TP .Net

Système de Radar

Lemettre Arnaud

Version 2.0

10 Pages

02/04/2009

EPITA-MTI2014-NET-TP-Radar

Multimédia
Technologies de l'Information

ÉCOLE D'INGÉNIEURS EN INFORMATIQUE

Propriétés du document

Auteur	**Lemettre Arnaud**
Version	2.0
Nombre de pages	10
Références	EPITA-MTI2014-NET-TP-Radar

Historique du document

Date de vision	Version	Auteur	Changements
27/02/2009	0.1	Lemettre	création
01/05/2011	2.0	Lemettre	Mise à jour du sujet

Site de référence

description	url
Site MTI	
Blog MTI	

Sommaire

Introduction

Le but de cette série de TP sera la réalisation d'un système de radars automatiques. Ce tp se décompose en plusieurs parties. Toutes les semaines vous aurez une partie à rendre. Cette partie sera évaluée. Toute fois si pour une raison quelconque vous n'avez pas réussi à faire convenablement cette partie une correction sera postée une journée avant le début du prochain TP. Vous aurez à charge de l'intégrer dans votre code si celui ne produit pas le fonctionnement attendu.

Ce travail est à faire individuellement, tout code similaire sur deux personnes sera considéré comme un travail non rendu et non négociable.

Durant cette série de TP les notions suivantes seront abordées :

- Unity
- Les winforms
- Mef
- Les workflows

Pour l'ensemble du sujet login_l correspondra à votre login EPITA.

Chaque partie à rendre ne demande pas plus de 3h de travail par semaine, Bonne chance ;)

Contexte

Dans ce TP nous allons aborder les concepts des applications lourdes. Bien entendu tous ces concepts peuvent être également appliqués aux Web applications. Le but de ce TP est donc de réaliser une application simulant un système de radar automatique. La mise à jour des données se fera par des appels simulés. Chaque radar calculera si la voiture dépasse la vitesse limite et notifiera cela à la simulation.

L'ensemble de la solution devra utiliser le Framework 4.5 de .Net.

Le rendu de chaque lot, correspond au rendu de chaque partie.

Ce TP sera corrigé par **moulinette**.

Partie 1

Renseignements

Avec ce que nous avons vu en cours, vous devez réaliser le système qui s'occupera de simuler le flot des voitures. Mais aussi la façon dont devra interagir l'ensemble du système. Cette partie sera l'occasion de constituer la data access mais aussi la couche de business management.

Travail à Faire

Dbo

Dans un premier temps il nous faut des objets métiers, dans notre cas une voiture. Cette classe devra être dans une bibliothèque de classe nommée Dbo. Cette classe se nommera Car. Ayant pour propriétés :

Getter / setter	Type
NumberId	Int
Speed	Int

Cette classe aura pour namespace :

Dbo

Il faudra également rajouter un override sur la méthode ToString(). Celle-ci devra renvoyer une concaténation de numberId et de Speed, ayant cette représentation :

```
NumberId : Speed km/h
```

DataAccess

Maintenant nous devons créer la dataaccess. Pour cela, créer un projet de type bibliothèque de classes nommée DataAccess.

Vous devez créer ensuite une interface à la racine ayant pour fonction :

```
List<Dbo.Car> GetListCar();
```

Et nommée IData

Cette data access devra permettre d'accéder aux données. La particularité de celle-ci est de pouvoir déterminer s'il faut choisir entre un service de base de données ou un service. Pour réaliser ces deux types d'accès nous allons simuler les accès à travers des classes mock.

Pour cela créer un nouveau dossier nommé Mock qui contiendra deux classes :

- MockDatabase
- MockService

Ces classes devront implémenter l'interface suivante :

IData

La classe MockDatabase implémentera donc la méthode GetListCar qui renverra 20 voitures.

Les valeurs NumberId initialisées devront être comprises entre 1000 et 2000. Chacun des nombres devra être différent. Les valeurs Speed initialisées devront être comprises entre 30 et 180.

La classe MockService implémentera donc la méthode GetListCar qui renverra 10 voitures.

Les valeurs NumberId initialisées devront être comprises entre 1000 et 2000. Chacun des nombres devra être différents. Les valeurs Speed initialisées devront être comprises entre 30 et 180.

Le choix entre ces deux classes devra se faire au travers de unity. Pour cela créer un dossier nommé Config qui contiendra une classe nommée ContainerInjection et qui implémentera UnityContainer. Cette classe aura pour namespace : DataAccess.Config

Vous devrez implémenter une fonction qui permettra de configurer le container. Celle-ci devra se nommer Configure et aura pour signature :

Nom de la fonction	**Configure**	
Paramètre en entrée	bool	Permet de déterminer si on utlise le service
Paramètre en sortie	void	
Espace de nom	DataAccess.Config	
Classe	ContainerInjection	
Portée	public	

Signature:

```
public void Configure(bool isMockService)
```

Lorsque le paramètre booléen sera true, l'instance renvoyée devra être MockService sinon cela sera MockDatabase.

Pour récupérer les données, tous les appels devront passer par une classe à la racine du projet nommée Car. Celle-ci aura un constructeur :

Nom de la fonction	**Car**	
Paramètre en entrée	bool	Par défaut, ce paramètre est true.
Paramètre en sortie	NA	
Espace de nom	DataAccess	
Classe	Car	
Portée	public	

Cette classe devra comporter une propertie nommée :

Getter / setter	**Type**
IsService	Bool

Ainsi qu'une méthode GetListCar qui permettra de retourner une liste de voitures. Cette méthode devra pouvoir renvoyer les résultats en utilisant unity et le paramétrage de IsService.

Nom de la fonction	**GetListCar**	
Paramètre en entrée		

Paramètre en sortie	List<Dbo.Car>	L'ensemble des voitures auto générées
Espace de nom	DataAccess	
Classe	Car	
Portée	public	

A la fin de cette partie le projet DataAccess devra être équivalent à l'image ci-dessous :

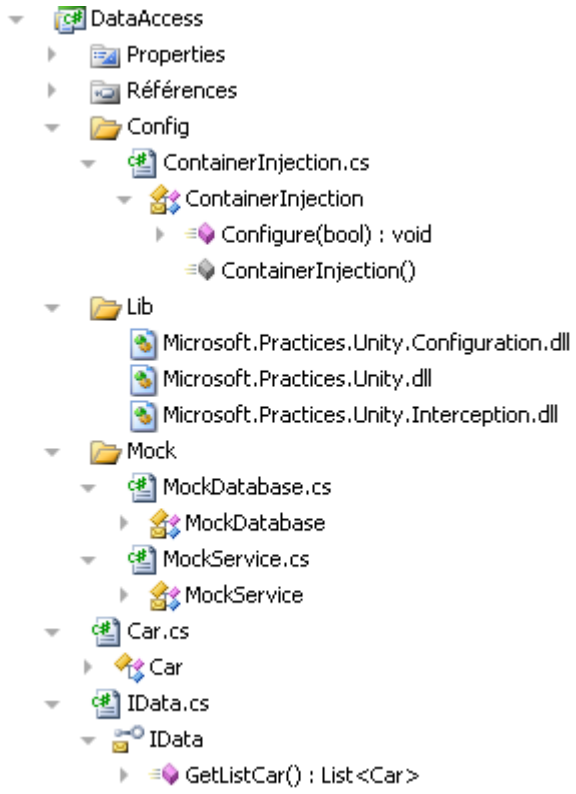

Figure 1 : Architecture du projet DataAccess

Interface

Avant de continuer de remonter les couches, nous devons rajouter un projet Interface de type bibliothèque de classes contenant les interfaces au sens programmation de notre projet. Pour cela vous devez créer une nouvelle interface respectant le formalisme suivant :

Nom de l'interface	IEngine
Espace de nom	Interface
Portée	Public
Liste des méthodes	void InitModule(List<Dbo.Car> list, int initSpeed); List<Dbo.Car> GetData(); List<Dbo.Car> Compute();

Le projet doit être de la forme suivante :

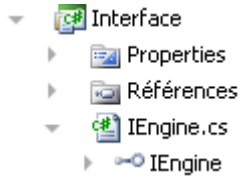

Figure 2 : Architecture du projet interface

BusinessManagement

Maintenant, nous allons créer la couche métier qui abritera la logique de calcul des radars.

Dans un 1er temps, nous allons créer une couche métier sous la forme d'un plugin.

La structure de ce plugin devra être dans un projet de bibliothèque de classes nommé : BusinessManagement1 et contenir à sa racine une classe nommée : Engine. Celle-ci devra implémenter IEngine déclarée dans le projet Interface.
Comme cela va devenir une extension de notre logiciel avec le framework MEF, cette classe devra être décorée avec l'attribut de classe Export, ainsi qu'avec un attribut de metadata ayant comme valeur :

IsTrueEngine , false

Cette classe devra comporter deux attributs privés de classe de portée privée :

```
List<Dbo.Car> _list;
int _speedLimit;
```

Cette classe comportera 3 méthodes :

Nom de la fonction	**InitModule**	
Paramètre en entrée	`List<Dbo.Car>`	La liste de voitures à tester
	`int`	La vitesse limite du radar
Paramètre en sortie	`void`	
Espace de nom	BusinessManagement1	
Classe	Engine	
Portée	public	

Cette méthode initialisera les variables de classe déclarées précédemment.

Nom de la fonction	**GetData**	
Paramètre en entrée		
Paramètre en sortie	`List<Dbo.Car>`	La liste des voitures à tester
Espace de nom	BusinessManagement1	
Classe	Engine	
Portée	public	

Cette méthode devra récupérer la liste de voitures fournies par la DataAccess au travers de la classe Car et mettre à jour la variable privée.

Nom de la fonction	**Compute**	
Paramètre en entrée		
Paramètre en sortie	`List<Dbo.Car>`	La liste des voitures dépassant la limite de vitesse
Espace de nom	BusinessManagement1	

Classe	Engine
Portée	public

Cette méthode devra retourner uniquement le 1er élément de la liste de voitures stockées dans la variable de classe. Si la liste est vide la méthode doit retourner *Null*. Bien qu'on ne retourne qu'un seul élément celui-ci doit être dans une liste (Modification pour le prochain TP).

Le projet doit être comparable à :

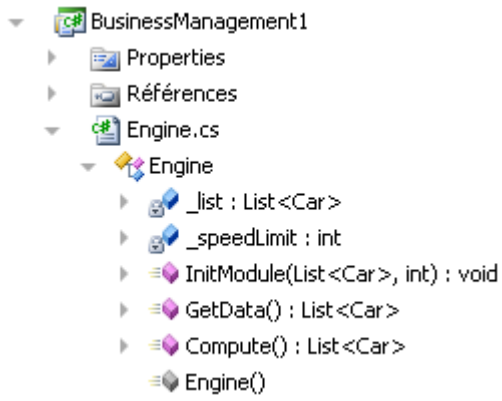

Figure 3 : Architecture du projet Business

Modalité de rendu

Les fichiers seront à rendre dans une tarball ayant pour nom :

login_l.zip

Cette tarball devra comprendre :

- Un dossier contenant la solution Visual Studio qui devra compiler.
  Nom : login_lTPRadar

Le tout à envoyer sur l'adresse mti.rendu.dotnet@gmail.com avec les balises suivantes :

[MTI2014][NET][login_l][Radar] partie 1