

UML : Dossier E (début)

Les modèles de Conception

La phase d'analyse a permis de déterminer ce que doit faire le système informatique. Il convient ensuite, lors de la phase de conception, d'indiquer **comment** ce système peut être implémenté. Pour représenter cela il est possible d'utiliser les diagrammes UML. La modélisation UML à l'aide de diagrammes de conception permet de concevoir, comprendre, communiquer et même documenter du code (dans la mesure où l'on possède un AGL capable de faire de la rétro-ingénierie). Dans ce dossier nous ne ferons pas la différence entre la *conception préliminaire* et la *conception détaillée*.

Toutefois, en phase de conception, l'important n'est pas de bien connaître la notation UML mais de savoir penser et concevoir en terme d'objets : ce qui est très différents. Ainsi, dans le dossier présent, nous allons commencer par voir comment utiliser les différents diagrammes UML en phase de conception. Puis, dans le dossier F, nous aborderons les problèmes généraux relatifs à la conception objet. Nous aborderons alors les principes de l'affectation des responsabilités et nous ferons un tour d'horizon des principaux design patterns.

1 Le Diagramme des Classes de Conception

Le Diagramme de Classes de Conception (ou DCC) est généralement un sur-ensemble du diagramme des classes persistantes (ou modèle de domaine) auquel on a rajouté les classes graphiques et des classes de contrôle. Le diagramme des classes de conception doit permettre de décrire de façon statique les classes utilisées dans le code du programme. Il peut être utilisé soit en pro-ingénierie afin de déterminer l'architecture logicielle du programme soit en rétro-ingénierie pour documenter un code déjà écrit. Beaucoup de logiciels (dont Eclipse) gèrent automatiquement, et en temps réel, les interactions bidirectionnelles entre le code source et le diagramme de classes de conception.

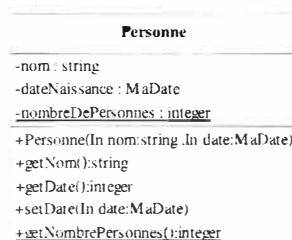
Tous les concepts vus sur le diagramme des classes persistantes dans le dossier B restent valables, toutefois il est possible d'utiliser d'autres concepts spécifiques à la conception que nous allons voir brièvement :

1.1 Les classes

Sur le plan de la notation la description des classes de conception est en tout point identique à la description des classes persistantes. Toutefois il est possible de rajouter la visibilité des attributs et des opérations (+ public, # protégé, - privé). On peut également signaler les attributs et opérations de classe (ou static) en les soulignant.

Dans un diagramme de classes il n'est pas obligatoire de détailler les paramètres des opérations. Néanmoins, si on souhaite le faire, il est possible d'indiquer la direction des arguments (IN, OUT, IN-OUT).

exemple :



Enfin les attributs dérivés sont assez peu utilisés dans un diagramme de classes de conception car ils peuvent être remplacés par des opérations.



1.2 Les associations

1.2.1 Navigabilité et nommage des associations

Lors de la phase d'analyse, la notion de **navigabilité** sert uniquement à exprimer le couplage et la dépendance entre les différentes classes (voir paragraphe sur le diagramme de paquetage dans le dossier B). Au niveau de la conception la navigabilité est un concept relativement plus important. Dans un diagramme de conception, elle permet de représenter le sens de navigation de l'association, c'est à dire la visibilité des classes par rapport aux autres classes de l'association. Le sens de navigation de l'association est indiqué grâce à une flèche. Si l'association ne possède pas de flèche, elle est par défaut bidirectionnelle (navigable dans les deux directions).

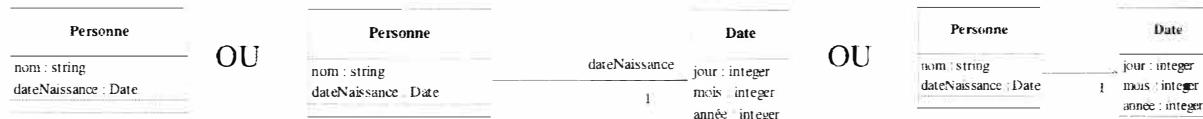
Dans un diagramme de classes de conception, du fait de la navigabilité, on nommera les associations par leur **rôle** plutôt que par un groupe verbal. Au niveau de l'implémentation, ce rôle deviendra, dans un langage de programmation, une variable d'instance. Bien entendu, on indiquera des rôles que si la navigabilité le permet.

exemple : Une personne possède une date de naissance. Cette propriété est ici représentée, sur l'association, à l'aide du rôle `dateNaissance`. Le sens de navigation de l'association fait que la Personne voit la date ; alors qu'à partir d'une `MaDate` on ne peut pas remonter vers la personne.



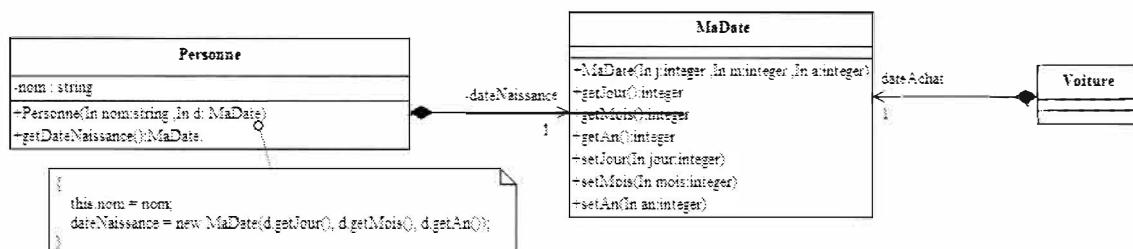
Dans la pratique, on trouve souvent des diagrammes de classes de conception où les propriétés sont exprimées par l'intermédiaire d'un attribut. Ce type de notation allège sensiblement le diagramme de classes car il permet au concepteur de ne pas représenter toutes les classes sur le diagramme ; *en particulier les classes de l'API de programmation*. Toutefois cette notation est moins précise et ne permet pas de représenter tous les cas de figure (notamment lorsque l'association comprend des multiplicités sur ses deux extrémités). Pour cette raison, certains font figurer tout de même l'association sans pour autant lui donner un rôle.

suite de l'exemple : la propriété est ici représentée grâce à l'attribut `dateNaissance`. Au niveau de l'implémentation ces notations sont identiques à la précédente.



1.2.2 Agrégation de composition

Si en conception l'agrégation partagée peut être vue comme unurre sémantique (Rumbaugh parle de placebo), l'agrégation de composition est quant à elle très utilisée. Elle permet de représenter le concept de "**non-partagabilité**" et de "**destruction automatique**". La règle générale est que, bien qu'une classe puisse être un composant de plusieurs autres classes, *chaque instance ne doit être un composant que de l'une d'entre elles*.



Il est à noter que lorsqu'on a une agrégation composition, il faut que les assesseurs de la classe agrégat renvoient des **recopies défensives** des objets de la classe agrégée.

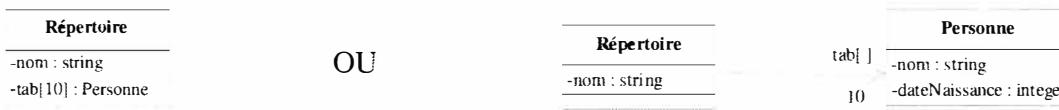
1.2.3 Représentation des attributs ‘collections’

Lorsqu’une classe comprend un ensemble d’objets d’une autre classe, il est possible (mais pas obligatoire) d’indiquer la façon dont cet ensemble est implémenté au niveau du code. Ainsi on va pouvoir indiquer si le concepteur utilise un tableau ou une collection.

exemple : Un répertoire est composé d’un ensemble de 10 Personnes. Si on ne veut pas indiquer la façon dont on code cet ensemble, on peut réaliser le diagramme suivant :



On peut également indiquer que le répertoire gère les personnes à l’aide d’un tableau.



On peut également indiquer que les personnes sont gérées à l’aide d’une collection ; par exemple un `ArrayList` en Java. Comme `ArrayList` est un type générique (ou template), on peut lui imposer de ne contenir que des Personnes.



Remarques : - dans la deuxième modélisation (à droite), non normalisée, l’utilisateur doit définir lui-même le type de la collection avec la contrainte `{ArrayList}`
 - comme nous le verrons plus tard, plutôt que d’utiliser la classe `ArrayList`, il peut être plus judicieux d’utiliser l’interface `List`

1.2.4 Les classes-associations

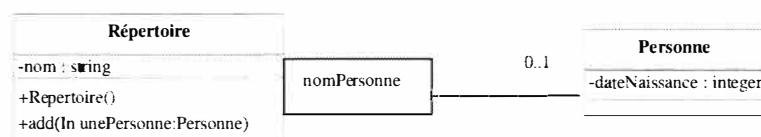
Lors de la phase d’analyse, une classe-association se différencie d’une classe à part entière (que l’on appelle communément classe coordinatrice) car elle est identifiée par les identifiants des classes qui sont reliées à l’association (contrairement à la classe coordinatrice qui possède son propre identifiant). Cependant, dans un diagramme des classes de conception, on ne représentera pas de classes-associations qui ne peuvent pas être traduites en programmation orientée objet. On les remplacera donc généralement par des classes coordinatrices (même si ce n’est pas la seule solution). Bien sûr, le développeur ne devra alors pas oublier de programmer à la main la contrainte sur l’identifiant qui a pu être décelée au cours de la phase d’analyse grâce à la classe-association.

Il faut ensuite déterminer le sens de navigation des différentes associations qui sont reliées à la classe coordinatrice (qui remplace la classe-association). Puis, en fonction du sens de navigation de ces associations, déterminer où placer les méthodes qui vont permettre de retrouver la liste des instances de cette classe coordinatrice – Fowler suggère même d’ajouter dans une des classes qui est relié à la classe coordinatrice, une méthode permettant de retrouver la liste des instances de l’autre classe qui est reliée à la classe coordinatrice.

1.2.5 Qualification des associations

L’association qualifiée permet de représenter en UML le concept de tableau associatif en programmation. Il se rapproche également du concept d’identification relative en Merise /2.

exemple : Dans un répertoire deux personnes ne peuvent pas avoir le même nom. Cela signifie que dans un répertoire pour un nom de personne donné, on a zéro ou une Personne.

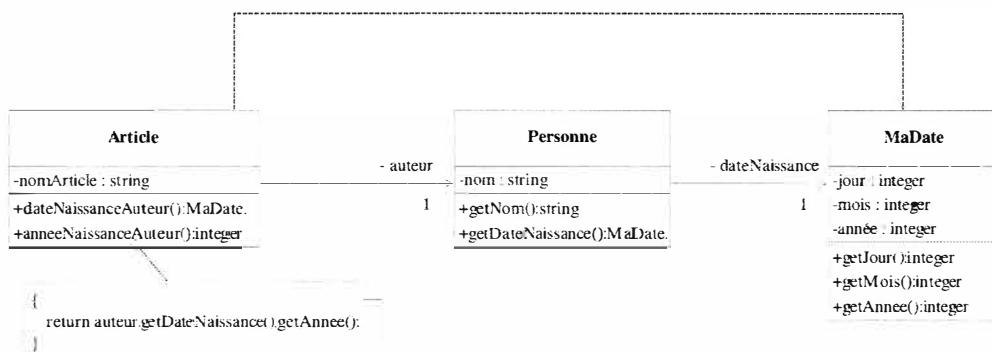


1.3 Dépendances

Une dépendance existe entre deux éléments si des changements apportés à la définition de l'un d'entre eux **peuvent entraîner des modifications** dans l'autre. Pour cette raison, il faut essayer de limiter le nombre de dépendance vers les *classes instables* (concept de Faible Couplage ou encore de Protection des Variations). Une dépendance est indiquée à l'aide d'une flèche en pointillés. Dans un diagramme des classes de conception les dépendances entre les différentes classes peuvent être très nombreuses. Ainsi pour ne pas surcharger le diagramme on ne représente que celles sur lesquelles le concepteur veut mettre l'accent.

exemple: la classe Personne dépend de la classe MaDate puisqu'un de ses attributs (`dateNaissance`) est de type MaDate. Toutefois, il ne paraît pas utile de représenter cette dépendance car elle est triviale (il existe déjà une association navigable entre ces deux classes).

Par contre, la classe Article dépend de la classe MaDate puisqu'une de ses opérations `-dateNaissanceAuteur` - retourne un MaDate (et aussi puisqu'une de ses opérations `-anneeNaissanceAuteur` – utilise une méthode de la classe MaDate). La représentation de cette dépendance est ici beaucoup plus pertinente.



Remarques :

- les relations d'héritage et les associations entraînent des dépendances implicites et triviales qu'il n'est pas utile de représenter
- il faut éviter de créer des dépendances dont on peut se passer. Surtout si elles portent sur des classes susceptibles d'évoluer (attention tout de même à ne pas perdre du temps à essayer de masquer des classes qui ont peu de risque de changer ; par exemple les bibliothèques Java)
- il faut éviter le code "catastrophe ferroviaire" qui ressemble à un ensemble de wagons accrochés les uns aux autres ; ce code peu soigné entraîne généralement des dépendances inutiles. Suivre la loi Déméter qui préconise que les assesseurs doivent cacher la structure interne de la classe
- il faut être attentif aux dépendances cycliques et il faut essayer d'éviter les dépendances bidirectionnelles (elles peuvent toujours être rompues à l'aide d'une interface)
- les dépendances ne sont pas obligatoirement transitives
- la représentation des dépendances n'est pas spécifique au diagramme de classes. Comme nous l'avons vu précédemment, il est aussi possible de les indiquer sur un diagramme de paquetages
- pour montrer le type de dépendance ou pour faciliter la génération de code à partir d'un AGL, on peut ajouter un stéréotype sur la relation de dépendance (par exemple « `call` » signifie que la dépendance est issue d'un appel d'opération, ou « `create` » d'un appel à un constructeur).

1.4 L'héritage et les classes abstraites

Dans un diagramme de classes, la classification multiple et l'héritage multiple sont en théorie autorisés. Toutefois, en conception détaillée, que l'on soit en rétro-ingénierie ou en prototypage, il faut que le diagramme de classes, pour qu'il soit exploitable, soit compatible avec le langage de programmation que l'on utilise.

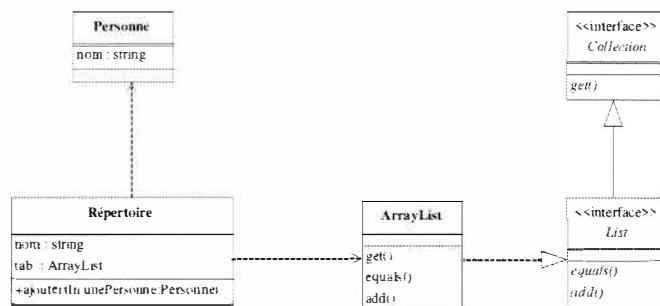
Or, en pratique quasiment aucun langage de programmation ne permet la classification multiple et très peu permettent l'héritage multiple. Par exemple, en Java les deux sont interdits. Ainsi, lorsqu'on travaillera en Java, **on n'indiquera pas de contraintes d'héritage dans le diagramme de classes de conception**. Par contre, il sera quand même possible d'exprimer la contrainte de couverture (ou Totalité) grâce aux classes abstraites. La contrainte de non-exclusion peut être modélisée, quant à elle, soit grâce à une spécialisation covariante (Pierre-Alain Muller), soit grâce à la délégation dont nous reparlerons plus tard.

1.5 Les interfaces

Une interface est une classe abstraite qui ne possède pas d'attribut d'instance et dont toutes les opérations sont abstraites. Naturellement, le nom d'une interface est comme pour toutes les classes abstraites écrit en italique. Dans un diagramme de classes de conception, pour mieux discerner les interfaces il est possible d'utiliser le stéréotype (ou mot-clé) « interface »

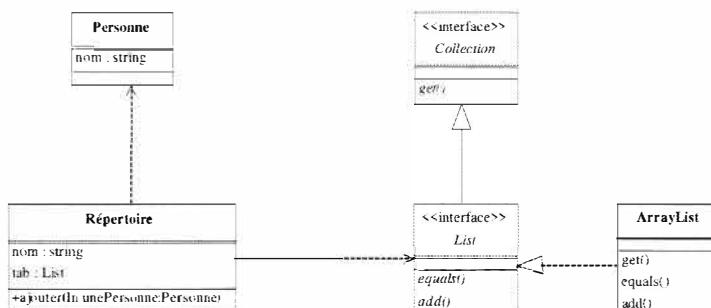
Bien entendu une interface peut hériter des opérations d'autres interfaces (dans ce cas, l'héritage peut même être multiple). L'implémentation d'une interface par une classe est notée à l'aide d'une flèche triangulaire en pointillés. On rappelle que l'implémentation multiple est autorisée dans tous les langages de programmation. Et on rappelle également qu'une classe qui implémente une interface doit redéfinir toutes les opérations de l'interface.

exemple : la classe Répertoire utilise (donc dépend) la classe ArrayList qui implémente l'interface List.



Dans l'exemple précédent, la classe Répertoire dépend de la classe ArrayList. En réalité elle a besoin d'une liste quelconque (pour stocker les personnes) et cette liste doit lui procurer l'opération ‘add’. Ce service qui est fourni par l'interface List est implémenté dans le cas présent par la classe ArrayList.

Pour cette raison, il est possible, si on le désire, d'indiquer que la classe Répertoire dépend directement de l'interface List. Cela permettra d'avoir un code plus **adaptable**.



Remarque : bien entendu, au niveau de l'implémentation, dans la classe Répertoire il faudra que la variable tab, qui est de type List, référence malgré tout une instance de ArrayList. En Java cela peut être fait de la façon suivante :

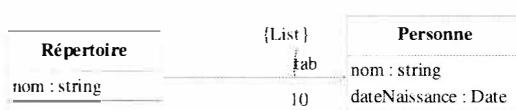
```
private List tab = new ArrayList();
```

Interfaces, collections et généricité ...

Depuis la version 1.5 de Java, grâce à la généricité, il est possible d'indiquer qu'une instance de l'interface List (ou qu'un objet de la classe ArrayList) ne contient qu'un seul type d'objets, par exemple des Personne. Ainsi, le code

```
private List<Personne> tab = new ArrayList<Personne>()
```

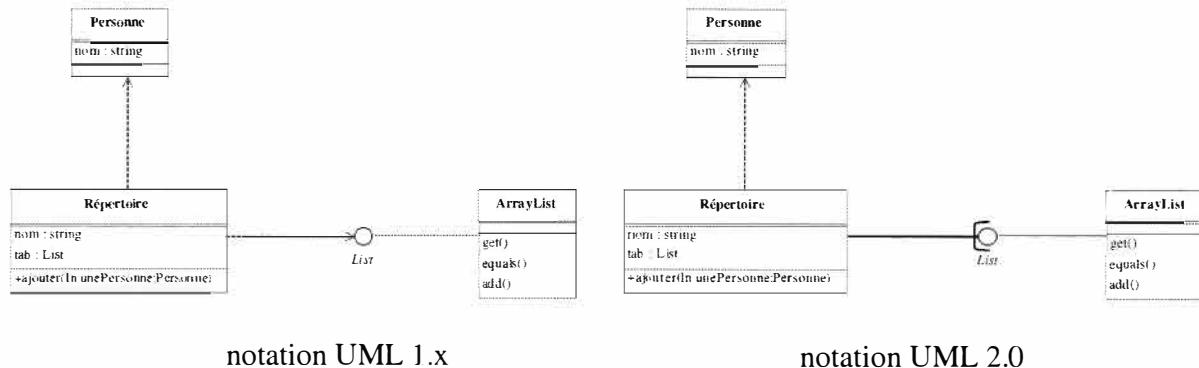
devrait pouvoir s'écrire en UML avec la notation suivante :



Autre notation : la lollipop

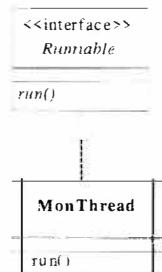
Les interfaces peuvent également être représentées par une icône en forme de petit cercle (lollipop). Ce genre de notation, normalisée par UML, est assez fréquemment utilisé dans les diagrammes des classes de conception (ainsi que dans les diagrammes de composants).

Dans l'exemple suivant, la sucette signifie qu'`ArrayList` implémente l'interface `List`. En UML 1.x, le fait que le Répertoire requiert l'interface `List` est indiqué grâce à une relation de dépendance classique mais dans UML2 cette notation a été remplacée par une prise de courant



1.6 Les classes actives

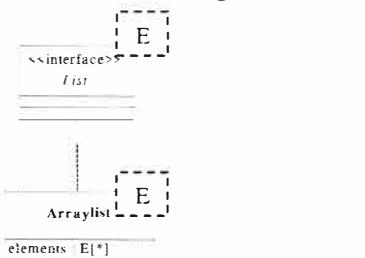
Un objet actif s'exécute dans son propre thread et le contrôle. La classe d'un objet actif est une classe active.



1.7 Les classes paramétrables (ou templates)

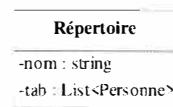
Une classe paramétrable, aussi appelée templates, sont des modèles de classes. Elles correspondent aux classes génériques d'Ada et aux templates de C++

exemple :



```

public class Repertoire
{
    private List<Personne> tab = new ArrayList<Personne>();
}
  
```



Exercices

(E01) Diagrammes de classes de conception - Rétro-ingénierie

```

public interface InterfaceDate {
    public int getJour();
    public int getMois();
    public int getAnnee();
    public int getAge();
}



---


public class MaDate implements InterfaceDate {
    private int T[] = new int[3];

    public MaDate() {
        Calendar now = Calendar.getInstance();
        T[0] = now.get(Calendar.DAY_OF_MONTH);
        T[1] = now.get(Calendar.MONTH)+1;
        T[2] = now.get(Calendar.YEAR);
    }

    public MaDate (int jour, int mois, int annee) {
        T[0]=jour;T[1]=mois; T[2]=annee;
    }

    public int getJour() {
        return T[0];
    }

    public int getMois() {
        return T[1];
    }

    public int getAnnee() {
        return T[2];
    }

    public String toString() {
        return (T[0] + "/" + T[1] + "/" + T[2]);
    }

    public int getAge(){
        InterfaceDate now = new MaDate();
        return differenceAge(now, this);
    }

    private static int compareTo(InterfaceDate date1, InterfaceDate date2) {
        return ((date1.getAnnee()*10000+date1.getMois()*100+date1.getJour())-
                (date2.getAnnee()*10000+date2.getMois()*100+date2.getJour()));
    }

    public static int differenceAge(InterfaceDate dateSup, InterfaceDate dateInf) {
        if (compareTo(dateSup,dateInf)<0)
            return -differenceAge(dateInf,dateSup);
        else { int difference = dateSup.getAnnee() - dateInf.getAnnee();
            if ((dateSup.getMois()<dateInf.getMois()) ||
                (dateSup.getMois()==dateInf.getMois() & dateSup.getJour()<dateInf.getJour()))
                difference = difference - 1;
            return difference;
        }
    }
}



---


public abstract class Habitation {
    protected int numero;
    private int superficie;
    private InterfaceDate dateDerniereRenovation;
    private Immeuble immeuble;
    private static int nbHabitations = 0;

    public Habitation(int superficie, InterfaceDate uneDate) {
        nbHabitations++;
        numero = nbHabitations;
        this.superficie = superficie;
        dateDerniereRenovation = new MaDate(uneDate.getJour(), uneDate.getMois(),
                                            uneDate.getAnnee());
    }

    public int getNumero() {
        return numero;
    }

    public int getSuperficie() {
        return superficie;
    }
}

```

```

public void setImmeuble(Immeuble immeuble) {
    this.immeuble = immeuble;
}

public Immeuble getImmeuble() {
    return immeuble;
}

public String getNomTaxation() {
    if (immeuble == null)
        return "pas de taxe";
    else    return immeuble.getTaxation().getNom();
}

public void setDateDerniereRenovation(MaDate nouvelleDate) {
    dateDerniereRenovation = new MaDate(nouvelleDate.getJour(), nouvelleDate.getMois(),
                                         nouvelleDate.getAnnee());
}

public MaDate getDateDerniereRenovation() {
    return new MaDate(dateDerniereRenovation.getJour(), dateDerniereRenovation.getMois(),
                      dateDerniereRenovation.getAnnee());
}

public abstract float loyerMensuel();

public float loyerAnnuel() {
    return loyerMensuel() * 12;
}

public String toString() {
    return "Habitation n°" + getNumero() + " qui a été rénovée le " + dateDerniereRenovation;
}

```

```

public class Appartement extends Habitation {
    private int nbPieces;

    public Appartement(int superficie, InterfaceDate uneDate, int nbPieces) {
        super(superficie, uneDate);
        this.nbPieces = nbPieces;
    }

    public int getNbPieces() {
        return nbPieces;
    }

    public float loyerMensuel() {
        return getSuperficie() * 5 * (1 + nbPieces / 10);
    }
}

```

```

public class AppartementDeLuxe extends Appartement {

    public AppartementDeLuxe(int superficie, InterfaceDate uneDate, int nbPieces) {
        super(superficie, uneDate, nbPieces);
    }

    public float loyerMensuel()
    {
        return super.loyerMensuel() + 500;
    }
}

```

```

public class Loft extends Habitation {
    private int hauteurPlafond;

    public Loft(int superficie, InterfaceDate uneDate, int hauteurPlafond) {
        super(superficie, uneDate);
        this.hauteurPlafond = hauteurPlafond;
    }

    public int getHauteurPlafond() {
        return hauteurPlafond;
    }

    public float loyerMensuel() {
        return getSuperficie() * 3 * hauteurPlafond;
    }
}

```

```

public class Immeuble {
    private List<Habitation> vect;
    private Taxation taxation;

    public Immeuble(Taxation taxation) {
        vect = new ArrayList<Habitation>();
        this.taxation = taxation;
    }

    public void vider() {
        vect.clear();
    }

    public boolean ajouter(Habitation hab) {
        if (place(hab.getNumero()) != -1)
            return false;
        else
            return vect.add(hab);
    }

    public void retirer(int num) {
        int i = place(num);
        if (i != -1)
            vect.remove(i);
    }

    public Habitation rechercher(int num) {
        int i = place(num);
        if (i != -1)
            return vect.get(i);
        else
            return null;
    }

    public Taxation getTaxation() {
        return taxation;
    }

    public float loyerAnnuelTotal() {
        float sum = 0;
        for (Habitation a : vect)
            sum += a.loyerAnnuel();
        return sum;
    }

    public float taxesAnnuelles() {
        if (nbHabitations() > 100)
            return taxation.getMontantTaxe();
        else if (nbHabitations() >= 10)
            return loyerAnnuelTotal() * taxation.getTauxTaxe();
        else
            return loyerAnnuelTotal() * 0.1f;
    }

    public String toString() {
        String tampon = new String();
        for (Habitation a : vect)
            tampon += a.toString();
        return tampon;
    }

    private int nbHabitations() {
        return vect.size();
    }

    private int place(int num) {
        int i = 0;
        while (i < vect.size() && (vect.get(i)).getNumero() != num)
            i++;
        if (i == vect.size())
            return -1;
        else
            return i;
    }
}

```

```
public class Taxation {  
  
    private String nomTaxe;  
    private float montantTaxe;  
    private float tauxTaxe;  
  
    public Taxation(String nomTaxe, float montantTaxe, float tauxTaxe) {  
        this.nomTaxe = nomTaxe;  
        this.montantTaxe = montantTaxe;  
        this.tauxTaxe = tauxTaxe;  
    }  
  
    public String getNom() {  
        return nomTaxe;  
    }  
  
    public float getMontantTaxe() {  
        return montantTaxe;  
    }  
  
    public float getTauxTaxe() {  
        return tauxTaxe;  
    }  
}
```

Questions :

- 1) Réaliser à la main un diagramme de classes de l'application en mode esquisse. On fera apparaître sur le diagramme uniquement les dépendances non triviales.
- 2) A l'aide du logiciel de votre choix, réaliser un diagramme de classes de l'application en mode plan.
- 3) Proposer un ou des moyens qui permettraient de réduire les dépendances inutiles.

UML : Dossier E (suite)

Les modèles de Conception

2 Le diagramme de séquence des interactions (DSI)

Le diagramme de séquence est sans aucun doute le diagramme UML le plus utilisé après le diagramme de classes. Il permet de décrire de façon dynamique un scénario ou une méthode en renseignant les classes de conception participantes et en décrivant le partage des responsabilités entre les objets de ces classes. On peut utiliser le diagramme de séquence des interactions soit en pro-ingénierie pour trouver les opérations qui doivent apparaître dans le diagramme de classes de conception, soit en rétro-ingénierie pour illustrer le fonctionnement interne de ces opérations ou pour documenter la façon dont ces opérations sont utilisées dans un scénario.

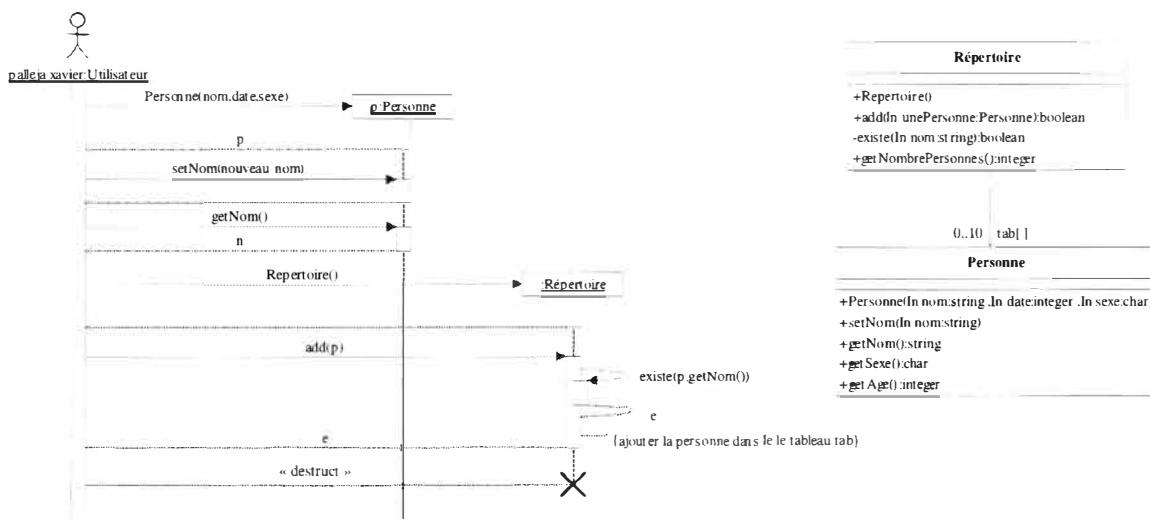
Pour être lisible par tous, y compris par les AGL, les diagrammes de séquence des interactions doivent respecter un certain nombre de **règles de notation** que nous allons commencer par présenter. Ensuite, nous verrons que dans un processus de développement tel que UP, les diagrammes de séquence des interactions permettent *de faire le lien entre l'analyse et la conception*. Grâce à eux il va être possible, d'une part de développer les scénarios d'analyse des cas d'utilisation, et d'autre part, de faire la liaison entre ces scénarios et les objets issus du diagramme des classes de conception. Enfin, pour représenter les interactions, plutôt que d'utiliser le diagramme de séquence, il est possible d'utiliser le *diagramme de communication* (anciennement diagramme de collaboration dans UML 1.x).

2.1 Règles de notation du diagramme de séquence des interactions

Le diagramme de séquence permet d'illustrer le fonctionnement interne d'une méthode ou d'un scénario. Grâce aux structures de contrôle (alternatives ou itératives) il est possible de détailler une méthode ou même de faire cohabiter plusieurs scénarios sur un même diagramme.

2.1.1 Principes de base

En conception, le diagramme de séquence indique quelles sont les opérations qui sont appelées sur les différents objets. Le diagramme de séquence doit permettre de voir à quel moment (ou dans quel ordre chronologique) les opérations sont utilisées. Il doit aussi permettre d'identifier l'objet qui a déclenché l'appel de l'opération.



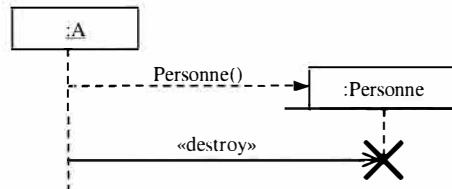
Règle générale : Dans un diagramme de séquence de conception, tout message donne lieu à une opération définie dans la classe de l'objet qui reçoit le message

Remarques :

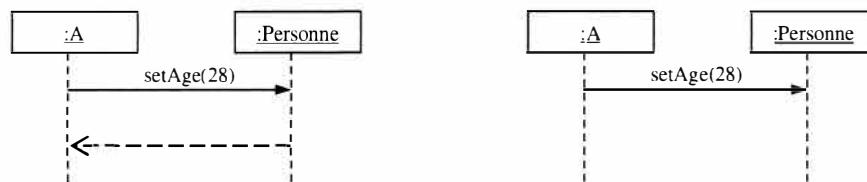
Dans UML 2.0 le diagramme de séquence a subi un certain nombre de modifications. Les objets sont remplacés par des *participants* (qui ne sont pas soulignés). Et, comme nous le verrons plus tard, les *appels à des constructeurs* sont indiqués en pointillés, on peut se passer des acteurs en utilisant les *messages trouvés*, et les *cadres d'interaction* permettent l'imbrication des structures de contrôle.

Remarques :

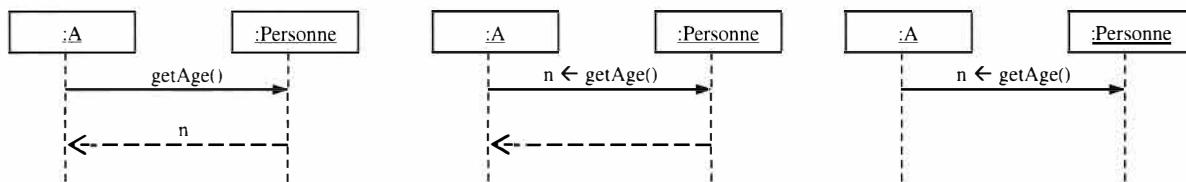
- Sur un diagramme de séquence, il est possible de faire figurer les opérations qui permettent de créer (constructeur) ou de détruire (destructeur en C++) un objet. Depuis UML 2.0, l'appel à un constructeur est illustré à l'aide d'une flèche en pointillés.



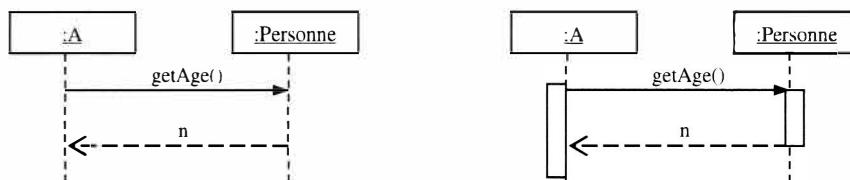
- Certaines personnes représentent des flèches de retour pour tous les appels. C'est d'ailleurs le cas sur le diagramme de la page précédente qui a été fait avec le logiciel Objecteering. Mais il n'est pas obligatoire d'indiquer systématiquement ces flèches de retour, surtout lorsque celles-ci ne remmènent aucune information.
par exemple, les deux notations qui suivent sont identiques.



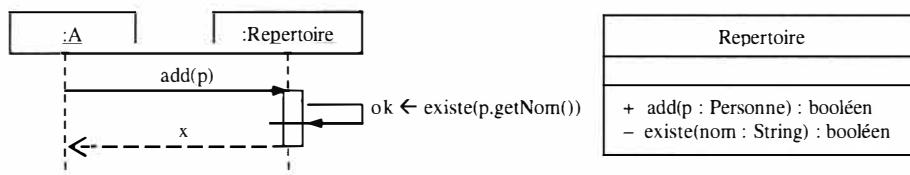
- Lorsqu'on veut représenter une opération de type «fonction» qui possède un paramètre de retour, la flèche de retour peut alors être justifiée même si on peut très bien s'en passer.
les trois notations qui suivent représentent la même chose.



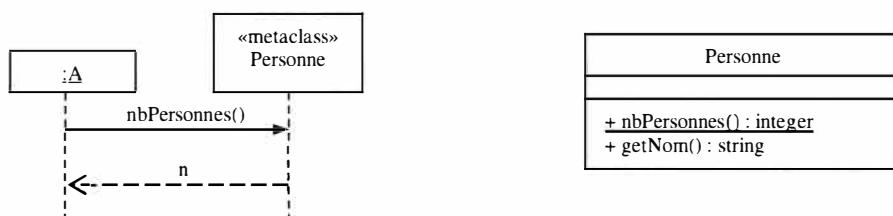
- On peut également représenter les barres d'activation qui montrent la durée de l'activité des participants, c'est à dire le temps pendant lequel la méthode est au sommet de la pile.



- Les opérations appelées sur un objet sont obligatoirement publiques. Les opérations réflexives peuvent être privées (mais pour être sûr il faudrait détailler tous les scénarios).

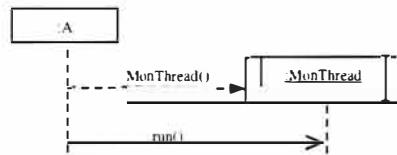


- Pour représenter l'appel d'une méthode de classe (statique), il faut indiquer que le participant récepteur est une classe. Plus précisément UML dit que le récepteur est une instance d'une métaclasses (car toute classe est une instance de la classe Class).



- Il est possible de distinguer graphiquement les envois de messages **synchrones** (que nous avons vu jusqu'à présent) de ceux qui sont **asynchrones**.

- Un message synchrone déclenche une opération uniquement lorsque le destinataire accepte le message. Une fois le message envoyé, l'expéditeur est bloqué jusqu'à ce que le destinataire accepte le message et termine son travail. Un message synchrone est représenté par une flèche pleine.
- Un message asynchrone n'attend pas de réponse et n'interrompt donc pas l'exécution de l'expéditeur. Celui-ci envoie le message sans savoir quand, ni même si, le message sera traité par le destinataire. Dans UML 1.x, un message asynchrone était représenté par une demi-flèche mais dans UML 2.0 la demi-flèche a été remplacée par une flèche bâton (même si la demi-flèche reste utilisée dans la pratique). Au niveau de la programmation, on rencontre généralement les messages asynchrones dans les applications multithreads et dans les middlewares orientés messages. Dans l'exemple suivant, la méthode run() appliquée sur une classe active est asynchrone

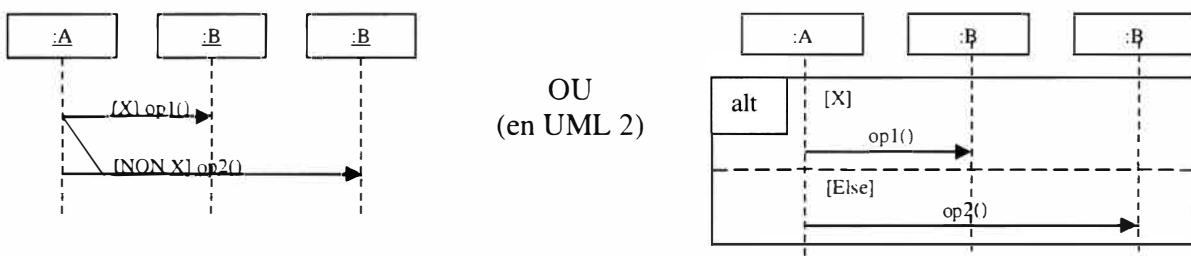


En réalité, UML propose d'autres formes de synchronisations telles que les messages dérobants (dont l'opération est déclenchée uniquement si le destinataire s'est préalablement mis en attente) ou encore les messages minutés (qui ne bloquent l'expéditeur que pendant un temps donné). Toutefois ces synchronisations sont relativement peu utilisées dans la pratique.

2.1.2 Les structures de contrôle

Lorsqu'on décide de décrire de façon détaillée l'algorithme d'une méthode ou bien lorsqu'on désire faire cohabiter plusieurs scénarios dans un même diagramme, il est possible de faire figurer des structures de contrôle dans le diagramme de séquence. Pour cela, UML1.x utilisait des marqueurs d'itération (*) et les conditions de garde ([]). UML 2.0 à lui introduit les cadres d'interactions (qui peuvent posséder les étiquettes alt, loop, opt, ref, sd, par ...).

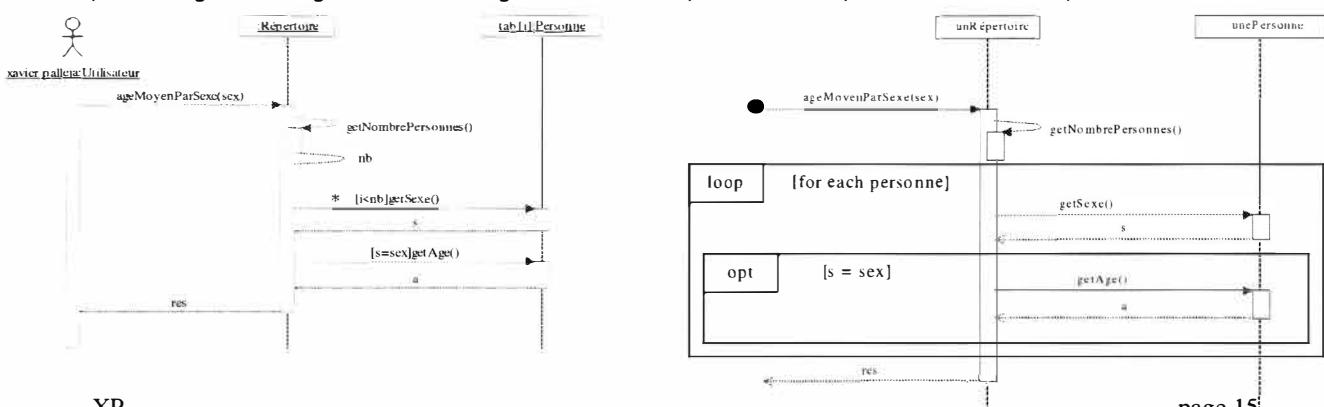
Conditionnelle



Itérative



Les deux exemples suivants illustrent l'utilisation des structures de contrôle pour décrire le fonctionnement de la méthode `ageMoyenParSexe(s)` de la classe `Repertoire`. On voit bien que la notation UML2 est plus riche car, elle permet de représenter l'imbrication des structures de contrôle (elle permet également, grâce au message trouvé, de ne pas avoir à représenter un acteur).



2.2 Le diagramme des interactions au sein d'un processus de développement

En conception, dans un processus de développement tel que UP, le Diagramme de Séquence des Interactions permet de faire le lien entre l'analyse et la conception.

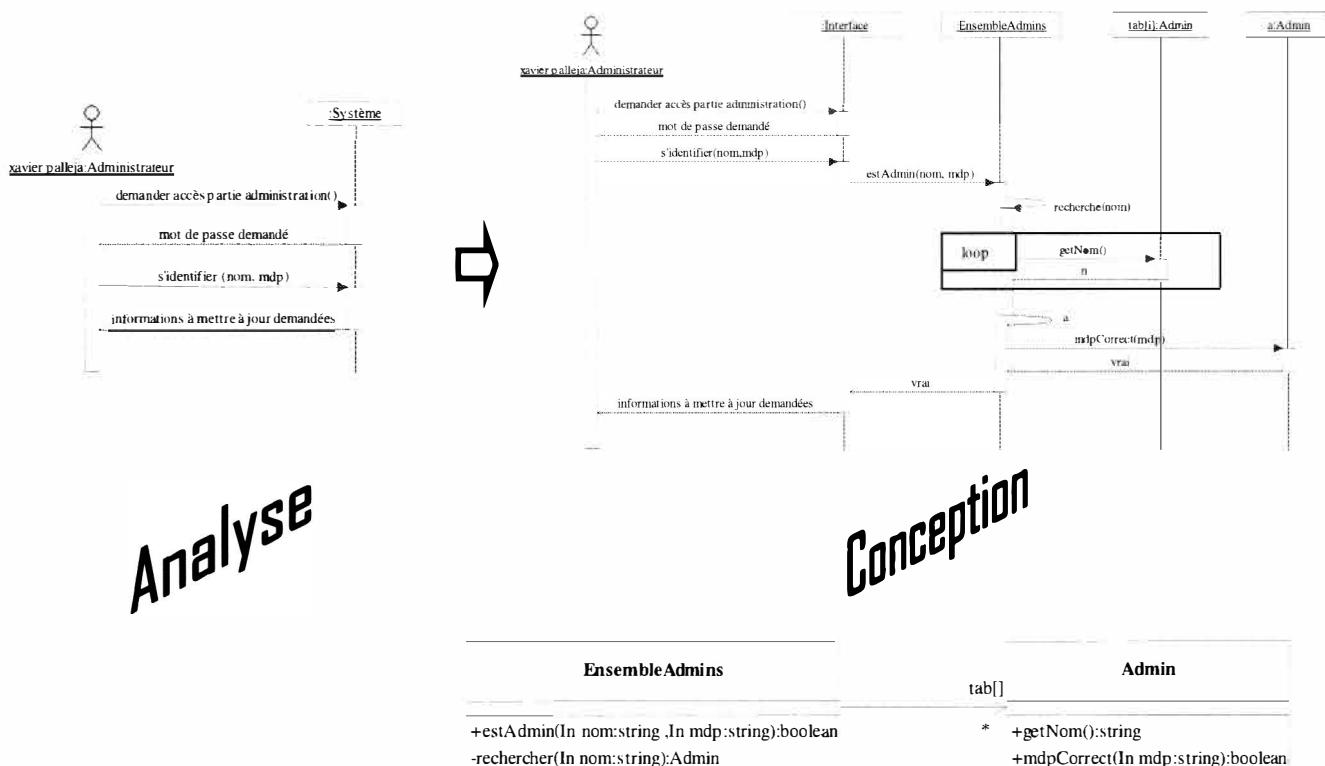
Nous allons commencer par voir comment, grâce à ce diagramme, il est possible **d'enrichir les scénarios issus de la phase d'analyse** (voir à ce sujet le dossier A) et de trouver les opérations qui doivent apparaître dans le diagramme de classes de conception. Pour rendre les diagrammes des interactions plus lisibles et pour aider le concepteur à choisir la meilleure architecture, il est possible d'estampiller les classes des objets participants en fonction de leur type («entité», «contrôle», «dialogue»). Pour cela on utilise généralement les **stéréotypes proposés par I. Jacobson**. Quoi qu'il en soit, lors de la phase de conception, le développeur doit faire des choix sur la façon dont les objets doivent collaborer. A ce sujet, deux stratégies s'opposent : **le contrôle centralisé, et le contrôle distribué**. Enfin, nous verrons que la finalité du diagramme de séquence peut être détournée de son objectif original pour modéliser du code écrit dans un **langage non orienté objet** ou encore pour représenter le **workflow** entre les travailleurs métier.

2.2.1 Le Diagramme de Séquences des Interactions (DSI) pour modéliser un scénario

Dans un processus de développement, les diagrammes de séquence des interactions sont théoriquement utilisés en pro-ingénierie. Ils sont construits à partir des Diagrammes de Séquence Système (DSS) issus des cas d'utilisation. Ils permettent de décrire de façon détaillée le fonctionnement interne de la boîte noire "Système" en indiquant les différentes interactions qui ont lieu entre les objets qui se trouvent dans ce système.

Le DSI permet de mettre en évidence les méthodes qu'il est utile de définir dans le diagramme des classes de conception. Lorsque la conception est centrée sur les objets des classes entité (qui, comme nous le verrons plus tard, correspondent généralement aux objets des classes persistantes) il n'est pas utile d'encombrer le diagramme avec plusieurs objets des classes graphiques. Pour cette raison il est possible d'utiliser un objet anonyme d'une pseudo-classe (ou paquetage) '*InterfaceGraphique*' qui permet au concepteur de ne pas avoir à se soucier de la façon dont l'acteur du scénario communique avec le système.

Exemple : soit la modélisation du site Web vu dans le dossier A. En conception, il est possible de détailler le DSS du scénario nominal du cas d'utilisation "S'identifier" grâce à un DSI :



2.2.2 Utilisation des stéréotypes de I. Jacobson et architectures en couches

On rappelle que les stéréotypes permettent de cataloguer les classes selon des critères définis par le développeur. Il est possible d'utiliser des stéréotypes déjà prédefinis (par exemple «*actor*» et «*interface*») ou bien, si cela n'est pas suffisant, de définir ses propres stéréotypes. Dans tous les cas, on peut associer à un stéréotype une représentation graphique particulière (par exemple le stickman pour le stéréotype «*actor*» ou bien la lollipop pour le stéréotype «*interface*»).

Dans un diagramme de conception (un diagramme de classes de conception ou bien un DSI) peuvent cohabiter des classes qui ont des profils très différents. Ainsi, on peut retrouver dans un même diagramme de conception des *classes entités* qui sont généralement issues du diagramme des classes persistantes (mais pas obligatoirement ; en effet certaines entités utiles au fonctionnement de l'application peuvent être non persistantes). On y trouve aussi des objets de '*dialogue*' qui prennent en charge l'interface graphique de l'application. Enfin, il peut y avoir des objets de '*contrôle*' qui, dans *une architecture en couches*, vont garantir une certaine évolutivité en isolant la logique métier des classes de dialogue.

Afin de distinguer ces trois profils, il est possible d'attribuer un stéréotype à chacune des classes du diagramme. Ces stéréotypes peuvent, soit être définis textuellement entre guillemets, soit être représentés par des icônes. Même si les créateurs d'UML n'ont pas défini l'aspect que doivent avoir ces icônes, les utilisateurs d'UML ont pris l'habitude d'utiliser les représentations graphiques suivantes qui ont été proposées par **I. Jacobson** :

-  « entité » : Classes qui regroupent les objets métiers de l'entreprise. Généralement ces classes sont persistantes mais ce n'est pas toujours le cas.
-  « dialogue » : Classes qui jouent le rôle d'interface avec l'utilisateur Généralement des fenêtres, de frames, des formulaires, des écrans ...
-  « contrôle » : Classes qui assurent la logique de l'application ; Elles **s'occupent du bon déroulement des scénarios**. Ces classes contiennent généralement uniquement des opérations (peu voire pas d'attributs). Elles **vérifient que les informations envoyées par l'interface graphique sont exploitables et garantissent une certaine indépendance entre les données et l'interface** (paradigme MVC – Modèle, Vue, Contrôleur).

Exemple de diagramme des classes de conception utilisant des stéréotypes :

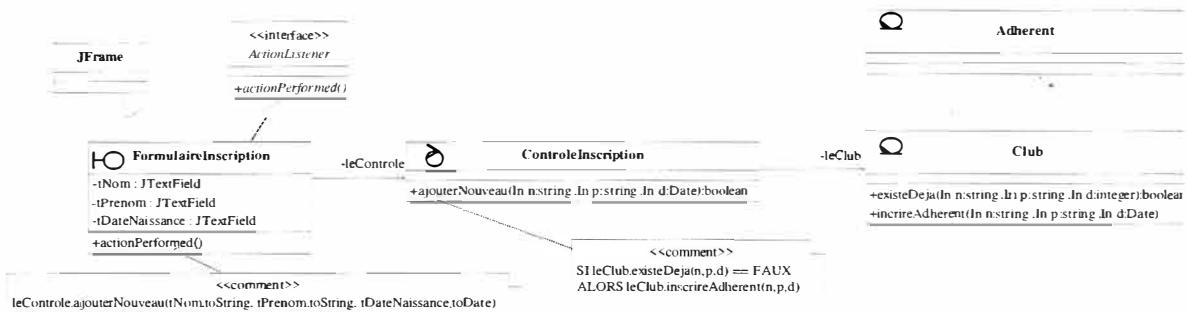
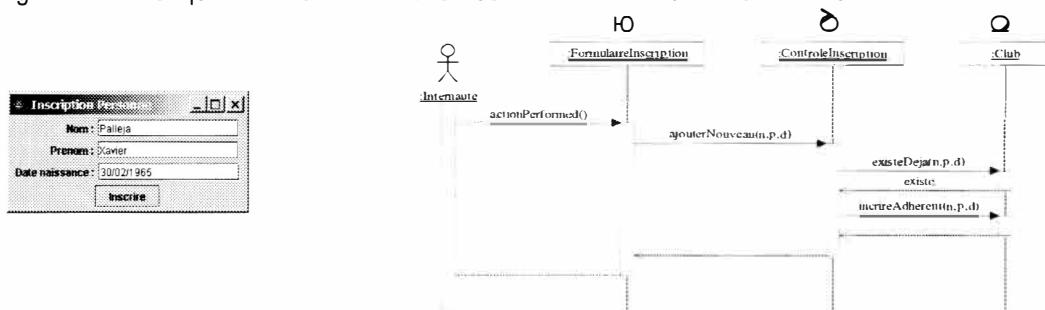


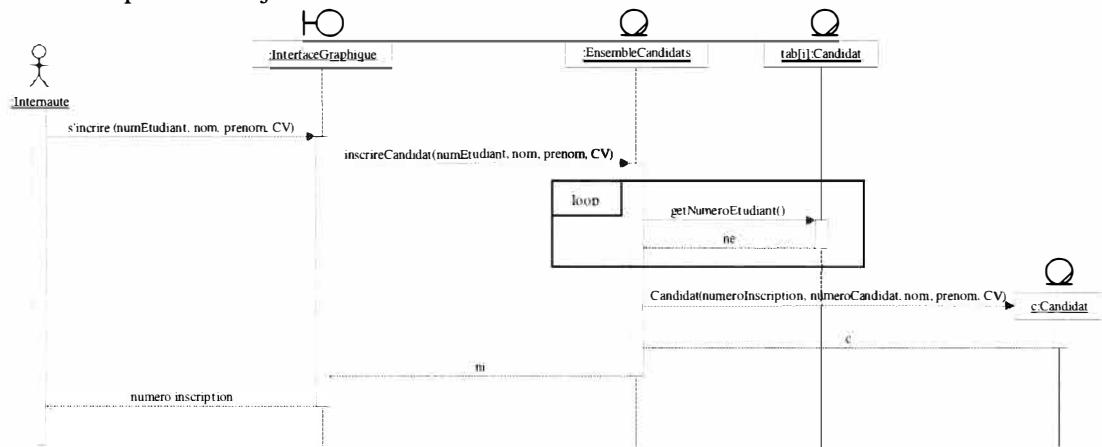
diagramme de séquence des interactions illustrant l'architecture mise en œuvre



Remarques :

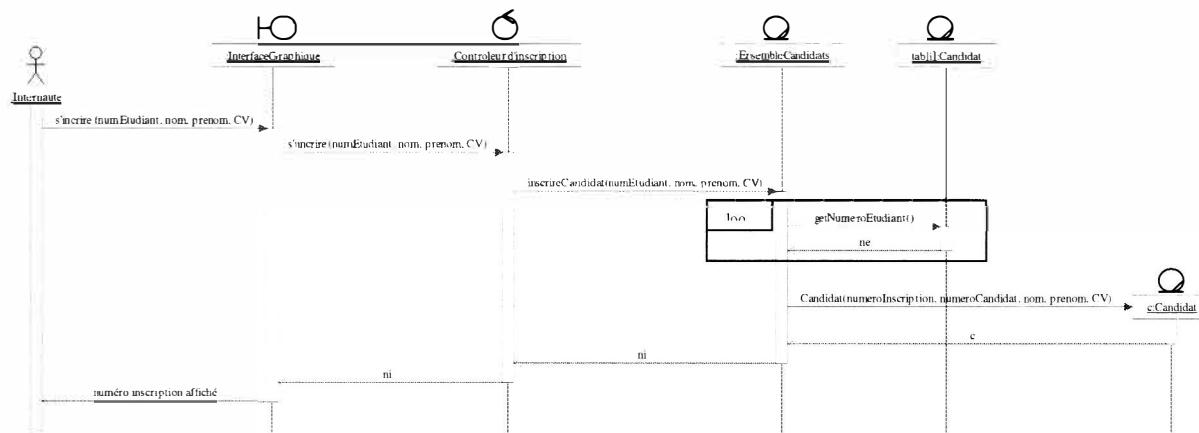
- Le concept de stéréotype est utilisé dans de nombreux autres diagrammes d'UML.
- Il existe beaucoup d'autres stéréotypes prédefinis (comme par exemple «*interface*»). Si cela n'est pas suffisant, le concepteur peut en inventer de nouveaux.
- Le paradigme MVC est fréquemment utilisé dans le développement Web. Dans les applications JEE, le modèle est assumé par des EJB ou des JavaBeans, les contrôleurs par des servlets et les vues par des JSP. On parle alors de modèle MVC2 (voir dossier F).

Les stéréotypes de Jacobson peuvent être utilisés pour illustrer le type d'architecture qui a été choisie par le concepteur. Ainsi, si on prend comme exemple l'exercice E04, on a vu qu'il était possible d'implémenter un scénario de plusieurs façons différentes. Si les objets «dialogue» communiquent directement avec les objets «entité», on parle alors **d'architecture en deux couches** (la couche **présentation** pour les objets «dialogue» et la couche **métier** pour les objets «entité»).



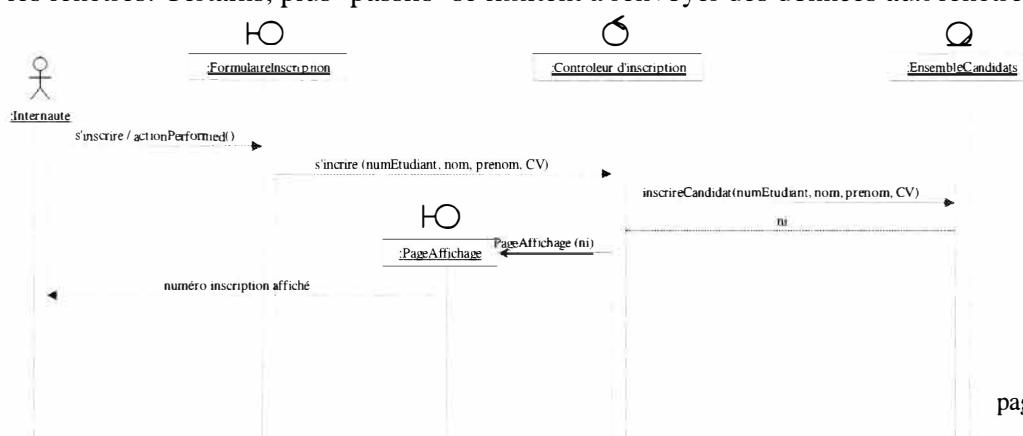
Il est à remarquer que dans le diagramme ci-dessus, si on avait estimé que l'objet de la classe EnsembleCandidat était de type «contrôle» on aurait alors eu une architecture en trois couches. Dans ce cas-là, il aurait fallu que EnsembleCandidat soit le «contrôle» de tous les cas d'utilisation.

Si par contre, on souhaite assurer une certaine évolutivité à l'application, il est alors possible d'utiliser des classes de «contrôle» pour isoler la logique métier des classes de dialogue. On obtient alors une **architecture en trois couches** (en plus des couches **présentation** et **métier**, la couche **application** regroupe tous les objets de type «contrôle») .



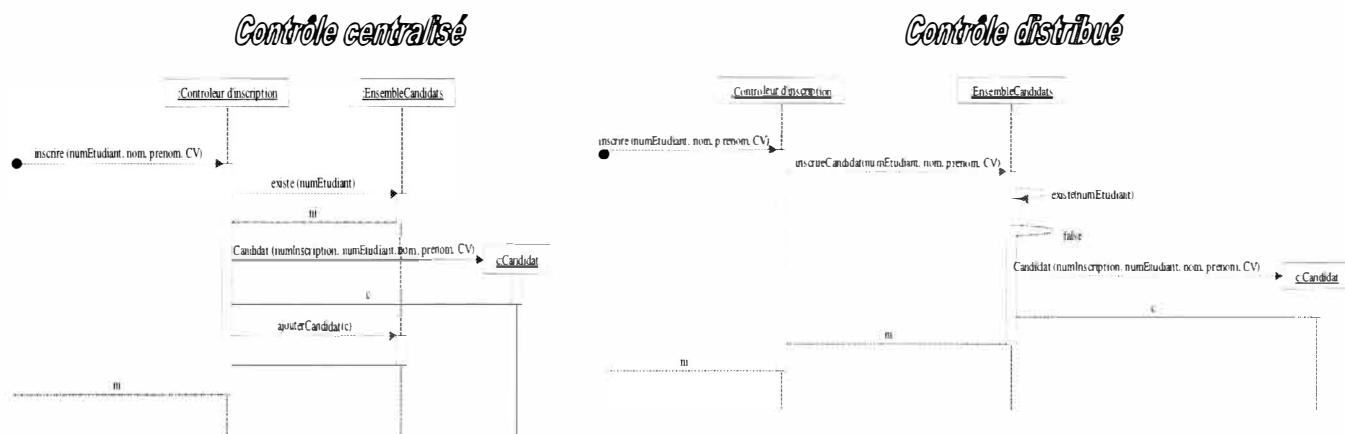
Que l'on choisisse une architecture en deux couches ou bien une architecture en trois couches, le diagramme de séquence des interactions peut également permettre de représenter les différents objets graphiques de l'application. Pour cela, on décompose la pseudo-classe IntefaceGraphique – qui est en fait un paquetage – en plusieurs classes de conception.

Dans l'exemple suivant, on a affaire à un **contrôleur ‘actif’** qui se charge lui-même de gérer et créer les fenêtres. Certains, plus ‘passifs’ se limitent à renvoyer des données aux fenêtres.



2.2.3 Contrôle centralisé / Contrôle distribué

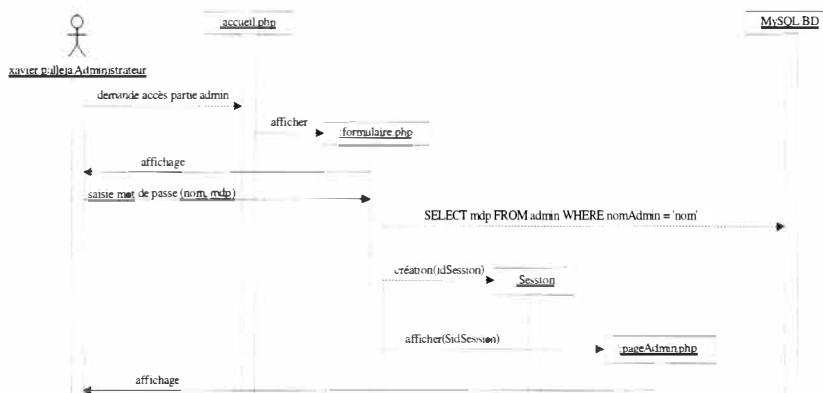
Quelle que soit l'architecture choisie (deux ou trois couches), le concepteur peut adopter soit un contrôle **centralisé** soit un contrôle **distribué**. Lorsque le contrôle est distribué, les traitements sont répartis entre différents objets, chacun d'eux se chargeant d'une partie de l'algorithme. Avec un contrôle centralisé, un objet effectue la majeure partie des traitements, les autres n'étant là que pour fournir des données. Même si "l'esprit objet" se prête davantage à un contrôle distribué, l'utilisation d'objets de «contrôle» peut favoriser une vision centralisée.



2.2.4 Autres utilisations du diagramme de séquence en conception

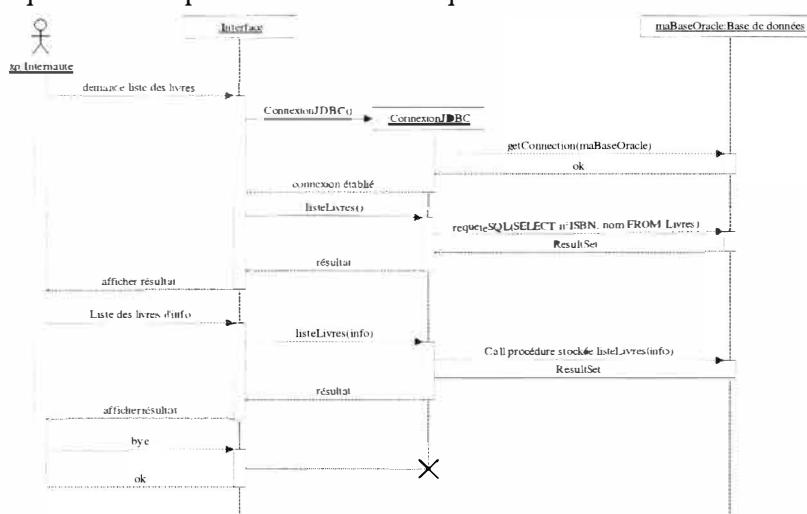
Recherche et description des messages dans un langage non orienté objet

Le diagramme de séquence des interactions peut être utilisé même lorsque le langage utilisé en programmation n'est pas orienté objet. Dans l'exemple suivant, les envois de messages permettent de mettre en évidence l'enchaînement des différents scripts php4.



Recherche et description des messages représentant l'accès à une base de données

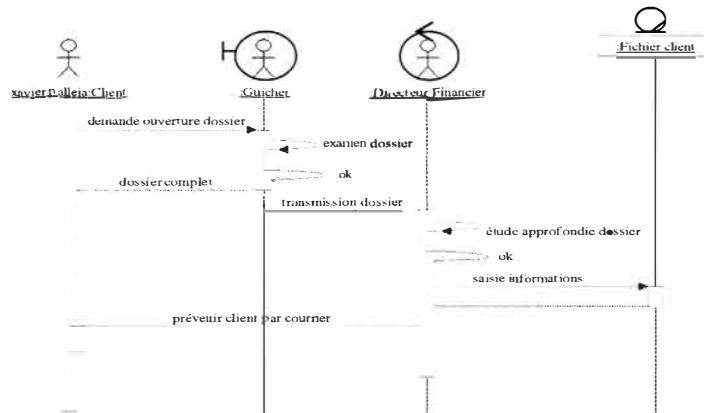
Comme nous venons de le voir, le diagramme de séquence des interactions peut permettre d'indiquer à quels moments une application accède à une base de données. Il est alors possible de décrire les requêtes ou les procédures stockées qui sont exécutées sur le SGBD.



Utilisation du DSI dans la conception d'un système d'information (métier)

Enfin, on peut également utiliser le DSI à la place du diagramme d'activités pour modéliser le fonctionnement du futur Système d'Information. Les objets du diagramme vont alors représenter les différents acteurs internes du système étudié. Les envois de messages entre ces acteurs, vont permettre de décrire le workflow.

Exemple de DSI permettant de décrire le scénario nominal du cas "ouverture de compte" dans le **système d'information agence bancaire** que nous avons vu précédemment (dossier A).



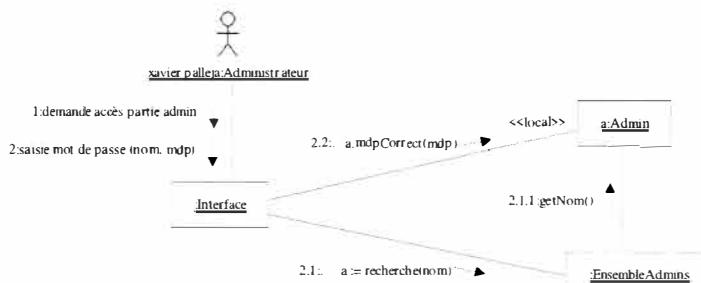
2.3 Autre diagramme décrivant les interactions : Le diagramme de communication

La description des interactions peut aussi être réalisée à l'aide du diagramme de communication (diagramme de collaboration dans UML 1.x). Le diagramme de communication est très proche du diagramme de séquence. Toutefois, si ce dernier met plutôt l'accent sur la chronologie des différentes interactions, le diagramme de communication privilégie, quant à lui, la mise en évidence des liens entre les objets qui participent au scénario.

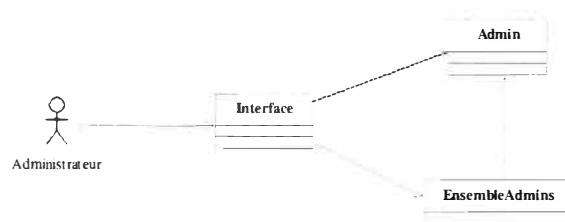
Ces liens, qui sont des instances d'associations, vont aider le concepteur à trouver les différentes associations du diagramme de classes. De plus, la direction des messages qui se trouvent sur un lien va apporter des éléments importants dans le choix de la navigabilité d'une association. Les liens temporaires (que l'on peut repérer grâce au stéréotype «*local*») vont, dans le diagramme de classes, donner lieu à des relations de dépendance.

Enfin, même si ce n'est pas son point fort, le diagramme de communication peut aussi permettre, tout comme le diagramme de séquence, de représenter l'imbrication des appels des méthodes. Cela est réalisé grâce à une notation décimale imbriquée normalisée. Seule cette notation permet de lever l'ambiguïté liée aux auto-appels.

Par exemple, dans notre site Web, le scénario nominal du cas d'utilisation "S'identifier" peut être représenté comme suit : (ici on a une architecture à trois niveaux et un contrôle centralisé)

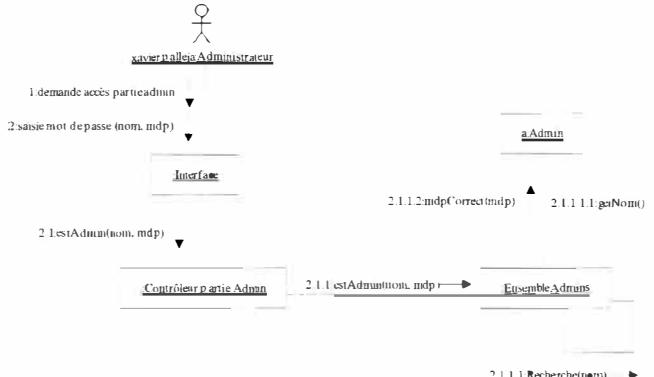


De ce diagramme on peut déduire aisément le diagramme de classes :



Toutefois, lorsque le contrôle est distribué, la multiplication des appels de méthodes peut rendre le système de numérotation relativement indigeste.

Toujours dans notre exemple, on utilise ici un contrôle distribué (avec une architecture en trois couches). Ici la numérotation peut être difficile à suivre.



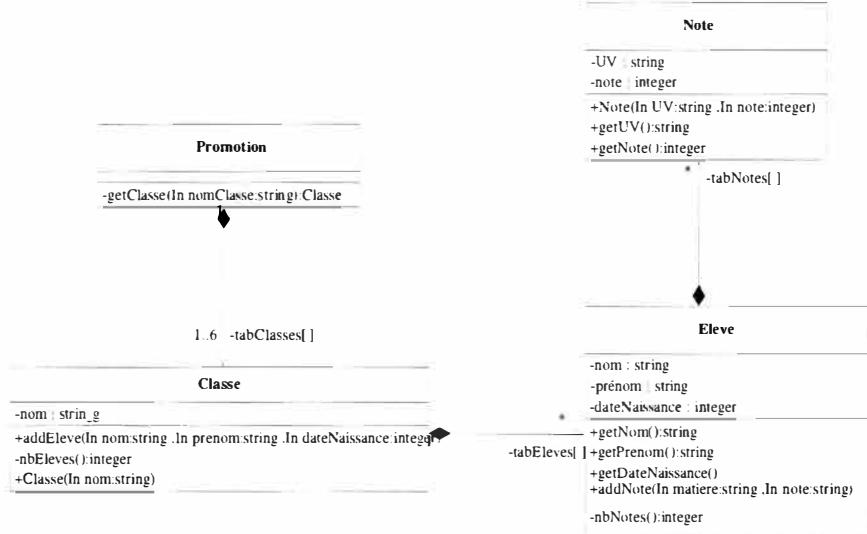
Exercices

(E01 suite) Diagrammes de séquences - Rétro-ingénierie

A partir du code de l'exercice E01, réaliser le diagramme de séquences de la méthode `totalPaieAvecAides()` de la classe `Entreprise`. On fera apparaître sur le diagramme le fonctionnement des méthodes qui sont appelées directement par `totalPaieAvecAides()`.

(E03) Diagramme de séquences - Notes de l'IUT de Pointe-à-Pitre

On décide de réaliser un programme Java permettant de gérer les notes des étudiants de licence professionnelle de l'IUT de Pointe-à-Pitre. Devant le succès de sa LP, l'IUT de Pointe-à-Pitre a décidé de diviser sa promotion de licence en un ensemble de classes. L'attribut `note` de la classe `Note` correspond à la moyenne annuelle de l'étudiant pour une UV. L'analyse statique a permis de dégager le diagramme des classes de conception suivant :



N.B. : `tabEleves[]` et `tabNotes[]` sont des tableaux de 100 éléments dont seulement les x premières cases sont remplies.

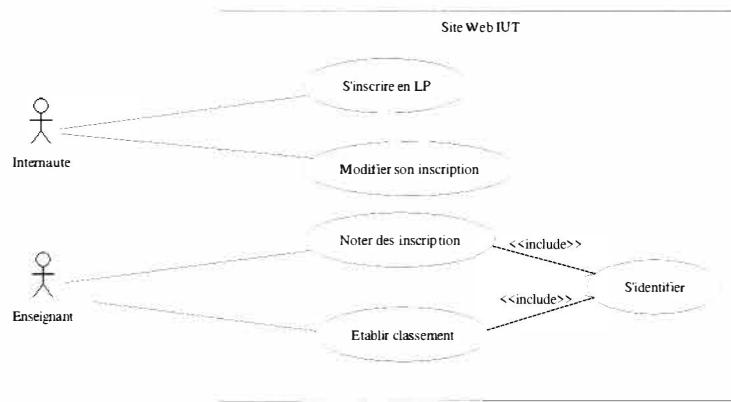
On désire écrire dans la classe `Promotion` une opération publique qui calcule la moyenne générale d'une classe dont le nom est passé en paramètre. La signature de cette opération doit être la suivante : `+ moyenneClasse (nomClasse : string) : real`

Travail à faire :

A l'aide d'un diagramme de séquence, illustrez la façon dont vous programmez la méthode `moyenneClasse()`.

Remarque : vous pouvez rajouter dans le diagramme de classes toutes les méthodes dont vous avez besoin. Si vous utilisez ces nouvelles méthodes dans le diagramme de séquences vous devrez alors illustrer leur fonctionnement.

(E04) Diagramme de séquences - Gestion des inscriptions en LP



Cas d'utilisation "S'inscrire en LP" :

- Dans le scénario nominal l'Internaute saisit son numéro d'étudiant, son nom, son prénom et des informations qui concernent son CV. Le système valide sa candidature et retourne le numéro d'inscription de l'internaute
 - Scénario d'erreur : s'il existe déjà une candidature qui possède le même numéro d'étudiant, l'inscription est rejetée et l'internaute en est averti.

Cas d'utilisation "Modifier son inscription" :

- Dans le scénario nominal l'Internaute saisit son numéro d'étudiant et son numéro d'inscription. Le système affiche toutes les informations qui concernent la candidature. L'internaute modifie alors les informations du CV. Le système indique que les modifications ont été prises en compte.
 - Scénario d'erreur : si le numéro d'inscription ne correspond pas au numéro de l'étudiant le système indique qu'il y a une erreur de saisie

Cas d'utilisation "Noter des inscriptions" :

- Dans le scénario nominal l'enseignant saisit le numéro d'inscription du candidat qu'il souhaite noter. Le système lui affiche le CV du candidat. L'enseignant attribut une note à la candidature. Le système indique que la note a été enregistrée.

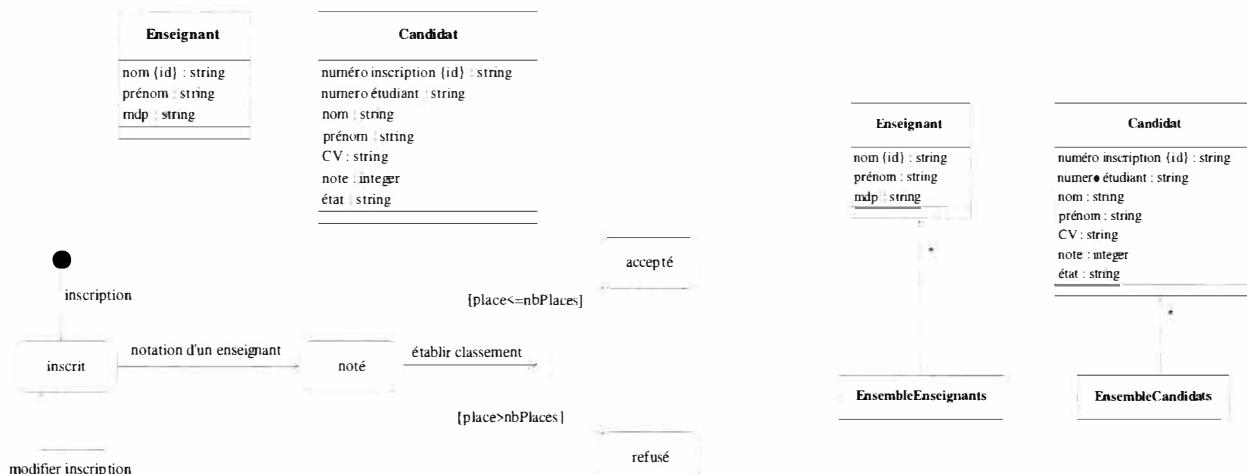
Cas d'utilisation "Etablir classement" :

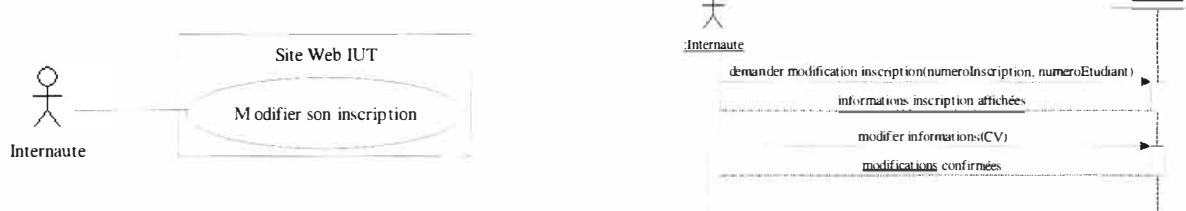
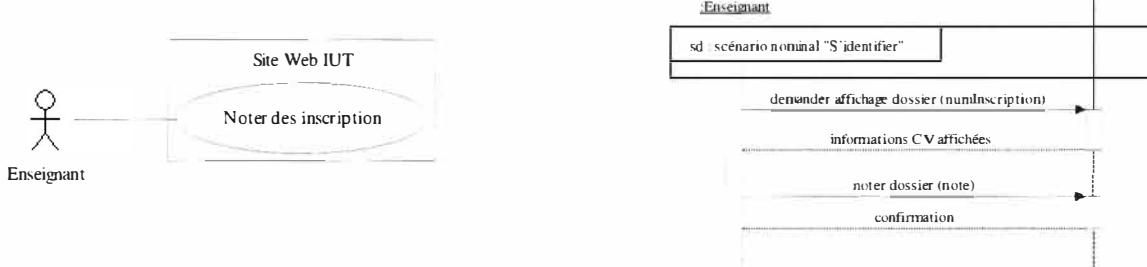
- Dans le scénario nominal l'enseignant saisit le nombre de places disponibles en LP, et demande au système de classer les candidats inscrits en fonction de la note qu'ils ont obtenue. Le système classe les étudiants leur attribut un statut (accepté ou refusé) et affiche le résultat

Cas d'utilisation "S'identifier" :

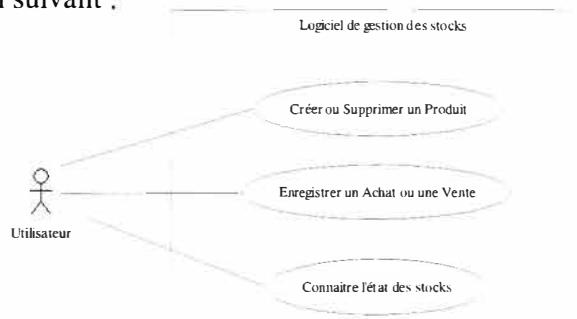
- Dans le scénario nominal l'enseignant saisit son nom et son mot de passe. Le système donne alors l'accès à la partie administration et demande à l'utilisateur ce qu'il souhaite faire.
 - Scénario d'erreur : si le nom de l'enseignant n'est pas connu ou bien si le mot de passe n'est pas correct un message d'erreur est renvoyé à l'enseignant

Les phases d'analyse et de conception préliminaire ont permis de dégager les diagrammes suivants (diagramme des classes persistantes, ébauche du diagramme des classes de conception et diagramme états-transitions de la classe Candidat) :



Cas d'utilisation "S'inscrire en LP" :**Cas d'utilisation "Modifier son inscription" :****Cas d'utilisation "S'identifier" :****Cas d'utilisation "Noter des inscriptions" :****(E05) Programmation d'une application en trois couches**

On désire réaliser une application permettant de gérer les stocks de produits d'une entreprise marchande. Les fonctionnalités attendues par cette application sont décrites par le diagramme des cas d'utilisation suivant :

**Remarque :**

En pratique, il serait plus judicieux de regrouper ces trois cas dans un seul cas "gérer les produits". Mais pour des raisons pédagogiques nous allons quand même garder ces trois cas.

Cas d'utilisation "Créer ou Supprimer un produit" :

- Dans le scénario nominal, l'utilisateur demande la création d'un nouveau produit. Le système affiche un formulaire où l'utilisateur peut saisir le nom du produit, le prix unitaire du produit et la quantité en stock. L'utilisateur valide ensuite le formulaire. Le système enregistre alors le nouveau produit
- Scénario d'erreur : s'il existe déjà un produit qui possède le même nom, la création est annulée et l'utilisateur en est averti.
- Scénario d'erreur : si le prix du produit n'est pas correct (non numérique ou négatif), la création est annulée et l'utilisateur en est averti.

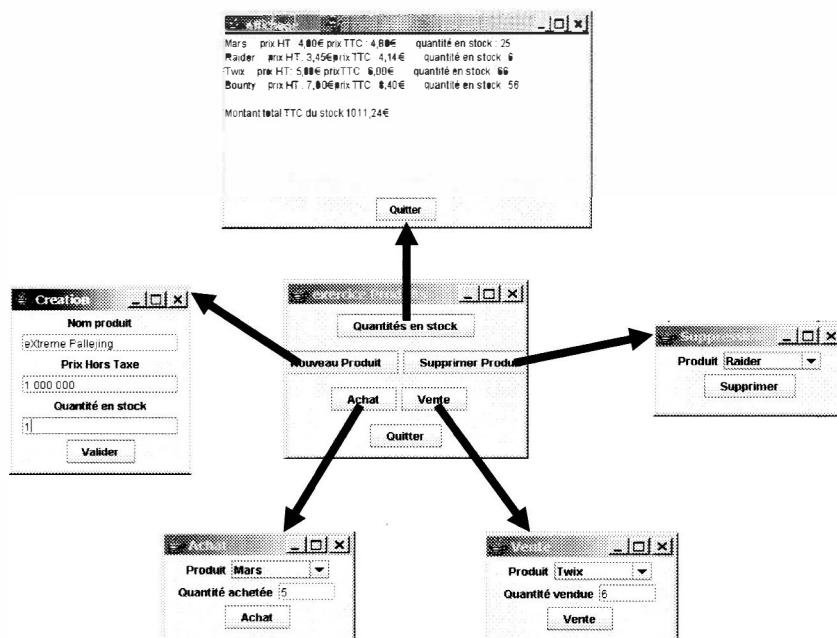
Cas d'utilisation "Enregistrer un achat ou une vente" :

- Dans le scénario nominal, l'utilisateur demande l'enregistrement d'un achat. Le système affiche un formulaire où l'utilisateur peut saisir le nom du produit acheté ainsi que la quantité achetée. L'utilisateur valide ensuite le formulaire et le système enregistre alors l'achat.
- Scénario d'erreur : si la quantité est négative ou nulle ou non numérique l'achat n'est pas enregistré et l'utilisateur en est averti.

Cas d'utilisation "Connaître l'état des stocks" :

- Dans le scénario nominal, l'utilisateur demande l'affichage de l'état des stocks. Le système affiche alors dans une fenêtre le nom, le prix unitaire HT, le prix unitaire TTC et la quantité en stock de chaque produit. Le système affiche également la valeur totale TTC du stock.

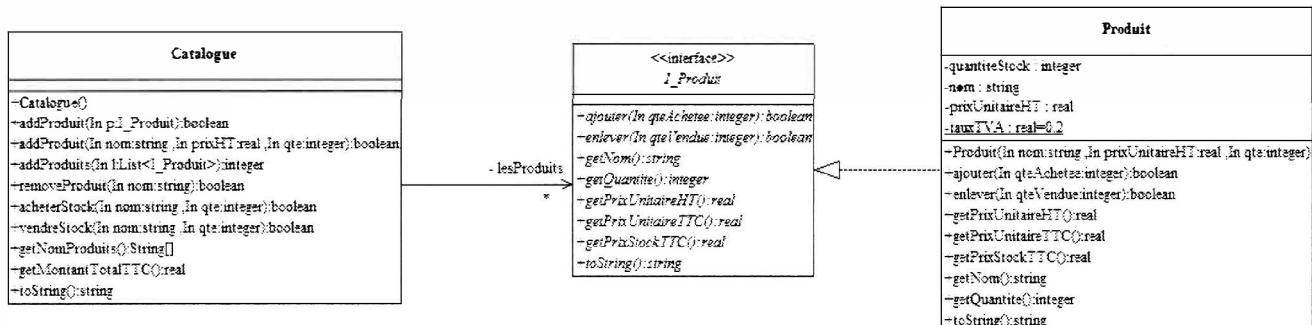
L'interface graphique du programme doit ressembler à ceci :



Travail à faire :

Dans un premier temps, nous n'allons pas nous préoccuper de la persistance des données ; on souhaite simplement développer notre application selon une architecture en trois couches.

- les classes de la *couche de Présentation* sont disponibles sur le réseau
- la *couche Métier* doit être composée de deux classes. Une classe Produit et une classe Catalogue qui contient la liste de tous les produits en stock. La phase de conception a permis de déterminer que ces classes doivent posséder, au moins, les méthodes suivantes.



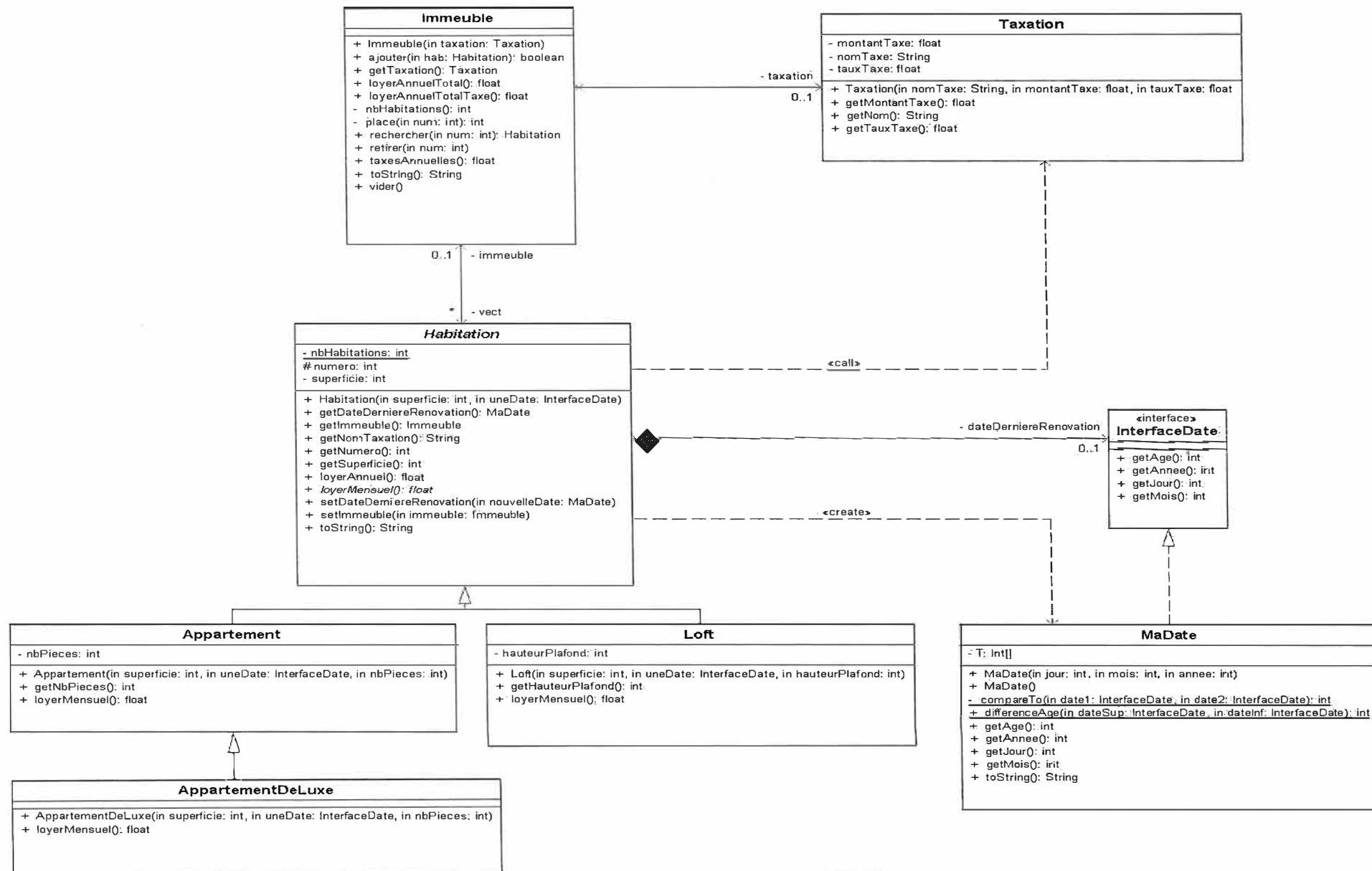
- une *couche Application* composée de toutes les classes de « contrôle ». On souhaite posséder **un contrôleur pour chacun des cas d'utilisation**. Il est important que tous les contrôleurs **partagent le même objet Catalogue** (en effet les produits pour lesquels il doit être possible d'enregistrer un achat doivent être les mêmes que les produits pour lesquels on doit pouvoir consulter l'état des stocks). Pour cette raison il ne faut pas que chaque contrôleur crée une instance différente de Catalogue ...

On vous demande de créer un paquetage différent pour chacune de ces couches.

(E02) Diagrammes de classes de conception - Utiliser l'AGL Eclipse

1. Démarrez votre machine sous le système d'exploitation Windows XP (ou bien sous une version moins élaborée de Windows). Puis, copiez le fichier "TP Eclipse.zip" sur la racine du disque D:\ de votre machine.
2. Puis, dézippez "TP Eclipse.zip" sur la racine du disque D (cliquer sur "TP Eclipse.zip" avec le bouton droit, puis dans le menu contextuel, 7-Zip - Extraire Ici).
3. Lancez ensuite Eclipse 3.2 avec le raccourci « Lancement Eclipse 3.2.lnk » qui se trouve dans le répertoire D:\TP Eclipse\clipse 3.2.
4. Créez un projet '*Locations*'. Ne créez pas de paquetage dans ce projet et importez directement dans le projet les classes de l'exercice précédent que vous trouverez sur l'ENT (répertoire LP/UML/Exercice E02).
5. Demandez à éclipse d'utiliser une version récente de Java (cet Eclipse est ancien et utilise par défaut une vieille version de Java). Aller dans menu *Fenêtre – Préférences* ... puis cliquez sur l'intitulé *Java – Compilateur* et choisir la dernière version (au moins le 5.0) pour le *niveau de compatibilité du compilateur*
6. Demandez à éclipse d'afficher les diagrammes selon la norme UML 2.x. Aller dans menu *Fenêtre – Préférences* ... puis cliquez sur l'intitulé *UML* et indiquez respectivement les valeurs *UML Standard* et *UML 2.0* pour la *Présentation des diagrammes* et la *Notation*.
7. Demandez ensuite à Eclipse de générer automatiquement le diagramme de classes de votre projet (dans le menu choisir : *Fichier – Nouveau – Autre* ... puis sélectionnez *UML Diagrams* et enfin *UML Class Diagram*). Vous demanderez de générer au moins les liens d'héritage et les associations (il est préférable de lui demander de ne pas représenter toutes les dépendances qui risquent d'alourdir le diagramme).
8. Pour modifier les options affichées dans le diagramme de classes, vous pouvez cliquer sur le diagramme avec le bouton droit de la souris puis choisir *preferences* dans le menu contextuel.
9. Modifiez ensuite le diagramme de classes et observez les changements opérés dans le code source ; par exemple, dans la classe AppartementDeLuxe ajoutez un attribut superficiePiscine de type entier. Pour cela, dans le diagramme de classes, cliquez sur la classe AppartementDeLuxe avec le bouton droit de la souris et sélectionnez *nouveau* dans le menu contextuel. Ajoutez ensuite une classe dans votre diagramme (par exemple AppartementDeLuxeTresCher qui hérite de AppartementDeLuxe et qui possède un attribut superficieParc).
10. Inversement, modifiez le code source (en ajoutant par exemple une nouvelle méthode) puis observez les changements opérés dans le diagramme de classes.
11. Demandez maintenant à Eclipse de représenter le diagramme de séquence qui correspond au fonctionnement de la méthode loyerAnnuelTotalTaxe() de la classe Immeuble.
Pour cela, dans le diagramme de classes, cliquez sur la méthode voulue avec le bouton droit de la souris et sélectionnez *Open* puis *Nouveau/Ouvrir* un diagramme de séquence dans le menu contextuel. Il est ensuite possible de détailler le fonctionnement de chacune de méthode du scénario obtenu. Par exemple, demandez de détailler dans le même diagramme la méthode loyerAnnuelTotal(). Pour cela, cliquez avec le bouton droit de la souris sur la flèche qui correspond à l'appel de la méthode. Puis dans le menu contextuel sélectionnez *Open* puis *Extend Diagram for this statement*. Sur le diagramme obtenu, Faire ensuite la même chose avec la méthode taxesAnnuelles().
12. Modifier ensuite le code de votre application pour que les immeubles qui possèdent moins de 10 habitations, aient une taxe annuelle équivalente à 10% du loyer annuel total si toutes les habitations ont été rénovées lors des 10 dernières années (pour cela on regardera l'âge de la dernière rénovation des habitations) ; et de 15% du loyer annuel total s'il existe une des habitations qui n'a pas été rénovée lors des 10 dernières années. Générez ensuite le diagramme de séquence de la méthode taxesAnnuelles() de la classe Immeuble.
13. Modifiez maintenant le code de la méthode taxesAnnuelles() pour que le montant de la taxe des immeubles qui possèdent un nombre d'habititations inférieur à 10, ne soit plus calculé en fonction de l'âge de rénovation des habitations au moment où on lance la méthode mais de l'âge de la rénovation qu'auront les habitations au 31 décembre de l'année en cours. Régénérez ensuite le diagramme de séquence de la méthode taxesAnnuelles() de la classe Immeuble.
14. **Avant de partir, supprimez le répertoire "D:\TP Eclipse" qui se trouve sur la racine de D.**

(E02) Diagrammes de classes de conception - Utiliser l'AGL Eclipse



(E02 suite) Diagrammes de séquences - Rétro-ingénierie

