Post Dobles

Un doble es un objeto que se hace pasar por el original ☺. Este objeto puede ser “Indutrial” o “Artesanal”.

Los dobles “Industriales” son las herramientas que nos proporcionan aquellos frameworks o librerías para crear, gestionar y administrar los dobles que necesitamos. Los más conocidos Xunit, EasyMock, RhinoMock y otros mucho. Cada lenguaje tiene uno o varios frameworks con los que nos podemos sentir cómodos y son sencillos de usar.

Los dobles “Artasenales” son aquellos objetos que nos creamos nosotros mismos (hechos a mano) para que se hagan pasar por otro objeto e imiten su funcionalidad sin implementarla.

Según la funcionalidad que imiten los dobles pueden ser:

Stub: Son aquellos objetos que no comprueban la funcionalidad del objeto que estamos probando únicamente devuelve el resutlado. Otra definición que leí en un foro es que se utilizan para los test que comprueban los cambios de estado. Este tipo de objeto **NO** se usa para validar los Assert.

Mocks: Son aquellos objetos que comprueban la funcionalidad del objeto que estamos probando y nos devuelve el resultado según su lógica. También se utilizan para probar el funcionamiento de un objeto. Este tipo de objeto **SI** se usa para validar los Asserts.

Fakes: Son aquellos objetos que nos creamos nosotros (pueden ser Mocks o stubs) para que se hagan pasar por un objeto que necesitamos.

Cargar un fichero en una tabla en bbdd con los campos

ID no null entero

Titulo no null varchar(50)

Director no null varchar(50)

Cantidad no null entero

Precio no null float(2,2)

El fichero tiene los campos separados por |

Enviar un mail con las películas de un director pasado como parámetro.

Ciclo 1: Comprobar que existe el fichero de entrada

Cuando creamos el test nos damos cuenta que necesitamos una clase fichero con el propósito de gestionar nuestro fichero de entrada, en este caso que nos diga si existe. Como el fichero de entrada es un objeto externo necesitaremos crear un mock o un stub. Podemos utilizar una librería o framework o podemos crearnos nuestro propio doble. Como vamos a comprobar si existe o no usamos un stub que nos devolverá cuando lo llamemos true.

Si usamos un doble artifical en este caso con la librería RhinoMock.

        [TestMethod]

        public void Existe()

        {

            Rhino.Mocks.MockRepository mock = new Rhino.Mocks.MockRepository();

            FicheroEntrada ficheroEntrada = mock.Stub<FicheroEntrada>();

            Assert.IsTrue(ficheroEntrada.Existe("ficheroEntrada"));

        }

Clase FicheroEntrada

 public bool Existe(string nombreFichero)

        {

            return File.Exists(nombreFichero);

        }

Como el ficheo de entrada no existe el test se queda en rojo, como el objeto fichero de entrada es externo y lo hemos sustituido por un stub, vamos a hacer que el test se ponga verde añadiendo al stub el resultado que queremos cuando se llame al método Existe.

 [TestMethod]

        public void Existe()

        {

            Rhino.Mocks.MockRepository mock = new Rhino.Mocks.MockRepository();

            IFicheroEntrada ficheroEntrada = mock.Stub<IFicheroEntrada>();

            Rhino.Mocks.Expect.Call(ficheroEntrada.Existe("")).IgnoreArguments().Return(true);

            mock.Replay(ficheroEntrada);

            Assert.IsTrue(ficheroEntrada.Existe("ficheroEntrada"));

            mock.Verify(ficheroEntrada);

        }

Modificaciones al insertar el resultado del stub.

RhinoMockTrabaja con Interface por lo que hemos tenido que extraer una interface de la clase FicheroEntrada para trabajar sobre ella.

Definimos verificamos las expectativas que esperamos cuando se llame a un método del stub

Con esto le estamos diciendo que cuando nuestro test llame al método Existe() de la clase FicheroEntrada este debe devolver True independientemente de lo que le pasemos.

Despues de refactorizar podemos añadir el nombre del fichero a la Interface y nos queda el test como

        [TestMethod]

        public void Existe()

        {

            Rhino.Mocks.MockRepository mock = new Rhino.Mocks.MockRepository();

            IFicheroEntrada ficheroEntrada = mock.Stub<IFicheroEntrada>();

            Rhino.Mocks.Expect.Call(ficheroEntrada.Existe("")).IgnoreArguments().Return(true);

            mock.Replay(ficheroEntrada);

            Assert.IsTrue(ficheroEntrada.Existe(ficheroEntrada.nombreFicheroEntrada));

            mock.Verify(ficheroEntrada);

        }

Fijaros que al ponerle IgnoreParameter le podemos pasar cualquier cosa le da lo mismo y devolverá siempre true.

Ciclo 1 (bis): Comprobar que existe el fichero de entrada

Ahora vamos a crearnos nosotros nuestro propio stub, es decir, una clase que hereda de la clase FicheroEntrada y que sobrescribe los métodos de la clase padre que queremos emular.

 public class FicheroEntradaStub : FicheroEntrada

    {

        public override bool Existe(string nombreFichero)

        {

            return true;

        }

    }

Y nos queda el test

[TestMethod]

public void Existe\_Con\_Stub\_Artesanal()

        {

            FicheroEntradaStub fakeFicheroEntrada = new FicheroEntradaStub();

            IFicheroEntrada ficheroEntrada = fakeFicheroEntrada;

            Assert.IsTrue(ficheroEntrada.Existe(ficheroEntrada.nombreFicheroEntrada));

        }

Para poder sobrescribir el método Existe de la clase FicheroEntrada hemos tenido que hacer virtual al método Existe();

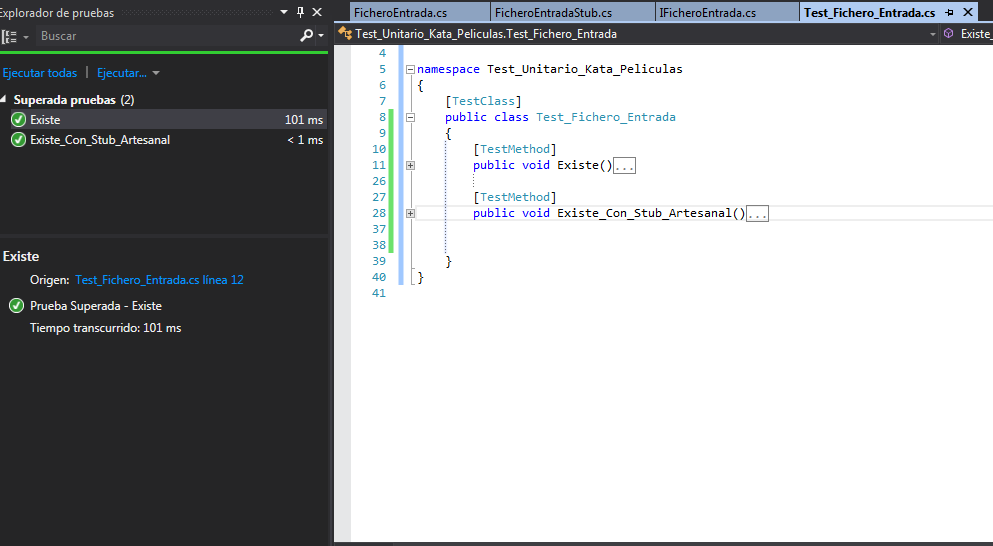
  public virtual bool Existe(string nombreFichero)

        {

            return File.Exists(nombreFichero);

        }

Como vimos en el post anterior podemos saber que hace la clase sin necesidad de comentar solo con leer los nombres Kata\_Peliculas.Test\_Fichero\_Entrada.Existe()



Ciclo 2: Abrir fichero de entrada

Al igual que el caso anterior el test se pone rojo porque no tenemos el fichero de entrada

 [TestMethod]

        public void Abrir()

        {

            IFicheroEntrada ficheroEntrada = new FicheroEntrada();

            StreamReader swEntrada = ficheroEntrada.Abrir();

            Assert.AreNotEqual(null, swEntrada);

        }

 public System.IO.StreamReader Abrir()

        {

            return new StreamReader(new FileStream(nombreFicheroEntrada, FileMode.Open, FileAccess.Read));

        }

por lo que usamos un stub y el test nos queda

[TestMethod]

        public void Abrir()

        {

            Rhino.Mocks.MockRepository mock = new Rhino.Mocks.MockRepository();

            IFicheroEntrada ficheroEntrada = mock.DynamicMock<IFicheroEntrada>();

            Rhino.Mocks.Expect.Call(ficheroEntrada.Abrir()).Return(mock.Stub<StreamReader>());

            mock.Replay(ficheroEntrada);

            StreamReader contenidoFichero = ficheroEntrada.Abrir();

            Assert.IsNotNull(contenidoFichero);

            mock.Verify(ficheroEntrada);

        }

El test está en verde y refactorizamos.

Ciclo 3: Recorrer fichero de datos

Al realizar el primer test es un babyStep donde el método nos devuelve true

        [TestMethod]

        public void Recorrer\_Fichero()

        {

            FicheroEntrada ficheroEntrada = new FicheroEntrada();

            bool result = ficheroEntrada.RecorrerFichero(new Streamreader());

            Assert.IsTrue(result);

        }

 public bool RecorrerFichero(StreamReader contenidoFichero)

        {

Return True

}

Ciclo 4: Error fichero de datos vacio

Cuando se produce un error al leer el fichero provocara un error. En principio intentamos utilizar un stub para recorrer el fichero pero no conseguíamos que el test se pusiera en rojo por lo que obtamos por hacer un objeto fake del contenido del fichero, es decir, un fake de streamreader

 [TestMethod]

        public void Error\_Fichero\_Vacio()

        {

            try

            {

                FakeStreamContenidoFichero fakeStream = new FakeStreamContenidoFichero(true);

                FicheroEntrada ficheroEntrada = new FicheroEntrada();

                bool result = ficheroEntrada.RecorrerFichero(fakeStream.contenidoFichero);

                Assert.Fail("Error fichero vacio");

            }

            catch

            {

                //Ok

            }

        }

Este es el fake

    public class FakeStreamContenidoFichero

    {

        public FakeStreamContenidoFichero(bool erroneo)

        {

            MemoryStream ms = new MemoryStream();

            StreamWriter sw = new StreamWriter(ms);

            if (!erroneo)

            {

                sw.WriteLine("1|REGRESO AL FUTURO|ROBERT ZEMECKIS|3|9.99");

                sw.WriteLine("2|STAR WARS|GEORGE LUCAS|6|5.49");

                sw.WriteLine("3|TITANIC|JAMES CAMERON|1|10");

                sw.WriteLine("4|INDIANA JONES|STEVEN SPIELBERG|4|4.95");

                sw.WriteLine("5|PARQUE JURASICO|STEVEN SPIELBERG|3|2.5");

                sw.WriteLine("6|TERMINATOR|JAMES CAMERON|5|6.05");

                sw.Flush();

                ms.Seek(0, SeekOrigin.Begin);

                contenidoFichero = new StreamReader(ms);

            }

            else

            {

                contenidoFichero = new StreamReader(new MemoryStream());

            }

        }

        public StreamReader contenidoFichero { get; private set; }

    }

Y asi queda el método

        public bool RecorrerFichero(StreamReader contenidoFichero)

        {

            while (!contenidoFichero.EndOfStream)

            {

                try

                {

                    contenidoFichero.ReadLine();

                }

                catch

                {

                    throw new Exception("Error al ller registro");

                }

            }

            return true;

        }

Ciclo 5: Pruebas de los campos de las peliculas

Creamos una nueva clase de test cuyo objetivo es contener los test refernte a los registros de las peliculas.

 [TestMethod]

        public void Validar\_ID\_Registro\_leido\_De\_Fichero()

        {

            Pelicula pelicula = new Pelicula("1|REGRESO AL FUTURO|ROBERT ZEMECKIS|3|9.99");

            Assert.AreEqual(1, pelicula.ID);

        }

        [TestMethod]

        public void Validar\_Titulo\_Registro\_leido\_De\_Fichero()

        {

            Pelicula pelicula = new Pelicula("1|REGRESO AL FUTURO|ROBERT ZEMECKIS|3|9.99");

            Assert.AreEqual("REGRESO AL FUTURO", pelicula.Titulo);

        }

        [TestMethod]

        public void Validar\_Director\_Registro\_leido\_De\_Fichero()

        {

            Pelicula pelicula = new Pelicula("1|REGRESO AL FUTURO|ROBERT ZEMECKIS|3|9.99");

            Assert.AreEqual("ROBERT ZEMECKIS", pelicula.Director);

        }

        [TestMethod]

        public void Validar\_Cantidad\_Registro\_leido\_De\_Fichero()

        {

            Pelicula pelicula = new Pelicula("1|REGRESO AL FUTURO|ROBERT ZEMECKIS|3|9.99");

            Assert.AreEqual(3, pelicula.Cantidad);

        }

        [TestMethod]

        public void Validar\_Precio\_Registro\_leido\_De\_Fichero()

        {

            Pelicula pelicula = new Pelicula("1|REGRESO AL FUTURO|ROBERT ZEMECKIS|3|9,99");

            Assert.AreEqual("9,99", pelicula.Precio.ToString());

        }

La clase Peliculas nos queda

  public class Pelicula

    {

        public Pelicula(string texto)

        {

            string[] datos = texto.Split('|');

            ValidarIDDatos(datos[0]);

            ValidarTituloDatos(datos[1]);

            ValidarDirectorDatos(datos[2]);

            ValidarCantidadDatos(datos[3]);

            ValidarPrecioDatos(datos[4]);

        }

        private void ValidarPrecioDatos(string precio)

        {

            try

            {

                Precio = float.Parse(precio);

            }

            catch

            {

                throw new Exception("Precio no valido");

            }

        }

        private void ValidarCantidadDatos(string cantidad)

        {

            try

            {

                Cantidad = int.Parse(cantidad);

            }

            catch

            {

                throw new Exception("Cantidad no numerico");

            }

        }

        private void ValidarDirectorDatos(string director)

        {

            if ((0 < director.Trim().Length) || (director.Trim().Length < 51))

            {

                Director = director;

            }

            else

            {

                throw new Exception("Director no valido");

            }

        }

        private void ValidarTituloDatos(string titulo)

        {

            if ((0 < titulo.Trim().Length) || (titulo.Trim().Length < 51))

            {

                Titulo = titulo;

            }

            else

            {

                throw new Exception("Titulo no valido");

            }

        }

        private void ValidarIDDatos(string iD)

        {

            try

            {

                ID = int.Parse(iD);

            }

            catch

            {

                throw new Exception("ID no numerico");

            }

        }

        public int ID { get; private set; }

        public string Titulo { get; set; }

        public string Director { get; set; }

        public int Cantidad { get; set; }

        public float Precio { get; set; }

    }

Ahora para refactorizar vamos a crear un objeto que nos sirva de enlace entre la bbdd y el código. En el que añadiremos los campos que ahora son propiedades. (Code First)

Añadimos un contexto de bbdd para enlazar el código con la bbdd. Recomindo leer este post donde explica claramente lo que hemos hecho. <http://jhonnyslopez.wordpress.com/2012/04/15/introduccion-a-entity-framework-4-3-1-code-first-con-c-visual-studio-11/>

 public class DbContexto : DbContext

    {

        public DbSet<Pelicula> Peliculas{ get; set; }

    }

Ciclo 6: Test sobre bbdd

Si realizamos los test sobre la bbdd directamente, podemos encontrarnos con que la bbdd no existe, van demasiado lentos… por lo que usaremos el dbContext que hemos generado para generar un mock que emule a la Clase DbContexto.

Defino el test

 [TestMethod]

        public void Insertar\_En\_Base\_De\_Datos()

        {

            try

            {

                DbContexto context = new DbContexto();

                KataPeliculas.Pelicula pelicula = new KataPeliculas.Pelicula("1|REGRESO AL FUTURO|ROBERT ZEMECKIS|3|9.99");

                context.Insertar(pelicula);

                Assert.Fail("Error al insertar pelicula");

            }

            catch

            {

                //Ok

            }

        }

Escribo código

 public class DbContexto : DbContext

    {

        public DbSet<Pelicula> Peliculas{ get; set; }

        public void Insertar(Pelicula pelicula)

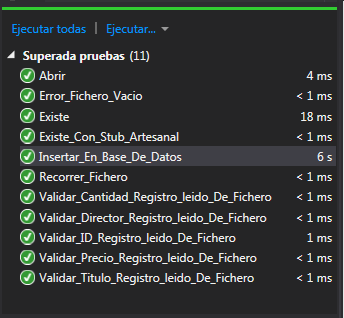
        {

            Peliculas.Add(pelicula);

        }

    }

El test esta en verde pero demasiado tiempo



Cambio en el test por un mock

 [TestMethod]

        public void Insertar\_En\_Base\_De\_Datos()

        {

            try

            {

                Rhino.Mocks.MockRepository mock = new Rhino.Mocks.MockRepository();

                KataPeliculas.IDbContexto context = mock.DynamicMock<IDbContexto>();

                Rhino.Mocks.Expect.Call<IDbContexto>(context).IgnoreArguments().Throw(new Exception("Error al insertar pelicula"));

                KataPeliculas.Pelicula pelicula = new KataPeliculas.Pelicula("1|REGRESO AL FUTURO|ROBERT ZEMECKIS|3|9.99");

                context.Insertar(pelicula);

            }

            catch

            {

                //Ok

            }

        }

Test para hacer commit en bbdd

        [TestMethod]

        public void Commit\_En\_Base\_De\_Datos()

        {

            try

            {

                DbContexto context = new DbContexto();

                context.Commit();

                Assert.Fail("Error al hacer commit");

            }

            catch

            {

                //OK

            }

        }

Código

        public void Commit()

        {

            this.SaveChanges();

        }

Como tarda demasiado usamos un mock y nos queda así refactorizado la clase de test

 [TestClass]

    public class Test\_Sobre\_BBDD\_Datos

    {

        private Rhino.Mocks.MockRepository mock;

        private KataPeliculas.IDbContexto context;

        public Test\_Sobre\_BBDD\_Datos()

        {

            mock = new Rhino.Mocks.MockRepository();

            context = mock.DynamicMock<IDbContexto>();

        }

        [TestMethod]

        public void Insertar\_En\_Base\_De\_Datos()

        {

            try

            {

                Rhino.Mocks.Expect.Call<IDbContexto>(context).IgnoreArguments().Throw(new Exception("Error al insertar pelicula"));

                KataPeliculas.Pelicula pelicula = new KataPeliculas.Pelicula("1|REGRESO AL FUTURO|ROBERT ZEMECKIS|3|9.99");

                context.Insertar(pelicula);

            }

            catch

            {

                //Ok

            }

        }

        [TestMethod]

        public void Commit\_En\_Base\_De\_Datos()

        {

            try

            {

                Rhino.Mocks.MockRepository mock = new Rhino.Mocks.MockRepository();

                KataPeliculas.IDbContexto context = mock.DynamicMock<IDbContexto>();

                Rhino.Mocks.Expect.Call<IDbContexto>(context).IgnoreArguments().Throw(new Exception("Error al hacer commit"));

                context.Commit();

            }

            catch

            {

                //OK

            }

        }

    }

Realizar consulta a bbdd

        [TestMethod]

        public void Realizar\_Consulta\_De\_Cantidad\_De\_Peliculas\_De\_Un\_Director()

        {

            DbContexto contexto = new DbContexto();

            int cantidad = contexto.ObtenerCantidadPeliculasPorDirector("ROBERT ZEMECKIS");

            Assert.AreEqual(1, cantidad);

        }

Escribimos el código

 public int ObtenerCantidadPeliculasPorDirector(string director)

        {

            return Peliculas.Select(s => s.Director == director).Count();

        }

Da error por no existirla bbdd ponemos un mock

 [TestMethod]

        public void Realizar\_Consulta\_De\_Cantidad\_De\_Peliculas\_De\_Un\_Director()

        {

                Rhino.Mocks.MockRepository mock = new Rhino.Mocks.MockRepository();

                KataPeliculas.IDbContexto context = mock.DynamicMock<IDbContexto>();

                Rhino.Mocks.Expect.Call(context.ObtenerCantidadPeliculasPorDirector("ROBERT ZEMECKIS")).IgnoreArguments().Return(1);

                mock.ReplayAll();

                int cantidad = context.ObtenerCantidadPeliculasPorDirector("ROBERT ZEMECKIS");

                Assert.AreEqual(1, cantidad);

        }

Asi quedaría la interface de la bbdd que utilizaremos después en las pruebas de integración.

public interface IDbContexto

    {

        void Insertar(Pelicula pelicula);

        void Commit();

        int ObtenerCantidadPeliculasPorDirector(string directo);

        System.Data.Entity.DbSet<Pelicula> Peliculas { get; set; }

    }

}

Hay que tener en cuenta que al hacer code first el objeto DbContext utiliza la cadena de conexión que tenemos configurada por en el framework

Ciclo 7: Envió de mail

[TestClass]

    public class Test\_Envio\_Mail

    {

        string servidor = "";

        string from="";

        string to="";

        string mensaje="";

        string asunto="";

        [TestMethod]

        public void Enviar()

        {

            KataPeliculas.Email mail = new KataPeliculas.Email(servidor, from, to);

            bool result = mail.Enviar(mensaje, asunto);

            Assert.IsTrue(result);

        }

    }

Codigo refactorizado

 public class Email

    {

        private string servidor;

        private string from;

        private string to;

        public Email(string servidor, string from, string to)

        {

            this.servidor = servidor;

            this.from = from;

            this.to = to;

        }

        public bool Enviar(string mensaje, string asunto)

        {

            return true;

        }

    }

Test en caso de error en el envio de correo

 [TestMethod]

        public void Error\_Al\_Enviar()

        {

            KataPeliculas.Email mail = new KataPeliculas.Email(servidor, from, to);

            bool result = mail.Enviar(mensaje, asunto);

            Assert.IsFalse(result, "No se pudo enviar el mail");

        }

Esta en rojo

   public bool Enviar(string mensaje, string asunto)

        {

            try

            {

                SmtpClient smtp = new SmtpClient(servidor);

                smtp.Send(from, to, asunto, mensaje);

                return true;

            }

            catch

            {

                return false;

            }

        }

Este test rompe el test anterior por lo que arreglamos el anterior haciendo que devuelva una excepción en caso de error y void si envía el mail

  public void Enviar(string mensaje, string asunto)

        {

            try

            {

                SmtpClient smtp = new SmtpClient(servidor);

                smtp.Send(from, to, asunto, mensaje);

            }

            catch

            {

                throw new Exception("Error al enviar el mail");

            }

        }

Lanzo los test de nuevo y el test enviar sigue roto pero ahora por no poder enviar el mail, por lo que nos creamos un mock al no existir el servidor de correo.

        [TestMethod]

        public void Enviar()

        {

            try

            {

                Rhino.Mocks.MockRepository mock = new Rhino.Mocks.MockRepository();

                IEmail mail = mock.DynamicMock<IEmail>(servidor, from, to);

                Rhino.Mocks.Expect.Call<IEmail>(mail).IgnoreArguments().Throw(new Exception("Error al Enviar Correo"));

                mail.Enviar(mensaje, asunto);

            }

            catch

            {

                //OK

            }

        }

Ya están en verde.

Ciclo 8: Cierre de fichero

  [TestMethod]

        public void Cerrar()

        {

            Rhino.Mocks.MockRepository mock = new Rhino.Mocks.MockRepository();

            IFicheroEntrada ficheroEntrada = mock.DynamicMock<IFicheroEntrada>();

            Rhino.Mocks.Expect.Call(ficheroEntrada.Cerrar(null)).IgnoreArguments().Return(true);

            mock.ReplayAll();

            bool result = ficheroEntrada.Cerrar(ficheroEntrada.Abrir());

            Assert.IsTrue(result);

            mock.VerifyAll();

        }

Codigo

 public void Cerrar(System.IO.StreamReader contenidoFichero)

        {

            contenidoFichero.Close(); ;

        }

Asi queda la interface para usar en los test de integración

 public interface IFicheroEntrada

    {

        bool Existe();

        System.IO.StreamReader Abrir();

        void Cerrar(StreamReader contenidoFichero);

        bool RecorrerFichero(StreamReader streamReader);

        string nombreFicheroEntrada { get; set; }

    }

Esto es todo, espero que hayáis aprendido tanto como nosotros haciendo esta kata. El código de la solución se encuentra en https://francalles@bitbucket.org/francalles/kata-peliculas.git