Фасад доступа к данным

На основе LINQ to SQL

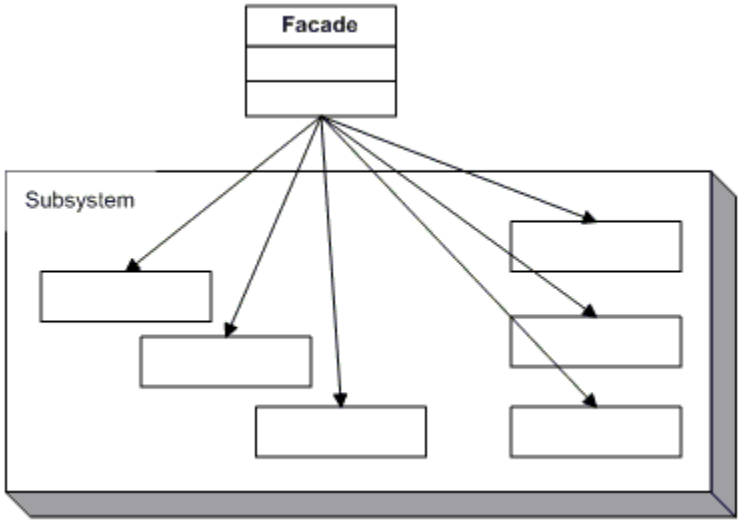
Сложность по шкале Microsoft 300-400

Сразу признаюсь, что я люблю Linq2Sql и не люблю Entity Framework. Люблю компактные API и не люблю развесистые классы с кучей методов служебных выставленных наружу. Может по этим причинам, а может быть по каким-то еще, но я использую схему доступа к данным, которую собираюсь описать ниже, уже несколько лет. По большей части меня она радует и выполняет свои задачи хорошо для большинства заданий возникающих на работе.

Вводная часть

Как уже ясно из названия, речь пойдет о паттерне Фасад. Вот что нам говорит википедия по поводу этого шаблона:

> **Шаблон Facade (Фасад)** — Шаблон проектирования, позволяющий скрыть сложность системы путем сведения всех возможных внешних вызовов к одному объекту, делегирующему их соответствующим объектам системы.



Однако сам по себе фасад достаточно тривиален и ограничиться только описанием применимости паттерна в случае баз данных было бы не очень интересно и информативно. Я планирую рассмотреть всю цепочку классов и данных, которые участвуют в получении данных, от записей в базы данных до доменных классов. Описать пути преобразования данных, оптимизацию запросов, сложные моменты в реализации, ограничения технологии, случаи, когда данный подход не оправдает себя.

Итогом работы будет являться использование фасада в таком духе:

```
var facade = new InfrastructureFacade();
var customers = facade.Get(Specification<Customer>.Null);
```

Т.е. надо будет только указать фасаду какой тип доменных данных вы хотите получить и с какими ограничениями. В данном случае ограничений на данные нет, т.е. в результате выполнения запроса получим всех заказчиков из базы.

Причем это будет практически единственный доступный способ получить данные из базы – простой для чтения и использования. Все служебные классы будут спрятаны от пользователей, в данном случае от других модулей.

До начала работы

Думаю, что надо перечислить все те компоненты и технологии, что будут участвовать в работе тестового приложения. Проект написан в Visual Studio 2010, .Net 4.0, так же желательно наличие NuGet, хотя это не обязательный пункт по идее все недостающие компоненты должны загрузиться в процессе первой сборки проекта. Скрипты написаны изначально для SQL Server 2008, но я уверен, что они заработают и на младших версиях SQL Server вплоть до SQL Server 2000. По поводу базовых компонентов кажется все.

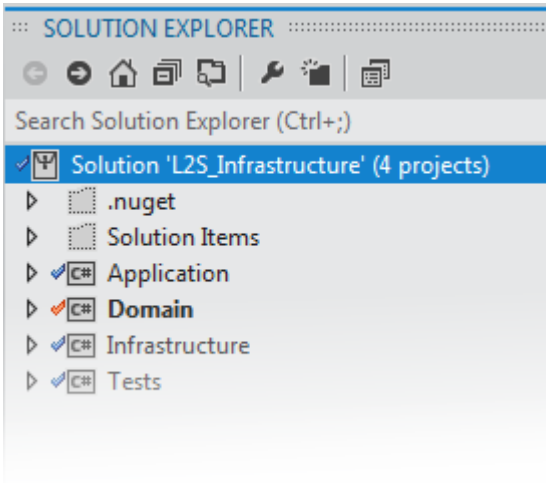
В проекте применяется автообнаружение классов с помощью StructureMap. На этом моменте я останавливаться не буду, так как это было подробно рассказано в предыдущих постах.

С идеологической точки зрения, программа построена по DDD (опять же см посты), так что есть доменная часть, для работы с пользовательскими сценариями используется паттерн UnitOfWork, который кратко будет рассмотрен ниже. Общение с базой данных идет только через Linq 2 Sql, при этом без использования хранимых процедур (о хранимых процедурах будет упомянуто отдельно), т.е. вся основная логика приложения должна находиться в C# коде.

Тестовое приложение можно забрать с Assembla из SVN репозитария.

Домен

Структура проекта



Пару слов про домен, который получился в тестовом проекте очень маленький и простой. В данном случае на него внимание не уделялось, и он принял утилитарный вид. Класс Customer полностью повторяет таблицу из базы данных. Класс Ware в некотором роде объединил InvoiceContent и Ware. Класс Invoice будет повторять таблицу. В целом никакой магии нет, кроме того, что Ware может создаваться через копирование с указанием количества заказанного товара.

```
public class Ware {
    public Guid WareId;
    public string Title;
    public decimal Price;
    public int Quantity;

    public Ware WithQuantity(int quantity) {
        return new Ware {
                            WareId = WareId,
                            Title = Title,
                            Price = Price,
                            Quantity = quantity
                        };
    }
}
```

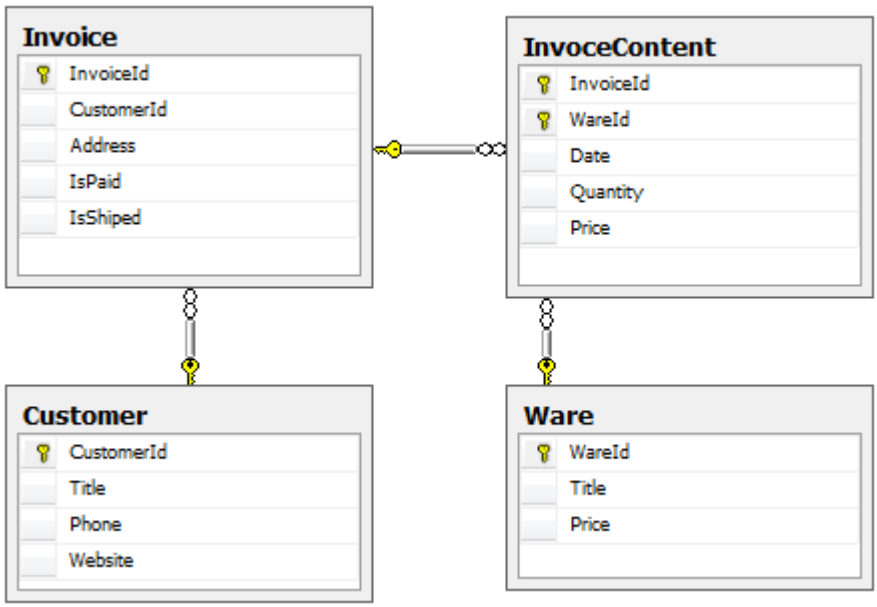
В дальнейшем рассмотрении так же будет участвовать спецификация на получение активных заказчиков.

```
public class ActiveCustomer : Specification<Customer> {
    public override bool IsSatisfiedBy(Customer obj) {
        return obj.Invoices.Any();
    }
}
```

Кажется, ничего не забыл, так что можно переходить к рассмотрению структуры базы данных.

База данных

База данных у нас будет состоять из Заказчиков, Заказов, Товаров и Содержимого заказа (Customer, Invoice, Ware, InvoiceContent – соответственно).



Общая диаграмма зависимостей таблиц представлена выше. Я думаю это уже минимальный классический набор сущностей для иллюстрации более-менее сложных концепций в программировании, на которых можно доступно и легко пояснить связи.

Остановимся лишь для чуть более подробного рассмотрения таблиц InvoiceContent и Ware. Свойствами товара является только название и цена в данный момент времени, по которому он будет продаваться или покупаться. Артикулы товара опущены, для нас хватит того, что все первичные ключи являются GUID полями. Состав заказа должен содержать ссылку на товар, но при этом содержать количество заказанных товаров и цену по которой совершена сделка, для исторического анализа.

Так как это тестовый пример, то здесь нет некоторых необходимых полей, как например, удалена ли запись или нет. Как правило, в больших системах не стоит удалять данные, их нужно помечать как удаленные. Это позволяет избежать многих проблем с фрагментированием базы, анализом данных, удаление связанных данных само по себе может быть большой проблемой. Но сейчас это не тема для разговора.

Скрипт для создания базы DatabaseScript.sql находится в проекте Infrastructure\Scripts. Останавливаться на нем подробно, я думаю, не имеет смысла, там все просто до безобразия.

После того как вы создали базу данных из скрипта, стоит обратить внимание на концепцию UnitOfWork.

Unit of Work

Теоретическая часть. Интерфейс.

Посмотрим, что нам скажет по поводу этого паттерна Мартин Фаулер, так как википедия молчит:

> **Unit of Work (единица работы)** – Обслуживает набор объектов, изменяемых в бизнес-транзакции (бизнес-действии) и управляет записью изменений и разрешением проблем конкуренции данных.

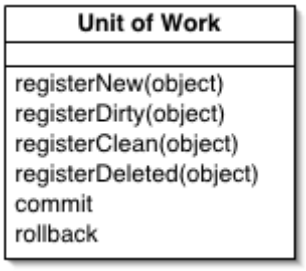
Когда необходимо писать и читать из БД, важно следить за тем, что вы изменили и если не изменили - не записывать данные в БД. Также необходимо вставлять данные о новых объектах и удалять данные о старых.

Можно записывать в БД каждое изменение объекта, но это приведёт к большому количеству мелких запросов к БД, что закончится замедлением работы приложения. Более того, это требует держать открытую транзакцию всё время работы приложения, что непрактично, если приложение обрабатывает несколько запросов одновременно. Ситуация ещё хуже, если необходимо следить за чтением из БД, чтобы избежать неконсистентного чтения.

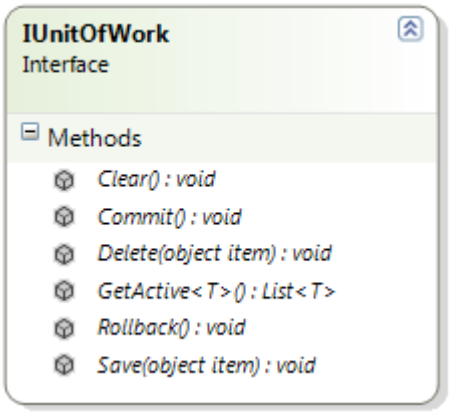
Реализация паттерна Unit of Work (здесь и далее UoW) следит за всеми действиями приложения, которые могут изменить БД в рамках одного бизнес-действия. Когда бизнес-действие завершается, UoW выявляет все изменения и вносит их в БД.

К слову, сам Linq2Sql внутри себя реализует UoW и работает с ним, чтобы отслеживать изменения пользователей.

Классическая реализация выглядят следующим образом:



В целом она не сильно отличается от того, что предложу я или MSDN.



Внимательный читатель наверно заметит, что в данной реализации есть примесь от репозитория, когда надо получить все живые (не удаленные) объекты заданного типа из UoW. На практике это весьма помогает, а о чистоте можно подискутировать отдельно, если у кого возникнет желание.

Интерфейс UoW находится в сборке Domain, тогда как реализация в Infrastructure – реализуем принцип инверсии зависимостей. К тому же нам в работе инфраструктурного фасада понадобятся некоторые специфичные методы, которые лучше всего объявить с модификатором Internal.

```
namespace Domain {
public interface IUnitOfWork {
    void Save(object item);
    void Delete(object item);
    void Commit();
    void Rollback();
    void Clear();
    List<T> GetActive<T>();
}
}
```

Немного об ответственности методов UoW.

- Save – сохраняет новые объекты, созданные пользователем, а так же отвечает за то, чтобы пометить существующие объекты как обновленные. Так же в этом методе можно вернуть к жизни удаленные объекты, об это стоит помнить при реализации.
- Delete – помечает объект на удаление. При этом, если объект новый, то его можно просто удалить из списков «живых» объектов, т.е. никаких операций с базой в дальнейшем произведено не будет.
- Commit – применение всех изменений к базе.

- Rollback – откат всех изменений в UoW, т.е. фактически очищаются списки новых, удаленных, обновленных элементов.
- Clear – полное очищение UoW, в целом достаточно редкая операция, может быть полезна при полной перезагрузки данных.

В интерфейсе не отражен, но при реализации нам понадобится метод **Register**, который заносит объекты в UoW, как изначальные, существующие в базе данных. В интерфейсе его нет, так как пользовательский код не должен иметь возможности заносить объекты в UoW с такой маркировкой, это противоречит логике.

Реализация Unit of Work

Реализация на самом деле может быть сделана многими способами. Даже сам интерфейс может быть дополнительно оптимизирован в зависимости от того, что у вас содержится в базе. Если у вас первичные ключи исключительно суррогатные и целочисленные, то можно ввести дополнительные параметры для регистрации, поиску и обновлению по этому ключу. Но помятуя размышления на тему суррогатных целочисленных ключей, лучше рассмотреть более общий случай.

Итак, в общем случае нам потребуется минимально три коллекции для хранения существующих в базе элементов, сохраненных/модифицированных и удаленных. На мой взгляд, логичнее всего это хранить в словарях, где ключом будет тип данных.

```
namespace Infrastructure {
public class UnitOfWork : IUnitOfWork {
    private readonly Dictionary<Type, List<object>> persisted
                                        = new Dictionary<Type, List<object>>();
    private readonly Dictionary<Type, List<object>> saved
                                        = new Dictionary<Type, List<object>>();
    private readonly Dictionary<Type, List<object>> deleted
                                        = new Dictionary<Type, List<object>>();

    . . .
}
}
```

Начнем наверно с базового метода, который будет использоваться в дальнейшем – с регистрации объектов в UoW, тех, что приходят из базы данных. Для этих целей удобно будет использовать два метода: для регистрации одиночных объектов и для регистрации массивов, что ускорит использование UoW.

```
public void Register(object item) {
    var type = CheckTypeExistence(item, persisted);

    persisted[type].Add(item);
}

public void RegisterAll<T>(List<T> item) {
    if(!item.Any()) return;

    var type = CheckTypeExistence(item.First(), persisted);

    persisted[type].AddRange(item.Cast<object>());
}
```

Тут есть нюанс, что надо четко понимать какой метод будет вызван в каждой ситуации, так как IList<T>, Enumerabler<T> и прочее это все же объект, а не коллекция List<T>.

В целом код достаточно примитивный, кроме того, что используется метод **CheckTypeExistence()**, который инициализирует коллекции для пришедшего типа, если он не зарегистрирован.

```
private Type CheckTypeExistence(object item, Dictionary<Type, List<object>> list) {
    var type = item.GetType();

    if (!list.ContainsKey(type)) {
        saved[type] = new List<object>();
        deleted[type] = new List<object>();
        persisted[type] = new List<object>();
    }
    return type;
}
```

Метод в реализации тоже прост, как лом в разрезе и в дополнительных пояснениях не нуждается. Главное чтобы у вас складывалась цельная картина происходящего, что ничего экстраординарного в этом методе не происходит не только на мой, но и на ваш взгляд. Однако метод сам по себе важен для последующей работы: лучше заранее инициализировать все нужные коллекции в словарях, чтобы потом не терять времени. Хотя данный метод будет вызываться при любых изменениях данных, но лучше перебдеть, чем недобдеть и потом искать обходные пути с хаками.

Как вариант можно предоставить метод **RegistryType()** для того, чтобы разработчики заранее инициализировали все типы данных. Но это может вылиться в то, что будет создан класс, где будут регистрироваться все доменные классы вне зависимости от необходимости. Плюс ко всему, и что самое важное (!!!) это добавляет энтропии в класс и заставляет задумываться разработчика (используйте «принцип наименьшего удивления» и «бритву Оккама»):

- надо ли заранее регистрировать тип?
- обязательное ли это действие?
- есть автоматическое регистрирование?
- в чем профиты использования?

Все вместе это замедлит разработку и создаст ненужные трудности и источники ошибок. Использование данного подхода может быть оправдано только для экстремальной оптимизации кода, что возможно потребует пересмотра все структуры программы и данных в базе.

Сохранение и удаление элементов в фасаде:

```
public void Save(object item) {
    var type = CheckTypeExistence(item, saved);

    if (!saved[type].Contains(item))
        saved[type].Add(item);

    deleted[type].Remove(item);
}

public void Delete(object item) {
    var type = CheckTypeExistence(item, deleted);

    saved[type].Remove(item);

    if (!deleted[type].Contains(item))
        deleted[type].Add(item);
}
```

Как я уже упоминал при описании методов в рассмотрении интерфейса, при сохранении надо поместить элемент в сопутствующую библиотеку и «воскресить» элемент, если тот был удален. С удалением тоже фокусов нет.

В дальнейшем для работы нам еще понадобятся методы, которые не вошли в интерфейс: получение списка «живых» элементов, а так же методы для совершения общих действий над коллекциями.

```
public List<T> GetActive<T>() {
    var type = typeof (T);

    if (!persisted.ContainsKey(type)) {
        return new List<T>();
    }

    return persisted[type]
        .Union(saved[type])
        .Except(deleted[type])
        .Cast<T>().Distinct().ToList();
}
```

В данном случае я решил не создавать коллекции с запрошенным типом, если он не зарегистрирован, а просто вернуть пустой список. Дальнейший код трудностей вызывать не должен.

Для коллекций, нам понадобятся методы:

- ForEachInserted<T>(Action<T> action)
- ForEachUpdated<T>(Action<T> action)
- ForEachDeleted<T>(Action<T> action)

Общая реализация выглядит примерно следующим образом для каждого метода:

```
public void ForEachInserted<T>(Action<T> action) {
    var type = typeof (T);
    var list = saved[type].Except(persisted[type]).Cast<T>().ToList();
    list.ForEach(action);
}
```

Можно проводить многие оптимизации и создавать специфические вещи в зависимости от ваших задач и типов данных, здесь приведена наверно одна из простейших реализаций на базе .Net 4.

О применении

Пару слов о применении данного шаблона разработки. Первое что приходит в голову, и в связи с чем его используют, это следующая ситуация: представьте что у вас есть приложение с несколькими экранами, каждый делает определенную работу, на каждый есть свой UseCase – определенный способ использования. Так вот перед началом работы с каждым сценарием работы вы создаете или запрашиваете от фасада UoW с необходимыми данными, и все изменения до финального сохранения записываются в UoW. При такой модели меньше действий происходит с БД и меньше вероятности запороть базу. Если в процессе работы вышла ошибка, то обрывочные или не целостные данные в базу не уйдут (если конечно ошибка была не в процессе записи в базу).

UoW может использоваться для передачи данных между сценариями работы (экранами, UseCases), что может облегчить общее взаимодействие систем приложения. Более подробно можно почитать в книге «Шаблоны корпоративных приложений» Мартина Фаулера.

Инфраструктурный фасад, который мы начнем рассматривать чуть позже, в данном конкретном случае опирается на UoW, для минимизации обращений к базе и отслеживании состояния объектов.

Фасад

Теория

Кажется, все приготовления прошли, необходимые концепции упомянул и теперь можно перейти непосредственно к реализации фасада. Интерфейс фасада состоит всего из двух методов, которых нам хватит для всех целей приложения.

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

Как вы можете заметить, интерфейс фасада содержится в модуле домена, тогда как реализация будет в модуле инфраструктуры – применяем инверсию зависимостей, так как домен не должен зависеть от других частей системы.

Про Unit of Work (UoW) рассказано было ранее, паттерн Specification так же уже рассматривался, так что на этих типах данных останавливаться не будем.

Общий принцип работы такой:

- Получаем фасад;
- Вызываем метод получения доменных данных, в который передаем ограничивающие условия, которым должны соответствовать элементы;
- Что-то делаем с данными, работаем с UoW;
- Вызываем метод сохранения UoW.

Всё что делается под капотом фасада, в каком порядке сохраняются данные, какие, куда и как – всё это уже ответственность других элементов, которые мы так же рассмотрим подробно чуть позже. На данный момент ограничимся только их кратким рассмотрением, под капотом работают:

- **Адаптеры чтения и записи** – знают, какой транслятор вызвать, в каком порядке сохранить данные, работают с UoW, некоторые высокоуровневые вещи по работе с доменными данными и автогенеренными классами L2S, как обработать спецификации, отфильтровать ненужные элементы.
- **Трансляторы данных** – знают, как правильно создать доменные классы в соответствии с пришедшей спецификацией, а так же как правильно отобразить доменные данные на автогенеренные классы L2S.
- **Оптимизаторы запросов** – знают, как достроить запрос к базе данных на основе пришедших спецификаций.

Фактически, фасад управляет обращением к адаптерам чтения и записи, скрывает работу по обращению к ним, так как они все унифицированы и имеют как минимум метод Get() такой же сигнатуры, как и сам фасад.

Реализация

Подходим плавно к реализации фасада, которая расположена в модуле Infrastructure. Сейчас окидываю взглядом реализацию фасада и вижу, что он очень простой. Собственно таким он и должен быть, ведь этот класс самостоятельной работы выполнять не должен, а только умно делегировать работу.

Реальный фасад фактически не реализует никаких дополнительных методов, которые могут быть использованы при работе с реальным классом, а не при работе через интерфейс. Итак, реальный фасад (читай класс InfrastructureFacade) оснащается двумя конструкторами:

```
public InfrastructureFacade() {
    unitOfWork = new UnitOfWork();
}

public InfrastructureFacade(IUnitOfWork unitOfWork) {
    this.unitOfWork = (UnitOfWork) unitOfWork;
}
```

Привнесем некоторой гибкости при работе, когда можно не перегружать весь UoW. Далее реализация свойства UnitOfWork, которое не представляет интереса. Зато остановимся подробнее на методах Get() и Commit()

Метод **Get()** позволяет получить данные из базы данных, причем отфильтровать их с соответствии с заданной спецификацией.

```
public List<T> Get<T>(Specification<T> specification) {
    return GetAdapter<T>()
        .Get(unitOfWork, specification)
        .FindAll(specification);
}

internal IReadAdapter<T> GetAdapter<T>() {
    var instance = ObjectFactory.Container
        .ForGenericType(typeof (IReadAdapter<>))
        .WithParameters(typeof (T))
        .GetInstanceAs<IReadAdapter<T>>();

    if (instance.IsNull()) {
        var type = typeof (T);
        throw new ArgumentOutOfRangeException("", type
                    ,string.Format("Type {0} not registered/founded"
                    ,type.Name));
    }

    return instance;
}
```

Не будем пока рассматривать интерфейсы адаптеров (IReadAdapter), примите на веру, что там есть метод Get с такой же сигнатурой. Работу метода можно описать достаточно просто:

- найди адаптер по типу данных шаблона,
- получи данные,
- отфильтруй список.

Последняя фильтрация нужна из-за того, что не все бизнес-условия можно перевести в SQL, возможно некоторые проверки будет удобнее провести на клиенте.

Код в методе получения адаптера должен быть уже знаком по статье рассматривающей авторегистрацию классов, тут никаких нюансов нет, по сравнению с упомянутой статьей. Люди знакомые с StructureMap, так же не должны тут увидеть подвохов, все достаточно стандартно. Основная сложность скорее в регистрации адаптеров, но она была уже рассмотрена ;)

Посмотрим, как устроен метод **Commit()**.

```
public void Commit(IUnitOfWork unitOfWork) {
    var adapters = ObjectFactory.Container.GetAllInstances<IWriteAdapter>().ToList();
    adapters.Sort((x, y) => x.Sequence.CompareTo(y.Sequence));
    adapters.ForEach(i => i.Save((UnitOfWork) unitOfWork));
}
```

Если при работе с Get() нам нужен только один адаптер, то при сохранении, надо получить полный набор адаптеров записи, при этом необходимо их отсортировать в порядке сохранения данных в базу. Особенность некоторых иерархичных структур такова, что L2S или же трансляторы не в состоянии построить полный граф элементов, если начинать строить его с произвольного элемента. Т.е. построение такого графа добавить чрезмерной сложности и ветвистости коду, а сложность и ветвистость источник трудноуловимых ошибок. Гораздо легче и быстрее сохранять некоторые элементы раньше других, благодаря чему проблему удается решить малой кровью и ошибка в зависимостях ключей может произойти, только если данных вообще нет, либо мы выставили неправильный порядок сохранения данных.

О порядке сохранения данных

Небольшой пример на пальцах. Допустим, у нас есть тип данных адрес. Для нормализации таблиц и поддерживая хорошую структуру данных, мы можем создать классы/таблицы: Страна, Область, Город, Адрес. Зависимость их пояснений не требует. Так вот при создании совершенно нового адреса, система может запнуться, так как никоим образом не гарантируется порядок получения адаптеров сохранения и Адрес или Город может сохраняться в базу данных до того, как была сохранена Страна. Или мы можем получить дубликаты Стран, Областей.

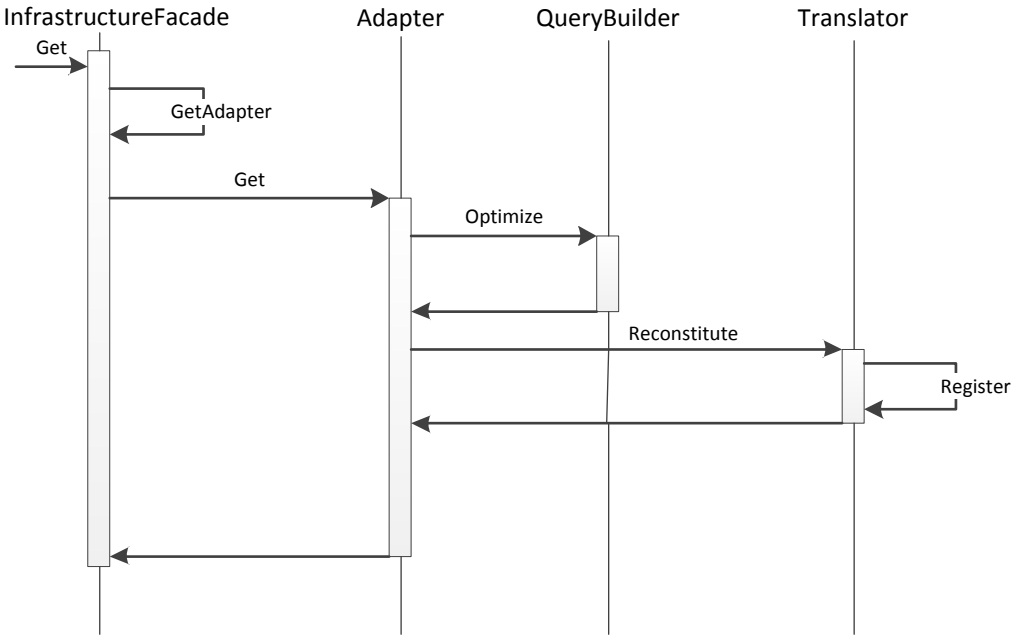
Можно представить, что всегда надо восстанавливать полный путь до корня, до Страны и это возможно сработает, когда ключами является GUID генерируемый клиентом, с identity такой фокус не пройдет! Так как до сохранения Страны мы не знаем какой номер она получит в результате сохранения.

Развивая мысль, можно допустить, что у нас есть второй адрес по этой же стране созданный вместе с первым, и он не сохранится из-за нарушения уникальности названия страны. Или вы бы не стали делать такую проверку в базе?

Это я к тому, что большинство систем строятся на использовании суррогатных ключей с identity, хотя в случае со страной само имя является натуральным уникальным ключом. Однако я надеюсь, что идея ясна.

Но вернемся к методу Commit(), после того, как все Адаптеры записи упорядочены, для каждого из них вызывается метод Save(), аргументом которого является UoW. Каждый адаптер работает только со своим типом данных, по которому он получает пользовательские значения из UoW.

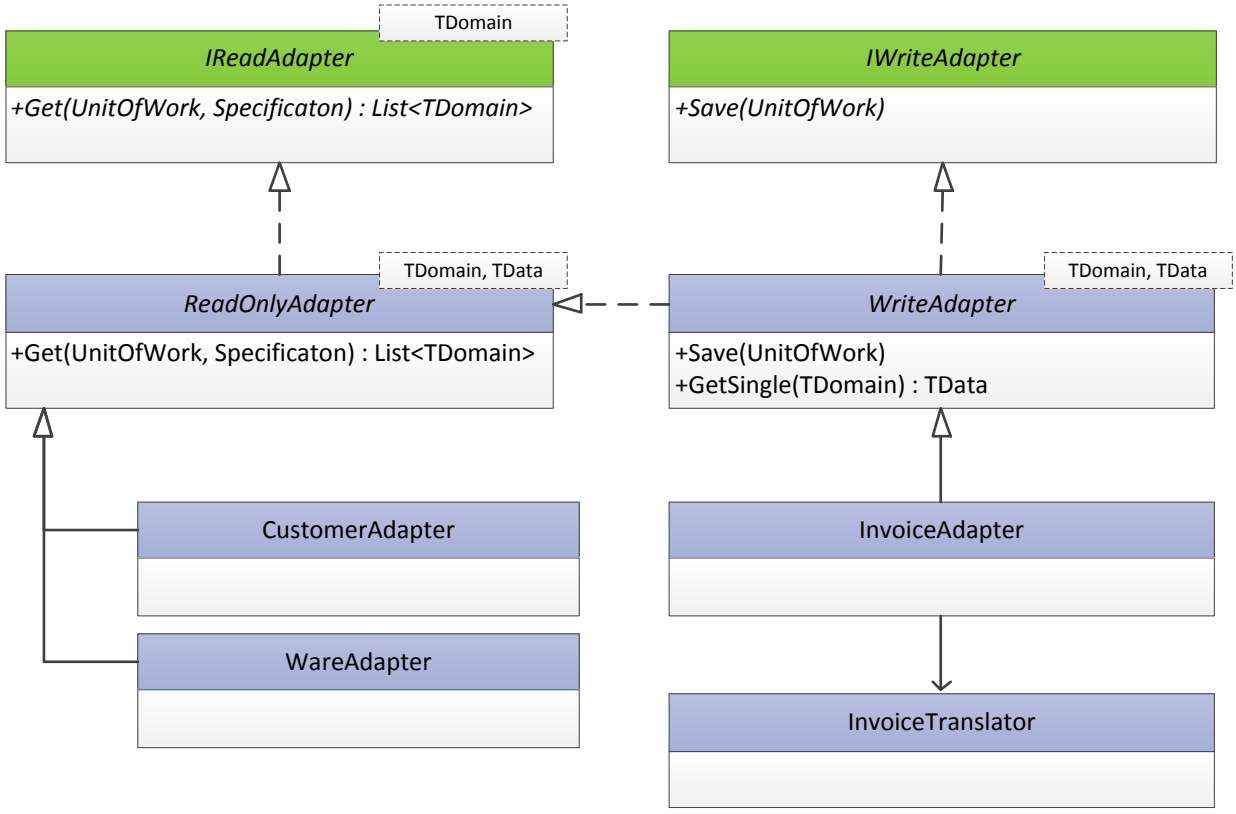
Хм. Вот и весь фасад! Но самое интересное, конечно же, осталось за кадром пока что, в реализации адаптеров, трансляторов и оптимизаторов запросов. Однако уже сейчас полезно будет нарисовать общую диаграмму взаимодействия всех этих элементов при получении данных.



Обо всем этом подробно будет написано ниже.

Адаптеры чтения

Для любителей и знающих UML предоставляю схему ниже, чтобы сразу оценить взаимодействие и структуру классов и наследования. TDomain – доменный класс, TData – сгенерированный Linq2Sql класс на основании таблицы данных.



Да, сейчас мы будем говорить только о том, как читать данные из базы данных.

IReadAdapter

Начнем с самого верха, с интерфейса IReadAdapter. Как видно из схемы, он состоит только из объявления сигнатуры одного, уже знакомого нам метода. От данного интерфейса кроме абстрактного класса ReadOnlyAdapter никто не наследуется, но данный интерфейс, само его наличие, играет важную роль в деле регистрации и получения конкретных адаптеров (CustomerAdapter, WareAdapter) в фасаде. Если вы вернетесь в фасад и посмотрите на код для получения адаптера, то вы увидите, что используется только один дженерик параметр. И это единственный параметр доступный клиентской части. Клиент (домен) ничего не знает о базе данных и о том, какие классы сгенерировал L2S, так что это является механизмом с помощью которого мы в принципе можем получить нужный адаптер.

```
public interface IReadAdapter<TDomain> {
    List<TDomain> Get(UnitOfWork unitOfWork, Specification<TDomain> specification);
}
```

Только что был хороший пример того, как маленький кусок кода делает большую часть работы. Дальше будет большой важный класс, который фактически связывает трансляторы, построители запросов и делает еще много общей работы, так что конкретные адаптеры будут получаться маленькими и легкими.

ReadOnlyAdapter

Данный класс является одним из самых важных во всей схеме, поэтому уделим ему максимум внимания. Как следует из схемы, ReadOnlyAdapter реализует интерфейс IReadAdapter, поэтому ключевым методом можно считать метод **Get()**.

```
public virtual List<TDomain> Get(UnitOfWork unitOfWork, Specification<TDomain>
specification) {
    var isLazy = specification.ExtractSupersetSpecification<IsLazy<TDomain>>()
```

```
                                                                ?? new IsLazy<TDomain>();
    var list = (from dataObject in PrepareQuery(specification)
                select dataObject).ToList();

    var result = new List<TDomain>();
    list.ForEach(i => result.Add(translator.Reconstitute(i, unitOfWork, isLazy)));

    return result;
}
```

Сразу все описать не получится, так как у нас редкий случай рассмотрения программы от общего к частному. Когда ныряем все глубже и глубже в детали когда, тогда как все предыдущие статьи шли от частных применений к общему, к выделению подхода. С одной стороны сложно рассказывать с отсылками к будущему тексту, но это лучше наверно, чем рассказывать о частностях, которые будут долгое время оторваны от общего контекста понимания. Ладно, закончим с лирикой, перейдем снова к практическим вещам.

В первых строках метода мы пробуем получить указание на ленивую инициализацию доменных типов. Спецификации на ленивую инициализацию могут сократить обращения к базе данных, потому что мы либо не догружаем данные, которые не требуется, либо указываем L2S какие данные нам точно потребуются, чтобы движок составил сложный запрос с join'ами.

Далее идет разделение: получение данных с помощью оптимизированного запроса и трансформация данных в доменные записи.

Оптимизация запроса осуществляется с помощью метода **PrepareQuery()** аргументом которого является спецификация, в результате разбора которой можно построить запрос сужающий результирующий набор данных. В недрах метода идет работа с Реализация оптимизации запросов, о которых подробнее будет рассказано далее.

В последних строках, с помощью метода **Reconstitute()** происходит маппинг объектов базы на объекты домена. Самая первая реализация осуществляла маппинг объектов сразу, по мере чтения объектов из базы, однако при таком подходе невозможно будет распараллелить восстановление объектов, в случае их слабой связанности, а большинство случаев именно такие. Хотя в данном коде и не используется многопоточный маппинг, но его возможность становится более явной на мой взгляд.

Я думаю, вы уже заметили, что метод Reconstitute() относится к классу Базовый класс, который передается в класс в момент создания.

```
private readonly ReadOnlyTranslator<TDomain, TData> translator;

protected ReadOnlyAdapter(ReadOnlyTranslator<TDomain, TData> translator) {
    this.translator = translator;
}
```

До сих пор нигде не фигурировало соединение с базой, контекст L2S, рабочие таблицы. Все это появляется в методах связанных с PrepareQuery().

```
private IQueryable<TData> PrepareQuery(Specification<TDomain> specification) {
    var options = new DataLoadOptions();
    DoConfigureLoad(options);

    var context = new WarehouseDataContext(ConnectionHelper.CurrentConnection);
    context.LoadOptions = options;
```

```
    var preparedQuery = GetQueryBuilder().Optimize(GetBaseQuery(context), specification);
    return preparedQuery;
}

protected virtual void DoConfigureLoad(DataLoadOptions options) {}
```

Так мало строчек кода и так много всего происходит, но я попробую подробно и связно рассказать, что к чему.

В первых двух строчках создается служебная структура L2S, с помощью которой можно указать фреймворку какие связанные данные надо дополнительно загружать с основным контентом TData сразу, а не посредством дополнительных запросов. При этом по умолчанию никакой оптимизации не делается, все отдается на откуп конкретным классам, которые могут переопределить виртуальный метод **DoConfigureLoad()**. Это может порой привести к значительному росту скорости восстановления объектов. Подробные примеры использования будут рассмотрены в конкретных реализациях адаптера.

После того, как были созданы оптимизационные настройки, переходим к созданию контекста Linq2Sql. В нашем примере контекст называется WarehouseDataContext и после его создания указываются опции для загрузки данных. Никаких особенностей в этих строках нет.

Далее идет подготовка оптимизированного запроса по переданным спецификациям. Основной порядок действий, на мой взгляд, прекрасно читается в самом коде: Взять необходимый оптимизатор запросов, применить его к базовому запросу на основе спецификаций.

```
private IQueryBuilder<TDomain, TData> GetQueryBuilder() {
    var queryBuilders = ObjectFactory.Container
        .ForGenericType(typeof (IQueryBuilder<,>))
        .WithParameters(typeof (TDomain), typeof (TData))
        .GetInstanceAs<IQueryBuilder<TDomain, TData>>();

    return queryBuilders;
}

protected virtual IQueryable<TData> GetBaseQuery(DataContext dataContext) {
    var attemptsLeft = 5;
    while (attemptsLeft > 0) {
        try {
            return dataContext.GetTable<TData>();
        }
        catch (Exception) {
            attemptsLeft--;
        }
    }

    throw new InvalidOperationException(string.Format("Can't get table from Context for
{0}", typeof (TDomain).Name));
}
```

Нужный оптимизатор запросов получаем на основе шаблонных типов TDomain и TData с помощью StructureMap. Код уже должен быть знаком, так как похожие конструкции получения дженерик типов использовались неоднократно. Как вы можете догадаться, обнаружение и регистрация оптимизаторов запросов происходит в автоматическом режиме методом Scan() у StructureMap.

Получение базового запроса это простое получение таблицы из контекста L2S. В данном случае применена конструкция для получения таблицы в несколько попыток, и только по истечении отпущенных попыток показывается сообщение с ошибкой. В небольших проектах для локальных

сетей такой подход может быть излишним, но мне пригодилось это при работе с удаленной базой.

Конкретные адаптеры

Теперь можно обратить взор на то, как выглядят конкретные адаптеры для доменных типов. Начнем рассмотрение с CustomerAdapter.

```
public class CustomerAdapter : ReadOnlyAdapter<Customer, customer> {
    public CustomerAdapter() : base(new CustomerTranslator()) {}

    protected override void DoConfigureLoad(DataLoadOptions options) {
        options.LoadWith<customer>(c => c.invoices);
    }
}
```

Так как класс небольшой, то я привел его полный текст. Данный класс наследуется от абстрактного класса ReadOnlyAdapter, при этом указываются классы доменный (Customer) и сгенерированный (customer). Адаптер работает только с одним транслятором, поэтому конструктор конкретного класса можно оставить пустым, а транслятор передать конструктору базового класса.

Значение DataLoadOptions

Пусть по условиям задачи получается так, что в подавляющем большинстве случаев нам интересны заказчики с заказами, т.е. при загрузке заказчиков мы хотим сразу получать списки заказов. Для того, чтобы получать их одним запросом, мы даем подсказки L2S с помощью заполнения класса DataLoadOptions в методе DoConfigureLoad().

Представим, что я не перегружаю метод DoConfigureLoad() в конкретном адаптере. Забегая немного вперед представим, что в трансляторе некоторым образом маппятся (восстанавливаются) объекты не только Customer, но и Invoice. Данную работу можно представить в виде небольшого теста:

```
[TestFixture]
public class CustomerAdapterTests : TestsBase{
    [Test]
    public void CanReconstitute() {
        // Arrange
        var customerAdapter = new CustomerAdapter();

        // Act
        var customers = customerAdapter.Get(new UnitOfWork()
                                            ,new NullSpecification<Customer>());

        // Assert
        customers.Count.Should().Be(5);
    }
}
```

В данном случае нам важен не столько результат теста, сколько то, какие запросы будут отправлены к базе данных. Для этого воспользуемся профайлером со стандартными настройками, этого должно вполне хватить.

Запускаем тест и смотрим на результаты.

1. Выполнился запрос для получения всех элементов из базы данных
   ```
   SELECT [t0].[CustomerId], [t0].[Title], [t0].[Phone], [t0].[Website]
   FROM [dbo].[Customer] AS [t0]
   ```

2. Получаются и обрабатываются заказы для каждого заказчика.
   ```
   exec sp_executesql N'SELECT [t0].[InvoiceId], [t0].[CustomerId],
   [t0].[Address], [t0].[IsPaid], [t0].[IsShiped]
   FROM [dbo].[Invoice] AS [t0]
   WHERE [t0].[CustomerId] = @p0',N'@p0 uniqueidentifier',@p0='9E7E67FE-
   B11F-4029-A285-278A0B5F9C8A'
   ```
3. Повторять п.2 пока не закончатся заказчики.

Получается очень большой трейс, специально сделал картинку побольше, чтобы оценили масштаб. Хотя это только 5 заказчиков, а что будет, если у нас их будет сотни и тысячи?

EventClass	TextData
Audit Login	-- network protocol: LPC set quoted_identifier on set ...
Audit Logout	
RPC:Completed	exec sp_reset_connection
Audit Login	-- network protocol: LPC set quoted_identifier on set ...
SQL:BatchS...	SELECT [t0].[CustomerId], [t0].[Title], [t0].[Phone], [t...
SQL:BatchC...	SELECT [t0].[CustomerId], [t0].[Title], [t0].[Phone], [t...
Audit Logout	
RPC:Completed	exec sp_reset_connection
Audit Login	-- network protocol: LPC set quoted_identifier on set ...
RPC:Completed	exec sp_executesql N'SELECT [t0].[InvoiceId], [t0].[Cust...
Audit Logout	
RPC:Completed	exec sp_reset_connection
Audit Login	-- network protocol: LPC set quoted_identifier on set ...
RPC:Completed	exec sp_executesql N'SELECT [t0].[InvoiceId], [t0].[Cust...
Audit Logout	
RPC:Completed	exec sp_reset_connection
Audit Login	-- network protocol: LPC set quoted_identifier on set ...
RPC:Completed	exec sp_executesql N'SELECT [t0].[InvoiceId], [t0].[Cust...
Audit Logout	
RPC:Completed	exec sp_reset_connection
Audit Login	-- network protocol: LPC set quoted_identifier on set ...
RPC:Completed	exec sp_executesql N'SELECT [t0].[InvoiceId], [t0].[Cust...
Audit Logout	
RPC:Completed	exec sp_reset_connection
Audit Login	-- network protocol: LPC set quoted_identifier on set ...
RPC:Completed	exec sp_executesql N'SELECT [t0].[InvoiceId], [t0].[Cust...
Audit Logout	
Audit Logout	
RPC:Completed	exec sp_reset_connection
Audit Login	-- network protocol: LPC set quoted_identifier on set ...
SQL:BatchS...	SELECT [t0].[CustomerId], [t0].[Title], [t0].[Phone], [t...
SQL:BatchC...	SELECT [t0].[CustomerId], [t0].[Title], [t0].[Phone], [t...
Audit Logout	
RPC:Completed	exec sp_reset_connection

Теперь можно посмотреть, как себя поведет движок L2S с подсказками по тому, какие типы данных нам понадобятся сразу, т.е. метод DoConfigureLoad() перегружен в конкретном адаптере. Запускаем тест и видим следующее:

```
SELECT [t0].[CustomerId], [t0].[Title], [t0].[Phone], [t0].[Website],
[t1].[InvoiceId], [t1].[CustomerId] AS [CustomerId2], [t1].[Address],
[t1].[IsPaid], [t1].[IsShiped], (
    SELECT COUNT(*)
    FROM [dbo].[Invoice] AS [t2]
    WHERE [t2].[CustomerId] = [t0].[CustomerId]
    ) AS [value]
FROM [dbo].[Customer] AS [t0]
LEFT OUTER JOIN [dbo].[Invoice] AS [t1]
  ON [t1].[CustomerId] = [t0].[CustomerId]
ORDER BY [t0].[CustomerId], [t1].[InvoiceId]
```

EventClass	TextData
Audit Login	-- network protocol: LPC set quote...
Audit Logout	
RPC:Completed	exec sp_reset_connection
Audit Login	-- network protocol: LPC set quote...
SQL:BatchStarting	SELECT [t0].[CustomerId], [t0].[Tit...
SQL:BatchCompleted	SELECT [t0].[CustomerId], [t0].[Tit...
Audit Logout	

Результат, как говорится, на лицо. Один запрос, получающий нужные данные. Однако увлекаться данной опцией чрезмерно тоже не стоит и надо внимательно и аккуратно анализировать запросы сгенерированные L2S в результате ваших советов по загрузке.

Типичный адаптер чтения

Как показывает практика, типичный адаптер для чтения в большинстве случаев будет выглядеть таким образом:

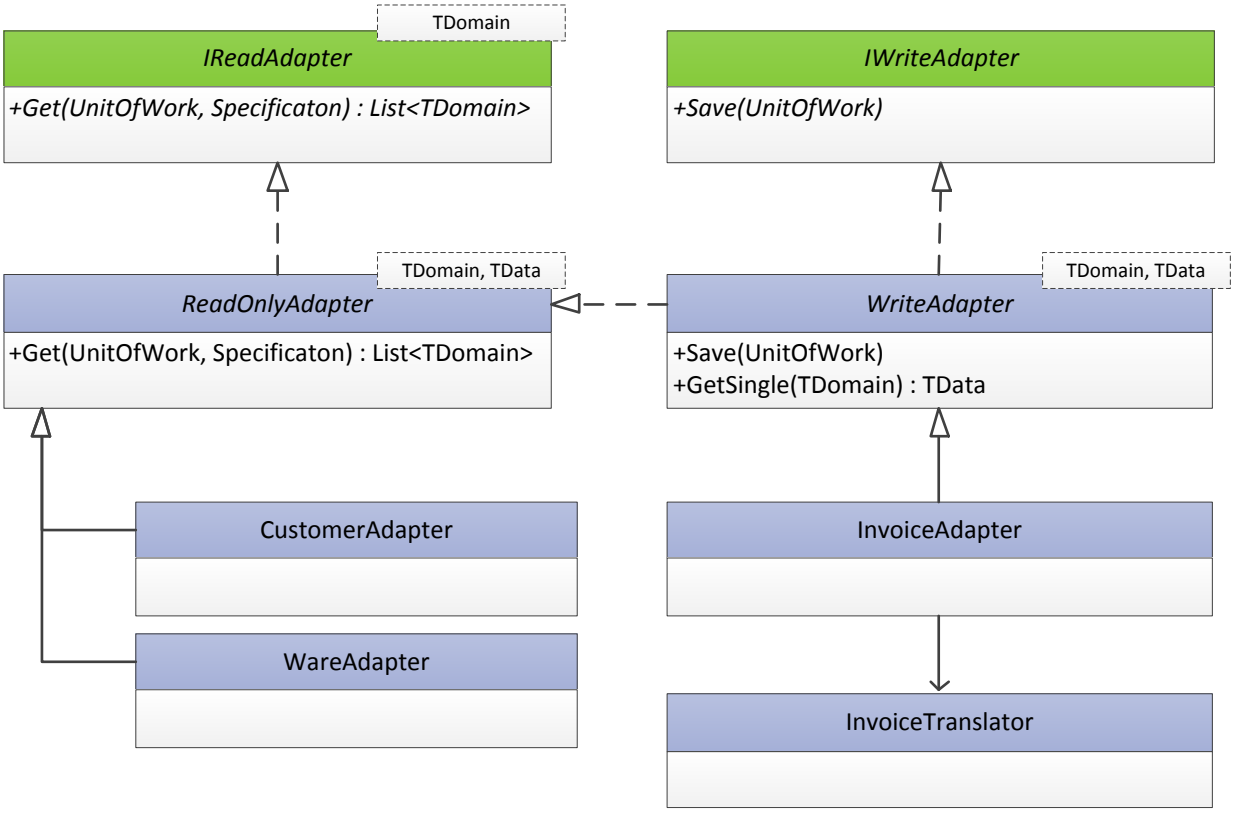
```
public class WareAdapter : ReadOnlyAdapter<Ware, ware> {
    public WareAdapter() : base(new WareTranslator()) {}
}
```

Просто и без лишних сложностей.

Да, пока что классы трансляторов остаются черным ящиком, но это ненадолго, скоро дойдет очередь и до них, а сейчас рассмотрим адаптеры для записи.

Адаптеры записи

Как и в случае с адаптерами для чтения, большую часть работы можно вынести в абстрактный класс. Интерфейс **IWriteAdapter** необходим опять же в целях обнаружения классов.



Если вы взглянете на метод InfrastructureFacade.Commit() то увидите, что получение всех адаптеров для записи получается из StructureMap скопом по одному только интерфейсу, что весьма удобно, так как они нужны всегда все разом для определения очередности работы.

IWriteAdapter

В интерфейс вынесены метод и свойство необходимые для работы в классе фасада, а именно Save() и Sequence.

```
public interface IWriteAdapter {
    void Save(UnitOfWork unitOfWork);
    int Sequence { get; }
}
```

Теперь можно переходить к рассмотрению «мяса», основной работы по сохранению данных.

WriteAdapter

В плане организации, из схемы видно, что адаптер записи наследуется от адаптера чтения, что логично, так как в большинстве случаев возможность записи это расширенные возможности адаптера, а множественного наследование классов в C# нет. В противном случае можно было бы для конкретных адаптеров писать два класса родителя, но не будем развивать эту тему.

Конечно же, самым интересным методом в классе будет метод **Save()**, но о нем чуть позже.

Адаптер записи, как уже упоминалось ранее, должен получить все элементы из UnitOfWork (UoW) своего типа, и дать указание базе на обновление, запись либо удаление данных. Для обновления данных и записи потребуется транслятор, который так же нужен родительскому классу, так что логичным представляется такой конструктор:

```
private readonly Translator<TDomain, TData> translator;

protected WriteAdapter(Translator<TDomain, TData> translator) : base(translator) {
    this.translator = translator;
}
```

Следует не забыть про поле, указывающее на последовательность сохранения данных. В данном случае и на практике мне хватало такой реализации (да, свойство виртуальное):

```
public virtual int Sequence {
    get { return 100; }
}
```

Чем меньше число, тем раньше будет вызван адаптер записи для обработки UoW.

Метод Save() потребует детального рассмотрения, есть в нем некоторые нюансы. Итак:

```
public void Save(UnitOfWork unitOfWork) {
    context = new WarehouseDataContext(ConnectionHelper.CurrentConnection);
    Table = context.GetTable<TData>();

    unitOfWork.ForEachInserted<TDomain>(
        domainObject => {
            var dataObject = new TData();
            translator.Persist(domainObject, dataObject, unitOfWork);
           // translator.PersistCreatedAuditableFields(dataObject);
            Table.InsertOnSubmit(dataObject);
            context.SubmitChanges();
           // unitOfWork.UpdateIdentity(domainObject, dataObject.id);
        });

    unitOfWork.ForEachUpdated<TDomain>(
        (domainObject) => {
            var dataObject = GetSingle(domainObject);
            if (dataObject != null) {
                translator.Persist(domainObject, dataObject, unitOfWork);
            }
        });

    context.SubmitChanges();

    unitOfWork.ForEachDeleted<TDomain>(
        (domainObject) => {
            var dataObject = GetSingle(domainObject);
            if (dataObject != null) {
                translator.AssertCanBeDeleted(dataObject);
                DeleteDataObject(dataObject);
            }
        });


    context.SubmitChanges();
    context.Dispose();

    AfterCommit(unitOfWork);
}
```

Беглый взгляд на метод, выделяет 3 логических части:

- Работа с новыми данными.
- Работа с обновленными данными.
- Работа с удаленными данными.

Работа с новыми данными

Для того чтобы постоянно не искать части кода в общем листинге, скопирую повторно код для удобства:

```
unitOfWork.ForEachInserted<TDomain>(
    domainObject => {
        var dataObject = new TData();
        translator.Persist(domainObject, dataObject, unitOfWork);
       // translator.PersistCreatedAuditableFields(dataObject);
        Table.InsertOnSubmit(dataObject);
        context.SubmitChanges();
       // unitOfWork.UpdateIdentity(domainObject, dataObject.id);
    });
```

В данном коде, мы получаем все новые (вставленные) элементы в UoW и для каждого осуществляем процесс маппинга на новый элемент в базе данных. В первой строке лямбда-функции создается элемент таблицы данных, которые будет заполнен данными из доменного типа во второй строке с помощью метода **Persist()** транслятора. Далее идет служебный код вставки элемента в логическую базу и окончательная запись в базу с помощью метода **SubmitChanges()**.

Обратим внимание на код в комментариях.

Аудит данных

Первая строка с комментарием показывает идею, как можно использовать трансляторы в осуществлении аудита. Данный код выполняется для всех новых элементов, так что мы можем предположить, что определенные поля не заполнены и их надо заполнить вполне конкретной, известной заранее и не зависимо от типа данных информацией.

Генерацию всех классов L2S делает с модификатором **partial**, что дает богатые возможности по дополнению и унификации всех/части классов. Например для аудита можно создать интерфейс IDataAuditable и по этому признаку в методе **PersistCreatedAuditableFields()** заполнять поля даты создания элемента и логин того, кто данные эти создает. Снимается разом много монотонной и нудной работы.

Таким же образом можно поступать и для отслеживания обновлений, так как адаптер записи то место, где точно происходят последние изменения и указания на запись данных.

Работа с целочисленными суррогатными ключами

Важный момент для тех, кто использует в качестве основного ключа целочисленный ряд с генерацией на стороне сервера (identity). В вашем случае, созданные на клиенте элементы не имеют своего идентификатора и любые дочерние элементы не смогут быть записаны в базу, до того, пока мы не узнаем реальное значение родительского ключа.

Решением является четкий, заранее заданный порядок сохранения типов сущностей. Причем сохранение в реальную базу надо делать сразу после маппинга, чтобы обновилось значение

ключа в dataObject. После этого надо уведомить UnitOfWork о новом ключе **unitOfWork.UpdateIdentity()**.

Вообще работа с такими ключами очень серьезно влияет на разработку как UoW, так и адаптеров записи. В каких-то аспектах использование целочисленных ключей облегчает жизнь, но как показывает практика, для составления непротиворечивой модели, обработка изменений ключей и прочие вещи, связанные с тем, что сервер генерирует жизненно важные вещи – пагубно влияют на процесс разработки, делая многие рабочие процессы усложненными. Я рекомендую несколько раз подумать, прежде чем проектировать базу с целочисленными суррогатными ключами использующими опцию identity.

При относительно простых структурах данных и использовании натуральных ключей (или же GUID) можно вынести context.SubmitChanges() за пределы лямбда-функции.

Работа с измененными данными

Работа с измененными данными не менее интересна и увлекательна, чем с новыми, здесь тоже есть свои нюансы и подвохи.

```
    unitOfWork.ForEachUpdated<TDomain>(
        (domainObject) => {
            var dataObject = GetSingle(domainObject);
            if (dataObject != null) {
                translator.Persist(domainObject, dataObject, unitOfWork);
            }
        });
```

Хотя кода поменьше, чем при сохранении нового элемента, но он имеет больше связей с конкретными адаптерами записи. Первой строкой идет метод **GetSingle()**, который должен вернуть объект типа TData, соответствующий доменному объекту, т.е. надо найти конкретную запись в базе.

```
public TData GetSingle(TDomain domainObject) {
    return context.GetTable<TData>().Single(GetIdPredicate(domainObject));
}

protected abstract Func<TData, bool> GetIdPredicate(TDomain domainObject);
```

Как видно из кода, метод GetIdPredicate() является обязательным к реализации в конкретных классах, в нем будет указано, каким образом, по каким полям однозначно определить что объект TData является отображением на TDomain. Как это выглядит на практике будет в разделе трансляторов.

Если удалось найти такой объект, то транслятор переносит в объект TData (dataObject) значения полей TDomain (domainObject).

В данном примере не показано уже, но можно точно так же выделять методы для автоматического обновления полей аудита, если это требуется в вашей системе и схема базы достаточно унифицирована для таких действий.

Еще можно заметить, что система никак не реагирует на ситуацию, когда объект в базе никак не был найден. Это сделано умышленно, так как непонятно как обрабатывать такую ситуацию. Например, запись может быть удалена другим пользователем, и тогда нечего обновлять и в целом это нормальная ситуация, что данные устарели. По бизнес-правилам возможно стоит

сообщить об этом пользователю, а возможно что и не надо и это поведение by design. Так что как обрабатывать решать вам в зависимости от того, какие требования предъявляются к системе.

Работа с удаленными данными

Осталось последнее действие – удаление данных.

```
unitOfWork.ForEachDeleted<TDomain>(
    (domainObject) => {
        var dataObject = GetSingle(domainObject);
        if (dataObject != null) {
            translator.AssertCanBeDeleted(dataObject);
            DeleteDataObject(dataObject);
        }
    });
```

Уже есть знакомые части в организации этого куска кода и уже рассмотренные моменты, как метод GetSingle(), однако и здесь не применяется простой подход в удалении объектов из базы.

Полное удаление сущностей из базы всегда является опасным и спорным моментом, стоит подумать и проанализировать трижды, прежде чем удалить что-то, предпринять защитные меры от случайного, непреднамеренного удаления, которое может повлечь за собой необратимые последствия в целостности данных. Для этих целей вызывается метод **AssertCanBeDeleted()**, который не проводит анализа конкретной сущности, а знает только можно ли в принципе удалять сущности типа TData. С одинаковым успехом туда можно передавать и TDomain.

```
protected virtual void DeleteDataObject(TData dataObject) {
    Table.DeleteOnSubmit(dataObject);
}
```

Удаление сделано с помощью виртуального метода, так как можно как удалять элемент физически, либо перегрузить метод и помечать его удаленным, после чего обновлять запись в базе данных.

Порядок действий и действия после сохранения

Порядок основных логический действий тоже выбран не просто так. В результате разных действий пользователя, наиболее полный сценарий работы выглядит как:

1. Создание сущности – создаем какой-либо новый элемент, для последующей работы, один или несколько.
2. Редактирование сущности – редактируем созданные либо существующие элементы.
3. Удаление – возможно это откат созданных элементов, или удаление ненужных элементов.

Правильность работы с коллекциями по большому счету регулируется UoW, чтобы коллекции для вставки, обновления и удаления были верными, но если есть ошибка в UoW, то возможной была бы ситуация когда мы сначала удаляем элемент, а потом его редактируем. Или, например, создали элемент и откатили его, а при работе адаптера он сначала удалился, а потом создался. Так что такой порядок обработки коллекций UnitOfWork мне кажется дополнительной защитой от логических казусов при работе с данными.

После того, как все логические коллекции обработаны, дается команда на применение изменений и идет вызов виртуального метода **AfterCommit()**. Название говорит само за себя. Честно сказать не припомню уже в связи с чем вводился в старых проектах этот метод, но можно пофантазировать, что надо каким-то образом оптимизировать или перестроить UoW.

Можно считать, что абстрактный адаптер записи рассмотрен, и можно перейти к конкретной реализации.

Конкретный адаптер записи

Конкретная реализация мала и красива

```
public class InvoiceAdapter : WriteAdapter<Invoice, invoice> {
    public InvoiceAdapter() : base(new InvoiceTranslator()) {
    }

    protected override Func<invoice, bool> GetIdPredicate(Invoice domainObject) {
        return t => t.InvoiceId == domainObject.InvoiceId;
    }
}
```

Здесь, так же как и с адаптером чтения, нужный транслятор передается в конструктор базового класса, облегчая работу с самим адаптером на уровне регистрации в StructureMap.

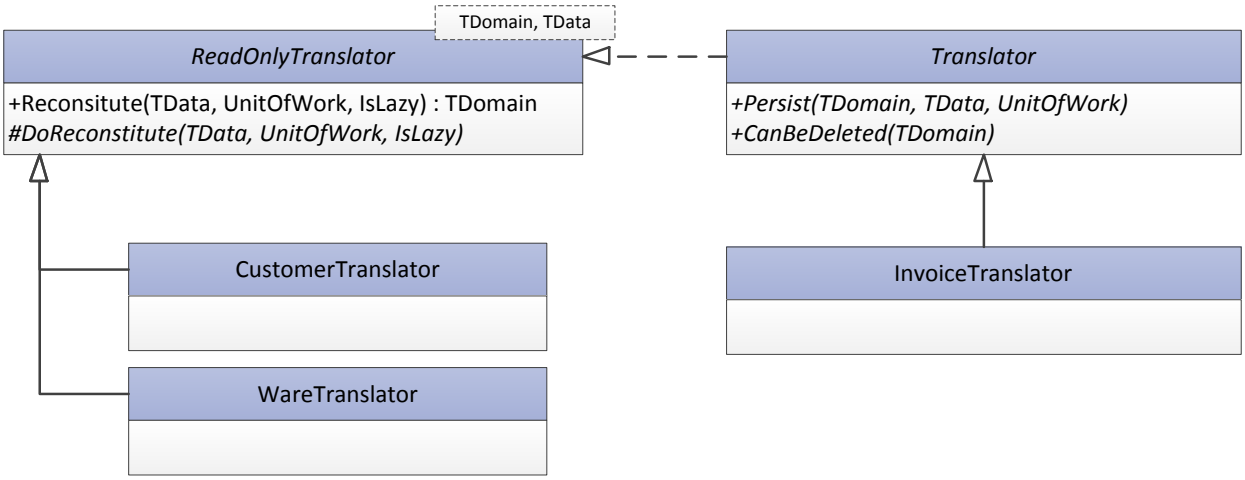
Обязательный перегруженный метод **GetIdPredicate()**, о котором уже упоминалось предстает здесь в своей реализации. Большинство реализаций будут в таком духе. Согласитесь, что написать пачку таких адаптеров не составит много труда и не займет много времени.

В данном примере не указаны все возможные перегрузки, да и не стоит, так как они в практической реализации просты очень.

Мы рассмотрели работу адаптеров записи и чтения, и настает черед трансляторов.

Трансляторы

Опять же для любителей и знающих UML предоставляю схему ниже, чтобы сразу оценить взаимодействие и структуру классов и наследования.



Хотя в схеме трансляторов и меньше компонентов, так как они динамически не создаются, не участвуют в автообнаружении и вообще ведут себя весьма пассивно по сравнению с другими классами, сказано о них будет не меньше чем об адаптерах.

Трансляторы чтения

Транслятор для чтения является абстрактным классов с двумя шаблонными параметрами TDomain – обозначающий доменные классы, TData – обозначающий классы Linq2Sql.

Базовый класс

Основной задачей транслятора чтения является выдача доменного объекта на основании записи в базе данных. Попутно это оказывается отличным местом для регистрации элементов в UnitOfWork. Общий алгоритм работы можно описать следующим образом:

- Убедиться, что пришли данные.
- Попробовать найти данные в UnitOfWork, может быть мы уже их переводили в доменный объект и тогда надо просто вернуть готовый объект из UoW.
- Если данные не нашли в UoW, то надо восстановить (маппировать) данные с помощью конкретного транслятора, на основании записи из базы и указаний по ленивой инициализации.
- Зарегистрировать полученный объект в UoW.
- Вернуть полученный объект.

Звучит не особенно сложно.

```
public virtual TDomain Reconstitute(TData dataObject, IUnitOfWork unitOfWork,
IsLazy<TDomain> isLazy) {
    if (ReferenceEquals(dataObject, null)) {
        return null;
    }

    var domainObject = unitOfWork.GetActive<TDomain>()
                                 .Find(i => GetPredicate(i,dataObject));
    if (domainObject != null) {
        return domainObject;
    }

    domainObject = DoReconstitute(dataObject, isLazy, unitOfWork);
    ((UnitOfWork) unitOfWork).Register(domainObject);

    return domainObject;
}

protected abstract TDomain DoReconstitute(TData dataObject, IsLazy<TDomain> isLazy,
IUnitOfWork unitOfWork);

protected abstract bool GetPredicate(TDomain domain, TData data);
```

Проверку на пустоту можно не пояснять.

Поиск в UoW. Искать элемент стоит только среди «живых» элементов, при этом в общем случае мы не знаем каким образом искать элемент, так как на доменные сущности не наложено никакого ограничения по реализации уникальных ключей. Накладывание такого ограничения может быть имеет смысл в некоторых случаях, но это сузит задачу и приведет к частному случаю, мы же рассмотрим более общий.

В общем случае потребуется от конкретного адаптера определить способ сравнения TData и TDomain, чтобы можно было определить, восстанавливали мы такой элемент уже или нет. Для этого определяем абстрактный метод **GetPredicate()**. Если нашли элемент, то сразу его возвращаем.

С помощью метода **DoReconstitute()** производится маппинг объектов в конкретном адаптере, это надо будет описывать для каждого типа данных. Об этом будет чуть позже подробнее. Так же

больше внимания уделим классу **IsLazy<TDomain>**, который на самом деле является спецификацией.

После того, как получили доменный объект необходимо его зарегистрировать, используя метод **Register()**, обращаясь к реальному классу UoW, а не его интерфейсу. В данном моменте стоит обратить внимание на то, что в результате маппинга объектов можно получить развесистую структуру дочерних классов, которые необходимо будет так же регистрировать в UoW. По умолчанию зарегистрируется только родительский элемент то, что явно передано на регистрацию в метод. Для того чтобы зарегистрировать дочерние элементы, в метод DoReconstitute() передается UoW, так что заботится об этом надо самостоятельно. Волноваться не надо, пример такого сложного восстановления будет рассмотрен.

Конкретные трансляторы

Простой/типовой транслятор

К такому типу трансляторов можно отнести следующую реализацию транслятора данных для товаров.

```
public class WareTranslator : ReadOnlyTranslator<Ware, ware> {
    protected override Ware DoReconstitute(ware dataObject, IsLazy<Ware> isLazy
                                           ,IUnitOfWork unitOfWork) {
        return new Ware {
                            WareId = dataObject.WareId
                        };
    }

    protected override bool GetPredicate(Ware domain, ware data) {
        return data.WareId == domain.WareId;
    }
}
```

Проводится ручной маппинг свойств класса базы данных, на доменный класс. В данном примере я ограничился только маппингом одного свойства. Но идея, думаю, ясна.

Так же в классе представлена реализация метода GetPredicate(), которая не представляет сложности и не требует пояснений.

Сложный транслятор

Примером такого транслятора может послужить восстановление заказчика из базы данных вместе с заказами. Т.е. у нас происходит одновременное восстановление двух сущностей. Давайте посмотрим, как это происходит:

```
public class CustomerTranslator : Translator<Customer, customer> {
    private readonly InvoiceTranslator invoiceTranslator;

    public CustomerTranslator() {
        invoiceTranslator = new InvoiceTranslator();
    }

    protected override Customer DoReconstitute(customer dataObject
                                               ,IsLazy<Customer> isLazy
                                               ,IUnitOfWork unitOfWork) {
        var domainObj = new Customer {
                                          CustomerId = dataObject.CustomerId
                                      };
        if (isLazy.NotContains<Invoice>()) {
            foreach (var invoice in entitySet) {
```

```
                var reconstitute = invoiceTranslator.Reconstitute(invoice
                                                                  , unitOfWork
                                                                  , isLazy.For<Invoice>());
                doReconstitute.Invoices.Add(reconstitute);
            }
        }
        return domainObj;
    }
    . . .
}
```

Уже в реализации конструктора видно, что будем восстанавливать более одной сущности. Хотя создание транслятора и не затратное по времени, но, тем не менее, лучше создать класс заранее и использовать его.

В методе DoReconstitute() я показываю начальное восстановление для заказчика и далее начинаются более интересные вещи. Как упоминалось ранее, класс IsLazy является наследником Specification и служит для передачи информации о том, к каким образом строить «ленивые» объекты.

При работе с IsLazy надо проверить, что спецификация не содержит типа, который мы собираемся восстанавливать. Для решения проблемы строгой типизации у класса IsLazy есть шаблонный метод **For<>()**, который позволяет конвертировать спецификацию для передачи в другие трансляторы, как в нашем случае для транслятора заказов.

Как вы видите, для восстановления информации о заказе используется уже готовый класс транслятора, что говорит о том, что нет необходимости заботиться о сохранении элементов в UoW. Главное не забыть правильно обработать полученный восстановленный объект в соответствии с нашими желаниями.

На мой взгляд, вполне логично описывать восстановление всех дочерних объектов при запросе родительского элемента. Писать в дальнейшем сервисы по связыванию элементов родственных мне нравится гораздо меньше. Однако при описанном мной подходе цепочка восстановления может получиться очень большой, так что не забывайте указывать спецификации для ленивой загрузки.

Трансляторы записи

Хотя чтения и является основной операцией при работе с большинством данных, но большинство программ все же пишутся для редактирования данных (пусть хотя бы в ограниченных рамках и для ограниченного круга лиц). Так что без трансляторов записи нам никуда.

Базовый класс

Вы наверно приятно удивитесь, глядя на столь короткую реализацию класса, после всего уже описанного. Самостоятельно класс базовый класс транслятора в представленном случае ничего не делает, а только объявляет абстрактные методы.

```
public abstract class Translator<TDomain, TBase> : ReadOnlyTranslator<TDomain, TBase>
    where TDomain : class
    where TBase : class {
    public abstract void Persist(TDomain domainObject, TBase dataObject, UnitOfWork
unitOfWork);
    public abstract void AssertCanBeDeleted(TBase dataObject);
}
```

Хотя, если вспомнить описание базового класса адаптера записи, то там упоминалась возможность написать именно в этом классе методы для реализации аудита – **PersistCreatedAuditableFields()**. Повторятhья не буду, а кто не запомнил, тот может вернуться и перечитать про Аудит данных.

В данном случае базовый класс обязывает наследников описать способ маппинга доменных объектов на автогенерированные классы L2S (метод Persist()) и явно описывать можно ли удалять объекты или нет (метод AssertCanBeDeleted()).

Можно было бы предоставить виртуальную реализацию методу AssertCanBeDeleted() указав, что объекты по умолчанию удалять нельзя. Однако я бы хотел, чтобы разработчики задумывались над этим вопросом каждый раз при создании транслятора записи, так как очень важно понимать, что будет с системой при удалении записи.

Так же вы можете заметить, что метод AssertCanBeDeleted() не возвращает булево значение. Это сделано специально для того, чтобы заставить строить систему исключений, ведь если на удаление попал элемент который нельзя удалять, значит не просто что-то пошло не так, а какой-то кусок программы неверно сделан и следует обратить самое пристальное внимание на разбор возникшей ситуации.

Теперь можно перейти к рассмотрению конкретного примера.

Конкретные трансляторы записи

Я думаю, что можно рассмотреть сразу сложный вариант сохранения, с зависимыми данными, когда одна доменная сущность записывается в несколько разных таблиц. Примером такого доменного типа является Invoice. Если бы мы делали маппинг доменного класса только на таблицу invoice, то у нас бы получался пустой заказ, что не имеет смысла. Можно было бы разнести во времени записывание общей информации о заказе, а потом о его составе, но случись что между этими действиями и мы получим нецелостные логические данные, хотя база будет в полном порядке. Именно поэтому конструировать/обновлять класс L2S для данного типа следует для всех связанных элементов.

```
public override void Persist(Invoice domainObject, invoice dataObject, UnitOfWork
unitOfWork) {
    dataObject.InvoiceId = domainObject.InvoiceId;
    dataObject.CustomerId = domainObject.Customer.CustomerId;
    dataObject.Address = domainObject.Address;

    dataObject.invoceContents.Clear();
    foreach (var ware in domainObject.Wares) {
         var invoceContent = new invoceContent {
                                                    InvoiceId = domainObject.InvoiceId,
                                                    WareId = ware.WareId,
                                                    Date = domainObject.Date,
                                                    Price = ware.Price,
                                                    Quantity = ware.Quantity,
                                                };

        dataObject.invoceContents.Add(invoceContent);
    }
}
```

В первых трех строках идет запись общей информации о заказе, а дальше идет формирование связанных элементов по элементам заказа. Обратите внимание, что формирование класса invoiceContent идет по данным двух элементов.

В зависимости от задачи и данных, вы можете как обновлять данные в базе, так и очищать старый список и записывать новые элементы. Мне оказалось легче пересоздавать список элементов заказа. =)

В результате всех действий у вас появится связанная модель данных для Linq2Sql, по которой он сможет уже самостоятельно определить, что и в каком порядке записывать.

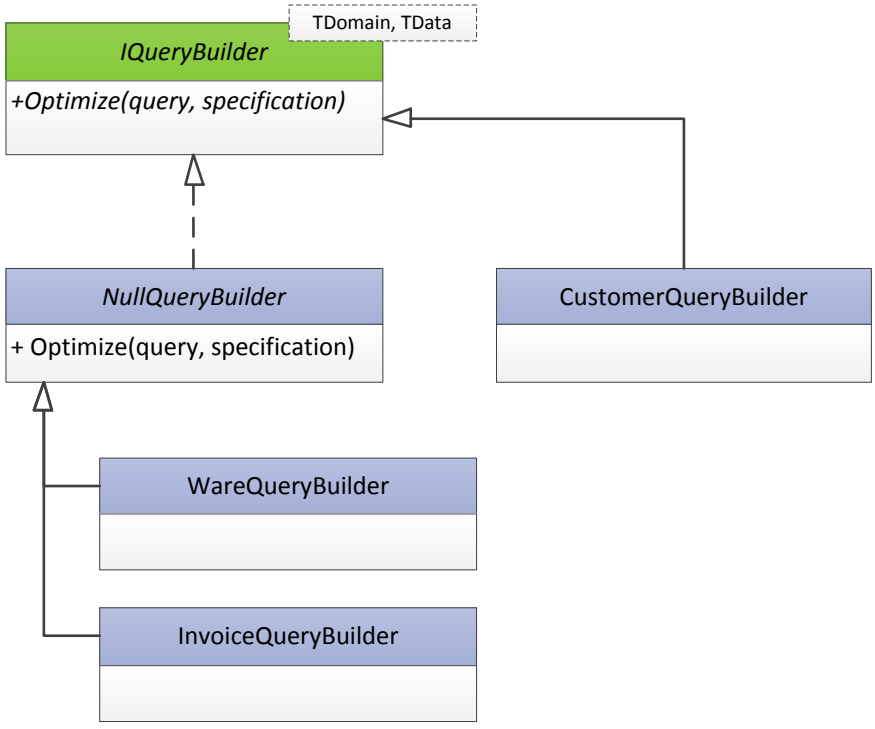
```
public override void AssertCanBeDeleted(invoice dataObject) {}
```

В данном случае метод AssertCanBeDeleted() пустой, так как элементы можно удалять.

Итак, получается, что уже описаны все механизмы для восстановления и записи данных, и настает очередь оптимизации запросов к базе данных.

Построение запросов

Уже традиционно покажу сразу UML схему, чтобы вы смогли ощутить какие классы будут рассмотрены и как они зависят друг от друга.



Хорошо уметь загружать связанные данные, хорошо уметь восстанавливать только нужные данные, но было бы еще замечательно уметь получать только нужные данные от базы данных, чтобы не выкачивать многие тысячи строк, которые нам может и не нужны вовсе. Для этих целей и существует в нашей схеме построители (или оптимизаторы) запросов.

Linq2Sql позволяет гибко работать с запросами в стиле fluent interface переводя их в sql запросы. Было бы упущением не воспользоваться этой возможностью в полной мере. Особенно учитывая

то, что мы уже работаем с логическими таблицами L2S, которые реализуют интерфейс IQueryable. Да, когда мы в адаптере чтения обращаемся к контексту с методом **GetTable<TData>()**, мы получаем объект, который можно донастраивать с помощью методов **Where()**, разбирая пользовательские спецификации. Собственно в этом и состоит идея.

Следуя только что сказанному, интерфейс для построения запросов может выглядеть следующим образом:

```
public interface IQueryBuilder<TDomain, TData> {
    IQueryable<TData> Optimize(IQueryable<TData> query, Specification<TDomain>
specification);
}
```

Модифицировали запрос, если надо, и отдали его обратно. Отлично, теперь рассмотрим как это может быть реализовано.

NullQueryBuilder

Согласитесь, что не все запросы необходимо оптимизировать каким-либо образом. Может пока нет нужды в сужении результатов запроса, а может мы просто не дошли до этого, но в любом случае нам понадобится некий общий класс, который просто вернет исходный запрос. Исходя из целей класса, логично его назвать NullQueryBuilder.

```
public abstract class NullQueryBuilder<TDomain, TData> : IQueryBuilder<TDomain, TData> {
    public IQueryable<TData> Optimize(IQueryable<TData> query
                                      , Specification<TDomain> specification) {
        return query;
    }
}
```

Данный класс сделан абстрактным в угоду специфики использования классов оптимизации. Как и большинство других шаблонных классов из одного семейства, данные классы удобно автоматически находить и регистрировать в проекте с помощью StructureMap. SM не позволяет попробовать взять шаблонный класс. Т.е. на примере текущей реализации:

```
private IQueryBuilder<TDomain, TData> GetQueryBuilder() {
    var queryBuilders = ObjectFactory.Container
        .ForGenericType(typeof (IQueryBuilder<,>))
        .WithParameters(typeof (TDomain), typeof (TData))
        .TryGetInstanceAs<IQueryBuilder<TDomain, TData>>();

    return queryBuilders;
}
```

В StructerMap нет метода **TryGetInstanceAs()**, благодаря которому мы могли бы протестировать наличие типа в коллекции для IQueryBuilder, и в случае чего вернули бы конкретный NullQueryBuilder. Таким образом, приходим к необходимости объявлять класс NullQueryBuilder абстрактным, а пустые оптимизаторы наследовать от него.

```
public class WareQueryBuilder : NullQueryBuilder<Ware, ware> {}
public class InvoiceQueryBuilder : NullQueryBuilder<Invoice, invoice> {}
```

Теперь можно заняться настоящим оптимизатором запросов.

Реализация оптимизации запросов

Прежде чем приступить к описанию запросов, скажу пару слов про спецификации, которые используются для построения оптимизированного запроса. Такие спецификации, как и все остальные наследуются от класса Specification, однако они так же должны предоставлять доступ к информации, по которой идет определение условия (хотя и не всегда).

```
ublic class ActiveCustomer : Specification<Customer> {
    public override bool IsSatisfiedBy(Customer obj) {
        return obj.Invoices.Any();
    }
}

public class CustomerTitleStartsWith : Specification<Customer> {
    public string StartPattern { get; private set; }

    public CustomerTitleStartsWith(string startPattern) {
        StartPattern = startPattern;
    }

    public override bool IsSatisfiedBy(Customer obj) {
        return obj.Title.StartsWith(StartPattern);
    }
}
```

В противном случае построить запрос было бы проблематично, так как строятся они следующим образом:

```
public class CustomerQueryBuilder : IQueryBuilder<Customer, customer> {
    public IQueryable<customer> Optimize(IQueryable<customer> query
                                        , Specification<Customer> spec) {
        var belongsToBase = spec.ExtractSupersetSpecification<ActiveCustomer>();
        if (belongsToBase.IsNotNull()) {
            query = query.Where(i => i.invoices.Any());
        }

        var startsWith  = spec.ExtractSupersetSpecification<CustomerTitleStartsWith>();
        if (titleStartsWith.IsNotNull()) {
            query = query.Where(i => i.Title.StartsWith(titleStartsWith.StartPattern));
        }

        return query;
    }
}
```

По коду можно уже заметить паттерн, по которому обрабатываются все спецификации. С помощью метода **ExtractSupersetSpecification()**, мы определяем наличие интересующей спецификации в переданной и если находим, то уже на основе ее и данных, которые та содержит, накладывать условия на итоговый набор данных.

Если в первом случае, со спецификацией ActiveCustomer, мы можем построить запрос исходя из логического значения спецификации, то во втором случае, без дополнительной информации, которая в себе содержит спецификация – не обойтись. Хотя, по задумке, спецификация не должна раскрывать условия, по которым она фильтрует данные. Но это не страшное изменение, главное, чтобы это условие нельзя было изменить после создания конкретной спецификации.

Например, в результате применения к исходному запросу спецификации ActiveCustomer, получаем такой запрос:

```
SELECT [t0].[CustomerId], [t0].[Title], [t0].[Phone], [t0].[Website]
FROM [dbo].[Customer] AS [t0]
WHERE EXISTS(
    SELECT NULL AS [EMPTY]
    FROM [dbo].[Invoice] AS [t1]
    WHERE [t1].[CustomerId] = [t0].[CustomerId]
    )
```

Получаемые запросы советую тоже порой пересматривать, так как порой встречается страшное . К сожалению так сразу не изобразишь «страшное» и то, как от него избавится. Так же советую изучить, что L2S может перевести в SQL.

Такое впечатление, что все ключевые технические моменты описаны и можно рассматривать тестовый проект в деталях, а так же более-менее четко представлять, как все реализовано и связано в таком подходе по получению данных из БД.

Независимые оптимизации

Под независимыми оптимизациями, я понимаю оптимизации, которые не зависят от типа данных. К ним, например, можно отнести использование методов **Top(), Take(), Skip()**. Так как они не привязаны к конкретным типам данных, а накладывают ограничения на количество записей, то для реализации этих методов логично будет выделить базовый класс, в котором будет проходить анализ по наличию таких спецификация и применению их к запросам.

Ограничения и особенности

У любой технологии и подхода есть ограничения, так и описанный подход не является серебряной пулей. Ограничения и особенности у данного подхода следующего характера:

- Данный подход никак не оптимизирует работу с хранимыми процедурами. Если у вас до сих пор пишутся CRUD процедуры, то это не облегчит работу с ними. Данный подход отменяет сами эти процедуры.
  Так же данный подход никак не регламентирует и не облегчает работу с большими наборами хранимых процедур, если вы их используете в клиентском коде.
- При данном подходе сложно сделать зависимость двух адаптеров от одного доменного класса. Т.е. если вы хотите в зависимости от ситуации записывать данные о заказчиках то в таблицу Customer, а то в таблицу TempCustomer, то это будет проблематично. Проблема решается созданием дополнительного класса TempCustomer. Что, впрочем, мне видится куда как более правильным решением вообще.
- Несмотря на то, что весь код пронизан использованием UnitOfWork, вполне возможно работать с описанной структурой и без него. Проверено на практике на текущем проекте. Работы по устранению UoW не так уж и много, как может показаться на первый взгляд.
- Придется порой работать с настройками элементов таблиц в графическом редакторе dbml. Иногда приходится явно указывать по каким полям искать исходные данные для обновления, что учитывать при обновлении/записи, а что нет. Следует ли проверять значения перед записью или нет. К сожалению, общих рекомендаций по решению описанных проблем выдать для пункта нельзя. Все зависит от ситуации, и решения нарабатываются долгим опытным путем.
- Данный подход не очень подойдет для высоконагруженных проектов. Так как скорость чтения у L2S не самая большая, хотя побольше чем у EF. Основным плюсом L2S является легкость записи связанных данных одной транзакцией.

Так как я стараюсь избегать использования хранимых процедур для простых действий с данными, то для меня описанный подход (с вариациями) служит верой и правдой уже около 5 лет.

Areas for improvement

Фантазировать на тему улучшения можно фантазировать наверно очень долго, а так же пытаться впихнуть невпихуемое в рамки одного проекта. Однако именно разные эксперименты сподвигли меня написать сей труд по работе фасада, и вот что можно улучшить в его работе. Некоторые вещи из этого списка я даже покажу, как сделать, но в других статьях.

Фасад

Возможно, стоит дополнить интерфейс фасада асинхронным получением данных. Но не для того, чтобы дергать его асинхронно из ViewModel, презентеров или контроллеров. Упомянутые структуры не должны по идее дергать фасад, а работать с сервисами обработки данных.

Трансляторы

Это первое, что бросается в глаза после начала реального использования. В текущих примерах проблема ручного сопоставления полей может быть не видна начинающим разработчикам, но опытные должны были сразу заметить эту монотонную часть работы. По началу, может показаться, что трудов не так уж и много, но когда начинаешь вручную описывать маппинг класса с 15 свойствами, то это утомляет. Велик шанс что-то забыть или же сопоставить не то поле. Требуется опять же больше тестов для проверки такого кода.

Да, от этого можно избавиться, используя различные автомапперы, например EmitMapper (о его использовании в представленной схеме будет отдельным рассказом), но тогда у вас должно быть полное совпадение по названию полей в доменном классе и в таблицах, или во вьюхах, а так же данные класса должны быть представлены свойствами. Вообще тема сопоставления имен в таблицах и в доменных классах достаточно широка.

Адаптеры

Над адаптерами тоже можно потрудиться. Уже сейчас разделено получение первичных данных из базы и восстановление их в доменные объекты. При таком разделении можно восстанавливать объекты многопоточно, что может дать неплохой выигрыш в подготовке данных из базы. Конечно же не для всех типов данных такое можно будет провернуть, но если данные могут восстанавливаться независимо, то почему бы и нет? Данным процессом, как и многими другими в данной реализации можно управлять с помощью конструирования специальных спецификаций. Кроме того, для того, чтобы не усложнять излишне схему, потребуется вынести код регистрации элементов в UoW в адаптер, так как в противном случае надо будет делать потокобезопасное регистрирование объектов. Создание потокобезопасных объектов тот еще труд и проблема, так что лучше всего просто перенести ответственность, на мой взгляд, это вызовет меньше проблем, так как чтение из UoW (если оно будет) разными потоками проблем создавать не должно.

Если создать строгие правила по именованию полей в базе данных, использовать те же имена, что и в коде, а для первичных ключей задавать имена по правилу: Имя таблицы + Id, то можно не описывать каждый раз метод **GetPredicate()**, в адаптерах записи.

```
protected override Func<invoice, bool> GetIdPredicate(Invoice domainObject) {
    return t => t.InvoiceId == domainObject.InvoiceId;
}
```

Такой код вполне можно генерировать на лету и сохранять его как делегат, чтобы создание шло только при первом обращении. Таким образом можно избавится еще от одного возможного места ошибок.

Да, могут быть составные натуральные ключи, но помня статью о составлении ключей, можно сказать, что суррогатный ключ для связи данных все равно будет присутствовать, так что такая оптимизация имеет право на жизнь.

В зависимости от того, с какими структурами данных вы работаете, их можно читать с помощью микро-ORM, а записывать с помощью L2S. Для этого видимо придется сильно переписать адаптеры чтения, но в каком ключе, в двух словах не описать, хотя пример использования для PetaPoco и будет рассмотрен в дальнейших статьях.

Больше в голову не приходит глобальных оптимизаций, таких вещей, что прям вот мешали бы или утомляли сильно. Разных оптимизаций касательно реализации не стоит рассматривать, так как это зависит уже от конкретного проекта и рассматривать или упоминать их нет смысла.

Заключение

Я постарался, как можно более полно рассказать о структуре получения данных, с описанием возможностей, ограничений, рассказывая, что может быть исправлено или дополнено в зависимости от ситуации. Надеюсь, у меня это получилось и по прочтении многих тысяч слов вы смогли нарисовать у себя в голове описанную модель и представить, как она работает.

Главное достоинство описанной модели, на мой взгляд, это простота конечного использования и высокая модульность самой разработки. Еще раз напомню конечное использование фасада:

```
var facade = new InfrastructureFacade();
var customers = facade.Get(Specification<Customer>.Null);
```

Простота использования превыше всего, скрытое стремление к волшебному методу **DoAllStuff()**.

Hard’n’heavy!

Violet Tape