Annexe B

.NET en pratique

L'objectif de cette annexe est de permettre à chacun de se familiariser un peu avec l'environnement de programmation .NET. Pour cela nous allons développer une petite application permettant à chacun de géré ses cartes de visite (ajout, suppression et consultation) et de les rendre accessibles aux autres utilisateurs (uniquement consultation). Pour atteindre ce but, l'application se compose de deux parties : le service d'annuaire qui doit pouvoir être utilisée à distance et différentes applications clientes de type client léger ou client lourd graphique ou en ligne de commande. Pour des raisons de simplicité nous ne nous intéresserons pas aux problèmes liés à l'authentification des différentes classes d'utilisateurs.

Une version html de ce texte est disponible à l'adresse `http ://rangiroa.essi.fr/riveill/enseignement/tp/carteVisite.html`. Elle contiendra les évolutions futures du sujet.

B.1 Les outils à utiliser

Comme nous allons faire des développements simples nous allons utiliser les outils mis 'gracieusement' en ligne par Microsoft et les installer les uns après les autres :

1. Le *framework .NET*[1] est généralement installé à l'adresse `C:\Windows\Microsoft .Net\Framework` et contient les principales classes nécessaires à l'exécution des programmes ainsi que le compilateur (**csc.exe**). Pensez à mettre à jour vos variables d'environnement avec l'adresse d'installation du *framework .NET*.
2. Le *kit de développement (SDK)*[2] est généralement installé à l'adresse `C:\Program Files\Microsoft.Net\SDK` et contient les principaux utilitaires : **xsd.exe** (pour générer la documentation associée aux classes), **ildasm.exe** (le désassembleur) ou **wsdl.exe** (le générateur de talon pour les services web).

[1]Framework .Net : `http ://www.microsoft.com/france/msdn/netframework/default.mspx`
[2]SDK : `http ://www.microsoft.com/france/msdn/netframework/default.mspx`

3. L'environnement de programmation *Web Matrix*[3] a comme principaux avantages d'avoir une faible empreinte mémoire, d'être gratuit, simple d'utilisation, d'inclure un serveur HTTP et de permettre la création et l'accès à des bases de données simples au format *access* sans aucune installation supplémentaire.

Ce n'est pas le seul environnement possible, en effet vous pouvez utiliser *Mono*[4], comme implémentation du framework et *SharpDevelop*[5] comme environnement de programmation. Par ailleurs, si vous le souhaitez, Microsoft met à disposition des programmeurs des versions gratuites de ses environnements de développement[6]. L'offre *MSDN Academic Alliance*[7] permet aux universités d'avoir accès à Visual Studio, SQL Serveur et à l'ensemble des outils de développement commercialisés par Microsoft.

B.2 Architecture de l'application "Cartes de Visite"

Nous allons développer au cours de ce TD une application de gestion de cartes de visite qui respectera l'architecture décrite dans la figure B.1.

Cette application sera développée en C#.

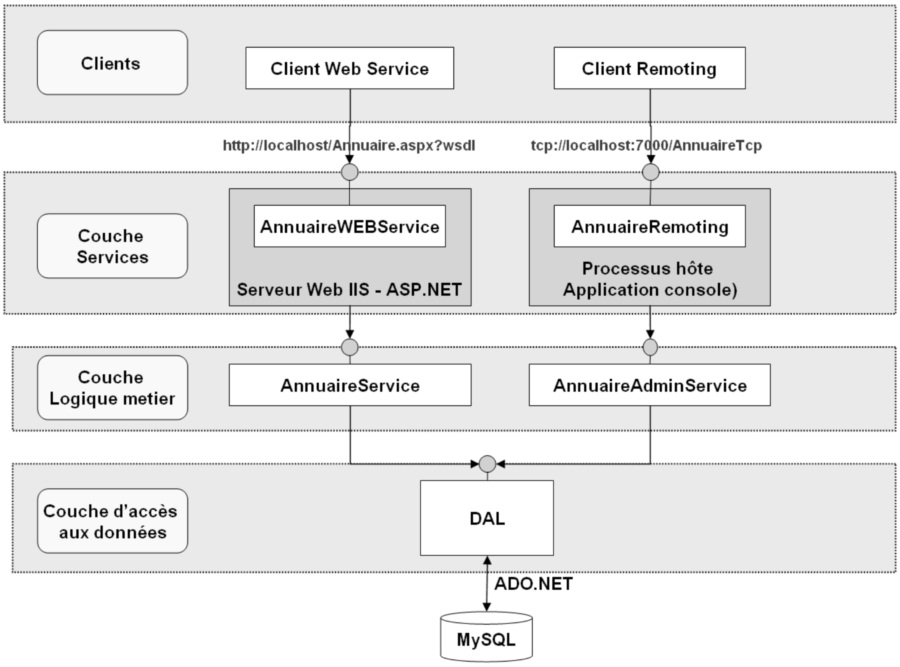

Figure B.1 – Architecture de l'application

[3]Web Matrix : `http ://www.asp.net/webmatrix/`
[4]Mono : `http ://www.go-mono.org`
[5]SharpDevelop : `http ://sourceforge.net/projects/sharpdevelop/`
[6]Visual Studio Express : `http ://www.microsoft.com/france/msdn/vstudio/express/default.mspx`
[7]MSDN AA : `http ://www.microsoft.com/france/msdn/abonnements/academic/default.mspx`

B.3 Les objets métiers

Nous choisissons de représenter les données de chaque carte dans des instances d'une classe `CarteDeVisite` dont le code C# est donné ci-dessous :

CarteDeVisite (CarteDeVisite.cs)

```
using System;
using System.Xml;
using System.Xml.Serialization;

namespace MonPremierTP {
   [Serializable][XmlType("carte-de-visite")]
   public class CarteDeVisite {
      [XmlElement("nom")] public string Nom;
      [XmlElement("prenom")] public string Prenom;
      [XmlElement("email")] public string Email;
      [NonSerialized()]
      public string Telephone;
   }
}
```

Nous verrons ultérieurement comment les différentes fiches seront initialisées par des données issues de la base de données. La description et l'utilisation des différents attributs sont présentés dans la section B.6.5.

Pour génerer la bibliothèque CarteDeVisite.dll, la commande est la suivante :

```
csc /target:library CarteDeVisite.cs
```

B.4 La couche d'accès aux données

Cette couche permet de d'alimenter en données les différents objets métiers manipulés par l'application cartes de visite. Nous donnons uniquement ici l'interface, la description de l'implémentation sera faite ultérieurement. Voici la version 0.1 :

AnnuaireDAL (AnnuaireDAL.cs)

```
// volontairement le namespace n'est pas répéter...
// à vous de l'indiquer et d'utiliser les bonnes directives 'using'
[assembly:AssemblyVersion("0.1.0.0")]
public class AnnuaireDAL {
   public AnnuaireDAL() {} // constructeur
   public CarteDeVisite ExtraireCarte(string nomRecherche) {
      // opération de recherche d'une carte dans la BD
      return null;
   }
   public void AjouterCarte(CarteDeVisite carte){
      // opération d'insertion d'une nouvelle carte
   }
}
```

Pour générer la bibliothèque AnnuaireDAL.dll, la commande est la suivante :

```
csc /target:library /reference:CarteDeVisite.dll AnnuaireDAL.cs
```

B.5 La couche métier

Quand nous aurons implémenté la classe *AnnuaireDAL* décrite dans la section précédente, nous disposerons d'un moyen d'accéder aux données persistantes du service d'annuaire sans avoir à connaître la technologie de stockage employée. L'interrogation et la mise à jour des données seront faites au travers des interfaces de la *DAL (Data Access Layer)* en ne manipulant que des entités liées au domaine métier.

Pour créer l'indépendance entre conservation des données et manipulation des données, nous allons créer la couche contenant la logique du service d'annuaire. Il s'agit d'implanter deux services, l'un pour consulter les cartes de visites existantes et un autre pour en insérer de nouvelles. L'idée est ici de spécifier clairement les services sous la forme d'interface. Cette couche ne sera manipulée qu'au travers de ces seules interfaces et non par un accès direct à leur implantation. Voici les interfaces des deux services :

IAnnuaireService (IAnnuaireService.cs)

```
public interface IAnnuaireService {
   CarteDeVisite Rechercher(string nom);
}
public interface IAnnuaireAdminService {
   CarteDeVisite Rechercher(string nom);
   void Ajouter(CarteDeVisite carte);
}
```

Une fois ces interfaces définies, il ne nous reste qu'à les implanter en utilisant la couche d'accès aux données décrite précédemment.

AnnuaireService (AnnuaireService.cs)

```
public class AnnuaireService : IAnnuaireService {
   // constructeur de la classe
   public AnnuaireService() {}
   // implémentation de la méthode métier
   public CarteDeVisite Rechercher(string nom) {
      AnnuaireDAL dal=new AnnuaireDAL();
      return dal.ExtraireCarte(nom);
   }
}

public class AnnuaireAdminService : IAnnuaireAdminService {
   // constructeur de la classe
   public AnnuaireAdminService(){}
   // implémentation de la méthode métier
   public CarteDeVisite Rechercher(string nom) {
      AnnuaireDAL dal=new AnnuaireDAL();
```

```
      return dal.ExtraireCarte(nom);
   }
   public void Ajouter(CarteDeVisite carte) {
      AnnuaireDAL dal=new AnnuaireDAL();
      // on vérifie que la carte n'existe pas déjà
      CarteDeVisite test=dal.ExtraireCarte(carte.Nom);
      if (test==null)
         dal.AjouterCarte(carte);
      else
         throw new Exception("La carte existe déjà");
   }
}
```

B.6 Premières manipulations en C#

Ces expérimentations ne sont pas directement liées avec la mise en œuvre de l'application cartes de visite mais permettent de se familiariser avec le langage C# et le framework .Net.

B.6.1 Construire le diagramme UML

A cette étape du TP, nous avons écrit 4 fichiers : `CarteDeVisite.cs`, `AnnuaireDAL.cs`, `IAnnuaireService.cs` et `AnnuaireService.cs` et il peut être souhaitable de construire le diagramme UML de l'ensemble.

B.6.2 Générer les fichiers de documentation

Pour commenter le code, C# offre trois types de commentaires :

- les commentaires introduits par les balises '//', la fin étant la fin de ligne ;
- les commentaires introduits par les balises '/*' et terminé par les balises '*/' ;
- les commentaires introduits par les balises '///', la fin étant la fin de ligne.

Seul, ce dernier type va nous intéresser car il permette de générer des fichiers de documentation associés au code développé. Dans *Visual Studio* ou *SharpDevelop*, le fait de mettre trois slash à la suite dans un endroit valide fait apparaitre un menu avec les balises XML de documentation dont nous présentons ici les principales :

- La balise *summary* sert à donner la description complète de l'élément que l'on souhaite documenter. Il peut être utilisé sur une classe, une méthode, d'une propriété ou une variable.
- La balise *param* permet de documenter les paramètres d'une méthode ou d'une propriété. Il prend en complément uniquement le nom de la variable, le type de la variable est automatiquement déterminé par le générateur de documentation.
- La balise *returns* permet de documenter la valeur de retour d'une fonction seulement.
- La balise *value* permet de décrire la valeur de retour ou d'entrée d'une propriété. Elle joue le même rôle que la balise *param* pour une fonction.
- La balise *paramref* permet d'indiquer que le mot dans le commentaire est un paramètre de la fonction afin que le générateur de documentation puis mettre un lien vers le commentaire du paramètre.

– La balise *exception* permet d'informer sur le(s) type(s) d'exception(s) que la fonction peut lever. La propriété *cref* du tag permet de spécifier le type d'exception documenté.
– Il existe bien d'autres balises : `<c>`, `<code>`, `<example>`, `<include>`, `<list>`, `<para>`, `<permission>`, `<remarks>`, ¡see>, `<seealso>`. A vous de découvrir leur utilisation en lisant la documentation.

Voici un exemple de programme commenté avec ces balises :

```
// xml_AnnuaireDAL.cs
// à compiler par : csc /doc:xml_AnnuaireDAL.xml

[assembly:AssemblyVersion("0.2.0.0")]
/// Mon texte pour la classe AnnuaireDAL
/// <summary>
/// Description de la classe AnnuaireDAL .
/// </summary>
public class AnnuaireDAL {
   /// <remarks>
   /// Des commentaires plus longs peuvent être associés à un type ou un membre
   /// grâce à la balise remarks</remarks>
   /// <example> Cette classe permet d'isoler la partie métier de la persistance
   /// des données, elle comporte essentiellement deux méthodes.
   /// <code>
   /// public CarteDeVisite ExtraireCarte(string nomRecherche)
   /// public void AjouterCarte(CarteDeVisite carte)
   /// </code>
   /// </example>
   /// <summary>
   /// Constructeur de la classe AnnuaireDAL.
   /// </summary>
   public AnnuaireDAL() {} // constructeur

   /// texte pour la méthode ExtraireCarte
   /// <summary>
   /// Cette méthode très sophistiquée reste totalement à implémenter. Le paramètre
   /// d'entrée est <paramref name="nomRecherche" />
   /// </summary>
   /// <param name="nomRecherche">
   /// le paramètre d'entrée est le nom du titulaire de la carte de visite recherchée
   /// </param>
   /// <returns>
   /// le résultat est la carte de visite associée au nom
   /// <returns>
   /// <exception cref="NomInconnu">
   /// Une exception est levée si le nom est inconnu
   /// </exception>
   public CarteDeVisite ExtraireCarte(string nomRecherche) {
      // opération de recherche d'une carte dans la BD
      return null;
   }
```

```
   /// texte pour la méthode AjouterCarte
   /// <summary>
   /// Description de AjouterCarte.</summary>
   /// <param name="carte"> Emplacement de la description du paramètre carte</param>
   /// <seealso cref="String">
   /// Vous pouvez utiliser l'attribut cref sur n'importe quelle balise pour
   /// faire référence à un type ou un membre,
   /// et le compilateur vérifiera que cette référence existe. </seealso>
   public void AjouterCarte(CarteDeVisite carte){
      // opération d'insertion d'une nouvelle carte
   }

   /// <summary>
   /// Une propriété introduite juste pour montrer la balise "value".
   /// </summary>
   /// <value>donne toujours 0492965148</value>
   public int UnePropriete {
      get {
        return 0492965148;
      }
   }
}
```

Mettez ce code dans le fichier `commentaire1.cs` et compilez le : `csc.exe /doc :commentaire1.xml /reference :CarteDeVisite.dll commentaire1.cs` et examinez le fichier XML produit.

Maintenant, à vous de jouer en commentant les différentes classes précédemment écrites. Faites bien la différence entre les commentaires débutants par '///', pris en compte dans la génération du fichier de documentation XML et les commentaires débutant par '//' ou '/*', seulement présent dans le fichier source.

Attention : lorsque vous générez la documentation, vous compiler aussi le code en question.

B.6.3 Compilation du code

Compilez les différents fichiers pour obtenir une dll.

```
csc.exe /out:annuaire.dll /t:library /r:AnnuaireDAL.dll /r:CarteDeVisite.dll \
        AnnuaireService.cs IAnnuaireService.cs
```

Observez avec `ildasm` la *dll* produite et commentez. Qu'observez-vous : nombre de classes, fonctions membres ?

```
ildasm.exe annuaire.dll
```

B.6.4 Utilisation de la réflexivité

Nous allons utiliser la réflexivité pour imprimer à l'écran les différents éléments de la dll précédemment produite. Voici un exemple :

```
using System;
using System.IO;
using System.Reflection;

public class Meta {
   public static int Main () {
      // lire l'assembly
      Assembly a = Assembly.LoadFrom ("annuaire.dll");

      // lire les modules de l'assembly
      Module[] modules = a.GetModules();
      // inspecter tous les modules de l'assembly
      foreach (Module module in modules) {
         Console.WriteLine("Dans le module {0}, on trouve", module.Name);
         // lire tous les types du module
         Type[] types = module.GetTypes();
         // inspecter tous les types
         foreach (Type type in types) {
            Console.WriteLine("Le type {0} a cette(ces) méthode(s) : ", type.Name);
            // inspecter toutes les méthodes du type
            MethodInfo[] mInfo = type.GetMethods();
            foreach (MethodInfo mi in mInfo) {
               Console.WriteLine (" {0}", mi);
            }
         }
      }
      return 0;
   }
}
```

Compilez cette classe (`csc <nom de fichier>.cs`) et exécutez le code, puis faites une lecture attentive du source. Avez-vous tout compris ?

B.6.5 Sérialisation binaire et XML

Voici un cours programme qui permet de sérialiser/désérialiser une carte de visite en XML et en format binaire.

```
using System;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
using System.Xml;
using System.Xml.Serialization;

class Serialisation {
   static void Main(string[] args) {
      CarteDeVisite uneCarte = new CarteDeVisite();

      uneCarte.Nom = "RIVEILL";
      uneCarte.Prenom = "Michel";
      uneCarte.Email = "riveill@unice.fr";
```

```
      uneCarte.Telephone = "04 92 96 51 48";

      // on sérialise en binaire et on sauvegarde dans un fichier
      Stream stream = File.Open("Demo.bin", FileMode.Create);
      BinaryFormatter formatter = new BinaryFormatter ();
      formatter.Serialize (stream, uneCarte);
      stream.Close();

      // on sérialise en XML et on sauvegarde dans un fichier
      Stream stream2 = File.Open("Demo.xml", FileMode.Create);
      XmlSerializer serializer = new XmlSerializer (typeof(CarteDeVisite));
      serializer.Serialize(stream2, uneCarte);
      stream2.Close();

      // on désérialise depuis un fichier (mode binaire)
      stream = File.Open ("Demo.bin", FileMode.Open);
      formatter = new BinaryFormatter();
      CarteDeVisite obj = (CarteDeVisite) formatter.Deserialize (stream);
      stream.Close ();

      // on désérialise depuis un fichier (mode XML)
      stream2 = File.Open ("Demo.xml", FileMode.Open);
      serializer = new XmlSerializer (typeof(CarteDeVisite));
      CarteDeVisite obj2 = (CarteDeVisite) serializer.Deserialize (stream2);
      stream2.Close ();

      Console.WriteLine ("\t\torigine\t\tbinaire\t\tXml");

      // Nom, prenom et email ont bien été sérialisés
      Console.WriteLine ("uneCarte.nom (a ete serialisee)");
      Console.WriteLine ("\t\t{0}\t{1}\t{2}", uneCarte.Nom, obj.Nom, obj2.Nom);

      Console.WriteLine ("uneCarte.prenom (a ete serialisee)");
      Console.WriteLine ("\t\t{0}\t{1}\t{2}", uneCarte.Prenom, obj.Prenom, obj2.Prenom);

      Console.WriteLine ("uneCarte.email (a ete serialisee)");
      Console.WriteLine ("\t\t{0}\t{1}\t{2}", uneCarte.Email, obj.Email, obj2.Email);

      // téléphone n'a pas été sérialisé
      Console.WriteLine ("uneCarte.telephone (n'a pas ete serialisee)");
      Console.WriteLine ("\t\t{0}\t\t{1}\t\t{2}", uneCarte.Telephone,
                         obj.Telephone, obj2.Telephone);
   }
}
```

Compilez et exécutez ce code. Analysez-le.

Remarque : Si vous sérialisez un tableau ou une collection, tous les objets du tableau ou de la collection sont sérialisés.

B.7 La couche de stockage physique des données

On reprend la conception de l'application. Les données stockées dans l'annuaire sont des cartes de visite contenant un nom, un prénom, un courriel et un numéro de téléphone. Pour que ceux qui ne sont pas très aguerri avec la manipulation des bases de données il est possible de créer celles-ci directement avec *Web Matrix*. Voici la manière de procéder :

1. ouvrir *web matrix*
2. aller dans la fenêtre 'Workspace'
3. sélectionner l'onglet 'Data'
4. cliquer sur l'icone 'Nouvelle Base de Données'
5. créer une base de données Access
6. créer la table en sélectionnant table, dans la BD créée
7. cliquer sur l'icone 'Nouvelle Table'
8. Pour finir, créer les entrées de la table qui doivent au moins contenir les champs suivants ainsi qu'une clé primaire non décrite ici :

```
string Nom;
string Prenom;
string Email;
string Telephone;
```

Par exemple la table obtenue peut avoir le format décrit dans la figure B.2.

Column Name	Data Type	Size	Allow Nulls
key	AutoNumber	0	False
nom	Text	50	False
prenom	Text	50	True
email	Text	50	True
telephone	Text	50	True

Figure B.2 – Format de la base de données

Il est alors possible d'initialiser la base de données en sélectionnant l'onglet 'Data'. Un exemple de contenu est donné dans la figure B.3.

B.8 Accès direct à la base de données par un client léger

Dans un premier temps et pour nous familiariser avec l'environnement, nous allons construire un premier client capable d'accéder à la base de données. La procédure est relativement simple.

1. ouvrir web matrix
2. créer un nouveau fichier ('File', 'New File') puis 'General', 'ASP.Net Page', et finir de remplir le formulaire (le suffixe du fichier sera aspx).

	key	nom	prenom	email	telephone
▶	1	RIVEILL	Michel	riveill@unice.f	(null)
	2	BLAY	Mireille	(null)	(null)
	3	PINNA	Anne-Marie	(null)	(null)
	4	TIGLI	Jean-Yves	(null)	(null)
	5	HUGUES	Anne-Marie	(null)	(null)
*					

Figure B.3 – Contenu de la base de données

3. une fenêtre d'édition comportant 3 modes : 'dessin', 'html' et 'code' est à votre disposition ('all' est la fusion du mode 'html' et 'code'). En mode 'dessin' mettez sur la page une `TextBox` (saisie de chaine de caractère), un `DataGrid` et un `Button`.
4. en mode 'html' vous pouvez renommer les identificateurs des différents éléments. Par exemple pour moi, le *TextBox* a `"nom"` comme identificateur, le *DataGrid* a `"listeCartes"` comme identificateur et le *Button* a `"Button"` comme id et `"Chercher"` comme Text.
5. écrire la méthode ExtraireCarteSQL à l'aide de l'utilitaire fournit par *Web Matrix* (procédure décrite ci-après).

Pour écrire la méthode ExtraireCarteSQL à l'aide de l'utilitaire présent dans *Web matrix* il faut être en mode 'code', puis faire glisser la zone `SELECT Data Method` de la colonne *ToolBox* à l'endroit ou vous voulez insérer la méthode. Une boite de dialogue s'ouvre. Elle permet de sélectionner la base de données, puis de construire la requête `SELECT` souhaitée.

Pour cela, sélectionnez les bonnes colonnes, remplissez la clause `WHERE` pour obtenir la requête décrite dans la figure B.4.

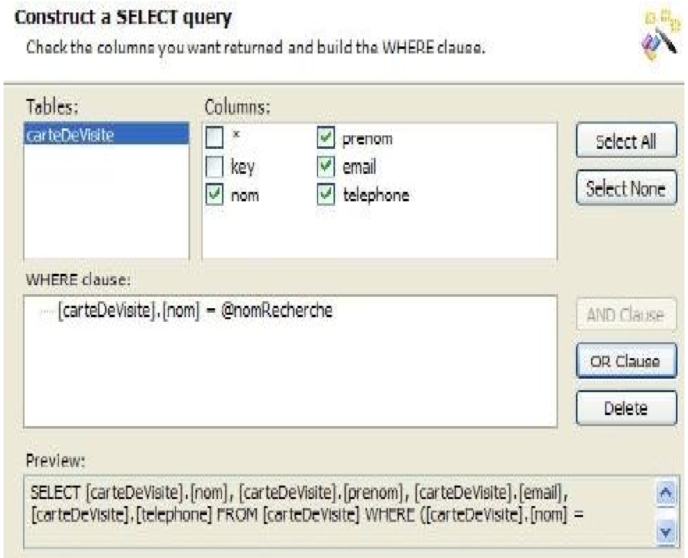

Figure B.4 – Clause `WHERE`

Après avoir cliqué sur le bouton suivant, testez la requête écrite. Si elle convient, donnez le nom de la méthode et demandez le résultat sous la forme d'un `DataSet`.

Pour terminer l'exemple, il reste à programmer la méthode appelée lorsque le bouton sera sélectionné. Pour cela, en mode 'dessin', cliquez 2 fois sur le `Button`. La fenêtre d'édition bascule en mode 'code' et l'on peut saisir le code associé à l'évènement 'clic sur le `Button`'. Le code de cette méthode est le suivant :

```
listeCartes.DataSource = ExtraireCarteSQL (nom.Text);
listeCartes.DataBind();
listeCartes.Visible = true;
```

Petites explications : la première ligne met à jour la liste des données du *DataSet* en appelant la méthode `ExtraireCarteSQL` construite dans l'étape 6 précédente, le paramètre est le texte de la `TextBox` contenue dans la page. Les deux lignes mettent à jour le `DataSet` et le rendent visible.

Attention : cette méthode, certes très efficace en terme de programmation ne respecte pas du tout l'architecture initiale, elle a juste permis d'utiliser la base de données créée et de nous familiariser avec *Web Matrix* pour la création de page web dynamique. La suite du TP reprend la construction pas à pas des différentes couches de l'application cartes de visite.

B.9 La couche d'accès aux données

Cette couche à pour rôle de faire abstraction de la technologie utilisée pour stocker les données. Elle doit masquer le moteur de base de données utilisé à la couche suivante, la couche métier que nous avons déjà écrit. L'avantage de cette abstraction est qu'il est possible de remplacer le moteur de SGBD utilisé par toute autre moteur de base de données ou encore par un système de stockage simple à l'aide d'un système de gestion de fichier. Si une telle modification avait lieu, seule cette couche d'accès aux données devrait être réécrite et la modification serait transparente au reste de l'application.

L'API ADO.NET est utilisée pour dialoguer avec le serveur de base de données utilisé (ici Access). Il faut maintenant implémenter la classe en charge du dialogue avec la base de données pour lire et insérer des cartes de visite. Pour ne pas avoir à écrire directement les requêtes SQL d'interrogation de la base de données nous utiliserons *Web Matrix* pour construire les fonctions nécessaire comme dans la section B.8. Pour cela :

1. reprendre la classe `AnnuaireDal`,
2. compléter la méthode `ExtraireCarte` qui s'appuie sur la fonction `ExtraireCarteSQL` de la section B.8 précédente.
3. compléter la méthode `AjouterCarte` selon le même principe en utilisant la commande `SQL INSERT` pour créer la fonction `AjouterCarteSQL`. Les fonctions `ExtraireCarteSQL` et `AjouterCarteSQL` font partie du fichier `AnnaireDal.cs`.

```
public CarteDeVisite ExtraireCarte(string nomRecherche) {
   CarteDeVisite carte = new CarteDeVisite ();
   System.Data.DataSet requete = ExtraireCarteSQL (nomRecherche);
```

```
   carte.Nom = requete.Tables["Table"].Rows[0]["nom"];
   carte.Prenom = requete.Tables["Table"].Rows[0]["prenom"];
   carte.Email = requete.Tables["Table"].Rows[0]["email"];
   carte.Telephone = requete.Tables["Table"].Rows[0]["telephone"];
   return carte;
}

public void AjouterCarte(CarteDeVisite carte) {
   AjouterCarteSQL (carte.Nom, carte.Prenom, carte.Email, carte.Telephone);
}
```

B.10 La logique métier : AnnuaireService.dll

Nous avons maintenant les différents éléments pour construire la bibliothèque `AnnuaireService.dll` par compilation du fichier `annuaireService.cs`. Voici un exemple de compilation séparée :

```
REM production des différents modules
csc /t:module CarteDeVisite.cs
csc /t:module /addModule:CarteDeVisite.netmodule AnnuaireDAL.cs
csc /t:module /addModule:CarteDeVisite.netmodule IAnnuaireService.cs

REM liaison des modules produits avec la logique métier
csc /out:AnnuaireService2.dll /t:library /addModule:IAnnuaireService.netmodule \
    /addModule:CarteDeVisite.netmodule /addmodule:AnnuaireDAL.netmodule AnnuaireService.cs
```

Voici un exemple de compilation globale :

```
csc /out:AnnuaireService.dll /t:library IAnnuaireService.cs CarteDeVisite.cs \
    AnnuaireDAL.cs AnnuaireService.cs
```

B.11 Utilisation depuis un client local

Nous allons maintenant, construire un client local capable d'ajouter une nouvelle fiche et de consulter les fiches présentes dans la base.

```
public class ClientLocal {
   public ClientLocal() { }
   static int Main() {
      try {
         IAnnuaireAdminService annuaire = new AnnuaireAdminService ();
         CarteDeVisite carte = annuaire.Rechercher ("RIVEILL");

         Console.WriteLine ("l'email de RIVEILL est {0}", carte.Email);
      } catch (Exception e) {
         Console.WriteLine (e.ToString());
      }
      return 0;
   }
}
```

B.12 Publications des services

Il s'agit d'exposer maintenant le service d'annuaire afin de le rendre accessible aux clients distants. Nous allons exposer ces services de deux manières différents. Le service d'ajout de carte de visite sera publié via le *.NET Remoting* et le service de consultation sera accessible au travers d'un Web Service. Dans le cas de cet exemple d'annuaire, il s'agit d'enrober la logique métier de l'application dans des conteneurs permettant l'accès distant au service. La figure suivante présente les différents objets qui seront créés dans le cas de l'approche *.Net Remoting* et dans le cas de l'approche par service web.

Les différents objets utilisé sont décrits dans la figure B.5.

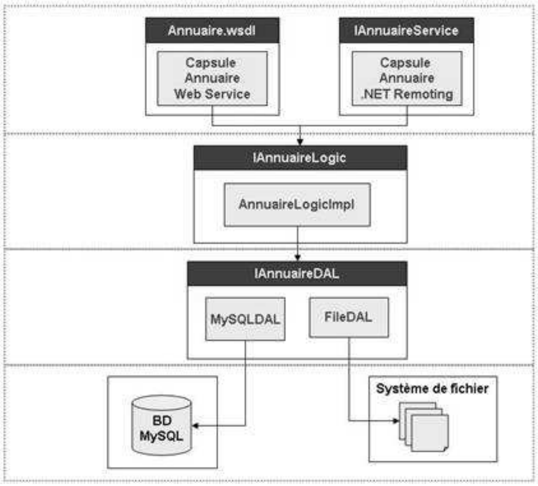

Figure B.5 – Objets utilisés pour les différents appel à distance

B.12.1 Publication .NET Remoting

Pour exposer un service via le .NET Remoting il faut créer un conteneur (objet servant dans la terminologie de l'OMG) capable de distribuer cet objet. Pour cela nous allons construire un exécutable qui publiera l'objet sur le port 7000 via un canal TCP en mode singleton (un seul objet serveur actif à la fois partagé par tous les clients).

AnnuaireServiceRemoting (AnnuaireServiceRemoting.cs)

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;

public class AnnuaireServiceRemoting: MarshalByRefObject, IAnnuaireService {
   IAnnuaireAdminService annuaire = new AnnuaireAdminService ();
   public CarteDeVisite Rechercher(string nom) {
      return annuaire.Rechercher(nom);
```

```
   }
   public void Ajouter(CarteDeVisite carte) {
      annuaire.Ajouter(carte);
   }
}
```

Server (Server.cs)

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;

public class Server {
   [STAThread]
   static void Main(string[] args) {
      IChannel chan = new TcpChannel (7000);
      ChannelServices.RegisterChannel (chan, false);

      RemotingConfiguration.RegisterWellKnownServiceType(
         typeof (AnnuaireServiceRemoting),
         "TcpService",
         WellKnownObjectMode.Singleton);

      Console.WriteLine("Press Enter to disconnect the annuaire.");
      Console.ReadLine();
   }
}
```

B.12.2 Publication WEB Service

Pour pouvoir exécuter un service Web, il faut un serveur Web. Pour cela, nous allons utiliser Web Matrix qui permet d'activer un serveur Web interne. Le tutorial de Web Matrix (http ://rangiroa.essi.fr/cours/tutorial-webmatrix) explique en détail la marche à suivre. Voici un cours résumé :

1. créer la classe `AnnuaireWebService` dans le fichier `AnnuaireWebService.asmx` (File, New File) puis 'General', 'XML Web Service', et finir de remplir la boîte de dialogue.
2. un canevas de service web est crée. Il vous reste à compléter le code pour qu'il accède à l'annuaire. Le code est presque le même que pour le servant dans le cadre du 'Remoting'. Seules les méthode devant être accédées à distance sont précédées de l'attribut `[WebMethod]`.

```
<%@ WebService language="C#" class="AnnuaireWebService" Debug="true" %>

using System;
using System.Web.Services;
using System.Xml.Serialization;

public class AnnuaireWebService : System.Web.Services.WebService, IAnnuaireService {
   private IAnnuaireService service;
```

```
    public AnnuaireWebService() {
       service=new AnnuaireService();
    }

    [WebMethod]
    public CarteDeVisite Rechercher(string nom) {
       return service.Rechercher(nom);
    }
 }
```

3. Copier la dll dans un répertoire `bin`.
4. Sauvegarder et exécuter le service web (appuyer sur la touche F5). Une boite de dialogue vous demande le port d'activation du Web Service (80 par défaut)
5. Un navigateur s'ouvre avec une page vous permettant d'interroger le service web.

```
http://localhost:80/AnnuaireWebService.asmx pour avoir accès au service web
http://localhost:8086/AnnuaireWebService.asmx?WSDL pour avoir accès à la description WSDL
```

B.13 Utilisation de ces services

B.13.1 Par un client lourd

Dans le cas de .Net Remoting, le client n'a besoin de connaître que l'interface du service. Dans le cas de la consommation du service web il est nécessaire d'obtenir un proxy en C# sur ce service web. Ceci peut être fait par l'utilisation de l'utilitaire WSDL :

```
wsdl.exe /out:proxy_webservice.cs /n:proxy http://machine:port/URL?WSDL
REM la directive /n permet de préciser l'espace de nom
```

Voici de manière comparative les différentes manières de construire le client :

Client local

```
class Client {
   static void Main (string[] args) {
      // creation de l'objet
      IAnnuaireService annuaire = new  AnnuaireService ();

      CarteDeVisite c = annuaire.Recherche ("auteur");
   }
}
```

Client .NET Remoting

```
class Client {
   static void Main (string[] args) {
      // creation d'une connection TCP
      TcpChannel channel = new TcpChannel ();
      ChannelServices.RegisterChannel (channel, false);
```

```
      // creation du proxy
      IAnnuaireService annuaire = (IAnnuaireService) Activator.GetObject
            (typeof (IAnnuaireService),
            "tcp://127.0.0.1:7000/TcpService");

      CarteDeVisite c = annuaire.Recherche ("auteur");
   }
}
```

Client service web

```
class Client {
   static void Main (string[] args) {
      // creation du proxy
      proxy.AnnuaireWebService annuaire = new proxy.AnnuaireWebService ();

      proxy.cartedevisite c  = annuaire.Recherche ("auteur");
   }
}
```

B.13.2 Par un client léger

A part l'architecture qui n'avait pas été respectée, nous l'avons déjà fait. Pour respecter celle-ci, nous avons deux possibilités.

Solution 1 Créer un client léger qui appelle le service précédemment créé, via le proxy généré précédemment. Voici par exemple, la partie code d'une telle page.

Pour pouvoir mettre au point le programme, l'entête de la page (onglet 'all') doit contenir la directive debug ainsi que le chargement du proxy généré :

```
<%@ Page Language="C#" Debug="true" %>
<%@ assembly Src="proxy_annuaireWS.cs" %>
```

La partie code est la suivante :

```
   void Button_Click(object sender, EventArgs e) {
      // création du proxy et appel du service web
      proxy.AnnuaireWebService annuaire = new proxy.AnnuaireWebService ();
      proxy.cartedevisite carte = annuaire.Rechercher (nom.Text);

      // mise à jour des 'Label' contenu dans la page
      LabelPrenomValeur.Text = carte.Prenom;
      LabelEmailValeur.Text = carte.Email;

      // rendre visible les 'Label" de la page
      LabelPrenom.Visible = true;
      LabelPrenomValeur.Visible = true;
      LabelEmail.Visible = true;
      LabelEmailValeur.Visible = true;
}
```

La partie 'html' de la page contient un `TextBox` pour saisir le nom et quatre `Label` : deux pour afficher le texte *"prénom"* et *"email"* et deux autres pour afficher les valeurs associées.

```
...
<asp:TextBox id="nom" runat="server"></asp:TextBox>
...
<asp:Label id="LabelPrenom" runat="server" text="Prénom : " \
              visible="False"></asp:Label>
<asp:Label id="LabelPrenomValeur" runat="server" text="Label" \
              visible="False"></asp:Label>
...
<asp:Label id="LabelEmail" runat="server" text="Email : " \
              visible="False"></asp:Label>
<asp:Label id="LabelEmailValeur" runat="server" text="Label" \
              visible="False"></asp:Label>
```

Solution 2 Créer un client léger qui utilise directement la *dll* précédemment produite. Ceci peut être fait très facilement en déplaçant la *dll* dans un répertoire `bin`, puis en utilisant la directive 'import' pour charger le 'namespace' du TP :

```
<%@ import Namespace="TP" %>
```

Attention : si votre *dll* utilise des modules, il faut aussi déplacer dans ce répertoire les différents modules.

Le code est le suivant (très voisin de l'étape précédente... en l'absence de la directive import, il est aussi possible d'utiliser les noms de classe complet) :

```
void Button_Click(object sender, EventArgs e) {
   IAnnuaireService annuaire = new AnnuaireService ();
   CarteDeVisite carte = annuaire.Rechercher (nom.Text);

   LabelPrenomValeur.Text = carte.Prenom;
   LabelEmailValeur.Text = carte.Email;

   LabelPrenom.Visible = true;
   LabelPrenomValeur.Visible = true;
   LabelEmail.Visible = true;
   LabelEmailValeur.Visible = true;
}
```

B.14 Pour poursuivre le TP

- Utiliser le déploiement et la gestion de version afin de pouvoir faire évoluer le code et gérer de multiples versions de chaque composants.
- Nous avons construit très rapidement cet exemple, en particulier les cas d'erreur ne sont pas traités. Il faudrait en particulier quand des champs des cartes de visite sont vides.

- Il est possible de construire d'autres classes d'accès aux données afin d'utiliser une sauvegarde dans un fichier XML et non plus une base de données.
- Créer d'autres tables afin d'avoir par exemple pour chaque personne son compte bancaire afin d'effectuer des transfert entre compte de manière transactionnelle.
- Mettre en place un mécanisme d'authentification afin que seule les personnes autorisées puissent accéder au données sensible.