Los patrones Inversión de Control (IoC) e Inyección de Dependencias (DI)

Article · January 2011

CITATIONS
0

READS
5,144

2 authors:

Alejandro Tamayo
University of Havana
16 PUBLICATIONS **6** CITATIONS
SEE PROFILE

Miguel Katrib
University of Havana
45 PUBLICATIONS **57** CITATIONS
SEE PROFILE

Los patrones Inversión de Control (IoC) e Inyección de Dependencias (DI). Cómo hacer consultas LINQ más abstractas.

Alejandro Tamayo Castillo, Miguel Katrib Mora

Entradilla

Muchas aplicaciones cuentan, en el momento de su despliegue, con la opción de seleccionar un tipo de base de datos (SQL Server, MySQL, PostgreSQL, entre otras) en la que almacenar la información. En este artículo se muestra cómo desarrollar una Aplicación .NET con soporte transparente (*pluggable*) para el almacenamiento de los datos sin depender directamente de un "mapeador" relacional (ORM) como **Entity Framework** o **NHibernate** al permitir modelar/realizar consultas LINQ de forma abstracta, es decir sin depender de las características particulares del proveedor de datos que se utilice.

"*La mayoría de las nuevas ideas en cuanto a desarrollo de software son realmente nuevas variaciones de viejas ideas*"

Martin Fowler

Aunque los patrones de diseño **Inversión de Control** (IoC) e **Inyección de Dependencias** (DI) llevan más de diez años de creados, todavía mantienen su vigencia y utilidad. Este artículo puede servir de punto de partida a los que lean por primera vez sobre este tema. Pero si usted ya conoce sobre estos patrones es una buena oportunidad para reflexionar sobre su validez y ver cómo han sido aplicados en la práctica.

¿Qué es IoC/DI y por qué lo necesito?

Es muy probable que el lector se haya topado alguna vez con el problema de "conectarse a una base de datos". El clásico código C# del Listado 1~~Listado 1~~ muestra la forma de conectarse a una base de datos de SQL Server utilizando ADO.NET.

```
var sqlConnection = new SqlConnection(@"Data Source=.;Initial Catalog=CineDB;
                                        Integrated Security=True");
var dataAdapter = new SqlDataAdapter(@"SELECT Film.Title, Director.Name
                                        FROM Director INNER JOIN Film
                                        ON Director.Id = Film.IdDirector",
                                        sqlConnection);
var dataTable = new DataTable("Film");
dataAdapter.Fill(dataTable);
foreach (DataRow row in dataTable.Rows)
{
    Console.WriteLine("Film: {0} Director: {1}",
       row["Title"], row["Name"]);
}
```

Listado 1 Conexión a una base de datos de SQL Server utilizando ADO.NET

La primera cuestión que hay que analizar en este código (ignorando el *hardcoding* de las cadenas que están escritas directamente en el código como ejemplificación) se encuentra en la creación de los objetos que se utilizan para la conexión a la base de datos (en este caso el `SqlConnection` y el `SqlDataAdapter`). Fíjese que en este ejemplo el desarrollador tiene la responsabilidad sobre la creación (haciendo uso explícito de la operación **new**) de las instancias de conexión y por consiguiente del tipo de objeto a utilizar (en este ejemplo `SqlConnection`, pero pudiese ser `SQLiteConnection` para una base de datos **SQLite**). Realmente resulta un poco incómodo escribir toda esta parafernalia una y otra vez cuando lo que queremos es solamente realizar una consulta.

Lo anterior podría mejorarse si se "encapsula" la conexión como propiedad global en un cierto ensamblado y usar entonces el patrón **Factoría** (*Factory*) como se muestra simplificadamente en el Listado 2~~Listado 2~~.

```
class ConnectionAdmin
{
    public string Provider { get; set; }
    public string ConnectionString { get; set; }
    private static IDbConnection _connection = null;
    public static IDbConnection Connection
    {
        get
        {
            if (_connection == null)
            {
                return ProviderFactory.CreateConnection(Provider,
                    ConnectionString);
            }
            return _connection;
        }
    }
}

class ProviderFactory
{
    public static IDbConnection CreateConnection(string provider,
            string connectionString)
    {
        switch (provider)
        {
            case "System.Data.SqlClient":
                return new SqlConnection(connectionString);
            case "System.Data.SQLite":
                return new SQLiteConnection(connectionString);
            default:
                throw new NotImplementedException();
        }
    }
}
```

Listado 2 Ejemplo de Factoría para la gestión de la conexión a base de datos de diferentes tipos

Dado que ADO.NET define las interfaces `IDbConnection`, `IDbCommand`, `IDbDataAdapter` para que sean implementadas por los respectivos proveedores de bases de datos que utilizan esta tecnología

entonces es posible implementar tal factoría "cableando" estáticamente en el código del `switch` todas las alternativas deseadas (como se ve en el Listado 2~~Listado 2~~). Sin embargo, sería deseable que la factoría pudiese gestionar dinámicamente a cuál proveedor conectarse (incluyendo tal vez proveedores no previstos en el momento de escribir la aplicación).

¿Qué papel puede jugar aquí esto de **Inversión de Control** (IoC) [1]? Cuando el desarrollador explícitamente crea e inicializa un objeto se dice que tiene el control de su creación. La idea con IoC consiste en cederle el control de la creación a una entidad, generalmente conocida como **Contenedor,** que es quien se encargará de realizar ese trabajo y el código de la aplicación que lo usa solo se limitará a consumir la instancia creada[1].

```
/* Inicialización de la conexión */
var connection = container.Resolve<IDbConnection>(new
{
    connectionString = @"Data Source=.;Initial Catalog=CineDB;
                Integrated Security=True"
});

/* Obtenciones posteriores de la conexión */
var dataAdapter = container.Resolve<IDbDataAdapter>(new
{
    selectCommandText = @"SELECT Film.Title, Director.Name
                FROM Director INNER JOIN Film
                ON Director.Id = Film.IdDirector",
    selectConnection = container.Resolve<IDbConnection>()
});
```

Listado 3 Ejemplo de uso de un contenedor IoC (Castle Windsor)

El uso de tal contenedor puede apreciarse en el Listado 3~~Listado 3~~. Primeramente se "resuelve" la conexión y se inicializa con la cadena de conexión deseada. Esta operación puede realizarse por ejemplo cuando la aplicación inicia. Posteriormente cuando se desea realizar una consulta, simplemente se resuelve un adaptador de datos con el código SQL de la consulta y una conexión que también se resuelve mediante el contenedor, pero esta vez sin pasar ninguna cadena de conexión ya que se supone que esta conexión se inicializó en algún momento previo. Note que en ningún momento se utiliza una implementación concreta, sino que se trata con interfaces para lograr la abstracción deseada.

¿Y por dónde le entra el agua al coco? Realmente por muy mágico que pueda parecer el contenedor, tiene que estar previamente determinada la relación entre interfaz e implementación concreta. En este artículo se utiliza de manera práctica **Castle Windsor** [2] que es uno de los tantos contenedores IoC que existen para .NET. El contenedor en este caso es un objeto de tipo `IWindsorContainer` que es responsabilidad del desarrollador inicializar (más adelante se verá un ejemplo). En el Listado 4~~Listado 4~~ podrá encontrar un ejemplo de cómo se realiza la asociación entre interface e implementación.

```
container.Register(Component.For<IDbConnection>()
    .ImplementedBy<SqlConnection>()
    .LifeStyle.Is(LifestyleType.Singleton));
```

[1] De ahí la expresión de Invertir el Control en la creación del objeto.

```
container.Register(Component.For<IDbDataAdapter>()
    .ImplementedBy<SqlDataAdapter>()
    .LifeStyle.Is(LifestyleType.Transient));
```

Listado 4 Registro de implementaciones en el contenedor Castle Windsor

Note que se ha registrado la implementación para SQL Server de las interfaces `IDbConnection` e `IDbDataAdapter` pero se han definido dos formas diferentes de instanciación: para la conexión se ha solicitado que la instanciación sea **Singleton** para tener una única conexión en la aplicación (la sintaxis `container.Resolve<IDbConnection>()` siempre devolverá el mismo objeto) y para el adaptador de datos pues **Transient** que es el equivalente a realizar un **new** cada vez que le solicitemos un objeto al contenedor. Es por ello que cuando se resuelve la conexión por segunda vez ya se devuelve inicializada. Sin embargo, hasta aquí las consultas se siguen expresando como una cadena en el lenguaje SQL en la cual hay detalles que dependen del proveedor de base de datos y no de los conceptos que propiamente se quieren consultar. Esto obliga al desarrollador a tener en cuenta posibles comportamientos diferentes en dependencia del proveedor de datos, además de los errores que pueden ocurrir en ejecución cuando sean analizadas estas cadenas que conforman la consulta. De ahí la valía de la propuesta de LINQ con el .NET Framework 3.5 ya tratado ampliamente en esta revista. La consulta LINQ se expresa en código C# (y por consiguiente analizada estáticamente por el compilador) y es el proveedor LINQ quien se encarga de generar automáticamente el código SQL equivalente.

Pero LINQ sigue atado al proveedor, bien programado directamente o usando los asistentes de Visual Studio, el desarrollador tiene que indicar un proveedor de datos concreto. Sería muy útil poder declarar una consulta LINQ "abstracta" que no dependa de ningún proveedor de datos en particular para que luego se pueda dinámicamente (en ejecución) asociarla a un proveedor de datos. ¿Se podrá lograr este objetivo con IoC usando un contenedor como el Castle Windsor?

Consultas LINQ abstractas

Para lograr esta suerte de consulta LINQ abstracta, separemos nuestra aplicación .NET en al menos tres ensamblados independientes:

- La aplicación principal
- Un ensamblado que contendrá las interfaces de la aplicación (`MovieApp.Interfaces.dll` por ejemplo)
- Un ensamblado por cada proveedor de base de datos que se quiera soportar (Por ejemplo `MovieApp.SqlConnection.dll` y `MovieApp.SQLiteConnection.dll`).

La aplicación principal referenciará estáticamente el ensamblado que contiene las interfaces, como mismo lo harán los ensamblados que contienen el código concreto para la conexión a los distintos tipos de bases de datos. Pero en ningún momento se referenciarán estáticamente la aplicación principal y los ensamblados por cada proveedor de base de datos. El objetivo es que estos se vinculen dinámicamente en tiempo de ejecución para así poder decidir, sin recompilar la aplicación principal, a qué tipo de base de datos se desea conectar. Este ensamblado de interfaces representará el papel de intermediario.

Para ilustrar las consultas utilizaremos la base de datos de cine que hemos presentado en otros trabajos de DotNetManía (Figura 1~~Figura 1~~)

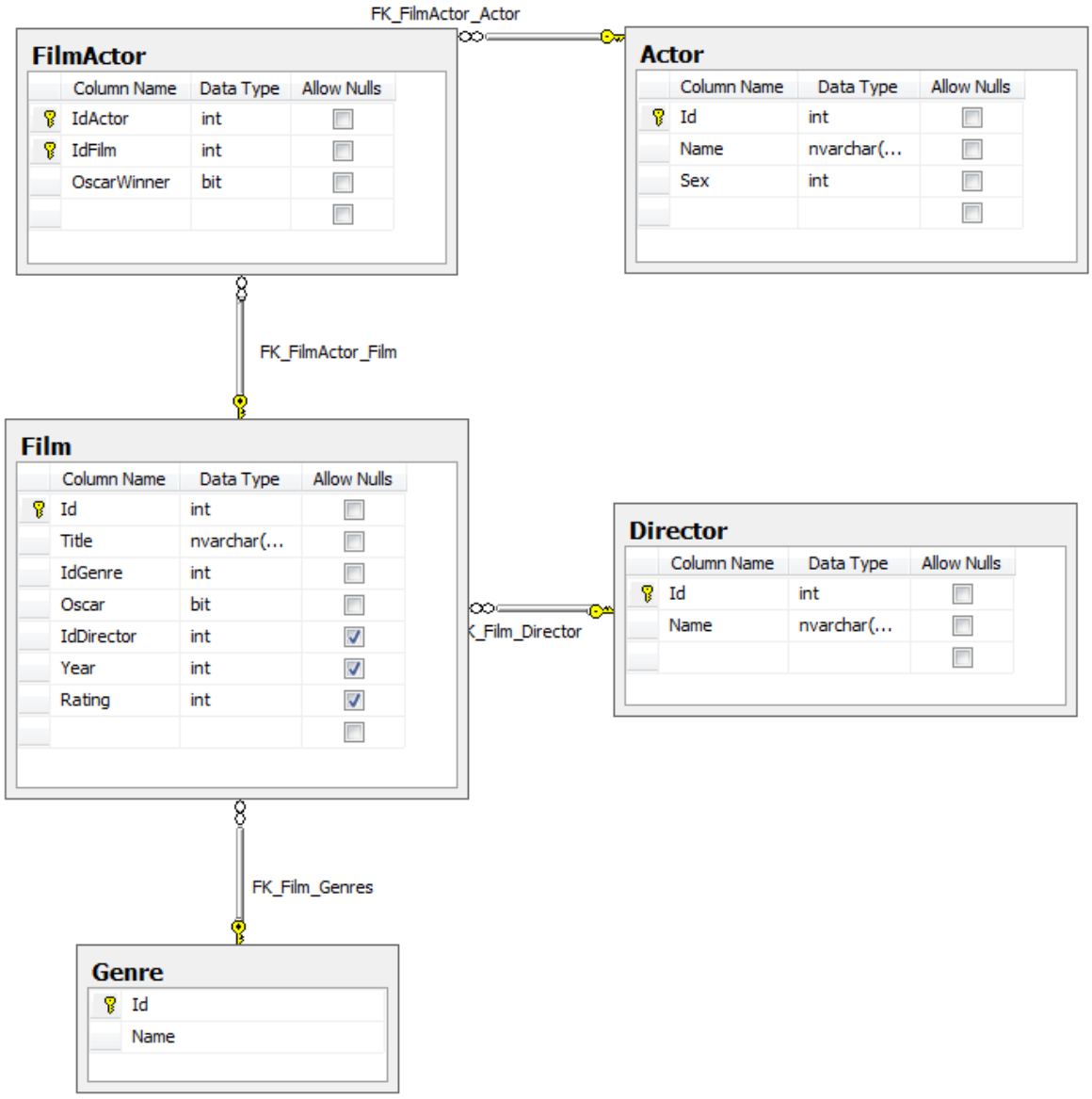

Figura 1 Esquema de la base de datos CineDB

Si se utiliza **LINQ to SQL** para realizar la conexión a la base de datos (suponiendo que esté en SQL Server) habría que escribir un código similar al del Listado 5~~Listado 5~~ para realizar una consulta equivalente a la escrita anteriormente en SQL. En principio este código está ligado al contexto de datos de tipo `SqlServerDBDataContext` y una vez más el desarrollador controla la creación del contexto de datos.

```
var dataContext = new SqlServerDBDataContext();
var query = from film in dataContext.Films
            join dir in dataContext.Directors
            on film.IdDirector equals dir.Id
```

```
            select new
            {
                Title = film.Title,
                Director = dir.Name
            };
foreach (var item in query)
    Console.WriteLine("Film: {0} Director: {1}",
        item.Title, item.Director);
```

Listado 5 Consulta LINQ que solicita las películas y el director asociado

Para abstraer la consulta LINQ primeramente será necesario crear una abstracción de las entidades que participan en la dicha consulta. En el caso de C# esta abstracción se logra mediante la declaración de interfaces que para este ejemplo serían `IActor`, `IDirector`, `IFilm`, `IFilmActor` e `IGenre` (Listado 6~~Listado 6~~). La idea que se persigue consiste en escribir de alguna forma una consulta LINQ que en tiempo de compilación dependa solamente de las interfaces y que solo en ejecución utilice alguna implementación concreta de las mismas.

```
public interface IFilm: IEntity
{
    int Id { get; set; }
    string Title { get; set; }
    int IdGenre { get; set; }
    bool Oscar { get; set; }
    int? IdDirector { get; set; }
    int? Year { get; set; }
    int? Rating { get; set; }
}

public interface IActor: IEntity
{
    int Id { get; set; }
    string Name { get; set; }
    int Sex { get; set; }
}
```

Listado 6 Interfaces que describen las entidades que se gestionarán en la base de datos

Una vez creadas estas interfaces habrá que resolver cómo integrar dichas interfaces a las entidades generadas por el ORM que se utilice y evidentemente abstraer el concepto de ORM. En principio no existe una interfaz (o clase) única implementada por los distintos ORM de .NET. Por ejemplo, **LINQ to SQL** se basa en la clase `DataContext`, pero **Entity Framework** en `ObjectContext` y **NHibernate** en `ISession`.

Inyección de Dependencias (DI), Patrón Unit of Work y patrón Repository

Hasta ahora tenemos mapeado, en el ensamblado que contiene las interfaces, a las entidades que participarán en la consulta LINQ. Sin embargo habría que mapear también a los respectivos contextos de datos para lograr una abstracción real. Para lograr este objetivo primeramente es necesario introducir algunos patrones adicionales.

El patrón **Unit of Work** [3] tiene como objetivo desde el punto de vista conceptual mantener el listado de los objetos afectados en la base de datos, así como escribir los cambios y solucionar los problemas de concurrencia que puedan aparecer. Cada uno de los tipos `DataContext`, `ObjectContext` o `ISession` de una manera u otra implementan este patrón. Por tanto, para realizar una abstracción conceptual primero definiremos la interfaz `IUnitOfWork` (Listado 7~~Listado 7~~).

```
public interface IUnitOfWork : IDisposable
{
    IRepository<TEntity> GetRepository<TEntity>()
        where TEntity : class;
    TTable GetTable<TTable>(Type type);
    void InsertOnSubmit<TEntity>(TEntity entity)
        where TEntity : class;
    void DeleteOnSubmit<TEntity>(TEntity entity)
        where TEntity : class;
    void SubmitChanges();
}
```

Listado 7~~7~~ Interface IUnitOfWork

Como podrá observar existen tres funcionalidades fundamentales: obtener un repositorio, gestionar entidades y guardar los cambios.

Un repositorio, definido por el patrón **Repository** [4], es un conjunto de entidades de un mismo tipo (en este artículo las entidades de un mismo tipo serían los Films, los Directores o los Actores) que actúa como intermediario entre entidades de la base de datos y las operaciones del dominio. El repositorio actúa como un conjunto de objetos en memoria los cuáles no se escribirán permanentemente en la base de datos hasta que no se ejecute el método `SubmitChanges` del **Unit of Work**. Note que de un mismo Unit of Work se pueden obtener más de un repositorio a la vez y hasta que no se ejecute `SubmitChanges` no se harán persistentes los cambios en la base de datos. Este es precisamente el comportamiento que siguen LINQ to SQL o Entity Framework que son dos ORM basados en estos patrones. En .NET el repositorio puede abstraerse como se muestra en el Listado 8.

```
public interface IRepository<TEntity>
    where TEntity : class
{
    IQueryable<TEntity> GetAll();
    void InsertOnSubmit(TEntity entity);
    void DeleteOnSubmit(TEntity entity);
}
public interface IRepository
{
    IQueryable GetAll();
    void InsertOnSubmit(object entity);
    void DeleteOnSubmit(object entity);
}
```

Listado 8~~8~~ Interface IRepository

Como podrá observar el repositorio está ligado a un `IUnitOfWork` de forma tal que todas las entidades estén contenidas en el `IUnitOfWork` y cada repositorio simplemente visualiza una partición de las mismas. Es importante resaltar que `GetAll()` devuelve un `IQueryable` y no un `IEnumerable`. Este comportamiento va a ser decisivo para la modelación de consultas LINQ sobre interfaces. La consulta original que se desea expresar de forma abstracta se muestra en el Listado 9~~Listado 9~~. Note que esta consulta está basada en las interfaces y el `Context` que será alguna implementación de `IUnitOfWork` será el encargado de devolver la implementación concreta para cada una de estas interfaces.

```
var query =
    from film in Context.GetRepository<IFilm>().GetAll()
    join dir in Context.GetRepository<IDirector>().GetAll()
    on film.IdDirector equals dir.Id
    select new FilmDirector { Title = film.Title, Director = dir.Name };
```

Listado 9 Consulta LINQ dependiente solo de interfaces

La idea consiste en poder obtener repositorios que contengan las entidades que implementen cierta interfaz. Cada método `GetAll` devuelve un `IQueryable<T>` donde `T` es del tipo interfaz que representa alguna entidad, por ejemplo, `IQueryable<IFilm>` e `IQueryable<IDirector>`. En la tecnología LINQ un tipo `IQueryable<T>` no se genera a código IL sino que se expresa en un **árbol de expresión**.

En este caso el árbol de expresión que representa la consulta LINQ del Listado 9 depende de interfaces y no de entidades particulares. Para hacer "funcionar" realmente dicha consulta habría que sustituir en dicho árbol cada referencia a una de estas interfaces por una implementación concreta. El Listado 10~~Listado 10~~ muestra la implementación de `IUnitOfWork`[2] utilizando LINQ to SQL dentro del ensamblado `MovieApp.SqlConnection.dll`. Hay que resaltar que el método `GetRepository<T>()` del `IUnitOfWork` utiliza Castle Windsor para devolver una instancia del repositorio. El cuerpo de este método podría haber sido implementado como `return new Repository<T>(this)` que sería la manera clásica de instanciar la clase `Repository<T>` ya que esta clase requiere para su construcción un parámetro de tipo `IUnitOfWork`. Sin embargo, utilizando la vía de IoC no fue necesaria la especificación explícita del `this` como parámetro de construcción ya que se supone que está registrada, en el contenedor, una implementación *Singleton* del `IUnitOfWork`. La habilidad de sustituir automáticamente las dependencias requeridas para construir el objeto, siempre y cuando estas se tengan registradas con una implementación concreta en el contenedor, es lo que se conoce como **Inyección de Dependencias** (DI) [5]. Note que esta labor es relativamente compleja ya que estas dependencias pueden depender de otras implementaciones y así sucesivamente formando un grafo que en otro caso el programador tendría que resolver manualmente.

```
partial class SqlServerDBDataContext : IUnitOfWork
{
    public IRepository<T> GetRepository<T>() where T : class
    {
        return Bootstrapper.Container.Resolve<IRepository<T>>();
```

[2] Por problemas de espacio solo se muestra la implementación de algunos métodos. Para ver el código completo descárguese el proyecto desde el sitio de la revista.

```
    }

    public TTable GetTable<TTable>(System.Type type)
    {
        return (TTable)this.GetTable(type);
    }
}

partial class Director : IDirector
{
}
partial class Film : IFilm
{
}
public class Repository<T> : IRepository<T>, IRepository where T : class
{
    readonly IUnitOfWork dataContext;

    public Repository(IUnitOfWork dataContextProvider)
    {
        dataContext = dataContextProvider;
    }
    private Type _getMap(Type t)
    {
        return Bootstrapper.Container.Kernel.GetHandler(t)
                .ComponentModel.Implementation;
    }
    public virtual IQueryable<T> GetAll()
    {
        return GetTable().Cast<T>();
    }
    public virtual ITable GetTable()
    {
        return dataContext.GetTable(_getMap(typeof(T)));
    }
}
```

Listado 10 Asociando las interfaces IUnitOfWork e IRepository con el contexto de datos para LINQ to SQL

Cada ensamblado externo contendrá un código de inicialización (método `Install` en el Listado 11~~Listado 11~~) que se ejecutará cuando se vincule dinámicamente éste a la aplicación principal. Castle Windsor provee un mecanismo para gestionar estos vínculos dinámicos por lo que si el desarrollador elige utilizarlo, el código de inicialización debe expresarse implementando la interfaz `IWindsorInstaller`. Note que de esta forma el ensamblado externo puede registrar en el contenedor IoC las implementaciones concretas para las interfaces que se requieren resolver, donde en este ejemplo serían las implementaciones concretas para el contexto LINQ to SQL.

```
public class Bootstrapper : IWindsorInstaller
{
    public static IWindsorContainer Container { get; private set; }

    #region IWindsorInstaller Members

    public void Install(IWindsorContainer container,
```

```
        IConfigurationStore store)
    {
        Container = container;

        container.Register(
            Component.For(typeof(IRepository<>))
            .ImplementedBy(typeof(Repository<>))
            .LifeStyle.Is(LifestyleType.Singleton));

        container.Register(
            Component.For<IUnitOfWork>()
            .ImplementedBy<SqlServerDBDataContext>()
            .LifeStyle.Is(LifestyleType.Singleton));

        container.Register(
            Component.For<IDirector>()
            .ImplementedBy<Director>()
            .LifeStyle.Is(LifestyleType.Transient));

        container.Register(
            Component.For<IFilm>()
            .ImplementedBy<Film>()
            .LifeStyle.Is(LifestyleType.Transient));

      /* entidades restantes */
    }

    #endregion
}
```

Listado 11 Punto de partida para MovieApp.SqlConnection.dll

La aplicación principal tiene que contener también un código de inicialización (Listado 12~~Listado 12~~) pero que en este caso solamente se encargue de vincular e inicializar las dependencias que existen en los ensamblados externos. Con Castle Windsor solamente debemos rellenar un Xml que exprese los ensamblados y dependencias que se quieren cargar y automáticamente el contenedor Windsor realizará dicha tarea, aunque será responsabilidad del desarrollador llamar a este inicializador principal.

```
public static class WindsorBootstrapper
{
    public static IWindsorContainer Container {get; private set;}

    public static void Initialize()
    {
        //initialize container with some IWinsorInstallers
        Container = new WindsorContainer(new XmlInterpreter());
        //delegate initialization to the configured Windsor Installers
        Container.Install(Container.ResolveAll<IWindsorInstaller>());
    }
}
```

Listado 12 Bootstrapper

Si en ejecución se le pasa un Xml vacío al contenedor, éste tratará de localizar la información de las dependencias en el fichero de configuración de la aplicación .NET (`.config`). Esto permitirá variar el tipo de conexión de base de datos simplemente modificando el valor del fichero de configuración sin necesidad de recompilar la aplicación principal (ya que la carga de los ensamblados externos se realiza dinámicamente).

El patrón Inyección de Dependencias se utiliza en este caso para mantener la definición separada de la implementación en ensamblados diferentes, permitiendo así el desarrollo de aplicaciones desacopladas.

```
var Context = WindsorBootstrapper.Container.Resolve<IUnitOfWork>();
var query =
    from film in Context.GetRepository<IFilm>().GetAll()
    join dir in Context.GetRepository<IDirector>().GetAll()
    on film.IdDirector equals dir.Id
    select new FilmDirector { Title = film.Title, Director = dir.Name };
```

Listado 13 Consulta LINQ dependiente solo de interfaces

Volviendo al asunto de cómo hacer funcionar la consulta LINQ (Listado 13~~Listado 13~~), la magia de reemplazar las interfaces del árbol de expresión por implementaciones concretas la realiza el método LINQ `Cast<T>()` (ver Listado 10~~Listado 10~~). Este método reemplaza en el árbol de expresión de un `IQueryable<T>` el tipo `T` por un cierto `TResult`. En nuestro caso se utiliza para reemplazar, por ejemplo, `IFilm` por una implementación concreta `Film` para así hacer funcional la consulta LINQ. Solo faltaría acceder al mapa que contiene las relaciones entre interfaces e implementaciones concretas. Para ello se ha creado el método auxiliar `_getMap()` dentro de la clase `Repository<T>`, que haciendo uso de la API de Castle Windsor, devuelve el tipo concreto registrado en el contenedor para una interfaz determinada. Note que haciendo estos reemplazos la consulta LINQ quedaría como si hubiese sido modelada con la implementación concreta que es al final el objetivo que estábamos persiguiendo con toda esta orquestación. El resultado de la conversión a SQL realizada por el proveedor de datos LINQ to SQL es el que aparece en el Listado 14~~Listado 14~~.

```
SELECT [t0].[Title], [t1].[Name] AS [Director]
FROM [dbo].[Film] AS [t0]
INNER JOIN [dbo].[Director] AS [t1] ON [t0].[IdDirector] = ([t1].[Id])
```

Listado 14 Código SQL generado a partir de la consulta LINQ del Listado 13

De este modo al quedar la implementación concreta para la conexión a la base de datos en un ensamblado externo a la aplicación principal, es posible tener diferentes implementaciones (por ejemplo utilizando LINQ to Objects, DBLinq, Entity Framework, NHibernate, entre otras) de las interfaces `IUnitOfWork`, `IRepository` y las diferentes `IEntity` que mapean el modelo y así lograr una aplicación con soporte para múltiples tipos de bases de datos sin tener que "casar" explícitamente en el código de la aplicación el uso de un ORM particular pero aprovechando las bondades de la modelación de consultas de esa gran tecnología que es LINQ.

Más allá de un ORM

Como prueba de concepto de que este enfoque de utilizar IoC y DI resulta ser una abstracción real, vamos a crear otro proveedor de datos utilizando **POCO** (*Plain Old CLR Object*). La idea es no utilizar ORM alguno, sino una lista en memoria de objetos .NET. Para ello hay que crear un nuevo ensamblado `MoveApp.PocoObjects.dll` e implementar nuevamente las interfaces `IUnitOfWork` e `IRepository<T>` pero ahora orientadas a **LINQ to Objects**.

```
class PocoContext: IUnitOfWork
{
    List<object> Items = new List<object>();

    public IRepository<TEntity> GetRepository<TEntity>()
        where TEntity : class
    {
        return
            Bootstrapper.Container.Resolve<IRepository<TEntity>>();
    }

    public TTable GetTable<TTable>(Type type)
    {
        return (TTable)(object)Items;
    }

    public void InsertOnSubmit<TEntity>(TEntity entity)
        where TEntity : class
    {
        Items.Add(entity);
    }

    public void DeleteOnSubmit<TEntity>(TEntity entity)
        where TEntity : class
    {
        Items.Remove(entity);
    }

    public void SubmitChanges()  { }
    public void Dispose() { }
}

class PocoRepository<TEntity> : IRepository<TEntity>, IRepository
    where TEntity : class
{
    readonly IUnitOfWork dataContext;

    public PocoRepository(IUnitOfWork pocoContext)
    {
        dataContext = dataContextProvider;
    }

    public IQueryable<TEntity> GetAll()
    {
        return dataContext.GetTable<List<object>>(null)
            .OfType<TEntity>()
```

```
            .AsQueryable<TEntity>();
    }
    /* el resto */
}
```

Como se puede apreciar, los datos se almacenan en una única lista en memoria (`List<object>`). El método más significativo en este caso sería el `GetAll()` del repositorio que utiliza los métodos extensores `OfType()` para filtrar los elementos de la lista según su tipo y luego el método extensor `AsQueryable<T>()` para permitir la realización de consultas LINQ to Objects. Adicionalmente habría que crear las clases `Film`, `Director`, `Actor`, etc pero en este caso no guardan interés ninguno ya que son meras implementaciones de sus respectivas interfaces con propiedades generadas automáticamente por el Visual Studio 2010.

El patrón Inyección de Dependencias (DI) se puede apreciar una vez más en el método `GetRepository<TEntity>()` que intenta resolver una instancia de `IRepository<TEntity>`. En el contenedor IoC (Castle Windsor) se tuvieron que haber registrado las clases `PocoContext` y `PocoRepository<TEntity>` como implementación de `IUnitOfWork` e `IRepository<TEntity>` respectivamente. Para instanciar la clase `PocoRepository<TEntity>` es imprescindible pasar como parámetro en el constructor una instancia de `IUnitOfWork`. Sin embargo, en el método `GetRepository<TEntity>()` no se tuvo que pasar explícitamente la instancia de `PocoContext` ya que el contenedor es capaz de "Inyectar" esa dependencia en la construcción del objeto repositorio.

Luego de insertar algunos datos de ejemplo en la lista, el lector podrá comprobar que las mismas consultas abstractas declaradas previamente van a funcionar de manera transparente. La utilidad práctica de este "proveedor de datos" que simplemente almacena objetos en memorias está ligada a las pruebas unitarias para verificar la calidad del software [6].

Conclusiones

Hemos revisado los patrones **Inversion of Control** (IoC), **Dependency Injection** (DI), **Unit of Work** y **Repository**, que siguen mostrándonos su vigencia y utilidad. IoC surgió conceptualmente en 1998 y la implementación para .NET se viene promoviendo desde el 2008. Incluso componentes novedosos como **MEF** (*Managed Extensibility Framework*) utilizan o basan su funcionamiento de alguna forma en estos patrones [7]. De modo que conocer o repasar estos patrones de diseño debe resultar de utilidad al lector. Su adecuada aplicación puede ayudar a producir soluciones más elegante y flexibles. En el caso de la gestión de bases de datos, independientemente del ORM que se seleccione, si se apuesta por LINQ claro está, utilizar una implementación similar a la que hemos esbozado en este artículo nos permitiría cambiar de proveedor sin tener que modificar el modelo conceptual de las consultas y la lógica del negocio de la aplicación.

Bibliografía

1. **Fowler, Martin.** Inversion of Control Containers and the Dependency Injection pattern. [En línea] 23 de Enero de 2004. http://www.martinfowler.com/articles/injection.html.

2. **Steele, Patrick.** Inversion of Control Patterns for the Microsoft .NET Framework. [En línea] 1 de Agosto de 2010. http://visualstudiomagazine.com/articles/2010/08/01/inversion-of-control-patterns-for-the-microsoft-net-framework.aspx.

3. **Fowler, Martin.** Unit of Work. [En línea] http://martinfowler.com/eaaCatalog/unitOfWork.html.

4. —. Repository. [En línea] http://martinfowler.com/eaaCatalog/repository.html.

5. —. Dependency Injection. [En línea] http://martinfowler.com/articles/injection.html.

6. 100% UNIT TESTABLE LINQ TO SQL REPOSITORY. [En línea] http://weblogs.asp.net/rashid/archive/2009/02/19/100-unit-testable-linq-to-sql-repository.aspx.

7. MEF is not An IoC container; but MEF uses IoC. [En línea] http://msmvps.com/blogs/peterritchie/archive/2010/02/24/mef-is-not-an-ioc-container-but-mef-uses-ioc.aspx.