Архитектуры программ

Model View ViewModel (MVVM)

Понятие архитектуры

- *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.

Проектирование программного обеспечения

- Принципы **SOLID (S**ingle Responsibility, **O**pen Closed, **L**iskov substitution, **I**nterface Segregation and **D**ependency Inversion Principles**)**
- **Шаблоны проектирования**

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 (передача управления)	Компонентная; многоуровневая
. . .	

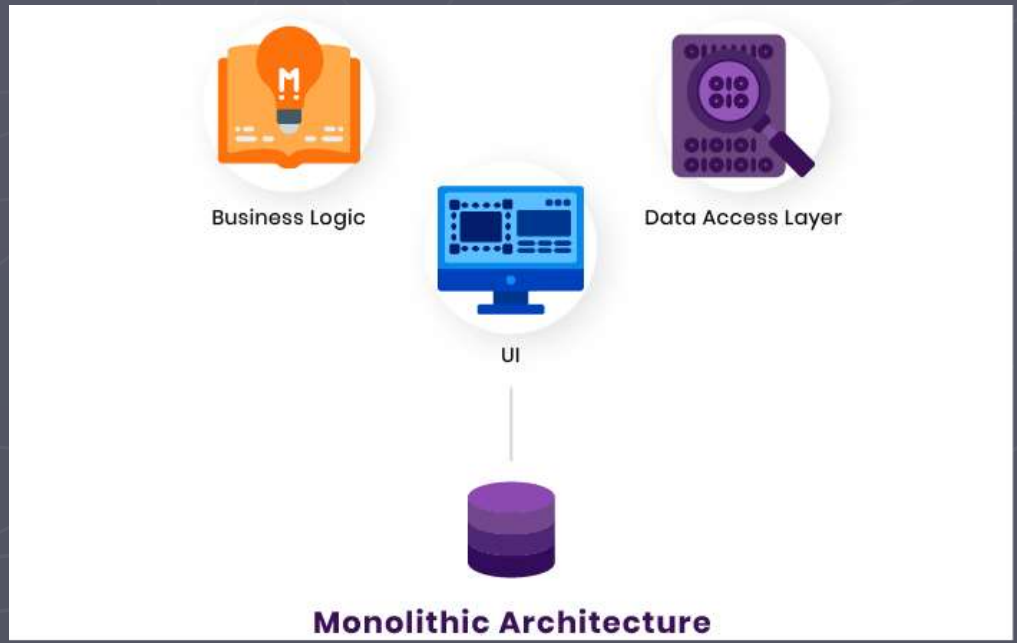
Монолитная архитектура

- 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).

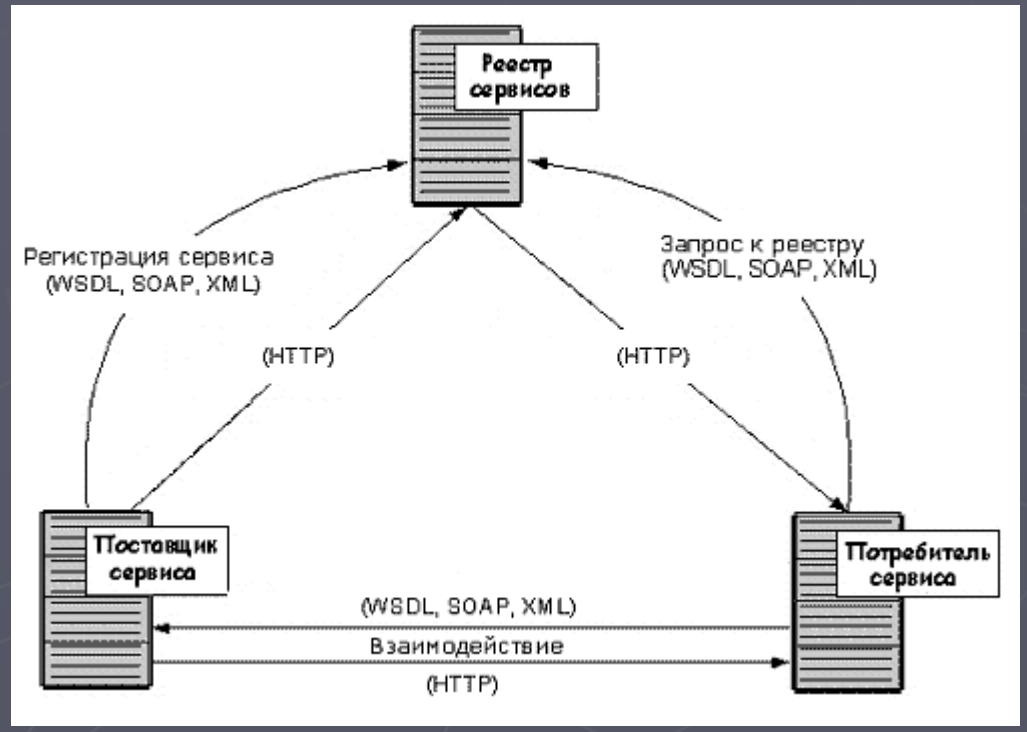
- Плюсы монолитной архитектуры
  - Упрощенная разработка и развертывание
  - Меньше сквозных проблем
  - Лучшая производительность

- Минусы монолитной архитектуры
  - Кодовая база со временем становится громоздкой
  - Сложно внедрять новые технологии
  - Ограниченная гибкость



Сервис-ориентированная архитектура (SOA)

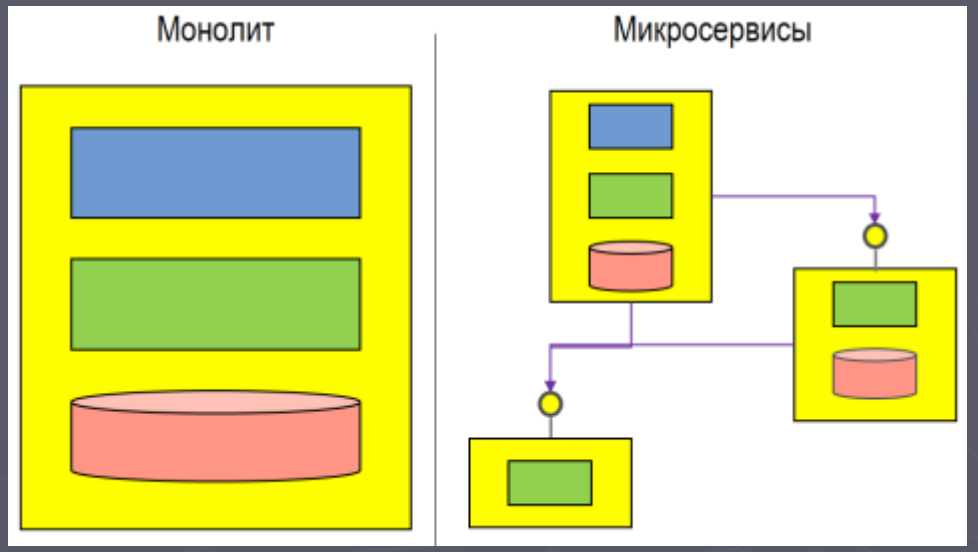
- Принципы :
  - Сочетаемость приложений, ориентированных на пользователей.
  - Многократное использование бизнес-сервисов.
  - Независимость от набора технологий.
  - Автономность (независимые эволюция, масштабируемость и развёртываемость).



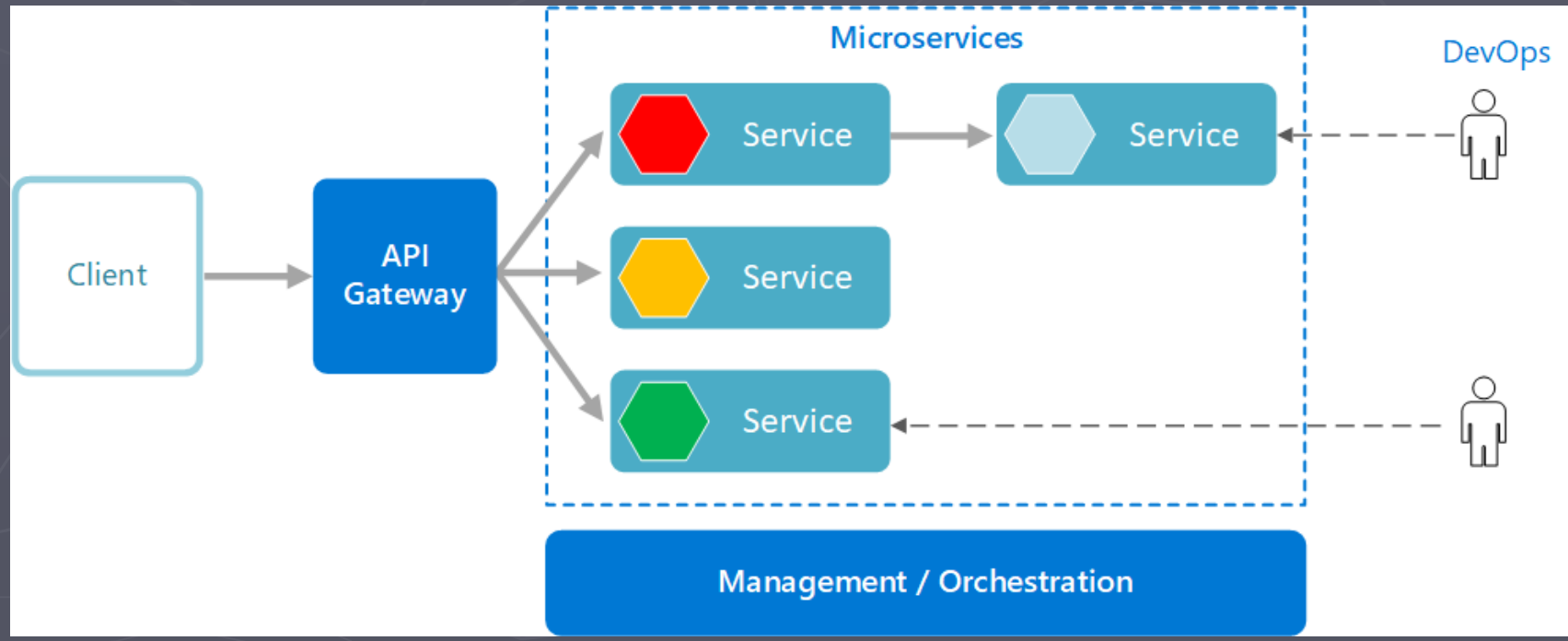
Микросервисная архитектура

Каждый микросервис:

- небольшой, независимый модульный сервис
- включает в себя бизнес-логику и представляет собой совершенно независимый компонент
- взаимодействуют через API.



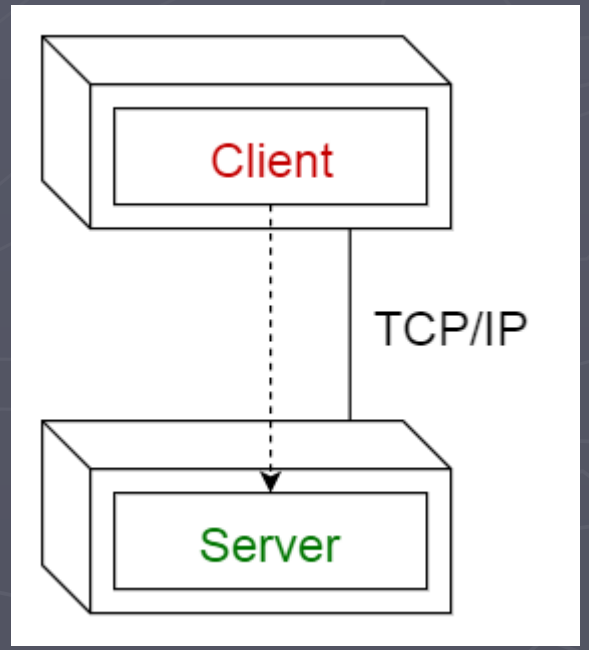
Сервисы одной системы могут быть написаны на различных языках программирования и общаться друг с другом, используя различные протоколы.



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**.

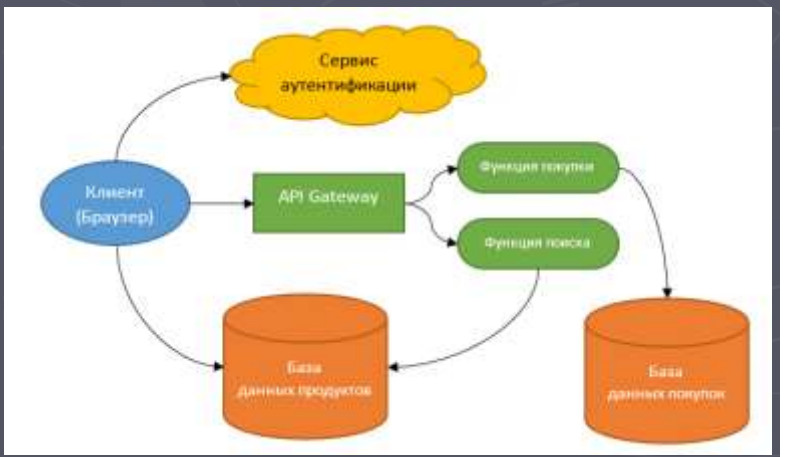
- Эта архитектурная модель, в основном, решает задачу развёртывания приложения.



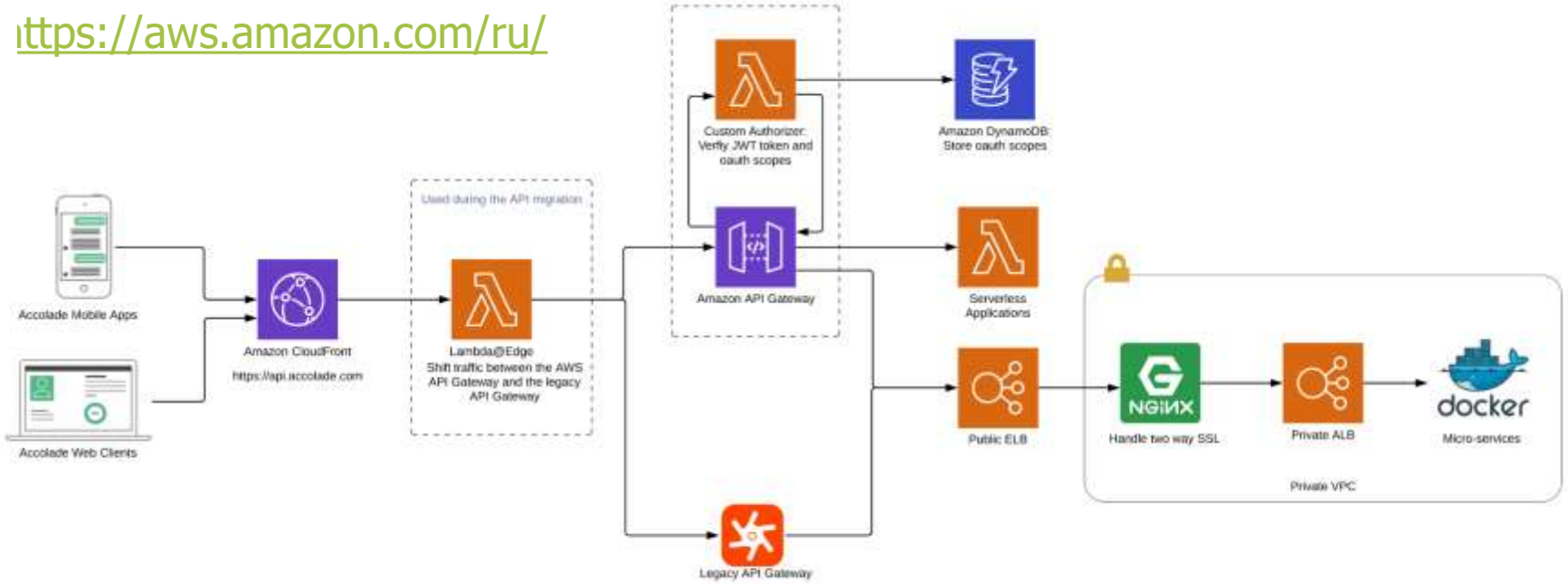
Бессерверный архитектурный стиль

- «бэкенд как услуга (BaaS)»
- «функция как услуга (FaaS)»

Бессерверные вычисления - это модель выполнения облачных вычислений, в которой поставщик облачных услуг выступает в качестве сервера, динамически управляющего распределением ресурсов машины.

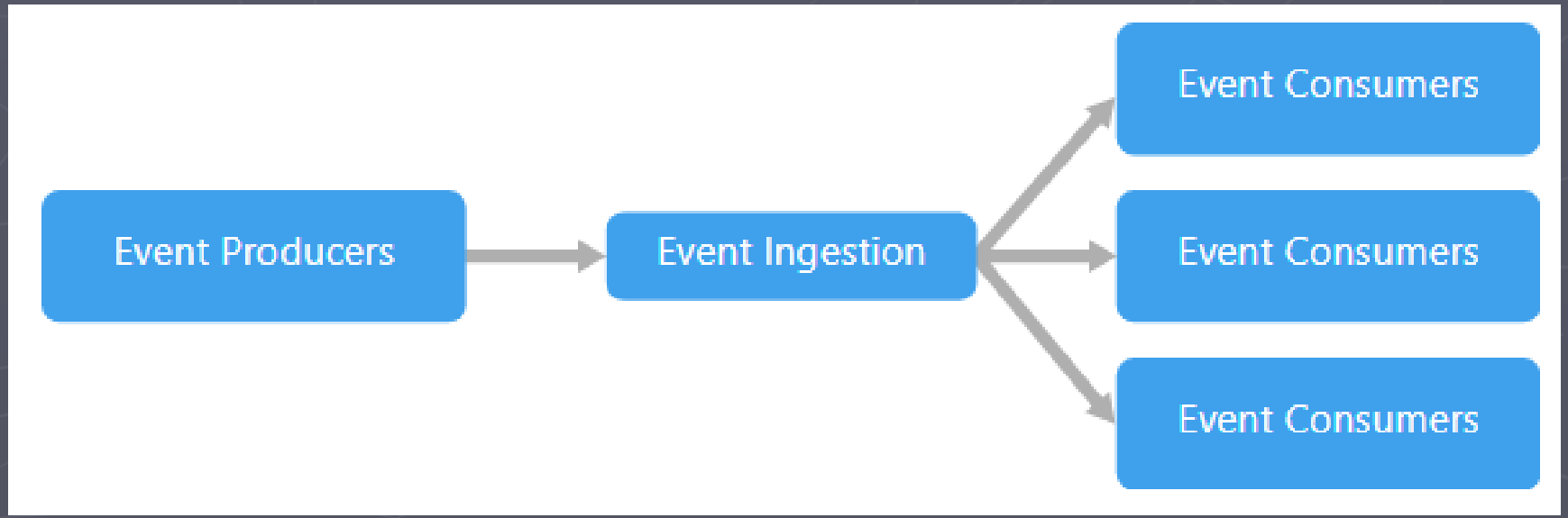


https://aws.amazon.com/ru/



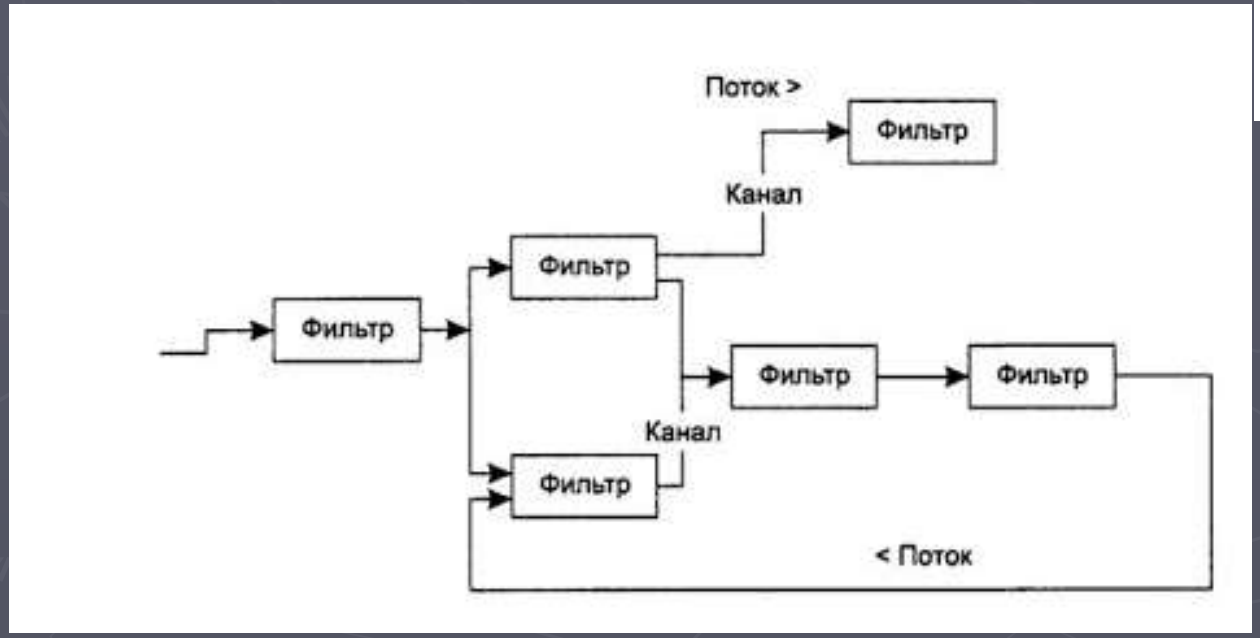
Архитектура, управляемая событиями - EDA Event-driven architecture

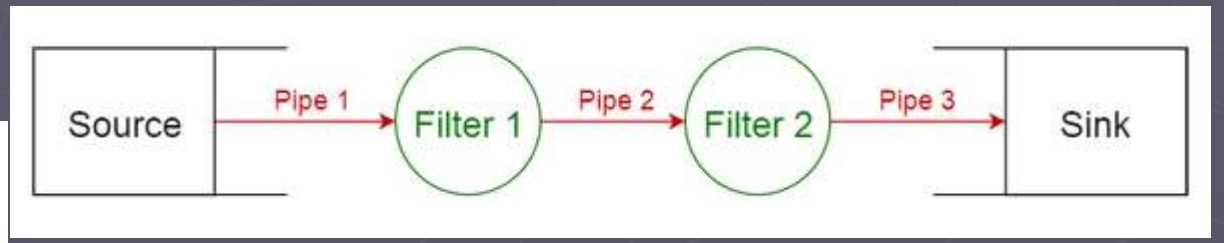
- Управляемая событиями архитектура состоит из производителей событий, которые генерируют поток событий, и потребителей событий, которые прослушивают события.
- Обработчики являются изолированными независимыми компонентами, отвечающими (в идеале) за какую-нибудь одну задачу, и содержат бизнес-логику, необходимую для работы.
- Модели
  - Pub/sub
  - Event streaming

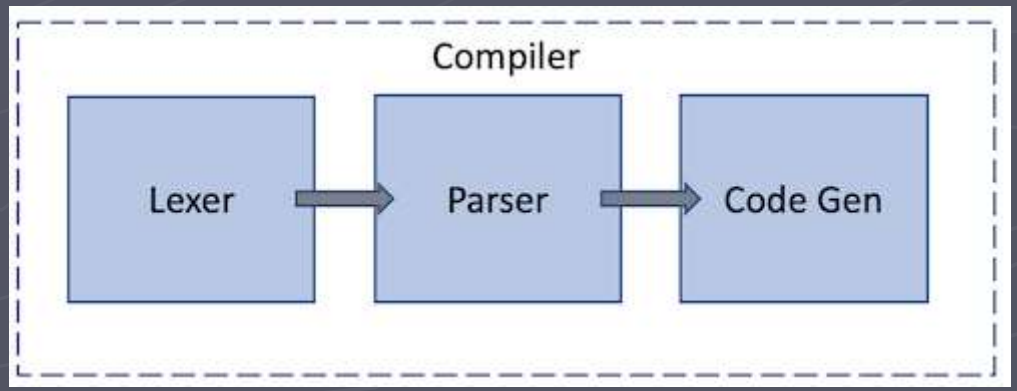


Каналы и фильтры (Pipes and Filters)

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.





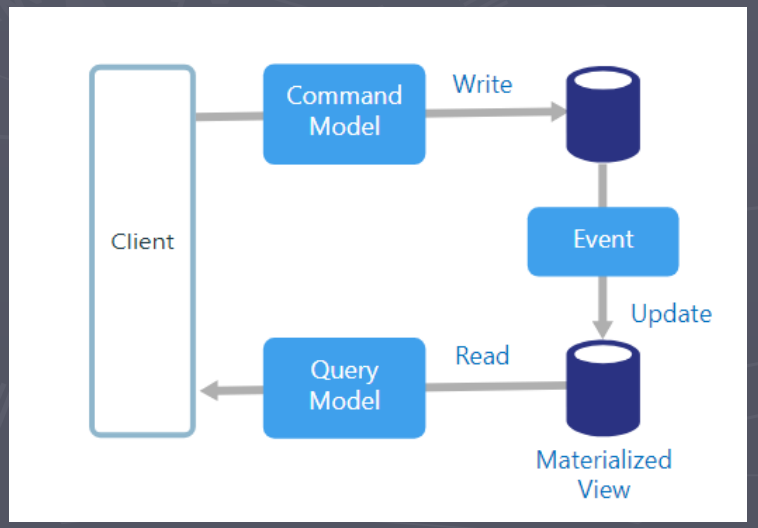


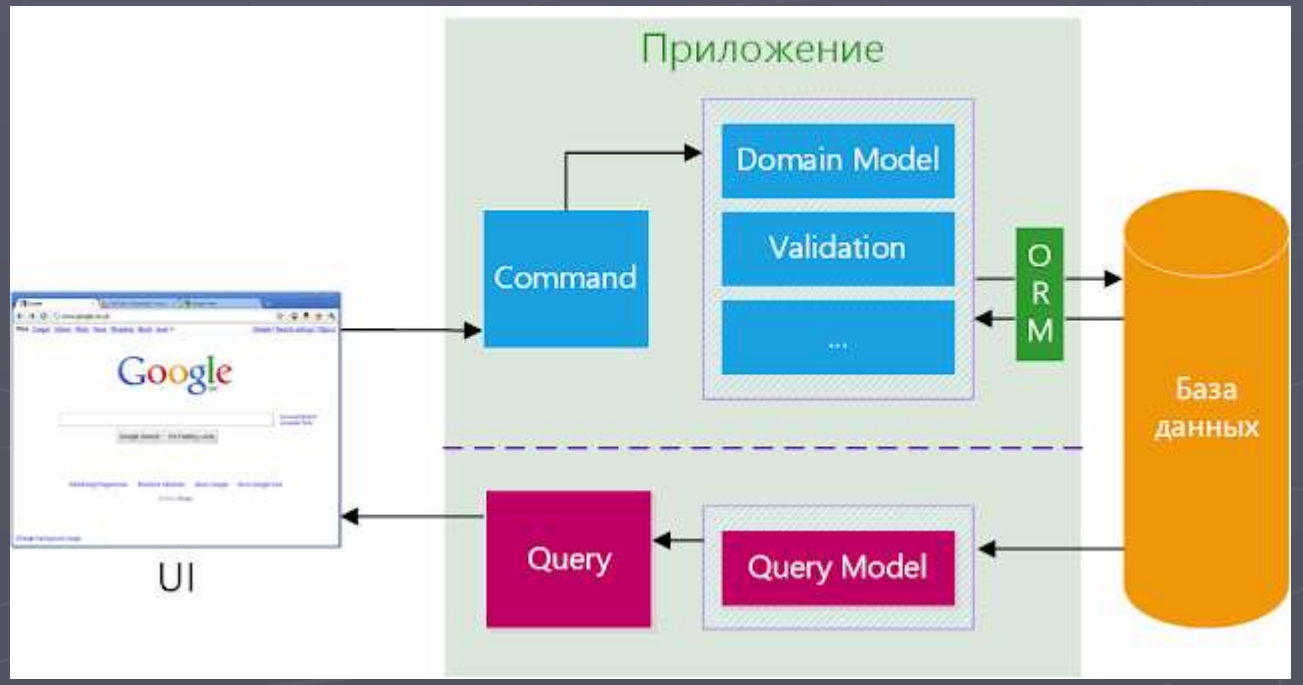
Command and Query Responsibility Segregation (CQRS)

- подход при котором код, изменяющий состояние, отделяется от кода, читающего это состояние.
- В основе этого подхода

лежит принцип

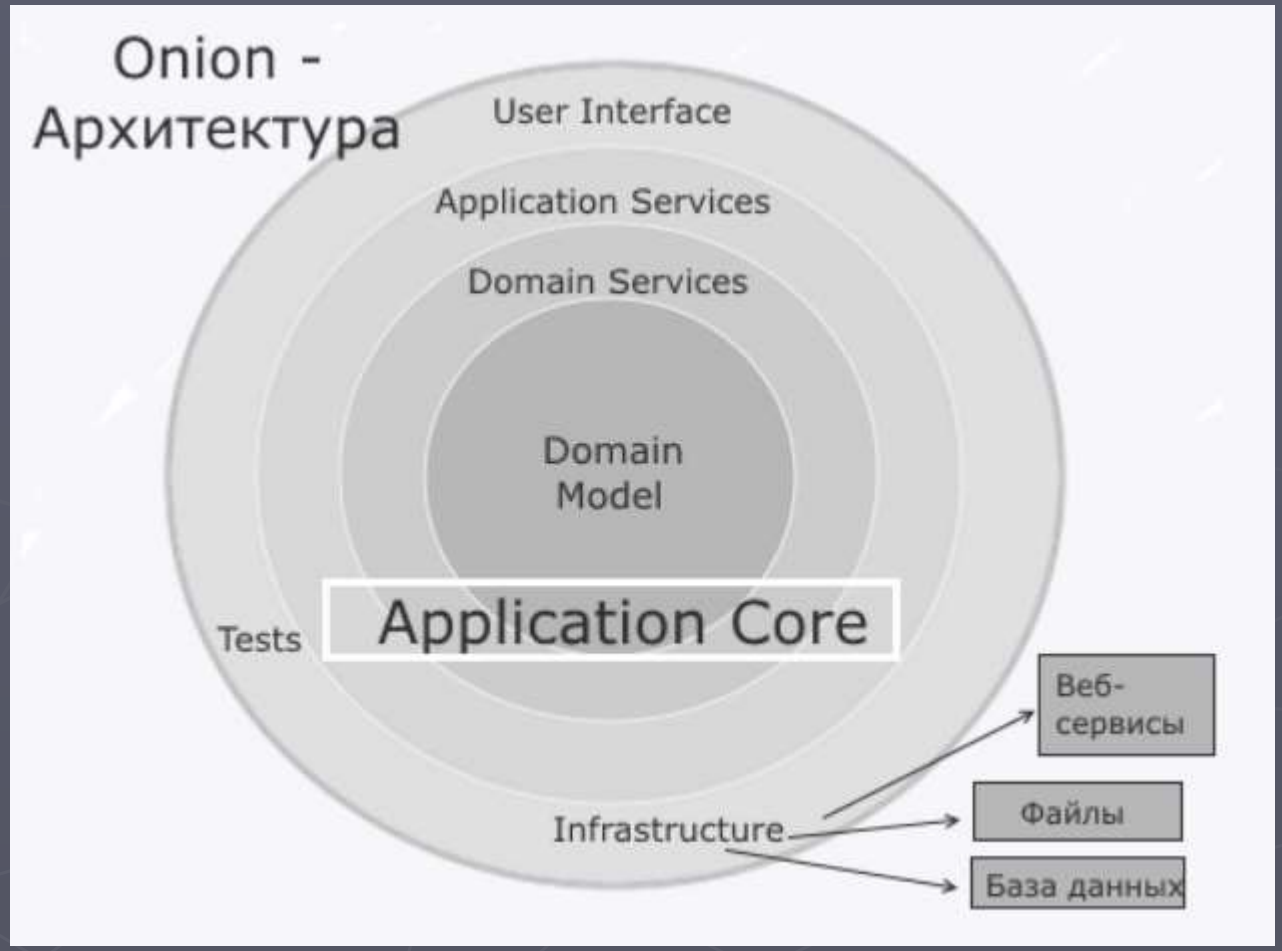
Command-query separation -CQS





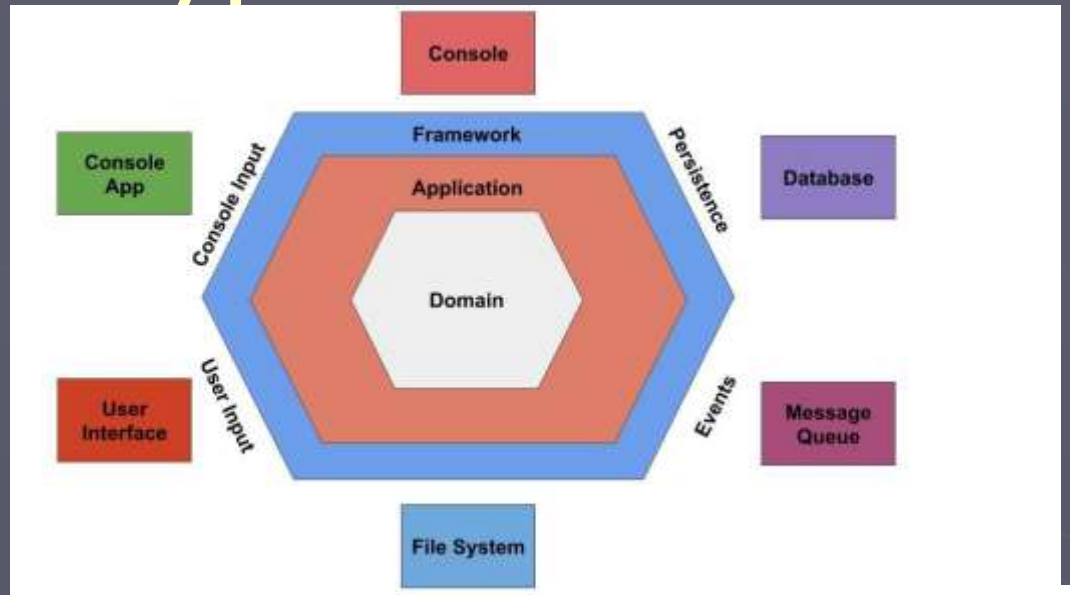
Onion-архитектура

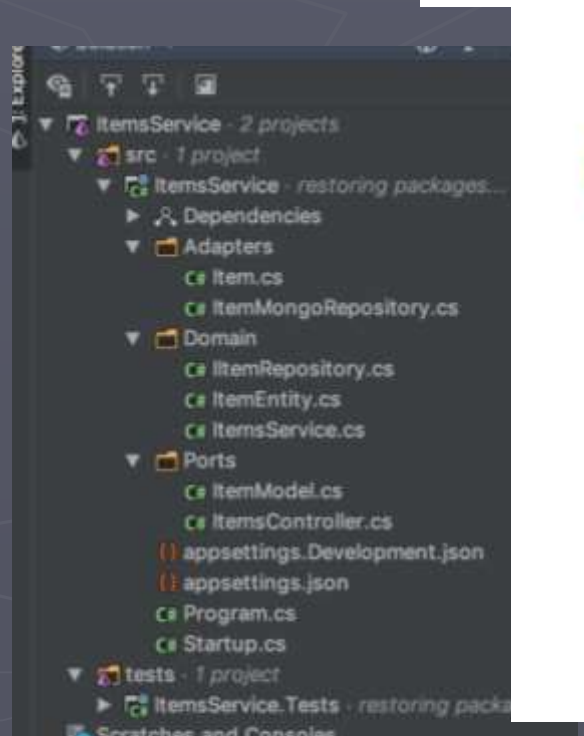
- Ее ключевая особенность в управлении связанностью.
- Фундаментальное правило — любой модуль приложения может зависеть от более близких к центру луковицы модулей, но не может зависеть от более дальних.

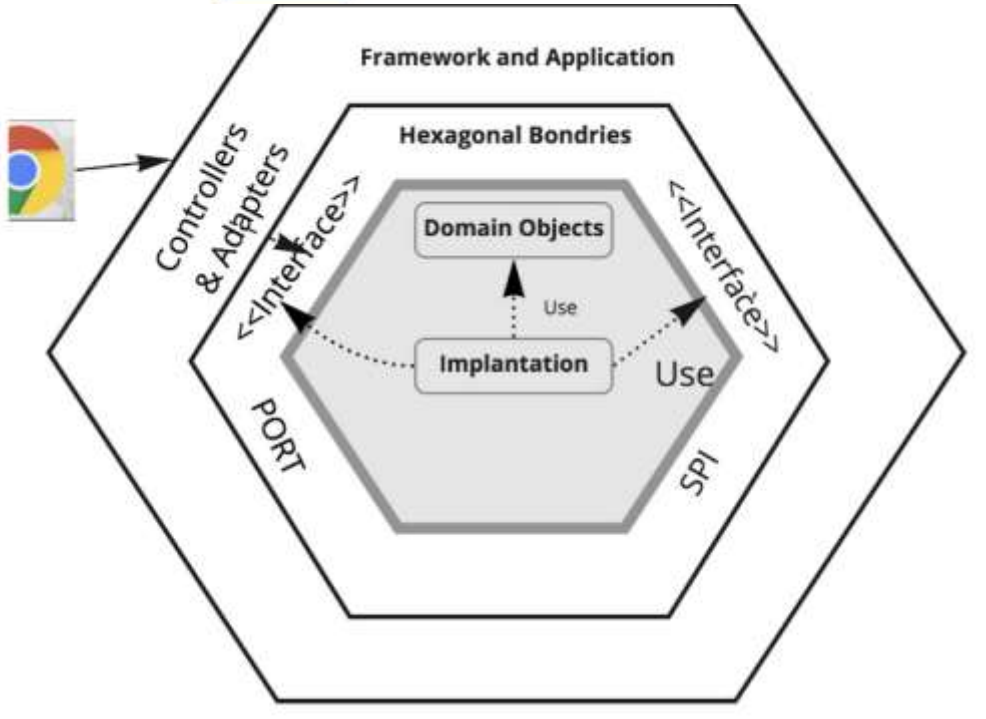


Гексагональная архитектура

- Гексагональная архитектура, или архитектура портов и адаптеров, является архитектурным шаблоном, используемым при разработке программного обеспечения. Он направлен на создание слабосвязанных компонентов приложения, которые можно легко подключить к своей программной среде с помощью портов и адаптеров.



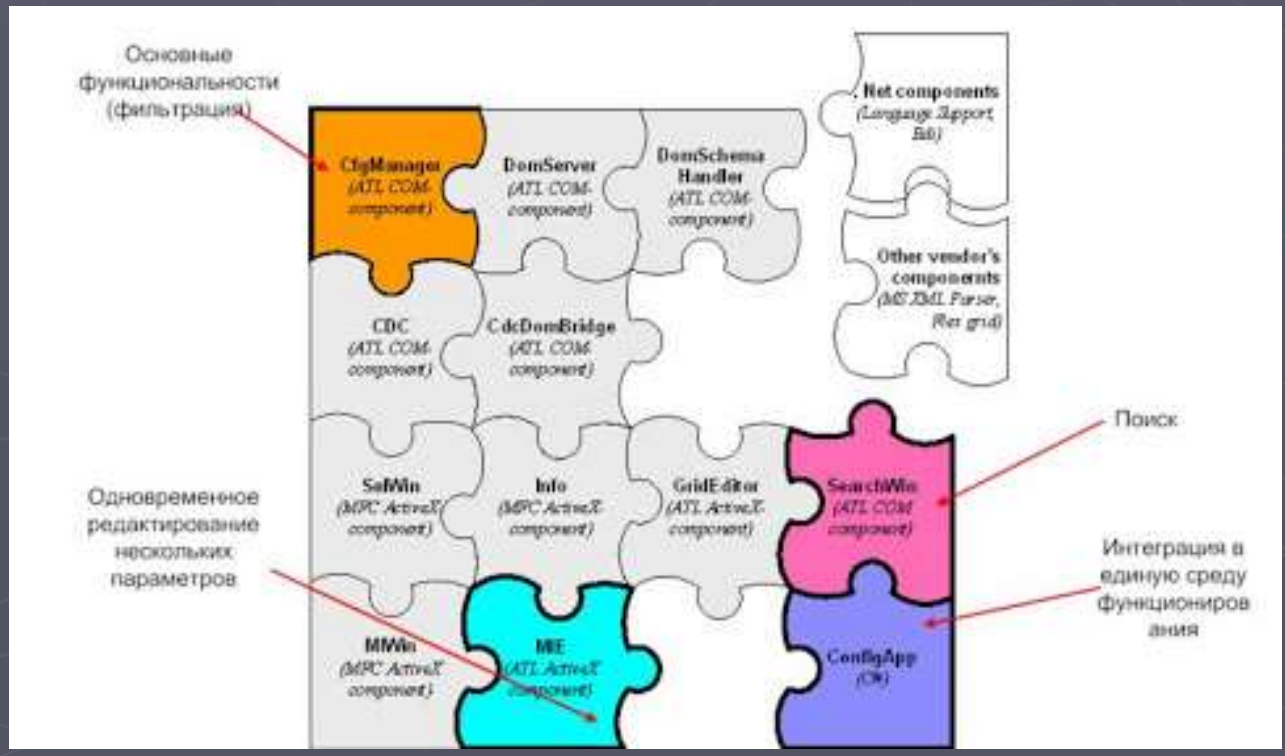




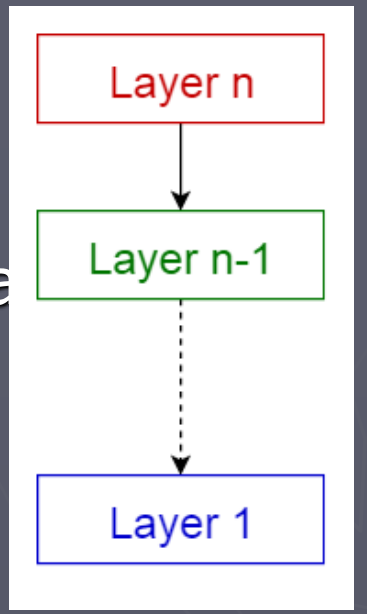
Компонентная архитектура

- Из приложения выделятся отдельные, относительно независимые части, называемые **компонентами**. Компоненты пригодны для повторного использования.

- Меняется подход к созданию приложений. Теперь это выделение и организация связей между компонентами.



Многоуровневая (многослойная, многозвенная) архитектура

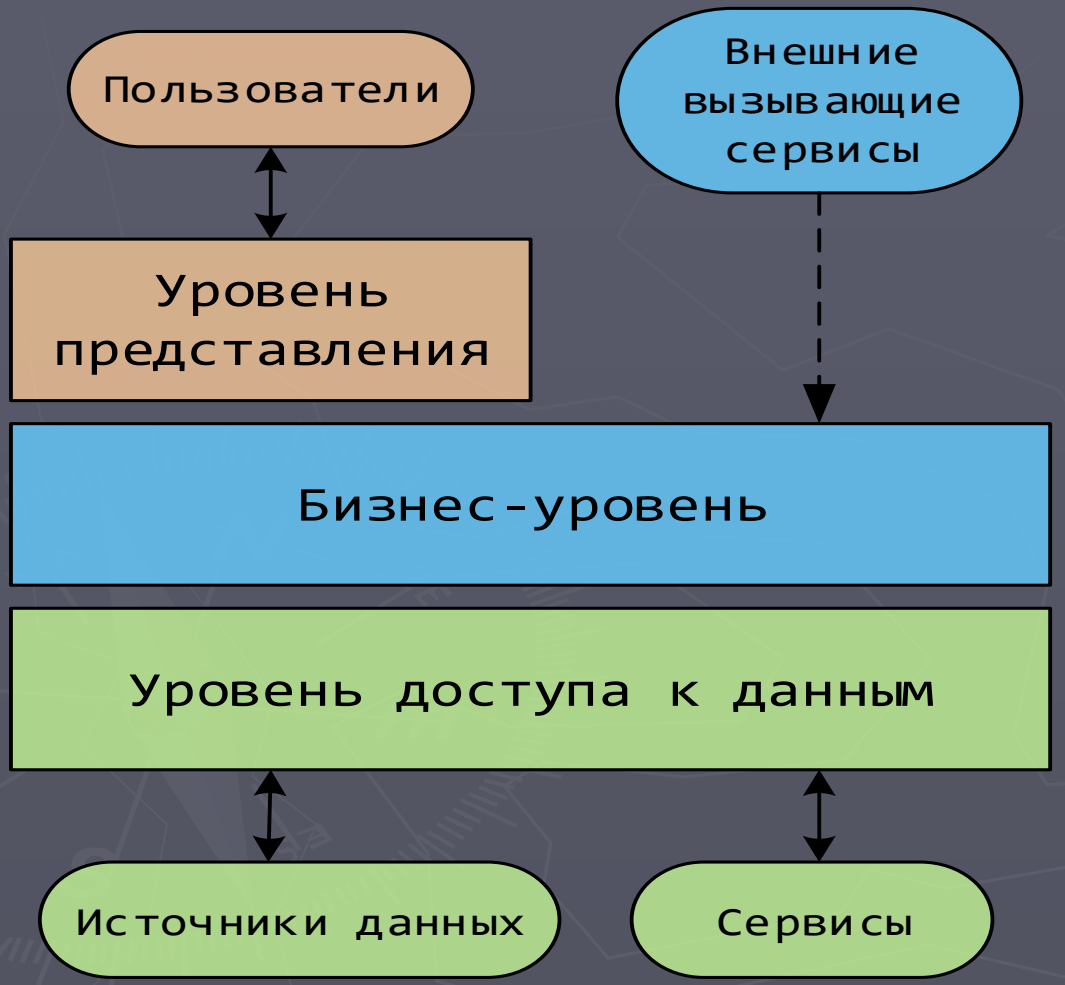


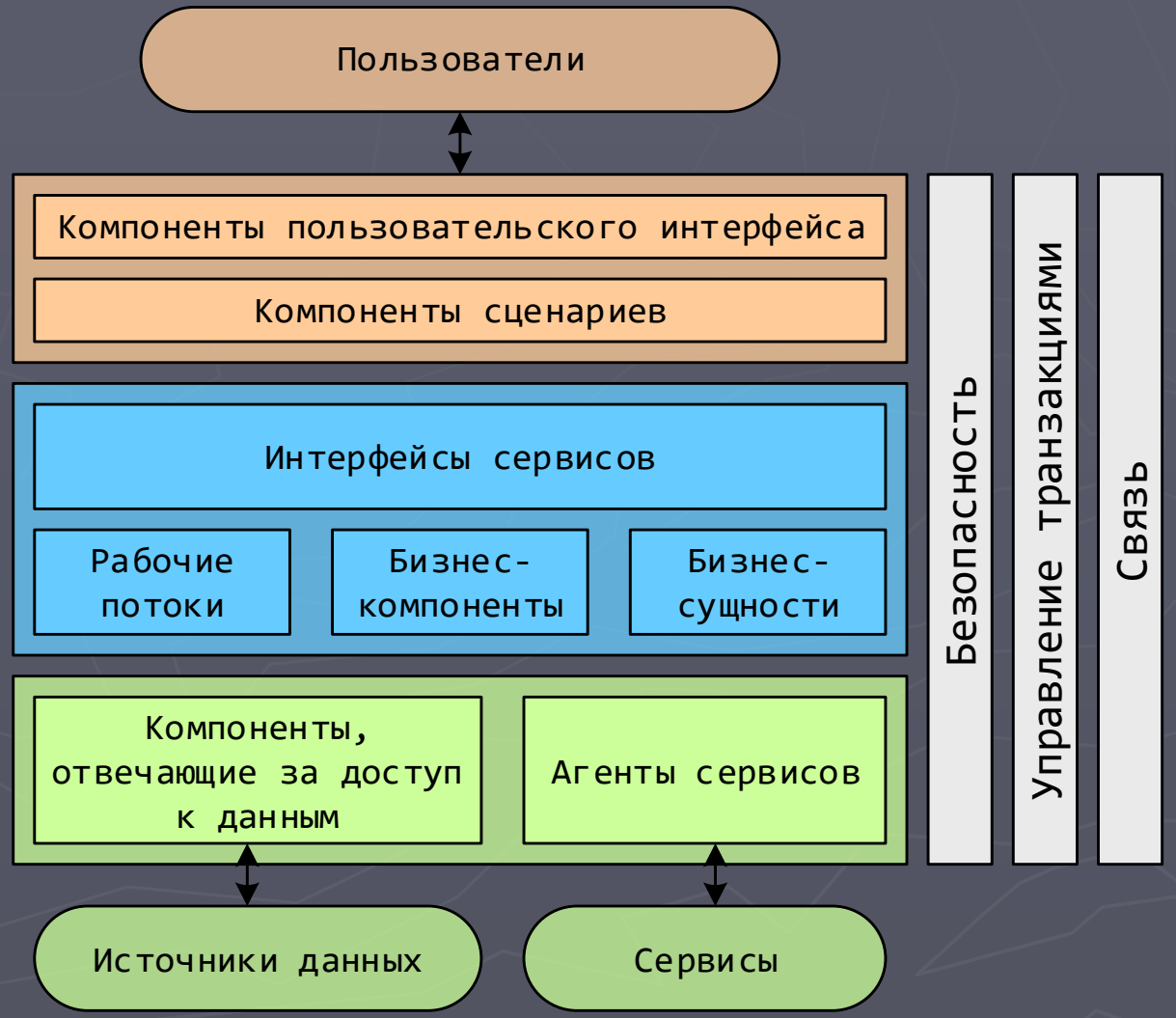
- *Многоуровневая архитектура* (multilayered architecture) базируется на следующих принципах:
  - Каждая часть системы соотносится с определённым **уровнем** (layer).
  - Для каждого уровня заданы выполняемые им функции.
  - Уровни выстроены в **стековую структуру**.
  - Нижние уровни независимы от верхних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

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.

Трёхуровневая (3-х или 4-х) архитектура

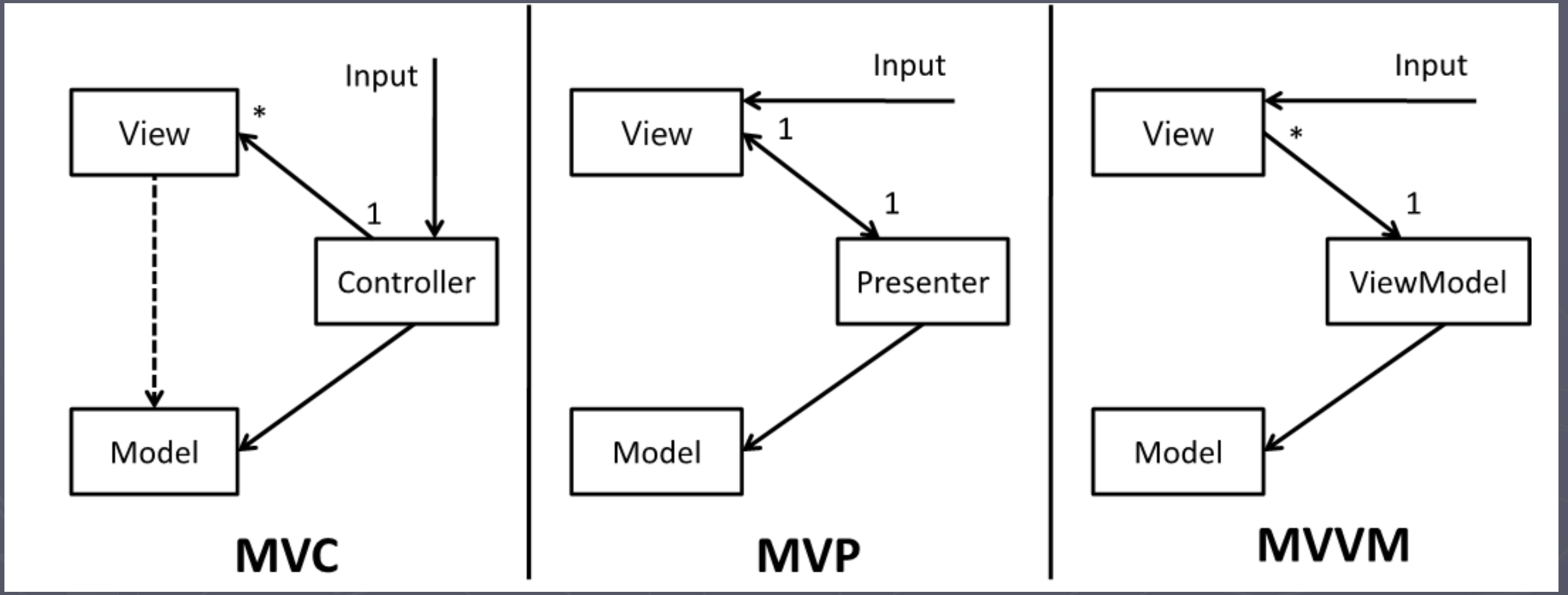




Выделенное представление

- *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



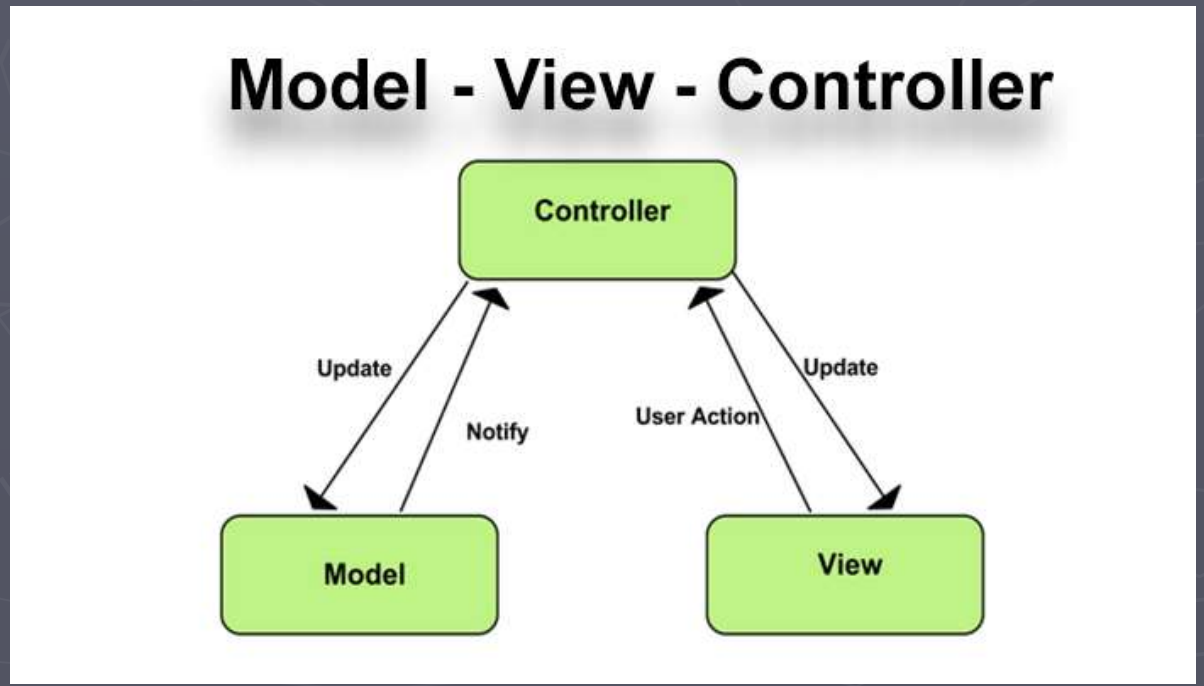
Web

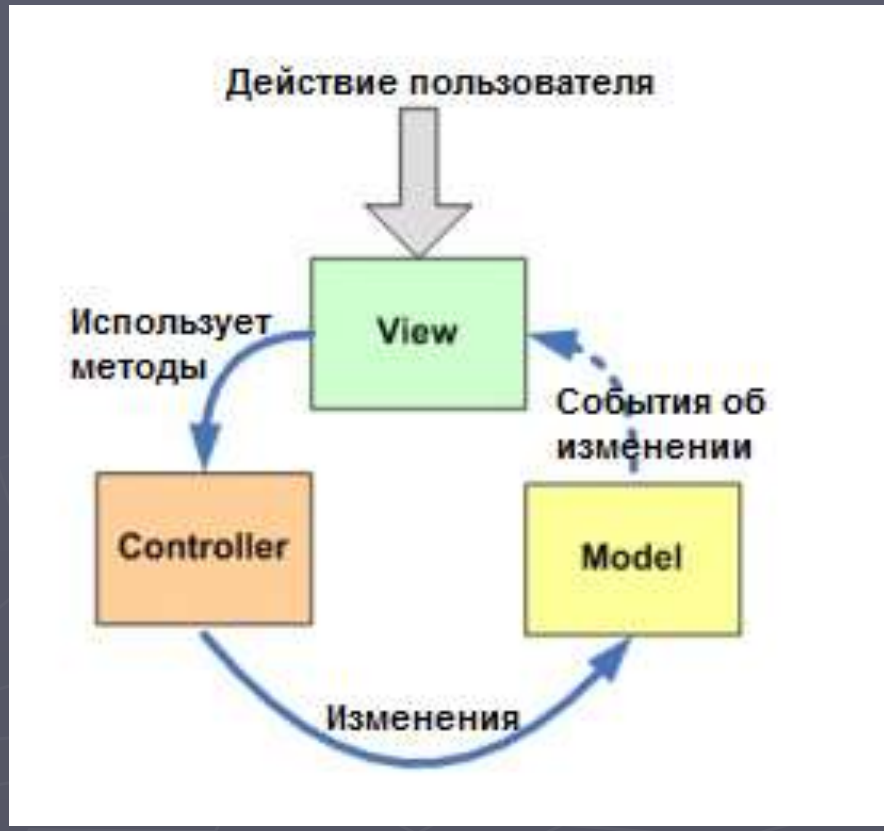
WinForm

WPF

Model-View-Controller

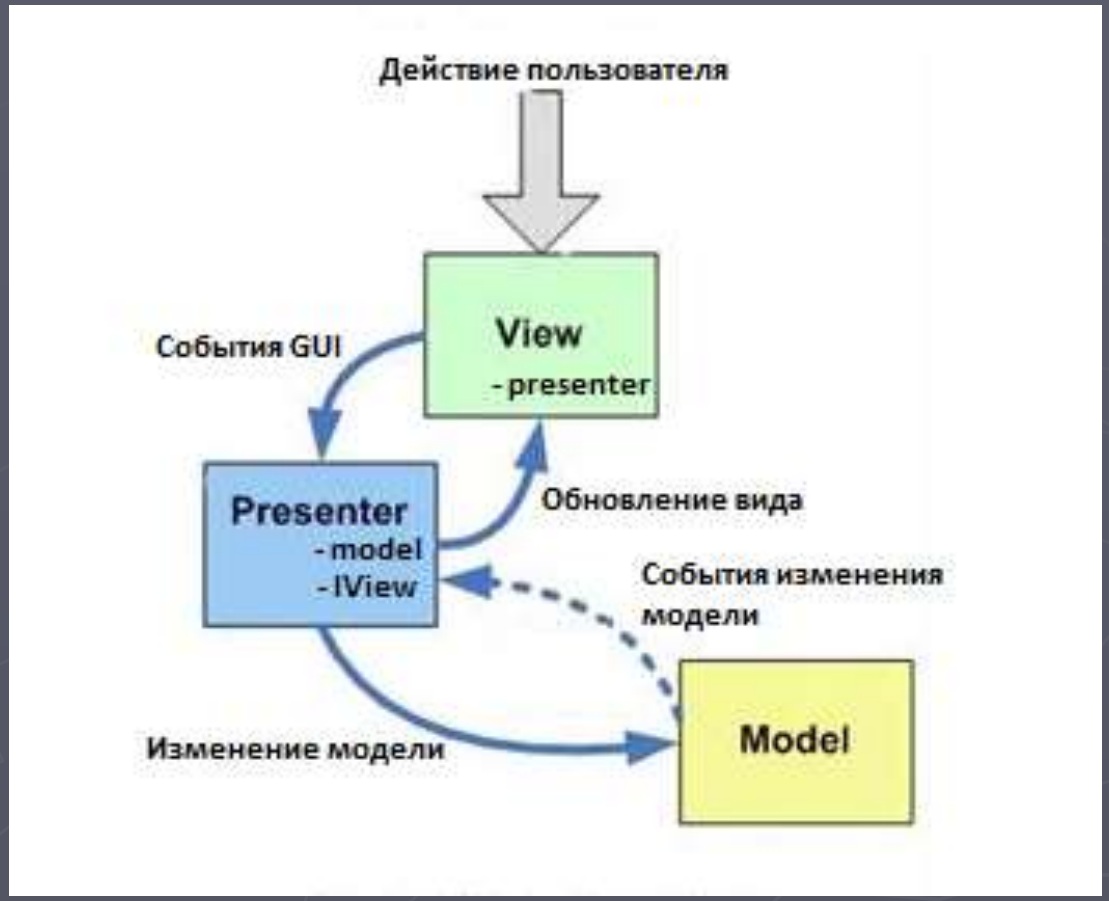
- два варианта:
- **контроллер-супервизор (supervising controller)**
- **пассивное представление (passive view)**.

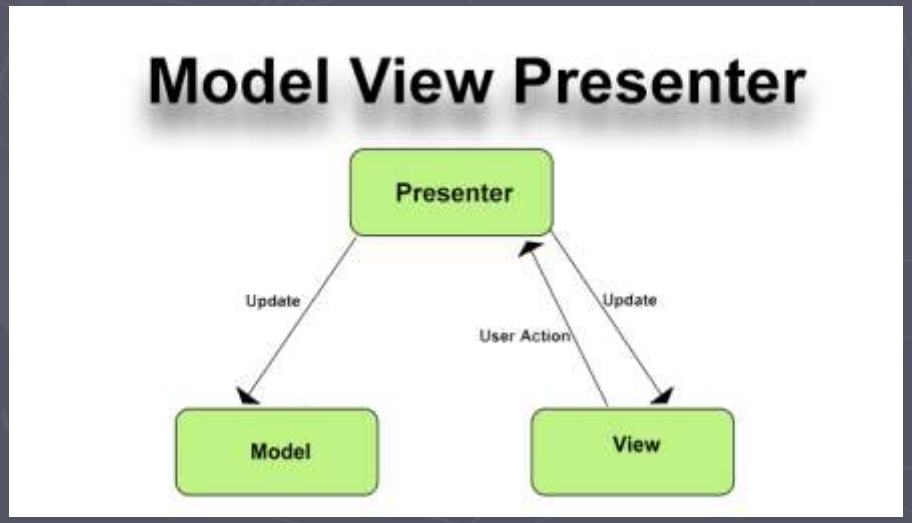




Model-View-Presenter

- **Признаки презентера:**
  Двухсторонняя коммуникация с представлением;
- Представление взаимодействует напрямую с презентером, путем вызова соответствующих функций или событий экземпляра презентера;
- Презентер взаимодействует с View путем использования специального интерфейса, реализованного представлением;

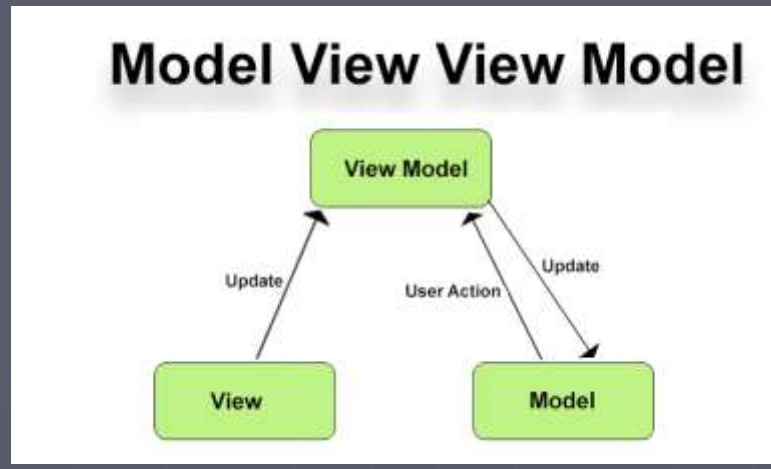




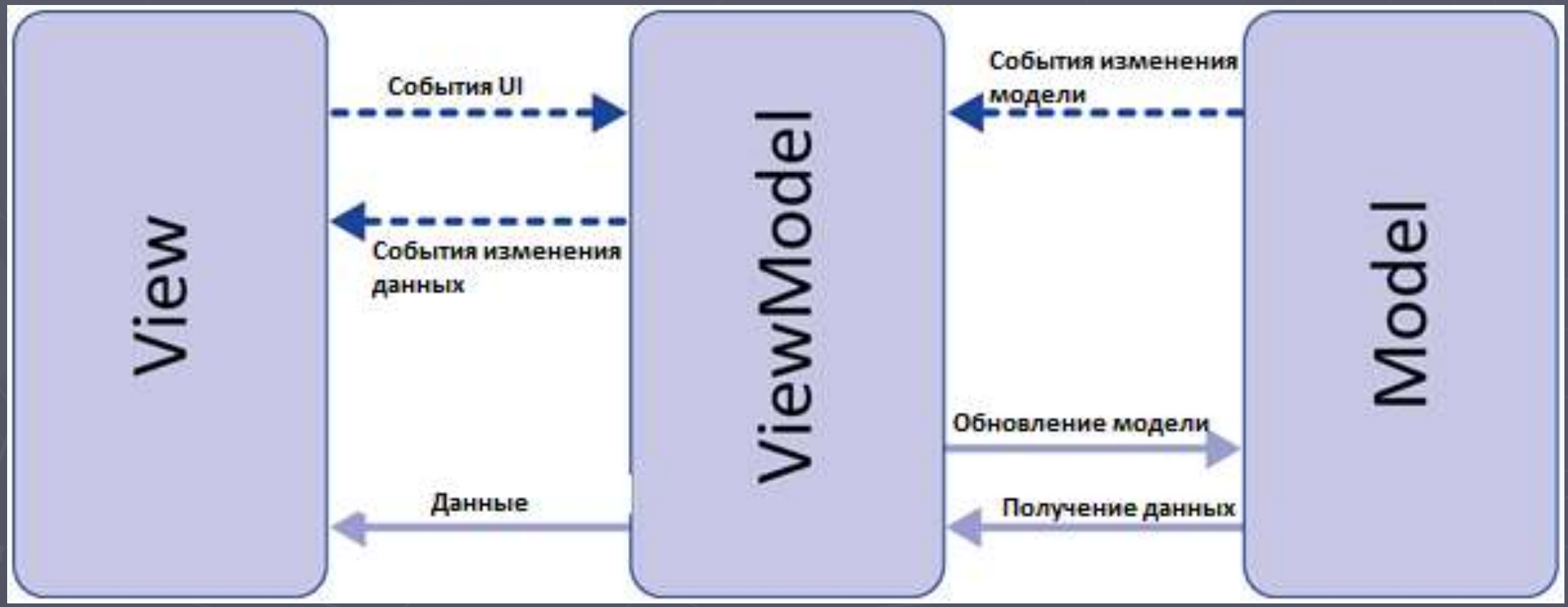
Отвечает за данные выборки

UI (дизайн)

Логика работы (программист)



MVVM (Model-View-ViewModel)



Область использования MVVM

- WPF
- Silverlight
- WinRT
- HTML5 (Knockout/Angular)
- Xamarin
- Windows 10
- Мобильные Android, iOS

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

Выбор паттерна

MVVM

- Используется в ситуации, когда возможно связывание данных без необходимости ввода специальных интерфейсов представления

MVP

- Используется в ситуации, когда невозможно связывание данных (нельзя использовать Binding)

MVC

- Используется в ситуации, когда связь между представлением и другими частями приложения невозможна

Пример плохого метода с точки зрения шаблона

```
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();
        }
```

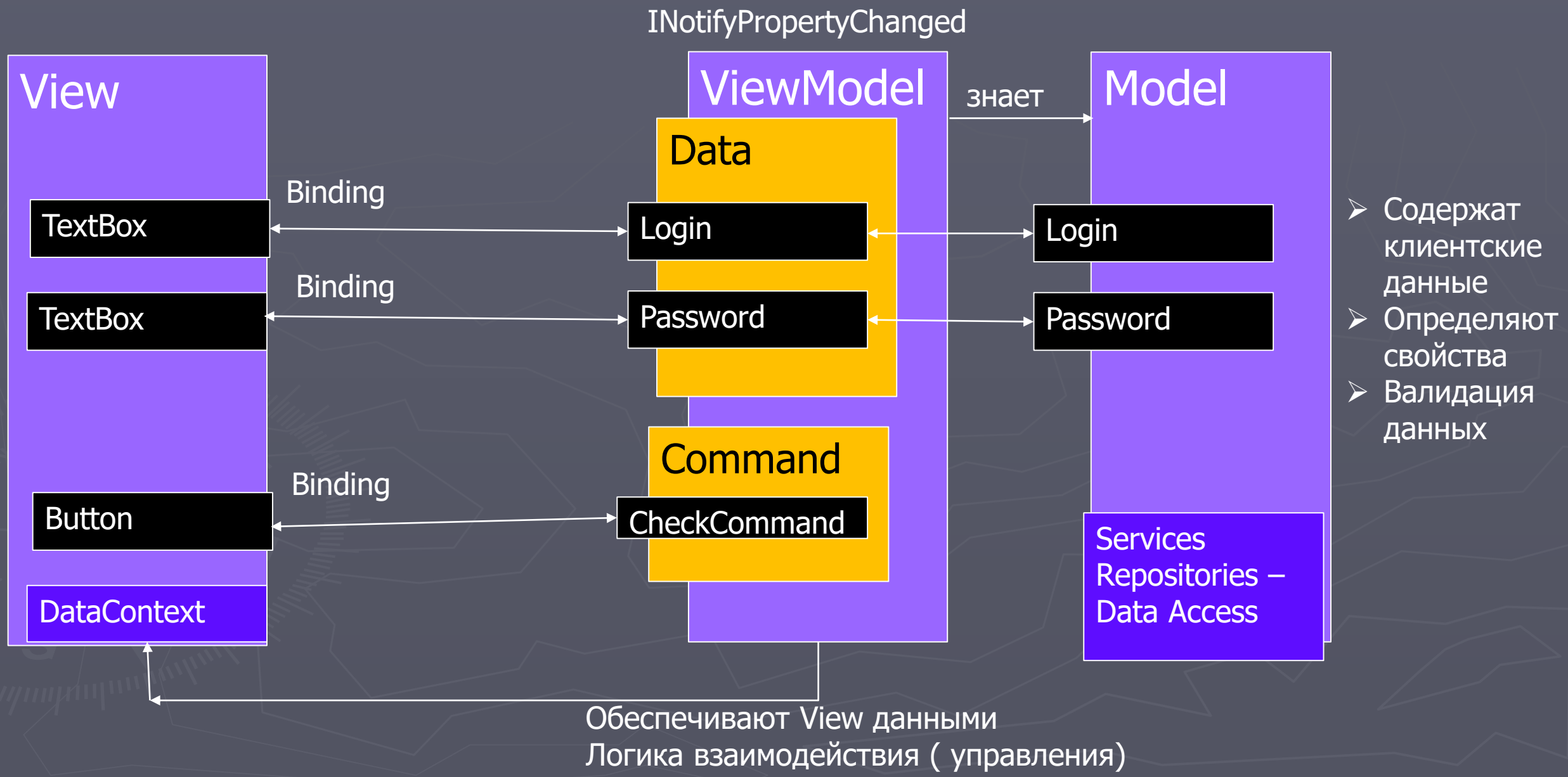
Доступ к UI

Доступ к данным

Управление

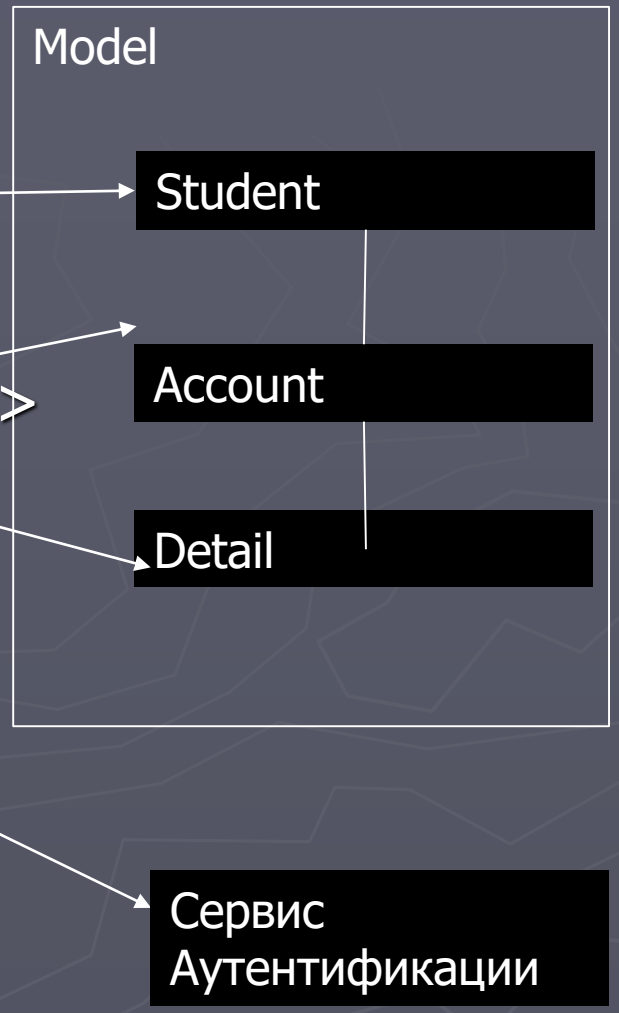
Доступ к UI

Архитектура MVVM



- Содержат клиентские данные
- Определяют свойства
- Валидация данных

```
public Student Student {get;set}
//обращение
public ObservableCollection<AccountItem>
   Accounts {get;set;}
//обертка
public bool LoggIn{get;set;}
```

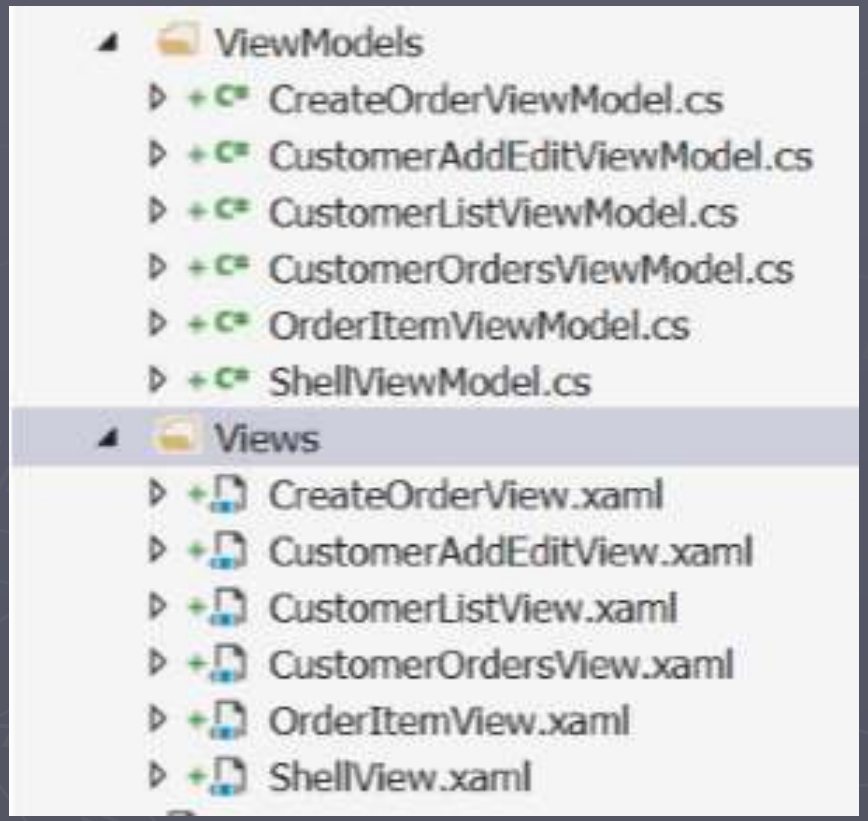


Принципы именования классов в MVVM

- 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

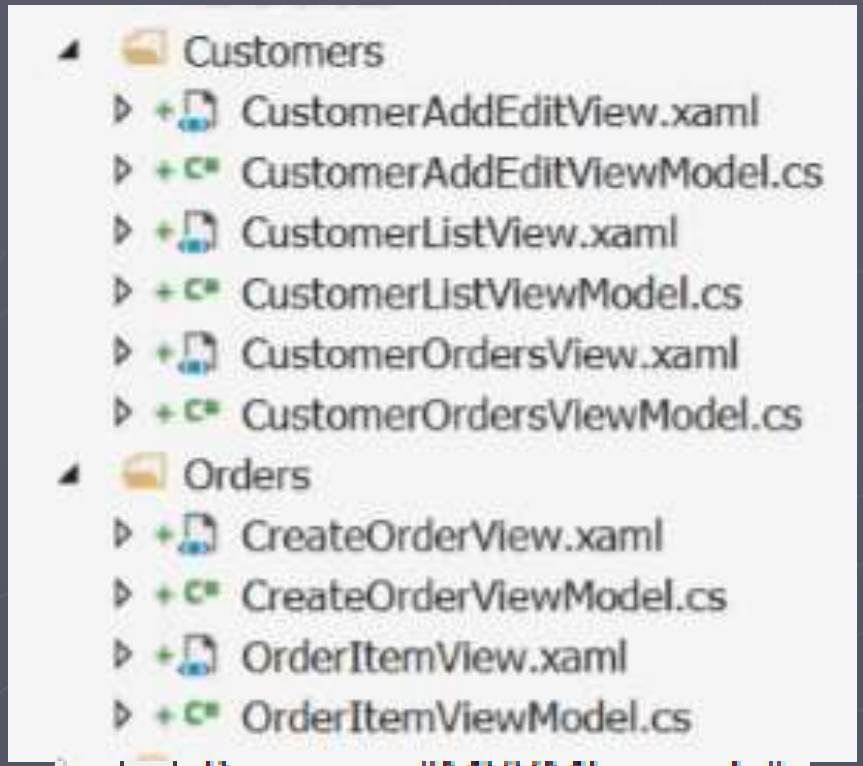
- Папки по типу

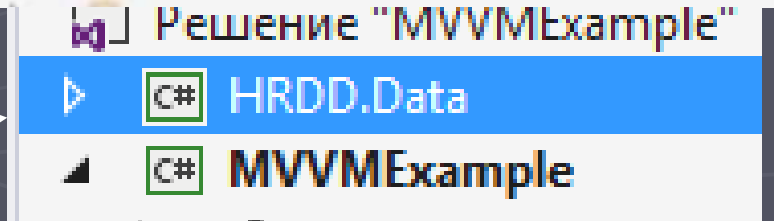


Легко находить в большом проекте

Типы модели в отдельную библиотеку классов

Папки по функциям (или по Use Case)





Подходы к проектированию

- View First
  - Определение в Xaml или Code Behind DataContext

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

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

  - ViewMode First

Пример разработки проекта на основе MVVM

- 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 First }
        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 HRDDDbContext : 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

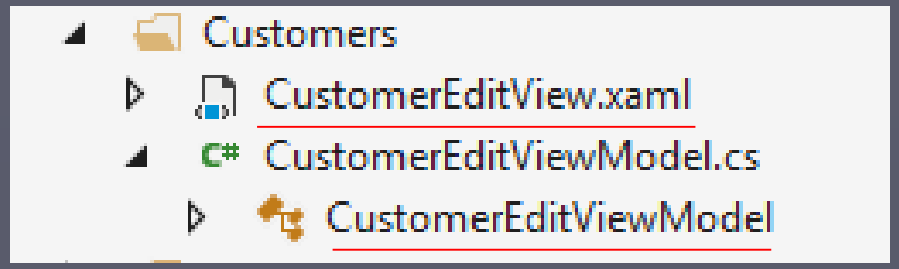
```
public interface ICustomersRepository
    {
        Task<List<Customer>> GetCustomersAsync();
        Task<Customer> GetCustomerAsync(Guid id);
        Task<Customer> AddCustomerAsync(Customer customer);
        Task<Customer> UpdateCustomerAsync(Customer customer);
        Task DeleteCustomerAsync(Guid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

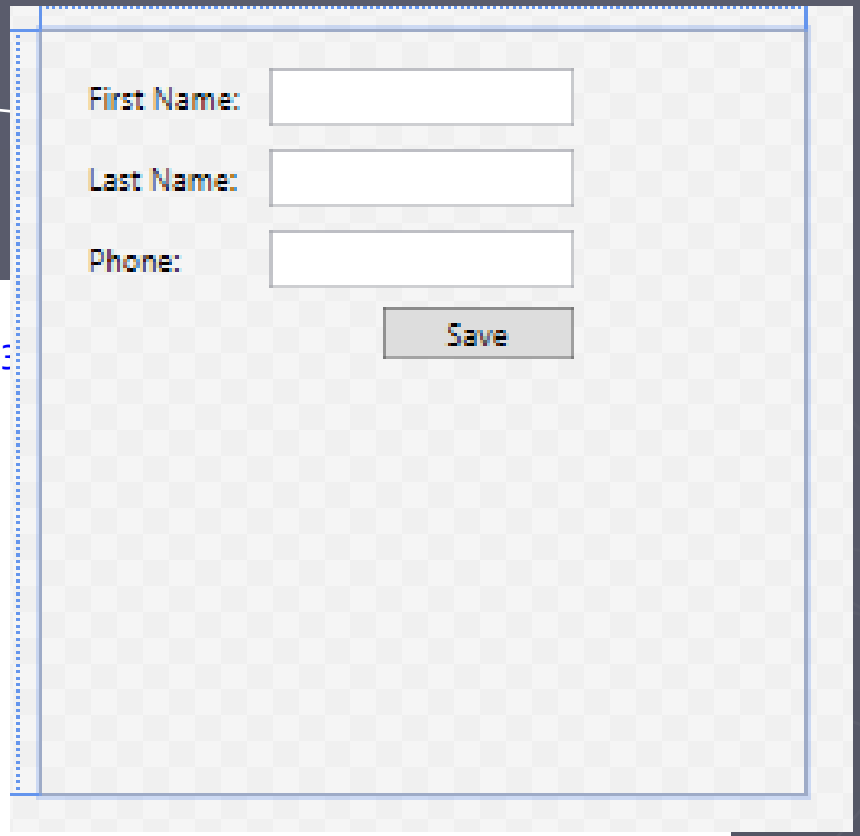
3) Реализация репозитория

```
 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.FirstOrDefaultAsync(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

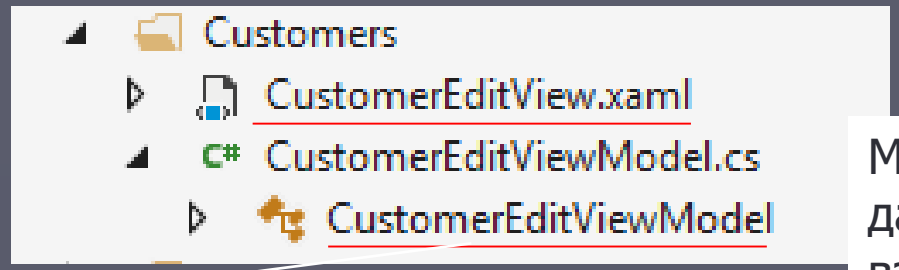
4) добавляем View






```
 <UserControl.DataContext>
        <local:CustomerEditViewModel CustomerId="11DA4696-CEA3
    </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>
```

5) Добавляем ViewModel



Манипуляции с данными и логика взаимодействия

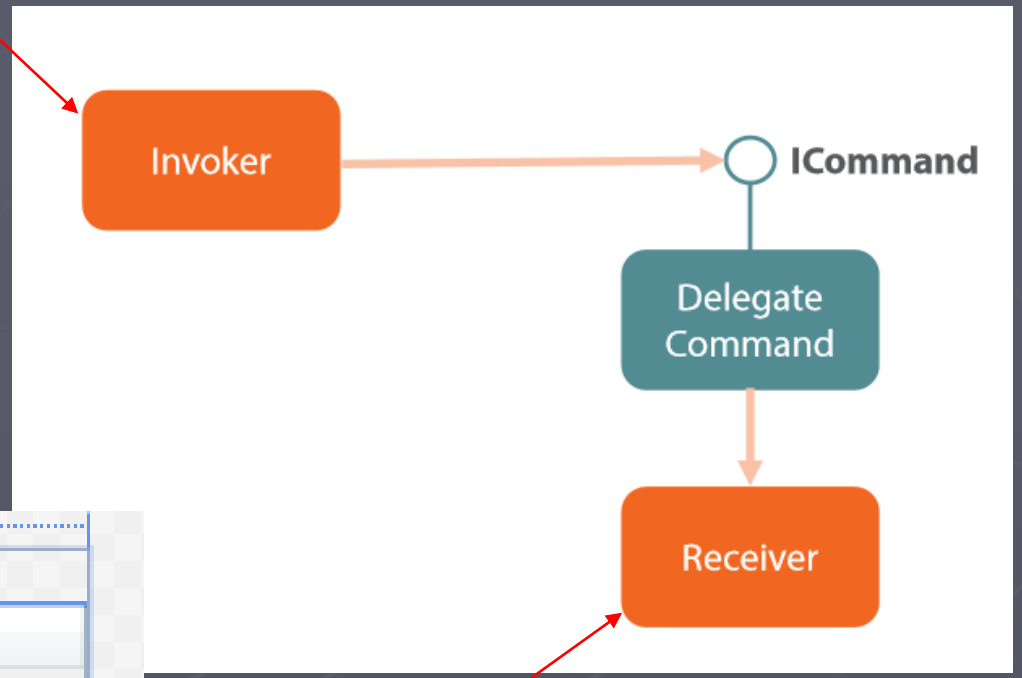
```
public class CustomerEditViewModel : INotifyPropertyChanged
    {   private Customer _customer;
        private ICustomersRepository _repository = new CustomersRepository();

        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

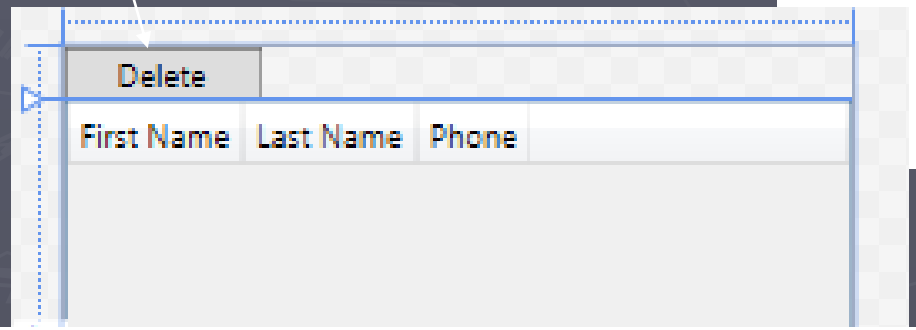
Сохранение и загрузка

Использование Command

View



Добавить команду удаления



ViewModel

View

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

```
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 { };
    }
}
```

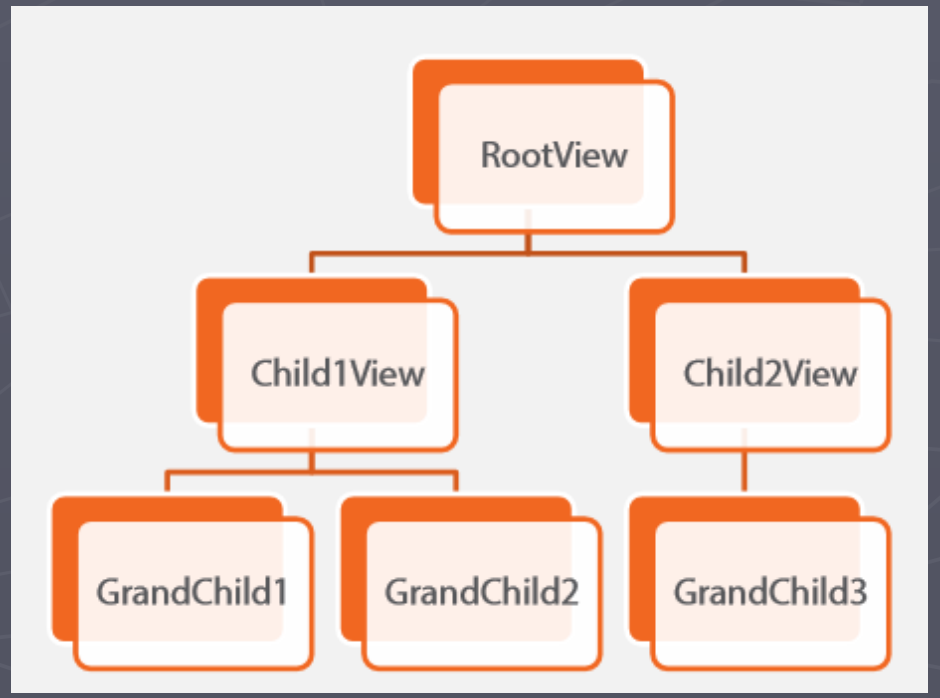
ViewModel

```
 public class Customer :INotifyPropertyChanged
    {
//...
        public string FirstName
        {
            get
            {
                return _firstName;
            }
            set
            {
                if (_firstName != value)
                {
                    _firstName = value;
               PropertyChanged(this, new PropertyChangedEventArgs("FirstName"));
                }
            }
        }
```

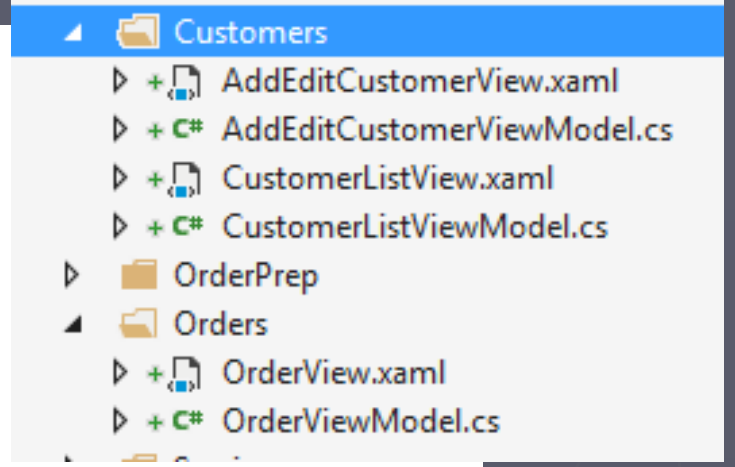
Model

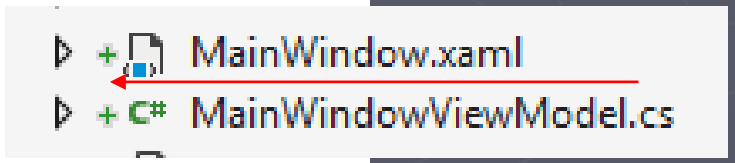
Иерархия и управление

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

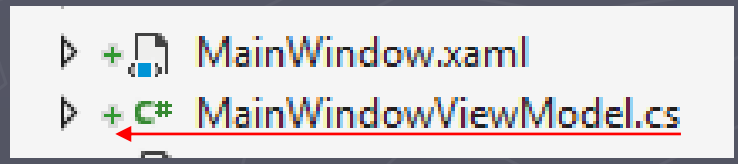


```
<Window x:Class="ZzaDesktop.MainWindow"
      xmlns:cust="clr-namespace:MVVMDemo.Customers"
        xmlns:order="clr-namespace:MVVMDemo.Orders"
        xmlns:prep="clr-namespace:MVVMDemo.OrderPrep"
        xmlns:local="clr-namespace:MVVMDemo"
        Title="MainWindow"
        Height="350"
        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>
```



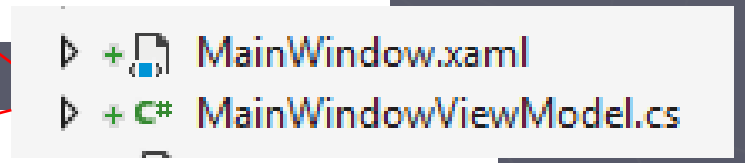


```
class MainWindowViewModel : BindableBase
    {
        private CustomerListViewModel _customerListViewModel = new CustomerListViewModel();
        private OrderViewModel _orderViewModel = new OrderViewModel();
        private OrderPrepViewModel _orderPrepViewModel = new OrderPrepViewModel();
   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

        private BindableBase _CurrentViewModel;

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



Добавление команд навигации

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




```
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

- WPF MVVM Toolkit
- Prism
- Caliburn Micro
- MVVM Light
- Cinch