Informe de investigación sobre Unit Test aplicado a Desarrollo con capas utilizando EF como Capa de datos

# Introducción

En este documento, se mostrara una de las formas posibles para realizar código pensando en el test unitario. El test unitario es importante en el desarrollo de un sistema para verificar rápidamente que los cambios introducidos, no generaron bugs en las implementaciones de los métodos.

Para realizar tests unitarios, se debe sacrificar tiempo de desarrollo a dicha tarea, para reducir las horas de desarrollo de pruebas unitarias es necesario desarrollar siempre con el concepto de código testeable.

Código testeable es aquel que es fácil de observar, dados ciertas entradas es fácil determinar la salida, por ejemplo el siguiente método

public int Add(int x, int y)

{

    return x + y;

}

La otra cualidad que debe tener el código testeable, es que debe estar aislado, es decir cada método debe centrarse en una acción específica y dejar el resto a otros métodos.

# Investigación

La investigación fue realizada en internet, se busco en diversos foros y sitios especializados en la tecnología, encontrando la información mas completa en el paper ¨Testability and Entity Framework 4.0¨ de Scott Allen. Si bien el paper habla sobre la versión 4.0 de EF, sirvió de base para entender bien el contexto de las pruebas unitarias, luego se amplio esa información con la obtenida en el articulo “Using Microsoft Fakes to Unit Test Entity Framework” y se buscaron ejemplos para consolidar lo investigado.

# Resultados

Es necesario contar con una arquitectura que implemente el patrón repositorio, para definir los métodos a utilizar por los distintos repositorios y reutilizar el código, permitiendo en las clases de testing trabajar sobre estos métodos definidos en la interfaz. Se puede ver un ejemplo de este patrón en el proyecto “Empleados”

A la hora de realizar las pruebas unitarias sobre la capa de datos, se debe crear una estructura falsa de datos para aislar el código de las distintas acciones que pueden fallar, como por ejemplo el acceso a la base de datos. Para crear esta estructura, es necesario contar con una Interfaz IRepositorio la cual define los los métodos para acceder a los datos, para una mejor comprensión, hago un ejemplo:

public interface IRepositorio<TEntity> where TEntity : class

{

TEntity FindById(int id);

IList<TEntity> FindAll();

IList<TEntity> FindWhere(Expression<Func<TEntity, bool>> predicate);

TEntity Create(TEntity t);

int Delete(Expression<Func<TEntity, bool>> predicate);

int Update(TEntity t);

}

Esta interfaz debe ser implementada por los repositorios, y por la clase que creara la estructura de datos falsa para usar en las pruebas unitarias, (también es posible crear una interfaz especifica que implemente esta interfaz por cada repositorio que contenga un tipo de entidades).

En la capa de reglas, que serán las encargadas de instanciar los repositorios, se debe definir un constructor con parámetros del tipo IRepositorio, los cuales se usaran en el momento de hacer el test para usar la estructura de datos falsa (el constructor por defecto instanciara los repositorios persistentes en la base de datos).

Para crear la estructura de datos falsa, se debe crear una clase de repositorio que implemente IRepositorio y traiga datos por defecto, por ejemplo:

public class FakeProductsRepository : IProductsRepository

{

public List<Product> FindAll()

{

return new List<Product>

{

new Product { Name = "PASTE" }

new Product { Name = "BRUSH" }

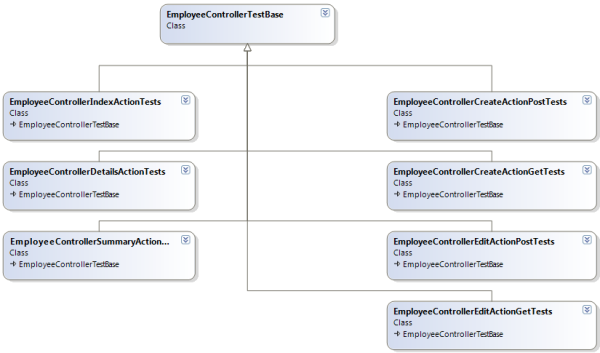
},

}

}

De esta forma se salta la conexión con la base de datos y se prueba únicamente la lógica del método.

Para probar los métodos, se debe crear un proyecto de unit test, y dentro de este las clases necesarias para las estructuras de datos falsas, y las pruebas necesarias. Es recomendable crear una clase base por cada tipo de datos que se va a testear, la cual va a instanciar a la clase correpondiente de la capa de negocio, enviando por parámetro el repositorio falso previamente creado; y luego tener clases que hereden de esta para cada conjunto de operaciones relacionadas al objeto en si (se muestra un diag. De clases a modo de ejemplo).



Si se realiza todo lo aquí comentado, las clases de los tests individuales serán simples, ya que se dedicaran a probar una pequeña porción de código aislado, por lo que no deberán fallar por cuestiones no relacionadas a la lógica del método.

Agrego ejemplos de clases de tests:

[TestClass]

public class EmployeeControllerCreateActionPostTests

           : EmployeeControllerTestBase {

    [TestMethod]

    public void ShouldAddNewEmployeeToRepository() {

        \_controller.Create(\_newEmployee);

        Assert.IsTrue(\_repository.Contains(\_newEmployee));

    }

    [TestMethod]

    public void ShouldCommitUnitOfWork() {

        \_controller.Create(\_newEmployee);

        Assert.IsTrue(\_unitOfWork.Committed);

    }

    // ... more tests

    Employee \_newEmployee = new Employee() {

        Name = "NEW EMPLOYEE",

        HireDate = new System.DateTime(2010, 1, 1)

    };

}

[TestClass]

public class EmployeeControllerIndexActionTests

           : EmployeeControllerTestBase {

    [TestMethod]

    public void ShouldBuildModelWithAllEmployees() {

        var result = \_controller.Index();

        var model = result.ViewData.Model

                      as IEnumerable<Employee>;

        Assert.IsTrue(model.Count() == \_employeeData.Count);

    }

    [TestMethod]

    public void ShouldOrderModelByHiredateAscending() {

        var result = \_controller.Index();

        var model = result.ViewData.Model

                     as IEnumerable<Employee>;

        Assert.IsTrue(model.SequenceEqual(

                       \_employeeData.OrderBy(e => e.HireDate)));

    }

    // ...

}

# Bibliografia

http://msdn.microsoft.com/en-us/library/ff714955.aspx

http://vaideeswaranr.blogspot.com.ar/2013/02/using-microsoft-fakes-to-unit-test.html

http://stackoverflow.com/questions/19287055/unit-testing-and-entity-framework