

# Partie I.4 : Conception abstraite avec UML

*Objectifs et Activités de la conception*

*Conception abstraite avec UML*

*Classes actives*

*Réalisation des cas d'utilisation*

*Diagrammes de séquence pour la conception*

*Compléments sur les machines à états*

*Documents à l'issue de la conception*



# Objectifs de la conception

---

- ❑ Passer d'un modèle d'analyse à une suite de modèles de plus en plus détaillés, exécutables, explicites
  
- ❑ Implémenter les diagrammes de classe et les modèles d'opérations
  - Introduire des aspects algorithmiques
  - Préciser le détail des interactions (interfaces)
  - Choisir les classes et méthodes qui implémenteront les interactions
  - Prendre en compte les limitations des langages
    - ☞ Héritage disponible ? Simple ou multiple ?
    - ☞ Quelles règles de visibilité ? Quels paquetages ?

# Objectifs de la conception (2)

---

- Prendre en compte des exigences de réalisation (performances, robustesse, sécurisation, synchronisation...)
  - ☞ Classes « techniques » et méthodes supplémentaires à prévoir
- Mettre en œuvre des schémas d'architectures logicielles (patterns de conception, architectures N-tiers)
- Utiliser des bibliothèques ou des « composants sur étagères » (« Components Off The Shelf » - COTS) pour certaines classes
  - ☞ Méthodes éventuelles à développer pour interfacer le composant
    - ☞ Tests du composant à prévoir !

# Objectifs de la conception (3)

---

- ❑ Quand le modèle obtenu est suffisamment détaillé
  - Passage au langage de programmation cible
  
- ❑ Systématiquement:
  - Documenter les choix réalisés
  - Tracer les choix par rapport aux exigences
  - Vérifier la cohérence des choix, veiller à une certaine simplicité
  - Écrire les documents de conception, faire le lien avec les documents d'analyse:
    - Classes du D.C.S. -> Classes de la conception
    - Associations du D.C.S. -> Attributs, méthodes, tables ?
    - Opérations de l'analyse -> méthodes des classes

# La conception en détail

---

- ❑ Choisir la représentation des associations (et vérifier leur navigabilité)
  - Attribut de la classe origine ?
  - Structure de données indépendante (table de hachage, base de données, ...)
  - Méthode statique de recherche ?
- ❑ Choisir entre attribut stocké ou calculé (dérivé)
- ❑ Représenter une fonction par un attribut (mémoisation) ?
- ❑ Identifier les attributs supplémentaires nécessaires (optimisation, décomposition d'un attribut complexe)
- ❑ Choisir les méthodes d'accès nécessaires, leur visibilité

# La conception en détail (2)

---

- ❑ Quels types prédéfinis pour certaines classes ou énumérations ?
  - Real (OCL) → en Java: `float` ? `double` ?
  - Date : on utilise la version Java ? On en fait une version plus simple en, encapsulant la date Java ?
- ❑ Quelle hiérarchie d'exceptions ?
- ❑ Quelles classes auxiliaires nécessaires
  - Classes actives (d'interface)
  - Classes techniques:
    - Analyseur syntaxique ? Parser XML ? Compérateurs Java ?
  - Introduction d'héritage supplémentaire ?
  - Classe pour représenter certains attributs (ex: classe `Adresse`)
  - Valeurs de retour multiples ?
    - Java : une seule valeur de retour → Classe pour encapsuler les résultats multiples du modèle d'opération...

# La conception en détail (3)

---

- ❑ Associer les opérations aux classes (voir ci-après)
- ❑ Identifier les méthodes complexes
  - les décrire par du pseudo-code ou des diagrammes de séquence
  - En déduire les méthodes auxiliaires nécessaires
  - Vérifier les exceptions nécessaires
  - Identifier le profil précis des méthodes
- ❑ Identifier les méthodes techniques nécessaires
  - Égalité ? (`String` en Java: 5 comparaisons d'égalités possibles !)
  - Relations d'ordre pour les tris ?
  - Opérateurs d'affectation ou de clonage
  - Quel jeu de constructeurs ?

# La conception en détail (4)

---

- ❑ Vérifier la cohérence globale de la conception
  - Par rapport à chaque cas d'utilisation
  - Par rapport aux diagrammes de séquence
  - Par rapport aux machines à états (cycle de vie des objets):  
retrouve-t-on les états indiqués ?
  - ☞ ***Avoir une vision d'ensemble avant de programmer***
  - ☞ ***Éviter d'avoir à retoucher la signature des méthodes***
  
- ❑ **En déduire l'architecture des classes, le squelette des classes et des méthodes.**

# Problèmes de visibilité

---

- ❑ Quelles relations de visibilité entre classes
  - Quels besoins en visibilité
  - En déduire les structures en paquetages nécessaires
  
- ❑ Comment établir la visibilité entre instances
  - Si  $x.op(...)$  contient un appel  $y.f(...)$  Comment  $x$  connaît-t-il  $y$  ?
    - $x$  stocke la référence de  $y$  dans un attribut
    - $y$  a été passée en paramètre lors de l'appel à  $op$
    - $y$  est une variable locale pour une instance créée par  $op$

Quelle rémanence pour les liens ? Quels attributs en plus ?

☞ à prévoir à la conception !

# La conception en UML

---

- ❑ Une conception « abstraite », **graphique** (diagrammes)
  - Diagrammes de séquence ou diagrammes d'interaction entre objets pour illustrer les enchaînements d'opérations
  - Machines à états élaborées
  
- ❑ *Avantage : on reste dans le modèle !*
  - Vérification de la cohérence avec le modèle d'analyse
  - Référentiel uniforme, partagé pour l'équipe
  - Plus rapide qu'un développement de code détaillé
    - ☞ Déterminer l'organisation générale du traitement
    - ☞ Échapper aux détails et contraintes d'un vrai langage
    - ☞ Éviter de compiler et mettre au point du code qui risque d'être jeté !
  - Génération de code assistée (« forward engineering »)
  - Production automatique d'une partie de la documentation

# La conception en UML (2)

---

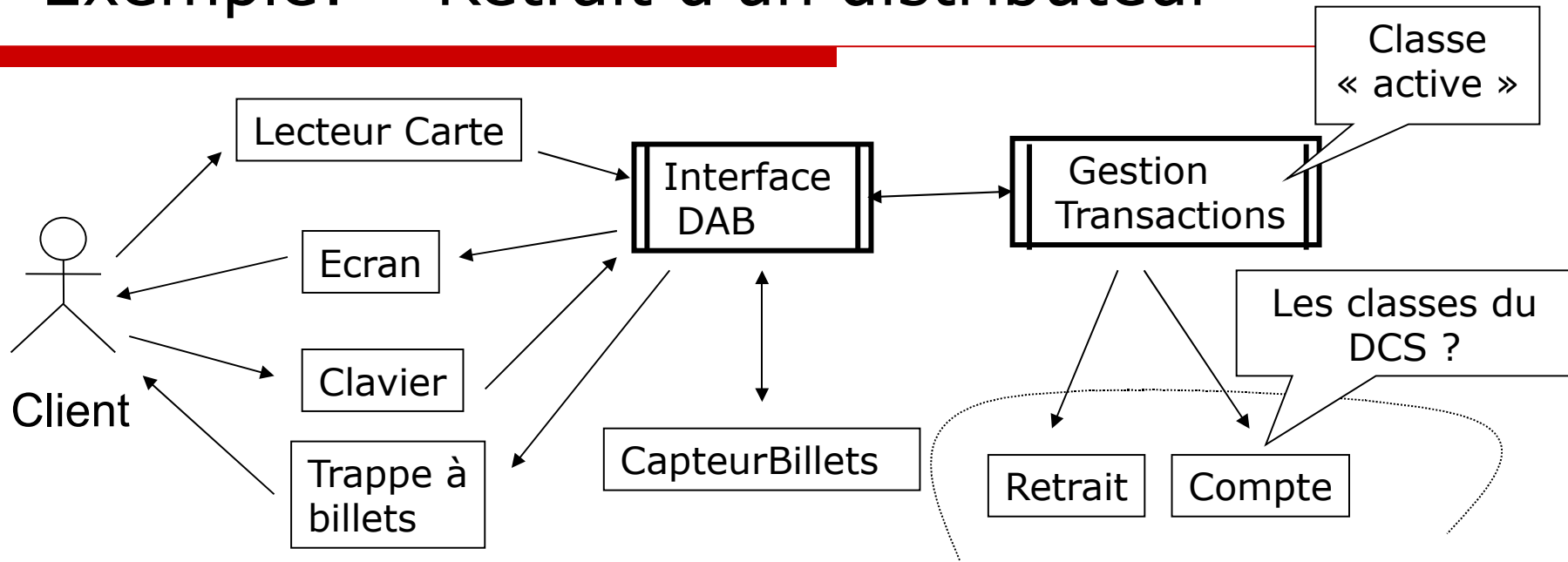
- ❑ Vue d'ensemble avant de passer aux détails:
  - donner une vue « en largeur » d'une opération** en décrivant schématiquement les interactions entre instances
  - Plus de chances d'identifier tous les aspects,
  - Plus d'opportunités de factoriser les méthodes
  - ☞ *Moins d'allers-retours conception ↔ codage*
  - ☞ *Moins de code*
  - ☞ *Code plus simple donc plus robuste*
  
- ☞ *« Esquisse » du code futur*
- ☞ *Difficulté: s'arrêter au bon niveau de détail !*

# Réalisation des cas d'utilisation

---

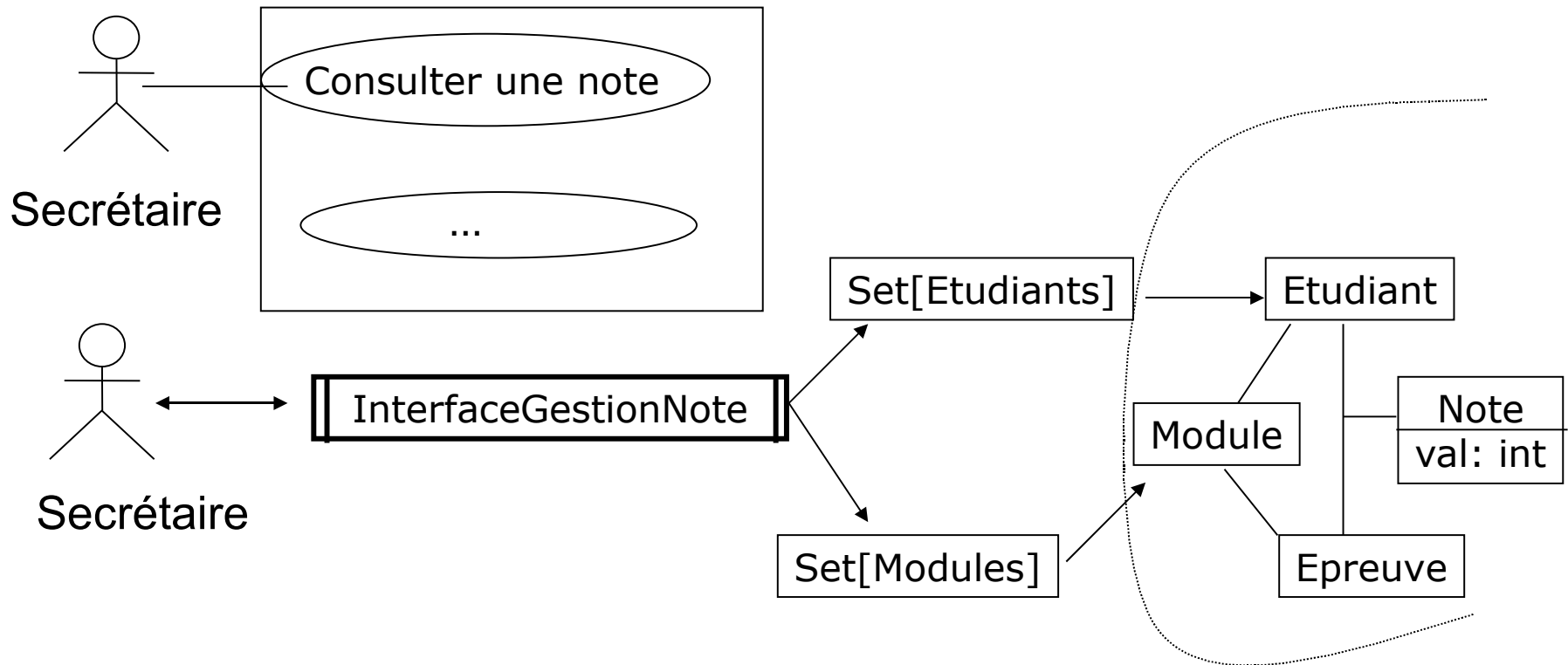
- ❑ Ils définissent les opérations qui doivent être réalisées
- ❑ Pour chaque cas d'utilisation, on établit :
  - *Une description textuelle* (plus détaillée qu'en analyse)
  - *Un diagramme de classe spécifique* :
    - ☞ Les classes concernées du diagramme de classes général
    - ☞ Les classes d'interface et classes auxiliaires
  - *Des diagrammes de séquence* pour l'opération à réaliser
    - ☞ Si besoin, des diagrammes pour les méthodes ajoutées
    - ☞ On utilise une version *plus algorithmique* des diagrammes
  - Si besoin, les exigences spécifiques (synchronisation)
- ❑ Après traitement de **tous** les cas d'utilisation, on déduit le squelette des classes : attributs, signature et visibilité des méthodes, exceptions signalées, paquetages induits

# Exemple: « Retrait d'un distributeur »



- ☞ « classe active » : ses instances ont leur propre flot de contrôle et peuvent donc initier des interactions.
- ☞ Les flèches symbolisent ici le sens des communications nécessaires entre classes
- ☞ Les autres classes sont des classes « techniques » pour l'interface

# Autre Exemple



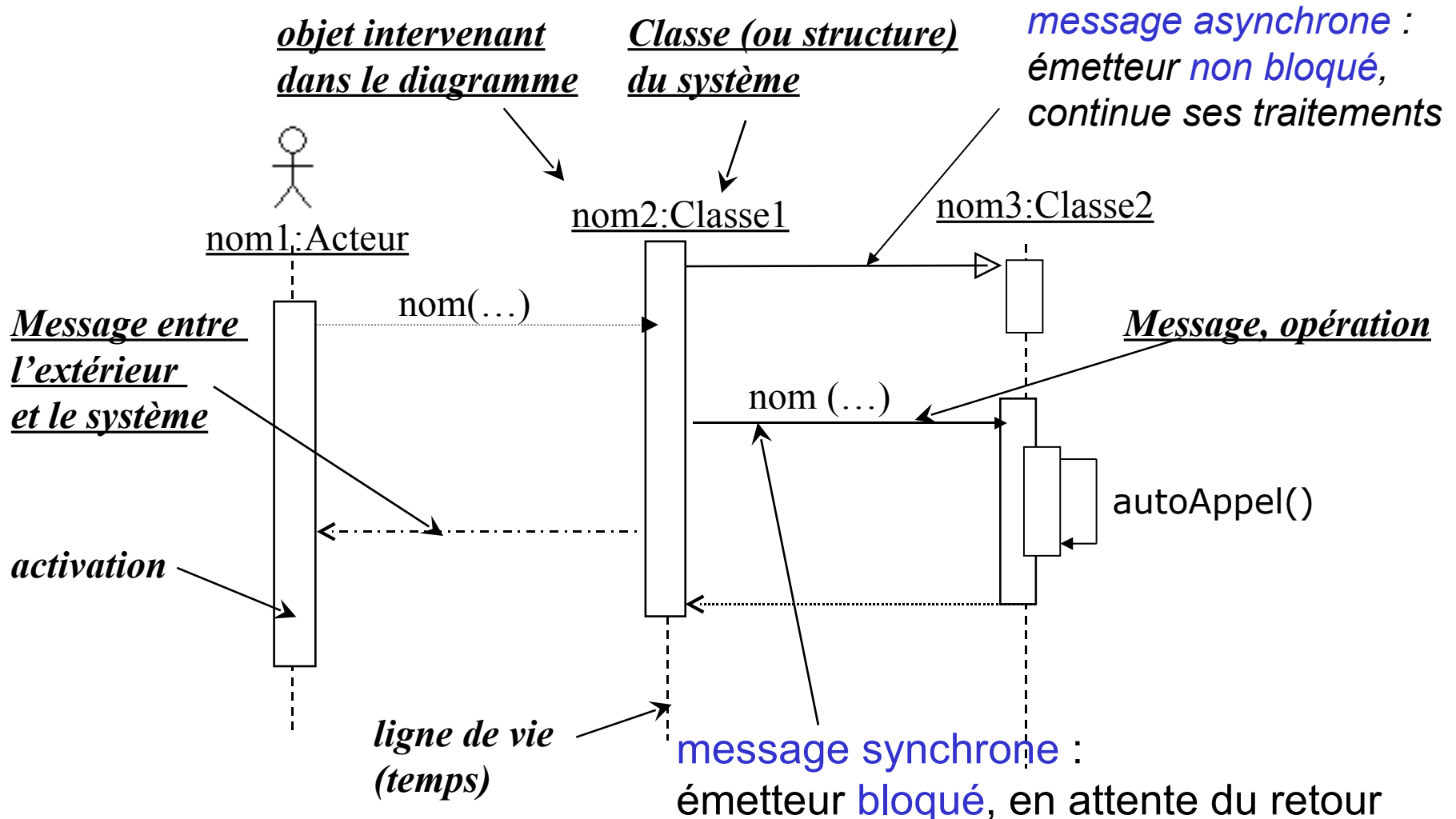
L'interface de gestion des notes n'a pas la visibilité directe d'une instance de `Etudiant` ou de `Epreuve` mais doit passer par un intermédiaire !

# Classes auxiliaires

---

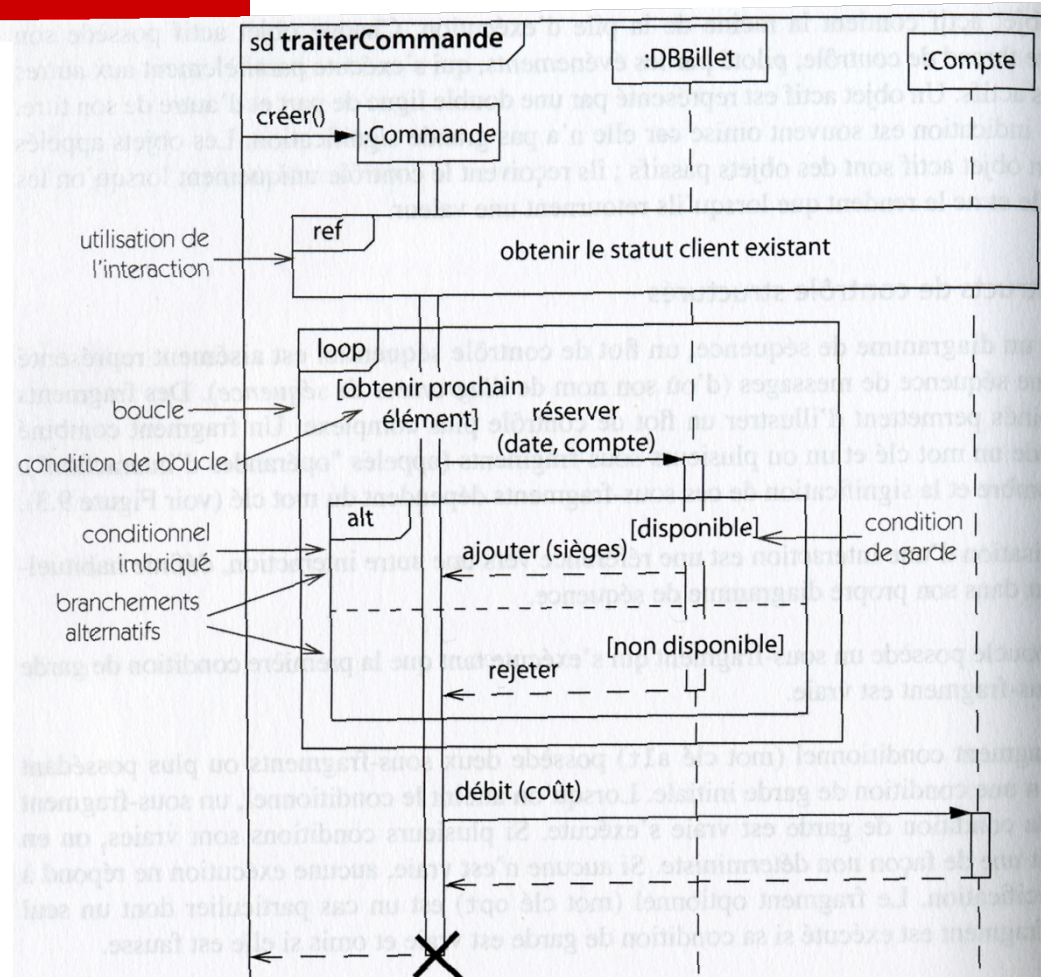
- ❑ Souvent on sera amené à introduire des classes qui n'existaient qu'implicitement dans le DCS
  - Détails de l'interface comme dans l'exemple précédent
  - Représentation des collections d'instances:  
pour communiquer avec une instance, il faut avoir une référence sur cette instance... Comment ? Souvent par recherche dans une collection d'instances !
  
- ❑ Suivant le type d'interface on introduira d'autres classes :
  - ☞ Une Interface Web ne demande pas les mêmes classes et méthodes qu'une interface « ligne » (comment s'enchaînent les interactions...)

# Diagrammes de séquence: syntaxe étendue



# Diagrammes de séquence « conception »

Diagramme de séquence  
« orienté conception » pour la  
réservation de spectacles



« UML 2.0, Guide de référence »

Rumbaugh & alli, CampusPress, 2005

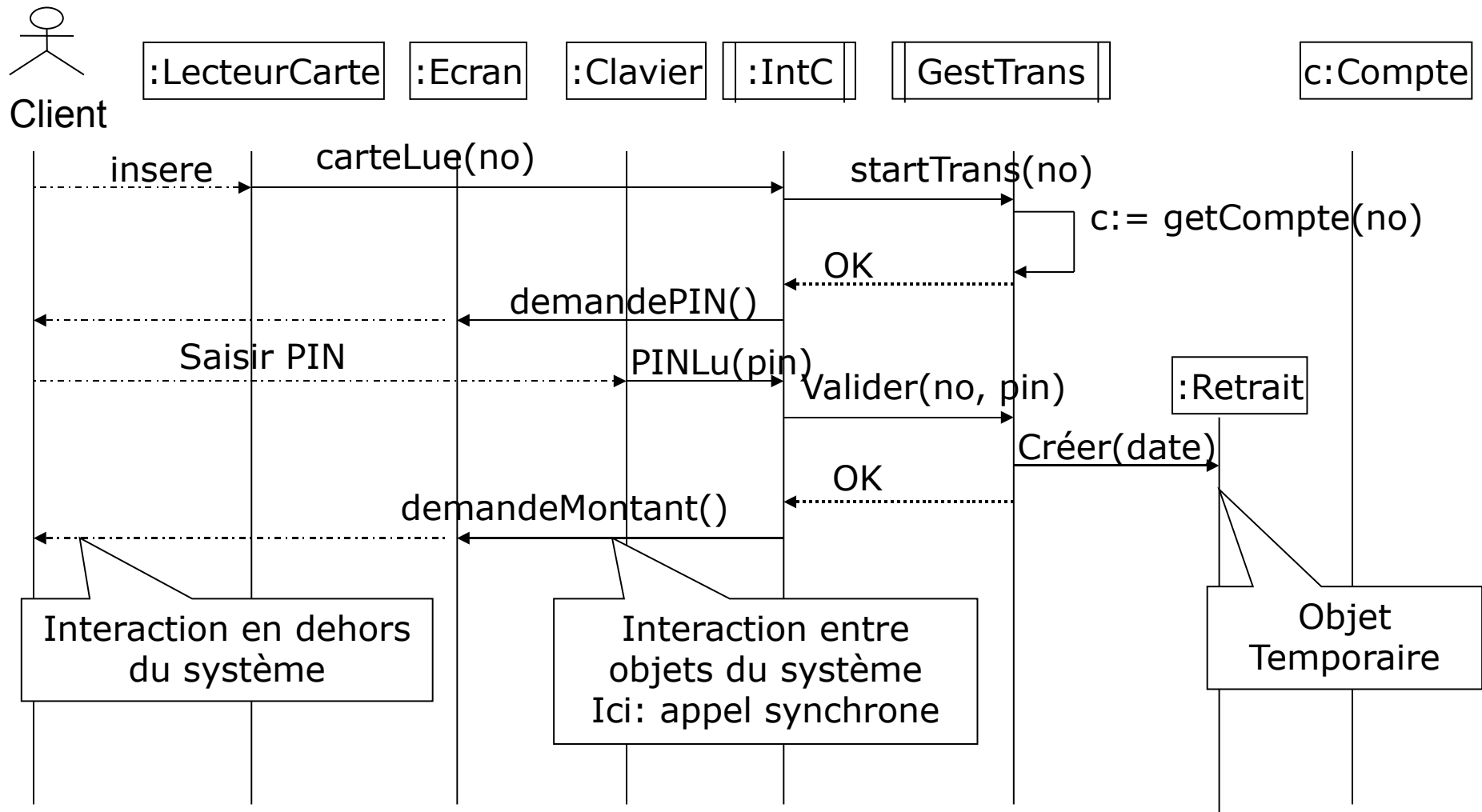
# Diagrammes de séquence « conception » (2)

---

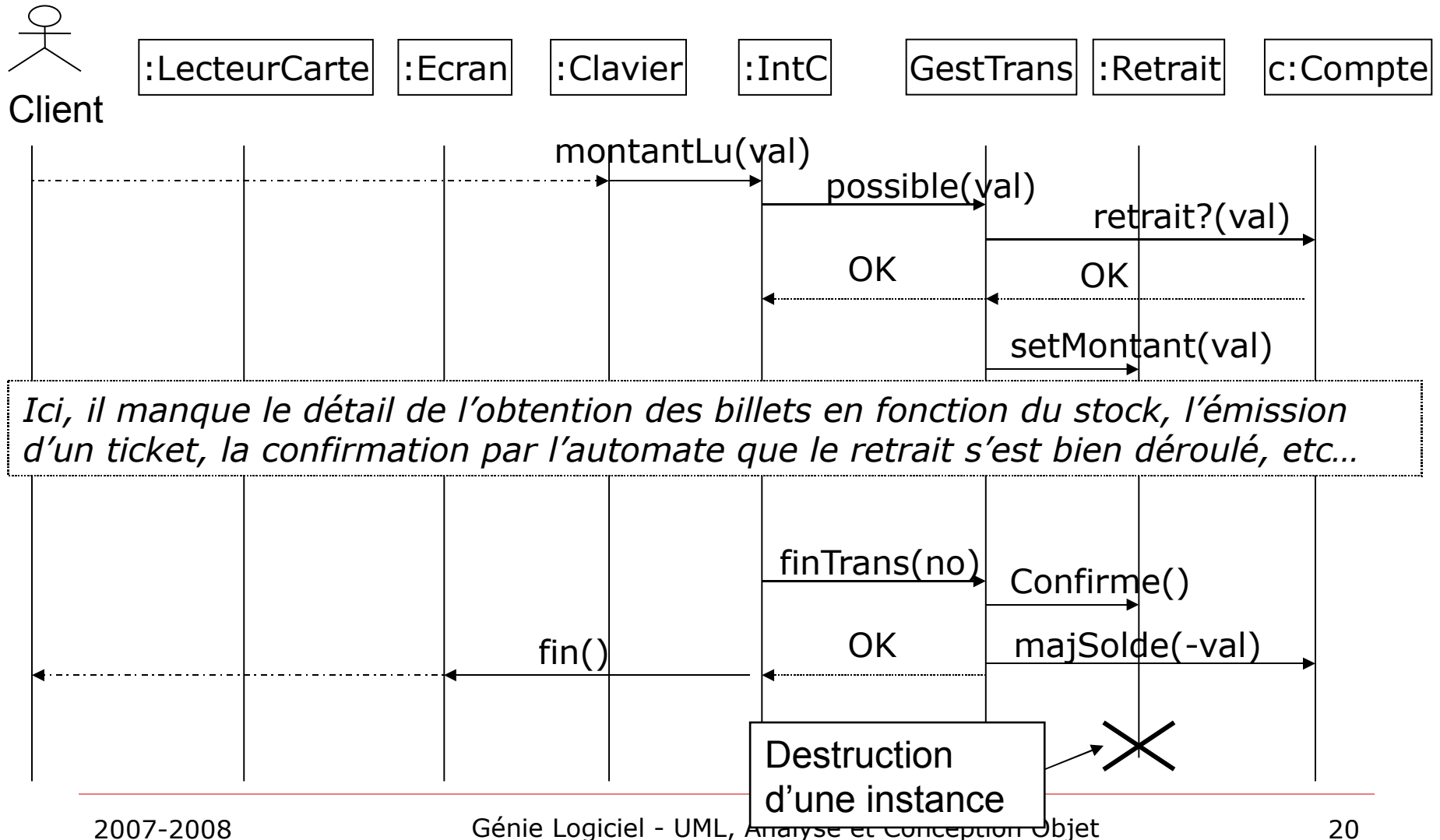
- ❑ Opérateurs de contrôle: `loop`, `par`, `alt`, `opt`, `ref`
- ❑ Ne pas abuser de ces notations, qui ne sont en général pas suffisamment précises pour remplacer du pseudo-code et qui compliquent le diagramme !
  - ☞ Utile si on veut engendrer du code automatiquement ?
  - ☞ Utile pour décrire à assez haut niveau des aspects algorithmiques
  - ☞ Rien n'empêche de recourir aux commentaires UML pour préciser certains aspects.

*Se demander si cela apporte plus d'information que de bruit*  
*Se demander si le diagramme reste suffisamment lisible*

# Diagramme de séquence pour « Retrait »



# Diagramme de séquence pour « Retrait »(2)



# Analyse du diagramme de séquence de « Retrait »

---

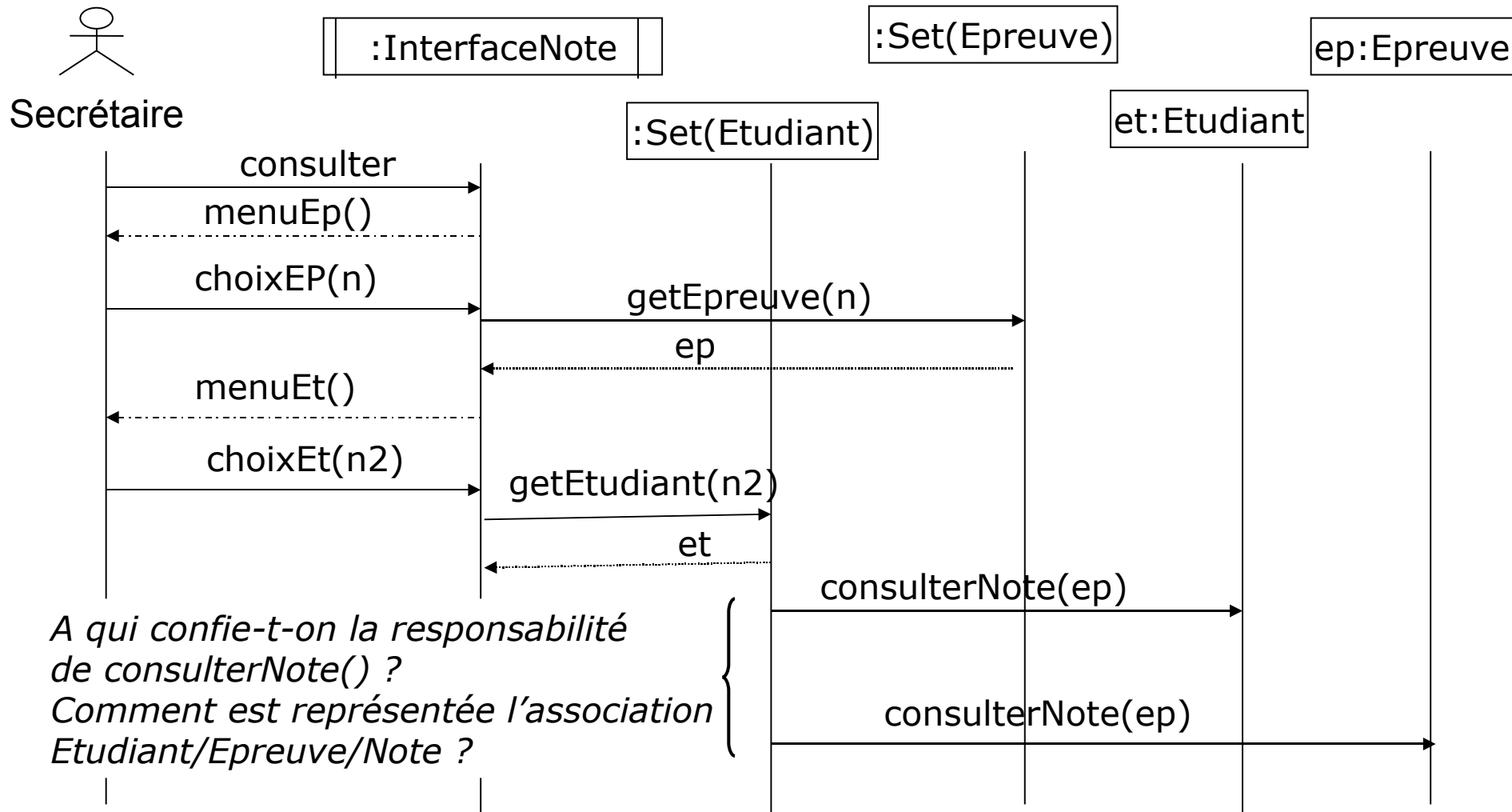
- ❑ Identification des classes nécessaires et de leur visibilité:
  - À partir de `InterfaceClient`, on a besoin de pouvoir retrouver l'instance de `GestionnaireTransaction`
  - `GestionnaireTransaction` doit pouvoir retrouver un compte à partir d'un numéro de CB. Donc besoin d'une méthode statique de recherche dans `Compte` ?
  - On a besoin de créer des instances temporaires de `Retrait`, mais pas d'un lien permanent
  - etc...
- ❑ Identification d'un certain nombre de méthodes avec leurs paramètres, valeur de retour
- ❑ Choix d'une manière d'enchaîner les opérations
- ❑ Choix d'une répartition des responsabilités entre classes

# Description diagrammatique des appels

---

- ❑ Permet de donner une vue « en largeur » du déroulement d'une opération:
  - Quels sont les objets concernés ?
  - Quelles méthodes de quelles classes sont concernées ?
  - ☞ *utile pour comprendre la mécanique d'ensemble et la répartition des responsabilités, avant de rentrer dans le détail*
  - ☞ *Choix à faire sur les appels qu'on fait apparaître (omettre certains détails pour améliorer la compréhension)*
  
- ❑ Par opposition à du pseudo-code ou du code qui donnera une vue « en profondeur » d'une opération:
  - Plus de détail sur la façon dont **une** méthode procède
  - Mais vision limitée à l'opération courante, sans compréhension globale des interactions entre objets

# Diagramme de séquence pour « Consulter une note »



# Exercices pour « Consulter une note »

---

## □ Hypothèses:

- Il n'y a pas de Set (Etudiant), ni de Set (Epreuve) mais une base de données qui stocke les résultats
- Les cas d'utilisation comportent une opération pour calculer la moyenne d'une épreuve
- Les cas d'utilisation comportent une opération pour calculer la moyenne d'un étudiant
- Les cas d'utilisation comportent les deux opérations précédentes

*Cela change-t-il quelque chose ?*

*Certains choix deviennent-ils meilleurs que d'autres ?*

## □ En annexe: la conception de « Utiliser un poste automatique »

# La conception en UML (fin)

---

## Limites de l'approche:

- Quand les diagrammes de séquence n'apportent plus rien par rapport au code ou pseudo-code
- Quand les diagrammes deviennent trop compliqués à lire, ou ne sont qu'une approximation
- Les diagrammes de séquence reflètent mal certains aspects de la mise en œuvre (gestion des exceptions ?)
- Écrire le code manuellement mais garder la possibilité de remonter les modifications dans le modèle ?
  - ☞ Rétro-Ingénierie !