



Vous souhaitez utiliser un outil de développement performant qui permette à la fois de coder et de visualiser graphiquement votre conception. Vous avez entendu parler de Rational/XDE, de Together, et autres ... Mais vous n'êtes pas un expert UML, et vous vous posez pas mal de questions sur les correspondances détaillées entre les nombreux concepts et diagrammes UML proposés par ces outils et le code C# qui va en découler. Ou simplement, vous voulez mieux maîtriser le paramétrage de la génération de code de votre IDE préféré ...

Cet article vise à jeter un pont entre les concepts de la modélisation UML et le monde de l'implémentation dans le langage orienté objet phare de la plateforme .NET : C#.

UML est un langage de modélisation visuelle, C# est un langage de programmation textuel. 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 C#, et c'est ce que nous allons tenter de détailler dans la suite de cet article.

Nota : cet article est largement inspiré de l'annexe 1 de la nouvelle édition du livre de Pascal Roques « UML par la pratique », Eyrolles 2003 (à paraître en Février).

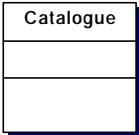
La structure statique

Les concepts structuraux (ou statiques) tels que classe et interface sont évidemment fondamentaux aussi bien en UML qu'en C#. Ils sont représentés en UML dans les diagrammes de classes, et constituent le squelette d'un code orienté-objet.

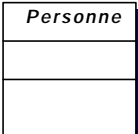
Nous aborderons tour à tour les concepts UML de classe, d'interface, de package, d'attribut et d'opération.

Classe

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

UML	C#
	<pre>public class Catalogue { ... }</pre>

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 C#.

UML	C#
	<pre>abstract public class Personne { ... }</pre>

Interface

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

UML	C#
	<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, alors que les méthodes C# comment plutôt par une majuscule.

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 en C#, mais sous un nom différent : namespace.

UML	C#
	<pre>namespace Catalogue { ... }</pre>

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#.

Attribut

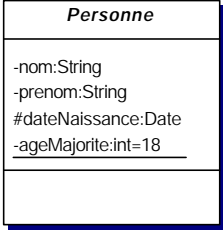
Les attributs UML deviennent simplement des variables (ou membres) en C#.

Leur type est soit un type primitif (int, etc.), soit une classe fournie par la plateforme .NET (string, DateTime, etc.). 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 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>

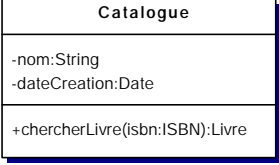
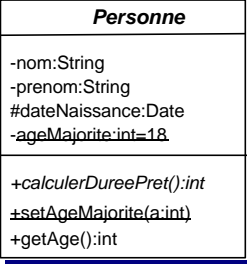
 <pre> classDiagram class Personne { -nom:String -prenom:String #dateNaissance:Date -ageMajorite:int=18 } </pre>	<pre> abstract public class Personne { private string nom; private string prenom; protected DateTime dateNaissance; private static int ageMajorite = 18; } </pre>
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Opération

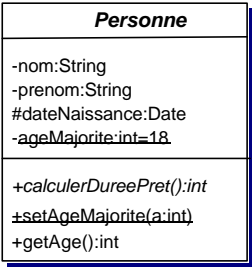
Les opérations UML deviennent très directement des méthodes en C#.

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 C#.

UML	C#
 <pre> classDiagram class Catalogue { -nom:String -dateCreation:Date +chercherLivre(isbn:ISBN):Livre } </pre>	<pre> public class Catalogue { private string nom; private DateTime dateCreation; public Livre ChercherLivre(ISBN isbn) { ... } ... } </pre>
 <pre> classDiagram class Personne { -nom:String -prenom:String #dateNaissance:Date -ageMajorite:int=18 +calculerDureePret():int +setAgeMajorite(a:int) +getAge():int } </pre>	<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>

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

UML	C#
 <pre> classDiagram class Personne { -nom:String -prenom:String #dateNaissance:Date -ageMajorite:int=18 +calculerDureePret():int +setAgeMajorite(a:int) +getAge():int } </pre>	<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>

Les relations entre concepts statiques

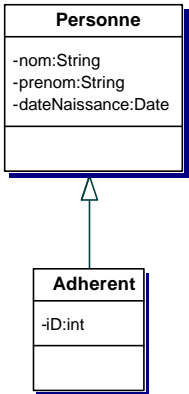
Les relations UML entre concepts statiques sont très riches. On peut distinguer les relations de :

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

Elles ne se traduisent pas toutes de façon simple par un mot-clé en C# (comme dans les autres langages objets).

Généralisation

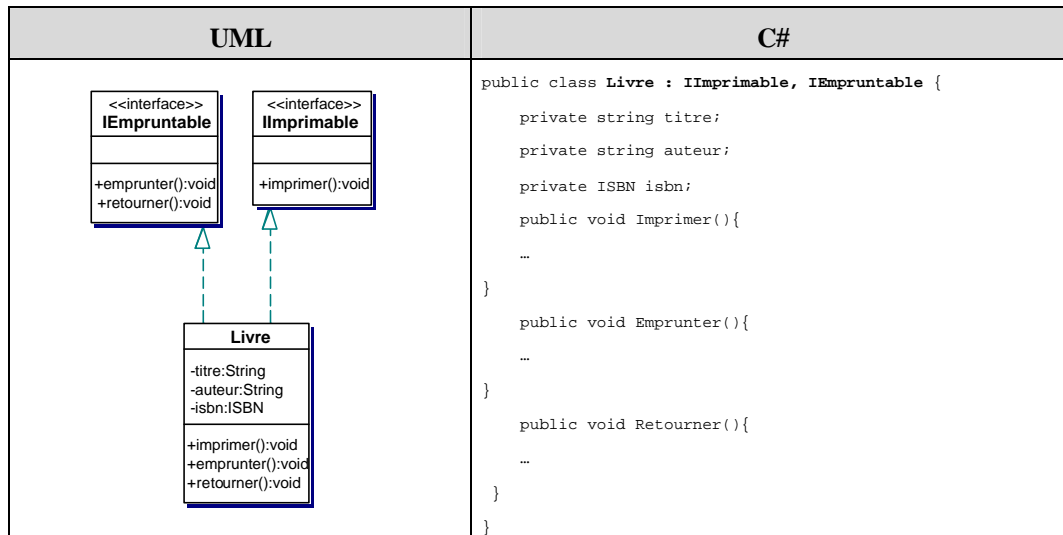
Le concept UML de généralisation se traduit directement par le mécanisme de l'héritage dans les langages objets. C# utilise la syntaxe du C++ pour l'héritage, mais interdit l'héritage multiple entre classes.

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

Réalisation

Une classe UML peut implémenter plusieurs interfaces.

Contrairement à C++, mais à l'identique de Java, le langage C# propose directement ce mécanisme.

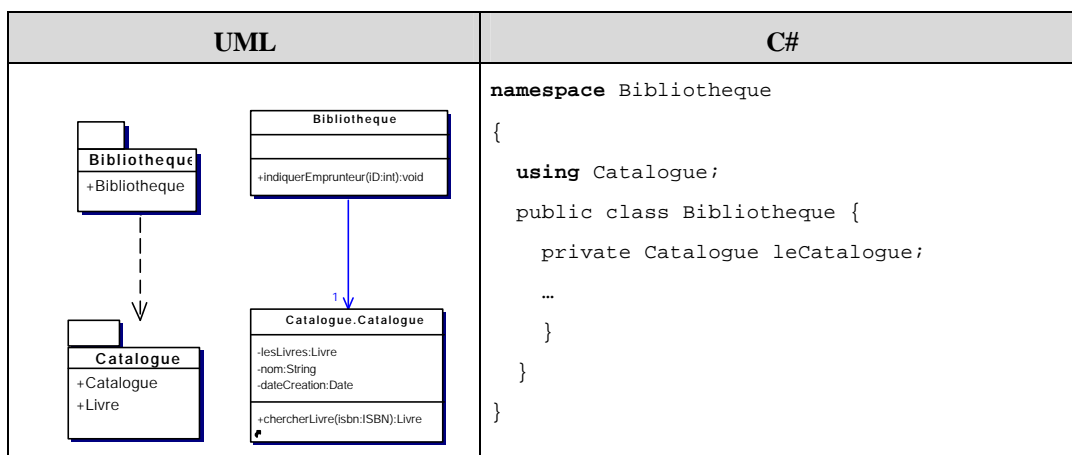


Dépendance

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 C#.

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



Association

Les associations navigables UML se traduisent par du code C# qui dépend notamment de la multiplicité de l'extrémité concernée, mais aussi de l'existence ou pas d'une contrainte {ordered} ou d'un qualificatif. Essayons de voir tout cela du plus simple au plus complexe :

- ?? Une association navigable avec une multiplicité 1 se traduit par une variable d'instance, tout comme un attribut, mais avec un type référence vers une instance de classe du modèle au lieu d'un type simple.
- ?? 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 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 : *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.

Toutes ces règles importantes à comprendre si vous devez générer du code automatiquement à partir d'un outil UML sont résumées sur le schéma synthétique suivant.

UML	C#
	<pre>public class A1 { private B1 leB1; ... }</pre>
	<pre>public class A2 { private B2[] lesB2; ... }</pre>
	<pre>public class A3 { private ArrayList lesB3 = new ArrayList(); ... }</pre>
	<pre>public class A4 { private Hashtable lesB4 = new Hashtable (); ... }</pre>

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>public class Homme { private Femme epouse; ... } public class Femme { private Homme mari; ... }</pre>

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

UML	C#
	<pre>public class Personne { private Personne[] subordonne; private Personne chef ; ... }</pre>

Agrégation et composition

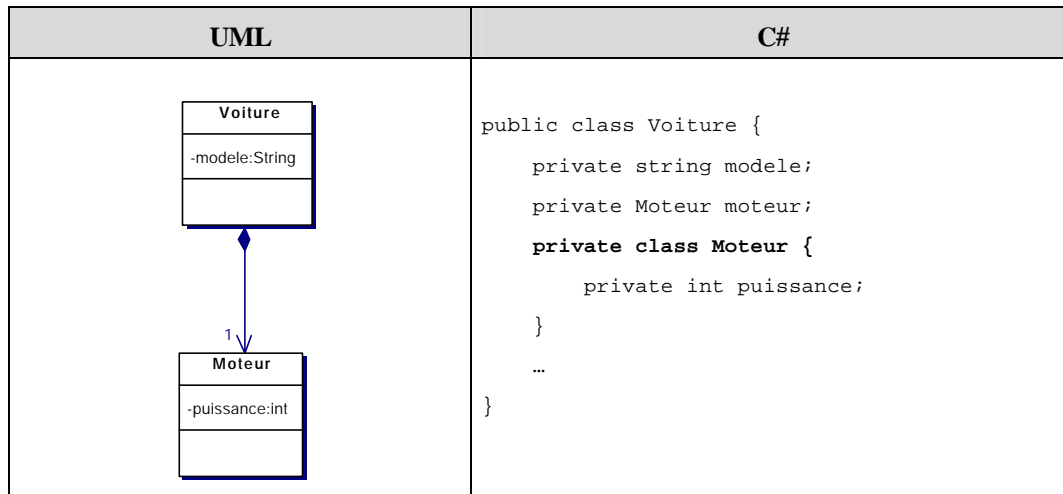
L'agrégation est un cas particulier d'association non symétrique exprimant une relation de contenance. Les agrégations n'ont pas besoin d'être nommées : implicitement elles signifient «contient », «est composé de ». La sémantique des agrégations n'est pas fondamentalement différente de celle des associations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

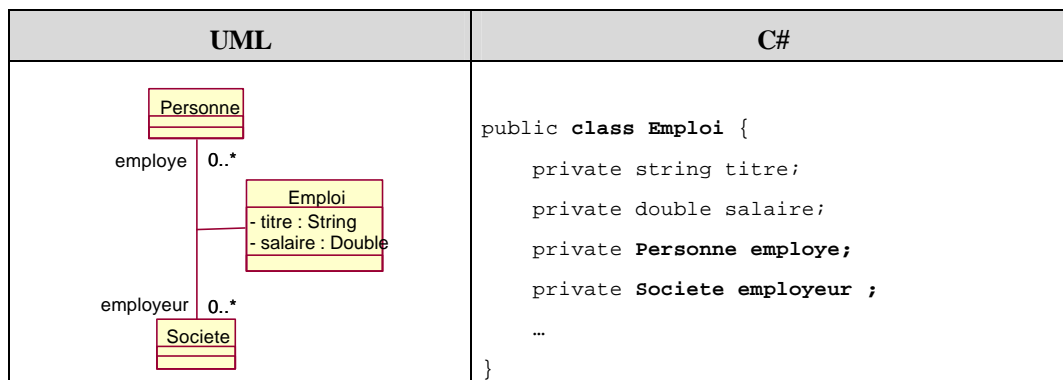
Dans certains langages objet, par exemple C++, la composition implique donc la propagation du destructeur, mais cela ne s'applique pas à C#.

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



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.



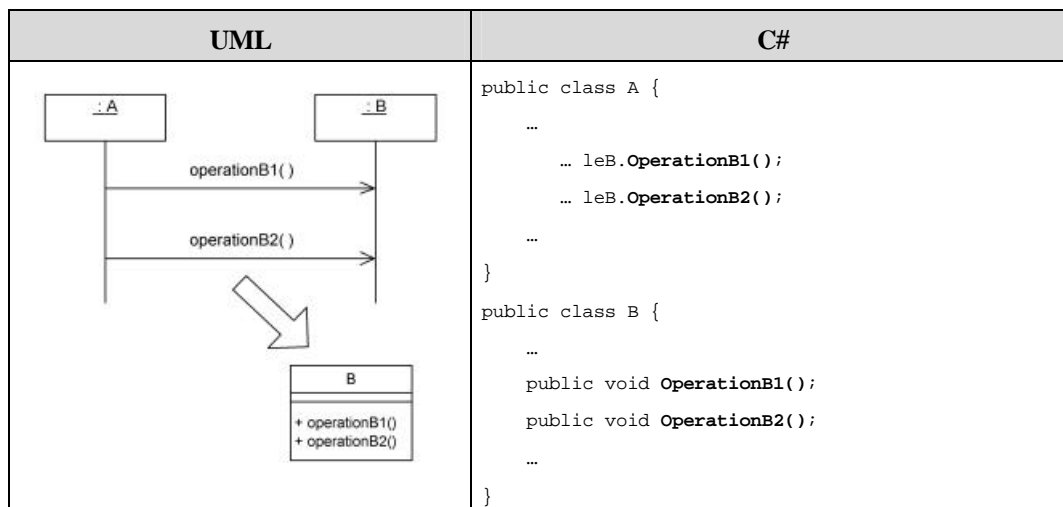
La dynamique

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 digrammes 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 ...

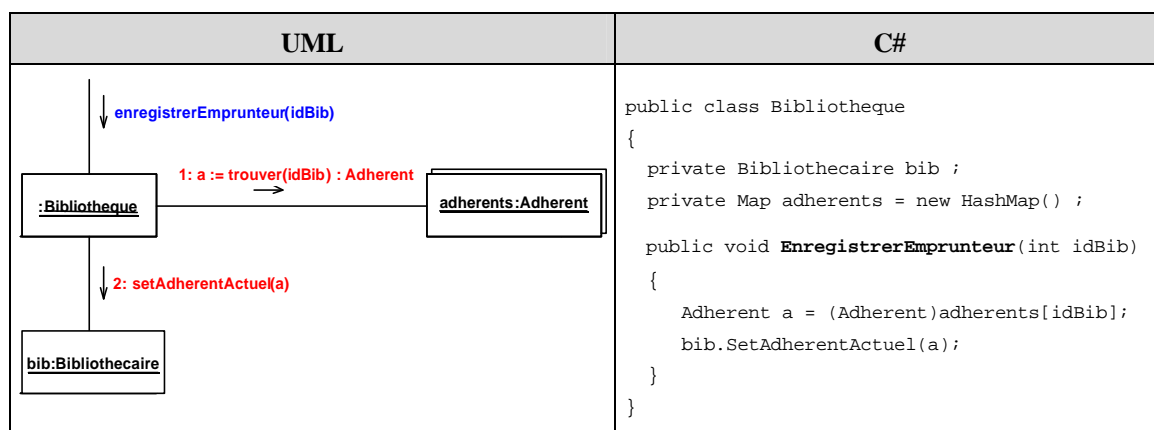
Interactions

Les diagrammes d'interactions UML (séquence ou collaboration) sont particulièrement utiles au concepteur 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.



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 Bibliotheque).



État

La notion d'état n'existe pas dans les langages orientés objet et C# n'échappe pas à la règle. Cependant, les concepteurs objets expérimentés ont mis au point des techniques éprouvées permettant de rendre compte de cette notion avec les éléments de base existants. Nous rejoignons ici le concept maintenant bien connu de «Design Pattern», popularisé par le célèbre livre du GoF...

Les outils les plus avancés nous proposent ainsi de traduire les éléments de modélisation UML saisis sous forme de diagrammes d'états par des ajouts judicieux de classes et de méthodes.

Mais ce sujet complexe méritera à lui seul l'écriture d'un autre article ultérieurement ...

Auteur : Pascal ROQUES – Copyright © - DotNetGuru.org