Министерство науки и высшего образования Российской Федерации

АМУРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

А.В. Нацвин, Т.А. Юрьева

РАЗРАБОТКА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ С ГРАФИЧЕСКИМ ИНТЕРФЕЙСОМ НА БАЗЕ ФРЕЙМВОРКА AVALONIA

Учебно-методическое пособие

Благовещенск

Издательство АмГУ

2023

ББК 32.973.018.2

Н 35

Рекомендовано

учебно-методическим советом университета

Рецензент:

Акилова И.М., доцент кафедры информационных и управляющих систем ФГБОУ ВО АмГУ

Нацвин А.В.

Разработка программного обеспечения с графическим интерфейсом на базе фреймворка Avalonia: учебно-методическое пособие /А.В. Нацвин, Т.А. Т.А. Юрьева – Благовещенск: Изд-во АмГУ, 2023. – 102 с.

Учебно-методическое пособие содержит краткие теоретические сведения по дисциплине «Проектирование пользовательского интерфейса» и методические материалы и задания для выполнения лабораторных работ.

Пособие предназначено для студентов направлений подготовки 09.03.01 – «Информатика и вычислительная техника» и 09.03.02 – «Информационные системы и технологии».

ВВЕДЕНИЕ

Одной из профессиональных компетенций, согласно требованиям федерального государственного образовательного стандарта 09-ых направлений подготовки бакалавриата, является способность проектирования пользовательского интерфейса хотя-бы по образцу или концепции, а также его графический дизайн. В этой связи в учебные планы направлений подготовки «Информатика и вычислительная техника» и «Информационные системы и технологии» в Амурском государственном университете были введены дисциплины «Проектирование интерфейса «Человек-компьютер»» и «Проектирование пользовательского интерфейса» соответственно. В рамках этих дисциплин предусматривается выполнение лабораторных работ.

Структура и содержание лабораторного практикума были ориентированы на достижение следующих учебных задач:

1. Познакомиться с шаблоном проектирования MVVM (Model-View-ViewModel), а также библиотекой ReactiveUI, упрощающей его реализацию;

2. Изучить возможности настройки компилятора в целях оптимизации потребления ресурсов ПК;

3. Разработать программное обеспечение, базирующееся на кросс-платформенном фреймворке Avalonia;

4. Применить стилизацию к оформлению разработанного пользовательского интерфейса.

Каждый элемент содержания сопровождается подробной методикой выполнения работы, с иллюстрациями исходного кода и промежуточных результатов, ссылки на дополнительные источники информации, на справочные материалы и демонстрации примеров реализаций каких-либо функций.

После выполнения основных учебных задач студентам предлагается самостоятельно создать программу с применением фреймворка Avalonia согласно варианту.

Лабораторная работа №1

Шаблон проектирования MVVM, настройка IDE и создание проекта с использованием фреймворка Avalonia

Шаблон проектирования MVVM

Шаблон проектирования или паттерн (design pattern) – это повторяемая конструкция, предназначенная для решения определенной группы задач в рамках некоторого контекста. Основными преимуществами паттернов являются упрощение разработки за счет готовых абстракций, а так же облегчение коммуникации между разработчиками за счет использования знакомых терминов. Паттерны, в свою очередь, могут быть как низкоуровневыми (идиомы) так и высокоуровневыми (архитектурными). В свою очередь, алгоритмы также являются шаблонами (например, алгоритмы сортировки), но не проектирования, а вычисления. Шаблоны проектирования имеют довольно обширную классификацию:

1) Основные
    а) Основные шаблоны
    б) Порождающие шаблоны
    в) Структурные шаблоны
    г) Поведенческие шаблоны
2) Частные
    а) Шаблоны параллельного программирования
    б) Шаблоны генерации объектов
    в) Шаблоны программирования гибких объектов
    г) Шаблоны выполнения задач
    д) Шаблоны архитектуры системы
    е) Предприятие
    ж) Шаблоны проектирования потоковой обработки
    з) Шаблоны проектирования распределённых систем
    и) Шаблоны баз данных и др.
3) Иные шаблоны

Рассмотрим паттерн, часто применяемый при разработке приложений на языке C# Model-View-ViewModel. MVVM – это шаблон проектирования архитектуры приложения, представленный в 2005 году Джоном Госсманом как модификация шаблона Presentation Model, изначально предназначенный для технологии WPF (Windows Presentation Fondation) и впоследствии UWP (Universal Windows Platform). На данный момент применение MVVM не ограничивается решениями Microsoft и применяется на многих платформах, например, Linux и iOS.

MVVM состоит из трех слоев: модель (Model), представление (View) и модель представления (ViewModel) (рис. 1).

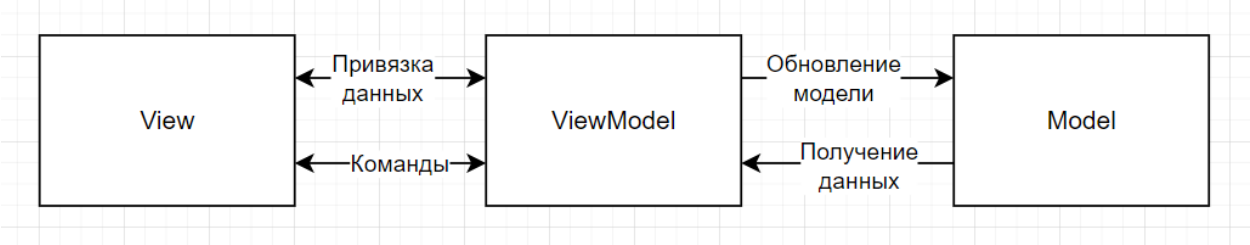


Рис. 1 Схема приложения MVVM

Модель описывает данные используемые приложением и логику обработки этих данных. Однако модель, согласно паттерну, не должна содержать взаимодействие с элементами пользовательского интерфейса.

Представление – это пользовательский интерфейс. Исходный код интерфейса не включает никакой логики взаимодействия с данными, однако может включать обработчики событий для изменения самого интерфейса.

Модель отображения реализует свойства и команды, а так же хранит данные полученные из модели и при этом предоставляет их отображению, отображение же решает, как эти данные будут показаны.

Преимуществом такого разделения приложения является возможность разработки, тестирования, модификации каждого компонента отдельным специалистом, а именно, разработчик задает логику программы, а дизайнер создает интерфейс.

На данный момент существует множество GUI-библиотек и оберток над библиотеками C, C++ и т.д. для языка C#, например WPF, UWP (Microsoft), QT# (Nokia), GTK# (The GTK+ Team) и сравнительно недавно появившаяся UNO Platform (Nventive and Community). При этом паттерн MVVM полноценно поддерживают только решения от Microsoft, в довесок к этому большинство разработок либо ограничены платформой .NET Framework, либо имеют скудную документацию и практически не развиваются, так как делаются небольшими командами энтузиастов или появились сравнительно недавно. На этом фоне стоит отметить один из популярных проектов с поддержкой MVVM и кроссплатформенности – фреймворк Avalonia.

Установка Avalonia

Avalonia – это фреймворк для создания приложений с графическим интерфейсом пользователя на языке XAML и C# для платформы .NET. Во многом данный проект похож на технологии WPF/UWP/Xamarin, что дает преимущество в виде возможности использования справочных материалов предназначенных для этих технологий. Avalonia поддерживает Windows, Linux, macOS, кроме того присутствует экспериментальная поддержка мобильных платформ Android и iOS. Минимальные требования к платформе для запуска приложений представлены в таблице 1.

Таблица 1

Операционная система	Версия	Платформа
Windows	8+	.NET Core 3.1+
Linux	Debian 9 (Stretch) + Ubuntu 16.04 + Fedora 30 +	.NET Core 3.1+
MacOS	MacOS High Sierra 10.13+	.NET Core 3.1+

В первую очередь для работы потребуется установленный SDK .NET (таб. 2). .NET (ранее .NET Core) – это платформа для разработки кроссплатформенного программного обеспечения, совместимая с операционными си-

стемами Windows, Linux и macOS. Данная платформа основана на .NET Framework и развивается параллельно ему сотрудниками компании Microsoft и организацией .NET Foundation.

Таблица 2

.NET SDK	QR-Code
Для среды Visual Studio 2019 последним является .NET 5.	
Для среды Visual Studio 2022 (также и для Visual Studio Code) последним является .NET 7.	

.NET является обязательным компонентом для разработки, но выбор IDE (среды разработки) и ОС уже предоставляется разработчику. В рамках данного курса разработка будет вестись в Visual Studio 2019/2022 (Так же есть поддержка Visual Studio Code и JetBrains Rider) на платформе Windows 10. Для работы в Visual Studio, потребуется плагин, добавляющий шаблоны приложений Avalonia (таб. 3).

Таблица 3

IDE	QR-Code	Плагин	QR-Code
Visual Studio 2019		Avalonia for Visual Studio 2019	
Visual Studio 2022		Avalonia for Visual Studio 2022	

После установки плагина в меню создания проектов появится 4 новых шаблона Avalonia (рис. 2).

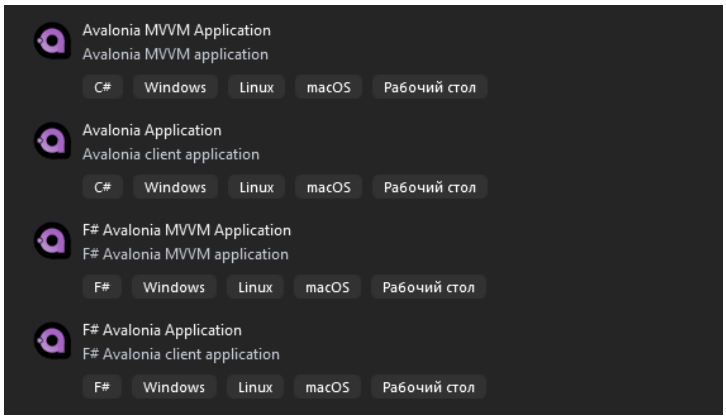


Рис. 2 Меню выбора шаблона проектов

В свою очередь, основными справочными источниками в данном курсе выступят официальная справка Avalonia и справка Microsoft для WPF/UWP (таб. 4).

Таблица 4

Справка	QR-Code
Официальная справка Avalonia (ENG)	
Официальная справка Microsoft	

Создание первого проекта

Выберете шаблон «`Avalonia MVVM Application`» для языка C# в качестве названия укажите «`Demo`». В проводнике перейдите в папку проекта (По умолчанию путь: `C:\Users\%username%\source\repos`) и найдите файл проекта «`Название_проекта`».`csproj` (`Demo.csproj`)

(или сделайте двойной щелчок мыши на названии проекта в обозревателе решений). В данном файле найдите следующий параметр: `<TargetFramework>net6.0</TargetFramework>` и замените «`net6.0`» на тот, который установлен у вас в системе (в случае несовпадений версий компиляция проекта завершиться ошибкой). Для того чтобы заработал Avalonia XAML Editor, а именно окно предпросмотра, необходимо выполнить сборку решения (Меню `Сборка` → `Собрать решение`; или сочетание клавиш Ctrl+Shift+B). Запустите отладку стандартного проекта приложения (клавиша F5), в результате должно открыться следующее окно (рис. 3).

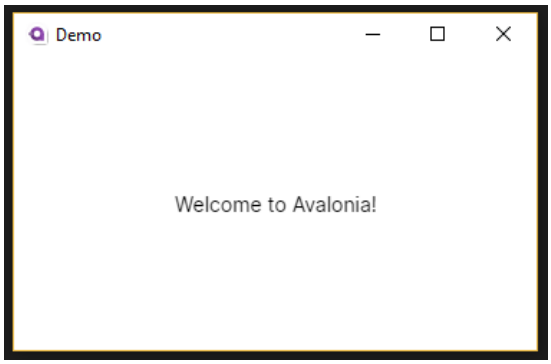


Рис. 3 Окно программы

Файловая иерархия проекта

Проект, созданный в Avalonia, представляет собой следующую иерархию (рис. 4). В шаблоне MVVM есть каталоги для каждой из концепций: модель (Model), представление (View) и модель-представление (ViewModel) Назначение файлов и каталогов подробнее представлено в таблице 5.

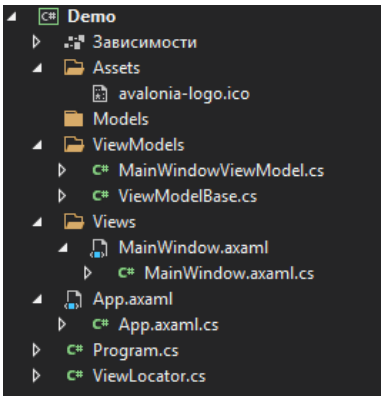


Рис. 4 Иерархия файлов и каталогов проекта

Таблица 5

Объект	Назначение
Каталог Assets	Каталог содержит файлы, включенные в приложение, например, значки (.ico), картинки (.png, .bmp и тд.). Файлы из этого каталога будут автоматически добавлены в качестве ресурсов.
Каталог Models	Изначально каталог пуст, в будущем в этом каталоге будут располагаться модели.
Каталог ViewModels	Данный каталог предназначен для хранения моделей представления. Предварительно в каталоге создан файл базового класса для моделей представления. Так же в каталоге присутствует модель представления для главного окна приложения, которое является производным от базового класса.
Каталог Views	Содержит файлы исходного кода для окон приложения (представления). Пока имеется только исходный код главного окна.
Файл App.axaml	Это файл, где будут размещены стили и шаблоны XAML, которые будут применяться ко всему приложению.
Файл Program.cs	Данный файл является точкой входа для приложения.
Файл ViewLocator.cs	Используется для поиска соответствующих view для viewmodel.

Настройки публикации приложения

Фреймворк Avalonia, как и множество других кроссплатформенных библиотек, имеет один существенный недостаток в виде большого потребления оперативной памяти и дискового пространства. Высокий расход ресурсов неизбежная цена за комфорт, как разработки, так и эксплуатации. Так вместе с откомпилированным проектом Avalonia поставляются все библиотеки не-

обходимые для функционирования GUI, что означает его легкую переносимость с одной машины на другую. В свою очередь, среда .NET позволяет оптимизировать занимаемое дисковое пространство и потребление оперативной памяти с помощью настройки публикации приложения, кроме того приложение можно сделать полностью автономным добавив все необходимые библиотеки в том числе и входящие в пакет .NET. Настройки публикации приложения доступны как через редактирование файла проекта .csproj так и через указание ключей при компиляции через .NET CLI (Command Line Interface).

Рассмотрим вариант публикации приложения через CLI, а так же несколько из доступных параметров. Для того чтобы открыть CLI нужно выбрать пункт Средства → Командная строка → Командная строка разработчика. Аналогом стандартной сборки приложения из меню является команда `dotnet build`. Однако данная команда не создает приложение готовое для развертывания на другой машине, вместо этого необходимо использовать `dotnet publish` при этом проект скомпилируется в Debug-варианте. Для компиляции проекта в Release-варианте нужно указать параметр `-c Release`.

Следующим параметром для компиляции служит указание среды выполнения приложения `-r <RID>`. Данный параметр ограничит приложение конкретной средой исполнения путем исключения зависимостей для других платформ, например, для Windows x64 параметр выглядит как `-r win-x64`. Полный список RID можно получить в справке Microsoft (таб. 6).

Таблица 6

Справка	QR-Code
.NET Каталог RID	

В свою очередь, следующий параметр --self-contained [true|false] определяет, будет ли вместе с приложением опубликована среда .NET. Применение данного параметра может сделать приложение полностью автономным (значение `true`), однако при этом возрастет занимаемое пространство.

Параметр `-p:PublishTrimmed=true` позволяет удалить неиспользуемые функции из итоговой сборки. Однако не все библиотеки (например, WPF) поддерживают данный параметр.

Следующий параметр `-p:PublishSingleFile=true позволяет упаковать все файлы dll` в единый исполняемый файл.

Еще один параметр `-p:PublishReadyToRun=true` уменьшают время запуска и задержки в приложении путем компиляции сборок в формате ReadyToRun (R2R). Двоичные файлы R2R повышают производительность при запуске, уменьшая объем работы, которую компилятор JIT должен выполнять при загрузке вашего приложения. Двоичные файлы содержат аналогичный машинный код по сравнению с тем, что мог бы создать JIT. Однако двоичные файлы R2R больше, поскольку они содержат как код на промежуточном языке (IL), который все еще необходим для некоторых сценариев, так и собственную версию того же кода. R2R доступен только при публикации приложения, предназначенного для определенных сред выполнения (RID), таких как Linux x64 или Windows x64.

Более подробно про другие возможные параметры публикации приложения можно прочитать в официальной справке Microsoft (таб. 7).

Таблица 7

Справка	QR-Code
dotnet publish	

В качестве примера выше созданный проект Demo был опубликован несколькими способами, результаты которых представлены в таблице 8.

Таблица 8

Вариант компиляции	Рабочий набор (МБ) (RAM)	Размер (МБ) (ROM)
dotnet publish -c Release	124,7	109
dotnet publish -c Release -r win-x64 --self-contained true	99,4	97,1
dotnet publish -c Release -r win-x64 --self-contained true -p:PublishTrimmed=true	98,8	56,5
dotnet publish -c Release -r win-x64 --self-contained false -p:PublishSingleFile=true	97,6	24
dotnet publish -c Release -r win-x64 --self-contained true -p:PublishReadyToRun=true	84,6	36,6

Представленные результаты не могут быть полностью объективными и могут отличаться, так как многое зависит, как от платформы, на котором было запущено приложение, так и от самого приложения.

Контрольные вопросы

1. *Что такое паттерны (шаблоны) проектирования?*
2. *Что представляет собой шаблон MVVM?*
3. *Что такое Avalonia, имеются ли у нее аналоги?*
4. *Что такое .NET?*

Лабораторная работа №2

Первая программа с архитектурой MVVM на Avalonia.

Источники

Данная лабораторная работа основана на примере приложения из официальной справки Avalonia ToDo List App (таб. 9).

Таблица 9

Справка	QR-Code
Avalonia ToDo (ENG)	

Создание отображения главного окна

Avalonia как и WPF/UWP использует для разметки интерфейса язык XAML (англ. eXtensible Application Markup Language). Данный язык программирования, основанный на XML, разработан компанией Microsoft и предназначен для декларативного программирования (описывается ожидаемый результат, а не способ его получения). Пользовательский интерфейс программируется подобно WEB-странице на языке HTML, однако, логика приложения по-прежнему пишется процедурным языком, а для задания свойств элементов используются атрибуты на манер свойств в объектно-ориентированных языках.

Создайте проект приложения «`Avalonia MVVM Application`» по аналогии с лабораторной работой №1 (в качестве названия проекта укажите «`Demo_2`»). Далее выполните следующие действия: нажать правой кнопкой мыши (Далее ПКМ) на каталоге «`Views`» → `Добавить` → `Создать элемент` … (рис. 5).

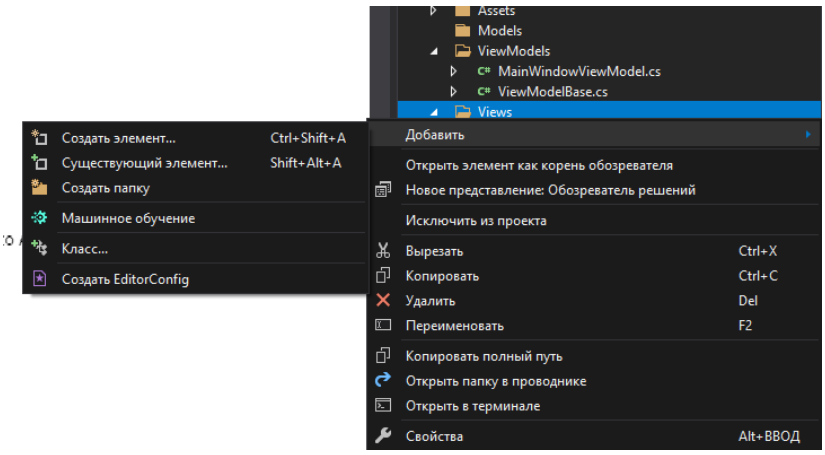


Рис. 5 Меню создания объектов

Выберите пункт «Установленные» → «Элементы Visual C#» → «Avalonia» → «User Control (Avalonia)». В качестве названия пользовательского элемента укажите «TaskListView.axaml» (рис. 6).

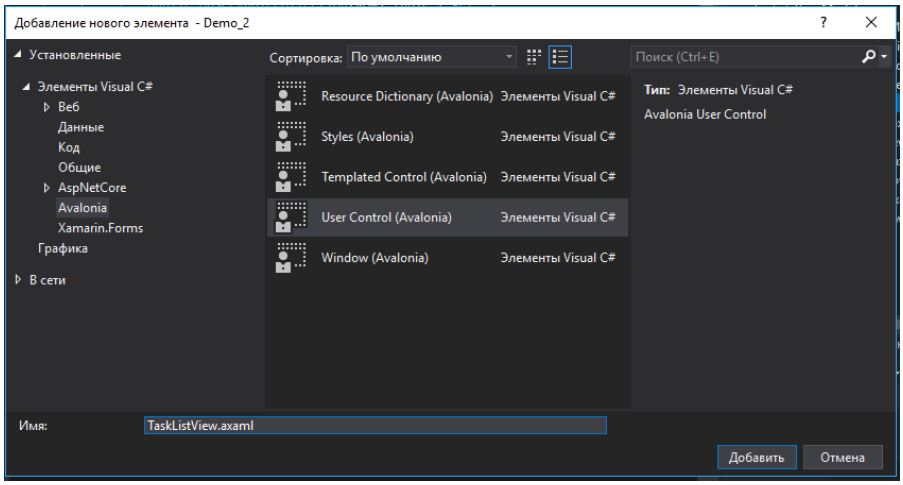


Рис. 6 Меню создания объектов

В технологиях Microsoft WPF/UWP файл разметки имеет расширение xaml, в то время как в Avalonia было принято (в угоду уменьшения ошибок) сделать свой формат axaml. К только что созданному файлу axaml как и к MainWindow.axaml, будет создан соответствующий файл отделенного кода cs (еще его называют code behind) (не путать с ViewModel). Данный файл предназначен для логики, связанной с интерфейсом, и может содержать в себе, например, обработчики событий элементов управления.

Кроме того, обязательно следует учитывать, что в проекте Avalonia правильное наименование классов моделей и моделей отображения имеет значение. Отображение и модель отображения должны иметь одинаковые имена, при этом отображение может иметь или не иметь в своем названии префикс `View`, а модель отображения обязана иметь в своем названии префикс `ViewModel`. Так если отображение имеет название `MyWindow`**`View`**`.axaml` (или `MyWindow.axaml`), то соответствующая ему модель отображения должна называться `MyWindow`**`ViewModel`**`.cs`. В свою очередь, сам механизм подмены префикса описан в файле «`ViewLocator.cs`» (рис. 7).

```
var name = data.GetType().FullName!.Replace("ViewModel", "View");
```

Рис. 7 Механизм подмены префикса

В созданном файле «`TaskListView.axaml`» замените содержимое следующим (рис. 8).

```
<UserControl xmlns="https://github.com/avaloniaui"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
             xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
             mc:Ignorable="d" d:DesignWidth="300" d:DesignHeight="400"
             x:Class="Demo_2.Views.TaskListView">
    <DockPanel>
        <Label DockPanel.Dock="Top"
               FontWeight="Bold"
               HorizontalContentAlignment="Center">Список дел</Label>
        <Button DockPanel.Dock="Bottom"
                HorizontalAlignment="Stretch"
                HorizontalContentAlignment="Center"
                Foreground="Green">
            Добавить элемент
        </Button>
        <StackPanel>
            <CheckBox Margin="4">Почитать теорию</CheckBox>
            <CheckBox Margin="4">Сделать лабораторную работу</CheckBox>
        </StackPanel>
    </DockPanel>
</UserControl>
```

Рис. 8 Листинг отображения пользовательского объекта

Теги (они же объекты) в XAML, как и в XML, HTML и т.д. могут записываться в следующем виде:

1) Открывающий и закрывающие тэги с параметрами (необязательно) с кодом внутри:

```
<объект свойство="значение">контент</объект>
```

2) Один тэг с «слешем» в конце:

```
<объект свойство="значение" Content="контент"/>
```

Первый тег `<UserControl>` представляет пустой элемент управления, который можно использовать для создания других элементов управления. Объект `<UserControl>` как и `<Window>` может иметь только один дочерний элемент, чаще всего таким элементом является панель.

В качестве параметров у данного тэга, как корневого элемента, указаны пространства имен XML – это, как и в C#, некое абстрактное хранилище методов, классов и т.д. предназначенное для избегания конфликтов имен. В XAML первым указывается пространство имен по умолчанию, для проектов Avalonia это `xmlns="https://github.com/avaloniaui"`. В свою очередь, второе объявление `xmlns:x` определяет словарь языка XAML и сопоставляет его с выделенным пространством имен `x` (данное сопоставление является общепринятым в справке Microsoft). Стандартное применение данного пространства имен заключается в использовании x:Key (уникальный ключ для ресурсов), x:Class (класс, представляющий code behind), x:Name (уникальное имя элемента интерфейса) и т.д. Таким образом, в записи `x:Class="Demo2.Views.ListOfWorkView"` говорится о привязанном к AXAML-файлу классе (code behind). В свою очередь, следующий параметр `mc:Ignorable` это пространство имен определения XAML которые предназначены для создания дизайна и игнорируются во время выполнения программы. Соответственно `d:DesignWidth="200"` и `d:DesignHeight="400"` устанавливают ширину и высоту окну, однако после компиляции эти значения будут пропущены. Более подробно о пространствах имен можно узнать из справки Microsoft (таб. 10).

Таблица 10

Справка	QR-Code
Пространства имен XAML	

Панели, встреченные далее – это элемент-компоновщик, т.е. в его функции входит отрисовка, положение, пропорции и расположение относительно друг друга дочерних элементов. В панель может быть вложена как другая панель, так и элементы интерфейса. Первый компоновщик, встреченный в приведенном коде – это `<DockPanel>`. Данный компоновщик определяет область, в которой дочерние элементы могут быть упорядочены подобно WEB-странице: слева (`Left`), справа (`Right`), вверху (`Top`) и внизу (`Bottom`). Для этого нужно указать параметр `DockPanel.Dock="параметр"`. В ином случае элементы будут располагаться по центру.

В свою очередь, `<StackPanel>` упорядочивает дочерние элементы по принципу стека. По умолчанию элементы располагаются вертикально, однако, указав параметр `Orientation="Horizontal"` можно упорядочить элементы горизонтально.

Последние три объекта, которые были указаны в коде – это `<Label>` (текстовая метка), `<Button>` (кнопка) и `<CheckBox>` (текстовая метка, которую можно выбрать, поставив флажок), являются уже элементами управления. Список доступных элементов довольно обширен, полный перечень, как и примеры, их использования доступны в официальной справке Avalonia, кроме того, полный перечень свойств доступен в официальной справке Microsoft (таб. 11).

Таблица 11

Справка	QR-Code
Элементы интерфейса (ENG)	
Элементы интерфейса	

Параметры `HorizontalAlignment` и `HorizontalContentAlignment`, указанные в тегах, выравнивают сам элемент и контент внутри него согласно значению параметра (Stretch – растянуть, Center – по центру, Left по левому краю, Right – по правому краю). Свойство `Margin` устанавливает размер свободного пространства вокруг элемента (в пикселях). Данное свойство принимает одинарное значение, например `Margin="5"` (отступ 5 пикселей с каждой стороны), двойное `Margin="5, 7"` (отступ слева 5 пикселей, сверху 7 пикселей) и четверное `Margin="5, 7, 0, 4"` (слева 5, сверху 7, справа 0, снизу 4). Данные параметры относятся к компоновке элементов и присутствуют практически у всех элементов интерфейса. О применении данных свойств можно прочитать в справке (таб. 12).

Таблица 12

Справка	QR-Code
Выравнивание, поля и отступы (ENG)	

Далее приведите файл `MainWindowView.axaml` к следующему виду (рис. 9).

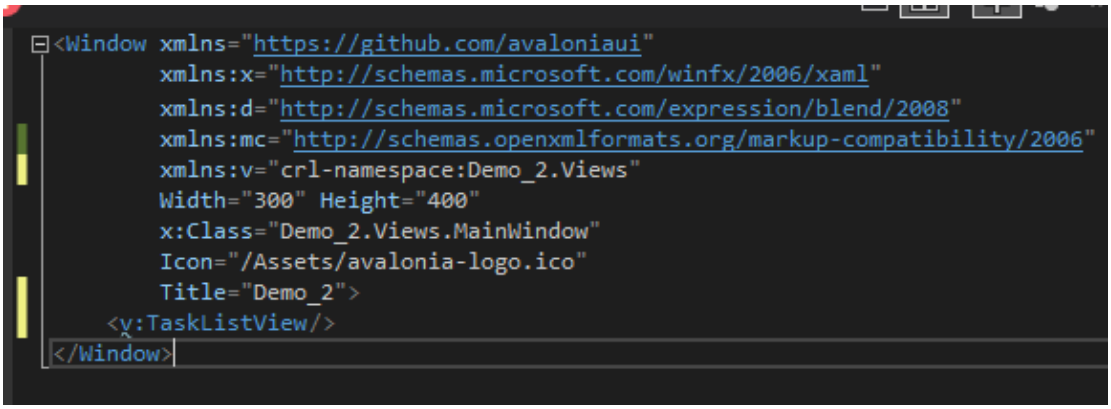


Рис. 9 Листинг отображения главного окна

В данном коде основной момент заключен в использовании сопоставления пространства имен XML (xmlns) и C# (namespace), а именно в строке `xmlns:v="clr-namespace:Demo_2.Views"`. Данная функция является возможностью языка XAML, позволяющая использовать объекты написанные на C# как тэги XML (таб. 7). В свою очередь, для окна указаны параметры ширины (`Width`), высоты (`Height`) и заголовок (`Title`). Стоит отметить, что свойства ширины и высоты есть практически у всех элементов интерфейса (выражаются в пикселях). Остальные свойства связанные с оформлением будут рассмотрены в лабораторной работе 4.

В строке `<v:TaskListView/>` создается объект класса `TaskListView`, располагающийся в `TaskListView.cs`, из сопоставленного пространства имен `v`. В результате создания этого объекта вызывается метод `InitializeComponent()` который в свою очередь на основе соответствующего `axaml`-файла создает пользовательский элемент интерфейса.

В качестве эксперимента так же можете включить встроенную «темный» режим темы `FluentTheme` приложения открыв файл `App.axaml` и изменив значение в строке `<FluentTheme Mode="Light">` на `Dark`.

Запустите приложение (клавиша F5), в результате должно открыться окно (рис. 10) созданного нами приложения.

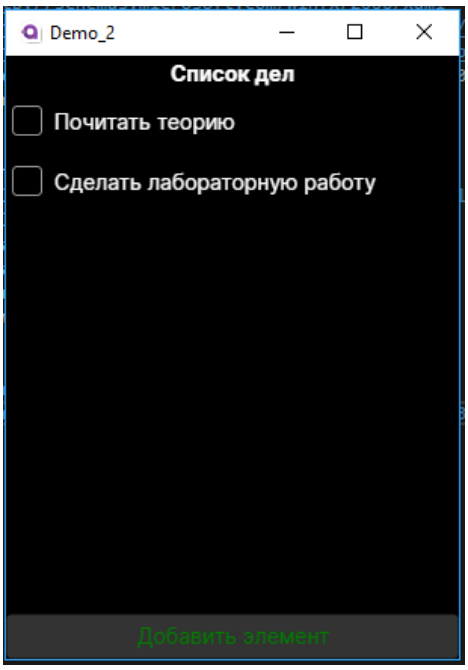


Рис. 10 Окно скомпилированной программы

Создание моделей

Обычно свойства элементов интерфейса (например, текст в `label` или `button`) вводятся напрямую в конструкторе или меняются в программном коде при обращении к элементу интерфейса напрямую. Однако в паттерне MVVM эта задача перекладывается на привязки, позволяющие интерфейсу меняться динамически.

Для хранения элементов нашего списка создайте класс «`Row`» в пространстве (в папке) имен «`Models`». В классе реализуйте два свойства, `public Val` и `public Ch`, с методами `Get` и `Set` (подробно про `Get` и `Set` в таблице 13).

Таблица 13

Справка	QR-Code
Get и Set	

В первое свойство будет представлять название задачи в виде строки, второе же логическое значение, указывающее, выделен ли `CheckBox` (рис. 11).

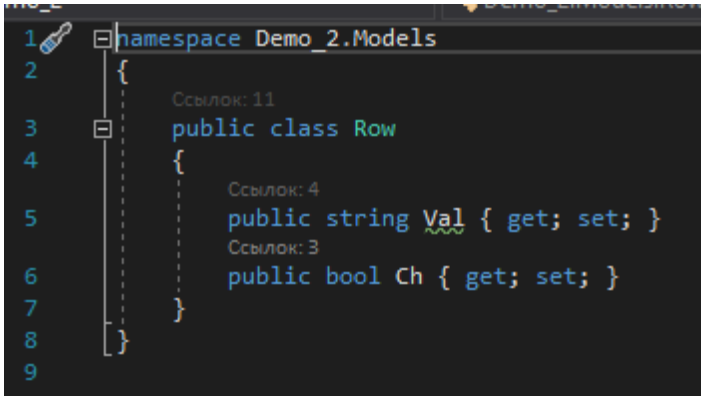


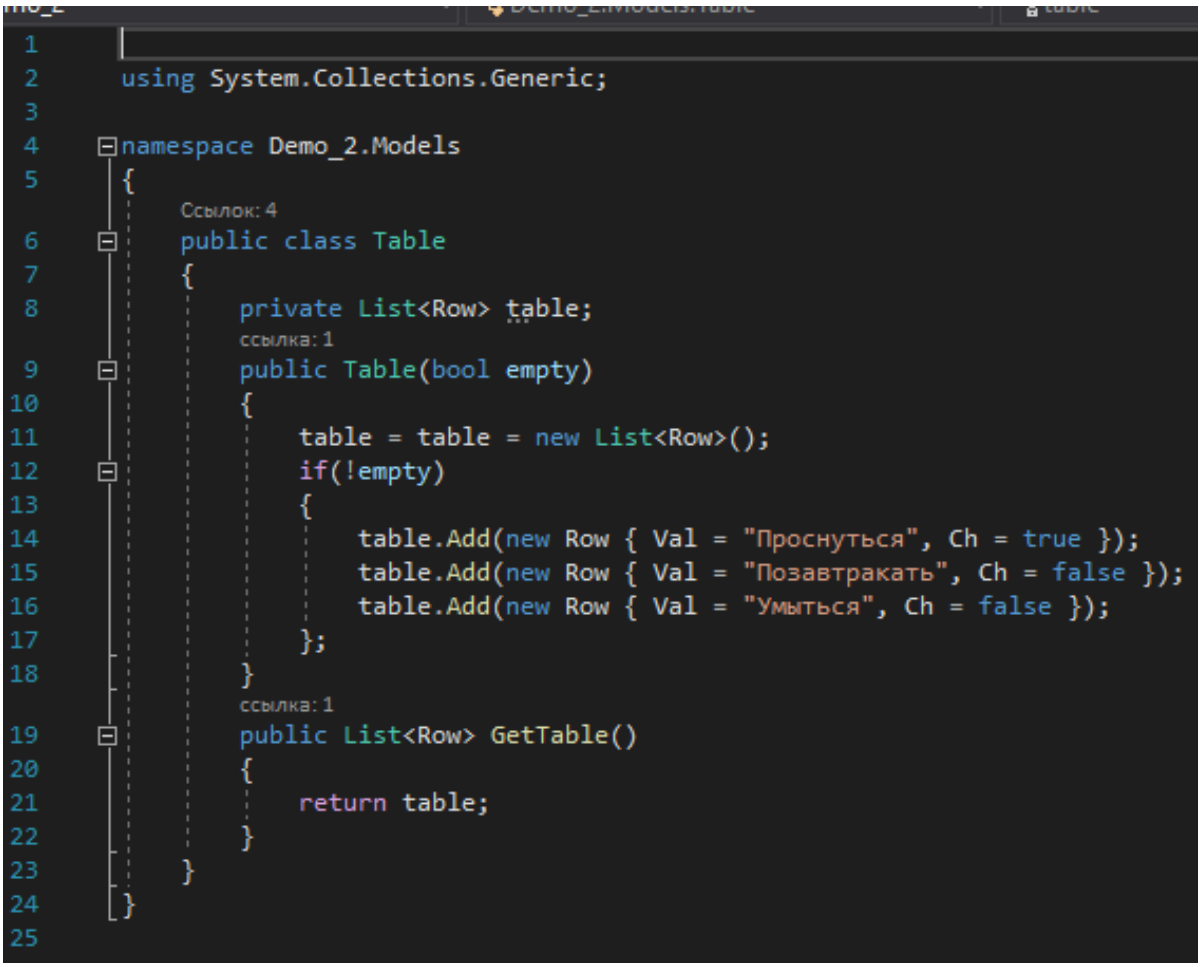
Рис. 11 Код класса Row

Следующий класс назовем «`Table`», этот класс будет являться списком элементов «`Row`». Данный класс также следует расположить в пространстве имен `Models`. В свою очередь, в качестве самой «таблицы» выступит объект класса `List<T>`, данный класс является представителем универсальных классов (таб. 14).

Таблица 14

Справка	QR-Code
Обобщения, универсальные классы	

Кроме того, реализуем возможность создавать таблицу пустой или с заранее созданными строками (рис. 12).



```
using System.Collections.Generic;

namespace Demo_2.Models
{
    Ссылок: 4
    public class Table
    {
        private List<Row> table;
        ссылка: 1
        public Table(bool empty)
        {
            table = table = new List<Row>();
            if(!empty)
            {
                table.Add(new Row { Val = "Проснуться", Ch = true });
                table.Add(new Row { Val = "Позавтракать", Ch = false });
                table.Add(new Row { Val = "Умыться", Ch = false });
            };
        }
        ссылка: 1
        public List<Row> GetTable()
        {
            return table;
        }
    }
}
```

Рис. 12 Код класса Table

Создание модели отображения

Следующими важными классами приложения являются модели представления, связывающие интерфейс и логику программы. Для этого создайте класс `TaskListViewModel` в пространстве имен `ViewModels` со следующим содержимым (рис. 13). В конструкторе преобразуем `List<T>` в `ObservableCollection<T>` так как данная коллекция позволяет оповещать другие объекты о своем изменении. Согласно описанию конструктора класса, данные, из объекта указанного в качестве параметра, будут скопированы вместо создания ссылки. Это значит, что при добавлении элемента в преобразованную коллекцию в исходном `List<T>` ничего не изменится.

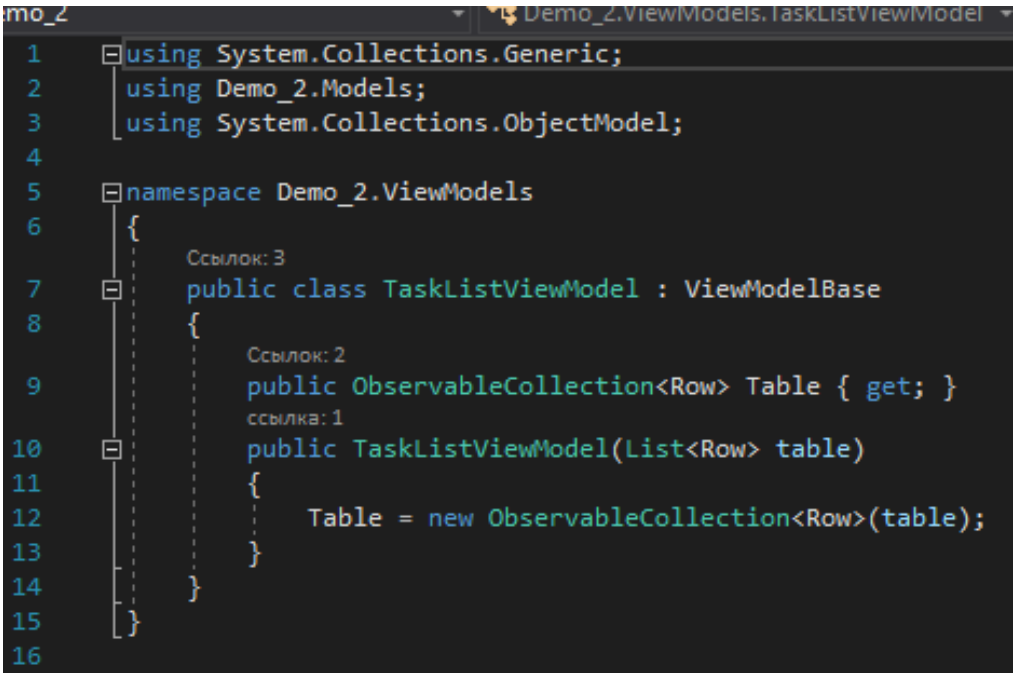


Рис. 13 Код класса TaskListViewModel

Модель представления главного окна уже создана, поэтому откройте `MainWindowViewModel.cs` и измените его согласно рисунку 14.

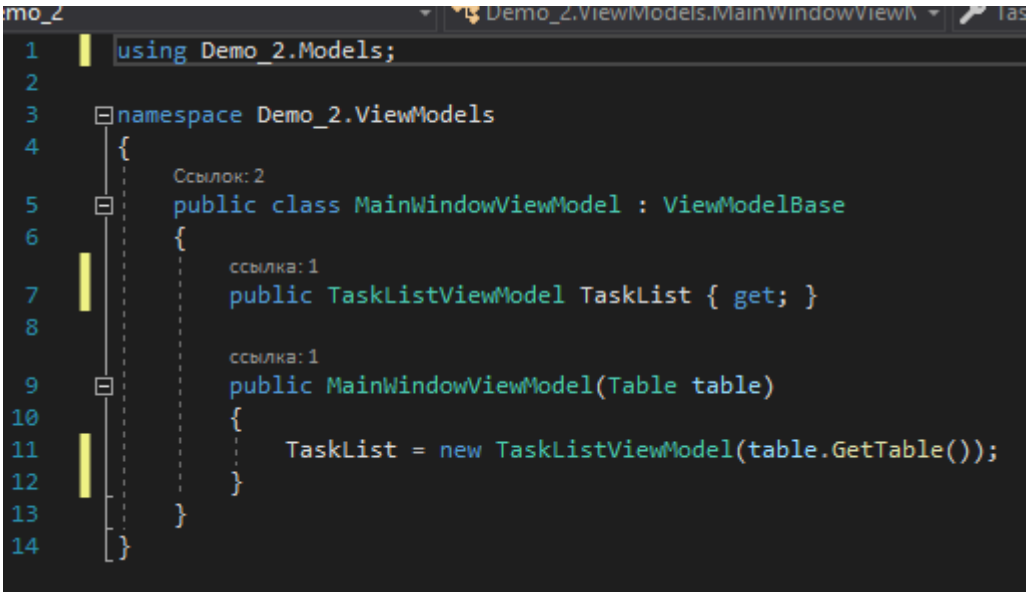


Рис. 14 Код класса MainWindowViewModel

В данном коде из главной модели представления была создана модель представления для `TaskListViewModel`, которой была передана ссылка на коллекцию.

Следующим шагом нужно инициализировать саму модель и передать ее конструктору модели представления главного окна. Для этого откройте файл `App.axaml.cs` и измените метод `OnFrameworkInitializationCompleted()` согласно рисунку 15. Значение параметра `empty` укажем `false` – это означает, что будет создана непустая таблица.

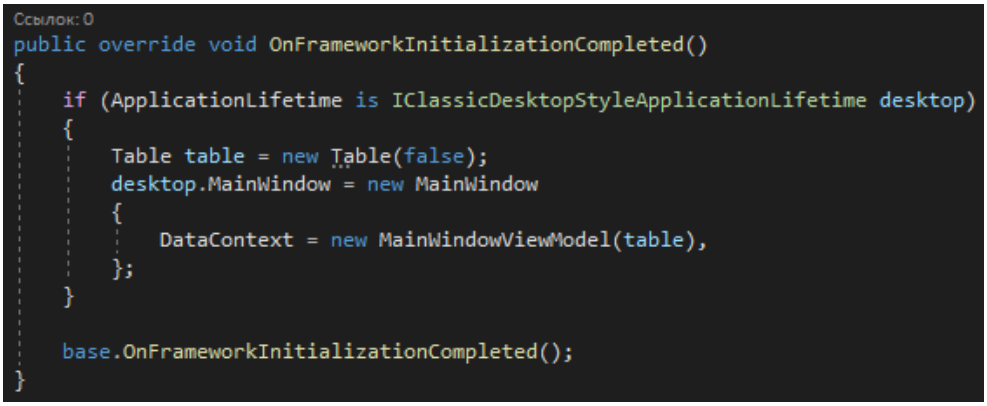


Рис. 15 Код метода OnFrameworkInitializationCompleted().

Введение привязок в отображение

Перепишем свойство тега `<Window>`, и введем привязку свойства `Content` к объекту `TaskListViewModel` (т.е. в качестве контента окна станет модель отображения, при этом соответствующее отображение подключится автоматически). Для этого удалите строку `<v:TaskListView/>` и заменить ее на свойство `Content="{Binding TaskList}"` объекта `<Window>`. Таким образом, можно выделить два варианта добавления элемента интерфейса: обратившись к его отображению напрямую или создать его модель отображения в качестве свойства и сделать к нему привязку. В свою очередь, строка `xmlns:v="clr-namespace:Demo_2.Views"` потеряла смысл, так как нам более не требуется обращаться к пространству имен содержащему отображения, поэтому ее можно удалить. Расширение XAML разметки `{Binding}` (англ. привязка) позволяет синхронизировать свойства элементов интерфейса и свойства объекта `DataContext` (обычно

ViewModel). В свою очередь, привязку можно задать как в коде C# так и с помощью конструкции свойство="{Binding Свойство}" (таб. 15).

Таблица 15

Справка	QR-Code
Привязки (ENG)	
Привязки из кода C# (ENG)	

На данный момент, при запуске нашей программы, список не изменится, так как в отображении пользовательского элемента до сих пор находятся CheckBox-ы с явно указанными значениями. Чтобы список изменялся динамически нужно ввести привязки к нашей «таблице».

Для привязки списка замените <StackPanel> с его содержимым кодом, который будет автоматически генерировать CheckBox-ы (рис. 16).

```
    </Button>
    <ItemsControl Items="{Binding Table}">
        <ItemsControl.ItemTemplate>
            <DataTemplate>
                <CheckBox Margin="4"
                          IsChecked="{Binding Ch}"
                          Content="{Binding Val}"/>
            </DataTemplate>
        </ItemsControl.ItemTemplate>
    </ItemsControl>
</DockPanel>
```

Рис. 16 Код генерирующий CheckBox-ы.

Первым классом в приведенном коде является <ItemsControl>, базовым классом для которого является <Control>. Задача данного класса – это отображение (перечисление) всех элементов списка (подобно foreach в

C#), на которые указывает свойство Items. В свою очередь, вложенный в него тег <ItemsControl.ItemTemplate> указывает шаблон отображения этих самых элементов, (в случае отсутствия шаблона, по умолчанию вызывается метод ToString()). <ItemTemplate> в качестве параметра принимает объект <DataTemplate>, хранящий в себе шаблон отображения данных. Запустите приложение, в результате окно приобретет следующий вид (рис. 17).

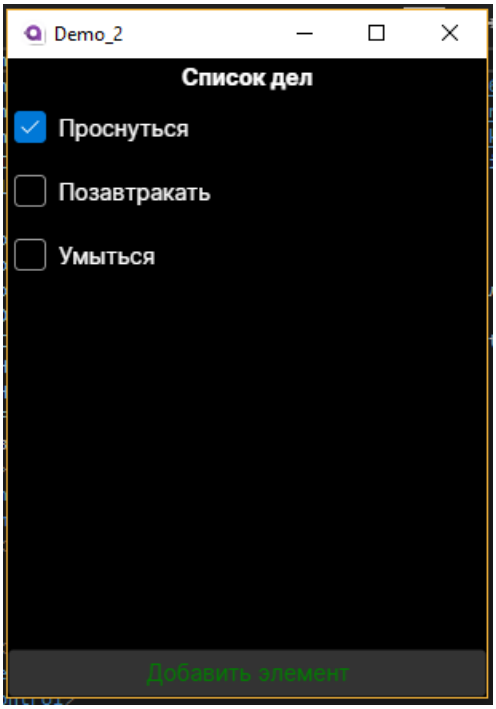


Рис. 17 Приложение с использованием привязок.

Привязка команд

Перед тем как создавать привязку команды к кнопке создайте еще один UserControl и назовите его AddTaskView. Данное отображение будет предоставлять форму, заполнив которую, пользователь сможет добавить новую запись в списке дел. Данное отображение заменит собой предыдущее отображение после того как пользователь нажмет на кнопку «Добавить элемент». Откройте соответствующий файл axaml и приведите его в соответствии с рисунком 18.

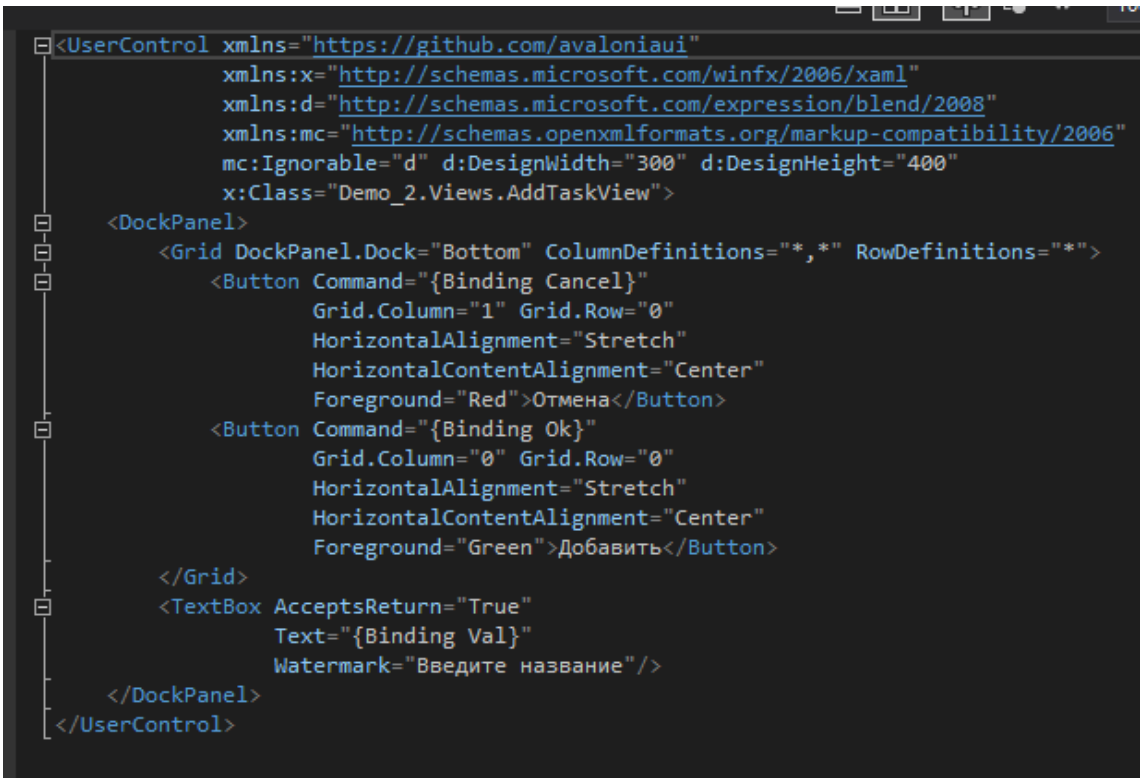


Рис. 18 Код отображения AddTaskView.

Новым элементом в данном коде является <Grid>. Grid – это панель, представляющая собой таблицу (сетку). Основными параметрами сетки являются ColumnDefenition и RowDefenition указывающие количество столбцов и строк таблицы и их размеры, разделяемые запятой (данный способ объявления является возможностью Avalonia, позволяющей не создавать строки вручную). В качестве размеров ячеек можно указывать * (равная ширина относительно родителя), Auto (ширина по контенту) и конкретное значение в пикселях (например, 20). В свою очередь, чтобы указать к какой ячейке относятся объекты внутри сетки нужно в присоединяемых свойствах самого объекта указать Grid.Row="номер_строки" и Grid.Column="номер_колонки" и для «растяжки» элемента на несколько строк и столбцов используются свойства Grid.RowSpan и Grid.ColumnSpan.

Для кнопок внутри «сетки» указаны команды, которые будут выполнены при нажатии, теперь определим эти команды. Для определения команд

создадим соответствующую модель отображения `AddTaskViewModel` со следующим кодом (рис. 19).

```
using System.Reactive;
using ReactiveUI;
using Demo_2.Models;

namespace Demo_2.ViewModels
{
    Ссылок: 2
    public class AddTaskViewModel : ViewModelBase
    {
        string val;
        ссылка: 1
        public AddTaskViewModel()
        {
            var okEnabled = this.WhenAnyValue(
                x => x.Val,
                x => !string.IsNullOrWhiteSpace(x));

            Ok = ReactiveCommand.Create(() => new Row { Val = Val }, okEnabled);
            Cancel = ReactiveCommand.Create(() => { });
        }

        Ссылок: 2
        public string Val
        {
            get => val;
            set => this.RaiseAndSetIfChanged(ref val, value);
        }

        Ссылок: 2
        public ReactiveCommand<Unit, Row> Ok { get; }
        Ссылок: 2
        public ReactiveCommand<Unit, Unit> Cancel { get; }
    }
}
```

Рис. 19 Модель отображения AddTaskViewModel.

Привязываемое свойство ViewModel обязательно должно иметь модификатор доступа `public` и может иметь следующие виды:

1) Если не требуется уведомление об изменении свойства, т.е. значение будет единожды установлено, и более изменяться не будет:

```
private Тип свойство;
public Тип Свойство{
     Get{ return свойство;}
}
```

2) Если требуется уведомление об изменении:

```
private Тип свойство;
public Тип Свойство{
     Get{return свойство;}
     Set{this.RaiseAndSetIfChanged(ref свойство, Value);
}
```

В Avalonia по умолчанию для реализации механизма MVVM используется библиотека ReactiveUI (таб. 16) (во многом упрощающая реализацию паттерна), однако Avalonia позволяет использовать и стандартные механизмы .NET или иные библиотеки (например, MVVM Light, Prism и т.д.).

Таблица 16

Справка	QR-Code
Справка ReactiveUI (ENG)	

Код, представленный выше, как последующие фрагменты будут содержать фрагменты, связанные с механизмами передачи функций в качестве параметра (механизм делегирования), очень часто для этого применяется лямбда-выражение – анонимная функция, которую можно передать в качестве параметра (таб. 17).

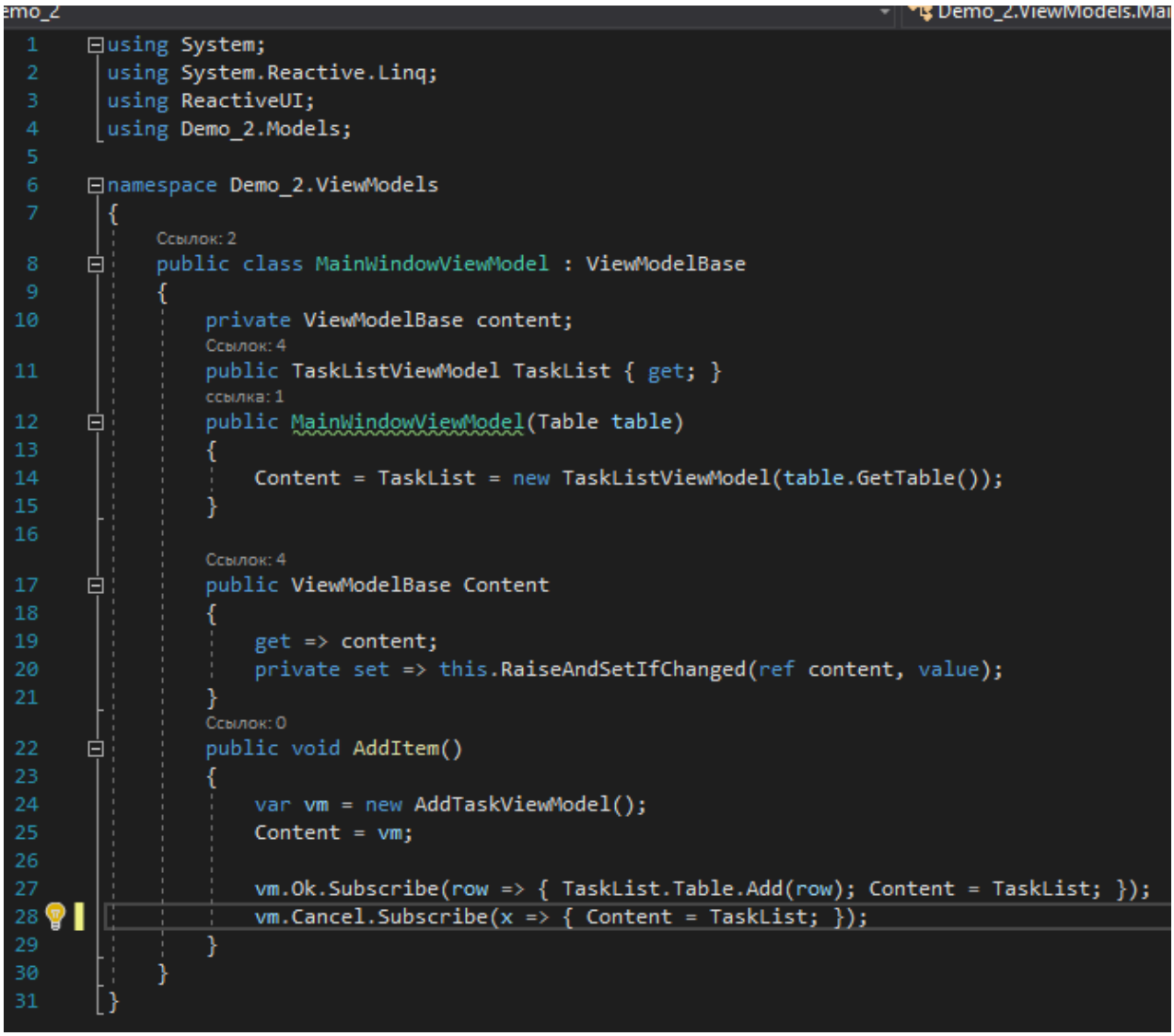
Таблица 17

Справка	QR-Code
Делегаты и лямбда-выражения	
Лямбда-выражения	

Стоит отметить, что в Avalonia по умолчанию все модели отображения наследуются от класса `ViewModelBase`, который сам наследуется от `ReactiveObject` – базового класса для моделей отображения в ReactiveUI, предоставляющего необходимые интерфейсы для написания свойств.

Функция `RaiseAndSetIfChanged` сигнализирует об изменении значения свойства (переменной) и должна располагаться в методе `Set` свойства. В качестве параметров функции используются ссылка (`ref`) на поле и новое значение поля. В свою очередь, функция `WhenAnyValue` выполняет асинхронную задачу (подробнее про асинхронность в следующей лабораторной работе). Данная функция предназначена для получения уведомления об изменении свойств объектов (в данном случае подписка идет на свойства текущего объекта (`this`)) (подробнее о функции в справке ReactiveUI (таб. 13)). Метод `Create()` статического класса `ReactiveCommand` создает синхронную команду, которая будет выполнена при нажатии кнопки, к которой она будет привязана. Первый вариант команды (`Ok`) поддерживает параметр, говорящий о том, может ли быть выполнена команда в данный момент, второй вариант команды (`Cancel`), не делает ничего, но событие нажатия все равно будет сгенерировано. В свою очередь, объявление самой команды включает в себя два обобщения включающие входной и выходной тип данных, по умолчанию эти типы объявляются как `Unit` и являются по своей сути «пустышками», однако если указать иной тип, то мы сможем передать значение или принять результат из команды. В случае команды `Ok` будет возвращен объект `Row`.

Для того чтобы обрабатывать действия кнопок измените *Main*`WindowViewModel` согласно рисунку 20.



```
using System;
using System.Reactive.Linq;
using ReactiveUI;
using Demo_2.Models;

namespace Demo_2.ViewModels
{
    Ссылок: 2
    public class MainWindowViewModel : ViewModelBase
    {
        private ViewModelBase content;
        Ссылок: 4
        public TaskListViewModel TaskList { get; }
        ссылка: 1
        public MainWindowViewModel(Table table)
        {
            Content = TaskList = new TaskListViewModel(table.GetTable());
        }

        Ссылок: 4
        public ViewModelBase Content
        {
            get => content;
            private set => this.RaiseAndSetIfChanged(ref content, value);
        }
        Ссылок: 0
        public void AddItem()
        {
            var vm = new AddTaskViewModel();
            Content = vm;

            vm.Ok.Subscribe(row => { TaskList.Table.Add(row); Content = TaskList; });
            vm.Cancel.Subscribe(x => { Content = TaskList; });
        }
    }
}
```

Рис. 20 Модель отображения MainWindowViewModel.

Новым элементом здесь будут вызовы функций `Subscribe()`. Данный метод описывает код, который будет выполнен, когда произойдет событие выполнения команды. Кроме того, механизм команд позволяет передать результат выполнения этой самой команды. Данная функция, а также множество других (например, `Where()` и т.д.) принадлежат пространству имен `System.Reactive` целью которых является предоставление набора статистических методов для операций запроса над наблюдаемыми последовательностями. Реактивные расширения (Rx) - это библиотека для составления асинхронных и основанных на событиях программ с использованием наблюдаемых последовательностей, и операторов запросов в стиле LINQ (таб. 18).

Таблица 18

Справка	QR-Code
dotnet/reactive (ENG)	
LINQ	

Откомпилируйте, запустите и проверьте работоспособность приложения.

Контрольные вопросы

1. *Прочитайте справку про свойства Get и Set. Для чего они применяются?*
2. *Прочитайте справку про Лямбда-выражения. Что такое анонимные функции?*
3. *Что такое ReactiveUI? Есть ли у него аналоги?*
4. *Что такое пространства имен XAML. Как работает сопоставление?*

Лабораторная работа №3

Создание приложения с применением шаблона асинхронного программирования (TAP) .

Источники

Данная лабораторная работа основана на примере приложения из официальной справки Avalonia Music Store (таб. 19).

Таблица 19

Справка	QR-Code
Avalonia Music Store (ENG)	

Асинхронное программирование

Написание приложений (в том числе и с графическим интерфейсом) неразрывно связано с операциями ввода-вывода (например, получение/отправка данных из сети, запросы к базе данных, работа с принтером, чтение/запись на накопитель и т.д.) и сложными вычислениями, которые очень часто занимают довольно продолжительное время. Одним из признаков современных приложений с GUI является их отзывчивость, например, окно архиватора, при запаковке/распаковке архива не зависает (не блокируется) до тех пор, пока не завершится задача, а отображает ее состояние полосой прогресса и статистикой (рис. 21). Кроме того приложение позволяет приостановить задачу или ее отменить не дожидаясь окончания выполнения.

Вышеуказанную проблему зависания можно решить двумя способами: применением многопоточности и асинхронности. Многопоточность – это свойство приложения, при котором процесс может состоять из нескольких потоков выполняющихся «параллельно», т.е. без предписанного порядка во времени. Параллельность в данном случае значит, что на самом деле коман-

ды все равно выполняются последовательно, при этом процессор выделяет свое «внимание» на определенный промежуток времени каждому потоку (квант времени). Во время написания приложения с многопоточностью программист сам указывает приложению, как распределить нагрузку.

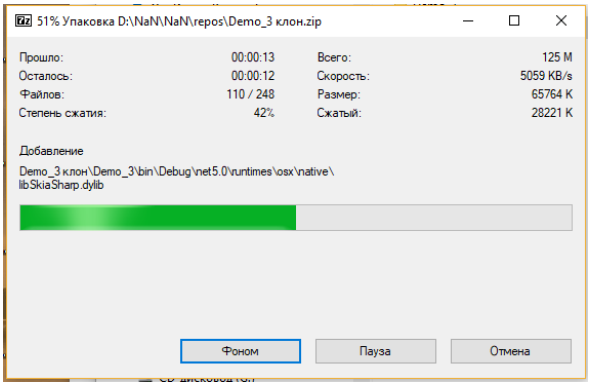


Рис. 21 Архиватор 7-Zip.

Применяя многопоточность для решения вопроса отзывчивости, интерфейс оставляют в основном потоке, а все трудоемкие задачи отправляют в фоновые потоки для выполнения в параллельном режиме. При этом если требуется уведомить интерфейс, о каком либо изменении, то используются делегаты и метод `Control.BeginInvoke()/Invoke()` (WinForms) или `Dispatcher.BeginInvoke()/Invoke()` (WPF) для изменений интерфейса в потоке UI (таб. 20).

Таблица 20

Справка	QR-Code
Метод Invoke (WinForms)	
Метод Invoke (WPF)	

Однако от приложения в некоторых случаях можно добиться большей производительности, например, после отправки запроса к базе данных на удаленном сервере, поток ожидает результат, хотя мог бы в это время выполнить полезную нагрузку. Данную задачу можно решить применением асинхронности. Асинхронность – это выполнение операций в неблокирующем стиле, т.е. без ожидания результата. Сама программа при этом разбивается на части и становится машиной состояний выполняющаяся в разных потоках. Во время выполнения сложной функции, если будет встречена асинхронная операция, то произойдет ее вызов, далее выполняется код до момента востребованности результатов асинхронной операции (если такой имеется). Далее в случае если операция не выполнена, то сохраняется текущее состояние функции и поток освобождается и возвращается в пул, в ином же случае код продолжает свое выполнение в этом же контексте. Когда асинхронная операция завершает работу (функция из которой она была вызвана сохранена), вызывается продолжение основной функции с того места где потребовался результат ее выполнения уже в новом потоке. В .NET есть 3 шаблона асинхронности EAP (Event-based Asynchronous Pattern), APM (Asynchronous Programming Model) и TAP (Task-based Asynchronous Pattern) (первые два шаблона на данный момент считаются устаревшими). Асинхронный код, написанный в паттерне TAP, имеет следующие особенности:

1) асинхронный метод помечается ключевым словом `async`, при этом он должен содержать внутри один или несколько `await`;

2) асинхронный метод должен возвращать `Task<T>` или `Task`. Кроме того метод может возвращать и тип `void`, однако такое решение имеет множество недостатков (например, исключения выброшенные в асинхронном методе должны быть обработаны только в этом же методе, и таким методам нельзя применить `await`) и должно применяться только там где это необходимо;

3) при написании асинхронных методов принято к их имени добавлять Async.

Асинхронный метод может выглядить следующим образом:

```
public async Task MyMethodAsync(){
    await …
}
```

При этом хоть и метод возвращает Task наличие return нужно только в случае использования Task<T> чтобы вернуть значение из функции. В свою очередь, конструкция получения результата асинхронной операции выглядит следующим образом:

```
var x = await MyMethodAsync();
```

В данном примере результат асинхронной операции получается сразу, однако можно метод вызвать заранее, а результат получить позднее:

```
Task task = MyMethodAsync();
…
var x = await task;
```

Говоря об асинхронности, стоит так же упомянуть про контекст синхронизации, а также планировщик заданий. Контекст синхронизации (SynchronizationContext) – это класс, обеспечивающий базовую функциональность для построения планировщиков заданий для определенных платформ.

Планировщик заданий – это объект, в цели которого входит постановка задачи в очередь на выполнение, автоматическая балансировкой нагрузки, а также их продолжение. При этом, абстрагируясь от понятия «поток», планировщик самостоятельно выбирает какую задачу в каком потоке выполнять. Примером планировщика задач может послужить вышеуказанный Dispatcher, метод BeginInvoke() которого является оберткой над методом Post обеспечивающий вызов кода в контексте UI.

В ReactiveUI так же есть два встроенных планировщика:

RxApp.MainThreadSheduler – это планировщик задач, выполняющий задачи в потоке пользовательского интерфейса.

RxApp.TaskpoolSheduler – это планировщик выполняющий код в ThreadPool библиотеки TPL (Task Parallel Library).

Подробно об асинхронном программировании можно узнать в официальной справке Microsoft (таб. 21).

Таблица 21

Справка	QR-Code
Асинхронное программирование	
Применение Async/Await (ENG)	

Создание главного окна

Создайте проект приложения «`Avalonia MVVM Application`» по аналогии с лабораторной работой №1 (в качестве названия проекта укажите «`Demo_3`»).

Первым шагом переключите встроенную тему `FluentTheme` в темный режим, открыв файл `App.axaml` и изменив значение в строке `<FluentTheme Mode="Light">` на `Dark`. Далее установите ширину и высоту окна равную 700 пикселей и добавьте `WindowStartupLocation="CenterScreen"` для того, чтобы приложение при запуске появлялось всегда в центре экрана. В свою очередь, для придания небольшой прозрачности окну, на манер стилей Windows 10/11, а также расширения стиля окна на его заголовок (подобно интерфейсам WEB-браузеров) в теге `<Window>` добавьте следующие свойства (рис. 22).

```
Background="Transparent"
TransparencyLevelHint="AcrylicBlur"
ExtendClientAreaToDecorationsHint="True"
```

Рис. 22 Свойства, добавляющие прозрачность.

Первое свойство устанавливает, что окно должно быть прозрачным, второе устанавливает режим прозрачности (акриловое размытие), и последнее свойство устанавливает требование расширения области окна на заголовок. Для большего эффекта добавьте следующий код (рис. 23).

```
<Panel>
    <ExperimentalAcrylicBorder IsHitTestVisible="False">
        <ExperimentalAcrylicBorder.Material>
            <ExperimentalAcrylicMaterial
                BackgroundSource="Digger"
                TintColor="Black"
                TintOpacity="1"
                MaterialOpacity="0.65" />
        </ExperimentalAcrylicBorder.Material>
    </ExperimentalAcrylicBorder>

</Panel>
```

Рис. 23 Объект, добавляющий эффект размытия на манер windows 10/11.

После `</ExperimentalAcrylicBorder>` добавьте в `Panel` еще одну с отступом в 40 пикселей, в которую поместим `Grid` с одной строкой и двумя ячейками. В первую ячейку поместите надпись, в другую кнопку (рис. 24).

```
<Panel Margin="40">
    <Grid ColumnDefinitions="*,*" RowDefinitions="*">
        <Label Content="Коллекция альбомов"
               Grid.Column="0"
               Grid.Row="0"/>
        <Button Content="Search Music"
                Command="{Binding SearchMusicCommand}"
                HorizontalAlignment="Right"
                VerticalAlignment="Top"
                Grid.Column="1"
                Grid.Row="0">
            <PathIcon Data="{StaticResource music_pic}"/>
        </Button>
    </Grid>
</Panel>
```

Рис. 24 Заготовка интерфейса.

В приведенном выше примере появился новый тег <PathIcon> предназначенный для представления векторных значков, свойство Data принимает данные в виде разметки пути написанных на специальном языке (таб. 22). Данный тег позволит нам заменить текст кнопки на векторное изображение.

Таблица 22

Справка	QR-Code
Синтаксис разметки пути	

Сами векторные данные вынесем как объект StreamGeometry в файл App.axaml и сделаем их общим ресурсом для приложения. Для этого добавьте в App.axaml после тега </Application.Styles> следующий код (рис. 25).

```
<Application.Resources>
    <StreamGeometry x:Key="music_pic"></StreamGeometry>
</Application.Resources>
```

Рис. 25 Объявление ресурса.

Далее, поместите внутрь тега <StreamGeometry> векторную разметку:

```
M11.5,2.75 C11.5,2.22634895 12.0230228,1.86388952 12.5133347,2.04775015
L18.8913911,4.43943933 C20.1598961,4.91511241 21.0002742,6.1277638
21.0002742,7.48252202 L21.0002742,10.7513533 C21.0002742,11.2750044
20.4772513,11.6374638 19.9869395,11.4536032 L13,8.83332147 L13,17.5 C13,17.5545945
12.9941667,17.6078265 12.9830895,17.6591069 C12.9940859,17.7709636 13,17.884807
13,18 C13,20.2596863 10.7242052,22 8,22 C5.27579485,22 3,20.2596863 3,18
C3,15.7403137 5.27579485,14 8,14 C9.3521238,14 10.5937815,14.428727
11.5015337,15.1368931 L11.5,2.75 Z M8,15.5 C6.02978478,15.5 4.5,16.6698354 4.5,18
C4.5,19.3301646 6.02978478,20.5 8,20.5 C9.97021522,20.5 11.5,19.3301646 11.5,18
C11.5,16.6698354 9.97021522,15.5 8,15.5 Z M13,3.83223733 L13,7.23159672
L19.5002742,9.669116 L19.5002742,7.48252202 C19.5002742,6.75303682
19.0477629,6.10007069 18.3647217,5.84393903 L13,3.83223733 Z
```

Кроме того, разработчики Avalonia представили готовый набор иконок, созданных для темы Fluent (таб. 23).

Таблица 23

Справка	QR-Code
Иконки для темы Fluent (ENG)	

Код для иконок можно написать, как самостоятельно, используя соответствующий язык, так и с использованием конвертации популярных векторных форматов (например, SVG) в разметку XAML. Возможность конвертации предоставляет векторный графический редактор Inkscape.

Вернемся к созданию команды для кнопки, для этого в модели отображения главного окна добавьте конструктор, а также добавьте свойство класса `ReactiveCommand<Unit,Unit> SearchMusicCommand {get;}` и проинициализируйте его из конструктора (рис. 26).

```
1   using System;
2   using System.Collections.Generic;
3   using System.Collections.ObjectModel;
4   using System.Reactive;
5   using System.Reactive.Linq;
6   using System.Text;
7   using System.Windows.Input;
8   using ReactiveUI;
9   using System.Reactive.Concurrency;
10  using Demo_3.Models;
11
12
13  namespace Demo_3.ViewModels
14  {
        Ссылок: 3
15      public class MainWindowViewModel : ViewModelBase
16      {
            ссылка: 1
17          public ReactiveCommand<Unit, Unit> SearchMusicCommand { get; }
            ссылка: 1
18          public MainWindowViewModel()
19          {
20              SearchMusicCommand = ReactiveCommand.Create(() => { });
21          }
22      }
23  }
```

Рис. 26 Код MainWindowViewModel.

После компиляции приложения получится следующее окно (рис. 26).

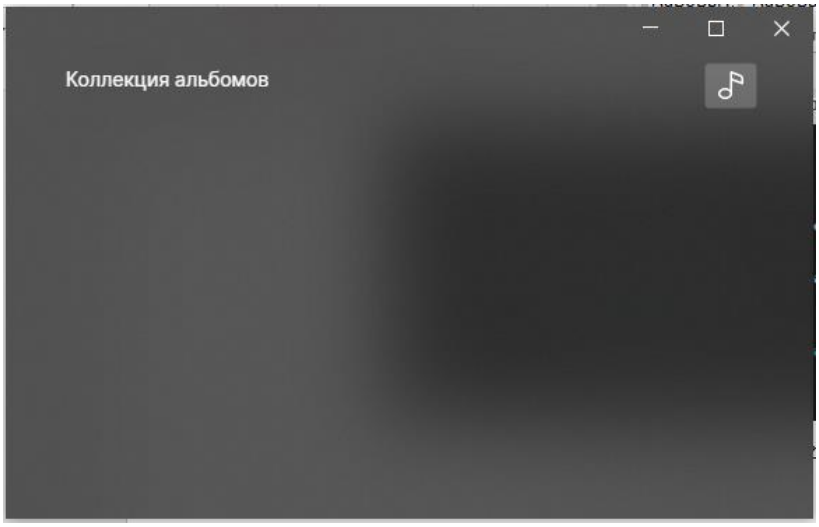


Рис. 26 Главное окно программы.

Создание диалогового окна

На данном этапе кнопка только генерирует событие своего нажатия, сделаем так, чтобы при ее нажатии открывалось окно. Для этого создайте в каталоге `Views` новый объект `Window`, назовите его `SubWindow.axaml` и измените его код согласно рисунку 27.

```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        x:Class="Demo_3.Views.SubWindow"
        Width="700" Height="700"
        Title="SubWindow"
        TransparencyLevelHint="AcrylicBlur"
        ExtendClientAreaToDecorationsHint="True"
        WindowStartupLocation="CenterOwner"
        Icon="/Assets/avalonia-logo.ico">
    <Panel>
        <ExperimentalAcrylicBorder IsHitTestVisible="False">
            <ExperimentalAcrylicBorder.Material>
                <ExperimentalAcrylicMaterial
                    BackgroundSource="Digger"
                    TintColor="Black"
                    TintOpacity="1"
                    MaterialOpacity="0.65" />
            </ExperimentalAcrylicBorder.Material>
        </ExperimentalAcrylicBorder>

        <Panel Margin="40">

        </Panel>
    </Panel>
</Window>
```

Рис. 27 Код SubWindow.

Для открытия диалоговых окон из `ViewModel` ReactiveUI предоставляет такой объект как `Interaction<TInput, TOutput>`. Метод `Handle(TInput object)` объекта `Interaction` вызывает событие открытия диалогового окна и возвращает `TOutput object` результат диалогового окна. Измените код `MainWindowViewModel` согласно рисунку 28.

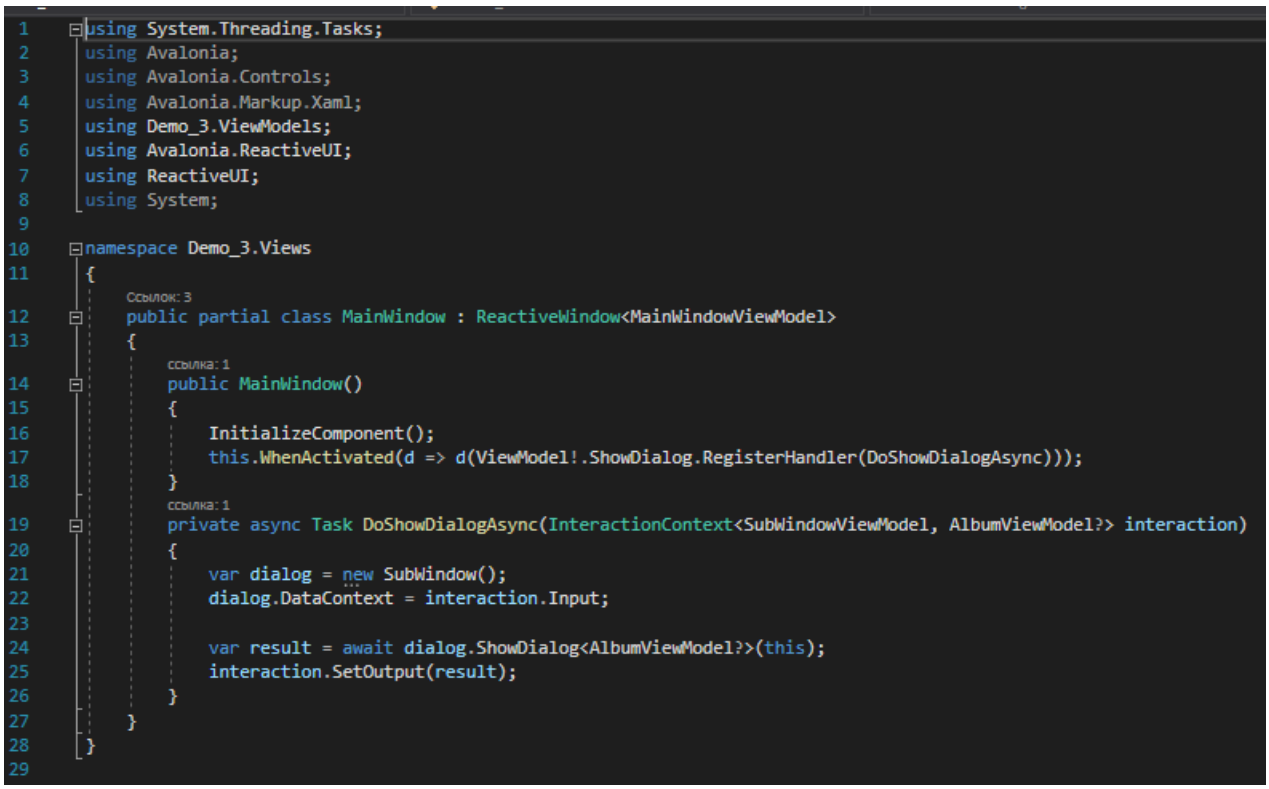
```
public ReactiveCommand<Unit, Unit> SearchMusicCommand { get; }
Ссылок: 3
public Interaction<SubWindowViewModel, AlbumViewModel?> ShowDialog { get; }
ссылка: 1
public MainWindowViewModel()
{
    ShowDialog = new Interaction<SubWindowViewModel, AlbumViewModel?>();
    SearchMusicCommand = ReactiveCommand.CreateFromTask(async () =>
    {
        var store = new SubWindowViewModel();
        var result = await ShowDialog.Handle(store);
    });
}
```

Рис. 28 Конструктор MainWindowViewModel.

Метод `CreateFromTask()` отличается от `Create` тем, что принимает в качестве аргумента асинхронный метод вместо синхронного. В данном коде создается модель отображения диалогового окна, после она передается в качестве входного параметра методу `handle()`. В свою очередь, ключевое слово `await` позволяет не блокировать поток UI ожиданием завершения диалога с пользователем, а получить результат `handle()` позднее.

Прежде чем идти дальше, нужно создать в папке `View` классы `SubWindowViewModel` и `AlbumViewModel` (пока пустые), также при этом указать, что данные классы наследуются от `ViewModelBase`.

Вызов `handle()` не откроет само окно, для этого нужно в code behind (`MainWindow.xaml.cs`) окна `MainWindow` написать и зарегистрировать обработчик, который его вызовет (рис. 29).



```
using System.Threading.Tasks;
using Avalonia;
using Avalonia.Controls;
using Avalonia.Markup.Xaml;
using Demo_3.ViewModels;
using Avalonia.ReactiveUI;
using ReactiveUI;
using System;

namespace Demo_3.Views
{
    Ссылок: 3
    public partial class MainWindow : ReactiveWindow<MainWindowViewModel>
    {
        ссылка: 1
        public MainWindow()
        {
            InitializeComponent();
            this.WhenActivated(d => d(ViewModel!.ShowDialog.RegisterHandler(DoShowDialogAsync)));
        }
        ссылка: 1
        private async Task DoShowDialogAsync(InteractionContext<SubWindowViewModel, AlbumViewModel?> interaction)
        {
            var dialog = new SubWindow();
            dialog.DataContext = interaction.Input;

            var result = await dialog.ShowDialog<AlbumViewModel?>(this);
            interaction.SetOutput(result);
        }
    }
}
```

Рис. 29 Код C# формы.

Первое, что нужно отметить, это то, что вместо `Window` наше окно наследуется от `ReactiveWindow<TViewModel>`. Это позволит нам обратится к соответствующей модели отображения и подписаться на события ее свойств. Для подписки будет использован метод `WhenActivated()` (позволяет подписаться на событие в момент активации отображения). Созданный обработчик события `DoShowDialogAsync` принимает в качестве параметра объект `InteractionContext<TInput, TOutput>` из которого можно получить входное значение для диалога `(Input)` и отправить результат методом `SetOutput()`. При этом данный метод асинхронно ждет результата диалога, чтобы его передать обратно в место вызова. После компиляции приложения, можно открыть диалоговое окно кнопкой (рис. 30).

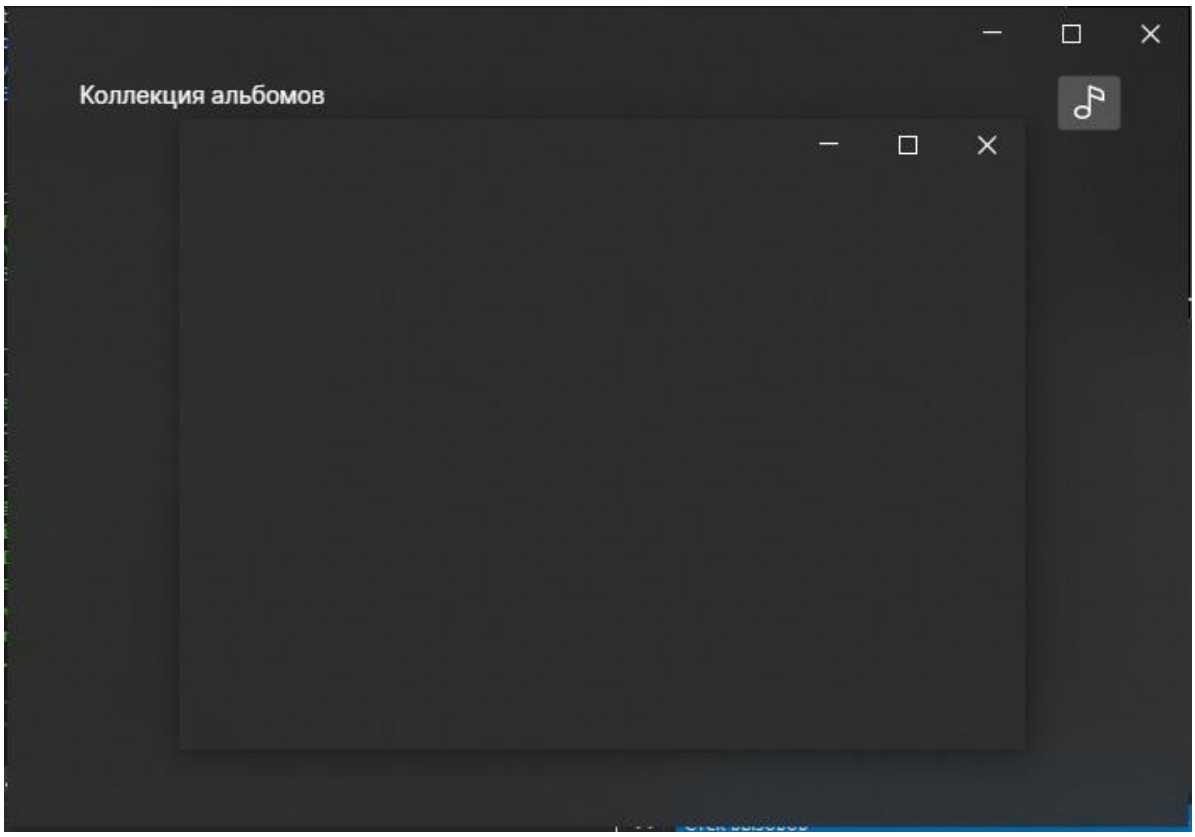


Рис. 30 Диалоговое окно.

В диалоговом окне будет происходить поиск музыкальных альбомов по названию или группе. Далее пользователю будет предоставлен выбор из предложенных вариантов, после которого происходит подтверждение выбора и добавление альбома в коллекцию кнопкой.

В свою очередь, для этого нам понадобятся следующие объекты:

1) `<TextBox>` (поле для ввода текстовых данных) для ввода информации и поиска;
2) `<ProgressBar>` (полоса прогресса) для сообщения пользователю, что его поисковый запрос выполняется;
3) `<ListBox>` для отображения результата поиска;
4) `<Button>` для подтверждения добавления в коллекцию.

Кроме того, для размещения объектов понадобятся панели. Для общей разметки можно использовать `<DockPanel>` которая позволит интерфейсу при изменении формы окна фиксировать положение условной «шапки»,

«подвала» и «сайдбаров» на своих местах. Для упорядочивания строки поиска и прогрессбара можно использовать <StackPanel> или <Grid>.

Добавьте в <Panel> файла SubWindow.axaml следующий код «шапки» окна (рис. 31).

```
<Panel Margin="40">
    <DockPanel>
        <StackPanel DockPanel.Dock="Top">
            <TextBox Text="{Binding SearchText}"
                     Watermark="Поиск альбома (введите название или группу)...." />
            <ProgressBar IsIndeterminate="True"
                         IsVisible="{Binding IsBusy}" />
        </StackPanel>
    </DockPanel>
</Panel>
```

Рис. 31 Шапка окна поиска.

За счет свойства DockPanel.Dock="Top" происходит фиксация <StackPanel> и его содержимого в верхней части окна. В поле <TextBox> происходит привязка к свойству Text, содержимое которого будет использовано для поиска. В свою очередь, свойство Watermark устанавливает текст, (обычно справочный) который будет отображаться в поле до тех пор, пока пользователь не начнет ввод. В прогрессбаре свойство IsIndeterminate устанавливает «характер» состояния в виде неопределенного, т.е. прогрессбар не будет отображать конкретное значение. Свойство IsVisible устанавливает видимость объекта для пользователя, поэтому к нему будет выполнена привязка, чтобы прогрессбар отображался только тогда, когда выполняется поиск. Добавим соответствующие свойства класса в SubWindowViewModel (рис. 32).

```
using System;
using System.Collections.ObjectModel;
using System.Linq;
using System.Reactive;
using System.Reactive.Linq;
using System.Threading;
using Demo_3.Models;
using ReactiveUI;


namespace Demo_3.ViewModels
{
    Ссылок: 5
    public class SubWindowViewModel : ViewModelBase
    {
        private bool _isBusy;
        Ссылок: 0
        public bool IsBusy
        {
            get => _isBusy;
            set => this.RaiseAndSetIfChanged(ref _isBusy, value);
        }
        private string? _searchText;
        Ссылок: 0
        public string? SearchText
        {
            get => _searchText;
            set => this.RaiseAndSetIfChanged(ref _searchText, value);
        }
    }
}
```

Рис. 32 Свойства для строки поиска и прогрессбара.

Добавьте после тэга </StackPanel> объекты `<Button>` и `<List>` со следующими свойствами (рис. 33).

```
<Button Command="{Binding AddMusicCommand}"
        Content="Добавить в коллекцию"
        DockPanel.Dock="Bottom"
        HorizontalAlignment="Center" />
<ListBox Items="{Binding SearchResults}"
         SelectedItem="{Binding SelectedAlbum}"
         Background="Transparent"
         Margin="0 20"/>
```

Рис. 33 Кнопка «Добавить в коллекцию» и список результатов поиска.

Также добавите свойство для выбранного альбома и коллекцию результатов поиска в SubWindowViewModel (рис. 34). Кроме того, добавьте в коллекцию несколько элементов для проверки работоспособности.

```
private AlbumViewModel? _selectedAlbum;
Ссылок: 0
public AlbumViewModel? SelectedAlbum
{
    get => _selectedAlbum;
    set => this.RaiseAndSetIfChanged(ref _selectedAlbum, value);
}
Ссылок: 3
public ObservableCollection<AlbumViewModel> SearchResults { get; } = new();
ссылка: 1
public SubWindowViewModel()
{
    SearchResults.Add(new AlbumViewModel());
    SearchResults.Add(new AlbumViewModel());
    SearchResults.Add(new AlbumViewModel());
}
```

Рис. 34. Код SubWindowViewModel.

Перед тем как проверить работоспособность приложения создайте `UserControl AlbumView.axaml` в каталоге `Views`. В результате получится следующее диалоговое окно (рис. 35).

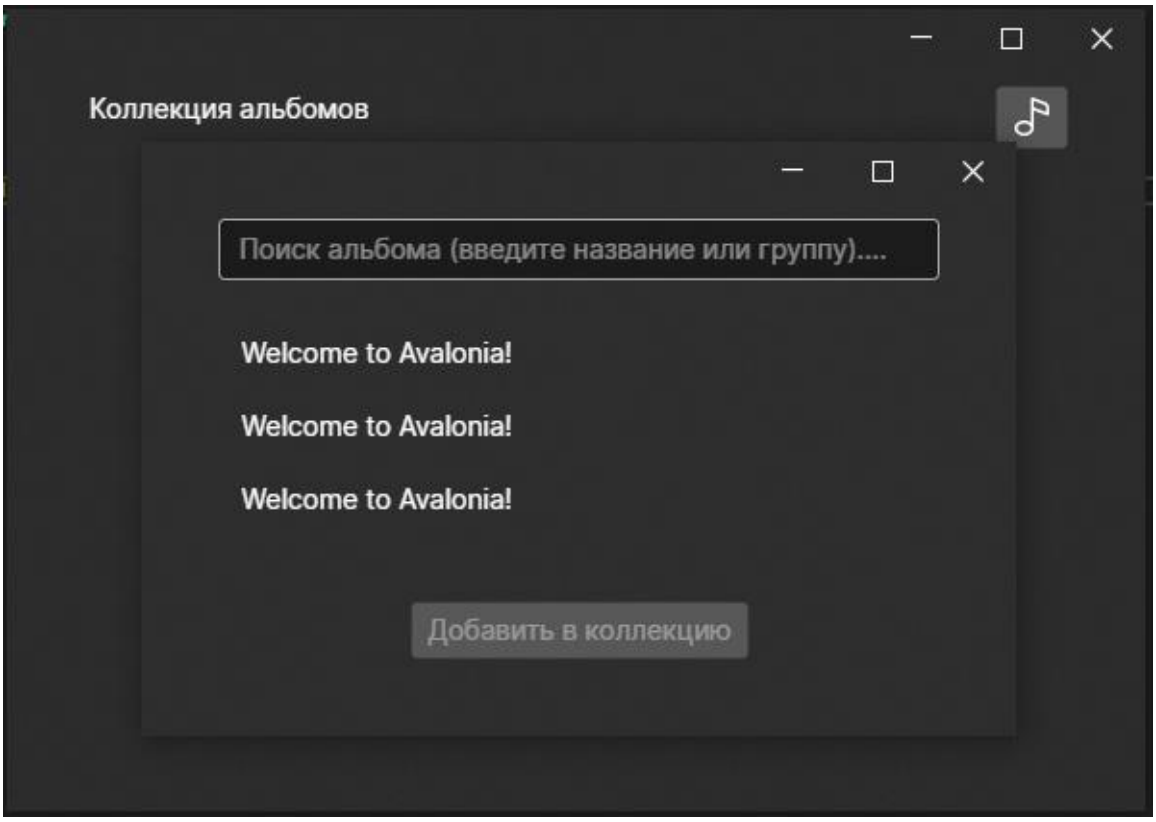


Рис. 35. Диалоговое окно.

При создании объекта модели отображения альбома и его привязки Avalonia автоматически пытается найти соответствующее отображение и выве-

сти его в соответствующем месте. Теперь, вместо стандартной надписи, отобразим обложку, название альбома и исполнителя. Для этого добавьте в `AlbumView.axaml` следующий код (рис. 36).

```
<StackPanel Spacing="5" Width="200">
    <Border CornerRadius="10" ClipToBounds="True">
        <Panel Background="#7FFF22DD">
            <Image Width="200" Stretch="Uniform" Source="{Binding Cover}" />
            <Panel Height="200" IsVisible="{Binding Cover, Converter={x:Static ObjectConverters.IsNull}}">
                <PathIcon Height="75" Width="75" Data="{StaticResource music_pic}" />
            </Panel>
        </Panel>
    </Border>
    <TextBlock Text="{Binding Title}" HorizontalAlignment="Center" />
    <TextBlock Text="{Binding Artist}" HorizontalAlignment="Center" />
</StackPanel>
```

Рис. 36. Отображение альбома.

В данном коде создается `<StackPanel>` с пустым пространством вокруг (`Spacing`) в 5 пикселей и шириной в 200 пикселей (`Width`). Внутри располагается объект `<Border>` предназначенный для показа рамки или фона у которого скруглены края (`CornerRadius`) и установлено усечение внутренних элементов по размеру родителя (`ClipToBounds`). Кроме того, здесь же располагаются два текстовых элемента `<TextBlock>` к свойствам Text которых сделаны привязки. В `<Border>` помещена панель с пурпурным фоном (`Background`) (так как `<Border>` не поддерживает множество элементов). Первым объектом внутри панели является `<Image>` предназначенный для показа изображения шириной (Width) в 200 пикселей с сохранением пропорций (`Stretch="Uniform"`) и источником изображения в виде объекта к которому происходит привязка (если свойство `null`, то объект не выводится). Вторым объектом выступает панель высотой 200 пикселей содержащая уже знакомую нам иконку, но увеличенную до 75x75 пикселей. Свойство видимости зависит от того чему равно Cover. Само свойство видимости имеет тип `bool` поэтому `Cover` нужно преобразовать в логическое значение с помощью конвертора (таб. 24). В данном случае конвертер вернет истину в том случае если `Cover=null`.

Таблица 24

Справка	QR-Code
Конвертация привязываемых значений (ENG)	

Если сейчас проверить вывод диалогового окна, то элементы в нем будут располагаться в столбик. Чтобы элементы заполняли все отведенное пространство нужно изменить `<ListBox>` из `SubWindowViewModel` следующим образом (рис. 37).

```
<ListBox Items="{Binding SearchResults}"
         SelectedItem="{Binding SelectedAlbum}"
         Background="Transparent"
         Margin="0 20">
<ListBox.ItemsPanel>
        <ItemsPanelTemplate>
            <WrapPanel />
        </ItemsPanelTemplate>
    </ListBox.ItemsPanel>
</ListBox>
```

Рис. 37. ListBox с шаблоном отображения данных.

`ListBox` по умолчанию использует `StackPanel` для отображения элементов внутри себя (в один столбик), при этом, с помощью свойства `ItemsPanel`, можно данную панель переопределить. Для этого потребуется объект `<ItemsPanelTemplate>` предназначенный для указания панели, которая будет использоваться для размещения элементов. Таким образом, в коде выше мы указали <ListBox>-у что элементы следует упорядочить как `<WrapPanel>`. <WrapPanel> – это панель упорядочивающая дочерние элементы слева на право. При этом если элементы не помещаются в строке, то они автоматически переносятся на новую строку. После запуска приложения диалоговое окно буде выглядеть следующим образом (рис. 38).

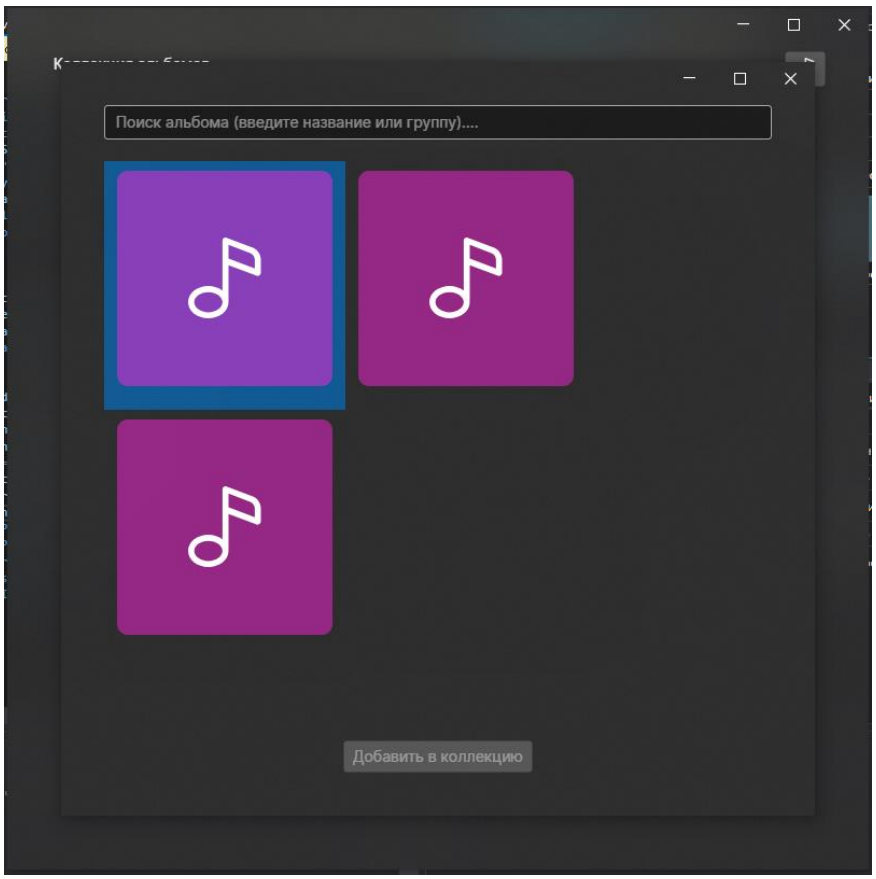


Рис. 38. Диалоговое окно с контентом.

Добавим функциональности кнопке, для этого добавьте в `SubWindowViewModel` команду (рис. 39), которая будет возвращать в место вызова (рис. 29, переменная result) выбранный альбом (рис. 40). На генерируемое событие нажатия кнопки создайте в `SubWindow.axaml.cs` обработчик событий, который вызовет встроенный метод `Close()` окна (рис. 41).

```
Ссылок: 2
public ReactiveCommand<Unit, AlbumViewModel?> AddMusicCommand { get; }
```

Рис. 39. Объявление свойства.

```
public SubWindowViewModel()
{
    AddMusicCommand = ReactiveCommand.Create(() => { return SelectedAlbum; });
}
```

Рис. 40. Инициализация.

```
Ссылок: 3
public partial class SubWindow : ReactiveWindow<SubWindowViewModel>
{
    ссылка: 1
    public SubWindow()
    {
        InitializeComponent();
        this.WhenActivated(d => d(ViewModel!.AddMusicCommand.Subscribe(Close)));

    }
}
```

Рис. 41. Обработчик событий.

Удалите три строки «заглушки» (рис. 34) заполняющие коллекцию результатов вывода.

Реализация модели и модели отображения

Реализуем поиск альбомов в музыкальном сервисе iTunes, для этого добавим к проекту библиотеку iTunesSearch. Добавление библиотеки происходит следующим образом: `Проект` → `Управление пакетами NuGet` → вкладка `Обзор` → введите в строку поиска `iTunesSearch` → Установить (рис. 42).

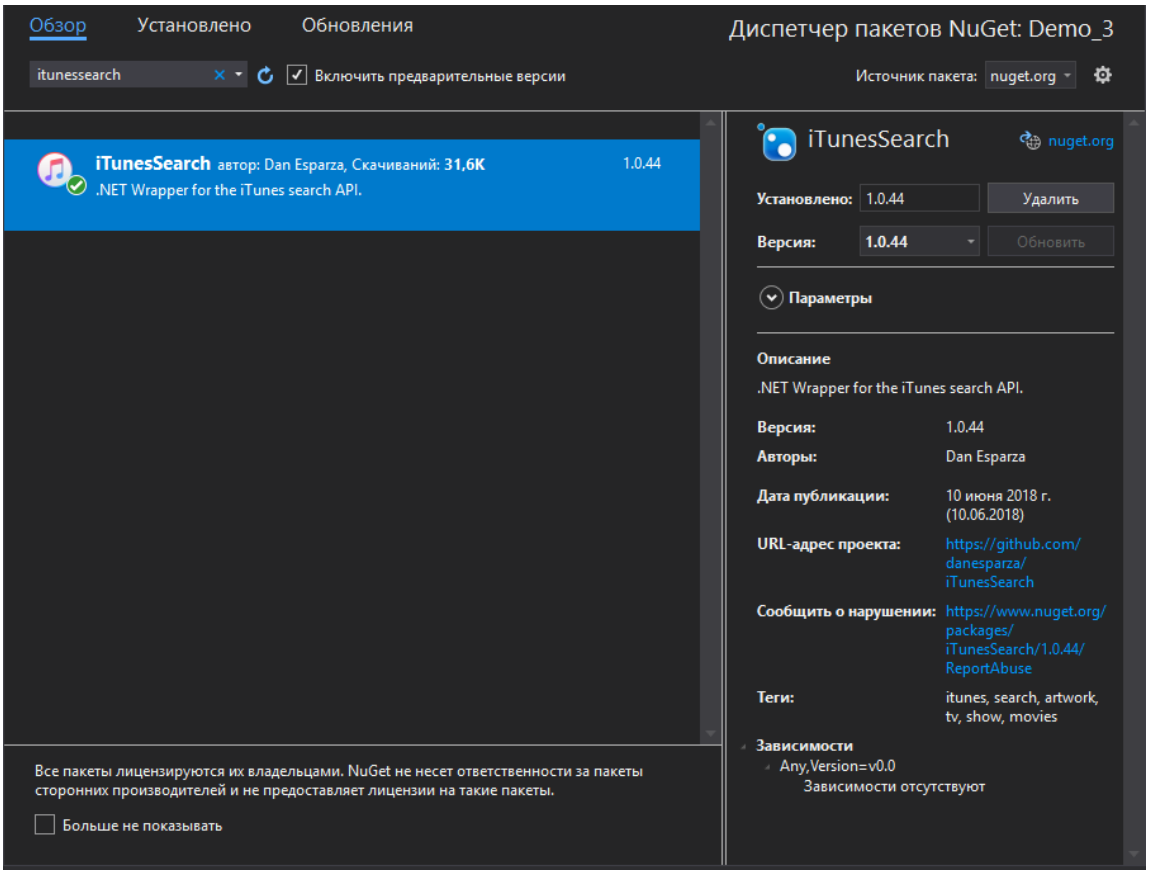
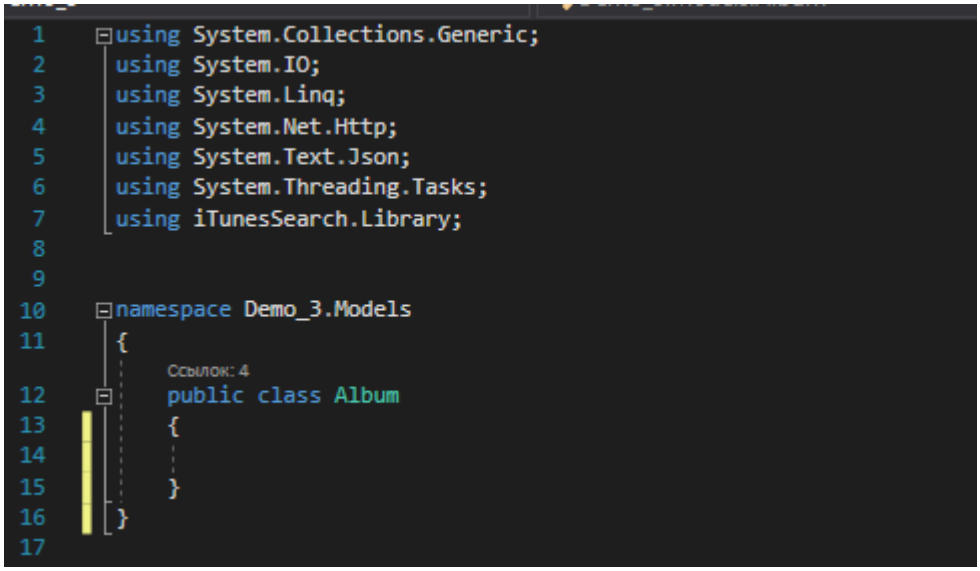


Рис. 42. Диспетчер пакетов NuGet.

Создайте класс `Album` в каталоге `Models` (рис. 43), со свойствами (рис. 44), конструктором (рис. 45) и методами (рис. 46-52).

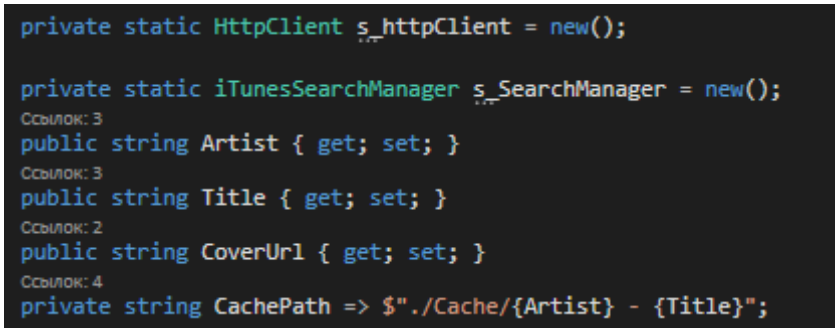


```
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Net.Http;
using System.Text.Json;
using System.Threading.Tasks;
using iTunesSearch.Library;


namespace Demo_3.Models
{
    Ссылок: 4
    public class Album
    {

    }
}
```

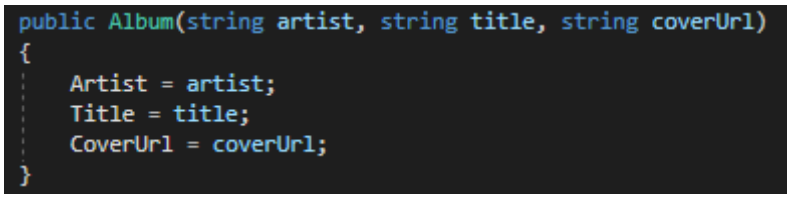
Рис. 43. Класс Album.



```
private static HttpClient s_httpClient = new();

private static iTunesSearchManager s_SearchManager = new();
Ссылок: 3
public string Artist { get; set; }
Ссылок: 3
public string Title { get; set; }
Ссылок: 2
public string CoverUrl { get; set; }
Ссылок: 4
private string CachePath => $"./Cache/{Artist} - {Title}";
```

Рис. 44. Свойства.



```
public Album(string artist, string title, string coverUrl)
{
    Artist = artist;
    Title = title;
    CoverUrl = coverUrl;
}
```

Рис. 45. Конструктор.

```
public async Task<Stream> LoadCoverBitmapAsync()
{
    if (File.Exists(CachePath + ".bmp"))
    {
        return File.OpenRead(CachePath + ".bmp");
    }
    else
    {
        var data = await s_httpClient.GetByteArrayAsync(CoverUrl);

        return new MemoryStream(data);
    }
}
```

Рис. 46. Функция загрузки обложки из кэша.

```
ссылка: 1
public async Task SaveAsync()
{
    if (!Directory.Exists("./Cache"))
    {
        Directory.CreateDirectory("./Cache");
    }

    using (var fs = File.OpenWrite(CachePath))
    {
        await SaveToStreamAsync(this, fs);
    }
}
```

Рис. 47. Функция сохранения данных о альбоме.

```
private static async Task SaveToStreamAsync(Album data, Stream stream)
{
    await JsonSerializer.SerializeAsync(stream, data).ConfigureAwait(false);
}
```

Рис. 48. Функция сохранения данных об альбоме (запись в файл JSON).

```
public Stream SaveCoverBitmapStream()
{
    return File.OpenWrite(CachePath + ".bmp");
}
```

Рис. 49. Функция сохранения обложки.

```
public static async Task<IEnumerable<Album>> LoadCachedAsync()
{
    if (!Directory.Exists("./Cache"))
    {
        Directory.CreateDirectory("./Cache");
    }
    var results = new List<Album>();
    foreach (var file in Directory.EnumerateFiles("./Cache"))
    {
        if (!string.IsNullOrWhiteSpace(new DirectoryInfo(file).Extension)) continue;

        await using var fs = File.OpenRead(file);
        results.Add(await Album.LoadFromStream(fs).ConfigureAwait(false));
    }
    return results;
}
```

Рис. 50. Функция чтения данных о альбоме (чтение из файла).

```
public static async Task<Album> LoadFromStream(Stream stream)
{
    return (await JsonSerializer.DeserializeAsync<Album>(stream).ConfigureAwait(false))!;
}
```

Рис. 51. Функция чтения данных об альбоме (чтение из файла JSON).

```
public static async Task<IEnumerable<Album>> SearchAsync(string searchTerm)
{
    var query = await s_SearchManager.GetAlbumsAsync(searchTerm).ConfigureAwait(false);
    return query.Albums.Select(x =>
        new Album(x.ArtistName, x.CollectionName, x.ArtworkUrl100.Replace("100x100bb", "600x600bb")));
}
```

Рис. 52. Функция поиска информации об альбоме в iTunes.

Задача метода `LoadCoverBitmapAsync()` (рис. 46) получить поток байтов обложки. В случае если обложка найдена по указанному пути, то загружается она, иначе обложка загружается из интернета по ссылке указанной в `CoverUrl`.

Следующий метод `SaveAsync()` (рис. 47), выполняет сохранение описания альбома, в рамках которого происходят следующие действия. Во-первых, проверка наличия папки для так называемого «кэша», при этом, если папка отсутствует, она будет создана. Во-вторых, открывается поток для записи в файл, имя которого содержит название исполнителя и название альбома, и передается вместе со ссылкой на текущий объект методу `SaveTo-`

`StreamAsync()` (рис.48), который в свою очередь, сохраняет информацию об альбоме в файл формата JSON. JSON – это текстовый формат обмена структурированными данными, основанный на языке JavaScript. Альтернативой JSON для хранения данных также может выступать формат XML. Подробнее про сериализацию и десериализацию JSON и XML можно прочитать в официальной справке Microsoft. (таб. 25).

Таблица 25

Справка	QR-Code
Сериализация и десериализация JSON	
Сериализация и десериализация XML	

В свою очередь, у метода `LoadFromStream()` был вызван метод `ConfigureAwait()`, предназначенный для управления контекстом синхронизации. По умолчанию асинхронный метод возвращается в тот контекст, из которого был вызван на выполнение, при этом, приложение в UI-контексте обычно имеет один поток, т.е. в конкретное время может выполняться только одна задача. При смешивании асинхронного и синхронного кода может привести к DeadLock (взаимоблокировке). Например, если для ожидания результата функции используется следующая конструкция:

```
var a = MyMethodAsync().Result;
```

вместо

```
var a = await MyMethodAsync();
```

Вызов Result заблокирует контекст синхронизации, пока асинхронная операция не закончит свою работу. В свою очередь, когда операция дойдет

до await внутри `MyMethodAsync()` ей будет нужно поместиться в очередь контекста синхронизации, который вызовет оставшуюся часть. Но он не сможет этого сделать, так как единственный поток будет занят ожиданием результата всей функции.

Таким образом, при вызове любой библиотечной функции, во избежание DeadLock требуется использовать `ConfigureAwait(false)`, что перебросит окончание выполнения функции в контекст синхронизации по умолчанию (`ThreadPool`). Однако у такого решения есть ограничение, если вызываемая функция работает с пользовательским интерфейсом, то это может привести к ошибке доступа к данным принадлежащим другому потоку. Также, в качестве альтернативы можно использовать команду `Task.Run()`, что неявно выполнит функцию в контексте по умолчанию.

Следующая функция (рис. 49) выполняет сохранения массива байтов обложки в файл с расширением .bmp. При этом путь и имя у такого файла будет аналогично соответствующему файлу JSON. Использование данной функции будет реализовано позже.

Функция загрузки кэша (рис. 50) во-первых проверяет наличие папки кэша, в ином случае ее создает, во-вторых получает список файлов в каталоге, и выбирает только те файлы, у которых нет расширения. Далее происходит открытие файла для чтения, создающее поток байтов. Однако вместо обычного `using` используется асинхронный вариант `await using`, позволяющий освободить неуправляемые ресурсы асинхронно (таб. 26). И в конце происходит десериализация файла JSON в объект класса Album (рис. 51).

Таблица 26

Справка	QR-Code
Механизм асинхронного освобождения ресурсов	

Последний метод (рис. 52) осуществляет поиск альбомов, и создает массив результатов, который содержит только текстовую информацию.

Для того чтобы поиск автоматически выполнялся при вводе текста добавьте следующий код в конструктор `SubWindowViewModel` (рис. 53).

```
this.WhenAnyValue(x => x.SearchText)
    .Where(x => !string.IsNullOrWhiteSpace(x))
    .Throttle(TimeSpan.FromMilliseconds(400))
    .ObserveOn(RxApp.MainThreadScheduler)
    .Subscribe(DoSearch!);
```

Рис. 53. Подписка на изменение свойства SearchText.

Метод `WhenAnyValue()` (из `ReactiveUI`) выбирает событие изменения необходимого нам свойства (`SearchText`). Поскольку пользователь может печатать довольно быстро, не стоит запускать поиск после каждого нажатия клавиши. Для этого вводится ограничение (`Trottle`) в данном случае на 400 мс, т.е. значение будет обработано только тогда, когда пользователь не будет печатать 400 мс или дольше. Вызов `ObserveOn` требуется, чтобы убедится, что `DoSearch()` будет вызван в потоке UI (из-за применения `Trottle` вызов может произойти в случайном потоке, но интерфейс можно обновлять только в потоке UI).

Перед тем как создать сам метод поиска, нужно реализовать модель отображения для альбома, в которой содержится информация об исполнителе и названии. Добавьте в `AlbumViewModel` следующий код (рис. 54).

```
private readonly Album _album;
Ссылок: 0
public string Artist => _album.Artist;
Ссылок: 0
public string Title => _album.Title;
Ссылок: 2
public AlbumViewModel(Album album)
{
    _album = album;
}
```

Рис. 54. Создание модели отображения альбома.

В свою очередь, сами поля не будут поддерживать уведомление (функция `RaiseAndSetIfChanged()` отсутствует), так как данные альбома однажды полученные меняться не будут. Реализуем сам метод автоматического поиска альбомов (рис. 53) добавив в `SubWindowViewModel` следующий код (рис. 55).

```
private CancellationTokenSource? _cancellationTokenSource;
ссылка: 1
private async void DoSearch(string s)
{
    IsBusy = true;
    SearchResults.Clear();
    if (!string.IsNullOrWhiteSpace(s))
    {
        var albums = await Album.SearchAsync(s);

        foreach (var album in albums)
        {
            var vm = new AlbumViewModel(album);

            SearchResults.Add(vm);
        }
    }

    IsBusy = false;
}
```

Рис. 55. Поиск альбомов.

В данном методе свойство `IsBusy` устанавливается в значение `true` для того, чтобы во время поиска пользователь видел прогрессбар. Коллекция `SearchResults` предварительно очищается, чтоб в дальнейшем ее заполнить новыми результатами поиска. Далее вызывается метод `SearchAsync` ищущий альбомы в `iTunes`, которые после добавляются в коллекцию. В свою очередь, строка `CancellationTokenSource? _cancellationTokenSource;` пока никакой функциональности не несет и будет использована позже. Запустите приложение и проверьте функцию поиска (рис. 56).

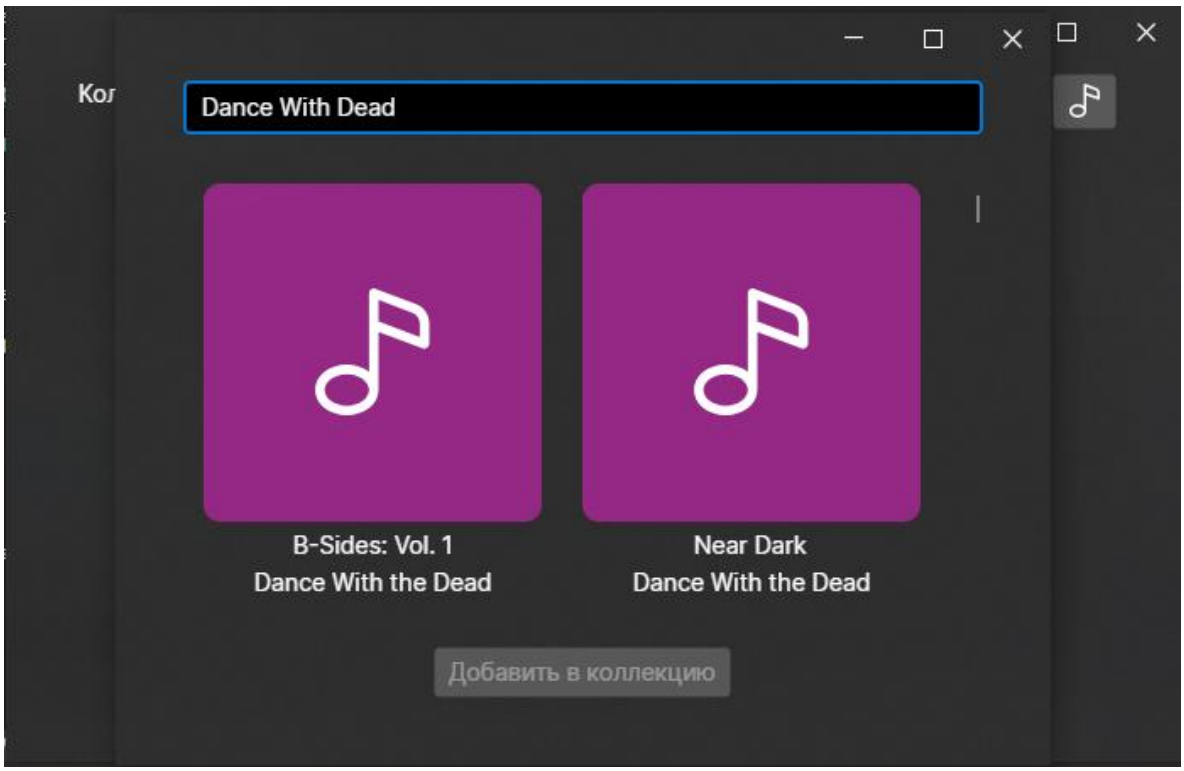


Рис. 56. Поиск альбомов.

Пока вместо обложек отображается созданная нами «заглушка», напишем функцию для загрузки настоящих обложек. Добавьте в `AlbumViewModel` следующий код (рис. 57).

```
private Bitmap? _cover;
Ссылок: 3
public Bitmap? Cover
{
    get => _cover;
    private set => this.RaiseAndSetIfChanged(ref _cover, value);
}
Ссылок: 2
public async Task LoadCover()
{
    await using (var imageStream = await _album.LoadCoverBitmapAsync())
    {
        Cover = await Task.Run(() => Bitmap.DecodeToWidth(imageStream, 400));
    }
}
```

Рис. 57. Загрузка обложки.

`Bitmap` – это класс, предоставляемый библиотекой `Avalonia`, объекты которого предназначены для хранения изображений. Метод `LoadCov-`

er загружает изображение со помощью ранее написанного метода и после уменьшает его размер в памяти методом DecodeToWidth() (так как размер обложек в интерфейсе был ограничен 400x400 пикселей, то нет смысла хранить их в большем размере). Bitmap.DecodeToWidth() не является асинхронным методом и применить к нему await не получится, поэтому создается задача методом Task.Run, внутри которой она может быть выполнена асинхронно.

Добавьте в SubWindowViewModel следующий код (рис. 58).

```
private async void LoadCovers(CancellationToken cancellationToken)
{
    foreach (var album in SearchResults.ToList())
    {
        await album.LoadCover();

        if (cancellationToken.IsCancellationRequested)
        {
            return;
        }
    }
}
```

Рис. 58. Загрузка обложек для результата поиска.

Задача вышеописанной функции это запуск асинхронной загрузки обложки для каждого найденного альбома с возможностью отмены операции. Метод ToList() предназначен для конвертации объекта ObservableCollection<T> в List<T> путем создании копии данных в памяти. Данный метод выгоден тем, что по время загрузки обложек коллекция SearchResults может поменяться (пользователь решил изменить поисковый запрос) и цикл foreach выдаст исключение, но так как перечисление будет проводиться на неизменной копии, исключение будет предотвращено. Далее в коде используется свойство (токен) cancellationToken ранее созданного объекта private CancellationTokenSource? _cancellationTokenSource (рис. 54). Данный объект предназначен для

отправки сигнала отмены асинхронным операциям (таб. 27). Стоит отметить, что в официальной справке явно сказано вызывать метод `Dispose()` когда токен более не понадобится, однако при его использовании момент вызова утилизации становится не предсказуем, поэтому его вызов опускают и перекладывают эту задачу на сборщик мусора.

Таблица 27

Справка	QR-Code
Сигнал отмены	

Добавим возможность загрузки обложек во время поиска, для этого добавьте следующий код в начало метода `DoSearch()` (`SubWindowViewModel`) выполняющий задачу отмены предыдущей команды поиска и создание нового токена (рис. 59). Также добавьте код после цикла `foreach`, отменяющую загрузку обложек при вызове отмены (рис. 60).

```
_cancellationTokenSource?.Cancel();
_cancellationTokenSource = new CancellationTokenSource();
var cancellationToken = _cancellationTokenSource.Token;
```

Рис. 59. Отмена предыдущей операции и создание нового токена.

```
if (!cancellationToken.IsCancellationRequested)
{
    LoadCovers(cancellationToken);
}
```

Рис. 60. Использование токена.

Запустите приложение и проверьте функцию поиска и загрузки обложек (рис. 61).

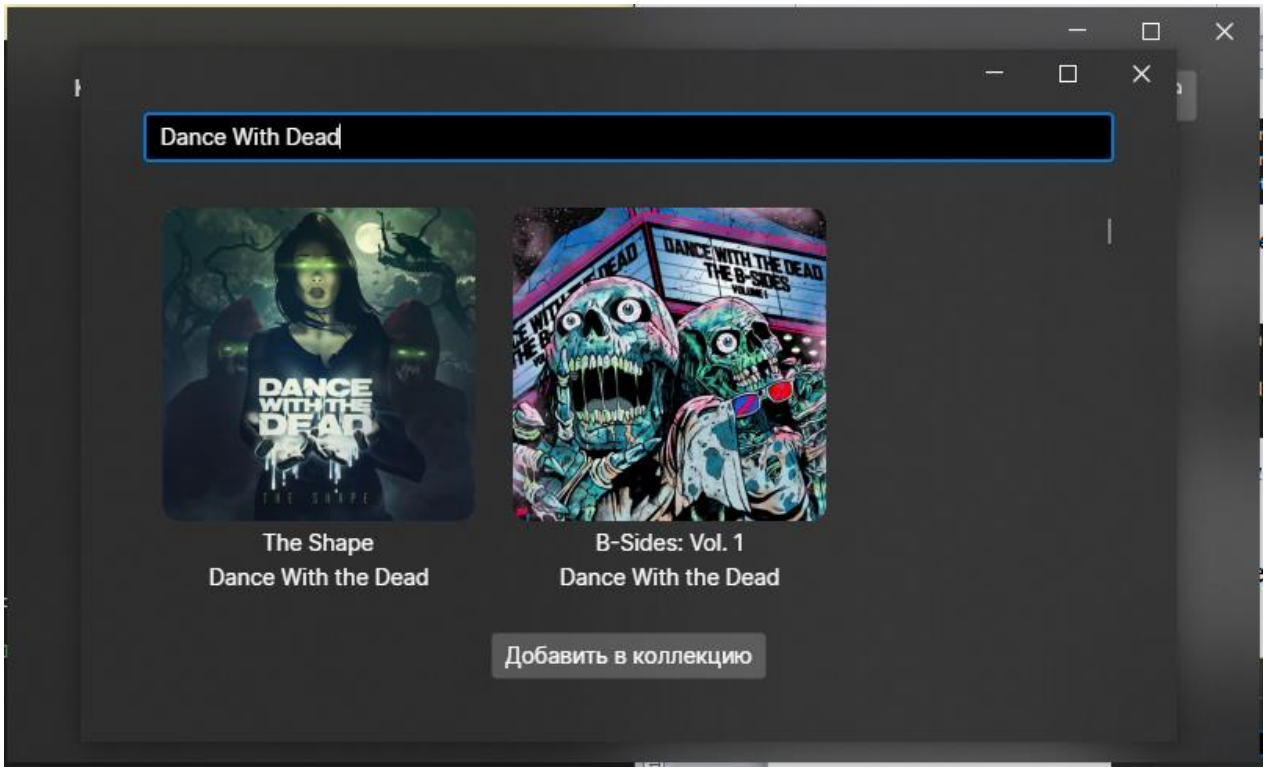


Рис. 61. Загрузка обложек.

Реализуем добавление в коллекцию выбранного нами альбома, для этого добавьте пространство имен `xmlns:local="clr-namespace:Demo_3.Views"` и следующий код в `MainWindow.axaml` после тега `<TextBlock>` (рис. 62).

```
<ScrollViewer Margin="0 40 0 0">
    <ItemsControl Items="{Binding Albums}">
        <ItemsControl.ItemsPanel>
            <ItemsPanelTemplate>
                <WrapPanel />
            </ItemsPanelTemplate>
        </ItemsControl.ItemsPanel>

        <ItemsControl.ItemTemplate>
            <DataTemplate>
                <local:AlbumView Margin="0 0 20 20"/>
            </DataTemplate>
        </ItemsControl.ItemTemplate>
    </ItemsControl>
</ScrollViewer>
```

Рис. 62. Отображение коллекции.

Первым новым тегом в данном коде будет `<ItemsControl>` предназначенный для отображения списка элементов без возможности их выбора. При использовании данного тега есть два недостатка: во-первых, данный тег не поддерживает скролл (если контент не помещается, то полоса прокрутки не появится), во-вторых, нет возможности указать расстояние между элементами внутри. Полосы прокрутки можно добавить к любому элементу, обернув его в `<ScrollViewer>`. В свою очередь, чтобы добавить пространство между элементами нужно изменить шаблон отображения элемента списка `ItemTemplate`. Однако в таком случае придется указать пространство имен, откуда был взят этот самый элемент, и обращаться к нему напрямую.

Далее добавьте свойства `CollectionEmpty` и `Albums` в `MainWindowViewModel` (рис. 63).

```
private bool _collectionEmpty;
ссылка: 1
public bool CollectionEmpty
{
    get => _collectionEmpty;
    set => this.RaiseAndSetIfChanged(ref _collectionEmpty, value);
}
Ссылок: 4
public ObservableCollection<AlbumViewModel> Albums { get; } = new();
```

Рис. 63. Свойства CollectionEmpty и Albums.

В конец конструктора добавьте подписку на событие изменения количества элементов в коллекции `Albums`, которая будет изменять свойство `CollectionEmpty` (рис. 64).

```
this.WhenAnyValue(x => x.Albums.Count).Subscribe(x => CollectionEmpty = x == 0);
```

Рис. 64. Подписка на изменение коллекции.

Кроме того, измените команду `SearchMusicCommand` следующим образом (рис. 62). Данное изменение дает возможность добавить в коллекцию выбранный альбом.

```
SearchMusicCommand = ReactiveCommand.CreateFromTask(async () =>
{
    var store = new SubWindowViewModel();
    var result = await ShowDialog.Handle(store);
    if (result != null)
    {
        Albums.Add(result);
    }
});
```

Рис. 62. Добавление в коллекцию выбранного альбома.

Откомпилируйте приложение, попробуйте добавить несколько альбомов.

Реализуем функцию сохранения альбомов в кэш на локальном диске и их последующую загрузку при запуске программы. Для этого добавьте в `AlbumViewModel` функцию сохранения (рис. 65).

```
public async Task SaveToDiskAsync()
{
    await _album.SaveAsync();

    if (Cover != null)
    {
        var bitmap = Cover;

        await Task.Run(() =>
        {
            using (var fs = _album.SaveCoverBitmapStream())
            {
                bitmap.Save(fs);
            }
        });
    }
}
```

Рис. 65. Запись данных об альбоме на локальный диск.

Измените команду `SearchMusicCommand`, добавив в условие `if` вызов новой функции (рис. 66).

```
if (result != null)
{
    Albums.Add(result);
    await result.SaveToDiskAsync();
}
```

Рис. 66. Вызов записи данных.

Добавьте функцию чтения обложек с диска в память (рис. 67).

```
ссылка: 1
private async void LoadAlbums()
{
    var albums = await Album.LoadCachedAsync();

    foreach (var album in albums)
    {
        Albums.Add(new AlbumViewModel(album));
    }

    foreach (var album in Albums)
    {
        await album.LoadCover();
    }
}
```

Рис. 67. Чтение данных с диска.

Зарегистрируем задачу на асинхронное выполнение в главном потоке (так как данная задача работает с интерфейсом), добавив строчку в конце конструктора (рис. 68).

```
RxApp.MainThreadScheduler.Schedule(LoadAlbums);
```

Рис. 68. Регистрация задачи в планировщике главного потока.

Запустите приложение, попробуйте добавить несколько альбомов в коллекцию. При повторном запуске выбранные вами ранее альбомы должны автоматически загрузиться. Пример главного окна с автоматической загрузкой представлен на рисунке 69.

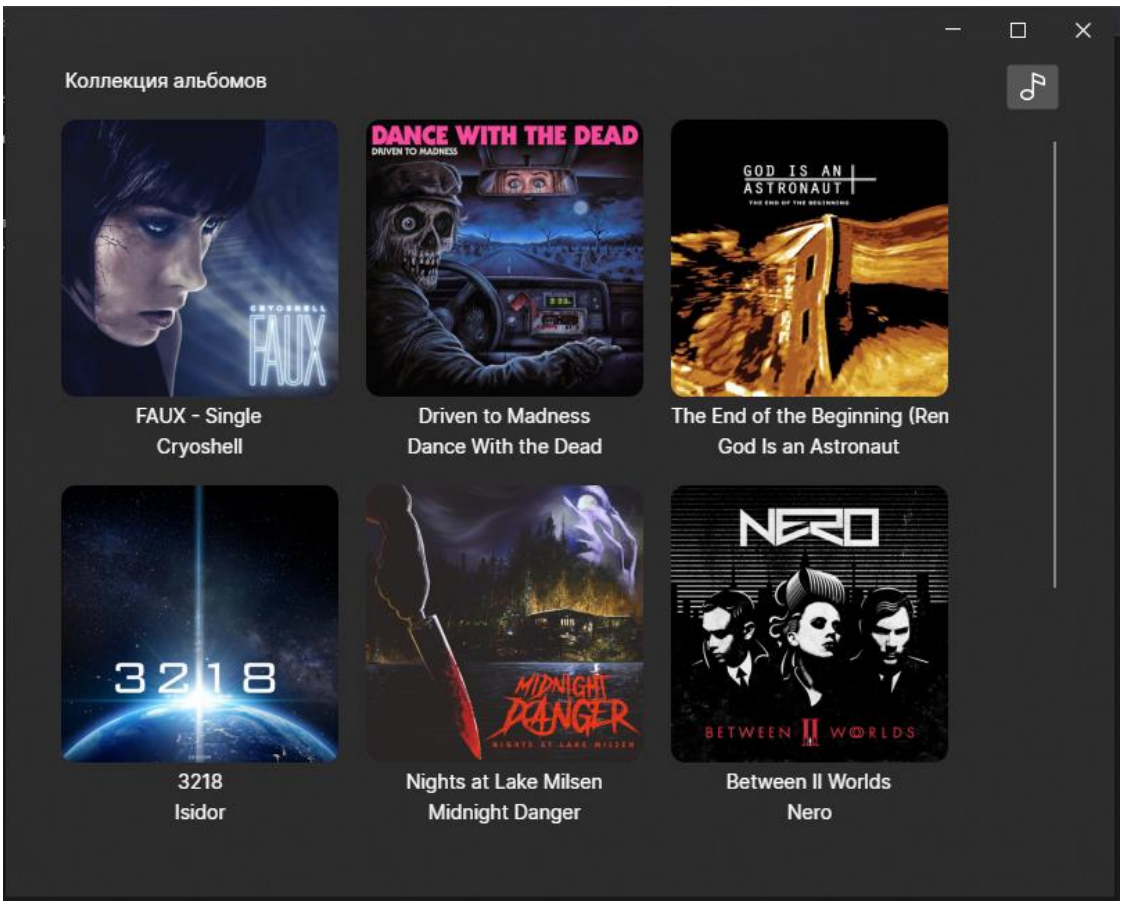


Рис. 69. Итоговое главное окно приложения.

Контрольные вопросы

1. *Прочитайте справку про синтаксис разметки пути, какие есть возможности у «мини-языка»?*
2. *Прочитайте справку про асинхронное программирование. Для чего оно применяется? Каким образом можно поставить задачу на выполнение?*
3. *Прочитайте справку про отмену задачи с помощью CancellationToken. Каким образом можно отметить задачу?*
4. *Прочитайте справку про конверторы. Для чего они применяются?*

Лабораторная работа №4

Работа со стилями в фреймворке Avalonia.

Источники

Данная лабораторная работа основана на проекте калькулятора и темы оформления Citrus из ресурса GitHub (таб. 28).

Таблица 28

GitHub	QR-Code
Приложение ACalc	
Тема оформления Citrus	

Создание отображения главного окна

Разработка интерфейса приложения достаточно трудоемки процесс, включающий различные этапы от исследования предметной области до построения макета, сопровождающиеся дискуссиями и мозговым штурмом, поэтому в данной лабораторной работе ограничимся готовым макетом (рис. 70).

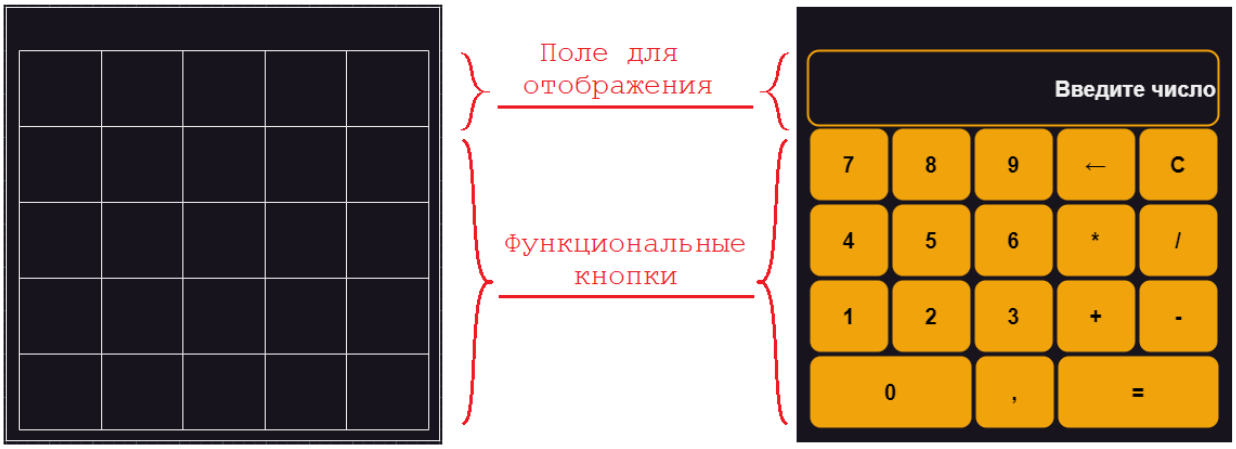


Рис. 70. Макет главного окна.

Создайте проект приложения «Avalonia MVVM Application» по аналогии с лабораторной работой №1 (в качестве названия проекта укажите «Demo_4»).

Откройте файл MainWindow.axaml и создайте разметку будущего интерфейса калькулятора. Для этого установите ширину и высоту окна в 400 px, и замените <TextBlock> на <Panel> с отступами "5 40 5 5". Внутрь панели поместите таблицу 5x5 с пропорциональным размером колонок и столбцов, кроме того добавьте свойство ShowGridLines="True". Данное свойство сделает границы ячеек видимыми, что будет полезно при дальнейшей разработке макета. Запустите приложение, главное окно программы приобретет следующий вид (рис. 71).

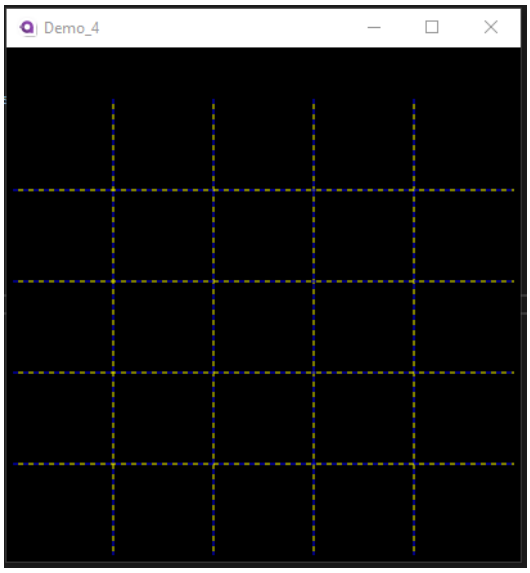


Рис. 71. Главное окно программы.

В качестве поля для ввода воспользуемся элементом <TextBox>, который будет отображать вводимые числа и результат операций. Поместим его в первую ячейку таблицы с помощью присоединяемых свойств Grid.Column="0" и Grid.Row="0", а также растянем на всю таблицу с помощью свойства Grid.ColumnSpan="5". Далее установим данному объекту фокус ввода параметром TabIndex="0" (рис. 72).

```
<TextBox Watermark="Введите число"
         TabIndex="0"
         Grid.Column="0"
         Grid.Row="0"
         Grid.ColumnSpan="5"
         IsReadOnly="True"/>
```

Рис. 72. Текстовое поле для отображения чисел.

TabIndex – это свойство, предназначенное для переключения фокуса ввода между элементами интерфейса с помощью клавиши Tab, при этом для указания порядка нужно присвоить `TabIndex`-у каждого элемента, который будет получать фокус, значение от 0 (получает фокус первым) до `Int.maxValue` (`2147483647`). Если же значения не указывать то, по умолчанию фокус ввода передается в порядке объявления элементов.

Сам ввод, при этом, будет реализован с помощью кнопок и горячих клавиш, поэтому заблокируем возможность печатать в поле свойством `IsReadOnly="true"`, что дополнительно защитит приложение от некорректного ввода пользователем.

Теперь нужно создать кнопки, выполняющие стандартные функции калькулятора. В качестве примера возьмем клавишу «0». Данная клавиша согласно макету будет располагаться в левом нижнем углу (`Grid.Column="0"`, `Grid.Row="4"`) и занимать две ячейки (`Grid.ColumnSpan="2"`). Укажем кнопке индекс табуляции 1, (`TabIndex="1"`), т.е. при нажатии клавиши Tab фокус перейдет с ранее описанного текстового поля на эту кнопку. Так же установим кнопке горячую клавишу 0 на цифровом блоке клавиатуры (`HotKey="Numpad0"`) (рис. 73).

```
<Button Content="0"
        TabIndex="1"
        Grid.Column="0"
        Grid.Row="4"
        HotKey="Numpad0"
        Grid.ColumnSpan="2"/>
```

Рис. 73. Кнопка 0.

Список горячих клавиш, а так же модификаторов можно получить из справки Avalonia (таб. 29).

Таблица 29

Справка	QR-Code
HotKeys (ENG)	

Откомпилируйте и запустите приложение, в результате получится следующее окно (рис. 74).

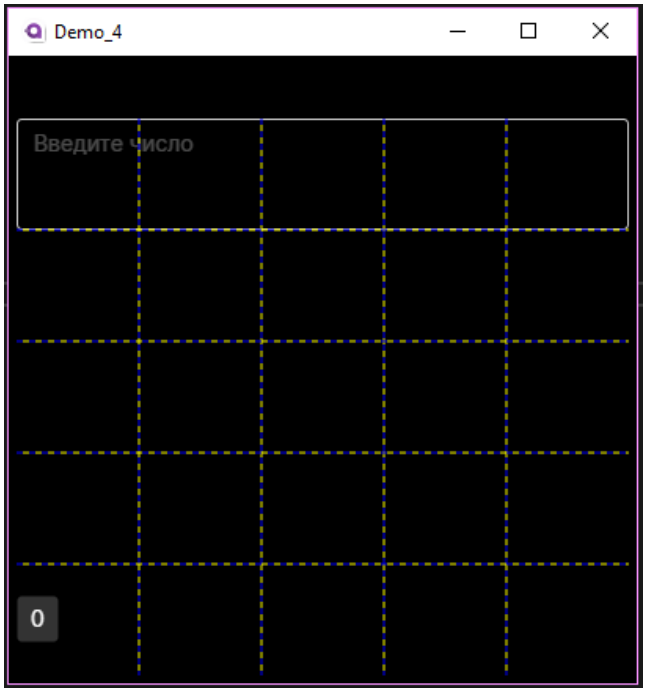
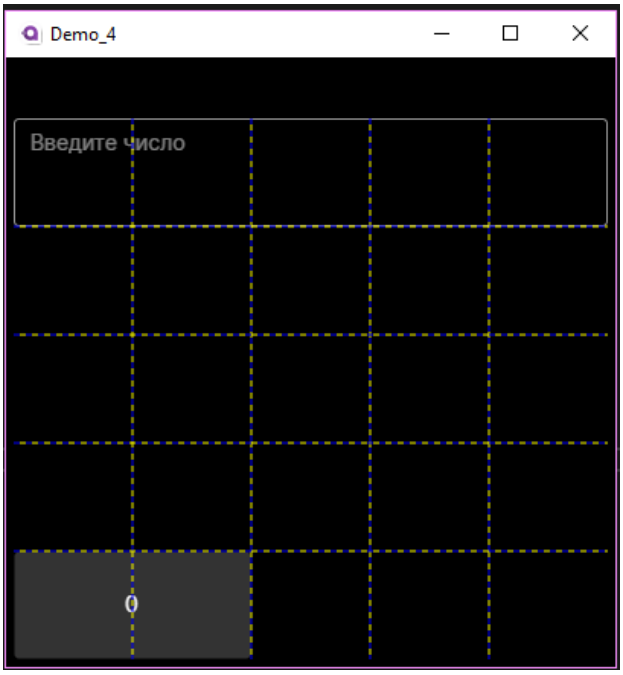


Рис. 74. Главное окно программы.

Текстовое поле автоматически растягивается и заполняет доступное ему пространство, однако кнопка по умолчанию имеет размер по контенту внутри, что можно исправить, указав для этого в свойствах `HorizontalAlignment="Stretch"` и `VerticalAlignment="Stretch"`. В свою очередь, для того чтобы цифра внутри кнопки располагалась по центру, добавим свойства

VerticalContentAlignment= "Center" и HorizontalContentAlignment="Center" (рис. 75).



Первый Рис. 75. Главное окно программы.

Стоит отметить, что такой способ оформления подойдет только в том случае если элементов немного или они уникальны, в нашем же случае, потребуется создать 18 однотипных кнопок, и у каждой кнопки указывать данные свойства будет довольно накладно, поэтому для этого можно использовать стили.

Применение стилей

Стиль – это объект, который может настраивать внешний вид одного, группы или всех элементов управления приложения. При этом в Avalonia стили основаны на сочетании таковых из WPF/UWP и CSS.Стили могут располагаться в любом элементе внутри тега <элемент.Styles></элемент.Styles> и при этом влиять как на сам элемент, так и на все объекты во внутренней иерархии.

Общий шаблон стиля в Avalonia выглядит следующим образом:

```
<Style Selector="селектор">
```

```
        <Setter
             Property="Свойство"
             Value="Значение">
</Style>
```

В `Property` указывается имя свойства элемента, которому нужно установить соответствующее значение `Value`, например:

```
<Setter
     Property="Background"
     Value="White">
```

Селектор – это строка указывающая критерий выбора элементов управления. В Avalonia существует множество правил использования селекторов (таб. 30), однако в нашем случае ограничимся следующими конструкциями:

```
Selector="класс"
```

или

```
Selector="класс:псевдокласс /template/ внутр_класс#имя"
```

Таблица 30

Справка	QR-Code
Styles (ENG)	

Первый вариант селектора предназначен для изменения свойств элементов в нормальном состоянии, в свою очередь второй вариант предназначен для изменения свойств у определенных состояний элемента.

Класс – это название целевого класса, для объектов которого создается стиль, например, `Button`.

Псевдокласс – это параметр, заимствованный из CSS, указывающий для какого состояния элемента будет применен стиль, например, `:pointover` (над элементом расположен курсор мыши). В свою очередь в WPF/UWP аналогом является триггер.

`/template/` - это параметр, устанавливающий, что будет изменен шаблон отображения элемента.

Внутр_класс – это класс объектов, находящихся внутри иерархии объекта целевого класса, свойства которого будут изменены. Например, для кнопки это `ContentPresenter`.

Имя – это название, указанное в свойстве `Name` элемента. Данный параметр указывается только в случае, когда нужно обратиться к конкретному объекту.

Инструменты разработки Avalonia

В свою очередь, для выбора соответствующего элемента для изменения свойств, следует обратиться к инструментам разработки Avalonia, по функциям напоминающего панель разработчика в WEB-браузере. Данная панель вызывается только после запуска Debug-версии приложения с помощью клавиши F12 (рис. 74).

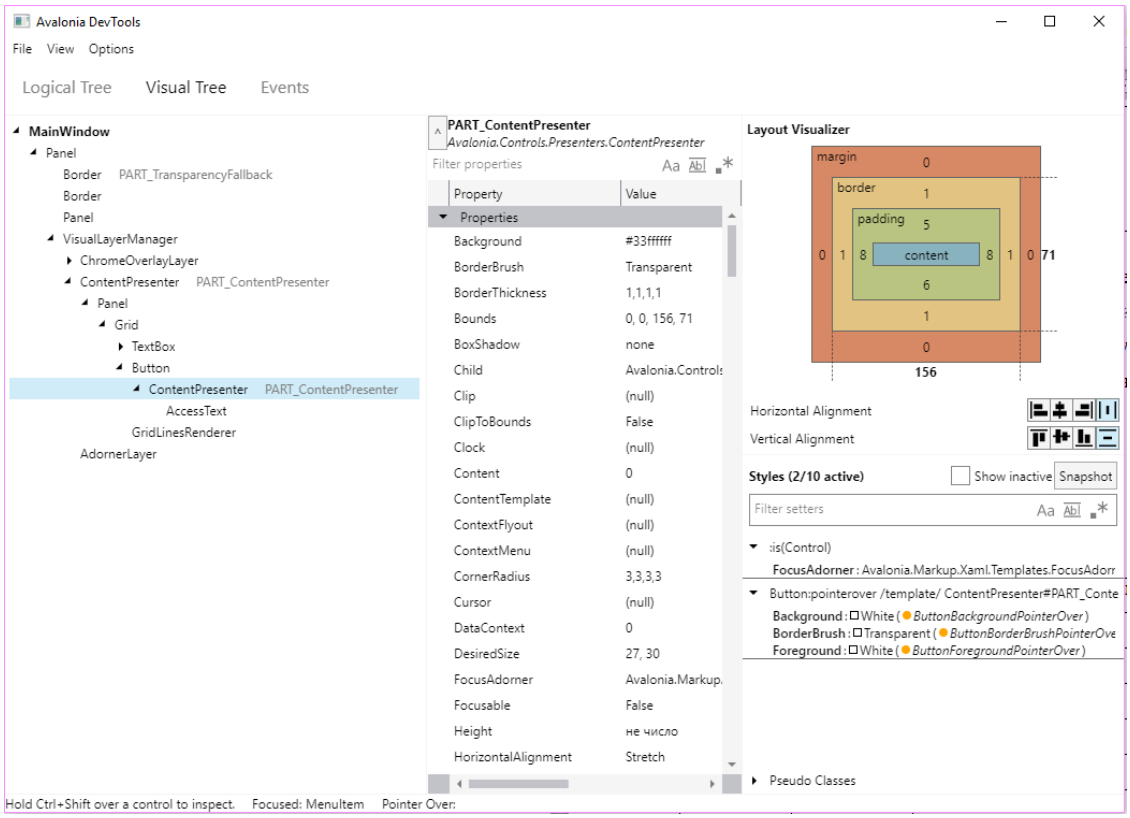


Рис. 74. Средства диагностики Avalonia.

В панели диагностики присутствуют три вкладки: «Logical Tree», «Visual Three» и «Events».

Вкладка «Logical Tree» отображает представление дерева (иерархии) элементов управления. Вкладка «Visual Three» показывает представление логического дерева, а также объекты, из которых состоят сами элементы. При этом в обеих вкладках при выборе того или иного элемента можно увидеть его свойства, представленные в виде таблицы, и отредактировать их. В свою очередь, изменения свойств будут моментально отображаться в форме, но не будут сохранены в исходной программе. Кроме того вкладки также включают блок визуализации макета и блок стилей.

Блок визуализации макета предназначен для отображения текущих свойств разметки выбранного элемента и включает следующие параметры:

1) Ограничения по размеру элемента (если ограничения подчеркнуты, они имеют фиксированное значение);
2) `Margin` – отступ от рамки объекта до родительского контейнера;
3) `Border` – толщина рамки;
4) `Padding` – отступ от объекта до его рамки;
5) Размер самого элемента.

В свою очередь панель стилей показывает список стилей, которые влияют на свойства выбранного элемента в определенном состоянии. Каждый стиль представлен заголовком, отображающим его селектор, и списком свойств. При этом если свойство привязано к ресурсу, то рядом с его значением будет отображаться желтая точка, за которой следует имя ресурса. Стоит отметить, что некоторые свойства могут отображаться зачеркнутыми, это означает, что свойство перекрыто (переопределено) стилем с более высоким приоритетом.

Последняя вкладка «Events» отображает маршрутизируемые события сгенерированные элементами интерфейса, а также цепочку следования этого события. События могут быть трех видов: bubble event, tunnel event и direct event.

Bubble event (пузырьковое событие) – это событие которое не обрабатывается самими элементом. При этом событие начинает переходить вверх по иерархии элементов, до тех пор, пока его не обработают или оно не встретит корневой элемент. Например, если кнопка не обработала событие нажатия, то его может обработать панель ее содержащая.

В свою очередь, Tunnel event возникает перед пузырьковым событием, и является его противоположностью, т.е. движется от внешнего элемента к внутреннему элементу. Туннельное событие обычно используют для проверки ввода пользователя, отменяя (перехватывая) при этом пузырьковое событие.

Direct event – это событие, которое возникает и обрабатывается только одним элементом (по аналогии с событиями из WinForms). Более подробно маршрутизируемые события представлены в официальной справке Avalonia (таб. 31).

Таблица 31

Справка	QR-Code
Routed Events (ENG)	

В свою очередь, для описания селектора нужно открыть вкладку «Visual Tree», зажать сочетание клавиш «Ctrl+Shift» и навести курсор на необходимый компонент на форме, при этом в визуальном дереве выделится соответствующий объект (рис. 75).

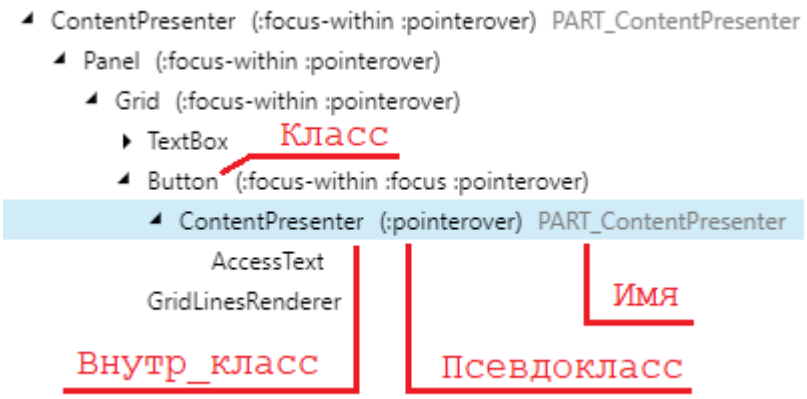


Рис. 75. Средства диагностики Avalonia.

Создадим стиль для обычного состояния кнопок всего приложения, для этого добавьте следующий код в файл `App.axaml` (рис. 76).

```
<Application.Styles>
    <FluentTheme Mode="Dark"></FluentTheme>
    <Style Selector="Button">
        <Setter Property="HorizontalAlignment" Value="Stretch"></Setter>
        <Setter Property="VerticalAlignment" Value="Stretch"></Setter>
        <Setter Property="VerticalContentAlignment" Value="Center"></Setter>
        <Setter Property="HorizontalContentAlignment" Value="Center"></Setter>
    </Style>
</Application.Styles>
```

Рис. 76. Стиль для кнопок.

В свою очередь, указание данных параметров в кнопке более не требуется, поэтому их можно удалить. Создайте по аналогии с «нулем» остальные кнопки согласно макету. В качестве параметра горячих клавиш укажите следующие значения (таб. 32).

Таблица 32

Клавиша	Key	Клавиатура
0	Numpad0	Цифровой блок 0
1	Numpad1	Цифровой блок 1
2	Numpad2	Цифровой блок 2
3	Numpad3	Цифровой блок 3
4	Numpad4	Цифровой блок 4
5	Numpad5	Цифровой блок 5

Клавиша	Key	Клавиатура
6	Numpad6	Цифровой блок 6
7	Numpad7	Цифровой блок 7
8	Numpad8	Цифровой блок 8
9	Numpad9	Цифровой блок 9
,	Decimal	Цифровой блок запятая
+	Add	Цифровой блок +
-	Subtract	Цифровой блок -
*	Multiply	Цифровой блок *
/	Divide	Цифровой блок /
=	Enter	Клавиша Enter
←	Back	Клавиша Backspace
C	Delete	Клавиша Delete

Уберите параметр показывающий сетку таблицы, и запустите приложение. В результате главное окно примет следующий вид (рис. 77).

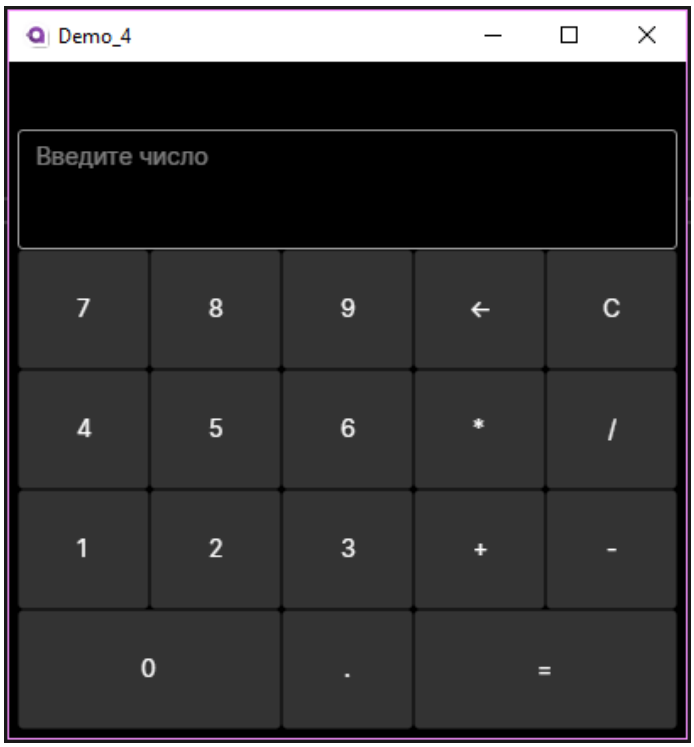


Рис. 77. Главное окно программы.

Создание файла стилей

Располагать стили можно не только внутри объектов, но и внутри отдельных `axaml` файлов которые будут включены в проект в виде ресурсов или в виде отдельных динамических библиотек (`dll`). Создайте каталог который будет хранить стили, для этого нажмите ПКМ на проекте (корневом каталоге решения) → `Добавить → Создать папку` и введите имя `Themes`. Далее следует подключить данный каталог как содержащий ресурсы Avalonia, для этого откройте файл `Demo_4.csproj` (см. лабораторную работу 1) и добавьте строчку `<AvaloniaResource Include="Themes\**" />` рядом со строкой, подключающей каталог `Assets` (рис. 78).

```
<ItemGroup>
  <AvaloniaResource Include="Assets\**" />
  <AvaloniaResource Include="Themes\**" />
  <None Remove=".gitignore" />
</ItemGroup>
<ItemGroup>
```

Рис. 78. Каталог проекта.

Внутри данного каталога создайте стиль Avalonia следующим образом: нажмите ПКМ на каталоге `Themes` → `Добавить → Создать элемент ... → Avalonia → Styles (Avalonia)` → введите имя `Theme.axaml`. Иерархия каталогов должна выглядеть следующим образом (рис. 79).

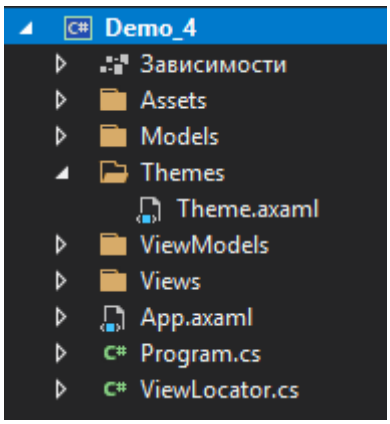


Рис. 79. Каталог проекта.

Откройте только что созданный файл стиля (рис. 80). Приведенный в файле объект `<Styles></Styles>` выполняет ту же функцию что и `<элемент.Styles></элемент.Styles>` т.е. хранит внутри себя стили.

```
<Styles xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <Design.PreviewWith>
        <Border Padding="20">
            <!-- Add Controls for Previewer Here -->
        </Border>
    </Design.PreviewWith>

    <!-- Add Styles Here -->
</Styles>
```

Рис. 80. Файл Theme.axaml.

Стоит отметить, что в Avalonia встроены две готовые темы оформления, от которых могут наследоваться стили: устаревшая Default и современная Fluent (рис. 81).

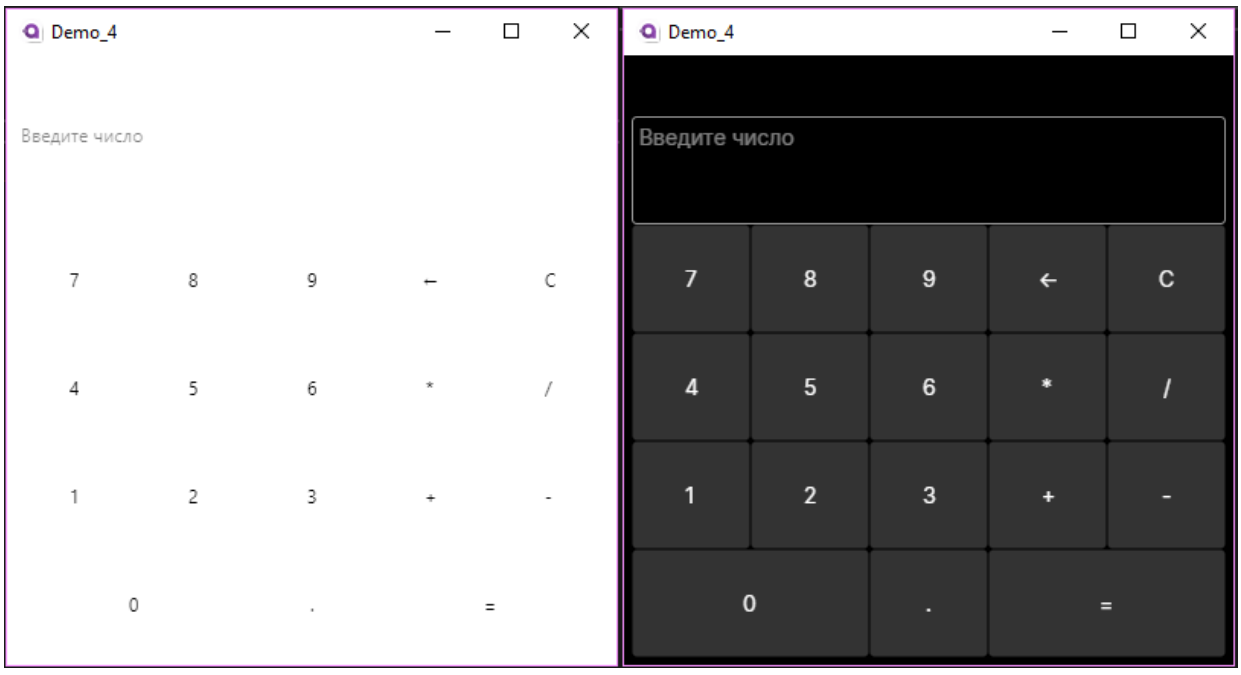


Рис. 81. Встроенные темы оформления: Default – слева; Fluent - справа.

Для подключения темы в раздел `<элемент.Styles></элемент.Styles>` необходимо в начало поместить одну из следующих строк:

для темы Fluent:

```
<FluentTheme Mode="режим"></FluentTheme>
```

для темы Default:

```
<StyleInclude
Source="avares://Avalonia.Themes.Default/DefaultTheme.xaml"/>
```

Добавьте в созданный файл стилей после `</Design.PreviewWith>` тему Fluent, при этом режим этой темы можно указать любой.

Во второй лабораторной работе были представлены два свойства, которые были доступны только во время разработки – это `d:DesignWidth` и `d:DesignHeight`. Альтернативным способом записи данных свойств являются свойства `Width` и `Height` объекта `Design`. Кроме того, объект `<Design>` предоставляет еще несколько свойств, среди которых есть `DataContext` и `PreviewWith`.

`DataContext` – это свойство, позволяющее использовать модель отображения прямо в окне предпросмотра, при этом не запуская отладку. Подробнее о данной возможность можно узнать из официальной справки (таб. 33).

В свою очередь свойство `PreviewWith` предоставляет возможность увидеть, как влияет стиль на указанные элементы в окне предпросмотра.

Таблица 33

Справка	QR-Code
IDE support (ENG)	

В нашей программе используются два вида элементов интерфейсов – кнопки и текстовое поле, поэтому добавим их в `PreviewWith`, чтобы настроить стили (рис. 82).

Рис. 82. Предпросмотр стилей.

Перенесите стиль, созданный для кнопки из `App.axaml` в созданный файл после тега темы Fluent, при этом в окне предпросмотра кнопка должна растянуться.

Ранее при написании интерфейса (см. лабораторная работа 3) присутствовала задача множественного использования объекта, решенная применением общего ресурса. При стилизации можно также применять ресурсы, сделав цветовую схему, шрифты, размеры и т.д. общими для всех элементов интерфейса. Ресурсы размещаются аналогично стилям, т.е. внутри любого объекта в теге `<объект.Resource></объект.Resource>` при этом у ресурса нужно указать параметр `x:Key` по которому можно в дальнейшем к нему обратится.

Кроме того ресурсы могут находиться и в своем специальном файле, называемом словарем ресурсов (ResourceDictionary). Для создания словаря нажмите ПКМ на каталоге `Themes` → `Добавить` → `Создать элемент ...` → `Avalonia` → `Resource Dictionary (Avalonia)` → введите имя `Scheme.axaml`. Подключите словарь к файлу стилей, добавив следующий код в `Theme.axaml` (рис. 83).

```
<Style>
    <Style.Resources>
        <ResourceDictionary >
            <ResourceDictionary.MergedDictionaries>
                <ResourceInclude Source="avares://Demo_4/Themes/Scheme.axaml"/>
            </ResourceDictionary.MergedDictionaries>
        </ResourceDictionary>
    </Style.Resources>
</Style>
```

Рис. 83. Подключение словаря стилей.

Указание пути к файлам в Avalonia осуществляется по следующему правилу: если ресурс включен в сборку приложения, т.е. папка, содержащая или сам объект добавлен как ресурс, то путь к ним указывается относительно корневого каталога проекта, в ином случае, если ресурс из другой сборки (`dll`) используется `avares:` URI-схема. Более подробно про указание пути к ресурсу можно прочитать в официальной справке Avalonia (таб. 34).

Таблица 34

Справка	QR-Code
Assets (ENG)	

Добавьте ресурсы в библиотеку, для этого откройте файл `Scheme.axaml` и приведите его к следующему виду (рис. 84).

```
<ResourceDictionary xmlns="https://github.com/avaloniaui"
                    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
                    xmlns:s="clr-namespace:System;assembly=mscorlib">
    <FontFamily x:Key="Font">Arial Rounded MT Bold</FontFamily>
    <s:Double x:Key="FontSize">30</s:Double>
    <Color x:Key="BaseColor">#ffa51f</Color>
    <Color x:Key="LightColor">#ffd89e</Color>
    <Color x:Key="DarkColor">#f90</Color>
    <Color x:Key="FontColor1">Black</Color>
    <Color x:Key="FontColor2">White</Color>
</ResourceDictionary>
```

Рис. 84. Добавление ресурсов.

Сделаем кнопку более похожей на макет, для этого измените перенесенный стиль кнопки следующим образом (рис. 85).

Рис. 85. Стиль кнопки в обычном состоянии.

Однако если навести курсор и нажать на кнопку в окне предспросмотра она будет иметь серый цвет, исправьте это, добавив еще два стиля, влияющих на кнопку когда на нее наведен курсор (`:pointover`) и когда она находится в нажатом состоянии (`:pressed`) (рис. 86).

```
<Style Selector="Button:pointerover /template/ ContentPresenter">
    <Setter Property="Background" Value="{DynamicResource LightColor}"></Setter>
</Style>
<Style Selector="Button:pressed /template/ ContentPresenter">
    <Setter Property="Background" Value="{DynamicResource DarkColor}"></Setter>
</Style>
<Style>
```

Рис. 86. Стили для кнопки в различных состояниях.

Следующим элементом для стилизации выступит TextBox, для этого добавьте следующий стиль (рис. 87).

Рис. 87. Стили для текстового поля в обычном состоянии.

Для изменения состояния текстового поля воспользуемся инструментами разработчика Avalonia, и заметим, что за рамку отвечает объект `Border` с именем `PART_BorderElement` во внутренней иерархии. При этом псевдоклассы отвечающие за состояние имеют значения :pointover (для со-

стояния, когда над объектом находится курсор мыши) и :focus (для состояния, когда в поле поставлен текстовый курсор).

```
<Style Selector="TextBox:pointerover /template/ Border#PART_BorderElement">
    <Setter Property="BorderBrush" Value="{DynamicResource LightColor}"></Setter>
</Style>
<Style Selector="TextBox:focus /template/ Border#PART_BorderElement">
    <Setter Property="BorderBrush" Value="{DynamicResource DarkColor}"></Setter>
</Style>
```

Рис. 88. Стили для текстового поля в различных состояниях.

Последним шагом будет подключение данного стиля к приложению, для этого замените в файле App.axaml строчку с темой Fluent на следующую:

```
<StyleInclude
Source="avares://Demo_4/Themes/Theme.axaml"></StyleInclude>
```

После запуска приложение примет следующий вид (рис. 89).

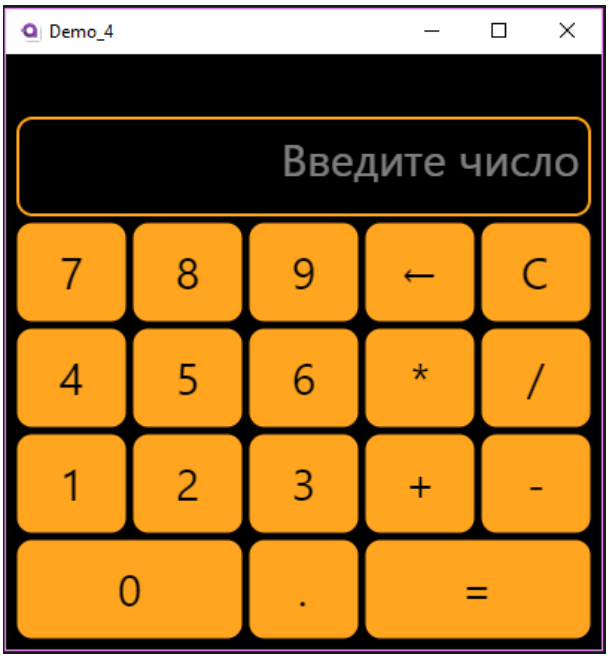


Рис. 88. Главное окно программы после применения стилей.

Реализация моделей и модели отображения

Во всем перечне кнопок разрабатываемого калькулятора можно выделить два блока схожих по функционалу клавиш – цифровой блок (клавиши 0-9) и операторный блок (+, -, *, /). При реализации модели и модели отобра-

жения команды, выполняемые при нажатии клавиш данных блоков, можно объединить, указав значение в xaml-коде которое будет передано команде.

В качестве возвращаемого значения для цифровых клавиш установим соответствующие им значения типа `Int`, для этого откройте файл MainWindow.axaml и добавьте в пространства имен формы строку, `xmlns:s="clr-namespace:System;assembly=mscorlib"`. Для клавиш математических операций создадим пользовательский тип данных содержащий константы. Для этого создайте новый класс в каталоге `Models` и назовите его `MOperator`. Замените ключевое слово `class` на `enum` и добавьте константы для каждой математической операции (рис. 89).

```
namespace Demo_4.Models
{
    Ссылок: 10
    public enum MOperator
    {
        Add,
        Subtract,
        Multiply,
        Divide
    }
}
```

Рис. 89. Тип данных MOperator.

Откройте файл MainWindow.axaml и добавьте в пространства имен формы строку, делающую доступными классы из каталога `Models` `xmlns:m="clr-namespace:Demo_4.Models;assembly=Demo_4"`.

Перед тем как вводить привязки, добавьте в `MainWindowViewModel.cs` две команды `addNumber` и `Command` (рис. 90). В приведенных командах вместо `Unit` в `<TInput, …>` был указаны Int и MOperator в качестве типа входного значения. В соответствии с этим метод `Create()` был заменен на метод `Create<TInput>()` поддерживающий передачу значения функции.

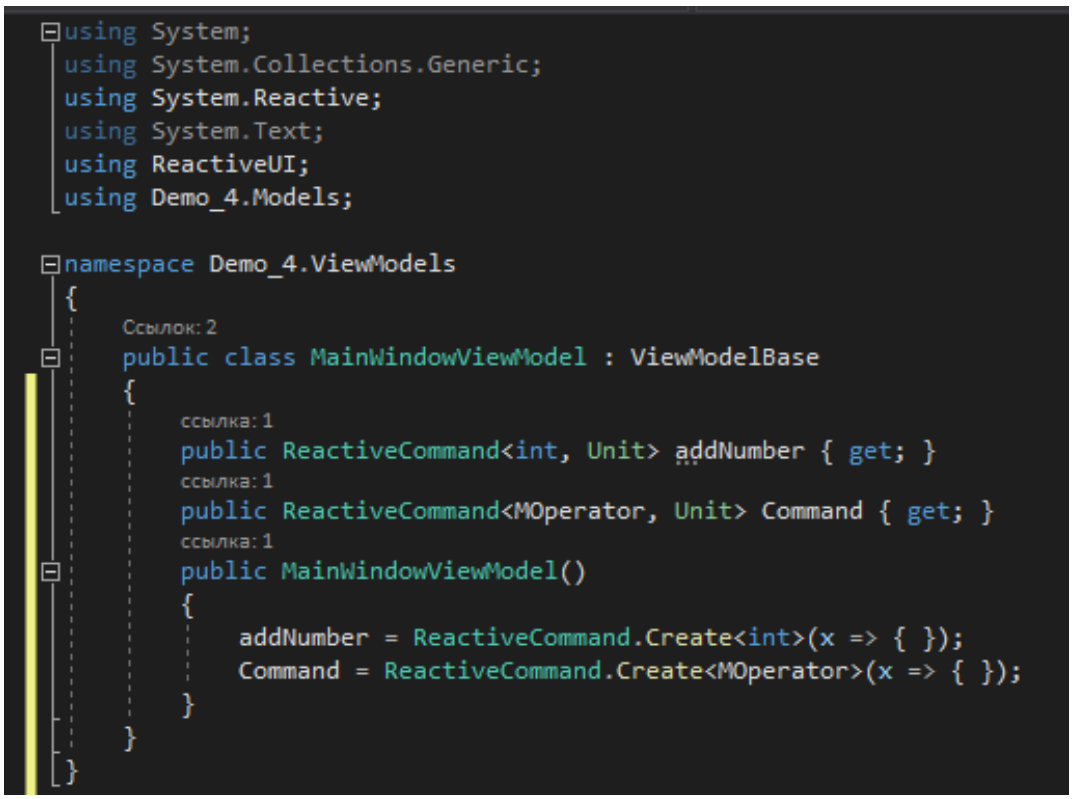


Рис. 90. Модель отображения главного окна.

Для передачи строкового значения или привязки к значению используется свойство: `CommandParameter="значение"`, при этом для остальных типов необходимо в тег `<Button>` поместить тег `</Button.CommandParameter>` внутри которого разместить передаваемое значение. Более подробно данный способ представлен в справке Avalonia (таб. 35). Измените цифровые и командные клавиши так, чтобы при их нажатии передавался соответствующий параметр. В качестве образца ориентируйтесь на реализацию клавиш 0 и + представленных на рисунках 91-92.

Таблица 35

Справка	QR-Code
Binding to commands (ENG)	

```
<Button Content="0"
        TabIndex="1"
        Grid.Column="0"
        Grid.Row="4"
        HotKey="Numpad0"
        Grid.ColumnSpan="2"
        Command="{Binding addNumber}">
    <Button.CommandParameter>
        <s:Int32>0</s:Int32>
    </Button.CommandParameter>
</Button>
```

Рис. 91. Клавиша 0.

```
<Button Content="+"
        TabIndex="13"
        Grid.Column="3"
        Grid.Row="3"
        HotKey="Add"
        Command="{Binding Command}">
    <Button.CommandParameter>
        <m:MOperator>Add</m:MOperator>
    </Button.CommandParameter>
</Button>
```

Рис. 92. Клавиша +.

Для остальных уникальных клавиш добавьте в модель отображения главного окна, следующие команды (рис. 93) и проинициализируйте их в конструкторе (рис. 94). Введите соответствующие привязки в XAML: клавиша «←» – Back, клавиша «C» – Del, клавиша «.» – Dot и клавиша «=» - Equal.

```
public ReactiveCommand<Unit, Unit> Del { get; }
ссылка: 1
public ReactiveCommand<Unit, Unit> Back { get; }
ссылка: 1
public ReactiveCommand<Unit, Unit> Dot { get; }
ссылка: 1
public ReactiveCommand<Unit, Unit> Equal { get; }
```

Рис. 93. Команды.

```
Dot = ReactiveCommand.Create(() => {});
Del = ReactiveCommand.Create(() => {});
Back = ReactiveCommand.Create(() => {});
Equal = ReactiveCommand.Create(() => {});
```

Рис. 93. Инициализация команд.

Кроме того добавьте свойство `ShownValue` (рис. 94) в модель отображения и введите его привязку к свойству `Text` элемента `<TextBox>` (рис. 94).

```
private string shownValue = "";
Ссылок: 6
public string ShownValue
{
    get => shownValue;
    set => this.RaiseAndSetIfChanged(ref shownValue, value);
}
```

Рис. 93. Свойство ShownValue.

```
<TextBox Watermark="Введите число"
         TabIndex="0"
         Grid.Column="0"
         Grid.Row="0"
         Grid.ColumnSpan="5"
         IsReadOnly="True"
         Text="{Binding ShownValue}"/>
```

Рис. 94. Привязка.

Реализуем модель калькулятора, для этого создайте класс `Calc` в каталоге `Models` и приведите его к следующему виду (рис. 95).

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Demo_4.Models
{
    Ссылок: 3
    public class Calc
    {
        private double[] equation;
        private MOperator mOperator;
        private int current;
        private string result;
        ссылка: 1
        public Calc()
        {
            result = "0";
            equation = new double[2];
            current = 0;
            mOperator = MOperator.Add;
        }
    }
}
```

Рис. 95. Класс Calc.

В приведенном классе представлены следующие свойства:

1) `double[] equation` – массив содержащий два числа между которыми будет происходить математическая операция;

2) `MOperator mOperator` – выбранная математическая операция;

3) `int current` – указатель, показывающий, в которую ячейку массива `equation` производится запись;

4) `string result` – значение возвращаемое для вывода на экран калькулятора.

Добавьте в класс следующие методы (рис. 96-102).

```
public string addNumberFunc(int x)
{
    if (result == "0") result = "";
    if (double.TryParse(result + x.ToString(), out equation[current]))
    {
        result += x.ToString();
    }
    return result;
}
```

Рис. 96. Функция ввода цифры.

```
public string DotFunc()
{
    if (double.TryParse(result + ',', out equation[current]))
    {
        result += ',';
    }
    return result;
}
```

Рис. 97. Функция добавления запятой.

```
public string BackFunc()
{
    if (result == "") result = "0";
    if (result != "0" || result != "-")
    {
        result = result.Remove(result.Length - 1, 1);
        double.TryParse(result, out equation[current]);
    }
    return result;
}
```

Рис. 97. Функция стирания последнего символа.

```
public string DelFunc()
{
    current = 0;
    result = "0";
    mOperator = MOperator.Add;
    equation[0] = equation[1] = 0;
    return result;
}
```

Рис. 98. Функция очистки окна калькулятора.

```
public string EqualFunc()
{
    if (mOperator == MOperator.Add) equation[0] = equation[0] + equation[1];
    if (mOperator == MOperator.Subtract) equation[0] = equation[0] - equation[1];
    if (mOperator == MOperator.Multiply) equation[0] = equation[0] * equation[1];
    if (mOperator == MOperator.Divide) equation[0] = equation[0] / equation[1];
    result = equation[0].ToString();
    current = 0;
    return result;
}
```

Рис. 99. Функция «равно».

```
public string CommandFunc(MOperator x)
{
        mOperator = x;
        current = 1;
        result = "0";
    return result;
}
```

Рис. 100. Функция установки математического оператора.

Приведенные функции выполняют следующие задачи:

1) `addNumberFunc()` – добавляет в конец вводимого числа переданную в качестве параметра цифру. При вводе осуществляется проверка формируемого числа на соответствие типу double.

2) `DotFunc()` – добавляет в конец формируемого числа запятую. При вводе также осуществляется проверка формируемого числа на соответствие типу double.

3) `BackFunc()` – стирает последний введенный символ. При стирании также осуществляется проверка формируемого числа на соответствие типу double.

4) `DelFunc()` – очищает все переменные и переводит калькулятор в начальное состояние.

5) `EqualFunc()` – производит математическое вычисление в соответствии с выбранной математической операцией.

6) `CommandFunc()` – устанавливает текущую математическую операцию.

Подключим созданную модель к модели отображения, для этого добавьте в модель отображения поле `private Calc calc;`, проинициализируйте его из конструктора строкой `calc = new Calc();` и измените команды следующим образом (рис. 101). Откомпилируйте и проверьте работоспособность приложения.

```
addNumber = ReactiveCommand.Create<int>(x => { ShownValue = calc.addNumberFunc(x); });
Dot = ReactiveCommand.Create(() => { ShownValue = calc.DotFunc(); });
Del = ReactiveCommand.Create(() => { ShownValue = calc.DelFunc(); });
Back = ReactiveCommand.Create(() => { ShownValue = calc.BackFunc(); });
Equal = ReactiveCommand.Create(() => { ShownValue = calc.EqualFunc(); });
Command = ReactiveCommand.Create<MOperator>(x => { ShownValue =  calc.CommandFunc(x); });
```

Рис. 101. Функция установки математического оператора.

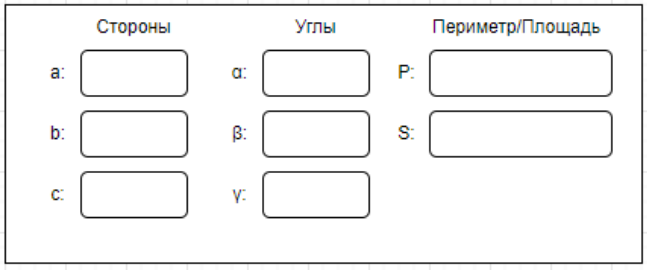
Контрольные вопросы

1. *Что такое стиль? Для чего он применяется?*
2. *Что такое селектор и как он формируется?*
3. *Как предварительно посмотреть созданный стиль, не подключая его к приложению?*
4. *Как подключить каталог, содержащий ресурсы, которые необходимо включить в сборку?*
5. *Как передать параметр команде?*

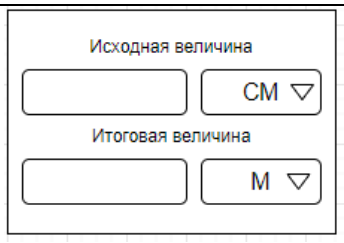
Самостоятельная работа

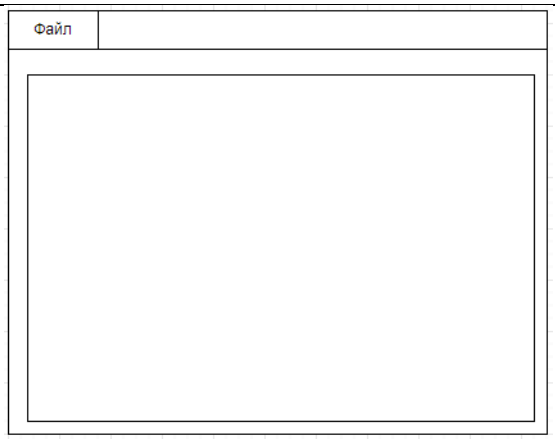
Создайте программу с применением фреймворка Avalonia согласно варианту. Каждый вариант включает в себя макет формы, задание (что должна делать программа), приблизительный перечень элементов интерфейса, из которых создается макет формы и примечание, в котором описана дополнительная справочная информация. Обратите внимание, что в Avalonia многие классы аналогичны WPF, но при этом они имеют собственную реализацию.

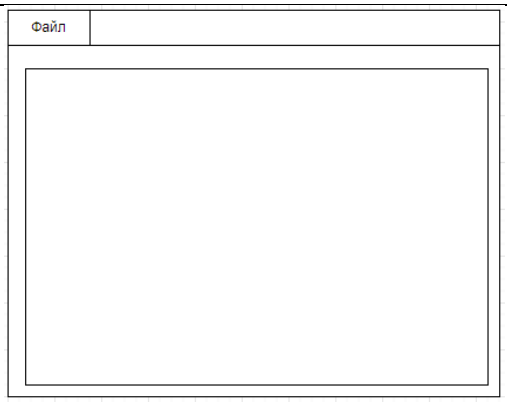
Вариант 1	
Макет	
Задание	Программа предназначена для решения квадратных уравнений, в том числе и с отрицательным дискриминантом.
Элементы интерфейса	TextBox, Label, StackPanel, Grid.
Примечание	Так как на форме отсутствует кнопка, расчет будет вестись автоматически при изменении свойств с помощью функции WhenAnyValue(). Для того чтоб сделать верхний индекс воспользуйтесь символами таблицы Unicode.

Вариант 2	
Макет	

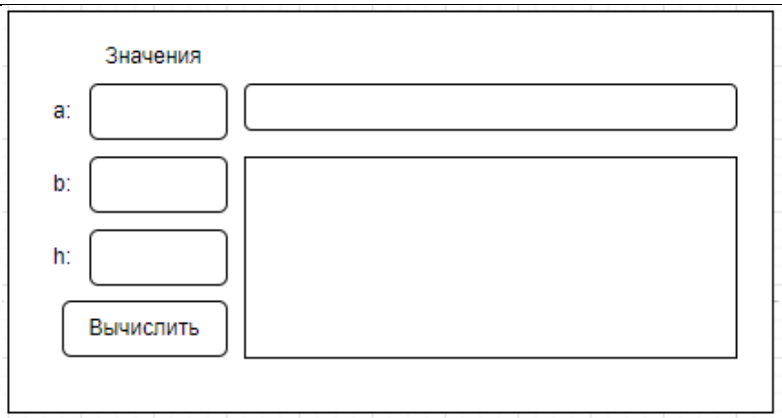
Задание	Программа предназначена для расчета характеристик треугольника по трем сторонам.
Элементы интерфейса	TextBox, Label, Grid.
Примечание	Так как на форме отсутствует кнопка, расчет будет вестись автоматически при изменении свойств с помощью функции WhenAnyValue(). Для того чтоб сделать латинские символы воспользуйтесь таблицей Unicode.

Вариант 3	
Макет	
Задание	Программа предназначена для перевода величин. Выбор величины (например, мера длинны, единица измерения информации и т.п.) осуществите самостоятельно.
Элементы интерфейса	TextBox, Label, ComboBox, Grid.
Примечание	Так как на форме отсутствует кнопка, расчет будет вестись автоматически при изменении свойств с помощью функции WhenAnyValue(). По новым элементам найдите и изучите справку самостоятельно.

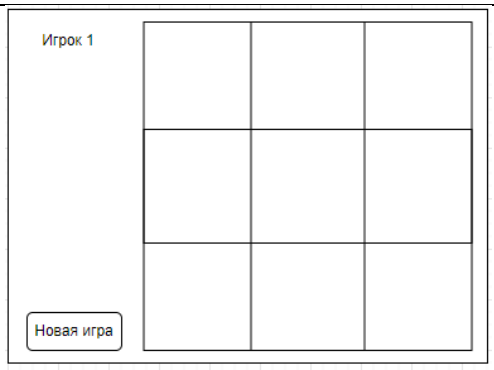
Вариант 4	
Макет	
Задание	Программа предназначена для вывода графических файлов. В меню должен быть доступен пункт «Открыть» открывающее диалоговое окно выбора файла.
Элементы интерфейса	Menu, Image, DockPanel.
Примечание	Функция открытия диалогового окна в Avalonia доступна только в асинхронном варианте. По новым элементам найдите и изучите справку самостоятельно.

Вариант 5	
Макет	
Задание	Программа предназначена для вывода текстовых файлов. В меню должен быть доступен пункт «Открыть» открывающее диалоговое окно выбора файла.
Элементы интерфейса	Menu, TextBox, DockPanel.

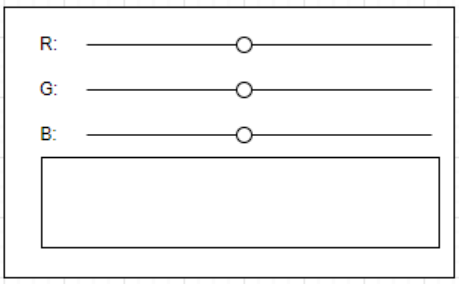
Примечание	Функция открытия диалогового окна в Avalonia доступна только в асинхронном варианте. TextBox поддерживает режим многострочности. По новым элементам найдите и изучите справку самостоятельно.

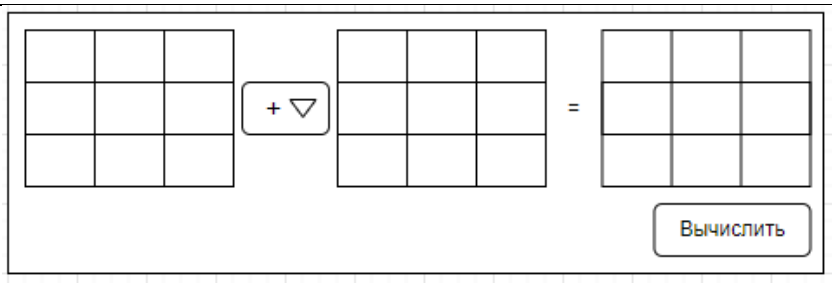
Вариант 6	
Макет	
Задание	Программа предназначена для расчета функции на указанном интервале с указанным шагом, при этом процесс расчета должен отображаться полосой прогресса и в текстовом поле. Функцию придумайте самостоятельно.
Элементы интерфейса	TextBox, Label, Button, ProgressBar, Grid.
Примечание	Напишите код в асинхронном стиле. TextBox поддерживает режим многострочности. По новым элементам найдите и изучите справку самостоятельно.

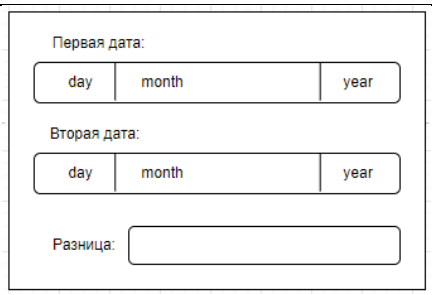
Вариант 7	
Макет	
Задание	Программа предназначена для отображения точек на координатной плоскости.
Элементы интерфейса	TextBox, Label, Button, Canvas, StackPanel, DockPanel.
Примечание	По новым элементам найдите и изучите справку самостоятельно.

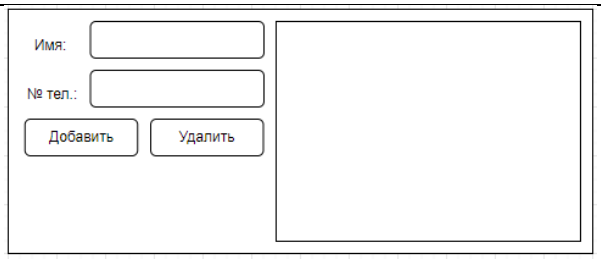
Вариант 8	
Макет	
Задание	Программа представляет собой игру «крестики нолики». В окне должен отображаться текущий игрок и кнопка новой игры. В конце игры должен отображаться победитель.
Элементы интерфейса	Label, Button, Grid.
Примечание	В ячейки поместите кнопки, внутрь которых поместите скрытый крестик и нолик. Видимость крестика и нолика управляет-

	ся нажатием самой кнопки.

Вариант 9	
Макет	
Задание	Программа предназначена для показа цвета в зависимости от ползунков RGB.
Элементы интерфейса	Slider, Panel, StackPanel.
Примечание	Изучите кисти (Brush, SolidColorBrush и т.д.). По новым элементам найдите и изучите справку самостоятельно.

Вариант 10	
Макет	
Задание	Программа выполняет математические операции с матрицами 3x3.
Элементы интерфейса	DataGrid, Grid, ComboBox, Button, Label.
Примечание	Установите пакет Avalonia.DataGrid с помощью менеджера пакетов NuGet. Привязку таблиц можно осуществить к двухмерным массивам, коллекциям и т.д. По новым элементам найдите и изучите справку самостоятельно.

Вариант 11	
Макет	
Задание	Программа должна вычислять разницу между двумя введенными датами.
Элементы интерфейса	DatePicker, Grid, TextBox, Label.
Примечание	Так как на форме отсутствует кнопка, расчет будет вестись автоматически при изменении свойств с помощью функции WhenAnyValue(). По новым элементам найдите и изучите справку самостоятельно.

Вариант 12	
Макет	
Задание	Программа предоставляет пользователю ввод имени и номера телефона, которые добавляются в список. Кроме того программа позволят удалить выбранный в списке элемент. Поле для ввода номера телефона должно предоставлять ввод по шаблону.
Элементы интерфейса	ListBox, Grid, StackPanel, Label, TextBox, Button, MaskedTextBox.
Примечание	По новым элементам найдите и изучите справку самостоятельно.

ЛИТЕРАТУРА

1. Назаркин, О. А. Разработка графического пользовательского интерфейса в соответствии с паттерном Model-View-Viewmodel на платформе Windows Presentation Foundation. Основные средства WPF: учебное пособие по дисциплине «Проектирование человеко-машинного интерфейса» / О. А. Назаркин. – Липецк: Липецкий государственный технический университет, ЭБС АСВ, 2014. – 61 с. – URL: https://www.iprbookshop.ru/55141.html (дата обращения: 09.06.2023).

2. Компаниец, В. С. Проектирование и юзабилити-исследование пользовательских интерфейсов : учебное пособие / В. С. Компаниец, А. Е. Лызь. – Ростов-на-Дону, Таганрог : Издательство Южного федерального университета, 2020. – 107 с. – URL: https://www.iprbookshop.ru/115528.html (дата обращения: 09.06.2023).

3. Разработка Windows-приложений в среде программирования Visual Studio.Net: учебно-методическое пособие по дисциплине Информатика и программирование / составители Ю. А. Воронцов, А. Г. Ерохин. – Москва : Московский технический университет связи и информатики, 2016. – 20 с. – URL: https://www.iprbookshop.ru/61536.html (дата обращения: 09.06.2023).

4. Техническая документация / Microsoft Learn. – Режим доступа: https://learn.microsoft.com/ru-ru/docs/ (дата обращения: 19.04.2023)

СОДЕРЖАНИЕ

Нацвин Алексей Викторович,

ассистент кафедры общей математики и информатики.

Татьяна Александровна Юрьева,

доц. кафедры общей математики и информатики АмГУ, канд. пед. наук