Лабораторная работа №5

Использование шаблона «Repository» для работы с БД

Задачи

Реализовать репозитории сущностей для ваших хранимых данных

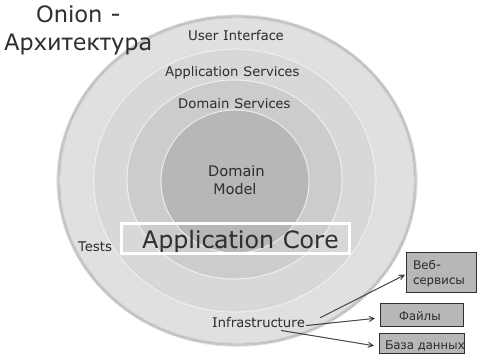
Применить шаблон в приложении с графическим интерфейсом, в соответствии с вашей темой

# Теоретические сведения

## Многослойная архитектура

Термин "Onion Architecture" ("луковая" архитектура) был предложен Джеффри Палермо (Jeffrey Palermo) еще в 2008 году. Спустя годы данная концепция стала довольно популярной и является одной из наиболее применяемых типов архитектуры при построении приложения на ASP.NET.

Onion-архитектура представляет собой разделение приложения на уровни. При чем есть один независимый уровень, который находится в центре архитектуры. От этого уровня зависит второй уровень, от второго - третий и так далее. То есть получается, что вокруг первого независимого уровня наслаивается второй-зависимый. Вокруг второго наслаивается третий, который также может зависеть и от первого. Образно это может быть выражено в виде лука, в котором также есть сердцевина, вокруг которого наслаиваются все остальные слои, вплоть до шелухи.



Количество уровней может отличаться, но в центре всегда находится модель домена (Domain Model), то есть те классы моделей, которые используются в приложении и объекты которых хранятся в базе данных:

Первый уровень вокруг модели домена образуют интерфейсы, которые управляют работой с моделью домена. Обычно это интерфейсы репозиториев, через которые мы взаимодействуем с базой данных.

Внешний уровень представляет такие компоненты, которые очень часто изменяются. Обычно внешний уровень образуют пользовательский интерфейс, тесты, какие-то вспомогательные классы инфраструктуры приложения. К этому уровню также относятся конкретные реализации интерфейсов, объявленных на нижележащих уровнях. Например, реализация интерфейса репозитория, который объявлен на уровне Domain Services. Вообще все внутрении уровни, которые можно объединить в Application Core, определяют только интерфейсы, а конкретная реализация этих интерфейсов располагается на внешнем уровне.

Также стоит отметить, что все внешние хранилища, как базы данных, файлы, внешние веб-сервисы, от которых мы можем получать данные, - все это является внешним по отношению к архитектуре.

## Работа с Базами данных

При разработке приложений, работающих с базами данных (БД) возникает одна сложность, необходимость иметь единый для всего приложения механизм работы с данными. Самые распатроненные подходы:

Использование непосредственного кода ADO.Net в месте ребаоты с БД

Использование шаблона Active Record

Использование ORM, такой как Entity Framework 6 или NHibernate

Использование шаблона Repository

Некоторые из вас уже сталкивались с кодом ADO.Net, для работы с БД, он предоставляет низкоуровневый механизм, для работы с внешними СУБД. Существует множество клиентских библиотек, поддерживающих стандарт ADO.Net, и обеспечивающих интерфейс к различным базам данных.

Шаблон проектирования Active Record подразумевает внедрение логики работы с БД непосредственно в классы данных.

ORM (Object relationship management) – фреймворки, которые скрывают в себе всю рабту с БД (обычно низ лежащий ADO.Net) и предоставляющие наборы методов, для формирования запросов в БД.

### Шаблон «Repository»

Шаблон  **Repository**, позволяет отделить логику работы с данными, и описать ее общим шаблоном. Java разработчики используют другое название для этого шаблона – DAO (Data access objects). В системе описывается интерфейс хранилища данных, который явно не указывает, какое хранилище будет использоваться. Таким образом есть возможность менять механизм сохранения данных, не меняя остальной код приложения. Есть возможность реализации репозитория, который будет сохранять данные в

XML-файлах

Базе данных MS SQL Server

Базе данных Oracle

Не реляционной базе MongoDB

Доставть и сохранять данные со стороннего Web сервиса,

Возможны абсолютно любые комбинации. В классической литературе рекомендуют использовать этот шаблон в связке с Entity Framework, и упоминается возможность прозрачной замены сервиса хранения.

Преимущества, которые дает использование этого шаблона:

Отделение логики сохранения данных от логики обработки и отобрадения

Возможность изменить механизм сохранения данных, не меняя остальной код

Возможность определить какая часть приложения использует конкретные методы работы с данными (поиск по коду обращений к методам репозитория)

Если не использовать подобный шаблон, а использовать непосредственно SQL код при отображении/изменении данных, то очень сложно отследить, какая часть вашего приложения что делает с данными.

Этот шаблон предполагает выделение классов-сущностей, которое имеют в себе только поля, по возможности не имеют никаких методов, и описание для каждого такого класса набора методов для работы с данными.

    public interface IBookRepository

    {

        IEnumerable<Book> GetBookList();

        Book GetBook(int id);

        void Create(Book item);

        void Update(Book item);

        void Delete(int id);

    }

Для получения книги, в данном примере необходимо вызвать метод: repository.GetBook(5)

Для каждой отдельной сущности создается отдельная реализация репозитория, например для книг и пользователей.

### Шаюлон «Unit of Work»

В приложениях нередко используется паттерн репозиторий для инкапсулирования логики работы с источниками данных. И нередко мы оперируем множеством сущностей и моделей, для управления которыми создается также множество классов-репозиториев. Паттерн Unit of Work позволяет упростить работу с различными репозиториями и дает уверенность, что все репозитории будут использовать одно и то же подключение к БД. Кроме того что Юнит содержит в себе репозиории и подключение, так-же он ответственен за закрытие соединения.

public class UnitOfWork : IDisposable

{

    private OrderContext db = new OrderContext();

    private BookRepository bookRepository;

    private OrderRepository orderRepository;

    public BookRepository Books

    {

        get

        {

            if (bookRepository == null)

                bookRepository = new BookRepository(db);

            return bookRepository;

        }

    }

    public OrderRepository Orders

    {

        get

        {

            if (orderRepository == null)

                orderRepository = new OrderRepository(db);

            return orderRepository;

        }

    }

    public void Save()

    {

        db.SaveChanges();

    }

    private bool disposed = false;

    public virtual void Dispose(bool disposing)

    {

        if (!this.disposed)

        {

            if (disposing)

            {

                db.Dispose();

            }

            this.disposed = true;

        }

    }

    public void Dispose()

    {

        Dispose(true);

        GC.SuppressFinalize(this);

    }

}

Класс UnitOfWork предоставляет доступ к репозиториям через отдельные свойства и определяет общий контекст для обоих репозиториев.

Кроме того, данный класс содержит дополнительные методы Save() и Dispose(), которые в иной ситуации мы могли б определить в репозиториях. Но так как этот функционал будет общим для обоих репозиториев, то его лучше вынести в класс UnitOfWork.

# Пример

Рассмотрим пример, на тестовом проекте. (RepositoryPattern.sln)

## Базовая сущность

Выделяем общий класс, для всех сущностей, так как каждая сущность будет иметь коллонку с идентификатором

public class BaseEntity<TKey>

{

public TKey Id { get; set; }

}

## Базовый интерфейс репозитория

public interface IBaseRepository<TKey, TEntity> where TEntity : BaseEntity<TKey>

{

TKey Insert(TEntity entity);

bool Update(TEntity entity);

int Upsert(TEntity entity);

int GetCount();

TEntity GetById(TKey id);

bool Delete(TKey id);

IList<TEntity> GetAll();

}

Он содержит все общие методы работы с любой сущностью.

Метод Upsert производит обновление, или добавление новой записи, если такая не найдена.

## Репозитории конкретных сущностей

public interface IBlogPostRepository : IBaseRepository<int, BlogPost>

{

IList<BlogPost> GetByUserId(int userId);

int GetCountByUserId(int userId);

IList<BlogPostWithAuthor> GetAllWithUserNick();

}

public interface IBlogUserRepository : IBaseRepository<int, BlogUser>

{

BlogUser GetByLoginPassword(string login, string password);

}

К примеру для постов блога мы определяем метод поиска всех записей конкретного пользователя.

Особый интерес вызывает метод **GetAllWithUserNick**, который возвращает не нашу сущность, а объект-представление, которое содержит информацию с двух таблиц (BlogPost и BlogUser), он возвращает информацию, необходимую для отображения списка записей в блоге, вместе с именами их авторов. По мимо этого, данный запрос не извлекает из БД содержимое всех посов, а только их заголовки, что существенно снижает нагрузку на СУБД.

Чисто для удобства повторного использования свойств класс BlogPostWithAuthor унаследован от BlogPost, но такого требования нет, и в большинстве случаев представление и не должно наследоваться от хранимого объекта.

## IUnitOfWork

public interface IUnitOfWork : IDisposable

{

IBlogPostRepository BlogPostRepository { get; }

IBlogUserRepository BlogUserRepository { get; }

void Commit();

void RollBack();

}

Здесь определены все возможные репзитории в нашем приложении, также предусмотрена транзакционность. Вызов метода Commit записывает изменения в БД, а метод RollBack отменяет все изменения текущего UnitOfWork.

## SqlServer UnitOfWork

Релизация для работы с Sql Server

namespace WorkWithDB.DAL.SqlServer

{

public class UnitOfWork : IUnitOfWork

{

private readonly SqlTransaction \_transaction;

private readonly SqlConnection \_connection;

private IBlogUserRepository \_blogUserRepository;

private IBlogPostRepository \_blogPostRepository;

/// <summary>

///

/// </summary>

public UnitOfWork ()

{

var connectionString = ConfigurationManager.ConnectionStrings["Default"].ConnectionString;

\_connection = new SqlConnection(connectionString);

\_connection.Open();

\_transaction = \_connection.BeginTransaction(/\*IsolationLevel.ReadCommitted\*/);

}

public IBlogPostRepository BlogPostRepository

{

get

{

if (\_blogPostRepository == null)

\_blogPostRepository = new BlogPostRepository(\_connection, \_transaction);

return \_blogPostRepository;

}

}

public IBlogUserRepository BlogUserRepository

{

get

{

if (\_blogUserRepository == null)

\_blogUserRepository = new BlogUserRepository(\_connection, \_transaction);

return \_blogUserRepository;

}

}

public void Dispose()

{

try

{

if (\_transaction != null) \_transaction.Dispose();

}

finally

{

\_connection.Dispose();

}

}

public void Commit()

{

if (\_transaction != null) \_transaction.Commit();

}

public void RollBack()

{

if (\_transaction != null) \_transaction.Rollback();

}

}

}

При создании объекта происходит открытие соединяя и создание новой транзакции.

Примечание!!!

При использовании MS SQL Server для корректной работы транзакций необходимо включить поддержку Snapshot isolation для базы данных, иначе транзакции будут блокировать другие соединения.

ALTER DATABASE shipperdb SET allow\_snapshot\_isolation ON

ALTER DATABASE shipperdb SET read\_committed\_snapshot ON

## UnitOfWorkFactory

public static class UnitOfWorkFactory

{

private static Func<IUnitOfWork> \_factory;

public static IUnitOfWork CreateInstance()

{

if(\_factory == null)

throw new InvalidOperationException("Библиотека работы с данными не инициализирована");

return \_factory.Invoke();

}

public static void \_\_Initialize(Func<IUnitOfWork> factory)

{

\_factory = factory;

}

}

Позволяет заменить реализацию механизма хранения по всему приложению, и обспечивает абстракцию, если нам необходимо работать с БД – пишем следующий код:

using (IUnitOfWork scope = UnitOfWorkFactory.CreateInstance())

{

var id = scope.BlogUserRepository.Insert(new BlogUser() { Name = "Bogdan", Nick = "winnie2", UserPassword = "!QAZ2wsx#EDC4rfv" });

var allUsers = scope.BlogUserRepository.GetAll();

id = allUsers.Select(u => u.Id).FirstOrDefault();

var user = scope.BlogUserRepository.GetById(id);

scope.Commit();

}

Commit нужен только, если мы производили изменение данных.

## BaseRepository<TKey, TEntity>

Содержит в себе все общие методы по работе с БД, как выполнение запросов, работа с параметрами.

Методы:

ExecuteScalar – выполнение запросов, возвращающих единое значение

ExecuteNonQuery – выполнение операций, не возвращающих результата, как-то добавление, или удаление записей, так-же служебные команды СКЛ сервера

ExecuteSingleRowSelect – запрос, который возвращает только одну строку

ExecuteSelect – запрос, который возвращает список объектов

## BlogPostRepository

Релизация для записей блога

Примечательны методы:

public IList<BlogPostWithAuthor> GetAllWithUserNick()

{

return base.ExecuteSelect("Select bp.Id, bp.UserId, bp.Title, bp.Created, u.Nick " +

" from BlogPost bp " +

" JOIN BlogUser u on bp.UserId = u.Id" +

" order by bp.Created desc",

reader => new BlogPostWithAuthor

{

Id = (int)reader["Id"],

UserId = (int)reader["UserId"],

Title = (string)reader["Title"],

Created = (DateTimeOffset)reader["Created"],

AuthorNick = (string)reader["Nick"],

}

);

}

public IList<Entity.BlogPost> GetByUserId(int userId)

{

return base.ExecuteSelect(

"Select bp.Id, bp.UserId, bp.Content, bp.Created from BlogPost bp where bp.UserId = @userId",

new SqlParameters()

{

{"userId", userId}

}

);

}

Первый производит выборку для нестандартного типа данных, поэтому мы явно указываем, как преобразовывать строчку с базы в объект, а метод GetByUserId использует общий механизм для всех постов, и стандартную реализацию преобразования

protected override BlogPost DefaultRowMapping(SqlDataReader reader)

{

return new BlogPost

{

Id = (int)reader["Id"],

UserId = (int)reader["UserId"],

Content = (string)reader["Content"],

Created = (DateTimeOffset)reader["Created"],

Title = (string)reader["Title"],

};

}

## SqlParameters

Использование параметров необходимо для избегания возможности SQL-иньекции, это очень важно!

## Использование репозиториев

using (var unitOfWork = UnitOfWorkFactory.CreateInstance())

{

var post = \_post   
 ?? new BlogPost(){  
 UserId = StateHolder.CurrentUser.Id,

Created = DateTimeOffset.Now

};

post.Content = Content;

post.Title = Title;

unitOfWork.BlogPostRepository.Upsert(post);

unitOfWork.Commit();

}

Прежде всего, мы открываем соединение, и не забываем поместить его в блок using, для закрытия соединения.

Вся работа с данными производится внутри этого блока, в конце работы необходимо зафиксировать транзакцию, иначе она будет отменена, при закрытии соединения.

В случае возникновения любой исключительной ситуации (Exception) транзакция будет отменена, благодаря использованию using.

Приведенный выше код производит создание новой записи, или обновление существующей.

## Подключение к базе

Подключение к базе настраивается в файле App.config главного проекта:

<connectionStrings>

<add name="Default" providerName="System.Data.SqlClient" connectionString="Data Source=.\SQLEXPRESS;Integrated Security=True;Initial Catalog=SampleBlog;"/>

</connectionStrings>

Это подключение к базе SQL Express, так-жеможно использовать LocalDB, он интсалируется автоматически вместе со студией, и подходит для локальной разработки

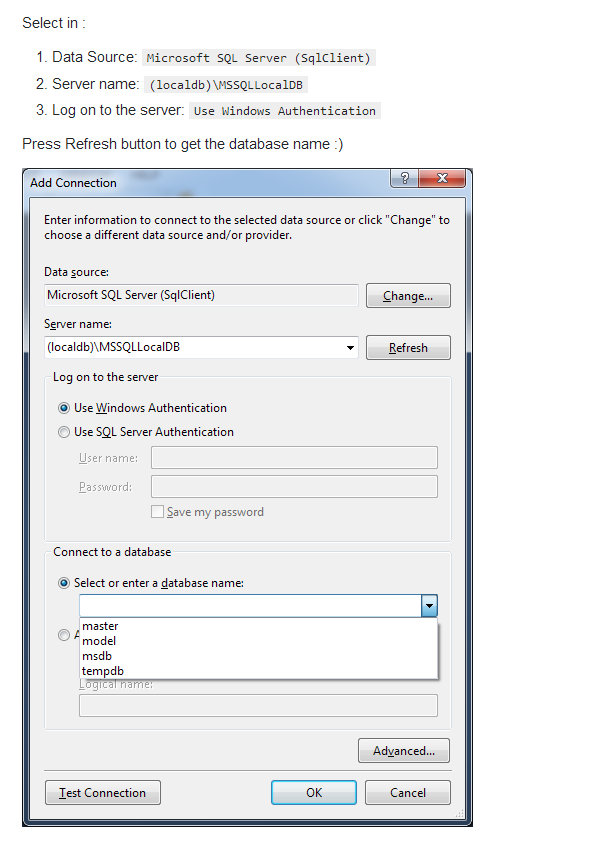
<http://intenseagile.com/2015/05/01/connect-to-localdb-from-visual-studio.html>

<http://stackoverflow.com/questions/21563940/how-to-connect-to-localdb-in-visual-studio-server-explorer>

<connectionStrings>

<add name="DefaultConnection" connectionString="Data Source=(LocalDb)\v11.0;Integrated Security=True;Database= SampleBlog; " providerName="System.Data.SqlClient" />

</connectionStrings>



## Создание базы

К проекту приложены скрипты создания таблиц, и начальных данных.

1. Скрипт, который создает базу данных с таблицами и данными (не подойдет для LocalDB):  
   SQL\SampleBlog 20151209 0300.sql
2. Два скрипта, которые в существующую базу добавляют таблицы и данные

SQL\ 1. Create Tables.sql – создает таблицы, и активирует транзакции

SQL\ 2. Fill Data.sql – заполняет текстовые данные