Testing

Asp.net MVC

Testing

Le processus de création d'une application se focalise principalement sur les fonctionnalités qui doivent être développées afin de répondre à une demande, de résoudre un problème spécifique,…

Mais avant de mettre en production ladite application, il est impératif de s'assurer que ces fonctionnalités soient implémentées de manière fiable et robuste.

- **Les Tests Manuels**

  C'est la manière la plus facile de tester une application et la plus littérale : On lance l'application et on tente d'utiliser l'application comme un utilisateur « normal » le ferait.

  C'est pourtant la manière la moins robuste de tester et celle-ci a son lot de points négatifs.

  - Basé sur le jugement humain, ces tests sont subjectifs. Comment faire la différence entre le string 1 et l'int 1 sur l'écran?
  - L'ordinateur est plus efficace et plus précis. Le test manuel se fait au travers de l'UI et doit, pour pouvoir être intéressant, respecter des étapes dans un ordre précis et non pas comme le testeur pense que ça doit fonctionner
  - Les tests manuels prennent du temps

Testing

- **Les Test automatisés**

  Et si nous demandions à un logiciel de tester une logiciel?

  Il « suffit » d'écrire les tests de définir les résultats souhaités et laisser l'exécution des tests nous révéler les échecs et les succès.

Testing

Il existe trois catégories de tests permettant d'identifier les bugs :

- Unit Tests:

  Ils permettent de tests les fonctionnalités basiques. En mettant en place une série de tests unitaires, nous pouvons nous assurer que les aspects fonctionnels de notre application sont correctement codés

  Exemple : Vérifier le type de retour d'une fonctions

- Integration Tests :

  Ils permettent de vérifier que 1 ou 2 composants fonctionnent correctement ensemble. Ils peuvent par exemple être utilisés pour vérifier que deux classes interagissent correctement ensemble.

  Ils sont également utilisés afin de vérifier que toute l'application , incluant la base de données et les services externes, fonctionne et délivre un résultat attendu

- Acceptance Tests :

  Ils testent les fonctionnalités et les composants en ciblant les besoins énoncés par le « client »

Testing

Unit Test

Un test unitaire est généralement composé de trois phases :

1. *Arrange*

   Nous créons les instances, variables, …nécessaires pour le test. Nous assignons également des valeurs aux propriétés nécessaires.

   Les tests unitaires doivent être conçu afin de permettre l'automatisation de ceux-ci et donc de pouvoir les exécuter rapidement et facilement durant le cycle de vie de l'application.

   C'est très important car des bugs peuvent apparaitre durant le développement et les tests unitaires doivent nous assurer de la robustesse de notre code.

2. *Act*

   Nous appelons la fonctionnalité que nous désirons tester et nous stockons les résultats dans des variables afin de pouvoir vérifier celles-ci.

3. *Assert*

   Nous vérifions les résultats attendus. Si la vérification passe : le test réussi. Sinon, celui-ci échoue.

Testing

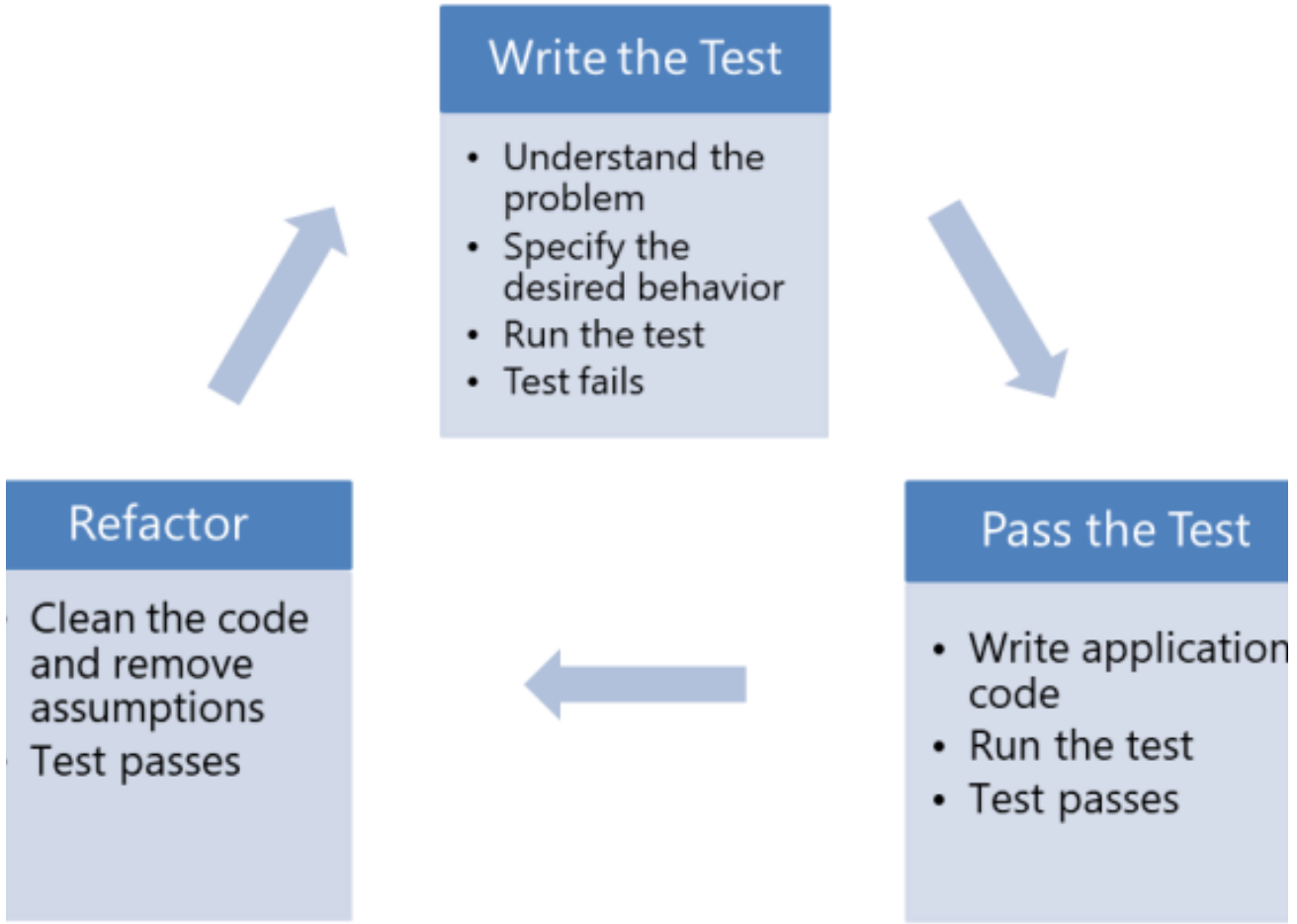
Test Driven

Il est de plus en plus fréquent d'utiliser les test unitaires dans la méthodologie de développement tel la méthodologie Agile, itérative ou Waterfall.

Mais une méthodologie spécifique est liée aux tests unitaires : Test Driven (TDD).

Les principes :

- Écrire le code avant les tests
- Avancer fonctionnalité par fonctionnalité
- Ecrire uniquement le code nécessaire pour passer les tests



Composants MVC et Unit Testing

Testing

Composants MVC et Unit Testing

Afin de permettre les test unitaires, il est nécessaire de tendre vers des composants évitant l'interdépendance forte : des « Loosely coupled Components ».

En Orienté objet, deux classes peuvent être « faiblement couplées » si une classe peut appeler une méthode de l'autre classe sans code spécifique à celle-ci.

Il est donc facile de remplacer une classe par un autre composant implémentant les même fonctionnalités.

Comment ?

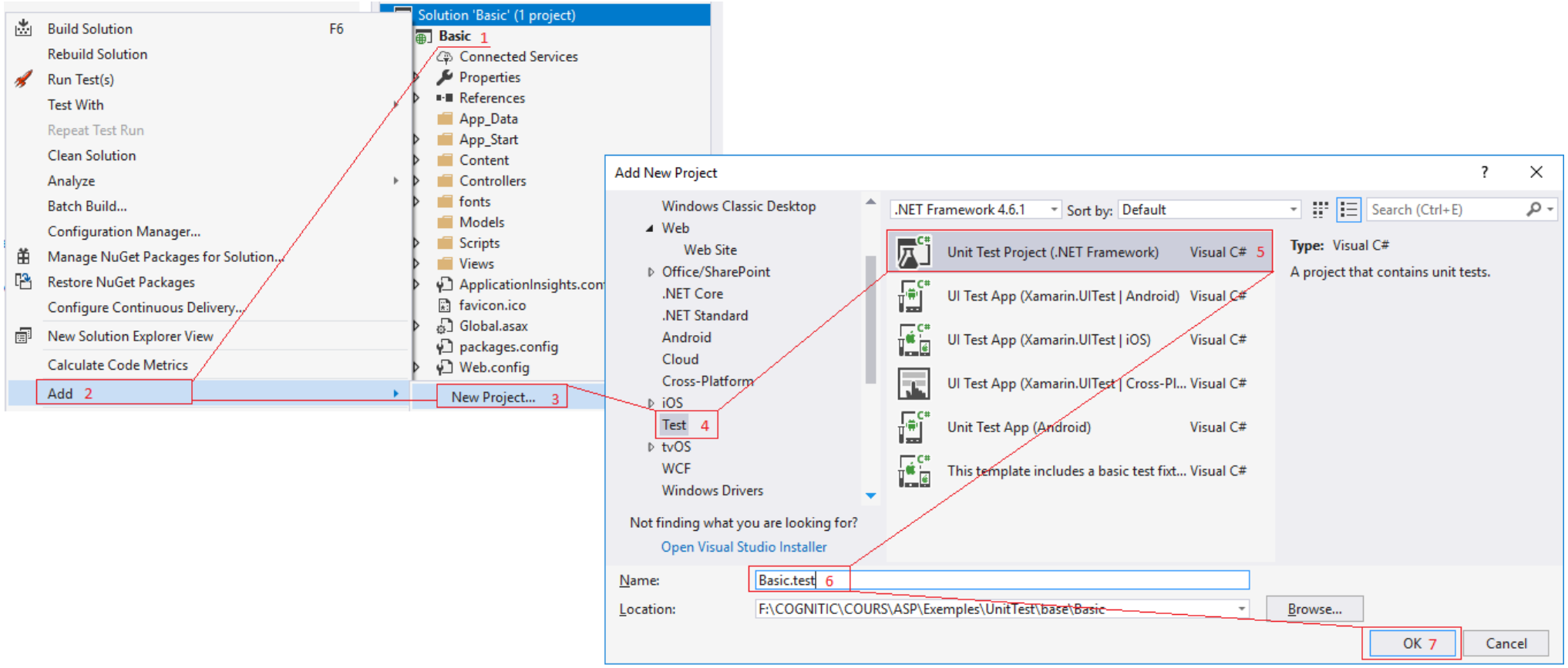
En utilisant les interfaces.

Chaque classe qui implémente une interface doit implémenter les propriétés et les méthodes.

Cela créé donc un faible couplage puisqu'on peut utiliser l'interface dans notre classe plutôt que la classe concrète.

Composants MVC et Unit Testing

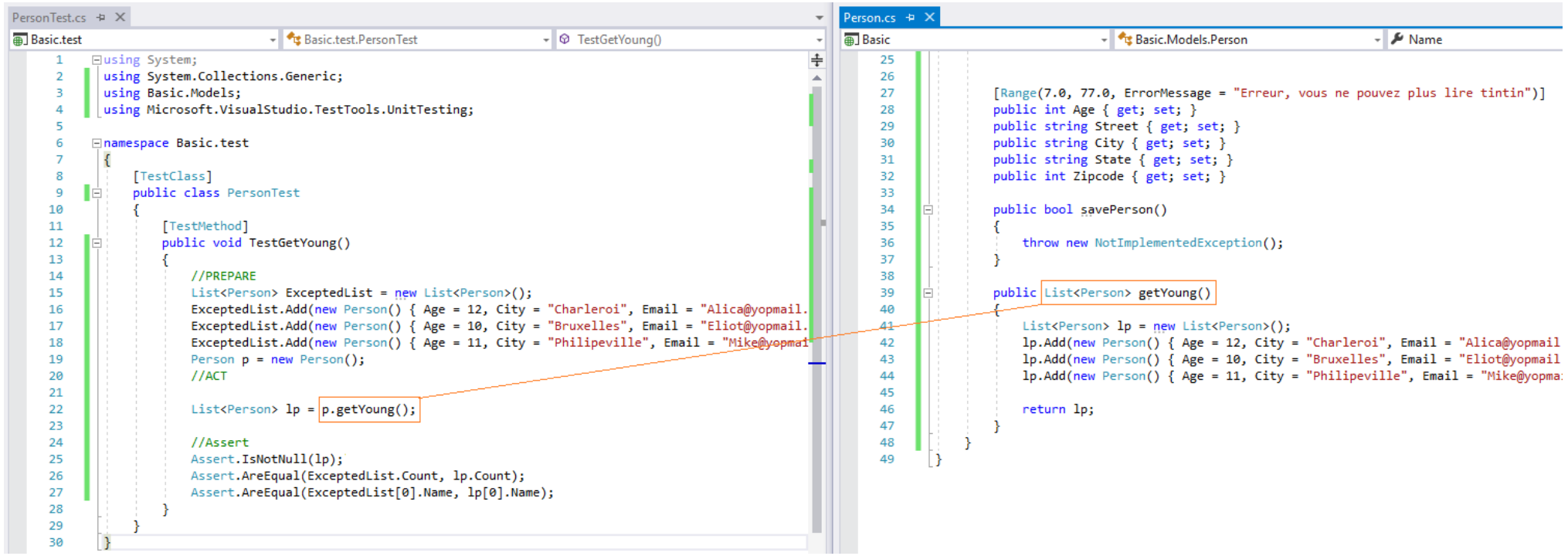
Pour tester une application MVC, nous devons ajouter un nouveau projet à notre solution.



Composants MVC et Unit Testing

Pour commencer, le plus simple est de se focaliser sur les classes du Modèle puisqu'en MVC, ces classes ne dépendent pas des autres composants.

On peut donc facilement instancier ces classes en mémoire, attribuer les valeurs aux propriétés, appeler les méthodes et vérifier le résultat renvoyé.



Composants MVC et Unit Testing

L'annotation *[TestMethod]* renseigne au moteur de test qu'il s'agit d'une méthode permettant le testing. Les *Assert* permettent de poser les vérifications de succès ou d'échec du test.

Les tests unitaires suivent le pattern A/A/A

ARRANGE : Déclaration des variables et objets nécessaires au testing

ACT : Exécution de la méthode à tester

ASSERT : Vérifications

Composants MVC et Unit Testing

Méthode	Description
AreEqual<T>(T, T) AreEqual<T>(T, T, string)	Vérifie que les deux <T> ont la même valeur
AreNotEqual<T>(T, T) AreNotEqual<T>(T, T, string)	Vérifie que les deux <T> n'ont pas la même valeur
AreSame<T>(T, T) AreSame<T>(T, T, string)	Vérifie que les deux variables pointent sur le même objet
AreNotSame<T>(T, T) AreNotSame<T>(T, T, string)	Vérifie que les deux variables ne pointent pas sur le même objet
Fail() Fail(string)	Echec
Inconclusive() Inconclusive(string)	Indique que le résultat du test est incertain
IsTrue(bool) IsTrue(bool, string)	Évalue l'expression booléenne et renvoi le résultat
IsFalse(bool) IsFalse(bool, string)	Évalue l'expression booléenne et renvoi l'opposé du résultat
IsNull(object) IsNull(object, string)	Vérifie si l'objet est null

Composants MVC et Unit Testing

Ces méthodes static permettent de vérifier certains Aspect des tests. Si une *Assertion* échoue, une exception est lancée et fait échouer le test.

Chaque test est exécuté séparément si un échoue, les autres continue à fonctionner.

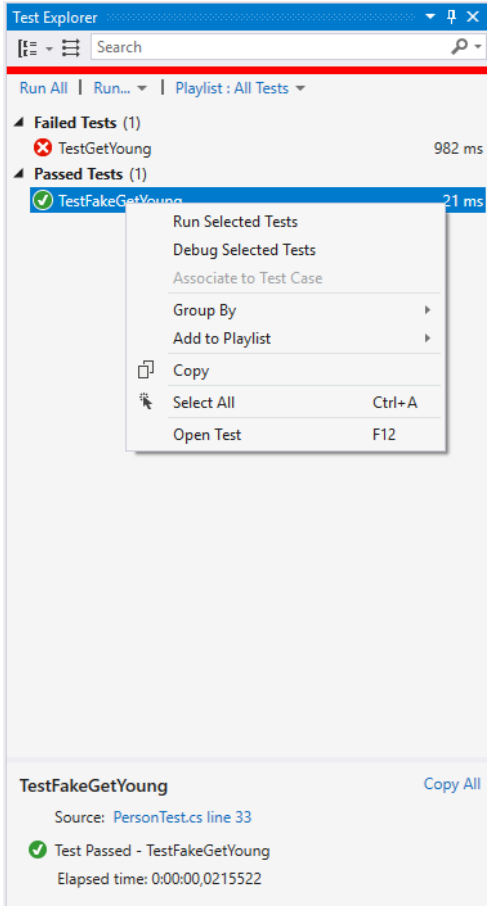
Le string inclus dans les signatures permet d'envoyer un message vers l'exception afin de mieux tracer les erreurs.

Méthode	Description
IsNotNull(object) IsNotNull(object, string)	Vérifie que l'objet n'est pas null
IsInstanceOfType(object, Type) IsInstanceOfType(object, Type, string)	Vérifie si l'objet est une instance de <T>
IsNotInstanceOfType(object, Type) IsNotInstanceOfType(object, Type, string)	Vérifie si l'objet n'est pas une instance de <T>

Composants MVC et Unit Testing

Pour lancer notre test :

- Build de la solution
- Affichage de l'explorateur de test
  - Menu → Test → Window → Test explorer
  - Ou ctrl+E,T



Composants MVC et Unit Testing

Facile?

Dans l'exemple précédent, nous simulons une liste dans la méthode même que l'on doit tester. Ce n'est évidemment pas le cas dans le monde réel.

Un pattern à la rescousse : Le Repository

Un repository est basé sur une interface qui définit les propriétés et les méthodes nécessaires pour une application qui désire enregistrer des données.

Ce pattern nous permet par la suite de définir une source de donnée qui peut être un context EntityFramework, une couche Ado.net ou une FakeDb.

Composants MVC et Unit Testing

Interface

```
1   using System;
2   using System.Collections.Generic;
3   using System.Linq;
4   using System.Text;
5   using System.Threading.Tasks;
6
7   namespace Basic.Models.Interfaces
8   {
9       public interface IPersonContext
10      {
11          IQueryable<Person> GetAll();
12          Person GetById(int id);
13          Person Add(Person Entity);
14          bool Delete(Person Entity);
15          int SaveChanges();
16      }
17  }
```

Implémentation

```
1   using Basic.Models.Global;
2   using System;
3   using System.Collections.Generic;
4   using System.Linq;
5   using System.Web;
6
7   namespace Basic.Models.Interfaces
8   {
9       public class PersonContext : IPersonContext
10      {
11          CampingEntities camping = new CampingEntities();
12          public Person Add(Person Entity)
13          {
14              return MapToPersonLocal(camping.Person.Add(MapToPerson(Entity)));
15          }
16
17          public bool Delete(Person Entity)...
29
30          public IQueryable<Person> GetAll()...
34
35          public Person GetById(int id)...
39
40          public int SaveChanges()...
53
54
55          private Person MapToPersonLocal(Global.Person person)...
69          private Global.Person MapToPerson(Person entity)...
100     }
101 }
```

Composants MVC et Unit Testing

Afin de pouvoir effectuer les tests unitaires, nous ne pouvons pas utiliser l'implémentation précédente car elle se base directement sur notre couche Entity et donc notre test ferait partie des *Test d'intégration*.

Nous devons créer un FakeRepository afin de s'abstraire de la couche Db (EntityFramework, Ado.net,…)

Nous simulerons une collection (source de données) pour effectuer nos tests.

```
6   namespace Basic.Models.Interfaces
7   {
8       public class FakePersonContext : IPersonContext
9       {
10          //Permet de simuler une db
11          SetMap _map = new SetMap();
12
13          public IQueryable<Person> Persons
14          {
15              set { _map.Use<Person>(value); }
16          }
17
18          public bool ChangesSaved { get; set; }
19          public Person Add(Person Entity)
20          {
21              _map.Get<Person>().Add(Entity);
22              return Entity;
23          }
24          public IQueryable<Person> GetAll()
25          {
26              return _map.Get<Person>().AsQueryable();
27          }
28
29          public Person GetById(int id)
30          {
31              Person item = (from p in this.GetAll()
32                             where p.Id == id
33                             select p).First();
34
35              return item;
36          }
37          public int SaveChanges()...
42
43          public bool Delete(Person Entity)...
55
56      }
57  }
```

Composants MVC et Unit Testing

Voici donc notre méthode de test

Il ne diffère pas beaucoup de notre test précédent MAIS celui-ci à l'avantage de permettre de tester le fonctionnement réel du repository sans devoir modifier le code de celui-ci afin d'ajouter des fakes datas

```
[TestMethod]
public void TestFakeGetYoung()
{
    //PREPARE
    var context = new FakePersonContext();
    context.Persons = new[]
    {
    new Person() { Age = 12, City = "Charleroi", Email = "Alica@yopmail.com", FirstName = "Alicia", Name = "Stein", Zipcode =
    new Person() { Age = 10, City = "Bruxelles", Email = "Eliot@yopmail.com", FirstName = "Eliot", Name = "Ness", Zipcode = 1
    new Person() { Age = 11, City = "Philipeville", Email = "Mike@yopmail.com", FirstName = "Mike", Name = "Kelkel", Zipcode
    }.AsQueryable();

    Person p = new Person();
    //ACT

    List<Person> lp = p.getYoung();

    //Assert
    Assert.IsNotNull(lp);
    Assert.AreEqual(3, lp.Count);
    Assert.AreEqual("Stein", lp[0].Name);
}
```

Composants MVC et Unit Testing

Nous devons aller plus loin car nous devons nous assurer que l'appel de notre code se fait dans le bon « context »

Unit test : test sans prendre en compte la couche data.

Autres cas : le code doit interagir avec la couche Data (Entity Framework, Ado.net,…)

Le premier choix qui s'offre à nous est d'utiliser un constructeur afin de définir le repository lors de l'accès à notre classe et à ses méthodes

```
private IPersonContext _context;

public Person()
{
    _context = new PersonContext();
}
public Person(IPersonContext ip)
{
    _context = ip;
}

public List<Person> getYoung()
{
    //Avant le context
    //List<Person> lp = new List<Person>();
    //lp.Add(new Person() { Age = 12, City = "Charleroi", Ema
    //lp.Add(new Person() { Age = 10, City = "Bruxelles", Ema
    //lp.Add(new Person() { Age = 11, City = "Philipeville",

    //return lp;

    //Après l'ajout des constructeurs
    return _context.GetAll().ToList();
}
```

Composants MVC et Unit Testing

Echec du test d'intégration : Erreur couche DB

```
[TestMethod]
public void TestGetYoung()
{
    //PREPARE
    List<Person> ExceptedList = new List<Person>();
    ExceptedList.Add(new Person() { Age = 12, City = "Charleroi", Email = "Alica@yopmail.com",
    ExceptedList.Add(new Person() { Age = 10, City = "Bruxelles", Email = "Eliot@yopmail.com",
    ExceptedList.Add(new Person() { Age = 11, City = "Philipeville", Email = "Mike@yopmail.com"
    Person p = new Person();
    //ACT

    List<Person> lp = p.getYoung();

    //Assert
    Assert.IsNotNull(lp);
    Assert.AreEqual(ExceptedList.Count, lp.Count);
    Assert.AreEqual(ExceptedList[0].Name, lp[0].Name);
}
```

Succès du test unitaire : Méthode de la classe valide

```
[TestMethod]
public void TestFakeGetYoung()
{
    //PREPARE
    var context = new FakePersonContext();
    context.Persons = new[]
    {
    new Person() { Age = 12, City = "Charleroi", Email = "Alica@yopmail.com", FirstName = "Alicia",
    new Person() { Age = 10, City = "Bruxelles", Email = "Eliot@yopmail.com", FirstName = "Eliot",
    new Person() { Age = 11, City = "Philipeville", Email = "Mike@yopmail.com", FirstName = "Mike",
    }.AsQueryable();

    Person p = new Person(context);
    //ACT

    List<Person> lp = p.getYoung();

    //Assert
    Assert.IsNotNull(lp);
    Assert.AreEqual(3, lp.Count);
    Assert.AreEqual("Stein", lp[0].Name);
}
```