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 ».

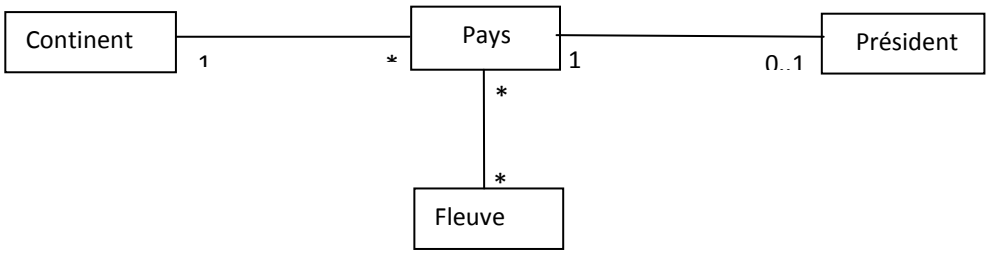
Les méthodes (ou opération s) de la classe « Compte » sont : « débiter », « créditer » et « afficher ».
Les méthodes (ou opération s) de la classe « Personne » sont : « créerCompte » et « afficher ».

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.

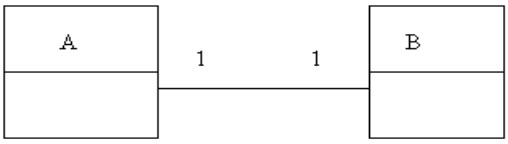
```
class Personne
{
	private String nom;
	private String prenom;
	private float revenu;
	private ArrayList comptes = new ArrayList();
	…
}
```

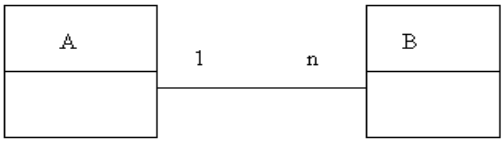
Remarque : La collection de comptes est construite, mais pas les comptes à l'intérieur !

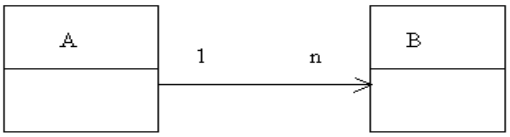
Interprétation des modèles UML

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

UML	JAVA

Cas d'association un à un.
 
 	
	public class A{
	private B leB;
	}
	
	public class B{
	private A leA;
	}

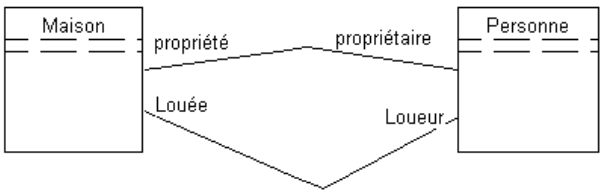
Association de un à n.
 
 	
	public class A{
	private ArrayList lesB = new ArrayList() ;
	}
	
	public class B{
	private A leA;
	}

Navigabilité restreinte.
 
 	
	public class A{
	private ArrayList lesB = new ArrayList() ;
	}
	
	public class B{
	// pas de référence à un objet de la classe A
	}

Notion de rôle dans un diagramme UML :

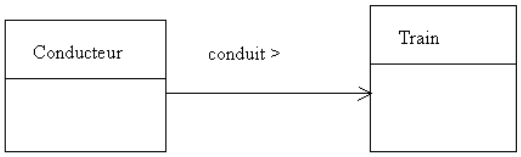
Les classes jouent un rôle dans l'association

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:

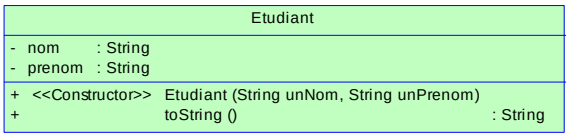
Navigabilité entre associations.

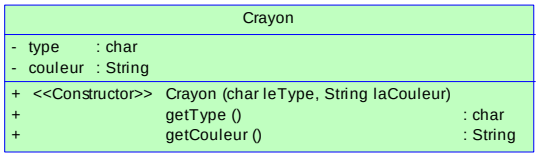


On peut assimiler une association comme une relation permettant un parcours de l'information. On "passe" d'une classe à l'autre en parcourant les associations; ainsi par défaut les associations sont "navigables" dans les deux sens. Il peut être pertinent au moment de l'analyse de réduire la navigabilité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

Exercice corrigé sur les relations entre classes

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






```
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;
}

}
```

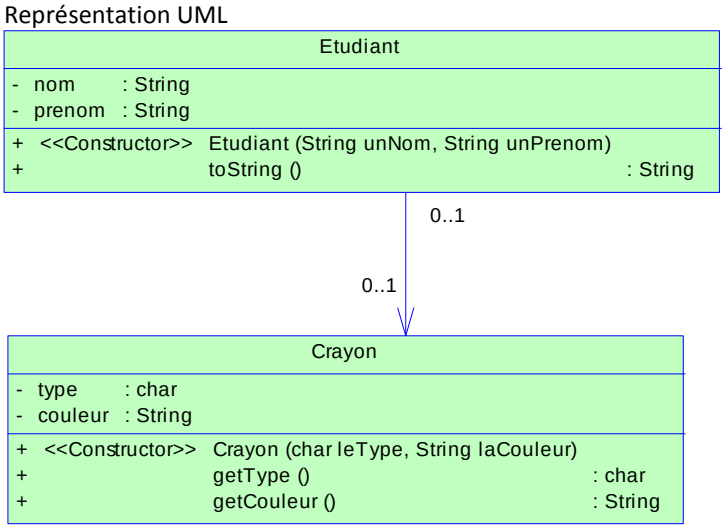
```
public class Etudiant {
   private String nom;
   private String prenom;

// 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());
              }
```