

# Première implémentation des modèles UML en langages Java et C#

## Table des matières

1	Introduction .....	2
2	Les bases sur le modèle statique du logiciel .....	2
2.1	Implémentation d'une Classe.....	2
2.1.1	Règles de passage au code.....	2
2.2	Implémentation d'une Classe abstraite .....	3
2.3	Implémentation des Attributs .....	4
2.4	Implémentation des Opérations.....	6
2.5	Implémentation d'une Interface .....	9
2.6	Implémentation d'un Package.....	10
3	Implémentation des relations entre concepts statiques.....	11
3.1	Implémentation d'un héritage .....	11
3.2	Implémentation d'un héritage entre interfaces .....	11
3.3	Implémentation d'une Dépendance .....	12
3.4	Implémentation d'une Réalisation .....	14
3.5	Exemples de relations possibles avec une interface .....	15
4	Implémentation d'une association binaire .....	16
4.1	Règles générales .....	16
4.2	Règle pour le rôle navigable avec une multiplicité 1 .....	16
4.3	Règle pour le rôle navigable avec une multiplicité N .....	16
4.4	Association navigable dans un seul sens.....	17
4.5	Exemple d'une association unidirectionnelle avec une multiplicité 1 .....	17
4.5.1	Tableau de synthèse des associations binaires unidirectionnelles .....	19
4.6	Association navigable dans les deux sens .....	20
4.6.1	Association bidirectionnelle de 1 vers 1.....	20
4.7	Association réflexive bidirectionnelle de 1 vers N .....	21
5	Agrégation et composition .....	22
5.1	Première possibilité : Instanciation dans le constructeur .....	22
5.2	Deuxième possibilité : classe imbriquées.....	23
6	Classe d'association .....	24
7	Les diagrammes dynamiques UML .....	25
7.1	Diagrammes d'interactions .....	25
7.2	Diagrammes de séquence .....	26
8	Conclusion.....	27
9	TP :Etude de Cas: deux DCC sur la Gestion de Formations (Roques).....	28
10	Diagrammes de communication A_Revoir Bibliothèque .....	28
10.1	Obsolète: Exemple de passage du diagramme de classes au code C# .....	30

## 1 Introduction

Cet documentation vise à jeter un pont entre les concepts de la modélisation UML et le monde de l'implémentation dans le langage orienté objet.

UML est un langage de modélisation visuelle, Java et C# sont des langages de programmation textuels. UML est plus riche que les langages de programmation dans le sens où il offre des moyens d'expression plus abstraits et plus puissants. Cependant, il existe généralement une façon privilégiée de traduire les concepts UML en déclarations Java ou C#.

## 2 Les bases sur le modèle statique du logiciel

Les concepts structuraux (ou statiques) tels que classe et interface sont évidemment fondamentaux aussi bien en UML qu'en Java ou C#. Ils sont représentés en UML dans les Diagrammes de Classes de Conception (DCC), et constituent le squelette d'un code orienté-objet, à savoir toutes les déclarations.

Nous aborderons tour à tour les concepts UML de classe, d'attribut, d'opération, d'interface, de package.

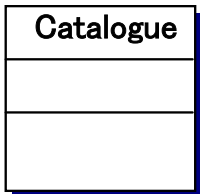
### 2.1 Implémentation d'une Classe

La classe est le concept fondamental de toute technologie objet. Le mot-clé correspondant existe aussi bien en Java qu'en C#. De plus, chaque classe UML devient par défaut un fichier .java ou fichier .cs.

#### 2.1.1 Règles de passage au code

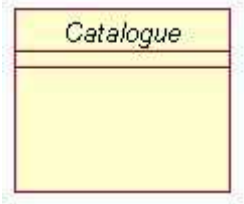
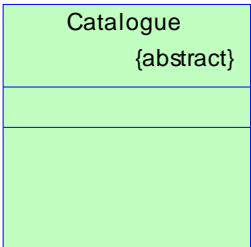
En première approche, on peut appliquer les règles suivantes:

- La classe UML devient une classe Java,
- les attributs UML d'une classe deviennent des variables d'instances Java,
- les opérations UML d'une classe deviennent des méthodes Java.

Les opérations UML d'une classe deviennent des méthodes Java.		
UML	C#	
	<pre>public <b>class</b> Catalogue {     ... }</pre>	
	<table><tr><th>Java</th></tr><tr><td><pre>public <b>class</b> Catalogue {     ... }</pre></td></tr></table>	Java
Java		
<pre>public <b>class</b> Catalogue {     ... }</pre>		

## 2.2 Implémentation d'une Classe abstraite

Une classe abstraite est simplement une classe qui ne s'instancie pas directement mais qui représente une pure abstraction afin de factoriser des propriétés. Elle se note en *italiques* en UML et se traduit par le mot-clé `abstract` en Java ou en C#.

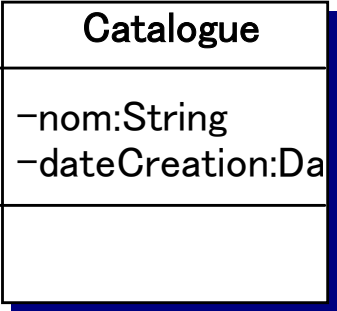
UML	C#
<div>  </div> <p>ou</p> <div>  </div>	<pre>abstract public <b>class</b> Catalogue {     ... }</pre>
	<div>Java</div> <pre>abstract public <b>class</b> Catalogue {     ... }</pre>

## 2.3 Implémentation des Attributs

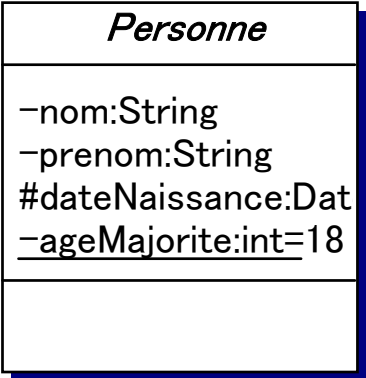
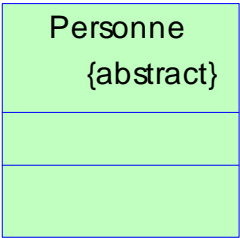
Les attributs UML deviennent simplement des variables (ou membres) en Java et C#. Leur type est soit un type primitif (int, etc.), soit une classe fournie par la plateforme (String, Date) Java ou (string, DateTime, etc.) .NET. Attention à ne pas oublier dans ce cas la directive d'importation du package correspondant. La visibilité des attributs est montrée graphiquement en UML en les faisant précéder par + pour public, # pour protégé (protected), - pour privé (private).

Les attributs de classe en UML deviennent des membres statiques en Java ou en C# (static).

Attention : les attributs de type référence à un autre objet ou à une collection d'objets sont abordés de façon détaillée dans le paragraphe sur les "associations".

UML	C#
	<pre>using System;  public class Catalogue {     private string nom;     private DateTime dateCreation;     ... }</pre>
	<p style="text-align: center;"><b>Java</b></p> <pre>import java.util.Date;  public class Catalogue {     private String nom;     private Date dateCreation;     ... }</pre>

Autre exemple:

UML	C#
	<pre>abstract public class Personne {     private string nom;     private string prenom;     protected DateTime dateNaissance;     private static int ageMajorite = 18; }</pre>
<p>ou</p> 	<div>Java</div> <pre>abstract public class Personne {     private String nom;     private String prenom;     protected Date dateNaissance;     private static int ageMajorite = 18; }</pre>

Voir aussi les attributs dérivés.

## 2.4 Implémentation des Opérations

Les opérations UML deviennent très directement des méthodes en Java et en C#.

Leur visibilité est définie graphiquement avec les mêmes conventions que les attributs (+,-,#).

Les opérations de classe deviennent des méthodes statiques (`static`) ; les opérations abstraites (en *italiques*) se traduisent par le mot-clé `abstract` en Java ou en C#.

UML	C#
<div> <div>Catalogue</div> <div> - nom:String  - dateCreation:Date  + chercherLivre(isbn:ISBN):Li </div> </div>	<pre> public class Catalogue {     private string nom;     private DateTime dateCreation;     <b>public Livre ChercherLivre(ISBN isbn)</b>     {         ...     }     ... } </pre>
	<div>Java</div> <pre> public class Catalogue {     private String nom;     private Date dateCreation;     <b>public Livre chercherLivre(ISBN isbn)</b>     {         ...     } } </pre>

UML	C#
<div> <div> <b>Personne</b> </div> <div> -nom:String  -prenom:String  #dateNaissance:Date  -<u>ageMajorite:int=18</u> </div> <div> +calculerDureePret():int  +setAgeMajorite(a:int)  +getAge():int </div> </div>	<pre> abstract public class Personne {     private string nom;     private string prenom;     protected DateTime dateNaissance;     private static int ageMajorite = 18;     public abstract int CalculerDureePret();     public static void SetAgeMajorite(int aMaj) {         ...     }     public int GetAge() {         ...     } } </pre>
	<div>Java</div> <pre> abstract public class Personne {     private String nom;     private String prenom;     protected Date dateNaissance;     private static int ageMajorite = 18;     public abstract int calculerDureePret();     public static void setAgeMajorite(int aMaj) {         ...     }     public int getAge() {         ...     } } </pre>

Nota : avec l'utilisation des « property » C# au lieu des accesseurs à la Java, on obtiendrait plutôt le code suivant :

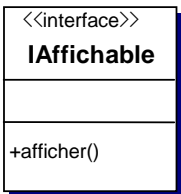

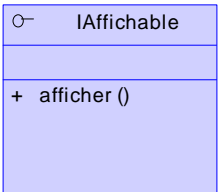
UML	C#
<div> <div><b>Personne</b></div> <div> -nom:String  -prenom:String  #dateNaissance:Date  -<u>ageMajorite:int=18</u> </div> <div> +calculerDureePret():int  +setAgeMajorite(a:int)  +getAge():int </div> </div>	<pre> abstract public class Personne {     private string nom;     private string prenom;     protected DateTime dateNaissance;     private static int ageMajorite = 18;     public abstract int CalculerDureePret();     public static void SetAgeMajorite(int aMaj) {         ...     }     public int Age{         get{return System.DateTime.Now - dateNaissance;}     } } </pre>



## 2.5 Implémentation d'une Interface

La notion UML d'interface, popularisée initialement par Java (et CORBA !) peut être représentée de plusieurs façons graphiques différentes en UML. Elle se traduit par le mot-clé "interface" aussi bien en Java qu'en C#.

Pour les notions d'interface fournie (**réalisation**) et d'interface requise (**dépendance**), voir le chapitre sur les relations entre concepts statiques.

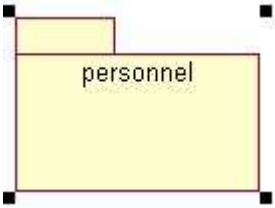
UML	C#
 <p>ou</p>  <p>ou</p> 	<pre>interface IAffichable {     void Afficher(); }</pre>
	Java
	<pre>interface IAffichable {     void afficher(); }</pre>

Nota : nous avons essayé de respecter au maximum les conventions de nommage de chaque langage. C'est pourquoi vous verrez par exemple les opérations UML commencer par une minuscule (comme en Java), alors que les méthodes C# comment plutôt par une majuscule.

## 2.6 Implémentation d'un Package

Le package est le mécanisme général de regroupement d'éléments de modélisation en UML. Le package en tant que regroupement de classes ou d'interfaces existe aussi bien en Java qu'en C#, avec les mots clé suivants:

"package" en Java  
"namespace" en C#

UML	C#
	<code>namespace Personnel</code> <code>{</code> <code>...</code> <code>}</code>
	Java
	<code>package personnel;</code> <code>...</code>

Attention, par contre, la notion UML de sous-système (package stéréotypé « subsystem » pouvant posséder des interfaces et une dynamique) n'a pas d'équivalent en C#.

### 3 Implémentation des relations entre concepts statiques

(sauf association)

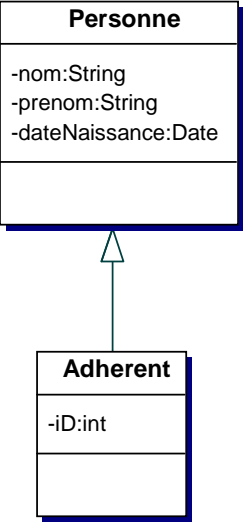
Les relations UML entre concepts statiques sont très riches. On peut distinguer les relations de :

- Généralisation (héritage)
- Réalisation
- Dépendance
- Association, avec ses variantes : agrégation et composition.

Elles ne se traduisent pas toutes de façon simple par un mot-clé dans le langage objets Java ou C# .

#### 3.1 Implémentation d'un héritage

Le concept UML de généralisation se traduit directement par le mécanisme de l'héritage dans les langages objets. C# utilise la syntaxe du C++ pour l'héritage, mais interdit l'héritage multiple entre classes . Java utilise le mot réservé "extends" et interdit aussi l'héritage multiple.

UML	C#	
 <pre> classDiagram     class Personne {         -nom:String         -prenom:String         -dateNaissance:Date     }     class Adherent {         -iD:int     }     Personne &lt; -- Adherent         </pre>	<pre> public class Personne {     ... }  public class Adherent : Personne {     private int iD; }         </pre>	
	<th>Java</th>	Java
	<pre> public class Personne {     ... }  public class Adherent extends Personne {     private int iD; }         </pre>	

#### 3.2 Implémentation d'un héritage entre interfaces

L'héritage multiple pour les interfaces est accepté en UML, Java et C#.

### 3.3 Implémentation d'une Dépendance

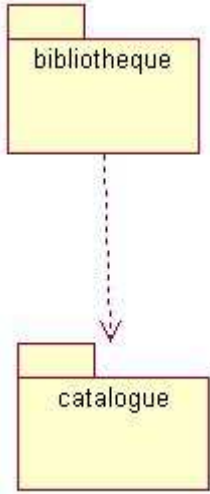
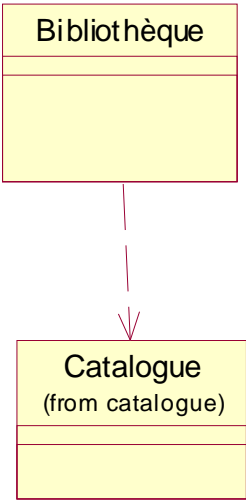
La dépendance est un concept très général en UML et peut concerner de nombreux types d'éléments de modélisation (cas d'utilisation, classes, interfaces, packages, composants...).

Une dépendance entre une classe A et une classe B existe par exemple dès que A possède une méthode prenant comme paramètre une référence sur une instance de B, ou si A utilise une opération de classe de B. Il n'existe pas de mot-clé correspondant en Java ou en C#.

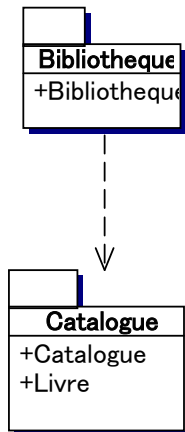
La dépendance entre packages se traduit de façon indirecte par des directives d'importation en Java (`import`) ou des directives d'usage en C# (`using`).

Le nom des packages est en minuscule en UML et en Java.

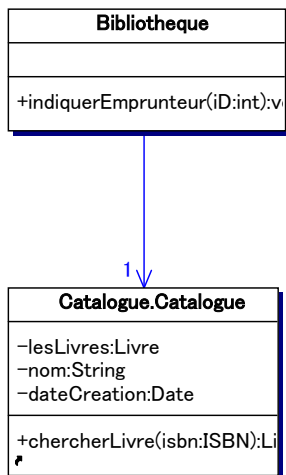
Le nom des namespace en C# commence par une majuscule

UML	C#
	<p><b>Dépendance entre package :</b></p> <p>-----</p> <pre>namespace Bibliotheque {     using Catalogue;     public class Bibliotheque {         private Catalogue leCatalogue;         ...     } }</pre>
	<p><b>Java</b></p> <p><b>Dépendance entre classes :</b></p> <p>-----</p> <pre>package bibliotheque;  import catalogue.*;  public class Bibliotheque {     private Catalogue leCatalogue;     ... }</pre>

Autre formalisme:



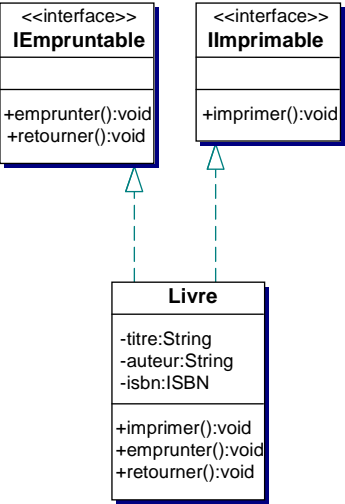
Formalisme accepté dans le diagramme de classes: :



### 3.4 Implémentation d'une Réalisation

Une classe UML peut implémenter plusieurs interfaces.

Contrairement à C++, les langages Java et C# propose directement ce mécanisme (mais pas l'héritage multiple entre classe).

UML	C#
 <pre> classDiagram     class IEmpruntable {         &lt;&lt;interface&gt;&gt;         +emprunter():void         +retourner():void     }     class IImprimable {         &lt;&lt;interface&gt;&gt;         +imprimer():void     }     class Livre {         -titre:String         -auteur:String         -isbn:ISBN         +imprimer():void         +emprunter():void         +retourner():void     }     IEmpruntable .. &gt; Livre     IImprimable .. &gt; Livre         </pre>	<pre> public class Livre : IImprimable, IEmpruntable {     private string titre;     private string auteur;     private ISBN isbn;     public void Imprimer(){         ...     }     public void Emprunter(){         ...     }     public void Retourner(){         ...     } }         </pre>
	<div data-bbox="963 1218 1031 1249" data-label="Section-Header"> <h4>Java</h4> </div> <pre> public class Livre implements IImprimable, IEmpruntable {     private String titre;     private String auteur;     private ISBN isbn;     public void imprimer(){         ...     }     public void emprunter(){         ...     }     public void retourner(){         ...     } }         </pre>

### 3.5 Exemples de relations possibles avec une interface

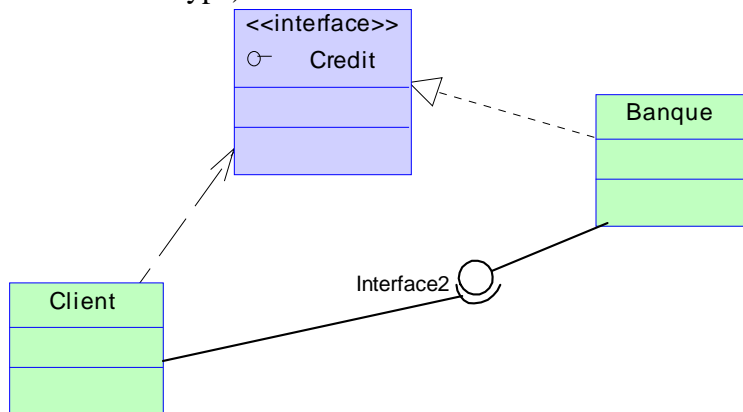
Les exemples suivants illustrent les relations de réalisation (interface fournie) et de dépendance avec une interface (interface requise).

La classe Banque fournit effectivement les services décrits dans l'interface: La classe Banque "réalise" l'interface Crédit.

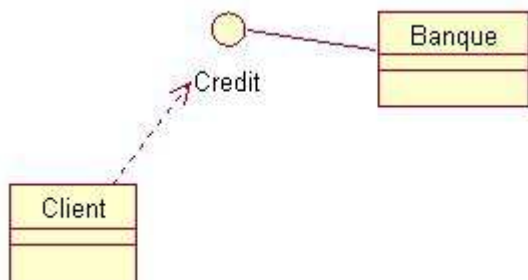
Client utilise tout ou partie des services décrits dans l'interface : la classe Client dépend de l'interface Crédit, la classe Client requiert l'interface Crédit.

#### Représentation avec les deux notations possibles:

- L'interface est représentée de la même manière qu'une classe (mais avec un stéréotype).



- L'interface est représentée sous formalisme Lollipop (alias socket) (**depuis UML 1.X avec Rose2002**)



## 4 Implémentation d'une association binaire

On s'intéresse au code Java ou C# correspondant à des associations navigables UML

### 4.1 Règles générales

(Règles générales de passage au code)

Les associations, agrégations ou compositions s'implémentent en ajoutant des attributs dans les classes du code.

De plus, ce code dépend de la multiplicité de l'extrémité (rôle) concernée , mais aussi de l'existence ou pas d'une contrainte {ordered} ou d'un qualificatif.

### 4.2 Règle pour le rôle navigable avec une multiplicité 1

(Une multiplicité 1 côté navigable)

Une association navigable avec une multiplicité 1 (maximun) se traduit par l'ajout d'une variable d'instance (nom du rôle) qui référence une instance d'une classe du modèle.

Rappel : en général, les attributs classiques ont un type primitif.

Autrement formulé : les rôles navigables produisent des variables d'instances, tout comme les attributs, mais avec un type utilisateur au lieu d'un type simple.

### 4.3 Règle pour le rôle navigable avec une multiplicité N

Une multiplicité « \* » va se traduire par un attribut de type collection de références d'objets au lieu d'une simple référence sur un objet. La difficulté consiste à choisir la bonne collection parmi les très nombreuses classes de base que propose Java et C#. Bien qu'il soit possible de créer des tableaux d'objets, ce n'est pas forcément la bonne solution. En la matière, on préfère plutôt recourir à des collections, parmi lesquelles les plus utilisées sont :

- En Java : *ArrayList* (anciennement *Vector* ) et *HashMap* (anciennement *HashTable*). Utilisez *ArrayList* si vous devez respecter un ordre et récupérer les objet à partir d'un indice entier; utilisez *HashMap* si vous souhaitez récupérer les objets à partir d'une clé arbitraire. Attention, le JDK 1.5 introduit les collections typées appelées « generics ».
- En C#: *ArrayList*, *SortedList* et *HashTable*. Utilisez *ArrayList* si vous devez respecter un ordre et récupérer les objets à partir d'un indice entier ; utilisez *HashTable* ou *SortedList* si vous souhaitez récupérer les objets à partir d'une clé arbitraire.

Dans la suite du document, nous prenons un exemple simple pour illustrer les règles de passages.



#### 4.4 Association navigable dans un seul sens

(Association à navigabilité restreinte)

(Association navigable unidirectionnelle)

On distingue les cas suivants:

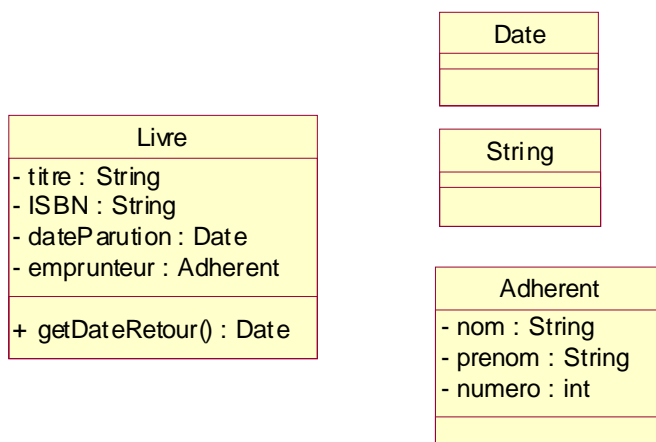
- association unidirectionnelle de 1 vers 1,
- association unidirectionnelle de 1 vers plusieurs (1, N).

#### 4.5 Exemple d'une association unidirectionnelle avec une multiplicité 1

La multiplicité maximum est 1.

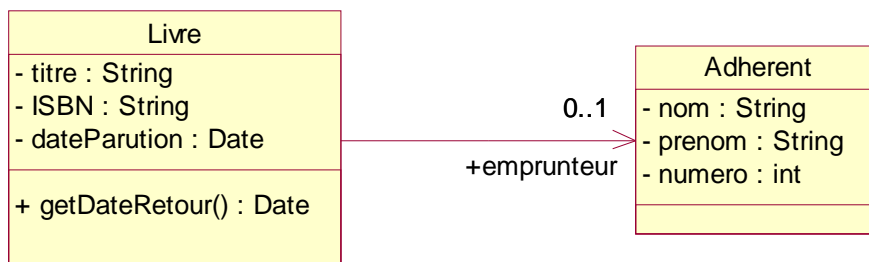
Voici une première représentation du DCC qui doit être traduit en langage de programmation.

Notez que le Livre a un attribut qui est une référence vers un objet Adherent.

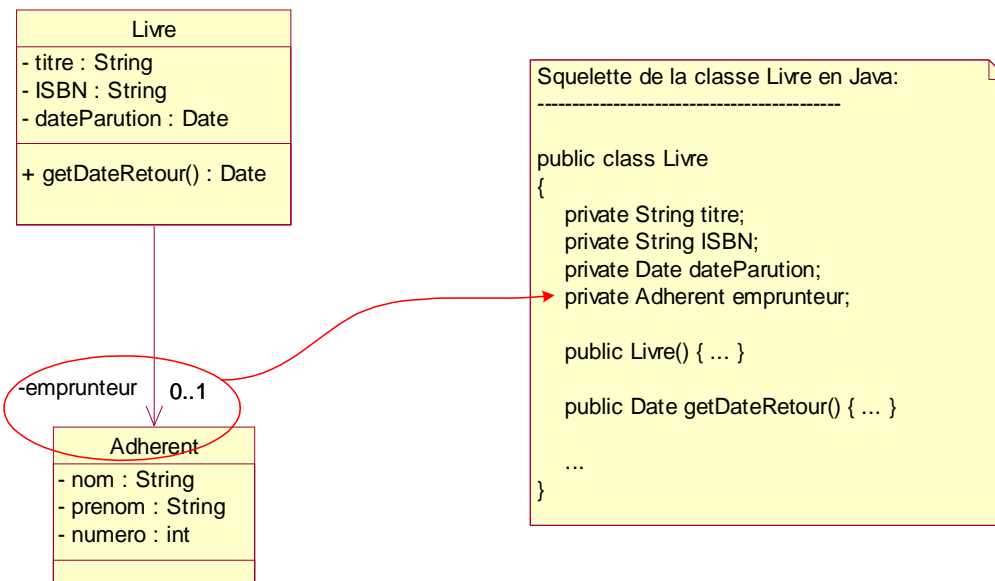


Rappel : Les attributs de type référence vers un (ou des) objets sont équivalents à des associations vers la classe correspondantes.

Voici une représentation équivalente du DCC avec une ligne d'association qui indique que le Livre a un attribut référençant un objet Adherent.



Nous donnons ici le squelette du code Java de la classe Livre:



Nous donnons ici le squelette du code de la classe Livre en Java et en C# .

Java	C#
<pre> public class Livre {     private String titre;     private String ISBN;     private Date dateParution;     private Adherent emprunteur;      public Livre() { ... }      public Date getDateRetour() { ... }      ... }                 </pre>	<pre> public class Livre {     private string titre;     private string ISBN;     private DateTime dateParution;     private Adherent emprunteur;      public Livre() { ... }      public DateTime getDateRetour() { ... }      ... }                 </pre>

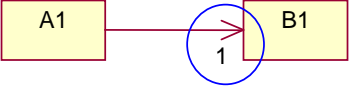
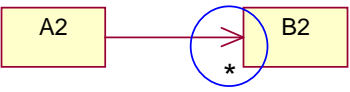
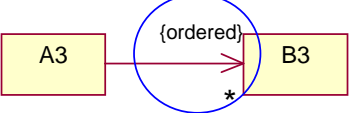
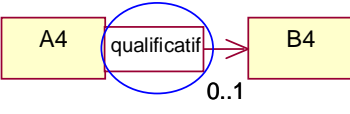
Nota: Comme l'association est navigable dans un seul sens, dans la classe Adherent, il n'y a pas d'attribut qui référence un objet Livre.

#### 4.5.1 Tableau de synthèse des associations binaires unidirectionnelles

Toutes les règles de passage au code sont importantes à comprendre si vous devez générer du code à partir de diagramme UML.

Le tableau de synthèse ci-après concerne les associations binaires unidirectionnelles et considère la multiplicité de l'extrémité, ainsi que l'existence d'un qualificatif ou d'une contrainte {ordered}.

Les règles de passage au code sont résumées sur le schéma synthétique suivant.

UML	Java	C#
	<pre>public class A1 {     private B1 leB1;     ... }</pre>	<pre>public class A1 {     private B1 leB1;     ... }</pre>
	<pre>public class A2 {     private B2 [] lesB2;     ... }</pre>	<pre>public class A2 {     private B2[] lesB2;     ... }</pre>
	<pre>public class A3 {     private List&lt;B3&gt; lesB3 = new ArrayList&lt;B3&gt;();     ... }</pre>	<pre>public class A3 {     private ArrayList lesB3 = new ArrayList();     ... }</pre>
	<pre>public class A4 {     private Map&lt;Q,B4 &gt; lesB4 = new HashMap&lt;Q,B4 &gt; ();     ... }</pre>	<pre>public class A4 {     private Hashtable lesB4 = new Hashtable ();     ... }</pre>

## 4.6 Association navigable dans les deux sens

(Association bidirectionnelle)

On distingue les cas suivants:

- association bidirectionnelle de 1 vers 1,
- association bidirectionnelle de 1 vers plusieurs (1 , N).
- association bidirectionnelle de plusieurs vers plusieurs (N , N).

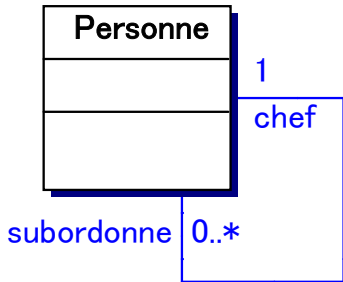
### 4.6.1 Association bidirectionnelle de 1 vers 1

Une association bidirectionnelle se traduit simplement par une paire de références, une dans chaque classe impliquée dans l'association. Les noms des **rôles** aux extrémités d'une association servent à nommer les variables de type référence.

UML	C#	
<pre> classDiagram     class Homme     class Femme     Homme "0..1" -- "0..1" Femme : mari, epouse         </pre>	<pre> public class Homme {     private Femme <b>epouse</b>;     ... }  public class Femme {     private Homme <b>mari</b>;     ... }         </pre>	
	<th>Java</th>	Java
	<pre> public class Homme {     private Femme <b>epouse</b>;     ... }  public class Femme {     private Homme <b>mari</b>;     ... }         </pre>	

## 4.7 Association réflexive bidirectionnelle de 1 vers N

Une association réflexive se traduit simplement par une référence sur un objet de la même classe.

UML	C#
 <pre> classDiagram     class Personne {     }     Personne "1" -- "0..*" Personne : chef / subordonne         </pre>	<pre> public class Personne {     private <b>Personne</b>[] subordonne;     private <b>Personne</b> chef ;     ... }         </pre>
	<p style="text-align: center;"><b>Java</b></p> <pre> public class Personne {     private <b>Personne</b>[] subordonne;     private <b>Personne</b> chef ;     ... }         </pre>

## 5 Agrégation et composition

L'agrégation est un cas particulier d'association non symétrique exprimant une relation de contenance. **Les agrégations n'ont pas besoin d'être nommées : implicitement elles signifient « contient », « est composé de ».** La sémantique des agrégations n'est pas fondamentalement différente de celle des associations simples, elles se traduisent donc comme indiqué précédemment en Java ou en C#.

Une composition est une agrégation plus forte impliquant que :

- une partie ne peut appartenir qu'à un seul composite (agrégation non partagée);
- la destruction du composite entraîne la destruction de toutes ses parties (le composite est responsable du **cycle de vie** des parties).

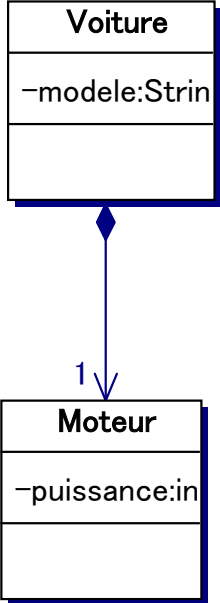
Dans certains langages objet, par exemple C++, la composition implique donc la propagation du destructeur, mais cela ne s'applique pas à Java ni à C#.

### 5.1 Première possibilité : Instanciation dans le constructeur

Pour traduire la composition, on peut créer un objet dans le constructeur de l'autre. S'il y a destruction de ce dernier, il y aura perte de référence du deuxième objet qui sera supprimé à son tour.

## 5.2 Deuxième possibilité : classe imbriquées

La notion de classe imbriquée peut s'avérer intéressante pour traduire la composition. Elle n'est cependant pas du tout obligatoire.

UML	C#	
 <pre> classDiagram     class Voiture {         -modele:Strin     }     class Moteur {         -puissance:in     }     Voiture "1" *-- "1" Moteur         </pre>	<pre> public class Voiture {     private string modele;     private Moteur moteur;     <b>private class Moteur</b> {         private int puissance;     }     ... }         </pre>	
	<th>Java</th>	Java
	<pre> public class Voiture {     private String modele;     private Moteur moteur;     <b>private static class Moteur</b> {         private int puissance;     }     ... }         </pre>	

## 6 Classe d'association

Il s'agit d'une association promue au rang de classe. Elle possède tout à la fois les caractéristiques d'une association et d'une classe et peut donc porter des attributs qui se valorisent pour chaque lien. Ce concept UML avancé n'existe pas dans les langages de programmation objet, il faut donc le traduire en le transformant en classe normale, et en ajoutant des variables de type référence.

UML	C#
<pre> classDiagram     class Personne     class Emploi {         - titre : String         - salaire : Double     }     class Societe     Personne "0..*" -- "0..*" Societe : employe     Societe "0..*" -- "0..*" Emploi : employeur </pre>	<pre> public class Emploi {     private string titre;     private double salaire;     private <b>Personne</b> employe;     private <b>Societe</b> employeur ;     ... } </pre>
	Java
	<pre> public class Emploi {     private String titre;     private double salaire;     private <b>Personne</b> employe;     private <b>Societe</b> employeur ;     ... } </pre>



## **7 Les diagrammes dynamiques UML**

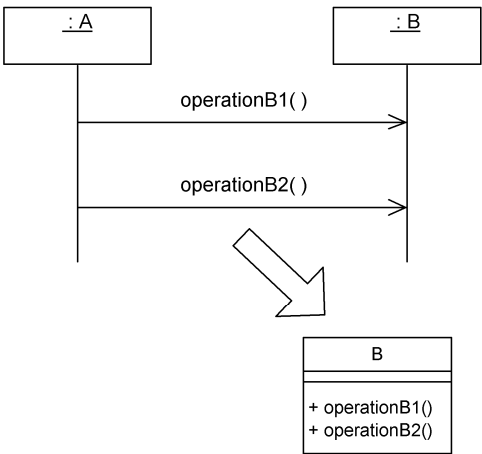
Contrairement à ce qui est généralement admis (en grande partie à cause des lacunes de la plupart des outils de modélisation UML), les diagrammes dynamiques UML peuvent se traduire également en code, même si c'est d'une manière moins directe que les diagrammes de classes. Il est donc intéressant de le savoir, pour comprendre l'avantage de choisir certains outils plus avancés, ou simplement pour effectuer efficacement le travail à la main ...

### **7.1 Diagrammes d'interactions**

Pour l'analyse et la conception, les diagrammes d'interactions UML (séquence ou communication) sont particulièrement utiles pour représenter graphiquement ses décisions d'allocation de responsabilités. Chaque diagramme va ainsi représenter un ensemble d'objets de classes différentes collaborant dans le cadre d'un scénario d'exécution du système. Dans ce genre de diagramme, les objets communiquent en s'envoyant des messages qui invoquent des méthodes sur les objets récepteurs. Il est ainsi possible de suivre visuellement les interactions dynamiques entre objets, et les traitements réalisés par chacun.

Pour l'implémentation, les séquences d'appels de méthodes des **diagrammes d'interactions** permettent d'écrire le corps de certaines méthodes.

## 7.2 Diagrammes de séquence

UML	C#	
 <pre> sequenceDiagram     participant A as :A     participant B as :B     A-&gt;&gt;B: operationB1()     A-&gt;&gt;B: operationB2()     </pre> <pre> classDiagram     class B {         + operationB1()         + operationB2()     }     </pre>	<pre> public class A {     ...     leB.OperationB1();      leB.OperationB2();     ... }  public class B {     ...     public void OperationB1();     public void OperationB2();     ... }     </pre>	
	<th>Java</th>	Java
	<pre> public class A {     ...     leB.operationB1();     ...     leB.operationB2();     ... }  public class B {     ...     public void operationB1() {         ...     }     ...     public void operationB2() {         ...     }     ... }     </pre>	

## **8 Conclusion**

La génération de code à partir des différents diagrammes est une opération délicate qu'il faut maîtriser. Ainsi, en fonction du langage cible utilisé (Java ou C#), le mapping entre les différents types et les spécificités de chaque langage (propriétés en C#, Date, ...) auront un impact plus ou moins grand. Des outils peuvent prendre en charge cette opération de génération de code en prenant soin de proposer différentes alternatives (ArrayList ou HashMap) à l'utilisateur.

Remarque: Visual Studio .NET propose dans sa version *Architect* la possibilité de créer ses diagrammes directement dans l'outil avec MS Visio. Un partenariat existe entre [Microsoft et Rational](#) concernant l'intégration de Rose dans VS.NET pour les versions à venir.

## 9 TP :Etude de Cas: deux DCC sur la Gestion de Formations (Roques)

Nous allons donner des exemples de passage d'un modèle UML (de niveau conception détaillée) à du code C# .

Voir étude :

580\_Roques\_Demande\_de\_Formation\_IMPLEMENTATION\_Classes\_PDF\_JavaUML

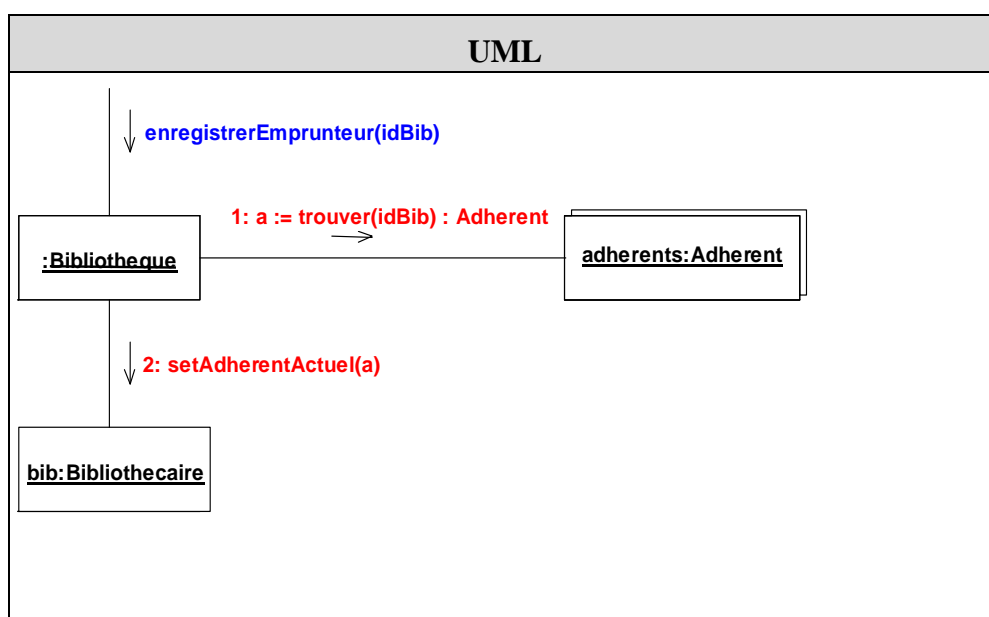
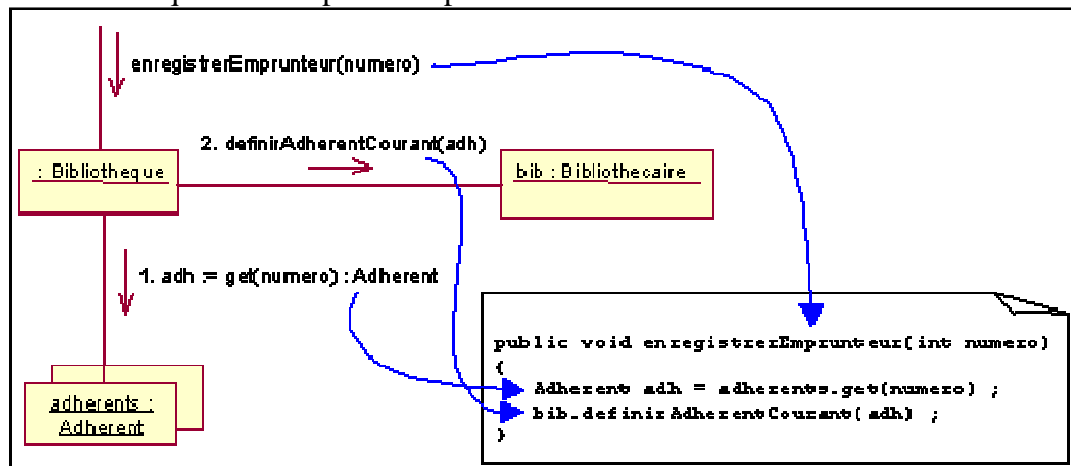
## 10 Diagrammes de communication A\_Revoir Bibliothèque

De même, une interaction complexe entre objets de différentes classes peut donner de façon très directe le **corps de la méthode** appelée (ici, EnregistrerEmprunteur ( ) de la classe Bibliothèque).

L'exemple suivant illustre la correspondance entre la numérotation des messages dans un diagramme de communication et le code correspondant : Corps de la méthode enregistrerEmprunteur.

La bibliothèque:

OUI Il manque le DCC pour comprendre.



### Java

```
public class Bibliotheque
{
    ...
    private Bibliothecaire bib ;
    private Map<Adherent,idAdh> adherents = new HashMap<Adherent,idAdh> () ;
    ...

    public void enregistrerEmprunteur(int idBib)
    {
        Adherent a = (Adherent)adherents[idBib];
        bib.setAdherentActuel(a);
    }
}
```

### C#

```
public class Bibliotheque
{
    private Bibliothecaire bib ;
    private HashTable adherents = new HashTable() ;

    public void EnregistrerEmprunteur(int idBib)
    {
        Adherent a = (Adherent)adherents[idBib];
        bib.SetAdherentActuel(a);
    }
}
```

## 10.1 Obsolète: Exemple de passage du diagramme de classes au code C#

La Bibliothèque:

