УДК 519.683.8
Горшунов А.І.
Білоруський державний університет інформатики і радіоелектроніки

АРХИТЕКТУРНАЯ МОДЕЛЬ ПОСТРОЕНИЯ ВЕБ-ПРИЛОЖЕНИЙ CQRS

Горшунов А.І. Архітектурна модель побудови веб-застосунків CQRS. У статті пропонується базова імплементація архітектурної моделі побудови застосунків за методологією CQRS (Command and Query Responsibility Segregation, поділ відповідальності на команди і запити), аналізуються особливості проектування, процесу розробки бізнес-додатків; переваги і недоліки даної методології.
Ключові слова: архітектура, процес розробки, CQRS, C#, програмне забезпечення.

Harshunou A.I. CQRS as architectural model web applications development. The article proposes basic implementation of architectural model building applications using CQRS methodology (Command and Query Responsibility Segregation), analyzes the characteristics of design, business application development process; the advantages and disadvantages of this methodology.
Keywords: architecture, development process, CQRS, C#, software.

Горшунов А.И. Архитектурная модель построения веб-приложений CQRS. В статье предлагается базовая реализация архитектурной модели построения приложений по методологии CQRS (Command and Query Responsibility Segregation, разделение ответственности на команды и запросы), анализируются особенности проектирования, процесса разработки бизнес-приложений; преимущества и недостатки данной методологии.
Ключевые слова: архитектура, процесс разработки, CQRS, C#, программное обеспечение.

Постановка проблемы. Процесс разработки программного обеспечения изменяется с развитием технологий программирования: системы усложнятся, количество данных и пользователей растёт, изменяется и процесс разработки непосредственно программистами.

Заказчики приложений редко могут дать подробное описание всех бизнес-правил и функционала приложений, т.к. сами зачастую не знают всех требований и ожиданий клиента, формируя «сырое» техническое задание и перекладывая ответственность и выяснение требований непосредственно на команду разработчиков во время этапа разработки.

Основной проблемой разработчиков становится поддержка постоянно меняющихся бизнес-правил, интеграций, оптимизация производительности и расширение функциональности приложений, и, как следствие, проблема качества кода.

В крупных бизнес-приложениях зачастую существует множество предметных областей и сложных правил, которые сложно спроектировать и описать единой моделью. Зачастую бизнес-аналитики сами не в состоянии описать модель и выявить сущности приложения либо провести декомпозицию. В итоге аналитика и моделирование предметной области частично (а иногда и полностью) перекладывается на плечи разработчиков.

Также сами разработчики могут понимать и разделять приложение на компоненты и сущности по-разному, создавая тем самым проблемы дублирования кода, запросов, интерфейсов и даже перемешивание слоёв приложения, инфраструктурного кода и кода доменной модели.

Таким образом, учитывая особенности процесса разработки бизнес-приложений, разработчикам необходимо спрогнозировать возможность изменений, унифицировать терминологию и построить архитектуру приложения так, чтобы минимизировать «архитектурные» риски (вероятность полного переписывания приложения) в связи с изменившимися бизнес-правилами.

Целью работы является изучение и демонстрация возможностей применения архитектурной модели построения бизнес-приложений CQRS.

Основные результаты исследования. Модель Command-Query Separation (CQS) предлагает разделение работы с объектом на Запросы (Query) и Команды (Commands). При этом необходимо соблюдать следующие правила:

- запросы возвращает данные и никогда не меняют состояние объекта;

© Горшунов А.І.

- команды изменяют состояние объекта, в идеальном случае, не должны ничего возвращать.

Из этого следует, что во время отсутствия Команд:

- одинаковые Запросы гарантировано вернут одинаковый результат;
- любое количество любых Запросов не изменят состояние объекта;
- удаление Запроса из кода абсолютно прозрачно для объекта и не может дать побочных действий.

Эти свойства CQS используются в «программировании по контракту». В нем методы, осуществляющие проверку состояния объекта, являются Запросами. Это гарантирует их прозрачность для остального кода и возможность безопасного удаления части из них в Release версии кода (например, для оптимизации). Кроме того, подход, предлагаемый CQS, позволяет сделать код приложения более понятным именно благодаря разделению Команд и Запросов. Соответственно, в дальнейшем такое приложение легче поддерживать и модифицировать.

Необходимо отметить, что в некоторых случаях CQS не может быть использован в принципе. В качестве примеров можно рассмотреть реализации алгоритмов стек и очередь. Их методы Pop() и Dequeue() по определению должны и возвращать данные и изменять состояние самого объекта. Кроме того, в случае многопоточного программирования иногда удобнее иметь методы, противоречащие принципам CQS.

Область использования CQS достаточно широка. Например, он подходит для взаимодействия с элементами управления пользовательского интерфейса. Однако, в случае применения данного принципа для работы с источниками данных используется термин Command-Query Responsibility Segregation (CQRS). По сути это синоним, уточняющий область применения. CQRS несет в себе все принципы CQS и не ограничивает способ хранения данных. Это может быть СУБД, массив в памяти и т. д.

Таким образом, мы получаем набор объектов, которые меняют состояние системы, и набор объектов, которые возвращают данные. Типовой дизайн системы, где есть пользовательский интерфейс, бизнес-логика и база данных, показан на рисунке 1:

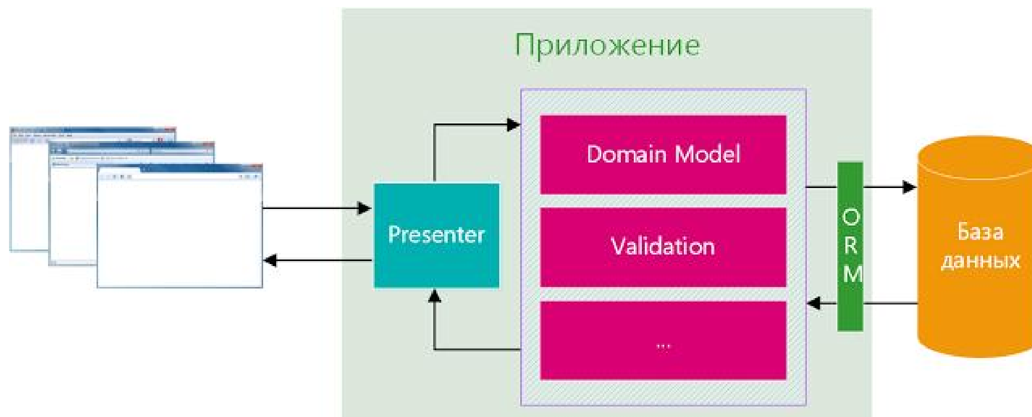

Рис.1. Классический пример архитектуры бизнес-приложения

CQRS предполагает, что не нужно смешивать объекты Команда и Запрос, а нужно их явно выделить (см. рисунок 2):

© Горшунов А.І.

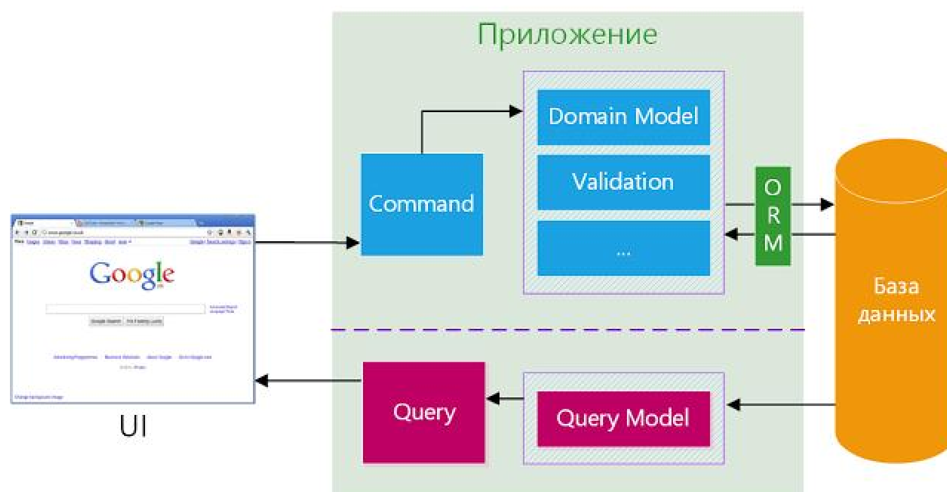

Рис. 2. Архитектурная модель CQRS

Дальше мы будем использовать эти базовые понятия и структуры, чтобы придать системе возможность горизонтального масштабирования. Но уже сейчас, явно выделив Команду и Запрос в вашем коде, вы получите более стройный дизайн системы. Если класс является Командой, то:

- изменяет состояние системы;
- ничего не возвращает;
- хорошо описывает предметную область, как действия пользователей над системой;
- контекст команды хранит нужные для её выполнения данные.

Если класс является Запросом, то:

- не изменяет состояние системы;
- контекст запроса хранит нужные для её выполнения данные;
- возвращает результат

В качестве демонстрации был выбран язык C#.

Основные интерфейсы в CQRS могут выглядеть так:

```
public interface IQuery<out TOutput>
{
   TOutput Ask();
}
public interface IQuery<in TSpecification, out TOutput>
{
   TOutput Ask(TSpecification spec);
}
public interface IAsyncQuery<TOutput> : IQuery<Task<TOutput>> { }
public interface IAsyncQuery<in TSpecification, TOutput> : IQuery<TSpecification, Task<TOutput>> { }
public interface ICommandHandler<in TInput>
{
    void Handle(TInput input);
}
public interface ICommandHandler<in TInput, out TOutput>
{
   TOutput Handle(TInput input);
}
public interface IAsyncCommandHandler<in TInput> : ICommandHandler<TInput, Task> { }
public interface IAsyncCommandHandler<in TInput, TOutput>: ICommandHandler<TInput,
Task<TOutput>>{}
```

© Горшунов А.І.

В этом случае в отсутствии Команд все Запросы всегда возвращают одинаковые результаты на одинаковых входных данных. Такая организация сильно упрощает отладку, потому что в Запросе нет состояния, которое могло бы изменить возвращаемый результат.

Приведем пример реального использования. Отчеты – не единственная частая задача чтения данных. Более общее определение типовых операций чтения это – фильтрация, постраничный вывод, создание проекций (представление агрегатов в необходимом на клиентской стороне виде). Для постраничного вывода из любой таблицы в базе данных и поддержкой фильтрации можно использовать всего одну реализацию IQuery:

```
public class ProjectionQuery<TSpecification, TSource, TDest>: IQuery<TSpecification,
IEnumerable<TDest>>
    , IQuery<TSpecification, int>
    where TSource : class, IHasId
    where TDest : class
{
    protected readonly ILinqProvider LinqProvider;
    protected readonly IProjector Projector;

    public ProjectionQuery(ILinqProvider linqProvier, IProjector projector)
    {
        if (linqProvier == null) throw new ArgumentNullException(nameof(linqProvier));
        if (projector == null) throw new ArgumentNullException(nameof(projector));
        LinqProvider = linqProvier;
        Projector = projector;
    }
    protected virtual IQueryable<TDest> GetQueryable(TSpecification spec) => LinqProvider
      .GetQueryable<TSource>().ApplyIfPossible(spec).Project<TSource,
TDest>(Projector).ApplyIfPossible(spec);

    public virtual IEnumerable<TDest> Ask(TSpecification spec) => GetQueryable(spec).ToArray();

    int IQuery<TSpecification, int>.Ask(TSpecification spec) => GetQueryable(spec).Count();
}
public class PagedQuery<TSortKey, TSpec, TEntity, TDto>: ProjectionQuery<TSpec, TEntity, TDto>,
    IQuery<TSpec, IPagedEnumerable<TDto>>
    where TEntity : class, IHasId
    where TDto : class, IHasId
    where TSpec : IPaging<TDto, TSortKey>
{
    public PagedQuery(ILinqProvider linqProvier, IProjector projector) : base(linqProvier, projector) { }

    public override IEnumerable<TDto> Ask(TSpec spec) =>
GetQueryable(spec).Paginate(spec).ToArray();

    IPagedEnumerable<TDto> IQuery<TSpec, IPagedEnumerable<TDto>>.Ask(TSpec spec)
        => GetQueryable(spec).ToPagedEnumerable(spec);

    public IQuery<TSpec, IPagedEnumerable<TDto>> AsPaged()
        => this as IQuery<TSpec, IPagedEnumerable<TDto>>;
}
```

Метод ApplyIfPossible проверит осуществляется фильтрация на уровне агрегата или проекции (бывает нужно и так, и так). Метод Project создаст проекцию с помощью AutoMapper.

© Горшунов А.І.

Принципы CQRS очень хорошо подходят для реализации по протоколу HTTP. Спецификация HTTP четко говорит GET-запросы должны возвращать данные с сервера. POST, PUT, PATCH – изменять состояние. Хорошим тоном в web-программировании считается перенаправление на GET после выполнения POST-операции, например, отправки формы. Таким образом: GET - это Запрос POST/PUT/PATCH/DELETE – это Команда.

Подобная архитектура может показаться «перегруженной». Действительно подобный подход накладывает определенные ограничения к квалификации разработчиков:

- требуется понимание паттернов программирования и хорошее знание платформы;
- первоначальные вложения в код инфраструктуры.

Преимущества:

- четкое отделение домена от инфраструктуры;
- минимизация объема кода в проекте, устранение дублирования кода, устранение циклических зависимостей, устранение рутины, использование соглашений, вместо избыточных конфигураций;
- регламентирование бизнес-логики и общей структуры проекта;
- абстракция от серверной инфраструктуры, поддержка горизонтального масштабирования.

Выводы. В результате исследования была изучена архитектурная модель построения бизнес-приложений SQRS, предложена базовая реализация модели, проанализированы преимущества и недостатки данной модели. Приведены примеры реализации задач проекции и постраничного вывода данных. Данная модель хорошо себя за рекомендовала на существующих проектах. Описанные в работе подходы к проектированию использованы при выполнении ряда проектов медицинской тематики (американские веб-сайты, специализирующиеся на поиске докторов, локаций госпиталей, сервисов: https://www.bannerhealth.com, https://www.eehealth.org, https://blog.hap.org), при этом продемонстрирована их эффективность в процессе разработки, гибкость к изменяющимся требованиям заказчика, скорость разработки, унификация архитектуры и терминологии предметной области.

1. Мартин Фаулер. Архитектура корпоративных программных приложений - М.: «Вильямс», 2012. - 544 с.
2. Мартин Р., Мартин М. Принципы, паттерны и методики гибкой разработки на языке C#. - Пер. с англ. - СПб.: Символ-Плюс, 2011. - 768 с.
3. CQRS Documents by Greg Young: https://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf
4. CQRS by Martin Fowler: http://martinfowler.com/bliki/CQRS.html
5. Dino Esposito. Cutting Edge - CQRS for the Common Application: https://msdn.microsoft.com/en-us/magazine/mt147237.aspx

© Горшунов А.І.