Procedure Façade

В прошлый раз, когда речь шла об инфраструктурном слое, я упомянул, что в текущем своем виде он не подходит для удобного использования процедур, если таковые имеются. Однако многие проекты до сих пор могут вынужденно использовать процедуры как наследие предыдущих версий. Впрочем, с некоторым удивлением можно обнаружить и новые проекты, которые активно используют процедуры вплоть до CRUD реализаций.

В этом посте я хочу рассказать, как можно сделать использование процедур как можно более простым для вашего кода и лишить себя многой тупой работы и кода. Приведенный ниже пример будет основываться на Linq2Sql контексте, но общая идея, я уверен, подойдет и для EF. Для тех, кто не читал цикл о фасаде доступа к данным, рекомендую сначала прочитать его, так все примеры ниже будут основаны на описанном примере и внимание будет уделено только работе с процедурами.

Идея

Общая идея совсем не нова и основывается на принципе Convention over Configuration. Т.е. все работает «как бы магическим образом», основываясь только на специальных правилах именования. Когда-то давно это не казалось хорошей идеей, однако такая договоренность значительно упрощает жизнь и сейчас данный принцип используется во многих проектах и фреймворках. Взять хотя бы тот же ASP.NET MVC, который полностью построен на специальных названиях, папках и тому подобное, никто ведь не жалуется, даже скорее наоборот. Конечно, работа проектов на соглашениях кажется «магией», пока не начинаешь работать с ними и глубоко разбираться в механизмах лежащих глубоко внутри. Впрочем, для правильного и эффективного использования в любом случае придется овладеть «магией», которая превратится в простую ловкость рук.

Если кратко, то любой ORM уже генерирует достаточно кода для вызова процедур и есть возможность давать произвольные имена процедурам, получается что легче всего определять имена процедур по методам вызова из слоя домена, из сервисов.

Я уверен, что процедуры должны дергаться только в домене, не давая им просачиваться в слой моделей/представлений. Так же я уверен в том, что не должно быть никаких действий между вызовом процедуры и получением ответа от нее. Т.е. если я говорю фасаду запустить процедуру GetAllCustomers(), то должна дернутся ровно одна процедура с таким именем, а не произойти чистка заказчиков, создание каких-нибудь специальных заказчиков, получение всех заказчиков, удаление терминированных и заказ пиццы. Таким образом получается прозрачное видение того, какие процедуры и как часто участвуют в работе.

Подготовка

Прежде чем перейти к рассказу о реализации, необходима небольшая подготовка проекта из поста о фасаде доступа к данным. Прежде всего потребуются сами тестовые процедуры, которые будут демонстрировать работы с разными типами процедур. У нас будут процедуры:

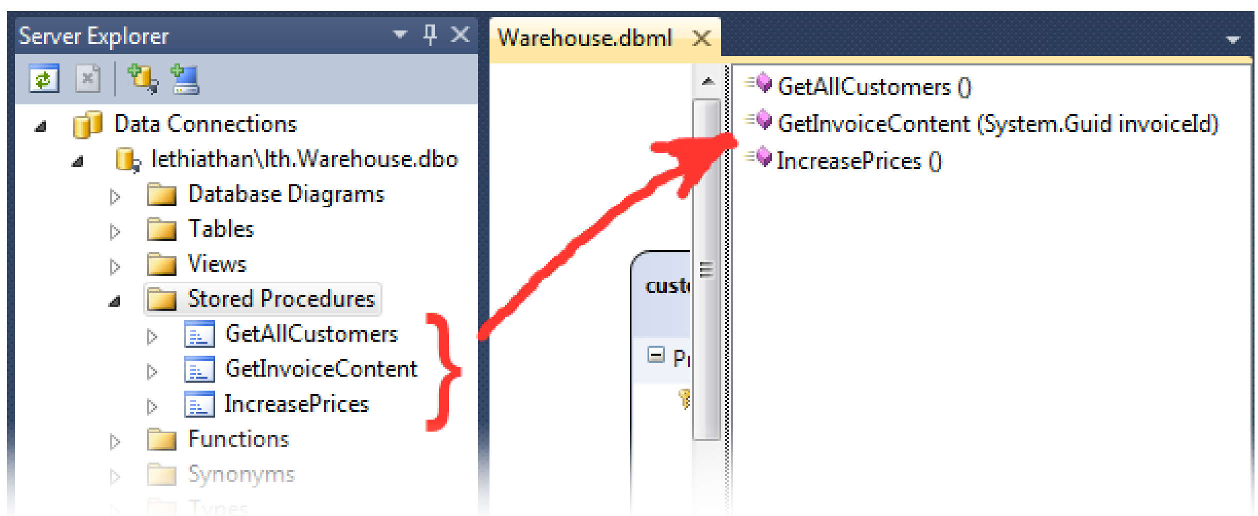
- Без параметров. Возвращают набор данных.

- С параметрами. Возвращают набор данных.
- Без параметров. Модифицируют данные, возможна долгая работа.

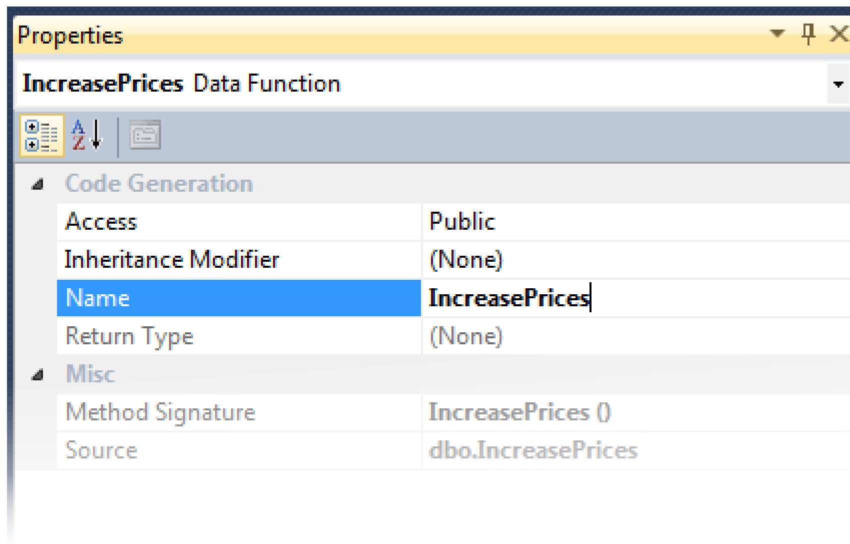
```
create procedure GetAllCustomers
as
select CustomerId
      ,Title
      ,Phone
      ,Website
  from Customer (nolock)
go

create procedure GetInvoiceContent(
      @InvoiceId uniqueidentifier
)
as
select
         w.WareId
        ,w.Title
        ,w.Price
  from invoceContent c (nolock)
  join ware w (nolock)
       on c.wareId = w.wareId
 where c.InvoiceId = @InvoiceId
 go

create procedure IncreasePrices
as
set nocount on
update ware set Price = Price + 1
go
```

Как вы можете видеть, процедуры совсем-совсем простые. Так же код процедур можно найти в файле Infrastructure > Scripts > [Procedures.sql]

После того, как вы создали процедуры в исходной базе данных, требуется добавить их в контекст базы приложения. Открываем файл .dbml и ServerExplorer, после чего просто перетаскиваем все три процедуры на панель процедур и функций. Должно получиться в итоге как на скриншоте.



В этот момент ORM сгенерировал методы в коде по именам процедур из базы. Если у вас процедуры называются страшными нечеловечьими именами, то вам всегда поможет F4 (Proprties) в которых вы сможете дать удобоваримое и понятное имя процедуре.



Следующим шагом будет подготовка шаблона сервиса, который будет использовать в своей работе эти процедуры. Для демонстрационных целей все процедуры будут вызываться в одном сервисе. Итак, требуется создать в сборке Domain класс для сервиса. Я назвал его, не мудрствуя лукаво, Service, так как других доменных сервисов у нас не будет.

Все готово для дальнейшего действия.

Раннее использование

Возможно лень, возможно время, общие знания и подходы диктовали достаточно прямолинейную и не оптимальную реализацию работы с процедурами из фасада. Помня о том, что вызывать напрямую процедуры из контекста моветон и а-та-та, напрашивалось только одно – сделать отдельный класс с перечислением всех используемых процедур, выделение интерфейса, работа с ним из инфраструктурного фасада. Т.е. в коде это выглядело бы примерно так:

Выделяем интерфейс в модуль домена:

```
namespace Domain {
public interface IProcedureFacade {
    List<Customer> RetrieveAllMyCustomers();
    List<Ware> ContentFor(Invoice invoice);
    void IncreasePriceOnOne();
}
}
```

Для большей красноречивости можно привести полную реализацию фасада процедур:

```
namespace Infrastructure {
public class ProcedureFacade : IProcedureFacade {
    private readonly IUnitOfWork unitOfWork;

    public ProcedureFacade() {
        unitOfWork = ObjectFactory.GetInstance<IInfrastructureFacade>().UnitOfWork;
    }
```

```
    public List<Customer> RetrieveAllMyCustomers() {
        var isLazy = new IsLazy<Customer>();
        var translator = new CustomerProcTranslator();
        using (var ctx = new WarehouseDataContext(ConnectionHelper.CurrentConnection)) {
            return ctx.GetAllCustomers()
                .Select(i => translator.Reconstitute(i, unitOfWork, isLazy))
                .ToList();
        }
    }

    public List<Ware> ContentFor(Invoice invoice) {
        var isLazy = new IsLazy<Ware>();
        var translator = new WareProcTranslator();
        using (var ctx = new WarehouseDataContext(ConnectionHelper.CurrentConnection)) {
            return ctx.GetInvoiceContent(invoice.InvoiceId)
                .Select(i => translator.Reconstitute(i, unitOfWork, isLazy))
                .ToList();
        }
    }

    public void IncreasePriceOnOne() {
        using (var ctx = new WarehouseDataContext(ConnectionHelper.CurrentConnection)) {
            ctx.IncreasePrices();
        }
    }
}
}
```

Я думаю можно заметить в целом один и тот же шаблон по реализации, который надо набить по большому счету руками. Вторично приходится указывать переменные для вызова процедуры, требуется делать обвязку в using, использование трансляторов. Так же сюда потом будет подмывать написать чуток логики по сортировке, например, или еще какую фильтрацию, которой здесь совершенно не место.

Так же будут соблазны создавать методы, которые объединят под одним именем несколько процедур, что не прибавит вам радости при модификации кода.

Если процедур много, то класс и файл начинает разрастаться так, что уже сложно его поддерживать и искать связанные логически процедуры. Может начаться процесс по логическому разделению ответственностей по файлам, использовать модификатор partial для класса ProcedureFacade и много еще каких извращений и головных болей может доставить такая реализация.

Так же можно заметить, что я специально назвал методы не так же как они называются в базе данных. Это сделано намеренно, для иллюстрации, что с такими именами все будет работать, но распутывание клубка вызовов будет страдать.

В вырожденном случае, реализация сервиса может выглядеть так

```
public class Service : IService {
    private IInfrastructureFacade facade;

    public Service() {
        facade = ObjectFactory.GetInstance<IInfrastructureFacade>();
    }

    public List<Customer> AllCustomers() {
```

```
        // some additional actions
        return facade.Procedures.RetrieveAllMyCustomers();
    }

    public List<Ware> GetInvoiceContent(Invoice invoice) {
        // some additional actions
        var invoiceContent = facade.Procedures.ContentFor(invoice);
        // some additional actions
        return invoiceContent;
    }

    public void RisePrice() {
        // some additional actions
        facade.Procedures.IncreasePriceOnOne();
        // some additional actions
    }
}
```

Можно понаблюдать за тем, как названия методов вносят еще один уровень неопределенности, и все вместе это напоминает испорченный телефон, с которым очень тяжело работать через некоторое время.

Можно стараться строго себя контролировать и давать везде одинаковые имена, но смена настроения, импульсивное нажатие на F2 и всё! Работа не нарушена, читабельность тоже, а реверс-инжиниринг для самого себя усложнен.

Итого:

- очень много повторяющегося кода
- сложность процесса вспоминания кода
- высокая трудоемкость создания нового кода
- высоки соблазны по неправильной организации логики

Но, как вы догадываетесь, есть средство избежать этих недостатков.

Новый подход и решение

Для начала хочу показать, как я хочу видеть код, при вызове процедур:

```
    public List<Customer> GetAllCustomers() {
        // some additional actions
        return facade.Execute<Customer>();
    }
```

Что значит найди и выполни процедуру GetAllCustomers, а результат выдай мне в виде списка объектов типа Customer. Если тип не передавать, то будет считаться что ответа от процедуры мы не ждем, параметры же остаются параметрами.

Конечно для того, чтобы это заработало придется написать немало кода, но(!) только один раз. Что называется: сделал и забыл.

Начнем с модификации интерфейса фасада. Добавим туда 2 перегруженных метода **Execute()**, а так же один метод для выполнения с увеличенным или бесконечным таймаутом.

```
public interface IInfrastructureFacade {
    IUnitOfWork UnitOfWork { get; }
    List<T> Get<T>(Specification<T> specification);
    void Commit(IUnitOfWork unitOfWork);

    List<T> Execute<T>(params object[] args);
    void Execute(params object[] args);
    void ExecuteLong(params object[] args);
}
```

А так же можно сократить потенциально бесконечный интерфейс для фасада процедур до следующего вида:

```
namespace Domain {
public interface IProcedureFacade {
    object Execute(string name, int timeOut, List<StackFrame> stack, params object[]
args);
}
}
```

Да-да, мы будем анализировать стек функций, для того, чтобы найти какую процедуру запускать. Думаю, что стоит начать с рассмотрения реализации IProcedureFacade.

Procedure Facade

Весь класс состоит из одного небольшого важного метода. Если выражаться человечьим языком, то суть его сводится к следующему:

- Попытаться найти метод по переданному имени в собственной реализации, и если нашли, то выполнить его и вернуть результат. Если же нет, то
- Попытаться найти метод по переданному имени в сгенерированном контексте, и если нашли, то выполнить его и вернуть результат. Если же нет, то
- Построить сообщение на основе стека вызовов для того, чтобы программист разобрался, что же он делает не так и где ошибка.

Согласитесь, что звучит крайне просто? Для реализации надо быть знакомым с рефлексией.

```
public class ProcedureFacade : IProcedureFacade {
    private readonly Type contextType;
    private readonly Type type;


    public ProcedureFacade() {
        var context = new WarehouseDataContext();
        contextType = context.GetType();
        type = GetType();
    }


    public object Execute(string name, int timeOut, List<StackFrame> stack, params
object[] args) {
        var methodInfo = type.GetMethod(name, BindingFlags.NonPublic |
BindingFlags.Instance | BindingFlags.Public);
        if (methodInfo != null) {
            return methodInfo.Invoke(this, args);
        }

        methodInfo = contextType.GetMethod(name);
        if (methodInfo != null) {
            var armDataContext = new
WarehouseDataContext(ConnectionHelper.CurrentConnection);
```

```
            armDataContext.CommandTimeout = timeOut;
            return methodInfo.Invoke(armDataContext, args);
        }

        var arg = String.Join(", ", args.Select(i => i.ToString()));
        var builder = new StringBuilder();
        foreach (var frame in stack) {
            builder.AppendLine(frame.ToString());
        }

        throw new InvalidOperationException("Method named " + name + " not found. Was
called with args " + arg + "\r\n" + builder);
    }
}
```

В конструкторе сохраняются заранее типы контекста и самого фасада, это несколько сократит по времени работу основного метода.

Я думаю, что по визуальным блокам уже можно сказать, какой блок какую функциональность несет. Сделаю, правда, несколько замечаний по самой реализации. В самой первой строке идет имя метода в недрах самого класса, и метод ищется среди всех возможных методов в плане видимости: публичные и приватные. Сделано для большего удобства и чтобы не ограничивать в реализации. Для контекста же ищутся только публичные методы, так как другими они быть просто не могут.

Еще важна последовательность, в которой происходит поиск метода. Я думаю, что все же могут быть ситуации, когда надо будет каким-либо образом поднастроить параметры или еще что-то перед вызовом процедуры с помощью контекста и тогда, именно эта последовательность нас выручит.

Так же можно заметить, что контекст для базы каждый раз создается заново. Это сделано из соображений многопоточности и возможностей одновременного общения к базе нескольких процедур. В противном случае были бы часты ошибки в духе: контекст все еще работает с результатом запроса, поэтому новая операция не может быть выполнена. Мне кажется это может быть решено с помощью опции MARS, но что-то на практике я так и не проверил, хотя и надо бы. Как проверю, соберу достаточные результаты, так обязательно об этом напишу.

Остаток метода формирует очень важную информацию, для понимания ошибок, если таковые произошли.

Infrastructure Façade

Не растекаясь сильно мыслью по древу, перейдем к рассмотрению реализации более высокоуровневой абстракции. Начнем с самого простого, вызова процедуры, которая ничего не возвращает в ответ. Т.е. препарируем метод **Execute()**

```
public void Execute(params object[] args) {
    var procName = GetCallingMethodName();
    var callingHistory = GetCallingHistory();
    try {
        Procedures.Execute(procName, 30, callingHistory, args);
    }
    catch (TimeoutException e) {
        throw new TimeoutException("Try to call operation ExecuteLong() for really time
consuming operations");
    }
}
```

Так, по логике метода сначала определяется имя процедуры для дальнейшего запуска. Потом берется вся история вызовов. Был соблазн как-то это объединить, но все же лучше так оставить. Далее идет собственно вызов метода Execute() из процедурного фасада, который мы уже рассмотрели. Передается имя процедуры, стандартный таймаут в 30 секунд, история вызовов на всякий случай и аргументы. Только что подумал, что историю вызовов можно передавать в виде делегата, так как не каждый раз она нужна. Тут бы и память, и время сэкономили. Можете считать это домашним заданием.

```
private string GetCallingMethodName() {
    var stackTrace = new StackTrace(2, false);
    var stackFrame = stackTrace.GetFrame(0);
    return stackFrame.GetMethod().Name;
}
```

Эмпирический опыт показал, что стоит пропустить 2 «кадра» стека и получишь исходную информацию о методе, который вызвал Execute(). О нюансах такой работы будет позже.

Как устроен метод **GetCallingHistory()** вы уже наверно догадались:

```
private List<StackFrame> GetCallingHistory() {
    var stackTrace = new StackTrace(0, false);
    var stackFrame = stackTrace.GetFrames();
    return stackFrame.ToList();
}
```

Тут вообще ничего интересного нет, так что можно переходить к рассмотрению следующих частей класса.

Метод **Execute()**, который возвращает данные будет значительно сложнее, и не сразу все сложилось как следует. Итак, сначала код, потом комментарии:

```
public List<T> Execute<T>(params object[] args) {
    var procName = GetCallingMethodName();
    var callingHistory = GetCallingHistory();

    object result = null;
    try {
        result = Procedures.Execute(procName, 30, callingHistory, args);
    }
    catch (TimeoutException e) {
        throw new TimeoutException("Try to call operation ExecuteLong() for really time
consuming operations");
    }

    var typedResult = result as List<T>;
    if (typedResult != null) {
        return typedResult;
    }

    var enums = result as IEnumerable<object>;

    var adapter = TryGetTranslator<T>();
    if (adapter.IsNotNull() && enums.IsNotNull()) {
        var translator = ((dynamic) adapter).Translator;

        var res = new List<T>();
        foreach (var item in enums) {
            res.Add(translator.Reconstitute((dynamic) item, null));
        }
        return res;
```

```
    }

    if (enums != null) {
        var objects = enums.ToList();

        return objects.ConvertAll(i => (T) i);
    }
    return new List<T>();
}
```

Начало метода повторяет предыдущий, с тем лишь отличием, что теперь у нас есть результат выполнения процедуры в виде самого общего объекта. Дальше мы пробуем способы по получению данных, в зависимости от того, насколько все удачно может для нас сложиться.

Конечно, самое простое и желанное для нас, что каким-то волшебным образом мы получим список данных нужного типа T, который передал пользователь. Если нам повезло, а такое может быть, то приведенный результат возвращается пользователю и сказки конец, все рады. Но, как показала практика , таких случаев весьма мало, и приходится подходить к получению желаемых данных более изобретательно.

Вторая наша надежда, что результат можно хоть как-то перечислить, т.е. у нас набор данных. Если наши надежды оправдываются, то пробуем получить транслятор для шаблонного типа **T** с помощью метода **TryGetTranslator()**.

```
private IReadAdapter<T> TryGetTranslator<T>() {
    IReadAdapter<T> instance = null;
    try {
        instance = ObjectFactory.Container
            .ForGenericType(typeof (IReadAdapter<>))
            .WithParameters(typeof (T))
            .GetInstanceAs<IReadAdapter<T>>();
    }
    catch (StructureMapException e) {}

    return instance;
}
```

Завсегдатаям блога такое уже не в новинку, единственно что, StructureMap не имеет перегрузки TryXXX для шаблонных методов, так что я просто обернул вызов в try()…catch() – легко! Хотя по логике в данном случае нам нужен транслятор, но найти можно только адаптеры, так что я подумал, что лучше назвать метод по желаемому действию. В конце концов он помечен как private, и мелкий, что может нивелировать фактическую ошибку. Поборники полного соответствия и буквы закона могут переименовать метод.

Далее идет работа с транслятором и так как они достаточно общие по сути, но семантика объявления не столь схожа, то придется во всю задействовать приведение объектов к типу **dynamic**, даже не знаю какой ценой в противном случае далось бы такое решение. Когда все транслировалось в переданный пользователем тип, то можно возвращать коллекцию объектов. Это чаще всего применяется при трансляции в доменные типы. Как скажем, когда мы попросим набор объектов типа Customer.

Если же адаптера подходящего не нашлось, но список все же есть, то пробуем тупо привести к нужному типу. Такое срабатывает для value-типов, или же когда запрашивает dynamic.

Когда ничего не помогает, то возвращаем пустой список. Хотя наверно желательнее будет возвратить null.

Последним можно рассмотреть **ExecuteLong()**, хотя уже так же будет понятно чем он отличается от приведенных уже методов – только временем таймаута.

```
public void ExecuteLong(params object[] args) {
    var procName = GetCallingMethodName();
    var callingHistory = GetCallingHistory();

    Procedures.Execute(procName, 0, callingHistory, args);
}
```

Технологически мы рассмотрели все модификации. Теперь можно посмотреть как будет выглядеть класс сервисов.

```
public class Service : IService {
    private IInfrastructureFacade facade;

    public Service() {
        facade = ObjectFactory.GetInstance<IInfrastructureFacade>();
    }

    public List<Customer> GetAllCustomers() {
        // some additional actions
        return facade.Execute<Customer>();
    }

    public List<Ware> GetInvoiceContent(Invoice invoice) {
        // some additional actions
        var invoiceContent = facade.Execute<Ware>(invoice);
        // some additional actions
        return invoiceContent;
    }

    public void IncreasePrice() {
        // some additional actions
        facade.Execute();
        // some additional actions

    }
}
```

Все имена полностью соответствуют именам процедур, красота и порядок.

Нюансы и советы по использованию

Как это часто бывает в реальной жизни, есть некоторые нюансы использования данного подхода. Начнем с самого простого: надо вызвать несколько процедур в одном сервисном методе, как быть?

Быть очень просто – надо создать новые методы с нужными именами. При этом, если вы создаете дополнительные процедуры с модификатором private, то велика вероятность что в процессе компиляции, JIT оптимизации ваш метод заинлайнят и работать ничего не будет. Для исправления ситуации рекомендую использовать атрибуты, говорящие о том, что инлайн делать не стоит.

```
public List<Customer> GetAllCustomers() {
    // some additional actions
```

```
    PrepareData();
    return facade.Execute<Customer>();
}

[MethodImpl(MethodImplOptions.NoInlining)]
private void PrepareData() {
    facade.Execute();
}
```

Это вообще основное, что стоит запомнить при использовании.

Ограничением для использования является на данный момент работа только с одним логическим контекстом. Однако я думаю, это можно исправить несложными действиями.

Если в качестве желаемого типа задать dynamic, то вернется сгенерированный контекстом тип данных. Т.е. если взять

```
public List<Customer> GetAllCustomers() {
    // some additional actions
    var res = facade.Execute<dynamic>();
    // some additional actions
}
```

То переменная **res** будет иметь фактический тип GetAllCustomersResult, и можно будет обращаться к любым полям класса, но о поддержке IntelliSense придется на время забыть.

Если процедура возвращает единственное значение value-типа, то для его получения так же стоит задать желаемый тип dynamic, а далее брать первое значение и обращаться к нужному полю сгенерированного класса.

Использование описанного подхода не работает для процедур, которые принимают параметры табличного типа. Это ограничение справедливо только для Linq2Sql, так как он не может сгенерировать вызов для такого типа процедур.

Итого

Описанная работа с процедурами успешно проходит эксплуатацию в реальном проекте уже месяц, и мне нравится как это работает. Единственный серьезный минус – невозможность легкой работы с процедурами с табличными параметрами. Но это уже вопрос к Linq2Sql.

Hard’n’Heavy!

Violet Tape