SPRING IOC POUR .NET

serge.tahe@istia.univ-angers.fr, avril 2005

Objectifs du document :

- découvrir les possibilités de configuration et d'intégration du framework Spring (http://www.springframework.org) pour une application .Net (http://www.springframework.net/)
- définir et utiliser la notion d'IoC (Inversion of Control), également appelée injection de dépendance (Dependency Injection)

Les idées exprimées dans ce document ont pour origine un livre lu au cours de l'été 2004, un magnifique travail de **Rod Johnson : J2EE Development without EJB** aux éditions **Wrox**.

---

1 Introduction

Un document **Spring IoC pour Java** a déjà été écrit [http://tahe.developpez.com/java/springioc]. Le document présent reprend le même contenu mais l'adapte à la plate-forme .Net. Le langage .Net utilisé pour les exemples est VB.Net.

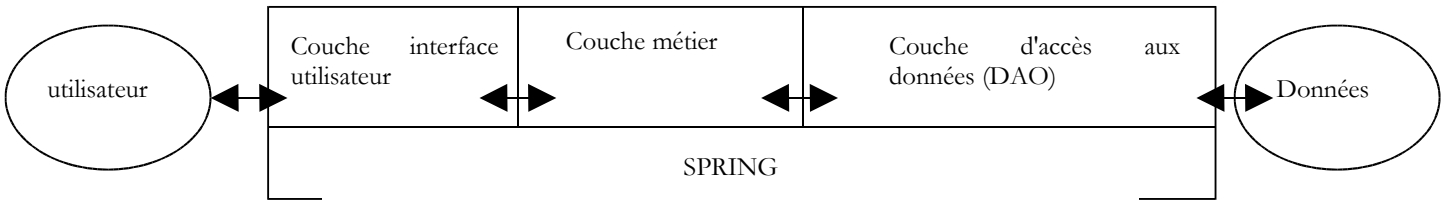
Spring.net n'en est qu'à la version 0.6 RC3 (avril 2005). On trouvera en annexe où le trouver. **Spring.net** étant en cours de développement, à ce jour seules certaines des caractéristiques de Spring/Java ont été portées, mais ce sont celles qui forment le coeur de Spring : l'**inversion de contrôle** (IoC) et la **programmation orientée aspects** (AOP). Nous ne présentons ici que la notion d'inversion de contrôle.

Il est possible que certains détails techniques présentés ici évoluent dans le futur. L'intérêt de Spring ne réside pas dans ces détails techniques mais bien plutôt dans sa philosophie. C'est celle-ci que le lecteur est invité à découvrir.

La partie pratique du document propose plusieurs exemples. Ils utilisent les classes de test **Nunit**. Celles-ci sont l'équivalent pour .Net des classes de test **Junit** de Java. En annexe, on trouvera où trouver [Nunit] et comment l'installer.

2 Configurer une application avec Spring

Considérons une application 3-tier classique :



On supposera que l'accès la couche DAO est contrôlé par une interface [IArticlesDao] :

```
....
Namespace istia.st.articles.dao

  Public Interface IArticlesDao
    ' liste de tous les articles
    Function getAllArticles() As IList
    ' ajoute un article
    Function ajouteArticle(ByVal unArticle As Article) As Integer
    ' supprime un article
    Function supprimeArticle(ByVal idArticle As Integer) As Integer
    ' modifie un article
    Function modifieArticle(ByVal unArticle As Article) As Integer
    ' recherche un article
    Function getArticleById(ByVal idArticle As Integer) As Article
    ' supprime tous les articles
    Sub clearAllArticles()
    ' insère des articles au sein d'une transaction
    Sub doInsertionsInTransaction(ByVal articles As Article())
    ' change le stock d'u article
    Function changerStockArticle(ByVal idArticle As Integer, ByVal mouvement As Integer) As Integer
  End Interface
End Namespace
```

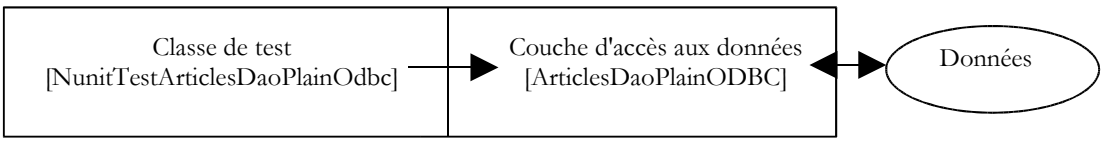
Dans la couche d'accès aux données ou couche DAO (Data Access Object), il est fréquent de travailler avec un SGBD. Considérons le cas où celui-ci est accédé via un pilote ODBC. Le squelette d'une classe accédant à cette source ODBC pourrait être le suivant :

```
NameSpace istia.st.articles.dao

Imports System.Data.Odbc
...
  Public Class ArticlesDaoPlainODBC
    Implements istia.st.articles.dao.IArticlesDao

    ' champs privés
    Private connexion As OdbcConnection = Nothing
    Private DSN As String

    Public Sub New(ByVal DSN As String, ByVal user As String, ByVal passwd As String)
      'on récupère le nom de la source ODBC
      Me.DSN = DSN
      ' on crée la chaîne de connexion
      Dim connectString As String = String.Format("DSN={0};UID={1};PWD={2}", DSN, user, passwd)
      'on instancie la connexion
      connexion = New OdbcConnection(connectString)
    End Sub

....
  End Class
End NameSpace
```

Pour faire une opération sur la source ODBC, toute méthode a besoin d'un objet [OdbcConnection] qui représente la connexion à la base par laquelle vont transiter les échanges entre celle-ci et l'application. Pour construire cet objet, on a besoin de trois informations :

`DSN As String`	le nom de la source ODBC
`user As String`	l'identité sous laquelle on crée la connexion
`passwd As String`	le mot de passe associée à cette identité

Notre classe [ArticlesDaoPlainODBC] obtient ces informations via l'agent extérieur qui instancie un membre de la classe. On peut se demander comment celui-ci obtient les trois informations nécessaires à l'instanciation de la classe [ArticlesDaoPlainODBC]. Prenons un exemple. Supposons qu'on veuille écrire une classe de test de la couche [Dao]. On aurait l'architecture suivante :



Le squelette d'une classe de test Nunit [http://www.nunit.org/] pourrait être le suivant :

```
Imports System
Imports System.Collections
Imports NUnit.Framework
Imports istia.st.articles.dao
Imports ArticlesDaoSqlmap = istia.st.articles.dao.ArticlesDaoSqlMap
Imports Article = istia.st.articles.domain.Article
Imports System.Threading

  <TestFixture()> _
  Public Class NunitTestArticlesDaoPlainOdbc

    ' l'objet à tester
    Private articlesDao As IArticlesDao

    <SetUp()> _
    Public Sub init()
      ' on crée une instance de l'objet à tester
      articlesDao = New ArticlesDaoPlainODBC("odbc-firebird-articles", "SYSDBA", "masterkey")
    End Sub

    <Test()> _
    Public Sub testGetAllArticles()
      ' vérification visuelle
      listArticles()
    End Sub

    ' listing écran
    Private Sub listArticles()
      Dim articles As IList = articlesDao.getAllArticles
      For i As Integer = 0 To articles.Count - 1
```

```
      Console.WriteLine(CType(articles(i), Article).ToString)
    Next
  End Sub


 End Class
```

L'environnement de test Nunit est un portage vers la plate-forme .Net de l'environnement JUnit qui existe pour la plate-forme Java. Dans la classe ci-dessus, la méthode ayant l'attribut <SetUp()> est exécutée avant chaque méthode de test. Celle qui a l'attribut <TearDown()> est elle, exécutée après chaque test. Il n'y en a pas dans l'exemple ci-dessus. Ici, on voit que la méthode [init] qui a l'attribut <SetUp()>, instancie un objet [ArticlesDaoPlainODBC] en lui passant en "dur" les trois informations dont le constructeur de cet objet a besoin.

Notre classe de test est à la merci d'un changement d'une des informations codées en "dur". Il serait préférable que celles-ci soient inscrites dans un fichier de configuration, afin d'éviter des recompilations inutiles lorsqu'elles changent. La solution habituelle pour configurer une application est l'utilisation d'un fichier où l'on va retrouver toutes les informations susceptibles de changer dans le temps. Il existe une grande variété de fichiers de configuration. La tendance actuelle est d'utiliser des fichiers XML. C'est l'option prise par Spring. Le fichier configurant un objet [ArticlesDaoPlainODBC] pourrait être le suivant :

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<!DOCTYPE objects PUBLIC "-//SPRING//DTD OBJECT//EN"
"http://www.springframework.net/dtd/spring-objects.dtd">

<objects>
  <!-- la classe d'implémentation de l'interface IArticlesDao -->
  <description>Gestion d'une table d'articles</description>
  <object id="articlesdao" type="istia.st.articles.dao.ArticlesDaoPlainODBC, articlesdao">
    <constructor-arg index="0">
      <value>odbc-firebird-articles</value>
    </constructor-arg>
    <constructor-arg index="1">
      <value>SYSDBA</value>
    </constructor-arg>
    <constructor-arg index="2">
      <value>masterkey</value>
    </constructor-arg>
  </object>
</objects>
```

Le fichier de configuration de Spring décrit des objets à instancier. Il ne dit pas, en général, à quel moment ils seront instanciés. Le moment de leur instanciation est alors décidé par le code qui exploite ce fichier. Les objets décrits dans un fichier de configuration Spring peuvent être instanciés et initialisés de deux façons différentes :

- en indiquant, comme ci-dessus, les paramètres à passer au constructeur de l'objet
- en fournissant des valeurs aux propriétés de l'objet (Property). Dans ce cas, l'objet doit posséder un constructeur par défaut que Spring utilisera pour l'instanciation.

Les objets qui, dans une application, ont pour rôle de rendre un service sont souvent créés en un seul exemplaire. On les appelle des **singletons**. Ainsi dans notre exemple d'application multi-tier présentée au début de ce document, l'accès à la base des articles sera assuré par un unique exemplaire de la classe [ArticlesDaoPlainODBC]. Pour une application web, ces objets de service servent plusieurs clients à la fois. On ne crée pas un objet de service par client.

Le fichier de configuration Spring ci-dessus permet de créer un objet **service** unique de type [ArticlesDaoPlainODBC] dans un paquetage nommé [istia.st.articles.dao]. Les trois informations nécessaires au constructeur de cet objet sont définies à l'intérieur d'une balise **<object>...</object>**. On aura autant de telles balises **<object>** que de singletons à construire.

Détaillons la configuration :

```
<objects>
...
</objects>
```

<objects> est la balise racine d'un fichier de configuration Spring. Elle annonce la description des objets singleton à instancier.

```
  <description>Gestion d'une table d'articles</description>
```

La balise **<description>** est facultative. On peut l'utiliser par exemple pour décrire le rôle du fichier de configuration.

```
<object id="articlesdao" type="istia.st.articles.dao.ArticlesDaoPlainODBC, articlesdao">
...
</object>
```

La balise **<object>** sert à décrire un objet à instancier. Elle a ici deux attributs :

- **name** : identifiant de l'objet. C'est via ce nom, que le code extérieur référencera l'objet.
- **class** : de la forme "nom de classe, nom d'assemblage". La première information est le nom complet de la classe à instancier. La seconde, le nom de la DLL qui contient cette classe. Dans notre exemple, la classe se trouve dans un fichier nommé [articlesdao.dll]

Le contenu de la balise **<object>** sert à décrire le mode d'instanciation de l'objet :

```
 <object id="articlesdao" type="istia.st.articles.dao.ArticlesDaoPlainODBC, articlesdao">
   <constructor-arg index="0">
     <value>odbc-firebird-articles</value>
   </constructor-arg>
   <constructor-arg index="1">
     <value>SYSDBA</value>
   </constructor-arg>
   <constructor-arg index="2">
     <value>masterkey</value>
   </constructor-arg>
 </object>
```

Rappelons la signature du constructeur de la classe [ArticlesDaoPlainODBC] :

```
   Public Sub New(ByVal DSN As String, ByVal user As String, ByVal passwd As String)
```

L'objet Spring [articlesdao] sera instancié par le constructeur ci-dessus avec les trois informations du fichier de configuration : [odbc-firebird-articles, SYSDBA, masterkey].

A quel moment va intervenir la construction des objets définis dans le fichier Spring ? On trouve dans toute application, une méthode assurée d'être la première à s'exécuter. C'est généralement dans celle-ci que la construction des singletons est demandée. L'initialisation d'une application peut être dévolue à la méthode **main** de cette même application si elle en a une. Pour une application ASP.NET, ce peut-être la méthode [**Application_Start**] du fichier [**global.asax**]. Pour notre classe de test [Nunit], l'initialisation de l'application a lieu dans la méthode associée à l'attribut **<Setup()>**.

Comment utiliser le fichier de configuration ci-dessus dans notre classe [Nunit] ? Voici un exemple :

```
Imports System
Imports System.Collections
Imports NUnit.Framework
Imports istia.st.articles.dao
Imports ArticlesDaoSqlmap = istia.st.articles.dao.ArticlesDaoSqlMap
Imports Article = istia.st.articles.domain.Article
Imports System.Threading
Imports Spring.Objects.Factory.Xml
Imports System.IO

 <TestFixture()> _
 Public Class NunitSpringTestArticlesDaoPlainOdbc

   ' l'objet à tester
   Private articlesDao As IArticlesDao

   <SetUp()> _
   Public Sub init()
     ' on récupère une instance du fabricant d'objets Spring
     Dim factory As XmlObjectFactory = New XmlObjectFactory(New FileStream("spring-config-plainodbc.xml",
FileMode.Open))
     ' on demande l'instanciation de l'objet articles dao
     articlesDao = CType(factory.GetObject("articlesdao"), IArticlesDao)
   End Sub

   <Test()> _
   Public Sub testGetAllArticles()
     ' vérification visuelle
     listArticles()
   End Sub

   ' listing écran
   Private Sub listArticles()
     Dim articles As IList = articlesDao.getAllArticles
     For i As Integer = 0 To articles.Count - 1
       Console.WriteLine(CType(articles(i), Article).ToString)
     Next
   End Sub
 End Class
```

Commentaires :

- pour instancier les objets du fichier de configuration de Spring, on passe par un objet de type [XmlObjectFactory]. C'est un objet de type "Factory", c.a.d. un objet qui sert à créer d'autres objets (Factory=usine, fabrique). Spring dispose de plusieurs types de "Factory" selon le fichier de configuration utilisé. Ici, celui-ci est un fichier XML et donc on utilise un type [XmlObjectFactory].
- de façon assez logique, un objet de type [XmlObjectFactory] a besoin du nom du fichier XML de configuration, ici [spring-config-plainodbc.xml]. Très exactement, le type [XmlObjectFactory] s'instancie avec un flux de lecture créé à partir du fichier XML dont on donne le nom.
- une fois l'objet de type [XmlObjectFactory] créé, un objet du fichier de configuration est obtenu par **[XmlObjectFactory].getObject("identifiant")** où "identifiant" est l'attribut [id] d'un des objets du fichier de configuration.
- si l'objet demandé n'a pas déjà été instancié, Spring l'instancie à l'aide des informations de son fichier de configuration et en rend une référence au programme appelant. Si l'objet a déjà été instancié, Spring se contente de rendre la référence de l'objet déjà existant. C'est le principe du singleton.
- on remarquera que la classe de test [Nunit] ne connaît pas le nom de la classe d'accès aux données. Ce nom est dans le fichier de configuration. La classe de test se contente de demander un objet implémentant l'interface [IArticlesDao] :

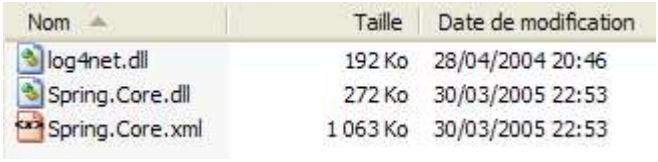
```
    ' l'objet à tester
    Private articlesDao As IArticlesDao
```

```
    <SetUp()> _
    Public Sub init()
...
      articlesDao = CType(factory.GetObject("articlesdao"), IArticlesDao)
    End Sub
```

C'est là tout l'intérêt de Spring. Si on change de classe d'implémentation, notre classe de test n'aura pas à être changée. On modifiera simplement le fichier de configuration de Spring. La classe de test elle, se contente de travailler avec une interface et non avec une classe.

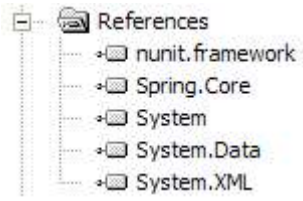
Terminons cette présentation par quelques points pratiques.

Quand on écrit "Spring va instancier...", que veut-on dire exactement ? Pour la plate-forme .Net, Spring est contenu dans trois fichiers :



Pour un projet .Net construit avec Visual Studio et qui doit utiliser Spring, on procèdera comme suit :

- les trois fichiers ci-dessus seront placés dans le dossier [bin] du projet
- [Spring.Core.dll] doit faire partie des références du projet :



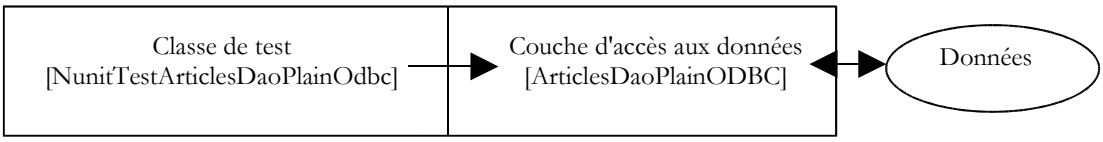
- que les classes faisant appel à Spring, devront importer certains espaces de noms dont souvent le suivant :

```
Imports Spring.Objects.Factory.Xml
```

Autre point pratique : où place-t-on le fichier de configuration de Spring ? Il y a plusieurs endroits possibles. L'un d'eux est le dossier [bin] du projet. C'est là qu'a été placé le fichier [spring-config-plainodbc.xml] de l'exemple étudié.

3 Inversion de contrôle IoC

Attardons-nous maintenant sur la notion d'**inversion de contrôle** (IoC, Inversion of Control) utilisée par Spring pour configurer les applications. Pour illustrer ce concept, revenons à l'architecture de notre application de test précédente :

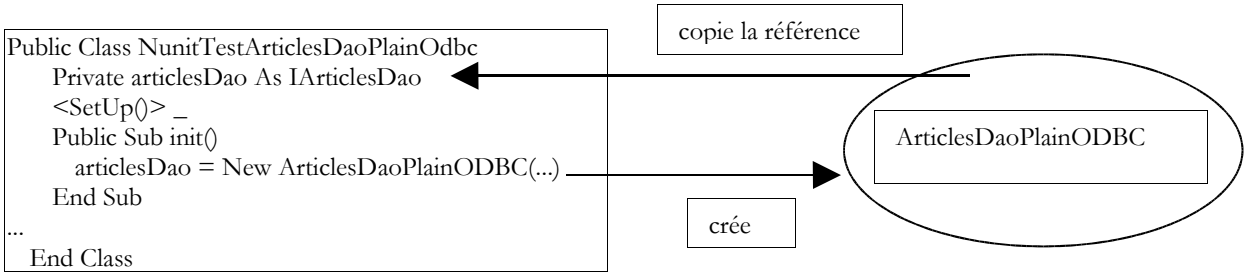


Pour accéder aux données du SGBD, la classe de test doit utiliser les services d'un objet implémentant l'interface [IArticlesDao], par exemple un objet de type [ArticlesDaoPlainODBC]. Nous avons étudié deux solutions possibles pour instancier un tel objet :

- dans la première solution, la classe de test demandait elle-même l'instanciation d'un objet de type [ArticlesDaoPlainODBC] :

```
  <TestFixture()> _
  Public Class NunitTestArticlesDaoPlainOdbc

    ' l'objet à tester
    Private articlesDao As IArticlesDao

    <SetUp()> _
    Public Sub init()
      ' on crée une instance de l'objet à tester
      articlesDao = New ArticlesDaoPlainODBC("odbc-firebird-articles", "SYSDBA", "masterkey")
    End Sub
...
  End Class
```

On a une **dépendance en dur dans le code sur le nom de classe**. Si la classe d'implémentation de l'interface [IArticlesDao] venait à changer, le code de la méthode [init] **devrait être modifié**. On a les relations suivantes entre les objets :

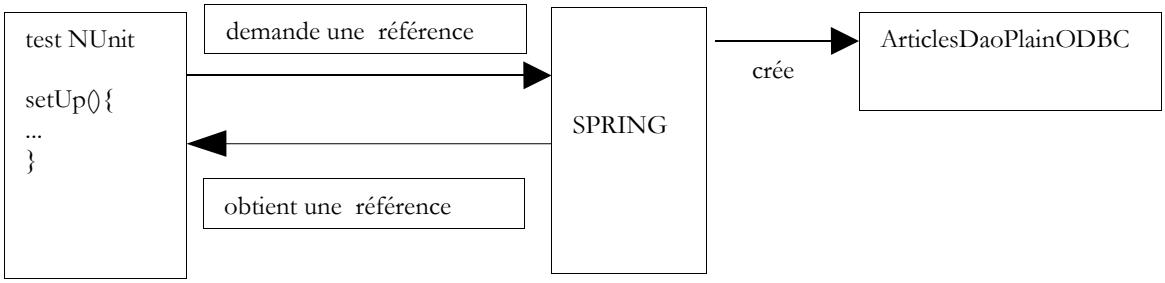


La classe [NunitTestArticlesDaoPlainOdbc] prend elle-même l'initiative de la création de l'objet [ArticlesDaoPlainODBC] dont elle a besoin. Pour en revenir au terme "inversion de contrôle", on dira que c'est elle qui a le "contrôle" pour créer l'objet dont elle a besoin.

- la seconde solution procède différemment. La classs de test est devenue la suivante :

```
  <TestFixture()> _
  Public Class NunitSpringTestArticlesDaoPlainOdbc

    ' l'objet à tester
    Private articlesDao As IArticlesDao

    <SetUp()> _
    Public Sub init()
       ' on récupère une instance du fabricant d'objets Spring
      Dim factory As XmlObjectFactory = New XmlObjectFactory(New FileStream("spring-config-plainodbc.xml",
FileMode.Open))
       ' on demande l'instanciation de l'objet articles dao
      articlesDao = CType(factory.GetObject("articlesdao"), IArticlesDao)
    End Sub
     ...
  End Class
```

Ce mécanisme pourrait être schématisé comme suit :

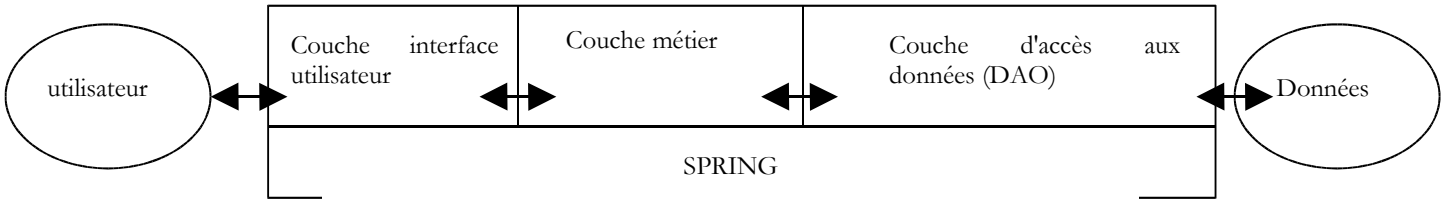


Ici, la classe de test ne prend pas l'initiative de demander la création d'un objet [ArticlesDaoPlainODBC]. Elle se contente de demander à Spring, une référence sur un tel objet. Si l'objet existe, Spring en rend alors une référence. S'il n'existe pas, il le crée. La classe de test a perdu le contrôle de la création de l'objet [ArticlesDaoPlainODBC]. Elle demande simplement une référence sur cet objet. Cette demande va ici forcer Spring à créer l'objet. Mais on pourrait imaginer, dans un autre contexte, que l'objet demandé a déjà été créé à la demande de l'application. Spring ne recrée alors pas l'objet mais rend une référence sur l'objet déjà existant (singleton). La notion d'Inversion de contrôle (IoC) signifie ici :

- que l'application ne prend jamais l'initiative de créer les singletons dont elle a besoin. Elle en demande simplement des références.
- c'est Spring qui prend la décision de créer un singleton lors de la première demande de référence sur celui-ci

4 Injection de dépendance

L'injection de dépendance peut être considérée comme une conséquence de l'inversion de contrôle. Nous allons l'illustrer sur un nouvel exemple. Considérons l'application web 3-tier suivante :



On supposera que l'accès la couche DAO est contrôlé par l'interface [IArticlesDao] étudiée précédemment et que l'ensemble de l'application est une application web d'achats d'articles sur le web. Les articles sont ceux gérés par la couche [Dao]. On suppose que l'accès à la couche métier est contrôlé par l'interface suivante :

```
Public Interface IArticlesDomain
  ' acheter un panier d'articles
  Sub acheter(ByVal panier As Panier)
  ' obtenir la liste des articles
  Function getAllArticles() As IList
  ' obtenir un article particulier
  Function getArticleById(ByVal idArticle As Integer) As Article
  ' pour enregistrer les erreurs
  ReadOnly Property erreurs() As ArrayList
End Interface
```

On ne s'attardera pas sur la signification des différentes méthodes. On notera simplement que la méthode [getAllArticles] qui doit obtenir la liste de tous les articles en vente, a besoin d'avoir accès aux données. Pour les obtenir, elle doit s'adresser à l'interface [IArticlesDao]. Le squelette d'une classe d'implémentation de l'interface [IArticlesDomain] pourrait ressembler à ceci :

```
Imports istia.st.articles.dao
...
Namespace istia.st.articles.domain
  Public Class AchatsArticles
    Implements IArticlesDomain

    'champs privés
    Private _articlesDao As IArticlesDao
    Private _erreurs As ArrayList

    ' constructeur
    Public Sub New(ByVal articlesDao As IArticlesDao)
      _articlesDao = articlesDao
    End Sub
...
  End Class
End Namespace
```

Nous l'avons dit, certaines méthodes de l'interface de la couche métier ont besoin de demander des données à la couche [Dao]. Notre classe d'implémentation [AchatsArticles] de l'interface [IArticlesDomain] a donc besoin d'une référence sur une implémentation de l'interface [IArticlesDao]. Ci-dessus, c'est le champ privé [_articlesDao] qui est cette référence. Celle-ci est fournie au moment de la construction d'un objet [AchatsArticles].

Supposons que la classe [AchatsArticles] ait été écrite et qu'on veuille la tester avec un test de type [Nunit]. Construisons celui-cii pour qu'il utilise un fichier de configuration Spring :

```
...
Imports Spring.Objects.Factory.Xml
Imports System.IO
...
  <TestFixture()> _
  Public Class NunitSpringTestArticlesDomain

    ' l'objet à tester
    Private articlesDomain As IArticlesDomain

    <SetUp()> _
    Public Sub init()
       ' on récupère une instance du fabricant d'objets Spring
      Dim factory As XmlObjectFactory = New XmlObjectFactory(New FileStream("spring-config-domain.xml",
FileMode.Open))
       ' on demande l'instanciation de l'objet articles dao
      articlesDomain = CType(factory.GetObject("articlesdomain"), IArticlesDao)
    End Sub
...
  End Class
```

Quel serait le contenu du fichier de configuration [spring-config-domain.xml] de Spring ? Il pourrait être le suivant :

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<!DOCTYPE objects PUBLIC "-//SPRING//DTD OBJECT//EN"
"http://www.springframework.net/dtd/spring-objects.dtd">

<objects>
  <description>Gestion d'une table d'articles</description>

  <!-- la classe d'implémentation de l'interface IArticlesDao -->
  <object id="articlesdao" type="istia.st.articles.dao.ArticlesDaoPlainODBC, articlesdao">
    <constructor-arg index="0">
      <value>odbc-firebird-articles</value>
    </constructor-arg>
    <constructor-arg index="1">
      <value>SYSDBA</value>
    </constructor-arg>
    <constructor-arg index="2">
      <value>masterkey</value>
    </constructor-arg>
  </object>

  <!-- la classe d'implémentation de l'interface IArticlesDomain -->
  <object id="articlesdomain" type="istia.st.articles.domain.AchatsArticles, articlesdomain">
    <constructor-arg index="0">
      <ref object="articlesdao" />
    </constructor-arg>
  </object>
</objects>
```

Ce fichier est celui déjà utilisé pour instancier le singleton de type [IArticlesDao] de la couche [Dao] auquel on a ajouté le code pour instancier le singleton de type [IArticlesDomain] de la couche métier. Comment celui-ci sera-t-il construit ?

- un code externe demande à Spring une référence sur le singleton nommé " articlesdomain " dans le fichier de configuration. C'est le cas de la méthode [init] de notre classe de test :

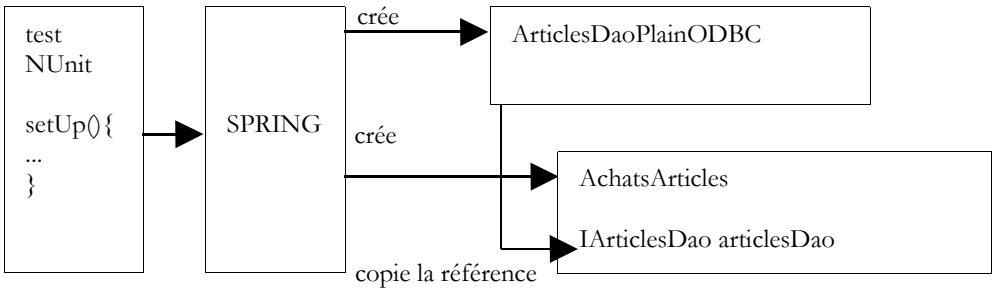
```
    <SetUp()> _
    Public Sub init()
       ' on récupère une instance du fabricant d'objets Spring
      Dim factory As XmlObjectFactory = New XmlObjectFactory(New FileStream("spring-config-domain.xml",
FileMode.Open))
       ' on demande l'instanciation de l'objet articles dao
      articlesDomain = CType(factory.GetObject("articlesdomain"), IArticlesDao)
    End Sub
```

- Spring trouve dans son fichier de configuration, la définition du singleton en question. Il découvre que pour l'instancier, il a besoin d'un autre singleton appelé " articlesdao " :

```
  <object name="articlesdomain" class="istia.st.articles.domain.AchatsArticles, articlesdomain">
    <constructor-arg index="0">
      <ref object="articlesdao" />
    </constructor-arg>
  </object>
```

- Spring va alors instancier le singleton " articlesdao ". Nous avons déjà expliqué comment il le faisait.
- Ceci fait, il peut instancier le singleton " articlesdomain " et en rendre une référence au code qui l'a demandée.

Ce mécanisme pourrait être schématisé comme suit :



On voit que Spring a géré la dépendance qu'avait le singleton " articlesdomain " vis à vis du singleton " articlesdao ". Afin d'utiliser Spring de cette façon, il nous a fallu construire une classe avec un constructeur qui accepte comme argument le singleton dépendant :

```
Imports istia.st.articles.dao
...
Namespace istia.st.articles.domain
  Public Class AchatsArticles
    Implements IArticlesDomain

    'champs privés
    Private _articlesDao As IArticlesDao

    ' constructeur
    Public Sub New(ByVal articlesDao As IArticlesDao)
      _articlesDao = articlesDao
    End Sub
...
  End Class
End Namespace
```

Le terme " injection de dépendance " recouvre à la fois :

- la façon particulière de construire les classes à instancier en fonction de leurs dépendances
- la façon qu'a Spring de gérer ces dépendances au moment de l'instanciation de ces classes

5 Spring IoC par la pratique

Nous abordons maintenant avec des exemples, la mise en pratique de ce que nous avons vu précédemment.

5.1 Exemple 1

La classe [Personne.vb] est la suivante :

```
Namespace istia.st.springioc.demos
  Public Class Personne

    ' champs privés
    Private _nom As String
    Private _age As Integer

    ' constructeur par défaut
    Public Sub New()
    End Sub

    ' propriétés associées aux champs privés
    Public Property nom() As String
      Get
        Return _nom
```

```
    End Get
    Set(ByVal Value As String)
      _nom = Value
    End Set
  End Property

  Public Property age() As Integer
    Get
      Return _age
    End Get
    Set(ByVal Value As Integer)
      _age = Value
    End Set
  End Property

  ' chaîne d'identité
  Public Overrides Function tostring() As String
    Return String.Format("[{0},{1}]", nom, age)
  End Function

  ' méthode init
  Public Sub init()
    Console.WriteLine("init personne {0}", Me.ToString)
  End Sub

  ' méthode close
  Public Sub close()
    Console.WriteLine("destroy personne {0}", Me.ToString)
  End Sub

 End Class
End Namespace
```

La classe présente :

- deux champs privés **nom** et **age** accessibles via des propriétés
- une méthode **toString** pour récupérer la valeur de l'objet [Personne] sous la forme d'une chaîne de caractères
- une méthode **init** qui sera appelée par Spring à la création de l'objet, une méthode **close** qui sera appelée à la destruction de l'objet

Pour créer des objets de type [Personne], nous utiliserons le fichier Spring [spring-config-1.xml] suivant :

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<!DOCTYPE objects PUBLIC "-//SPRING//DTD OBJECT//EN"
"http://www.springframework.net/dtd/spring-objects.dtd">

<objects>
  <object id="personne1" type="istia.st.springioc.demos.Personne, demo1" init-method="init"
    destroy-method="close">
    <property name="nom">
      <value>Simon</value>
    </property>
    <property name="age">
      <value>40</value>
    </property>
  </object>
  <object id="personne2" type="istia.st.springioc.demos.Personne, demo1" init-method="init"
    destroy-method="close">
    <property name="nom">
      <value>Brigitte</value>
    </property>
    <property name="age">
      <value>20</value>
    </property>
  </object>
</objects>
```

Ce fichier

- définit deux objets de clés respectives "personne1" et "personne2" de type [Personne]
- il initialise les propriétés [nom, age] de chaque personne
- il définit les méthodes à appeler lors de la construction initiale de l'objet [init-method] et lors de la destruction de l'objet [destroy-method]

Pour nos tests, nous utiliserons des classes de test NUnit. La première sera la suivante :

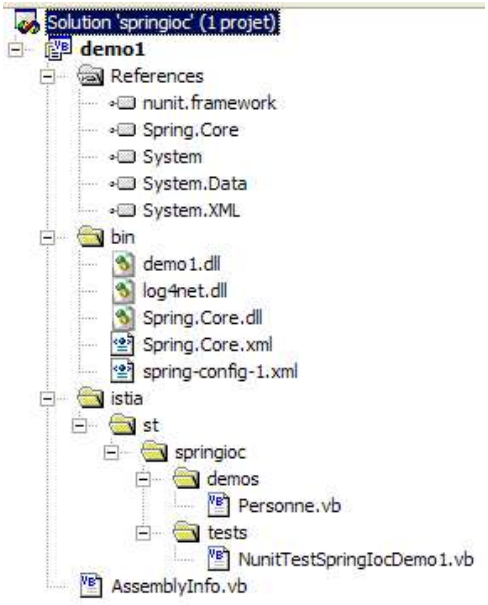
```
Imports System
Imports Spring.Objects.Factory.Xml
Imports System.IO
Imports NUnit.Framework
Imports istia.st.springioc.demos
```

```
Namespace istia.st.springioc.tests

  <TestFixture()> _
  Public Class NunitTestSpringIocDemo1
    ' l'objet à tester
    Private factory As XmlObjectFactory

    <SetUp()> _
    Public Sub init()
      ' on crée une instance de factory
      factory = New XmlObjectFactory(New FileStream("spring-config-1.xml", FileMode.Open))
      ' log
      Console.WriteLine("setup test")
    End Sub

    <Test()> _
    Public Sub demo()
      ' récupération par leur clé des objets [Personne] du fichier Spring
      Dim personne1 As Personne = CType(factory.GetObject("personne1"), Personne)
      Console.WriteLine("personne1=" + personne1.ToString())
      Dim personne2 As Personne = CType(factory.GetObject("personne2"), Personne)
      Console.WriteLine("personne2=" + personne2.ToString())
      personne2 = CType(factory.GetObject("personne2"), Personne)
      Console.WriteLine("personne2=" + personne2.ToString())
    End Sub

    <TearDown()> _
    Public Sub destroy()
      ' on détruit les singletons
      factory.Dispose()
      ' on libère la ressource factory
      factory = Nothing
      ' suivi
      Console.WriteLine("teardown test")
    End Sub

  End Class
End Namespace
```

Commentaires :

- pour obtenir les objets définis dans le fichier [spring-config-1.xml], nous utilisons un objet de type [XmlObjectFactory]. Il existe d'autres types de " factory " permettant d'accéder aux singletons du fichier de configuration. L'objet [XmlObjectFactory] est obtenu dans la méthode d'attribut [SetUp] de la classe de test et mémorisé dans une variable privée. Rappelons que la méthode dotée de l'attribut <Setup()> est exécutée avant chaque test.
- le fichier [spring-config-1.xml] sera placé dans le dossier [bin] de l'application
- Spring peut utiliser des fichiers de configuration ayant divers formats. L'objet [XmlObjectFactory] permet d'analyser un fichier de configuration au format XML.
- l'exploitation d'un fichier Spring donne un objet de type [XmlObjectFactory], ici l'objet **factory**. Avec cet objet, un singleton identifié par la clé **C**, s'obtient par **factory.getObject(C)**.
- la méthode [demo] demande et affiche la valeur des singletons de clé "personne1" et "personne2".

La structure du projet Visual Studio est la suivante :

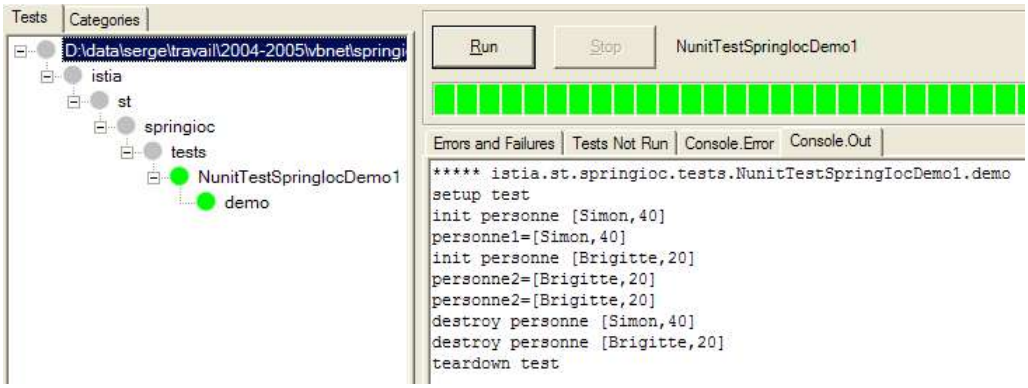


Commentaires :

- le projet VS s'appelle [demo1]
- le dossier [istia] contient les codes source. Les codes compilés iront dans le dossier [bin] dans la DLL [demo1.dll] :

- le fichier [spring-config-1.xml] est dans le dossier [bin] du projet.
- le dossier [bin] contient les fichiers nécessaires à l'application :
  - **Spring.Core.dll, Spring.Core.xml** pour les classes Spring
  - **log4net.dll** pour les logs de Spring
  - **demo1.dll** qui est la DLL produite par la génération du projet

La DLL [demo1.dll] produite par la génération du projet est chargée dans l'outil de test graphique [Nunit-Gui 2.2]. Celui-ci affiche automatiquement les classes Nunit présentes dans la DLL :



L'exécution de la méthode [demo] du test NUnit donne les résultats indiqués ci-dessus et repris ci-dessous :

```
1. ***** istia.st.springioc.tests.NunitTestSpringIocDemo1.demo
2. setup test
3. init personne [Simon,40]
4. personne1=[Simon,40]
5. init personne [Brigitte,20]
6. personne2=[Brigitte,20]
7. personne2=[Brigitte,20]
8. destroy personne [Simon,40]
9. destroy personne [Brigitte,20]
10.teardown test
```

Commentaires :

- ligne 2 : le test démarre avec l'excécution de la méthode d'attribut <Setup()>
- ligne 3 : l'opération

```
Dim personne1 As Personne = CType(factory.GetObject("personne1"), Personne)
```

a forcé la création du bean [personne1]. Parce que dans la définition du singleton [personne1] on avait écrit [init-method="init"], la méthode [init] de l'objet [Personne] créé a été exécutée. Le message correspondant est affiché.

```
init personne [Simon,40]
```

- ligne 4 : l'opération

```
     Console.WriteLine("personne1=" + personne1.ToString())
```

a fait afficher la valeur de l'objet [Personne] créé.

```
personne1=[Simon,40]
```

- lignes 5-6 : le même phénomène se répète pour le bean de clé [personne2].

```
init personne [Brigitte,20]
personne2=[Brigitte,20]
```

- ligne 7 : la dernière opération

```
personne2 = CType(factory.GetObject("personne2"), Personne)
Console.WriteLine("personne2=" + personne2.ToString())
```

n'a donné qu'une ligne d'affichage :

```
personne2=[Brigitte,20]
```

Elle n'a donc pas provoqué la création d'un nouvel objet de type [Personne]. Si cela avait été le cas, on aurait eu l'affichage de la méthode [init], ce qui n'est pas le cas ici. C'est le principe du **singleton**. Spring, par défaut, ne crée qu'un seul exemplaire des beans de son fichier de configuration. C'est un service de références d'objet. Si on lui demande la référence d'un objet non encore créé, il le crée et en rend une référence. Si l'objet a déjà été créé, Spring se contente d'en donner une référence.

- lignes 8-10 : la méthode d'attribut <TearDown()> s'exécute (ligne 10). Dedans, l'opération

```
    ' on détruit les singletons
    factory.Dispose()
```

provoque la destruction des singletons créés par Spring. Cela provoque pour chacun d'eux l'exécution de leur méthode [close] parce qu'on avait écrit, dans le fichier de configuration, pour chacun d'eux : " destroy-method=close ". C'est ce qu'indiquent les affichages :

```
destroy personne [Simon,40]
destroy personne [Brigitte,20]
```

Les bases d'une configuration Spring étant maintenant acquises, nous serons désormais un peu plus rapides dans nos explications.

5.2 Exemple 2

Considérons la nouvelle classe [Voiture] suivante :

```
Imports istia.st.springioc.demos

Namespace istia.st.springioc.demos

  Public Class Voiture
    ' champs privés
    Private _marque As String
    Private _type As String
    Private _propriétaire As Personne

    ' propriétés publiques
    Public Property marque() As String
      Get
        Return _marque
      End Get
      Set(ByVal Value As String)
        _marque = Value
      End Set
    End Property

    Public Property type() As String
      Get
        Return _type
      End Get
      Set(ByVal Value As String)
        _type = Value
      End Set
    End Property

    Public Property propriétaire() As Personne
      Get
        Return _propriétaire
      End Get
```

```
    Set(ByVal Value As Personne)
      _propriétaire = Value
    End Set
  End Property

  ' constructeur par défaut
  Public Sub New()

  End Sub

  ' constructeur à trois paramètres
  Public Sub New(ByVal marque As String, ByVal type As String, ByVal propriétaire As Personne)
    ' on initialise les champs privés via leurs propriétés associées
    With Me
      .marque = marque
      .type = type
      .propriétaire = propriétaire
    End With
  End Sub

  ' chaîne d'identité de l'objet Voiture
  Public Overrides Function tostring() As String
    Return String.Format("[{0},{1},{2}]", marque, type, propriétaire.ToString)
  End Function

  ' méthodes init-destroy
  Public Sub init()
    Console.WriteLine("init voiture {0}", Me.ToString)
  End Sub

  Public Sub destroy()
    Console.WriteLine("destroy voiture {0}", Me.ToString)
  End Sub

 End Class
End Namespace
```

La classe présente :

- trois champs privés **_type**, **_marque** et **_propriétaire**. Ces champs peuvent être initialisés et lus par des propriétés publiques. Ils peuvent être également initialisés à l'aide du constructeur **Voiture(String, String, Personne)**. La classe possède également un constructeur sans arguments qui permet de créer d'abord un objet [Voiture] non initialisé et de l'initialiser ensuite via ses propriétés publiques.
- une méthode **toString** pour récupérer la valeur de l'objet [Voiture] sous la forme d'une chaîne de caractères
- une méthode **init** qui sera appelée par Spring juste après la création de l'objet, une méthode **destroy** qui sera appelée à la destruction de l'objet

Pour créer des objets de type [Voiture], nous utiliserons le fichier Spring [**spring-config-2.xml**] suivant :

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<!DOCTYPE objects PUBLIC "-//SPRING//DTD OBJECT//EN"
"http://www.springframework.net/dtd/spring-objects.dtd">
<objects>
  <!-- des personnes -->
  <object id="personne1" type="istia.st.springioc.demos.Personne, demo1" init-method="init"
    destroy-method="close">
    <property name="nom">
      <value>Simon</value>
    </property>
    <property name="age">
      <value>40</value>
    </property>
  </object>
  <object id="personne2" type="istia.st.springioc.demos.Personne, demo1" init-method="init"
    destroy-method="close">
    <property name="nom">
      <value>Brigitte</value>
    </property>
    <property name="age">
      <value>20</value>
    </property>
  </object>
  <!-- une voiture -->
  <object id="voiture1" type="istia.st.springioc.demos.Voiture, demo1" init-method="init"
    destroy-method="destroy">
    <constructor-arg index="0">
      <value>Peugeot</value>
    </constructor-arg>
    <constructor-arg index="1">
      <value>307</value>
    </constructor-arg>
    <constructor-arg index="2">
      <ref object="personne2"/>
```

```
    </constructor-arg>
  </object>
</objects>
```

Ce fichier ajoute aux définitions précédentes un objet de clé "voiture1" de type [Voiture]. Pour initialiser cet objet, on aurait pu écrire :

```
  <object id="voiture1" type="istia.st.springioc.demos.Voiture, demo1" init-method="init"
    destroy-method="destroy">
    <constructor-arg index="0">
      <value>Peugeot</value>
    </constructor-arg>
    <constructor-arg index="1">
      <value>307</value>
    </constructor-arg>
    <constructor-arg index="2">
      <ref object="personne2"/>
    </constructor-arg>
  </object>
```

Plutôt que de choisir cette méthode déjà présentée, nous avons choisi ici, d'utiliser le constructeur **Voiture(String, String, Personne)** de la classe. Par ailleurs, le singleton [voiture1] définit la méthode à appeler lors de la construction initiale de l'objet [init-method] et celle à appeler lors de la destruction de l'objet [destroy-method].

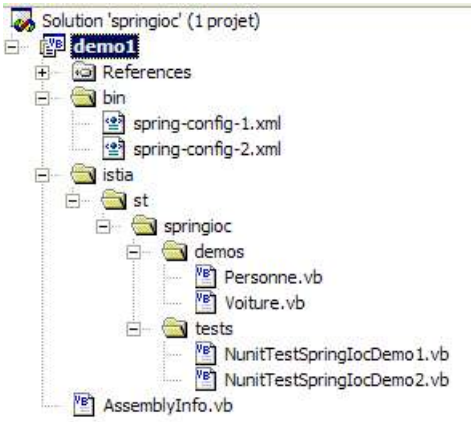
Pour nos tests, nous utiliserons la classe de test NUnit [NunitTestSpringIocDemo2] suivante :

```
Imports System
Imports Spring.Objects.Factory.Xml
Imports System.IO
Imports NUnit.Framework
Imports istia.st.springioc.demos

Namespace istia.st.springioc.tests

  <TestFixture()> _
  Public Class NunitTestSpringIocDemo2
    ' la fabrique de singletons
    Private factory As XmlObjectFactory

    <SetUp()> _
    Public Sub init()
      ' on crée une instance de factory
      factory = New XmlObjectFactory(New FileStream("spring-config-2.xml", FileMode.Open))
      ' log
      Console.WriteLine("setup test")
    End Sub

    <TearDown()> _
  Public Sub destroy()
      ' on détruit les singletons
      factory.Dispose()
      ' on libère le factory
      factory = Nothing
      ' suivi
      Console.WriteLine("teardown test")
    End Sub

    <Test()> _
    Public Sub demo2()
      ' récupération du singleton [voiture1]
      Dim voiture1 As Voiture = CType(factory.GetObject("voiture1"), Voiture)
      Console.WriteLine("Voiture1=[{0}]", voiture1)
    End Sub

  End Class
End Namespace
```

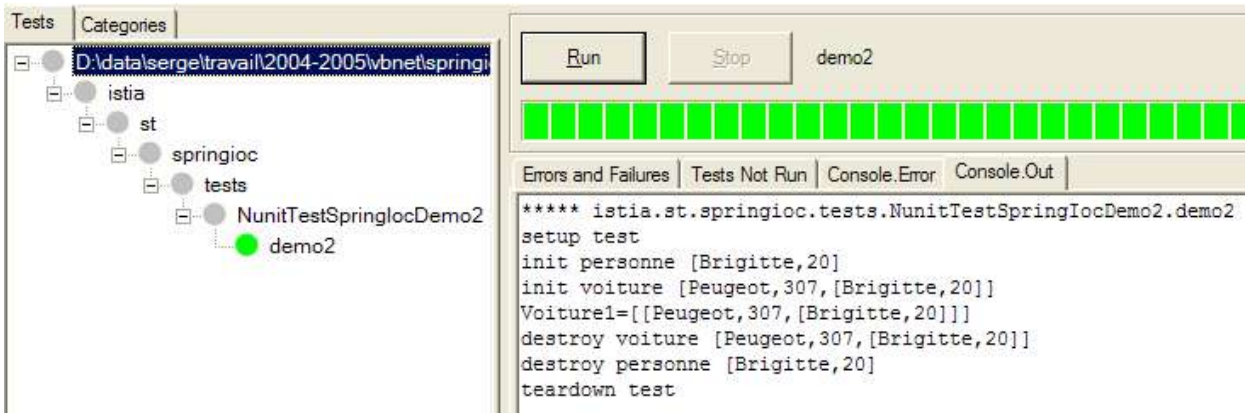
Commentaires :

- les méthodes d'attributs <Setup()> et <TearDown()> restent inchangées.
- la méthode [demo2] récupère une référence sur le singleton [voiture1] et affiche celui-ci.

La structure du projet Visual Studio reste ce qu'elle était auparavant. Seuls deux nouvelles classes sont apparues ainsi qu'un fichier de configuration Spring :



L'exécution du test NUnit donne les résultats suivants :



Commentons les affichages écran :

```
1. ***** istia.st.springioc.tests.NunitTestSpringIocDemo2.demo2
2. setup test
3. init personne [Brigitte,20]
4. init voiture [Peugeot,307,[Brigitte,20]]
5. Voiture1=[[Peugeot,307,[Brigitte,20]]]
6. destroy voiture [Peugeot,307,[Brigitte,20]]
7. destroy personne [Brigitte,20]
8. teardown test
```

Commentaires :

- ligne 2 : méthode d'attribut [Setup] exécutée avant chaque test, ici [demo]
- ligne 3 : Spring commence la création du singleton [voiture1]. Celui-ci a une dépendance vis à vis du singleton [personne2] qui n'existe pas. Ce dernier est donc créé et sa méthode [init] exécutée.
- ligne 4 : le singleton [voiture1] peut maintenant être créé. Sa méthode [init] est alors exécutée.
- ligne 5 : la méthode [demo2] fait afficher la valeur du singleton [voiture1]
- lignes 6-8 : la méthode [TearDown] du test est exécutée. Les singletons sont détruits d'où l'exécution de leurs méthodes [destroy]

5.3 Exemple 3

Nous introduisons la nouvelle classe [GroupePersonnes] suivante :

```
Imports istia.st.springioc.demos

Namespace istia.st.springioc.demos

  Public Class GroupePersonnes
    ' champs privés
    Private _membres() As Personne
    Private _groupesDeTravail As Hashtable

    ' propriétés publiques
```

```
    Public Property membres() As Personne()
      Get
        Return _membres
      End Get
      Set(ByVal Value() As Personne)
        _membres = Value
      End Set
    End Property

    Public Property groupesDeTravail() As Hashtable
      Get
        Return _groupesDeTravail
      End Get
      Set(ByVal Value As Hashtable)
        _groupesDeTravail = Value
      End Set
    End Property

    ' constructeur par défaut
    Public Sub New()

    End Sub

    ' chaîne d'identité de l'objet GroupeDePersonnes
    Public Overrides Function tostring() As String
      ' on parcourt la liste des membres du groupe
      Dim identité As String = "[membres=("
      Dim i As Integer
      For i = 0 To membres.Length - 2
        identité += membres(i).ToString + ","
      Next
      identité += membres(i).ToString + "), groupes de travail=("
      ' on parcourt le dictionnaire des groupes de travail
      Dim clés As IEnumerator = groupesDeTravail.Keys.GetEnumerator
      Dim clé As Object
      While clés.MoveNext
        clé = clés.Current
        identité += String.Format("[{0},{1}] ", clé, groupesDeTravail(clé))
      End While
      ' on rend le résultat
      Return identité + "]"
    End Function

    ' méthodes init-destroy
    Public Sub init()
      Console.WriteLine("init GroupeDePersonnes {0}", Me.ToString)
    End Sub

    Public Sub destroy()
      Console.WriteLine("destroy GroupeDePersonnes {0}", Me.ToString)
    End Sub

  End Class
End Namespace
```

Ses deux membres privés sont :

_membres : un tableau de personnes membres du groupe
_groupesDeTravail : un dictionnaire affectant une personne à un groupe de travail

Ces membres privés sont rendus accessibles via des propriétés publiques. On cherche ici, à montrer comment Spring permet d'initialiser des objets complexes tels que des objets possédant des champs de type tableau ou dictionnaire.

Le fichier Spring [spring-config-3.xml] de configuration sera le suivant :

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<!DOCTYPE objects PUBLIC "-//SPRING//DTD OBJECT//EN"
"http://www.springframework.net/dtd/spring-objects.dtd">
<objects>
  <!-- des personnes -->
  <object id="personne1" type="istia.st.springioc.demos.Personne, demo1" init-method="init"
    destroy-method="close">
    <property name="nom">
      <value>Simon</value>
    </property>
    <property name="age">
      <value>40</value>
    </property>
  </object>
  <object id="personne2" type="istia.st.springioc.demos.Personne, demo1" init-method="init"
    destroy-method="close">
    <property name="nom">
      <value>Brigitte</value>
```

```
    </property>
    <property name="age">
      <value>20</value>
    </property>
  </object>
  <!-- une voiture -->
  <object id="voiture1" type="istia.st.springioc.demos.Voiture, demo1" init-method="init"
    destroy-method="destroy">
    <constructor-arg index="0">
      <value>Peugeot</value>
    </constructor-arg>
    <constructor-arg index="1">
      <value>307</value>
    </constructor-arg>
    <constructor-arg index="2">
      <ref object="personne2" />
    </constructor-arg>
  </object>
  <!-- un groupe de personnes -->
  <object id="groupe1" type="istia.st.springioc.demos.GroupePersonnes" init-method="init"
    destroy-method="destroy">
    <property name="membres">
      <list>
        <ref object="personne1" />
        <ref object="personne2" />
      </list>
    </property>
    <property name="groupesDeTravail">
      <dictionary>
        <entry key="Brigitte">
          <value>Marketing</value>
        </entry>
        <entry key="Simon">
          <value>Ressources humaines</value>
        </entry>
      </dictionary>
    </property>
  </object>
</objects>
```

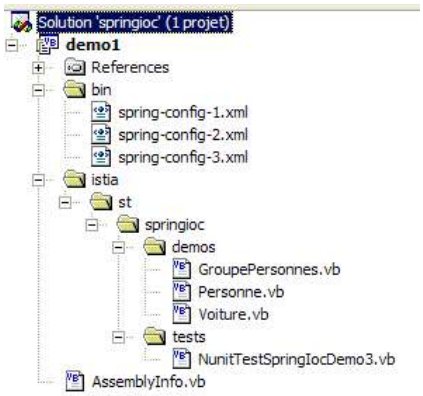
1. la balise **<list>** permet d'initialiser avec différentes valeurs un champ de type tableau ou de type **IList**.
2. la balise **<dictionary>** permet de faire la même chose avec un champ implémentant l'interface **IDictionary**

Pour nos tests, nous utiliserons la classe de test NUnit [NunitTestSpringIocDemo3] suivante :

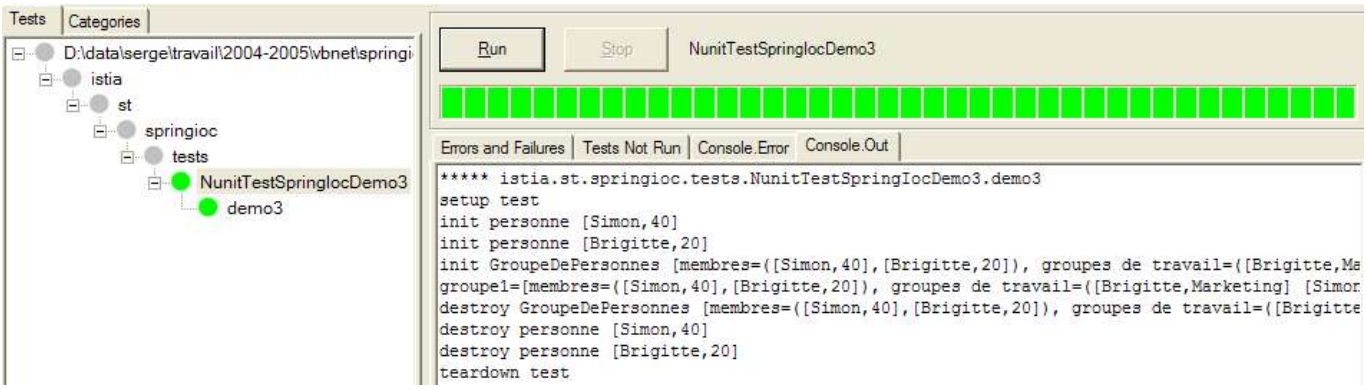
```
Imports System
Imports Spring.Objects.Factory.Xml
Imports System.IO
Imports NUnit.Framework
Imports istia.st.springioc.demos

Namespace istia.st.springioc.tests

  <TestFixture()> _
  Public Class NunitTestSpringIocDemo3
    ' l'objet à tester
    Private factory As XmlObjectFactory

    <SetUp()> _
    Public Sub init()
      ' on crée une instance de factory
      factory = New XmlObjectFactory(New FileStream("spring-config-3.xml", FileMode.Open))
      ' log
      Console.WriteLine("setup test")
    End Sub

    <TearDown()> _
    Public Sub destroy()
      ' on détruit les singletons
      factory.Dispose()
      ' suivi
      Console.WriteLine("teardown test")
    End Sub

    <Test()> _
  Public Sub demo3()
      ' récupération du singleton [groupe1]
      Dim groupe1 As GroupePersonnes = CType(factory.GetObject("groupe1"), GroupePersonnes)
      Console.WriteLine("groupe1={0}", groupe1)
    End Sub

  End Class
End Namespace
```

La méthode [demo3] récupère le singleton [groupe1] et l'affiche.

La structure du projet Visual Studio reste ce qu'elle était auparavant si ce n'est qu'il a deux classes ainsi qu'un fichier de configuration supplémentaires.



L'exécution de la méthode [demo3] du test NUnit donne les résultats suivants :



Commentaires :

```
1. ***** istia.st.springioc.tests.NunitTestSpringIocDemo3.demo3
2. setup test
3. init personne [Simon,40]
4. init personne [Brigitte,20]
5. init GroupeDePersonnes [membres=([Simon,40],[Brigitte,20]), groupes de travail=
   ([Brigitte,Marketing] [Simon,Ressources humaines] ]
6. groupe1=[membres=([Simon,40],[Brigitte,20]), groupes de travail=([Brigitte,Marketing]
   [Simon,Ressources humaines] ]
7. destroy GroupeDePersonnes [membres=([Simon,40],[Brigitte,20]), groupes de travail=
   ([Brigitte,Marketing] [Simon,Ressources humaines] ]
8. destroy personne [Simon,40]
9. destroy personne [Brigitte,20]
10.teardown test
```

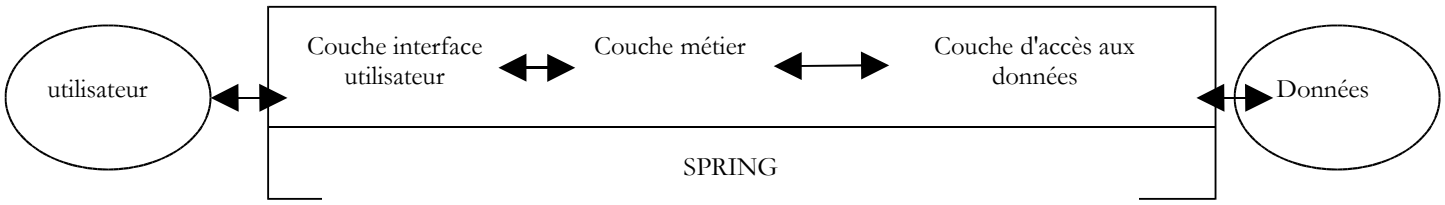
- ligne 2 : méthode d'attribut [Setup] exécutée avant chaque test, ici [demo]
- ligne 3-4 : Spring commence la création du singleton [groupe1]. Celui-ci a une dépendance vis à vis des singletons [personne1, personne2] qui n'existent pas. Ces derniers sont créés et leurs méthodes [init] exécutées.
- ligne 5 : le singleton [groupe1] peut maintenant être créé. Sa méthode [init] est alors exécutée.
- ligne 6 : la méthode [demo3] fait afficher la valeur du singleton [groupe1]
- lignes 7-10 : la méthode [TearDown] du test est exécutée. Les singletons sont détruits d'où l'exécution de leurs méthodes [destroy]

6 Spring pour configurer les applications à trois couches

Nous nous proposons ici d'étudier sur un exemple comment Spring peut être utilisé pour configurer les applications à trois couches. Un exemple courant de ce type d'applications sont les applications web.

6.1 Le problème

On souhaite construire une application 3-tier ayant la structure suivante :



- les trois couches seront rendues indépendantes grâce à l'utilisation d'interfaces
- l'intégration des trois couches sera réalisée par Spring
- on créera des paquetages séparés pour chacune des trois couches que l'on appellera [**control]**, [**domain]** et [**dao]**. Un paquetage supplémentaire [**tests]** contiendra les applications de tests.

6.1.1 La couche d'accès aux données

La couche DAO implémentera l'interface suivante :

```
Namespace istia.st.spring3tier.dao
  Public Interface IDao
    ' faire qq chose dans la couche [dao]
    Function doSomethingInDaoLayer(ByVal a As Integer, ByVal b As Integer) As Integer
  End Interface
End Namespace
```

- écrire deux classes **Dao1Impl1** et **Dao1Impl2** implémentant l'interface **IDao**. La méthode **Dao1Impl1**. **doSomethingInDaoLayer** rendra a+b et méthode **Dao1Impl2**. **doSomethingInDaoLayer** rendra a-b.
- écrire une classe de test NUnit testant les deux classes précédentes

6.1.2 La couche métier

La couche métier implémentera l'interface suivante :

```
Namespace istia.st.spring3tier.domain
  Public Interface IDomain
    ' faire qq chose dans la couche [domain]
    Function doSomethingInDomainLayer(ByVal a As Integer, ByVal b As Integer) As Integer
  End Interface
End Namespace
```

- écrire deux classes **Domain1Impl1** et **Domain1Impl2** implémentant l'interface **IDomain**. Ces classes auront un champ privé de type **IDao** qui pourra être initialisé via une propriété publique. La méthode **doSomethingInDomainLayer** de la classe [**DomainImpl1]** incrémentera a et b d'une unité puis passera ces deux paramètres à la méthode **doSomethingInDaoLayer** de l'objet de type **IDao1** reçu. La méthode **doSomethingInDomainLayer** de la classe [**Domain1Impl2**elle, décrémentera a et b d'une unité avant de faire la même chose.
- écrire une classe de test NUnit testant les deux classes précédentes

6.1.3 La couche interface utilisateur

La couche interface utilisateur implémentera l'interface suivante :

```
Namespace istia.st.spring3tier.control
  Public Interface IControl
    ' faire qq chose dans la couche [control]
    Function doSomethingInControlLayer(ByVal a As Integer, ByVal b As Integer) As Integer
  End Interface
End Namespace
```

- écrire deux classes **Control1Impl1** et **Control1Impl2** implémentant l'interface **IContro1**. Ces classes auront un champ privé de type **IDomain** qui pourra être initialisé via une propriété publique. La méthode **doSomethingInControlLayer** de la classe [**Control1Impl1]** incrémentera a et b d'une unité puis passera ces deux paramètres à la méthode

doSomethingInDomainLayer de l'objet de type **IDomain1** reçu. La méthode **doSomethingInControlLayer** de la classe **[Control11Impl2]** elle, décrémentera a et b d'une unité avant de faire la même chose.

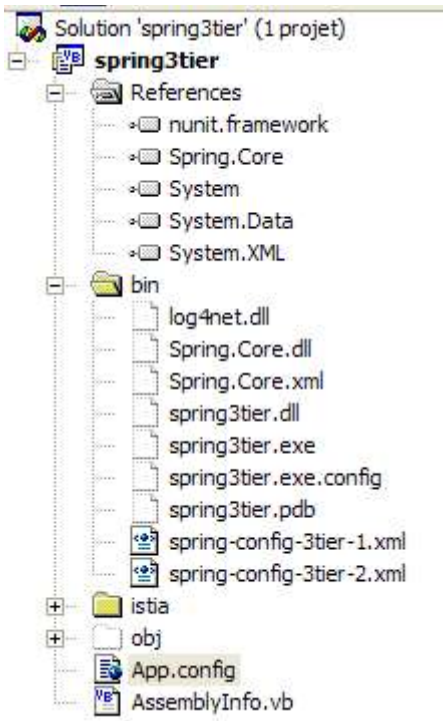
- écrire une classe de test NUnit testant les deux classes précédentes

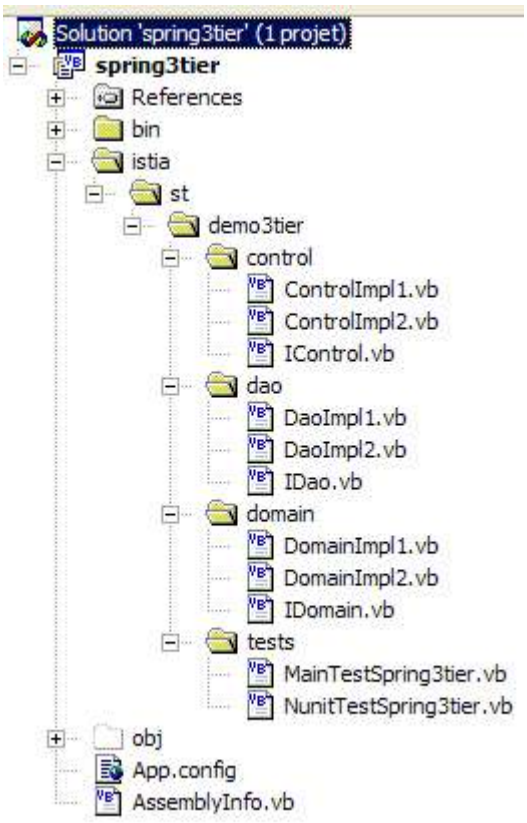
6.1.4 Intégration avec Spring

- écrire un fichier de configuration Spring qui décidera quelles classes chacune des trois couches précédentes devra utiliser
- écrire une classe de test NUnit utilisant différentes configurations Spring, afin de mettre en lumière la flexibilité de l'application écrite
- écrire une application autonome (méthode main) donnant deux paramètres à la méthode [doSomethingInControlLayer] de l'interface **IControl** implémentée et affichant le résultat rendu par l'interface.

6.2 Une solution

6.2.1 Le projet Visual Studio

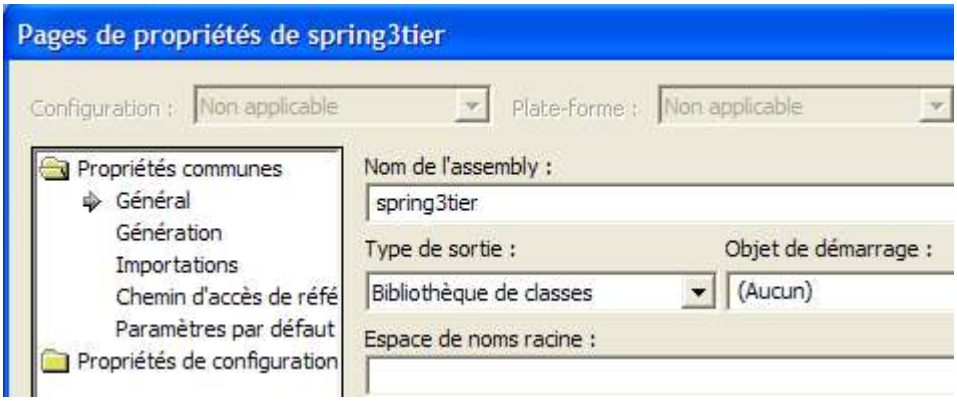




Commentaires :

- les fichiers nécessaires à Spring ont été placés dans le dossier [bin] : Spring.Core.dll, log4net.dll, Spring.Core.xml
- les fichiers de configuration de Spring [spring-config-3tier-*.xml] ont eux aussi été placés dans le dossier [bin]
- sous la racine [istia], on trouve les différentes classes de l'application

Le projet a été configuré pour générer la DLL [spring3tier.dll] dans le dossier [bin] :



6.2.2 Le paquetage [istia.st.spring3tier.dao]

L'interface **IDao** :

```
Namespace istia.st.spring3tier.dao
  Public Interface IDao
    ' faire qq chose dans la couche [dao]
    Function doSomethingInDaoLayer(ByVal a As Integer, ByVal b As Integer) As Integer
  End Interface
End Namespace
```

Une première classe d'implémentation :

```
Namespace istia.st.spring3tier.dao
  Public Class DaoImpl1
    Implements istia.st.spring3tier.dao.IDao

    ' faire qq chose dans la couche [dao]
    Public Function doSomethingInDaoLayer(ByVal a As Integer, ByVal b As Integer) As Integer Implements
IDao.doSomethingInDaoLayer

      Return a + b
    End Function
  End Class

End Namespace
```

- la classe n'a pas de champ privé
- La méthode [doSomethingInDaoLayer] rend la somme de ses paramètres comme demandé.

Une seconde classe d'implémentation :

```
Namespace istia.st.spring3tier.dao
  Public Class DaoImpl2
    Implements istia.st.spring3tier.dao.IDao

    ' faire qq chose dans la couche [dao]
    Public Function doSomethingInDaoLayer(ByVal a As Integer, ByVal b As Integer) As Integer Implements
IDao.doSomethingInDaoLayer

      Return a - b
    End Function
  End Class

End Namespace
```

- la classe n'a pas de champ privé
- la méthode [doSomethingInDaoLayer] rend la différence de ses paramètres comme demandé.

6.2.3 Le paquetage [istia.st.spring3tier.domain]

L'interface **IDomain** :

```
Namespace istia.st.spring3tier.domain
  Public Interface IDomain
    ' faire qq chose dans la couche [domain]
    Function doSomethingInDomainLayer(ByVal a As Integer, ByVal b As Integer) As Integer
```

```
  End Interface
End Namespace
```

Une première classe d'implémentation **IDomainImpl1** :

```
Imports istia.st.spring3tier.dao

Namespace istia.st.spring3tier.domain
  Public Class DomainImpl1
    Implements istia.st.spring3tier.domain.IDomain

    ' champs privés
    Private _dao As IDao

    ' propriété associée
    Public WriteOnly Property dao() As IDao
      Set(ByVal Value As IDao)
        _dao = Value
      End Set
    End Property

    ' constructeur par défaut
    Public Sub New()
    End Sub

    ' faire qq chose dans la couche [domain]
    Public Function doSomethingInDomainLayer(ByVal a As Integer, ByVal b As Integer) As Integer Implements
IDomain.doSomethingInDomainLayer
      a += 1
      b += 1
      Return _dao.doSomethingInDaoLayer(a, b)
    End Function
  End Class

End Namespace
```

- la classe a un champ privé qui est une référence au singleton de type [IDao] qui donne accès à la couche [Dao]. Ce champ sera initialisé par Spring (injection de dépendance) au moment de la construction de l'objet.
- La méthode [doSomethingInDomainLayer] incrémente ses paramètres puis les passe à la méthode [doSomethingInDaoLayer] du singleton [dao]

Une seconde classe d'implémentation **IDomainImpl2** :

```
Imports istia.st.spring3tier.dao

Namespace istia.st.spring3tier.domain
  Public Class DomainImpl2
    Implements istia.st.spring3tier.domain.IDomain

    ' champs privés
    Private _dao As IDao

    ' propriété associée
    Public WriteOnly Property dao() As IDao
      Set(ByVal Value As IDao)
        _dao = Value
      End Set
    End Property

    ' constructeur par défaut
    Public Sub New()
    End Sub

    ' faire qq chose dans la couche [domain]
    Public Function doSomethingInDomainLayer(ByVal a As Integer, ByVal b As Integer) As Integer Implements
IDomain.doSomethingInDomainLayer
      a -= 1
      b -= 1
      Return _dao.doSomethingInDaoLayer(a, b)
    End Function
  End Class

End Namespace
```

- la classe a un champ privé qui est une référence au singleton de type [IDao] qui donne accès à la couche [Dao]. Ce champ sera initialisé par Spring (injection de dépendance) au moment de la construction de l'objet.
- La méthode [doSomethingInDomainLayer] décrémente ses paramètres puis les passe à la méthode [doSomethingInDaoLayer] du singleton [dao]

6.2.4 Le paquetage [istia.st.spring3tier.control]

L'interface **IControl** :

```
Namespace istia.st.spring3tier.control
  Public Interface IControl
    ' faire qq chose dans la couche [control]
    Function doSomethingInControlLayer(ByVal a As Integer, ByVal b As Integer) As Integer
  End Interface
End Namespace
```

Une première classe d'implémentation **ControlImpl1** :

```
Imports istia.st.spring3tier.domain

Namespace istia.st.spring3tier.control
  Public Class ControlImpl1
    Implements istia.st.spring3tier.control.IControl

    ' champs privés
    Private _domain As IDomain

    ' propriété associée
    Public WriteOnly Property domain() As IDomain
      Set(ByVal Value As IDomain)
        _domain = Value
      End Set
    End Property

    ' constructeur par défaut
    Public Sub New()
    End Sub

    ' faire qq chose dans la couche [control]
    Public Function doSomethingInControlLayer(ByVal a As Integer, ByVal b As Integer) As Integer
Implements IControl.doSomethingInControlLayer
      a += 1
      b += 1
      Return _domain.doSomethingInDomainLayer(a, b)
    End Function
  End Class

End Namespace
```

- la classe a un champ privé qui est une référence au singleton de type [IDomain] qui donne accès à la couche [Domain]. Ce champ sera initialisé par Spring (injection de dépendance) au moment de la construction de l'objet.
- La méthode [doSomethingInControlLayer] incrémente ses paramètres puis les passe à la méthode [doSomethingInDomainLayer] du singleton [domain]

Une seconde classe d'implémentation **IControlImpl2** :

```
Imports istia.st.spring3tier.domain

Namespace istia.st.spring3tier.control
  Public Class ControlImpl2
    Implements istia.st.spring3tier.control.IControl

    ' champs privés
    Private _domain As IDomain

    ' propriété associée
    Public WriteOnly Property domain() As IDomain
      Set(ByVal Value As IDomain)
        _domain = Value
      End Set
    End Property

    ' constructeur par défaut
    Public Sub New()
    End Sub

    ' faire qq chose dans la couche [control]
    Public Function doSomethingInControlLayer(ByVal a As Integer, ByVal b As Integer) As Integer
Implements IControl.doSomethingInControlLayer
      a -= 1
      b -= 1
      Return _domain.doSomethingInDomainLayer(a, b)
    End Function
  End Class
```

```
End Namespace
```

- la classe a un champ privé qui est une référence au singleton de type [IDomain] qui donne accès à la couche [Domain]. Ce champ sera initialisé par Spring (injection de dépendance) au moment de la construction de l'objet.
- La méthode [doSomethingInControlLayer] décrémente ses paramètres puis les passe à la méthode [doSomethingInDomainLayer] du singleton [domain].

6.2.5 Les fichiers de configuration [Spring]

Le fichier [spring-config-3tier-1.xml] utilise les versions 1 des implémentations :

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<!DOCTYPE objects PUBLIC "-//SPRING//DTD OBJECT//EN"
"http://www.springframework.net/dtd/spring-objects.dtd">
<objects>
  <!-- la classe dao -->
  <object id="dao" type="istia.st.spring3tier.dao.DaoImpl1, spring3tier"></object>
  <!-- la classe domain -->
  <object id="domain" type="istia.st.spring3tier.domain.DomainImpl1, spring3tier">
    <property name="dao">
      <ref object="dao" />
    </property>
  </object>
  <!-- la classe control -->
  <object id="control" type="istia.st.spring3tier.control.ControlImpl1, spring3tier">
    <property name="domain">
      <ref object="domain" />
    </property>
  </object>
</objects>
```

Le fichier [spring-config-3tier-2.xml] utilise les versions 2 des implémentations :

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<!DOCTYPE objects PUBLIC "-//SPRING//DTD OBJECT//EN"
"http://www.springframework.net/dtd/spring-objects.dtd">
<objects>
  <!-- la classe dao -->
  <object id="dao" type="istia.st.spring3tier.dao.DaoImpl2, spring3tier"></object>
  <!-- la classe domain -->
  <object id="domain" type="istia.st.spring3tier.domain.DomainImpl2, spring3tier">
    <property name="dao">
      <ref object="dao" />
    </property>
  </object>
  <!-- la classe control -->
  <object id="control" type="istia.st.spring3tier.control.ControlImpl2, spring3tier">
    <property name="domain">
      <ref object="domain" />
    </property>
  </object>
</objects>
```

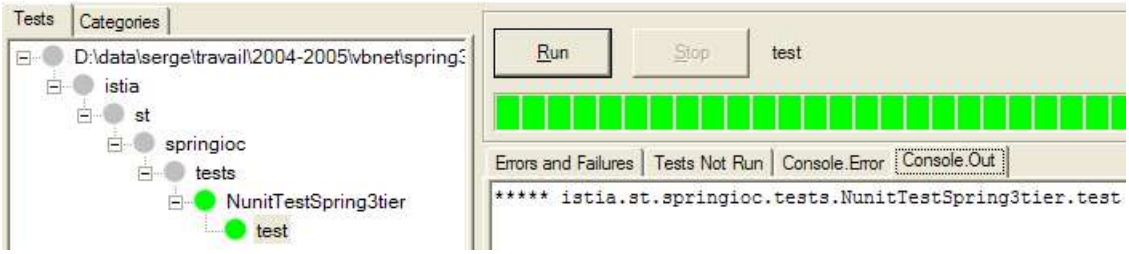
6.2.6 Le paquetage des tests [istia.st.spring3tier.tests]

Un test Nunit [NunitTestSpring3tier.vb] :

```
Imports System
Imports Spring.Objects.Factory.Xml
Imports System.IO
Imports NUnit.Framework
Imports istia.st.spring3tier.control
Imports istia.st.spring3tier.domain

Namespace istia.st.springioc.tests

  <TestFixture()> _
  Public Class NunitTestSpring3tier
    ' les fabriques de singleton
    Private factory1 As XmlObjectFactory
    Private factory2 As XmlObjectFactory

    <SetUp()> _
    Public Sub init()
      ' on crée les fabriques de singleton
      factory1 = New XmlObjectFactory(New FileStream("spring-config-3tier-1.xml", FileMode.Open))
      factory2 = New XmlObjectFactory(New FileStream("spring-config-3tier-2.xml", FileMode.Open))
    End Sub
```

```
    <TearDown()> _
    Public Sub destroy()
      ' on détruit les singletons
      factory1.Dispose()
      factory2.Dispose()
      ' on libère les fabriques de singletons
      factory1 = Nothing
      factory2 = Nothing
    End Sub

    <Test()> _
    Public Sub test()
      'on récupère une implémentation de l'interface IControl
      Dim control1 As IControl = CType(factory1.GetObject("control"), IControl)
      ' on utilise la classe
      Dim a1 As Integer = 10, b1 As Integer = 20
      Dim res1 As Integer = control1.doSomethingInControlLayer(a1, b1)
      Assert.AreEqual(34, res1)
      ' on récupère une autre implémentation de l'interface IControl
      Dim control2 As IControl = CType(factory2.GetObject("control"), IControl)
      ' on utilise la classe
      Dim a2 As Integer = 10, b2 As Integer = 20
      Dim res2 As Integer = control2.doSomethingInControlLayer(a2, b2)
      Assert.AreEqual(-10, res2)
    End Sub

  End Class
End Namespace
```

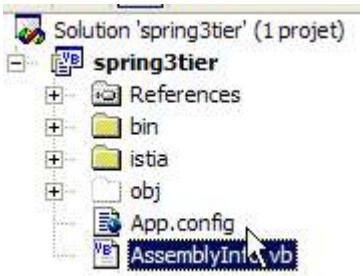
L'exécution de ce test donne les résultats suivants :



Le lecteur qui a une version avec couleurs de ce document verra que les résultats sont au "vert" indiquant par là la réussite du test.

6.3 Une autre type de fichier de configuration de Spring

Une application VB.net peut être configurée à l'aide d'un fichier appelé [App.config]. Ce fichier est placé à la racine du projet Visual Studio :



Spring peut tirer parti de ce fichier de configuration. Considérons le fichier [App.config] suivant, inspiré d'exemples de la documentation de [Spring.net] :

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<configuration>
  <configSections>
    <sectionGroup name="spring">
      <section name="context" type="Spring.Context.Support.ContextHandler, Spring.Core" />
      <section name="objects" type="Spring.Context.Support.DefaultSectionHandler, Spring.Core" />
    </sectionGroup>
  </configSections>
  <spring>
    <context type="Spring.Context.Support.XmlApplicationContext, Spring.Core">
      <resource uri="config://spring/objects" />
```

```
    </context>
    <objects>
      <!-- une première configuration -->
      <!-- la classe dao -->
      <object id="dao1" type="istia.st.spring3tier.dao.DaoImpl1, spring3tier"></object>
      <!-- la classe domain -->
      <object id="domain1" type="istia.st.spring3tier.domain.DomainImpl1, spring3tier">
        <property name="dao">
          <ref object="dao1" />
        </property>
      </object>
      <!-- la classe control -->
      <object id="control1" type="istia.st.spring3tier.control.ControlImpl1, spring3tier">
        <property name="domain">
          <ref object="domain1" />
        </property>
      </object>
      <!-- une deuxième configuration -->
      <!-- la classe dao -->
      <object id="dao2" type="istia.st.spring3tier.dao.DaoImpl2, spring3tier"></object>
      <!-- la classe domain -->
      <object id="domain2" type="istia.st.spring3tier.domain.DomainImpl2, spring3tier">
        <property name="dao">
          <ref object="dao2" />
        </property>
      </object>
      <!-- la classe control -->
      <object id="control2" type="istia.st.spring3tier.control.ControlImpl2, spring3tier">
        <property name="domain">
          <ref object="domain2" />
        </property>
      </object>
    </objects>
  </spring>
</configuration>
```

Note : les informations ci-dessous sont données avec réserve. Je ne suis pas sûr d'avoir correctement compris la signification de tous les éléments du fichier de configuration ci-dessus.

La syntaxe XML du fichier [App.config] exige qu'il respecte la syntaxe :

```
<configuration>
....
</configuration>
```

La gestion des différentes sections de [App.config] peut être déléguée à des programmes externes. C'est ce qui est fait ici dans la section [ConfigSections] :

```
  <configSections>
    <sectionGroup name="spring">
      <section name="context" type="Spring.Context.Support.ContextHandler, Spring.Core" />
      <section name="objects" type="Spring.Context.Support.DefaultSectionHandler, Spring.Core" />
    </sectionGroup>
  </configSections>
```

Le code ci-dessus signifie que la section appelée [**spring/context**] doit être gérée par la classe [**Spring.Context.Support.ContextHandler**] qu'on trouvera dans l'assemblage [**Spring.Core.dll**] et que la section appelée [**spring/objects**] doit être gérée par la classe [**Spring.Context.Support.DefaultSectionHandler**] qu'on trouvera là encore dans l'assemblage [**Spring.Core.dll**].

La section [spring/context] est la suivante :

```
  <spring>
    <context type="Spring.Context.Support.XmlApplicationContext, Spring.Core">
      <resource uri="config://spring/objects" />
    </context>
...
  </spring>
```

On y semble dire que la section [spring/objects] du fichier de configuration doit être gérée par la classe [**Spring.Context.Support.XmlApplicationHandler**] qu'on trouvera dans l'assemblage [**Spring.Core.dll**]. Cette classe doit exploiter une ressource XML dont l'emplacement est [config://spring/objects], c.a.d. la section [spring/objects] du fichier de configuration courant.

On retrouve dans la section [spring/objects] la syntaxe Spring à laquelle nous sommes maintenant habitués.

Nous avons dans la section **<objects>... </objects>** du fichier [**App.config**] défini deux configurations possibles pour notre application 3tier :

- une qui utilise la version 1 des implémentations d'interfaces
- une autre qui utilise la version 2 de ces mêmes implémentations

Maintenant comment exploiter le fichier [App.config] ?

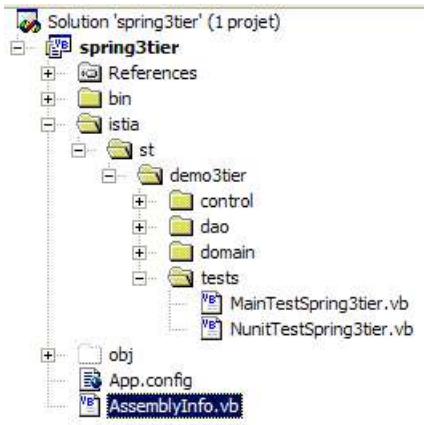
Le code suivant montre une application console qui exploite le fichier [App.config] précédent :

```
Imports System
Imports Spring.Context
Imports System.IO
Imports NUnit.Framework
Imports istia.st.spring3tier.control
Imports istia.st.spring3tier.domain
Imports System.Configuration

Namespace istia.st.springioc.tests

  Module MainTestSpring3tier

    Public Sub main()
      ' le contexte Spring qui va nous permettre de récupérer les singletons
      Dim contexte As IApplicationContext = CType(ConfigurationSettings.GetConfig("spring/context"),
IApplicationContext)
      ' on récupère une 1ère implémentation de l'interface IControl
      Dim control1 As IControl = CType(contexte.GetObject("control1"), IControl)
      ' on utilise la classe
      Dim a1 As Integer = 10, b1 As Integer = 20
      Console.WriteLine("res1({0},{1})={2}", a1, b1, control1.doSomethingInControlLayer(a1, b1))
      ' on récupère une autre implémentation de l'interface IControl
      Dim control2 As IControl = CType(contexte.GetObject("control2"), IControl)
      ' on utilise la classe
      Dim a2 As Integer = 10, b2 As Integer = 20
      Console.WriteLine("res2({0},{1})={2}", a2, b2, control2.doSomethingInControlLayer(a2, b2))
      ' pause
      Console.WriteLine("Tapez [entrée] pour continuer...")
      Console.ReadLine()
    End Sub

  End Module
End Namespace
```

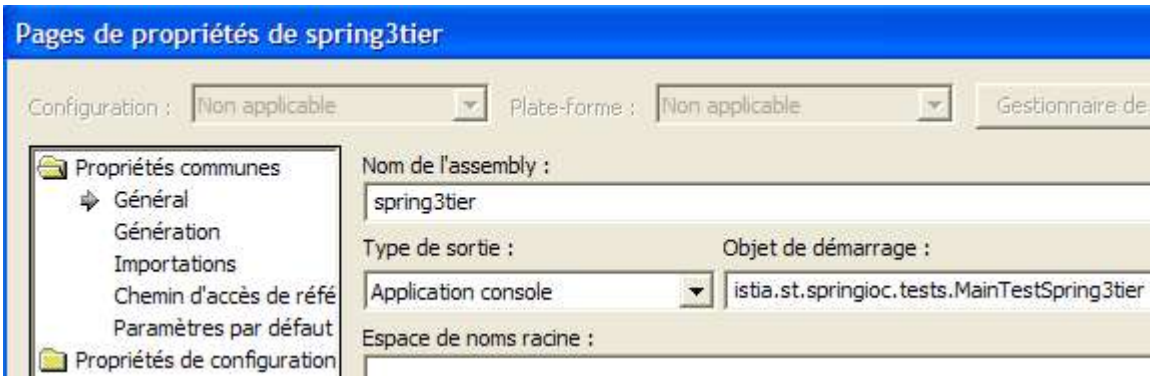
Commentaires :

- dans les exemples NUnit que nous avons utilisés jusqu'ici, nous utilisions un objet [XmlObjectFactory] pour obtenir les singletons dont nous avions besoin. Ici l'objet utilisé est de type [IApplicationContext], un interface de Spring. Il s'obtient à partir de [App.config] grâce à la classe [ConfigurationSettings], classe traditionnellement utilisée dans .Net pour exploiter les fichiers de configuration.
- on demande le gestionnaire de la section [spring/context]. Si on se reporte au fichier [App.config], on découvre que celui-ci est de type [XmlApplicationContext] et qu'il est chargé de gérer la section [spring/objects] de [App.config].
- une fois l'objet de type [IApplicationContext] récupéré, il s'utilise comme l'objet [XmlObjectFactory] que nous avions utilisé jusqu'à maintenant.

Le programme précédent s'appelle [MainTestSpring3tier.vb] et est placé dans le paquetage [tests] :



Le projet [spring3tier] est configuré pour que [MainTestSpring3tier] :

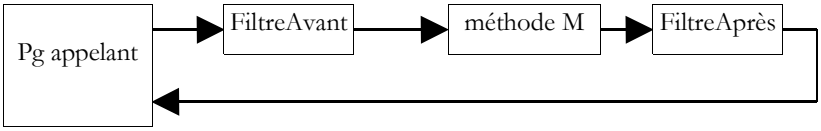


L'exécution du projet donne les résultats suivants :

```
res1(10,20)=34
res2(10,20)=-10
Tapez [entrée] pour continuer...
```

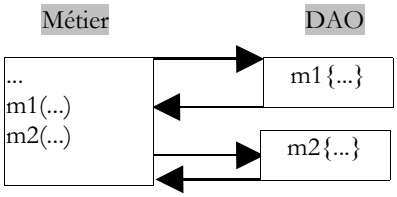
7 Conclusion

Le framework Spring permet une réelle souplesse aussi bien dans l'architecture des applications que dans leur configuration. Nous avons utilisé le concept IoC, l'un des deux piliers de Spring. L'autre pilier est AOP (Aspect Oriented Programming) que nous n'avons pas présenté. Il permet d'ajouter, par configuration, du "comportement" à une méthode de classe sans modifier le code de celle-ci. Schématiquement, AOP permet de filtrer les appels à certaines méthodes :

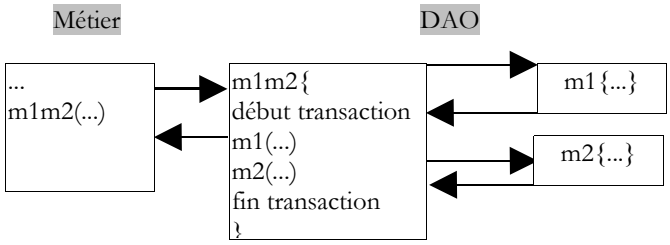


- le filtre peut être exécuté avant ou après la méthode M cible, ou les deux.
- la méthode M ignore l'existence de ces filtres. Ceux-ci sont définis dans le fichier de configuration de Spring.
- le code de la méthode M n'est pas modifié. Les filtres sont des classes Java à construire. Spring fournit des filtres prédéfinis, notamment pour gérer les transactions de SGBD.
- les filtres sont des beans et à ce titre sont définis dans le fichier de configuration de Spring en tant que beans.

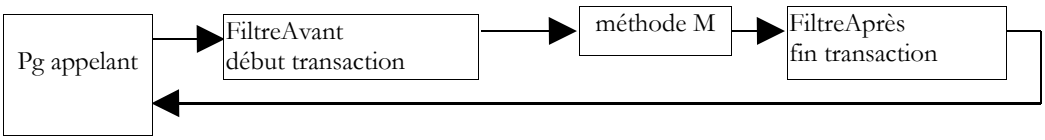
Un filtre courant est le filtre transactionnel. Prenons une méthode M de la couche métier réalisant deux opérations indissociables sur des données (unité de travail). Elle fait appel à deux méthodes M1 et M2 de la couche DAO pour réaliser ces deux opérations.



Parce qu'elle est dans la couche métier, la méthode M fait abstraction du support de ces données. Elle n'a pas, par exemple, à faire l'hypothèse que les données sont dans un SGBD et qu'elle a besoin de mettre les deux appels aux méthodes M1 et M2 au sein d'une transaction de SGBD. C'est à la couche DAO de s'occuper de ces détails. Une solution au problème précédent est alors de créer une méthode dans la couche DAO qui ferait elle-même appel aux méthodes M1 et M2, appels qu'elle engloberait dans une transaction de SGBD.



La solution du filtrage AOP est plus souple. Elle va permettre de définir un filtre qui, avant l'appel de M va commencer une transaction et après l'appel va opérer un commit ou rollback selon les cas.



Il y a plusieurs avantages à cette approche :

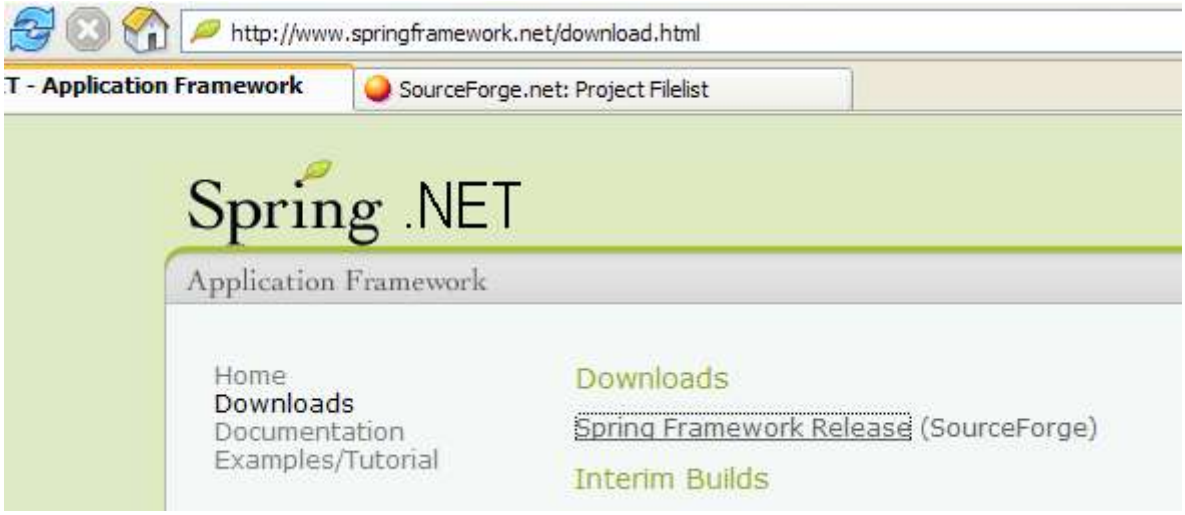
- une fois le filtre défini, il peut être appliqué à plusieurs méthodes, par exemple toutes celles qui ont besoin d'une transaction
- les méthodes ainsi filtrées n'ont pas à être réécrites
- les filtres à utiliser étant définis par configuration, on peut les changer

Pour davantage d'informations : http://www.springframework.net.

8 Annexes

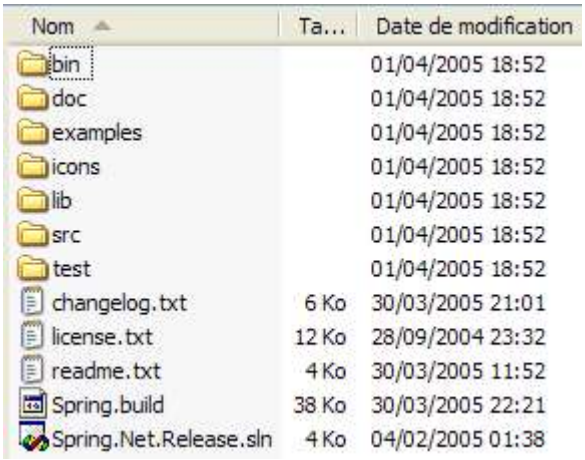
8.1 Où trouver Spring ?

Le site principal de Spring est [http://www.springframework.org/]. C'est le site de la version Java. La version .Net en cours de développement (avril 2005) est à l'url [http://www.springframework.net/].



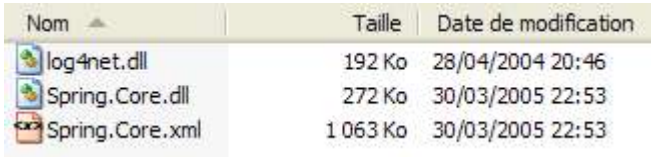
Le site de téléchargement est chez [SourceForge] :

Une fois le zip ci-dessus récupéré, le décompresser :



Nom	Ta...	Date de modification
bin		01/04/2005 18:52
doc		01/04/2005 18:52
examples		01/04/2005 18:52
icons		01/04/2005 18:52
lib		01/04/2005 18:52
src		01/04/2005 18:52
test		01/04/2005 18:52
changelog.txt	6 Ko	30/03/2005 21:01
license.txt	12 Ko	28/09/2004 23:32
readme.txt	4 Ko	30/03/2005 11:52
Spring.build	38 Ko	30/03/2005 22:21
Spring.Net.Release.sln	4 Ko	04/02/2005 01:38

Dans ce document, nous n'avons utilisé que le contenu du dossier [bin] :



Nom	Taille	Date de modification
log4net.dll	192 Ko	28/04/2004 20:46
Spring.Core.dll	272 Ko	30/03/2005 22:53
Spring.Core.xml	1 063 Ko	30/03/2005 22:53

Dans un projets Visual Studio utisant Spring, il faut faire systématiquement deux choses :

- mettre les fichiers ci-dessus dans le dossier [bin] du projet
- ajouter au projet une référence à l'assembly [Spring.Core.dll]

8.2 Où trouver Nunit ?

Le site principal de Nunit est [http://www.nunit.org/]. La version disponible en avril 2005 est la 2.2.0 :

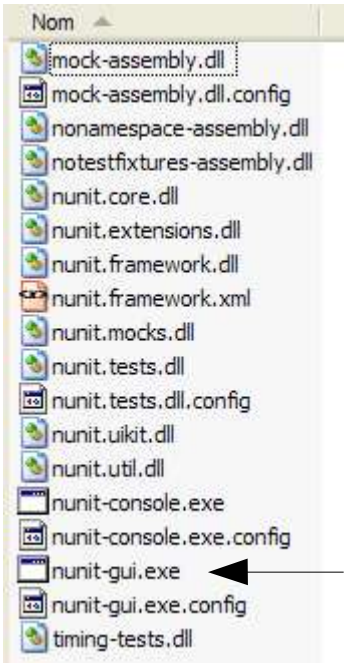
Current Production Release

Production releases are stable releases that have gone through a beta review and had most of the problems fixed. Most people will want to use a production release.

NUnit 2.2 Production Release	
win	NUnit-2.2.0.msi
mono	NUnit-2.2.0-mono.zip
src	NUnit-2.2.0-src.zip

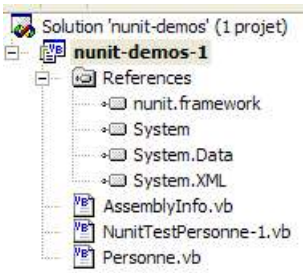
Téléchargez cette version et installez la. L'installation crée un dossier où on trouvera la version graphique de test :

Ce qui est intéressant se trouve dans le dossier [bin] :



La flèche ci-dessus désigne l'utilitaire graphique de test. L'installation a également ajouté de nouveaux éléments au référentiel d'assemblages de Visual Studio que nous allons découvrir maintenant.

Créons le projet Visual Studio suivant :



La classe testée est dans [Personne.vb] :

```
Public Class Personne

  ' champs privés
  Private _nom As String
  Private _age As Integer

  ' constructeur par défaut
  Public Sub New()
  End Sub

  ' propriétés associées aux champs privés
  Public Property nom() As String
    Get
      Return _nom
    End Get
    Set(ByVal Value As String)
      _nom = Value
    End Set
  End Property

  Public Property age() As Integer
    Get
      Return _age
    End Get
    Set(ByVal Value As Integer)
      _age = Value
    End Set
  End Property

  ' chaîne d'identité
  Public Overrides Function tostring() As String
    Return String.Format("[{0},{1}]", nom, age)
  End Function

  ' méthode init
  Public Sub init()
    Console.WriteLine("init personne {0}", Me.ToString)
  End Sub

  ' méthode close
  Public Sub close()
    Console.WriteLine("destroy personne {0}", Me.ToString)
  End Sub

End Class
```

La classe de test est dans [NunitTestPersonne-1.vb] :

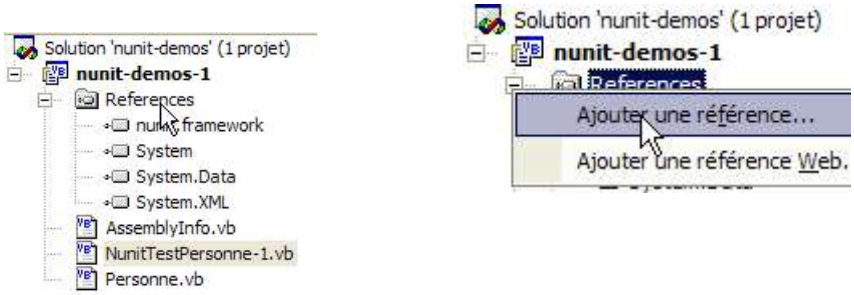
```
Imports System
Imports NUnit.Framework

<TestFixture()> _
 Public Class NunitTestPersonne

  ' objet testé
  Private personne1 As Personne

  <SetUp()> _
  Public Sub init()
    ' on crée une instance de Personne
    personne1 = New Personne
    ' log
    Console.WriteLine("setup test")
  End Sub

  <Test()> _
  Public Sub demo()
    ' log écran
    Console.WriteLine("début test")
```

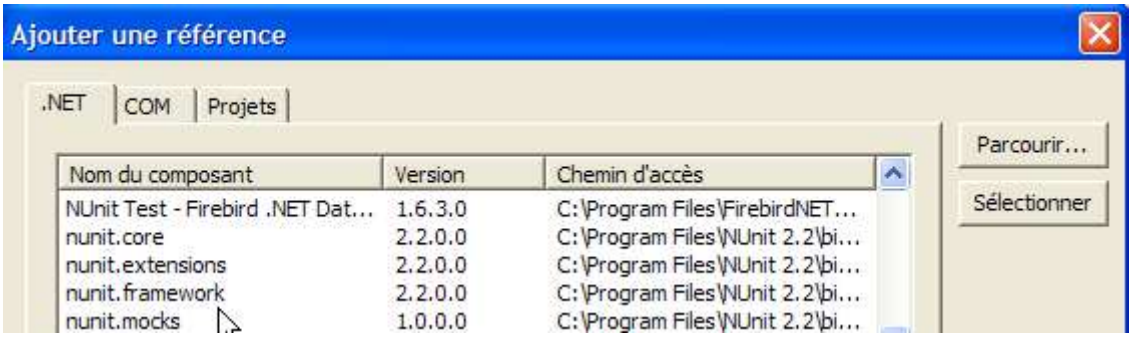
```
    ' init personne1
    With personne1
      .nom = "paul"
      .age = 10
    End With
    ' tests
    Assert.AreEqual("paul", personne1.nom)
    Assert.AreEqual(10, personne1.age)
    ' log écran
    Console.WriteLine("fin test")
  End Sub

  <TearDown()> _
  Public Sub destroy()
    ' suivi
    Console.WriteLine("teardown test")
  End Sub

End Class
```

Plusieurs choses sont à noter :

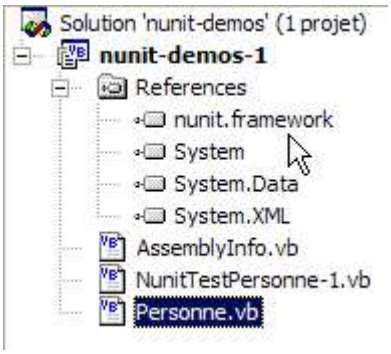
- les méthodes sont dotées d'attributs tels <Setup()>, <TearDown()>, ...
- pour que ces attributs soient reconnus, il faut que :
  - le projet référence l'assembly [nunit.framework.dll]
  - la classe de test importe l'espace de noms [NUnit.Framework]

La référence est obtenue en cliquant droit sur [References] dans l'explorateur de solutions :





L'assembly [nunit.framework.dll] doit être dans la liste proposée si l'installation de [Nunit] s'est bien passée. Il suffit de double-cliquer sur l'assembly pour l'ajouter au projet :



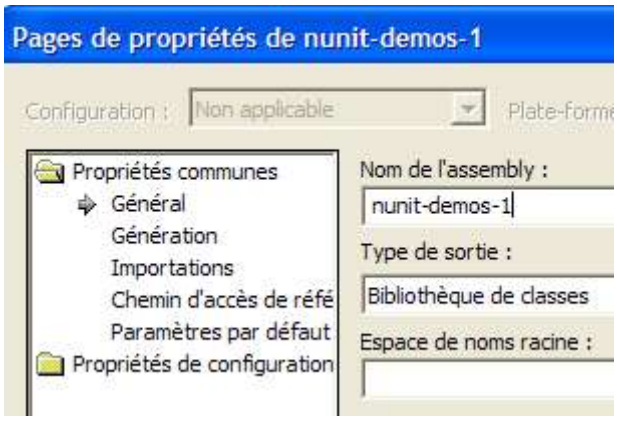
Ceci fait, la classe de test [NunitTestPersonne] doit importer l'espace de noms [NUnit.Framework] :

```
Imports NUnit.Framework
```

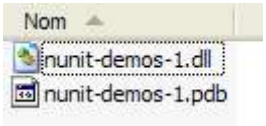
Les attributs de la classe de test [NunitTestPersonne] doivent alors être reconnus.

- l'attribut <Test()> désigne une méthode à tester
- l'attribut <Setup()> désigne la méthode à exécuter avant chaque méthode testée
- l'attribut <TearDown()> désigne la méthode à exécuter après chaque méthode testée
- la méthode **Assert.AreEqual** permet de tester l'égalité de deux entités.Il existe de nombreuses autres méthodes de type **Assert.xx**.
- l'utilitaire NUnit arrête l'exécution d'une méthode testée dès qu'une méthode [Assert] échoue et affiche un message d'erreur. Sinon il affiche un message de réussite.

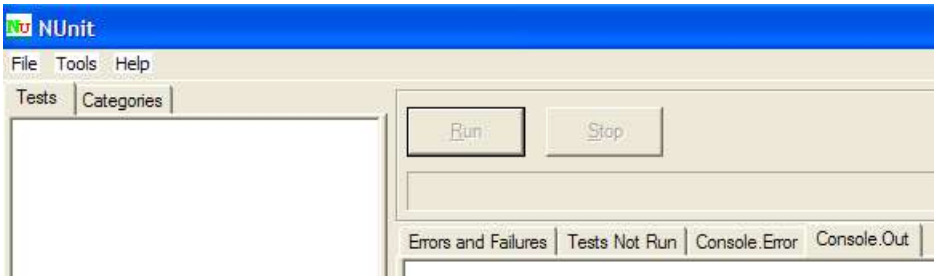
Configurons notre projet pour qu'il génère une DLL :



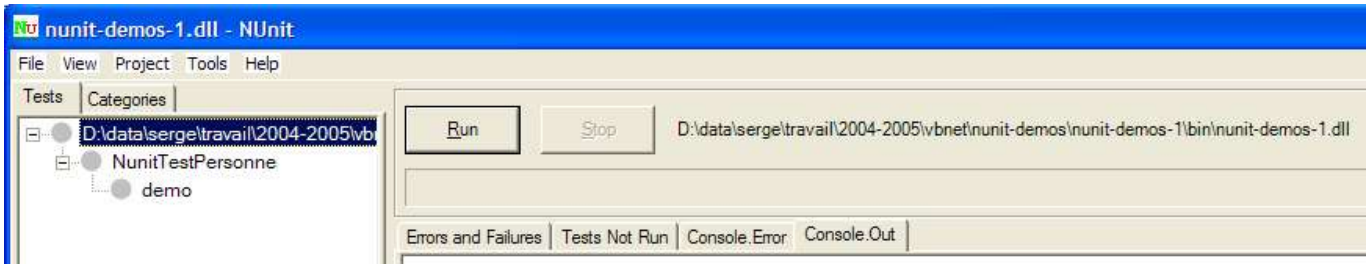
La DLL générée s'appellera [nunit-demos-1.dll] et sera placée par défaut dans le dossier [bin] du projet. Générons notre projet. Nous obtenons dans le dossier [bin] :



Lançons maintenant l'utilitaire de test graphique Nunit. Rappelons qu'il se trouve dans <Nunit>\bin et qu'il s'appelle [nunit-gui.exe]. <Nunit> désigne le dossier d'installation de [Nunit]. On obtient l'interface suivante :

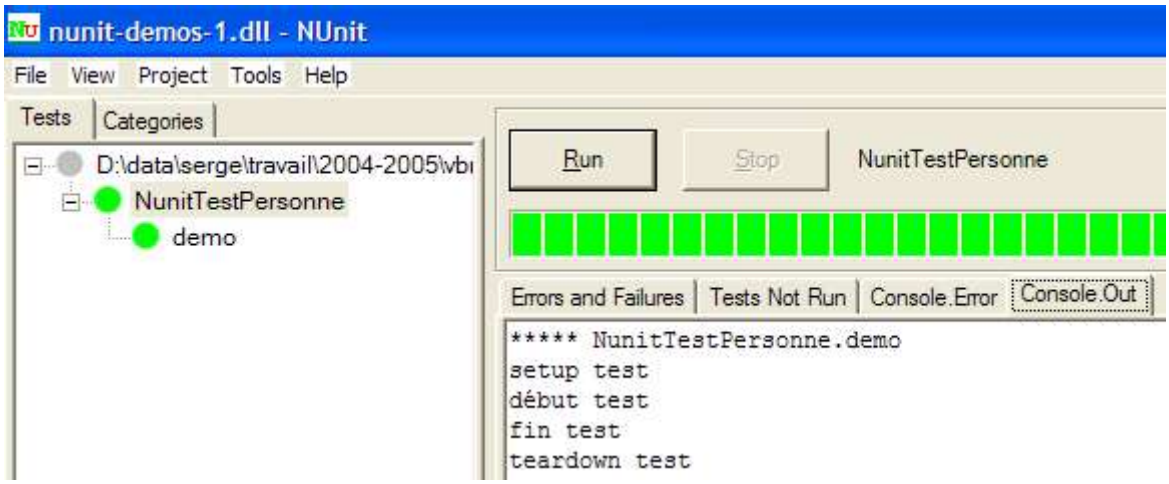


Utilisons l'option de menu [File/Open] pour charger la DLL [nunit-demos-1.dll] de notre projet :



[Nunit] est capable de détecter automatiquement les classes de test qui se trouvent dans la DLL chargée. Ici, il trouve la classe [NunitTestPersonne]. Il affiche alors toutes les méthodes de la classe ayant l'attribut <Test()>. Le bouton [Run] permet de lancer

les tests sur l'objet sélectionné. Si celui-ci est la classe [NunitTestPersonne], toutes les méthodes affichées sont testées. On peut demander le test d'une méthode particulière en la sélectionnant et en demandant son exécution par [Run]. Demandons l'exécution de la classe :



Un test réussi sur une méthode est symbolisé par un point vert à côté de la méthode dans la fenêtre de gauche. Un test raté est symbolisé par un point rouge.

La fenêtre [Console.Out] à droite montre les affichages écran produits par les méthodes testées. Ici, nous avons voulu suivre le déroulement d'un test :

```
1. setup test
2. début test
3. fin test
4. teardown test
```

- la ligne 1 montre que la méthode d'attribut <Setup()> est exécutée avant le test
- les lignes 2-3 sont produites par la méthode [demo] testée (voir le code plus haut)
- la ligne 4 montre que la méthode d'attribut <TearDown()> est exécutée après le test

Table des matières