WCF Comment éviter de décorer avec [ServiceKnownType]WCF - Comment éviter de décorer les contrats de service avec [ServiceKnownType] ou [KnownType] avec l'attribut `[EmbedTypeInSerializer]` ?

Silver Nakache 04 Juillet 2007.

Introduction

Lors du design des classes et des interfaces, il est d'usage de découpler les implémentations des contrats.

En effet, définir un ou plusieurs contrats formalise le protocole qui doit être mis en place pour établir une discussion entre les différentes couches logicielles. Dans une approche véritablement industrielle, l'architecte veille à ce que chaque couche puisse être validée par des implémentations différentes : des « mocks » dans un premier temps, des implémentations réelles bien sûr, des implémentations optimisées par la suite. Il veille aussi à conserver une architecture saine en adressant la problématique de « Levelization » du code.

WCF permet d'atteindre une agilité certaine dans la manière d'exprimer des contrats de services et permet de formaliser clairement la manière de travailler entre les différentes équipes de développement.

Dans l'idéal, ce que l'on souhaite exprimer, c'est la possibilité de répartir les couches logiques sur des couches physiques en jouant simplement avec le 'B' de WCF c'est-à-dire le 'Binding' pour s'adapter à l'infrastructure lors du déploiement : Serveur de production, Serveur de développement, Système embarqué, etc …

Il en va également de l'organisation des projets :

Pour atteindre le niveau d'agilité requis, le 'packaging' des projets est évidemment primordial, on retrouvera donc :

- Définition des Contrats de Services WCF : Des Projets/Assemblies contenant uniquement la définition du protocole, c'est-à-dire les contrats de service.
- Implémentations des Contrats de Services WCF : Des Projets/Assemblies contenant uniquement les implémentations de ces contrats de service.

- Définition du domaine Métier : Des Projets/Assemblies contenant uniquement la définition des objets métiers, permettant le couplage faible.
- Implémentation du domaine Métier : Des Projets/Assemblies contenant uniquement les implémentations des objets métiers : Mock Objets, Implémentation réelle, etc…

Et bien sûr, un projet 'WCF Host' qui permettra d'héberger les contrats de services par différents vecteurs : Service NT, Console, WinForm, WAS, IIS, etc …

Il est donc naturel d'exprimer un contrat de service sous une forme débarrassée d'implémentation, en voici un exemple :

```
[ServiceContract]
public interface IMyContract
{
   [OperationContract]
   IMyObject GetMyObject();

}
```

Le contre-exemple serait :

```
[ServiceContract]
public interface IMyContract
{
    [OperationContract]
    MyObject GetMyObject();

}
```

Identification de la problématique

Les sous-systèmes de Serialization/Déserialization WCF sont désormais très aboutis, les leçons sur les types difficiles à serializer du XMLSerializer ou du BinarySerializer sont maintenant très correctement adressés par les nouveaux « DataContractSerializer » et « NetDataContractSerializer ».

Ce n'est donc plus la Serialization des types qui pose problème mais leur transport.

Je m'explique :

Pour ceux qui étaient familiés avec .Net Remoting, le pattern Proxy/Stub permettait d'obtenir un véritable 'transparent-proxy' côté client, c'est-à-dire que lors du déploiement seul la définition des

interfaces était nécessaire. Le.Net Remoting Marshalling/Unmarshalling permettait de matérialiser l'appel sur le serveur.

WCF a été pensé dans une optique SOA, il en résulte des différences de fonctionnement dans la mécanique de Marshalling/Unmarshalling. Lors du Unmarshalling, le sous-système de désarialization de WCF a besoin d'identifier l'implémentation qui se cache derrière l'interface du contrat de service : **MyObject** derrière **IMyObject** dans notre cas.

WCF définit l'attribut `[ServiceKnownType(typeof(…))]` ou `[KnownType(typeof(…))]` permettant d'informer la sous-couche WCF de l'implémentation qui peut être potentiellement transportée.

- Première remarque, cela impose au développeur,au fournisseur de service, d'être exhaustif dans la définition des implémentations que l'interface est susceptible de transporter, et cela se complique d'autant plus quand l'objet retourné contient une « grappe » d'objets imbriqués, en voici un exemple :

```
namespace ContractServicesInterfaceDefinition
{
    [ServiceContract]
    [ServiceKnownType(typeof(MyObject))]
    [ServiceKnownType(typeof(MyMockObject))]
    [ServiceKnownType(typeof(MyOptimisedObject))]
    public interface IMyContract
    {
        [OperationContract]
        IMyObject GetMyObject();

    }
}
```

- Deuxième remarque, en termes d'organisation, l'utilisateur de service est amené à enrichir le protocole, l'interface de service en l'occurrence, demandant ainsi au fournisseur de service de compléter ses besoins. Il recompile donc le projet 'Définition des Contrats de Services WCF ' mais se voit pollué par une dépendance, 'les implémentations' dont il n'a que faire.

Comment donc permettre aux équipes de développeurs d'interagir sans contention ?
Comment éviter l'inflation des attributs `xxxKnownType` du côté du fournisseur de service ?

Solution Technique

Observons maintenant le résultat d'une sérialisation avec les deux nouveaux « serializer » fournis avec WCF.

Avec le DataContractSerializer :

```
- <MyObject xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
      xmlns="http://schemas.datacontract.org/2004/07/DomainImplementation">
  <_CircularRef i:nil="true" />
  <_MyString>Hello world</_MyString>
    </MyObject>
```

Avec le NetDataContractSerializer

```
- <MyObject xmlns:i="http://www.w3.org/2001/XMLSchema-instance" z:Id="1"
      z:Type="DomainImplementation.MyObject" z:Assembly="DomainImplementation, Version=1.0.0.0,
      Culture=neutral, PublicKeyToken=null"
      xmlns:z="http://schemas.microsoft.com/2003/10/Serialization/"
      xmlns="http://schemas.datacontract.org/2004/07/DomainImplementation">
  <_CircularRef z:Ref="1" i:nil="true" />
  <_MyString z:Id="2">Hello world</_MyString>
    </MyObject>
```

Il apparaît clairement que le NetDataContractSerializer est capable de transmettre le type de l'objet à transporter 'at Runtime'.

Maintenant l'objectif est de d'imposer à la sous-couche de sérialisation WCF à utiliser systématiquement le NetDataContractSerializer plutôt que les autres serializer. Pour des raisons sortant du périmètre de cet article, la sous-couche de serialization WCF ne sélectionne pas systématiquement le serializer adéquat.

Il n'existe pas non plus de moyen 'B' Binding de sélectionner le serializer à utiliser.

Seul recours, fabriquer un nouvel Attribut de contrat de service qui invitera à la sous-couche WCF d'appliquer le bon Serializer. Regardons maintenant la solution envisagée :

```
namespace ContractServicesInterfaceDefinition
{
    [ServiceContract]
    public interface IMyContract
    {
        [OperationContract]
```

```
        [EmbedTypeInSerializer]
        IMyObject GetMyObject();

    }
}
```

Dans cette solution, nous déléguons complètement le travail de recherche des implémentations transportées au NetDataContractSerializer. Nous changeons dynamiquement le comportement de la sous-couche de Serialization au moment même ou le système recherche le comportement à appliquer pour sa déserialization.

Implémentation du nouvel attribut `[EmbedTypeInSerializer]` **:**

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Xml;
using System.Xml.Serialization;
using System.ServiceModel;
using System.ServiceModel.Channels;
using System.Runtime.Serialization;
using System.ServiceModel.Description;

namespace WCFCustomAttributes
{

    [AttributeUsage(AttributeTargets.Method)]
    public class EmbedTypeInSerializerAttribute : Attribute, IOperationBehavior
    {
#region IOperationBehavior Members

        public void AddBindingParameters(OperationDescription description,
BindingParameterCollection parameters)
        {

        }

        public void ApplyClientBehavior(OperationDescription description,
System.ServiceModel.Dispatcher.ClientOperation proxy)
        {
            ForceNetDataContractSerializerOperationBehavior(description);
        }

        public void ApplyDispatchBehavior(OperationDescription description,
System.ServiceModel.Dispatcher.DispatchOperation dispatch)
        {
            ForceNetDataContractSerializerOperationBehavior(description);
        }

        public void Validate(OperationDescription description)
        {

        }

#endregion
```

```
        private static void ForceNetDataContractSerializerOperationBehavior(OperationDescription
description)
        {
            DataContractSerializerOperationBehavior dcsOperationBehavior =
description.Behaviors.Find<DataContractSerializerOperationBehavior>();
            if (dcsOperationBehavior != null)
            {
                description.Behaviors.Remove(dcsOperationBehavior);
                description.Behaviors.Add(new
NetDataContractSerializerOperationBehavior(description));
            }
        }

        public class NetDataContractSerializerOperationBehavior :
DataContractSerializerOperationBehavior
        {
            public NetDataContractSerializerOperationBehavior(OperationDescription
operationDescription) : base(operationDescription) { }
            public override XmlObjectSerializer CreateSerializer(Type type, string name, string
ns, IList<Type> knownTypes)
            {
                return new NetDataContractSerializer();
            }

            public override XmlObjectSerializer CreateSerializer(Type type, XmlDictionaryString
name, XmlDictionaryString ns, IList<Type> knownTypes)
            {
                return new NetDataContractSerializer();
            }
        }
    }
}
```

Nous atteignons ainsi notre objectif de simplicité côté définition du contrat de service, et de 'dialogue' côté équipe de développement.

Il n'en reste pas moins que pour permettre la deserialization côté client, WCF cherchera à 'binder' le type indiqué lors de la 'réhydratation':

… z:Type="**DomainImplementation.MyObject**" z:Assembly="**DomainImplementation, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null**" …

Le mécanisme naturel de 'probing' de .Net se met en place et cherchera à charger, lors de la réhydratation de l'objet, l'implémentation côté client : les binaires devront donc être déployés au pied l'application.

Et par la suite …

Pour beaucoup, notamment ceux qui sont familiers avec .Net Remoting, l'obligation de déployer une implémentation côté client apparaît contraignante. En théorie, cela ne devrait pas être nécessaire.

On peut donc imaginer que Microsoft résoudra ce problème en mettant en place un système de download automatique via une URI intercallée dans le message de serialization même,tel qu'un composant web par exemple.

En augmentant la syntaxe comme suit :

… z:Type="**DomainImplementation.MyObject**" z:Assembly="**DomainImplementation, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null**" **z:URI= "\\MACHINE\Download\DomainImplementation.dll"** …

Il devrait être possible de télécharger le composant juste avant l'étape de probing.

A suivre ….

Glossaire :

Levelization : Levelization is a simple organizational principle or metric applied to code, first promoted by the great John Lakos in his landmark book "Large Scale Software Design in C++"

WCF: Windows Communication Fundation