

Архитектуры программ

MVVM



Понятие архитектура

- ▶ *Архитектура программного обеспечения* (software architecture) – представление, которое даёт информацию о **компонентах** ПО, **обязанностях** отдельных компонентов и правилах организации **связей** между компонентами.
- ▶ Архитектура – это высокоуровневая модель системы.

Архитектурный стиль (шаблон)

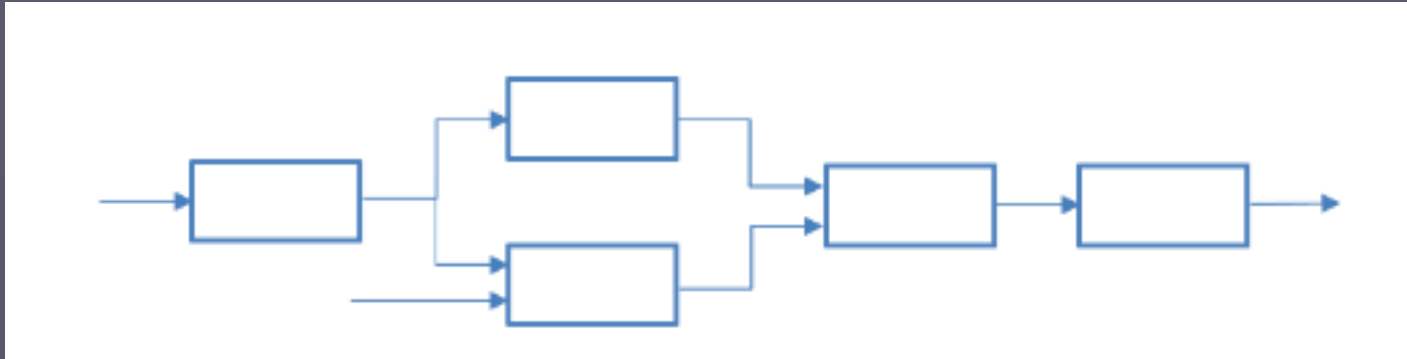
- ▶ Принципы, используемые в конкретной архитектуре, формируют *архитектурный стиль*.
- ▶ Архитектурный стиль подобен паттерну проектирования, но не на уровне модуля или класса, а на уровне всей создаваемой системы.

Группы архитектурных стилей

- ▶ Стили группируют по **фокусу применения**:

Группа	Примеры архитектур
Связь	Шина сообщений; сервис-ориентированная
Развёртывание	Монолитная; клиент-серверная; многозвенная
Структура	Компонентная; многоуровневая
. . .	

► «Каналы и фильтры» (Pipes and Filters)



Каждый поток обработки данных – это серия чередующихся фильтров и каналов, начинающаяся источником данных и заканчивающаяся их потребителем. Каналы обеспечивают передачу данных и синхронизацию. Фильтр же принимает на вход данные и обрабатывает их, трансформируя в некое иное представление, а затем передает дальше.

Монолитная архитектура

- ▶ Типичное приложение середины 1990-х: **набор форм.** (Почти) вся логика приложения сосредоточена в обработчиках событий (антипаттерн *Smart UI*). Редко – выделенные библиотеки с кодом.
- ▶ С точки зрения развертывания – это **монолитная архитектура** (т. е. физически приложение представлено одним файлом).

Компонентная архитектура

- ▶ Из приложения выделяются отдельные, относительно независимые части, называемые **компонентами**. Компоненты пригодны для повторного использования.
- ▶ Меняется подход к созданию приложений. Теперь это выделение и организация связей между компонентами.

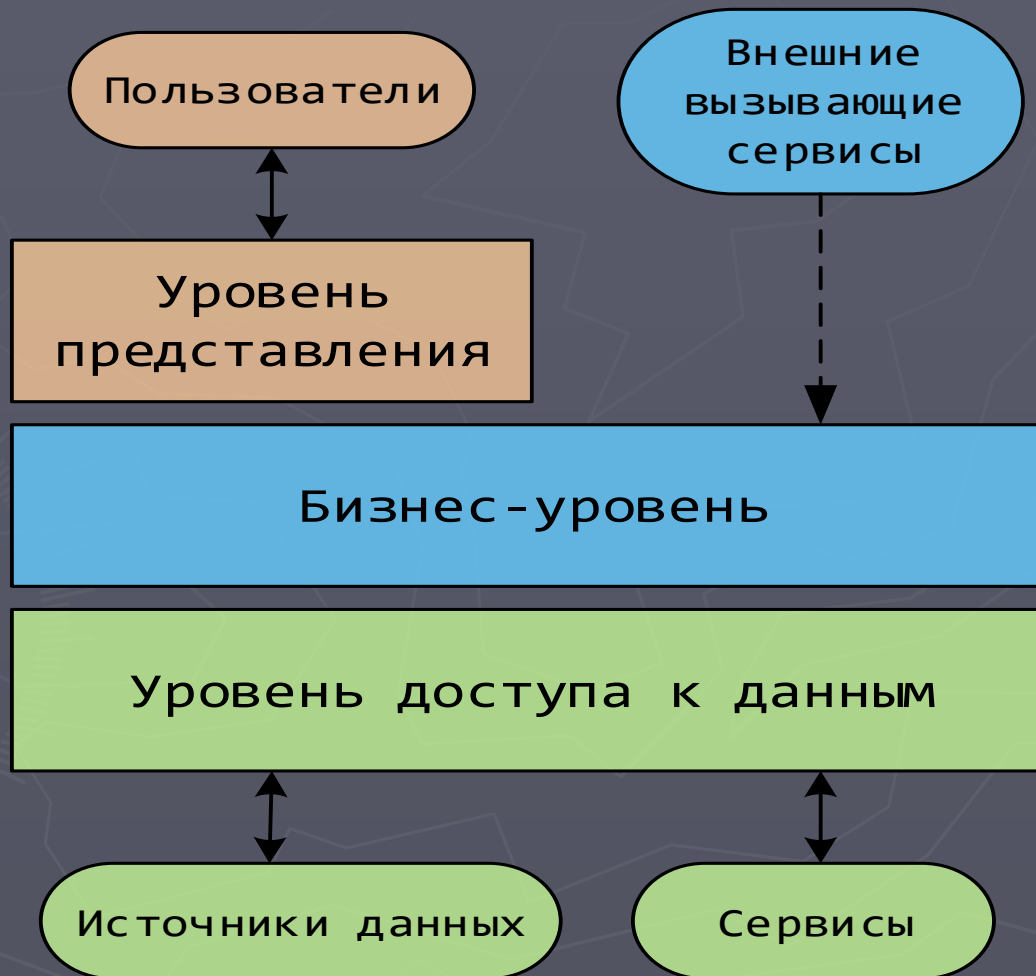
Клиент-серверная модель

- ▶ *Клиент-серверная модель* (client-server model) описывает отношение между двумя компьютерными программами, в котором одна программа – **клиент** – выполняет запросы к другой программе – **серверу**.
- ▶ Эта архитектурная модель, в основном, решает задачу развёртывания приложения.

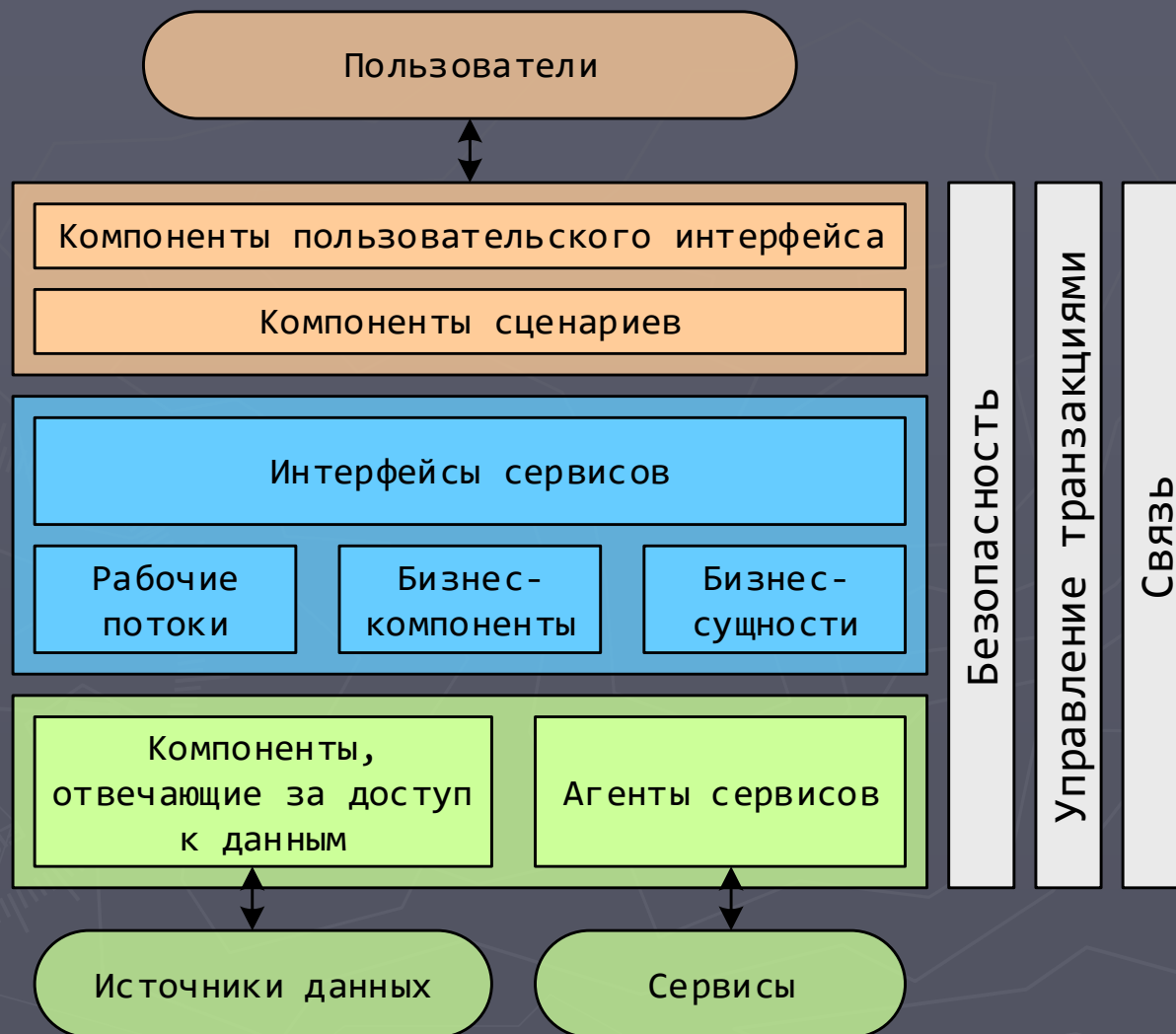
Многоуровневая архитектура

- ▶ *Многоуровневая архитектура* (multilayered architecture) базируется на следующих принципах:
 - Каждая часть системы соотносится с определённым **уровнем** (layer).
 - Для каждого уровня заданы выполняемые им функции.
 - Уровни выстроены в **стековую структуру**.
 - Нижние уровни независимы от верхних уровней.
 - Верхние уровни вызывают функции нижних уровней, при этом взаимодействуют только соседние уровни иерархии.

Трёхуровневая архитектура



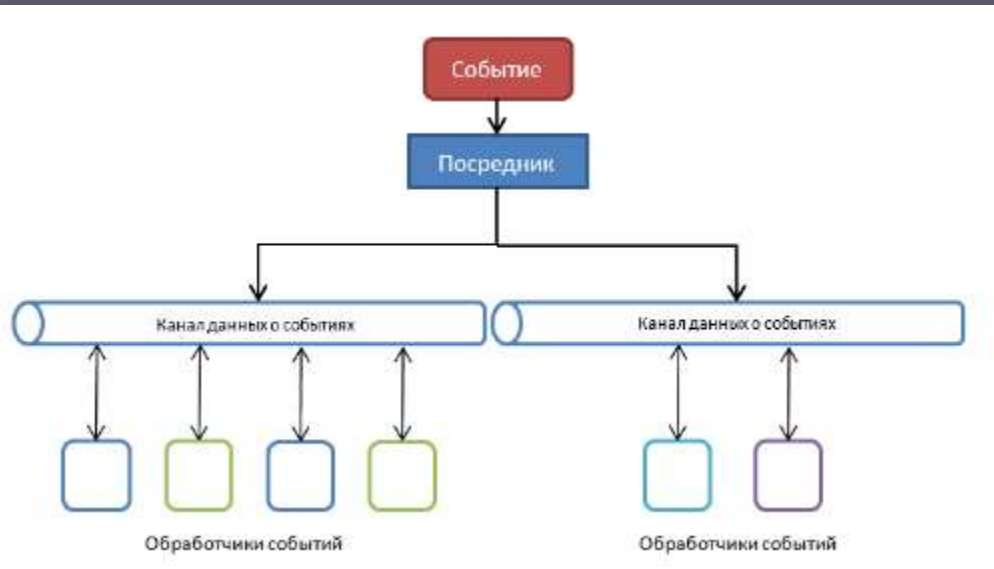
Трёхуровневая архитектура



Многозвенная архитектура

- ▶ *Многозвенная архитектура* (multitier architecture) – это стиль развёртывания приложений.
Подразумевает разделение компонентов на функциональные группы.
- ▶ Группа формируют **звено** (tier) – часть приложения, которая **физически** обособлена, выполняется в отдельном процессе или на отдельном физическом компьютере.

Архитектура, управляемая событиями - EDA

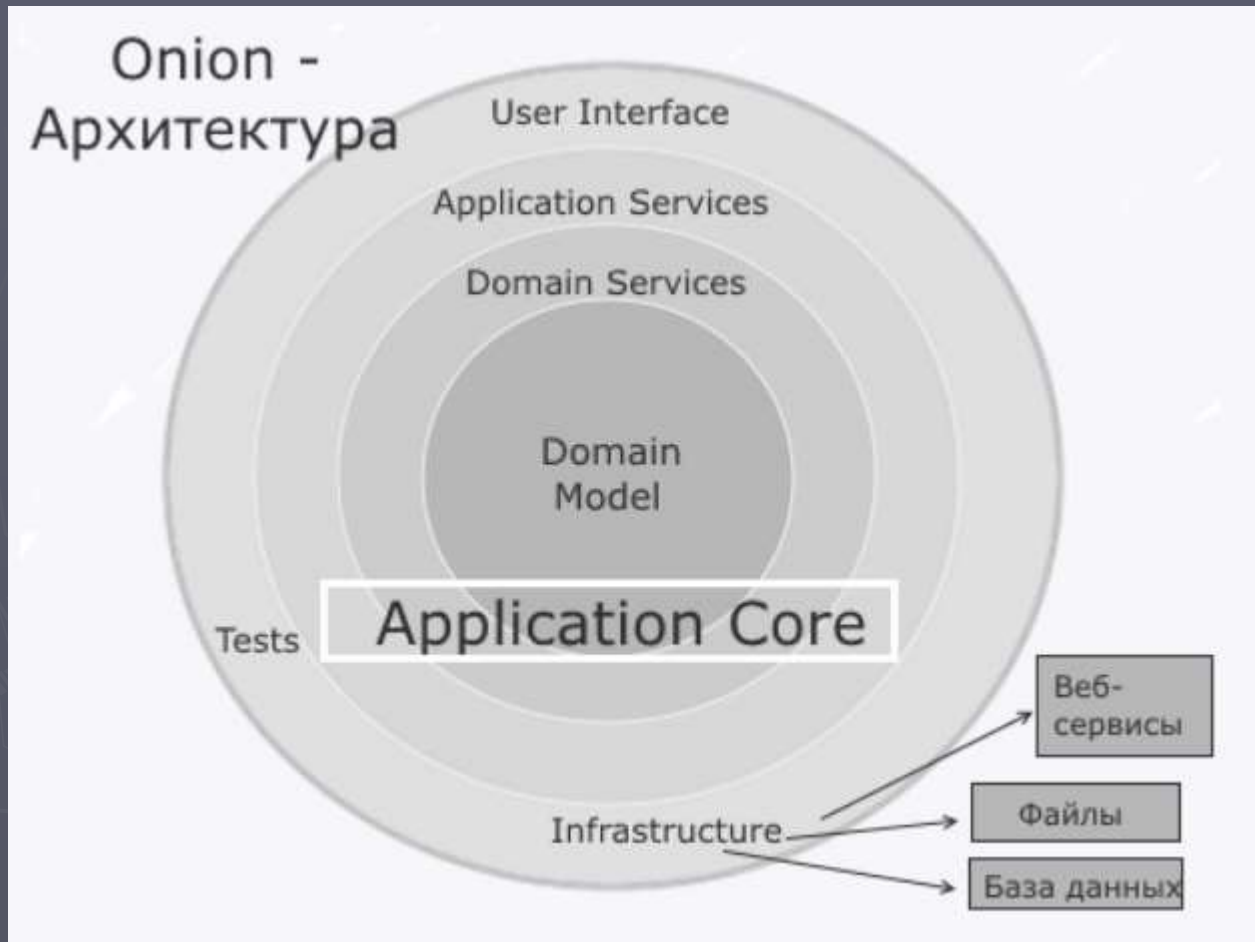


Обработчики являются изолированными независимыми компонентами, отвечающими (в идеале) за какую-нибудь одну задачу, и содержат бизнес-логику, необходимую для работы.

Микросервисная архитектура

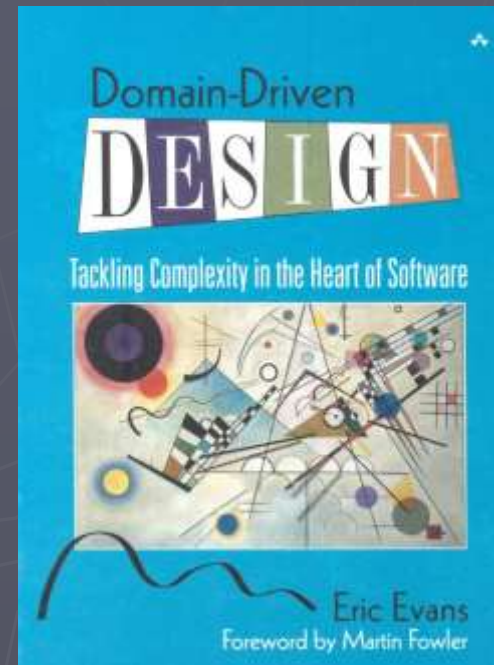
- ▶ Каждый микросервис включает в себя бизнес-логику и представляет собой совершенно независимый компонент.
- ▶ Сервисы одной системы могут быть написаны на различных языках программирования и общаться друг с другом, используя различные протоколы.

Onion-архитектура



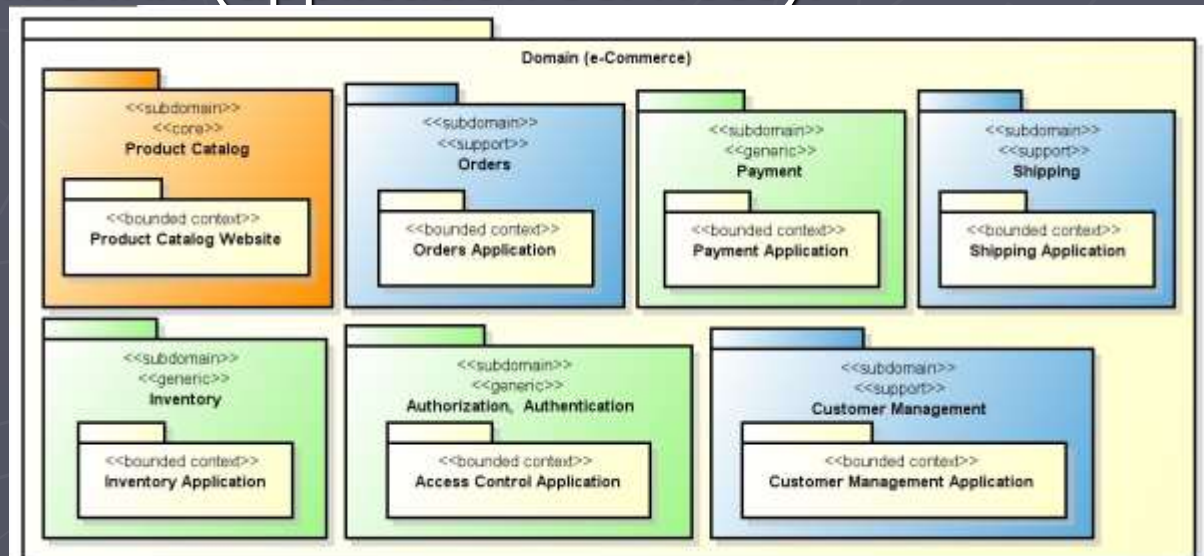
Domain-driven design (DDD)

- ▶ *Проектирование, основывающееся на домене* (domain-driven design) – вариант объектно-ориентированной архитектуры.
- ▶ Основной принцип: проектируемая система – это набор взаимодействующих объектов, описывающих предметную область.

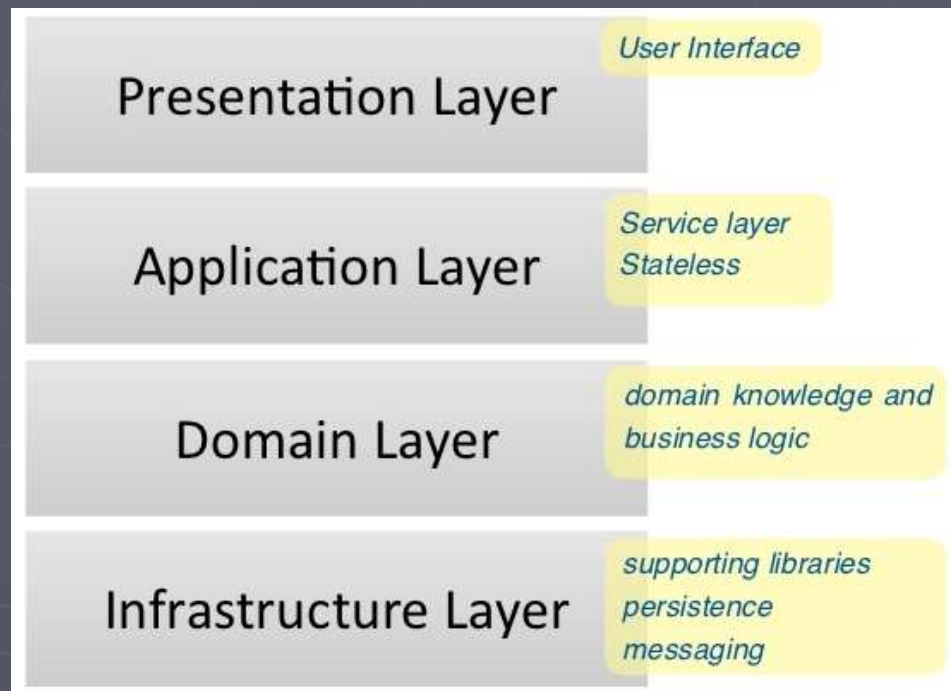


Терминология Domain-driven design

- Сущности (entities).
- Объекты-значения (value objects).
- Агрегаторы (aggregates).
- Хранилища (repositories).
- Фабрики (factories).
- Доменные сервисы (domain services).
- Сервисы приложения (application services).



Уровни в Domain-driven design



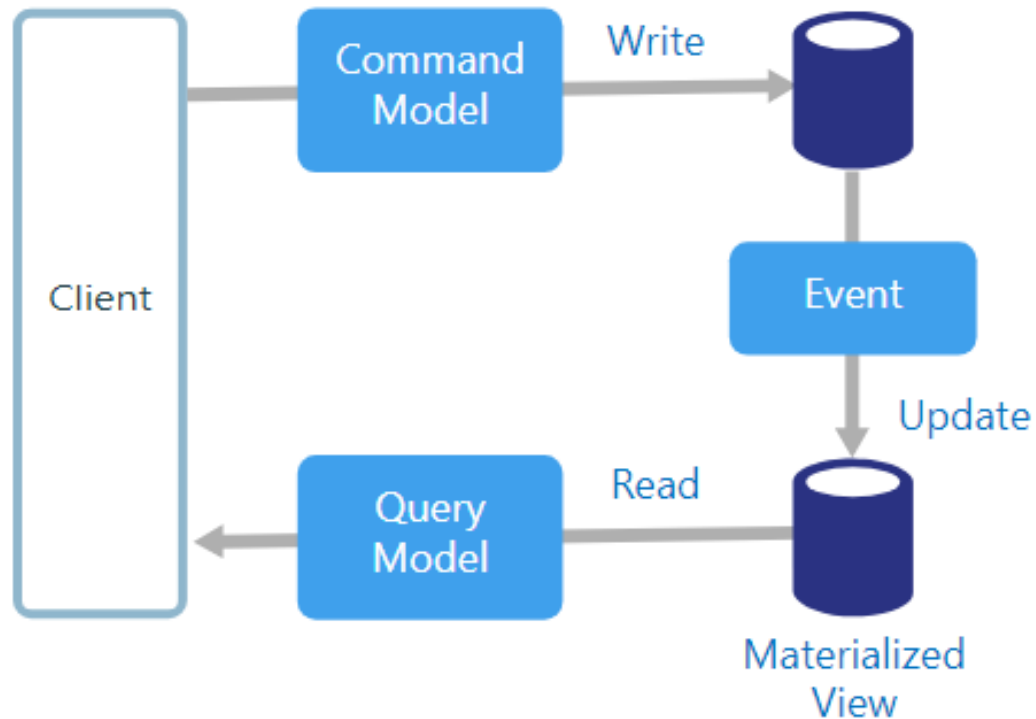
CQRS

- *Разделение ответственности на команды и запросы (Command Query Responsibility Segregation, CQRS) – в основе стиля лежит принцип использования различных объектов и потоков управления для **чтения** и **модификации** данных.*

сформулировал Грег Янг на основе принципа CQS, предложенного Бертраном Мейером

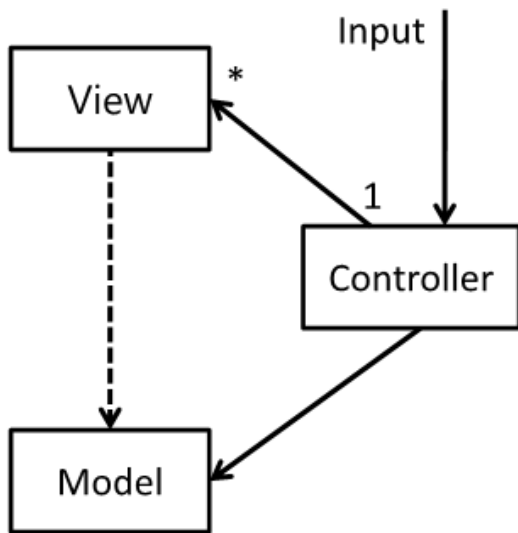
CQRS реализуется в **ограниченных контекстах** (*bounded context*) приложений, проектируемых на основе DDD

CQRS – один из вариантов

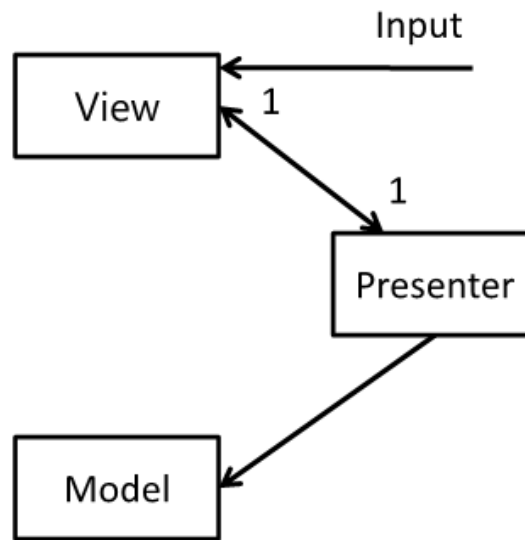


Выделенное представление

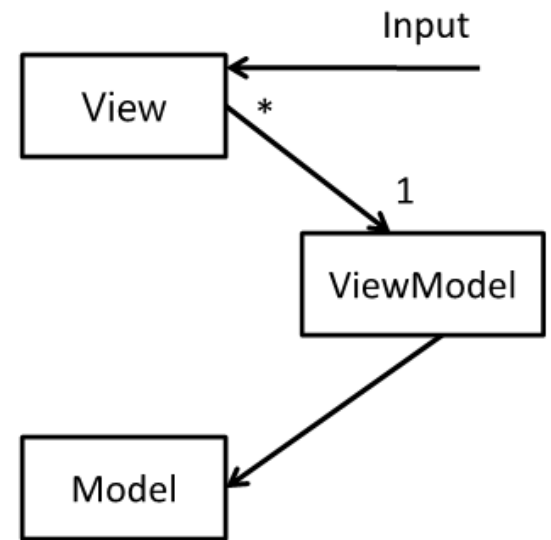
- ▶ *Выделенное представление* (separated presentation) – стиль **обработки запросов** или действий пользователя, а также манипулирования элементами интерфейса и данными. Этот стиль подразумевает отделение элементов интерфейса от логики приложения.
 - Model-View-Controller
 - Model-View-Presenter
 - Model-View-ViewModel



MVC



MVP



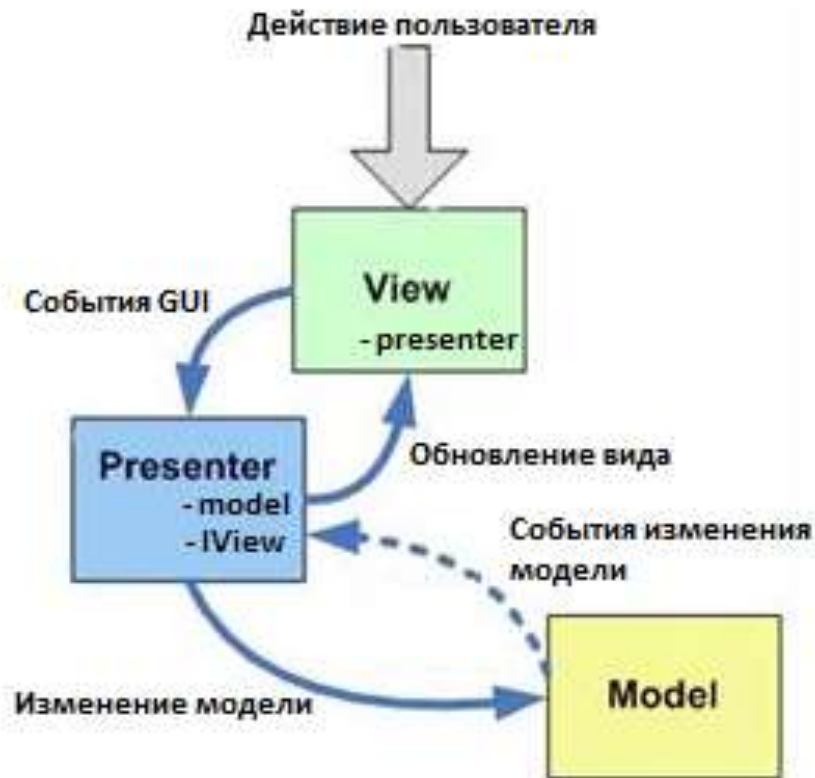
MVVM

Web

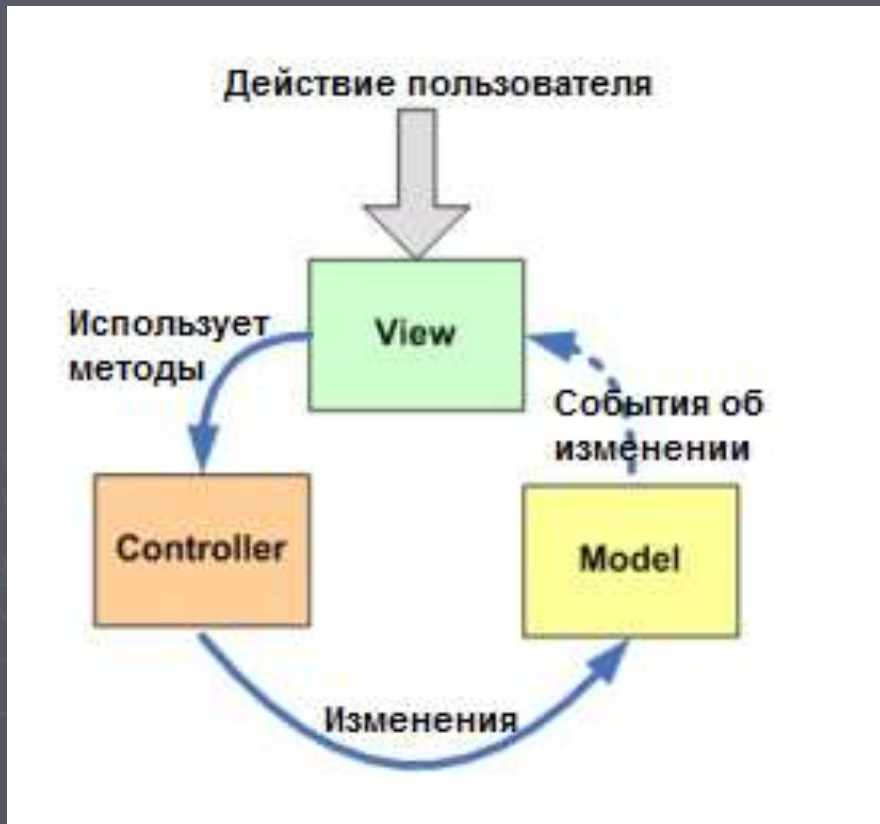
WinForm

WPF

Model-View-Presenter



Model-View-Controller



Отвечает за
данные
выборки

UI
(дизайн)

Логика работы
(программист)

MVVM (Model-View-ViewModel)—это
шаблон для разделения модели и её
представления
Для клиентских приложений

MVC / MVP

Область использования MVVM

- ▶ WPF
- ▶ Silverlight
- ▶ WinRT
- ▶ HTML5 (Knockout/Angular)
- ▶ Xamarin
- ▶ Windows 10

Назначение и преимущества

- ▶ Управляемость - разделение на уровни
 - удобство работы в команде - логика, UI
 - Обнаружение проблем
 - Быстрая замена View при сохранении ViewModel
- ▶ Тестируемость - написание автоматизированных тестов (unit test)
- ▶ Расширяемость
 - ▶ Быстрая замена View при сохранении ViewModel
 - архитектура

Пример

Доступ к UI

```
private void ComputeCustomerOrdersTotal(object sender, RoutedEventArgs e)
{
    var selectedCustomer = this.customerDataGrid.SelectedItem as Customer;

    var orders = (from order in dbContext.Orders.Include("OrderItems")
                  where order.CustomerId == selectedCustomer.Id select order);
    var sum = 0;
    foreach (var order in orders)
    {
        foreach (var item in order.OrderItems)
        {
            sum += item.UnitPrice * item.Quantity;
        }
    }
    this.customerOrderTotal.Text = sum.ToString();
}
```

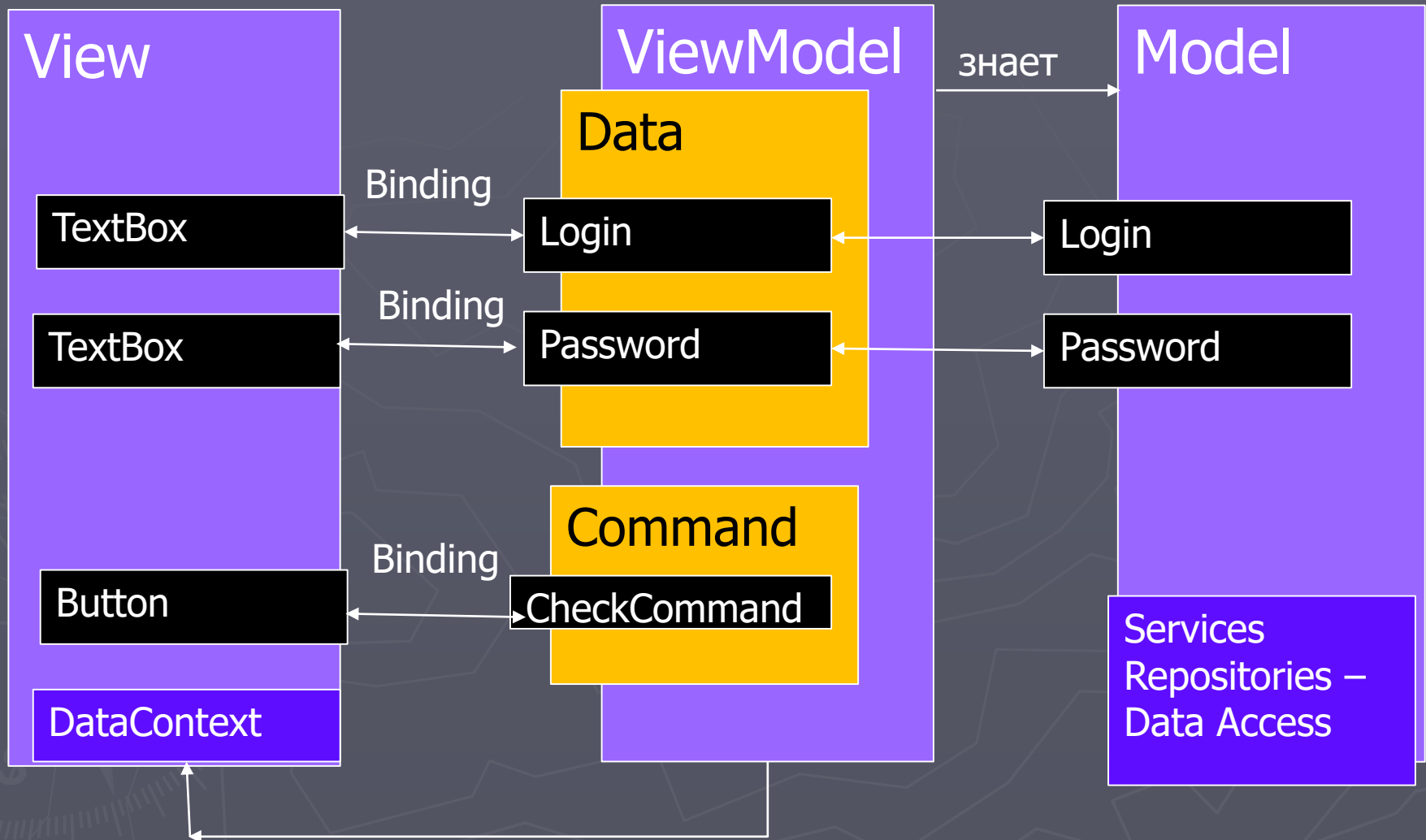
Доступ к
данным

Управление

Доступ к UI

Архитектура

INotifyPropertyChanged



Model Репозитории

- ▶ Содержат клиентские данные
- ▶ Определяют свойства
- ▶ Валидация данных



ViewModel

- ▶ Обеспечивают View данными
- ▶ Логика взаимодействия
 - Вызов данных, бизнес –классов, методов, сервисов
 - Логика управления

```
public Student Student {get;set}
```

```
//обращение
```

```
public ObservableCollection<AccountItem>
```

```
    Accounts {get;set;}
```

```
//обертка
```

```
public bool LoggIn{get;set;}
```

Model

Student

Account

Detail

Сервис
Аутентификации

Принципы именования классов в MVVM

► View

- ИмяView – (UserView) зависит от содержимого и означает что должна быть пара ViewModel

► Model

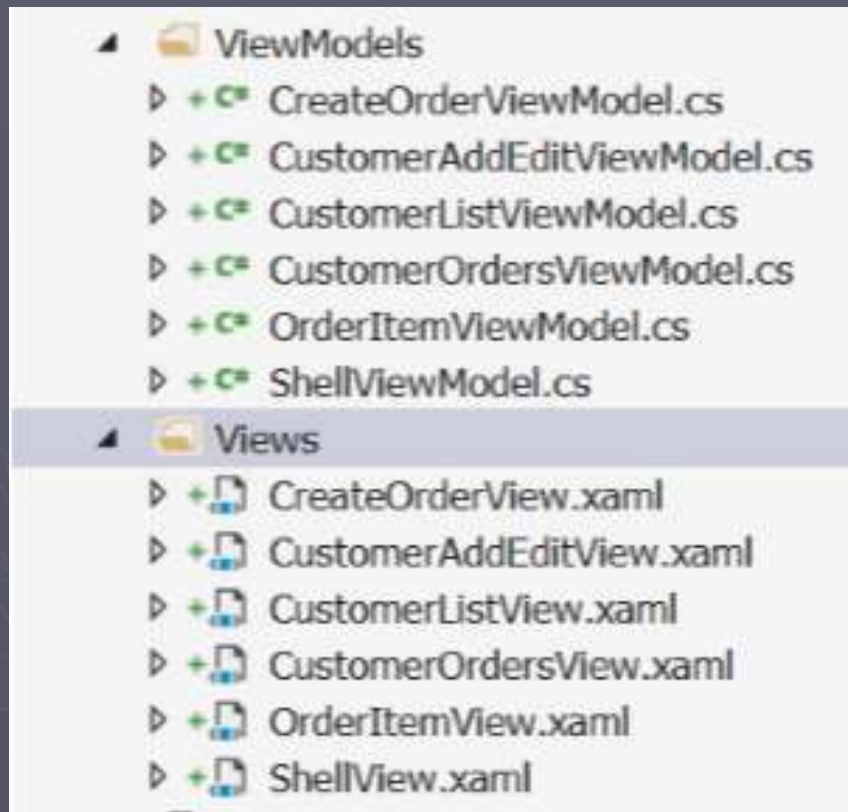
- Имя объекта данных или состояния (User)

► ViewModel

- ИмяViewModel (UserViewModel) -парно

Размещение компонентов

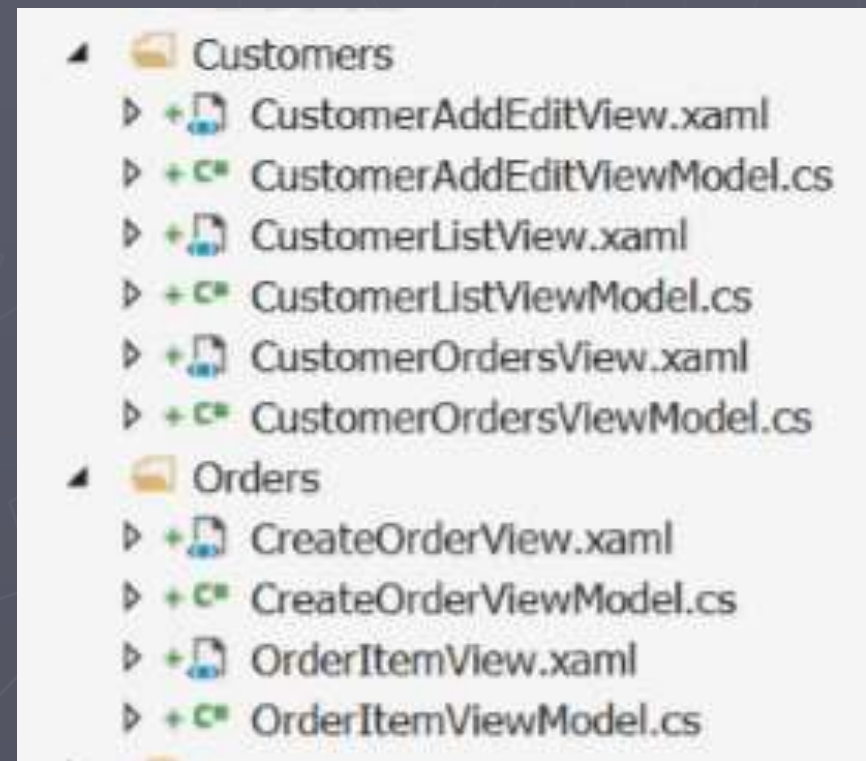
► Папки по типу



Легко находить в большом проекте

Типы модели в отдельную библиотеку классов

Папки по функциям (или по Use Case)



Решение "MVVMExample"

HRDD.Data

MVVMExample

Подходы к проектированию

► View First

- Определение в Xaml или Code Behind DataContext

```
<UserControl.DataContext>  
    <local:CustomerEditViewModel />  
</UserControl.DataContext>
```

```
this.DataContext = new CustomerListViewModel();
```

- ViewMode First

Пример разработки

► 1) Построение модели - EF

```
public class Customer
{
    public Customer()
    {
        Orders = new List<Order>();
    }
    [Key]
    public Guid Id { get; set; }
    public Guid? StoreId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string FullName { get { return F } }
    public string Phone { get; set; }
    public string Email { get; set; }
    public string Street { get; set; }
    public string City { get; set; }
    public string State { get; set; }
    public string Zip { get; set; }
    public List<Order> Orders { get; set; }
}
```

```
public class Product
{
    [Key]
    public int Id { get; set; }
    public string Type { get; set; }
    public string Name { get; set; }
    public string Description { get; set; }
    public string Image { get; set; }
    public bool HasOptions { get; set; }
    public bool IsVegetarian { get; set; }
    public bool WithTomatoSauce { get; set; }
    public string SizeIds { get; set; }
}
```

```
public class HRDDbContext : DbContext
{
    public DbSet<Customer> Customers { get; set; }
    public DbSet<Order> Orders { get; set; }
    public DbSet<Product> Products { get; set; }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        // Table names match entity names by default (don't pluralize)
        modelBuilder.Conventions.Remove<PluralizingTableNameConvention>();
        // Globally disable the convention for cascading deletes
        modelBuilder.Conventions.Remove<OneToManyCascadeDeleteConvention>();

        modelBuilder.Entity<Customer>()
            .Property(c => c.Id) // Client must set the ID.
            .HasDatabaseGeneratedOption(DatabaseGeneratedOption.None);
    }
}
```

► 2) Создание репозитория

```
public interface ICustomersRepository
{
    Task<List<Customer>> GetCustomersAsync();
    Task<Customer> GetCustomerAsync(Guid id);
    Task<Customer> AddCustomerAsync(Customer customer);
    Task<Customer> UpdateCustomerAsync(Customer customer);
    Task DeleteCustomerAsync(Guid customerId);
}
```

► 3) Реализация репозитория

```

public class CustomersRepository : ICustomersRepository
{
    HRDDDbContext _context = new HRDDDbContext();

    public Task<List<Customer>> GetCustomersAsync()
    {
        return _context.Customers.ToListAsync();
    }

    public Task<Customer> GetCustomerAsync(Guid id)
    {
        return _context.Customers.FirstOrDefault(c => c.Id == id);
    }

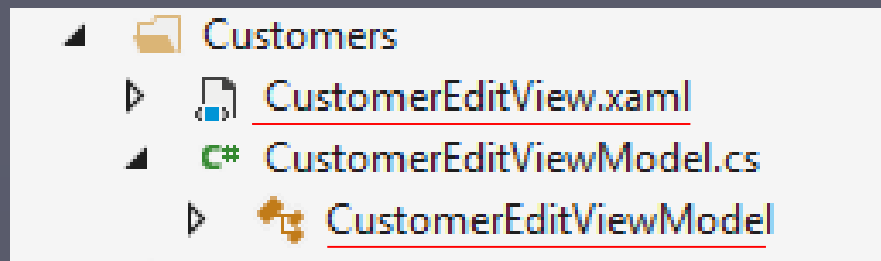
    public async Task<Customer> AddCustomerAsync(Customer customer)
    {
        _context.Customers.Add(customer);
        await _context.SaveChangesAsync();
        return customer;
    }

    //...

    public async Task DeleteCustomerAsync(Guid customerId)
    {
        var customer = _context.Customers.FirstOrDefault(c => c.Id == customerId);
        if (customer != null)
        {
            _context.Customers.Remove(customer);
        }
        await _context.SaveChangesAsync();
    }
}

```

приостановить выполнение метода до тех пор, пока эта задача не завершится // выполнение потока, в котором был вызван асинхронный метод, не прерывается



```
<UserControl.DataContext>
    <local:CustomerEditViewModel CustomerId="11DA4696-CEA3-4
</UserControl.DataContext>
<Grid>
...
    <TextBox x:Name="firstNameTextBox"
...
        Text="{Binding Customer.FirstName}"
.../>
...
    <TextBox x:Name="lastNameTextBox"
...
        Text="{Binding Customer.LastName}"
.../>
    <TextBox x:Name="phoneTextBox"
        ...
        Text="{Binding Customer.Phone}"
/>
    <Button x:Name="saveButton"
        Content="Save"
        Width="75"
        Command="{Binding SaveCommand}" />
</Grid>
```


Customers

- CustomerEditView.xaml
- CustomerEditViewModel.cs
 - CustomerEditViewModel

Манипуляции с
данными и логика
взаимодействия

```
public class CustomerEditViewModel : INotifyPropertyChanged
{
    private Customer _customer;
    private ICustomersRepository _repository = new CustomersRepository();

    public CustomerEditViewModel()
    {
        SaveCommand = new RelayCommand(OnSave);
    }

    public event PropertyChangedEventHandler PropertyChanged = delegate { };

    public Customer Customer
    {
        get { return _customer; }
        set {
            if (value != _customer)
            {
                _customer = value;
                PropertyChanged(this, new PropertyChangedEventArgs("Customer"));
            }
        }
    }

    public Guid CustomerId { get; set; }
    public ICommand SaveCommand { get; set; }
    public async void LoadCustomer()
    {
        Customer = await _repository.GetCustomerAsync(CustomerId);
    }
    private async void OnSave()
    {
        Customer = await _repository.UpdateCustomerAsync(Customer);
    }
}
```

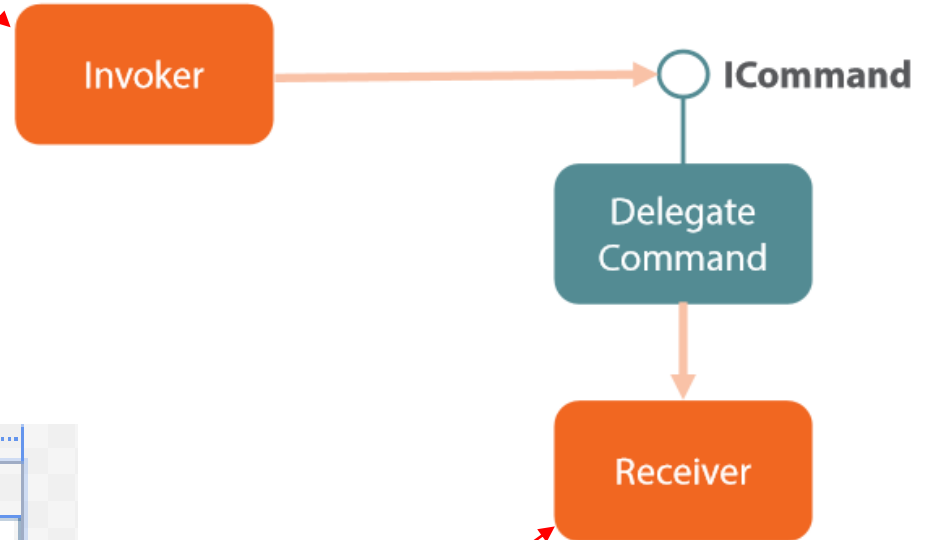
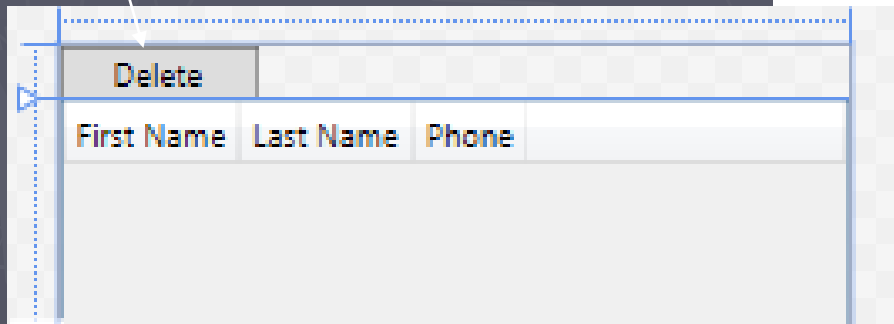
Представление
пользователя

Сохранение и
загрузка

Использование Command

View

Добавить команду
удаления



ViewModel

```
<Button Content="Delete"  
        Command="{Binding DeleteCommand}"  
        HorizontalAlignment="Left"  
        VerticalAlignment="Top"  
        Width="75" />
```

View

```
public class RelayCommand : ICommand
{
    Action _TargetExecuteMethod;
    Func<bool> _TargetCanExecuteMethod;

    public RelayCommand(Action executeMethod)
    {
        _TargetExecuteMethod = executeMethod;
    }

    public RelayCommand(Action executeMethod, Func<bool> canExecuteMethod)
    {
        _TargetExecuteMethod = executeMethod;
        _TargetCanExecuteMethod = canExecuteMethod;
    }

    public void RaiseCanExecuteChanged()
    {
        CanExecuteChanged(this, EventArgs.Empty);
    }
    #region ICommand Members
```

View

```
<DataGrid x:Name="customerDataGrid"
          AutoGenerateColumns="False"
          ItemsSource="{Binding Customers}"
          SelectedItem="{Binding SelectedCustomer}"
          Grid.Row="1">
```

ViewModel

```
public class CustomerListViewModel
{
    private ObservableCollection<Customer> _customers;
    private ICustomersRepository _repository = new CustomersRepository();

    public CustomerListViewModel()
    {
        //...
        DeleteCommand = new RelayCommand(OnDelete, CanDelete);
    }
    public RelayCommand DeleteCommand { get; private set; }

    //...
    private Customer _selectedCustomer;
    public Customer SelectedCustomer
    {
        get { return _selectedCustomer; }
        set { _selectedCustomer = value;
              DeleteCommand.RaiseCanExecuteChanged();
            }
    }
    private void OnDelete()
    { Customers.Remove(SelectedCustomer); }
    private bool CanDelete()
    { return SelectedCustomer != null; }
}
```

Property Change Notifications

- ▶ Обновление связанных данных
- ▶ DependencyProperties
- ▶ INotifyPropertyChanged(INPC)

```
public class CustomerListViewModel : INotifyPropertyChanged
{
    public ObservableCollection<Customer> Customers
    {
        get
        {
            return _customers;
        }
        set
        {
            if (_customers != value)
            {
                _customers = value;
                PropertyChanged(this, new PropertyChangedEventArgs("Customers"));
            }
        }
    }
}

// ...
public event PropertyChangedEventHandler PropertyChanged = delegate { };
}
```

ViewModel

```
public class Customer :INotifyPropertyChanged
```

```
{
```

```
//...
```

```
public string FirstName
```

```
{
```

```
    get
```

```
    {
```

```
        return _firstName;
```

```
    }
```

```
    set
```

```
    {
```

```
        if (_firstName != value)
```

```
        {
```

```
            _firstName = value;
```

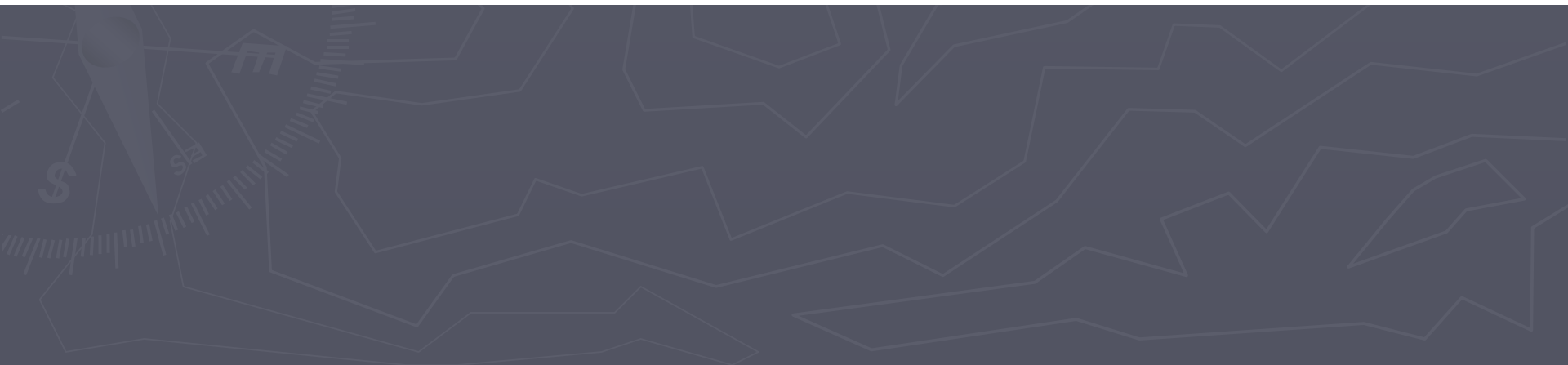
```
            PropertyChanged(this, new PropertyChangedEventArgs("FirstName"));
```

```
        }
```

```
    }
```

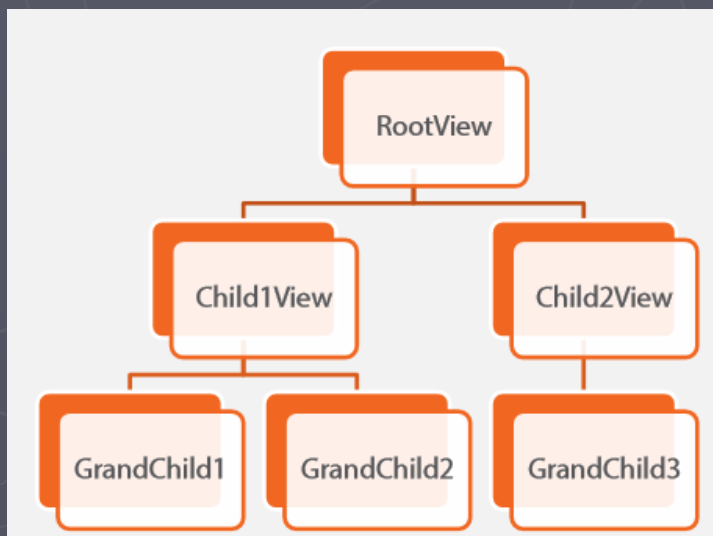
```
}
```

Model



Иерархия и управление

- ▶ Сложные View содержат дочерние вложенные View
 - У них могут быть или не быть ViewModels
- ▶ Родительские ViewModel могут конструировать дочерние ViewModel
 - Навигация
 - Данные

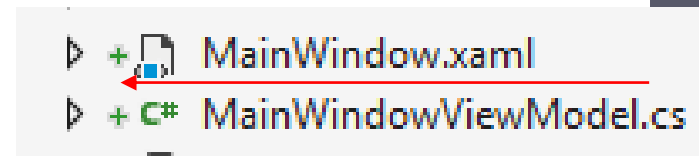
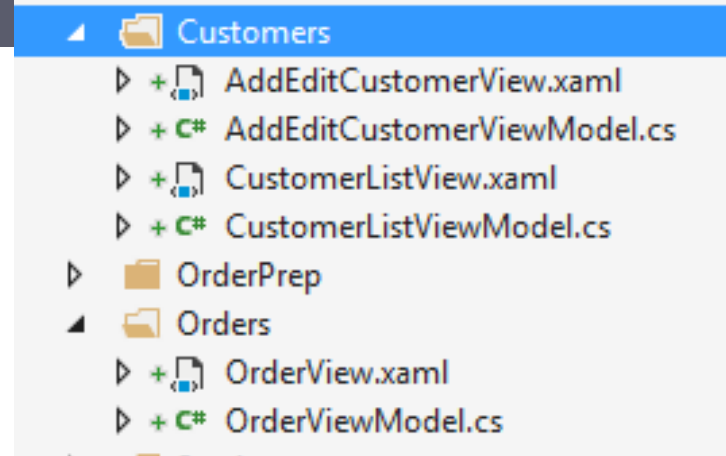


```

<Window x:Class="ZzaDesktop.MainWindow"
        xmlns:cust="clr-namespace:MVVMDemo.Customers"
        xmlns:order="clr-namespace:MVVMDemo.Orders"
        xmlns:prep="clr-namespace:MVVMDemo.OrderPrep"
        xmlns:local="clr-namespace:MVVMDemo"
        Title="MainWindow"
        Height="350"
        Width="525">
    <Window.DataContext>
        <local:MainWindowViewModel />
    </Window.DataContext>
    <Window.Resources>
        <DataTemplate DataType="{x:Type cust:CustomerListViewModel}">
            <cust:CustomerListView />
        </DataTemplate>
        <DataTemplate DataType="{x:Type order:OrderViewModel}">
            <order:OrderView />
        </DataTemplate>
        <DataTemplate DataType="{x:Type prep:OrderPrepViewModel}">
            <prep:OrderPrepView />
        </DataTemplate>
        <DataTemplate DataType="{x:Type cust:AddEditCustomerViewModel}">
            <cust:AddEditCustomerView />
        </DataTemplate>
    </Window.Resources>
    <Grid>

        <Grid x:Name="MainContent"
              Grid.Row="1">
            <ContentControl Content="{Binding CurrentViewModel}" />
        </Grid>
    </Grid>

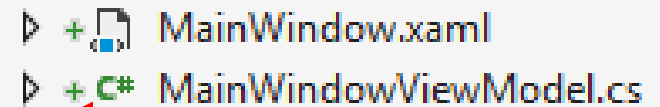
```




```
class MainWindowViewModel : BindableBase
{
    private CustomerListViewModel _customerListViewModel = new CustomerListViewModel();
    private OrderViewModel _orderViewModel = new OrderViewModel();
    private OrderPrepViewModel _orderPrepViewModel = new OrderPrepViewModel();
    private AddEditCustomerViewModel _addEditViewModel = new AddEditCustomerViewModel();

    private BindableBase _currentViewModel;

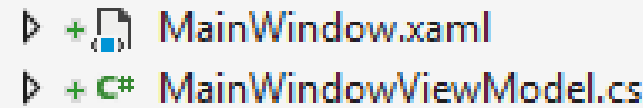
    public BindableBase CurrentViewModel
    {
        get { return _currentViewModel; }
        set { SetProperty(ref _currentViewModel, value); }
    }
}
```



▶ + [XAML icon] MainWindow.xaml
▶ + C# MainWindowViewModel.cs

Добавление команд навигации

```
<Button Content="Customers"
        Command="{Binding NavCommand}"
        CommandParameter="customers"
        Grid.Column="0" />
```



▶ + [icon] MainWindow.xaml
▶ + C# MainWindowViewModel.cs

```
public MainWindowViewModel()
{
    NavCommand = new RelayCommand<string>(OnNav);
    //...
}

public RelayCommand<string> NavCommand { get; private set; }
private void OnNav(string destination)
{
    switch (destination)
    {
        case "orderPrep":
            CurrentViewModel = _orderPrepViewModel;
            break;
        case "customers":
        default:
            CurrentViewModel = _customerListViewModel;
            break;
    }
}
```

Валидация

- ▶ Должна содержаться в Model или ViewModel, не во View
- ▶ Использование:
 - Exceptions
 - IDataErrorInfo
 - INotifyDataErrorInfo
 - ValidationRules

MVVM Toolkits / Frameworks

► Prism



► Caliburn Micro

► MVVM Light

