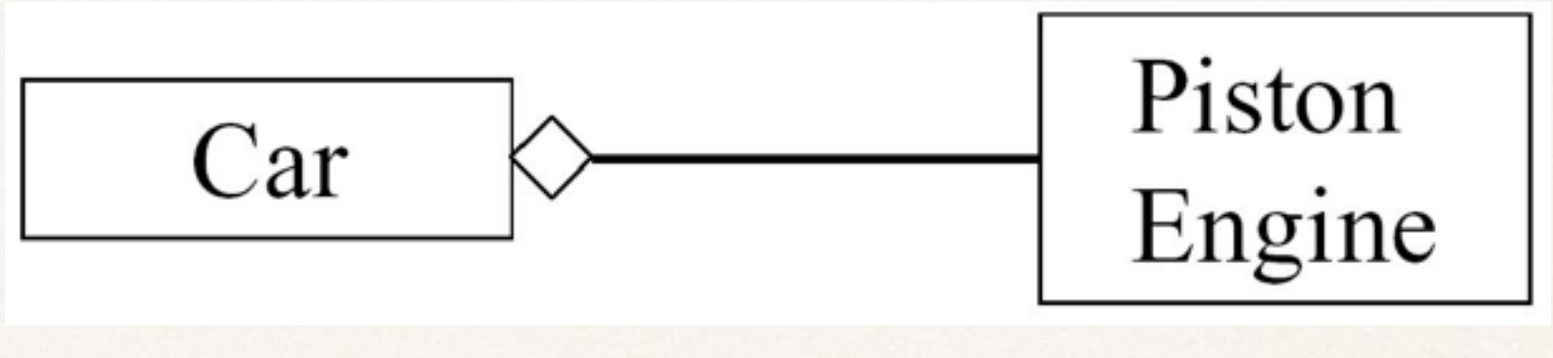
Principe ouvert / fermé
Open/Closed Principle (OCP)

> You should be able to extend a classes behavior, without modifying it.
> Robert C. Martin.

- **Les entités logicielles doivent être ouvertes à l'extension**
  - le code est extensible
- **mais fermées aux modifications**
  - Le code a été écrit et testé, on n'y touche pas.

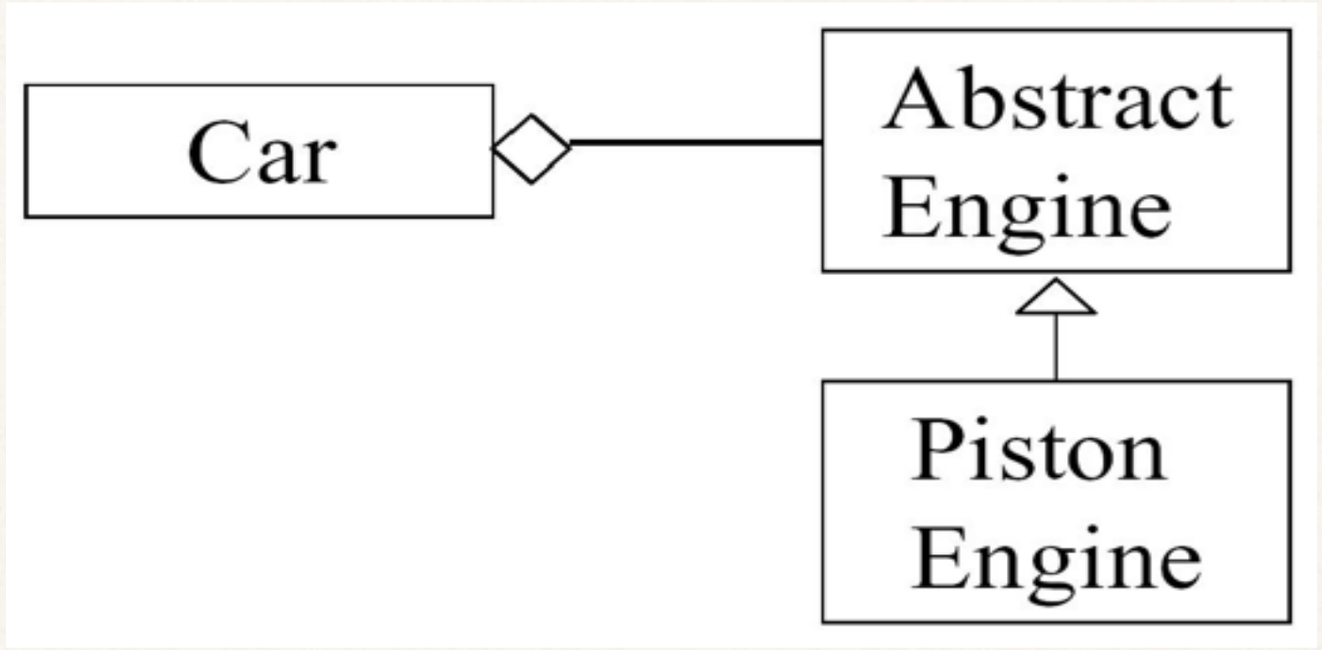
http://afsy.fr/avent/2013/02-principes-stupid-solid-poo

Open the door …



- Comment faire en sorte que la voiture aille plus vite à l'aide d'un turbo?
  - Il faut changer la voiture
    - avec la conception actuelle…

… But Keep It Closed!



- On retient :
  - Une classe ne doit pas dépendre d'une classe Concrète.
  - Elle peut dépendre d'une classe abstraite …
  - et utiliser le polymorphisme

The Open-Closed Principle(OCP) : allons plus loin (1)

- Le travail de cette méthode est de calculer le prix total d'un ensemble de «parties»

```
public double totalPrice(Part[] parts) {
        double total = 0.0;
        for (int i=0; i<parts.length; i++) {
                total += parts[i].getPrice();
        }
        return total;
}
```

The Open-Closed Principle(OCP) : allons plus loin (2)

- «But the Accounting Department decrees that motherboard parts and memory parts should have a premium applied when figuring the total price.»
- Que pensez-vous du code suivant?

```
public double totalPrice(Part[] parts) {
    double total = 0.0;
    for (int i=0; i<parts.length; i++) {
        if (parts[i] instanceof Motherboard)
            total += (1.45 * parts[i].getPrice());
        else if (parts[i] instanceof Memory)
            total += (1.27 * parts[i].getPrice());
        else
            total += parts[i].getPrice();
    }
    return total;
}
```

The Open-Closed Principle(OCP) : allons plus loin (3)

- Des exemples de classes *Part* et *ConcretePart*

```
// Class Part is the superclass for all parts.
public class Part {
      private double price;
      public Part(double price) {
          this.price = price;}
      public void setPrice(double price) {
          this.price = price;}
      public double getPrice() {
          return price;}
}

// Class ConcretePart implements a part for sale.
// Pricing policy explicit here!
public class ConcretePart extends Part {
      public double getPrice() {
      // return (1.45 * price); //Premium
      return (0.90 * price); //Labor Day Sale
      }
}
```

Mais si maintenant on veut modifier la politique de gestion des prix, par exemple en lisant dans une base de données, en modifiant les facteurs de calcul des prix ....

The Open-Closed Principle(OCP) : allons plus loin (4)

- Une meilleure idée est d'avoir une classe *PricePolicy* qui permettra de définir différentes politiques de prix:

```
// The Part class now has a contained PricePolicy object.
public class Part {
    private double price;
    private PricePolicy pricePolicy;

    public void setPricePolicy(PricePolicy pricePolicy) {
        this.pricePolicy = pricePolicy;}

    public void setPrice(double price) {this.price = price;}
    public double getPrice() {return pricePolicy.getPrice(price);}
}
```

The Open-Closed Principle(OCP) : allons plus loin (5)

```
/**
* Class PricePolicy implements a given price policy.
*/
public class PricePolicy {
      private double factor;

      public PricePolicy (double factor) {
      this.factor = factor;
      }

public double getPrice(double price) {return price * factor;}
}
```

D'autres politiques comme un calcul de la ristourne par «seuils» sont maintenant possibles ...

On pourrait aussi faire quoi, pour encore décroitre le couplage?

The Open-Closed Principle(OCP) : allons plus loin (6)

- With this solution we can dynamically set pricing policies at run time by changing the PricePolicy object that an existing Part object refers to

- Of course, in an actual application, both the price of a Part and its associated PricePolicy could be contained in a database

The Open-Closed Principle (OCP)

- Il est impossible que tous les éléments d'un système logiciel satisfasse l'OCP, mais l'objectif est de minimiser le nombre des éléments qui ne le satisfont pas.
- Le principe ouvert-fermé est vraiment au cœur de la conception OO.
- La conformité à ce principe donne un meilleur niveau de réutilisabilité et maintenabilité.

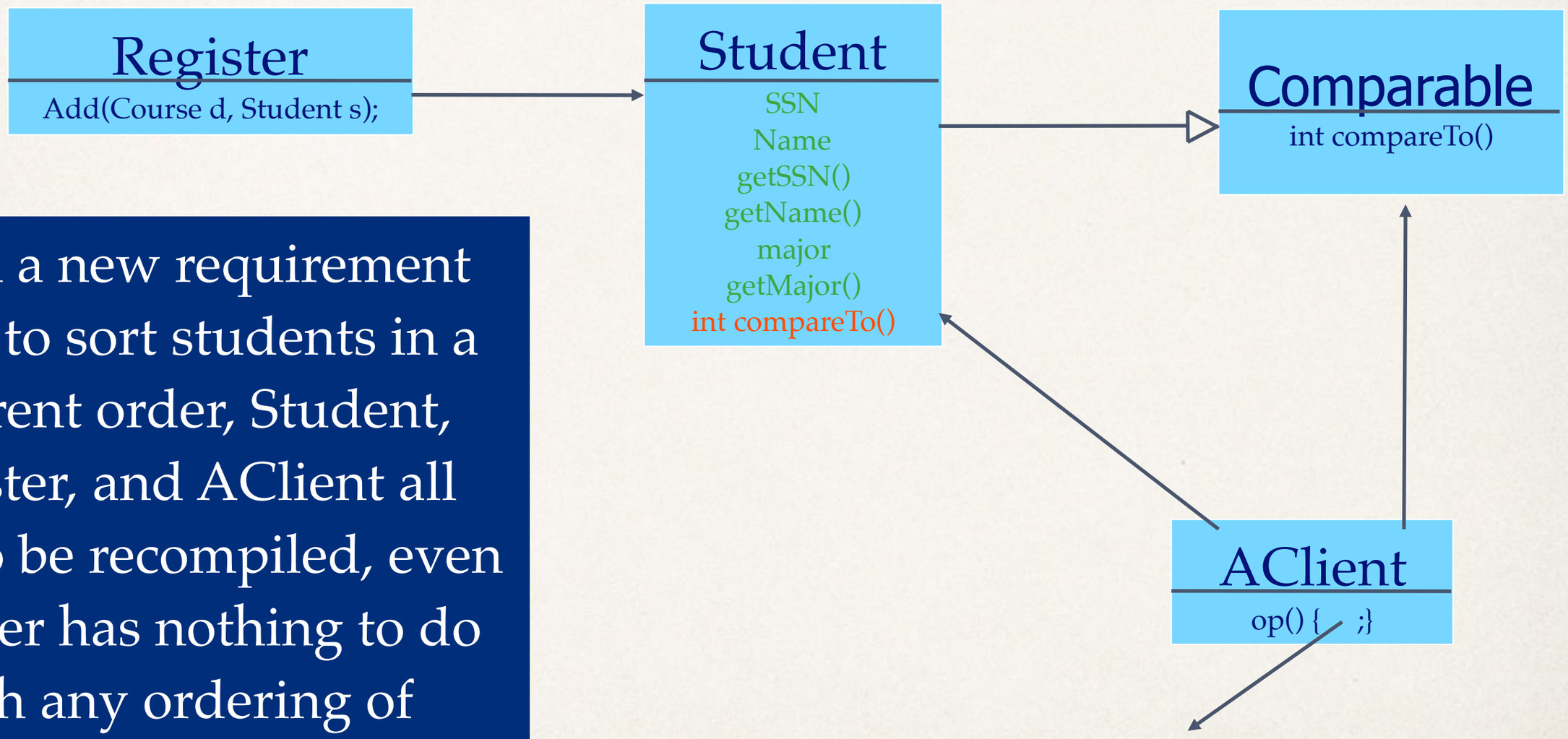
The single-responsibility principle

- **Example**:
  - Often we need to sort students by their name, or ssn. So one may make Class Student implement the Java Comparable interface.

```
class Student implements Comparable {
    int compareTo(Object o) { … }
};
```

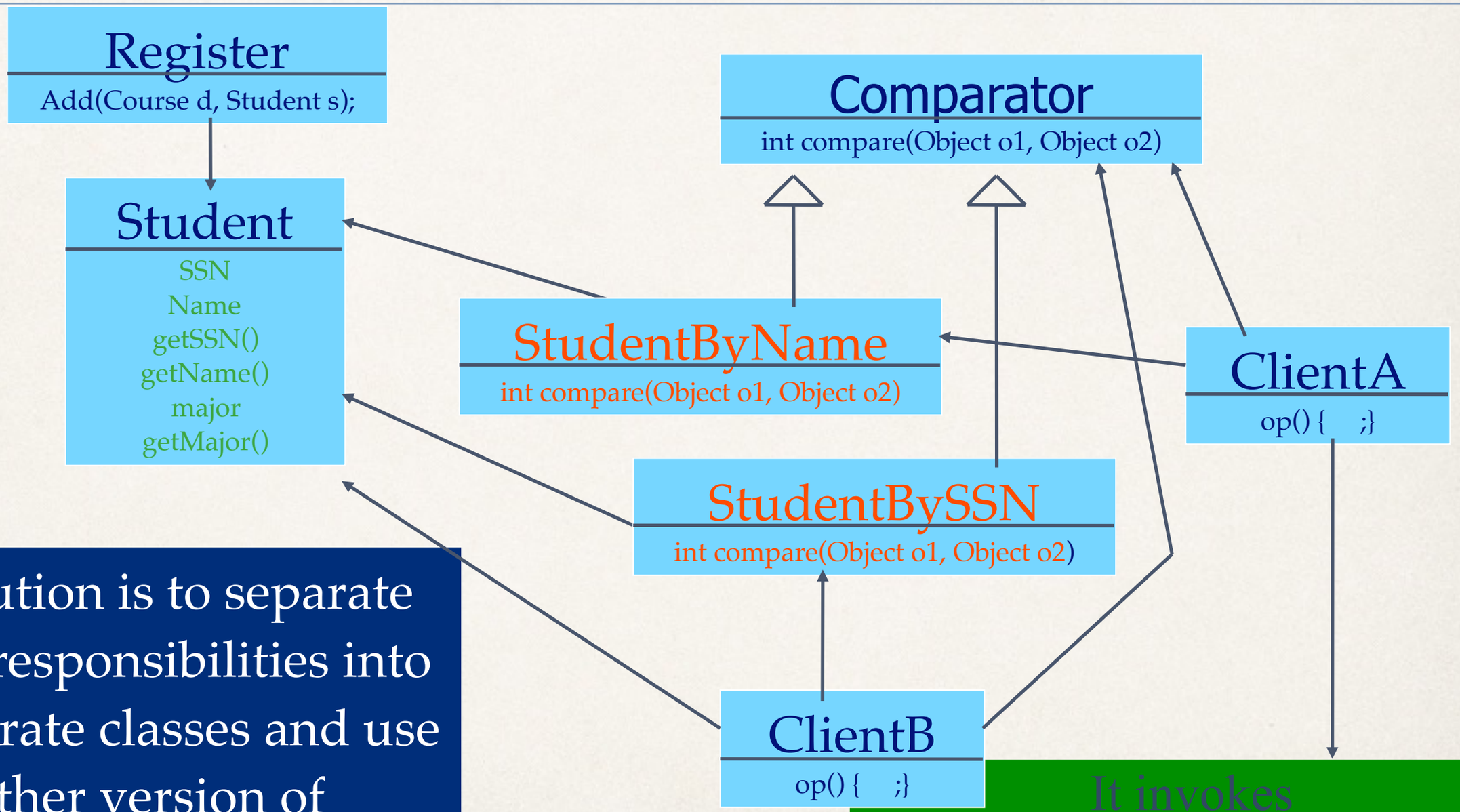
- **BUT**:
  - Student is a business entity, it does not know in what order it should be sorted since the order of sorting is imposed by the client of Student.
  - Worse: every time students need to be ordered differently, we have to recompile Student and all its client.
  - Cause of the problems: we bundled two separate responsibilities (i.e., student as a business entity with ordering) into one class – a violation of SRP

The single-responsibility principle



When a new requirement needs to sort students in a different order, Student, Register, and AClient all need to be recompiled, even Register has nothing to do with any ordering of Students.

The single-responsibility principle



The solution is to separate the two responsibilities into two separate classes and use another version of Collections.sort().

It invokes

Collections.sort(aListofStudents, new StudentByName());

www.cs.uofs.edu/~bi/2013f-html/se510/design-principles.ppt

Les codes : Classe Student

```
public class Student {

    private final String name;
    private final int section;

    // constructor
    public Student(String name, int section) {
        this.name = name;
        this.section = section;
    }
....}
```

```
Student alice  = new Student("Alice",  2);
Student bob    = new Student("Bob",    1);
Student carol  = new Student("Carol",  2);
Student dave   = new Student("Dave",   1);
Student[] students = {dave, bob,alice};
for (int i = 0; i < students.length; i++)
       System.out.println(students[i]);
```

Les codes : Comparateurs

```
Interface Comparator<T>
Type Parameters:
T - the type of objects that may be compared by this comparator

int compare(T o1, T o2)
```

Compares its two arguments for order. Returns a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second.

```
class ByName implements Comparator<Student> {
        public int compare(Student a, Student b) {
            return a.name.compareTo(b.name);
        }
```

```
class BySection implements Comparator<Student> {
        public int compare(Student a, Student b) {
            return a.section - b.section;
        }
```

```
Comparator<Student> byNameComparator =
    new ByName();
Comparator<Student> bySectionComparator=
    new BySection();
```

Les codes : Comparer des étudiants

```
Student[] students = {
            larry, kevin, jen, isaac, grant, helia,
            frank, eve, dave, carol, bob, alice
        };
```

```
// sort by name and print results
 Arrays.sort(students, byNameComparator);
 for (int i = 0; i < students.length; i++)
            System.out.println(students[i]);

// now, sort by section and print results
 Arrays.sort(students, bySectionComparator);
for (int i = 0; i < students.length; i++)
            System.out.println(students[i]);
```

Principe de substitution de Liskov
Liskov Substitution Principle (LSP)

- Les instances d'une classe doivent être remplaçables par des instances de leurs sous-classes sans altérer le programme.

Principe de substitution de Liskov

- « Si une propriété P est vraie pour une instance x d'un type T, alors cette propriété P doit rester vraie pour tout instance y d'un sous-type de T »

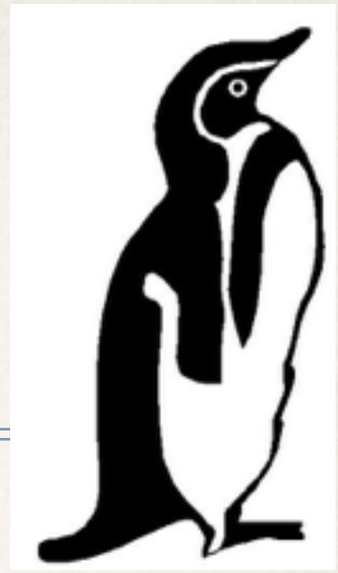
- Implications :
  - Le «contrat» défini par la classe de base (pour chacune de ses méthodes) doit être respecté par les classe dérivées
  - L'appelant n'a pas à connaître le type exact de la classe qu'il manipule : n'importe quelle classe dérivée peut être substituée à la classe qu'il utilise

- → Principe de base du polymorphisme :
  - Si on substitue une classe par une autre dérivée de la même hiérarchie: comportement (bien sûr) différent mais conforme.

Inheritance *Appears* Simple

```
class Bird {                              // has beak, wings,...
  public: virtual void fly();             // Bird can fly
};

class Parrot : public Bird {              // Parrot is a bird
  public: virtual void mimic();           // Can Repeat words...
};

// ...
Parrot mypet;
mypet.mimic();    // my pet being a parrot can Mimic()
mypet.fly();      // my pet "is-a" bird, can fly
```

Penguins Fail to Fly!



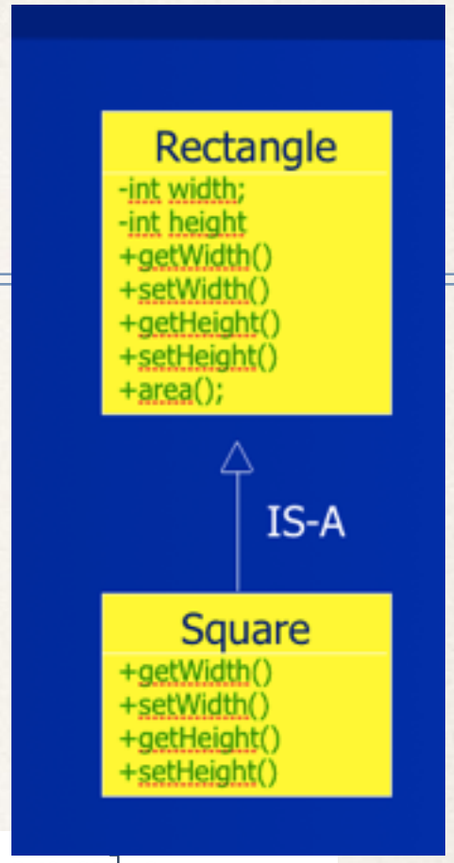
```
class Penguin : public Bird {
   public: void fly() {
      error ("Penguins don't fly!"); }
};
```

- Does not model: "*Penguins can't fly*"
- It models "*Penguins may fly, but if they try it is error*"
- Run-time error if attempt to fly → not desirable
- ***Think about Substitutability - Fails LSP***

```
void PlayWithBird (Bird& abird) {
    abird.fly();     // OK if Parrot.
    // if bird happens to be Penguin...OOOPS!!
}
```

Liskov Substitution Principe : contre-exemple

```
class Rectangle
  {
     int m_width;
     int m_height;
     public void setWidth(int width)
  {
         m_width = width;
     }
   public void setHeight(int h){
       m_height = ht;
   }
   public int getWidth(){
       return m_width;
   }
   public int getHeight(){
       return m_height;
   }
   public int getArea(){
   return m_width * m_height;
   }
}
```




```
class Square extends Rectangle
  {
public void setWidth(int width){
        super.setWidth(width);
        super.setHeight(width);
    }
public void setHeight(int height){
        super.setWidth(height);
        super.setHeight(height);
    }
  }
```

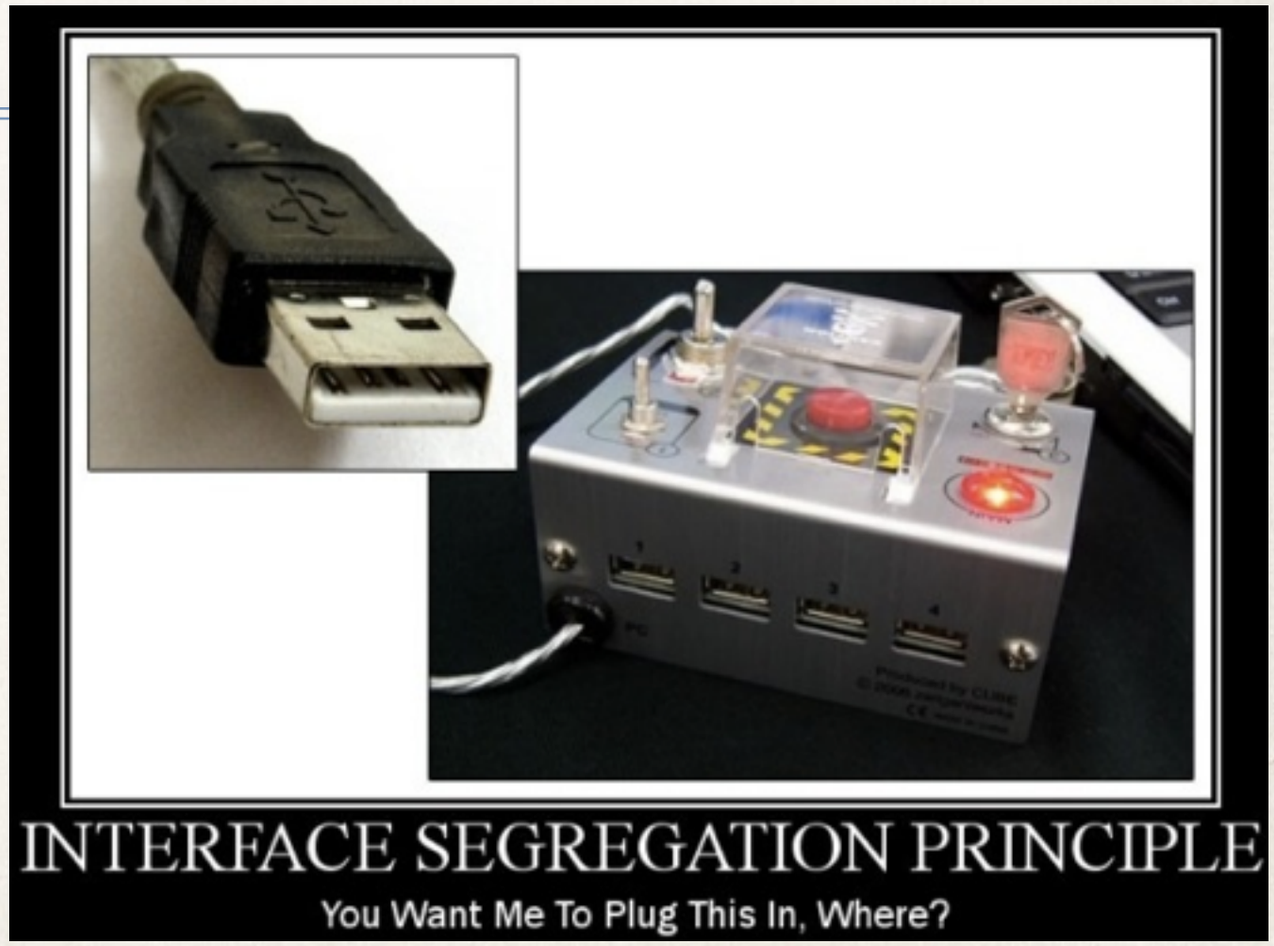
Liskov Substitution Principle

```
class LspTest
{
 private static Rectangle getNewRectangle(){
       // it can be an object returned by some factory ...
       return new Square();
 }

 public static void main (String args[]){
    Rectangle r = LspTest.getNewRectangle();
    r.setWidth(5);
    r.setHeight(10);

   // user knows that r it's a rectangle. It assumes that he's
 able to set the width and height as for the base class
    System.out.println(r.getArea());


// now he's surprised to see that the area is 100 instead of 50.
 }


}
```

"Clients should not be forced to depend upon interfaces that they do not use."
— Robert Martin, ISP paper linked from The Principles of OOD

SOLID: Interface Segregation Principle (ISP)

Make fine grained interfaces that are client specific.
Robert C. Martin.



http://williamdurand.fr/from-stupid-to-solid-code-slides/#/135

Program To An Interface, Not An Implementation

- An *interface* is the set of methods one object knows it can invoke on another object
- A class can implement many interfaces. (Essentially, an interface is a subset of all the methods that a class implements)
- A *type* is a specific interface of an object
- Different objects can have the same type and the same object can have many different types.
- An object is known by other objects only through its interface.
- Interfaces are the key to pluggability

Interface Example

```
/**
* Interface IManeuverable provides the specification
* for a maneuverable vehicle.
*/

public interface IManeuverable {
    public void left();
    public void right();
    public void forward();
    public void reverse();
    public void climb();
    public void dive();
    public void setSpeed(double speed);
    public double getSpeed();
}
```

Interface Example (Continued)

```
public class Car implements IManeuverable {
    // Code here.
}
```

```
public class Boat implements IManeuverable {
    // Code here.
}
```

```
public class Submarine implements IManeuverable {
    // Code here.
}
```

Interface Example (Continued)

- This method in some other class can maneuver the vehicle without being concerned about what the actual class is (car, boat, submarine) or what inheritance hierarchy it is in

```
public void travel(IManeuverable vehicle) {
    vehicle.setSpeed(35.0);
    vehicle.forward();
    vehicle.left();
    vehicle.climb();
}
```

Interface segregation principle

- Plusieurs «client-specific interfaces» sont mieux qu'une interface générale.
- Un client doit avoir des interfaces avec uniquement ce dont il a besoin
  - Incite à ne pas faire "extract interface" sans réfléchir
  - Incite à avoir des interfaces petites pour ne pas forcer des classes à implémenter les méthodes qu'elles ne veulent pas.
  - Peut amener à une multiplication excessive du nombre d'interfaces
    - à l'extrême : une interface avec une méthode (Penser à la cohésion...)
  - Utiliser l'expérience, le pragmatisme et le bon sens !

http://igm.univ-mlv.fr/ens/Master/M1/2013-2014/POO-DP/cours/1c-POO-x4.pdf

ISP Example: Timed door

```
class Door
{
   public:
   virtual void Lock() = 0;
   virtual void Unlock() = 0;
   virtual bool IsDoorOpen() = 0;
};
```

TimedDoor needs to sound an alarm when the door has been left open for too long. To do this, the TimedDoor object communicates with another object called a Timer.

ISP Example: Timed door

```
class Timer
{
   public:
   void Register(int timeout, TimerClient* client);
};
```

time of timeout

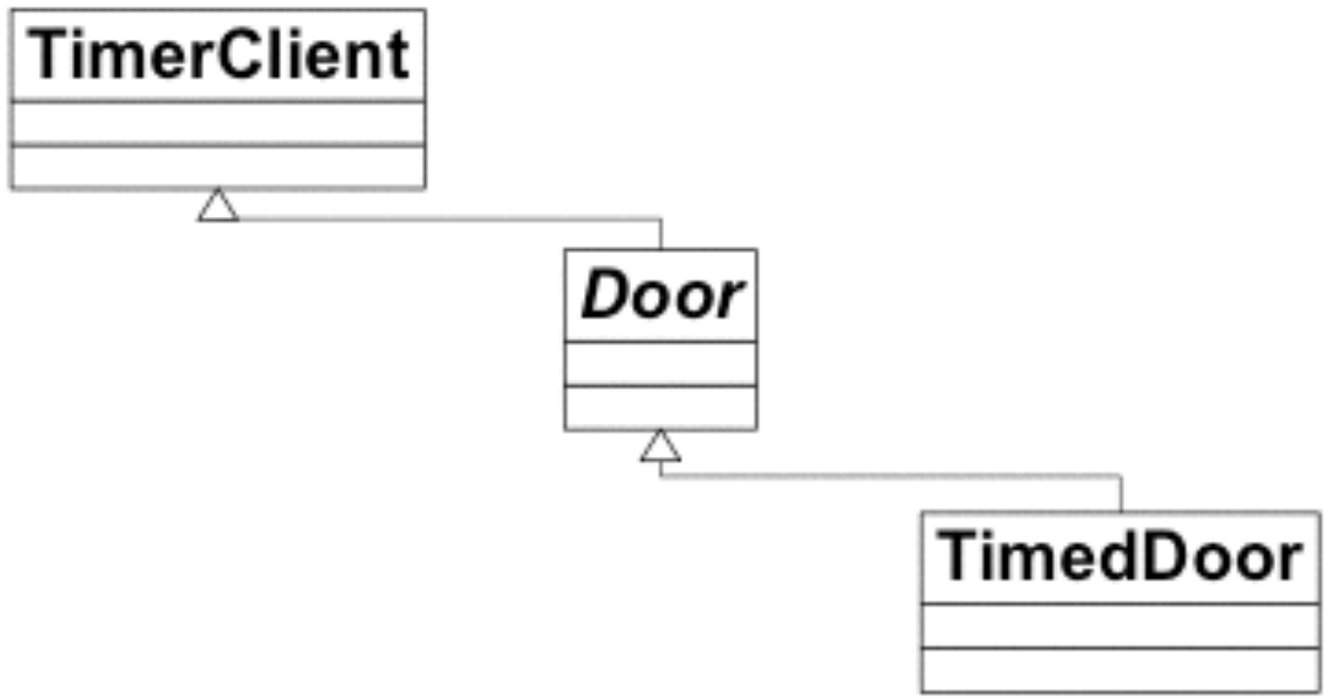
object to invoke TimeOut() on when timeout occurs

```
class TimerClient
{
   public:
   virtual void TimeOut() = 0;
};
```

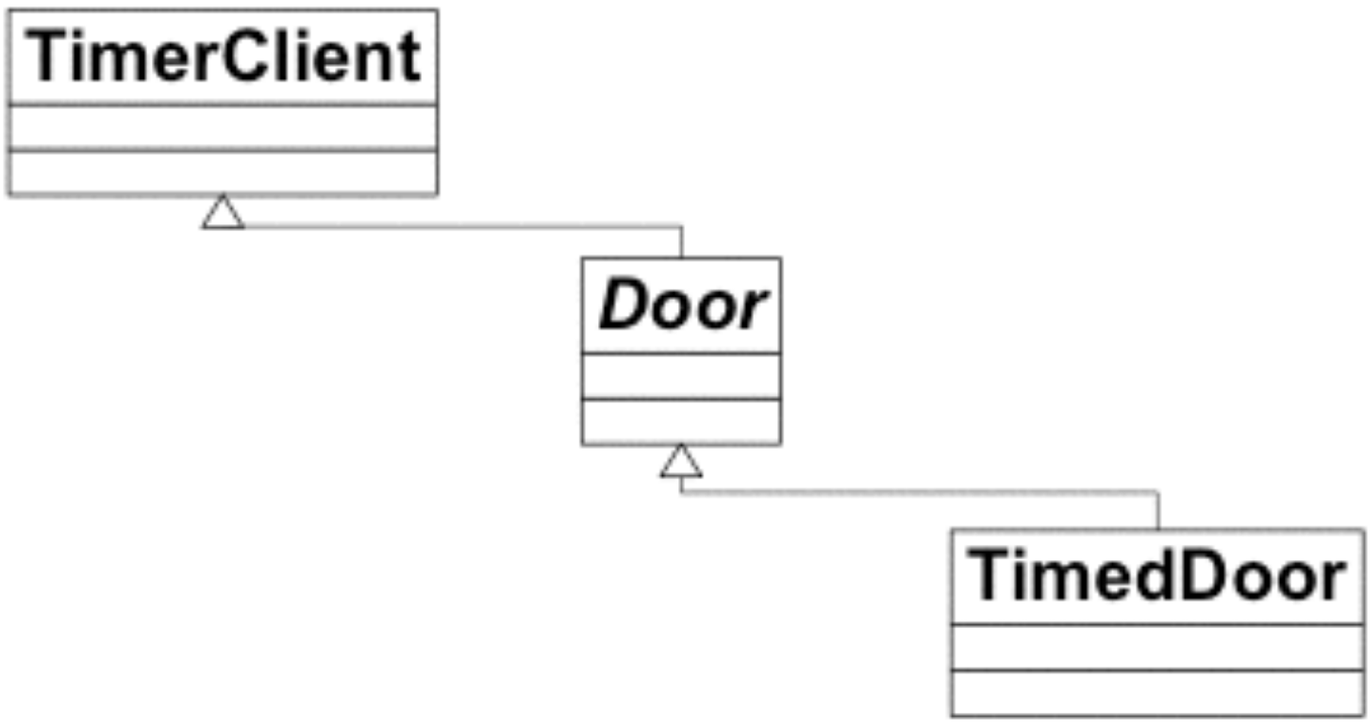
TimeOut method

How should we connect the TimerClient to a new TimedDoor class so it can be notified on a timeout?

InterfaceSegregationPrinciple Solution: yes or no?

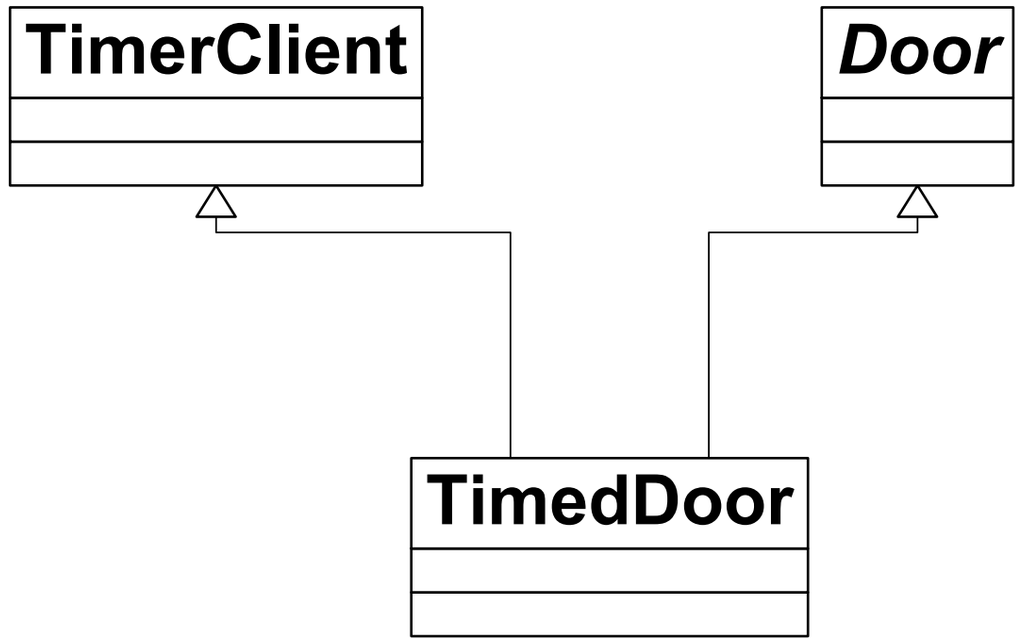


InterfaceSegregationPrinciple Solution: yes or no?

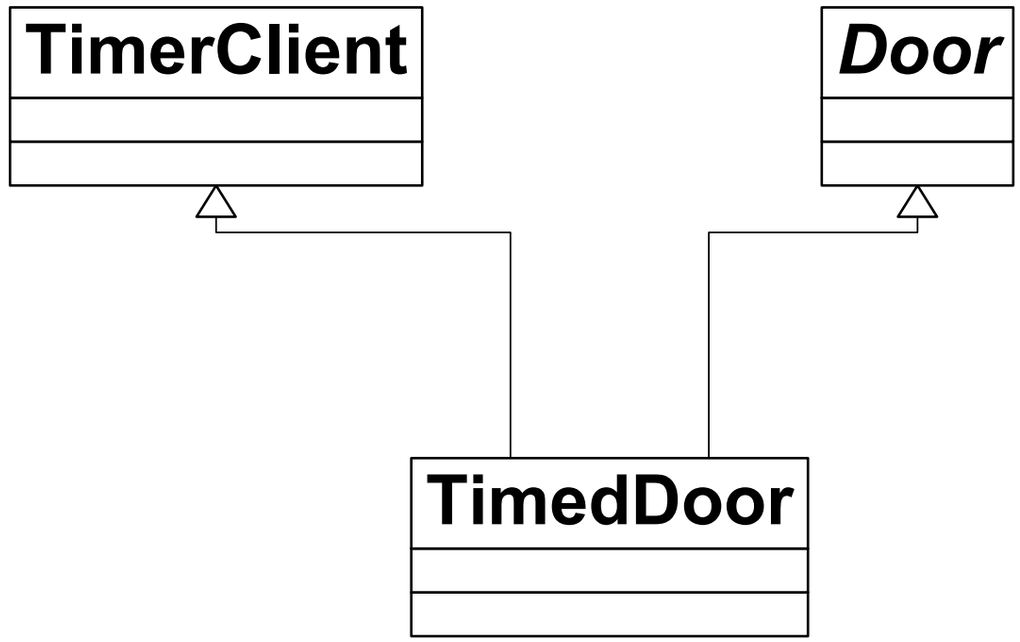


No, as it's polluting the Door interface by requiring all doors to have a TimeOut() method

ISP Solution: yes or no?



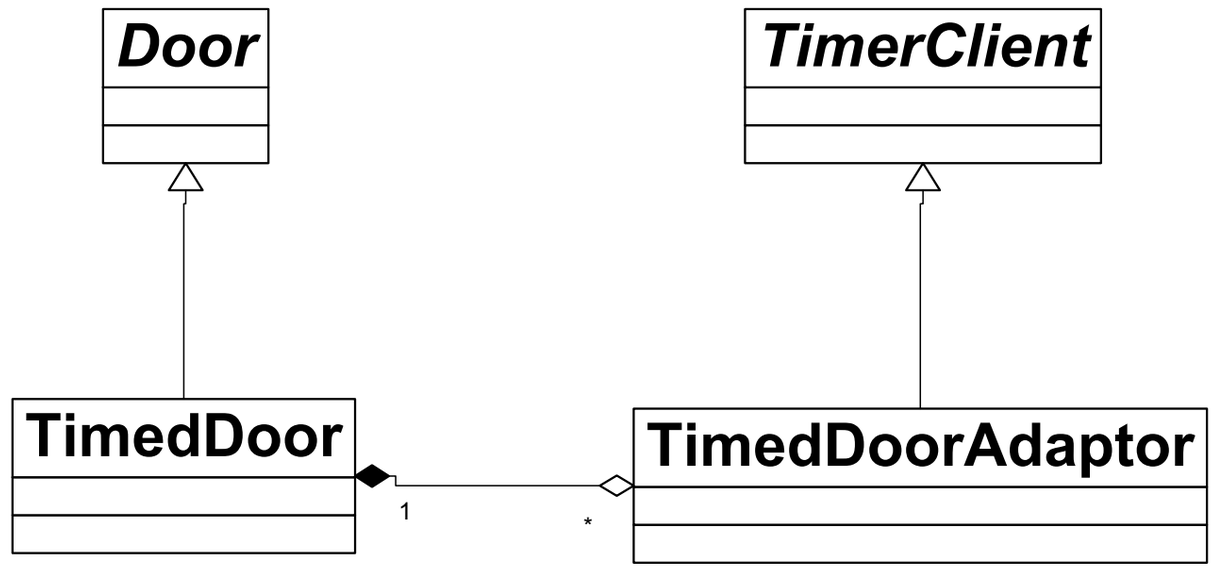
ISP Solution: yes or no?



Yes, separation through multiple inheritance

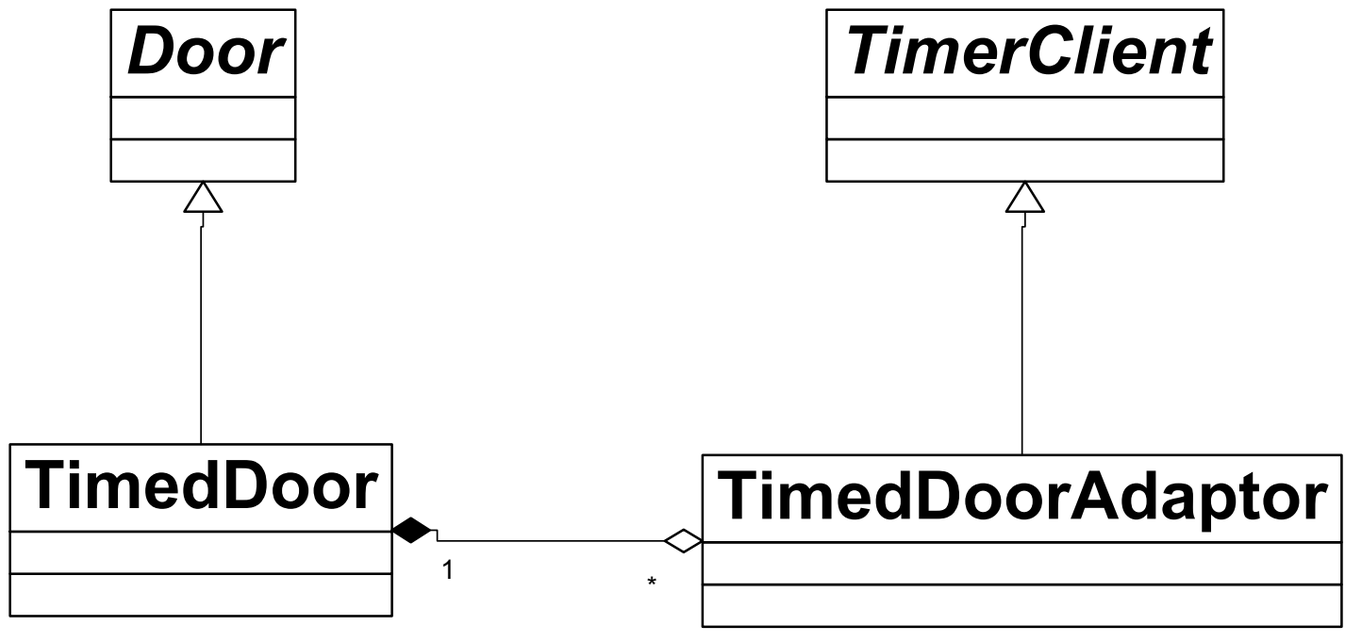
ISP solution: yes or no?

When the Timer sends the TimeOut message to the TimedDoorAdapter, the TimedDoorAdapter delegates the message back to the TimedDoor.



ISP solution: yes or no?

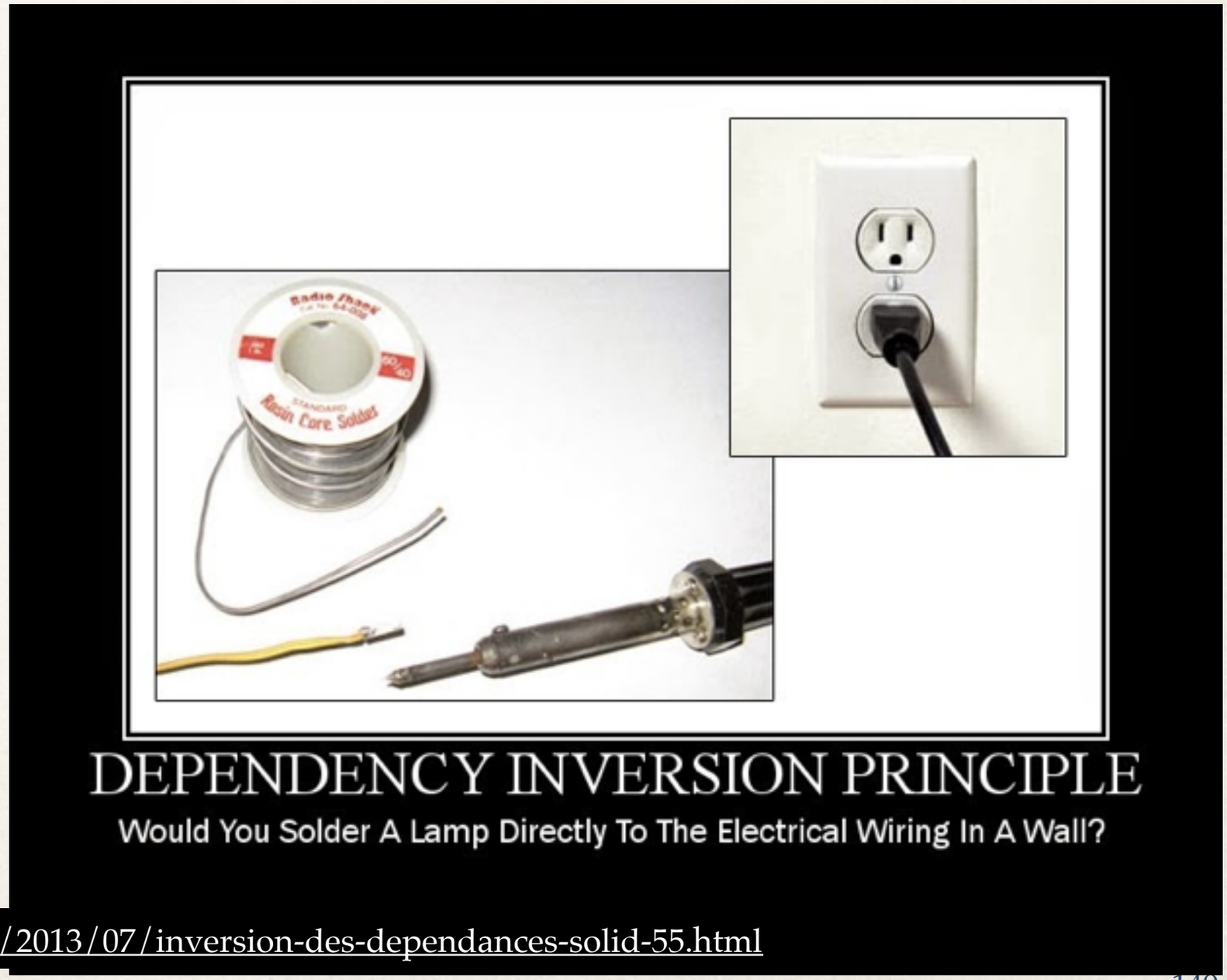
When the Timer sends the TimeOut message to the DoorTimerAdapter, the DoorTimerAdapter delegates the message back to the TimedDoor.



Yes, separation through delegation

Principe inversion de dépendance
Dependency Inversion Principle (DIP)

Depend on abstractions, not on concretions.
Robert C. Martin.



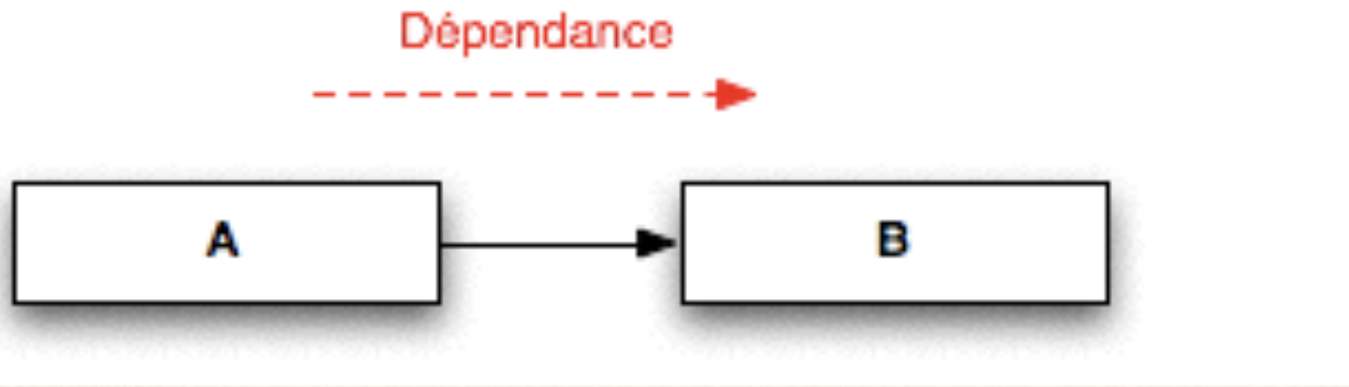
A LIRE !! http://cyrilgandon.blogspot.fr/2013/07/inversion-des-dependances-solid-55.html

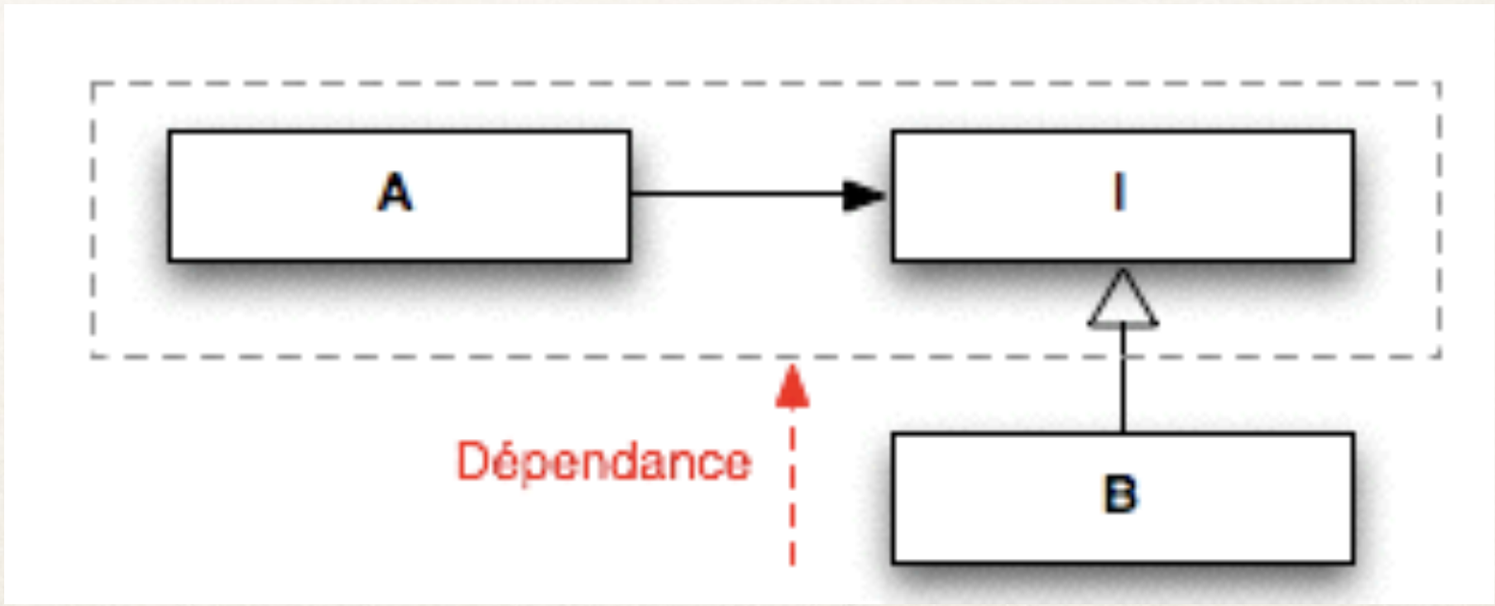
Inversion de dépendance

- Réduire les dépendances sur les classes concrètes
- «Program to interface, not implementation »
- Les abstractions ne doivent pas dépendre de détails.
  - → Les détails doivent dépendre d'abstractions.
- Ne dépendre QUE des abstractions, y compris pour les classes de bas niveau
- Permet OCP (principe) quand l'inversion de dépendance c'est la technique!

d'après http://igm.univ-mlv.fr/ens/Master/M1/2013-2014/POO-DP/cours/1c-POO-x4.pdf

Inversion de dépendance





Exemple de couplage fort

```java
package com.objis.spring.demoinjection;

public class Saxophone implements Instrument {
    public void jouer() {
        System.out.println("TOOT TOOT TOOT");
    }
}
```

```java
package com.objis.spring.demoinjection;

public class MusicienSansInjection {

        private String morceau;
        private Saxophone instrument ;

    public void joueInstrument() throws PerformanceException {
        System.out.println("Le Saxophone joue morceau " + morceau);
        instrument.jouer();
    }

    public MusicienSansInjection(String morceau) {
        this.morceau = morceau;
        instrument = new Saxophone();
    }
}
```

Problèmes couplage fort

- **Difficile de tester la Classe Musicien**
- **Difficile de réutiliser la Classe Musicien**

1) Masquer l'implémentation avec Interfaces !

Relâcher le couplage

- **Pour travailler avec un objet possédant un savoir-faire, nous déclarons une interface que l'objet doit implémenter.**

- **Cela créée un couplage faible entre l'objet demandeur et l'objet appelé. Ils n'ont pas besoin de se connaître mutuellement.**

Exemple de couplage faible

```java
package com.objis.spring.demoinjection;

public class Saxophone implements Instrument {
    public void jouer() {
        System.out.println("TOOT TOOT TOOT");
    }
}
```

```java
package com.objis.spring.demoinjection;

public class Piano implements Instrument {

    public void jouer() {
        System.out.println("PLINK PLINK PLINK");
    }
}
```

```java
package com.objis.spring.demoinjection;

public class Musicien implements Performeur {

    private String morceau;
    private Instrument instrument ;

    public void performe() throws PerformanceException {
        System.out.print("joue " + morceau + " : ");
        instrument.jouer();
    }

    public void setMorceau(String morceau) {
        this.morceau = morceau;
    }

    public void setInstrument(Instrument instrument) {
        this.instrument = instrument;
    }
}
```

Ici les classes sont indépendantes.

Couplage faible

GL VS UML ?

- Faire le lien entre UML et GL
- Les deux sont **complémentaires** et non pas contradictoires
- La conception et l'analyse s'intègrent dans les méthodes de développement agile