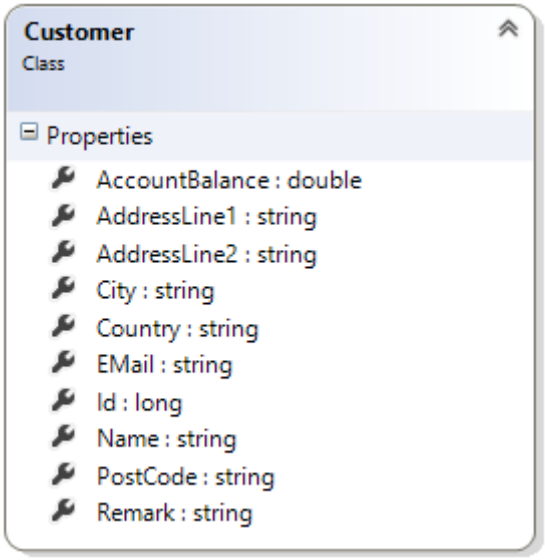
Laboratoire du 29 septembre – première problématique

Introduction

L'objectif de ce laboratoire est de vous faire découvrir une problématique qu'il vous faudra gérer dans vos projets futurs. Ce sera aussi l'occasion pour vous de vous familiariser avec un second mode de fonctionnement de l'ORM que vous avez manipulé dernièrement: Entity Framework. Enfin, vous aurez l'opportunité à travers ce laboratoire d'explorer un peu plus le framework de tests MSTest.

Consignes

Travaillez dans votre repository GIT. Si nécessaire, clonez le. S'il est déjà disponible sur votre machine, assurez-vous de partir d'une copie fraîche en réalisant un pull.
Créez une solution Visual Studio dans votre repository GIT. Ajoutez-y un projet de type class library (sous Classic Desktop).
Ajoutez ce projet à votre repository GitHUB.
Créez une classe Customer correspondant à cette représentation (les membres sont des propriétés auto-implémentées):



Cette classe représente votre modèle. Vous allez maintenant créer une base de données automatiquement sur base de ce modèle.

C'est Entity Framework qui va vous le permettre. Commencez donc par ajouter le package Entity Framework à votre projet via Nuget.

Créez une deuxième classe dans votre projet, nommez-là CompanyContext. Faites-la hériter de DbContext et ajoutez les using nécessaires. DbContext est une classe définie dans Entity Framework qui représente la manière d'accéder aux données.

Dans cette classe, créez un constructeur qui appellera le constructeur de la classe de base (DbContext) en lui passant en paramètre une ConnectionString qui décrira comment Entity Framework se connectera à votre base de données. Utilisez la version locale de SQL Server (LocalDb).

Toujours dans cette classe, créez une propriété générique de type DbSet<Customer> nommée Customers. Cette propriété représentera l'accès à la table Customer qui sera générée pour vous dans votre base de données.

Voici ce à quoi devrait ressembler votre classe CompanyContext.

```
using System;
using System.Collections.Generic;
using System.Data.Entity;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Model
{
    public class CompanyContext : DbContext
    {
        public DbSet<Customer> Customers { get; set; }
        public CompanyContext()
            : base(@"Data Source=(localdb)\MSSQLLocalDb;Initial Catalog=ConcurrencyDemo;")
        {

        }
    }
}
```

Compilez votre solution.

Créez ensuite un projet de tests unitaires. Ce projet devra référencer le premier car il en utilisera les classes. De plus, puisque nous allons tester le fonctionnement d'un objet qui hérite de DbContext et que DbContext est définie dans EntityFramework, vous aurez besoin d'une référence à EntityFramework dans votre projet de test. Ajoutez celle-ci, toujours via Nuget.
Créez ensuite dans le projet de tests une nouvelle classe nommée DbInitializer. Cette classe devra hériter de la classe générique DropCreateDatabaseAlways<T>. Le T correspond à votre classe DbContext (à savoir CompanyContext). Dans cette classe DbInitializer, faites un override de la méthode Seed. Dans cette méthode, créez une instance de la classe Customer et garnissez les propriétés de cette dernière. Ajoutez ensuite l'instance de Customer au contexte, en appelant la méthode Add que vous retrouverez par l'intermédiaire de la propriété Customers, comme illustré ci-dessous.

```
context.Customers.Add(customer);
```

Le contexte est garni avec une nouvelle instance de Customer, mais rien n'a encore été inséré en bas de données, cette nouvelle instance n'est qu'en mémoire. Afin que les ordres SQL correspondant aux opérations (qu'il s'agisse d'insertion, de modification, de suppression) soient générés et envoyés à la base de données, il faut appeler la méthode SaveChanges sur votre instance du contexte.

Votre DbInitializer devrait avoir la structure suivante.

```
using System;
using System.Collections.Generic;
using System.Data.Entity;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Model.Tests
{
    class DbInitializer:DropCreateDatabaseAlways<CompanyContext>
    {

        protected override void Seed(CompanyContext context)
        {
            Customer customer = new Customer()
            {
                Name = "Albert Dupont",
                AddressLine1 = "Rue des cerisiers, 16",
                City = "Arlon",
                Country = "Belgique",
                EMail = "info@me.com",
                Id = 3,
                Remark = "Ne pas avoir peur des chiens pour aller chez ce client",
                PostCode = "6700"
            };
            context.Customers.Add(customer);
            context.SaveChanges();
        }
    }
}
```

Ensuite, dans la classe créée par défaut dans ce nouveau projet (UnitTest.cs), créez une méthode publique nommée Setup() et décorez-la de l'attribut **TestInitialize**. TestInitialize sert à définir que la méthode décorée sera exécutée avant l'exécution des tests définis. Elle permet de mettre en place l'environnement de test. En l'occurrence, nous allons spécifier qu'avant tout test, la base de données doit être recrée et garnie avec un client. Cette définition est réalisée à l'aide des instructions illustrées ci-dessous..

```
[TestInitialize]
public void Setup()
{
    Database.SetInitializer(new DbInitializer());
    using (CompanyContext context = GetContext())
    {
        context.Database.Initialize(true);
    }
}
```

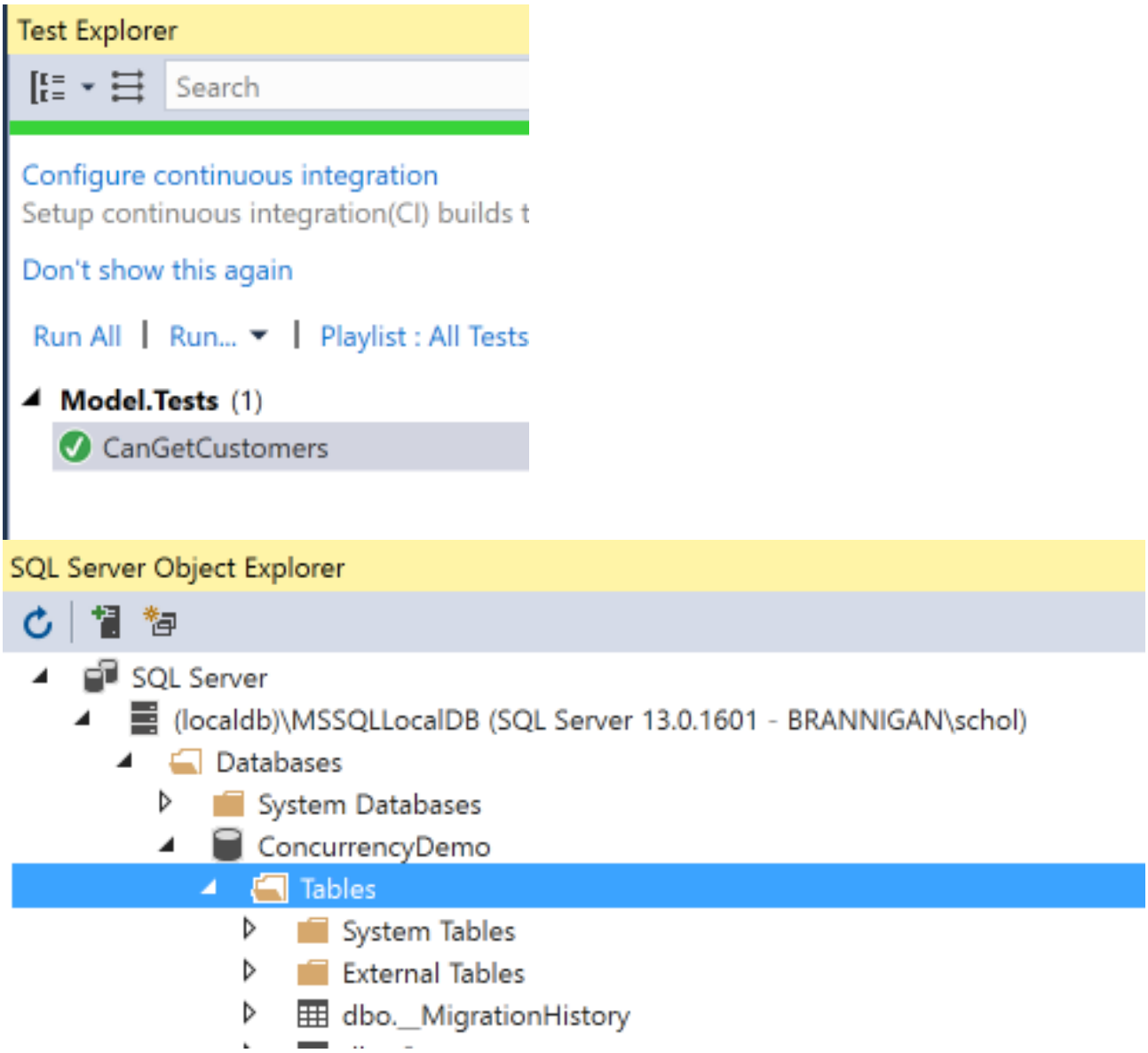
C'est EntityFramework qui offre cette fonctionnalité par le biais de ses initialiseurs[1]. C'est un aspect intéressant qui permet d'avoir des tests déterministes, car vous maîtrisez les données qui sont dans la base. Ce n'est évidemment pas un initialiseur souhaitable en production.

Créez ensuite un premier test unitaire, qui permet de s'assurer que la récupération des clients fonctionne. Ce test aura la structure suivante :

```
[TestMethod]
public void CanGetCustomers()
{
    using (var context = GetContext())
    {
        Assert.AreEqual(1, context.Customers.ToList().Count);
    }
}
```

La méthode GetContext est une factory, méthode utilitaire qui retourne une instance de CompanyContext. Dans le corps de la méthode, on compte le nombre de clients et on s'assure qu'il y en a bien 1. Ce devrait être le cas puisque notre initialiseur crée une base de données vierge et y ajoute un client avant chaque test. Compilez votre solution, puis lancez les tests (via le Test Explorer de Visual Studio). Ils devraient être au vert. Vous devriez de plus voir apparaître la DB en question dans votre SQL Server Object Explorer.

[1] http://www.entityframeworktutorial.net/code-first/database-initialization-strategy-in-code-first.aspx



Ce mode de fonctionnement d'Entity Framework est nommé "Code First": le développeur écrit le code et la base de données est générée sur base de celui-ci. A l'inverse, nous avons jusqu'ici utilisé le mode "Database First": du code est généré pour le développeur sur base d'une base de données existante.

Renommez votre première méthode de test afin de rendre son objectif plus parlant. Nommez la par exemple "InsertionFonctionnelle".

Ajoutez à votre solution le projet WPFClient qui a été mis à disposition sur GitHUB (solution labo 3). Copiez le répertoire de ce projet (pas celui de la solution), dans le répertoire de votre solution, au même niveau que les deux autres projets déjà existant. Ce projet devra référencer le projet contenant votre modèle (CompanyContext et la classe Customer) et devra également inclure une référence au package Nuget EntityFramework (pour les mêmes raisons que le projet de test nécessite ce package, voir plus haut).

Dans la méthode `MainWindow_Loaded` du code-behind de MainWindow.xaml, insérez les instructions permettant les opérations suivantes, dans cet ordre:

- Initialisation de la variable _context;
- Initialisation de la variable _customer : elle contiendra le seul client de la base de données.

- Le DataContext de “Formulaire” (voir code XAML) doit pointer sur la variable _customer.

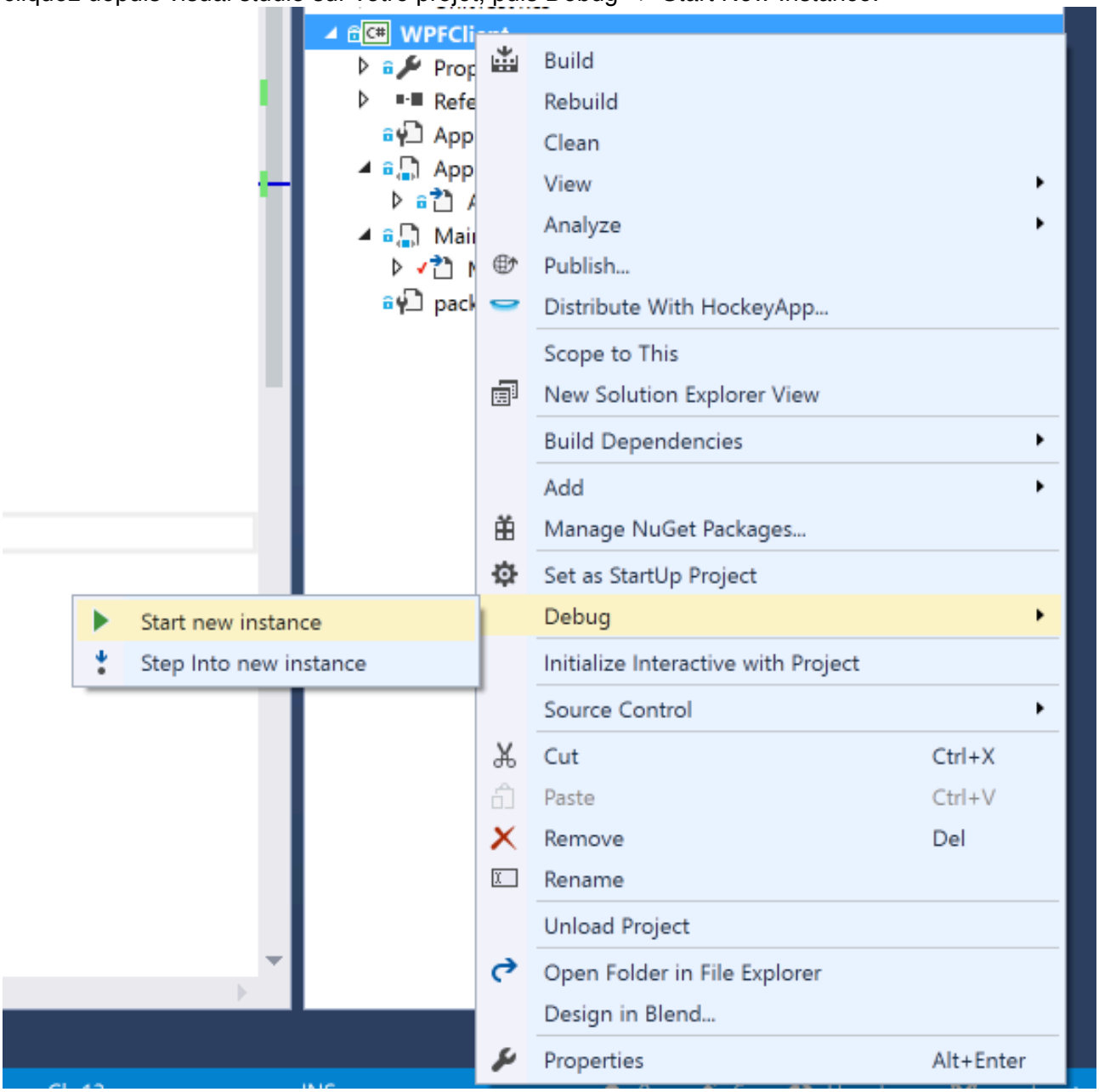
Lancez l’application, vous devriez alors voir apparaître le client que vous avez créé lors de vos tests.

Arrêtez le debug, puis modifiez encore le code-behind de MainWindow.xaml et ajoutez les instructions suivantes à la méthode `Button_Click`:

- Calcul du nouveau montant sur le compte du client en aditionnant le montant connu au montant spécifié dans la zone numérique `MontantAAjouterAuCompte`
- Sauvegarde des modifications

Exécutez votre application plusieurs fois en précisant un montant à ajouter et en vous servant de la connexion à votre base de données via le SQL Server Object Explorer afin de valider que la modification de la valeur sur le compte est bien sauvegardée en base de données.

Lancez ensuite deux instance de votre application WPF. Pour lancer plusieurs fois l’application, cliquez depuis visual studio sur votre projet, puis Debug => Start New Instance.



Une fois l'application lancée des deux côtés, spécifiez un montant différent sur vos écrans, puis sauvegardez.

Examinez alors le contenu de l'enregistrement que vous avez modifié à l'aide de SQL Server Object Explorer. Cela vous paraît-il cohérent? Que s'est-il passé ? Pouvez-vous modéliser cette situation dans un test automatisé ? Pouvez-vous ensuite trouver une solution au problème ?