

Annexe B

Diagramme de classes UML

UML (*Unified Modeling Language*) a été créé en 1997 pour être le langage standard de modélisation orienté-objet. UML contient différents diagrammes utilisés pour décrire de nombreux aspects du logiciel. Dans le cadre de ce cours, nous utiliserons uniquement le **diagramme de classes**¹.

Le diagramme de classes représente la structure statique du logiciel. Il décrit l'ensemble des classes qui sont utilisées ainsi que leurs associations. Il est inspiré des diagrammes Entité-Relation utilisés en modélisation de bases de données, en y ajoutant les aspects opérationnels (les méthodes) et quelques subtilités sémantiques (la composition, par exemple).

B.1 Représentation des classes et interfaces

B.1.1 Les classes

En UML, une classe est au minimum décrite par un nom. Graphiquement, elle est représentée par un rectangle, éventuellement divisé en 3 parties : son nom, ses attributs et ses opérations. On appelle **membres** de la classe ses attributs et méthodes (ou opérations).

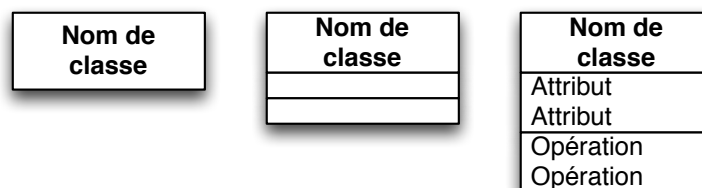


FIGURE B.1 – Représentations UML possible d'une classe, avec ou sans les membres

B.1.2 Les membres de classe

Chaque attribut est décrit au moins par un nom (unique dans une même classe) et par un type de données.

Une opération, ou méthode, est décrite au moins par un nom, par un ensemble d'arguments nécessaires à son invocation et par un type de retour. Chaque argument est décrit par un nom et un type de données.

1. le langage UML pourra être étudié en profondeur lors du cours Analyse et Conception de Systèmes Informatiques de l'axe Ingénierie des Systèmes Informatiques

Un niveau de **visibilité** est également attribué à chaque membre. La visibilité d'un membre d'une classe définit quelles autres classes y ont accès (en terme de lecture/écriture). UML utilise 3 niveaux de visibilité :

- **public** (noté par +), le membre est visible par toutes les classes
- **privé** (noté par -), le membre n'est visible par aucune classe sauf celle qui le contient
- **protégé** (noté par #), le membre est visible par toutes les sous-classes de celle qui le contient (cette visibilité est expliquée ultérieurement)

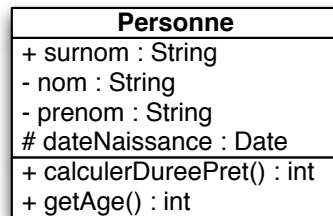


FIGURE B.2 – Exemple de classe Personne avec attributs et opérations

La figure B.2 est un exemple d'une classe Personne avec toutes les possibilités de visibilité.

B.1.3 Les classes abstraites

En UML, le mot-clé `{abstrait}` (ou `{abstract}`) est accolé aux classes et méthodes abstraites. Une autre manière souvent utilisée de représenter ces méthodes ou classes est d'écrire leur nom en italique. Les *attributs et méthodes de classe* sont soulignés. La classe Personne décrite dans la figure ci-dessous montre un exemple d'attribut et de méthode de classe.



FIGURE B.3 – Deux représentations possibles pour une classe abstraite Personne

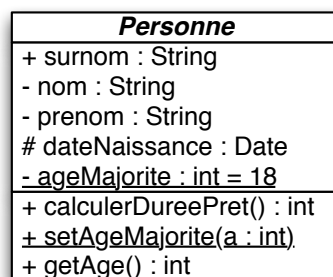


FIGURE B.4 – Exemple d'un attribut et d'une méthode de classe pour la classe Personne

B.1.4 Les interfaces

En UML, une interface est décrite par le mot-clé «interface» dans le bloc d'entête, comme présenté dans la figure B.5.

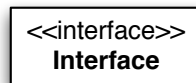


FIGURE B.5 – Exemple d’interface

B.2 Les relations

En conception orientée objet, les relations englobent notamment les relations d’héritage et de réalisation d’interface.

B.2.1 L’héritage

En UML, l’héritage se représente par une flèche à la pointe creuse. La figure B.6 décrit deux classes Super-classe et Sous-classe. La classe Sous-classe **hérite** de la classe Super-classe.

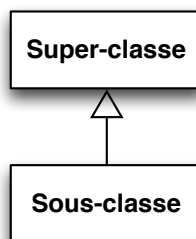


FIGURE B.6 – Exemple d’héritage

B.2.2 La réalisation

La réalisation d’une interface par une classe se représente par une flèche pointillée à pointe creuse, comme illustré dans la figure B.7.

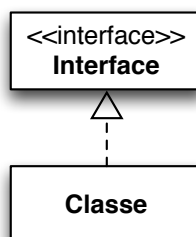


FIGURE B.7 – Exemple de réalisation de l’interface Interface par la classe Classe

B.3 Les associations

Certaines relations entre classes d’un même diagramme sont représentées en UML sous la forme d’**associations**. Le plus souvent une association ne relie que deux classes. Une association peut être identifiée par un nom et chacune de ses extrémités définit le nombre d’instances

des classes reliées qui sont impliquées dans cette association. On appelle **multiplicité** ce nombre d'instances qui peut prendre les valeurs suivantes :

Multiplicité	Interprétation
1	un et un seul
0..1	zéro ou un
N	exactement N
M..N	de M à N
*	zéro ou plus
0..*	zéro ou plus
1..*	un ou plus

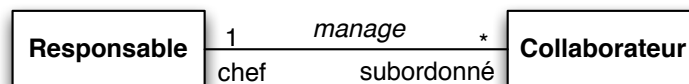


FIGURE B.8 – Exemple d'association manage

L'exemple de la figure B.8 décrit une association *manage* entre la classe *Responsable* et la classe *Collaborateur*. Un responsable gère plusieurs collaborateurs, ses subordonnés. Un collaborateur est géré par un seul responsable, son chef. Les éléments subordonné et chef sont des *rôles* d'association.

B.3.1 Direction des associations

Les associations peuvent être *dirigées*, ce qui contraint la visibilité et la navigation dans le modèle. La direction se représente par une flèche classique. Par défaut, s'il n'y a pas de flèche, l'association est bidirectionnelle (comme s'il y avait une flèche à chaque extrémité de l'association).

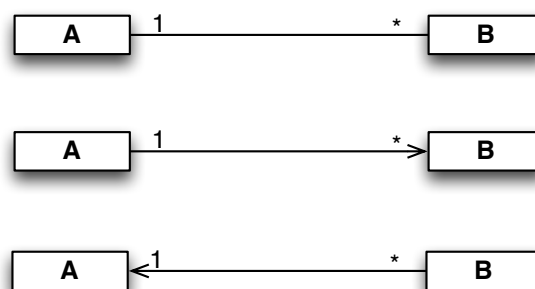


FIGURE B.9 – Exemples d'associations dirigées

La figure B.10 présente des exemples de directions, dont voici les interprétations. La première ligne signifie que A connaît tous les B auxquels elle est associée, et réciproquement, un B connaît le A auquel il est associé. La deuxième ligne signifie que seul le A connaît les B auxquels il est associé, mais pas l'inverse. Finalement, dans la troisième ligne, un B connaît le A auquel il est associé, mais pas l'inverse. En fait, ceci va impliquer la présence ou non d'un attribut *a* de type A dans la classe B ou *b* de type B dans la classe A en fonction de la direction. Par exemple, pour la deuxième ligne, A possède une liste d'objet de type B mais B ne possède pas d'attribut de type A.

B.3.2 Agrégation et composition

Deux sous-types d'associations permettent de préciser un sens particulier à ces relations : l'agrégation et la composition. Elles peuvent également être dirigées.

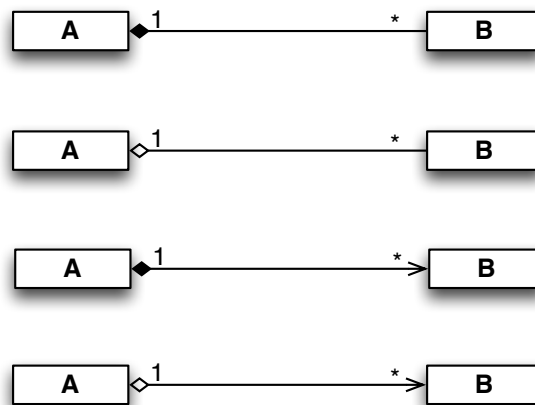


FIGURE B.10 – Exemples de compositions et d'agrégations.

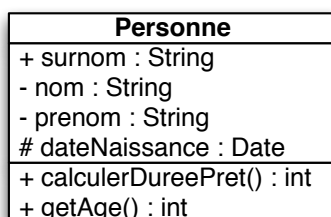
L'**agrégation** est une association avec relation de subordination, souvent nommée *possède* représentée par un trait reliant les deux classes et dont l'origine se distingue de l'autre extrémité (la classe subordonnée) par un *losange creux*. Une des classes "regroupe" d'autres classes. On peut dire que l'objet A utilise ou possède une instance de la classe B.

La **composition** est une association liant le cycle de vie des deux classes concernées. Une association de composition s'interprète comme une classe est composée de un ou plusieurs élément de l'autre classe. Elle est représentée par un trait reliant les deux classes et dont l'origine se distingue de l'autre extrémité (la classe composant) par un *losange plein*. On peut dire que l'objet A est composé instance de la classe B, et donc si l'objet de type A est détruit, les objets de type B qui le composent sont également détruit. Ce sera également souvent les objets de type A qui créeront les objets de type B.

B.4 Correspondance UML-Java

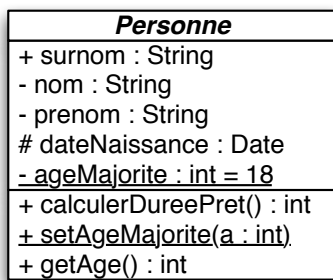
Java permet de programmer tout modèle sous forme de diagramme de classe UML tel que présenté ci-dessus. Voici quelques exemples de correspondance entre le modèle UML et le codage Java.

B.4.1 Classes et membres



```
public class Personne {
    public String surnom;
    private String prenom;
    private String nom;
    protected Date dateNaissance;
    public int calculerDureePret() {...}
    public int getAge() {...}
}
```

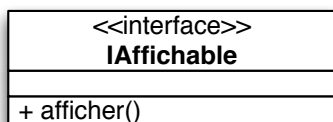
B.4.2 Classes abstraites



```

public abstract class Personne {
    public String surnom;
    private String prenom;
    private String nom;
    protected Date dateNaissance;
    private static int ageMajorite = 18;
    public int calculerDureePret() {...}
    public static void setAgeMajorite(int a) {...}
    public int getAge() {...}
}
  
```

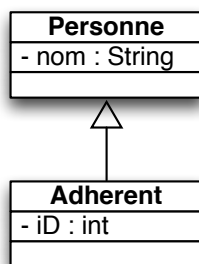
B.4.3 Interfaces



```

interface IAffichable {
    void afficher();
}
  
```

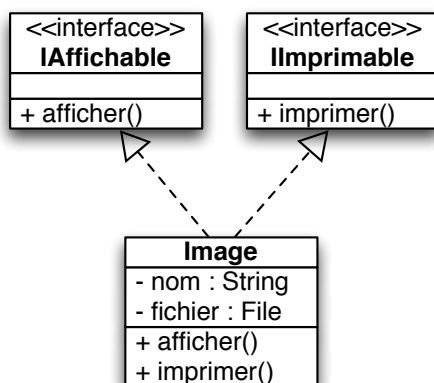
B.4.4 Héritage



```

public class Adherent extends Personne {
    private int iD;
}
  
```

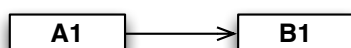
B.4.5 Réalisation



```

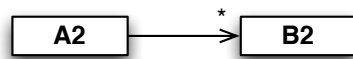
public class Image implements IAffichable, IImprimable {
    private String nom;
    private File fichier;
    public void afficher(){...}
    public void imprimer(){...}
}
  
```

B.4.6 Associations



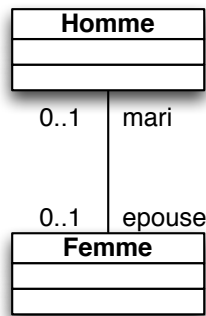
```

public class A1 {
    private B1 b1;
    ...
}
  
```



```

public class A2 {
    private ArrayList<B2> b2s ;
    ...
}
  
```

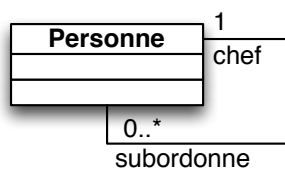


```

public class Homme {
    private Femme epouse ;
    ...
}
  
```

```

public class Femme {
    private Homme mari ;
    ...
}
  
```



```

public class Personne {
    private ArrayList<Personne> subordonnes ;
    private Personne chef ;
    ...
}
  
```

Une application réalisée selon l'approche objet est constituée d'objets qui collaborent entre eux pour exécuter des traitements. Ces objets ne sont donc pas indépendants, ils ont des relations entre eux. Comme les objets ne sont que des instances de classes, il en résulte que les classes elles-mêmes sont reliées.

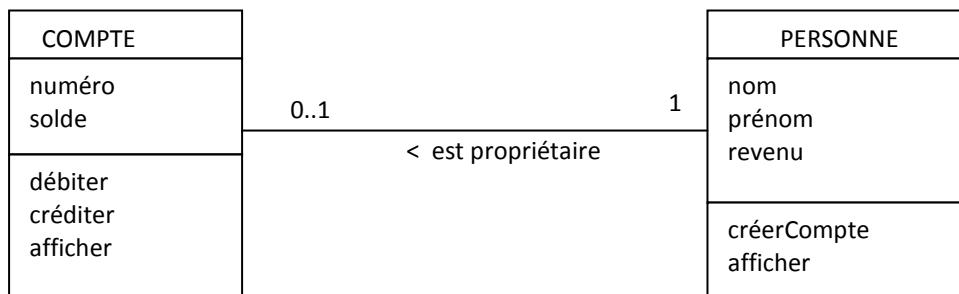
Relation d'association :

Deux classes sont en association lorsque certains de leurs objets ont besoin s'envoyer des messages pour réaliser un traitement.

Une association possède des **valeurs de multiplicité**, qui représentent le nombre d'instances impliquées.

UML (Unified Modeling Language) permet de représenter grâce à son diagramme de classes, les liens entre les classes.

Exemple de représentation en UML



Dans cet exemple, la classe « Compte » est relié à la classe « Personne » par une association nommée « est propriétaire ».

Le sens de lecture de l'association est représentée par le signe « < ». On lit donc « une personne est propriétaire d'un compte » et pas l'inverse.

Les attributs (ou propriétés) de la classe « Compte » sont : « numéro » et « solde ».

Les attributs de la classe « Personne » sont : « nom », « prénom » et « revenu ».

Les méthodes (ou opération s) de la classe « Compte » sont : « débiter », « créditer » et « afficher ».

Les méthodes (ou opération s) de la classe « Personne » sont : « créerCompte » et « afficher ».

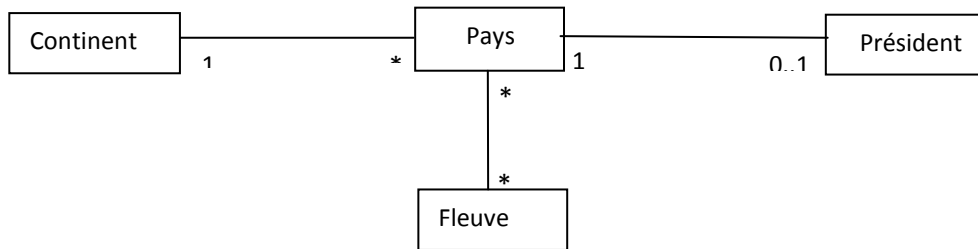
Les valeurs de multiplicité « 0..1 » indiquent qu'une personne a au minimum zéro compte et au maximum 1 compte. C'est à dire « Une personne a un compte ou aucun ».

Les valeurs de multiplicité « 1 » signifient q 'un compte appartient a une personne et une seule. On peut aussi écrire « 1..1 ».

ATTENTION, les valeurs de multiplicité en UML sont inversées par rapport aux cardinalités exprimés dans les diagrammes merisiens.

UML autorise une notation simplifiée des classes

Exemple : (notation simplifiée des classes sans indiquer d'attributs et de méthodes)



Remarque : Les valeurs de multiplicité « 0..* » sont parfois traduites par simplement le symbole « * » et les valeurs de multiplicité « 1..1 » sont parfois traduites par le symbole « 1 »

Ensemble des valeurs de multiplicité possibles en UML :

1..1	Un et un seul
1	Un et un seul
0..1	Zéro ou un
m..n	De m à n
*	De zéro à plusieurs
0..*	De zéro à plusieurs
1..*	De un à plusieurs

Traduction en java :

Les associations entre classes sont tout simplement représentées par des références. Les classes associées possèdent en attribut une ou plusieurs références vers l'autre classe. Le nombre de référence dépend de la cardinalité. Lorsque la cardinalité est « 1 » ou « 0..1 », il n'y a qu'une seule référence de l'autre classe.

Exemple :

```
public class Compte{
    private int numero;
    private float solde;
    private Personne proprietaire; //l'attribut proprietaire est une référence
                                   //sur la classe Personne

    public Compte(int num, Personne propr){ //constructeur
        numero = num;
        proprietaire = propr;
        solde = 0;
    }

    public void crediter(){
        ...
    }
    ... //autres méthodes
}
```

```

public class Personne
{
    private String nom;
    private String prenom;
    private float revenu;
    private Compte cpt;           //l'attribut cpte est une référence à un objet
                                   //Compte

    public Personne(String n, String p, float r){
        nom = n;
        prenom = p;
        revenu = r;
    }

    //autres méthodes
}

```

Remarque : cpt n'est pas initialisé dans le constructeur car une personne peut ne pas avoir de compte

Et si une personne pouvait avoir plusieurs comptes?

Dans la classe personne, on n'aurait plus une seule référence d'un compte, mais un ensemble de références vers des objets comptes :

- Soit sous la forme d'un tableau (si le nombre de compte est fixé dès le départ)
- Soit sous la forme d'une collection (ArrayList), qui est un type de tableau particulier dont la taille peut varier.

```

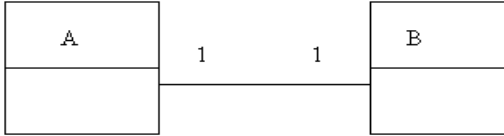
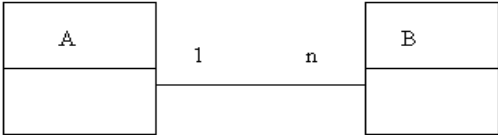
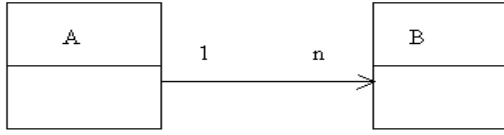
class Personne
{
    private String nom;
    private String prenom;
    private float revenu;
    private ArrayList comptes = new ArrayList();
    ...
}

```

Remarque : La collection de comptes est construite, mais pas les comptes à l'intérieur !

Interprétation des modèles UML

Ci-après, vous trouverez les grandes lignes de passage entre un modèle UML et Java dans ses cas les plus courants.

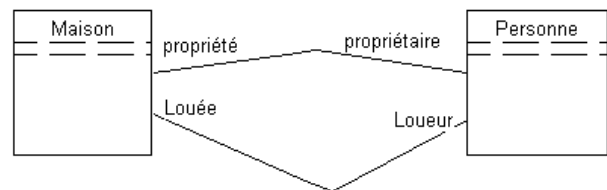
UML	JAVA
<p>Cas d'association un à un.</p> 	<pre>public class A{ private B leB; } public class B{ private A leA; }</pre>
<p>Association de un à n.</p> 	<pre>public class A{ private ArrayList lesB = new ArrayList() ; } public class B{ private A leA; }</pre>
<p>Navigabilité restreinte.</p> 	<pre>public class A{ private ArrayList lesB = new ArrayList() ; } public class B{ // pas de référence à un objet de la classe A }</pre>

Notion de rôle dans un diagramme UML :

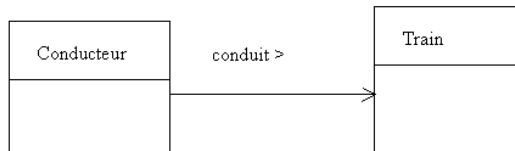
Les classes jouent un rôle dans l'association

Le nom de l'association traduit la nature du lien entre les objets de classes : " La personne est propriétaire d'un compte ".

On peut être amené à préciser le rôle que joue chaque classe dans l'association, notamment lorsqu'un couple de classe est relié par deux associations sémantiquement distinctes:



Navigabilité entre associations.



On peut assimiler une association comme une relation permettant un parcours de l'information. On "passe" d'une classe à l'autre en parcourant les associations; ainsi par défaut les associations sont "navigables" dans les deux sens. Il peut être pertinent au moment de l'analyse de réduire la navigabilité à un seul sens.

Exercice corrigé sur les relations entre classes

A partir des classes représentées au format UML, écrire le code en java.

Etudiant	
- nom	: String
- prenom	: String
+ <<Constructor>>	Etudiant (String unNom, String unPrenom)
+	toString () : String

Crayon	
- type	: char
- couleur	: String
+ <<Constructor>>	Crayon (char leType, String laCouleur)
+	getType () : char
+	getCouleur () : String

```
public class Crayon {
    private char type;
    private String couleur;

    public Crayon (char leType , String laCouleur) {
        this.type = leType;
        this.couleur = laCouleur;
    }

    public char getType() {
        return this.type;
    }
    public String toString() {
        return "Type:"+this.type+" Couleur:"+this.couleur;
    }
}
```

```
public class Etudiant {
    private String nom;
    private String prenom;

    // Constructeur
    public Etudiant (String unNom , String unPrenom){
        this.nom = unNom;
        this.prenom = unPrenom;
    }

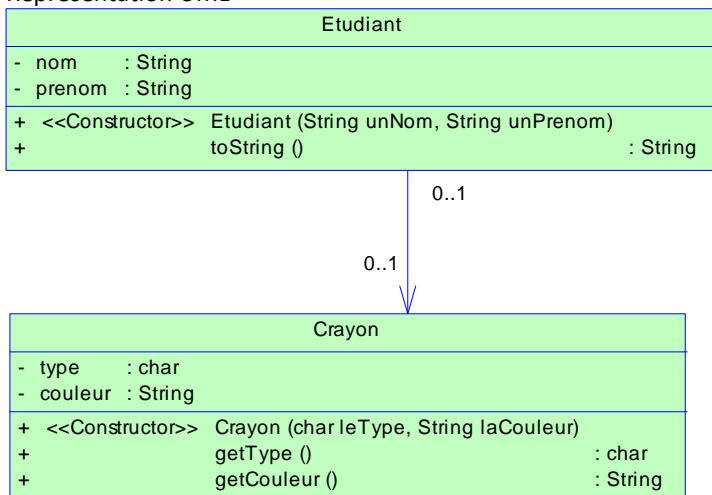
    // toString
    public String toString(){
        return this.prenom+" "+this.nom;
    }
}
```

```
class Test{
    public static void main(String args[]){
        Crayon c1 = new Crayon('F' , "Bleu");
        System.out.println(c1.toString());
        Etudiant e1 = new Etudiant("Friedman", "Milton");
        System.out.println(e1.toString());
    }
}
```

Dans l'exemple ci-dessus, les 2 classes sont indépendantes. Il existe des crayons ; il existe des étudiants mais les étudiants sont indépendants des crayons.

Nous allons faire évoluer notre cas pour dire qu'un étudiant possède un crayon

Représentation UML



Comment faire en Java ?

```
public class Etudiant {
    private String nom;
    private String prenom;
    private Crayon leCrayon;
    .../...
```

Comment se servir de cet attribut supplémentaire ?

Lorsque l'étudiant est créé, il ne possède aucun crayon.

Comment donner un crayon à l'étudiant (il faut que le crayon existe) ?

Il faut ajouter une méthode qui valorise (donne une valeur) la propriété privée `leCrayon` de l'étudiant. Ce sera donc une méthode de la classe `Etudiant`. Appelons là « `ajouteCrayon` »

```
// Ajoute à l'étudiant e1 le crayon c1
e1.ajouteCrayon(c1);
```

```
// Méthode qui ajoute un crayon à un étudiant
public void ajouteCrayon(Crayon unCrayon){
    this.leCrayon = unCrayon;
}
```

Si l'étudiant perd son crayon, il ne possède plus de crayon. Il faut donc affecter la valeur NULL à la propriété « leCrayon » de l'étudiant.

Il faut ajouter une méthode « perdCrayon » dans la classe « Etudiant ».

```
// L'étudiant e1 perd son crayon
e1.perdCrayon();
System.out.println(e1.toString());
```

```
// Méthode qui retire le crayon de l'étudiant
public void perdCrayon(){
    this.leCrayon = null;
}
```

Pour un affichage plus cohérent, on peut modifier la méthode permettant d'afficher un étudiant.

```
// toString
public String toString(){
    String message=this.prenom+" "+this.nom;
    if (leCrayon !=null)
        message=message+" possède le crayon "+leCrayon.toString();
    else
        message+=" ne possède pas de crayon";
    return message;
}
```

Un étudiant a en général plusieurs crayons

Nous allons donc attribuer une collections de crayons dans la classe « Etudiant »

```
class Etudiant {  
    private String nom;  
    private String prenom;  
    private ArrayList trousse;
```

Il faut donc modifier le constructeur « Etudiant »

```
// Constructeur  
public Etudiant (String unNom , String unPrenom){  
    this.nom = unNom;  
    this.prenom = unPrenom;  
    this.trousse=new ArrayList();  
}
```

Il faut aussi modifier les méthodes « ajouteCrayon » et « perdCrayon »

```
// Méthode qui ajoute un crayon à un étudiant  
public void ajouteCrayon(Crayon unCrayon){  
    this.trousse.add(unCrayon);  
}  
  
// Méthode qui retire le crayon de l'étudiant  
public void perdCrayon(Crayon unCrayon){  
    this.trousse.remove(unCrayon);  
}
```

La méthode « toString » est adaptée à cette nouvelle sorte d'étudiant. Une méthode privée qui nous donne le nombre de crayons de l'étudiant a aussi été ajoutée.

```
// Méthode qui compte le nombre de crayons  
private int nbCrayons(){  
    return this.trousse.size();  
}  
  
// toString  
public String toString(){  
    String message=this.prenom+" "+this.nom+" possède "  
    message+=nbCrayons()+" crayons."  
    for (int i=0; i<nbCrayons();i++){  
        message+="\n\t"+((Crayon)(trousse.get(i))).toString();  
    }  
    return message;  
}
```

Et voici un exemple de programme d'affichage :

```
public static void main(String args[]){  
    Crayon c1 = new Crayon('F' , "Bleu");  
    System.out.println(c1.toString());  
    Etudiant e1 = new Etudiant("Friedman", "Milton");  
    System.out.println(e1.toString());  
    // Ajoute à l'étudiant e1 le crayon c1  
    e1.ajouteCrayon(c1);  
    System.out.println(e1.toString());  
    // Second crayon  
    e1.ajouteCrayon(new Crayon('B' ,"Rouge"));  
    System.out.println(e1.toString());  
    // Il perd le premier crayon  
    e1.perdCrayon(c1);  
    System.out.println(e1.toString());  
}
```