عنوان: **آشنایی با NHibernate - قسمت هشتم**
نویسنده: وحید نصیری
تاریخ: ۱۳۸۸/۰۷/۲۵ ۱۸:۵۷:۰۰
آدرس: www.dotnettips.info
برچسب‌ها: NHibernate

معرفی الگوی Repository

روش متداول کار با فناوری‌های مختلف دسترسی به داده‌ها عموما بدین شکل است:
الف) یافتن رشته اتصالی رمزنگاری شده به دیتابیس از یک فایل کانفیگ (در یک برنامه اصولی البته!)
ب) باز کردن یک اتصال به دیتابیس
ج) ایجاد اشیاء Command برای انجام عملیات مورد نظر
د) اجرا و فراخوانی اشیاء مراحل قبل
ه) بستن اتصال به دیتابیس و آزاد سازی اشیاء

اگر در برنامه‌های یک تازه کار به هر محلی از برنامه او دقت کنید این 5 مرحله را می‌توانید مشاهده کنید. همه جا! قسمت ثبت، قسمت جستجو، قسمت نمایش و ...
مشکلات این روش:
-1 حجم کارهای تکراری انجام شده بالا است. اگر قسمتی از فناوری دسترسی به داده‌ها را به اشتباه درک کرده باشد، پس از مطالعه بیشتر و مشخص شدن نحوه‌ی رفع مشکل، قسمت عمده‌ای از برنامه را باید اصلاح کند (زیرا کدهای تکراری همه جای آن پراکنده‌اند).
-2 برنامه نویس هر بار باید این مراحل را به درستی انجام دهد. اگر در یک برنامه بزرگ تنها قسمت آخر در یکی از مراحل کاری فراموش شود دیر یا زود برنامه تحت فشار کاری بالا از کار خواهد افتاد (و متاسفانه این مساله بسیار شایع است).
-3 برنامه منحصرا برای یک نوع دیتابیس خاص تهیه خواهد شد و تغییر این رویه جهت استفاده از دیتابیسی دیگر (مثلا کوچ برنامه از اکسس به اس کیوال سرور)، نیازمند بازنویسی کل برنامه می‌باشد.
و ...

همین برنامه نویس پس از مدتی کار به این نتیجه می‌رسد که باید برای این‌کارهای متداول، یک لایه و کلاس دسترسی به داده‌ها را تشکیل دهد. اکنون هر قسمتی از برنامه برای کار با دیتابیس باید با این کلاس مرکزی که انجام کارهای متداول با دیتابیس را خلاصه می‌کند، کار کند. به این صورت کد نویسی یک نواختی با حذف کدهای تکراری از سطح برنامه و همچنین بدون فراموش شدن قسمت مهمی از مراحل کاری، حاصل می‌گردد. در اینجا اگر روزی قرار شد از یک دیتابیس دیگر استفاده شود فقط کافی است یک کلاس برنامه تغییر کند و نیازی به بازنویسی کل برنامه نخواهد بود.

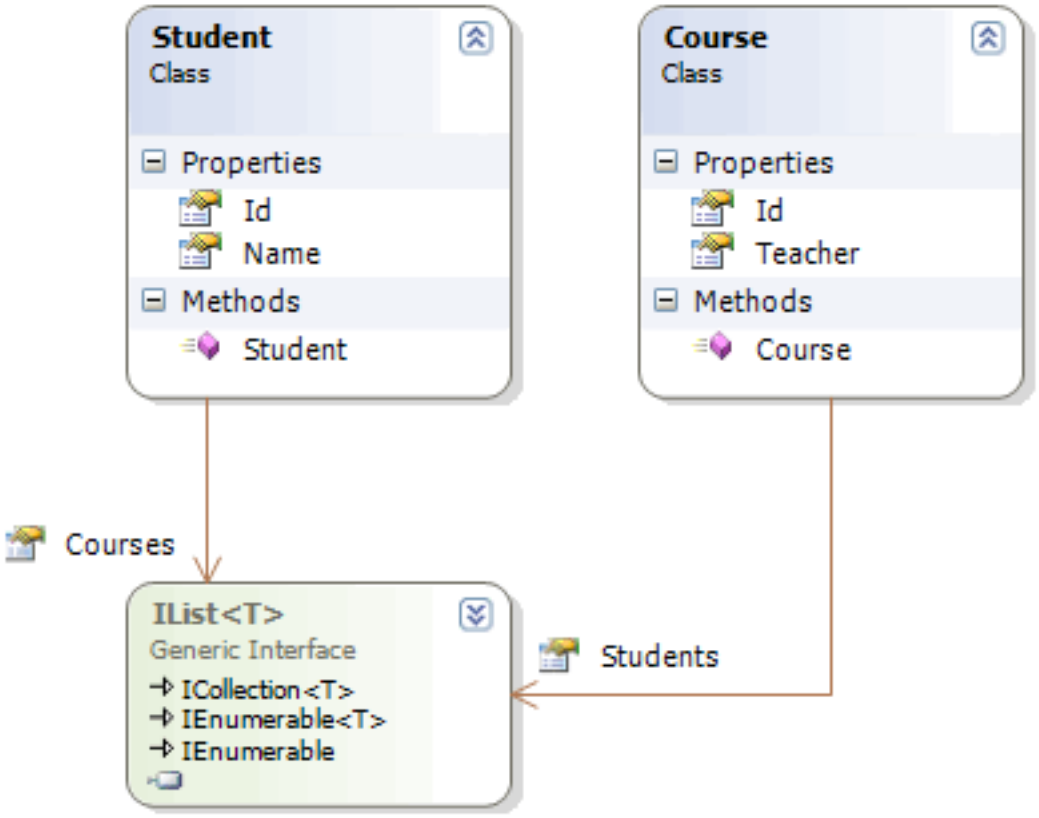
این روزها تشکیل این لایه دسترسی به داده‌ها (data access layer یا DAL) نیز مزموم است! و دلایل آن در مباحث چرا به یک ORM نیازمندیم برشمرده شده است. جهت کار با ORM ها نیز نیازمند یک لایه دیگر می‌باشیم تا یک سری اعمال متداول با آن‌ها را کپسوله کرده و از حجم کارهای تکراری خود بکاهیم. برای این منظور قبل از اینکه دست به اختراع بزنیم، بهتر است به الگوهای طراحی برنامه نویسی شیء گرا رجوع کرد و از رهنمودهای آن استفاده نمود.

الگوی Repository یکی از الگوهای برنامه نویسی با مقیاس سازمانی است. با کمک این الگو لایه‌ای بر روی لایه نگاشت اشیاء برنامه به دیتابیس تشکیل شده و عملا برنامه را مستقل از نوع ORM مورد استفاه می‌کند. به این صورت هم از تشکیل یک سری کدهای تکراری در سطح برنامه جلوگیری شده و هم از وابستگی بین مدل برنامه و لایه دسترسی به داده‌ها (که در اینجا همان NHibernate می‌باشد) جلوگیری می‌شود. الگوی Repository (مخزن)، کار ثبت، حذف، جستجو و به روز رسانی داده‌ها را با ترجمه آن‌ها به روش‌های بومی مورد استفاده توسط ORM مورد نظر، کپسوله می‌کند. به این شکل شما می‌توانید یک الگوی مخزن عمومی را برای کارهای خود تهیه کرده و به سادگی از یک ORM به ORM دیگر کوچ کنید؛ زیرا کدهای برنامه شما به هیچ ORM خاصی گره نخورده و این عملیات بومی کار با ORM توسط لایه‌ای که توسط الگوی مخزن تشکیل شده، صورت گرفته است.

طراحی کلاس مخزن باید شرایط زیر را برآورده سازد:
الف) باید یک طراحی عمومی داشته باشد و بتواند در پروژه‌های متعددی مورد استفاده مجدد قرار گیرد.
ب) باید با سیستمی از نوع اول طراحی و کد نویسی و بعد کار با دیتابیس، سازگاری داشته باشد.
ج) باید امکان انجام آزمایشات واحد را سهولت بخشد.
د) باید وابستگی کلاس‌های دومین برنامه را به زیر ساخت ORM مورد استفاده قطع کند (اگر سال بعد به این نتیجه رسیدید که ORM ایی به نام XYZ برای کار شما بهتر است، فقط پیاده سازی این کلاس باید تغییر کند و نه کل برنامه).
ه) باید استفاده از کوئری‌هایی از نوع strongly typed را ترویج کند (مثل کوئری‌هایی از نوع LINQ).

بررسی مدل برنامه

مدل این قسمت (برنامه NHSample4 از نوع کنسول با همان ارجاعات متداول ذکر شده در قسمت‌های قبل)، از نوع many-to-many می‌باشد. در اینجا یک واحد درسی توسط چندین دانشجو می‌تواند اخذ شود یا یک دانشجو می‌تواند چندین واحد درسی را اخذ نماید که برای نمونه کلاس دیاگرام و کلاس‌های متشکل آن به شکل زیر خواهند بود:




```
using System.Collections.Generic;

namespace NHSample4.Domain
{
 public class Course
 {
     public virtual int Id { get; set; }
     public virtual string Teacher { get; set; }
     public virtual IList<Student> Students { get; set; }

     public Course()
     {
```

```
        Students = new List<Student>();
    }
 }
}
```

```
using System.Collections.Generic;

namespace NHSample4.Domain
{
 public class Student
 {
    public virtual int Id { get; set; }
    public virtual string Name { get; set; }
    public virtual IList<Course> Courses { get; set; }

    public Student()
    {
        Courses = new List<Course>();
    }
 }
}
```

کلاس کانفیگ برنامه جهت ایجاد نگاشت‌ها و سپس ساخت دیتابیس متناظر

```
using FluentNHibernate.Automapping;
using FluentNHibernate.Cfg;
using FluentNHibernate.Cfg.Db;
using NHibernate.Tool.hbm2ddl;

namespace NHSessionManager
{
 public class Config
 {
    public static FluentConfiguration GetConfig()
    {
        return
          Fluently.Configure()
                  .Database(
                     MsSqlConfiguration
                       .MsSql2008
                       .ConnectionString(x => x.FromConnectionStringWithKey("DbConnectionString"))
                      )
                    .Mappings(
                      m => m.AutoMappings.Add(
                         new AutoPersistenceModel()
                             .Where(x => x.Namespace.EndsWith("Domain"))
                             .AddEntityAssembly(typeof(NHSample4.Domain.Course).Assembly))
                             .ExportTo(System.Environment.CurrentDirectory)
                             );
    }

    public static void CreateDb()
    {
        bool script = false;//آیا خروجی در کنسول هم نمایش داده شود
        bool export = true;//آیا بر روی دیتابیس هم اجرا شود
        bool dropTables = false;//آیا جداول موجود دراپ شوند
        new SchemaExport(GetConfig().BuildConfiguration()).Execute(script, export, dropTables);
    }
 }
}
```

چند نکته در مورد این کلاس:

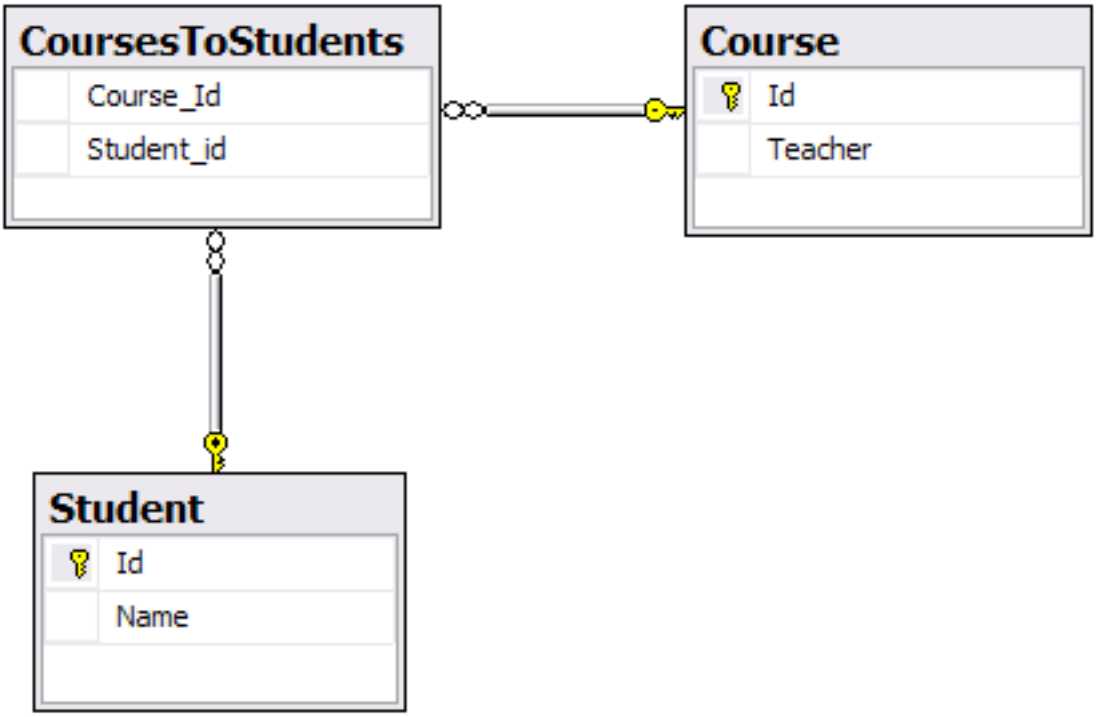
الف) با توجه به اینکه برنامه از نوع ویندوزی است، برای مدیریت صحیح کانکشن استرینگ، فایل App.Config را به برنامه افزوده و محتویات آن‌را به شکل زیر تنظیم می‌کنیم (تا کلید DbConnectionString توسط متد GetConfig مورد استفاده قرارگیرد):

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
<connectionStrings>
 <!--NHSessionManager-->
```

```
 <add name="DbConnectionString"
      connectionString="Data Source=(local);Initial Catalog=HelloNHibernate;Integrated Security =
true"/>
</connectionStrings>
</configuration>
```

ب) در NHibernate سنتی (!) کار ساخت نگاشت‌ها توسط یک سری فایل xml صورت می‌گیرد که با معرفی فریم ورک Fluent NHibernate و استفاده از قابلیت‌های Auto Mapping آن، این‌کار با سهولت و دقت هر چه تمام‌تر قابل انجام است که توضیحات نحوه‌ی انجام آن‌را در قسمت‌های قبل مطالعه فرمودید. اگر نیاز بود تا این فایل‌های XML نیز جهت بررسی شخصی ایجاد شوند، تنها کافی است از متد ExportTo آن همانگونه که در متد GetConfig استفاده شده، کمک گرفته شود. به این صورت پس از ایجاد خودکار نگاشت‌ها، فایل‌های XML متناظر نیز در مسیری که به عنوان آرگومان متد ExportTo مشخص گردیده است، تولید خواهند شد (دو فایل NHSample4.Domain.Course.hbm.xml و NHSample4.Domain.Student.hbm.xml را در پوشه‌ای که محل اجرای برنامه است خواهید یافت).

با فراخوانی متد CreateDb این کلاس، پس از ساخت خودکار نگاشت‌ها، database schema متناظر، در دیتابیسی که توسط کانکشن استرینگ برنامه مشخص شده، ایجاد خواهد شد که دیتابیس دیاگرام آن‌را در شکل ذیل مشاهده می‌نمائید (جداول دانشجویان و واحدها هر کدام به صورت موجودیتی مستقل ایجاد شده که ارجاعات آن‌ها در جدولی سوم نگهداری می‌شود).



پیاده سازی الگوی مخزن

اینترفیس عمومی الگوی مخزن به شکل زیر می‌تواند باشد:

```
using System;
using System.Linq;
using System.Linq.Expressions;
```

```
namespace NHSample4.NHRepository
{
 //Repository Interface
 public interface IRepository<T>
 {
     T Get(object key);

     T Save(T entity);
     T Update(T entity);
     void Delete(T entity);

     IQueryable<T> Find();
     IQueryable<T> Find(Expression<Func<T, bool>> predicate);
 }
}
```

سپس پیاده سازی آن با توجه به کلاس SingletonCore ایی که در قسمت قبل تهیه کردیم (جهت مدیریت صحیح سشن فکتوری)، به صورت زیر خواهد بود.
این کلاس کار آغاز و پایان تراکنش‌ها را نیز مدیریت کرده و جهت سهولت کار اینترفیس IDisposable را نیز پیاده سازی می‌کند :

```
using System;
using System.Linq;
using NHSessionManager;
using NHibernate;
using NHibernate.Linq;

namespace NHSample4.NHRepository
{
 public class Repository<T> : IRepository<T>, IDisposable
 {
     private ISession _session;
     private bool _disposed = false;

     public Repository()
     {
         _session = SingletonCore.SessionFactory.OpenSession();
         BeginTransaction();
     }

     ~Repository()
     {
         Dispose(false);
     }

     public T Get(object key)
     {
         if (!isSessionSafe) return default(T);

         return _session.Get<T>(key);
     }

     public T Save(T entity)
     {
         if (!isSessionSafe) return default(T);

         _session.Save(entity);
         return entity;
     }

     public T Update(T entity)
     {
         if (!isSessionSafe) return default(T);

         _session.Update(entity);
         return entity;
     }

     public void Delete(T entity)
     {
         if (!isSessionSafe) return;

         _session.Delete(entity);
     }

     public IQueryable<T> Find()
```

```
    {
        if (!isSessionSafe) return null;

        return _session.Linq<T>();
    }

    public IQueryable<T> Find(System.Linq.Expressions.Expression<Func<T, bool>> predicate)
    {
        if (!isSessionSafe) return null;

        return Find().Where(predicate);
    }

    void Commit()
    {
        if (!isSessionSafe) return;

        if (_session.Transaction != null &&
            _session.Transaction.IsActive &&
            !_session.Transaction.WasCommitted &&
            !_session.Transaction.WasRolledBack)
        {
            _session.Transaction.Commit();
        }
        else
        {
            _session.Flush();
        }
    }

    void Rollback()
    {
        if (!isSessionSafe) return;

        if (_session.Transaction != null && _session.Transaction.IsActive)
        {
            _session.Transaction.Rollback();
        }
    }

    private bool isSessionSafe
    {
        get
        {
            return _session != null && _session.IsOpen;
        }
    }

    void BeginTransaction()
    {
        if (!isSessionSafe) return;

        _session.BeginTransaction();
    }


    public void Dispose()
    {
        Dispose(true);
        // tell the GC that the Finalize process no longer needs to be run for this object.
        GC.SuppressFinalize(this);
    }

    protected virtual void Dispose(bool disposeManagedResources)
    {
        if (_disposed) return;
        if (!disposeManagedResources) return;
        if (!isSessionSafe) return;

        try
        {
            Commit();
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.ToString());
            Rollback();
        }
        finally
        {
            if (isSessionSafe)
```

```
            {
                _session.Close();
                _session.Dispose();
            }
        }

        _disposed = true;
    }
 }
}
```

اکنون جهت استفاده از این کلاس مخزن به شکل زیر می‌توان عمل کرد:

```
using System;
using System.Collections.Generic;
using NHSample4.Domain;
using NHSample4.NHRepository;

namespace NHSample4
{
 class Program
 {
     static void Main(string[] args)
     {
         //ایجاد دیتابیس در صورت نیاز
         //NHSessionManager.Config.CreateDb();


         //ابتدا یک دانشجو را اضافه می‌کنیم
         Student student = null;
         using (var studentRepo = new Repository<Student>())
         {
             student = studentRepo.Save(new Student() { Name = "Vahid" });
         }

         //سپس یک واحد را اضافه می‌کنیم
         using (var courseRepo = new Repository<Course>())
         {
             var course = courseRepo.Save(new Course() { Teacher = "Shams" });
         }

         //اکنون یک واحد را به دانشجو انتساب می‌دهیم
         using (var courseRepo = new Repository<Course>())
         {
             courseRepo.Save(new Course() { Students = new List<Student>() { student } });
         }

         //سپس شماره دروس استادی خاص را نمایش می‌دهیم
         using (var courseRepo = new Repository<Course>())
         {
             var query = courseRepo.Find(t => t.Teacher == "Shams");

             foreach (var course in query)
                 Console.WriteLine(course.Id);
         }

         Console.WriteLine("Press a key...");
         Console.ReadKey();
     }
 }
}
```

همانطور که ملاحظه می‌کنید در این سطح دیگر برنامه هیچ درکی از ORM مورد استفاده ندارد و پیاده سازی نحوه‌ی تعامل با NHibernate در پس کلاس مخزن مخفی شده است. کار آغاز و پایان تراکنش‌ها به صورت خودکار مدیریت گردیده و همچنین آزاد سازی منابع را نیز توسط اینترفیس IDisposable مدیریت می‌کند. به این صورت امکان فراموش شدن یک سری از اعمال متداول به حداقل رسیده، میزان کدهای تکراری برنامه کم شده و همچنین هر زمانیکه نیاز بود، صرفا با تغییر پیاده سازی کلاس مخزن می‌توان به ORM دیگری کوچ کرد؛ بدون اینکه نیازی به بازنویسی کل برنامه وجود داشته باشد.

دریافت سورس برنامه قسمت هشتم

ادامه دارد ...

نظرات خوانندگان

نویسنده: iMAN
تاریخ: ۱۳۸۸/۰۷/۲۵ ۲۱:۵۵:۳۱

واقعاً خسته نباشید آقای نصیری، مجموعه مقالات آشنایی با NHibernate شما عالی است، باعث شد من شروع به یادگیری NHibernate کنم. ممنونم

نویسنده: Mahdi
تاریخ: ۱۳۸۸/۰۷/۲۸ ۱۰:۵۲:۲۴

دوست عزیز. سری آموزش NH واقعا یک کار نمونه و عالی هست که انجام دادید. خسته نباشید و ممنون.

راستی گویا پیدا کردن شما روی سایتهای Social مثل Twitter یا Facebook و ... کار ساده ای نیست! به هر حال از آشنایی بیشتر با شما خوشوقت خواهم شد.

نویسنده: وحید نصیری
تاریخ: ۱۳۸۸/۰۷/۲۸ ۱۵:۰۱:۲۹

با سلام
و با تشکر از لطف دوستان.

بله. بنده در سایت‌های social به دلایل شخصی حضور ندارم. از لطف شما سپاسگزارم.

نویسنده: Majid
تاریخ: ۱۳۸۸/۱۱/۲۲ ۲۱:۲۸:۰۸

جناب نصیری عزیز
از شما به خاطر مطالب بسیار مفیدتان قدردانی می‌کنم
اگر امکان دارد طریقه استفاده از NHibernate در Windows Form (نمایش Objectها در Grid و کار با ابزارهای گزارش‌گیری و ...) را نیز آموزش دهید.
پاینده باشید.

نویسنده: Ahmad
تاریخ: ۱۳۸۹/۰۷/۲۲ ۰۱:۴۳:۳۳

سلام.
فرض کنید همچین کدی نوشته ایم:

```
using (var repository = new Repository())
{
try
{
// Some Code ...
repository.Update(myClass);
// Or
repository.Delete(myClass);
return true;

{
```

```
(catch (Exception
}
MessageBox.Show("خطا در ویرایش یا حذف");
{
{
```

در صورت بروز خطا هیچ وقت بلاک catch اجرا نمی شود.
البته هنگام return true متد Dispose کلاس Repository اجرا می شود.(نوش دارو پس از مرگ سهراب...)

نویسنده: وحید نصیری
تاریخ: ۱۳۸۹/۰۷/۲۲ ۱۱:۳۸:۴۵

سلام،
کاری را که شما دارید انجام می‌دید اشتباه است.
Using statement در سی شارپ به صورت خودکار به try/finally به همراه dispose شیء احاطه شده توسط آن ترجمه می‌شود. (+)
به عبارتی شما یک try/finally را در یک سطح بالاتر دارید و داخل آن یک try/catch قرار داده‌اید. اینکار صحیح نیست. Using را حذف کنید و try/catch/finally را جایگزین تمام موارد اضافه شده کنید.
ضمنا توصیه من این است که فقط try/catch را حذف کنید. Using سرجایش باشد تا هدف اصلی آن یعنی dispose اشیاء مرتبط حتما رخ دهد.
به تمام برنامه نویس‌ها آموزش داده می‌شود که exception handling چیست اما در پایان فصل به آن‌ها آموخته نمی‌شود که لطفا تا حد امکان از آن استفاده نکنید! بله، لطفا در استفاده از آن خساست به خرج دهید. چرا؟ چون کرش بر خلاف تصور عمومی چیز خوبی است! زمانیکه شما این try/catch را قرار دادید، flow برنامه متوجه نخواهد شد که در مرحله‌ی قبل مشکلی رخ داده و از ادامه برنامه و خسارت وارد کردن به سیستم جلوگیری کند. ضمنا این را هم به خاطر داشته باشید که exception ها در دات نت حبابی هستند. یعنی به فراخوان خود منتشر خواهند شد و در یک سطح بالاتر هم قابل catch هستند (با تمام جزئیات).

نویسنده: Ahmad
تاریخ: ۱۳۸۹/۰۷/۲۲ ۱۹:۵۶:۴۳

با تشکر از تذکر شما.ولی کد زیر هم در صورت بروز خطا چیزی را برنمی گرداند. یعنی متد Delete چه با موفقیت به پایان برسد یا خطایی در حذف رکورد رخ دهد باز return true; اجرا می شود.!؟

```
using(var repository = new Repository())
}
repository.Delete(myClass);
return true;
{
```

نویسنده: وحید نصیری
تاریخ: ۱۳۸۹/۰۷/۲۲ ۲۱:۲۱:۵۷

این مورد اصلا ربطی به try/catch , using و غیره ندارد. نیاز به solution کار شما است (با تمام کلاس‌ها و نگاشت و غیره) تا بتوان آن‌را دیباگ کرد. بهترین روش هم این است که خروجی SQL تولید شده را بررسی کنید تا متوجه شوید مشکل کار در کجاست.

نویسنده: وحید نصیری
تاریخ: ۱۳۸۹/۰۷/۲۲ ۲۲:۵۸:۲۲

ابزار حرفه‌ای مشاهده خروجی NHibernate برنامه زیر است (که در جهت دیباگ کار بسیار مفید است):
NHProf

کار کردن با آن هم بسیار ساده است. فایل How to use.txt آن‌را مطالعه کنید..

نویسنده: A
تاریخ: ۱۳۸۹/۰۹/۱۳ ۲۳:۵۱:۵۷

اگر نیاز به Transaction داشته باشیم در این مدل حتماً باید از TransactionScope استفاده کنیم؟ اگر این‌طور است فکر می‌کنم این مدل ضعیف باشد.
فکر می‌کنم (انتظار من این است) وقتی ORM وجود دارد باید بتوانیم کارها را در صف نگه داشته و یکجا اعمال کنیم.

البته منظورم این است که باید بتوان بین چند Table جداگانه (با ارتباط یا بی ارتباط) این کار را انجام داد.

من یک مدل درآورده‌ام که تا کنون نیاز من را بدون استفاده از TransactionScope برای کارهای تراکنشی برطرف کرده. شبیه همین مدل است با کمی تغییر. نمی‌دانم آیا می‌توانم آنرا مدل Repository بنامم یا خیر؟

نویسنده: وحید نصیری
تاریخ: ۱۳۸۹/۰۹/۱۴ ۰۰:۰۳:۰۹

حق با شما است. روش صحیح لایه بندی این قسمت بر اساس تعریف unit of work و سپس repository است. یک unit of work می‌تواند از اعمال حاصل چندین repository تشکیل شده و نهایتا در پایان کار همه را یکجا اعمال کند. در مثال فوق این دو مفهوم با هم تلفیق شده.
بنابراین اگر علمی‌تر می‌خواهید کار کنید در مورد unit of work تحقیق کنید (در سایت nhforge.org).

نویسنده: A
تاریخ: ۱۳۸۹/۰۹/۱۴ ۰۸:۴۵:۰۳

بله در پشت صحنه مدلی که گفتم شبیه UoW است. اما برنامه‌نویس تقریباً مانند Repository با آن کار می‌کند و خیلی از مسائل در پشت صحنه برای او حل می‌شود.

خیلی ممنون از اطلاعات. وبلاگتان هم بسیار عالیست.

نویسنده: shayan
تاریخ: ۱۳۸۹/۱۰/۲۸ ۱۶:۵۶:۴۸

با سلام،
فرض کنید در Table CoursesToStudents فیلدی به نام IsApproved را می خواهیم داشته باشیم، در اینصورت کلاس های نگاشت به چه صورت خواهد بود؟ در کدام کلاس نگاشت پیاده سازی می شود؟ اگر کلاس جداگانه ایی تعریف کنیم ایا باز هم رابطه ManyToMany برقرار خواهد بود؟

با تشکر

نویسنده: وحید نصیری
تاریخ: ۱۳۸۹/۱۰/۲۸ ۱۸:۱۵:۵۳

سلام،
زمانیکه با ORM هایی از نوع Code First کار می‌کنید مثل NHibernate یا مثل نگارش بعدی Entity framework ، ذهن خودتون رو از وجود جداول حاضر در بانک اطلاعاتی خالی کنید. جدولی به نام CoursesToStudents توسط ساز و کار درونی NHibernate مدیریت خواهد شد و لزومی ندارد برنامه در مورد آن اطلاعاتی داشته باشد.
موردی را که شما نیاز دارید کلاسی است به نام نتایج دوره؛ مثلا چیزی به نام CourseResult . این کلاس ارجاعاتی را به شیء دانشجو و شیء دوره دارد، به همراه نمره نهایی یا مثلا خاصیت قبول شده و برای مثال تاریخ امتحان و خواص دیگری که صلاح

می‌دانید.
زمانیکه NHibernate اسکریپت اعمال این نگاشت‌ها را تشکیل دهد (توسط امکانات کلاس SchemaExport که در مطلب بالا ذکر شده)، در جدول نهایی بانک اطلاعاتی شما به ازای ارجاعات به اشیاء یاد شده، یک کلید خارجی خواهید داشت. این کلاس جدید تاثیری روی سایر روابط ندارد.

نویسنده: Amir
تاریخ: ۱۳۸۹/۱۲/۰۱ ۱۳:۲۴:۳۷

سلام با تشکر از مطلب خوبتون
لینک زیر نحوه ترکیب Repository با EF رو توضیح داده که البته ادامه داره و در پست های بعدی مطلب رو تکمیل و به روز کرده
/http://huyrua.wordpress.com/2010/07/13/entity-framework-4-poco-repository-and-specification-pattern

نویسنده: وحید نصیری
تاریخ: ۱۳۸۹/۱۲/۰۱ ۱۵:۲۴:۳۴

ممنون. الگوهای طراحی برنامه نویسی شیءگرا یک حالت عمومی دارند. یعنی مختص به یک فناوری یا زبان خاص یا حتی یک محصول خاص نیستند. بگردید برای LINQ to SQL هم پیاده سازی الگوی Repository وجود دارد.
کلا استفاده‌ی از هر کدام از ORMs موجود بدون پیاده سازی الگوی Repository اشتباه است. به چند دلیل:
- مخفی کردن ساز و کار درونی یک ORM : برای مثال من جدا قصد ندارم این رو حفظ کنم که فلان ORM خاص چطور Insert انجام می‌دهد. من فقط می‌خواهم یک متد Insert داشته باشم. یکبار این رو در الگوی Repository پیاده سازی می‌کنم و بعد فراموش می‌کنم که این ORM الان EF است یا NH یا هرچی
- امکان تعویض کلی یک ORM : زمانیکه من در کدهای BLL خودم فقط از متد Insert پیاده سازی شده مطابق رهنمون‌های الگوی Repository استفاده کردم، دیگر BLL درکی از ORM نخواهد داشت. برای کوچ کردن به یک ORM دیگر فقط کافی است تا Repository را عوض کرد. مابقی برنامه دست نخورده باقی می‌ماند.
- نوشتن Unit test با استفاده از الگوی Repository ساده‌تر است: این الگو چون بر مبنای یک Interface پیاده سازی می‌شود، امکان Mocking این Interface در Unit tests ساده‌تر است.

نویسنده: شاهین کیاست
تاریخ: ۱۳۸۹/۱۲/۰۳ ۲۰:۴۹:۳۱

سلام.
من تا قبل فکر می کردم برای گرفتن کارایی از یک کلاس که اعمال آن پیچیده هست مثل همین Insert در یک ORM که مثال زدید باید از لایه Facade استفاده کرد.
لطفا اگر ممکن هست در رابطه با فرق این 2 یک توضیح بدید.
ممنون

نویسنده: وحید نصیری
تاریخ: ۱۳۸۹/۱۲/۰۳ ۲۲:۰۳:۲۳

برای اینکه از بحث دور نشیم، در NHibernate این الگوها قابل مشاهده است:
الگوی Facade که همان session و ISession معروف آن است و کار آن فراخوانی تعداد قابل ملاحظه‌ای زیر سیستم و مخفی کردن آن‌ها از دید کاربر نهایی است. به این صورت شما توانایی کار کردن با انواع بانک‌های اطلاعاتی را بدون درگیر شدن با جزئیات آن‌ها از طریق یک اینترفیس عمومی پیدا می‌کنید.
الگوی proxy که پایه و اساس lazy loading آن است.
الگوی object pool جهت مدیریت اتصالات آن به بانک اطلاعاتی
الگوی Interpreter جهت مدیریت کوئری‌های ویژه آن
و ...

الگوی Repository هم یک نوع نگارش سفارشی الگوی Facade است. در اینجا یک سیستم پیچیده (یک سطح بالاتر است از ISession که کارش مخفی کردن ساختار داخلی NHibernate است) یا همان ORM مورد نظر را دریافت کرده و یک اینترفیس ساده، قابل درک و عمومی را از آن را ارائه می‌دهد. هدف آن مخفی کردن ریز جزئیات روش کار با یک ORM خاص است. به همین جهت به آن Persistence Ignorance هم گفته می‌شود.
الگوی Repository یکی از الگوهای اصلی DDD یا Domain driven design است.

نویسنده: Hamidrezabina
تاریخ: ۱۳۹۰/۰۲/۰۱ ۱۹:۴۰:۴۲

با سلام . . .

چطوری میشه تمام مواردی که میخواهیم ثبت کنیم بر اساس یه transaction کار کنند ؟
مثلا من یه فاکتور فروش دارم که تمام موارد باید با هم ثبت بشوند یا هیچکدوم نشوند.
میشه بفرمائید چطوری باید پیاده سازیش کرد ؟

نویسنده: وحید نصیری
تاریخ: ۱۳۹۰/۰۲/۰۱ ۲۰:۴۵:۵۷

پیاده سازی الگوی مخزن در مطلب بالا از دیدی که مطرح کردید ایراد دارد. چون به ازای هر موجودیت یک تراکنش لحاظ می‌کند.
روش صحیح پیاده سازی مورد نظر شما استفاده از الگوی unit of work است
این الگو یک سطح بالاتر از الگوی مخزن قرار می‌گیرد
اگر می‌خواهید با نحوه‌ی پیاده سازی آن آشنا شوید به این پروژه مراجعه کنید
/http://efrepository.codeplex.com
هر چند برای EF نوشته شده ولی از دیدگاه طراحی اینترفیس و روابط نهایی برای تمام ORM های دیگر هم صادق است و فرقی نمی‌کند

نویسنده: وحید نصیری
تاریخ: ۱۳۹۰/۰۲/۰۳ ۱۶:۳۸:۵۸

جهت یادآوری...
در مورد unit of work در سایت http://nhforge.org جستجو کنید.