**1.WindowsForms. Модель обработки событий. Взаимодействие между формами.**

Для создания графических интерфейсов с помощью платформы .NET применяются разные технологии - Window Forms, WPF). Однако наиболее простой и удобной платформой до сих пор остается Window Forms или формы.

Для взаимодействия с пользователем в Windows Forms используется механизм событий. События в Windows Forms представляют стандартные события на C#, только применяемые к визуальным компонентам и подчиняются тем же правилам, что события в C#.

Для добавления обработчика используется стандартный синтаксис C#: this.Load += new System.EventHandler(this.Form1\_Load)

Допустим, первая форма по нажатию на кнопку будет вызывать вторую форму. Во-первых, добавим на первую форму Form1 кнопку и двойным щелчком по кнопке перейдем в файл кода. Итак, мы попадем в обработчик события нажатия кнопки, который создается по умолчанию после двойного щелчка по кнопке.

private void button1\_Click(object sender, EventArgs e)

{

Form2 newForm = new Form2();

newForm.Show();

}

Пока вторая форма не знает о существовании первой. Чтобы это исправить, надо второй форме как-то передать сведения о первой форме. Для этого воспользуемся передачей ссылки на форму в конструкторе.

public Form2(Form1 f)

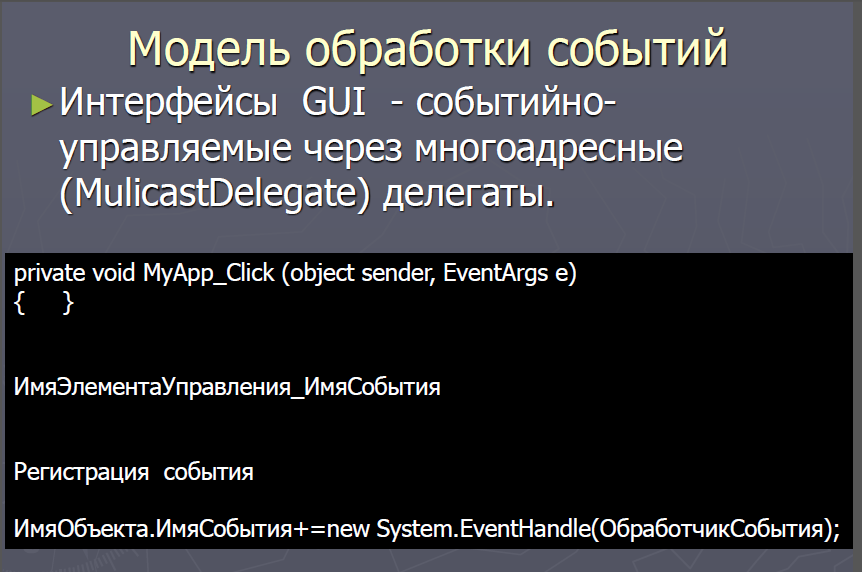
{

InitializeComponent();

f.BackColor = Color.Yellow;

}

Форма – это объект, который задается свойствами, определяющими их внешний вид, методами, определяющими их поведение, и событиями, определяющими их взаимодействие с пользователем.



**2.Элементы управления и их объектная модель.**

Элемент управления играет роль графической связи между пользователем и программой. Элемент управления может предоставлять или обрабатывать данные, принимать данные, введенные пользователем, реагировать на события, а также выполнять любые другие функции, связывающие пользователя и приложение.

Все элементы управления наследуются от класса Control и поэтому имеют ряд общих свойств:

**Anchor:** Определяет, как элемент будет растягиваться

**BackColor:** Определяет фоновый цвет элемента

**BackgroundImage:** Определяет фоновое изображение элемента

**ContextMenu:** Контекстное меню, которое открывается при нажатии на элемент правой кнопкой мыши. Задается с помощью элемента ContextMenu

**Cursor:** Представляет, как будет отображаться курсор мыши при наведении на элемент

**Dock:** Задает расположение элемента на форме

**Enabled:** Определяет, будет ли доступен элемент для использования. Если это свойство имеет значение False, то элемент блокируется.

**Font:** Устанавливает шрифт текста для элемента

**ForeColor:** Определяет цвет шрифта

**Location:** Определяет координаты верхнего левого угла элемента управления

**Name**: Имя элемента управления

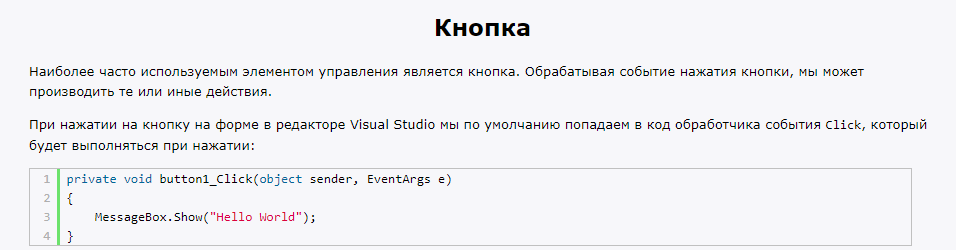
**Size:** Определяет размер элемента

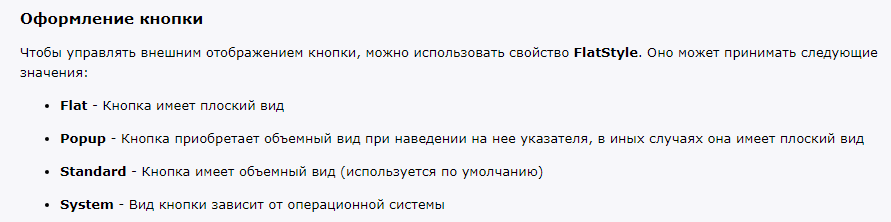
**Width:** ширина элемента

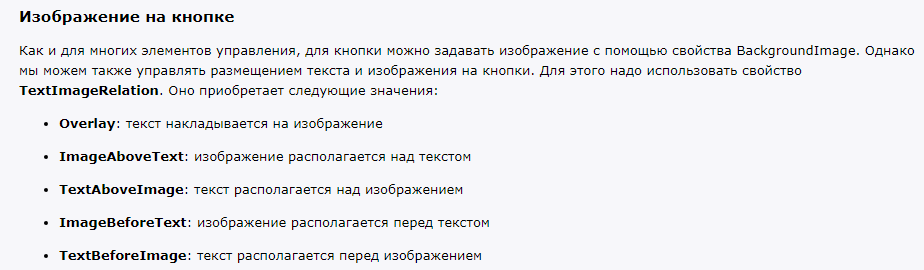
**Height:** высота элемента

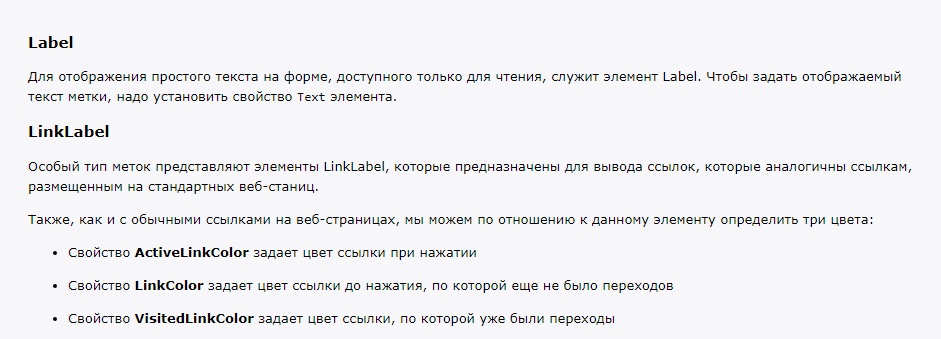
**TabIndex:** Определяет порядок обхода элемента по нажатию на клавишу Tab

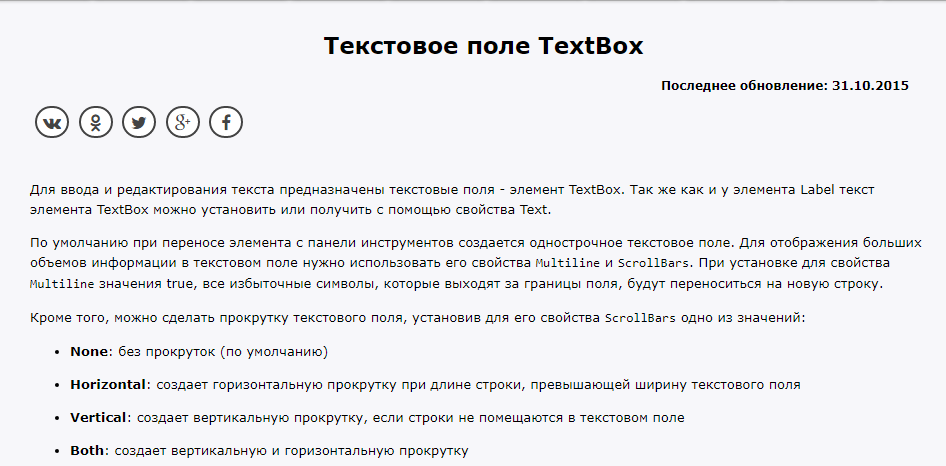
**Tag:** Позволяет сохранять значение, ассоциированное с этим элементом управления

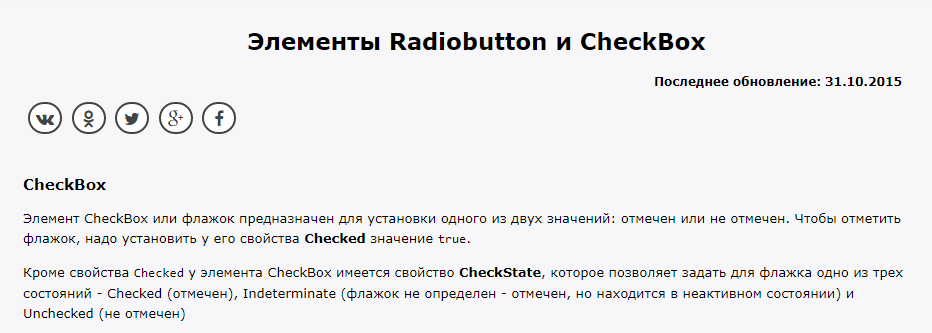


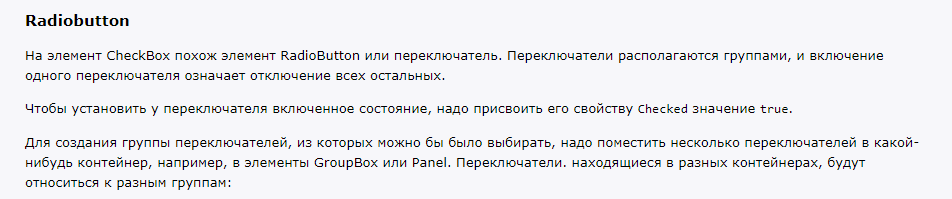


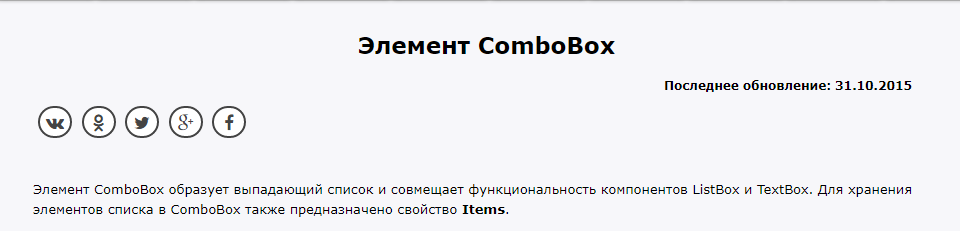
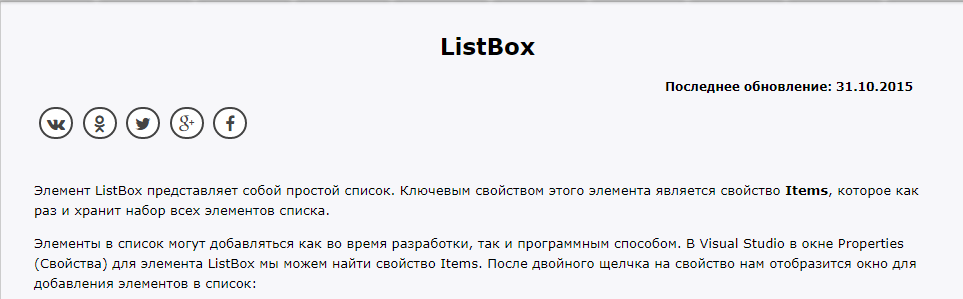


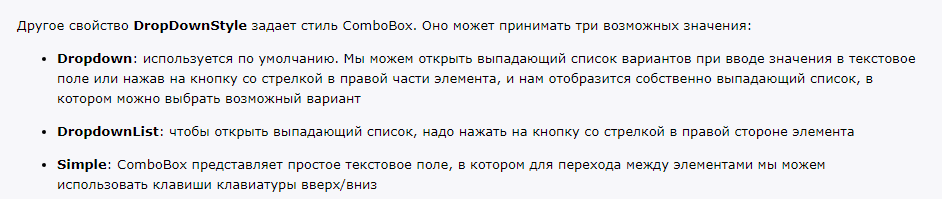




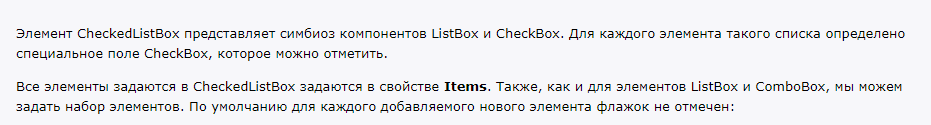


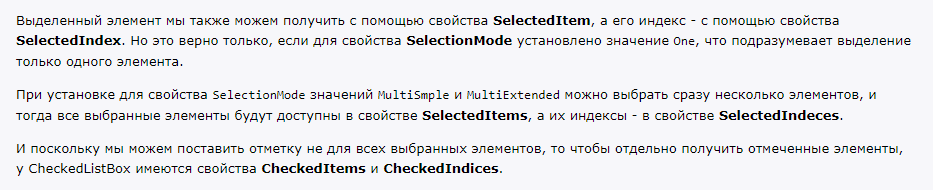


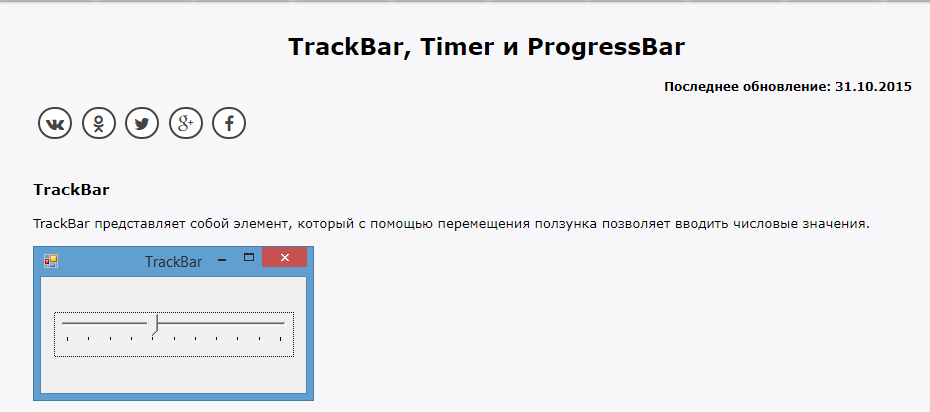


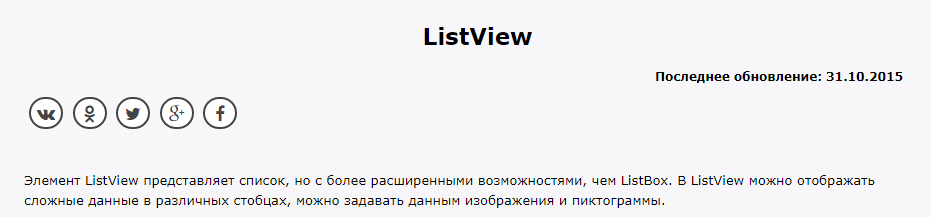


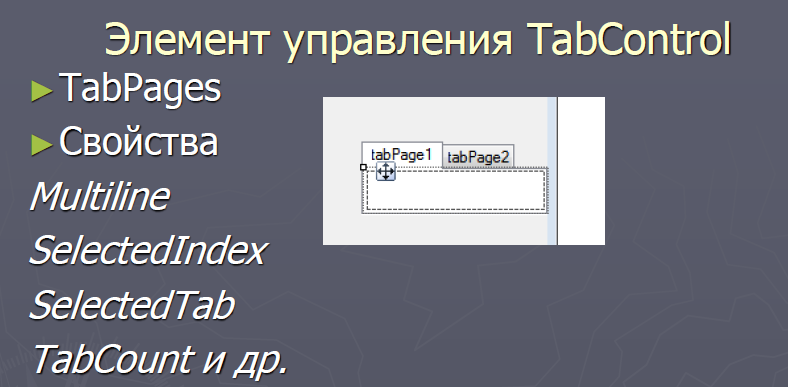












**3.Регулярные выражения. Поиск и чтение результатов.**

Регулярные выражения – это язык для описания текста и внесения в него изменений. Регулярное выражение применяется к строке. Результатом применения является фрагмент строки, либо новая строка, либо группы подстрок, либо логический результат – в зависимости от того, какая операция выполняется.

Регулярные выражения очень мощный и в то же время простой механизм обработки текстовой информации.

Для работы с регулярными выражениями в С# существует класс System.Text.RegularExpressions.Regex. Многие методы этого класса также существуют в двух версиях – статической и экземпляра.

Основная функциональность регулярных выражений в .NET сосредоточена в пространстве имен **System.Text.RegularExpressions**. А центральным классом при работе с регулярными выражениями является класс **Regex**. Например, у нас есть некоторый текст и нам надо найти в нем все словоформы какого-нибудь слова. С классом Regex это сделать очень просто.

string s = "Бык тупогуб, тупогубенький бычок, у быка губа бела была тупа";

Regex regex = new Regex(@"туп(\w\*)");

MatchCollection matches = regex.Matches(s);

if (matches.Count > 0)

{

foreach (Match match in matches)

Console.WriteLine(match.Value);

}

else

{

Console.WriteLine("Совпадений не найдено");

}

Создаём класс реджекс в него кладём регулярное выражение, потом матколлекшн и все совпадение сохраняются в нём же , после проходим циклом и выводим

### **Параметр RegexOptions**

Класс Regex имеет ряд конструкторов, позволяющих выполнить начальную инициализацию объекта. Две версии конструкторов в качестве одного из параметров принимают перечисление RegexOptions. Некоторые из значений, принимаемых данным перечислением:

* **Compiled**: при установке этого значения регулярное выражение компилируется в сборку, что обеспечивает более быстрое выполнение
* **CultureInvariant**: при установке этого значения будут игнорироваться региональные различия
* **IgnoreCase**: при установке этого значения будет игнорироваться регистр
* **IgnorePatternWhitespace**: удаляет из строки пробелы и разрешает комментарии, начинающиеся со знака #
* **Multiline**: указывает, что текст надо рассматривать в многострочном режиме. При таком режиме символы "^" и "$" совпадают, соответственно, с началом и концом любой строки, а не с началом и концом всего текста
* **RightToLeft**: приписывает читать строку справа налево
* **Singleline**: устанавливает однострочный режим, а весь текст рассматривается как одна строка

**4.Регулярные выражения. Классы символов. Привязки и кванторы.**

[…] — **символьные классы**, позволяют перечислить символы, которые могут находиться в данной позиции текста. Например gr[ea]y совпадет со строками «grey» и «gray».

[^…] — **исключающие символьные классы**, позволяют перечислить символы, которые **не могут** находиться в данной позиции текста. Например g[^ae]rdy не совпадет со строками «gardy» и «gerdy», но совпадет со строками «gurdy», «g3rdy» и «girdy».

**5.Атрибуты валидации модели (диапазоны, типы, сообщения об ошибках, регулярные выражения,**

### Атрибут Required

Применение этого атрибута к свойству модели означает, что данное свойство должно быть обязательно установлено.

[Required (ErrorMessage="Поле должно быть установлено")]

### Атрибут StringLength

Чтобы пользователь не мог ввести очень длинный текст, используется атрибут StringLength. Особенно это актуально, если в базе данных установлено ограничение на размер хранящихся строк.

[StringLength (50, MinimumLength=3,ErrorMessage="Длина строки должна быть от 3 до 50 символов")]

### Атрибут RegularExpression

Применение данного атрибута предполагает, что вводимое значение должно соответствовать указанному в этом атрибуте регулярному выражению.

[RegularExpression(@"[A-Za-z0-9.\_%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,4}", ErrorMessage = "Некорректный адрес")]

### Атрибут Range

Атрибут Range определяет минимальные и максимальные ограничения для числовых данных.

[Range(1700,2000,ErrorMessage="Недопустимый год")]

**6.Атрибуты валидации. Создание атрибута валидации (пример).**

Для создания атрибута нам надо унаследовать свой класс от класса **ValidationAttribute** и реализовать его метод IsValid().

public class UserNameAttribute : ValidationAttribute

{

public override bool IsValid(object value)

{

if (value != null)

{

string userName = value.ToString();

if (!userName.StartsWith("T"))

return true;

else

this.ErrorMessage = "Имя не должно начинаться с буквы T";

}

return false;

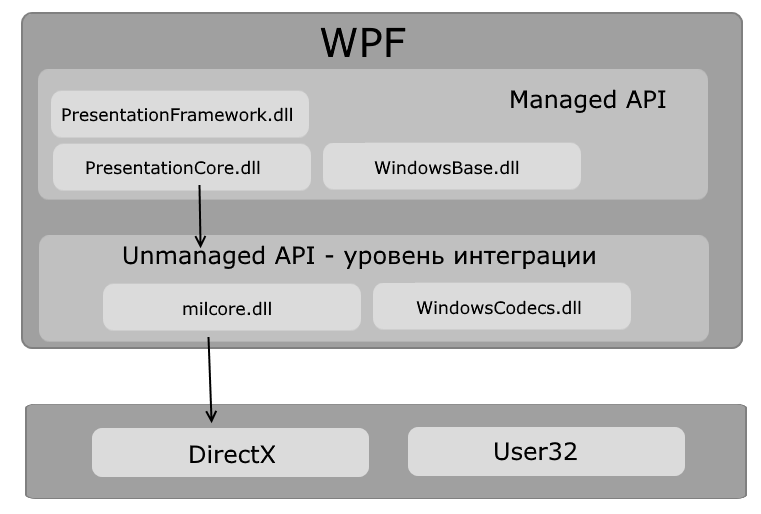
}

}

**7.WPF -Windows Presentation Foundation. Архитектура. Объектная модель.Отличие от WinForms.**

Технология WPF (Windows Presentation Foundation) является часть экосистемы платформы .NET и представляет собой подсистему для построения графических интерфейсов.

Схематически архитектуру WPF можно представить следующим образом:

Как видно на схеме, WPF разбивается на два уровня: managed API и unmanaged API (уровень интеграции с DirectX). Managed API (управляемый API-интерфейс) содержит код, исполняемый под управлением общеязыковой среды выполнения .NET - Common Language Runtime. Этот API описывает основной функционал платформы WPF



Ключевые возможности

Веб

-подобная модель компоновки

-Богатая модель рисования

-Развитая текстовая модель

-Поддержка аудио и видео

-Приложения на основе страниц

-Декларативный пользовательский интерфейс(XAML) (сочетание)

-Стили и шаблоны. Команды

**8.WPF. Контейнеры компоновки**

**Canvas**

Элемент управления Canvas поддерживает абсолютное позиционирование и предоставляет минимум встроенных возможностей макета для содержащихся в нем элементов управления. Элемент управления Canvas позволяет размещать содержащиеся в нем элементы управления со сдвигом относительно любого угла панели.

<Canvas Background="LightBlue">

<Label Canvas.Top="30" Canvas.Left="20">Enter here:</Label>

<TextBox Canvas.Top="30" Canvas.Left="120" Width="100" />

<Button Canvas.Top="70" Canvas.Left="130" Content="Click Me!" Padding="5" />

</Canvas>

DockPanel

Элемент управления DockPanel обеспечивает поддержку закрепления для упрощения расположения панелей инструментов и других элементов управления, которые требуется закрепить вдоль одной из границ панели. Элемент управления DockPanel предоставляет свойство с именем DockStyle для содержащихся в нем элементов управления, которое определяет, как они будут расположены.

<DockPanel>

<Border Height="25" Background="AliceBlue" DockPanel.Dock="Top">

<TextBlock>Menu</TextBlock>

</Border>

<Border Height="25" Background="Aqua" DockPanel.Dock="Top">

<TextBlock>Toolbar</TextBlock>

</Border>

<Border Height="30" Background="LightSteelBlue" DockPanel.Dock="Bottom">

<TextBlock>Status</TextBlock>

</Border>

<Border Height="80" Background="Azure" DockPanel.Dock="Left">

<TextBlock>Left Side</TextBlock>

</Border>

<Border Background="HotPink">

<TextBlock>Remaining Part</TextBlock>

</Border>

</DockPanel>

**Grid**

Наиболее знакомым контейнерным элементом управления является элемент управления Grid. По умолчанию каждое новое окно Window, открытое в Windows Presentation Foundation (WPF) для Visual Studio (конструктор), включает элемент управления Grid. Элемент управления Grid позволяет позиционировать элементы управления внутри ячеек, задаваемых пользователем. Элементы управления, помещенные в ячейки, поддерживают фиксированное поле между двумя или более краями элемента управления и краями ячеек при изменении размера окна Window.

<Grid ShowGridLines="True">

<Grid.ColumnDefinitions>

<ColumnDefinition />

<ColumnDefinition />

</Grid.ColumnDefinitions>

<Grid.RowDefinitions>

<RowDefinition />

<RowDefinition />

<RowDefinition />

</Grid.RowDefinitions>

<Label Grid.Column="0" Grid.ColumnSpan="2" Grid.Row="0" VerticalAlignment="Center" HorizontalAlignment="Center" Content="Title" />

<Label Grid.Column="0" Grid.Row="1" VerticalAlignment="Center" Content="Firstname:" Margin="10" />

<TextBox Grid.Column="1" Grid.Row="1" Width="100" Height="30" />

<Label Grid.Column="0" Grid.Row="2" VerticalAlignment="Center" Content="Lastname:" Margin="10" />

<TextBox Grid.Column="1" Grid.Row="2" Width="100" Height="30" />

</Grid>

**StackPanel**

Элемент управления StackPanel располагает содержащиеся в нем элементы управления либо в вертикальном столбце, либо в горизонтальной строке, в зависимости от значения свойства Orientation.

<StackPanel Orientation="Vertical">

<Label>Label</Label>

<TextBox>TextBox</TextBox>

<CheckBox>CheckBox</CheckBox>

<CheckBox>CheckBox</CheckBox>

<ListBox>

<ListBoxItem>ListBoxItem One</ListBoxItem>

<ListBoxItem>ListBoxItem Two</ListBoxItem>

</ListBox>

<Button>Button</Button>

</StackPanel>

**WrapPanel**

Элемент управления WrapPanel аналогичен элементу управления StackPanel в том, что в нем элементы управления располагаются в столбце или в строке в зависимости от свойства Orientation. В дополнение к подобному расположению, элемент управления WrapPanel поддерживает возможность переноса в следующий столбец или строку для содержащихся в нем элементов управления.

<WrapPanel>

<Button Width="100" Margin="5">Button</Button>

<Button Width="100" Margin="5">Button</Button>

<Button Width="100" Margin="5">Button</Button>

<Button Width="100" Margin="5">Button</Button>

<Button Width="100" Margin="5">Button</Button>

<Button Width="100" Margin="5">Button</Button>

<Button Width="100" Margin="5">Button</Button>

<Button Width="100" Margin="5">Button</Button>

</WrapPanel>

**9.XAML. Декларативная разработка. Простые и сложные свойства.**

XAML (eXtensible Application Markup Language) - язык разметки, используемый для инициализации объектов в технологиях на платформе .NET. Применительно к WPF (а также к Silverlight) данный язык используется прежде всего для создания пользовательского интерфейса декларативным путем.

С помощью атрибутов мы можем задать различные свойства кнопки. Height и Width являются простыми свойствами. Они хранят числовое значение. А например, свойства HorizontalAlignment или Background являются более сложными по своей структуре.

Чтобы выровнять кнопку по центру, применяется перечисление HorizontalAlignment, а для установки фонового цвета - класс SolidColorBrush. Хотя в коде XAML мы ничего такого не увидели и устанавливали эти свойства гораздо проще с помощью строк: Background="Red". Дело в том, что по отношению к коду XAML применяются специальные объекты - **type converter** или конвертеры типов, которые могут преобразовать значения из XAML к тем типам тех объектов, которые используются в коде C#.

**10.WPF. Понятие свойств зависимостей. Правила определения. Aattached properties**

Однако рассмотренные свойства элементов, как например, Width или Height, являются не просто стандартными свойствами языка C#. Они фактически скрывают **свойства зависимостей** или **dependency property**. Без свойств зависимостей были бы невозможны многие ключевые особенности WPF, как привязка данных, стили, анимация и т.д.

public class Phone : DependencyObject

{

public static readonly DependencyProperty TitleProperty;

static Phone()

{

TitleProperty = DependencyProperty.Register("Title", typeof(string), typeof(Phone));

}

public string Title

{

get { return (string)GetValue(TitleProperty); }

set { SetValue(TitleProperty, value); }

}

}

Прикрепляемые свойства (attached properties) также являются свойствами зависимостей с той разницей, что они определяются в одном классе, а применяются в другом. Например, при установке столбца или строки грида, в которых размещается элемент управления, используются свойства Grid.Row и Grig.Column, которые как раз и представляют прикрепляемые свойства. То есть эти свойства определены в классе Grid, но используются в других вложенных элементах.

<Button x:Name="button1" Content="Hello" Grid.Column="1" Grid.Row="0" />

**11.WPF.Обработка событий. Виды маршрутизации событий.**

События, возникнув на одном элементе, могут обрабатываться на другом. События могут подниматься и опускаться по дереву элементов.

Так, маршрутизируемые события делятся на три вида:

* **Прямые** (direct events) - они возникают и отрабытывают на одном элементе и никуда дальше не передаются. Действуют как обычные события.
* **Поднимающиеся** (bubbling events) - возникают на одном элементе, а потом передаются дальше к родителю - элементу-контейнеру и далее, пока не достигнет наивысшего родителя в дереве элементов.
* **Опускающиеся, туннельные** (tunneling events) - начинает отрабатывать в корневом элементе окна приложения и идет далее по вложенным элементам, пока не достигнет элемента, вызвавшего это событие.

Все маршрутизируемые события используют класс **RoutedEventArgs** (или его наследников), который представляет доступ к следующим свойствам:

* **Source**: элемент логического дерева, являющийся источником события.
* **OriginalSource**: элемент визуального дерева, являющийся источником события. Обычно то же самое, что и Source
* **RoutedEvent**: представляет имя события
* **Handled**: если это свойство установлено в True, событие не будет подниматься и опускаться, а ограничится непосредственным источником.

**12.WPF.Панели.Ресурсы статические и динамические**

**логических ресурсах**, которые могут представлять различные объекты - элементы управления, кисти, коллекции объектов и т.д. Логические ресурсы можно установить в коде XAML или в коде C# с помощью свойства Resources.

В чем смысл использования ресурсов? Они повышают эффективность: мы можем определить один раз какой-либо ресурс и затем многократно использовать его в различных местах приложения. В связи с этим улучшается поддержка - если возникнет необходимость изменить ресурс, достаточно это сделать в одном месте, и изменения произойдут глобально в приложении.

Свойство **Resources** представляет объект **ResourceDictionary** или словарь ресурсов, где каждый хранящийся ресурс имеет определенный ключ.

Причем каждый ресурс обязательно имеет свойство **x:Key**, которое и определяе ключ в словаре.

**13.WPF.Команды (Command). Модель команд. Встроенные команды**

**Команды** представляют механизм выполнения какой-нибудь задачи, например, копирования текста - когда мы нажимаем Ctrl+C, то мы копируем текст в буффер. В процессе копирования выполняется ряд действий, и все вместе эти действия объединяются в одну команду.

Все команды реализуют интерфейс System.Window.Input.ICommand:

public interface ICommand

{

event EventHandler CanExecuteChanged;

void Execute (object parameter);

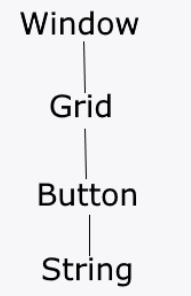
bool CanExecute (object parameter);

}

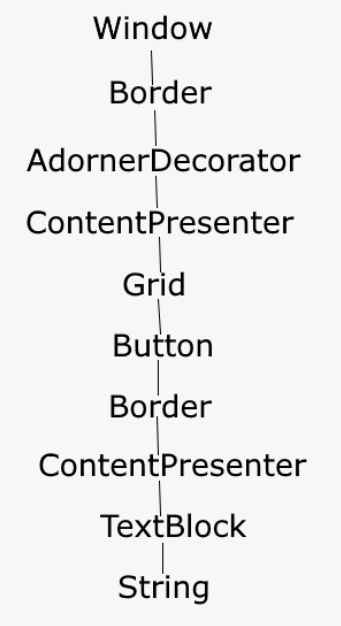
Некоторые элементы управления вводом умеют обрабатывать события команд самостоятельно. Например, класс TextBox обрабатывает команды Cut, Сору и Paste (а также команды Undo и Redo и часть команд класса EditingCommands, которые позволяют выделять текст и перемещать курсор в разные позиции)

**14.WPF.Логические и визуальные деревья. Шаблоны.**

Визуализация в WPF тесно связана с такими понятиями как логическое и визуальное дерево. Эти деревья являются своего рода каркасом приложения. Так мы можем представить приложение как некий набор вложенных элементов.



То есть в Window есть Grid, в Gride - элемент Button, в кнопке в качестве содержимого установлен некоторый текст в виде объекта String. В итоге получается некое дерево элементов, которое называется **логическим**. В WPF оно представлено классом **System.Windows.LogicalTreeHelper**. Логическое дерево имеет дело с визуализацией как таковой, оно образует модель доступа к дочерним элементам.



От него отличается **визуальное дерево**, представленное классом **System.Windows.Media.VisualTreeHelper**. Так, визуальное дерево для вышеприведенной разметки xaml будет выглядеть следующим образом. Визуальное дерево получается гораздо сложнее, оно показывает, как с визуальной точки зрения устроен элемент, из каких частей он состоит.

Визуальное дерево элемента управления опредлеляет, как будет выглядеть этот элемент или иными словами его **шаблон**. Шаблон элемента - это своего рода визуальный скелет элемента управления. Например, для элемента Button упрощенно шаблон выглядит следующим образом:

**15.WPF.Стили.**

Стили позволяют определить набор некоторых свойств и их значений, которые потом могут применяться к элементам в xaml. Стили хранятся в ресурсах и отделяют значения свойств элементов от пользовательского интерфейса. Также стили могут задавать некоторые аспекты поведения элементов с помощью триггеров. Аналогом стилей могут служить каскадные таблицы стилей (CSS), которые применяются в коде html на веб-страницах.

<Window.Resources>

<Style x:Key="BlackAndWhite">

<Setter Property="Control.FontFamily" Value="Verdana" />

<Setter Property="Control.Background" Value="Black" />

<Setter Property="Control.Foreground" Value="White" />

<Setter Property="Control.Margin" Value="10" />

</Style>

</Window.Resources>

Результат будет тот же, однако теперь мы избегаем не нужного повторения. Более того теперь мы можем управлять всеми нужными нам свойствами как единым целым - одним стилем.

Стиль создается как ресурс с помощью объекта **Style**, который представляет класс **System.Windows.Style**. И как любой другой ресурс, он обязательно должен иметь ключ. С помощью коллекции **Setters** определяется группа свойств, входящих в стиль. В нее входят объекты **Setter**, которые имеют следующие свойства:

**16.WPF.Триггеры. Типы триггеров**

Тригерры позволяют декларативно задать некоторые действия, которые выполняются при изменении свойств стиля. Существует три вида триггеров:

* **Триггеры свойств**: вызываются в ответ на изменения свойствами зависимостей своего значения

<Trigger Property="IsMouseOver" Value="True">

<Setter Property="FontSize" Value="14" />

<Setter Property="Foreground" Value="Red" />

</Trigger>

* **Триггеры данных**: вызываются в ответ на изменения значений любых свойств (они необязательно должны быть свойствами зависимостей)

<DataTrigger Binding="{Binding ElementName=checkBox1, Path=IsChecked}"

Value="True">

<Setter Property="IsEnabled" Value="False"/>

</DataTrigger>

* **Триггеры событий**: вызываются в ответ на генерацию событий

<EventTrigger RoutedEvent="Click">

<EventTrigger.Actions>

<BeginStoryboard>

<Storyboard>

<DoubleAnimation Storyboard.TargetProperty="Width" Duration="0:0:1" To="220" AutoReverse="True" />

<DoubleAnimation Storyboard.TargetProperty="Height" Duration="0:0:1" To="80" AutoReverse="True" />

</Storyboard>

</BeginStoryboard>

</EventTrigger.Actions>

</EventTrigger>

* **Мультитриггеры**: вызываются при выполнении ряда условий

<MultiTrigger>

<MultiTrigger.Conditions>

<Condition Property="IsMouseOver" Value="True" />

<Condition Property="IsPressed" Value="True" />

</MultiTrigger.Conditions>

<MultiTrigger.Setters>

<Setter Property="FontSize" Value="14" />

<Setter Property="Foreground" Value="Red" />

</MultiTrigger.Setters>

</MultiTrigger>

**17.WPF. Привязки (binding). Направление и обновление привязок. Интерфейс INotifyPropertyChanged**

Привязка подразумевает взаимодействие двух объектов: источника и приемника. Объект-приемник создает привязку к определенному свойству объекта-источника. В случае модификации объекта-источника, объект-приемник также будет модифицирован.

### **Режимы привязки**

Свойство **Mode** объекта Binding, которое представляет режим привязки, может принимать следующие значения:

* **OneWay**: свойство объекта-приемника изменяется после модификации свойства объекта-источника.
* **OneTime**: свойство объекта-приемника устанавливается по свойству объекта-источника только один раз. В дальнейшем изменения в источнике никак не влияют на объект-приемник.
* **TwoWay**: оба объекта - применки и источник могут изменять привязанные свойства друг друга.
* **OneWayToSource: объект-приемник, в котором объявлена привязка, меняет объект-источник.**
* **Default**: по умолчанию (если меняется свойство TextBox.Text, то имеет значение TwoWay, в остальных случаях OneWay).

Чтобы объект мог полноценно реализовать механизм привязки, нам надо реализовать в его классе интерфейс INotifyPropertyChanged. Когда объект класса изменяет значение свойства, то он через событие PropertyChanged извещает систему об изменении свойства. А система обновляет все привязанные объекты.

**18.Работа с данными СУБД в соединенной среде. Создание и настройка Command. Выполнение SQL, хранимой процедуры, операции с каталогами.**

**string** command = "SELECT \* FROM Employees";   
 SqlDataAdapter adapter = **new** SqlDataAdapter(command, connect);

**19.Работа с данными СУБДв соединенной среде. Создание и выполнение параметризированного запроса.**

**20.Классы DataAdapter. Connection. DataReader. Command. Transaction. Constraint.**

Чтобы извлечь записи из базы и использовать их для наполнения таблицы в DataSet, нужно использовать другой объект ADO.NET — **DataAdapter**. DataAdapter служит мостом между одним DataTable в DataSet и источником данных. Он включает все доступные команды для выполнения запросов и обновления источника данных.

string connectionString = ConfigurationManager.ConnectionStrings["DefaultConnection"].ConnectionString; Подключение

Класс DataReader позволяет читать данные, возвращенные командой SELECT, по одной строке за раз, в однонаправленном, доступном только для чтения потоке. Иногда это называют курсором. Использование DataReader — простейший путь получения данных, но ему недостает возможностей сортировки и связывания автономного объекта DataSet.

***Класс Command*** позволяет выполнить SQL-оператор любого типа. Хотя класс Command можно использовать для решения задач определения данных (таких как создание и изменение баз данных, таблиц и индексов), все же более вероятно его применение для выполнения задач манипулирования данными (вроде извлечения и обновления записей в таблице).

**21.Подключение к СУБД, открытие и чтение данных с помощью DataSet. Настройка нетипизированных объектов набора DataSet.**

string connectionString = ConfigurationManager.ConnectionStrings["DefaultConnection"].ConnectionString;

SqlCeDataAdapter da = new SqlCeDataAdapter();  
DataSet ds = new DataSet();  
DataTable dt = new DataTable();  
  
da.SelectCommand = new SqlCommand(@"SELECT \* FROM FooTable", connString);  
da.Fill(ds, "FooTable");  
dt = ds.Tables["FooTable"];  
  
foreach (DataRow dr in dt.Rows)  
{  
 MessageBox.Show(dr["Column1"].ToString());  
}

**22.Работа с данными в отсоединенной среде. Объекты DataTable. Создание DataAdapter.**

**string** connectionString =   
 WebConfigurationManager.ConnectionStrings["Northwind"].ConnectionString;  
 SqlConnection connect = **new** SqlConnection(connectionString);  
 **string** command = "SELECT \* FROM Employees";  
   
 SqlDataAdapter adapter = **new** SqlDataAdapter(command, connect);  
  
 *// Заполнить DataSet*  
 *DataSet dataset =* **new** DataSet();  
 adapter.Fill(dataset, "Employees");

**23.Entity Framework. Поянтие Entity Data Model (EDM). Архитектура.**

Entity Framework представляет ORM-технологию, которая позволяет абстрагироваться от структуры базы данных и может выполнять автоматически сопоставление таблиц и их данных с моделями классов, определенных разработчиком.

Другим ключевым понятием является **Entity Data Model**. Эта модель сопоставляет классы сущностей с реальными таблицами в БД.

Entity Data Model состоит из трех уровней: концептуального, уровень хранилища и уровень сопоставления (маппинга).

На концептуальном уровне происходит определение классов сущностей, используемых в приложении.

Уровень хранилища определяет таблицы, столбцы, отношения между таблицами и типы данных, с которыми сопоставляется используемая база данных.

Уровень сопоставления (маппинга) служит посредником между предыдущими двумя, определяя сопоставление между свойствами класса сущности и столбцами таблиц.

**24.Entity Framework. Подходы к проектированию. Database-First. Model-First.Code-First**

Entity Framework предполагает три возможных способа взаимодействия с базой данных:

* **Database first**: Entity Framework создает набор классов, которые отражают модель конкретной базы данных
* **Model first**: сначала разработчик создает модель базы данных, по которой затем Entity Framework создает реальную базу данных на сервере.
* **Code first**: разработчик создает класс модели данных, которые будут храниться в бд, а затем Entity Framework по этой модели генерирует базу данных и ее таблиц

**25.Entity Framework. Настройка конфигураций при CodeFirst. Способы получения связанных данных.**

Настройка конфигураций при Code First

-Аннотации (настройка сопоставления моделей и таблиц с помощью атрибутов)

-Fluent API (набор методов, которые определяются сопоставление между классами и их свойствами и таблицами и их столбцами)

modelBuilder.Entity<User>().Property(p=>p.Name).HasMaxLength(30);

modelBuilder.Entity<User>().HasKey(it => it.Login);

"жадная загрузка" или **eager loading**

using(DB db= new DB())

{IEnumerable<User> users = db.User.Include(p => p.Actor);

foreach( User p in users)

{

MessageBox.Show(p.Actor.Role);

}

}

"ленивая загрузка" или **lazy loading**

при первом обращении к объекту, если связанные данные не

нужны, то они не подгружаются. Однако при первом же обращении

к навигационному свойству эти данные автоматически

подгружаются из бд.

**explicit loading**("явная загрузка")

vart = db.Users.FirstOrDefault();

db.Entry(t).Collection("Actors").Load()

**26.Entity Framework. Аннотации. Соглашение конфигураций Fluent API. выше**

**27.Интерфейс работы с базами данных. Паттерн Repository**

Одним из наиболее часто используемых паттернов при работе с данными является паттерн 'Репозиторий'. Репозиторий позволяет абстрагироваться от конкретных подключений к источникам данных, с которыми работает программа, и является промежуточным звеном между классами, непосредственно взаимодействующими с данными, и остальной программой.

interface IRepository<T> : IDisposable

where T : class

{

IEnumerable<T> GetBookList(); // получение всех объектов

T GetBook(int id); // получение одного объекта по id

void Create(T item); // создание объекта

void Update(T item); // обновление объекта

void Delete(int id); // удаление объекта по id

void Save(); // сохранение изменений

}

public class SQLBookRepository : IRepository<Book>

{

private BookContext db;

public SQLBookRepository()

{

this.db = new BookContext();

}

public IEnumerable<Book> GetBookList()

{

return db.Books;

}

public Book GetBook(int id)

{

return db.Books.Find(id);

}

**28.Назначение и принцип использования паттерна Unit Of Work**

Паттерн Unit of Work позволяет упростить работу с различными репозиториями и дает уверенность, что все репозитории будут использовать один и тот же контекст данных.

Класс UnitOfWork предоставляет доступ к репозиториям через отдельные свойства и определяет общий контекст для обоих репозиториев.

Кроме того, данный класс содержит дополнительные методы Save() и Dispose(), которые в иной ситуации мы могли б определить в репозиториях. Но так как этот функционал будет общим для обоих репозиториев, то его лучше вынести в класс UnitOfWork.

**29.LINQ to Entities**

Хотя при работе с базой данных мы оперируем запросами LINQ, но база данных понимает только запросы на языке SQL. Поэтому между LINQ to Entities и базой данных есть проводник, который позволяет им взаимодействовать. Этим проводником является провайдер **EntityClient**.

И тот же запрос с помощью методов расширений LINQ:

using(PhoneContext db = new PhoneContext())

{

var phones = db.Phones.Where(p=> p.CompanyId == 1);

}

Оба запроса в итоге транслируются в одной выражение sql:

SELECT [Extent1].[Id] AS [Id],

[Extent1].[Name] AS [Name],

[Extent1].[Price] AS [Price],

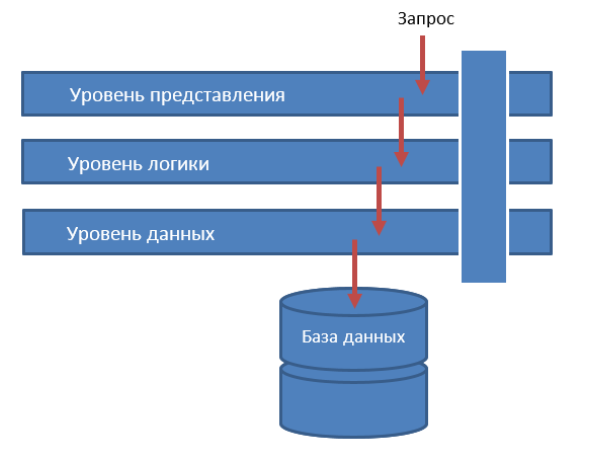
[Extent1].[CompanyId] AS [CompanyId]

FROM [dbo].[Phones] AS [Extent1]

WHERE 1 = [Extent1].[CompanyId]}

**30.Архитектурные стили проектирования. Многоуровневая архитектура**

Является одной из самых известных архитектур, в которой каждый слой выполняет определенную функцию. В зависимости от ваших нужд вы можете реализовать любое количество уровней, но слишком большое их количество приведет к чрезмерному усложнению системы. Часто выделяют три основных уровня: уровень представления, уровень логики и уровень данных.



**31.WPF. Архитектурный паттерн Model-View-ViewModel.**

Паттерн **MVVM (Model-View-ViewModel)** позволяет отделить логику приложения от визуальной части (представления). Данный паттерн является архитектурным, то есть он задает общую архитектуру приложения.

MVVM состоит из трех компонентов: модели (Model), модели представления (ViewModel) и представления (View).

### **Model**

Модель описывает используемые в приложении данные. Модели могут содержать логику, непосредственно связанную этими данными, например, логику валидации свойств модели. В то же время модель не должна содержать никакой логики, связанной с отображением данных и взаимодействием с визуальными элементами управления.

Нередко модель реализует интерфейсы INotifyPropertyChanged или INotifyCollectionChanged, которые позволяют уведомлять систему об изменениях свойств модели. Благодаря этому облегчается привязка к представлению, хотя опять же прямое взаимодействие между моделью и представлением отсутствует.

### **View**

View или представление определяет визуальный интерфейс, через который пользователь взаимодействует с приложением. Применительно к WPF представление - это код в xaml, который определяет интерфейс в виде кнопок, текстовых полей и прочих визуальных элементов.

Хотя окно (класс Window) в WPF может содержать как интерфейс в xaml, так и привязанный к нему код C#, однако в идеале код C# не должен содержать какой-то логики, кроме разве что конструктора, который вызывает метод InitializeComponent и выполняет начальную инициализацию окна. Вся же основная логика приложения выносится в компонент ViewModel.

Однако иногда в файле связанного кода все может находиться некоторая логика, которую трудно реализовать в рамках паттерна MVVM во ViewModel.

Представление не обрабатывает события за редким исключением, а выполняет действия в основном посредством команд.

### **ViewModel**

ViewModel или модель представления связывает модель и представление через механизм привязки данных. Если в модели изменяются значения свойств, при реализации моделью интерфейса INotifyPropertyChanged автоматически идет изменение отображаемых данных в представлении, хотя напрямую модель и представление не связаны.

ViewModel также содержит логику по получению данных из модели, которые потом передаются в представление. И также VewModel определяет логику по обновлению данных в модели.

Поскольку элементы представления, то есть визуальные компоненты типа кнопок, не используют события, то представление взаимодействует с ViewModel посредством команд.

Например, пользователь хочет сохранить введенные в текстовое поле данные. Он нажимает на кнопку и тем самым отправляет команду во ViewModel. А ViewModel уже получает переданные данные и в соответствии с ними обновляет модель.

Итогом применения паттерна MVVM является функциональное разделение приложения на три компонента, которые проще разрабатывать и тестировать, а также в дальнейшем модифицировать и поддерживать.

**32.Этапы разработки ПО.**

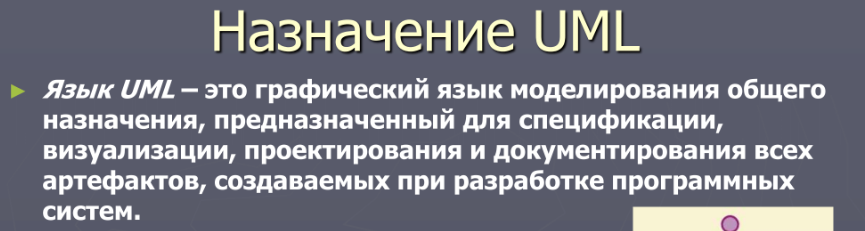
Типовой проект включает в себя следующие этапы разработки программного обеспечения:

* анализ требований к проекту;
* проектирование;
* реализация;
* тестирование продукта;
* внедрение и поддержка.



**33.Понятие и назначение UML. Классификация диаграмм (использования, последовательности, коммуникаций).**





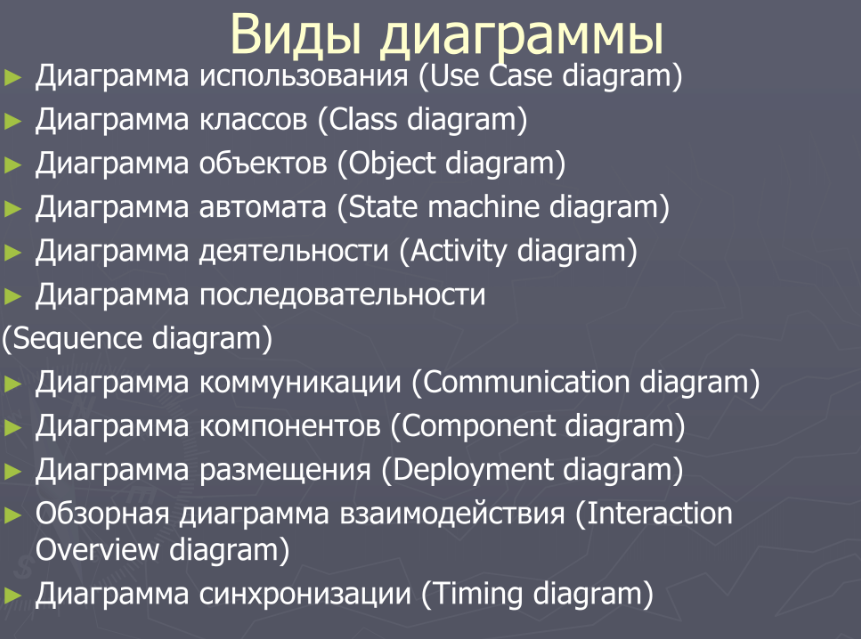


Диаграмма компонентов UML представляет собой полностью статическую структурную диаграмму. Предназначается она для того, чтобы продемонстрировать разбиение определенной программной системы на разнообразные структурные компоненты, а также связи между ними. Диаграмма компонентов UML в качестве таковых может использовать всевозможные модели, библиотеки, файлы, пакеты, исполняемые файлы и еще множество других элементов.

Диаграмма деятельности UML отображает разложение определенной деятельности на несколько составных частей. В данном случае понятием «деятельность» называется спецификация определенного исполняемого поведения в виде параллельного, а также координированного последовательного выполнения различных подчиненных элементов – вложенных типов деятельности и различных действий, объединенных потоками, идущими от выходов определенного узла к входам другого.

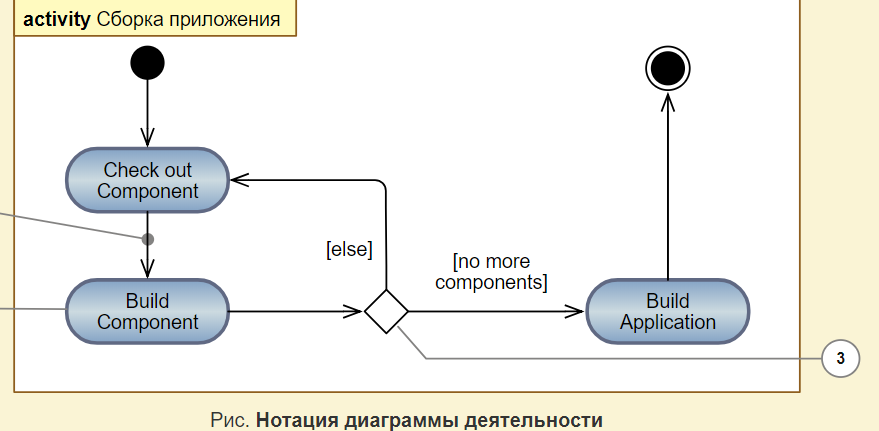
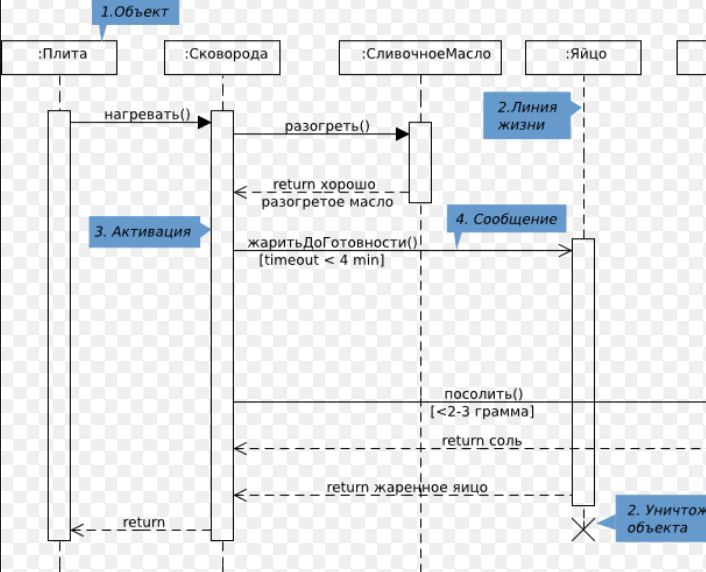
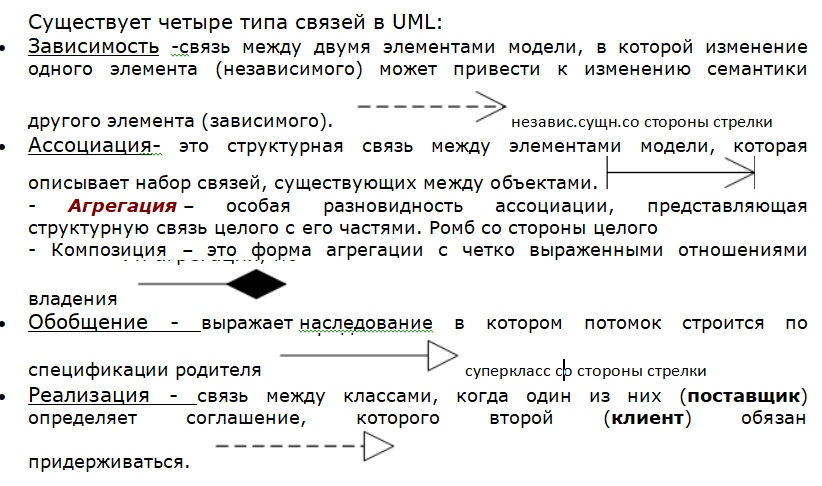


Диаграмма последовательности UML демонстрирует взаимодействия между несколькими объектами, которые упорядочиваются в соответствии с временем их проявления. На такой диаграмме отображается упорядоченное во времени взаимодействие между несколькими объектами. В частности, на ней отображаются все объекты, которые принимают участие во взаимодействии, а также полная последовательность обмениваемых ими сообщений.



**34.Диаграммы классов. Виды отношений между классами.**

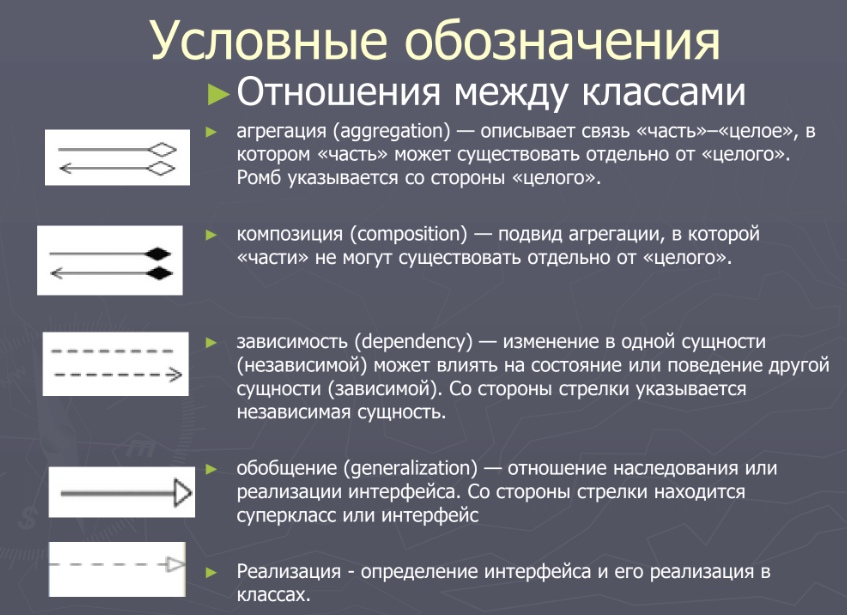
***Класс*** – это описание набора объектов с одинаковыми атрибутами, операциями, связями и семантикой.   
Диаграмма классов UML представляет собой статическую структурную диаграмму, предназначенную для описания структуры системы, а также демонстрации атрибутов, методов и зависимостей между несколькими различными классами.

[](https://vk.com/photo90330416_456248257)

Графически класс изображается в виде прямоугольника, разделенного на 3 блока горизонтальными линиями:

* имя класса
* атрибуты (свойства) класса
* операции (методы) класса.

Существует четыре типа связей в UML:

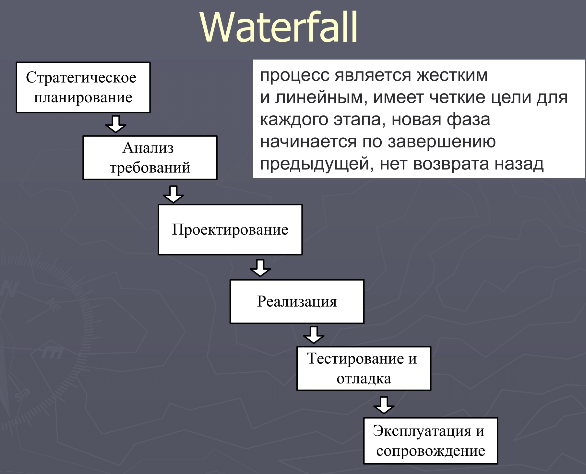
* Зависимость (– семантически представляет собой связь между двумя элементами модели, в которой изменение одного элемента (независимого) может привести к изменению семантики другого элемента (зависимого). )
* Ассоциация (– это структурная связь между элементами модели, которая описывает набор связей, существующих между объектами)
* Обобщение (– ***обобщение*** – выражает специализацию или ***наследование***, в котором специализированный элемент (потомок) строится по спецификациям обобщенного элемента (родителя))
* Реализация (– ***реализация*** – это семантическая связь между классами, когда один из них (**поставщик**) определяет соглашение, которого второй (**клиент**) обязан придерживаться)
* [](https://vk.com/photo90330416_456248254)

**35.Методологии разработки ПО (XP, Agile, SADT, RUP, Waterfall,Spiral).**

Экстремальное программирование — возможность вести разработку в условиях постоянно меняющихся требований. Вот несколько признаков:

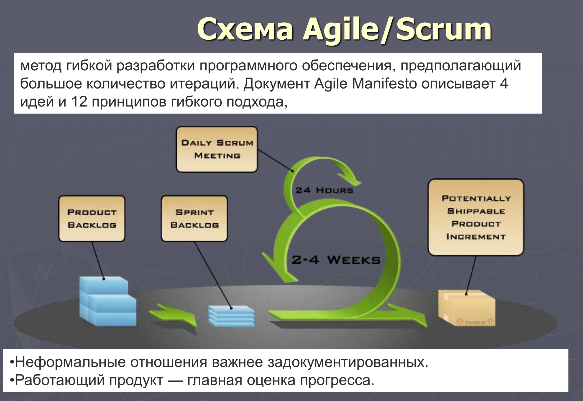
* Игра в планирование. В начале проекта есть только приблизительный план, после каждой итерации его чёткость возрастает.
* Высокая частота релизов. Новая версия продукта имеет незначительные изменения по сравнению с предыдущей, но время на выпуск при этом минимально.
* Контакт с клиентом. Для удовлетворения требований конечной аудитории необходимо оперативное реагирование на замечания и пожелания.
* Рефакторинг. Улучшение качества кода без уменьшения функциональности.
* Стандарт выполнения кода. Или применяются общие правила, или разногласия в оформлении не подлежат обсуждению и критике.
* Коллективная ответственность. Несмотря на то, что каждый член команды выполняет свой участок работ, за код в целом отвечает весь коллектив.

Модель Waterfall относится к классическому пониманию разработки ПО. Весь процесс является жестким и линейным, имеет четкие цели для каждого этапа, новая фаза начинается по завершению предыдущей, нет возврата назад. Преимущества водопадной методологии — децентрализация и строгий контроль над сроками и качеством исполнения.

* 

Agile — метод гибкой разработки программного обеспечения, предполагающий большое количество итераций. Документ Agile Manifesto описывает 4 идей и 12 принципов гибкого подхода, коротко его можно описать всего двумя пунктами:

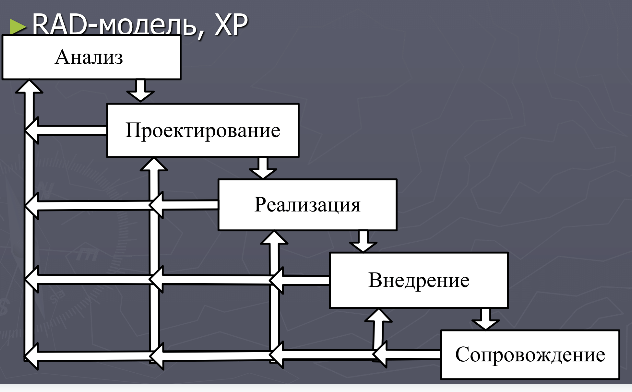
* Неформальные отношения важнее задокументированных. То есть устные договоренности между сотрудниками, между заказчиком и исполнителем важнее всего, что отражено в планах, договорах и техническом задании. Иначе говоря, клиент всегда прав.
* Работающий продукт — главная оценка прогресса. Важны не инструменты, решения, производительность и изящество, а тот факт, что все запланированные возможности реализованы.



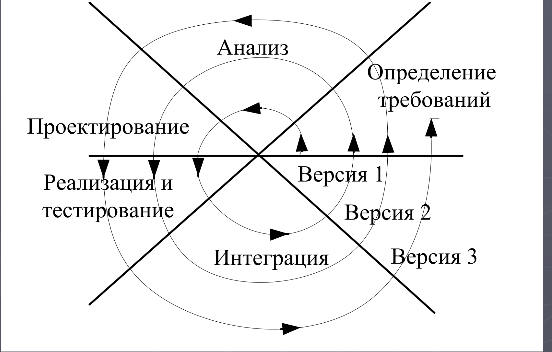
RUP — итеративный подход, который решает проблемы, которые есть у Waterfall. Чем хорош RUP:

* Учитывает изменяющиеся требования. Как бы ни был хорош руководитель проекта, учесть всё в начале невозможно.
* Интеграция функций происходит постепенно, то есть каждая «деталь» проходит цикл разработки, проверки и внедрения в проект. Как следствие, снижаются риски и стоимость производства.
* Ранний выпуск продукта. ПО выходит с уменьшенной функциональностью, чтобы занять нишу на рынке и противостоять конкурентам, после чего обрастает «мясом».
* Повторное использование. При наращивании функциональности проще выделить типовые решения, которые сократят разработку.
* Постоянное обучение. Из-за частых итераций разработчики не имеют больших пауз между доработкой кода, поэтому профессиональный рост происходит плавно и безболезненно.
* Постоянное улучшение продукта. Итерации позволяют оценить проект не только с точки зрения соответствия плану и ТЗ, но и найти пути увеличения эффективности и качества продукта.

RAD — методология, которая во главу угла ставит скорость и удобство разработки. Одно из главных условий — использование языка быстрой разработки. Это название абстрактного языка программирования, с помощью которого программист способен решать задачи быстрее, чем с представителями третьего поколения (C / C ++, Pascal или Fortran). RAD предполагает использование целого комплекса инструментов помимо языка быстрой разработки: системы сбора требований, среды разработки, фреймворки, программы для группового общения, ПО для тестирования.



## **Spiral** Модель спирального жизненного цикла — это сложная организация жизненного цикла ПО, которая фокусируется на раннем выявлении и уменьшении проектных рисков. Разработка начинается в небольшом масштабе, решаются локальные задачи, оцениваются риски и пути их уменьшения. Следующий шаг охватывает более комплексные задачи — следующий виток спирали.



 Функциональная модель SADT отображает функциональную структуру объекта, т.е. производимые им действия и связи между этими действиями. Основные элементы этой методологии основываются на следующих концепциях.  Функциональная модель SADT отображает функциональную структуру объекта, т.е. производимые им действия и связи между этими действиями. Основные элементы этой методологии основываются на следующих концепциях.

**36.Понятие рефакторинга. Антипаттерны.**

В основе поддержки читаемости и модифицируемости кода лежит рефакторинг — как в частном случае структур (frameworks), так и для программного обеспечения в целом. **Рефакторинг** представляет собой процесс такого изменения программной системы, при котором не меняется внешнее поведение кода, но улучшается его внутренняя структура. Это способ систематического приведения кода в порядок, при котором шансы появления новых ошибок минимальны. В сущности, при проведении рефакторинга кода вы улучшаете его дизайн уже после того, как он написан.

* Рефакторинг улучшает композицию программного обеспечения
* Рефакторинг облегчает понимание программного обеспечения
* Рефакторинг помогает найти ошибки
* Рефакторинг позволяет быстрее писать программы

## Анти-паттерны — это шаблоны ошибок, которые совершаются при решении различных задач.

## -Программирование копи-пастом (Copy and Paste Programming) Данный анти-паттерн является, наверное, самым древним в программировании. Когда от программиста требуется написание двух схожих функций, самым «простым» решением является написание одной функции, её копирование и внесение некоторых изменений в копию.

## -Спагетти-код (Spaghetti code) Спагетти-код — слабо структурированная и плохо спроектированная система, запутанная и очень сложная для понимания. Такой код так же очень часто содержит в себе множество примеров анти-паттерна программирования копи-пастом.

## -Золотой молоток (Golden hammer) Золотой молоток — уверенность в полной универсальности любого решения. На практике, это — применение одного решения (чаще всего какого-либо одного паттерна проектирования) для всех возможных и невозможных задач.

## -Магические числа (Magic numbers) Магическое число — константа, использованная в коде для чего либо (чаще всего — идентификации данных), само число не несёт никакого смысла без соответствующего комментария. Числа не несут абсолютно никакой семантики.

## -Изобретение велосипеда (Reinventing the wheel) Смысл этого анти-паттерна в том, что программист разрабатывает собственное решение для задачи, для которой уже существуют решения, очень часто лучшие чем придуманное программистом.

**37.Паттерны проектирования. Классификация. Порождающие(Abstarct Factory, Factory Method)**

Паттерн представляет определенный способ построения программного кода для решения часто встречающихся проблем проектирования. В данном случае предполагается, что есть некоторый набор общих формализованных проблем, которые довольно часто встречаются, и паттерны предоставляют ряд принципов для решения этих проблем.

### **Порождающие паттерны**

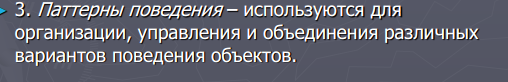
Порождающие паттерны —Эти паттерны отвечают за удобное и безопасное создание новых объектов или даже целых семейств объектов.. Среди них выделяются следующие:

* **Абстрактная фабрика (Abstract Factory)**
* **Строитель (Builder)**
* **Фабричный метод (Factory Method)**
* **Прототип (Prototype)**
* **Одиночка (Singleton)**

Другая группа паттернов - **структурные паттерны** - рассматривает, как классы и объекты образуют более крупные структуры - более сложные по характеру классы и объекты. К таким шаблонам относятся:

* **Адаптер (Adapter)**
* **Мост (Bridge)**
* **Компоновщик (Composite)**
* **Декоратор (Decorator)**
* **Фасад (Facade)**
* **Приспособленец (Flyweight)**
* **Заместитель (Proxy)**

Третья группа паттернов называются **поведенческими** - они определяют алгоритмы и взаимодействие между классами и объектами, то есть их поведение. Среди подобных шаблонов можно выделить следующие:

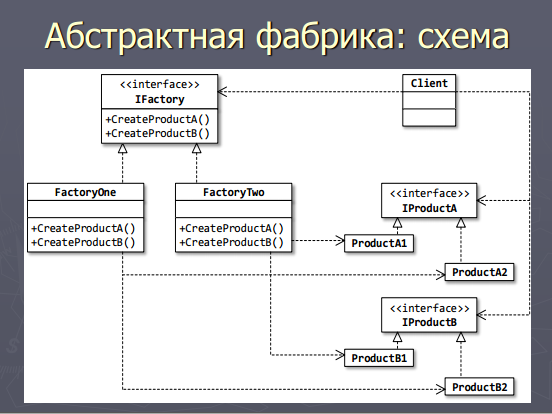


* **Цепочка обязанностей (Chain of responsibility)**
* **Команда (Command)**
* **Интерпретатор (Interpreter)**
* **Итератор (Iterator)**
* **Посредник (Mediator)**
* **Хранитель (Memento)**
* **Наблюдатель (Observer)**
* **Состояние (State)**
* **Стратегия (Strategy)**
* **Шаблонный метод (Template method)**
* **Посетитель (Visitor)**

### **Когда использовать абстрактную фабрику**



* Абстрактные классы **AbstractProductA** и **AbstractProductB** определяют интерфейс для классов, объекты которых будут создаваться в программе.
* Конкретные классы **ProductA1 / ProductA2** и **ProductB1 / ProductB2** представляют конкретную реализацию абстрактных классов
* Абстрактный класс фабрики **AbstractFactory** определяет методы для создания объектов. Причем методы возвращают абстрактные продукты, а не их конкретные реализации.
* Конкретные классы фабрик **ConcreteFactory1** и **ConcreteFactory2** реализуют абстрактные методы базового класса и непосредственно определяют какие конкретные продукты использовать
* Класс клиента **Client** использует класс фабрики для создания объектов. При этом он использует исключительно абстрактный класс фабрики AbstractFactory и абстрактные классы продуктов AbstractProductA и AbstractProductB и никак не зависит от их конкретных реализаций



### 

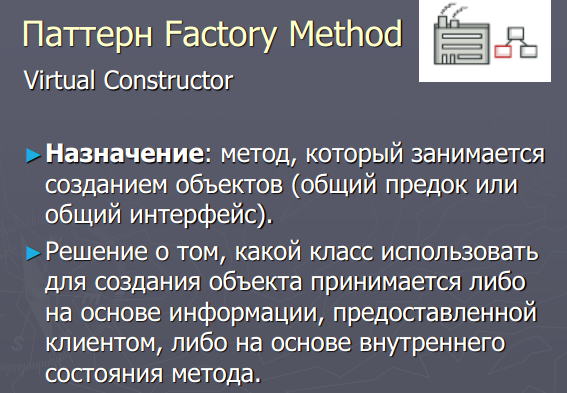
**Фабричный метод** — это порождающий паттерн проектирования, который определяет общий интерфейс для создания объектов в суперклассе, позволяя подклассам изменять тип создаваемых объектов.

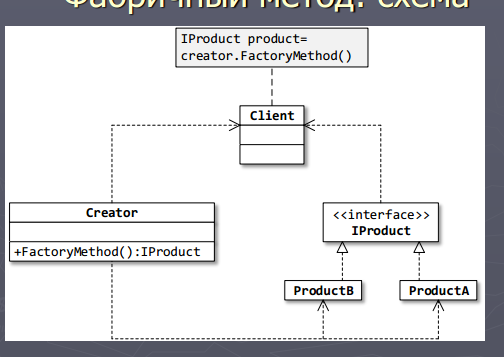
### **Когда надо применять паттерн**

* Когда заранее неизвестно, объекты каких типов необходимо создавать

Благодаря этому, код производства можно расширять, не трогая основной. Так, чтобы добавить поддержку нового продукта, вам нужно создать новый подкласс и определить в нём фабричный метод, возвращая оттуда экземпляр нового продукта.

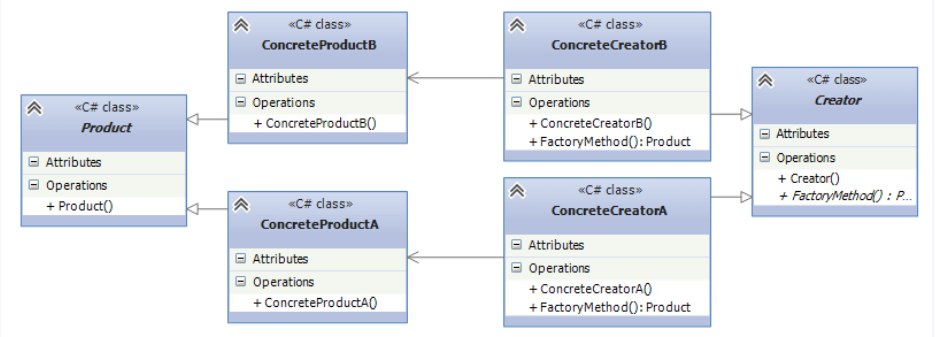
* Когда система должна быть независимой от процесса создания новых объектов и расширяемой: в нее можно легко вводить новые классы, объекты которых система должна создавать.
* Когда создание новых объектов необходимо делегировать из базового класса классам наследникам

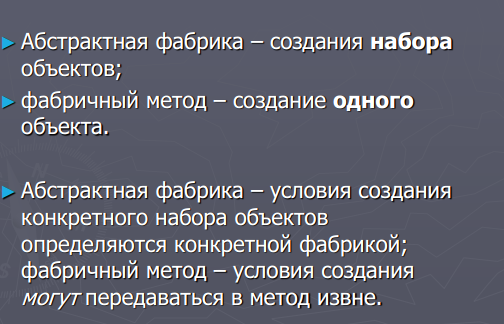




* Абстрактный класс **Product** определяет интерфейс класса, объекты которого надо создавать.
* Конкретные классы **ConcreteProductA** и **ConcreteProductB** представляют реализацию класса Product. Таких классов может быть множество
* Абстрактный класс **Creator** определяет абстрактный фабричный метод FactoryMethod(), который возвращает объект Product.
* Конкретные классы **ConcreteCreatorA** и **ConcreteCreatorB** - наследники класса Creator, определяющие свою реализацию метода FactoryMethod(). Причем метод FactoryMethod() каждого отдельного класса-создателя возвращает определенный конкретный тип продукта. Для каждого конкретного класса продукта определяется свой конкретный класс создателя.

Таким образом, класс Creator делегирует создание объекта Product своим наследникам. А классы ConcreteCreatorA и ConcreteCreatorB могут самостоятельно выбирать какой конкретный тип продукта им создавать.



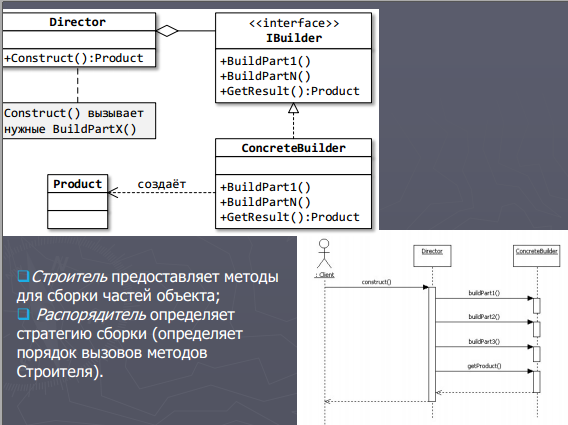


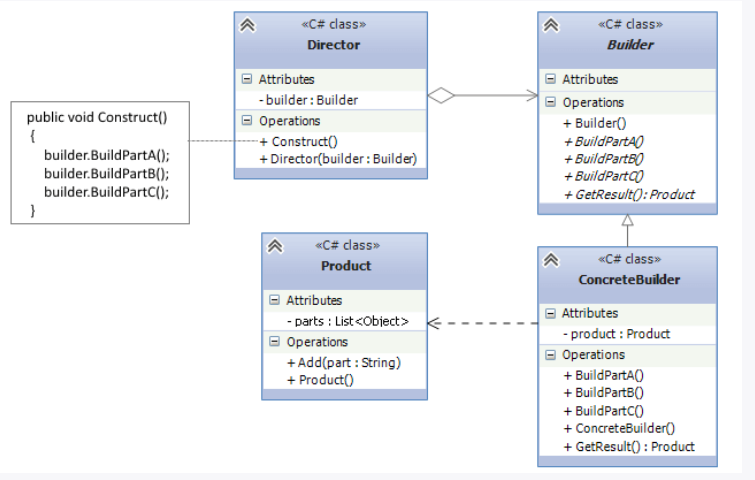
**38.Порождающие паттерны проектирования :Builder, Prototype.**

### **Когда использовать паттерн Строитель?**

**Строитель** — это порождающий паттерн проектирования, который позволяет создавать сложные объекты пошагово. Строитель даёт возможность использовать один и тот же код строительства для получения разных представлений объектов.

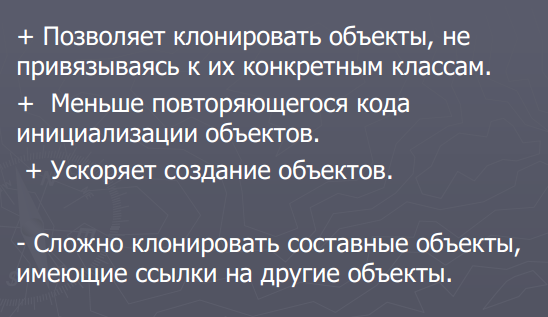
Паттерн предлагает разбить процесс конструирования объекта на отдельные шаги (например, построитьСтены, вставитьДвери и другие). Чтобы создать объект, вам нужно поочерёдно вызывать методы строителя. Причём не нужно запускать все шаги, а только те, что нужны для производства объекта определённой конфигурации.

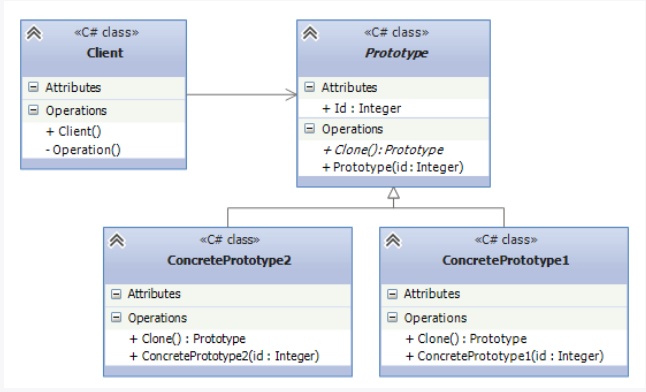
* Когда процесс создания нового объекта не должен зависеть от того, из каких частей этот объект состоит и как эти части связаны между собой
* Когда необходимо обеспечить получение различных вариаций объекта в процессе его создания
* **Product**: представляет объект, который должен быть создан. В данном случае все части объекта заключены в списке parts.
* **Builder**: определяет интерфейс для создания различных частей объекта Product
* **ConcreteBuilder**: конкретная реализация Buildera. Создает объект Product и определяет интерфейс для доступа к нему
* **Director**: распорядитель - создает объект, используя объекты Builder
* 



**Прототип** — это порождающий паттерн проектирования, который позволяет копировать объекты, не вдаваясь в подробности их реализации.

* Когда конкретный тип создаваемого объекта должен определяться динамически во время выполнения
* Когда нежелательно создание отдельной иерархии классов фабрик для создания объектов-продуктов из параллельной иерархии классов (как это делается, например, при использовании паттерна Абстрактная фабрика)
* Когда клонирование объекта является более предпочтительным вариантом нежели его создание и инициализация с помощью конструктора. Особенно когда известно, что объект может принимать небольшое ограниченное число возможных состояний.





**39.Порождающие паттерны проектирования: Singleton–4 реализации**

Одиночка (Singleton, Синглтон) - порождающий паттерн, который гарантирует, что для определенного класса будет создан только один объект, а также предоставит к этому объекту точку доступа.

Когда надо использовать Синглтон? Когда необходимо, чтобы для класса существовал только один экземпляр

Синглтон позволяет создать объект только при его необходимости. Если объект не нужен, то он не будет создан. В этом отличие синглтона от глобальных переменных

Классическая реализация данного шаблона проектирования на C# выглядит следующим образом:

class Singleton

{

private static Singleton instance;

private Singleton()

{}

public static Singleton getInstance()

{

if (instance == null)

instance = new Singleton();

return instance;

}

}

Чтобы решить проблему многопоточности, перепишем класс синглтона следующим образом:

class OS

{

private static OS instance;

public string Name { get; private set; }

private static object syncRoot = new Object();

protected OS(string name)

{

this.Name = name;

}

public static OS getInstance(string name)

{

if (instance == null)

{

lock (syncRoot)

{

if (instance == null)

instance = new OS(name);

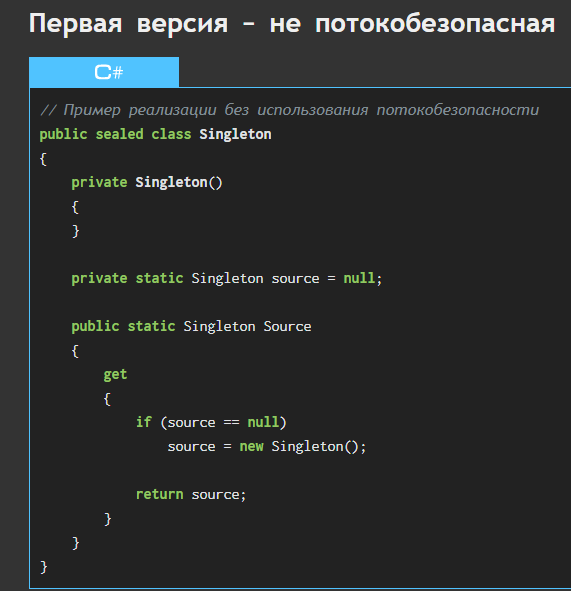
}

}

return instance;

}

}



Вышеуказанная реализация не является потокобезопасной. Два разных потока могли бы пройти условие if (source == null), создав два экземпляра, что нарушает принцип Singleton. Обратите внимание, что на самом деле экземпляр, возможно, уже был создан до того, как условие будет пройдено, но модель памяти не гарантирует, что новое значение экземпляра будет видно другим потокам, если не будет приняты соответствующие блокировки.

**Потокобезопасная реализация без использования lock**

public class Singleton

{

private static readonly Singleton instance = new Singleton();

public string Name { get; private set; }

private Singleton()

{

Name = System.Guid.NewGuid().ToString();

}

public static Singleton GetInstance()

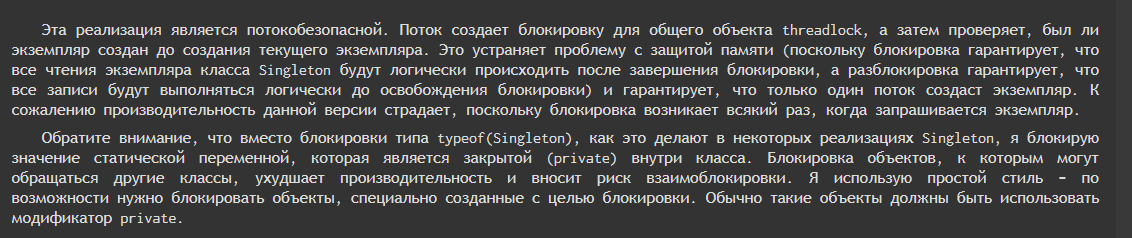
{

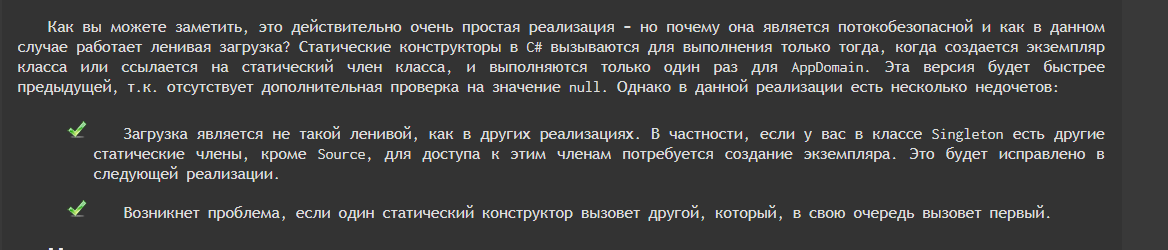
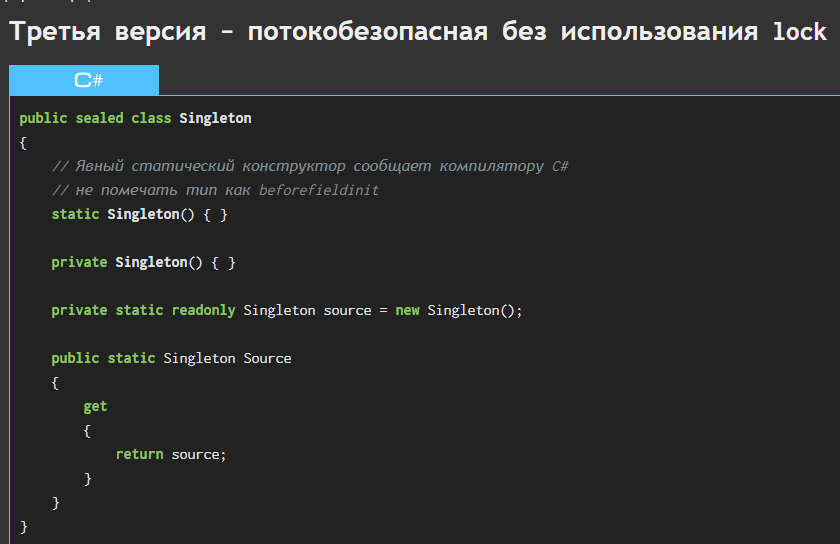
return instance;

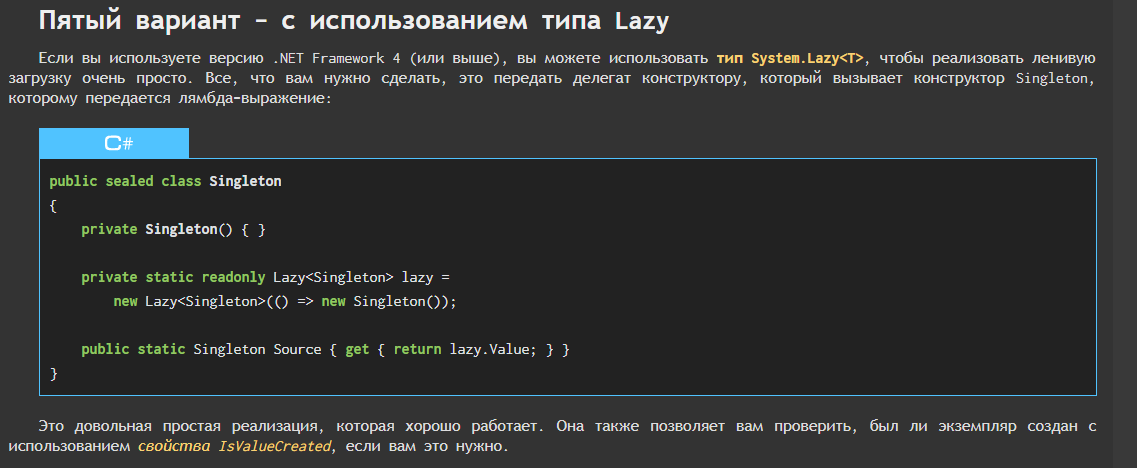
}

}









**Реализация через класс Lazy<T>**

public class Singleton

{

private static readonly Lazy<Singleton> lazy =

new Lazy<Singleton>(() => new Singleton());

public string Name { get; private set; }

private Singleton()

{

Name = System.Guid.NewGuid().ToString();

}

public static Singleton GetInstance()

{

return lazy.Value;

}

}

**40.Порождающие паттерны проектирования: Lazy initialization, Object pool**

Шаблон "Отложенная инициализация"(ленивая) позволяет отложить действия, связанные с созданием объекта, до момента, когда непосредственно потребуется результат этих действий. Таким образом, инициализация выполняется «по требованию», а не заблаговременно. Аналогичная идея находит применение в самых разных областях: например, [компиляция «на лету»](https://ru.wikipedia.org/wiki/JIT) . Частный случай ленивой инициализации — создание объекта в момент обращения к нему — является одним из [порождающих шаблонов проектирования](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D1%80%D0%BE%D0%B6%D0%B4%D0%B0%D1%8E%D1%89%D0%B8%D0%B5_%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD%D1%8B_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F). Как правило, он используется в сочетании с такими шаблонами как [Фабричный метод](https://ru.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B1%D1%80%D0%B8%D1%87%D0%BD%D1%8B%D0%B9_%D0%BC%D0%B5%D1%82%D0%BE%D0%B4_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)), [Одиночка](https://ru.wikipedia.org/wiki/%D0%9E%D0%B4%D0%B8%D0%BD%D0%BE%D1%87%D0%BA%D0%B0_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)) и [Заместитель](https://ru.wikipedia.org/wiki/%D0%97%D0%B0%D0%BC%D0%B5%D1%81%D1%82%D0%B8%D1%82%D0%B5%D0%BB%D1%8C_(%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)).

Данный шаблон используется если

* создание объекта связано с большими затратами ресурсов;
* есть вероятность, что объект или его часть не будут использованы.

Достоинства : 1) Инициализация выполняется только в тех случаях, когда она действительно необходима; 2) ускоряется начальная инициализация.

Недостатки: 1) Невозможно явным образом задать порядок инициализации объектов; 2) возникает задержка при первом обращении к объекту.

Реализация: - Использование Lazy<T> позволяет задержать создание объекта до первого обращения к свойству Value. Для создания объекта используется либо конструктор типа T, либо функция, передаваемая конструктору Lazy<T>. <https://metanit.com/sharp/tutorial/20.1.php> - Класс System.Lazy<T> служит для поддержки отложенной инициализации объектов. Он содержит свойство IsValueCreated и свойство Value типа T. <http://andrey.moveax.ru/post/patterns-oop-creational-lazy-initialization> ------------------------------------------------------------------------------------------------------------------------------------Пул объектов (Object pool) - это набор готовых к использованию объектов. Если клиенту требуется объект, он берёт его из пула, минуя процесс порождения. Когда объект больше не нужен, он не уничтожается, а возвращается в пул (при этом состояние объекта сбрасывается до начального). \*) Шаблон, в основном, применяется для повышения производительности: объекты часто создаются и уничтожаются; в системе существует ограниченное количество объектов типа, хранимого в Пуле; создание и/или уничтожение объекта являются очень затратными операциями.

Если в пуле нет ни одного свободного объекта, возможна одна из трёх стратегий: 1)Расширение пула. 2)Отказ в создании объекта, аварийная остановка. 3)В случае [многозадачной](https://ru.wikipedia.org/wiki/%D0%9C%D0%BD%D0%BE%D0%B3%D0%BE%D0%B7%D0%B0%D0%B4%D0%B0%D1%87%D0%BD%D0%BE%D1%81%D1%82%D1%8C) системы можно подождать, пока один из объектов не освободится.

Пример работы: Пул потоков, для работы с которым используется класса System.Threading. ThreadPool.

Реализация:

* определяем интерфейс *IPoolable,* который должны реализовывать объекты для взаимодействия с Пулом;
* разрабатываем архитектуру работы Пула с объектами, включая:
* их создание, хранение и удаление;
* сброс в исходное состояние при возврате, используя интерфейс *IPoolable*;
* реакцию на отсутствие свободных объектов;
* реализуем Пул;
* в клиентском коде для получения объекта обращаемся к Пулу, а после использования обязательно возвращаем объект обратно.

**41.Структурные паттерны: Adapter, Decorator**

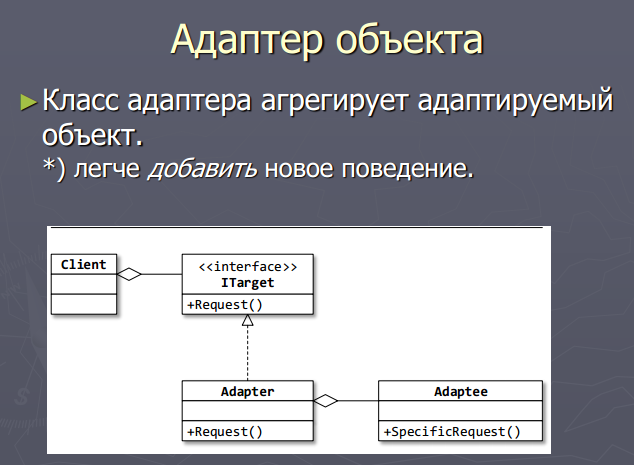
**Структурные шаблоны** — шаблоны проектирования, в которых рассматривается вопрос о том, как из классов и объектов образуются более крупные структуры. Шаблоны уровня класса полезны в тех случаях, когда необходимо объединить процессы нескольких библиотек. При этом затрагиваются такие механизмы: 1) Наследование. Определение реализации подклассами, а интерфейса – базовым классом. 2)Композиция, при которой структуры образуются через объединение нескольких объектов.

**Паттерн Адаптер** - это структурный паттерн проектирования, который позволяет объектам с несовместимыми интерфейсами работать вместе. .Представьте, что у вас есть несколько фото, которые хранятся на карте памяти. Вам нужно перенести их на компьютер, а чтобы это сделать, необходим адаптер, совместимый и с картой, и с портами ПК. Именно такую функцию выполняет паттерн Adapter: обеспечивает взаимосвязь классов и несовместимых интерфейсов.

Применение: 1)Когда необходимо использовать имеющийся класс, но его интерфейс не соответствует потребностям 2)Когда надо использовать уже существующий класс совместно с другими классами, интерфейсы которых не совместимы

+) Отделяет и скрывает от клиента подробности преобразования различных интерфейсов.

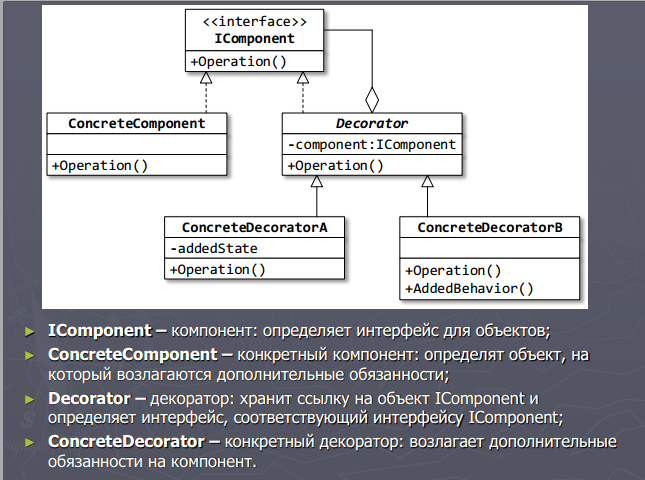
–) Усложняет код программы из-за введения дополнительных классов.

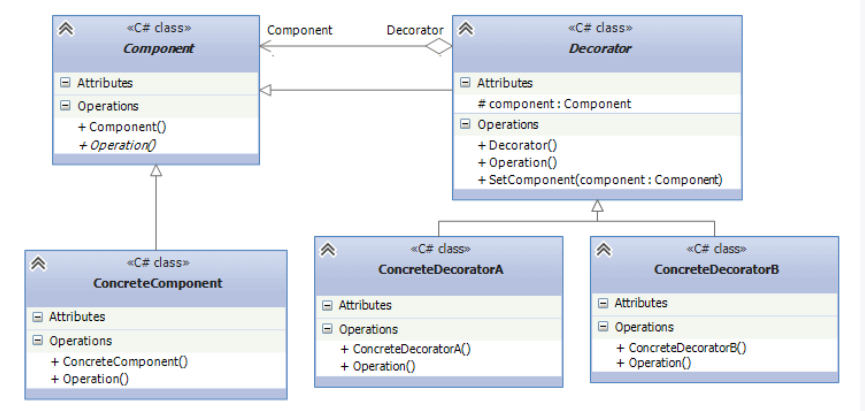
1. Адаптер имеет интерфейс, который совместим с одним из объектов.
2. Поэтому этот объект может свободно вызывать методы адаптера.
3. Адаптер получает эти вызовы и перенаправляет их второму объекту, но уже в том формате и последовательности, которые понятны второму объекту.
4. <https://metanit.com/sharp/patterns/4.2.php>
5. 
6. 

**Надо применять паттерн Decorator**, когда надо динамически добавлять к объекту новые функциональные возможности. При этом данные возможности могут быть сняты с объекта. Когда применение наследования неприемлемо. Например, если нам надо определить множество различных функциональностей и для каждой функциональности наследовать отдельный класс, то структура классов может очень сильно разрастись. Декоратор имеет альтернативное название — *обёртка*. Оно более точно описывает суть паттерна: вы помещаете целевой объект в другой объект-обёртку, который запускает базовое поведение объекта, а затем добавляет к результату что-то своё. Оба объекта имеют общий интерфейс, поэтому для пользователя нет никакой разницы, с каким объектом работать — чистым или обёрнутым. Вы можете использовать несколько разных обёрток одновременно — результат будет иметь объединённое поведение всех обёрток сразу.

+) Большая гибкость, чем у наследования; Позволяет добавлять обязанности на лету; Можно добавлять несколько новых обязанностей сразу/

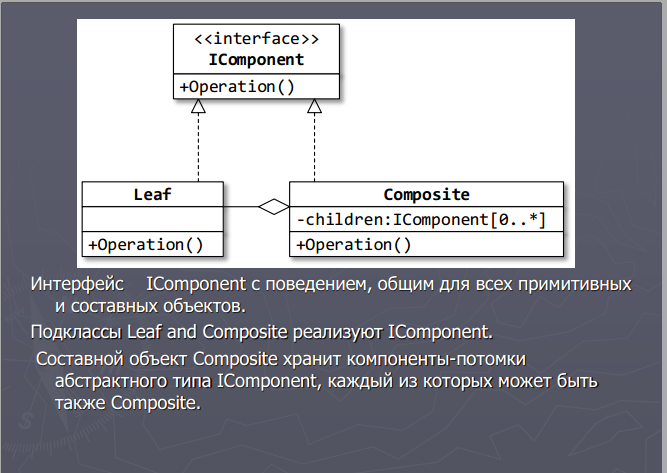
–) Трудно конфигурировать многократно обёрнутые объекты; Обилие крошечных классов.





**42.Структурные паттерны: Composite, Facade**

Структурные паттерны Composite объединяют различные объекты в древовидные структуры, позволяя в дальнейшем работать с ними, как с одним объектом.Паттерн Компоновщик (Composite) объединяет группы объектов в древовидную структуру по принципу "часть-целое и позволяет клиенту одинаково работать как с отдельными объектами, так и с группой объектов. То есть целое и его части должны реализовать один и тот же интерфейс

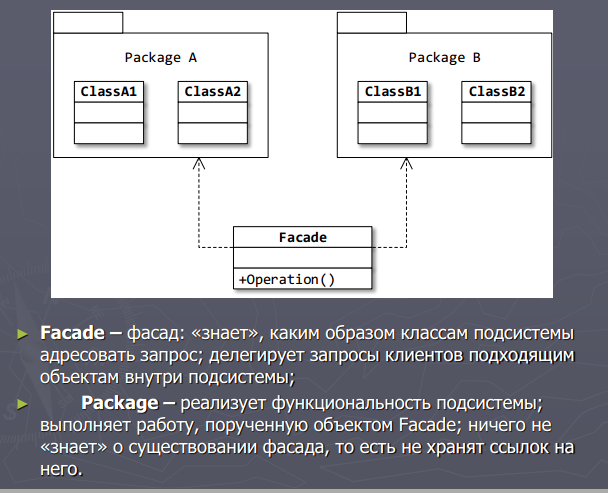
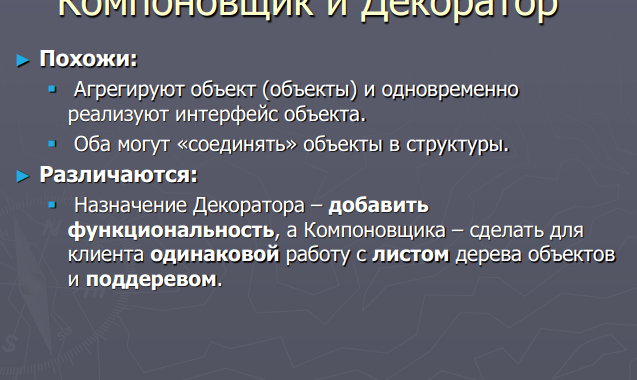


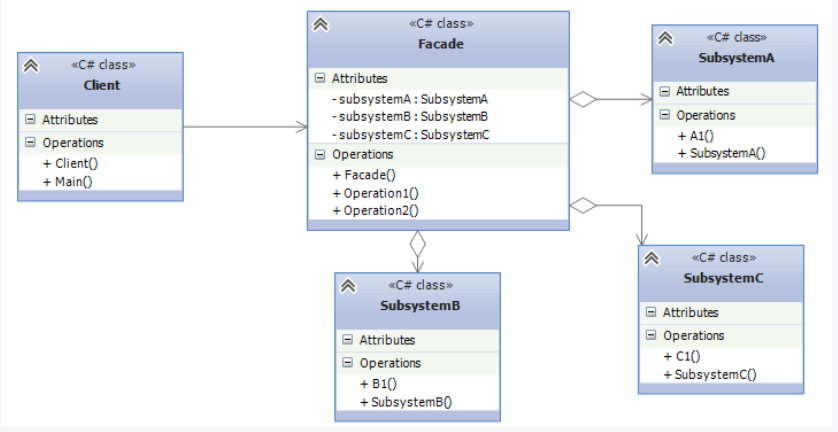
### **Когда использовать фасад?**

Фасад — это простой интерфейс для работы со сложной подсистемой, содержащей множество классов. Фасад может иметь урезанный интерфейс, не имеющий 100% функциональности, которой можно достичь, используя сложную подсистему напрямую. Но он предоставляет именно те фичи, которые нужны клиенту, и скрывает все остальные.

Фасад полезен, если вы используете какую-то сложную библиотеку со множеством подвижных частей, но вам нужна только часть её возможностей.

К примеру, программа, заливающая видео котиков в социальные сети, может использовать профессиональную библиотеку сжатия видео. Но все, что нужно клиентскому коду этой программы — простой метод encode(filename, format). Создав класс с таким методом, вы реализуете свой первый фасад.

* Когда имеется сложная система, и необходимо упростить с ней работу. Фасад позволит определить одну точку взаимодействия между клиентом и системой.
* Когда надо уменьшить количество зависимостей между клиентом и сложной системой. Фасадные объекты позволяют отделить, изолировать компоненты системы от клиента и развивать и работать с ними независимо.
* Когда нужно определить подсистемы компонентов в сложной системе. Создание фасадов для компонентов каждой отдельной подсистемы позволит упростить взаимодействие между ними и повысить их независимость друг от друга.
* 
* 



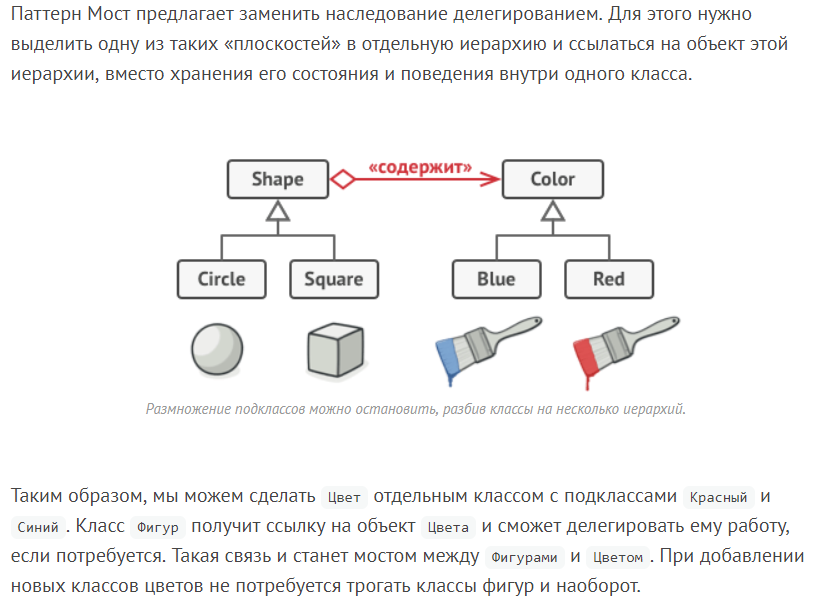
**43.Структурные паттерны: Proxy,Bridge**

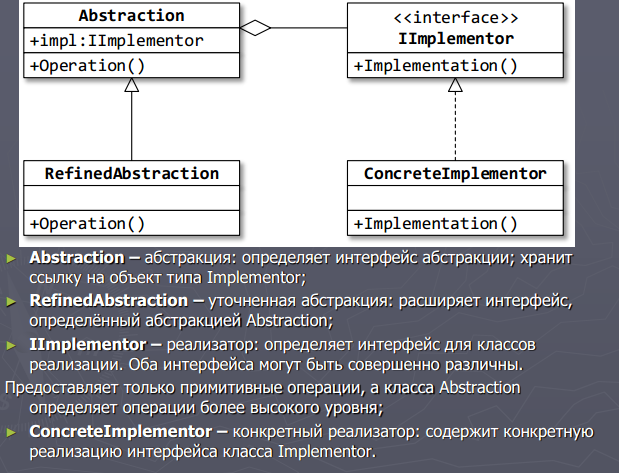
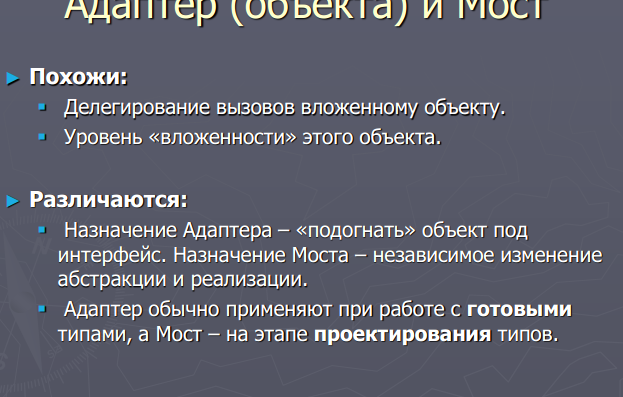
### **Заместитель** — это структурный паттерн проектирования, который позволяет подставлять вместо реальных объектов специальные объекты-заменители. Эти объекты перехватывают вызовы к оригинальному объекту, позволяя сделать что-то до или после передачи вызова оригиналу.

* Когда надо осуществлять взаимодействие по сети, а объект-проси должен имитировать поведения объекта в другом адресном пространстве. Использование прокси позволяет снизить накладные издержки при передачи данных через сеть. Подобная ситуация еще называется **удалённый заместитель (remote proxies)**
* Когда нужно управлять доступом к ресурсу, создание которого требует больших затрат. Реальный объект создается только тогда, когда он действительно может понадобится, а до этого все запросы к нему обрабатывает прокси-объект. Подобная ситуация еще называется **виртуальный заместитель (virtual proxies)**
* Когда необходимо разграничить доступ к вызываемому объекту в зависимости от прав вызывающего объекта. Подобная ситуация еще называется **защищающий заместитель (protection proxies)**
* Когда нужно вести подсчет ссылок на объект или обеспечить потокобезопасную работу с реальным объектом. Подобная ситуация называется **"умные ссылки" (smart reference)**

### **Когда использовать данный паттерн Bridge?**

**Мост** — это структурный паттерн проектирования, который разделяет один или несколько классов на две отдельные иерархии — абстракцию и реализацию, позволяя изменять их независимо друг от друга.

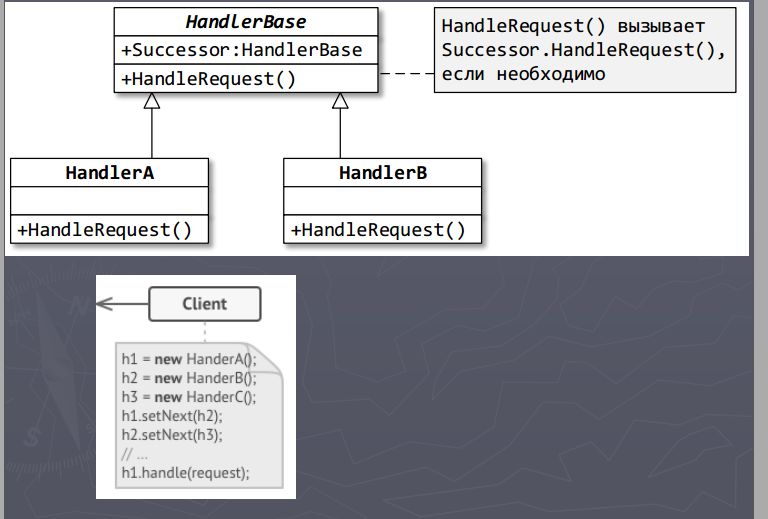


* Когда надо избежать постоянной привязки абстракции к реализации
* Когда наряду с реализацией надо изменять и абстракцию независимо друг от друга. То есть изменения в абстракции не должно привести к изменениям в реализации
* 
* 

**44.Паттерны поведения: Chain of Responsibility, Command**

**Цепочка обязанностей** — это поведенческий паттерн проектирования, который позволяет передавать запросы последовательно по цепочке обработчиков. Каждый последующий обработчик решает, может ли он обработать запрос сам и стоит ли передавать запрос дальше по цепи.

Цепочка обязанностей базируется на том, чтобы превратить отдельные поведения в объекты. В нашем случае каждая проверка переедет в отдельный класс с единственным методом выполнения. Данные запроса, над которым происходит проверка, будут передаваться в метод как аргументы.А теперь по-настоящему важный этап. Паттерн предлагает связать объекты обработчиков в одну цепь. Каждый из них будет иметь ссылку на следующий обработчик в цепи. Таким образом, при получении запроса обработчик сможет не только сам что-то с ним сделать, но и передать обработку следующему объекту в цепочке.Передавая запросы в первый обработчик цепочки, вы можете быть уверены, что все объекты в цепи смогут его обработать. При этом длина цепочки не имеет никакого значения.И последний штрих. Обработчик не обязательно должен передавать запрос дальше, причём эта особенность может быть использована по-разному.

* Когда имеется более одного объекта, который может обработать определенный запрос
* Когда надо передать запрос на выполнение одному из нескольких объект, точно не определяя, какому именно объекту
* Когда набор объектов задается динамически
* 

### 

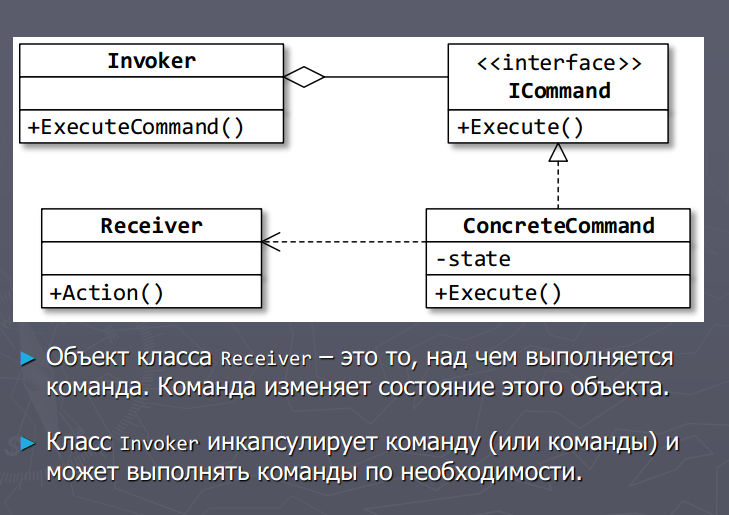
**Паттерн "Команда" (Command)** позволяет инкапсулировать запрос на выполнение определенного действия в виде отдельного объекта. Этот объект запроса на действие и называется командой. При этом объекты, инициирующие запросы на выполнение действия, отделяются от объектов, которые выполняют это действие.

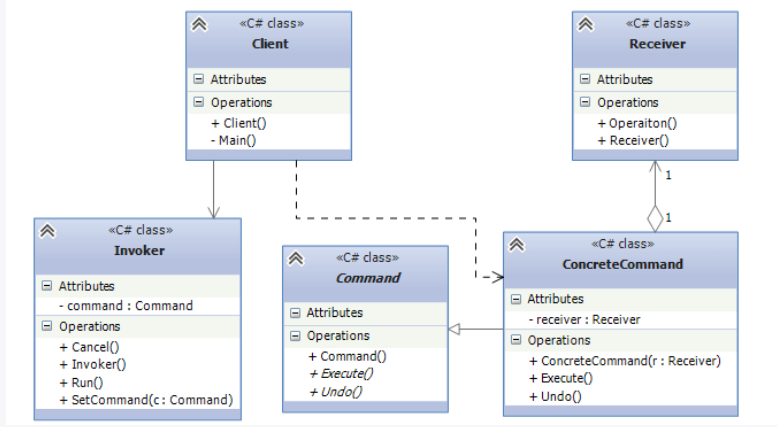
Команды могут использовать параметры, которые передают ассоциированную с командой информацию. Кроме того, команды могут ставиться в очередь и также могут быть отменены. Команда (Command) – один из «классических» поведенческих паттернов, описанных ещё у «Банды Четырёх» [1]. Он используется для создания гибкого механизма действий над чем-либо или команд. В этом механизме класс отправитель команды и класс получатель не зависят друг от друга.

Когда надо передавать в качестве параметров определенные действия, вызываемые в ответ на другие действия. То есть когда необходимы функции обратного действия в ответ на определенные действия.

Когда необходимо обеспечить выполнение очереди запросов, а также их возможную отмену.

Когда надо поддерживать логгирование изменений в результате запросов. Использование логов может помочь восстановить состояние системы - для этого необходимо будет использовать последовательность запротоколированных команд.

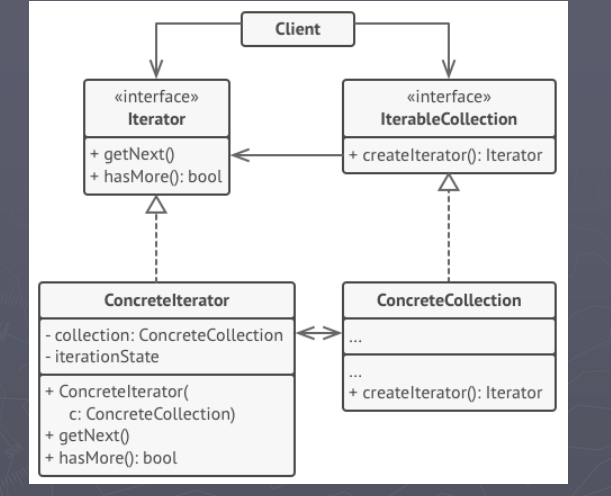


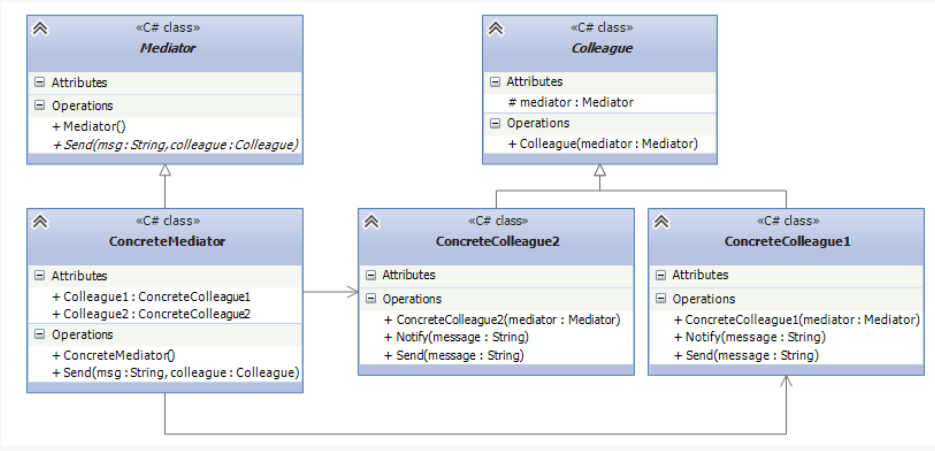


**45.Паттерны поведения:Iterator, Mediator,**

Когда используется паттерн Посредник?(как обходить рим)

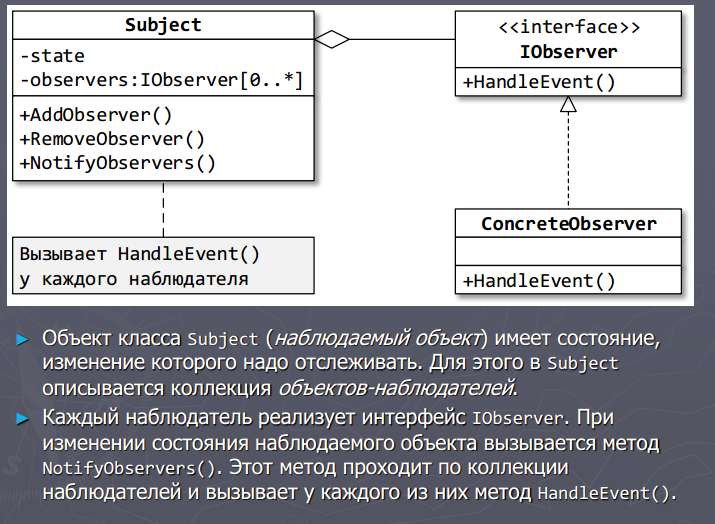
**Итератор** — это поведенческий паттерн проектирования, который даёт возможность последовательно обходить элементы составных объектов, не раскрывая их внутреннего представления.

* Когда имеется множество взаимосвязаных объектов, связи между которыми сложны и запутаны.
* Когда необходимо повторно использовать объект, однако повторное использование затруднено в силу сильных связей с другими объектами.
* 



**Посредник** — это поведенческий паттерн проектирования, который позволяет уменьшить связанность множества классов между собой, благодаря перемещению этих связей в один класс-посредник.

Паттерн Посредник заставляет объекты общаться не напрямую друг с другом, а через отдельный объект-посредник, который знает, кому нужно перенаправить тот или иной запрос. Благодаря этому, компоненты системы будут зависеть только от посредника, а не от десятков других компонентов.



* Устраняет зависимости между компонентами, позволяя повторно их использовать.
* Упрощает взаимодействие между компонентами.
* Централизует управление в одном месте.
* Посредник может [**сильно раздуться**](https://refactoring.guru/ru/smells/large-class).

**46.Паттерны поведения:Memento, Observer, NullObject**

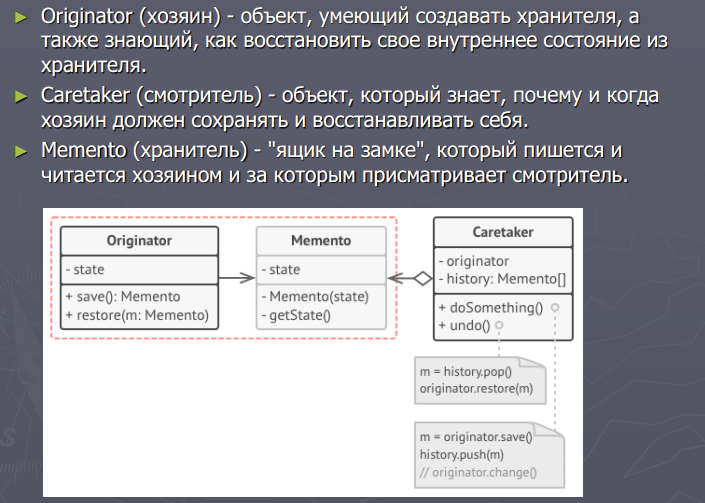
Когда использовать Memento?

* Когда нужно сохранить состояние объекта для возможного последующего восстановления
* Когда сохранение состояния должно проходить без нарушения принципа инкапсуляции

То есть ключевыми понятиями для данного паттерна являются сохранение внутреннего состояния и инкапсуляция, и важно соблюсти баланс между ними.

Memento: хранитель, который сохраняет состояние объекта Originator и предоставляет полный доступ только этому объекту Originator

Например, в игре происходит управление героем, все состояние которого заключено в нем самом - оружие героя, показатель жизней, силы, какие-то другие показатели. И нередко может возникнуть ситуация, сохранить все эти показатели во вне, чтобы в будущем можно было откатиться к предыдущему уровню и начать игру заново. В этом случае как раз и может помочь паттерн Хранитель.

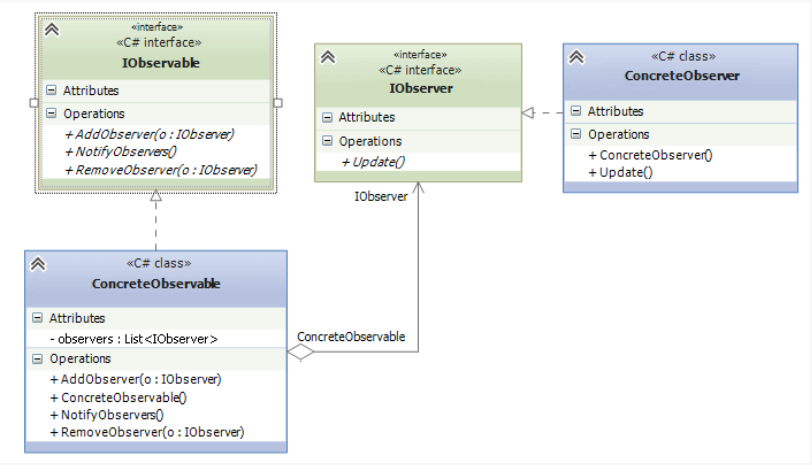
* Уменьшение связанности системы
* Сохранение инкапсуляции информации
* Определение простого интерфейса для сохранения и восстановления состояния
* 

### **Наблюдатель** — это поведенческий паттерн проектирования, который создаёт механизм подписки, позволяющий одним объектам следить и реагировать на события, происходящие в других объектах.

Давайте называть Издателями те объекты, которые содержат важное или интересное для других состояние. Остальные объекты, которые хотят отслеживать изменения этого состояния, назовём Подписчиками.

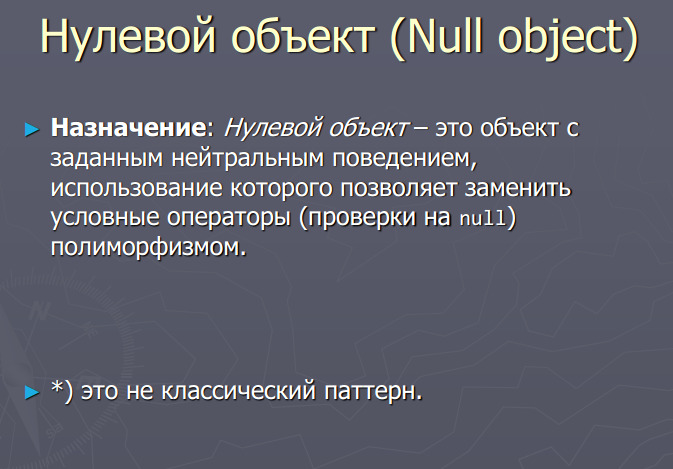
Паттерн Наблюдатель предлагает хранить внутри объекта издателя список ссылок на объекты подписчиков, причём издатель не должен вести список подписки самостоятельно. Он предоставит методы, с помощью которых подписчики могли бы добавлять или убирать себя из списка.

* Издатели не зависят от конкретных классов подписчиков и наоборот.
* Вы можете подписывать и отписывать получателей на лету.
* Реализует *принцип открытости/закрытости*.
* Подписчики оповещаются в случайном порядке.
* Когда система состоит из множества классов, объекты которых должны находиться в согласованных состояниях
* Когда общая схема взаимодействия объектов предполагает две стороны: одна рассылает сообщения и является главным, другая получает сообщения и реагирует на них. Отделение логики обеих сторон позволяет их рассматривать независимо и использовать отдельно друга от друга.
* Когда существует один объект, рассылающий сообщения, и множество подписчиков, которые получают сообщения. При этом точное число подписчиков заранее неизвестно и процессе работы программы может изменяться.

Целью Null-object является инкапсулирование *отсутствия* объекта путём замещения его другим объектом, который ничего не делает.

Данный шаблон проектирования рекомендуется использовать, когда:

* Объект требует взаимодействия с другими объектами. Null Object не устанавливает нового взаимодействия — он использует уже установленное взаимодействие.
* Какие-то из взаимодействующих объектов должны бездействовать
* Требуется абстрагирование «общения» с объектами, имеющими NULL-значение.



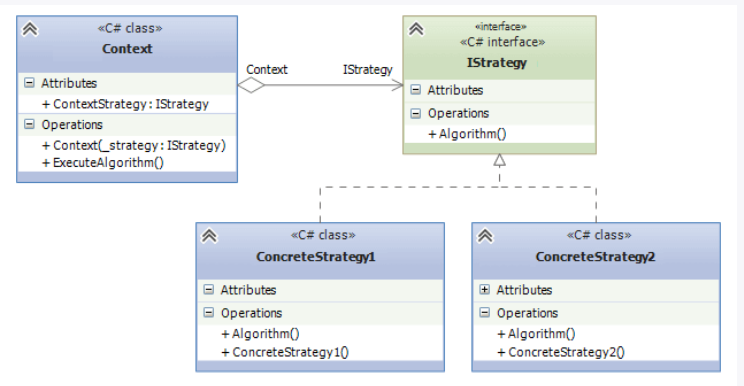
**47.Паттерны поведения:Visitor, Strategy**

* **Посетитель** — это поведенческий паттерн проектирования, который позволяет добавлять в программу новые операции, не изменяя классы объектов, над которыми эти операции могут выполняться. Когда имеется много объектов разнородных классов с разными интерфейсами, и требуется выполнить ряд операций над каждым из этих объектов
* Паттерн Посетитель предлагает разместить новое поведение в отдельном классе, вместо того чтобы множить его сразу в нескольких классах. Объекты, с которыми должно было быть связано поведение, не будут выполнять его самостоятельно. Вместо этого вы будете передавать эти объекты в методы посетителя.
* Код поведения, скорее всего, должен отличаться для объектов разных классов, поэтому и методов у посетителя должно быть несколько. Названия и принцип действия этих методов будет схож, но основное отличие будет в типе принимаемого в параметрах объекта, например:
* Начинающий риэлтор, для каждого типа домов своё коммерческое предложение
* Когда классам необходимо добавить одинаковый набор операций без изменения этих классов
* Когда часто добавляются новые операции к классам, при этом общая структура классов стабильна и практически не изменяется

### **Стратегия** — это поведенческий паттерн проектирования, который определяет семейство схожих алгоритмов и помещает каждый из них в собственный класс, после чего алгоритмы можно взаимозаменять прямо во время исполнения программы.

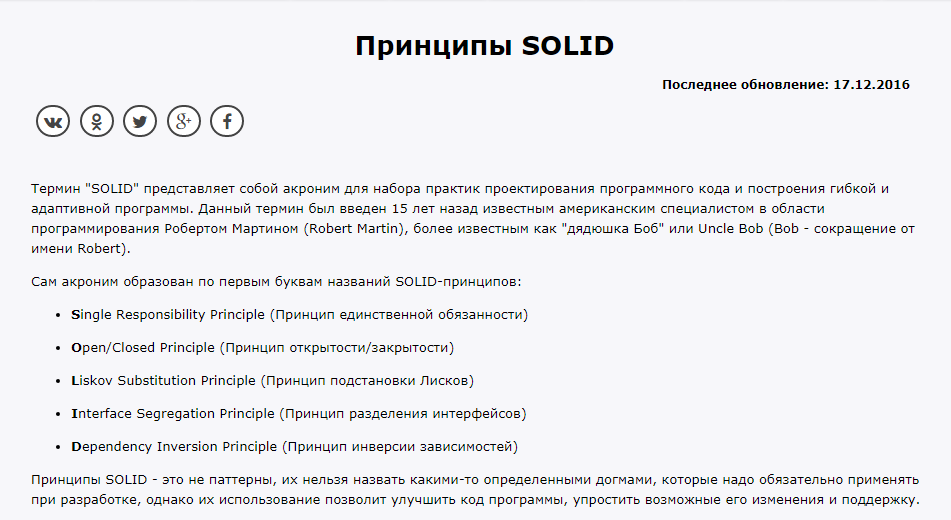
Чем доехать до аэропорта

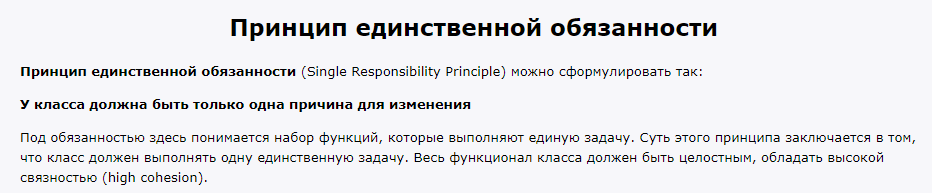
* Когда есть несколько родственных классов, которые отличаются поведением. Можно задать один основной класс, а разные варианты поведения вынести в отдельные классы и при необходимости их применять
* Когда необходимо обеспечить выбор из нескольких вариантов алгоритмов, которые можно легко менять в зависимости от условий
* Когда необходимо менять поведение объектов на стадии выполнения программы
* Когда класс, применяющий определенную функциональность, ничего не должен знать о ее реализации

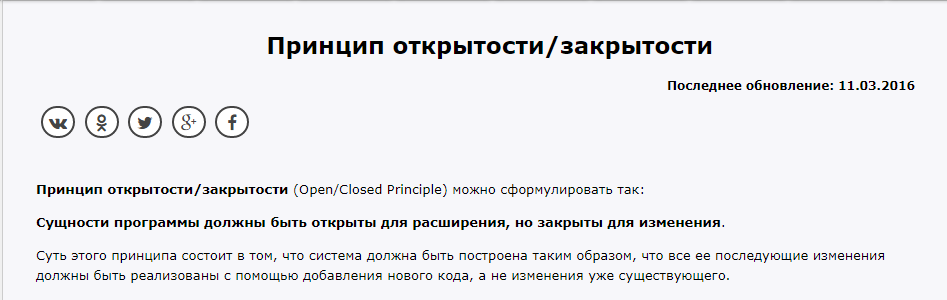


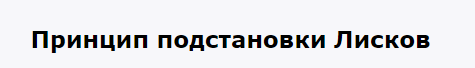
**48.Принципы проектирования SOLID.**

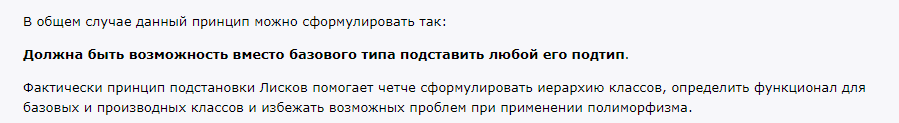
Принципы SOLID - это не паттерны, их нельзя назвать какими-то определенными догмами, которые надо обязательно применять при разработке, однако их использование позволит улучшить код программы, упростить возможные его изменения и поддержку.



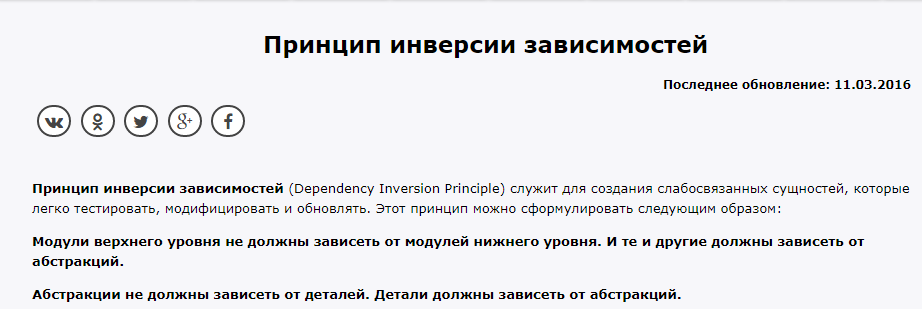












Сам акроним образован по первым буквам названий SOLID-принципов:

* **S**ingle Responsibility Principle (Принцип единственной обязанности, пример принтер)

**У класса должна быть только одна причина для изменения**

Под обязанностью здесь понимается набор функций, которые выполняют единую задачу. Суть этого принципа заключается в том, что класс должен выполнять одну единственную задачу. Весь функционал класса должен быть целостным, обладать высокой связностью (high cohesion).

* **O**pen/Closed Principle (Принцип открытости/закрытости, привет повар)

**Сущности программы должны быть открыты для расширения, но закрыты для изменения**.

Суть этого принципа состоит в том, что система должна быть построена таким образом, что все ее последующие изменения должны быть реализованы с помощью добавления нового кода, а не изменения уже существующего.

* **L**iskov Substitution Principle (Принцип подстановки Лисков, пример квадрат и прямоугольник)

**Должна быть возможность вместо базового типа подставить любой его подтип**.

Фактически принцип подстановки Лисков помогает четче сформулировать иерархию классов, определить функционал для базовых и производных классов и избежать возможных проблем при применении полиморфизма.

* **I**nterface Segregation Principle (Принцип разделения интерфейсов, отправка сообщений разными способами)

**Клиенты не должны вынужденно зависеть от методов, которыми не пользуются.**

При нарушении этого принципа клиент, использующий некоторый интерфейс со всеми его методами, зависит от методов, которыми не пользуется, и поэтому оказывается восприимчив к изменениям в этих методах. В итоге мы приходим к жесткой зависимости между различными частями интерфейса, которые могут быть не связаны при его реализации.

* **D**ependency Inversion Principle (Принцип инверсии зависимостей, книга и ее печать)

Модули верхнего уровня не должны зависеть от модулей нижнего уровня. И те и другие должны зависеть от абстракций.

Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.