Command Query Responsibility Segregation

ИЛИ ПРОСТО ЦЭКУЭРЭС

Но для начала выкинем R из CQRS

CQS – Command Query Separation, разделение команд и запросов

Основная идея CQS заключается в том, что любые методы могут быть только двух типов:

- Queries – возвращают результат, не изменяя состояние объекта

- Commands - изменяют состояние объекта, не возвращая значение

Не-CQS-ный код

```
public class User
{
    1 reference
    public string Email { get; private set; }

    0 references
    public bool IsEmailValid(string email)
    {
        bool isMatch = Regex.IsMatch("some email pattern", email);

        if (isMatch)
            Email = email; // Command

        return isMatch; // Query
    }
}
```

Превращается в CQS-ный код

```
public class User
{
    1 reference
    public string Email { get; private set; }

    1 reference
    public bool IsEmailValid(string email) // Query
    {
        return Regex.IsMatch("some email pattern", email);
    }

    0 references
    public void ChangeEmail(string email) // Command
    {
        if (IsEmailValid(email))
            throw new ArgumentOutOfRangeException(email);

        Email = email;
    }
}
```

А вот теперь CQRS

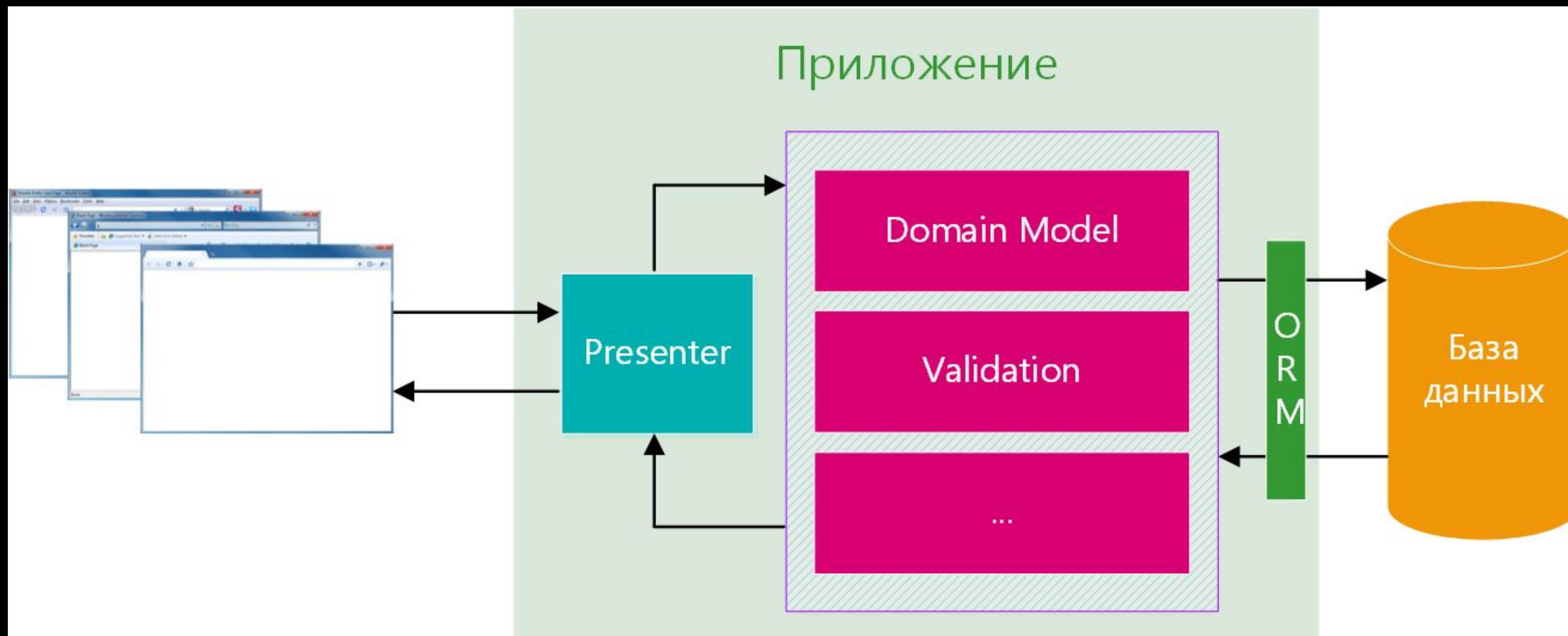
CQRS – Command Query Responsibility Segregation, разделение ответственности на команды и запросы

Та же идея, что и в CQS, но на более высоком уровне – на уровне всей системы

Для изменения состояния системы используем Command

Для выборки данных о состоянии системы используем Query

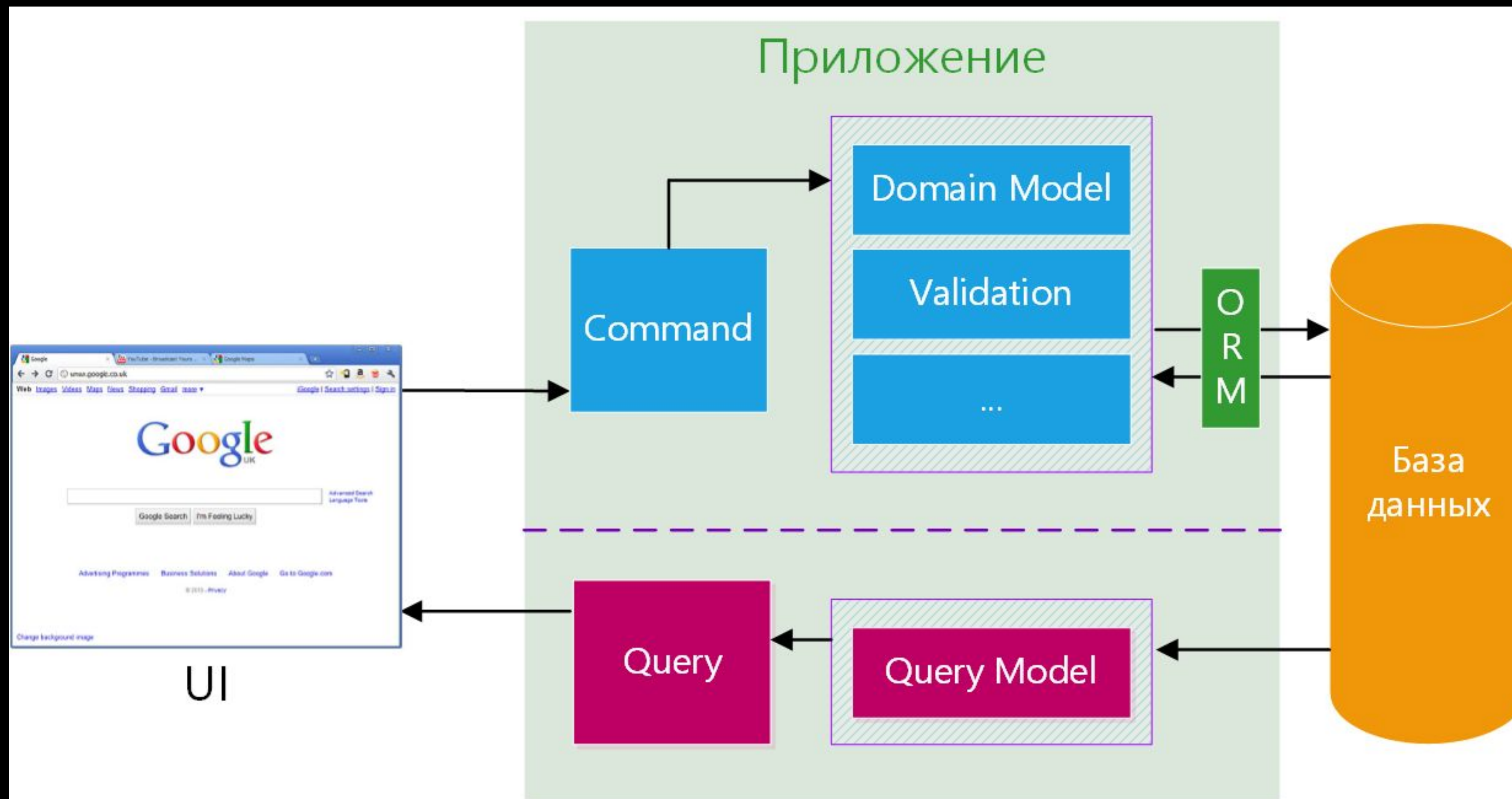
CRUD-сценарий



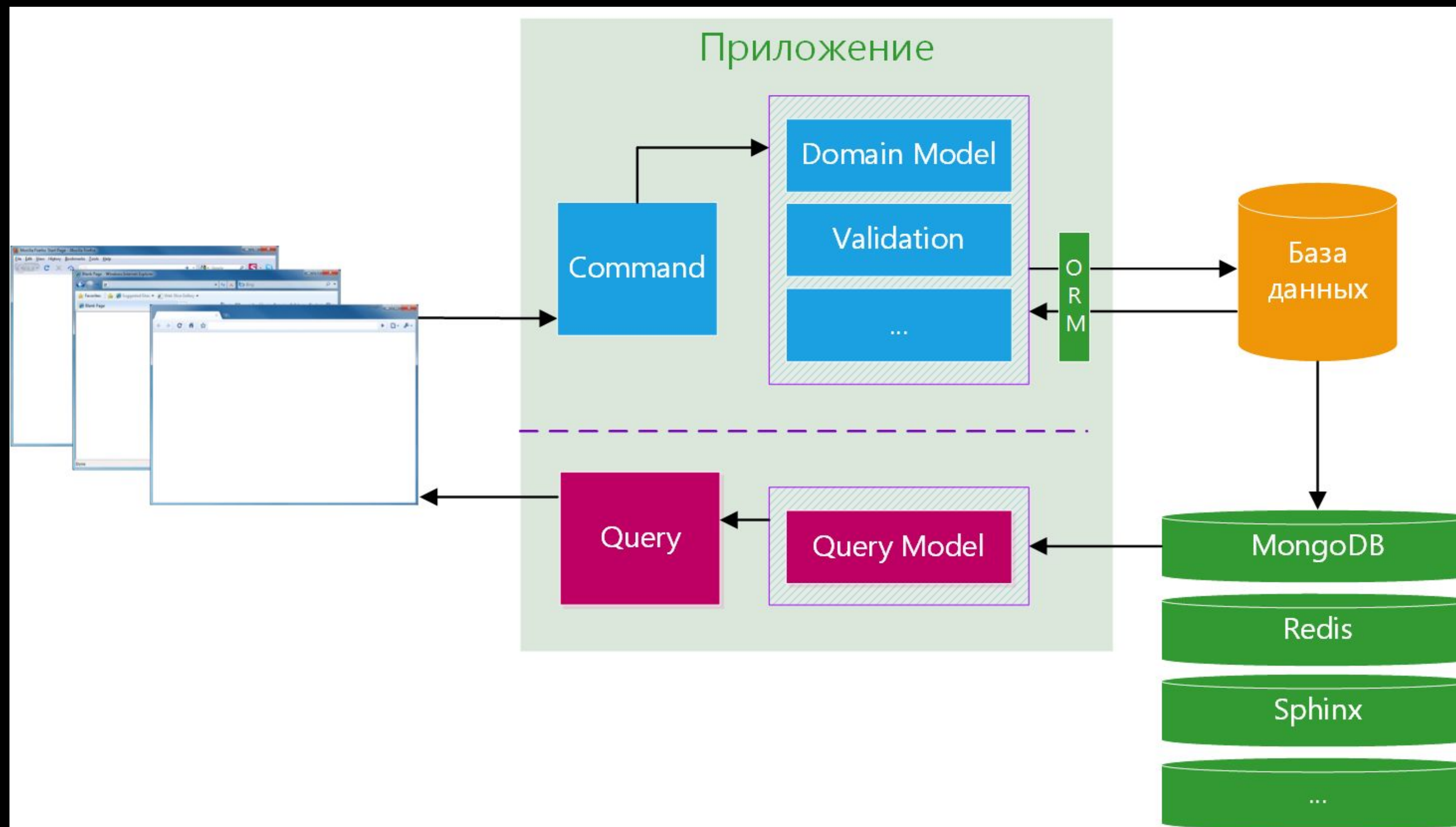
Что происходит дальше?

- У нас есть концептуальная модель, которая взаимодействует с основными объектами нашего домена
- Мы стараемся сделать наше хранилище наиболее приближенным к нашим данным
- Нам требуется выбирать и отображать все более сложно связанные данные, в том числе отчеты
- Наши объекты становятся все более сложными с большим количеством вспомогательных полей
- Вслед за объектами усложняется и хранилище
- Появляется лишнее для команд, нужное только для запросов

CQRS-сценарий



CQRS-сценарий с разными хранилищами



Эволюция команд и запросов на практике
Акт первый

```
public interface IRepository<TEntity>
{
    0 references
    void Create(TEntity entity);

    0 references
    TEntity Get(int id);

    0 references
    void Update(TEntity entity);

    0 references
    void Delete(TEntity entity);
}
```

Эволюция команд и запросов на практике
Акт второй

```
public class AccountRepository : IRepository<Account>
{
    0 references
    public IEnumerable<Account> GetActiveAccounts()
    {
        // ...
    }

    0 references
    public void ChangeAccountAddress(int id, string newAddress)
    {
        // ...
    }

    0 references
    public IEnumerable<Account> GetPremiumAccountsByManager()
    {
        // ...
    }
}
```

Эволюция команд и запросов на практике
Акт третий

```
public interface ISpecification<in TEntity>
{
    1 reference
    Func<TEntity, bool> IsSatisfiedBy();
}

0 references
public class ActiveAccountSpecification : ISpecification<Account>
{
    1 reference
    public Func<Account, bool> IsSatisfiedBy()
    {
        return x => x.IsActive && x.Credit > 0;
    }
}

0 references
public class AccountRepository : IRepository<Account>
{
    0 references
    public IEnumerable<Account> GetAccounts(ISpecification<Account>[] specifications)
    {
        // ...
    }
}
```

Эволюция команд и запросов на практике
Акт четвертый

```
public class FindPremiumAccountsByManagerQuery : IQuery<FindPremiumAccountsByManagerContext, User>
{
    private readonly ISession _session;

    0 references
    public FindPremiumAccountsByManagerQuery(ISession session)
    {
        _session = session;
    }

    0 references
    public User Ask(FindPremiumAccountsByManagerContext context)
    {
        return _session.Query<User>()…;
    }
}
```

Эволюция команд и запросов на практике
Занавес

- Меньше зависимостей в каждом классе
- Соблюдается принцип единственной ответственности
- Маленький класс проще заменить
- Маленький класс проще тестировать
- В целом дизайн кода однотипный и понятный
- При расширении функциональности системы сложность дизайна растет почти линейно

CQRS может быть на любом уровне

Тенденция рефакторинга:

- Растет количество классов
- Растет количество методов в каждом классе
- Растет количество зависимостей каждого класса
- Разбиваем их на Command и Query

Вместо больших классов с множеством зависимостей мы движемся в сторону большого количества маленьких однотипных классов, каждый из которых отвечает за единственное бизнес-правило

Наш недо-CQRS

```
public interface ICommandContext
{
}
```

```
public interface ICommand<in TCommandContext>
    where TCommandContext : ICommandContext
{
    3 references
    void Execute(TCommandContext commandContext);
}
```

Наш недо-CQRS

```
public interface ICommandFactory
{
    1 reference
    ICommand<TCommandContext> Create<TCommandContext>()
        where TCommandContext : ICommandContext;
}
```

```
public interface ICommandBuilder
{
    2 references
    void Execute<TCommandContext>(TCommandContext commandContext)
        where TCommandContext : ICommandContext;
}
```

Наш недо-CQRS

```
public class CommandBuilder : ICommandBuilder
{
    private readonly ICommandFactory _factory;



    0 references
    public CommandBuilder(ICommandFactory factory)
    {
        _factory = factory;
    }



    2 references
    public void Execute<TCommandContext>(TCommandContext commandContext)
        where TCommandContext :  ICommandContext
    {
        _factory.Create<TCommandContext>().Execute(commandContext);
    }
}
```

Наш недо-CQRS

```
public interface ICriterion
{
}
```

```
public interface IQuery<in TCriterion, out TResult>
    where TCriterion : ICriterion
{
    13 references
    TResult Ask(TCriterion criterion);
}
```

```
public interface IQueryFactory
{
    1 reference
    IQuery<TCriterion, TResult> Create<TCriterion, TResult>()
        where TCriterion : ICriterion;
}
```

Наш недо-CQRS

```
public interface IQueryFor<out TResult>
{
    11 references
    TResult With<TCriterion>(TCriterion criterion)
        where TCriterion : ICriterion;
}
```

```
public interface IQueryBuilder
{
    10 references
    IQueryFor<TResult> For<TResult>();
}
```

Наш недо-CQRS

```
public class QueryFor<TResult> : IQueryFor<TResult>
{
    private readonly IQueryFactory _factory;


    0 references
    public QueryFor(IQueryFactory factory)
    {
        _factory = factory;
    }

    11 references
    public TResult With<TCriterion>(TCriterion criterion)
        where TCriterion : ICriterion
    {
        return _factory.Create<TCriterion, TResult>().Ask(criterion);
    }
}
```

Как это выглядит в бою?

```
[HttpGet]
1 reference
public IActionResult List()
{
    IEnumerable<Certificate> certificates = _queryBuilder
        .For<IEnumerable<Certificate>>()
        .With(
            new FindByUserId
            {
                UserId = _userContextProvider.UserContext.User.Id
            });

    return View(_mapper.Map<IEnumerable<CertificateViewModel>>(certificates));
}
```

Как это выглядит в бою?

```
public void Execute(CreateBillForm form)
{
    LegalPerson legalPerson = _queryBuilder.Get<LegalPerson>(form.LegalPersonId);

    AccountOrder accountOrder = _accountOrderGenerator.Create(legalPerson, form.Amount);

    _commandBuilder.Execute(
        new CreateAccountOrderCommandContext
        {
            AccountOrder = accountOrder
        });

    form.AccountOrderId = accountOrder.Id;
}
```

Почему недо-CQRS?

Работа по большей части с CRUD

При выполнении команды хочется тут же получить какой-то результат и использовать его для отрисовки UI

После добавления элемента часто необходимо перенаправить пользователя на страницу с только что добавленным элементом

Мы изменяем поля CommandContext-ов, мы сожалеем