# Introduction : un modèle simple

Le modèle utilisé ne présente pas de complexité particulière en termes de compréhension, mais possède néanmoins un SEAM évident et offre la possibilité de jouer avec la covariance (entre autre).

Notre Client s’appelle Albert. C’est un bon client : ces spécifications changent tout le temps mais il ne revient pas dessus ☺. Nous commençons par une modélisation simple. L’objectif est de réaliser ce système étape par étape, en y introduisant des complexités croissantes, dont un container.

On se place dans un système existant avec un utilisateur courant. Cet utilisateur possède un identifiant unique, un nom, et un niveau d’authentification (certaines opérations peuvent nécessiter le niveau d’authentification HighlySecure).

public enum AuthenticationLevel

{

None = 0,

Authenticated = 1,

HighlySecure = 2

}

public interface ICurrentUser

{

string UserId { get; }

string UserName { get; }

AuthenticationLevel AuthenticationLevel { get; }

}

Albert veut que le Système puisse proposer des Services à l’Utilisateur courant, ces Services étant offerts par des Partenaires divers, un même Service pouvant être proposé par plusieurs Partenaires et un Partenaire peut proposer plusieurs Services. On ne souhaite pas mettre le Partenaire en avant : ce qui nous intéresse au niveau Métier, ce sont les Services offerts. L’abstraction IPartner est donc purement informative et limitée au strict nécessaire. (Compte tenu de la problématique Métier, on ne peut décemment pas oublier cette entité ☺).

public interface IPartner

{

string UniqueName { get; }

Uri MainUri { get; }

}

Les Services sont représentés par des interfaces, par exemple un IBookStore ou un ICabOrderer (ou un IEatAtHome).

Chaque type de Service possède ainsi une interface métier qui lui est parfaitement adapté, MAIS chaque Partenaire doit pouvoir apporter sa propre implémentation du Service (ce qui semble logique).

La première approche est non-générique, mais permet de poser les bases de l’articulation.

public interface IPartnersServicesProvider

{

IEnumerable<IPartnerServiceGateway> GetPartnerServices();

}

public interface IPartnerServiceGateway

{

IPartner Partner { get; }

IEnumerable<string> Categories { get; }

int PerUseCost { get; }

Type ServiceType { get; }

object ObtainService();

}

La notion de Gateway encapsule la méta-information commune aux différents Services auxquels l’utilisateur courant peut accéder.

Les Catégories permettent de filtrer les Services (LINQ) en plus du Type de Service et fournit l’implémentation du Service, si besoin est, sous la forme d’un objet non-typé.

Reste une question : où est l’information relative aux Services eux-mêmes ? On introduit l’abstraction suivante (« Storage » évite la polémique du « Repository » ☺) qui fournit la donnée brute du Système :

public interface IPartnerStorage

{

IEnumerable<PartnerServiceData> GetPartnerServicesData();

}

public class PartnerServiceData

{

public string PartnerUniqueName { get; set; }

public string PartnerMainUri { get; set; }

public string CommaSepratedCategories { get; set; }

public int PerUseCost { get; set; }

public string AssemblyQualifiedServiceTypeName { get; set; }

public string AssemblyQualifiedImplementation { get; set; }

}

Le PartnerServiceData est un DTO basique. Rien n’est spécifié à ce niveau sur la relation entre un utilisateur et la possibilité d’un Service. Lorsque le besoin évoluera, il n’est pas impossible que l’implémentation du Storage filtre les Services…

(A noter ici la possibilité de découvrir dynamiquement des implémentations de Services avec des attributs pour déclarer la méta-information.)

# Première implémentation

Tout est ridiculement simple, c’est à survoler très rapidement. Sauf un point, le point de création des Services dans les PartnerServiceGateway  :

public object ObtainService()

{

Type t = Type.GetType( \_data.AssemblyQualifiedImplementation );

return Activator.CreateInstance( t );

}

C’est ici que tout commence.

A partir de ce modèle, on peut ajouter :

- des contraintes User / Services (et/ou User / Partenaires)

- une implémentation plus sympa avec des génériques (et peut-être bien montrer les Génériques).

- faire de la Gateway elle-même un Composition Root (en associant par exemple une implémentation de Gateway PAR Partenaire, avec des dépendances quelconques à d’autres Services). Je vois deux options :

- Avec une évolution du Modèle en ajoutant un AssemblyQualifiedGatewayType au PatternServiceData (mais ce serait répété le pattern déjà existant☹).

- Avec une « named registration » et une convention sur le namespace ☺ (un point pour toi) !! Auquel cas, on fait de la Gateway existante une classe de base abstraite (qui prend la data en paramètre) et on peut y caser le DependencyOverride  (un partout ☺ ).

- On peut adapter ce système à un autre Client qui fournit un déjà un User mais dont le UserId est un entier : on introduit les « Single Contracts ».

- (Un détail : je peux aussi parler du Type.GetType qui renvoie des exceptions qui ne sont pas toujours des TypeLoadExceptions ni ne respecte le flag throwOnError et de la correction que j’utilise) :

/// <summary>

/// Simple implementation that corrects <see cref="Type.GetType(string,bool)"/>.

/// </summary>

/// <param name="assemblyQualifiedName">The assembly qualified name of a type.</param>

/// <param name="throwOnError">

/// True to ALWAYS throw a <see cref="TypeLoadException"/> if the type is not found.

/// False prevents any exception to be thrown and simply returns null.

/// </param>

/// <returns>The type or null if not found and <paramref name="throwOnError"/> is false.</returns>

/// <exception cref="TypeLoadException">

/// When <paramref name="throwOnError"/> is true, always throws this kind of exception.

/// The original error (not a <see cref="TypeLoadException"/>) is available in the <see cref="Exception.InnerException"/>.

/// </exception>

public Type GetType( string assemblyQualifiedName, bool throwOnError )

{

try

{

return Type.GetType( assemblyQualifiedName, throwOnError );

}

catch( Exception ex )

{

if( !throwOnError ) return null;

if( ex is TypeLoadException ) throw;

throw new TypeLoadException( String.Format( R.ExceptionWhileResolvingType, assemblyQualifiedName ), ex );

}

}

- Il reste à glisser là-dedans des problématiques de Lifetime. Je pense que l’on pourrait injecter ce modèle dans une application ASP.Net MVC et traiter le Storage en IDisposable.   
Qu’en penses-tu ?