МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ХАРКІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ РАДІОЕЛЕКТРОНІКИ

С. П. Новоселов, О. В. Сичова

ОСНОВИ РОЗРОБКИ КРОСПЛАТФОРМНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ НА AVALONIA

Навчальний посібник

ВИДАВНИЦТВО ХАРКІВ 2024 І. С. ІВАНЧЕНКА

УДК 681.5:004.42
Н76

Рекомендовано Вченою радою
Харківського національного університету радіоелектроніки
(протокол № 6/1 від 30.04.2024 р.)

Новоселов С. П.

Н76 Основи розробки кросплатформного програмного забезпечення на Avalonia : Навчальний посібник / С. П. Новоселов, О. В. Сичова. – Харків: Видавництво Іванченка І. С., 2024. – 267 с.

ISBN 978-617-8332-57-0
DOI: 10.30837/978-617-8332-57-0

У навчальному посібнику містяться відомості для розробки додатків на Framework Avalonia. Розглядаються різноманітні аспекти, починаючи від налаштування робочого середовища, встановлення .NET на платформі Linux, і завершуючи реалізацією доступу до бази даних та відображенням структурованої інформації. Посібник також надає вичерпну інформацію щодо створення інтерфейсу користувача засобами Avalonia, взаємодії між візуальними елементами, використання елементів користувача, роботи з XAML та командами Avalonia. Подана інформація щодо прив'язки даних, роботи з властивостями стилю, маршрутизації подій, шаблонів даних і моделей відображення.

Навчальний посібник призначено для підготовки здобувачів освіти першого (бакалаврського), другого (магістерського) рівнів освіти в галузі електроніка, автоматизація та електронні комунікації, а також інформаційні технології. Може бути корисним розробникам програмного забезпечення, які цікавляться конструюванням користувацьких інтерфейсів з використанням Framework Avalonia. Особливо корисним цей посібник буде для тих, хто працює на платформі Linux і хоче використовувати Visual Studio Code або MS Visual Studio як інструменти розробки.

УДК 681.5:004.42

Рецензенти:

– **Кобилін О. А.**, канд техн. наук, доцент, завідувач кафедри інформатики ХНУРЕ;

– **Цимбал О. М.**, д-р техн. наук, професор, професор кафедри комп'ютерно-інтегрованих технологій, автоматизації та робототехніки, ХНУРЕ;

– **Нефьодов Л. І.**, д-р техн. наук, професор, завідувач кафедри автоматизації та комп'ютерно-інтегрованих технологій ХНАДУ.

ISBN 978-617-8332-57-0
DOI: 10.30837/978-617-8332-57-0

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

IDE – Integrated Development Environment;
MAUI – Multi-Platform App UI;
MVVM – Model-View-ViewModel;
VM – View Model;
WPF – Windows Presentation Foundation;
XAML – Extensible Application Markup Language;
XML – Extensible Markup Language;
ПК – персональний комп’ютер.

ВСТУП

У навчальному посібнику розглядаються питання використання інструментів Avalonia для створення програмного забезпечення. Avalonia – це кросплатформна технологія створення додатків з графічним інтерфейсом користувача для .NET на основі XAML, що забезпечує гнучку систему стилів і підтримує широкий спектр операційних систем, таких як Windows через .NET Framework і .NET Core, Linux через Xorg, macOS, Android. Також, за допомогою Avalonia можна створювати серверні Web застосунки.

Альтернативним засобом створення кросплатформних додатків з використанням мови програмування C# є MAUI від Microsoft. MAUI – це еволюція Xamarin.Forms, що дозволяє розробникам створювати програми за допомогою уніфікованого API, використовуючи платформи, що лежать в основі набору інструментів власного інтерфейсу користувача.

MAUI потрібно розглядати як абстракцію над існуючими елементами керування інтерфейсу користувача операційної системи. Таким чином, Xamarin.Forms може забезпечити лише найменший спільний знаменник елементів керування інтерфейсу користувача та API, доступних на підтримуваних платформах.

На відміну від MAUI, Avalonia UI – це цілий власний набір інструментів інтерфейсу користувача, який відповідає за рендеринг кожного пікселя інтерфейсу користувача. Цей підхід ближче до того, як розроблено фреймворк інтерфейсу користувача Flutter від Google, і пропонує численні переваги, зокрема ідеальні піксельні програми на кожній платформі та спрощений процес підтримки нових платформ.

Навчальний посібник складається з семи розділів. В першому розділі подано особливості Framework Avalonia та інформацію про Model-View-ViewModel. Другий розділ присвячений встановленню Visual Studio Code і .NET на платформі Linux. В третьому розділі подано опис віджетів для організації діалогу з користувачем засобами Avalonia. Четвертий розділ присвячений реалізації взаємодії між візуальними елементами в Avalonia. У п’ятому розділі описані базові функції XAML, який дозволяє створювати складні інтерфейси за допомогою вкладених тегів і властивостей, що забезпечує чистий та структурований код. Він також підтримує прив’язку даних, анімацію, стилі,

шаблони та інші потужні можливості для розробки інтерфейсів користувача. У шостому розділі розглядається застосування різноманітних елементів користувача в Avalonia, шаблони даних і моделі відображення. Avalonia DataGrid, який є корисним інструментом для роботи з табличними даними у додатках, надаючи можливості відображення, редагування та керування даними у зручному форматі, поданий у сьомому розділі.

Навчальний посібник має велику кількість прикладів та практичних порад і буде корисним ресурсом для всіх, хто цікавиться розробкою кросплатформних додатків на Avalonia. Повні тексти прикладів, що подані в навчальному посібнику, читач може знайти на ресурсі Github за посиланням https://github.com/s-novoselov/THE_BASICS_OF_AVALONIA у вигляді лістингів програм.

Поданий в навчальному посібнику матеріал рекомендується використовувати у навчальному процесі для здобувачів освіти першого (бакалаврського), другого (магістерського) рівнів освіти в галузі електроніка, автоматизація та електронні комунікації, а також інформаційні технології.

1 ОСОБЛИВОСТІ FRAMEWORK AVALONIA

1.1 Основні відомості про Avalonia

Avalonia – проєкт з відкритим вихідним кодом, легкий в опануванні, призначений для кросплатформної розробки програмних засобів з графічним інтерфейсом із використанням технології .NET C#. Наразі, можна використовувати Avalonia для створення десктопних додатків для ОС Windows, Mac і Linux, а також мобільних додатків для Android та iOS (рис. 1.1).

Рисунок 1.1 – Підтримка різних операційних систем

Авалонія перебуває в стадії швидкого розвитку та має широку підтримку спільноти. Містить доволі детальну документацію, що постійно розширюється, і велику кількість зразків та прикладів із відкритим кодом, які демонструють, як нею користуватися.

Avalonia підтримує Visual Studio та Rider для створення файлів XAML, реалізуючи всі переваги XAML IntelliSense. Visual Studio також має візуальний попередній перегляд графічного інтерфейсу в процесі створення додатка. Avalonia підтримує зв'язування, MVVM, елементи керування без графічного відображення та шаблони даних так само, як і фреймворк XAML.

Avalonia містить інструмент, який полегшує візуальну розробку та налагодження під назвою DevTools. Цей інструмент також

є багатоплатформним, тобто можна використовувати його на Mac і Linux, а також має повністю відкритий вихідний код.

Avalonia має збіжності з WPF або UWP, тому розробники, що використовують дані технології, досить швидко можуть адаптуватись до нового інструменту.

Переваги Avalonia:

– Avalonia дозволяє створювати програми, які виглядають і поводяться однаково на більшості популярних платформ для настільних ПК (Windows, Mac, різні варіанти Linux), а також дозволяє налаштовувати специфічні платформи;

– Avalonia можна використовувати з поширеними фреймворками типу Model-View;

– Avalonia підтримується Schneider Electric, Unity, Grit World та іншими відомими компаніями (рис. 1.2).

Рисунок 1.2 – Компанії, що підтримують Avalonia

Avalonia (так само, як WPF і Silverlight) є повністю композиційною: можна створити необхідний віджет, наприклад, кнопку з примітивів Avalonia, подібно до створення складної сторінки, в той час, як пакети його конкурентів (Web, Xamarin і Java) не є композиційними в однаковій мірі.

Інтерфейс користувача Avalonia, на відміну від JavaScript, має всі переваги сильно типізованої мови. Крім того, Avalonia має всі переваги мови скомпільованої в двійковий код.

Avalonia є кросплатформним нащадком відомих пакетів Microsoft (WPF і Silverlight), які вивели розробку інтерфейсу користувача на абсолютно новий рівень, створивши набір нових концепцій, які дозволяють створити повноцінний

інтерфейс-додаток швидше і простіше порівняно з іншими технологіями. Серед таких концепцій можна виділити:

– візуальні та логічні дерева;

– додані властивості або залежності, які можуть бути визначені за межами об’єкта, до якого вони відносяться, та не займають додаткової пам’яті, якщо їм не призначено значення, відмінне від значення за замовчуванням, і мають спеціальну подію, яка запускається, коли їх значення змінюються;

– прикріплені маршрутизовані події, які можуть бути визначені за межами об’єктів, які їх запускають, і можуть поширюватися та оброблятися вгору та вниз на логічних деревах;

– прив’язка до відповідного шаблону MVVM;

– використання шаблонів керування;

– використання шаблонів даних;

– стилі;

– можна змінювати та доповнювати поведінку класу C#, не змінюючи сам клас за допомогою подій.

Усі ці парадигми були реалізовані для перелічених раніше операційних систем, що підтримуються в Avalonia.

Avalonia підтримується на таких платформах:

– Windows 8 і вище (Avalonia також працює коректно в Windows 7, але офіційно не підтримується);

– MacOS High Sierra 10.13 і вище;

– для Linux:

а) Debian 9 (Stretch) і вище;

б) Ubuntu 16.5 і вище;

в) Fedora 30 і вище;

– iOS 13.0 і вище;

– Web Application.

Наведені нижче середовища підтримують Avalonia XAML з IntelliSense:

– Visual Studio 2017 і вище (з або без Resharper 2020.3). Також підтримується Avalonia Visual Designer.

– JetBrains Rider 2020.3 і вище.

Необхідно також зазначити ще декілька особливостей Avalonia. Фреймворк Avalonia (як і WPF) на 100% є композиційним – просту кнопку можна зібрати з примітивів, таких як геометричні контури, рамки та зображення, так

само, як можна створювати дуже складні сторінки чи види. Як показує практика, кінцевий продукт дуже залежить від того, як буде виглядати та поводитися кожен елемент керування, а також, які його властивості можна налаштувати. Більше того, прості примітиви можна організувати в більш складні, що надає багато свободи дій розробнику. Ні фреймворк HTML/JavaScript/TypeScript, ні Xamarin не є композиційними в однаковій мірі – насправді їх примітивами є кнопки, прапорці та меню, які мають багато властивостей, які можна змінити для налаштування (деякі властивості можуть бути специфічними для платформи або браузера). У цьому плані розробник Avalonia має набагато більше свободи створювати все, що задовольняє потребам клієнта.

WPF запропонував багато нових парадигм розробки, які можуть допомогти розробляти візуальні додатки значно швидше і чистіше – серед них візуальні та логічні дерева, прив'язки, прикріплені властивості, приєднані маршрутизовані події, шаблони даних і керування, стилі, поведінка. Дуже мало з цих парадигм реалізовано у веб-фреймворках і Xamarin, і вони там значно менш потужні, тоді як в Avalonia – усі вони реалізовані, а деякі, наприклад, властивості та прив'язки реалізовані навіть більш потужним способом, ніж у WPF.

1.2 Основи Model-View-ViewModel

Model-View-ViewModel – шаблон проєктування, який застосовується в процесі проєктування архітектури застосунків (додатків). Публічно вперше був представлений Джоном Госсманом (John Gossman) у 2005 році як модифікація шаблону Presentation Model. MVVM орієнтований на такі сучасні платформи розробки, як Windows Presentation Foundation та Silverlight від компанії Microsoft.

MVVM полегшує відокремлення розробки графічного інтерфейсу від розробки бізнес логіки (бек-енд логіки), відомої як модель (можна також сказати, що це відокремлення представлення від моделі). Модель представлення є частиною, яка відповідає за перетворення даних для їх подальшої підтримки і використання. З цієї точки зору, модель представлення більше схожа на модель, ніж на представлення і обробляє більшість, якщо не всю, логіку відображення даних. Модель представлення може також реалізовувати патерн медіатор, організовуючи доступ до бек-енд логіки навколо множини правил використання, які підтримуються представленням.

MVVM використовується для відокремлення моделі та її відображення. Необхідністю цього є надання можливості змінювати їх незалежно одну від одної. Наприклад, розробник працює над логікою роботи з даними, а дизайнер – з користувацьким інтерфейсом (рис. 1.3).

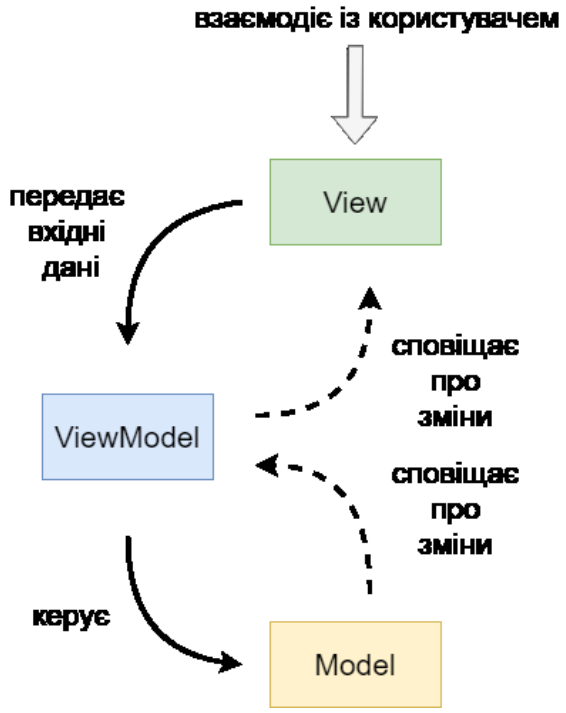


Рисунок 1.3 – Діаграма взаємодії між компонентами шаблона MVVM

MVVM була створена з метою поділу праці дизайнера і програміста, що є неможливим, коли Java-розробник намагається побудувати GUI в Swing або розробник на Visual C++ намагається створити інтерфейс користувача в MVC. Розробники кмітливі і мають безліч навичок, але створення зручних і привабливих інтерфейсів вимагає абсолютно інших талантів, ніж ті, якими вони володіють. Ця робота більше підходить для дизайнерів інтерфейсів. Хороші дизайнери інтерфейсів краще знають, чого бажають користувачі, ніж експерти в області проєктування і написання коду. Зрозуміло, буде краще, якщо дизайнер інтерфейсів створить інтерфейс, а розробник напише код, який реалізує логіку цього інтерфейсу, але технології типу Swing або MVC не дозволяють цього.

MVVM зручно використовувати замість класичного MVC та йому подібних у тих випадках, коли на платформі, де ведеться розробка, присутнє «зв’язування даних».

В MVC/MVP зміни у користувацькому інтерфейсі не впливають безпосередньо на модель, а йдуть через Контролер/Presenter. У таких технологіях, як WPF та Silverlight, присутня концепція «зв'язування даних», що дозволяє зв'язувати дані із візуальними елементами в обидві сторони.

Архітектура MVVM вирішує цю проблему чітким розподілом відповідальності:

– розробка інтерфейсу користувача здійснюється дизайнером інтерфейсів за допомогою технології, більш-менш природної для такої роботи (XML);

– логіка інтерфейсу користувача реалізується розробником як компонент ViewModel;

– функціональні зв'язки між інтерфейсом користувача та ViewModel реалізуються через біндинги (bindings), які, по суті, є правилами типу «якщо кнопка А була натиснута, повинен бути викликаний метод onButtonAClick() з ViewModel». Біндинги можуть бути написані в коді або визначені декларативним шляхом (Android використовує обидва типи).

Архітектура MVVM використовується в тому чи іншому вигляді усіма сучасними технологіями, наприклад Microsoft WPF і Silverlight, Oracle JavaFX, Adobe Flex, AJAX.

Шаблон MVVM ділиться на три частини:

– модель (Model), як і в класичному шаблоні MVC, Модель являє собою фундаментальні дані, що необхідні для роботи застосунку;

– вигляд (View) як і в класичному шаблоні MVC, Вигляд – це графічний інтерфейс, тобто вікно, кнопки тощо;

– модель вигляду (ViewModel, що означає «Model of View») з одного боку є абстракцією Вигляду, а з іншого надає обгортку даних з Моделі, які мають зв'язуватись. Тобто вона містить Модель, яка перетворена до Вигляду, а також містить у собі команди, якими може скористатися Вигляд для впливу на Модель. Фактично ViewModel призначена для того, щоб:

а) здійснювати зв'язок між моделлю та вікном;

б) відслідковувати зміни в даних, що зроблені користувачем;

в) відпрацьовувати логіку роботи View (механізм команд).

Розглянемо приклад реалізації шаблону у Windows Presentation Foundation.

Оголосимо модель, що містить логіку аплікації, не залежну від представлення:

```
public class OrderDto
{
	public string Name { get; set; }
}

public interface IOrdersModel
{
	OrderDto[] LoadOrders();
}

public class OrdersModel : IOrdersModel
{
	public OrderDto[] LoadOrders()
	{
		return new OrderDto[]
		{
			new OrderDto(){ Name = "Item1" },
			new OrderDto(){ Name = "Item2" },
		};
	}
}
```

Додамо модель вигляду – компонент, що зв’язує логіку аплікації (модель) із виглядом:

```
public class OrderViewModel
{
	private readonly IOrdersModel ordersModel;

    // колекція, що вміє сповіщати про зміну своїх даних
    public ObservableCollection<string> Orders { get; set; } = new ObservableCollection<string>();

    public OrderViewModel(IOrdersModel ordersModel)
    {
		this.ordersModel = ordersModel;
	}

	// метод-обробник, виконується при взаємодії користувача із виглядом
    public ICommand LoadOrders => new RelayCommand((object sender) =>
    {
		// керування моделю
```

```
        var orders = ordersModel.LoadOrders();

            // оновлення колекції до якої прив’язаний вигляд
            foreach (var order in orders)
            {
            Orders.Add(order.Name);
            }
    });
}
```

Тепер додамо вигляд – для взаємодії із користувачем:

```
<Window>
      <Button Command="{Binding LoadOrders}" />

      // зв’язування даних
      <ListBox ItemsSource="{Binding Orders}"/>
</Window>
```

1.3 Контрольні запитання та завдання

1. Охарактеризуйте фреймворк Avalonia.
2. Чим фреймворк Avalonia відрізняється від інших сучасних технологій кросплатформної розробки?
3. В чому полягають переваги Avalonia?
4. Які платформи підтримують фреймворк Avalonia?
5. В чому полягає особливості шаблону проєктування MVVM?

2 НАЛАШТУВАННЯ РОБОЧОГО СЕРЕДОВИЩА

2.1 Visual Studio Code

2.1.1 Встановлення Visual Studio Code на ОС Linux

Visual Studio Code – це невеликий за розміром програмний засіб, що включає лише мінімальну кількість компонентів, які використовуються для більшості робочих процесів розробки. Після встановлення, включені лише основні функції, такі як редактор, засоби керування файлами, інструменти керування вікнами та налаштування параметрів. Служба мови JavaScript/TypeScript і інструмент Node.js також є частиною базового встановлення.

Для завантаження інсталяційного пакету необхідно перейти за посиланням:

```
https://code.visualstudio.com/download
```

Відкриється сторінка із вибором варіанта завантаження, що подано на рис. 2.1.

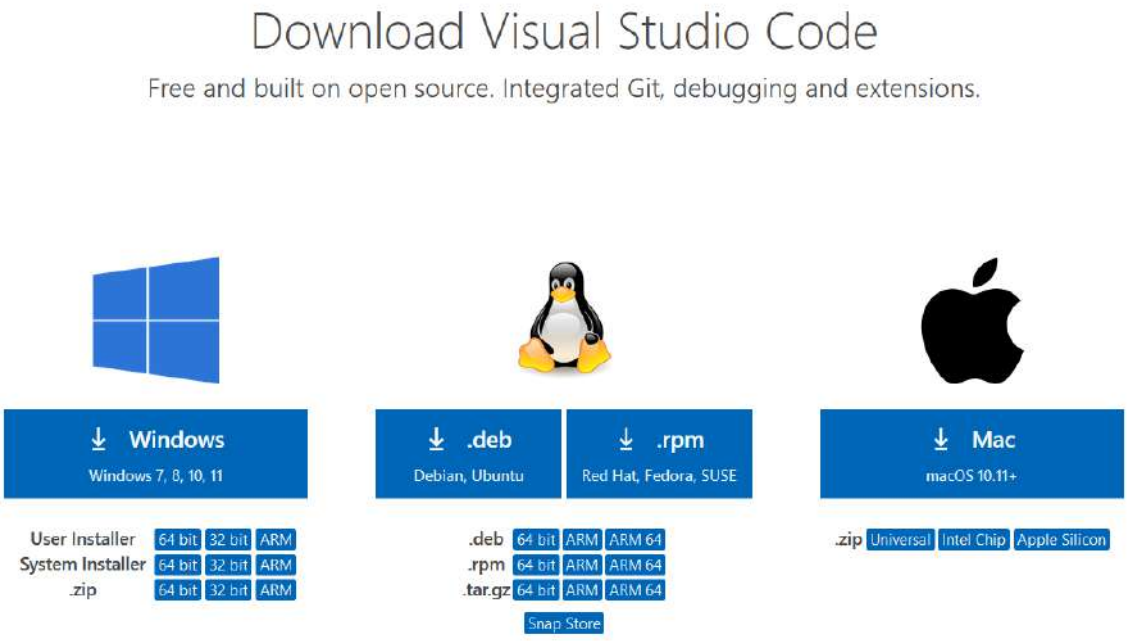


Рисунок 2.1 – Сторінка вибору варіанту завантаження Visual Studio Code для різних платформ

Для ОС Linux існує декілька варіантів встановлення Visual Studio Code.

Перший варіант: Visual Studio Code офіційно розповсюджується, як пакет Snap у Snap Store. Отримати його можна з магазину Snap. В даному випадку встановити його можна, виконавши наступну команду:

```
sudo snap install --classic code # or code-insiders
```

Після встановлення Snap Daemon буде самостійно піклуватися про автоматичне оновлення VS-коду у фоновому режимі. Користувач також отримає повідомлення про оновлення продукту, коли воно буде доступне в репозиторії.

Другий варіант: використання дистрибутиву на основі Debian та Ubuntu.

Найпростіший спосіб встановити Visual Studio Code для дистрибутивів на основі Debian, або Ubuntu – це завантаження та встановлення пакета .deb (64-біт). Посилання для завантаження:

```
https://go.microsoft.com/fwlink/?LinkID=760868
```

Встановити можна або через графічний центр програмного забезпечення, якщо він доступний, або через командний рядок за допомогою команди:

```
sudo apt install ./<file>.deb
```

Замість <file> необхідно написати повну назву пакету, що було завантажено, наприклад:

```
code_1.65.2-1646927742_amd64.deb
```

В даному випадку команда для встановлення пакету буде наступна:

```
sudo apt install ./code_1.65.2-1646927742_amd64.deb
```

Якщо використовується класичний дистрибутив ОС Linux, доведеться використовувати наступну послідовність команд:

```
sudo dpkg -i <file> .deb
sudo apt-get install -f       # Встановлення залежностей
```

Встановлення пакета .deb автоматично встановить Apt Repository та цифровий ключ, щоб увімкнути автоматичне оновлення за допомогою менеджера пакетів системи.

Якщо користувач звик працювати з більшими, монолітними інструментами розробки (IDE), він можете бути здивований, що звичайні сценарії роботи не повністю підтримуються з «коробки». Наприклад, немає діалогового вікна «Файл => Новий проєкт» із попередньо встановленими шаблонами проєкту. Більшості користувачам VS Code потрібно буде встановити додаткові компоненти в залежності від їхніх конкретних потреб. На рис. 2.2 подано приклад поширених додаткових компонентів для VS Code.

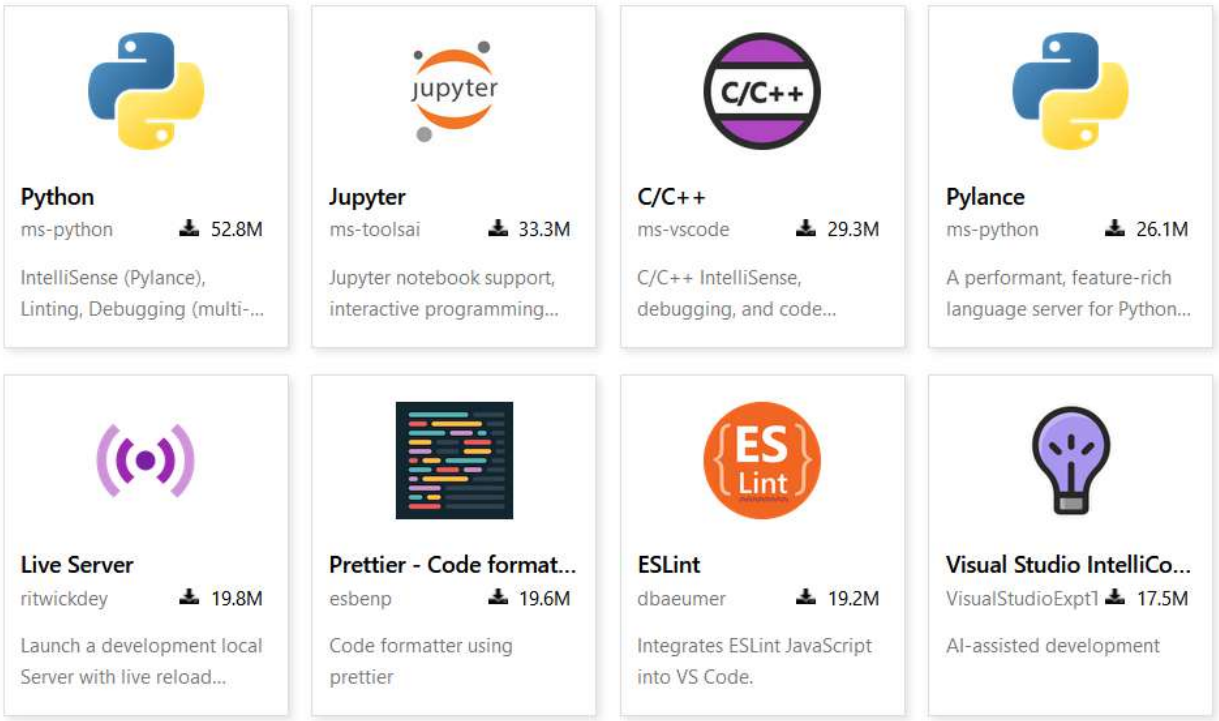


Рисунок 2.2 – Приклад поширених додаткових компонентів

Ось декілька компонентів, що найчастіше встановлюються:

– Git [https://git-scm.com/download] – VS Code має вбудовану підтримку для керування вихідним кодом за допомогою Git, але вимагає окремого встановлення Git;

– Node.js [https://nodejs.org] – кросплатформне середовище виконання для створення та запуску програм JavaScript;

– TypeScript [https://www.typescriptlang.org] – компілятор TypeScript, tsc, для трансляції TypeScript у JavaScript.

2.1.2 Інтерфейс користувача Visual Studio Code

По суті, Visual Studio Code є редактором коду. Як і багато інших редакторів коду, VS Code використовує звичайний інтерфейс користувача: редактор програмного коду та менеджер файлів, що розташовано зліва від нього, де відображаються всі файли і папки, до яких програміст має доступ.

На рис. 2.3 подано типовий вигляд інтерфейсу користувача.

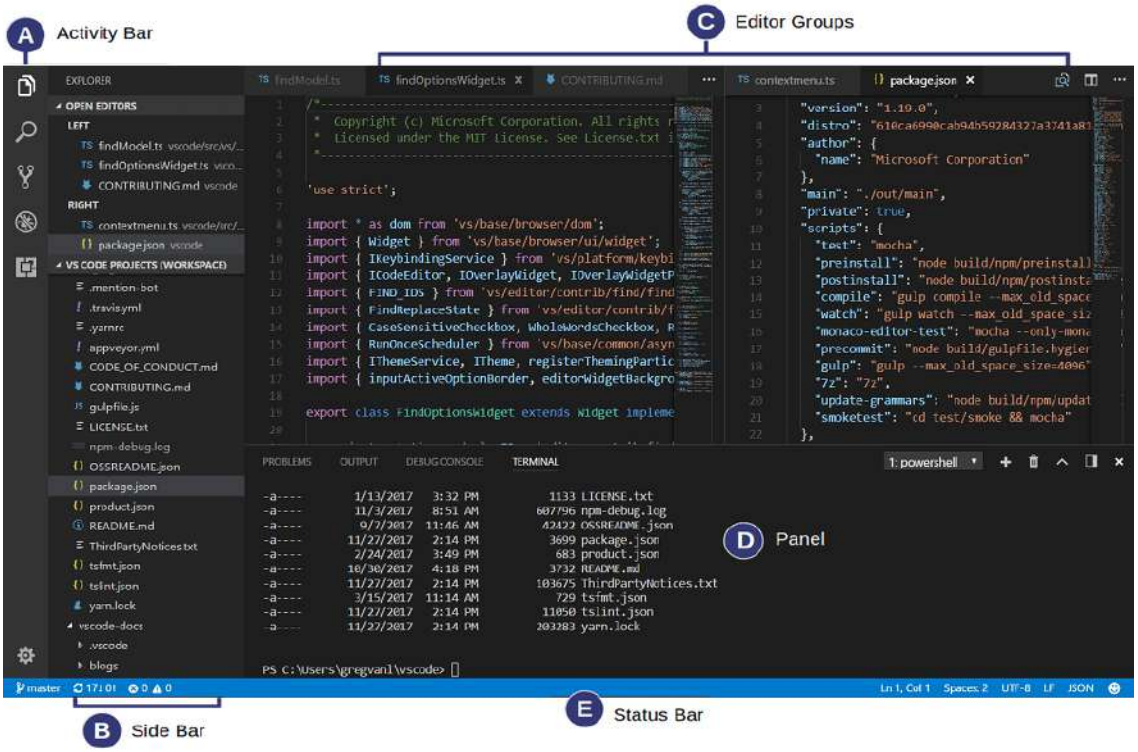


Рисунок 2.3 – Основні елементи інтерфейсу користувача

VS Code має простий та інтуїтивно зрозумілий інтерфейс, який максимально збільшує простір, наданий редактору, залишаючи достатньо місця для перегляду та доступу до повного контексту робочої папки або проєкту. Інтерфейс користувача розділений на п’ять областей:

– редактор (Editor) – основна область редагування файлів. Програміст може відкривати скільки завгодно редакторів поруч по вертикалі та горизонталі;

– бічна панель (Side bar) – містить різні види папок і файлів, подібно до провідника файлового менеджеру Explorer, щоб допомогти користувачеві під час роботи над проєктом;

– рядок стану (Status Bar) – інформація про відкритий проєкт і файли, які редагуються;

– панель активності (Activity Bar). Розташована в дальній лівій частині, вона дає змогу перемикатися між представленнями та надає додаткові індикатори, що залежать від контексту та від кількість вихідних змін, коли увімкнено Git;

– панелі (Panel) – редактор може відображати різні панелі під областю редактора для виведення налагоджувальної інформації, помилок і попереджень або вбудованого терміналу. Панель також можна перемістити вправо, щоб збільшити простір по вертикалі.

Кожен раз, коли запускається VS Code, він відкривається в тому ж стані, в якому був, коли востаннє його закривали. Папка, макет і відкриті файли зберігаються.

Відкриті файли в кожному редакторі відображаються із заголовками вкладок (Tabs) у верхній частині області редактора.

Більше інформації про основні компоненти інтерфейсу користувача можна знайти на офіційному сайті програми за посиланням:

`https://code.visualstudio.com/docs/getstarted/userinterface`

Треба особливо виділити такий компонент інтерфейсу, як Палітра команд (Command Palette) (рис. 2.4).

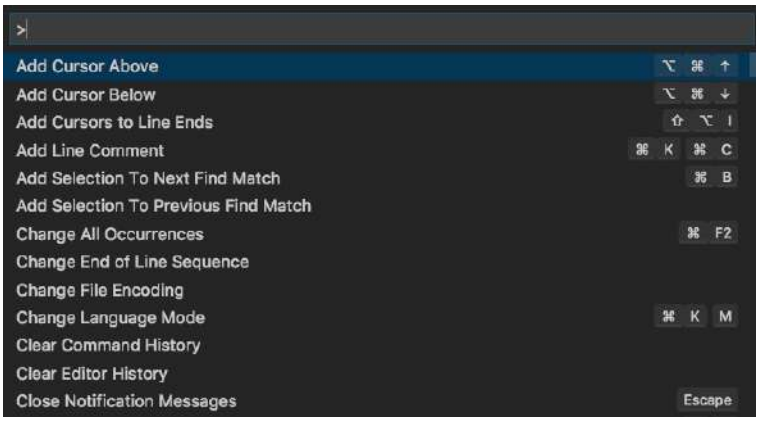


Рисунок 2.4 – Палітра команд

Палітра команд у VS Code однаково доступна з меню графічного інтерфейсу і з клавіатури. Найважливіша комбінація клавіш, яку потрібно знати, – це Ctrl+Shift+P, яка відкриває палітру команд. Звідси програміст має доступ до

всіх функцій VS Code, включаючи комбінації клавіш для найпоширеніших операцій.

Палітра команд надає доступ до багатьох команд. Користувач може виконувати команди редактора, відкривати файли, шукати символи та переходити до потрібного рядку у файлі, використовуючи те саме інтерактивне вікно. Ось кілька поширених команд:

– Ctrl+P – дозволить перейти до будь-якого файлу або символу, ввівши його назву;

– Ctrl+Tab – прокручує останній набір відкритих файлів;

– Ctrl+Shift+P – переведе безпосередньо до команд редактора;

– Ctrl+Shift+O – дозволить перейти до певного символу у файлі;

– Ctrl+G – дозволить перейти до певного рядка у файлі.

Якщо ввести символ «?» у поле введення, то на екран буде виведено список всіх доступних команд, які можна виконати в даній палітрі (рис. 2.5).

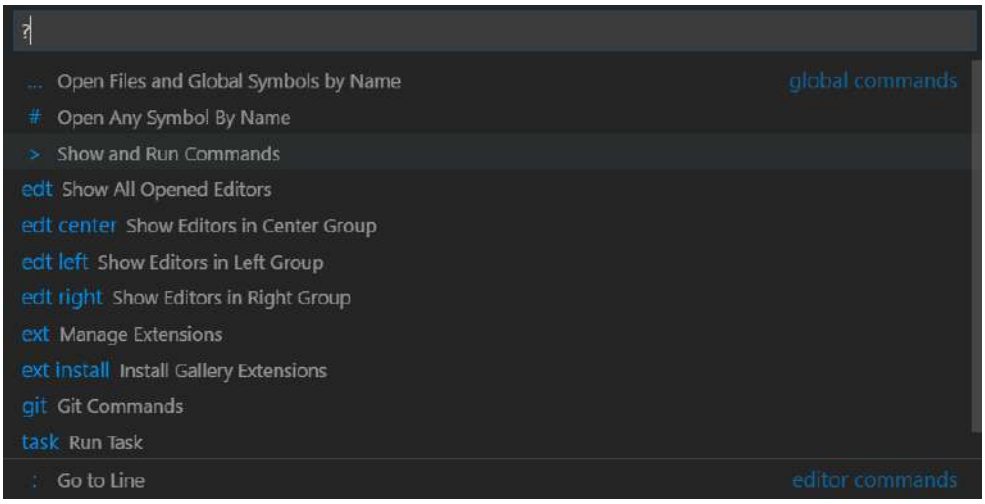


Рисунок 2.5 – Список всіх доступних команд

2.1.3 Інтегрований термінал

Visual Studio Code містить повнофункціональний інтегрований термінал, який зручно починається в корені робочої області. Він забезпечує інтеграцію з редактором для підтримки таких функцій, як посилання та виявлення помилок (рис. 2.6).

Щоб відкрити термінал, необхідно виконати одну з наступних дій:

– використати комбінацію клавіш Ctrl+` із символом зворотного апострофу;

– використати команду меню View Terminal;

– на панелі команд (Ctrl+Shift+P) скористатися командою «Перегляд: переключити термінал»;

– можна створити новий термінал через меню «Термінал» за допомогою «Термінал Новий термінал».

Рисунок 2.6 – Інтегрований термінал

Для відкриття зовнішнього терміналу необхідно скористатися комбінацією клавіш Ctrl+Shift+C.

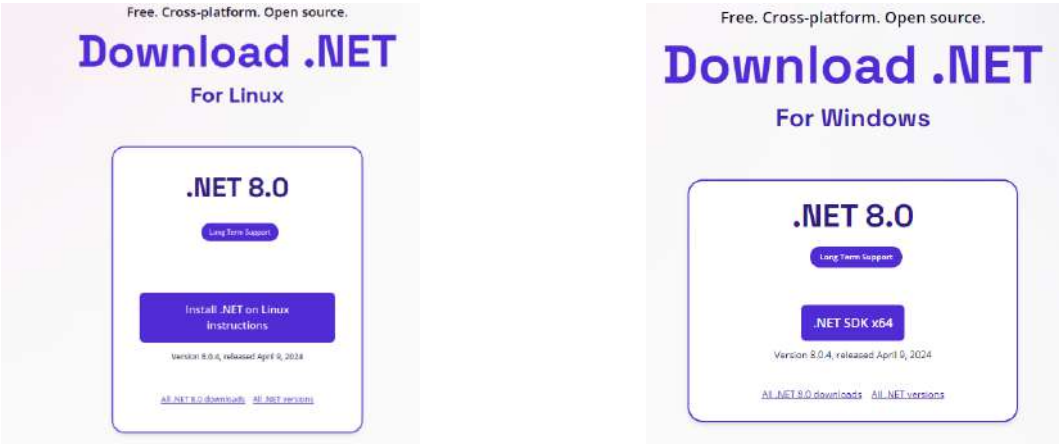
2.2 Встановлення .NET на платформі Linux

2.2.1 Розгортання SDK .NET

Для розробки програм з використанням Avalonia необхідно встановити SDK .NET. Для завантаження SDK необхідно перейти на офіційну сторінку за посиланням:

```
https://dotnet.microsoft.com/en-us/download
```

В результаті відкриється вибір платформи для розгортання SDK в залежності від версії ОС, з якої здійснено вхід в браузер (рис. 2.7).



а) б)

Рисунок 2.7 – Вибір платформи для розгортання SDK

Обираємо платформу Linux та натискаємо на кнопку «Install .NET on Linux».

.NET доступний у різних дистрибутивах Linux. Більшість платформ і дистрибутивів Linux випускають основну версію щороку, і більшість із них надає менеджер пакетів, який використовується для встановлення .NET. У даному підрозділі описано, що зараз підтримується та який менеджер пакетів використовується. Усі випуски .NET підтримуються, доки або версія .NET не закінчиться, або дистрибутив Linux не закінчиться. Для найкращої сумісності виберіть версію довгострокового випуску (LTS).

На момент написання даного посібника наступні версії .NET більше не підтримуються, але завантаження для них все ще залишаються опублікованими:

– .NET Core 3.0;

– .NET Core 2.2;

– .NET Core 2.1;

– .NET Core 2.0.

Далі буде описана ручна установка, без використання менеджеру пакетів для встановлення .NET у Linux. Встановити .NET можна одним із наступних способів:

– використовуючи пакети Snap;

– скриптовим встановленням за допомогою install-dotnet.sh;

– ручним розпакуванням бінарних файлів.

Ми обираємо другий варіант встановлення, що є одним із поширених варіантів та підходить для більшості дистрибутивів.

Скрипт dotnet-install використовується для автоматизації та встановлення пакета SDK та Runtime без права адміністратора. Він можете бути завантажений за посиланням:

```
https://dot.net/v1/dotnet-install.sh
```

Для запуску сценарію потрібен Bash.

Перш за все необхідно змінити права доступу до файлу сценарію та зробити його доступним для запуску. Робиться це за допомогою команди в каталозі, де зберігається сценарій:

```
sudo chmod u+x dotnet-install.sh
```

Сценарій за замовчуванням встановлює останню версію довгострокової підтримки SDK (LTS), тобто .NET 6. Щоб інсталювати поточний випуск, який може не бути версією (LTS), використовується параметр -c Current:

```
./dotnet-install.sh -c Current
```

Якщо необхідно встановити певну версію SDK, потрібно змінити параметр `-c`, щоб вказати конкретну версію. Наступна команда встановлює .NET SDK 6.0:

```
./dotnet-install.sh -c 6.0
```

Наступним кроком необхідно скористатися командою експорту, щоб додати DOTNET_ROOT, яка посилається на розташування папки із встановленим SDK в змінну PATH. Це зробить команди .NET CLI доступними у терміналі.

Більш правильним вважається зробити так, щоб параметр DOTNET_ROOT підтягувався кожен раз на старті операційної системи. Для цього потрібно відредагувати свій профіль оболонки, щоб додати необхідні команди. Існує кілька різних оболонок, доступних для Linux, і кожна з них має свій профіль.

Наприклад:

– оболонка Bash:

```
/.bash_profile,
/.bashrc
```

– оболонка Korn:

```
/.kshrc або .profile
```

– Z Shell:

```
/.zshrc або .zprofile
```

Необхідно відредагувати відповідний вихідний файл для конкретної оболонки та додати :$HOME/.dotnet в кінець існуючого оператора PATH. Якщо оператор PATH не включено, потрібно додати новий рядок:

```
export PATH=$PATH:$HOME/.dotnet.
```

Також в кінець файлу потрібно додати:

```
export DOTNET_ROOT=$HOME/.dotnet
```

2.2.2 Додавання підтримки мови програмування C# до Visual Studio Code

Підтримка C# у Visual Studio Code оптимізована для кросплатформної розробки .NET Core. VS Code підтримує налагодження програм C#, що працюють на .NET Core або Mono.

Підтримка мови C# потребує додаткової інсталяції відповідного розширення з Marketplace. Його можна встановити безпосередньо з VS Code, знайшовши назву «C#» у перегляді розширень (Ctrl+Shift+X), як показано на рис 2.8. Інший варіант – відкрити існуючий проєкт із файлами C#, та VS Code сам запропонує встановити необхідне розширення.

Для перевірки правильності встановлення SDK .NET та відповідного розширення в Visual Studio Code створимо новий проєкт на C#, виконаємо компіляцію та запустимо програму на виконання.

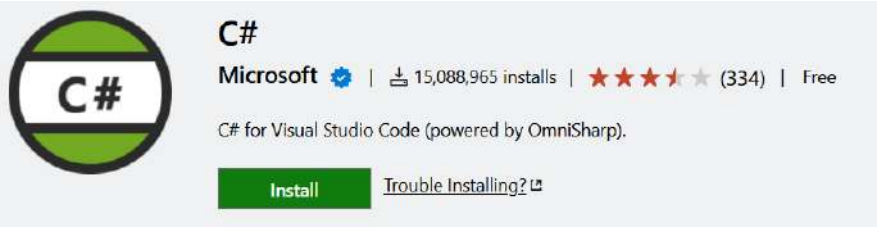


Рисунок 2.8 – Розширення C# в Visual Studio Code

Створимо проєкт консольної програми .NET під назвою ‘HelloWorld’. Для цього виконаємо наступні кроки:

1. Запустимо Visual Studio Code.

2. В головному меню обираємо «Файл» => «Відкрити папку»

3. У діалоговому вікні «Відкрити папку» створюємо папку HelloWorld та обираємо її (натискаємо «Вибрати папку»). Ім’я папки стає ім’ям проєкту та ім’ям простору імен за замовчуванням. Код, що буде додаватись пізніше, передбачає, що він буде розміщений у просторі імен проєкту – HelloWorld.

4. В діалоговому вікні про довіру авторам файлів в цій папці обираємо «Так, я довіряю авторам».

5. Відкриваємо термінал у Visual Studio Code, вибравши «Перегляд» => «Термінал» у головному меню. Термінал відкривається з командним рядком в папці HelloWorld.

6. В терміналі введемо, наприклад, таку команду:

```
dotnet new console --framework net6.0
```

Шаблон проєкту створює просту програму, яка відображає «Hello World» у вікні консолі, викликаючи метод Console.WriteLine(String) у Program.cs.

7. Замінимо вміст Program.cs таким кодом:

```
namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

Коли вперше редагується файл .cs, Visual Studio Code запропонує додати відсутні ресурси для створення та налагодження нашої програми (рис. 2.9).

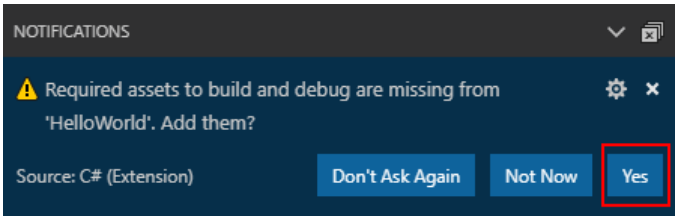


Рисунок 2.9 – Visual Studio Code пропонує додати відсутні ресурси

Обираємо «Так», і Visual Studio Code створить папку .vscode з файлами launch.json і tasks.json.

Якщо підказку не отримано або випадково була закрита, не вибравши «Так», можна виконати такі дії, щоб створити launch.json і tasks.json:

– у головному меню обираємо «Виконати» => «Додавання конфігурації».

– обираємо «.NET 5+ і .NET Core» у підказці «Вибрати середовище».

Код, що наведено вище, визначає клас Program з одним методом Main, який приймає масив String як аргумент. Main – це точка входу програми, метод, який автоматично викликається середовищем виконання під час запуску програми. Будь-які аргументи командного рядка, які надаються під час запуску програми, доступні в масиві args.

В останній версії C# нова функція під назвою оператори верхнього рівня (top-level statements) дозволяє опускати клас Program і метод Main.

Більшість існуючих програм C# не використовують оператори верхнього рівня, тому в даному прикладі ми не використовували цю нову функцію, але вони доступні у C# 10. Використовувати їх у своїх програмах чи ні, залежить від уподобань стилю.

Для запуску нашої програми необхідно в терміналі набрати таку команду:

```
dotnet run
```

В результаті, програма виводить на екран ‘Hello World!’ і закінчується (рис. 2.10).

Розширимо програму, щоб запитувати користувача ввести своє ім’я та відображати його разом із датою та часом.

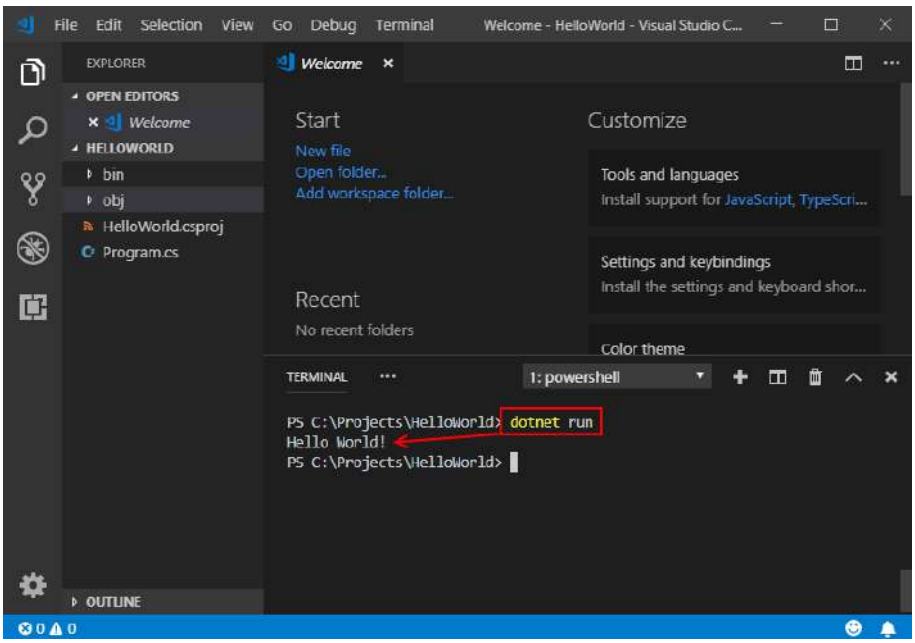


Рисунок 2.10 – Результат роботи програми «Hello world»

Для цього виконаємо такі кроки:

1. Відкриємо файл Program.cs.

2. Замінимо вміст методу Main у Program.cs, який є рядком, що викликає Console.WriteLine, на такий код:

```
Console.WriteLine("What is your name?");
var name = Console.ReadLine();
var currentDate = DateTime.Now;
Console.WriteLine($"{Environment.NewLine}Hello, {name}, on {currentDate:d} at {currentDate:t}!");
Console.Write($"{Environment.NewLine}Press any key to exit...");
Console.ReadKey(true);
```

Цей код відображає підказку у вікні консолі та чекає, поки користувач не введе рядок, а потім натисне клавішу Enter. Програма зберігає цей рядок у змінній з ім’ям «name». Далі отримуємо значення властивості DateTime.Now, яка містить поточний місцевий час, і призначає його змінній з назвою «currentDate». В результаті виконання програми ці значення відображаються у вікні консолі. Наприкінці роботи відображається підказка у вікні консолі та викликається метод Console.ReadKey(Boolean), щоб дочекатися введення даних користувачем.

NewLine – це незалежний від платформи та мови спосіб представлення розриву рядка. Альтернативами є «\n» у C# та «vbCrLf» у Visual Basic.

Знак долара ($) перед рядком дозволяє вставляти в рядок такі вирази, як імена змінних, у фігурних дужках. Значення виразу вставляється в рядок замість виразу. Цей синтаксис називають інтерпольованими рядками.

3. Зберігаємо зміну коду.

У коді Visual Studio необхідно явно зберегти зміни. На відміну від Visual Studio, зміни файлів не зберігаються автоматично під час створення та запуску програми.

4. Запускаємо програму ще раз:

```
dotnet run
```

5. Для тестування програми необхідно відповідати на запит, ввівши ім’я та натиснувши клавішу Enter (рис. 2.11).

6. Щоб вийти з програми необхідно натиснути будь-яку клавішу.

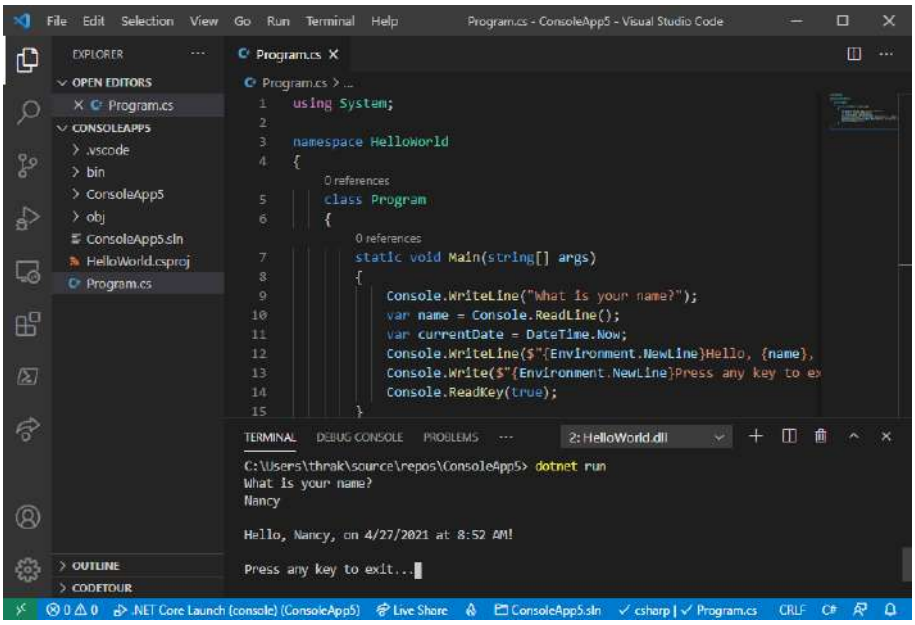


Рисунок 2.11 – Результат роботи модифікованої програми «Hello world»

2.2.3 Налагодження програми засобами Visual Studio Code

DebugDebug і Release – це вбудовані конфігурації збірки .NET. Конфігурацію збірки Debug використовують для налагодження, а конфігурацію Release – для остаточного випуску програми.

У конфігурації Debug програма компілюється з повною символьною інформацією про налагодження та без оптимізації. Оптимізація ускладнює налагодження, оскільки зв’язок між вихідним кодом і згенерованими інструкціями є складнішим. Конфігурація релізу програми не містить символьної інформації про налагодження та повністю оптимізована.

За замовчуванням параметри запуску Visual Studio Code використовують конфігурацію збірки Debug, тому не потрібно змінювати її перед налагодженням. Запустіть код Visual Studio.

Точка зупинки тимчасово перериває виконання програми до запуску рядка з точкою зупинки. Перевіримо це на практиці, взявши за приклад попередню версію програми:

1. Відкриємо файл Program.cs.

2. Встановимо точку зупинки на рядку, що відображає назву, дату та час, клацнувши на лівому полі вікна коду (рис. 2.12).

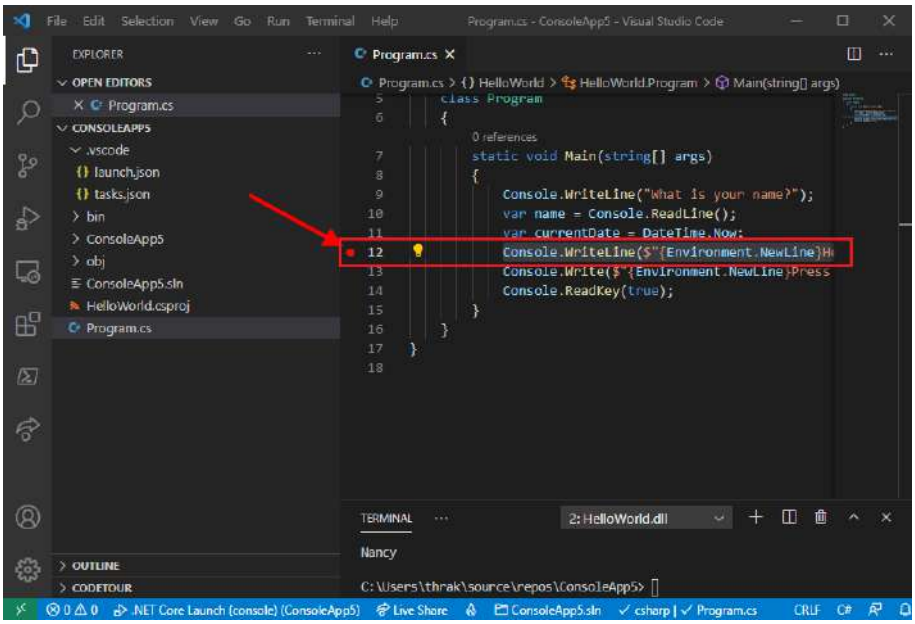


Рисунок 2.12 – Встановлення точки зупинки

Ліве поле знаходиться зліва від номерів рядків. Іншими способами встановити точку зупинки є натискання клавіші F9 або вибір команди «Виконати» => «Перемкнути точку зупинки» в головному меню, коли вибрано потрібний рядок коду. Visual Studio Code вказує рядок, на якому встановлюється точка зупинки, відображаючи червону крапку на лівому полі.

Особливості налаштування терміналу: точка зупинки знаходиться після виклику методу Console.ReadLine. Консоль налагодження не приймає вхідні дані терміналу для програми, що виконується. Щоб використовувати вихідні дані терміналу під час налагодження, можна використовувати вбудований термінал (одне з вікон Visual Studio Code) або зовнішній термінал. Для виконання більшості програм в даному навчальному посібнику використовується вбудований термінал.

Для налаштування використання вбудованого терміналу необхідно відкрити файл .vscode/launch.json та змінити значення параметра console з internalConsole на вбудований термінал.

```
‘console’: ‘integratedTerminal’,
```

3. Далі необхідно зберегти зміни.

Виконання налагодження

Для початку налагодження програми необхідно виконати наступні кроки:

1. Відкрити «Налагодження», вибравши однойменну піктограму в меню зліва (рис. 2.13).

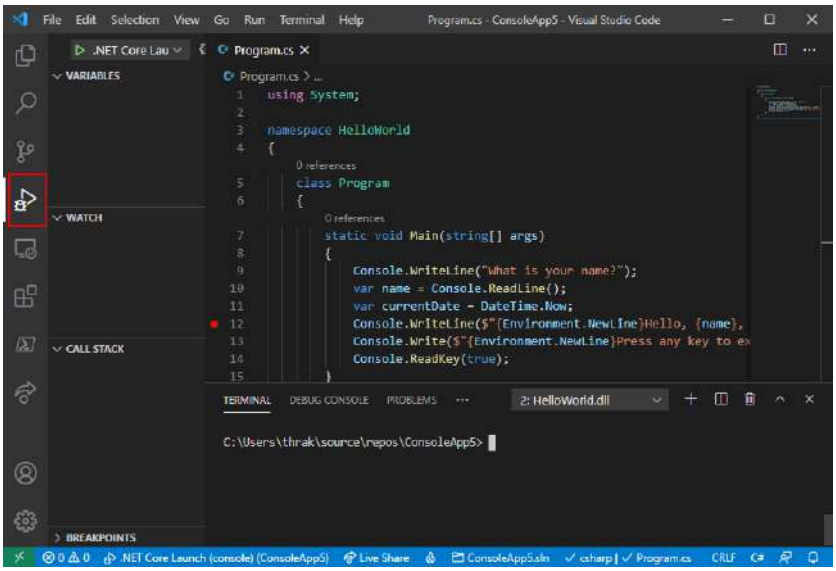


Рисунок 2.13 – Вкладка «Налагодження» в Visual Studio Code

2. Вибрати зелену стрілку у верхній частині панелі, поруч із запуском .NET Core (консоль). Запустити програму в режимі налагодження можна також

натисканням клавіші F5 або вибір «Виконати» => «Почати налагодження» з головного меню (рис. 2.14).

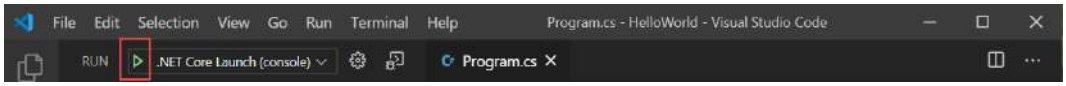


Рисунок 2.14 – Запуск програми в режимі налагодження

3. Вибрати вкладку «Термінал», щоб побачити запит «Як вас звати?», що відображає програма, перш ніж чекати відповіді (рис. 2.15).

4. Ввести рядок у вікні терміналу у відповідь на запит імені, а потім натиснути кнопку Enter.

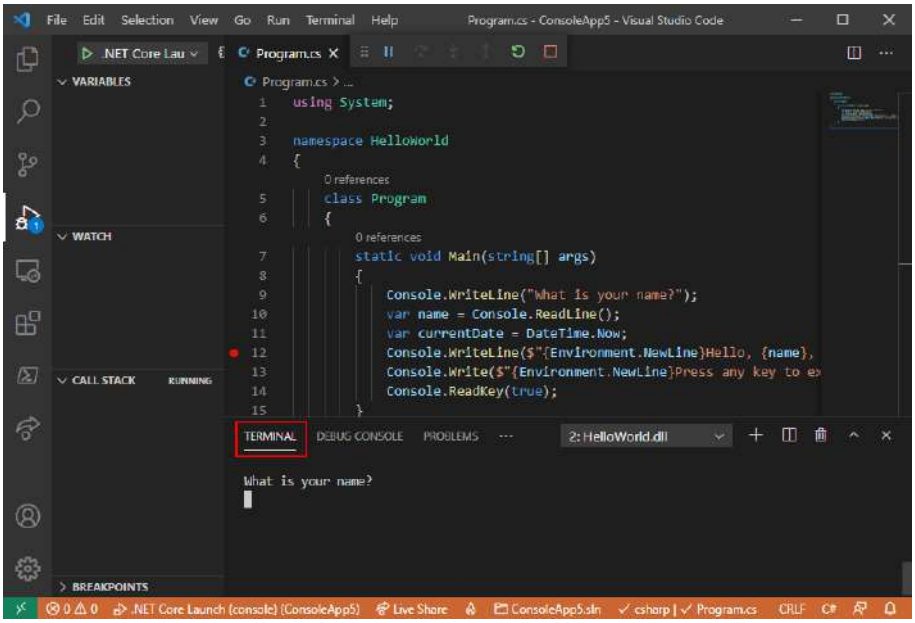


Рисунок 2.15 – Вибрати вкладку «Термінал»

Виконання програми припиняється, коли вона досягне точки зупинки перед запуском методу Console.WriteLine. У розділі вибору вікна «Локальні» => «Змінні» – відображаються значення змінних, які визначені в поточному методі, який запущений (рис. 2.16).

Робота з вікном консолі налагодження

Вікно консолі налагодження дозволяє взаємодіяти з програмою, яку в даний час налагоджують. Програміст можете змінити значення змінних, щоб побачити, як це впливає на його програму.

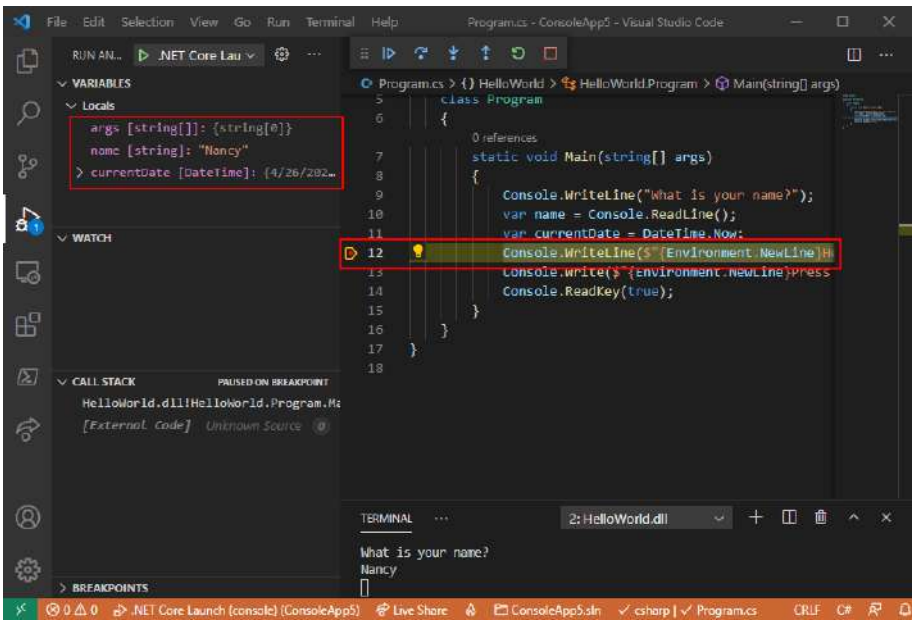


Рисунок 2.16 – Відображення значення змінних в режимі налагодження

Порядок роботи з вікном налагодження наступний:

1. Вибрати вкладку «Консоль налагодження».
2. Ввести:

```
name = "Test Name"
```

у командному інтерфейсі внизу вікна консолі налагодження та натиснути клавішу Enter (рис. 2.17).

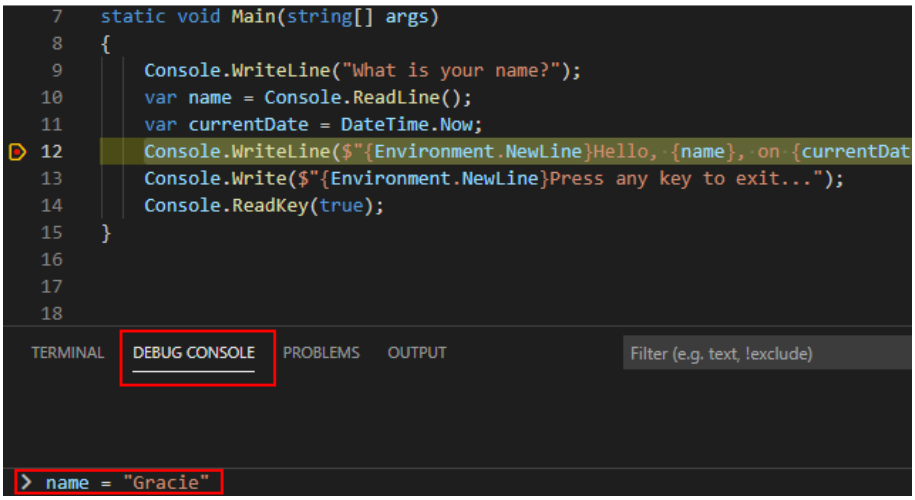


Рисунок 2.17 – Введення значення змінної

3. Ввести:

```
currentDate = DateTime.Parse('2019-11-16T17:25:00Z').ToUniversalTime()
```

у нижній частині вікна консолі налагодження та натиснути клавішу Enter.

У вікні «Змінні» відображаються нові значення змінних імені (name) та поточної дати (currentDate).

4. Продовжити виконання програми натисканням кнопки «Продовжити» на панелі інструментів. Інший спосіб– натиснути клавішу F5 (рис. 2.18).

Рисунок 2.18 – Продовження виконання програми

5. Знову вибрати вкладку «Термінал». Значення, що відображаються у вікні консолі, відповідають змінам, які внесли в консолі налагодження (рис. 2.19).

6. Натиснути будь-яку клавішу, щоб вийти з програми та припинити налагодження.

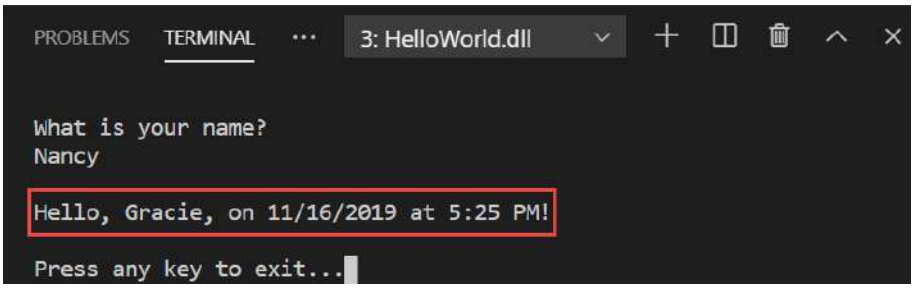


Рисунок 2.19 – Виведення змінених значень в термінал

Встановлення умовної точки зупинки

Написана програма відображає рядок, який вводить користувач, але що станеться, якщо користувач нічого не введе? Це можна перевірити за допомогою корисної функції налагодження, яка називається умовною точкою зупинки.

1. Для використання даної можливості необхідно клацнути правою кнопкою миші на червоній крапці, яка представляє точку зупинки. У контекстному меню вибираємо «Редагувати точку зупинки», щоб відкрити діалогове вікно, яке дозволяє ввести умовний вираз (рис. 2.20).

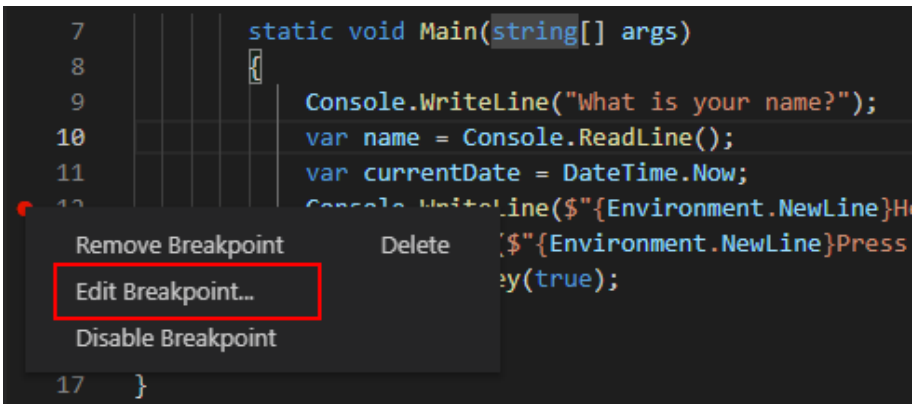


Рисунок 2.20 – Редагування точки зупинки

2. Вибрати «Вираз» у випадному меню, ввести наступний умовний вираз і натиснути Enter (рис. 2.21):

```
String.IsNullOrEmpty(назва)
```

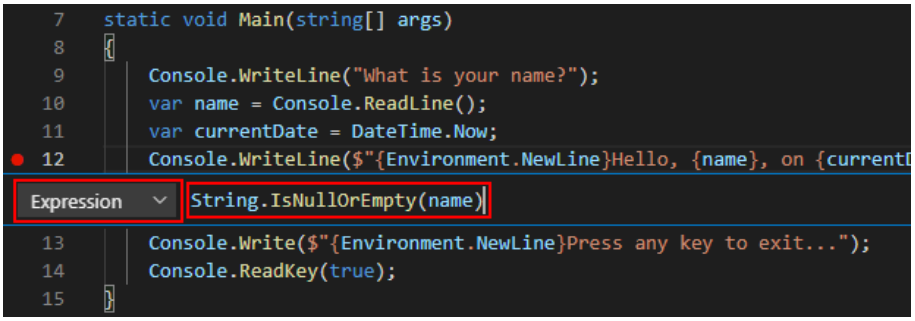


Рисунок 2.21 – Додавання виразу до точки зупинки

Кожного разу, коли досягається точка зупинки, debugger викликає метод String.IsNullOrEmpty(name), і він зупиняється на цьому рядку, лише якщо виклик методу повертає true.

Замість умовного виразу можна вказати кількість звернень, яка перериває виконання програми до того, як оператор буде виконано певну кількість разів. Інший варіант – вказати умову фільтрації, яка перериває виконання програми на основі таких атрибутів, як ідентифікатор потоку, ім’я процесу або ім’я потоку.

3. Запустимо програму з налагодженням, натиснувши кнопку F5.

4. Коли буде запропоновано ввести своє ім’я, потрібно натиснути клавішу Enter на вкладці «Термінал».

Оскільки умова, яку ми вказали (name має значення null, або String.Empty), була виконана, виконання програми припиняється, коли вона досягає точки

зупинки, тобто перед запуском методу Console.WriteLine. Вікно «Змінні» показує, що значенням змінної імені є "", або String.Empty.

5. Підтвердимо, що значення змінної є порожнім рядком, ввівши наступний оператор у командному рядку консолі налагодження та натиснувши Enter.

```
name == String.Empty
```

Результат становить true.

6. Щоб продовжити виконання програми необхідно натиснути кнопку «Продовжити» на панелі інструментів.

7. Виберімо вкладку «Термінал» і натиснемо будь-яку клавішу, щоб вийти з програми та припинити налагодження.

8. Очистимо точку зупинки, клацнувши по крапці на лівому полі вікна коду. Іншими способами очистити точку зупинки є натискання клавіші F9 або вибір «Виконати» => «Перемикання точки зупинки» у меню, коли вибрано рядок коду.

9. Якщо було отримане попередження про те, що умову точки зупинки буде втрачено, виберіть «Видалити точку зупинки».

Покрокове виконання програми

Visual Studio Code також дає змогу крок за кроком проходити програму та контролювати її виконання. Як правило, програміст встановлює точку зупинки і слідкує за ходом програми для невеликої частини коду програми. Виконаємо для прикладу покрокове налагодження нашої програми:

1. Встановимо точку зупинки на фігурній дужці, що відкривається, методу Main.

2. Натиснімо кнопку F5, щоб почати налагодження. Visual Studio Code виділяє рядок точки зупинки. На цьому етапі вікно змінних показує, що масив args порожній, а name і currentDate мають значення за замовчуванням.

3. Виберемо «Run => Step Into» або натиснемо на кнопку F11 (рис. 2.22). В даному випадку Visual Studio Code виділяє наступний рядок.

4. Знову виберемо «Run => Step Into» або натиснемо F11. Visual Studio Code запускає Console.WriteLine для запиту імені та виділяє наступний рядок

виконання. Наступний рядок – Console.ReadLine для name. Вікно «Змінні» не змінюється, а вкладка «Термінал» показує запит «Як вас звати?».

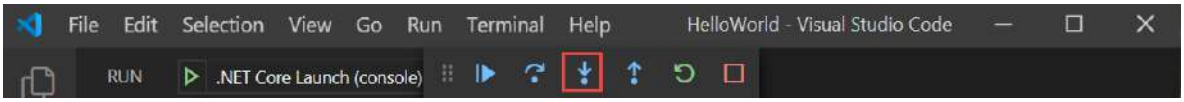


Рисунок 2.22 – Вибір режиму «Run => Step Into»

5. Знову виберемо «Run => Step Into» або натиснемо F11. Visual Studio виділяє рядок присвоєння значення змінній name. У вікні змінних показано, що ім’я все ще є нульовим.

6. Надамо відповідь на запит, ввівши рядок на вкладці «Термінал» і натиснувши Enter. На вкладці «Термінал» може не відображатися рядок, який вводиться під час його введення, але метод Console.ReadLine фіксує введені дані.

7. Обираємо «Run => Step Into» або натиснемо F11.

Visual Studio Code підкреслює рядок присвоєння змінній значення currentDate. У вікні змінних показано значення, яке повернуто викликом методу Console.ReadLine. На вкладці «Термінал» відображається рядок, який ввели в командному рядку.

8. Обираємо «Run => Step Into» або натиснемо F11. У вікні «Змінні» показано значення змінної currentDate після присвоєння властивості DateTime.Now.

9. Обираємо «Run => Step Into» або натиснемо F11. Visual Studio Code викликає метод Console.WriteLine(String, Object, Object). У вікні консолі відображається відформатований рядок.

10. Обираємо «Run => Step Out» або натиснемо Shift + F11 (рис. 2.23).

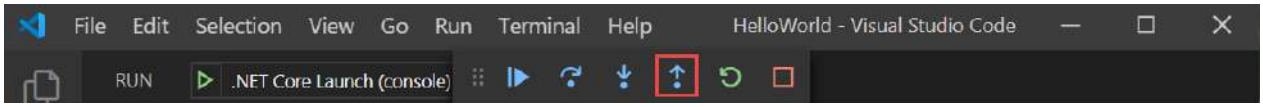


Рисунок 2.23 – Вибір режиму «Run => Step Out»

11. Обираємо вкладку «Термінал». Термінал відображає "Press any key to exit..."

12. Натискаємо будь-яку клавішу, щоб вийти з програми.

2.2.4 Створення першої програми з використанням фреймворку Avalonia

Для ініціалізації проєкту необхідно скористатися шаблонами .NET програм для Авалонії. Для цього потрібно буде клонувати репозиторій із шаблонами, а потім встановити завантажені шаблони. Робиться це за допомогою команди:

```
dotnet new -i Avalonia.Templates
```

В результаті отримаємо сформований набір шаблонів для подальшого створення проєктів (рис. 2.24).

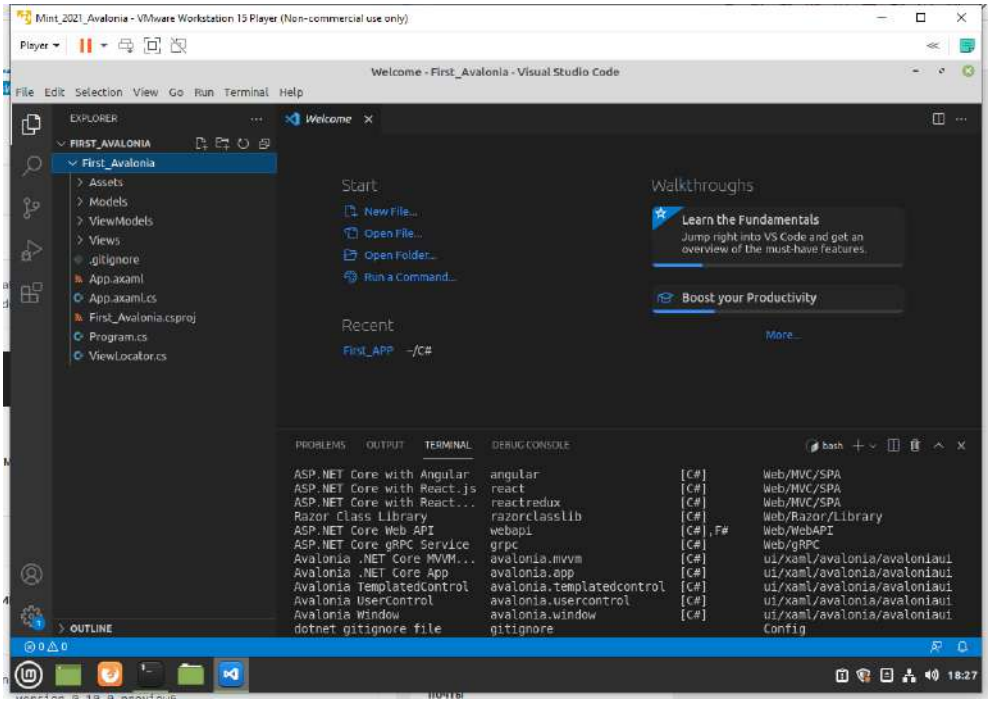


Рисунок 2.24 – Ініціалізація проєкту

Тепер, коли встановлені шаблони, можемо створити новий проєкт на основі MVVM шаблону Авалонії:

```
dotnet new avalonia.mvvm -o ACalc
```

Структура проєкту, що була автоматично згенерована, має вигляд, поданий на рис. 2.25. У папці Assets зберігаються ресурси, які ми використовуємо в даному проєкті. На даний момент там лежить лого Авалонії, що використовується як іконка додатка.

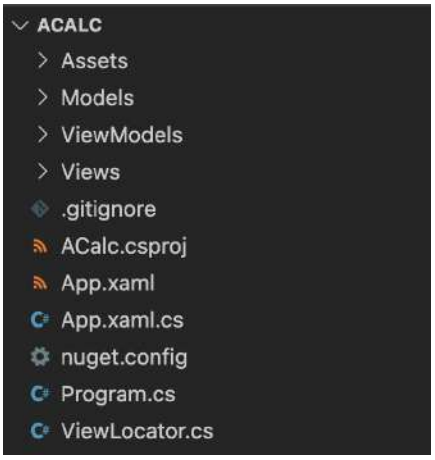


Рисунок 2.25 – Структура проєкту Avalonia

В папку Model ми будемо складати всі загальні моделі, які використовуються у нашому додатку. На даний момент вона порожня.

Папка ViewModels призначена для зберігання логіки, яка буде використовуватись у кожному вікні. Прямо зараз у цій папці зберігається ViewModel головного вікна та базовий клас для всіх ViewModel.

У папці Views зберігається розмітка вікон (а також code behind файл, в який хоч і можна покласти логіку, але краще для цього використовувати ViewModel). На даний момент у нас є лише головне вікно.

App.xaml – загальний конфіг програми. Незважаючи на те, що він має вигляд, як ще одне вікно, насправді цей файл слугує для завдання загальних налаштувань програми.

ViewLocator використовується для створення кастомних контролів.

Запуск програми відбувається командою:

```
dotnet run
```

2.3 Використання MS Visual Studio в якості IDE

Для використання MS Visual Studio в якості IDE для розробки програм Avalonia необхідно встановити спеціальне розширення.

Розширення Avalonia для Visual Studio включає конструктор XAML, який можна використовувати для відображення попереднього перегляду XAML під час його написання (рис. 2.26).

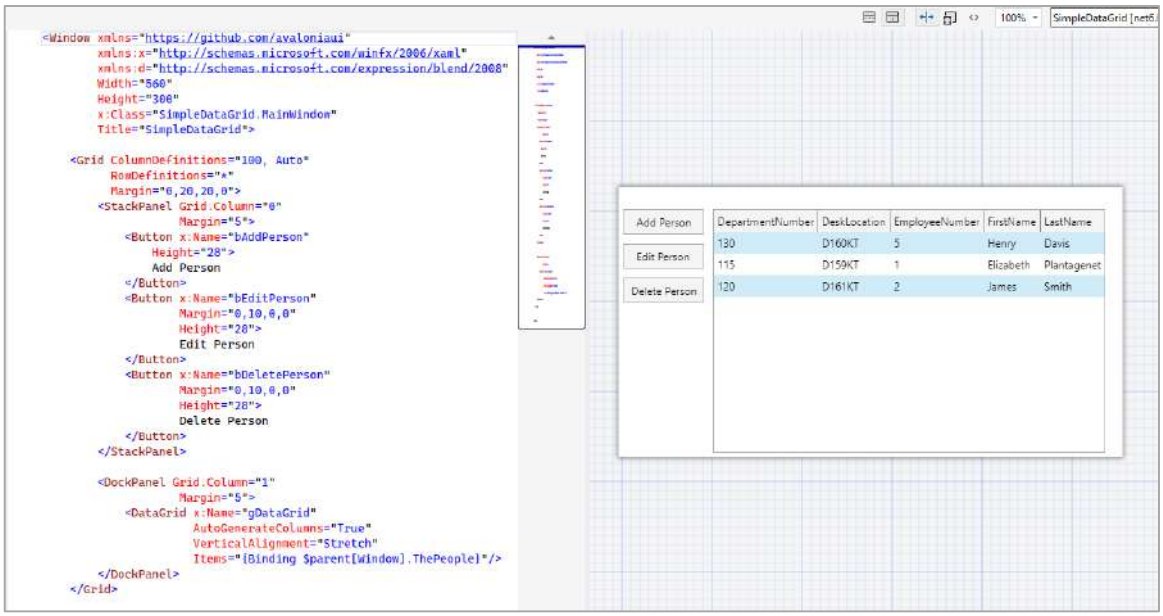


Рисунок 2.26 – Конструктор XAML в MS Visual Studio

Версія розширення відрізняється для різних версій MS Visual Studio. Якщо використовується VS2019 або VS2017, то потрібно встановити розширення для старіших версій.

Розширення Avalonia для Visual Studio містить шаблони проєктів і елементів керування, які можуть використовуватись, наприклад, на початку роботи.

Для завантаження розширення Avalonia для Visual Studio версій VS2019 або VS2017 потрібно перейти за посиланням:

`https://marketplace.visualstudio.com/items?itemName=AvaloniaTeam.AvaloniaforVisualStudio`

Якщо встановлена версія VS2022, потрібно встановити розширення для новішої версії:

`https://marketplace.visualstudio.com/items?itemName=AvaloniaTeam.AvaloniaVS`

На рис. 2.27 подано приклад сторінки завантаження розширення Avalonia для MS Visual Studio 2022.

Рисунок 2.27 – Приклад сторінки завантаження розширення Avalonia для MS Visual Studio 2022

Розширення містить шаблони для різних проєктів Visual Studio, пов'язаних з Avalonia, типи файлів Avalonia і Intellisense для файлів XAML Avalonia (які дещо відрізняються від файлів XAML WPF).

Після завершення встановлення розширення «Avalonia for Visual Studio» необхідно запустити Visual Studio та перевірити наявність доданого розширення (рис. 2.28).

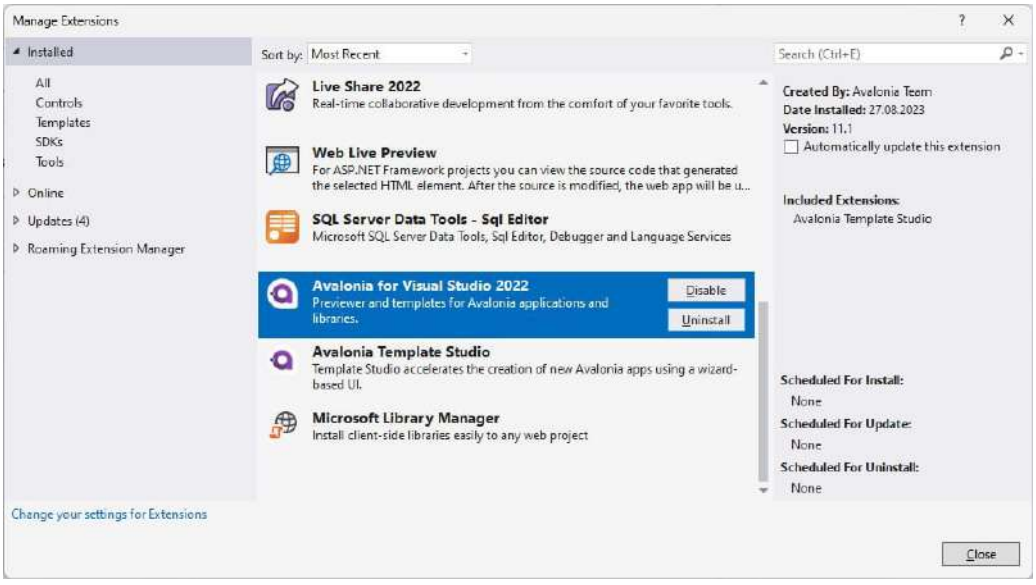


Рисунок 2.28 – Перевірка розширення «Avalonia for Visual Studio»

Для ініціалізації проєкту необхідно скористатися шаблонами .NET програм для Авалонії. Для цього потрібно буде клонувати репозиторій із шаблонами, а потім встановити завантажені шаблони. Робиться це за допомогою команди:

```
dotnet new -i Avalonia.Templates
```

Для створення нового проєкту необхідно завантажити Visual Studio та обрати тип проєкту «Avalonia Application» (рис. 2.29):

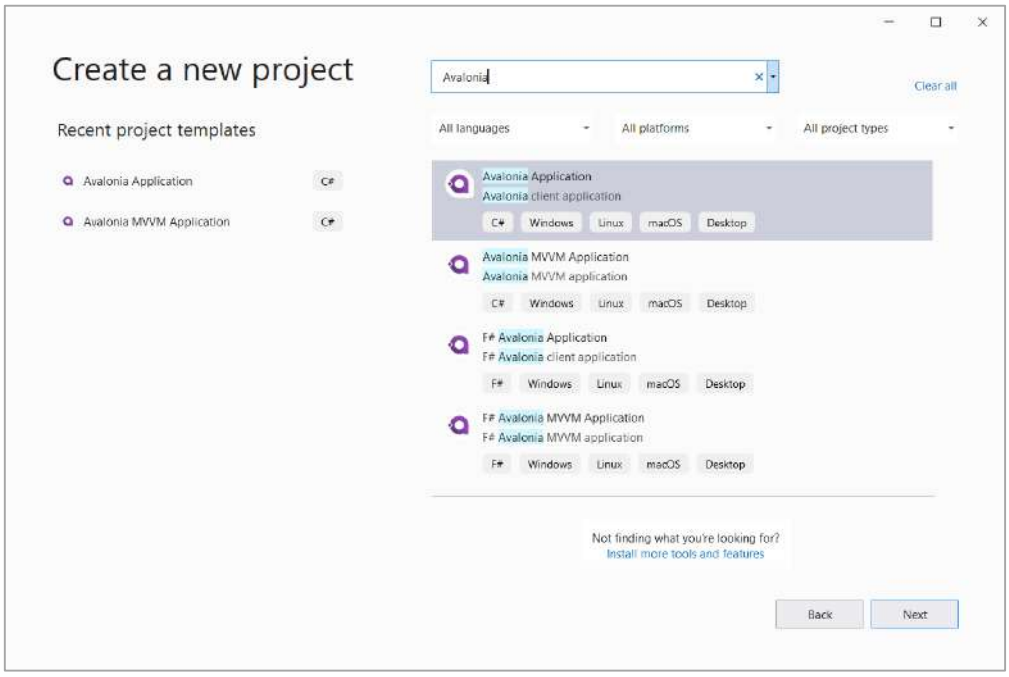


Рисунок 2.29 – Створення проєкту Avalonia

Натисніть кнопку «Далі», виберіть розташування та назву проєкту та натисніть кнопку «Створити» (рис. 2.30):

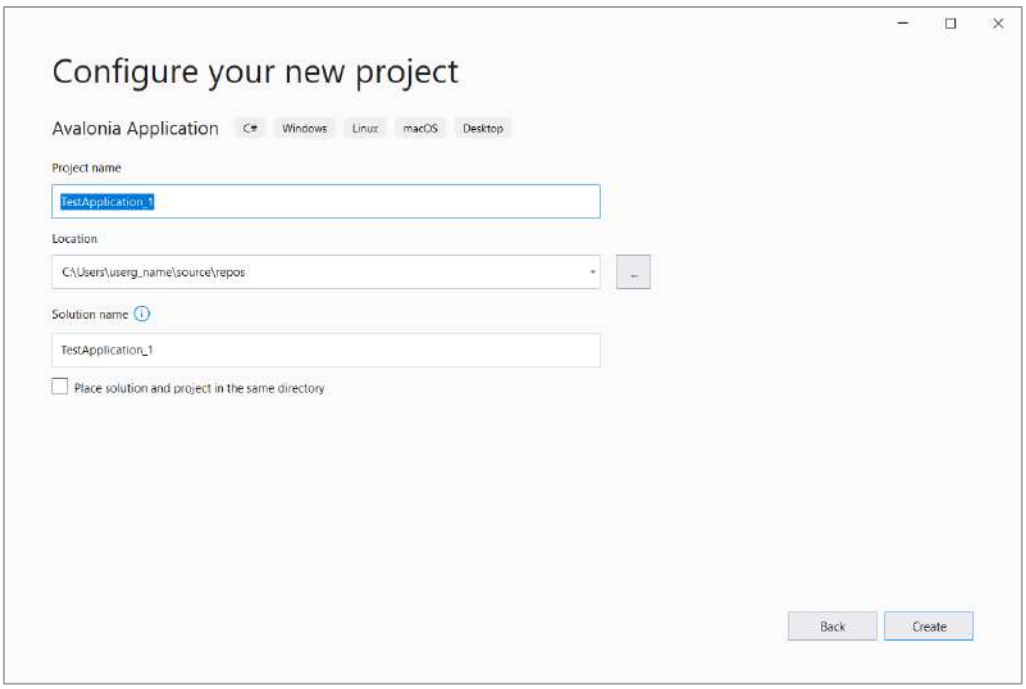


Рисунок 2.30 – Налаштування нового проєкту

Створений проєкт матиме залежності від трьох пакетів Avalonia (рис. 2.31).

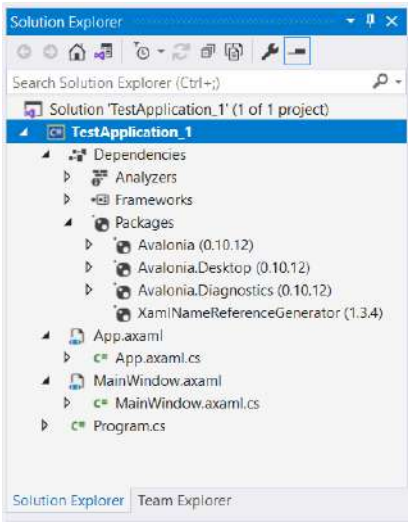


Рисунок 2.31 – Структура проєкту

На рис. 2.31 можна бачити три пакети Avalonia:

– Avalonia;

– Avalonia.Desktop;

– Avalonia.Diagnostics.

Також створюється п’ять файлів, що містять код:

– App.axaml;

– App.axaml.cs;

– MainWindow.axaml;

– MainWindow.axaml.cs;

– Program.cs.

Файли з розширеннями «.axaml» – це файли XAML, перейменовані на «.axaml», щоб відрізняти їх від файлів «.xaml» WPF/UWP. Синтаксис XAML Avalonia дуже схожий на синтаксис WPF XAML. Особливості стосуються, наприклад, селекторів стилів.

Серед п’яти описаних вище файлів, швидше за все, вам доведеться змінити файли MainWindow.axaml і MainWindow.axaml.cs, а потім, можливо, трохи змінити файли App.axaml і App.axaml.cs. Файл Program.cs, ймовірно за все, змінювати не доведеться.

Для перевірки роботи розширення можна запустити проєкт з порожнім вікном як воно є. В результаті першого запуску отримаємо такий вигляд програми (рис. 2.32).

Рисунок 2.32 – Перший запуск програми

Виконаємо просту модифікацію програми. Для цього відкрийте файл MainWindow.xaml і замініть його вміст (який за замовчуванням складається з тексту «Welcome to Avalonia!!») на такий код:

```
<Button x:Name="CloseWindowButton"
        Content="Close Window"
        HorizontalAlignment="Center"
        VerticalAlignment="Center"
        Padding="10,5"/>
```

На рис. 2.33 подано повний текст програми, яка була модифікована.

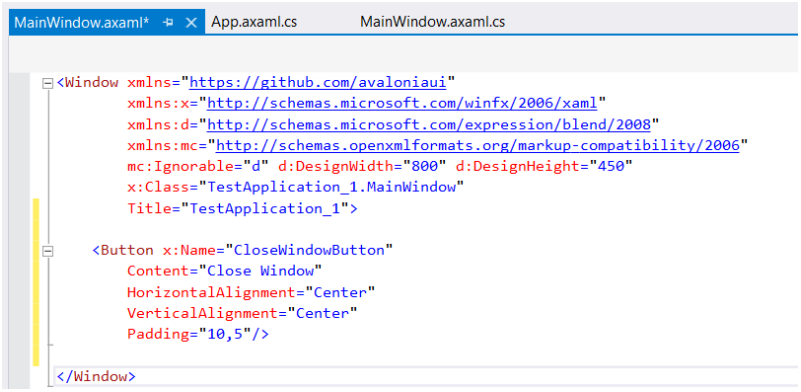


Рисунок 2.33 – Повний текст програми, яка була модифікована

Після запуску програми отримаємо наступний вигляд додатку (рис. 2.34). Програма містить посередині вікна кнопку з горизонтальним і вертикальним вирівнюванням. Текст «Закрити вікно» написано посередині кнопки (вміст кнопки), а на полях від тексту кнопки з боків є 10 пікселів праворуч і ліворуч, та

5 пікселів угорі та внизу. Така поведінка забезпечується властивістю Padding у вихідному коді (рис. 2.34).

Рисунок 2.34 – Зовнішній вигляд модифікованої програми

Наразі програма тільки відображає інформацію, але нічого не виконує. Якщо натиснути кнопку «Закрити вікно», нічого не станеться. Нам потрібно спробувати підключити подію натискання кнопки до дії, яка закриває вікно. У цьому першому прикладі ми збираємося використовувати найпростіший, але водночас і найгірший спосіб досягнення цієї мети – так званий «код позаду» ("code behind").

Розглянемо файл MainWindow.xaml.cs, що містить код C# для файлу MainWindow.xaml:

```
using Avalonia.Controls;
namespace TestApplication_1
{
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }
    }
}
```

Додамо до файлу з вихідним кодом рядок, що описує кнопку. Назвемо її «CloseWindowButton». У WPF створився би відповідний член класу. Avalonia, досі не містить цю функцію, але ми можемо легко знайти кнопку, додавши наступний рядок:

```
var button = this.FindControl<Button>("CloseWindowButton")
```

відразу після рядка

```
InitializeComponent();
```

що викликається в конструкторі. Далі можемо додати обробник до події Click кнопки:

```
button.Click += Button_Click;
```

Нарешті, в обробнику ми можемо викликати метод Close() в класі window:

```
private void Button_Click(object? sender, RoutedEventArgs e)
{
    this.Close();
}
```

Після додавання такої конструкції коду може з’явитись повідомлення про помилку. Це обумовлено тим, що у нас в коді немає посилання на відповідну бібліотеку. Для автоматичного додавання посилання необхідно встановити курсор на рядок з помилкою та натиснути комбінацію кнопок «Alt + Enter». На екрані з’явиться підказка з варіантами вирішення даної проблеми (рис. 2.35).

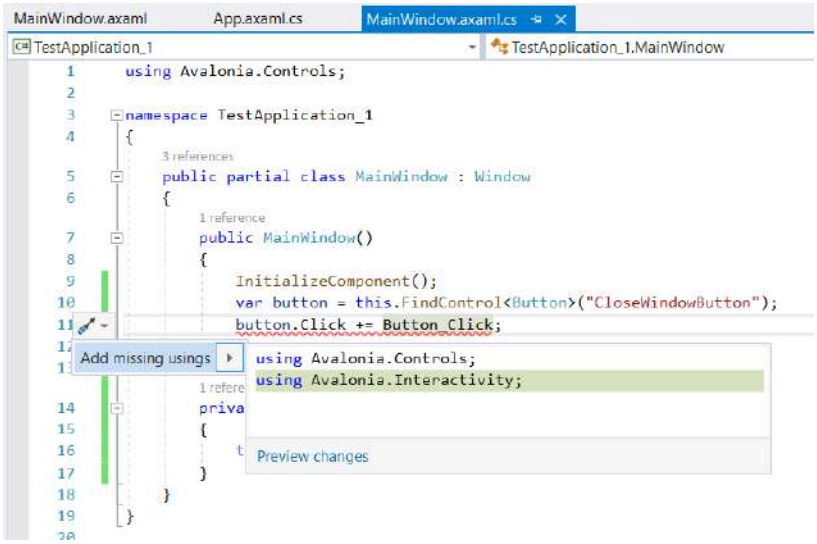


Рисунок 2.35 – Додавання посилання на потрібну бібліотеку

Після обирання варіанту «using Avalonia.Interactivity;», в коді з’явиться відповідний рядок, а повідомлення про помилку зникне (рис. 2.36).

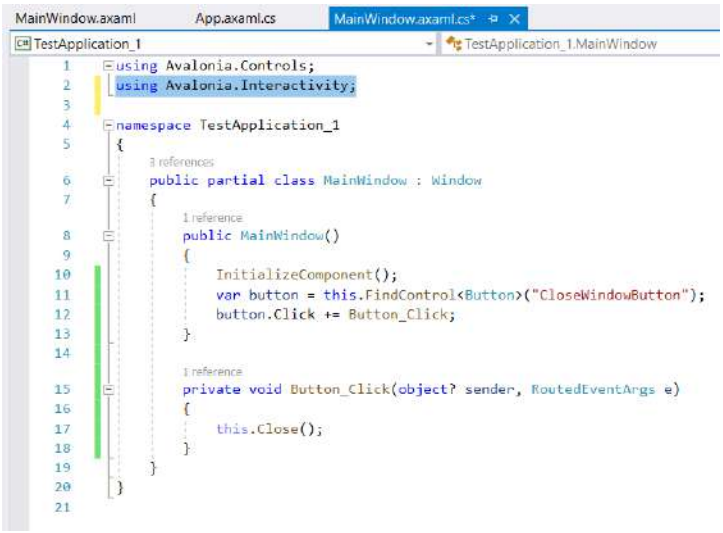


Рисунок 2.36 – Додавання «using Avalonia.Interactivity;» до списку посилань

В разі запуску програми на виконання, можна бачити, що кнопка «Закрити вікно» виконує свою функцію – закриває вікно після натискання на неї.

2.4 Контрольні запитання та завдання

1. Опишіть процес встановлення Visual Studio Code на ОС Linux.
2. В чому полягають особливості інтерфейсу користувача Visual Studio Code?
3. Опишіть процес розгортання SDK .NET на платформі ОС Linux?
4. Яким чином відбувається додавання мови програмування C# до Visual Studio Code?
5. В чому полягає процес налагодження програми засобами Visual Studio Code?
6. Як виконати налаштування вбудованого терміналу?
7. Як виконується налагодження програми засовами Visual Studio Code?
8. Яким чином організований процес покрокового виконання програми?
9. Опишіть процес створення програми з використанням фреймворку Avalonia.
10. Охарактеризуйте структуру проєкту Avalonia.
11. В чому полягає використання MS Visual Studio в якості IDE?

3 СТВОРЕННЯ ІНТЕРФЕЙСУ КОРИСТУВАЧА ЗАСОБАМИ AVALONIA

3.1 Класифікація елементів керування

Для створення інтерфейсу користувача використовується великій набір візуальних та невізуальних компонентів. Для зручності користування цими компонентами виконаємо їх класифікацію за призначенням.

Всі компоненти можна розділити на такі великі групи:

– Layout controls (компоненти керування макетом);

– Buttons (кнопки);

– Data display controls (компоненти керування відображенням даних);

– Text display and editing (компоненти відображення та редагування тексту);

– Value selection (компоненти вибору значення);

– Displaying images (компоненти відображення зображень);

– Date and time controls (компоненти керування датою та часом);

– Menus (компоненти створення меню).

Розглянемо першу групу компонентів, що використовуються для створення макету візуального інтерфейсу та керують розташуванням візуальних елементів на екранній формі (Layout controls). Елементи керування макетом надають розробнику можливість упорядкувати дочірні елементи керування відповідно до певних правил.

Серед таких компонентів можна виділити:

– *Border*: елемент керування, який прикрашає дочірню рамку та фон, його також можна використовувати для відображення заокруглених кутів, встановивши відповідну властивість;

– *Canvas*: панель, яка відображає дочірні елементи керування в довільних місцях. Елемент керування Canvas – це панель, яка розміщує свої дочірні елементи за явними координатами. Розробник вказуєте позиції окремих дочірніх елементів, встановлюючи прикріплені властивості Canvas.Left і Canvas.Top для кожного елемента. Об’єкти на Canvas можуть перекриватися, коли один об’єкт розташований поверх іншого;

– *DockPanel*: панель, яка розташовує дочірні елементи вгорі, внизу, ліворуч, праворуч або в центрі.

– *Expander*: елемент керування, який дозволяє розгортатись для відображення вкладеного вмісту;

– *Grid*: область гнучкої сітки, яка складається зі стовпців і рядків;

– *GridSplitter*: елемент, що перерозподіляє простір між стовпцями або рядками елемента керування Grid;

– *Panel*: базовий клас для елементів керування, які можуть містити кілька дочірніх елементів;

– *RelativePanel*: визначає область, у якій можна розташувати та вирівняти дочірні об’єкти відносно один одного або батьківської панелі;

– *ScrollBar*: елемент керування, який дозволяє генерувати подію прокручування, що може використовуватись іншими компонентами;

– *ScrollViewer*: елемент керування, що прокручує свій вміст, якщо вміст перевищує доступний простір;

– *SplitView*: елемент керування двома видами: панель, що згортається, і область для вмісту;

– *StackPanel*: панель, яка розташовує дочірні елементи горизонтально або вертикально, один біля одного;

– *UniformGrid*: панель із однаковими розмірами стовпців і рядків;

– *WrapPanel*: елемент керування, що розміщує дочірні елементи в послідовних позиціях зліва направо, розбиваючи вміст на наступний рядок на краю вікна, що його містить.

Наступна група компонентів відноситься до категорії кнопок (Buttons). Компоненти цієї групи мають спільну властивість – вони всі реагують на натискання вказівника (миші або сенсорного керування). Представниками цієї групи є:

– *Button*: кнопка, що використовується для визначення реакції користувача, та реагує на натискання, викликаючи декілька подій, наприклад, Click або PointerDown;

– *RepeatButton*: елемент керування, який повторно викликає подію клацання, коли його натискають і утримують;

– *RadioButton*: кнопка, яка дозволяє користувачеві вибрати один параметр із групи параметрів;

– *ToggleButton*: елемент керування, який користувач може вибрати (поставити прапорець) або зняти (зняти прапорець);

– *ButtonSpinner*: елемент керування, який містить дві кнопки, та може використовуватись для збільшення або зменшення будь-якої величини;

– *SplitButton*: функціонує як кнопка з основною та другорядною частинами, які можна натискати окремо. Основна частина поводиться як звичайна кнопка, а другорядна частина відкриває спливаюче вікно з додатковими діями;

– *ToggleSplitButton*: функціонує як кнопка перемикання з основною та другорядною частинами, які можна натискати окремо. Основна частина працює як звичайна кнопка-перемикач, а допоміжна частина відкриває спливаюче меню з додатковими діями.

Компоненти керування відображенням даних (Data display controls) допомагають відображати дані у вигляді таблиці чи списку. Використовуються для роботи, наприклад, з базами даних. Серед таких компонентів можна виділити:

– *DataGrid*: компонент керування відображає дані у сітці, що налаштовується, у вигляді таблиці;

– *ItemsControl*: компонент керування, що відображає колекцію елементів. Елемент керування ItemsControl є базовим класом для елементів керування, які відображають кілька елементів, наприклад ListBox;

– *ItemsRepeater*: елемент керування для відображення колекції на основі даних, який включає гнучку систему розміщення дочірніх елементів та видів, що налаштовуються;

– *ListBox*: компонент керування, що містить в собі окремі елементи, які можна обирати.

Компоненти відображення та редагування тексту (Text display and editing) містять наступний набір елементів:

– *AutoCompleteBox*: елемент керування, який надає текстове поле для введення даних користувача та випадний список, який містить можливі збіги на основі введення в текстовому полі;

– *TextBlock*: елемент керування, який відображає блок тексту, що не редагується;

– *TextBox*: елемент керування, який можна використовувати для відображення або редагування неформатованого тексту;

– *MaskedTextBox*: елемент керування, який можна використовувати для відображення або редагування текстових даних. Він використовує маску для розрізнення належного та неправильного формату введення користувача.

Компоненти обирання значення (Value selection) представлені наступним набором віджетів:

– *CheckBox*: елемент керування, який дозволяє користувачеві обрати потрібний параметр. Зазвичай він використовується для відображення логічної опції, де вибір позначено або не позначено, але він також підтримує режим із трьома станами, коли вибір позначено, невизначено або не позначено;

– *ComboBox*: елемент керування з випадним списком. Список містить набір опцій, які користувач може обрати в процесі роботи;

– *Slider*: елемент керування, який дозволяє користувачеві вибирати з діапазону значень, переміщаючи покажчик по доріжці.

Компоненти для відображення зображень (Displaying images) – це віджети, які відображають растрові або векторні зображення. Їх можна поділити на:

– *DrawingImage*: компонент, що відображає векторне зображення;

– *Image*: компонент, що відображає растрове зображення.

Компоненти керування датою та часом (Date and time controls) – це набір елементів керування, який дає можливість відображати та вибирати дату, час або відображати календар. Серед таких компонентів можна виділити наступні:

– *Calendar*: елемент керування, який дозволяє користувачеві вибрати дату за допомогою візуального відображення календаря;

– *CalendarDatePicker*: елемент керування для вибору дат із випадного меню календаря;

– *DatePicker*: елемент керування, який дозволяє користувачеві вибрати потрібну дату;

– *TimePicker*: елемент керування, який дозволяє користувачеві вибрати час.

Компоненти для створення меню (Menus) містять два представника:

– *ContextMenu*: контекстне меню, приєднане до елемента керування. Даний елемент керування можна застосувати до будь-якого іншого елемента керування, щоб надати меню, специфічне для саме цього елемента. Зазвичай, контекстне меню з’являється при натискається правою кнопкою миші на обраному компоненті.

– *Menu*: елемент керування меню верхнього рівня. Меню зазвичай розміщується в DockPanel у вікні, прикріпленому до верхньої частини вікна.

3.2 Основні віджети для організації діалогу з користувачем

3.2.1 TextBlock

TextBlock є єдиним елементом керування з-поміж описаних у розділі елементів керування, який не можна повторно шаблонувати – він є похідним від Control (а не від класу TemplatedControl). Таким чином, це скоріше примітивний, а не складений елемент керування в сенсі WPF. Це один з найважливіших будівельних блоків.

TextBlock представляє простий текст. Його найважливішою властивістю є «Text» типу «string» в C#. «Text» містить текст, який потрібно відобразити як TextBlock.

TextBlock має багато властивостей, які дозволяють налаштовувати текст, зокрема:

– *Foreground* – колір тексту;

– *FontSize* – розмір тексту;

– *FontFamily* – вказує назву шрифту;

– *FontWeight* – зазвичай перемикається між звичайним і напівжирним – для напівжирного тексту;

– *TextWrapping* – вказує, чи слід переносити текст до кількох рядків;

– *TextTrimming* – вказує, чи мають відображатися символи з крапками (‘...’), якщо частина тексту невидима – текст розтягується за межі розміру елемента керування.

Багато з перерахованих вище властивостей також застосовуються до інших елементів керування, які можуть мати текст, наприклад, кнопки, меню, списки тощо.

Для створення текстового блоку використовуються простий код XAML, наприклад:

```
<TextBlock Text="Hello World!"/>
```

На рис. 3.1 показано приклад використання віджету TextBlock.

MainWindow.axaml MainWindow.axaml.cs

```
<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"
        mc:Ignorable="d" d:DesignWidth="800" d:DesignHeight="450"
        x:Class="TestApplication_1.MainWindow"
        Title="TestApplication_1">

    <TextBlock Text="Hello World!"/>

</Window>
```

Рисунок 3.1 – Приклад використання віджету TextBlock

Після запуску програми вона буде мати наступний вигляд (рис. 3.2).

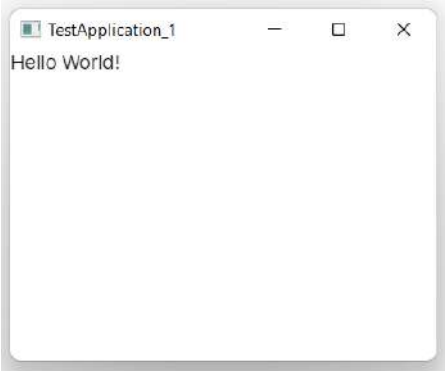


Рисунок 3.2 – Запуск програми з віджетом TextBlock

Можна бачити, що віджет відображає заданий текст в верхньому куті вікна програми.

3.2.2 Віджет TextBox

TextBox використовується для введення тексту у формі діалогу з користувачем. Якщо ви введете щось у TextBox, його властивість Text буде оновлено разом із введеним текстом. В наступному прикладі використовується особливість Avalonia реалізації технології зв’язування (binding) безпосередньо в інтерфейсі користувача файлу XAML:

```
<Grid Grid.Row="1"
      HorizontalAlignment="Center"
      VerticalAlignment="Center">
      <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
      </Grid.RowDefinitions>
```

```
        <TextBox x:Name="TheTextBox"
                 Width="150"
                 Height="27"
                 HorizontalAlignment="Left"
                 Grid.Row="0"/>
        <TextBlock  Padding="10,5"
                 Text="{Binding Path=Text, ElementName=TheTextBox}"
                 Grid.Row="1"
                 Height="25"/>
    </Grid>
```

Приклад роботи програми подано на рис. 3.3.

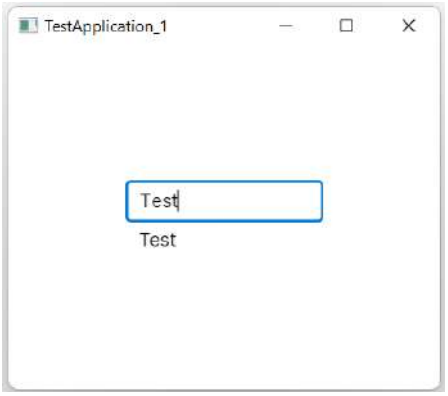


Рисунок 3.3 – Приклад роботи віджету TextBox

В процесі роботи програми користувач вводить текст у відповідний візуальний компонент. Все, що вводиться, одночасно відображається в іншому компоненті TextBlock. Зв’язок між віджетами реалізовано у властивостях TextBlock в рядку:

```
Text="{Binding Path=Text, ElementName=TheTextBox}"
```

Для зв’язку використовується ключове слово «Binding» та аргументи:

– Path=Text;

– ElementName=TheTextBox.

Тобто встановлюється зв’язок із віджетом, що має ім’я «TheTextBox», а дані беруться із його властивості «Text». З лістингу вище можна бачити, що для віджету «TextBox» задане ім’я «TheTextBox»:

```
<TextBox x:Name="TheTextBox".
```

Ми використали панель Grid з двома рядками – текстове поле знаходиться у верхньому рядку (Grid.Row="0"), а TextBlock – у другому рядку (Grid.Row="1"). Вирівнювання за центром робиться за допомогою атрибутів HorizontalAlignment="Center" та VerticalAlignment="Center".

3.2.3 Віджет Button

Кнопка або «Button» дуже часто використовується в інтерфейсі користувача. В даному підрозділі ми розглянемо простий приклад використання даного віджету. Для створення кнопки, наприклад, може використовуватись наступний код:

```
<Button Content="Button"
      Padding="10,5"
      Grid.Row="2"/>
```

Padding="10,5" означає, що кнопка поширюється від свого вмісту на 10 пікселів ліворуч і праворуч, та 5 пікселів зверху і знизу.

Grid.Row="2" означає, що кнопка знаходиться у третьому рядку сітки (перший рядок займає текстова мітка, а в другому знаходиться віджет для введення тексту).

Повний код даного прикладу має наступний вигляд (див. лістинг «Listing_3-02»):

```
<Grid Grid.Row="1"
      HorizontalAlignment="Center"
  VerticalAlignment="Center">

      <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
      </Grid.RowDefinitions>

      <TextBlock Text="Name:"
         Grid.Row="0"
         Height="17"/>
      <TextBox x:Name="TheTextBox"
                  Width="150"
                  Height="27"
```

```
                    HorizontalAlignment="Left"
                    Grid.Row="1"/>
        <Button Content="Button"
                    Margin="0,10,0,0"
                    Padding="10,5"
                    Grid.Row="2"/>
</Grid>
```

Після компіляції та запуску програми, отримаємо наступний вигляд інтерфейсу користувача (рис. 3.4).

Рисунок 3.4 – Інтерфейс користувача із використанням кнопки

Кнопки визначають пов’язану подію Click, яка запускається при натисканні кнопки. Існує три способи викликати код C# для обробки натискання кнопки:

– використання «code behind» – як було показано у розділі 3.1 (перший приклад використання Avalonia).

– використання властивості «Command», яку можна прив’язати до властивості View Model. Властивість View Model, у свою чергу, може визначати лямбда-вираз для виклику при натисканні кнопки, а також властивість, щоб контролювати, чи активовано кнопку чи ні. Це буде детально описано в наступних розділах.

– використання поведінки, яка прослуховує переспрямовану подію Click та викликає метод C# під час запуску події (про це також буде сказано пізніше) – це найкращий метод.

3.2.4 Віджет ListBox

ListBox відображає колекцію елементів з можливістю вибору одного елемента за раз. Якщо кількість елементів перевищує розмір ListBox, він відобразить смуги прокрутки.

Найкращий спосіб використовувати поле зі списком – прив’язати його властивість Items до колекції. Як це зробити буде показано нижче. У нашому випадку ми просто створили ListBoxItems у коді XAML, щоб заповнити його:

```
<ListBox x:Name="TheListBox"
      Grid.Row="1"
      Margin="0,10,0,0">
  <ListBoxItem Content="Item 1"/>
  <ListBoxItem Content="Item 2"/>
  <ListBoxItem Content="Item 3"/>
  <ListBoxItem Content="Item 4"/>
</ListBox>
```

Найважливішими властивостями ListBox є згадані вище елементи та властивості, пов’язані з виділенням: SelectedIndex і SelectedItem. SelectedIndex. На рис. 3.5 подано приклад використання віджета ListBox.

Рисунок 3.5 – Приклад використання віджета ListBox

Повний текст прикладу подано в лістингу «Listing_3-03».

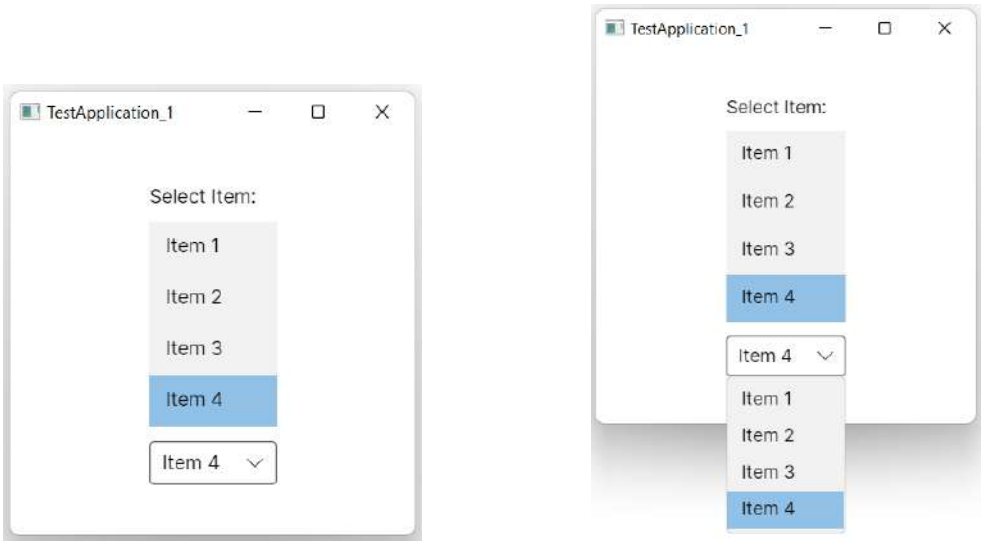
3.2.5 Віджет ComboBox

ComboBox також називається DropDownBox в інших фреймворках. Так само як ListBox, він зберігає колекцію елементів, але весь час відображається лише вибраний елемент – інші елементи відображаються лише у спливаючому вікні (точніше, у випадному меню), коли вказівник миші клацне стрілку з правого боку.

Для створення віджету ComboBox використаємо наступний код:

```
<ComboBox VerticalAlignment="Top"
          Grid.Row="2"
          SelectedIndex="{Binding Path=SelectedIndex,
                          ElementName=TheListBox}">
  <ComboBoxItem Content="Item 1"/>
  <ComboBoxItem Content="Item 2"/>
  <ComboBoxItem Content="Item 3"/>
  <ComboBoxItem Content="Item 4"/>
</ComboBox>
```

На рис. 3.6 подано приклад використання ComboBox в різних режимах роботи: в закритому стані (а), та в стані вибору елементу колекції (б).



а) б)

Рисунок 3.6 – Приклад використання віджета ComboBox:
а) в закритому стані; б) в стані вибору елемента колекції

З поданого рисунку можна бачити, що обидва віджети пов’язані між собою. Це зроблено за допомогою зв’язку між компонентами в файлі XAML.

Зв’язок між віджетами зроблено за допомогою наступного рядку коду у властивостях віджета ComboBox:

```
SelectedIndex="{Binding Path=SelectedIndex,
                ElementName=TheListBox}"
```

Так само, як і у випадку ListBox, основними властивостями ComboBoxs є Items (що мають бути прив'язані до колекції): SelectedIndex і SelectedItem. В даному рядку вказано, що властивість SelectedIndex ComboBox бере значення із однойменної властивості SelectedIndex компоненту ListBox, ім'я якого TheListBox.

Таким чином, після запуску наша програма працює навіть без жодного рядка вихідного коду у файлі «MainWindow.axaml.cs». Повний текст прикладу подано в лістингу «Listing_3-04».

До даного прикладу в рядок організації зв'язку можна додати атрибут Mode зі значенням TwoWay, що надасть можливість передавати параметр властивості SelectedIndex від одного віджета до іншого і навпаки:

```
SelectedIndex="{Binding Path=SelectedIndex,
      ElementName=TheListBox,
      Mode=TwoWay}"
```

3.2.6 Віджет ToggleButton

ToggleButton – це елемент керування, який має два стани – Checked та Unchecked, які контролюються його булевою властивістю IsChecked. Щоразу, коли натискається кнопка, її властивість IsChecked змінює значення з false на true і навпаки. Фон кнопки змінюється залежно від того, чи не встановлено прапорець.

Додамо до нашого тестового прикладу ToggleButton. Для цього трохи модифікуємо контейнер Grid та зробимо в ньому три колонки. Середній стовпчик буде використовуватись в якості розмежувального елементу залишаючись порожнім. Вся конструкція сітки буде виглядати наступним чином:

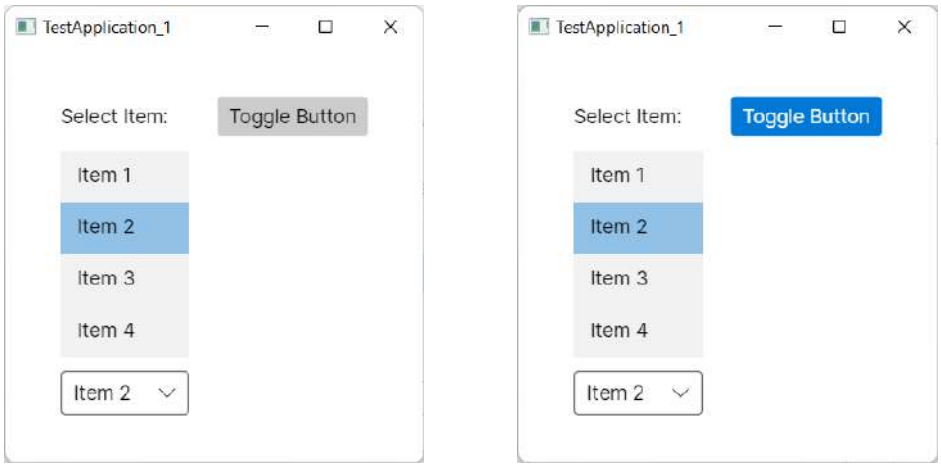
```
<Grid ColumnDefinitions="Auto, 20, Auto"
      RowDefinitions="Auto, Auto, Auto"
      HorizontalAlignment="Center"
      VerticalAlignment="Center">
```

Віджет ToggleButton розмістимо у правому крайньому стовпчику в першому рядку:

```
<ToggleButton x:Name="TheToggleButton"
```

```
Grid.Row="0"
Grid.Column="2"
Content="Toggle Button"/>
```

Приклад створеного інтерфейсу користувача подано на рис. 3.7, а. При натисканні на кнопку вона змінює колір і залишається в такому стані, поки користувач її знову не натисне (рис. 3.7, б).



а) б)

Рисунок 3.7 – Поведінка віджета ToggleButton:
а) стан кнопки після запуску; б) стан кнопки після натискання

3.2.7 Віджет CheckBox

CheckBox дуже схожий на кнопку перемикання (ToggleButton), але виглядає інакше. Для створення CheckBox використовується код XAML:

```
<CheckBox Content="Check Box"
      VerticalAlignment="Top"
      Grid.Row="1"
      Grid.Column="2"
      sChecked="{Binding Path=IsChecked,
            ElementName=TheToggleButton}"/>
```

На рис. 3.8 подано приклад використання віджета CheckBox. В даному прикладі використовується прив’язка, яка з’єднує властивість IsChecked компоненти CheckBox з властивістю IsChecked ToggleButton.

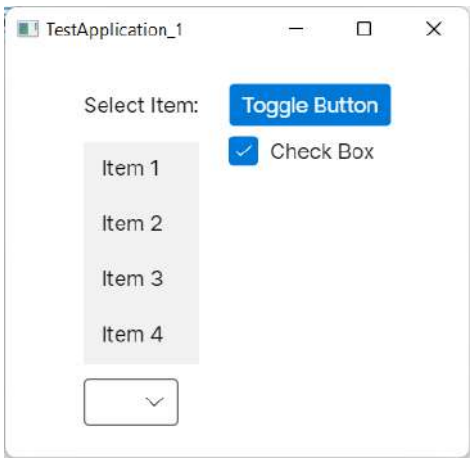
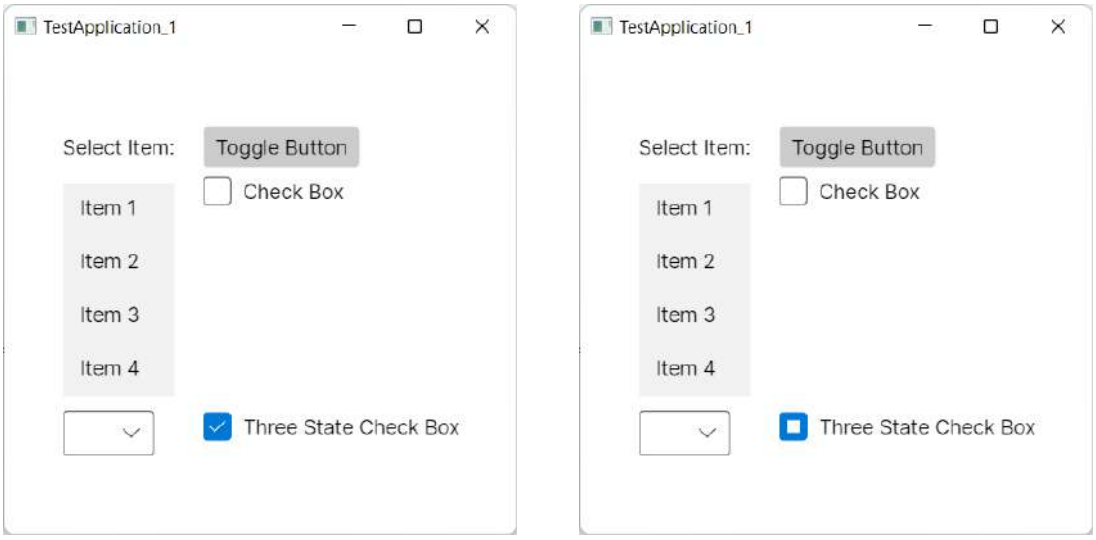


Рисунок 3.8 – Приклад використання віджета CheckBox

CheckBox також має властивість перемикача IsThreeState і, коли для нього встановлено значення true, CheckBox може перемикатися між трьома станами: false, true та undefined, що відповідає його властивості IsChecked, встановленій на null:

```
<CheckBox Content="Three State Check Box"
                        Grid.Row="2"
                        Grid.Column="2"
                        IsThreeState="True"/>
```

Зовнішній вигляд графічного інтерфейсу показаний на рис. 3.9.



а) б)

Рисунок 3.9 – Зовнішній вигляд графічного інтерфейсу з використанням віджету CheckBox: а) стан «true»; б) стан «undefined»

3.2.8 Віджет ContextMenu

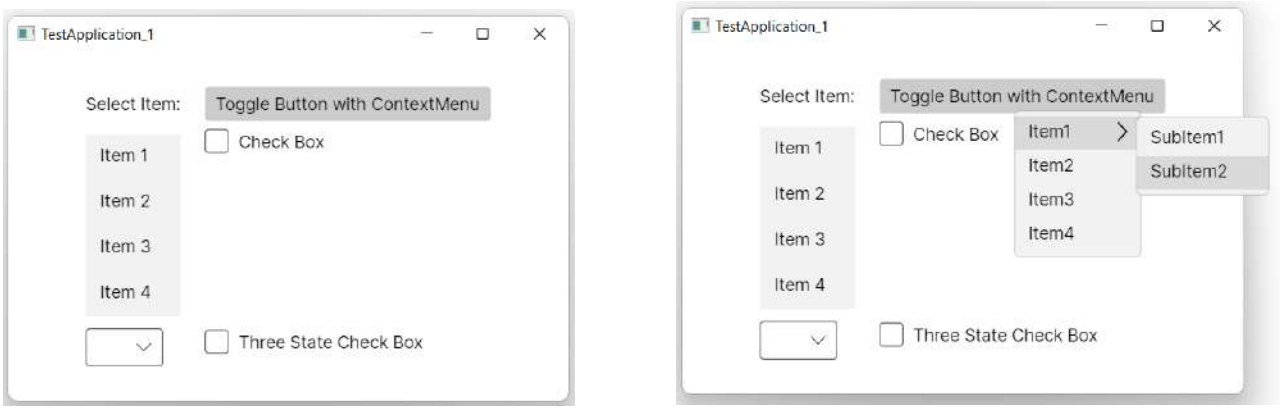
Контекстне меню відкривається, якщо натиснути правою кнопкою миші на певну область або елементі керування. На рис. 3.10 подано приклад застосування контекстного меню. Показано два стани роботи тестової програми:

– початковий стан, коли права кнопка миші на натиснута (рис. 3.10, а);

– робочій стан, коли контекстне меню відображається справа від точки натискання правої кнопки миші на контролі (рис. 3.10, б).

Для створення контекстного меню застосували наступний код:

```
<ToggleButton x:Name="TheToggleButton"
      Grid.Row="0"
      Grid.Column="2"
      Content="Toggle Button with ContextMenu">
      <ToggleButton.ContextMenu>
            <ContextMenu>
                  <MenuItem Header="Item1">
                        <MenuItem Header="SubItem1"/>
                        <MenuItem Header="SubItem2"/>
                  </MenuItem>
                  <MenuItem Header="Item2"/>
                  <MenuItem Header="Item3"/>
                  <MenuItem Header="Item4"/>
            </ContextMenu>
      </ToggleButton.ContextMenu>
</ToggleButton>
```



а) б)

Рисунок 3.10 – Приклад застосування контекстного меню: а) початковий стан, коли права кнопка миші не натиснута; б) контекстне меню відображається справа від точки натискання правої кнопки миші на контролі

Створення контекстного меню відбулось завдяки додаванню до опису кнопки пари тегів «<ToggleButton.ContextMenu> … </ToggleButton.ContextMenu>» з подальшим описом структури меню конструкцією «<ContextMenu> … </ContextMenu>»

3.2.9 Віджет Menu

Меню зазвичай розміщується у верхній частині вікна, але можуть з'являтися і в інших місцях графічного вікна програми. Меню зазвичай розміщується на панелі DockPanel або Grid у вікні, яке закріплене у верхній частині вікна. Даний віджет створюється за допомогою наступного коду XAML:

```
<Menu DockPanel.Dock="Top">
    <MenuItem Header="_FILE">
      <MenuItem Header="_New"/>
      <MenuItem Header="Open"/>
      <MenuItem Header="Save"/>
    </MenuItem>
    <MenuItem Header="EDIT">
      <MenuItem Header="Copy"/>
      <MenuItem Header="Paste"/>
    </MenuItem>
</Menu>
```

Зазвичай меню містить набір вкладених елементів MenuItems. Перший рівень MenuItems складають елементи, які будуть відображатися горизонтально вздовж меню. Другий рівень MenuItems – це пункти меню, які будуть випадати з верхнього рівня, а наступні вкладені MenuItems – це підменю.

Текст MenuItem відображається властивістю Header. Внутрішній вміст MenuItem – це місце, де розміщуються підпункти.

Роздільники додаються шляхом включення елемента керування Separator або MenuItem із заголовком «-». Приклад використання віджету Menu подано на рис. 3.11.

Клавіша прискорення – це клавіша на клавіатурі, яку можна натиснути для швидкого доступу до меню. ЇЇ також іноді називають гарячою клавішею, клавішею доступу або мнемонікою.

Якщо ви натиснете Alt у наведеному вище прикладі, ви побачите, що деякі літери підкреслені. Ви можете використовувати комбінацію Alt + підкреслена

літера для навігації по меню. В Avalonia, щоб ідентифікувати клавішу прискорення, потрібно використати «_» перед символом, до якого буде застосована клавіша прискорення. В нашому прикладі це конструкція «_FILE» та «_New».

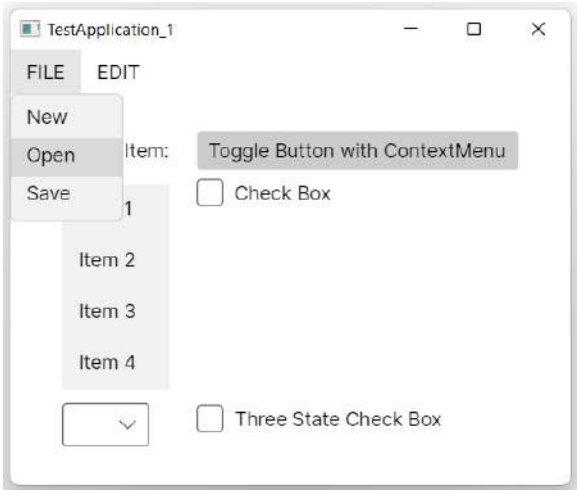


Рисунок 3.11 – Приклад використання віджета Menu

Так само як Button, команди можуть бути застосовані до MenuItems. Команда буде виконана, якщо натиснути або вибрати пункт меню за допомогою клавіатури або миші:

```
<Menu>
    <MenuItem Header="_File">
        <MenuItem Header="_Open..." Command="{Binding OpenCommand}"/>
    </MenuItem>
</Menu>
```

Значок меню можна відобразити, помістивши зображення у властивість Icon:

```
    <MenuItem Header="_Open...">
        <MenuItem.Icon>
            <Image Source="/Assets/Open.png"/>
        </MenuItem.Icon>
    </MenuItem>
```

Попередньо слід помістити зображення в папку «Assets». В нашому прикладі – це зображення «Open.png» (рис. 3.12).

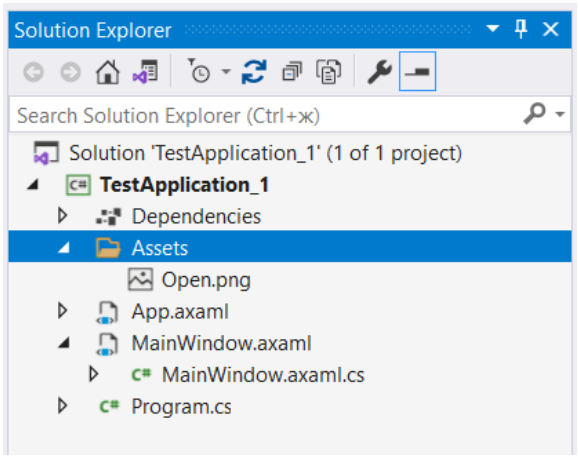


Рисунок 3.12 – Додавання файлу із зображенням до проєкту

Якщо папка «Assets» створюється вручну, необхідно додати всі свої ресурси до тегу «<ItemGroup>» у файлі .csproj, щоб Avalonia могла їх знайти. Відносного шляху достатньо. Якщо .csproj знаходиться в кореневій папці програм, потрібно буде додати таку властивість у файл .csproj:

```
<ItemGroup>
    <AvaloniaResource Include="Assets\**" />
</ItemGroup>
```

Коли проєкт Avalonia створюється з шаблону у Visual Studio, папка Assets буде автоматично включена в проєкт і тому наведена вище конструкція буде у файлі .csproj за замовчуванням. В результаті отримуємо таку структуру меню, як подано на рис. 3.13. Повний текст прикладу подано в лістингу «Listing_3-06».

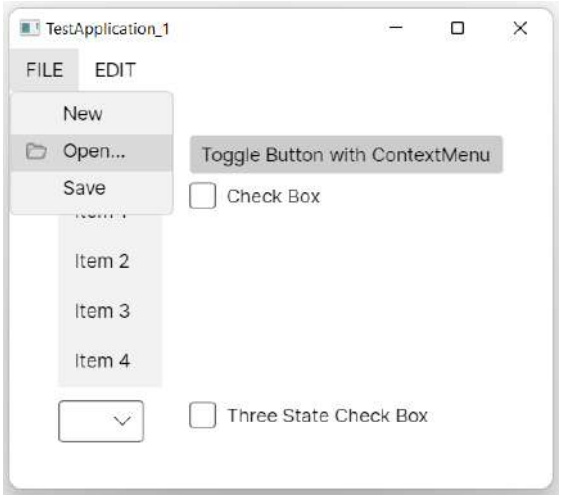


Рисунок 3.13 – Відображення зображення поруч із назвою пункту меню

3.2.10 Popup

Спливаюче вікно – це елемент керування, який відкриває невелике вікно поруч із так званим цільовим місцем розташування спливаючого вікна.

Нижче наведено приклад коду для створення спливаючого вікна:

```
<Grid Grid.Row="1">
    <Grid.RowDefinitions>
      <RowDefinition Height="Auto"/>
      <RowDefinition Height="*"/>
    </Grid.RowDefinitions>
    <ToggleButton x:Name="OpenClosePopupButton"
                  Content="Open/Close Popup"/>
    <Popup x:Name="ThePopup"
           Grid.Row="1"
           IsOpen="{Binding Path=IsChecked,
                  ElementName=OpenClosePopupButton, Mode=TwoWay}"
           StaysOpen="False"
           PlacementMode="Bottom"
           PlacementTarget="{Binding ElementName=OpenClosePopupButton}">
      <Grid x:Name="PopupsContent"
            Background="Red"
            Width="150"
            Height="70">
        <TextBlock Text="Popup’s Content"
                   HorizontalAlignment="Center"
                   VerticalAlignment="Center"/>
      </Grid>
    </Popup>
</Grid>
```

На рис. 3.14 подано приклад використання віджета Popup.

Відкривати спливаюче вікно чи ні, контролюється (і відображається) його властивістю IsOpen, яка в нашому випадку пов’язана з властивістю IsChecked ToggleButton, яка контролює (і відображає) стан спливаючого вікна.

Ми використовуємо двостороннє прив’язування, щоб прив’язати властивість Popup IsOpen до властивості IsChecked ToggleButton, щоб зміна кожного з них вплинула на іншу.

Властивість StaysOpen має значення false – це означає, що клацання за межами спливаючого вікна закриває це вікно.

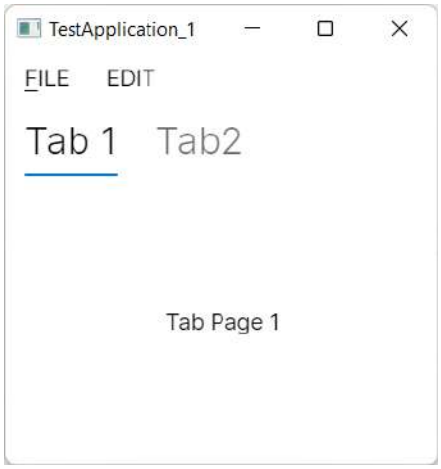
Рисунок 3.14 – Приклад використання віджета Popup

Властивість PlacementTarget визначає елемент, відносно якого розташовуватиметься спливаюче вікно.

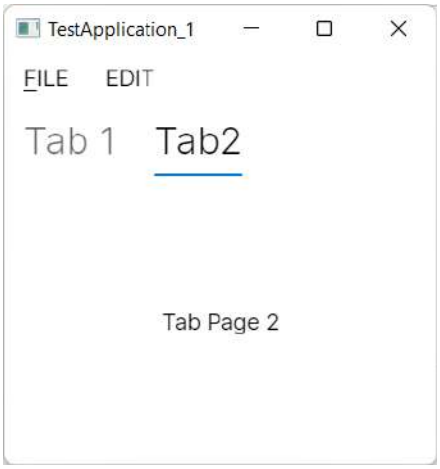
PlacementMode="Bottom" означає, що спливаюче вікно буде розташовуватися внизу цільового місця розташування.

3.2.11 Віджет TabControl

TabControl дозволяє відображати різні вкладки – кожна вкладка містить певний вміст. Перемикання між вкладками, наприклад, призведе до зміни тексту, що відображається, з «Tab Page 1» до «Tab Page 2» (рис. 3.15):



а)



б)

Рисунок 3.15 – Приклад використання віджета TabControl:
а) Tab Page 1; б) Tab Page 1

Для створення віджета TabControl використовується наступний код XAML:

```
<TabControl Grid.Row="1">
      <TabItem Header="Tab 1">
            <TextBlock Text="Tab Page 1"
                  HorizontalAlignment="Center"
                  VerticalAlignment="Center"/>
      </TabItem>
      <TabItem Header="Tab2">
            <TextBlock Text="Tab Page 2"
                  HorizontalAlignment="Center"
                  VerticalAlignment="Center"/>
      </TabItem>
</TabControl>
```

3.2.12 Робота з вікнами

В більшості проєктів використовується декілька вікон для організації інтерфейсу користувача. Розглянемо принцип створення та використання нового вікна у нашій тестовій програмі.

Для створення нового вікна, треба викликати контекстне меню правою кнопкою миші на назві проєкту в Solution Explorer, обрати опцію Add, та в іншому меню, що з’явиться, обрати опцію «New Item» (рис. 3.16).

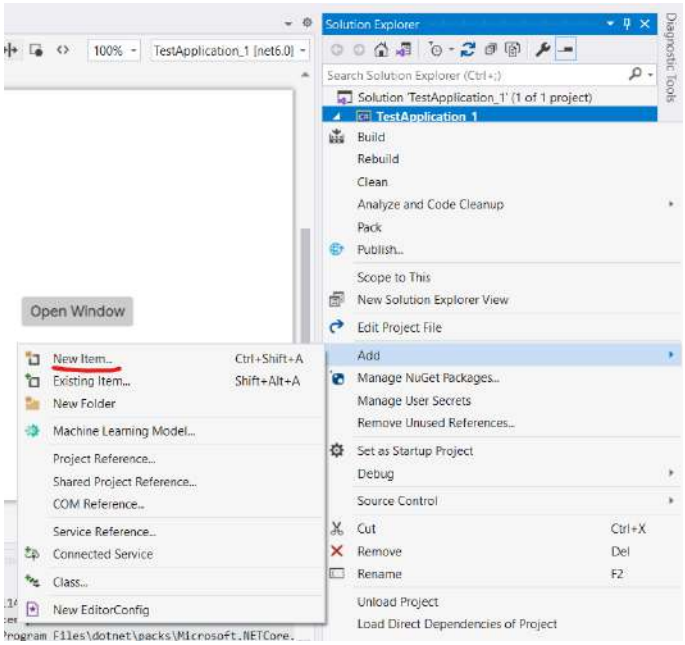


Рисунок 3.16 – Виклик діалогу створення нового вікна

У вікні, що відкриється (рис. 3.17), необхідно вибрати розділ «Avalonia», далі в списку обрати пункт «Window (Avalonia)», ввести назву нового вікна (в нашому випадку «Window2.xaml») і натиснути кнопку «Add».

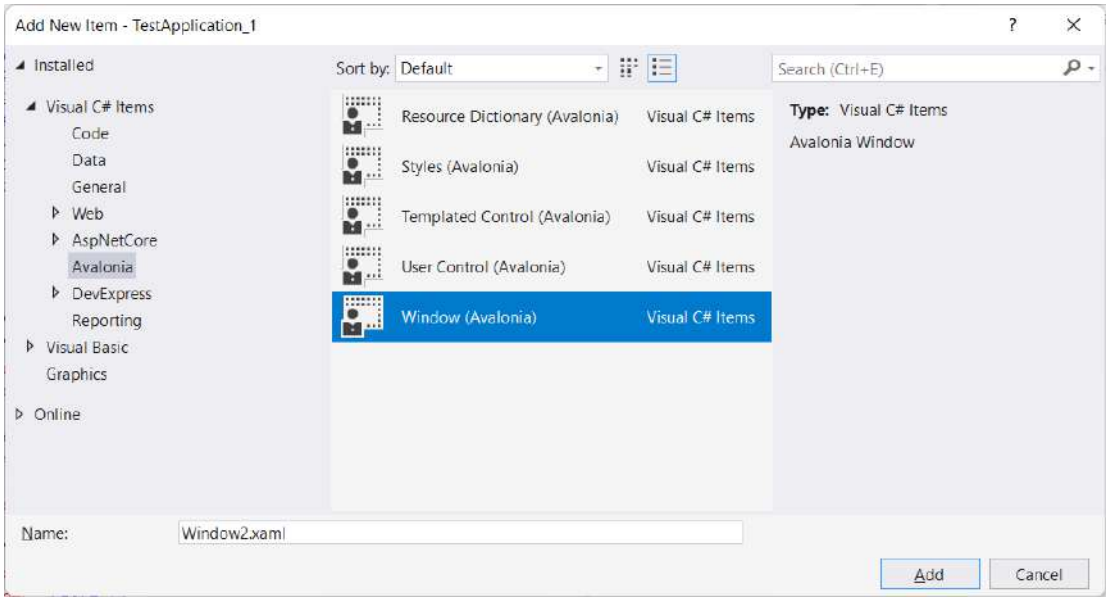


Рисунок 3.17 – Створення нового вікна

Після цього в проєкті буде створено два нових файли Window2.axaml та Window2.axaml.cs (рис. 3.18).

В першому вікні необхідно додати нову кнопку, для створення можливості виклику другого вікна:

```
<Button x:Name="OpenWindowButton"
        Content="Open Window"/>
```

Для цього замінимо весь код, що знаходиться між тегами <Grid> … </Grid>:

```
<Grid ColumnDefinitions="Auto"
      RowDefinitions="Auto"
      HorizontalAlignment="Center"
      VerticalAlignment="Center">
      <Button x:Name="OpenWindowButton"
            Content="Open Window"/>
</Grid>
```

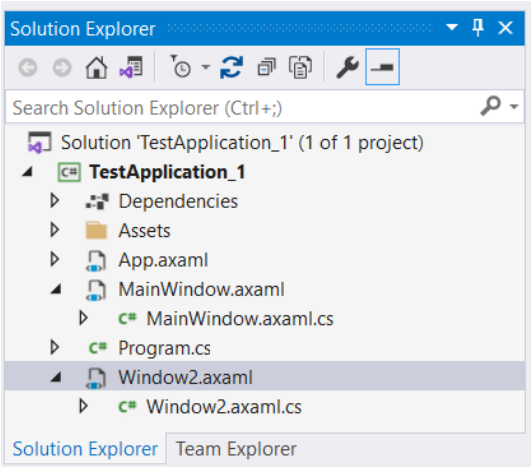


Рисунок 3.18 – Додані до проєкту нові файли

Код C#, який відкриває вікно, підключається за допомогою виділеного коду (code-behind) – не тому, що це хороший спосіб зробити це (насправді це найгірший, як згадувалося вище), а тому, що це найпростіший спосіб і найлегший для розуміння. В файлі MainWindow.axaml.cs до конструктора «MainWindow()» після InitializeComponent() додаємо наступний фрагмент коду:

```
var openWindowButton = this.FindControl<Button>("OpenWindowButton");
openWindowButton.Click += OpenWindowButton_Click;
```

У конструкторі MainWindow ми знаходимо кнопку за її назвою та приєднуємо обробник до події Click. Також необхідно створити нову функцію «OpenWindowButton_Click» з наступним вмістом:

```
private void OpenWindowButton_Click(object? sender, RoutedEventArgs e)
{
      // Create the window object
      Window sampleWindow = new Window2();
      sampleWindow.Width = 300;
      sampleWindow.Height = 250;

      // open the window
      sampleWindow.Show();
}
```

В даній функції спочатку створюється об’єкт типу Window2. Даному об’єкту визначаються нові значення довжини Width = 300 та ширини

Height = 250. Далі використовується метод Show() для відображення вікна на екрані. В результаті отримуємо такий приклад роботи програми (рис. 3.19).

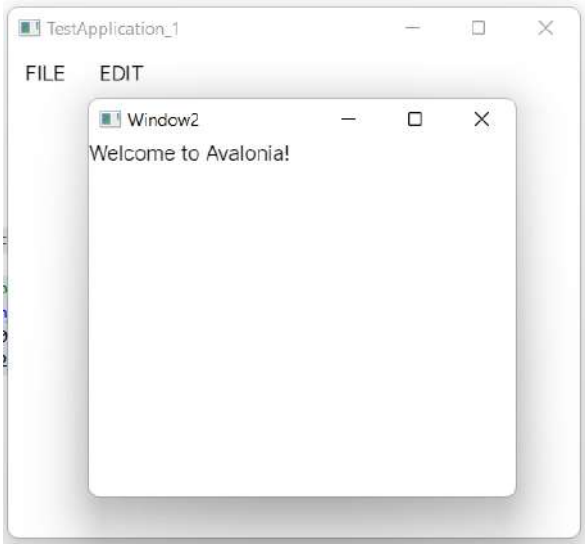


Рисунок 3.19 – Приклад запуску програму з двома вікнами

Якщо дослідити поведінку створеного вікна, то можна побачити, що можна повернутися до першого вікна не закриваючи другого, просто натиснувши мишею в площині вікна. Якщо така поведінка не є придатною, можна скористатися іншим методом виклику нового вікна – створення модального вікна.

Модальне вікно також називається діалоговим – це вікно, яке запобігає будь-яким діям зі своїми попередніми вікнами, поки воно не буде закрито. Для створення модального вікна використовується OpenDialog.

Виконаємо модифікацію вихідного коду та доповнимо його новою конструкцією:

```
<Button x:Name="OpenModalWindowButton"
      HorizontalAlignment="Center"
      Grid.Row="1"
      Content="Open Modal (Dialog) Window"/>
```

Ця конструкція створить на екрані нову кнопку, що буде розміщена під кнопкою виклику немодального вікна. Код, що відповідає за створення кнопок тепер буде наступний:

```
<Grid ColumnDefinitions="Auto"
      RowDefinitions="Auto, Auto"
      HorizontalAlignment="Center"
      VerticalAlignment="Center">

      <Button x:Name="OpenWindowButton"
            HorizontalAlignment="Center"
            Grid.Row="0"
            Content="Open Window"/>

      <Button x:Name="OpenModalWindowButton"
            HorizontalAlignment="Center"
            Grid.Row="1"
            Content="Open Modal (Dialog) Window"/>
</Grid>
```

Для оброки події натискання на кнопку «OpenModalWindowButton», доповнимо код в файлі MainWindow.axaml.cs такими рядками:

```
var openModalWindowButton =
      this.FindControl<Button>("OpenModalWindowButton");
openModalWindowButton.Click += OpenModalWindowButton_Click;
```

Також необхідно створити функцію для створення об’єкту та виклику модального вікна:

```
private void OpenModalWindowButton_Click(object? sender,
RoutedEventArgs e)
{
      // Create the window object
      Window sampleWindow = new Window2();
      sampleWindow.Width = 300;
      sampleWindow.Height = 250;

      // open the window
      sampleWindow.ShowDialog(this);
}
```

Єдина відмінність між цим і попередніми зразками полягає в тому, що тут ми викликаємо метод sampleWindow.ShowDialog(...) замість

sampleWindow.Show(), передаючи йому поточне вікно як батьківське вікно. Результат роботи програми подано на рис. 3.20.

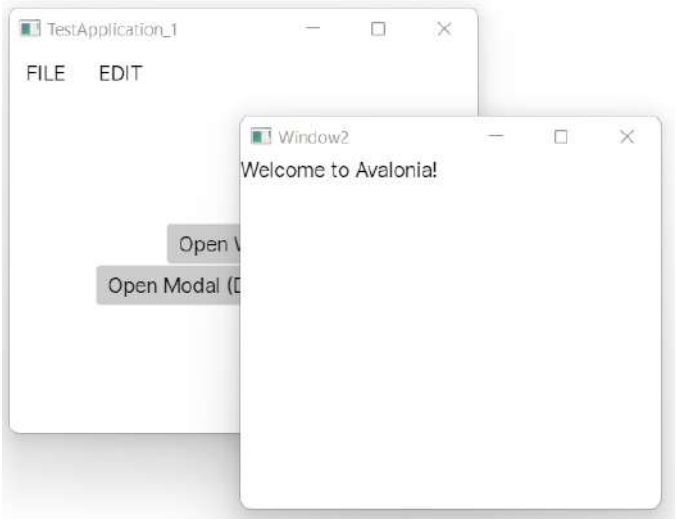


Рисунок 3.20 – Результат виведення модального діалогового вікна

Повністю код проєкту можна завантажити з папки «Listing_3-07» та протестувати.

3.3 Контейнери та панелі в Avalonia

Панелі – це примітивні елементи керування Avalonia, які призначені для розташування інших елементів керування всередині них. Крім кольору фону, самі панелі не мають жодного візуального представлення, але вони незамінні для впорядкування та розміщення інших елементів керування.

3.3.1 StackPanel

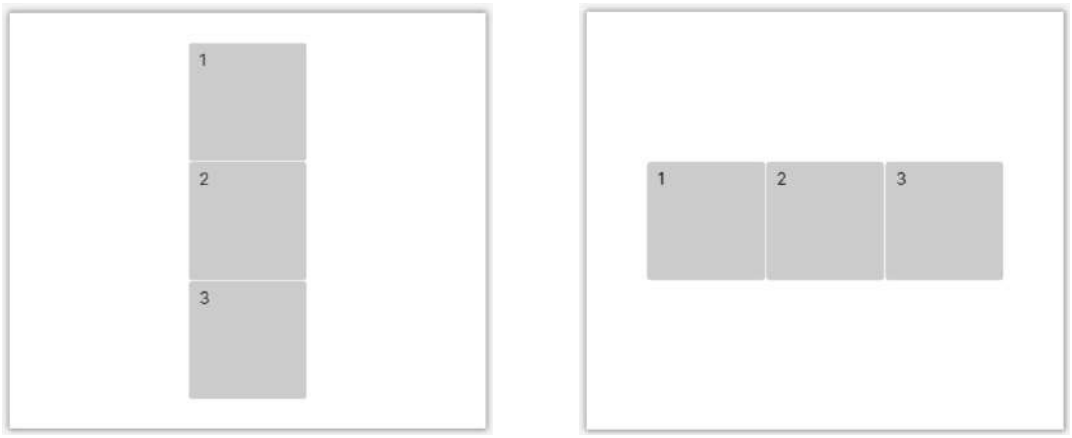
Елемент керування StackPanel – це панель, яка розміщує дочірні елементи, упорядковуючи їх горизонтально або вертикально. StackPanel зазвичай використовується для організації невеликого підрозділу інтерфейсу користувача на сторінці.

Можна використовувати властивість Orientation, щоб задати напрямок дочірніх елементів. Орієнтація за замовчуванням – вертикальна. Наступний код XAML показує, як створити вертикальну панель StackPanel елементів:

```
<StackPanel Orientation="Vertical"
      HorizontalAlignment="Center"
```

```
        VerticalAlignment="Center">
        <Button Content="1"
              Width="100"
              Height="100"/>
        <Button Content="2"
              Width="100"
              Height="100"/>
        <Button Content="3"
              Width="100"
              Height="100"/>
</StackPanel>
```

На рис. 3.21 подано приклад використання StackPanel для вертикальної та горизонтальної орієнтації.



а) б)

Рисунок 3.21 – Приклад використання StackPanel:
а) вертикальна орієнтація; б) горизонтальна орієнтація

Для досягнення горизонтальної орієнтації, що подана на рис. 3.21, б, потрібно замінити властивість Orientation на Orentation="Horizontal".

У StackPanel, якщо розмір дочірнього елемента не встановлено явно, він розтягується, щоб заповнити доступну ширину (або висоту, якщо орієнтація горизонтальна). У прикладі, що наведено нижче, ширина прямокутників не встановлена. Прямокутники розгортаються, щоб заповнити всю ширину StackPanel:

```
<StackPanel Spacing="5">
    <Rectangle Fill="Red" Height="44"/>
```

```
    <Rectangle Fill="Blue" Height="44"/>
    <Rectangle Fill="Green" Height="44"/>
    <Rectangle Fill="Orange" Height="44"/>
</StackPanel>
```

StackPanel має властивість Spacing, щоб забезпечити рівномірний інтервал між елементами. На рис. 3.22 подано результат роботи коду, який наведено вище.

Рисунок 3.22 – Результат використання властивості Spacing

Якщо змінити розмір вікна, ви побачите, що, коли вікно стає занадто маленьким, кінці StackPanels обрізаються. Віджет WrapPanel вирішує цю проблему.

3.3.2 Віджет WrapPanel

Елемент керування WrapPanel – це панель, яка розміщує дочірні елементи в послідовному положенні зліва направо, перериваючи вміст з наступного рядка на краю поля, що вписується в контейнер верхнього рівня. Подальше впорядкування відбувається послідовно зверху вниз або справа наліво, залежно від значення властивості Orientation.

Для створення WrapPanel замінимо назву віджета в попередньому коді XAML:

```
<WrapPanel Orientation="Vertical"
      HorizontalAlignment="Center"
      VerticalAlignment="Center">
      <Button Content="1"
            Width="100"
            Height="100"/>
      <Button Content="2"
```

```
            Width="100"
            Height="100"/>
        <Button Content="3"
            Width="100"
            Height="100"/>
</WrapPanel>
```

В результаті отримаємо адаптоване розміщення кнопок в залежності від розміру корисної площини вікна (рис. 3.23)

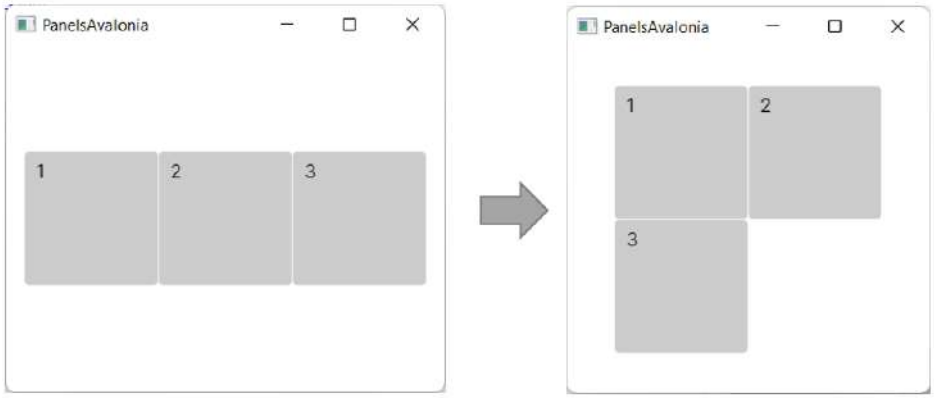


Рисунок 3.23 – Адаптоване розміщення кнопок в залежності від розміру корисної площини вікна

3.3.3 Віджет Grid

Віджет Grid – це різновид панелі, що використовується для організації інших віджетів в стовпцях і рядках. Властивості ColumnDefinition і RowDefinition використовуються для визначення абсолютної, відносної або пропорційної геометрії рядків і стовпців для сітки. Кожен віджет в сітці буде розміщено за допомогою додаткових властивостей Grid.Column і Grid.Row. Також можна мати віджети, які охоплюють декілька рядків та/або стовпців, використовуючи властивості ColumnSpan та RowSpan.

Нижче наведено приклад, який показує:

– налаштування сітки безпосередньо за допомогою властивостей ColumnDefinition і GridDefinition;

– принцип призначення комірки для кожного компонента;

– показані ефекти охоплення рядків/стовпців (наведено приклад сітки з трьома рівними рядками та трьома стовпцями: перший стовбець зроблено з фіксованою шириною, а інші два пропорційно охоплюють решту.

Приклад коду XAML:

```
<Grid ColumnDefinitions="100,1.5*,4*" RowDefinitions="Auto,Auto,Auto" Margin="4">
  <TextBlock Text="Col0Row0:" Grid.Row="0" Grid.Column="0"/>
  <TextBlock Text="Col0Row1:" Grid.Row="1" Grid.Column="0"/>
  <TextBlock Text="Col0Row2:" Grid.Row="2" Grid.Column="0"/>
  <CheckBox Content="Col2Row0" Grid.Row="0" Grid.Column="2"/>
  <Button Content="SpansCol1-2Row1-2" Grid.Row="1" Grid.Column="1" Grid.RowSpan="2" Grid.ColumnSpan="2"/>
</Grid>
```

В результаті компіляції програми отримаємо наступний вигляд головного вікна (рис. 3.24).

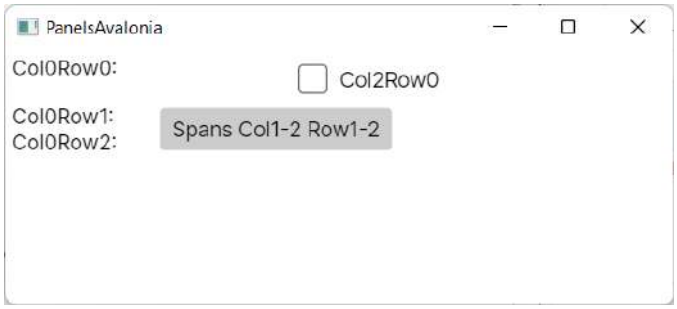


Рисунок 3.24 – Використання віджету Grid

У наведеному вище прикладі ми маємо два ключові слова: asterix та Auto:

– ключове слово «Auto» використовується для визначення геометрії рядка або стовпця за властивостями елемента керування;

– зірочка використовується для позначення пропорційного заповнення простору.

Множник, що використовується перед значенням пропорційного інтервалу, використовується для визначення відносного розміру пропорційних стовпців. Усі пропорційні стовпці поміщаються в простір, що залишився після обчислення всіх явних значень і значень «Авто». Отже, у наведеному вище прикладі стовпець 1 отримає 1,5 частини, а стовпець 2 отримає 4 частини залишку простору, який залишив стовпець 0. Нарешті, сама кнопка буде заповнюватися від початкової клітинки 1,1 над одним стовпцем і вниз на один

рядок, оскільки Grid.RowSpan і Grid.ColumnSpan налаштовані так, щоб вони займали дві одиниці замість однієї.

В режимі налагодження програми іноді корисно увімкнути візуалізацію сітки на екрані. В прикладі нижче показано, як використовується властивість «ShowGridLines="True"»:

```
<Grid ShowGridLines="True">
        <Grid.RowDefinitions>
            <RowDefinition Height="*"></RowDefinition>
            <RowDefinition Height="*"></RowDefinition>
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="*"></ColumnDefinition>
            <ColumnDefinition Width="*"></ColumnDefinition>
        </Grid.ColumnDefinitions>

        <Button Grid.Row="0" Grid.Column="0">Some text written on a
button</Button>
        <Button Grid.Row="0" Grid.Column="1">Some text written on a
button</Button>
        <Button Grid.Row="1" Grid.Column="0">Some text written on a
button</Button>
        <Button Grid.Row="1" Grid.Column="1">Some text written on a
button</Button>
    </Grid>
```

В результаті на екрані буде відображено наступний вигляд інтерфейсу користувача (рис. 3.25).

Властивості віджета Grid:

– ColumnDefinitions – колекція ColumnDefinitions, що визначає максимальну або мінімальну ширину стовпця;

– RowDefinitions – колекція RowDefinitions, що визначає максимальну або мінімальну висоту рядка;

– Grid.Column – властивість, що призначена для прикріплення елемента керування до певного стовпця;

– Grid.Row – властивість, що призначена для прикріплення елемента керування до певного рядка;

– Grid.ColumnSpan – властивість для визначення кількості стовпців, які охоплюватиме елемент керування;

– Grid.RowSpan – властивість для визначення кількості рядків, які охоплюватиме елемент керування.

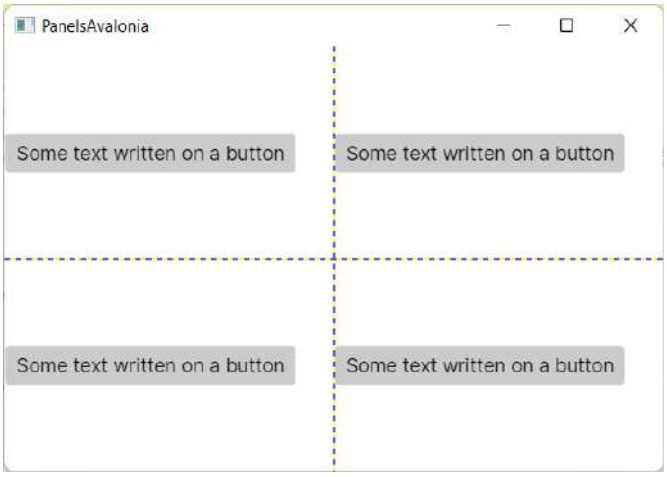


Рисунок 3.25 – Приклад візуалізації сітки на екрані

3.3.4 Віджет DockPanel

Панель DockPanel дозволяє розташувати дочірні елементи поруч зі своїми (визначеними) сторонами, тоді як останній дочірній (відкріплений) займе решту місця.

Значення dock визначається властивістю DockPanel.Dock, яка може приймати значення Left, Top, Right і Bottom.

У прикладі нижче ми маємо 8 кнопок, розташованих за годинниковою стрілкою: ліворуч, зверху, праворуч, знизу і знову ліворуч, зверху, праворуч, знизу, а потім остання кнопка, яка займає решту місця:

```
<DockPanel Margin="20">
      <Button Content="1"
            VerticalAlignment="Stretch"
            DockPanel.Dock="Left"
            Width="30"/>
      <Button DockPanel.Dock="Top"
            HorizontalAlignment="Stretch"
            Content="2"
            Height="30"/>
      <Button DockPanel.Dock="Right"
            VerticalAlignment="Stretch"
            Content="3"
            Width="30"/>
```

```
        <Button DockPanel.Dock="Bottom"
                HorizontalAlignment="Stretch"
                Content="4"
                Height="30"/>
        <Button DockPanel.Dock="Left"
                VerticalAlignment="Stretch"
                Content="5"
                Width="30"/>
        <Button DockPanel.Dock="Top"
                HorizontalAlignment="Stretch"
                Content="6"
                Height="30"/>
        <Button DockPanel.Dock="Right"
                VerticalAlignment="Stretch"
                Content="7"
                Width="30"/>
        <Button DockPanel.Dock="Bottom"
                HorizontalAlignment="Stretch"
                Content="8"
                Height="30"/>
        <Button Content="The Rest"
                HorizontalAlignment="Stretch"
                VerticalAlignment="Stretch"/>
    </DockPanel>
```

В результаті компіляції отримаємо наступний вигляд інтерфейсу програми (рис. 3.26).

Рисунок 3.26 – Приклад використання віджету DockPanel

3.3.5 Віджет Canvas

Canvas – це панель, яка розставляє свої дочірні елементи за явними координатами та дозволяє розміщувати в ній елементи керування за певними координатами з верхнього лівого кута. Розробник вказуєте положення окремих дочірніх елементів, встановлюючи для кожного елемента прикріплені властивості Canvas.Left і Canvas.Top.

Об'єкти на Canvas можуть перекриватися, коли один об'єкт розташовується поверх іншого. За замовчуванням Canvas відтворює дочірні об'єкти в тому порядку, в якому вони оголошені, тому останній дочірній елемент відображається зверху (кожен елемент має за замовчуванням z-індекс 0). Це те саме, що й інші вбудовані панелі. Однак Canvas також підтримує властивість Canvas.ZIndex, яку можна встановити для кожного з дочірніх елементів. Можна встановити цю властивість у коді, щоб змінити порядок створення елементів під час виконання.

Елемент з найвищим значенням Canvas.ZIndex відображається останнім і, отже, розташовується поверх будь-яких інших елементів, які мають однаковий простір або будь-яким чином перекриваються. Необхідно також зауважити, що підтримується значення альфа (прозорість), тому навіть якщо елементи перекриваються, вміст, показаний у областях накладання, може бути змішаний, якщо верхній має немаксимальне значення альфа.

Canvas не визначає розміри своїх дітей. Кожен елемент повинен вказати свій розмір. Ось приклад Canvas в XAML:

```
<Canvas>
  <Button Content="1"
          Width="100"
          Height="100"
          Canvas.Left="300"
          Canvas.Top="200"/>
</Canvas>
```

На рис. 3.27 подано результат використання віджету Canvas. Код XAML, що наведено нижче демонструє ефект накладання віджетів один на одного:

```
<Canvas Width="120" Height="120">
    <Rectangle Fill="Red" Height="44" Width="44"/>
```

```
        <Rectangle Fill="Blue" Height="44" Width="44" Canvas.Left="20"
Canvas.Top="20"/>
        <Rectangle Fill="Green" Height="44" Width="44" Canvas.Left="40"
Canvas.Top="40"/>
        <Rectangle Fill="Orange" Height="44" Width="44" Canvas.Left="60"
Canvas.Top="60"/>
    </Canvas>
```

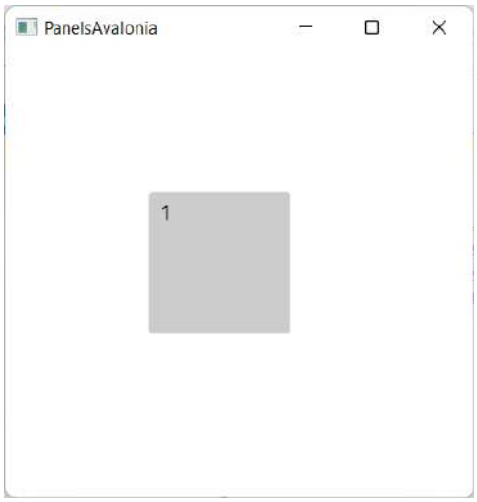


Рисунок 3.27 – Результат використання віджету Canvas

Результат компіляції подано на рис. 3.28.

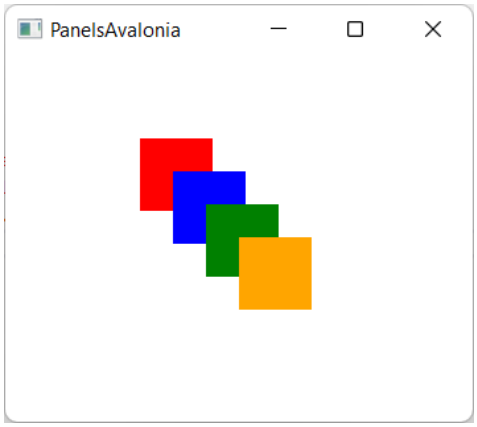


Рисунок 3.28 – Ефект накладання віджетів один на одного

Необхідно зауважити, що треба використовувати панель Canvas обережно. Хоча для деяких сценаріїв зручно точно контролювати положення елементів в інтерфейсі користувача, фіксована панель макета призводить до того, що ця область інтерфейсу буде менш адаптивною до загальних змін розміру вікна програми.

3.3.6 Віджет RelativePanel

Віджет RelativePanel не існує в WPF. Однак це може бути дуже корисним, особливо під час кодування для планшетів і телефонів. Він дозволяє вказати розташування елементів як щодо панелі, так і щодо інших іменованих елементів у межах тієї ж панелі.

RelativePanel надає багато так званих дочірніх властивостей, які дозволяють вибрати позицію її дочірнього по відношенню до самої панелі або щодо інших іменованих дочірніх елементів тієї ж панелі. Ось код XAML для прикладу:

```
<RelativePanel Margin="20"
               Background="LightBlue">
  <Button x:Name="Button1"
          Height="50"
          Content="Button1 - TopLeftCorner by default"/>
  <Button x:Name="Button2"
          Height="50"
          RelativePanel.AlignTopWithPanel="True"
          RelativePanel.AlignHorizontalCenterWithPanel="True"
          Content="Button2 - Mid Top"/>

  <Button x:Name="Button3"
          Height="50"
          RelativePanel.AlignBottomWithPanel="True"
          RelativePanel.AlignRightWithPanel="True"
          Content="Button3 - Bottom Right"/>

  <Button x:Name="Button4"
          Height="50"
          RelativePanel.AlignHorizontalCenterWithPanel="True"
          RelativePanel.AlignVerticalCenterWithPanel="True"
          Content="Button4 - Center"/>

  <Button x:Name="Button5"
          Height="50"
          RelativePanel.RightOf="Button4"
          RelativePanel.Below="Button4"
          Content="Button5 - Bottom right from Button4"/>
</RelativePanel>
```

На рис. 3.29 подано приклад застосування віджета RelativePanel. Елемент керування можна вирівняти за будь-якою стороною RelativePanel, використовуючи, наприклад, властивості RelativePanel.AlignBottomWithPanel і RelativePanel.AlignRightWithPanel. Можна також використовувати властивості RelativePanel.RightOf і RelativePanel.Below, щоб розмістити елемент керування в нижньому правому куті іншого елемента керування.

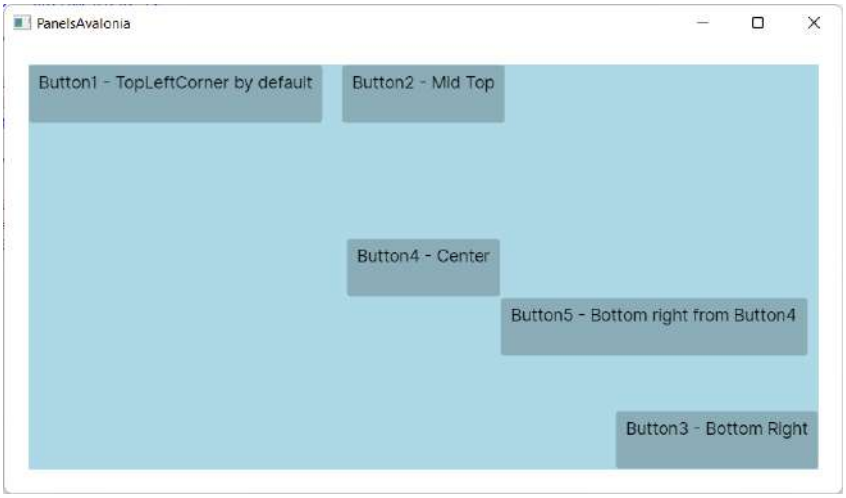


Рисунок 3.29 – Приклад застосування віджета RelativePanel

Існує декілька властивостей для визначення правил вирівнювання елемента відносно сусідніх контролів. В таблиці 3.1 подані варіанти властивостей для різних ситуацій вирівнювання елементів на екрані.

Таблиця 3.1 – Варіанти властивостей для різних ситуацій вирівнювання елементів на екрані

Вирівнювання відносно панелі	Вирівнювання відносно сусідніх елементів	Позиція елемента
AlignTopWithPanel	AlignTopWith	Above
AlignBottomWithPanel	AlignBottomWith	Below
AlignLeftWithPanel	AlignLeftWith	LeftOf
AlignRightWithPanel	AlignRightWith	RightOf
AlignHorizontalCenterWithPanel	AlignHorizontalCenterWith	
AlignVerticalCenterWithPanel	AlignVerticalCenterWith	

3.4 Контрольні запитання та завдання

1. Назвіть основні елементи керування, які застосовуються для створення графічного інтерфейсу в Avalonia.

2. Яку функцію виконує віджет TextBlock? Які властивості має елемент керування TextBlock?

3. Чим відрізняється елемент TextBox від TextBlock? Які властивості має елемент керування TextBox?

4. Наведіть приклад використання віджета Button.

5. Для чого використовується віджет ListBox?

6. Яку функцію виконує ComboBox в інтерфейсі користувача? Яким чином можна створити даний елемент? Наведіть приклад використання.

7. Наведіть приклад використання віджета ContextMenu.

8. Яким чином відбувається робота з вікнами в Avalonia?

9. Для чого використовуються контейнери в Avalonia? Наведіть приклад застосування контейнеру Grid.

4 ВЗАЄМОДІЯ МІЖ ВІЗУАЛЬНИМИ ЕЛЕМЕНТАМИ В AVALONIA

4.1 Прив’язка даних в Avalonia

4.1.1 Основні відомості

В Avalonia так само, як й у WPF прив’язка (binding) є потужним інструментом програмування, без якого не обходиться жодна серйозна програма.

Прив’язка передбачає взаємодію двох об’єктів: джерела та приймача. Об’єкт-приймач створює прив’язку до певної властивості об’єкта-джерела. В разі модифікації об’єкта-джерела, об’єкт-приймач також буде модифікований. Наприклад, найпростіша форма з використанням прив’язки:

```
<Window x:Class="BindingApp.MainWindow"
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      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"
      xmlns:local="clr-namespace:BindingApp"
      mc:Ignorable="d"
      Title="MainWindow" Height="250" Width="300">
      <StackPanel>
            <TextBox x:Name="myTextBox" Height="30" />
            <TextBlock x:Name="myTextBlock"
            Text="{Binding ElementName=myTextBox, Path=Text}"
            Height="30" />
      </StackPanel>
</Window>
```

Результат запуска програми подано на рис. 4.1. Поведінка програми наступна: при введенні будь якого тексту в поле TextBox, його вміст дублюється в віджеті TextBlock. На рис. 4.1 можна бачити повтор слова «Binding» в обох віджетах.

Прив’язка – надзвичайно потужна концепція, яка дозволяє зв’язати дві властивості так, що коли одна з них змінюється, змінюється й інша. Зазвичай прив’язування працює від вихідної властивості до цільової властивості – звичайне прив’язування OneWay, але існує також двостороннє прив’язування,

яке забезпечує синхронізацію двох властивостей залежно від змін. Є ще два режими прив’язки: OneWayToSource і OneTime, які використовуються значно рідше. Є також менш обговорювані, але настільки ж важливі прив’язки колекції, де одна колекція буде імітувати іншу або дві колекції будуть імітувати одна одну.

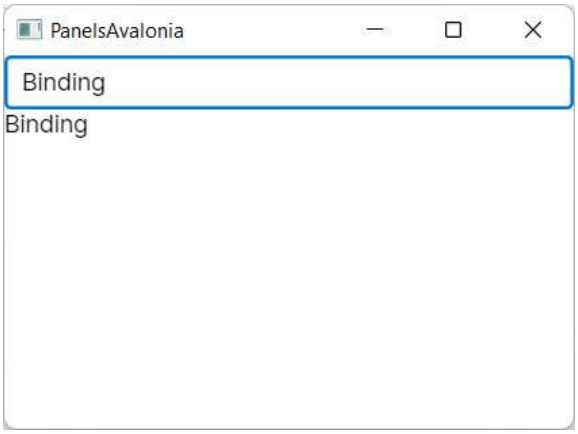


Рисунок 4.1 – Приклад форми з використанням прив’язки

Ціль прив’язки не обов’язково повинна бути такою самою, як і джерело прив’язки, може використовуватися перетворення між джерелом і цільовим елементом і навпаки, як буде показано нижче.

Прив’язка є основною концепцією шаблону MVVM. Як було зазначено в попередніх розділах, основна ідея шаблону MVVM полягає в тому, що складні візуальні об’єкти імітують властивості та поведінку дуже простих невізуальних об’єктів – так званих моделей перегляду (VM). Через це більшу частину бізнес-логіки можна розробити та перевірити на простих невізуальних об’єктах, а потім передати за допомогою прив’язок до дуже складного візуального об’єкта, який автоматично діятиме подібним чином.

Прив’язки Avalonia значно потужніші, менш помилкові та простіші у використанні, ніж прив’язки WPF – причина в тому, що вони були створені набагато пізніше, коли вже стали відомі недоліки існуючої технології, враховуючи останні досягнення в теорії розробки програмного забезпечення на основі технології Reactive Extensions.

Ще одна перевага прив’язок Avalonia полягає в тому, що на відміну від багатьох інших функцій Avalonia, вони досить добре задокументовані в Avalonia Data Bindings Documentation.

4.1.2 Концепція Avalonia Binding

Avalonia Binding – це складний об'єкт з багатьма можливостями. Прив'язки Avalonia (і WPF) найкраще пояснюються на рис. 4.2.

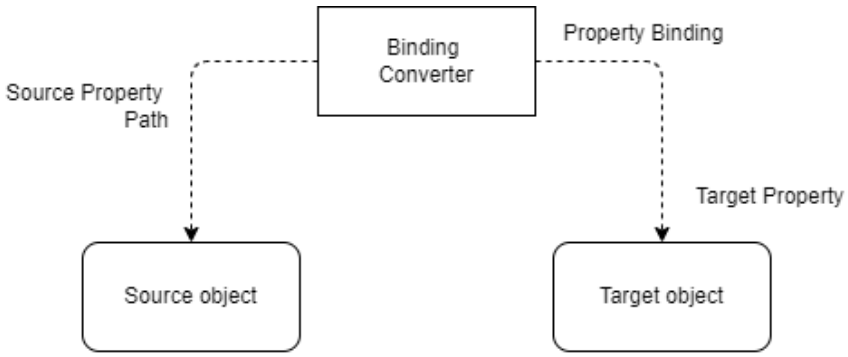


Рисунок 4.2 – Концепція Avalonia Binding

Розглянемо наступні параметри Avalonia Binding.

Об'єкт джерела прив'язки (*Binding Source Object*) – об'єкт, за допомогою якого можна отримати шлях до властивості джерела прив'язки.

Цільовий об'єкт прив'язки (*Binding Target Object*) – об'єкт, властивість якого (AvaloniaProperty) (прикріплена властивість (Attached), стиль (Style) або посилання (Direct)) служить цільовим елементом для прив'язки. Цільовий об'єкт може бути лише класом, похідним від AvaloniaObject (що означає, що це може бути будь-який з візуальних елементів Avalonia). AvaloniaObject є аналогом DependencyObject у WPF.

Шлях прив'язки (*Binding Path*) – шлях від вихідного об'єкта до вихідної властивості. Шлях складається з посилань шляху, кожне з яких може бути звичайною (C#) властивістю або властивістю Avalonia. У прив'язках XAML властивості Avalonia мають бути в дужках. Ось приклад шляху прив'язки в XAML:

```
MyProp1.(local:AttachedProperties.AttachedProperty1).MyProp2.
```

Цей шлях означає знайти властивість C# MyProp1 у вихідному об'єкті, потім знайти приєднану властивість AttachedProperty1 (визначене в класі AttachedProperties локального простору імен) в об'єкті, що повертається першим посиланням, а потім знайти властивість C# MyProp2 у цьому приєднаному значенні властивості.

Цільова властивість (*Target property*) – може бути лише одним із типів Attached, Style або Direct Property.

Тип прив'язки (*BindingMode*) може бути:

– односторонній (OneWay) – від джерела до цілі;

– двосторонній (TwoWay) – в обидва напрямки, коли вихідний об'єкт або цільовий змінено, інший також буде оновлено;

– OneWayToSource – коли цільовий об'єкт оновлюється, вихідний також оновлюється, але не навпаки.

– OneTime – синхронізує цільовий об'єкт з вихідним лише один раз – під час ініціалізації.

– за замовчуванням – залежить від бажаного режиму зв'язування цільової властивості. Коли властивості Attached, Style або Direct ініціалізовані, можна вказати бажаний режим прив'язки, який буде використовуватися в цьому випадку (коли BindingMode не вказано в самій прив'язці).

Конвертер (*Converter*) – потрібен лише в тому випадку, якщо вихідне та цільове значення відрізняються. Він використовується для перетворення значень з джерела в цільове і навпаки. Для звичайних прив'язок конвертер повинен реалізувати інтерфейс IValueConverter.

Існує також так зване багато-вихідне зв'язування (MultiBinding) як в Avalonia, так і в WPF. MultiBinding передбачає декілька джерел прив'язування і все ще одну й ту саму ціль зв'язування. Кілька джерел об'єднуються в одну ціль за допомогою спеціального перетворювача, який реалізує IMultiValueConverter у разі багатовихідного зв'язування.

Однією зі складних частин зв'язування є те, що існує кілька способів вказати вихідний об'єкт як в Avalonia, так і в WPF, але Avalonia має більше способів це зробити. Ось опис різних методів визначення вихідного об'єкта:

1. Якщо взагалі не вказано вихідний об'єкт – у такому випадку вихідним об'єктом за замовчуванням для цілі Binding буде призначено властивість DataContext. DataContext автоматично поширюється вниз по візуальному дереву, якщо його не змінити явно (за деякими винятками).

2. Можна вказати джерело явно в XAML, призначивши його Source властивості. Також можна призначити його безпосередньо в коді C# або в XAML, якщо використовувати розширення розмітки StaticResource.

3. Існує властивість ElementName, яку можна використовувати, щоб знайти вихідний елемент у тому самому файлі XAML за ім'ям.

4. Існує властивість RelativeSource, яка відкриває ще кілька цікавих способів місцезнаходження вихідного об’єкта залежно від його властивості Mode:

– для «Mode = Self» вихідний об’єкт буде таким самим, як цільовий об’єкт;

– «Mode = TemplatedParent» можна використовувати лише в ControlTemplate деяких Avalonia TemplatedControl. TemplatedParent у шаблоні елемента керування означає, що джерелом прив’язки є елемент керування, для якого використовується шаблон;

– «Mode = FindAncestor» означає, що пошук вихідного об’єкта буде відбуватися у «Візуальному дереві» (Visual Tree). Також слід використовувати властивість AncestorType в цьому режимі, щоб вказати тип вихідного об’єкта для пошуку. Якщо нічого іншого не вказано, перший об’єкт цього типу стане вихідним. Якщо також AncestorLevel встановлено на якесь додатне ціле число N, це вказує, що об’єкт N-го предка цього типу буде повернуто (за замовчуванням AncestorLevel == 1) як джерело для прив’язки.

У Avalonia (але не в WPF) властивість Tree RelativeSource можна встановити на TreeType.Logical (за замовчуванням це TreeType.Visual). У цьому випадку пошук предків здійснюється вгору по логічному дереву (яке є рідкішим і менш складним).

Для більшого розуміння теоретичного матеріалу розглянемо декілька практичних прикладів.

4.1.3 Прив’язка за замовчуванням (DataContext (default))

Якщо джерело не вказано в прив’язці, джерело Binding посилається до властивості DataContext елемента. У нашому прикладі DataContext встановлено у вікні, але оскільки він поширюється вниз по візуальному дереву (якщо не змінено явно), наш TextBlock має той самий DataContext, який є просто рядком, який відображається нашим TextBlock. Для демонстрації даного способу зв’язування використаємо наступний код XAML:

```
<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"
        mc:Ignorable="d" d:DesignWidth="320" d:DesignHeight="150"
```

```
        DataContext="This is the Window’s DataContext"
        x:Class="PanelsAvalonia.MainWindow"
        Title="PanelsAvalonia">
<Grid RowDefinitions="Auto, Auto, *" Margin="5">
      <TextBlock Text="Example #1 &#10;
DataContext (default) Binding Source:"
            HorizontalAlignment="Left"
            Margin="10,3"
            FontWeight="Bold"/>
      <TextBlock Text="{}Text={Binding}"
            FontSize="11"
            Grid.Row="1"
            Margin="10,3"
            HorizontalAlignment="Left"/>
      <TextBlock Text="{Binding}"
            Grid.Row="2"
            HorizontalAlignment="Center"
            VerticalAlignment="Bottom"
            Margin="0,0,0,20"/>
</Grid>
</Window>
```

В даному коді необхідно відмітити два важливих рядки (виділені напівжирним текстом):

```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        ...
        DataContext="This is the Window’s DataContext"
        ...>
        ...
        <Grid ...>
            <TextBlock Text="{Binding}"/>
        </Grid>
        ...
</Window>
```

В наведеному фрагменті виділені рядки коду:

– із визначенням параметру «DataContext» в розділі заголовку файлу;

– із використанням об’єкта за замовчуванням «"Text={Binding}"».

На рис. 4.2 подано приклад використання прив’язки за замовчуванням.

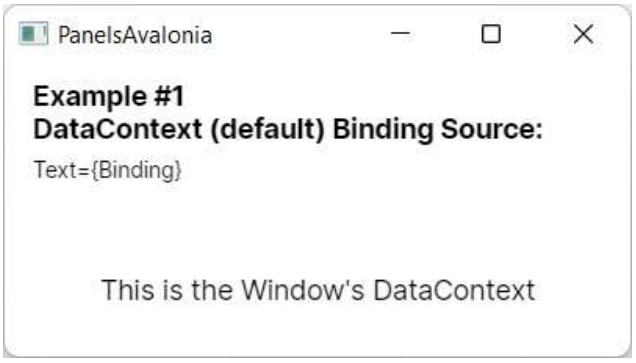


Рисунок 4.2 – Приклад використання прив’язки за замовчуванням

Текстовий блок в нижній частині форми після запуску відображає текст, що міститься в DataContext.

Повний код прикладу подано в лістингу «Listing_4-01».

4.1.4 Прив’язка за назвою елементу (Binding by ElementName)

Існує спосіб прив’язки до елемента за його назвою та властивістю «Tag». Тег – це властивість, що визначена для кожного елемента керування Avalonia, який може містити будь-який об’єкт. Елемент може мати назву. Наприклад, головний об’єкт – наше вікно у файлі XAML, має назву– «TheWindow», і ми використовуємо його для прив’язки до його тегу:

```
<Window ...
        Tag="This is the Window’s Tag"
        x:Name="TheWindow"
        ...>
        ...
             <TextBlock Text="{Binding #TheWindow.Tag}"
                        .../>
        ...
</Window>
```

Наведена вище конструкція

```
<TextBlock Text="{Binding #TheWindow.Tag}"
```

є скороченням Avalonia для

```
Text={Binding Path=Tag, ElementName=TheWindow}.
```

На рис. 4.3 подано приклад використання прив’язки за назвою елемента.

Рисунок 4.3 – Приклад використання прив’язки за назвою елемента

Повний код прикладу подано в лістингу «Listing_4-02».

4.1.5 Прив’язка до ресурсу (Binding.Source)

В наступному прикладі ми використовуємо розширення розмітки StaticResource, щоб встановити джерело прив’язки до рядка «This is the Window’s resource», визначеного як ресурс вікна:

```
<Window.Resources>
      <x:String x:Key="TheResource">
            This is the Window’s resource</x:String>
</Window.Resources>
<Grid RowDefinitions="Auto, Auto, *" Margin="5">
      <TextBlock Text="Example #3 &#10;Binding.Source:"
            HorizontalAlignment="Left"
            Margin="10,3"
            FontWeight="Bold"/>
      <TextBlock Text="{}
            Text={Binding Source={StaticResource TheResource}}"
            FontSize="11"
            Grid.Row="1"
            Margin="10,3"
            HorizontalAlignment="Left"/>
      <TextBlock Text="{Binding Source={StaticResource TheResource}}"
            Grid.Row="2"
            HorizontalAlignment="Center"
            VerticalAlignment="Bottom"
            Margin="0,0,0,20"/>
</Grid>
```

На рис. 4.4 подано приклад використання прив’язки до ресурсу.

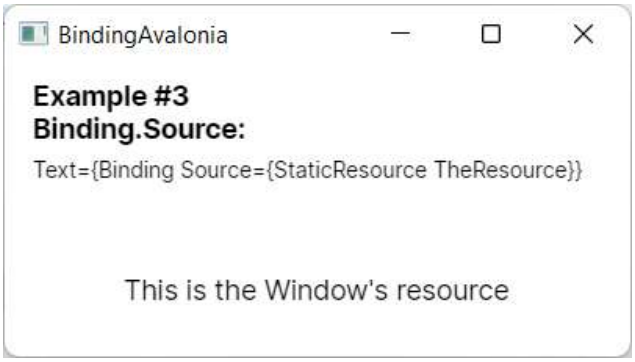


Рисунок 4.4 – Приклад використання прив’язки до ресурсу

Повний код прикладу подано в лістингу «Listing_4-03».

4.1.6 Прив’язка до самого себе за допомогою RelativeSource

Цей приклад показує, як елемент може використовувати сам себе, як вихідний об’єкт Binding за допомогою RelativeSource у режимі Self:

```
<Grid RowDefinitions="Auto, Auto, *" Margin="5">
      <TextBlock Text="Example #4 &#10;
            Binding with RelativeSource Mode=Self:"
            HorizontalAlignment="Left"
            Margin="10,3"
            FontWeight="Bold"/>
      <TextBlock Text="{}Text={Binding Path=Tag,
            RelativeSource={RelativeSource Self}}"
            FontSize="11"
            Grid.Row="1"
            Margin="10,3"
            HorizontalAlignment="Left"/>
      <TextBlock Text="{Binding Path=Tag,
            RelativeSource={RelativeSource Self}}"
            Tag="This is my own (TextBox’es) Tag"
            Grid.Row="2"
            HorizontalAlignment="Center"
            VerticalAlignment="Bottom"
            Margin="0,0,0,20"/>
</Grid>
```

На рис. 4.5 подано приклад використання прив’язки до самого себе за допомогою RelativeSource.

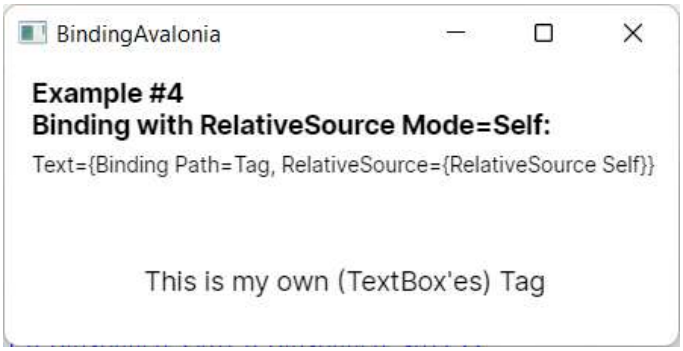


Рисунок 4.5 – Приклад використання прив’язки до самого себе за допомогою RelativeSource

Повний код прикладу подано в лістингу «Listing_4-04».

4.1.7 Прив’язка до TemplatedParent

Режим RelativeSource з TemplatedParent слід використовувати лише всередині ControlTemplate, і його використання означає, що прив’язка посилається на властивість (або шлях), визначену в елементі керування, що реалізується поточним шаблоном:

```
<TemplatedControl Tag="This is Control’s Tag"
                  ...>
    <TemplatedControl.Template>
        <ControlTemplate>
            <TextBlock Text="{Binding Path=Tag,
                      RelativeSource={RelativeSource TemplatedParent}}"/>
        </ControlTemplate>
    </TemplatedControl.Template>
</TemplatedControl>
```

Наведений вище код означає, що ми прив’язуємось до властивості Tag у TemplatedControl, яка реалізована ControlTemplate.

На рис. 4.6 подано приклад використання прив’язки до TemplatedParent.

Повний код прикладу подано в лістингу «Listing_4-05».

Рисунок 4.6 – Приклад використання прив’язки до TemplatedParent

4.1.8 Прив’язка до предка візуального дерева за допомогою RelativeSource з AncestorType

Визначення AncestorType означатиме для прив’язки, що RelativeSource перебуває в режимі FindAncestor:

```
<Grid ...
    Tag="This is the first Grid ancestor tag"
    ...>
    <StackPanel>
        <TextBlock Text="{Binding Path=Tag,
                   RelativeSource={RelativeSource AncestorType=Grid}}"/>
    </StackPanel>
</Grid>
```

На рис. 4.7 подано приклад використання прив’язки до предка візуального дерева за допомогою RelativeSource з AncestorType.

Рисунок 4.7 – Приклад використання прив’язки до предка візуального дерева за допомогою RelativeSource з AncestorType

Повний код прикладу подано в лістингу «Listing_4-06».

4.1.9 Прив'язка до передка візуального дерева за допомогою RelativeSource з AncestorType та AncestorLevel

Використовуючи AncestorLevel, можна вказати, що нам потрібен не перший предок потрібного типу, а N-й – де N може бути будь-яким натуральним числом. У коді нижче ми шукаємо другу сітку серед предків елемента:

```
<Grid ...
    Tag="This is the second Grid ancestor tag">
    <StackPanel>
        <Grid Tag="This is the first Grid ancestor tag">
            <StackPanel>
                <TextBlock Text="{Binding Path=Tag,
                 RelativeSource={RelativeSource AncestorType=Grid,
                        AncestorLevel=2}}"/>
            </StackPanel>
        </Grid>
    </StackPanel>
</Grid>
```

На рис. 4.8 подано приклад використання прив'язки до предка візуального дерева за допомогою RelativeSource з AncestorType та AncestorLevel.

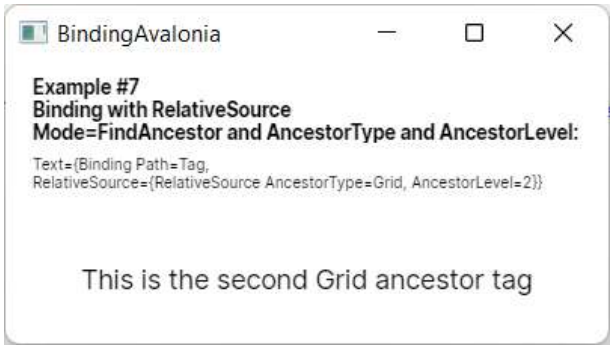


Рисунок 4.8 – Приклад використання прив'язки до предка візуального дерева за допомогою RelativeSource з AncestorType та AncestorLevel

Повний код прикладу подано в лістингу «Listing_4-07».

4.1.10 Використання скорочення Avalonia Binding Path для пошуку батьківського елемента в логічному дереві

Для скороченого запису прив'язки до батьківського елемента використовується на ступна конструкція коду:

```
<Grid Tag="This is the first Grid ancestor tag">
  <StackPanel Tag="This is the immediate ancestor tag">
      <TextBlock Text="{Binding $parent.Tag}"/>
  </StackPanel>
</Grid>
```

Зверніть увагу, що $parent.Tag означає знайти батьківського (першого предка) елемента та отримати від нього властивість Tag. Ця прив’язка має бути еквівалентною довшій версії:

```
<TextBlock Text="{Binding Path=Tag,
 RelativeSource={RelativeSource Mode=FindAncestor, Tree=Logical}}">
```

На рис. 4.9 подано приклад використання скорочення Avalonia Binding Path для пошуку батьківського елемента в логічному дереві.

Рисунок 4.9 – скорочення Avalonia Binding Path для пошуку батьківського елемента в логічному дереві

Повний код прикладу подано в лістингу «Listing_4-08».

4.1.11 Використання Avalonia Binding Path Shorthand для пошуку першого батьківського елемента в логічному дереві

Для прикладу розглянемо ситуацію пошуку першого батьківського елемента типу Grid. Нижче наведено фрагмент коду для демонстрації даного методу прив’язки:

```
<ContentControl Tag="This is the second logical tree ancestor tag"
   HorizontalAlignment="Center"
```

```
        VerticalAlignment="Bottom"
        Margin="0,0,0,20"
        Grid.Row="2">
        <Grid Tag="This is the second Grid ancestor tag">
            <StackPanel>
                <Grid Tag="This is the first Grid ancestor tag">
                    <StackPanel Tag="this is the immediate ancestor tag">
                        <Button Tag="This is the first logical tree ancestor tag">
                            <TextBlock Text="{Binding $parent[Grid].Tag}"/>
                        </Button>
                    </StackPanel>
                </Grid>
            </StackPanel>
        </Grid>
    </ContentControl>
```

З наведеного коду XAML можна бачити, що для пошуку елемента використовується запис:

```
$parent[Grid].Tag.
```

Серед набору тегів обрано той, що відноситься до об'єкта «Grig» та розміщується першим при проходженні дерева елементів знизу в гору, починаючи з місця використання прив'язки.

На рис. 4.10 подано приклад пошуку першого батьківського елементу типу Grid.

Повний код прикладу подано в лістингу «Listing_4-09».

Рисунок 4.10 – Приклад пошуку першого батьківського елемента типу Grid

4.1.12 Прив'язка до сітки Grid другого предка в логічному дереві за допомогою скорочення Avalonia Binding Path

Для вибору потрібного елементу в логічному дереві використовується порядковий номер того

```
[1] <Grid Tag="This is the second Grid ancestor tag">
    <StackPanel>
[0]   <Grid Tag="This is the first Grid ancestor tag">
        <StackPanel Tag="this is the immediate ancestor tag">
          <Button Tag="This is the first logical tree ancestor tag">
          <TextBlock Text="{Binding $parent[Grid;1].Tag}"/>
          </Button>
        </StackPanel>
      </Grid>
    </StackPanel>
  </Grid>
```

Цифрами в квадратних дужках відмічені рядки елементів Grid в порядку їх визначення в логічному дереві. В нашому варіанті використовується рядок з номером «1», що задано в записі:

```
<TextBlock Text="{Binding $parent[Grid;1].Tag}"/>
```

На рис. 4.11 подано приклад прив'язки до сітки Grid другого предка в логічному дереві.

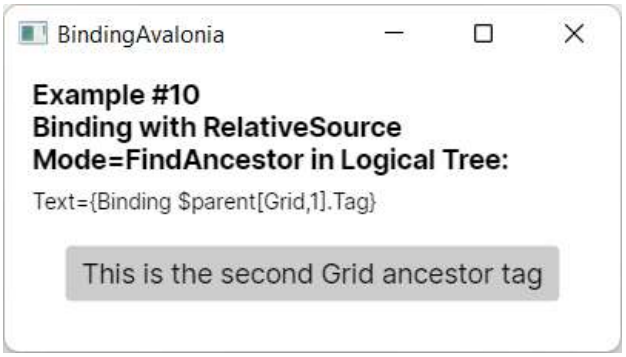


Рисунок 4.11 – Приклад прив'язки до сітки Grid другого предка в логічному дереві

Повний код прикладу подано в лістингу «Listing_4-10».

4.2 Візуальне дерево елементів

В попередньому розділі часто використовувався термін «Логічне дерево». Розглянемо більш детально його особливості.

Вивчивши попередні розділи можна зазначити, шо основні будівельні блоки Avalonia (і WPF) (примітиви) складаються з:

– примітивних елементів – основні елементи, такі як TextBlock, Border, Path, Image, Viewbox тощо, які не можуть бути розбиті на субелементи в концепції Avalonia.

– панелей – елементи, що відповідають за розміщення інших елементів всередині них.

Решта елементів керування (включаючи такі основні елементи керування, як, наприклад, Button, ComboBox, Menu тощо) та складні представлення створюються шляхом об’єднання різних примітивів разом із розміщенням їх у інших примітивах чи панелях. В Avalonia примітиви зазвичай успадковуються від класу Control, тоді як складніші елементи керування успадковуються від класу TemplatedControl. В свою чергу в WPF примітиви успадковуються від Visual, а складніші елементи керування успадковуються від Control (Control у WPF має властивість Template та пов’язану інфраструктуру, а в Avalonia вони мають TemplatedControl).

Композиція візуального об’єкта Avalonia (і WPF) може бути ієрархічною: на початку створюються деякі простіші об’єкти з примітивів, а потім створюються більш складні об’єкти з цих простіших об’єктів (і, можливо, також примітивів) тощо. Цей принцип ієрархічної композиції один із основних способів повторного використання візуальних компонентів.

Розглянемо приклад створення композиційного візуального елементу на прикладі простої кнопки. Наша кнопка буде мати зображення та текст. Зображення розмістимо зліва від текстової мітки. На рис. 4.12 подано фінальний результат нашої кнопки. Щоб створити такий елемент інтерфейсу, нам необхідно виконати ряд підготовчих дій. Вони були описані в попередніх розділах, але згадаємо їх ще раз.

Перше, що треба зробити – це додати новий ресурс (зображення) до нашого проєкту. Для цього створимо нову папку із назвою «Assets» – натискаємо правою кнопкою миші на назві проєкту «LogicalTree» та обираємо опцію «Add».

Рисунок 4.12 – Композиційний візуальний
елемент «Кнопка»

Далі обираємо «New Folder» та вводимо назву нової папки «Assets». Структура нашого проєкту повинна стати такою (рис. 4.13).

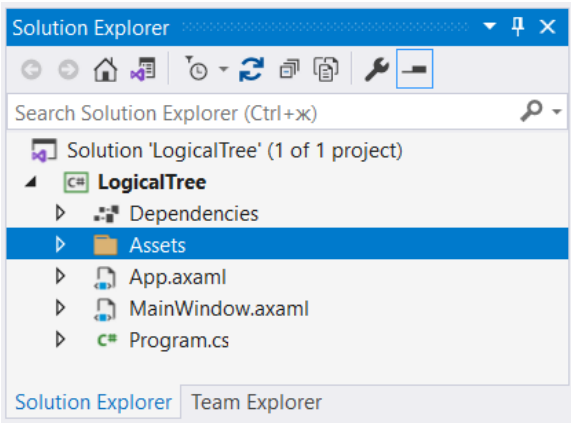


Рисунок 4.13 – Додавання нової папки «Assets» до проєкту

Наступним кроком додаємо в папку «Assets» необхідне зображення. В нашому прикладі використовується зображення папки в форматі .png (рис. 4.14).

Рисунок 4.14 – Зображення папки в форматі .png

Зображення можна додати простим копіюванням через буфер обміну. Після даної операції структура нашого проєкту буде виглядати як на рис. 4.15.

Для того, щоб доданим зображенням можна було користуватись, в проєкті необхідно додати посилання на папку «Assets» проєкту.

Для цього в файл «LogicalTree.csproj» вставимо такі рядки коду:

```
<ItemGroup>
    <AvaloniaResource Include="Assets\**" />
</ItemGroup>
```

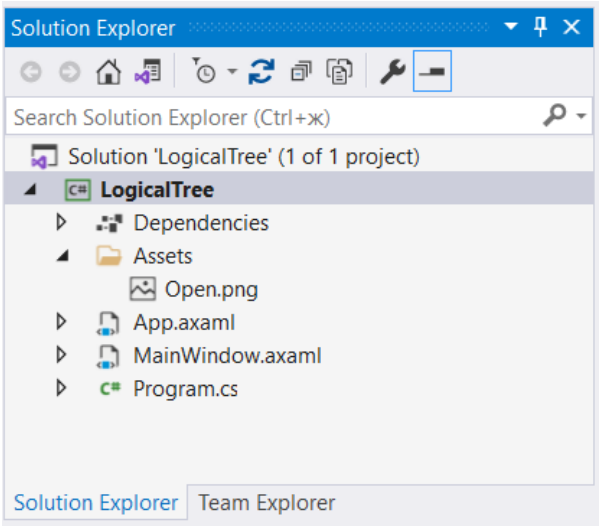


Рисунок 4.15 – Структура проєкту після додавання зображення папки

Код звичайної кнопки виглядає наступним чином:

```
<Button
      HorizontalAlignment="Center"
      VerticalAlignment="Center">
      Open Folder
</Button>
```

В нашому прикладі нам треба додати до тексту ще й зображення. Для цього нам потрібно змінити вміст тегу «Button» та використати контейнер «Grid» для розміщення додаткових елементів: зображення та текстової мітки. Новий код виглядає наступним чином:

```
      <Button
            HorizontalAlignment="Center"
            VerticalAlignment="Center">
            <Grid
                  ColumnDefinitions="Auto, Auto"
                  RowDefinitions="Auto">
                  <Image
```

```
                Source="/Assets/Open.png"
                Grid.Column="0"/>
            <TextBlock
                Margin="10,0,0,0"
                Grid.Column="1">
                Open Folder
            </TextBlock>
        </Grid>
    </Button>
```

На рис. 4.16 подано структуру нового композиційного елементу.

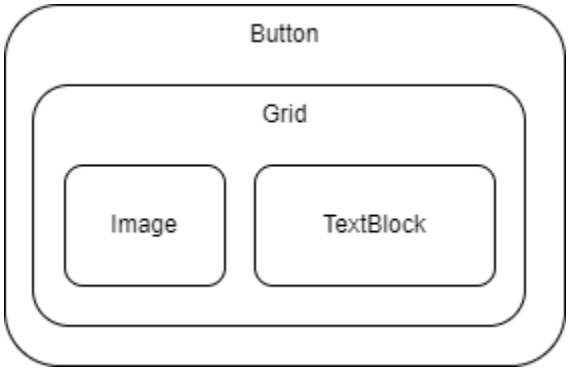


Рисунок 4.16 – Структура нового композиційного елементу

Нова кнопка складається з кількох примітивних елементів: з панелі Grid, яка має об’єкт Image для значка кнопки та TextBlock для тексту кнопки. Ця структура об’єктів визначає візуальне дерево компонентів (рис. 4.17).

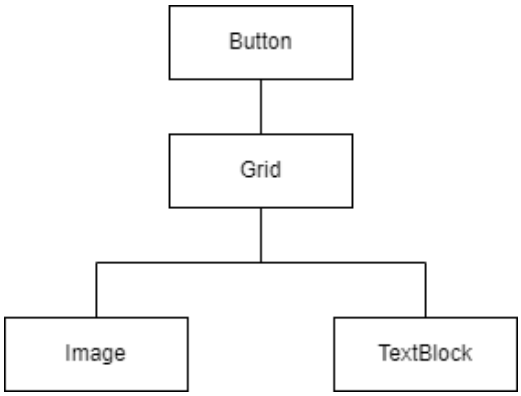


Рисунок 4.17 – Візуальне дерево компонентів

Звичайно, візуальні дерева реальних кнопок можуть бути складнішими: включати рамки та тіні для компоненту, містити накладну панель або такі панелі,

що змінюють прозорість або колір, коли курсор миші знаходиться на кнопці, щоб вказати, що ця кнопка активна при натисканні миші та багато інших речей.

Теми, що зазначені в проєкті, визначають вигляд і поведінку всіх основних елементів керування, включаючи, звичайно, кнопки. Вони роблять це за допомогою стилів і шаблонів. Важливо розуміти, що візуальне дерево кнопки визначається шаблоном ControlTemplate кнопки, розташованим у стилі кнопки, який, у свою чергу, розташований у FluentTheme. В нашому проєкті використовується тема «Light». Посилання на неї зустрічається в файлі App.axaml:

```
<Application.Styles>
    <FluentTheme Mode="Light"/>
</Application.Styles>
```

Avalonia має інструмент для дослідження візуальних та логічних дерев. Щоб його використати необхідно натиснути на вікно відкомпільованої програми, щоб отримати фокус миші, і натиснути клавішу F12. Відкриється вікно інструментів Avalonia для розробника (рис. 4.18).

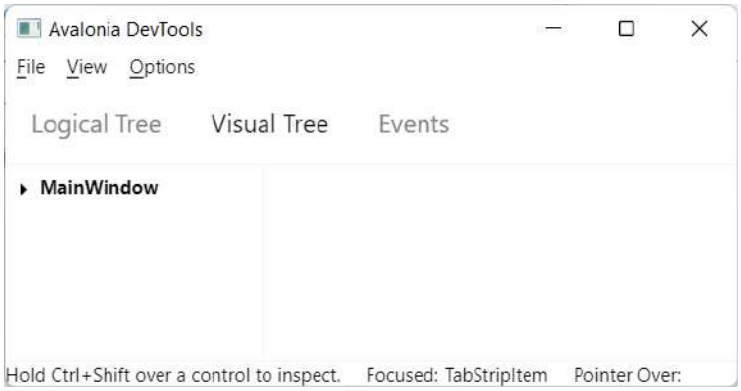


Рисунок 4.18 – Вікно інструментів Avalonia для розробника

Вікно інструментів дає нам можливість досліджувати будь-яку властивість або подію для будь-якого елемента у візуальних або логічних деревах.

Перевага інструменту полягає в тому, що він також написаний на Avalonia і тому є багатоплатформним. Він однаково відображатиметься в MacOS і Linux.

Інструмент показує лише інформацію, що відповідає одному вікну, тому, якщо використовується кілька вікон, дерева та властивості яких необхідно дослідити, доведеться використовувати кілька вікон інструментів.

Інструмент не відображатиметься для конфігурації, яка не має набору змінних препроцесора DEBUG, наприклад, конфігурацію випуску за замовчуванням. Фактично, наступні рядки з конструктора MainWindow (розташованого у файлі MainWindow.axaml.cs) створюють можливість запуску інструмента:

```
#if DEBUG
      this.AttachDevTools();
#endif
```

Крім того, якщо інструмент не потрібен, після видалення виклику this.AttachDevTool(), також можна видалити пакет Avalonia.Diagnostics зі своїх посилань.

У Avalonia логічне дерево відіграє більшу роль, ніж у WPF, тому за замовчуванням інструмент показує логічне дерево, і щоб перейти до візуального дерева, потрібно натиснути на вкладку «Візуальне дерево» (Visual Tree).

Після того, як ми перемкнули інструмент для відображення візуального дерева, покажемо курсором миші на зображення папки у кнопці і натиснемо разом клавіші Control і Shift (рис. 4.19).

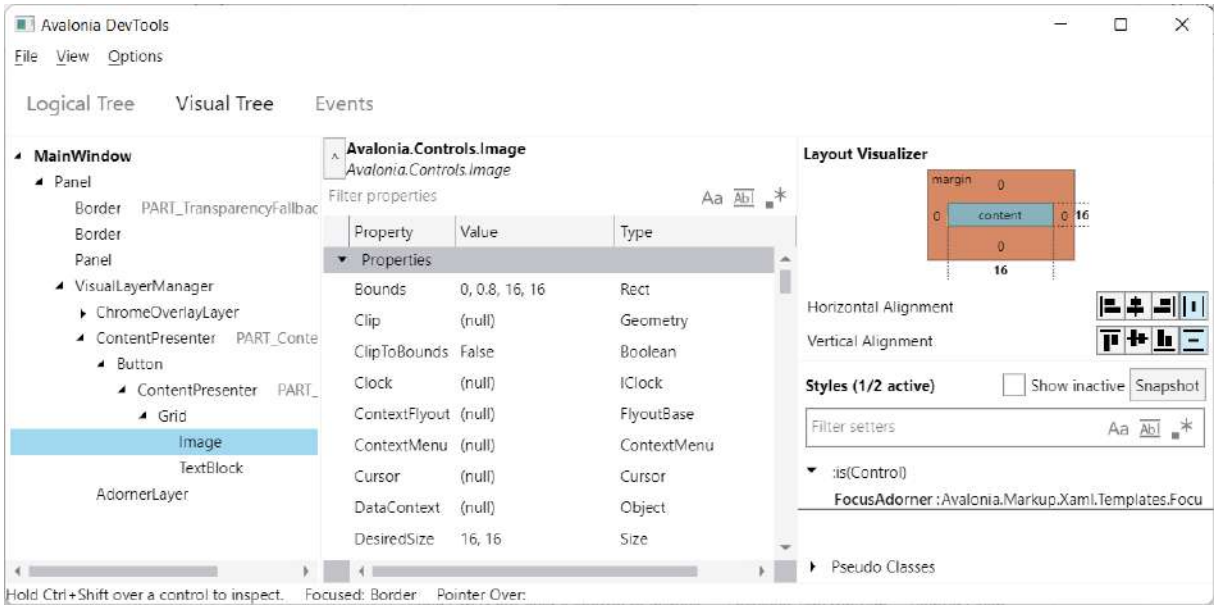


Рисунок 4.19 – Інтерактивний режим інструменту Avalonia DevTools

Візуальне дерево з лівого боку інструмента розгорнеться до елемента, що містить вказане зображення кнопки, а панель властивостей у середині

інструмента покаже властивості поточного вибраного елемента візуального дерева (у нашому випадку це буде елемент Image (рис. 4.19). Візуальне дерево фактично відображається для всього вікна, а розгортається та його частина, що відповідає поточному вибраному елементу.

Розглянемо просту кнопку без модифікації. З рис. 4.20 можна бачити, що візуальне дерево для кнопки з FluentTheme простіше, ніж розглянуте вище – воно складається лише з трьох елементів: Button (корінний елемент), потім ContentPresenter і потім елемент TextBlock.

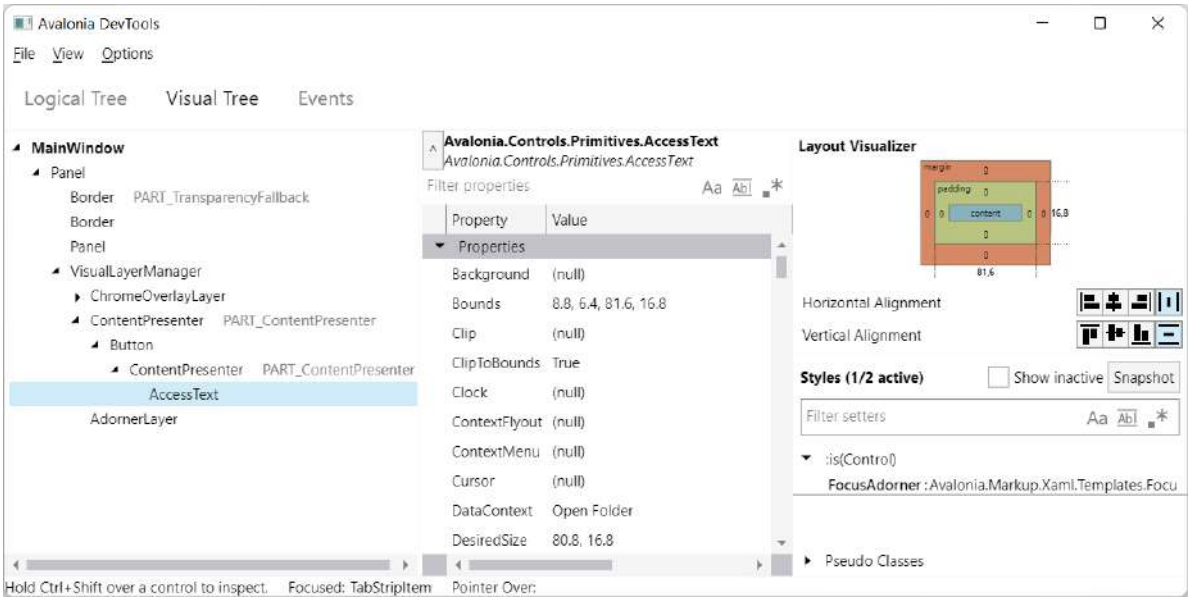


Рисунок 4.20 – Візуальне дерево для простої кнопки

Замість TextBlock можна вибрати інший елемент, наприклад, кнопку, яка є прабатьком TextBlock, щоб побачити її властивості на середній панелі інструмента. Якщо потрібно знайти конкретну властивість, наприклад, DataContext, можна ввести частину назви у верхньому полі введення таблиці властивостей, наприклад, «context», і програма відфільтрує властивості до тих, чиї назви містять слово ‘context’ (рис. 4.21).

Для дослідження властивостей візуального елемента «Кнопка» створимо простий код на C# у файлі MainWindow.xaml.cs. Зробимо обробник натискання на кнопку:

```
public MainWindow()
{
      InitializeComponent();
```

```
        this.AttachDevTools();
        _button = this.FindControl<Button>("SimpleButton");
        _button.Click += OnButtonClick;
    }
```

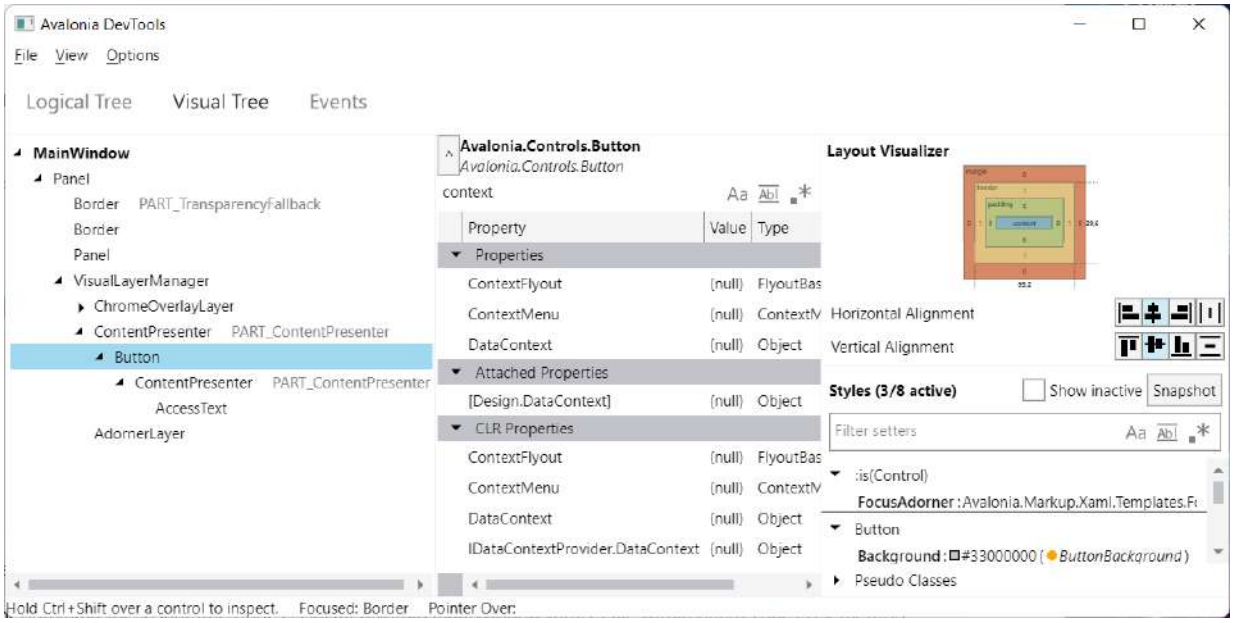


Рисунок 4.21 – Відображення властивостей за певною назвою

Саму кнопку створимо в класі MainWindow та зробимо приватною:

```
private Button _button;
```

Обробник події OnButtonClick матиме такий вигляд:

```
private void OnButtonClick(object? sender,
Avalonia.Interactivity.RoutedEventArgs e)
{
    IVisual parent = _button.GetVisualParent();
    var visualAncestors = _button.GetVisualAncestors().ToList();
    var visualChildren = _button.GetVisualChildren().ToList();
    var visualDescendants = _button.GetVisualDescendants().ToList();
}
```

Зауважимо, що для того, щоб зробити доступними методи розширення Visual Tree, ми повинні були додати використання Avalonia.VisualTree; посилання на простір імен у верхній частині файлу MainWindow.axaml.cs.

Всередині функції OnButtonClick кожен рядок відповідає візуальному дереву, яке показано на рис. 4.22.

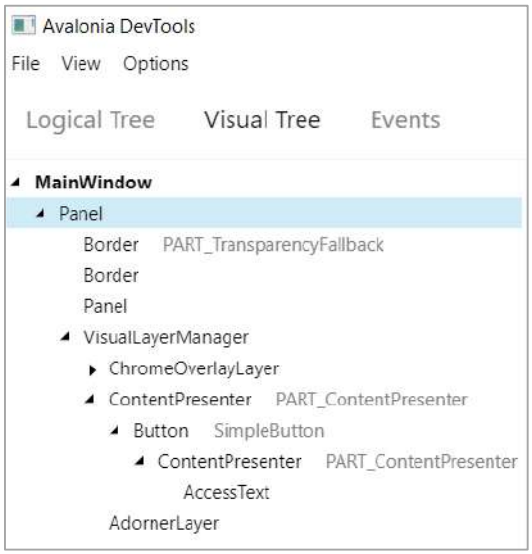


Рисунок 4.22 – Візуальне дерево

В результаті дослідження роботи програми за допомогою інструменту Debuger можна бачити вміст змінних, що задані всередині функції та відповідність їх візуальному дереву (рис. 4.23):

– батьківським елементом нашої кнопки є ContentPresenter;

– кнопка має чотирьох предків: ContentPresenter, VisualLayoutManager, Panel і Window;

– кнопка має лише один дочірній елемент – ContentPresenter;

– кнопка має двох нащадків: ContentPresenter і TextBlock.

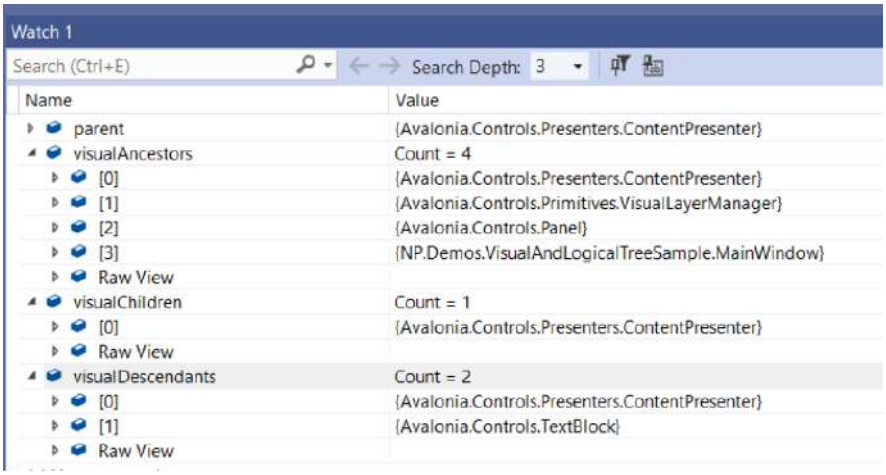


Рисунок 4.23 – Вміст змінних та відповідність їх візуальному дереву

4.3 Логічне дерево елементів

Логічне дерево є підмножиною візуального дерева – воно є менш інформативним, ніж візуальне дерево – містить менше елементів. Дерево точно відповідає коду XAML, але не розширює жодних шаблонів керування. Коли відображається ContentControl, він переходить безпосередньо від ContentControl до елемента, який представляє його Content (опускаючи все між ними). Коли відображається елемент ItemsControl, він переходить безпосередньо від елемента ItemsControl до елементів, які представляють вміст його елементів, також пропускаючи все, що знаходиться між ними.

Приклад відкритого логічного дерева для нашого тестового проєкту подано на рис. 4.24.

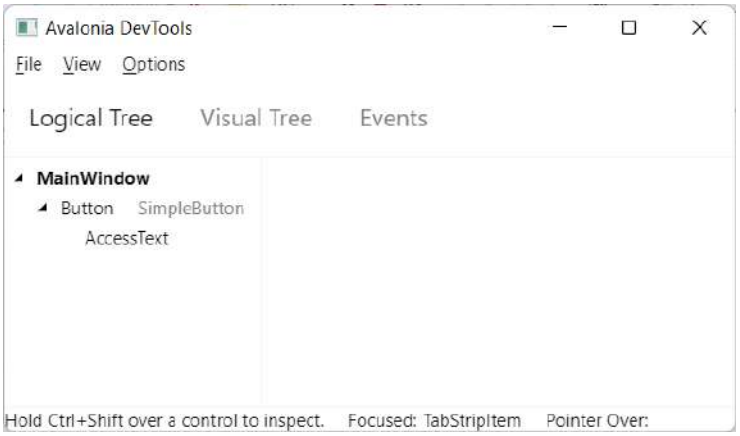


Рисунок 4.24 – Приклад відкритого логічного дерева
для програми з однією кнопкою

Для дослідження роботи із логічним деревом змінимо вміст тестового проєкту. В файлі MainWindow.xaml замість розмітки з кнопкою вставимо такий XAML код:

```
<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"
        mc:Ignorable="d" d:DesignWidth="300" d:DesignHeight="150"
        x:Class="LogicalTree.MainWindow"
        Title="LogicalTree"
```

```
            Width="300"
        Height="200">
  <Grid RowDefinitions="*, *">
    <Button x:Name="ClickMeButton"
            Content="Click Me"
            HorizontalAlignment="Center"
            VerticalAlignment="Center"/>

    <ItemsControl Grid.Row="1"
                  HorizontalAlignment="Center"
                  VerticalAlignment="Center">
      <Button x:Name="Item1Button"
              Content="Item1 Button"/>
      <Button x:Name="Item2Button"
              Content="Item2 Button"/>
    </ItemsControl>
  </Grid>
</Window>
```

Ми бачимо, що вміст вікна представлено панеллю Grid з двома рядками. У верхньому рядку міститься кнопка «Click Me», а в нижньому рядку міститься елемент ItemsControl з двома кнопками: «Item1 Button» і «Item2 Button». Назви кнопок у XAML збігаються з тими, що в них написано, тільки без пробілів: «ClickMeButton», «Item1Button» і «Item2Button» (рис. 4.25).

Рисунок 4.25 – Тестове вікно для дослідження інструменту Logical Tree

Після запуску програми та натискання клавіши F12 відкриється інструмент в якому можна відкрити вкладку «Логічне дерево» (рис. 4.26).

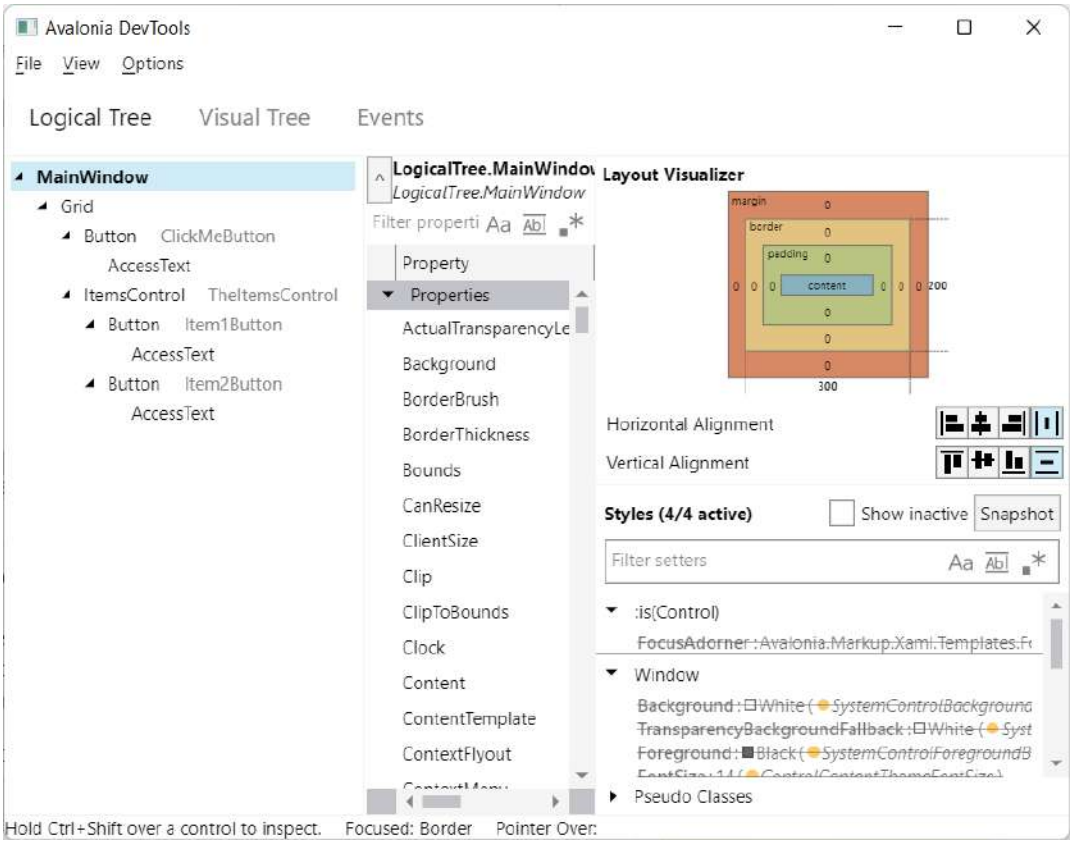


Рисунок 4.26 – Інструмент «Логічне дерево» для тестової програми

З рис. 4.26 можна побачити, що у візуальному дереві присутні лише елементи, які відповідають тегам XAML файлу MainWindow.axaml, а також AccessText у компоненті Buttons (оскільки кнопка є ContentControl, а AccessText є елементами, які представляють його вміст). Багато вузлів, які були б присутні у візуальному дереві, тут відсутні:

– немає елементів візуального дерева, які були створені через розширення шаблонів керування;

– немає меж вікна, панелей, VisualLayoutManager тощо.

Також з рис. 4.26 ми бачимо, що дерево відкривається від Window безпосередньо до Grid, оскільки елемент Grid є частиною файлу MainWindow.axaml. Аналогічно, ми відразу переходимо від кнопок прямо до TextBlock, оминаючи ContentPresenter, оскільки він походить із розширення шаблону кнопки.

Змінимо код обробника події OnButtonClick у файлі MainWindow.axaml.cs на наступний:

```
private void OnButtonClick(object? sender, RoutedEventArgs e)
{
```

```
        ItemsControl itemsControl =
           this.FindControl<ItemsControl>("TheItemsControl");

        var logicalParent = itemsControl.GetLogicalParent();
        var logicalAncestors = itemsControl.GetLogicalAncestors().ToList();
        var logicalChildren = itemsControl.GetLogicalChildren().ToList();
        var logicalDescendants =
          itemsControl.GetLogicalDescendants().ToList();
    }
```

Цей метод встановлюється як обробник події Click для «ClickMeButton» аналогічно до попереднього прикладу:

```
Button clickMeButton = this.FindControl<Button>("ClickMeButton");
clickMeButton.Click += OnButtonClick;
```

В результаті, ми отримуємо доступ до логічних елементів вікна: батька, предків, дітей і нащадків елемента ItemsControl. Також треба зауважити, що для того, щоб отримати ці методи розширення, ми повинні були додати використання простору імен Avalonia.LogicalTree у верхній частині файлу MainWindow.xaml.cs.

Таким чином, якщо увімкнути режим налагодження програми, ми зможемо дослідити логічну структуру робочого вікна програми. На рис. 4.25 подано вигляд вікна перегляду вмісту змінних у Visual Studio.

Рисунок 4.25 – Вікно перегляду вмісту змінних у Visual Studio

Ще однією можливістю вбудованого інструменту Avalonia є дослідження подій. Для цього використовується відповідна вкладка Events. На рис. 4.26 подано приклад перехоплення подій із кнопкою миші.

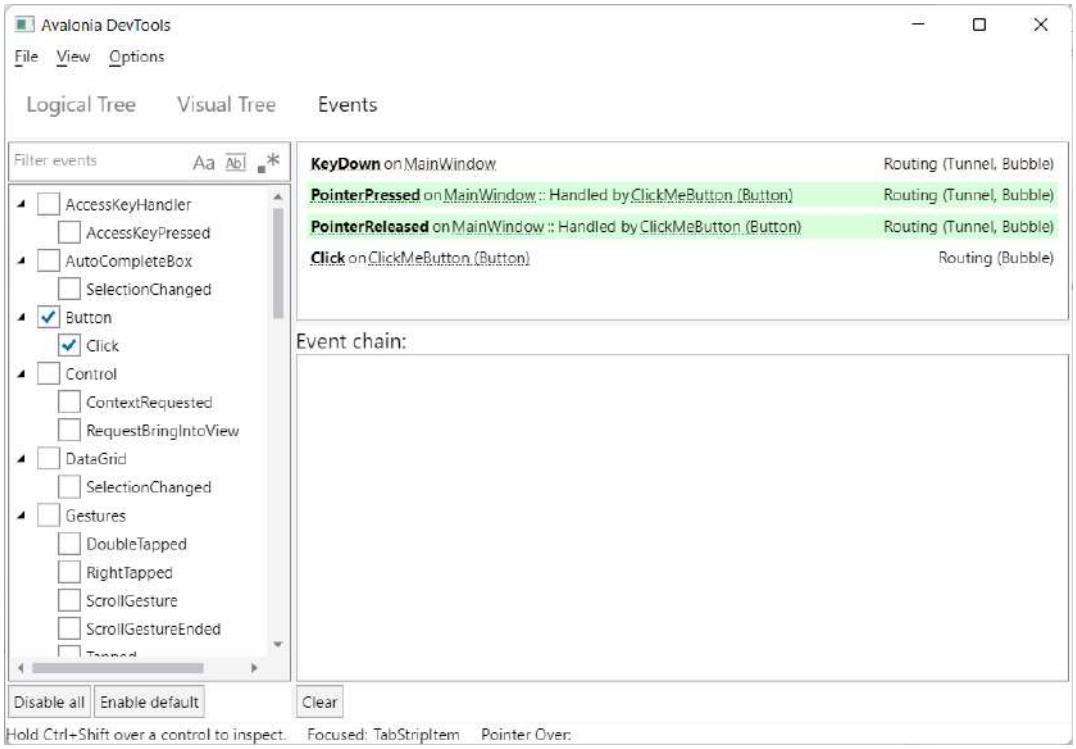


Рисунок 4.26 – Приклад перехоплення подій із кнопкою миші

На рис. 4.26 подано події, що відбуваються при натисканні (PointerPressed) та відпусканні (PointerReleased) лівої кнопки миші. Також можна бачити, що вказаним подіям передує подія KeyDown, а завершує їх – Click.

4.4 Додані властивості (Attached Properties)

Додані властивості – надзвичайно важлива та корисна концепція для розуміння. Вперше вона була представлена в WPF, а звідти перенесена до Avalonia в розширеній версії.

Attached Property базується на простій властивості читання/запису в C#. Властивість типу T, визначена в класі MyClass, може бути представлена двома методами – методами геттера і методами встановлення:

```
public class MyClass
{
```

```
  T Getter();
  void Setter(T value);
}
```

Зазвичай така властивість реалізується за допомогою поля підтримки типу T, визначеного в тому самому класі:

```
public class MyClass
{
  // the backing field
  T _val;

  T Getter() => _val;
  void Setter(T value) => _val = value;
}
```

Під час роботи над WPF архітектори WPF зіткнулися з цікавою проблемою. Кожен візуальний об’єкт повинен був визначати сотні, якщо не тисячі властивостей, більшість з яких кожен раз мали б значення за замовчуванням. Визначення резервного поля для кожної властивості в кожному об’єкті призведе до величезного споживання пам’яті, особливо непотрібного, оскільки приблизно 90% цих властивостей кожного разу матиме значення за замовчуванням.

Тому, щоб обійти цю проблему, вони придумали додані властивості (Attached Property). Замість того, щоб зберігати значення властивості в резервному полі всередині об’єкта, додана властивість зберігає значення у вигляді статичної хеш-таблиці або словника (або карти), де значення індексуються різними об’єктами, які можуть мати ці властивості. У хеш-таблиці знаходяться лише об’єкти зі значеннями властивостей, які не є значеннями за замовчуванням. Якщо запис для об’єкта не міститься в хеш-таблиці, передбачається, що властивість об’єкта має значення за замовчуванням. Статична хеш-таблиця доданої властивості може бути визначена практично в будь-якому класі – часто вона визначається в іншому класі, ніж той, який використовує його значення.

Таким чином, реалізація вкладеної властивості, наприклад з ім’ям MyAttachedProperty типу double у класі MyClass може буте реалізовано наступним чином:

```
public class MyClass
{

}

public static class MyAttachedPropertyContainer
{
    // Attached Property's default value
    private static double MyAttachedPropertyDefaultValue = 5.0;
    // Attached Property's Dictionary
    private          static          Dictionary<MyClass,          double>
MyAttachedPropertyDictionary = new Dictionary<MyClass, double>();
    // property getter
    public static double GetMyAttachedProperty(this MyClass obj)
    {
       if (MyAttachedPropertyDictionary.TryGetValue(obj, out double value)
       {
           return value;
       }
       else // there is no entry in the Dictionary for the object
       {
           return MyAttachedPropertyDefaultValue; // return default value
       }
    }

    // property setter
    public static SetMyAttachedProperty(this MyClass obj, double value)
    {
       if (value == MyAttachedPropertyDefaultValue)
       {
    // since the property value on this object 'obj' should become default,
    // we remove this object's entry from the Dictionary -
    // once it is not found in the Dictionary, -
    // the default value will be returned
           MyAttachedPropertyDictionary.Remove(obj);
        }
        else
        {
            // we set the object 'to have' the passed property value
            // by setting the Dictionary cell corresponding to the object
            // to contain that value
            MyAttachedPropertyDictionary[obj] = value;
        }
    }
}
```

Отже, замість кожного об’єкта типу MyClass, що містить значення, вони знаходяться в статичному словнику, індексованому об’єктами типу MyClass. Можна також вказати деяке значення за замовчуванням для властивості (у нашому випадку це 5.0), щоб лише об’єкти зі значенням властивості не за замовчуванням вимагали запису до словника.

Такий підхід економить багато пам’яті за рахунок дещо повільніших геттерів і сетерів для властивості. Також, крім економії пам’яті, вони дають багато інших переваг, наприклад:

– можна легко додати до них деякі зворотні виклики сповіщень про зміну властивостей, які запускатимуться, коли властивість змінюється в об’єкті;

– можна визначити додану властивість для класу, не змінюючи сам клас.

Остання особливість надзвичайно важлива. Яскравий приклад – звичайна кнопка не має властивості Height (Висота). Припустимо, що у нашій програмі є багато кнопок різних типів, і раптом користувачі вимагають, щоб багато з них мали плавні кути меж, а, крім того, різні кнопки повинні мати різну висоту. Розробник не хоче створювати новий похідний тип для кнопок і замінювати їх скрізь і повторно тестувати кожен з них, але може трохи змінити стилі кнопок. Таким чином, можна створити додану властивість ButtonHeightProperty, зв’язати властивість Height для меж кнопок ButtonHeightProperty кнопки та встановити для цієї властивості необхідні значення в стилях окремих кнопок.

Узагальнюючи попередній пункт, додані властивості дозволяють створювати та приєднувати поведінку до візуальних об’єктів – поведінки є складними класами, які дозволяють змінювати та розширювати функціональні можливості візуального об’єкта, не змінюючи клас візуального об’єкта.

Звичайно, дуже проста реалізація, розглянута вище, не бере до уваги багато інших питань, таких як потоки, зворотні виклики, реєстрація (щоб знати всі прикріплені властивості, визначені в нашому класі MyClass) тощо. Крім того, недоцільно визначати значення за замовчуванням як статичну змінну окремо за межами властивості, як було зроблено вище. Через ці міркування має сенс створити спеціальний тип (можливо з деякими загальними аргументами) AttachedProperty, який міститиме словник, значення за замовчуванням та багато інших функцій, необхідних для функціонування властивості. Це те, що реалізовано в WPF та Avalonia.

Створимо тестовий проєкт для дослідження використання доданої властивості. В якості такої властивості зробимо ButtonHeightProperty для

реалізації можливості зміни висоти кнопки в залежності від положення слайдера на екрані.

Значення повзунка слайдера може змінюватися між значеннями від 50 до 180. Коли користувач змінює положення повзунка, властивість ButtonHeight кнопки відповідно теж повинна змінюватись – висота стає більшою або меншою.

На першому етапі створимо кнопку та розташуємо поруч слайдер. Для цього нам необхідно використати наступний код XAML:

```
<Grid ColumnDefinitions="250, 50">
            <Button VerticalAlignment="Center"
                        HorizontalAlignment="Center"
                        Grid.Column="0"
                        Background="White"
                        Width="150"
                        Height="50">
                  <Border Background="Red"
                        BorderBrush="Black"
                        BorderThickness="2"
                        CornerRadius="15"
                        Padding="4">
                        <TextBlock
                              Foreground="White"
                              VerticalAlignment="Center"
                              HorizontalAlignment="Center">
                              Test Button
                        </TextBlock>
                  </Border>
            </Button>

            <Slider Minimum="0"
                        Maximum="10"
                        Grid.Column="1"
                        Orientation="Vertical"
                        HorizontalAlignment="Center"
                        VerticalAlignment="Center"
                        Height="150"
                        Width="50"/>
      </Grid>
```

В результаті даної розмітки формується наступне вікно користувача (рис. 4.27).

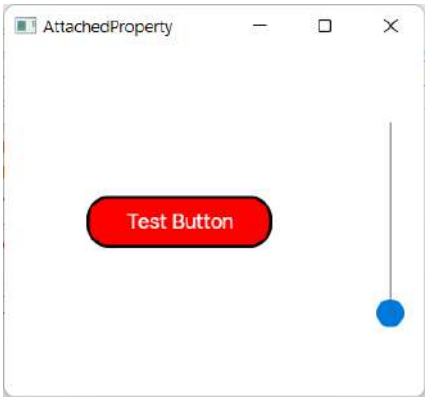


Рисунок 4.27 – Інтерфейс користувача із кнопкою та слайдером

Наступним кроком необхідно реалізувати можливість зміни висоти кнопки в залежності від положення слайдера. Для цього слід додати прив’язку в код XAML, а також створити додатковий клас для роботи із AttachedProperties.

Створюємо додатковий клас AttachedProperties (рис. 4.28). Вміст класу наступний:

```
public static class AttachedProperties
{
	public static double GetButtonHeight (AvaloniaObject obj)
	{
		return obj.GetValue(ButtonHeightProperty);
	}

	// Attached Property Setter
	public static void SetButtonHeight (AvaloniaObject obj, int value)
	{
		obj.SetValue(ButtonHeightProperty, value);
	}

	public static readonly AttachedProperty<int> ButtonHeightProperty =
	AvaloniaProperty.RegisterAttached<object, Control, int>
	(
		"ButtonHeight", // property name
		100 // property default value
	);
}
```

Розглянемо створений новий клас:

– public static int GetButtonHeight (AvaloniaObject obj) є геттером (подібним до того, що розглядалося вище);

– public static void SetButtonHeight (AvaloniaObject obj, int value) є установником;

– public static readonly AttachedProperty<int> ButtonHeightProperty – це статичне поле, що містить хеш-таблицю словника (або хеш-таблицю об’єкта зі значенням) і значення за замовчуванням приєднаної властивості та всі інші необхідні функції.

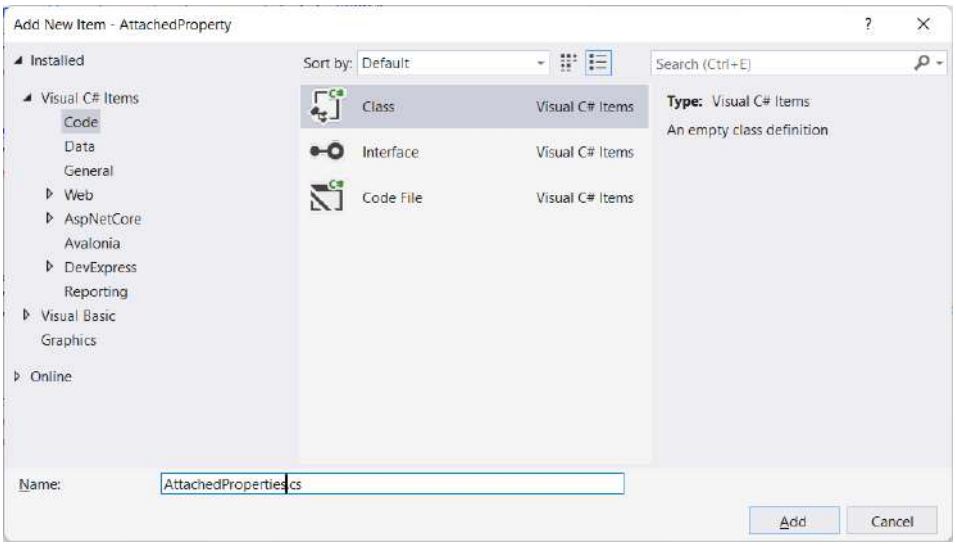


Рисунок 4.28 – Створення класу AttachedProperties

Тепер виконаємо модифікацію коду XAML у файлі MainWindow.cs.

В шапці файлу створюємо рядок з посиланням на простір імен «AttachedProperty» та присвоюємо йому назву «local»:

```
xmlns:local="clr-namespace:AttachedProperty"
```

Цей рядок визначає локальний простір імен XAML, щоб через цей простір імен ми могли посилатися на нашу приєднану властивість ButtonHeight.

Також в шапці додаємо рядок для встановлення початкових значень атрибуту ButtonHeight, для об’єкта вікна:

```
local:AttachedProperties.ButtonHeight ="50"
```

Зверніть увагу, як вказується Attached Property:

```
<namespace-name>:<class-name>.<AttachedProperty-name>.
```

Виконаємо прив’язку до локального ресурсу в коді створення віджету Border:

```
ButtonHeight="{Binding Path=(local:AttachedProperties.ButtonHeight), RelativeSource={RelativeSource AncestorType=Window}}"
```

та в коді створення віджету Slider:

```
Value="{Binding Path=(local:AttachedProperties.ButtonHeight),
      Mode=TwoWay,
      RelativeSource={RelativeSource AncestorType=Window}}"
```

Необхідно звернути увагу на формат властивості в прив’язці – повна назва доданої властивості міститься в дужках – це вимога як в Avalonia, так і в WPF – без дужок прив’язка не працюватиме.

Властивість Value повзунка пов’язується з властивістю ButtonHeight доданої властивості вікна-предка повзунка (який, зазвичай, є тим самим об’єктом Window, що і попередник вікна кнопки). Це прив’язування є двостороннім, тобто зміни властивості Value повзунка також змінять значення ButtonHeight доданої властивості (Attachness Property) у вікні.

Принцип роботи цього слайдера наступний – зміна значення повзунка шляхом переміщення так званого маркера призведе до зміни властивості ButtonHeight доданої властивості у вікні (через прив’язку повзунка), а це, у свою чергу, спричинить зміну властивості ButtonHeight кнопки (через її прив’язку).

Звичайно, у цьому простому випадку ми могли б підключити значення повзунка до властивості ButtonHeight кнопки безпосередньо, не залучаючи прикріпленої властивості у вікні, але тоді приклад не продемонстрував би, як працюють прикріплені властивості (і в багатьох випадках, наприклад, коли необхідна властивість не існує в елементі керування, прикріплені властивості є обов’язковими).

В результаті код XAML буде виглядати наступним чином:

```
<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"
   xmlns:local="clr-namespace:AttachedProperty"
```

```
    Width="300"
    Height="250"
    x:Class="AttachedProperty.MainWindow"
    local:AttachedProperties.ButtonHeight="50"
    Title="AttachedProperty">

    <Grid ColumnDefinitions="250, 50">
        <Button VerticalAlignment="Center"
            HorizontalAlignment="Center"
            Grid.Column="0"
            Background="White"
            Width="150"
            Height= "{Binding Path=(local:AttachedProperties.ButtonHeight),
                      RelativeSource={RelativeSource AncestorType=Window}}">
            <Border Background="Red"
                BorderBrush="Black"
                BorderThickness="2"
                CornerRadius="15"
                Padding="4">
                <TextBlock
                    Foreground="White"
                    VerticalAlignment="Center"
                    HorizontalAlignment="Center">
                    Test Button
                </TextBlock>
            </Border>
        </Button>

        <Slider Minimum="0"
            Maximum="25"
            Grid.Column="1"
            Orientation="Vertical"
            HorizontalAlignment="Center"
            VerticalAlignment="Center"
            Value="{Binding Path=(local:AttachedProperties.ButtonHeight),
                Mode=TwoWay,
                RelativeSource={RelativeSource AncestorType=Window}}"
            Height="150"
            Width="50"/>
    </Grid>
</Window>
```

На даному етапі можна перевірити зв’язок коду XAML із вихідним кодом на C#. В статичному полі ButtonHeightProperty класу AttachedProperties нами введено значення за замочуванням «10»:

```
"ButtonHeight", // property name
 100 // property default value
```

Якщо видалити рядок:

```
local:AttachedProperties.ButtonHeight="50"
```

та перезапустити програму, то можна побачити, що початкове значення повзунка стало 50, а не 100. Це пов’язано з тим, що значення за замовчуванням береться із цієї статичної змінної під час реєстрації приєднаної власності.

Наступним кроком є реєстрація події зміни значення слайдера. Для цього необхідно підписатися на відповідне сповіщення в коді на C# основного вікна:

```
// subscribe
_changeNotificationSubscriptionToken = AttachedProperties
      .ButtonHeightProperty
      .Changed
      .Subscribe(OnButtonHeightChanged);
```

Метод OnButtonHeightChanged(...) викликається, коли значення ButtonHeightProperty змінюється. Метод приймає один аргумент типу AvaloniaPropertyChangedEventArgsdouble, і цей аргумент містить всю необхідну інформацію:

– об’єкт, в якому відбулися зміни приєднаної властивості, надається властивістю Sender;

– властивість OldValue містить інформацію про попереднє значення.

– властивість NewValue містить інформацію про поточне значення.

Для тестування можна поставити точку зупинки налагодження в кінці методу OnButtonHeightChanged, запустити програму в режимі налагодження і спробувати перемістити повзунок на екрані. В результаті програма зупиниться на точці зупинки і можна дослідити поточні значення.

Інший, простіший спосіб зробити приблизно те ж саме (без можливості припинити підписку) – створити статичний конструктор у файлі MainWindow.axaml.cs і використовувати метод розширення AddClassHandler:

```
static MainWindow()
{
```

```
        AttachedProperties
            .ButtonHeightProperty
            .Changed
            .AddClassHandler<MainWindow>((x, e) =>
                x.OnAttachedPropertyChanged(e));
    }

    private void OnAttachedPropertyChanged(AvaloniaPropertyChangedEventArgs e)
    {
        double? oldValue = (double?)e.OldValue;
        double? newValue = (double?)e.NewValue;
    }
```

Зауважте, що тут не потрібно перевіряти, що відправник є таким самим, що й поточний об’єкт.

Також можна побачити, що метод OnAttachedPropertyChanged(...) має трохи менший тип безпечного підпису. Зазвичай цей спосіб відповідає 99% випадкам застосування, та за допомогою його можна досягти того, що нам потрібно, використовуючи AddClassHandler(...).

Із останнього прикладу можна бачити, що Avalonia використовує потужну парадигму IObservable Reactive Extensions, коли справа доходить до сповіщень про зміну приєднаних властивостей.

На рис. 4.29 подано результат роботи тестової програми. Повний текст програми можна знайти в лістингу «Listing_4-12_AttachedProperty».

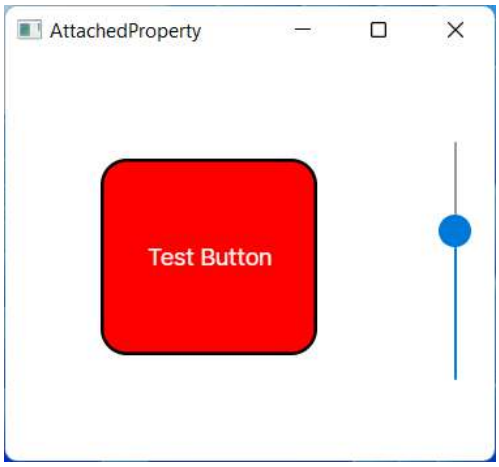


Рисунок 4.29 – Результат роботи тестової програми

4.5 Властивості стилю (Styled Properties)

WPF має концепцію Dependency Properties, яка в основному така сама, як і приєднані властивості, тільки вони визначені в тому ж класі, який їх використовує, і, відповідно, їх геттери та сетер розміщуються в однойменній властивості класу. Необхідно зауважити, що за допомогою Dependency Properties ми все ще маємо перевагу: не витрачаємо пам’ять на значення за замовчуванням і легко додаємо зворотні виклики, але ми втрачаємо перевагу додавання властивості без зміни класу.

Для демонстрації даного методу створимо новий проєкт, аналогічний до попереднього.

Приклад буде виконуватися точно так само, як і попередній, і код дуже схожий, тільки замість використання приєднаної властивості ButtonHeight, що визначена у файлі AttachedProperties.cs, ми використовуємо однойменну властивість Style, визначену в MainWindow.axaml.cs. Ви можете побачити, що геттер і сетер властивості Style є нестатичними і значно простішими:

```
public double ButtonHeight
{
      // getter
      get { return GetValue(ButtonHeightProperty); }
      // setter
      set { SetValue(ButtonHeightProperty, value); }
}

public static readonly StyledProperty<double> ButtonHeightProperty =
        AvaloniaProperty.Register<MainWindow, double>
(
      nameof(ButtonHeight)
);
```

Код XAML для даного прикладу наступний:

```
<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"
   xmlns:local="clr-namespace:StyleProperty"
```

```
    Width="300"
    Height="300"
    x:Class="StyleProperty.MainWindow"
    ButtonHeight="50"
    Title="StyleProperty">

    <Grid ColumnDefinitions="250, 50">
       <Button VerticalAlignment="Center"
          HorizontalAlignment="Center"
          Grid.Column="0"
          Background="White"
          Width="150"
          Height="{Binding Path=ButtonHeight,
          RelativeSource={RelativeSource AncestorType=Window}}">
          <Border Background="Red"
             BorderBrush="Black"
             BorderThickness="2"
             CornerRadius="15"
             Padding="4">
             <TextBlock
                Foreground="White"
                VerticalAlignment="Center"
                HorizontalAlignment="Center">
                Test Button
             </TextBlock>
          </Border>
          </Button>
          <Slider Minimum="50"
             Maximum="180"
             Grid.Column="1"
             Orientation="Vertical"
             HorizontalAlignment="Center"
             VerticalAlignment="Center"
             Value="{Binding Path=ButtonHeight,
             Mode=TwoWay,
             RelativeSource={RelativeSource AncestorType=Window}}"
             Height="150"
             Width="50"/>
    </Grid>
</Window>
```

Після запуску програми її поведінка така сама, що й у попередньому прикладі. Повний текст програми можна знайти в лістингу «Listing_4-13_StyleProperty».

4.6 Прямі властивості (Direct Properties)

Іноді потрібно використовувати просту властивість C#, підкріплену полем, але мати можливість підписатися на її зміни та використовувати властивість як ціль певного зв'язування – тому що, як цільові можна використовувати лише властивості Attached, Style і Direct для зв'язування в Авалонія. Прості властивості C# можна використовувати як джерело прив'язок, надаючи сповіщення про зміну через запуск події PropertyChanged інтерфейсу INotifyPropertyChanged.

Для демонстрації роботи із Direct Properties створимо новий проєкт, що буде за змістом дуже схожий на два попередніх. Розглянемо, як визначається властивість Direct у файлі MainWindow.xaml.cs:

```
private double _ButtonHeight = default;

public static readonly DirectProperty<MainWindow, double>
      ButtonHeightProperty =
      AvaloniaProperty.RegisterDirect<MainWindow, double>
(
      nameof(ButtonHeight),
      o => o.ButtonHeight,
      (o, v) => o.ButtonHeight = v
);

public double ButtonHeight
{
      get => _ButtonHeight;
      set
      {
            SetAndRaise(ButtonHeightProperty, ref _ButtonHeight, value);
      }
}
```

Код XAML для даного прикладу наступний:

```
<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"
  xmlns:local="clr-namespace:DirectProperty"
```

```
    Width="300"
    Height="300"
    x:Class="DirectProperty.MainWindow"
    ButtonHeight="50"
    Title="DirectProperty">
    <Grid ColumnDefinitions="250, 50">
      <Button VerticalAlignment="Center"
        HorizontalAlignment="Center"
        Grid.Column="0"
        Background="White"
        Width="150"
        Height="{Binding Path=ButtonHeight,
        RelativeSource={RelativeSource AncestorType=Window}}">
        <Border Background="Red"
           BorderBrush="Black"
           BorderThickness="2"
           CornerRadius="15"
           Padding="4">
           <TextBlock
              Foreground="White"
              VerticalAlignment="Center"
              HorizontalAlignment="Center">
              Test Button
           </TextBlock>
        </Border>
      </Button>

      <Slider Minimum="50"
        Maximum="180"
        Grid.Column="1"
        Orientation="Vertical"
        HorizontalAlignment="Center"
        VerticalAlignment="Center"
        Value="{Binding Path=ButtonHeight,
        Mode=TwoWay,
        RelativeSource={RelativeSource AncestorType=Window}}"
        Height="150"
        Width="50"/>
    </Grid>
</Window>
```

Повний текст програми можна знайти в лістингу «Listing_4-14_DirectProperty».

Класи AttachedProperty..., StyleProperty... та DirectProperty... походять від класу AvaloniaProperty. Як було зазначено вище, лише властивості Attached, Style і Direct можна зробити цільовими прив’язками інтерфейсу користувача Avalonia.

Якщо нам не потрібне попереднє значення змінної (OldValue у зразках, які ми мали вище), найкращим способом підписатися на зміни властивостей Attached, Style та Direct було б використовувати метод:

```
AvaloniaObject.GetObservable(AvaloniaProperty property)
```

Щоб продемонструвати використання методу GetObservable(...), ми можемо змінити наш зразок Attached Property таким чином:

```
public MainWindow()
{
      ...
      _changeNotificationSubscriptionToken = his.GetObservable(
      AttachedProperties.ButtonHeightProperty)
      .Subscribe(OnButtonHeightChanged);
}

private void OnButtonHeightChanged(double newValue)
{
      ...
}
```

З поданого коду можна побачити, що OldValue більше не доступний у зворотному виклику.

4.7 Взаємодія із структурованими властивостями

Властивості Attached, Style та Direct можна встановити лише для класів, які реалізують AvaloniaObject – це базовий клас, який реалізують усі візуальні елементи Avalonia. Існують такі властивості, де значення представлене у вигляді структур даних. Наприклад властивість CornerRadius у віджеті Border є структурою та має такий синтаксис:

```
public readonly struct CornerRadius : ValueType, IEquatable<CornerRadius>
```

Виходячи із документації, CornerRadius походить від ValueType, який в свою чергу – від object та IEquatable (рис. 4.30).

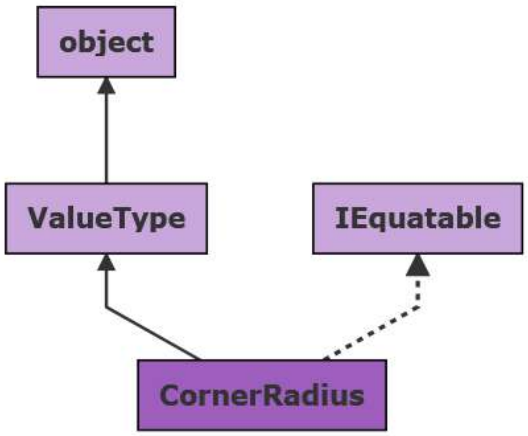


Рисунок 4.30 – Ієрархія класів для CornerRadius

Конструктор CornerRadius має такі реалізації:

– CornerRadius(double);

– CornerRadius(double, double);

– CornerRadius(double, double, double, double).

Розглянуті вище способи прив’язки не можуть біти застосовані для таких об’єктів. Тому необхідно використовувати інші підходи до реалізації задачі впливу на вміст структурованих властивостей. В даному посібнику для ознайомлення із можливостями Avalonia пропонується використовувати комбінацію Attached Property та Code-Behind.

Для демонстрації даного методу використовуватимо код проєкту AttachedProperty із прикладу «Listing_4-12_AttachedProperty». В існуючий код будуть внесені незначні зміни.

По-перше, в коді XAML видалимо посилання на додану властивість із властивості Height компонента Border. Даний рядок:

```
Height="{Binding Path=(local:AttachedProperties.ButtonHeight),
      RelativeSource={RelativeSource AncestorType=Window}}">
```

необхідно замінити на:

```
Height="50".
```

По-друге, в коді замінити назву властивості ButtonHeight на CornerRadius.

По-третє, в файлі MainWindow.axaml.cs в функцію OnCornerRadiusChanged додати два рядка:

```
Border border = this.FindControl<Border>("ButtonBorder");
border.CornerRadius = new CornerRadius(newValue);
```

Таким чином, код функції OnCornerRadiusChanged виглядатиме наступним чином:

```
private void OnCornerRadiusChanged(
AvaloniaPropertyChangedEventArgs<double> changeParams)

{
      // if the object on which this Style property changes
      // is not this very window, do not do anything
      if (changeParams.Sender != this)
      {
            return;
      }

      // check the old and new values of the Style property.
      double oldValue = changeParams.OldValue.Value;
      double newValue = changeParams.NewValue.Value;

      //Find control
      Border border = this.FindControl<Border>("ButtonBorder");
      border.CornerRadius = new CornerRadius(newValue);
}
```

Повний код прикладу подано в лістингу «Listing_4-14_CodeBehind».

В результаті компіляції програми отримаємо можливість зміни радіуса кутів межі кнопки в залежності від положення слайдера (рис. 4.31).

В результаті ми отримали можливість перехоплення зміни властивості слайдеру за допомогою концепції Attached Property та впливу на візуальну структуровану властивість CornerRadius для зміни її поточного значення.

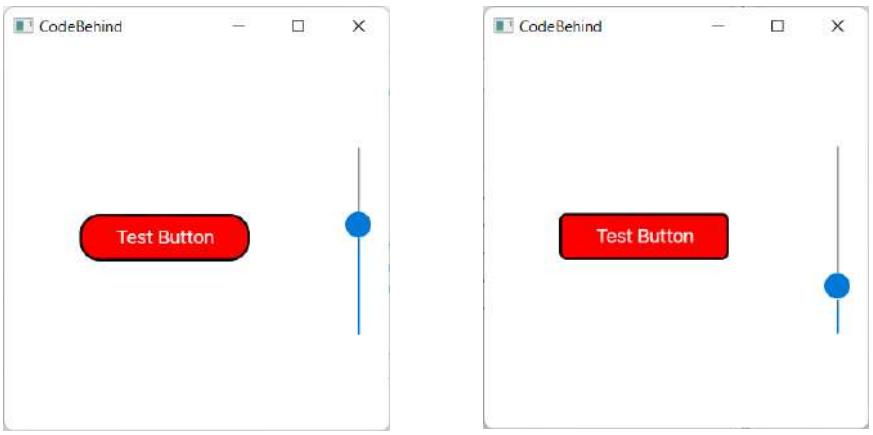


Рисунок 4.31 – Приклад зміни радіуса кутів кнопки
в залежності від положення слайдера

4.8 Прив’язка до команд та методів за технологією MVVM

Елементи керування, які виконують дію, наприклад Button, мають властивість Command, яку можна прив’язати до ICommand. Коли елемент керування активовано (наприклад, коли натиснута кнопка), буде викликано метод ICommand.Execute.

Реалізацію ICommand можна знайти в ReactiveCommand ReactiveUI. Якщо програма створена за допомогою шаблону програми Avalonia MVVM, вона буде доступна за замовчуванням.

Розглянемо приклад програми з однією кнопкою, натискання на яку будемо оброблювати в програмі. Зовнішній вигляд інтерфейсу подано на рис. 4.32.

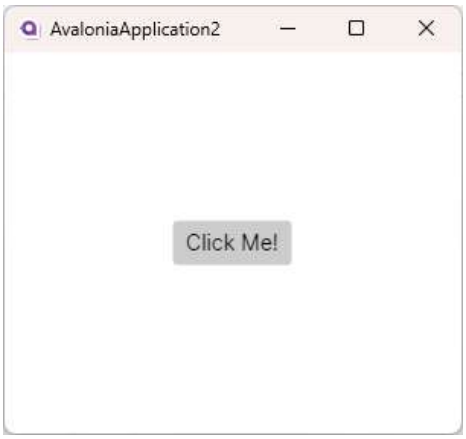


Рисунок 4.32 – Зовнішній вигляд інтерфейсу

Код інтерфейсу наступний:

```
<Button VerticalAlignment="Center"
            HorizontalAlignment="Center"
      Command="{Binding testBinding}">
      Click Me!
</Button>
```

Код файлу MainWindowViewModel.cs має наступний вміст:

```
public class MainWindowViewModel : ViewModelBase
{
      public ReactiveCommand<Unit, Unit> testBinding { get; }

      public MainWindowViewModel()
      {
            testBinding = ReactiveCommand.Create(testClick);
      }

      private void testClick()
      {
            throw new NotImplementedException();
      }
}
```

Поєднання візуальної частини з програмною відбувається завдяки прив’язці

```
Command="{Binding testBinding}"
```

у файлі MainWindow.axaml. Тут ми бачимо посилання на об’єкт testBinding типу ReactiveCommand<Unit, Unit>. При створенні даного об’єкта в параметрах вказується посилання на метод testClick(), в якому виконується обробка натискання на кнопку.

Якщо потрібно передати параметри з візуального шару в програму необхідно використовувати властивість CommandParameter :

```
<Button VerticalAlignment="Center"
      HorizontalAlignment="Center"
```

```
        Command="{Binding testBinding}"
        CommandParameter="Hello World">
            Click Me!
</Button>
```

В даному випадку код програми C# матиме такий вигляд:

```
public class MainWindowViewModel : ViewModelBase
{
        public ReactiveCommand<string, Unit> testBinding { get; }

        public MainWindowViewModel()
        {
            testBinding = ReactiveCommand.Create<string>(testClick);
        }

        private void testClick(string parameter)
        {
            throw new NotImplementedException();
        }
}
```

Для CommandParameter не виконується перетворення типу, тому, якщо вам потрібно, щоб ваш параметр типу був чимось іншим, ніж рядок, ви повинні надати об’єкт цього типу в XAML. Наприклад, щоб передати параметр int, ви можете використати:

```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:sys="clr-namespace:System;assembly=mscorlib">

    <Button Command="{Binding testBinding}">
        <Button.CommandParameter>
            <sys:Int32>42</sys:Int32>
        </Button.CommandParameter>
        Click Me!
    </Button>
</Window>
```

Іноді необхідно просто викликати метод, коли натискаєте кнопку, без створення команди. Це робиться наступним чином.

Код візуального інтерфейсу:

```
<Window xmlns="https://github.com/avaloniaui">
      <Button Command="{Binding RunTheThing}"
      CommandParameter="Hello World">Do the thing!</Button>
</Window>
```

Код програми мовою C#:

```
namespace Example
{
    public class MainWindowViewModel : ViewModelBase
    {
        public void RunTheThing(string parameter)
        {
            // Code for executing the command here.
        }
    }
}
```

4.9 Контрольні запитання та завдання

1. Які основні концепції прив’язки даних в Avalonia? Які різновиди прив’язки даних існують в Avalonia?

2. Як використовувати прив’язку за замовчуванням (DataContext) в Avalonia?

3. Яким чином використовувати прив’язку за назвою елемента (Binding by ElementName) в Avalonia?

4. Як використовувати прив’язку до ресурсу (Binding.Source) в Avalonia?

5. Що таке властивості стилю (Styled Properties) в Avalonia і як вони використовуються?

6. Що таке прямі властивості (Direct Properties) в Avalonia і як вони відрізняються від інших типів властивостей?

7. Як використовувати структуровані властивості (Structured Properties) для полегшення роботи зі стилями та оформленням в Avalonia?

8. Які можливості надає взаємодія з ресурсами (Resources) в Avalonia, і як їх можна використовувати для підтримки повторного використання та розширення функціоналу додатка?

9. Наведіть приклад використання елементів управління (Controls) та їх властивості для створення інтерактивного інтерфейсу користувача в Avalonia.

10. Як використовувати події (Events) для реагування на дії користувача і виконання певних дій у вашому додатку на основі дій користувача в Avalonia?

5 БАЗОВІ ФУНКЦІЇ XAML

5.1 Основи Avalonia XAML

XAML – це XML, який використовується для створення об’єктів C# (переважно візуальних). Класи C# відображаються як теги XML, тоді як властивості класу зазвичай відображаються як атрибути XML:

```
<my_namespace:MyClass Prop1="Hello World"
                      Prop2="123"/>
```

Код XAML у прикладі вище створює об’єкт типу MyClass з простору імен XML my_namespace і встановлює його властивості Prop1, як рядок «Hello World» і Prop2, як значення 123. Зверніть увагу, що властивості будуть розв’язані до їх типу C#, наприклад, якщо Prop2 має тип int, вона буде перетворена до 123 цілого значення, а якщо це рядковий тип, вона буде перетворена до рядка «123». Якщо властивість або тип, згаданий у файлі XAML, не існує, компіляція проєкту, який містить цей XAML, завершиться невдачею, і часто Visual Studio виявить помилку ще до компіляції та підкреслить відсутній тип або властивість червоною ламаною лінією.

Простір імен (у нашому прикладі це «my_namespace») зазвичай слід визначити вище або всередині тегу XML, де він використовується. Він може вказувати на простір імен C# або встановлювати простір імен C#.

Файл XAML можна пов’язати з файлом C# із застосуванням концепції «code-behind», щоб визначити відповідний клас, використовуючи оголошення «partial class». Код C# зазвичай містить визначення методів, які служать обробниками подій, що запускаються елементами, визначеними у файлі XAML. Цей спосіб зв’язування подій (описано у попередніх розділах цього посібника), які запускаються елементами XAML та обробниками подій C#, є найпростішим та «прозорішим» для розуміння. Однак він також є і найгіршим, оскільки він порушує класичну концепцію шаблону MVVM і майже ніколи не повинен використовуватися.

5.2 Простір імен XAML

Простір імен XAML – це рядок, який зазвичай визначається в елементі верхнього рівня файлу XAML (навіть якщо його можна визначити в будь-якому тегові) і вказує на простір імен C# у певній збірці або збірках .NET, до яких входить поточний XAML файл і що містяться у даному проєкті.

Розглянемо два верхні рядки стандартного файлу MainWindow.xaml:

```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml" ... >
```

Ці два рядки визначають два простори імен XAML для всього файлу. Один із цих просторів імен, який не потребує жодного префікса (він має порожній префікс), а інший має префікс «x». Обидва простори імен відносяться до типів, визначених у пакетах Avalonia.

Можна визначити багато елементів (скажімо, кнопку) в Avalonia без будь-якого префікса (наприклад, як <Button .../>), оскільки ці елементи розташовані в просторі імен Avalonia за замовчуванням, на який посилається «https://github.com/avaloniaui».

Простір імен, на який посилається префікс «x», містить різні типи, які використовуються трохи рідше. Наприклад – багато вбудованих типів C#, наприклад, рядок і об’єкт, можуть називатися в XAML як <x:String.../x:String> та <x:Object.../x:Object> (вони містяться в просторі імен Avalonia посилання на «http://schemas.microsoft.com/winfx/2006/xaml»).

Важлива примітка. Так звані URL-адреси простору імен XAML не повинні посилатися на будь-яку дійсну URL-адресу, яка дійсно існує в Інтернеті, і комп’ютер не повинен бути в мережі, щоб вони працювали.

Для вивчення основ роботи із простором імен створимо новий проєкт XAMLNamespace. В новому проєкті створимо ще два залежних проєкти: DependencyProject_1 та DependencyProject_2. При створенні нових проєктів необхідно обрати шаблон «Class Library» (рис. 5.1).

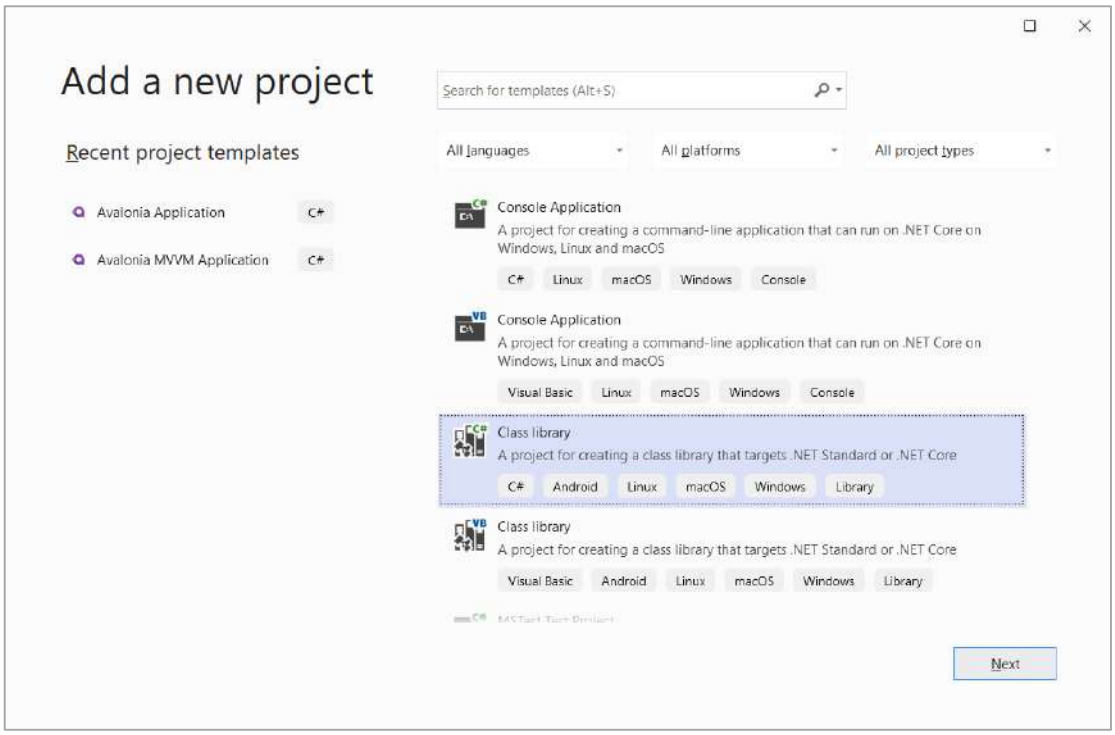


Рисунок 5.1 – Створення нового проєкту

Наразі структура нашого проєкту має вигляд, як подано на рис. 5.2.

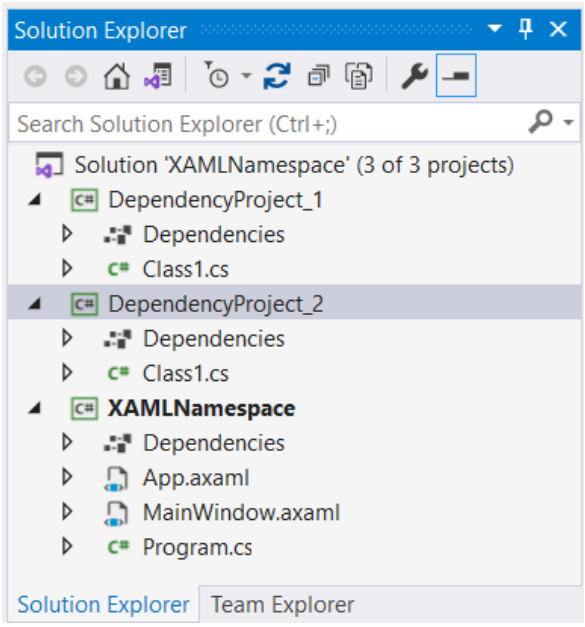


Рисунок 5.2 – Додавання залежних проєктів до рішення

В кожен із проєктів додаємо по одній папці SubFolder для розміщення файлів із описом візуальних компонентів (рис. 5.3).

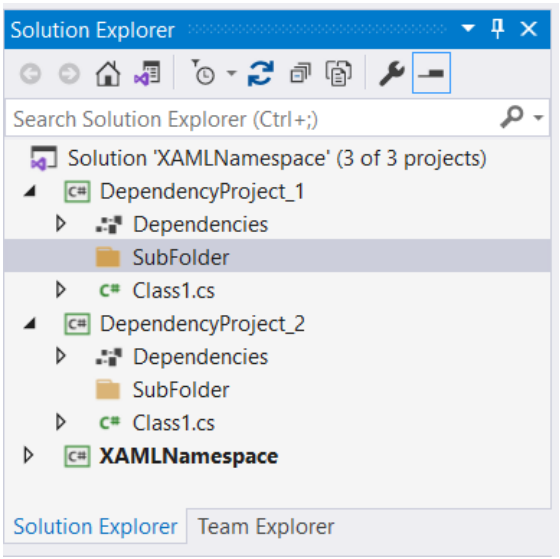


Рисунок 5.3 – Додавання SubFolder до проєктів

Наступним кроком необхідно створити два нових класи, та перейменувати два вже існуючих, призначивши їм імена кольорових ламп технологічного світлофора: BlueLamp, RedLamp, YellowLamp та GreenLamp (рис. 5.4).

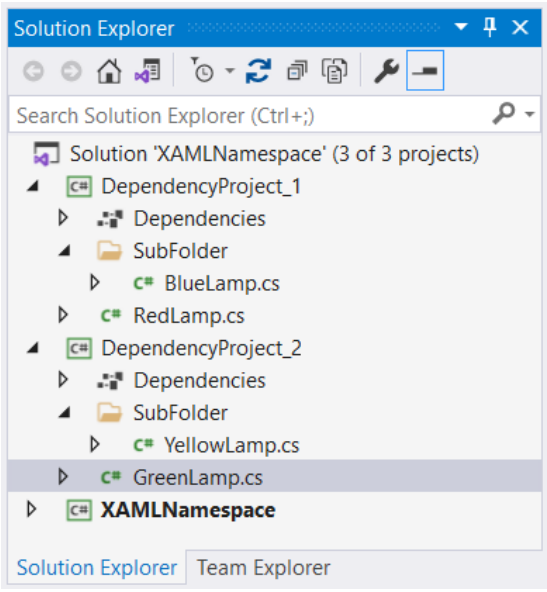


Рисунок 5.4 – Класи візуальних компонентів

Кожен клас містить схожий код для створення на екрані візуального компонента. Мета – створити графічний вигляд технологічного світлофора, або його ще називають «Світлова колона» (рис. 5.5).

Рисунок 5.5 – Зовнішній вигляд світлової колони

Для імітації кожної лампи використовується наступний код:

```
class BlueLamp : Border, IStyleable
{
	Type IStyleable.StyleKey => typeof(Border);

	public BlueLamp()
	{
		Background = new SolidColorBrush(Colors.Blue);
		BorderBrush = new SolidColorBrush(Colors.Black);
		BorderThickness = new Avalonia.Thickness(1);
		CornerRadius = new Avalonia.CornerRadius(4);
		Width = 100;
		Height = 40;
	}
}
```

Рядок коду:

```
Type IStyleable.StyleKey => typeof(Border);
```

в сукупності із реалізацією класом «…Lamp» інтерфейсу IStyleable гарантує, що стилі компонентів за замовчуванням (Border) з основної теми також будуть застосовані до класу BlueLamp, який походить від класу Avalonia Border. Конструктор призначає відповідний колір контролу і встановлює для нього висоту 40 та ширину 100 пікселів.

Аналогічно заповнюються інші класи для всіх кольорів світлової колони.

Із рис. 5.5 можна бачити, що колона крім індикаторних ламп має стійку, яка тримає всю конструкцію. Для її відображення в нашому проєкті передбачено окремий клас «Rack»:

```
class Rack : Path, IStyleable
{
   Type IStyleable.StyleKey => typeof(Path);
   public Rack()
   {
      Data = StreamGeometry.Parse("M30 300C24.4772 300 20 295.52284 " +
      "20 290C20 284.4771600000001 24.4772 280 30 280H120V60C120 " +
      "48.954 128.9544 40 140 40H160C171.0456 40 180 48.954 " +
     "180 60V280H270C275.522 280 280 284.4771600000001 280 290C280 " +
     "295.52284 275.522 300 270 300H180H120H30z");
     Stretch = Stretch.Fill;
     Height = 100;
     Width = 120;
     Stroke = new SolidColorBrush(Colors.Black);
     StrokeThickness = 1;
     Fill = new SolidColorBrush(Colors.Gray);
  }
}
```

Даний клас використовує об’єкт Path для створення на екрані зображення стійки кріплення світлової колони за заданими координатами.

Для використання можливостей Avalonia необхідно в кожний із доданих проєктів додати відповідні пакети «Avalonia» та «Avalonia.Desktop» (рис. 5.6).

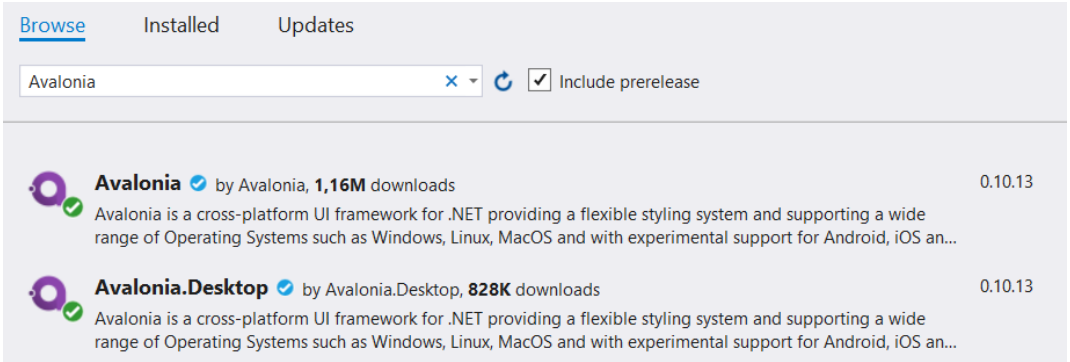


Рисунок 5.6 – Додавання пакетів до підпроєктів

Перед тим як використовувати створені класи необхідно додати посилання на додані проєкти в залежності головного проєкту за допомогою відповідного інструменту Visusal Studio (рис. 5.7).

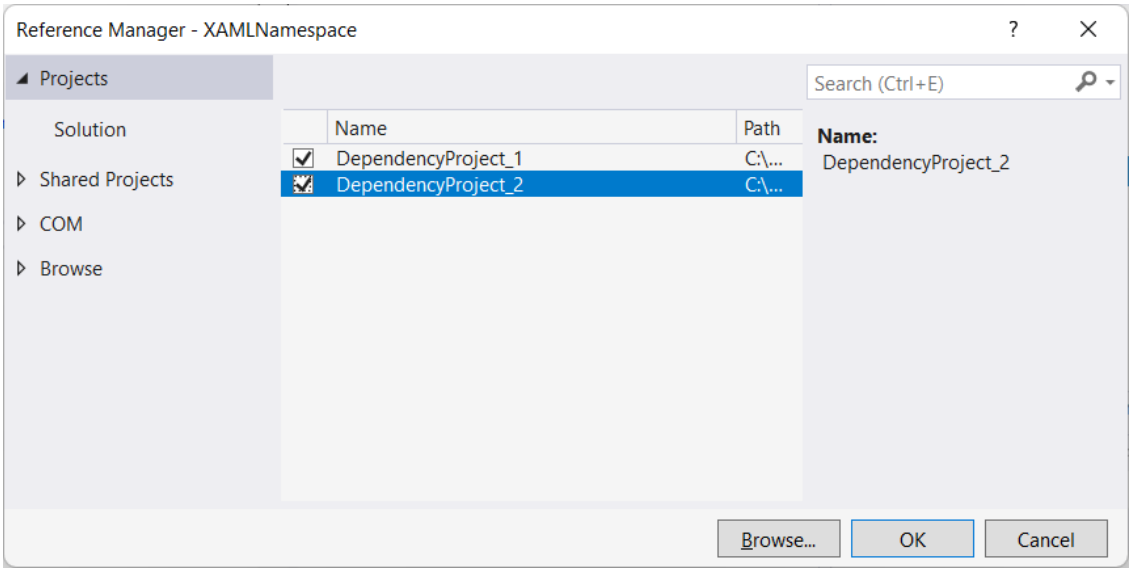


Рисунок 5.7 – Додавання посилання на додані проєкти

В результаті отримаємо таку структуру проєкту (рис. 5.8):

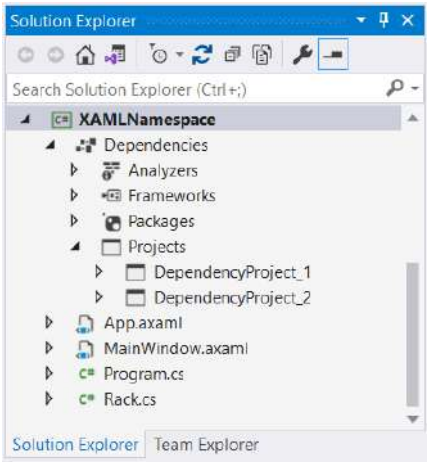


Рисунок 5.8 – Структура проєкту після додавання
пов’язаних проєктів

Для демонстрації різних способів посилання на простір імен створимо збірку із двох просторів. Для цього необхідно створити новий файл AssemblyInfo.cs із діалогового вікна додавання нового об’єкта у поточний проєкт (рис. 5.9).

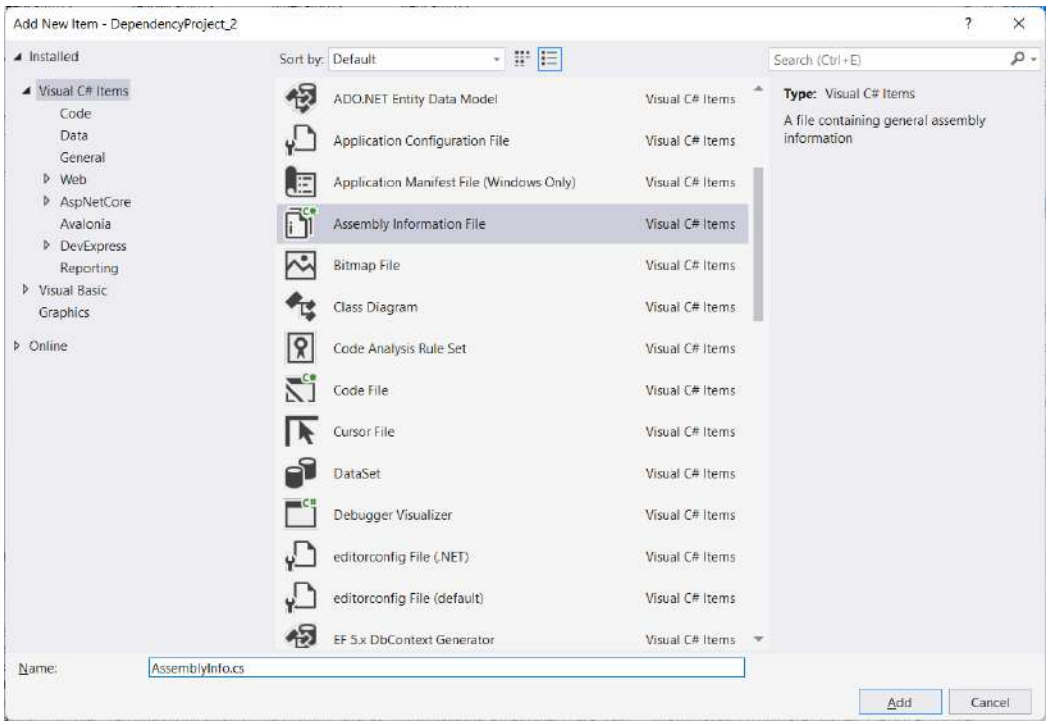


Рисунок 5.9 – Додавання файлу збірки

В новий файл, який було створено, слід додати два рядки із посиланням на простір імен другого підпроєкту:

```
[assembly: XmlnsDefinition("https://avaloniademos.com/xaml",
       "DependencyProject_2")]
[assembly: XmlnsDefinition("https://avaloniademos.com/xaml",
       "DependencyProject_2.SubFolder")]
```

В якості URL можна обрати будь-яку комбінацію, вона не буде посилатись на реальний URL в Internet. В результаті ми отримуємо можливість посилання на об’єднаний простір імен:

```
xmlns:dep2="https://avaloniademos.com/xaml"
```

із коду XAML.

В результаті ми отримали такий код XAML:

```
<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"
```

```
        xmlns:dep1="clr-namespace:DependencyProject_1;
          assembly=DependencyProject_1"
        xmlns:dep1_sub_Folder="clr-namespace:DependencyProject_1.SubFolder;
                assembly=DependencyProject_1"
        xmlns:local="clr-namespace:XAMLNamespace"
        xmlns:dep2="https://avaloniademos.com/xaml"
        Width="300"
        Height="450"
        x:Class="XAMLNamespace.MainWindow"
        Title="XAMLNamespace">

        <StackPanel HorizontalAlignment="Center"
                VerticalAlignment="Center">
                <dep1:RedLamp/>
                <dep1_sub_Folder:BlueLamp/>
                <dep2:YellowLamp/>
                <dep2:GreenLamp/>
                <local:Rack/>
          </StackPanel>
</Window>
```

Після компіляції та запуску програми отримаємо таке зображення, як подано на рис. 5.10.

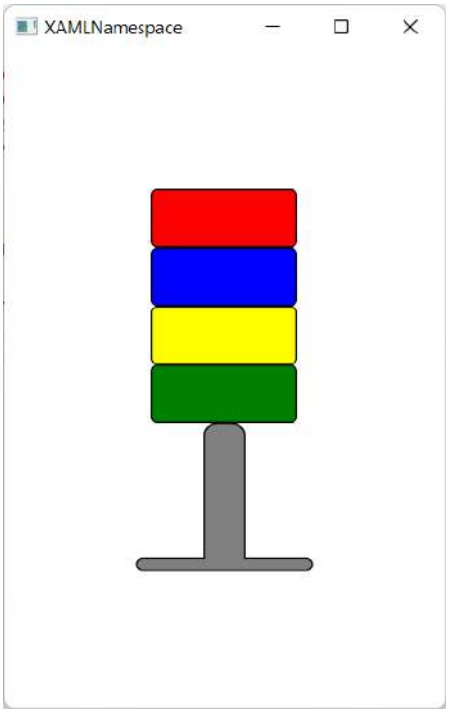


Рисунок 5.10 – Зображення світлової колони

Якщо проаналізувати вихідний код XAML, можна побачити, що є чотири визначені користувачем простори імен, задані такими рядками верхнього тегу XML:

```
xmlns:dep1="clr-namespace:DependencyProject_1;
      assembly=DependencyProject_1"
xmlns:dep1_sub_Folder="clr-namespace:DependencyProject_1.SubFolder;
      assembly=DependencyProject_1"
xmlns:local="clr-namespace:XAMLNamespace"
xmlns:dep2="https://avaloniademos.com/xaml"
```

На різні елементи інтерфейсу посилаються відповідні префікси простору імен XAML. Розглянемо рядок, який визначає префікс простору імен XAML dep1, за допомогою якого ми посилаємось на контрол RedLamp у файлі XAML:

```
xmlns:dep1="clr-namespace:DependencyProject_1;
      assembly=DependencyProject_1"
```

Значення простору імен містить дві частини, поділені ‘;’ (крапкою з комою). Перша частина відноситься до простору імен C#:

```
clr-namespace: DependencyProject_1
```

а друга частина належить до назви збірки:

```
assembly=DependencyProject_1
```

У випадку RedLamp і простір імен, і ім’я збірки мають однакову назву: DependencyProject_1.

BlueLamp визначається в тому самому проєкті (DependencyProject_1), але в папці SubFolder. Його простір імен C# – це не DependencyProject_1 (як для RedLamp), а DependencyProject_1.SubFolder.

Ось рядок, який визначає префікс простору імен XAML dep1_sub_Folder, за яким BlueLamp посилається у файлі MainWindow.xaml:

```
xmlns:dep1_sub_Folder="clr-namespace:DependencyProject_1.SubFolder;
      assembly=DependencyProject_1"
```

Простір імен clr змінився на DependencyProject_1.SubFolder, тоді як частина залишилась такою ж самою, оскільки BlueLamp було визначено в тій самій збірці DependencyProject_1.

Далі розглянемо запис <local:Rack/>. Код на C# для Rack визначено в основному проєкті XAMLNamespace. В Цьому проєкті знаходиться також і файл MainWindow.xaml. Через це ми можемо пропустити ім'я збірки під час визначення префікса «local» і вказати лише частину простору імені clr-namespace:

```
xmlns:local="clr-namespace:XAMLNamespace"
```

GreenLamp і YellowLamp визначаються в двох різних просторах імен DependencyProject_2. GreenLamp визначається в основному просторі імен проєкту – DependencyProject_2, а YellowLamp – у просторі імен DependencyProject_2.SubFolder. Однак DependencyProject_2 також має файл AssemblyInfo.cs, який визначає метадані збірки. У цьому файлі ми додали кілька рядків внизу:

```
[assembly: XmlnsDefinition("https://avaloniademos.com/xaml",
       "DependencyProject_2")]
[assembly: XmlnsDefinition("https://avaloniademos.com/xaml",
       "DependencyProject_2.SubFolder")]
```

Ці два рядки об'єднують два простори імен збірки: DependencyProject_2 і DependencyProject_2.SubFolder в одну URL-адресу:

```
"https://avaloniademos.com/xaml".
```

Як було зазначено вище, зовсім не має значення, чи існує ця URL-адреса чи комп'ютер знаходиться в мережі. Головне, щоб URL-адреса мала якесь значення, відповідне проєктам, які містять цю функцію.

Тепер префікс XAML dep2, за яким ми посилаємося на GreenLamp і YellowLamp, визначається посиланням на цю URL-адресу:

```
xmlns:dep2="https://avaloniademos.com/xaml"
```

У порівнянні з WPF є важлива додаткова функція Avalonia. У WPF можна посилатися за URL-адресою лише на функції, які не розташовані в тому самому проєкті, що й файл XAML, який хоче посилатися на нього, тоді як в Avalonia такого обмеження немає. Наприклад, якщо у нас є файл XAML у тому самому проєкті DependencyProject_2, ми все одно можемо поставити вище згаданий рядок URL у його верхній частині та посилатися на GreenLamp і YellowLamp, визначені в рамках одного проєкту за допомогою префікса dep2:.

Повністю із вихідним кодом проєкту можна ознайомитись із лістингу «Listing_5-1».

5.3 Організація доступу до властивостей об'єктів із XAML

5.3.1 Доступ до складних властивостей

Ми вже згадували вище, що доступ до властивостей коду C# можна отримати як до атрибутів XML відповідного елемента, наприклад:

```
<my_namespace:MyClass Prop1="Hello World"
                      Prop2="123"/>
```

Prop1 і Prop2 – це звичайні властивості C#, визначені в класі MyClass, які можна знайти в просторі імен C#, на який посилається префікс my_namespace цього файлу XAML. Prop1, ймовірно, має рядковий тип, тоді як Prop2 може бути будь-якого числового типу або рядка (XAML автоматично перетворить рядок «123» на правильний тип).

Але що станеться, якщо сама властивість буде якогось складного типу, що містить кілька власних властивостей? Розглянемо приклад створення складного типу та покажемо, як можна організувати доступ до нього із коду XAML. Для цього створим новий проєкт під назвою AccessProperties.

В новому проєкті створимо новий клас Person:

```
public class Person
{
    public int AgeInYears { get; set; }
    public string? FirstName { get; set; }
    public string? LastName { get; set; }
```

```
        public override string ToString()
        {
            return $"Person: {FirstName} {LastName}, Age: {AgeInYears}";
        }
    }
```

Тепер ми можемо відобразити його властивості, як вміст вікна. Ось що ми маємо у файлі MainWindow.xaml:

```
<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"
   xmlns:local="clr-namespace:AccessProperties"
   Width="300"
   Height="200"
   HorizontalContentAlignment="Center"
   VerticalContentAlignment="Center"
   x:Class="AccessProperties.MainWindow"
   Title="AccessProperties">
   <Window.Content>
      <local:Person FirstName="Joe"
                    LastName="Doe"
                    AgeInYears="25"/>
   </Window.Content>
</Window>
```

Необхідно звернути увагу на те, як ми призначаємо властивість Content вікна складеному типу:

```
   <Window.Content>
      <local:Person FirstName="Joe"
                    LastName="Doe"
                    AgeInYears="25"/>
   </Window.Content>
```

Ми використовуємо тег властивості Window.Content, що відокремлює крапкою назву класу від імені властивості.

Зауважте, що так само, як ми призначаємо властивості складених типів, ми також можемо призначати властивості примітивного типу, наприклад, ми можемо встановити ширину вікна за допомогою такого коду:

```
<Window ...>
  <Window.Width>
    <x:Double>300</x:Double>
  </Window.Width>
</Window>
```

замість використання атрибутів XML. Звичайно, такий запис застосовує більше коду, ніж використання атрибутів XAML, тому скоріше не використовується для властивостей примітивних типів.

Оскільки Window.Content є спеціальною властивістю, позначеною ContentAttribute, нам взагалі не потрібно було додавати <Window.Content>, і ми могли розмістити об'єкт <local:Person .../> прямо під тегом <Window...>. Існує лише одна властивість для кожного класу, яка може бути позначена атрибутом ContentAttribute, тому в багатьох випадках ми все одно змушені використовувати нотації <Class.Property>.

5.3.2 Спеціальні властивості XAML

Існує кілька спеціальних властивостей, позначених префіксом «x:», за умови, звичайно, що ми маємо префікс простору імен «x», визначений у верхній частині файлу наступним чином:

```
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

Найважливішими з них є x:Name і x:Key.

x:Name використовується для елементів у дереві XAML, щоб мати можливість легко їх знайти у C#, а також для надання певної документації для XAML і для того, щоб можна було легко ідентифікувати елемент всередині інструменту розробки Avalonia (Avalonia Development Tool).

Для того, щоб знайти елемент x:Name у коді C#: можна використовувати метод FindControl(...), наприклад, для кнопки, визначеної в XAML із використанням префіксу «x:»:

```
x:Named "CloseWindowButton"
```

ми можемо використовувати такий метод в коді C#, щоб знайти його:

```
var button = this.FindControl<button>("CloseWindowButton");</button>
```

Конструкція x:Key використовується для пошуку ресурсів Avalonia XAML.

5.4 Доступ до ресурсів Avalonia із XAML

5.4.1 Базові поняття

Розширення розмітки (Markup up extensions) – це деякі класи C#, які можуть значно спростити XAML. Вони використовуються для встановлення деяких властивостей XAML за допомогою позначень, які мають фігурні дужки («{« і «}»). Існує кілька дуже важливих вбудованих розширень розмітки Avalonia. Найважливішими є такі:

– StaticResource;

– DynamicResource;

– x:Static;

– Binding.

В даному підрозділі розглянемо приклади для всіх з них, окрім Binding, який було розглянуто в попередніх розділах.

Можна також створювати власні розширення розмітки, але це рідко використовується і не буде розглянуто в даному посібнику.

Ресурси XAML є одним із найважливіших методів повторного використання коду XAML та розміщення деяких загальних кодів XAML у загальних візуальних проєктах для використання в кількох програмах.

5.4.2 Порівняння StaticResource та DynamicResource

Розглянемо приклад, що показує основну відмінність між статичними та динамічними ресурсами: цільове значення статичного ресурсу не буде оновлюватися при оновленні самого ресурсу, а в разі динамічного – оновиться.

Створимо новий проєкт Avalonia під назвою ResourseExample. В новому проєкті використовуємо такий XAML файл:

```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        x:Class="NP.Demos.StaticVsDynamicXamlResourcesSample.MainWindow"
        Title="NP.Demos.StaticVsDynamicXamlResourcesSample"
        Width="300"
        Height="200">
  <Window.Resources>
    <ResourceDictionary>
      <!--We set the XAML resource-->
      <SolidColorBrush x:Key="StatusBrush"
                       Color="Green"/>
    </ResourceDictionary>
  </Window.Resources>

  <Grid>
    <Grid.RowDefinitions>
      <RowDefinition Height="*"/>
      <RowDefinition Height="Auto"/>
    </Grid.RowDefinitions>
    <StackPanel x:Name="ElementsPanel"
                Orientation="Vertical">
    <!--Refer to xaml resource using StaticResource Markup Expression -->
    <Border x:Name="Border1"
            Background="{StaticResource StatusBrush}"
            Height="30"
            Width="80"
            Margin="0,5"/>

    <!--Refer to xaml resource using StaticResource (without markup
expression) -->
    <Border x:Name="Border2"
            Height="30"
            Width="80"
            Margin="0,5">
        <Border.Background>
          <StaticResource ResourceKey="StatusBrush"/>
        </Border.Background>
    </Border>

    <!--Refer to xaml resource using DynamicResource Markup Expression -->
      <Border x:Name="StatusChangingBorder"
              Background="{DynamicResource StatusBrush}"
```

```
            Height="30"
            Width="80"
            Margin="0,5"/>
    </StackPanel>
    <Button x:Name="ChangeStatusButton"
            Grid.Row="1"
            Width="160"
            HorizontalAlignment="Right"
            HorizontalContentAlignment="Center"
            Content="Change Status Color"
            Margin="10"/>
  </Grid>
</Window>
```

Після першого запуску отримаємо наступний вигляд основного вікна програми (рис. 5.11).

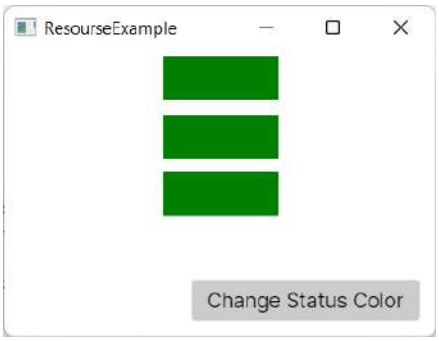


Рисунок 5.11 – Запуск тестової програми для демонстрації доступу до ресурсів XAML

Натискання на кнопку "Change Status Color" поки що не дає ніякого результату тому що ми ще не реалізували логіку роботи програми у коді C#.

Ми визначаємо ресурс XAML у тому самому файлі MainWindow.xaml, як ресурс вікна наступним чином:

```
<Window.Resources>
  <ResourceDictionary>
    <!--We set the XAML resource-->
    <SolidColorBrush x:Key="StatusBrush"
                     Color="Green"/>
  </ResourceDictionary>
</Window.Resources>
```

x:Key ресурсу XAML може використовуватися для посилання на певний ресурс як StaticResource так і DynamicResource. Ми використовуємо StaticResource, щоб встановити фон двох перших компонентів Border, а DynamicResource – для третього компонента.

Для першого компонента Border ми використовуємо розширення розмітки StaticResource:

```
<!--Refer to xaml resource using StaticResource Markup Expression -->
<Border x:Name="Border1"
      Background="{StaticResource StatusBrush}"
      Height="30"
      Width="80"
      Margin="0,5"/>
```

Для другого компонента Border ми використовуємо клас StaticResource без розширення розмітки (з поданого коду можна бачити, що відповідний XAML є значно більш докладним):

```
<Border x:Name="Border2"
        Height="30"
        Width="80"
        Margin="0,5">
  <Border.Background>
    <StaticResource ResourceKey="StatusBrush"/>
  </Border.Background>
</Border>
```

Нарешті, третій компонент Border використовує розширення розмітки DynamicResource:

```
<!--Refer to xaml resource using DynamicResource Markup Expression -->
<Border x:Name="StatusChangingBorder"
      Background="{DynamicResource StatusBrush}"
      Height="30"
      Width="80"
      Margin="0,5"/>
```

Для роботи прикладу необхідно підключити кнопку «StatusChangingBorder» до файлу MainWindow.xaml.cs, щоб мати змогу змінити ресурс «StatusBrush» із «зеленого» на «червоний»:

```
public MainWindow()
{
    InitializeComponent();
    Button button =
            this.FindControl<Button>("ChangeStatusButton");
    button.Click += Button_Click;
}

private void Button_Click(object? sender,
            Avalonia.Interactivity.RoutedEventArgs e)
{
    // getting a Window resource by its name
    var statusBrush = this.FindResource("StatusBrush");

    // setting the window resource to a new value
    this.Resources["StatusBrush"] =
            new SolidColorBrush(Colors.Red);
}
```

Тепер, після запуску програми ми отрумаємо можливість змінювати колір третього компонента натисканням на кнопку (рис. 5.12).

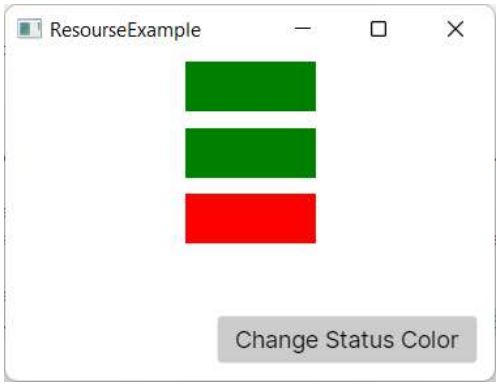


Рисунок 5.12 – Приклад роботи з ресурсами XAML Avalonia

Незважаючи на те, що ресурс однаковий для всіх трьох компонентів, змінюється лише фон останнього – той, який використовує DynamicResource.

Інші важливі відмінності між статичним і динамічним ресурсом полягають у наступному:

– DynamicResource може посилатися на ресурс XAML, визначений у XAML нижче застосування виразу DynamicResource, тоді як StaticResource має посилатися на ресурс, що описаний вище над ним;

– StaticResource можна використовувати для призначення простих властивостей C# різним об'єктам, що використовуються в XAML, тоді як метою оператора DynamicResource завжди має бути спеціальна властивість Avalonia для AvaloniaObject (наприклад, вкладені властивості);

– оскільки DynamicResource є потужнішим (надає сповіщення про зміни), він займає значно більше ресурсів пам'яті, ніж StaticResource. Через це, коли нам не потрібне сповіщення про зміну (властивість залишається незмінною протягом усього терміну дії програми), ми завжди повинні використовувати StaticResource. DynamicResources дуже корисні, коли необхідно динамічно змінювати теми або кольори нашої програми, наприклад, дозволити користувачеві перемикати тему або змінювати кольори залежно від часу доби.

Повний код розглянутого прикладу надано в лістингу «Listing_5-3».

5.4.3 Посилання на ресурси XAML, які визначені в різних файлах XAML і зразках проєктів

У цьому прикладі ми розглянемо, як звертатися до ресурсів XAML, розташованих в іншому файлі в межах одного або іншого проєкту. Для цього створюємо новий проєкт під назвою MultipleResourse. В проєкті створюємо додаткову папку Dependencies для розміщення іншого проєкту DependencyProject типу ClassLibrary, в я кому будуть зберігатись ресурси теми.

Також в новому проєкті створимо папку Themes та розмістимо в ньому новий файл XAML «BrushResources.axaml», що створюється на основі шаблону ResourceDictionary (рис. 5.13):

```
<ResourceDictionary
      xmlns="https://github.com/avaloniaui"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <!-- Add Resources Here -->
      <SolidColorBrush x:Key="RedBrush"
                       Color="Red"/>
</ResourceDictionary>
```

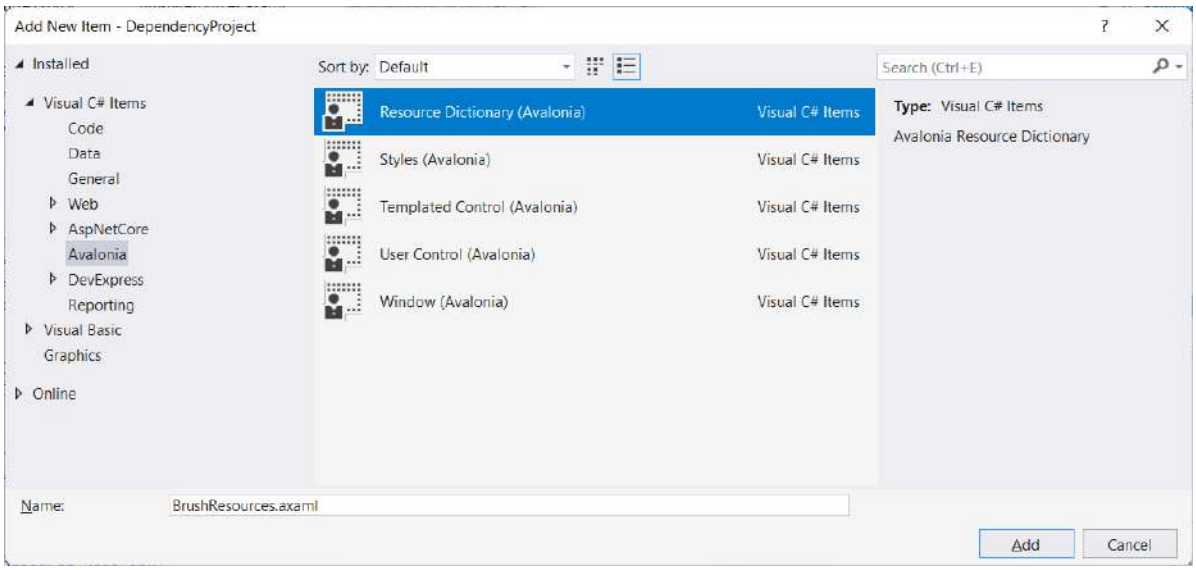


Рисунок 5.13 – Діалогове вікно створення файлу ресурсів

Далі створюємо папку Themes в основному проєкті та додаємо до неї новий файл LocalBrushResources.axaml, що створюється також на основі шаблону ResourceDictionary:

```
<ResourceDictionary
      xmlns="https://github.com/avaloniaui"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <!-- Add Resources Here -->
      <SolidColorBrush x:Key="GreenBrush"
                       Color="Green"/>
</ResourceDictionary>
```

Для кожного із файлів ресурсів необхідно переконатися, що обрана властивість «Build action» відповідає «Avalonia XAML» (рис. 5.14).

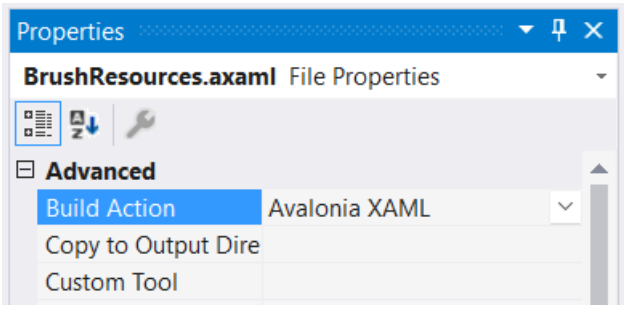


Рисунок 5.14 – Значення властивості «Build action»

Додаємо в залежності основного проєкту проєкт, що було створено. Таким чином структура рішення повинна бути такою, як на рис. 5.15.

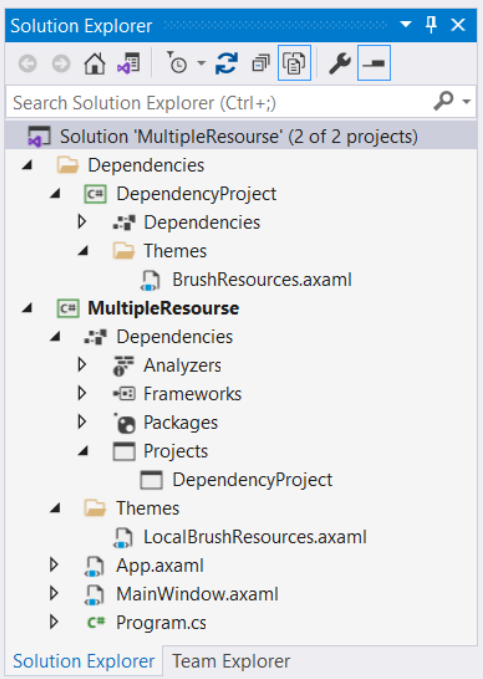


Рисунок 5.15 – Структура рішення MultipleResourse

Якщо в процесі компіляції проєкту виникне помилка «Problem with xaml compliation in AvaloniaUI», це означає, що процес MSBuild компілятора XAML Avalonia не обробив жодного XAML для SchemesView. Це може трапитись по кільком причинам:

– не було додано файл XAML як AvaloniaResource або EmbeddedResource;

– x: Директива класу відсутня або недійсна;

– проєкт безпосередньо не посилається на жоден пакет Avalonia. Починаючи з версії 0.9.x, потрібне пряме посилання через сумісність із пакетом SDK .NET Core 2.1, який не підтримує buildTransitive.

Перед компіляцією проєкту обов’язково слід перевірити чи додані необхідні пакети до проєкту DependencyProject (рис. 5.16):

Вміст основного файлу розмітки MainWindow.axaml має бути наступним:

```
<Window xmlns="https://github.com/avaloniaui"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      x:Class="MultipleResourse.MainWindow"
      Title="MultipleResourse"
      Width="100"
```

```
        Height="180">
    <Window.Resources>
      <ResourceDictionary>
        <ResourceDictionary.MergedDictionaries>
          <ResourceInclude
Source="avares://DependencyProject/Themes/BrushResources.axaml"/>
          <ResourceInclude Source="/Themes/LocalBrushResources.axaml"/>
        </ResourceDictionary.MergedDictionaries>

        <!-- BlueBrush is defined locally -->
        <SolidColorBrush x:Key="BlueBrush"
                         Color="Blue"/>
      </ResourceDictionary>
    </Window.Resources>
    <StackPanel HorizontalAlignment="Center"
                VerticalAlignment="Center">
      <Border x:Name="RedBorder"
              Width="70"
              Height="30"
              Background="{StaticResource RedBrush}"
              Margin="5"/>
      <Border x:Name="GreenBorder"
              Width="70"
              Height="30"
              Background="{StaticResource GreenBrush}"
              Margin="5"/>
      <Border x:Name="BlueBorder"
              Width="70"
              Height="30"
              Background="{StaticResource BlueBrush}"
              Margin="5"/>
    </StackPanel>
  </Window>
```

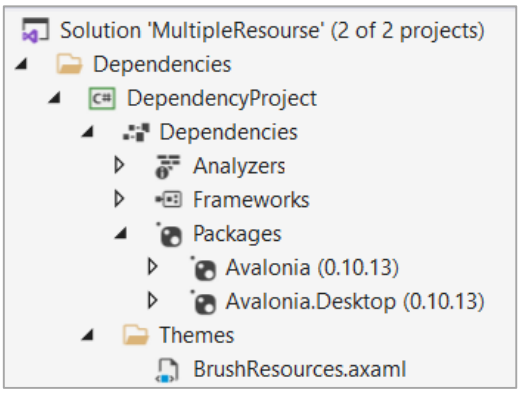


Рисунок 5.16 – Необхідні для роботи пакети Avalonia

Після вдалої компіляції на екрані повинно бути відображено наступне вікно (рис. 5.17).

Рисунок 5.17 – Демонстрація роботи програми MultipleResourse

Розберемо структуру нашого основного файлу розмітки. У нас є три компоненти Border, які розміщено вертикально – фон першого компонента отримує значення від ресурсу RedBrush, другий компонент – від GreenBrush і третій – від BlueBrush.

Розглянемо розділ «Ресурси» у вікні у верхній частині файлу MainWindow.axaml:

```
<Window.Resources>
  <ResourceDictionary>
      <ResourceDictionary.MergedDictionaries>

        <ResourceInclude
        Source="avares://DependencyProject/Themes/BrushResources.axaml"/>
        <ResourceInclude Source="/Themes/LocalBrushResources.axaml"/>
      </ResourceDictionary.MergedDictionaries>

      <!-- BlueBrush is defined locally -->
      <SolidColorBrush x:Key="BlueBrush"
                       Color="Blue"/>
  </ResourceDictionary>
</Window.Resources>
```

Теги <ResourceInclude .../> в тегові <ResourceDictionary.MergedDictionary> означають, що ми об’єднуємо словники ресурсів, визначені зовні, з поточним словником – так, як ми отримуємо всі їхні пари ключ-значення. На відміну від WPF ми використовуємо тег <ResourceDictionary Source="..."/>, а не <ResourceInclude Source="..."/>. Також треба зауважити, що для залежного проєкту ми не відокремлюємо збірку від решти URL-адрес та не використовуємо загадковий префікс «Component/» для URL-адреси. Це суто нотаційні, а не концептуальні відмінності, але їх все одно потрібно пам’ятати.

Особливості URL-адреси Avalonia XAML для об’єднаних файлів:

– "avares://Dependency1Proj/Themes/BrushResources.axaml" – URL-адреса файлу ресурсу Avalonia XAML, визначеного в іншому проєкті, має починатися з префіксу «avares://», за яким йде ім’я збірки, а потім шлях до файлу:

```
avares://assembly-name/path-to-the-avalonia-resource_file;
```

– "/Themes/LocalBrushResources.axaml" – URL-адреса файлу ресурсу Avalonia XAML, визначеного в тому самому проєкті, в якому він використовується, має складатися лише з косої риски, за якою має слідувати шлях до файлу ресурсу Avalonia з кореня поточного проєкту.

У кінці розділу ресурсів визначаємо ресурс BlueBrush – локальний для файлу MainWindow.axaml.

5.4.4 Розширення розмітки «x:Static»

Розширення розмітки «x:Static» дозволяє посилатися на статичні властивості, визначені в тому самому проєкті або в деяких залежних проєктах.

Для дослідження даної властивості створимо новий проєкт XStaticMarkupExtension, що буде містити залежний проєкт DependencyProject. Основний проєкт буде містити клас LocalProjectStaticBrushes, а залежний – DependencyProjectStaticBrushes.

В DependencyProject створрюємо клас DependencyProjectStaticBrushes.cs із таким вмістом:

```
using Avalonia.Media;

namespace DependencyProject
{
```

```
    public class DependencyProjectStaticBrushes
    {
        public static Brush RedBrush { get; set; } =
            new SolidColorBrush(Colors.Red);
    }
}
```

В основному проєкті створюємо клас LocalProjectStaticBrushes.cs із наступним вмістом:

```
using Avalonia.Media;

namespace XStaticMarkupExtension
{
    public class LocalProjectStaticBrushes
    {
        public static Brush GreenBrush { get; set; } =
            new SolidColorBrush(Colors.Green);
    }
}
```

Вміст обох файлів C# дуже простий – кожен визначає та встановлює значення для однієї статичної властивості: RedBrush та GreenBrush.

В результаті отримаємо наступну структуру проєкту (рис. 5.18).

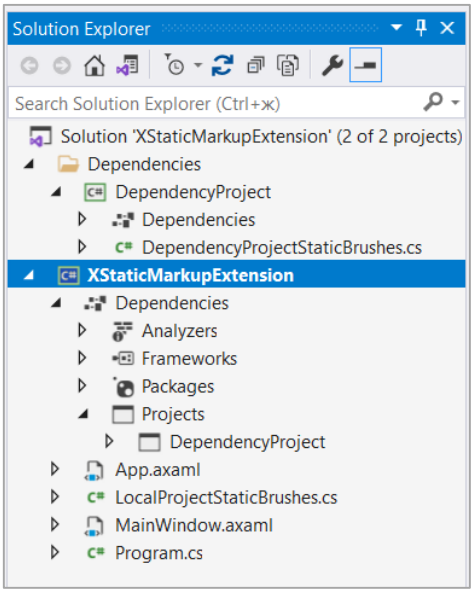


Рисунок 5.18 – Структура проєкту XStaticMarkupExtension

В основному файлі MainWindow.axaml описується структура вікна:

```
<Window xmlns="https://github.com/avaloniaui"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      Width="100"
      Height="180"
      x:Class="XStaticMarkupExtension.MainWindow"
      xmlns:dep1="clr-namespace:DependencyProject;
            assembly=DependencyProject"
      xmlns:local="clr-namespace:XStaticMarkupExtension"
      Title="XStaticMarkupExtension">
      <StackPanel HorizontalAlignment="Center"
                  VerticalAlignment="Center">
            <Border Width="70"
                    Height="30"
                    Background="{x:Static
                        dep1:DependencyProjectStaticBrushes.RedBrush}"
                    Margin="5" />
            <Border Width="70"
                    Height="30"
                    Background="{x:Static
                        local:LocalProjectStaticBrushes.GreenBrush}"
                  Margin="5" />
      </StackPanel>
</Window>
```

Результат компіляції проєкту подано на рис. 5.19.

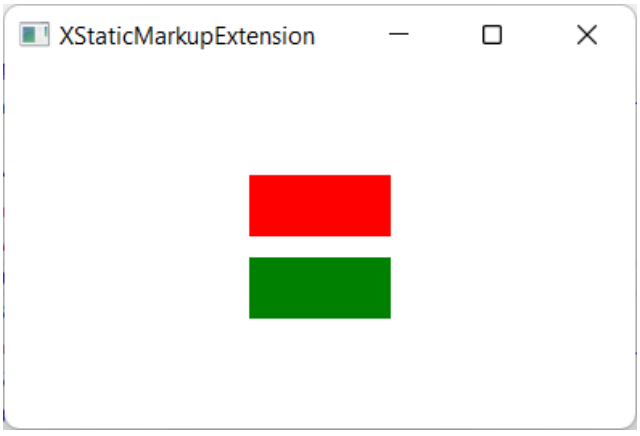


Рисунок 5.19 – Результат компіляції та запуску проєкту XStaticMarkupExtension

Розглянемо файл MainWindow.axaml. На рівні тегу Window визначено два важливих простору імен:

```
xmlns:dep1="clr-namespace:DependencyProject;assembly= DependencyProject"
xmlns:local="clr-namespace:XStaticMarkupExtension"
```

«dep1» відповідає залежному проєкту, а «local» відповідає локальному проєкту для файлу MainWindow.xaml, що знаходиться в одномовному проєкті. Використовуючи ці префікси простору імен і розширення x:Static, ми можемо встановити фонові властивості на двох компонентах Border:

```
Background="{x:Static dep1:DependencyProjectStaticBrushes.RedBrush}"
```

та

```
Background="{x:Static local:LocalProjectStaticBrushes.GreenBrush}".
```

Повний код проєкту наведено в лістингу «Listing_5-5».

5.4.5 Дженерики в Avalonia XAML

В програмуванні іноді необхідно обробляти різні типи даних схожими способами. З іншого боку, в мовах програмування зі статичною типізацією, параметри функцій типізовані. Це означає, що якщо функція приймає ціле число, його не можна перетворити на реальне.

Успадкування вирішує частину проблеми, оскільки ми можемо призначити дочірні об’єкти батьківським змінним:

```
fun main() {
    val a: Int = 10
    val b: Double = 1.5
    fraction(a) // 2.0
    fraction(b) // 0.3
}

fun fraction(n: Number) {
    println(n.toDouble() / 5)
}
```

У цьому випадку Number – це клас, який є батьківським для числових типів даних. Однак це не завжди підходить. Наприклад, ми хотіли б повернути дані попередньо невідомого типу з функції.

Деякі мови програмування дозволяють створювати так звані узагальнені (generics) функції і класи. Розглянемо приклад визначення і виклику узагальненої функції в мові Котліна.

Нещодавно Avalonia XAML стала також підтримувати дженерики. Для демонстрації такої можливості створим новий проєкт і дамо йому назву XamlGenerics. В новому проєкті створимо клас ValuesContainer.cs із таким вмістом:

```
public class ValuesContainer<TVal1, TVal2>
{
    public TVal1? Value1 { get; set; }
    public TVal2? Value2 { get; set; }
}
```

ValuesContainer визначає два значення Value1 загального типу TVal1 і Value2 загального типу TVal2.

Решта візуального коду знаходиться у файлі MainWindow.axaml, в якому реалізовано три приклади:

– перший пояснює створення одного об’єкта ValuesContainer;

– другий – список об’єктів ValuesContainer;

– третій – словник, який відображає цілі числа в об’єкті ValuesContainer.

Зразок об’єкта Single ValuesContainer:

```
<Grid RowDefinitions="Auto, Auto">
    <Grid.DataContext>
        <local:ValuesContainer x:TypeArguments="x:Double, x:String"
                               Value1="300"
                               Value2="Hello 1"/>
    </Grid.DataContext>
    <TextBlock Text="ValuesContainer Generics Sample:"
               FontWeight="Bold"/>
    <Grid Background="Yellow"
          Grid.Row="1"
          RowDefinitions="Auto, Auto"
          Width ="{Binding Path=Value1}"
```

```
            HorizontalAlignment="Left">
        <TextBlock Text="{Binding Path=Value1,
            StringFormat='Width of Yellow Rectangle=Value1=\{0\}'}"
                   Margin="5"/>
        <TextBlock Text="{Binding Path=Value2,
            StringFormat='Value2=\'\{0\}\''}"
                   Grid.Row="1"
                   Margin="5,0,5,5"/>
    </Grid>
</Grid>
```

Після запуску проєкту на екрані відобразиться наступне вікно (рис. 5.20).

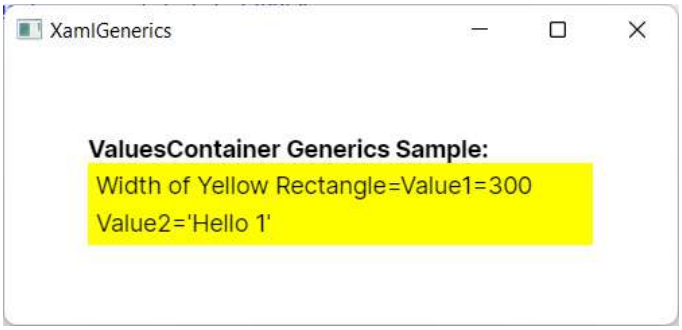


Рисунок 5.20 – Результат компіляції першого прикладу

В коді XAML ми визначаємо об'єкт ValuesContainer у секції DataContext об'єкта Grid, який містить такий код:

```
<Grid.DataContext>
    <local:ValuesContainer x:TypeArguments="x:Double, x:String"
                           Value1="300"
                           Value2="Hello 1"/>
</Grid.DataContext>
```

Властивість x:TypeArguments об'єкта ValuesContainer визначає список аргументів загальних типів, що розділені комами – ми визначаємо перший аргумент як Double, а другий як текстовий рядок (String). Потім ми встановлюємо Value1="300" і Value2="Hello 1". Зауважимо, що XML-рядок «300» буде автоматично перетворено в Double.

Оскільки DataContext є спеціальною властивістю, яка поширюється вниз по візуальному дереву, той самий DataContext буде визначено для всіх нащадків

Grid. Таким чином, ми можемо прив’язати властивості Text нащадків TextBlocks до Value1 і Value2, щоб відобразити ці значення. Крім того, щоб довести, що Value1 дійсно має подвійний тип (а не рядок), ми прив’язуємо ширину внутрішньої сітки (з жовтим фоном) до властивості Value1 DataContext:

```
<Grid Background="Yellow"
     ...
     Width ="{Binding Path=Value1}" ...>
```

Таким чином, ширина жовтого прямокутника буде 300 пікселів.

Розглянемо наступний зразок – список об’єктів ValuesContainer:

```
<Grid RowDefinitions="Auto, Auto">
   <Grid.DataContext>
      <collections:List
            x:TypeArguments="local:ValuesContainer(x:Int32, x:String)">
            <local:ValuesContainer x:TypeArguments="x:Int32, x:String"
                                   Value1="1"
                                   Value2="Hello 1"/>
            <local:ValuesContainer x:TypeArguments="x:Int32, x:String"
                                   Value1="2"
                                   Value2="Hello 2"/>
            <local:ValuesContainer x:TypeArguments="x:Int32, x:String"
                                   Value1="3"
                                   Value2="Hello 3"/>
      </collections:List>
   </Grid.DataContext>
   <TextBlock Text="List of ValuesContainer Generics Sample:"
                  FontWeight="Bold"/>
      <ItemsControl Items="{Binding}"
                  Grid.Row="1">
         <ItemsControl.ItemTemplate>
            <DataTemplate>
               <StackPanel Orientation="Horizontal">
                  <TextBlock Text="{Binding Path=Value1,
                     StringFormat=‘Value1=\{0\}’}"/>
                  <TextBlock Text="{Binding Path=Value2,
                     StringFormat=‘Value2=\’\{0\}\’’}"
                     Margin="10,0,0,0"/>
               </StackPanel>
            </DataTemplate>
```

```
        </ItemsControl.ItemTemplate>
    </ItemsControl>
</Grid>
```

Для його правильної роботи необхідно додати до шапки файлу XAML такий рядок:

```
xmlns:collections="clr-namespace:System.Collections.Generic;
assembly=System.Collections"
```

Результат компіляції проєкту подано на рис. 5.21.

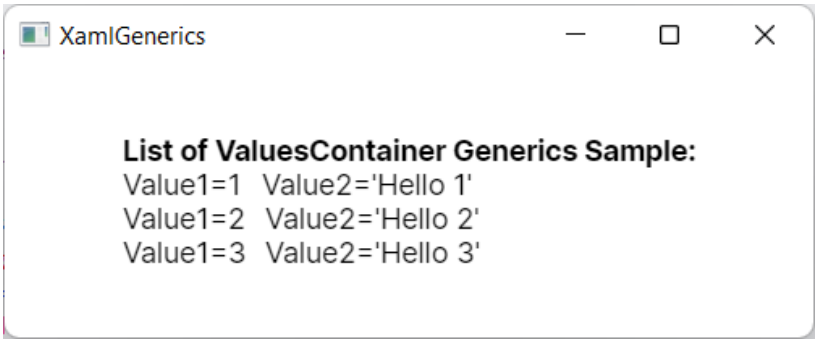


Рисунок 5.21 – Результат компіляції другого прикладу

Цього разу DataContext контейнера визначається у вигляді List<ValuesContainer<int, string>>, тобто ми маємо два рівні рекурсії аргументу типу:

```
<collections:List
    x:TypeArguments="local:ValuesContainer(x:Int32, x:String)">
  ...
</collections:List>
```

Оскільки List<...> має метод Add, ми можемо просто додати окремі об’єкти в List<...>:

```
<collections:List
          x:TypeArguments="local:ValuesContainer(x:Int32, x:String)">
  <local:ValuesContainer x:TypeArguments="x:Int32, x:String"
                         Value1="1"
                         Value2="Hello 1"/>
```

```
    <local:ValuesContainer x:TypeArguments="x:Int32, x:String"
                           Value1="2"
                           Value2="Hello 2"/>
    <local:ValuesContainer x:TypeArguments="x:Int32, x:String"
                           Value1="3"
                           Value2="Hello 3"/>
</collections:List>
```

Необхідно зауважити, що для кожного об’єкта ValuesContainer значення Value1 буде автоматично перетворено в int.

Далі ми прив’язуємо властивість Items елемента ItemsControl до списку і використовуємо ItemTemplate з ItemsControl для відображення Value1 і Value2 кожного окремого елемента:

```
<ItemsControl Items="{Binding}"
                  Grid.Row="1">
         <ItemsControl.ItemTemplate>
            <DataTemplate>
               <StackPanel Orientation="Horizontal">
                  <TextBlock Text="{Binding Path=Value1,
                     StringFormat=‘Value1=\{0\}’}"/>
                  <TextBlock Text="{Binding Path=Value2,
                     StringFormat=‘Value2=\’\{0\}\’’}"
                     Margin="10,0,0,0"/>
               </StackPanel>
            </DataTemplate>
         </ItemsControl.ItemTemplate>
      </ItemsControl>
```

Третій приклад присвячено використанню словника цілих чисел.

Для демонстрації даного прикладу необхідно додати до файлу XAML такий код:

```
<Grid RowDefinitions="Auto, Auto">
    <Grid.DataContext>
        <collections:Dictionary
            x:TypeArguments="x:String,
            local:ValuesContainer(x:Int32, x:String)">
            <local:ValuesContainer x:TypeArguments="x:Int32, x:String"
                                   x:Key="Key1"
                                   Value1="1"
```

```
                                Value2="Hello 1"/>
            <local:ValuesContainer x:TypeArguments="x:Int32, x:String"
                                x:Key="Key2"
                                Value1="2"
                                Value2="Hello 2"/>
            <local:ValuesContainer x:TypeArguments="x:Int32, x:String"
                                x:Key="Key3"
                                Value1="3"
                                Value2="Hello 3"/>
        </collections:Dictionary>
    </Grid.DataContext>
    <TextBlock Text="Dictionary of ValuesContainer Generics Sample:"
               FontWeight="Bold"/>
    <ItemsControl Items="{Binding}"
                  Grid.Row="1">
        <ItemsControl.ItemTemplate>
            <DataTemplate>
                <StackPanel Orientation="Horizontal">
                    <TextBlock Text="{Binding Path=Key,
                        StringFormat='Key=\'\{0\}\''}"/>
                    <TextBlock Text="{Binding Path=Value.Value1,
                        StringFormat='Value1=\{0\}'}"
                               Margin="10,0,0,0"/>
                    <TextBlock Text="{Binding Path=Value.Value2,
                        StringFormat='Value2=\'\{0\}\''}"
                               Margin="10,0,0,0"/>
                </StackPanel>
            </DataTemplate>
        </ItemsControl.ItemTemplate>
    </ItemsControl>
</Grid>
```

Для визначення словника використовуємо таку конструкцію:

```
Dictionary<string, ValuesContainer<int, string>>:
<collections:Dictionary x:TypeArguments="x:String,
      local:ValuesContainer(x:Int32, x:String)">
```

Словник заповнюється таким чином:

```
<collections:Dictionary
x:TypeArguments="x:String, local:ValuesContainer(x:Int32, x:String)">
    <local:ValuesContainer x:TypeArguments="x:Int32, x:String"
```

```
                            x:Key="Key1"
                            Value1="1"
                            Value2="Hello 1"/>
    <local:ValuesContainer x:TypeArguments="x:Int32, x:String"
                            x:Key="Key2"
                            Value1="2"
                            Value2="Hello 2"/>
    <local:ValuesContainer x:TypeArguments="x:Int32, x:String"
                            x:Key="Key3"
                            Value1="3"
                            Value2="Hello 3"/>
</collections:Dictionary>
```

Необхідно зауважити, що ми створюємо об’єкти ValuesContainer у словнику, але кожен з об’єктів має властивість x:Key, що встановлено на унікальне значення. Ця властивість x:Key визначає ключ для словника, тоді як об’єкт ValuesContainer стає значенням. В нашому випадку ключ словника має тип string, але якщо він був іншого типу, наприклад, int, то компілятор Avalonia XAML перетворив би значення ключа в ціле число.

Наведений вище код XAML заповнює наш словник трьома об’єктами KeyValuePair<string, ValuesContainer<int, string>>, у форматі json як представлено нижче:

```
[
{ Key="Key1"
  Value = new ValuesContainer{ Value1=1, Value2="Hello 1"},
{ Key="Key2"
  Value = new ValuesContainer{ Value1=2, Value2="Hello 2"},
{ Key="Key3"
  Value = new ValuesContainer{ Value1=3, Value2="Hello 3"}
]
```

Ось як ми прив’язуємо наш ItemsControl до цих об’єктів і відображаємо їх за допомогою його ItemTemplate:

```
<ItemsControl Items="{Binding}"
              Grid.Row="1">
    <ItemsControl.ItemTemplate>
        <DataTemplate>
```

```
            <StackPanel Orientation="Horizontal">
                <TextBlock Text="{Binding Path=Key,
                                          StringFormat='Key=\'\{0\}\''}"/>
                <TextBlock Text="{Binding Path=Value.Value1,
                                           StringFormat='Value1=\{0\}'}"
                           Margin="10,0,0,0"/>
                <TextBlock Text="{Binding Path=Value.Value2,
                                          StringFormat='Value2=\'\{0\}\''}"
                           Margin="10,0,0,0"/>
            </StackPanel>
        </DataTemplate>
    </ItemsControl.ItemTemplate>
</ItemsControl>
```

Властивість Text першого TextBlock прив'язано до ключа KeyValuePair<...>, другого TextBlock – до Value1, третього до Value2.

Результат роботи третього прикладу подано на рис. 5.22.

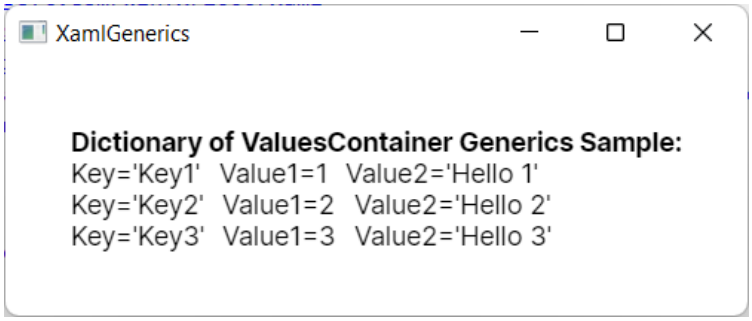


Рисунок 5.22 – Результат роботи третього прикладу

Повний код проєкту можна знайти в лістингу «Listing_5-6».

5.4.6 Використання ресурсів Assets в XAML

У Avalonia Lingo – Assets, як правило, є файлами двійкового зображення (наприклад, png або jpg). У цьому розділі ми покажемо, як звертатися до таких файлів із елементів керування у XAML. Для роботи із зображеннями створимо новий проєкт AssetsInXaml. В рішення додаємо залежний проєкт DependencyProject. В кожному проєкті створюємо папку Assets. В папку Assets проєкту DependencyProject скопіюємо файл Chip.png (рис. 5.23).

Chip.png

Рисунок 5.23 – Тестове зображення Chip.png

В папку Assets основного проєкту скопіюємо два файли зображення RobotArm.png та Lamp.png(рис. 5.24).

Lamp.png RobotArm.png

Рисунок 5.24 – Файли зображення Lamp.png та RobotArm.png

Після попередньої підготовки, структура рішення матиме вигляд, як подано на рис. 5.25.

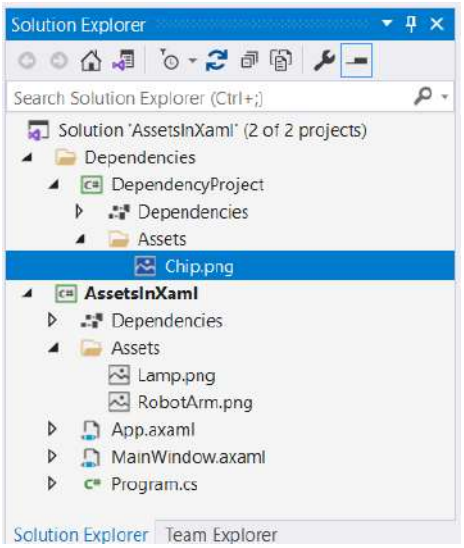


Рисунок 5.25 – Структура рішення AssetsInXaml

Для доступу до папок із ресурсами необхідно додати їх до файлів відповідного проєкту. Наприклад для основного проєкту додані рядки в файлі AssetsInXaml.cproj матимуть такий вигляд:

```
<ItemGroup>
     <AvaloniaResource Include="Assets\Lamp.png" />
```

```
        <AvaloniaResource Include="Assets\RobotArm.png" />
  </ItemGroup>
```

Для проєкту DependencyProject необхідно додати такі рядки:

```
  <ItemGroup>
        <AvaloniaResource Include="Assets\Chip.png" />
  </ItemGroup>
```

Для перевірки можливості доступу до ресурсів із зображенням, створимо такий XAML код:

```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        x:Class="AssetsInXaml.MainWindow"
        Title="AssetsInXaml"
            Width="100"
        Height="300">
  <StackPanel HorizontalAlignment="Center"
              VerticalAlignment="Center">
    <Image Source="/Assets/Lamp.png"
           Width="50"
           Height="50"
           Margin="5"/>
    <Image Source="avares://DependencyProject/Assets/Chip.png"
           Width="50"
           Height="50"
           Margin="5"/>
  </StackPanel>
</Window>
```

В результаті компіляції отримаємо наступне вікно користувача (рис. 5.26).

Виходячи із наведеного файлу, для доступу до зображень вони визначаються як локальний ресурс в тому проєкті, який містить файл MainWindow.axaml. В такому випадку для посилання можна використовувати спрощену версію вихідної URL-адреси:

```
Source="/Assets/Lamp.png"
```

Рисунок 5.26 – Приклад виводу зображення на екран

У той час коли зображення розташоване в іншому проєкті, має місце використання повної версію URL-адреси з avares://:

```
Source="avares://DependencyProject/Assets/Chip.png
```

Треба зауважити, що так само, як і у випадку файлів словника ресурсів XAML і на відміну від WPF, ім’я збірки (у нашому випадку «DependencyProject») є частиною URL-адреси і тому не використовується префікс компонента.

Розглянемо ще один приклад використання ресурсів, тільки зараз це буде відбуватись за допомогою коду C#. Для цього в файлі MainWindow.axaml.cs додамо такі рядки коду:

```
public MainWindow()
{
     InitializeComponent();
     // get the asset loader from Avalonia container
     var assetLoader = AvaloniaLocator.Current.GetService<IAssetLoader>();

     // get the Image control from XAML
     Image Image1 = this.FindControl<Image>("Image1");

     // set the image Source using assetLoader
     Image1.Source = new Bitmap (assetLoader?.Open(new
            Uri("avares://AssetsInXaml/Assets/RobotArm.png")));

     // get the Image control from XAML
     Image Image2 = this.FindControl<Image>("Image2");

     // set the image Source using assetLoader
```

```
        Image2.Source = new Bitmap (assetLoader?.Open(new
              Uri("avares://DependencyProject/Assets/Chip.png")));
    }
```

Для відображення різних способів виводу зображень на екран виконаємо модифікацію коду XAML. Додамо сітку Grid із двома стовбцями:

```
<Grid
      HorizontalAlignment="Center"
      VerticalAlignment="Center"
      ColumnDefinitions="120, 10, 120"
      RowDefinitions="Auto, Auto">
      <TextBlock
            Margin="0,0,0,10"
            HorizontalAlignment="Center"
            Grid.Row="0"
            Grid.Column="0">
                  XAML Resource
      </TextBlock>
      <TextBlock
            Margin="0,0,0,10"
            HorizontalAlignment="Center"
            Grid.Row="0"
            Grid.Column="2">
                  C# Resource
      </TextBlock>
      <StackPanel
            Grid.Row="1"
            Grid.Column="0"
            HorizontalAlignment="Center"
            VerticalAlignment="Center">
            <Image Source="/Assets/Lamp.png"
               Width="50"
               Height="50"
               Margin="5"/>
            <Image Source="avares://DependencyProject/Assets/Chip.png"
               Width="50"
               Height="50"
               Margin="5"/>
      </StackPanel>
      <StackPanel
            Grid.Row="1"
```

```
            Grid.Column="2"
            HorizontalAlignment="Center"
            VerticalAlignment="Center">
            <Image x:Name="Image1"
               Width="50"
               Height="50"
               Margin="5"/>
            <Image x:Name="Image2"
               Width="50"
               Height="50"
               Margin="5"/>
      </StackPanel>
</Grid>
```

В результаті компіляції отримаємо вигляд головного вікна, як подано на рис. 5.27.

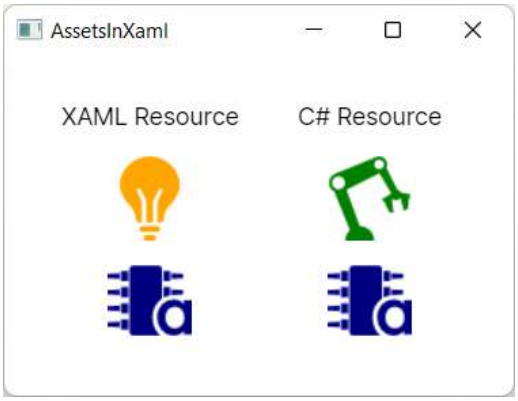


Рисунок 5.27 – Вигляд головного вікна після доповнення коду XAML

На даному рисунку зліва розташовані зображення виведені за допомогою XAML, а справа – за допомогою коду C#.

Зверніть увагу, що навіть для локального файлу «RobotArm.png» (файл, визначений у тому самому проєкті, що й файл MainWindow.xaml.cs, який його використовує), нам потрібно надати повну URL-адресу в форматі

```
«avares://assembly-name/»
```

із префіксом «avares://».

Повністю код проєкту можна побачити в лістингу «Listing_5-7».

5.5 Контрольні запитання та завдання

1. Що таке Avalonia XAML і для чого він використовується?
2. Як здійснюється доступ до властивостей об'єктів у XAML?
3. Що таке складні властивості в XAML і як до них отримати доступ?
4. Які спеціальні властивості XAML ви знаєте та для чого вони використовуються?
5. Як отримати доступ до ресурсів Avalonia із XAML?
6. Яка різниця між StaticResource і DynamicResource у XAML?
7. Як здійснити посилання на ресурси XAML, які визначені в різних файлах?
8. Як використовуються дженерики в Avalonia XAML?

6 ЗАСТОСУВАННЯ ЕЛЕМЕНТІВ КОРИСТУВАЧА В AVALONIA

6.1 Концепція маршрутизації подій

Так само, як і в WPF, Avalonia має концепцію приєднаних маршрутизованих подій, які поширюються вгору і вниз по візуальному дереву, але вони потужніші та з ними легше працювати, ніж WPF Routed Events.

На відміну від звичайних подій C#, Attached Routed Events:

– можуть бути визначені за межами класу, який їх запускає, і можуть бути «приєднані» до об’єктів;

– можуть поширюватися вгору і вниз по візуальному дереву WPF – у тому сенсі, що подія може бути запущена одним вузлом дерева та оброблена іншим вузлом дерева (одним із предків вузла, який запущений).

Існує три різні режими розповсюдження маршрутизованих подій:

– *прямий* – це означає, що подію можна обробляти лише в тому самому вузлі візуального дерева, який її запускає;

– *бульбашковий* – подія рухається від поточного вузла (вузла, який викликає подію) до кореня візуального дерева і може бути оброблена будь-де на шляху (рис. 6.1). Наприклад, якщо візуальне дерево складається з вікна, що містить сітку, що містить кнопку, і подія, що виникає, запускається на кнопці, то ця подія буде переміщатися у гору від кнопки до сітки, а потім до вікна;

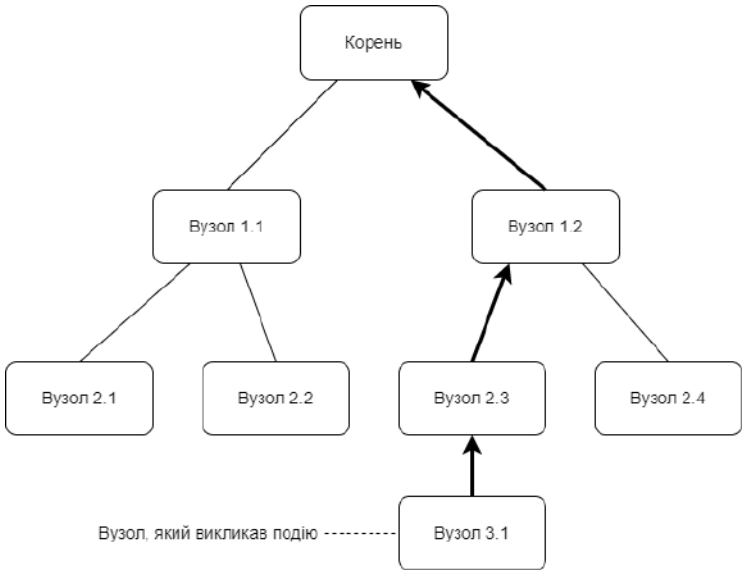


Рисунок 6.1 – Бульбашковий метод розповсюдження події

– тунелювання – подія рухається від кореневого вузла візуального дерева до поточного вузла (вузла, який викликає подію) (рис. 6.2). Використовуючи той самий приклад, що й вище, подія тунелювання спочатку буде викликана у вікні, потім у сітці і, нарешті, на кнопці.

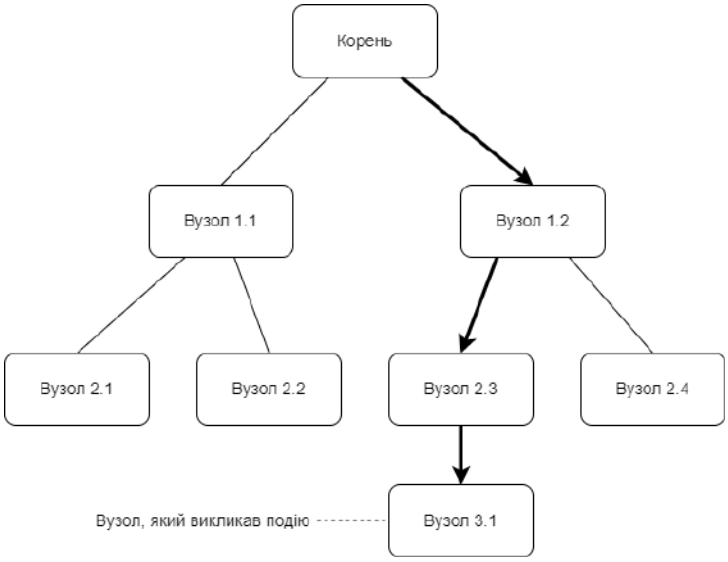


Рисунок 6.2 – Метод тунелювання

Маршрутизовані події Avalonia є більш потужними та логічними, ніж їхні аналоги WPF, через те, що в WPF подія має вибрати лише одну зі стратегій маршрутизації – вона може бути як прямою, бульбашковою або тунельною. Щоб дозволити деяку попередню обробку перед передачею основних (зазвичай бульбашкових) подій, багато подій, які запускаються перед ними, мають статус тунельних однорангових – так звані попередні події. Попередні події є абсолютно незалежними подіями в WPF, і між ними немає логічного зв’язку (окрім їхніх імен) та відповідності виникнення подій.

В Avalonia можна зареєструвати одну й ту саму подію, щоб мати кілька стратегій маршрутизації – так звані попередні події більше не потрібні – оскільки одну й ту саму подію можна спочатку викликати як подію тунелювання (використовується для попереднього перегляду), а потім як подію, що спалахує – виконання дії в реальному часі. Це також може призвести до помилок, наприклад, якщо ви обробляєте подію однаково в стані тунелювання та спалахування – подія може оброблятися двічі замість одного разу. Проста фільтрація під час підписки на обробник подій або проста перевірка в обробнику подій вирішить цю проблему.

Розглянемо приклад вбудованої маршрутизованої події.

В Avalonia вже існує багато маршрутизованих подій (як у WPF є багато вбудованих подій). Ми продемонструємо розповсюдження події маршрутизації за допомогою маршрутизованої події PointerPressedEvent, яка запускається, коли користувач натискає кнопку миші на якомусь візуальному елементі в Avalonia. Маршрутизована подія WPF LeftMouseButtonDown дуже схожа на PointerPressedEvent в Avalonia.

Для дослідження методів роботи із подіями створимо новий проєкт під назвою BuiltInRoutedEvent. Код XAML буде дуже простий:

```
<Window x:Name="TheWindow
        xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="BuiltInRoutedEvent"
        x:Class="BuiltInRoutedEvent.MainWindow"
        Background="Red"
        Width="200"
        Height="200">
  <Grid x:Name="TheRootPanel"
        Background="Green"
        Margin="35">
      <Border x:Name="TheBorder"
              Background="Blue"
              Margin="35"/>
  </Grid>
</Window>
```

Після запуску програми на екрані буде відображено три різнокольорові області (рис. 6.3).

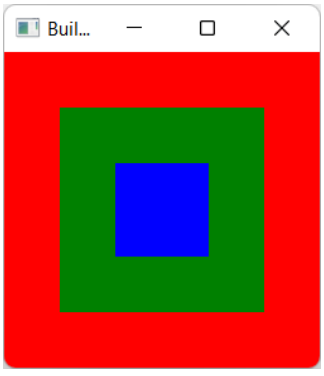


Рисунок 6.3 – Тестова форма для дослідження подій

У нас є Window (з червоним фоном), що містить Grid із зеленим фоном, що містить Border з синім фоном.

Для виникнення та обробки подій необхідно додати в проєкт код на C# для MainWindow:

```
public MainWindow()
{
      InitializeComponent();
#if DEBUG
      this.AttachDevTools();
#endif
      // add event handler for the Window
      this.AddHandler
      (
            Control.PointerPressedEvent, //routed event
            HandleClickEvent, // event handler
            RoutingStrategies.Bubble | RoutingStrategies.Tunnel // filter
      );

      Grid rootPanel = this.FindControl<Grid>("TheRootPanel");
      // add event handler for the Grid
      rootPanel.AddHandler (
            Control.PointerPressedEvent,
            HandleClickEvent,
            RoutingStrategies.Bubble | RoutingStrategies.Tunnel);

      Border border = this.FindControl<Border>("TheBorder");
      // add event handler for the Blue Border in the middle
      border.AddHandler(
            Control.PointerPressedEvent,
            HandleClickEvent,
            RoutingStrategies.Bubble | RoutingStrategies.Tunnel);
}
```

В даному коді до трьох візуальних компонентів (вікно, сітка, обмежена область) додаються обробники подій, в яких реєструються режими Bubble та Tunnel.

Для обробки подій додається такий код:

```
private void HandleClickEvent(object? sender, RoutedEventArgs e)
{
```

```
        Control senderControl = (Control)sender!;
        string eventTypeStr = e.Route switch
        {
              RoutingStrategies.Bubble => "Bubbling",
              RoutingStrategies.Tunnel => "Tunneling",
              _ => "Direct"
        };

        Debug.WriteLine($"{eventTypeStr} Routed Event {e.RoutedEvent!.Name}
raised on {senderControl.Name}; Event Source is {(e.Source as
Control)!.Name}");
}
```

В даному коді також включається режим видачі повідомлень у вікно налагодження програми (Debug).

Після запуску програми, якщо клікнути лівою кнопкою миші в центрі вікна, отримуємо такий вивід інформації у вікні Debug (рис. 6.4).

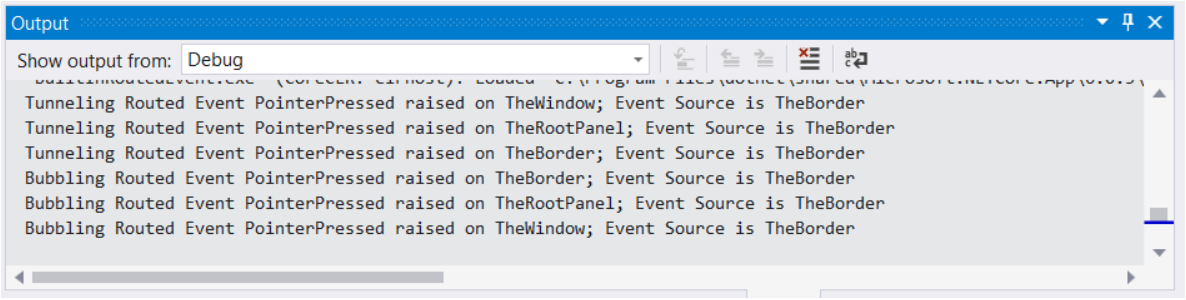


Рисунок 6.4 – Вивід інформації про виникнення подій у вікні Debug

З поданого рисунка можна бачити, що подія спочатку переміщується як «подія тунелювання» від вікна до центрального компонента, а потім як «подія, що випливає» у протилежному напрямку.

Розглянемо вихідний код C#. Спочатку ми призначаємо обробники вікна, сітки та межі за допомогою методу AddHandler. Давайте розглянемо один з них ближче:

```
// add event handler for the Window
this.AddHandler
(
    // routed event
    Control.PointerPressedEvent,
```

```
        // event handler
        HandleClickEvent,
        // routing strategy filter
        RoutingStrategies.Bubble | RoutingStrategies.Tunnel
    );
```

Першим аргументом AddHandler є RoutedEvent – статичний об'єкт, який містить карту візуальних об'єктів в обробники подій. Це аналогічно об'єкту AttachedProperty, який підтримує карту візуального об'єкта. Як і AttachedProperty, RoutedEvent можна визначити за межами класу і це не впливатиме на пам'ять, за винятком об'єктів, які мають для нього обробники.

Другим аргументом є метод обробника події HandleClickEvent. Ось реалізація методу:

```
private void HandleClickEvent(object? sender, RoutedEventArgs e)
{
    Control senderControl = (Control) sender!;
    string eventTypeString = e.Route switch
    {
        RoutingStrategies.Bubble => "Bubbling",
        RoutingStrategies.Tunnel => "Tunneling",
        _ => "Direct"
    };
    Debug.WriteLine($"{eventTypeStr} Routed Event {e.RoutedEvent!.Name}
    raised on {senderControl.Name};
    Event Source is {(e.Source as Control)!.Name}");
    ...
}
```

Все, що виконує даний код – це просто запис рядка до виводу Debug (для Visual Studio Debugger це означає, що він записує його в панель виведення).

Третій аргумент (RoutingStrategies.Bubble | RoutingStrategies.Tunnel) – це фільтр стратегії маршрутизації. Наприклад, якщо видалити з нього RoutingStrategies.Tunnel, він почне реагувати лише на запуск подій, що спалахують. За замовчуванням для нього встановлено значення RoutingStrategies.Direct | RoutingStrategies.Bubble.

Необхідно зауважити, що всі (або майже всі) вбудовані маршрутизовані події мають свої звичайні аналоги подій C#, які виникають, коли виникає

переспрямована подія. Ми могли б використати, наприклад, подію PointerPressed C#, щоб підключити її до обробника HandleClickEvent:

```
rootPanel.PointerPressed += HandleClickEvent;
```

Але в цьому випадку ми не зможемо вибрати фільтрацію RoutingStrategies (вона залишилася б за замовчуванням – RoutingStrategies.Direct | RoutingStrategies.Bubble). Крім того, ми не зможемо вибрати важливий аргумент handledEventsToo, який буде пояснений далі.

Для перехоплення події на певному елементі інтерфейсу додаємо наприкінці методу HandleClickEvent кілька додаткових рядків коду:

```
if (e.Route == RoutingStrategies.Bubble &&
      senderControl.Name == "TheBorder")
{
    e.Handled = true;
}
```

Мета цього коду полягає в тому, щоб встановити для події значення Handled, як тільки вона виконає всі тунелі та спалахує вперше на компоненті Border. Після запуску програми і натискання кнопкою миші на Border в панелі вихідних даних Visual Studio отримуємо наступні повідомлення (рис. 6.5).

```
Tunneling Routed Event PointerPressed raised on TheWindow; Event Source is TheBorder
Tunneling Routed Event PointerPressed raised on TheRootPanel; Event Source is TheBorder
Tunneling Routed Event PointerPressed raised on TheBorder; Event Source is TheBorder
Bubbling Routed Event PointerPressed raised on TheBorder; Event Source is TheBorder
```

Рисунок 6.5 – Повідомлення від компоненту Border

Оскільки подію було оброблено після першого виникнення на компоненті Border, обробники вище на візуальному дереві (ті, що знаходяться в сітці та вікні) більше не запускатимуться. Однак існує спосіб змусити їх запускати навіть спрямовану подію, яка вже була оброблена. Наприклад, щоб зробити це на рівні вікна, необхідно додати аргумент виклику AddHandler(...) у вікні:

```
// add event handler for the Window
this.AddHandler
(
```

```
        Control.PointerPressedEvent, //routed event
        HandleClickEvent, // event handler
        RoutingStrategies.Bubble | RoutingStrategies.Tunnel // filter
        ,true
    );
```

Останній аргумент називається handledEventsToo, і якщо він істинний, він запускає відповідний обробник також для подій, які оброблялися раніше. За замовчуванням це false. Після додавання даного оператору та запуску програми, знов натискаємо кнопку миші на Border. Результат подано на рис. 6.6. Останній рядок показує, що подію було згенеровано і оброблено у вікні, навіть якщо вона була позначена такою, що оброблена раніше.

```
Tunneling Routed Event PointerPressed raised on TheWindow; Event Source is TheBorder
Tunneling Routed Event PointerPressed raised on TheRootPanel; Event Source is TheBorder
Tunneling Routed Event PointerPressed raised on TheBorder; Event Source is TheBorder
Bubbling Routed Event PointerPressed raised on TheBorder; Event Source is TheBorder
Bubbling Routed Event PointerPressed raised on TheWindow; Event Source is TheBorder
```

Рисунок 6.6 – Обробка аргументу handledEventsToo

Тепер виконаємо дослідження розповсюдження подій за допомогою інструменту розробки Avalonia. Клацнувши мишею у вікні програми та натиснувши F12 викликаємо відповідне вікно інструментів. Далі клацнемо вкладку «Події» та з переліку подій, відображених на лівій панелі, виберемо PointerPressed, та відмінимо перевірку для решти з них (рис. 6.7).

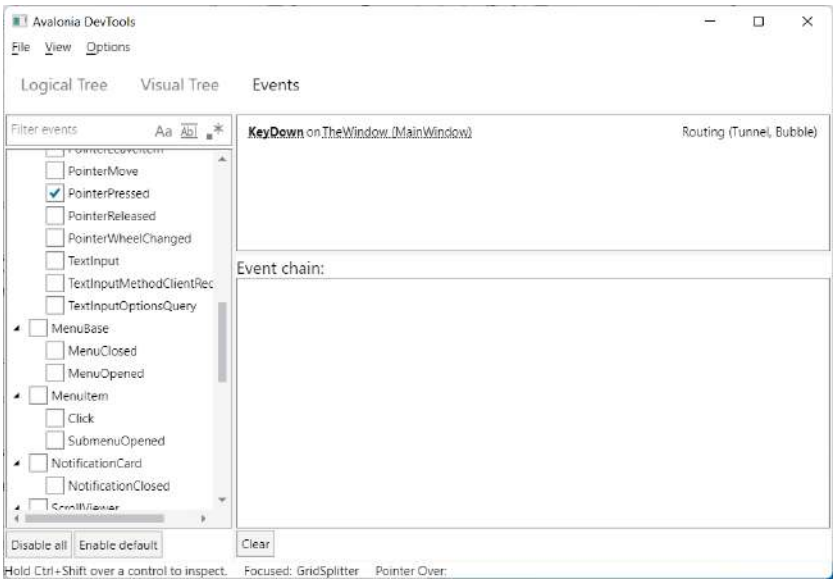


Рисунок 6.7 – Дослідження події PointerPressed

Після цього натискаємо на центральну рамку в програмі, і в головному вікні відобразиться запис події (рис. 6.8).

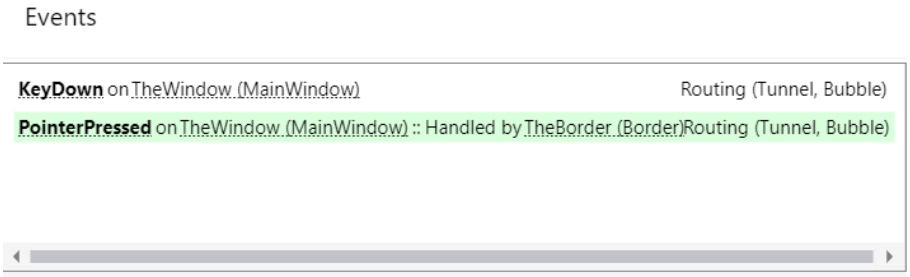


Рисунок 6.8 – Інформація про подію PointerPressed

Далі клацнемо мишкою запис про подію в головному вікні – панель «Ланцюжок подій» покаже, як подія розповсюджувалася по візуальному дереві (рис. 6.9).

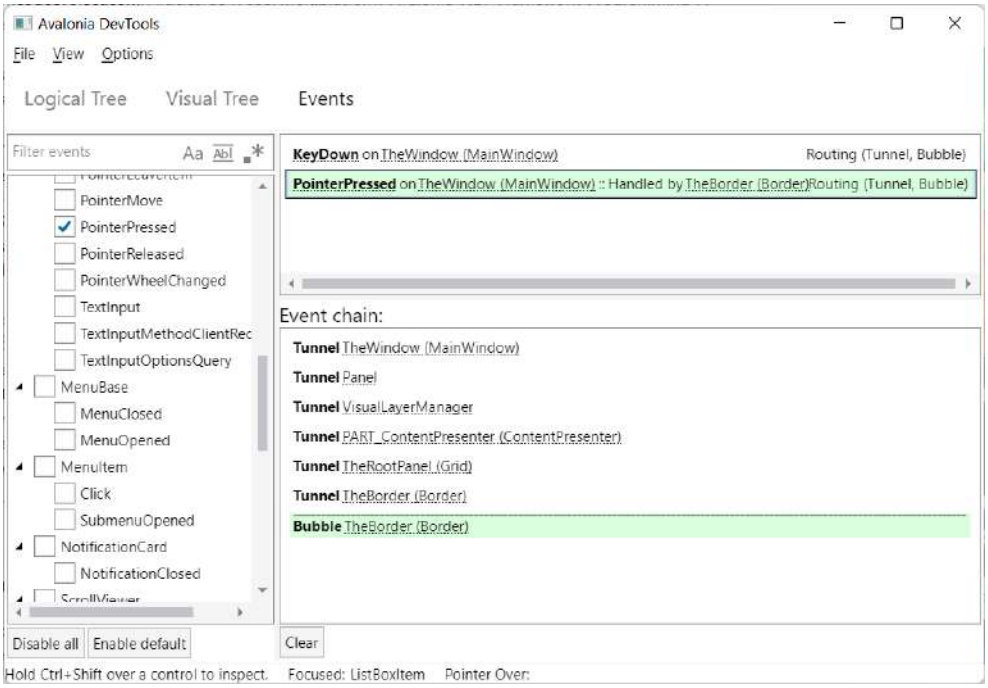


Рисунок 6.9 – Розповсюдження події

На жаль, наразі ланцюжок подій інструмента показує лише розповсюдження необробленої події. Він перестає відображатися в останній точці, коли подія була не оброблена – у нашому випадку, перший випадок обробки спалаху події (Bubble). Також можна побачити, що в інструменті показано більше випадків тунелювання нашої події, ніж у попередньому

прикладі. Це пов’язано з тим, що інструмент показує всі елементи візуального дерева, на якому виникає подія, тоді як ми підключили обробник лише до вікна, сітки та рамки.

Повний приклад прикладу наведено в лістингу «Listing_6-1»

6.2 Керування маршрутизованими подіями

Метою даного прикладу є запуск користувальницької маршрутизованої події MyCustomRoutedEvent, яка буде визначена у спеціально доданому для цього файлі, та відстеження її роботи.

Для даного прикладу створимо новий проєкт із назвою CustomRoutedEvent. Зміст XAML файлу такий же самий, що і в попередньому підрозділі:

```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        x:Class="CustomRoutedEvent.MainWindow"
        Title="CustomRoutedEvent"
            Background="Red"
            Width="200"
            Height="200">
      <Grid x:Name="TheRootPanel"
              Background="Green"
              Margin="35">
            <Border x:Name="TheBorder"
                        Background="Blue"
                        Margin="35"/>
      </Grid>
</Window>
```

Далі створюємо файл StaticRoutedEvents.cs із таким вмістом:

```
public static class StaticRoutedEvents
{
 public static readonly RoutedEvent<RoutedEventArgs> MyCustomRoutedEvent =
 RoutedEvent.Register<object, RoutedEventArgs>
 (
```

```
        "MyCustomRouted",
        RoutingStrategies.Tunnel //| RoutingStrategies.Bubble
    );
}
```

Як можна бачити, визначити подію дуже просто – необхідно просто викликати метод RoutedEvent.Register(...), передаючи назву події та стратегію маршрутизації.

Розглянемо використання нашої події. Для цього розглянемо код MainWindow.axaml.cs, де ми обробляємо MyCustomRoutedEvent:

```
// add event handler for the Window
this.AddHandler
(
    StaticRoutedEvents.MyCustomRoutedEvent, //routed event
    HandleCustomEvent, // event handler
    RoutingStrategies.Bubble | RoutingStrategies.Tunnel // routing strategy filter
);
```

Ми також додаємо наступний код, щоб згенерувати MyCustomRoutedEvent, коли миша натискається на синю область:

```
/// PointerPressed handler that raises MyCustomRoutedEvent
private void Border_PointerPressed(object? sender,
      PointerPressedEventArgs e)
{
    Control control = (Control)sender!;

    // Raising MyCustomRoutedEvent
    control.RaiseEvent(
            new RoutedEventArgs(StaticRoutedEvents.MyCustomRoutedEvent));
}
```

Наступний рядок коду викликає подію:

```
control.RaiseEvent(
      new RoutedEventArgs(StaticRoutedEvents.MyCustomRoutedEvent));
```

Повний код файлу MainWindow.axaml.cs наведено далі:

```
public partial class MainWindow : Window
{
   public MainWindow()
   {
      InitializeComponent();
#if DEBUG
      this.AttachDevTools();
#endif
      // add event handler for the Window
      this.AddHandler
      (
         StaticRoutedEvents.MyCustomRoutedEvent, //routed event
         HandleCustomEvent, // event handler
         RoutingStrategies.Bubble | RoutingStrategies.Tunnel // filter
      );

      Grid rootPanel = this.FindControl<Grid>("TheRootPanel");
      // add event handler for the Grid
      rootPanel.AddHandler (
         StaticRoutedEvents.MyCustomRoutedEvent,
         HandleCustomEvent,
         RoutingStrategies.Bubble | RoutingStrategies.Tunnel);
         Border border = this.FindControl<Border>("TheBorder");

      // add event handler for the Blue Border in the middle
      border.AddHandler(
         StaticRoutedEvents.MyCustomRoutedEvent,
         HandleCustomEvent,
         RoutingStrategies.Bubble | RoutingStrategies.Tunnel);

         // we add the handler to pointer pressed event in order
         // to raise MyCustomRoutedEvent from it.
      border.PointerPressed += Border_PointerPressed;
   }

   /// PointerPressed handler that raises MyCustomRoutedEvent
   rivate void Border_PointerPressed(object? sender,
      PointerPressedEventArgs e)
   {
      Control control = (Control)sender!;
```

```
        // Raising MyCustomRoutedEvent
        control.RaiseEvent(new
              RoutedEventArgs(StaticRoutedEvents.MyCustomRoutedEvent));
    }

    private void HandleCustomEvent(object? sender, RoutedEventArgs e)
    {
        Control senderControl = (Control)sender!;
        string eventTypeStr = e.Route switch
        {
            RoutingStrategies.Bubble => "Bubbling",
            RoutingStrategies.Tunnel => "Tunneling",
            _ => "Direct"
        };

        Debug.WriteLine($"{eventTypeStr} Routed Event {e.RoutedEvent!.Name}
        raised on {senderControl.Name};
        Event Source is {(e.Source as Control)!.Name}");
    }
}
```

Після компіляції проєкту і запуску програми, клацнемо синій квадрат посередині вікна. Панель виводу Visual Studio матиме надрукований текст, як подано на рис. 6.10:

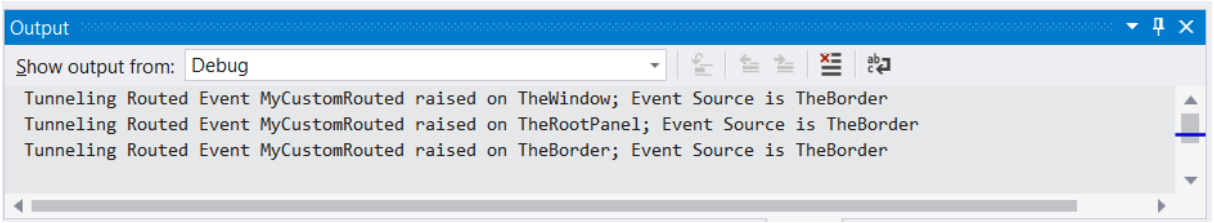


Рисунок 6.10 – Системне виведення у вікно налагодження

З рис. 6.10 можна побачити, що обробляється тільки проходження тунелю. Це через те, що ми визначили подію як подію чистого тунелювання, передавши її останній аргумент як RoutingStrategies.Tunnel. Якщо ми змінимо його на «RoutingStrategies.Tunnel | RoutingStrategies.Bubble» і перезапустимо рішення знову, ми побачимо проходи тунелювання та спалаху (рис. 6.11).

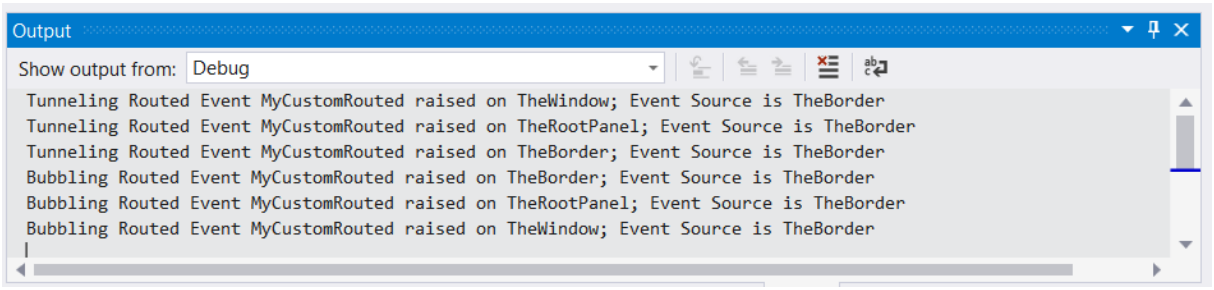


Рисунок 6.11 – Обробка проходження тунелювання та спалаху

6.3 Команди Avalonia

6.3.1 Поняття Команди

Коли хтось створює програму, прийнято розміщувати логіку, яка керує візуальними елементами, у деяких невізуальних класах (які називаються моделями перегляду (View Models)), а потім використовувати прив’язки та інші способи підключення візуальних елементів у XAML до моделей перегляду. Ідея полягає в тому, що невізуальні об’єкти набагато простіші та легші для тестування, ніж візуальні, тому, якщо розробник має справу переважно з невізуальними об’єктами, йому буде легше їх кодувати та тестувати. Такий шаблон називається MVVM.

Команда надає спосіб виконання деяких методів C# у моделі перегляду, коли натискається кнопка або елемент меню.

Кожна з кнопок Avalonia і MenuItem має властивість Command, яку можна прив’язати до команди, визначеної в моделі представлення. Така команда може виконати підключений до неї метод View Model. Avalonia не має власної реалізації команд, але рекомендується використовувати ReactiveUI ReactiveCommand. Можна також контролювати, чи буде кнопка (або MenuItem) увімкнена чи ні, за допомогою командного об’єкта, розміщеного в моделі перегляду.

Проте такий підхід до розміщення команд у моделях перегляду має серйозні недоліки:

– це змушує моделі перегляду залежати від візуальних збірок .NET (які реалізують команди). Це зламує жорсткий бар’єр, який ставиться між невізуальними моделями перегляду та візуальними. Після цього стає набагато

складніше контролювати (особливо в проєктах із багатьма розробниками), щоб візуальний код не «просочувався» у View Models;

– це без потреби забруднює моделі перегляду.

Таким чином, Avalonia забезпечує значно більш чистий спосіб виклику методу в моделі перегляду – прив'язуючи команду до імені методу.

6.3.2 Використання команд Avalonia для виклику методів у моделі перегляду

Створимо новий проєкт CommandTest для вивчення методів використання команд разом із методами в Avalonia. На основному екрані розмістимо дві кнопки, чек-бокс та текстову мітку для відображення поточного статусу. Для розміщення всіх цих компонентів створимо такий XAML код:

```
<Window x:Name="TheWindow"
        xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        x:Class="CommandTest.MainWindow"
        Title="CommandTest"
        Width="200"
        Height="300">

      <Grid x:Name="TheRootPanel"
         RowDefinitions="*, *, *, *"
         Margin="20">
         <CheckBox IsChecked="{Binding Path=CanToggleStatus, Mode=TwoWay}"
              Content="Can Toggle Status"
              HorizontalAlignment="Left"
              VerticalAlignment="Center"/>

         <TextBlock
              Text="{Binding Path=Status, StringFormat='Status={0}'}"
              Grid.Row="1"
              HorizontalAlignment="Left"
              VerticalAlignment="Center"/>

         <Button  Content="Toggle Status"
              Grid.Row="2"
              HorizontalAlignment="Right"
              VerticalAlignment="Center"
              IsEnabled="{Binding Path=CanToggleStatus}"
```

```
            Command="{Binding Path=ToggleStatus}"/>

        <Button  Content="Set Status to True"
            Grid.Row="3"
            HorizontalAlignment="Right"
            VerticalAlignment="Center"
            Command="{Binding Path=SetStatus}"
            CommandParameter="True"/>
    </Grid>
</Window>
```

Після першого запуску на екрані відобразиться вікно, такого вигляду, як подано на рис. 6.12.

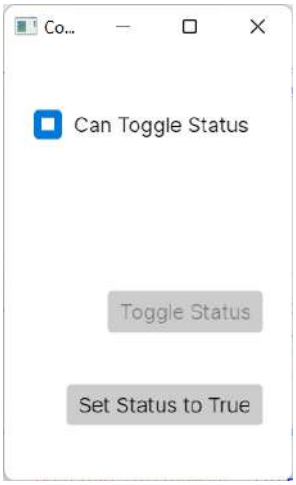


Рисунок 6.12 – Тестове вікно проєкту CommandTest

Поки що кнопки не працюють і тому необхідно створити код C# для виконання потрібних команд. Для цього наповнимо корисним кодом файл ViewModel.cs. Якщо при створенні проєкту його не було створено, то необхідно його додати вручну.

Даний файл містить декілька методів, які необхідні для обробки повідомлень, що виникають під час роботи з командами.

По-перше, необхідно створити об’єкт PropertyChanged, який використовуватиметься кожен раз, коли виникає подія INotifyPropertyChanged:

```
public event PropertyChangedEventHandler? PropertyChanged;
```

Взаємодія з даним об’єктом виконуватиметься із методу OnPropertyChanged:

```
private void OnPropertyChanged(string propName)
{
      PropertyChanged?.Invoke(this,
      new PropertyChangedEventArgs(propName));
}
```

Даний метод викликається кожен раз, коли обробляється та чи інша подія. Наприклад, метод для обробки події, що пов’язана зі зміною статусу:

```
private bool _status;

public bool Status
{
      get
      {
            return this._status;
      }
      set
      {
            if (this._status == value)
            {
                  return;
            }
            this._status = value;
            this.OnPropertyChanged(nameof(Status));
      }
}
```

Інший метод пов’язаний з обробкою стану Toggle відповідної кнопки із інтерфейсу користувача:

```
private bool _canToggleStatus = true;

public bool CanToggleStatus
{
      get
      {
```

```
            return this._canToggleStatus;
        }
        set
        {
            if (this._canToggleStatus == value)
            {
                return;
            }
            this._canToggleStatus = value;
            this.OnPropertyChanged(nameof(CanToggleStatus));
        }
    }
```

Наступний метод необхідний для перемикання статусу кнопки кожен раз, коли він викликається:

```
public void ToggleStatus()
{
    Status = !Status;
}
```

Останній метод потрібен для встановлення заданого стану змінній Status:

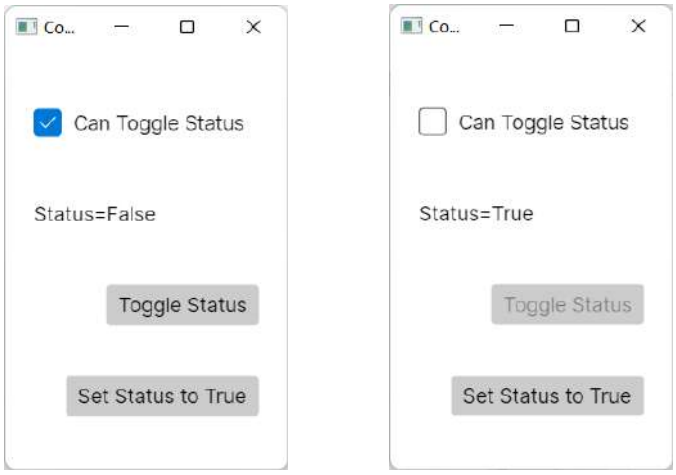
```
public void SetStatus(bool status)
{
    Status = status;
}
```

Повністю код прикладу можна знайти в лістингу «Listing_6-2».

Для того, щоб програма працювала правильно, необхідно пов’язати вихідний код C# та код XAML. Це виконується за допомогою наступного рядка у файлі MainWindow.axaml.cs:

```
public MainWindow()
{
    InitializeComponent();
    this.DataContext = new ViewModel();
}
```

Після запуску програми ми отримуємо інтерфейс, що подано на рис. 6.13.



а) б)

Рисунок 6.13 – Інтерактивний інтерфейс користувача: а) обробка натискання на кнопку «Toggle Status»; б) обробка віджета «Check Box»

Посередині вікна відображається значення поля Status. Коли користувач натискає кнопку «Toggle Status», вона перемикається між True і False. Якщо натиснути «Set Status to True», для значення статусу буде встановлено значення True, а зняття прапорця «Can Toggle Status» призведе до вимкнення кнопки «Toggle Status».

Таким чином ми забезпечили:

– обробку статусу логічної властивості;

– виклик методу ToggleStatus(), який перемикає властивість Status;

– виклик методу SetStatus(bool status), який встановлює для властивості Status будь-який аргумент, який йому було передано;

– реалізацію властивості CanToggleStatus, яка контролює чи ввімкнена дія ToggleStatus(), чи ні.

Щоразу, коли змінюється будь-яка властивість, запускається подія PropertyChanged, тому прив’язки Avalonia отримають сповіщення про зміну властивості.

Конструктор MainWindow, розташований у файлі MainWindow.axaml.cs, встановлює DataContext вікна як екземпляр нашого класу ViewModel:

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

DataContext – це спеціальна властивість StyledProperty, яка успадковується нащадками візуального дерева (якщо не змінено явно), тому вона також буде

доступна для всіх нащадків вікна. Ця особливість покладена в роботу коду XAML.

Прапорець у верхній частині вікна на рис. 6.13 має властивість IsChecked, яка має двосторонню прив’язку до CanToggleStatus ViewModel:

```
<CheckBox IsChecked="{Binding Path=CanToggleStatus, Mode=TwoWay}"
          Content="Can Toggle Status"
          .../>
```

Верхня кнопка через її команду викликає метод ToggleStatus() із ViewModel, а її властивість IsEnabled прив’язана до властивості CanToggleStatus у ViewModel:

```
<Button  Content="Toggle Status"
         ...
         IsEnabled="{Binding Path=CanToggleStatus}"
         Command="{Binding Path=ToggleStatus}"/>
```

Нижня кнопка призначена для демонстрації виклику методу з аргументом у моделі представлення:

```
<Button  Content="Set Status to True"
         Grid.Row="3"
         HorizontalAlignment="Right"
         VerticalAlignment="Center"
         Command="{Binding Path=SetStatus}"
         CommandParameter="True"/>
```

Її властивість Command прив’язана до методу SetStatus(bool status), який має один логічний аргумент – статус. Щоб передати цей аргумент, ми встановлюємо для властивості CommandParameter значення «True».

Повний код прикладу наведено у лістингу «Listing_6-3».

6.3.3 Елементи керування, створені користувачами (User Controls) Avalonia

Елементи керування створені користувачами (User Controls) не рекомендують використовувати, оскільки для елементів керування

рекомендується використовувати Custom Control (також називають користувацькими), що є потужнішими та мають кращу роздільність між візуальними та невізуальними проблемами, а для Views шаблону MVVM – кращі шаблони даних.

Але є задачі, в яких необхідно застосовувати саме цю технологію. Тому розглянемо правила використання таких елементів керування на прикладі нового проєкту UserControlSample. В новому проєкті створюємо новий файл User Controls (рис. 6.14).

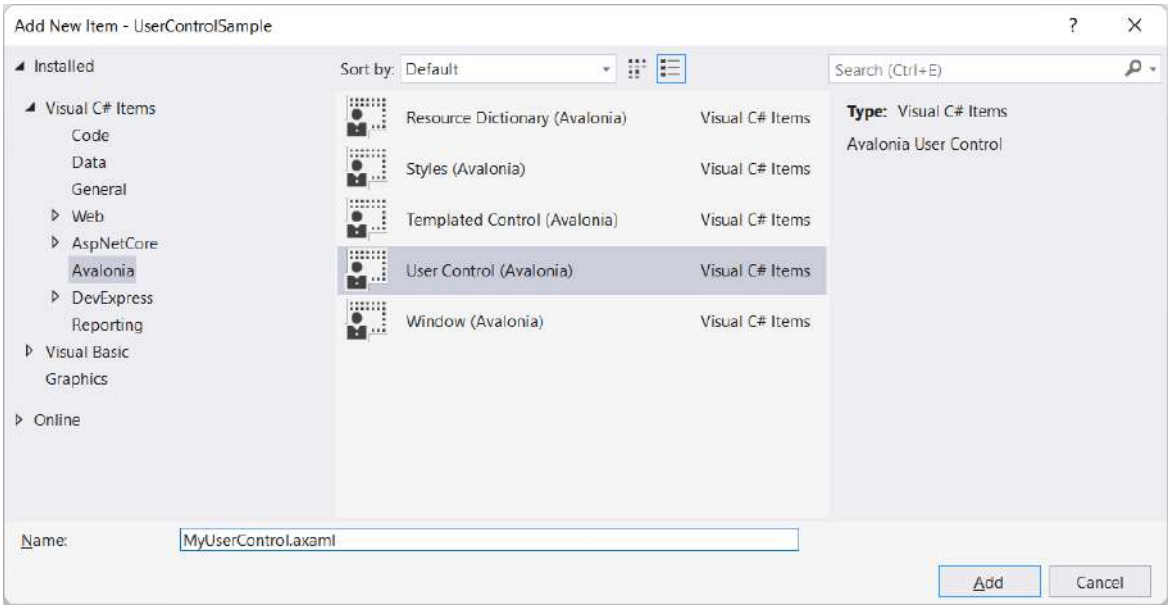


Рисунок 6.14 – Створення елементів користувача

Файл розмітки XAML має наступний вигляд:

```
<Grid RowDefinitions="Auto, Auto, *, Auto">
    <StackPanel Orientation="Horizontal"
                HorizontalAlignment="Left"
                VerticalAlignment="Center">
      <TextBlock Text="Enter Text: "
                 VerticalAlignment="Center"/>
      <TextBox x:Name="TheTextBox"
                 MinWidth="150"/>
    </StackPanel>
    <StackPanel Orientation="Horizontal"
                HorizontalAlignment="Left"
                VerticalAlignment="Center"
                Grid.Row="1"
```

```
                Margin="0,10">
      <TextBlock Text="Saved Text: "
                 VerticalAlignment="Center"/>
      <TextBlock x:Name="SavedTextBlock"/>
    </StackPanel>
    <StackPanel Orientation="Horizontal"
                HorizontalAlignment="Right"
                Grid.Row="3">
      <Button x:Name="CancelButton"
              Content="Cancel"
              Margin="5,0"/>
      <Button x:Name="SaveButton"
              Content="Save"
              Margin="5,0"/>
    </StackPanel>
  </Grid>
```

Розмітка не має ні прив’язок, ні команд – це просто пасивне розташування різних візуальних елементів.

На рис. 6.15 подано результат розмітки User Controls.

Рисунок 6.15 – Інтерфейс User Controls

Функціональність, завдяки якій все це працює, знаходиться у файлі коду MyUserControl.axaml.cs. В конструкторі даного класу реалізовані методи пошуку та підключення подій до візуальних компонентів:

```
public MyUserControl()
{
      InitializeComponent();

        // set _cancelButton and its Click event handler
        _cancelButton = this.FindControl<Button>("CancelButton");
        _cancelButton.Click += OnCancelButtonClick;
```

```
        // set _saveButton and its Click event handler
        _saveButton = this.FindControl<Button>("SaveButton");
        _saveButton.Click += OnSaveButtonClick;

        // set the TextBlock that contains the Saved text
        _savedTextBlock = this.FindControl<TextBlock>("SavedTextBlock");

        // set the TextBox that contains the new text
        _textBox = this.FindControl<TextBox>("TheTextBox");

        // initial New and Saved values should be the same
        NewValue = SavedValue;

        // every time the text changes, we should check if
        // Save and Cancel buttons should be enabled or not
        _textBox.GetObservable(TextBox.TextProperty).
            Subscribe(OnTextChanged);
    }
```

Посилання на візуальні елементи, визначені у файлі MyUserControl.xaml, отримуються всередині коду C# за допомогою методу FindControlTElement(«ElementName»), наприклад:

```
// set _cancelButton and its Click event handler
_cancelButton = this.FindControl<Button>("CancelButton");
```

Для події Click кнопки призначається обробник, наприклад:

```
_cancelButton.Click += OnCancelButtonClick;
```

Обробка тексту виконується на основі підписки на зміну тексту TextBox’es Text:

```
// every time the text changes, we should check if
// Save and Cancel buttons should be enabled or not
_textBox.GetObservable(TextBox.TextProperty).Subscribe(OnTextChanged);
```

Попередньо в шапці даного класу об’явлені всі потрібні об’єкти:

```
public partial class MyUserControl : UserControl
{
    private TextBox _textBox;
    private TextBlock _savedTextBlock;
    private Button _cancelButton;
    private Button _saveButton;
      ...
}
```

Для зберігання тексту, що було введено в візуальному компоненті TextBlock передбачена така функція:

```
private string? SavedValue
{
    get => _savedTextBlock.Text;
    set => _savedTextBlock.Text = value;
}
```

Компонент textBox отримує текстові дані через даний метод:

```
private string? NewValue
{
    get => _textBox.Text;
    set => _textBox.Text = value;
}
```

Далі показані приклади реалізації методів для обробки подій натискання на кнопки Cancel, Save, а також події, що виникає при введені тексту в компонент TexBlock:

```
// On Cancel, the TextBox value should become the same as SavedValue
private void OnCancelButtonClick(object? sender, RoutedEventArgs e)
{
    NewValue = SavedValue;
}

// On Save, the Saved Value should become the same as the TextBox Value
private void OnSaveButtonClick(object? sender, RoutedEventArgs e)
{
    SavedValue = NewValue;
```

```
        // also we should reset the IsEnabled states of the buttons
        OnTextChanged(null);
    }

    private void OnTextChanged(string? obj)
    {
        bool canSave = NewValue != SavedValue;
        // _cancelButton as _saveButton are enabled if TextBox'es value
        // is not the same as saved value and disabled otherwise.
        _cancelButton.IsEnabled = canSave;
        _saveButton.IsEnabled = canSave;
    }
```

Підключення компонентів користувача до основного коду виконується дуже легко. В даному прикладі це робиться за допомогою наступного рядка:

```
<local:MyUserControl Margin="20"/>
```

В результаті компіляції та запуску даного проєкту отримаємо такий зовнішній вигляд інтерфейсу користувача (рис. 6.16).

Рисунок 6.16 – Інтерфейс користувача
із використанням UserControl

Повністю код проєкту можна побачити в лістингу «Listing_6-4».

Основна проблема з елементом керування користувачами полягає в тому, що ми тісно пов'язуємо візуальне представлення, надане файлом MyUserControl.axaml, і логіку C#, що міститься у файлі MyUserControl.axaml.cs.

Використовуючи Custom Control, ми можемо повністю розділити їх, як буде показано нижче. Крім того, візуальне представлення можна відокремити від логіки C# за допомогою частини View-ViewModel шаблону MVVM, щоб можна було використовувати абсолютно різні візуальні представлення (надані різними шаблонами даних) з тією ж моделлю представлення, яка визначає бізнес-логіку. Такий приклад MVVM буде наведено нижче.

6.3.4 Avalonia ControlTemplates та Custom Control Templates

Виконаємо модифікацію попереднього прикладу. Створимо новий проєкт та перенесемо в нього той самий функціонал, але побудуємо додаток зовсім інакше. Уся функціональність C#, яка не є стандартною, буде знаходитися у файлі MyCustomControl.cs:

```
public class MyCustomControl : TemplatedControl
{
    public string? NewValue
    {
        get { return GetValue(NewValueProperty); }
        set { SetValue(NewValueProperty, value); }
    }

    public static readonly StyledProperty<string?> NewValueProperty =
        AvaloniaProperty.Register<MyCustomControl, string?>
        (
            nameof(NewValue)
        );

    public string? SavedValue
    {
        get { return GetValue(SavedValueProperty); }
        set { SetValue(SavedValueProperty, value); }
    }

    public static readonly StyledProperty<string?> SavedValueProperty =
        AvaloniaProperty.Register<MyCustomControl, string?>
        (
            nameof(SavedValue)
        );

    private bool _canSave = default;
```

```
        public static readonly DirectProperty<MyCustomControl, bool>
CanSaveProperty = AvaloniaProperty.RegisterDirect<MyCustomControl, bool>
            (
                nameof(CanSave),
                o => o.CanSave
            );

        public bool CanSave
        {
            get => _canSave;
            private set {
                SetAndRaise(CanSaveProperty, ref _canSave, value); }
        }

        private void SetCanSave(object? _)
        {
            CanSave = SavedValue != NewValue;
        }

        public MyCustomControl()
        {
            this.GetObservable(NewValueProperty).Subscribe(SetCanSave);
            this.GetObservable(SavedValueProperty).Subscribe(SetCanSave);
        }

        public void Save()
        {
            SavedValue = NewValue;
        }

        public void Cancel()
        {
            NewValue = SavedValue;
        }
    }
```

Більшість коду в даному прикладі присутня через визначення StyledProperty і DirectProperty.

Є дві властивості стилю: NewValue і SavedValue і одна пряма властивість: CanSave. Кожного разу, коли змінюється будь-яка зі стилізованих властивостей, пряма властивість переоцінюється як хибна тоді і тільки тоді, коли NewValue ==

SavedValue. Це досягається шляхом підписки на зміни NewValue та SavedValue у конструкторі класу:

```
public MyCustomControl()
{
    this.GetObservable(NewValueProperty).Subscribe(SetCanSave);
    this.GetObservable(SavedValueProperty).Subscribe(SetCanSave);
}
```

і встановивши його в методі зворотного виклику SetCanSave(...):

```
// CanSave is set to true when SavedValue is not the same as NewView
// false otherwise
private void SetCanSave(object? _)
{
    CanSave = SavedValue != NewValue;
}
```

Непотрібний аргумент «_» цьому методу передається для того, щоб його підпис відповідав тому, який вимагає метод Subscribe(...).

Також є два загальнодоступні методи, які можуть бути викликані командами кнопок: void Save() і void Cancel():

```
public void Save()
{
    SavedValue = NewValue;
}

public void Cancel()
{
    NewValue = SavedValue;
}
```

Різниця між цим файлом C# і файлом MyUserControl.axaml.cs (який ми описували в попередньому розділі) полягає в тому, що цей файл повністю не знає про реалізацію XAML і не має жодних посилань на елементи XAML.

Натомість XAML, створений як ControlTemplate у файлі MainWindow.axaml, посилається на властивості та методи, визначені у файлі MyCustomControl.cs за допомогою прив’язок і команд.

Перш за все, зверніть увагу, що ми отримали наш клас MyCustomControl від TemplatedControl:

```
public class MyCustomControl : TemplatedControl
{
    ...
}
```

Через це він має властивість Template типу ControlTemplate, яку ми можемо встановити для будь-якого об’єкта цього типу. Ось відповідний код XAML, розташований у файлі MainWindow.axaml:

```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        Width="300"
        Height="300"
        xmlns:local="clr-namespace:CustomControlSample"
        x:Class="CustomControlSample.MainWindow"
        Title="CustomControlSample">
   <local:MyCustomControl Margin="20">
      <local:MyCustomControl.Template>
         <ControlTemplate TargetType="local:MyCustomControl">
            <Grid RowDefinitions="Auto, Auto, *, Auto">
               <StackPanel Orientation="Horizontal"
                  HorizontalAlignment="Left"
                  VerticalAlignment="Center">
               <TextBlock Text="Enter Text: "
                  VerticalAlignment="Center"/>
               <TextBox x:Name="TheTextBox"
                  Text="{Binding Path=NewValue, Mode=TwoWay,
                     RelativeSource={RelativeSource TemplatedParent}}"
                     MinWidth="150"/>
               </StackPanel>
               <StackPanel Orientation="Horizontal"
                  HorizontalAlignment="Left"
                  VerticalAlignment="Center"
                  Grid.Row="1"
                  Margin="0,10">
               <TextBlock Text="Saved Text: "
                  VerticalAlignment="Center"/>
```

```
                    <TextBlock x:Name="SavedTextBlock"
                       Text="{TemplateBinding SavedValue}"/>
                 </StackPanel>
                 <StackPanel Orientation="Horizontal"
                    HorizontalAlignment="Right"
                    Grid.Row="3">
                    <Button x:Name="CancelButton"
                       Content="Cancel"
                       Margin="5,0"
                       IsEnabled="{TemplateBinding CanSave}"
                       Command="{Binding Path=Cancel,
                       RelativeSource={RelativeSource TemplatedParent}}"/>
                    <Button x:Name="SaveButton"
                       Content="Save"
                       Margin="5,0"
                       IsEnabled="{TemplateBinding CanSave}"
                       Command="{Binding Path=Save,
                       RelativeSource={RelativeSource TemplatedParent}}"/>
                 </StackPanel>
              </Grid>
           </ControlTemplate>
        </local:MyCustomControl.Template>
     </local:MyCustomControl>
  </Window>
```

Ми встановлюємо властивість Template для об’єкта ControlTemplate за допомогою таких рядків:

```
<local:MyCustomControl Margin="20">
  <local:MyCustomControl.Template>
    <ControlTemplate TargetType="local:MyCustomControl">
    ...
```

Необхідно зауважити, що ми заповнюємо властивість Template у рядку – це добре для створення прототипів, але погано для повторного використання. Зазвичай шаблон керування створюється як ресурс у якомусь файлі ресурсу, а потім ми використовуємо розширення розмітки {StaticResource <ResourceKey>} для встановлення властивості Template. Отже, рядки вище виглядатимуть так:

```
<local:MyCustomControl Margin="20"
       Template="{StaticResource MyCustomControlTemplate}">
```

Таким чином, ми зможемо повторно використовувати той самий шаблон для кількох елементів керування. Крім того, ми можемо розмістити шаблони елементів керування зі стилями та використовувати стилі для наших спеціальних елементів керування.

Зверніть увагу, як ми вказуємо TargetType ControlTemplate:

```
<ControlTemplate TargetType="local:MyCustomControl">
```

Це дозволить нам підключитися до властивостей, визначених класом MyCustomControl, за допомогою TemplateBinding або {RelativeSource TemplatedParent}.

TextBox прив’язаний до властивості NewValue елемента керування в режимі TwoWay, тому зміни одного вплинуть на інший:

```
<TextBox x:Name="TheTextBox"
         Text="{Binding Path=NewValue, Mode=TwoWay,
         RelativeSource={RelativeSource TemplatedParent}}"
         MinWidth="150"/>
```

TextBlock «SavedTextBlock» прив’язаний до SavedValue:

```
<TextBlock x:Name="SavedTextBlock"
           Text="{TemplateBinding SavedValue}"/>
```

Команди кнопок прив’язані до відповідних відкритих методів: Cancel() і Save(), а властивість кнопок IsEnabled прив’язана до властивості CanSave елемента керування:

```
<Button x:Name="CancelButton"
        Content="Cancel"
        Margin="5,0"
        IsEnabled="{TemplateBinding CanSave}"
        Command="{Binding   Path=Cancel,   RelativeSource={RelativeSource
TemplatedParent}}"/>
<Button x:Name="SaveButton"
```

```
            Content="Save"
            Margin="5,0"
            IsEnabled="{TemplateBinding CanSave}"
            Command="{Binding    Path=Save,    RelativeSource={RelativeSource
TemplatedParent}}"/>
```

6.4 Шаблони даних і моделі відображення

6.4.1 Вступ до концепції відображення та моделі відображення

Як було зазначено в перших розділах даного навчального посібника, MVVM – це абревіатура від шаблону Model-View-View-Model (Модель-Відображення-Відображення-Модель).

Відображення (View) – це візуальні елементи, які визначають зовнішній вигляд, відчуття та візуальну поведінку програми.

Модель відображення (View Model) – це повністю невізуальний клас або набір класів, який виконує дві основні ролі:

– вона забезпечує певну функціональність, яку відображення може імітувати або викликати за допомогою прив’язок, команд або інших засобів, наприклад, поведінки. Наприклад, модель відображення (View Model) може мати метод void SaveAction() і властивість IsSaveActionAllowed, тоді як просто відображення (View) матиме кнопку, яка викликає метод SaveAction(), властивість IsEnabled якого буде прив’язана до властивості IsSaveActionAllowed у моделі перегляду;

– вона обгортає модель (наприклад, дані, які надходять із серверної частини), надає сповіщення представленню, якщо модель змінена, і навпаки, а також може надавати функціональні можливості зв’язку між різними моделями відображення та моделями даних.

Шаблон VVM найкраще реалізовувати в Avalonia за допомогою ContentPresenter (для одного об’єкта) або ItemsPresenter (для колекції об’єктів) (рис. 6.17).

ContentControl – це елемент керування вмістом, наприклад TextBox, Button, TextBlack та інші елементи керування.

Від ItemsControl наслідуються такі елементи управління, як List, TabControl тощо.

Слід звертати увагу на різницю між ContentControl та ItemsControl в процесі написання шаблонів елементів керування.

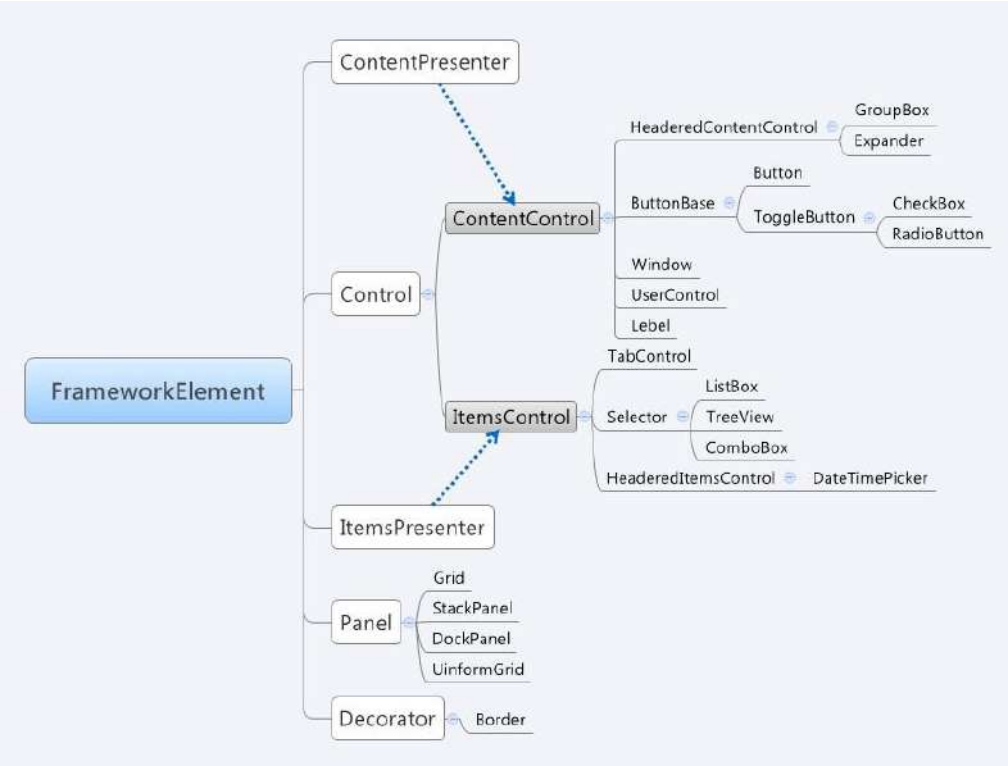


Рисунок 6.17 – Шаблон елементів керування

Як видно з рисунка вище, Control не може відображати текстовий зміст прямо – його необхідно відображати через ContentControl і ContentPresenter. З поданої вище структури ContentPresenter використовується для відображення свого текстового вмісту, а ContentControl використовується як контейнер для своїх внутрішніх вкладених елементів керування.

ContentPresenter за допомогою DataTemplate перетворює невізуальний об’єкт у візуальний об’єкт (відображення) (рис. 6.18).

Рисунок 6.18 – Властивість ContentPresenter

Властивість ContentPresenter зазвичай встановлюється як невізуальний об’єкт, тоді як ContentTemplate має бути встановлена як DataTemplate. ContentPresenter об’єднує їх у візуальний об’єкт (View), де DataContext надається властивістю ContentPresenter Content, тоді як візуальне дерево надається DataTemplate.

ItemsPresenter за допомогою DataTemplate перетворює колекцію невізуальних об’єктів у колекцію візуальних об’єктів, кожен із яких містить ContentPresenter, який перетворює окремий елемент моделі View колекції у візуальний об’єкт (рис. 6.19). Візуальні об’єкти впорядковані відповідно до панелі, наданої значенням властивості ItemsPresenter.ItemsPanel.

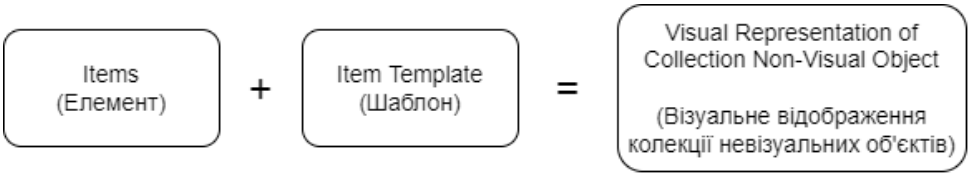


Рисунок 6.19 – Властивість ItemsPresenter

Властивість Items ItemsPresenter може містити колекцію невізуальних об’єктів. ItemTemplate містить об’єкт DataTemplate, а ItemsPresenter об’єднує їх у колекцію візуальних об’єктів.

6.4.2 Приклад реалізації ContentPresenter

Розглянемо реалізацію ContentPresenter на відповідному прикладі, створивши однойменне рішення в Visual Studio. На рис. 6.20 подано вікно реєстрації користувача з трьома полями вводу, текстовою міткою та кнопками керування.

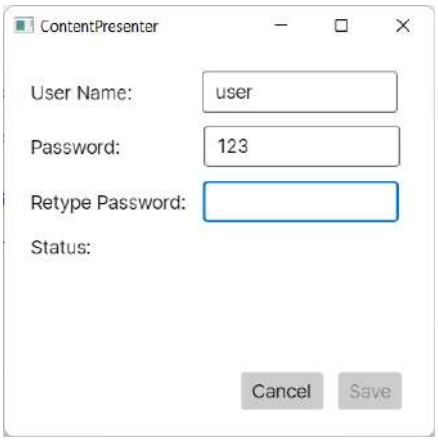


Рисунок 6.20 – Тестове вікно реєстрації користувача

Логіка роботи програми наступна:

– користувач, після заповнення «User Name» вводить пароль в поле «Password»;

– в полі «Retype Password» користувач повинен повторити введення також самого паролю;

– якщо паролі співпадають, засвітиться кнопка «Save»;

– після натискання на кнопку «Save» в полі статус висвітиться статус «Success!»;

– кнопка «Cancel» очищає всі поля введення та поле статусу.

На відміну від попередніх випадків, ми не створюємо для цього ані користувацькі, ані спеціальні елементи керування. Замість цього ми використовуємо повністю невізуальну модель представлення та шаблон даних, які стикуються в ContentPresenter.

Для розташування компонентів використовується наступний код XAML:

```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Width="330"
        Height="300"
        xmlns:local="clr-namespace:ContentPresenter"
        x:Class="ContentPresenter.MainWindow"
        Title="ContentPresenter">
   <Window.Resources>
      <local:ViewModel x:Key="TheViewModel"/>
         <DataTemplate x:Key="TheDataTemplate">
            <Grid RowDefinitions="Auto, Auto, Auto, Auto, *, Auto">
               <StackPanel Orientation="Horizontal"
                  HorizontalAlignment="Left"
                  VerticalAlignment="Center">
                  <TextBlock Text="User Name: "
                     VerticalAlignment="Center"/>
                  <TextBox x:Name="TheLogin"
                     Margin="50, 0"
                     Text="{Binding Path=LoginValue, Mode=TwoWay}"
                     MinWidth="150"/>
               </StackPanel>
               <StackPanel Orientation="Horizontal"
                  Grid.Row="1"
                  Margin="0,10"
                  HorizontalAlignment="Left"
                  VerticalAlignment="Center">
                  <TextBlock Text="Password:"
                     VerticalAlignment="Center"/>
```

```
            <TextBox x:Name="ThePassword"
               Margin="64, 0"
               Text="{Binding Path=PassValue, Mode=TwoWay}"
               MinWidth="150"/>
         </StackPanel>
         <StackPanel Orientation="Horizontal"
            Grid.Row="2"
            HorizontalAlignment="Left"
            VerticalAlignment="Center">
            <TextBlock Text="Retype Password: "
               VerticalAlignment="Center"/>
            <TextBox x:Name="TheRetypePassword"
               Margin="9, 0"
               Text="{Binding Path=RePassValue, Mode=TwoWay}"
               MinWidth="150"/>
         </StackPanel>
         <StackPanel Orientation="Horizontal"
            HorizontalAlignment="Left"
            VerticalAlignment="Center"
            Grid.Row="3"
            Margin="0,10">
            <TextBlock Text="Status: "
               VerticalAlignment="Center"/>
            <TextBlock x:Name="StatusTextBlock"
               Text="{Binding Path=StatusValue}"/>
         </StackPanel>
         <StackPanel Orientation="Horizontal"
            HorizontalAlignment="Right"
            Grid.Row="5">
            <Button x:Name="CancelButton"
               Content="Cancel"
               Margin="5,0"
               Command="{Binding Path=Cancel}"/>
            <Button x:Name="SaveButton"
               Content="Save"
               Margin="5,0"
               IsEnabled="{Binding Path=CanEnter}"
               Command="{Binding Path=Enter}"/>
         </StackPanel>
      </Grid>
   </DataTemplate>
</Window.Resources>
<ContentPresenter Margin="20"
```

```
        Content="{StaticResource TheViewModel}"
        ContentTemplate="{StaticResource TheDataTemplate}"/>
</Window>
```

Наприкінці файлу MainWindow.axaml визначаються екземпляри ViewModel і DataTemplate як ресурси та ContentPresenter, який поєднує їх:

```
<Window ...>
    <Window.Resources>
        ...
    </Window.Resources>
    <ContentPresenter Margin="20"
            Content="{StaticResource TheViewModel}"
            ContentTemplate="{StaticResource TheDataTemplate}"/>
</Window>
```

Визначення DataTemplate представлено наступним чином:

```
<Window ...>
   <Window.Resources>
      <local:ViewModel x:Key="TheViewModel"/>
         <DataTemplate x:Key="TheDataTemplate">
            <Grid RowDefinitions="Auto, Auto, *, Auto">
                  . . .
                  Визначення DataTemplate
                  . . .
             </Grid>
         </DataTemplate>
      </Window.Resources>
   <ContentPresenter Margin="20"
         Content="{StaticResource TheViewModel}"
         ContentTemplate="{StaticResource TheDataTemplate}"/>
</Window>
```

Реалізація моделі представлення (view model) та шаблону даних (data template) призначаються властивостям Content і ContentTemplate ContentPresenter за допомогою розширення розмітки StaticResource.

Далі наведено приклад, як ми визначаємо екземпляр ViewModel в якості ресурсу Window:

```
<Window ...>
    <Window.Resources>
        <local:ViewModel x:Key="TheViewModel"/>
        ...
    </Window.Resources>
...
</Window>
```

Необхідно зауважити, що об’єкт ViewModel, наданий як властивість Content ContentPresenter, стає DataContext для візуальних елементів, створених в DataTemplate, тому ми можемо прив’язати властивості DataTemplate до властивостей моделі View, не вказуючи вихідний об’єкт зв’язування (оскільки DataContext є джерелом зв’язування за замовчуванням).

Ми прив’язуємо TextBox до властивостей TheLogin, ThePassword та TheRetypePassword ViewModel у режимі TwoWay, щоб у разі зміни одного з них змінився інший. Всі три компоненти реалізовані однаково, тому розглянемо це на прикладі вводу логіну:

```
<TextBox x:Name="TheLogin"
      Margin="50, 0"
      Text="{Binding Path=LoginValue, Mode=TwoWay}"
MinWidth="150"/>
```

Прив’язуємо властивість Text компоненти «StatusTextBlock» до StatusValue:

```
<TextBlock x:Name="StatusTextBlock"
      Text="{Binding Path=StatusValue}"/>
```

Далі ми прив’язуємо команди кнопок до методів Save() і Cancel(), а також прив’язуємо властивість кнопок IsEnabled кнопки SaveButton до властивості Boolean CanEnter ViewModel:

```
<Button x:Name="CancelButton"
      Content="Cancel"
      Margin="5,0"
      Command="{Binding Path=Cancel}"/>
<Button x:Name="SaveButton"
```

```
        Content="Save"
        Margin="5,0"
        IsEnabled="{Binding Path=CanEnter}"
        Command="{Binding Path=Enter}"/>
```

Звичайно, ми можемо перетягнути DataTemplate в інший файл і, навіть, в інший проєкт, і повторно використовувати його в багатьох місцях. Реалізація методів в файлі наступна:

```
public class ViewModel : INotifyPropertyChanged
{
        public event PropertyChangedEventHandler? PropertyChanged;

        private void OnPropertyChanged(string propName)
        {
              PropertyChanged?.Invoke(this,
              new PropertyChangedEventArgs(propName));
        }

        private string? _statusValue;
        public string? StatusValue
        {
              get {
                    return this._statusValue; }
              private set {
                  if (this._statusValue == value) {
                      return; }
                  this._statusValue = value;
                  this.OnPropertyChanged(nameof(StatusValue));
                  this.OnPropertyChanged(nameof(CanEnter));
              }
          }

          private string? _loginValue;
          public string? LoginValue
          {
              get {
                  return this._loginValue; }
              set {
                  if (this._loginValue == value) {
                      return; }
                  this._loginValue = value;
```

```
        this.OnPropertyChanged(nameof(LoginValue));
        this.OnPropertyChanged(nameof(CanEnter));
    }
}

private string? _passValue;
public string? PassValue
{
    get {
        return this._passValue; }
    set {
        if (this._passValue == value) {
            return; }
        this._passValue = value;
        this.OnPropertyChanged(nameof(PassValue));
        this.OnPropertyChanged(nameof(CanEnter));
    }
}

private string? _rePassValue;
public string? RePassValue
{
    get {
        return this._rePassValue; }
    set {
        if (this._rePassValue == value){
            return; }
        this._rePassValue = value;
        this.OnPropertyChanged(nameof(RePassValue));
        this.OnPropertyChanged(nameof(CanEnter));
    }
}

public bool CanEnter => (PassValue == RePassValue);
public void Enter()
{
    if ((PassValue == RePassValue) && (LoginValue != "")){
        StatusValue = "Success!";
    }
}

public void Cancel()
{
```

```
            LoginValue = "";
            PassValue = "";
            RePassValue = "";
            StatusValue = "";
        }
    }
```

У нас є рядкові властивості StatusValue, LoginValue, PassValue і RePassValue, які запускають подію повідомлення PropertyChanged, коли будь-яка з них змінюється:

```
private string? _loginValue;
public string? LoginValue
{
      get {
            return this._loginValue; }
      set {
      if (this._loginValue == value) {
            return; }
      this._loginValue = value;
      this.OnPropertyChanged(nameof(LoginValue));
      this.OnPropertyChanged(nameof(CanEnter));
      }
}
```

Вони також повідомляють про можливу зміну булевої властивості CanEnter, яка є істинною тоді, коли PassValue та RePassValue збігаються:

```
public bool CanEnter => (PassValue == RePassValue);
```

Також є два загальнодоступні методи введення та скасування:

```
public void Enter()
{
      if ((PassValue == RePassValue) && (LoginValue != "")){
      StatusValue = "Success!";
      }
}

public void Cancel()
```

```
{
        LoginValue = "";
        PassValue = "";
        RePassValue = "";
        StatusValue = "";
}
```

Дані методи прив’язані до відповідних кнопок «CancelButton», «SaveButton».

Повний код прикладу знаходиться в папці ContentPresenter лістингу «Listing_6-5».

6.4.3 Приклад реалізації ItemsPresenter

Для дослідження методів реалізації ItemsPresenter розробимо новий проєкт в Visual Studio. Цей приклад описує, як використовувати ItemsPresenter для відображення колекції невізуальних об’єктів.

ItemsPresenter використовується в шаблоні елемента керування для визначення місця у візуальному дереві, куди має бути додана панель ItemsPanel, що визначена елементом DataTemplate.

Вигляд основного вікна програми подано на рис. 6.21.

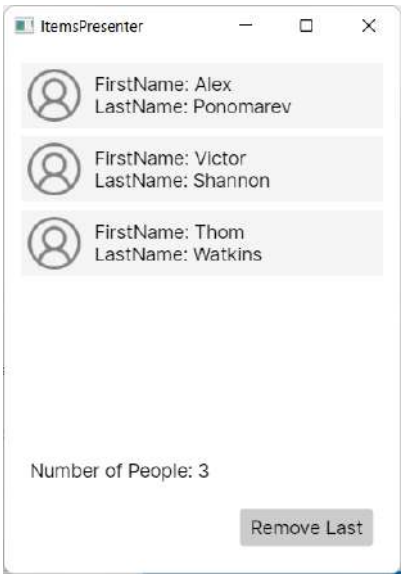


Рисунок 6.21 – Вигляд основного вікна програми

Інтерфейс складається із списку персон, текстової мітки для відображення кількості персон, та кнопки видалення запису із списку. Кожна опція списку

являє собою структуру компонентів, що складається із зображення, Ім’я та Прізвища. Така структура повторюється стільки разів, скільки записів буде додано до списку персон. Для створення структури елемента списку використовується такий код XAML:

```
<Border Background="WhiteSmoke"
      Margin="3">
      <Grid ColumnDefinitions="50, 250">
            <Image Source="/Assets/Codicons-Account.png"
                  Grid.Column="0"
                  Width="40"
                  Height="40"
                  Margin="5"/>
            <Grid Grid.Column="1"
                  Margin="5,0"
                  HorizontalAlignment="Stretch"
                  VerticalAlignment="Center"
                  RowDefinitions="Auto, Auto">
                  <TextBlock Grid.Row="0"
                        Text="{Binding Path=FirstName,
                        StringFormat=‘FirstName: {0}’}"/>
                  <TextBlock Grid.Row="1"
                        Text="{Binding Path=LastName,
                        StringFormat=‘LastName: {0}’}"/>
            </Grid>
      </Grid>
</Border>
```

Така структура кода XAML призведе до зображення на екрані одного інформаційного блока, що підготовлено для виведення інформації зі списку персон (рис. 6.22).

Рисунок 6.22 – Інформаційний блок для виведення інформації зі списку персон

На даний момент текст не відображається тому, що немає підключення до самого списку.

PersonViewModel – це найпростіший клас, що містить властивості FirstName та LastName. Структура даного класу показана нижче:

```
public class PersonViewModel
{
    public string FirstName { get; }

    public string LastName { get; }

    public PersonViewModel(string firstName, string lastName)
    {
        FirstName = firstName;
        LastName = lastName;
    }
}
```

ItemViewModel представляє модель відображення верхнього рівня, що містить колекцію об’єктів PersonViewModel у своїй властивості People типу ObservableCollectionPersonViewModel:

```
public class TestViewModel : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler? PropertyChanged;

    private void OnPropertyChanged(string propName)
    {
        PropertyChanged?.Invoke(this,
            new PropertyChangedEventArgs(propName));
    }

    public ObservableCollection<PersonViewModel> People { get; } =
        new ObservableCollection<PersonViewModel>();

    public int NumberOfPeople => People.Count;

    public ItemViewModel()
    {
        People.CollectionChanged += People_CollectionChanged;
```

```
            People.Add(new PersonViewModel("Alex", "Ponomarev"));
            People.Add(new PersonViewModel("Victor", "Shannon"));
            People.Add(new PersonViewModel("Thom", "Watkins"));
        }

        private void People_CollectionChanged(object? sender,
NotifyCollectionChangedEventArgs e)
        {
            OnPropertyChanged(nameof(NumberOfPeople));
            OnPropertyChanged(nameof(CanRemoveLast));
        }

        public bool CanRemoveLast => NumberOfPeople > 0;

        public void RemoveLast()
        {
            People.RemoveAt(NumberOfPeople - 1);
        }
    }
```

В даному класі в конструкторі створюється список із трьох персон, додаючи їх до об’єкту People:

```
        public ItemViewModel()
        {
            People.CollectionChanged += People_CollectionChanged;

            People.Add(new PersonViewModel("Alex", "Ponomarev"));
            People.Add(new PersonViewModel("Victor", "Shannon"));
            People.Add(new PersonViewModel("Thom", "Watkins"));
        }
```

Властивість NumberOfPeople містить поточну кількість елементів в колекції People, а властивість CanRemoveLast вказує, чи є якісь елементи в колекції:

```
    // number of people
    public int NumberOfPeople => People.Count;
    ...
    // can remove last item only if collection has some items in it
    public bool CanRemoveLast => NumberOfPeople > 0;
```

Щоразу, коли колекція People змінюється, ми сповіщаємо прив'язку, що ці дві властивості могли бути оновлені:

```
public ItemViewModel()
{
    People.CollectionChanged += People_CollectionChanged;
    ...
}

// whenever collection changes, fire notification for possible updates
// of NumberOfPeople and CanRemoveLast properties.
private void People_CollectionChanged(object? sender,
NotifyCollectionChangedEventArgs e)
{
    OnPropertyChanged(nameof(NumberOfPeople));
    OnPropertyChanged(nameof(CanRemoveLast));
}
```

Також використовується метод RemoveLast() для видалення останнього елемента в колекції People:

```
// remove last item of the collection
public void RemoveLast()
{
    People.RemoveAt(NumberOfPeople - 1);
}
```

Файл MainWindow.axaml містить код XAML для відображення програми:

```
<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:local="clr-namespace:ItemsPresenter"
        Width="300"
        Height="400"
        x:Class="ItemsPresenter.MainWindow"
        Title="ItemsPresenter">
      <Window.Resources>
         <local:ItemViewModel x:Key="TheViewModel"/>
            <DataTemplate x:Key="PersonDataTemplate">
               <Border Background="WhiteSmoke"
```

```
            Margin="3">
            <Grid ColumnDefinitions="50, 250">
               <Image Source="/Assets/Codicons-Account.png"
                  Grid.Column="0"
                  Width="40"
                  Height="40"
                  Margin="5"/>
               <Grid Grid.Column="1"
                  Margin="5,0"
                  HorizontalAlignment="Stretch"
                  VerticalAlignment="Center"
                  RowDefinitions="Auto, Auto">
                  <TextBlock Grid.Row="0"
                     Text="{Binding Path=FirstName,
                     StringFormat='FirstName: {0}'}"/>
                  <TextBlock Grid.Row="1"
                     Text="{Binding Path=LastName,
                     StringFormat='LastName: {0}'}"/>
               </Grid>
            </Grid>
         </Border>
      </DataTemplate>

      <DataTemplate x:Key="TestViewModelDataTemplate">
         <Grid RowDefinitions="*, Auto, Auto">
            <ItemsPresenter Items="{Binding Path=People}"
               ItemTemplate="{StaticResource PersonDataTemplate}">
               <ItemsPresenter.ItemsPanel>
                  <ItemsPanelTemplate>
                     <WrapPanel Orientation="Vertical"/>
                  </ItemsPanelTemplate>
               </ItemsPresenter.ItemsPanel>
            </ItemsPresenter>
            <TextBlock Text="{Binding Path=NumberOfPeople,
               StringFormat='Number of People: {0}'}"
               Grid.Row="1"
               HorizontalAlignment="Left"
               Margin="10"/>
            <Button Content="Remove Last"
               IsEnabled="{Binding Path=CanRemoveLast}"
               Command="{Binding Path=RemoveLast}"
               Grid.Row="2"
               HorizontalAlignment="Right"
```

```
                    Margin="10"/>
            </Grid>
         </DataTemplate>
      </Window.Resources>
      <ContentPresenter Content="{StaticResource TheViewModel}"
         ContentTemplate="{StaticResource TestViewModelDataTemplate}"
         Margin="10"/>
</Window>
```

Примірник моделі представлення визначається у верхній частині розділу ресурсів вікна:

```
<local:TestViewModel x:Key="TheViewModel"/>
```

Існують два шаблони даних, визначені як ресурси XAML Window:

– TestViewModelDataTemplate – шаблон даних для всієї програми. Він побудований на основі класу ItemViewModel і використовує PersonDataTemplate для відображення візуальних елементів, що відповідають кожній людині;

– PersonDataTemplate – відображає імена та прізвища окремого елемента PersonViewModel.

PersonDataTemplate дуже простий в реалізації – всього два TextBlocks для імені та прізвища один над одним:

```
<:Key="PersonDataTemplate">
  <Grid RowDefinitions="Auto, Auto"
        Margin="10">
    <TextBlock Text="{Binding Path=FirstName,
       StringFormat='FirstName: {0}'}"
       Grid.Row="0"/>
    <TextBlock Text="{Binding Path=LastName,
       StringFormat='LastName: {0}'}"
       Grid.Row="1"/>
  </Grid>
</DataTemplate>
```

TestViewModelDataTemplate містить ItemsPresenter (для якого був створений зразок):

```
<ItemsPresenter Items="{Binding Path=People}"
```

```
                ItemTemplate="{StaticResource PersonDataTemplate}">
  <ItemsPresenter.ItemsPanel>
    <ItemsPanelTemplate>
      <WrapPanel Orientation="Horizontal"/>
    </ItemsPanelTemplate>
  </ItemsPresenter.ItemsPanel>
</ItemsPresenter>
```

Його властивість Items прив'язана до колекції People класу ItemViewModel, а його властивість ItemTemplate встановлена як PersonDataTemplate.

Властивість ItemsPanel WrapPanel можна налаштувати на горизонтально орієнтовану панель, щоб продемонструвати можливість змінити спосіб розташування візуальних елементів у ItemsPresenter (за замовчуванням вони розташовуються вертикально).

Наведений код XAML також містить кнопку для видалення останнього елемента. Команда Button прив'язана до методу RemoveLast() моделі перегляду, а її властивість IsEnabled – до властивості CanRemoveLast моделі перегляду:

```
<Button Content="Remove Last"
        IsEnabled="{Binding Path=CanRemoveLast}"
        Command="{Binding Path=RemoveLast}"
        Grid.Row="2"
        HorizontalAlignment="Right"
        Margin="10"/>
```

Нарешті, ми об'єднаємо екземпляр View Model і DataTemplate за допомогою ContentPresenter:

```
<ContentPresenter Content="{StaticResource TheViewModel}"
       ContentTemplate="{StaticResource TestViewModelDataTemplate}"
       Margin="10"/>
```

Повністю код даного проєкту подано в лістингу «Listing_6-6».

6.5 Контрольні запитання та завдання

1. Що таке концепція маршрутизації подій в Avalonia і як вона працює?
2. Які методи керування маршрутизованими подіями ви знаєте та як їх використовувати?
3. Що таке команди в Avalonia і яка їх роль?
4. Як використовуються команди Avalonia для виклику методів?
5. Що таке Avalonia ControlTemplates та Custom Control Templates і як їх використовувати?
6. Що таке шаблони даних у Avalonia і яка їх основна концепція?
7. Як реалізувати ContentPresenter у Avalonia? Наведіть приклад.
8. Як реалізувати ItemsPresenter у Avalonia? Наведіть приклад.

7 РЕАЛІЗАЦІЯ ДОСТУПУ ДО БАЗИ ДАНИХ ТА ВІДОБРАЖЕННЯ СТРУКТУРОВАНОЇ ІНФОРМАЦІЇ В AVALONIA

7.1 Avalonia DataGrid

7.1.1 Загальні відомості про DataGrid

Елемент DataGrid зовнішнє дуже схожий на ListView при використанні GridView, але при цьому не містить великий функціонал. Наприклад, DataGrid може автоматично генерувати стовбці в залежності від введених даних. Також, в DataGrid за замовчуванням можна редагувати вміст в комірках, завдяки чому кінцевому користувачу можна змінювати вміст в DataGrid.

Частіш за все DataGrid використовується в поєднанні з базою даних, але, як і більшість елементів Avalonia, може функціонувати і з локальним джерелом даних, наприклад, із списком об’єктів. Комірки в DataGrid можуть містити, наприклад, текстові дані, чек-бокси, зображення, випадні списки тощо.

На рис. 7.1 подано приклад реалізації DataGrid.

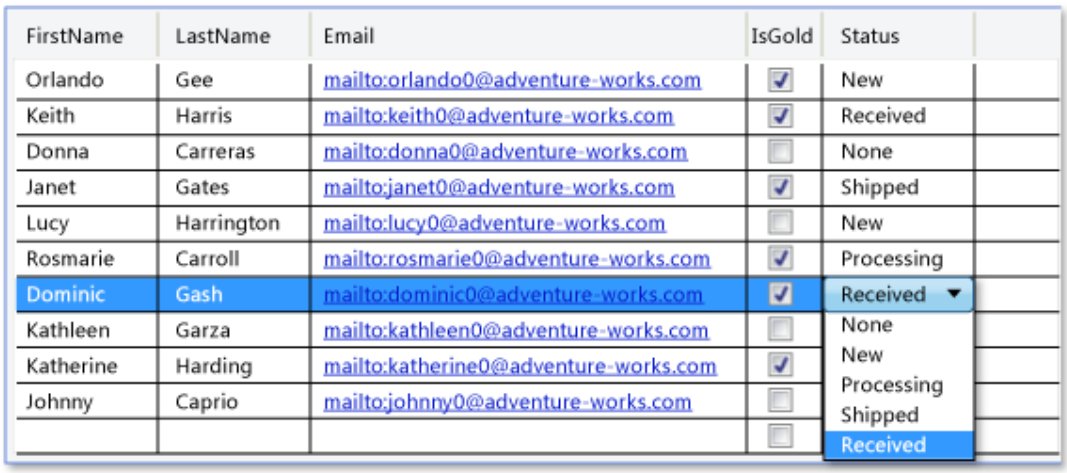


Рисунок 7.1 – Приклад реалізації DataGrid

7.1.2 Реалізація автоматичного генерування структури стовпців DataGrid

Для вивчення основ роботи з DataGrid створимо новий проєкт SimpleDataGrid, в якому реалізуємо простий інтерфейс виведення на екран таблиці з персональними даними користувачів програми, які взяті із вбудованого списку.

На першому етапі створюємо модель даних у вигляді окремого класу «Person» та розміщуємо його в окремій папці «Models» (рис. 7.2).

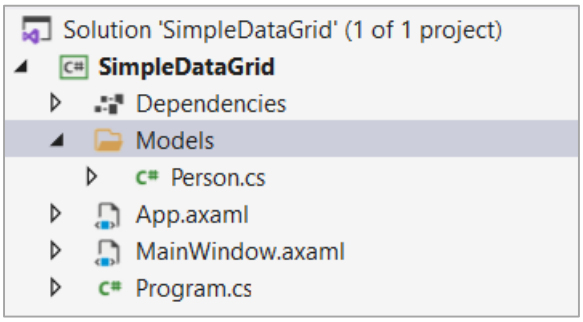


Рисунок 7.2 – Окремий клас «Person» з даними користувачів

Вміст файлу Person.cs наступний:

```
namespace SimpleDataGrid.Models
{
    public class Person
    {
        public int DepartmentNumber { get; set; }
        public string DeskLocation { get; set; }
        public int EmployeeNumber { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }
    }
}
```

В даному файлі створюються поля для зберігання даних про номер підрозділу (DepartmentNumber), ідентифікацію робочого місця (DeskLocation), код робітника у базі даних (EmployeeNumber), ім’я робітника (FirstName) та прізвище робітника (LastName).

Для заповнення списку тестовими даними створимо ще один клас «People», який буде викликатись на початку роботи програми. Вміст файлу People.cs наступний:

```
namespace SimpleDataGrid
{
    public class People : ObservableCollection<Person>
    {
        public void AddPerson(int departmentNumber,
```

```
            int employeeNumber,
            string deskLocation,
            string firstName,
            string lastName)
        {
            this.Add(new Person { DepartmentNumber = departmentNumber,
                EmployeeNumber = employeeNumber,
                DeskLocation = deskLocation,
                FirstName = firstName,
                LastName = lastName });
        }

        public People()
        {
            AddPerson(10, 1001, "R1F3R5T7", "John", "Plantagenet");
            AddPerson(10, 1002, "R1F1R2T3", "Richard", "Plantagenet");
            AddPerson(15, 1011, "R3F2R10T1", "Henry", "Tudor");
            AddPerson(20, 1021, "R7F4R10T7", "Elizabeth", "Tudor");
        }
    }
}
```

Ми використовуємо ObservableCollection для автоматичного заповнення колекції <Person> в нашому конструкторі.

В MainWindow.cs створюємо екземпляр класу People для можливості доступу до інформації з компонента DataGrid, що буде створений у головному вікні:

```
    public partial class MainWindow : Window
    {
        public People ThePeople { get; } = new People();

        public MainWindow()
        {
            InitializeComponent();
        }
    }
```

DataGrid має власні стилі, на які ми повинні посилатися в нашому App.xaml. Для використання DataGrid необхідно виконати модифікацію файлу

в його частині Application.Styles. Після відкриття файлу можна бачити стандартні властивості за замовчуванням:

```
<Application.Styles>
            <FluentTheme Mode="Light"/>
 </Application.Styles>
```

Необхідно закоментувати наявний рядок визначення стилю та додати наступні рядки коду:

```
<Application xmlns="https://github.com/avaloniaui"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             x:Class="SimpleDataGrid.App">
    <Application.Styles>
            <!-- <FluentTheme Mode="Light"/>-->
            <StyleInclude
Source="avares://Avalonia.Themes.Default/Accents/BaseLight.xaml"/>
            <StyleInclude
Source="avares://Avalonia.Themes.Default/DefaultTheme.xaml"/>
            <StyleInclude
Source="avares://Avalonia.Controls.DataGrid/Themes/Default.xaml"/>
    </Application.Styles>
</Application>
```

DataGrid може автоматично генерувати визначення стовпців на основі моделі даних, яка використовується в процесі його створення. В нашому прикладі достатньо визначити необхідність використання таблиці наступним чином:

```
<StackPanel>
      <DataGrid AutoGenerateColumns="True"
            Items="{Binding $parent[Window].ThePeople}"/>
</StackPanel>
```

Прив’язка в атрибуті Items вказує на властивість ThePeople через батьківський (перший предок) елемент Window.

В результаті компіляції отримуємо вигляд вікна, що подано на рис. 7.3.

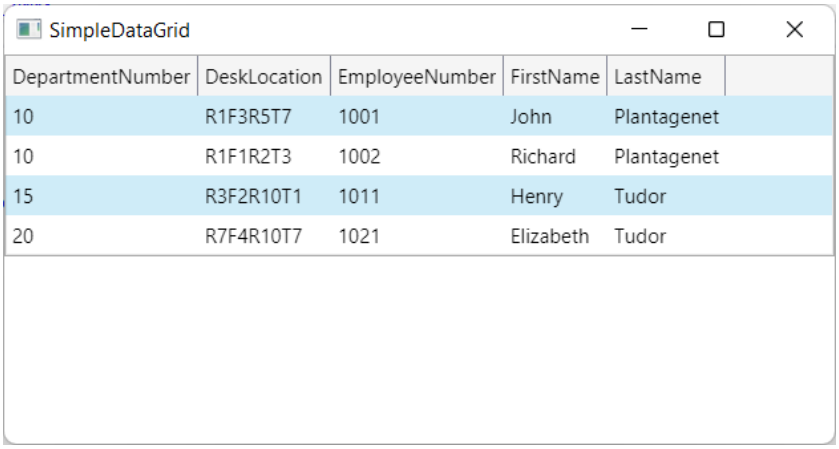


Рисунок 7.3 – Вигляд основного вікна з простою реалізацією таблиці

Повністю код прикладу подано в лістингу «Listing_7-1».

7.1.3 Реалізація діалогу користувача для додавання даних в колекцію

Управління даними на практиці практично завжди супроводжується такими функціями, як перегляд наявної інформації, додавання нових даних, видалення та редагування. Практично всі перелічені функції супроводжуються процедурою оновлення колекції даних після завершення їх модифікації. В будь-якому випадку необхідно реалізувати інтерфейс з діалоговими вікнами для взаємодії з користувачем програми.

В наступному прикладі підключимо в проєкт можливість додавати нову персону в колекцію даних <Person>. Для цього створимо нове вікно з полями введення для всіх полів класу Person з назвою «AddPerson.xaml». Приклад створення нового вікна описано в підрозділі 3.2.12 даного посібника. В результаті, структура проєкту буде наступна (рис. 7.4).

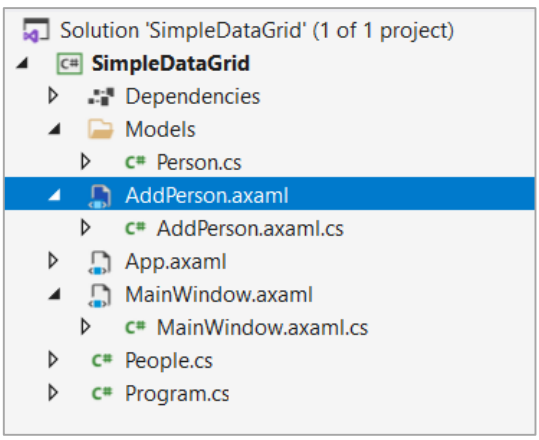


Рисунок 7.4 – Структура проєкту після створення нового вікна

В новому вікні в файлі AddPerson.xaml створимо розташування віджетів, як показано на рис. 7.5.

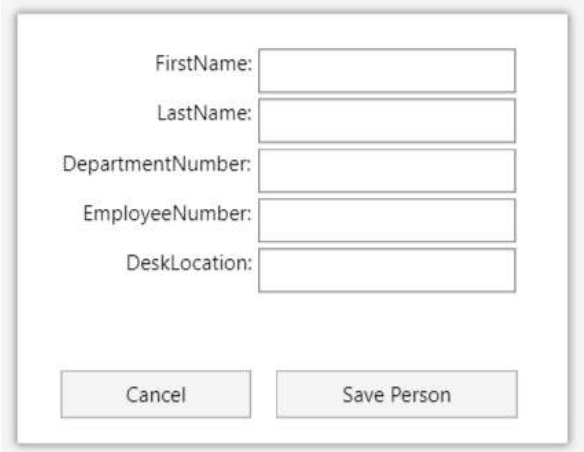


Рисунок 7.5 – Розташування віджетів у вікні
додавання персональних даних

Для того, щоб створити таку структуру, застосуємо контейнер Grid із наступними параметрами:

```
<Grid ColumnDefinitions="120, 3, Auto"
        RowDefinitions="Auto,Auto,Auto,Auto,Auto, *"
             Margin="15,20,20,0">

            .     .     .

</Grid>
```

З поданого коду XAML бачимо, що сітка має три стовпці з розмірами 120 пікселів, три пікселі, та останнім, що займає всю доступну область форми.

Рядки мають автоматичні параметри визначення ширини та займають стільки місця, скільки потребують компоненти, що в них розміщені. Останній рядок займає весь вільний простір. Всього передбачено шість рядків: п’ять для полів введення та один для розміщення кнопок керування.

Рядки введення мають назви. Всі назви розміщуються в першому стовпчику та мають таке визначення:

```
<TextBlock Grid.Column="0"
   Grid.Row="0"
```

```
            HorizontalAlignment="Right"> FirstName: </TextBlock>
        <TextBlock Grid.Column="0"
            Grid.Row="1"
            HorizontalAlignment="Right"> LastName: </TextBlock>
        <TextBlock Grid.Column="0"
            Grid.Row="2"
            HorizontalAlignment="Right"> DepartmentNumber: </TextBlock>
        <TextBlock Grid.Column="0"
            Grid.Row="3"
            HorizontalAlignment="Right"> EmployeeNumber: </TextBlock>
        <TextBlock Grid.Column="0"
            Grid.Row="4"
            HorizontalAlignment="Right"> DeskLocation: </TextBlock>
```

Навпроти кожної текстової мітки знаходиться поле введення:

```
<TextBox x:Name="tFirstName"
            Grid.Column="2"
            Margin="0,0,0,3"
            Grid.Row="0"
            Width="150"
            HorizontalAlignment="Left"/>
<TextBox x:Name="tLastName"
            Grid.Column="2"
            Margin="0,0,0,3"
            Grid.Row="1"
            Width="150"
            HorizontalAlignment="Left"/>
<TextBox x:Name="tDepartmentNumber"
            Grid.Column="2"
            Margin="0,0,0,3"
            Grid.Row="2"
            Width="150"
            HorizontalAlignment="Left"/>
<TextBox x:Name="tEmployeeNumber"
            Grid.Column="2"
            Margin="0,0,0,3"
            Grid.Row="3"
            Width="150"
            HorizontalAlignment="Left"/>
<TextBox x:Name="tDeskLocation"
            Grid.Column="2"
```

```
            Margin="0,0,0,3"
            Grid.Row="4"
            Width="150"
            HorizontalAlignment="Left"/>
```

Кожне поле окрім необхідних атрибутів візуального визначення, мають назву, наприклад: x:Name="tFirstName". Ця назва буде використовуватись для доступу до конкретного елемента інтерфейсу із коду C#.

Кнопки в коді XAML мають таку розмітку:

```
<Button x:Name="bCancel"
            Height="28"
            Margin="10,0,0,15"
            VerticalAlignment="Bottom"
            Grid.Column="0"
            Grid.Row="5">
      Cancel
</Button>
<Button x:Name="bSavePerson"
            Height="28"
            Margin="10,0,0,15"
            VerticalAlignment="Bottom"
            Grid.Column="2"
            Grid.Row="5">
      Save Person
</Button>
```

Передбачено дві кнопки: «Save Person» та «Cancel». Натискання на першу призводить до додавання в колекцію нового запису з персональними даними користувача. Натискання на другу – закриває вікно без зберігання введених даних.

В конструкторі класу AddPerson необхідно створити два об’єкти bSavePerson та bCancel для доступу до відповідних кнопок графічного інтерфейсу. До створених об’єктів додаємо реалізацію подій натискання на відповідні кнопки bSavePerson.Click та bCancel.Click:

```
public AddPerson()
{
      InitializeComponent();
```

```
        var bSavePerson = this.FindControl<Button>("bSavePerson");
        bSavePerson.Click += SaveButton_Click;

        var bCancel = this.FindControl<Button>("bCancel");
        bCancel.Click += Cancel_Click;
    }
```

Реалізація методу SaveButton_Click передбачає перевірку даних, що введено користувачем, конвертація її в Int (де потрібно) та заповнення полів об’єкта itemPerson типу Person:

```
    private void SaveButton_Click(object? sender, RoutedEventArgs e)
    {
        int departmentNumber = 0;
        bool saccessParseDN = int.TryParse(
            this.FindControl<TextBox>("tDepartmentNumber").Text,
            out departmentNumber);
        int DepartmentNumber = departmentNumber;

        int employeeNumber = 0;
        bool saccessParseEN = int.TryParse(
            this.FindControl<TextBox>("tEmployeeNumber").Text,
            out employeeNumber);
        int EmployeeNumber = employeeNumber;

        string DeskLocation = this.FindControl<TextBox>("tDeskLocation").Text;
        string FirstName = this.FindControl<TextBox>("tFirstName").Text;
        string LastName = this.FindControl<TextBox>("tLastName").Text;

        itemPerson = new Person();
        itemPerson.DepartmentNumber = DepartmentNumber;
        itemPerson.EmployeeNumber = EmployeeNumber;
        itemPerson.DeskLocation = DeskLocation;
        itemPerson.FirstName = FirstName;
        itemPerson.LastName = LastName;

        resultDialog = true;
        Close();
     }
```

Наприклад, поля DepartmentNumber та EmployeeNumber передбачають збереження числових даних, тому перед записом в них значення потрібно

перевірити та конвертувати в ціле число. Робиться це наступним чином із використанням методу TryParse класу int:

```
int departmentNumber = 0;
bool saccessParseDN = int.TryParse(
   this.FindControl<TextBox>("tDepartmentNumber").Text,
   out departmentNumber);
int DepartmentNumber = departmentNumber;
```

В класі AddPerson передбачені два поля:

```
public Person itemPerson;
public bool resultDialog;
```

для передачі введених персональних даних та результатів відпрацювання діалогового вікна. В методі SaveButton перед закриттям діалогового вікна в resultDialog заноситься «true»:

```
resultDialog = true;
Close();
```

Метод Cancel_Click містить тільки два рядки, серед яких один передає статус вікна – в resultDialog заноситься «false», а інший викликає метод Close():

```
private void Cancel_Click(object? sender, RoutedEventArgs e)
{
    resultDialog = false;
    Close();
}
```

Приклад відображення діалогового вікна подано на рис. 7.6.

Завершення роботи діалогового вікна може відбуватися двома способами: натисканням на кнопку SaveButton або на Cancel. В першому випадку resultDialog містить значення true. Таким чином, в класі, що викликав діалог, можна зробити висновок, що потрібно оновити список персональних даних. Якщо діалогове вікно закрито по кнопці Cancel, то в головному вікні у списку персон нічого додавати не треба, а просто відобразити вже наявні дані.

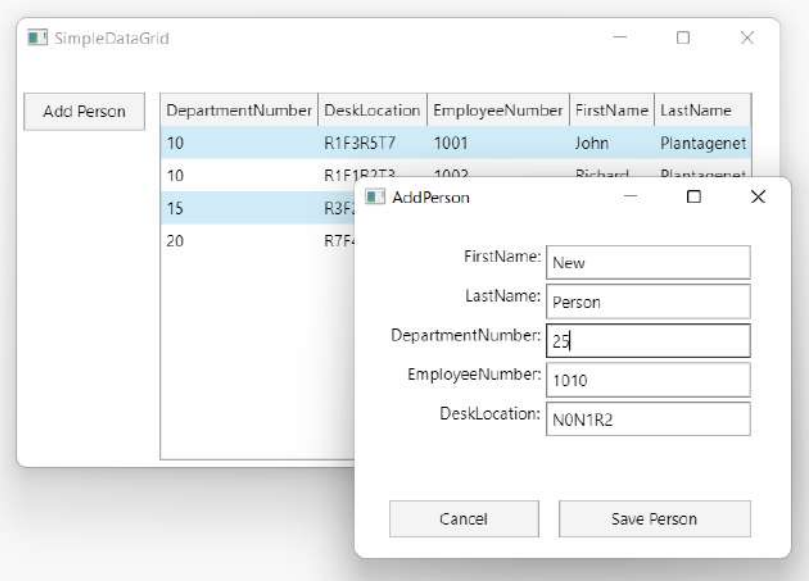


Рисунок 7.6 – Приклад відображення діалогового вікна

На рис. 7.7 подана діаграма станів для демонстрації операції додавання нового запису.

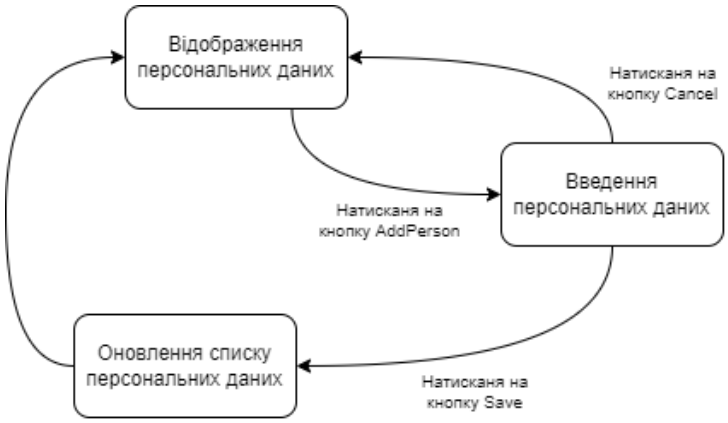


Рисунок 7.7 – Діаграма станів для операції додавання нового запису

Код головного вікна дуже простий:

```
public partial class MainWindow : Window
    {
        public People ThePeople { get; } = new People();

        public MainWindow()
        {
            InitializeComponent();

            var bAddPerson = this.FindControl<Button>("bAddPerson");
```

```
            bAddPerson.Click += AddPerson_Click;
        }

        async private void AddPerson_Click(object? sender,
            RoutedEventArgs e)
        {
            AddPerson addPerson = new AddPerson();
            await addPerson.ShowDialog(this);
            if (addPerson.resultDialog == true)
            {
                Person p = addPerson.itemPerson;
                ThePeople.Add(p);
            }
        }

        private void InitializeComponent()
        {
            AvaloniaXamlLoader.Load(this);
        }
    }
```

Після ініціалізації відбувається прив’язка коду до кнопки виклику діалогового вікна:

```
var bAddPerson = this.FindControl<Button>("bAddPerson");
            bAddPerson.Click += AddPerson_Click;
```

Обробка натискання на кнопку AddPerson викликається асинхронний метод AddPerson_Click:

```
async private void AddPerson_Click(object? sender, RoutedEventArgs e)
{
AddPerson addPerson = new AddPerson();
      await addPerson.ShowDialog(this);
      if (addPerson.resultDialog == true)
      {
            Person p = addPerson.itemPerson;
            ThePeople.Add(p);
      }
}
```

Даний метод зроблено із використанням префіксу async, що наділяє програму можливістю очікування завершення закриття діалогового вікна. Така поведінка гарантується наявністю оператора «await» перед викликом методу ShowDialog об’єкта addPerson:

```
await addPerson.ShowDialog(this);
```

Після закриття вікна, програма обробляє змінну resultDialog:

```
if (addPerson.resultDialog == true)
{
      Person p = addPerson.itemPerson;
      ThePeople.Add(p);
}
```

Тобто, якщо змінна містить значення «true», список ThePeople доповнюється новими даними. Завдяки використанню ObservableCollection, в процесі створення екземпляра Person, ми отримуємо динамічну колекцію, що надає повідомлення при додаванні, або видаленні елементів списку. В нашому випадку це приводить до автоматичного оновлення списку в елементі DataGrid.

Код XAML для головного вікна має такий вигляд:

```
<Window xmlns="https://github.com/avaloniaui"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
   Width="560"
   Height="300"
   x:Class="SimpleDataGrid.MainWindow"
   Title="SimpleDataGrid">

   <Grid ColumnDefinitions="100, Auto"
        RowDefinitions="*"
        Margin="0,20,20,0">
      <StackPanel Grid.Column="0"
         Margin="5">
         <Button x:Name="bAddPerson"
            Height="28">
            Add Person
         </Button>
      </StackPanel>
```

```
        <DockPanel Grid.Column="1"
           Margin="5">
           <DataGrid AutoGenerateColumns="True"
              VerticalAlignment="Stretch"
              Items="{Binding $parent[Window].ThePeople}"/>
        </DockPanel>
     </Grid>
  </Window>
```

StackPanel не підтримує розтягування дочірніх елементів, тому в коді, який ми взяли за основу із попереднього прикладу, замінюємо його на DockPanel. В даному контейнері передбачена реакція на розтягування останнього компонента в списку. За замовчуванням, останній компонент займає весь вільний простір. За це відповідає його властивість «LastChildFill».

Якщо для властивості LastChildFill встановлено значення true, що є параметром за замовчуванням, останній дочірній елемент DockPanel завжди заповнює простір, що залишився, незалежно від будь-якого іншого значення DockPanel, яке ви встановлюєте для останнього дочірнього елемента. Щоб закріпити дочірній елемент в іншому напрямку, необхідно встановити для властивості LastChildFill значення false, а також вказати явний напрямок закріплення для останнього дочірнього елемента.

Таким чином віджет DataGrid завжди буде розтягуватись вниз до повного зайняття всього вільного простору основного вікна (рис. 7.8).

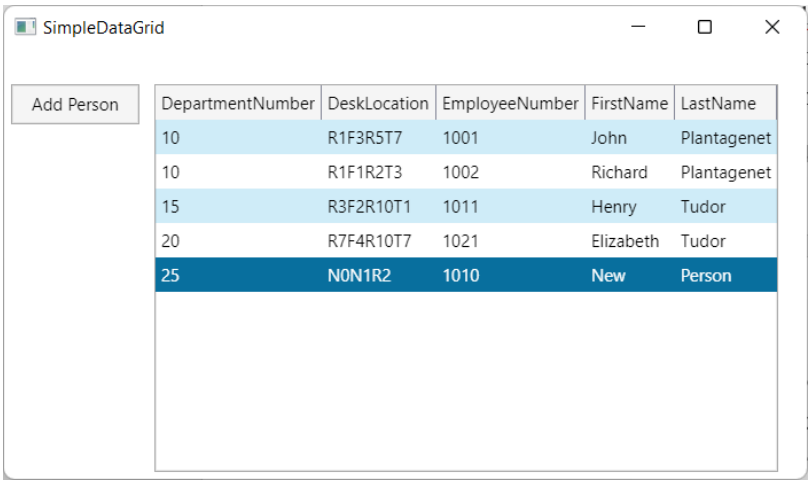


Рисунок 7.8 – Заповнення списку після додавання нового рядка даних

Повністю код прикладу подано в лістингу «Listing_7-2».

7.2 Використання бази даних для роботи з DataGrid в Avalonia

7.2.1 Створення та підключення до бази даних

Для збереження введеної інформації після виходу з програми використовують бази даних. В даному прикладі в якості СУБД будемо використовувати SQLite, що дає можливість розгортання її прямо на ПК користувача без встановлення додаткових серверів та програмних засобів.

SQLite є однією з найбільш використовуваних систем управління базами даних. Головною перевагою SQLite є те, що для бази даних не потрібно сервера.

База даних являє собою звичайний локальний файл, який ми можемо переміщати разом з головним файлом програми. Крім того, для запитів до бази даних ми можемо використовувати стандартні вирази мови SQL, які так само, з деякими змінами, можуть застосовуватися і в інших СУБД, таких, як Oracle, MS SQL Server, MySQL, Postgres тощо.

Ще однією перевагою є широке розповсюдження SQLite – область застосування охоплює безліч платформ і технологій: WPF, Windows Forms, UWP, Xamarin, Android, iOS та інші.

Для використання розробленої структури бази даних необхідно встановити необхідні бібліотеки для роботи з цією базою даних. Перш за все, для роботи SQLite в C# необхідно встановити через Nuget пакет System.Data.SQlite. На рис. 7.9 подано приклад додавання пакетів до програми.

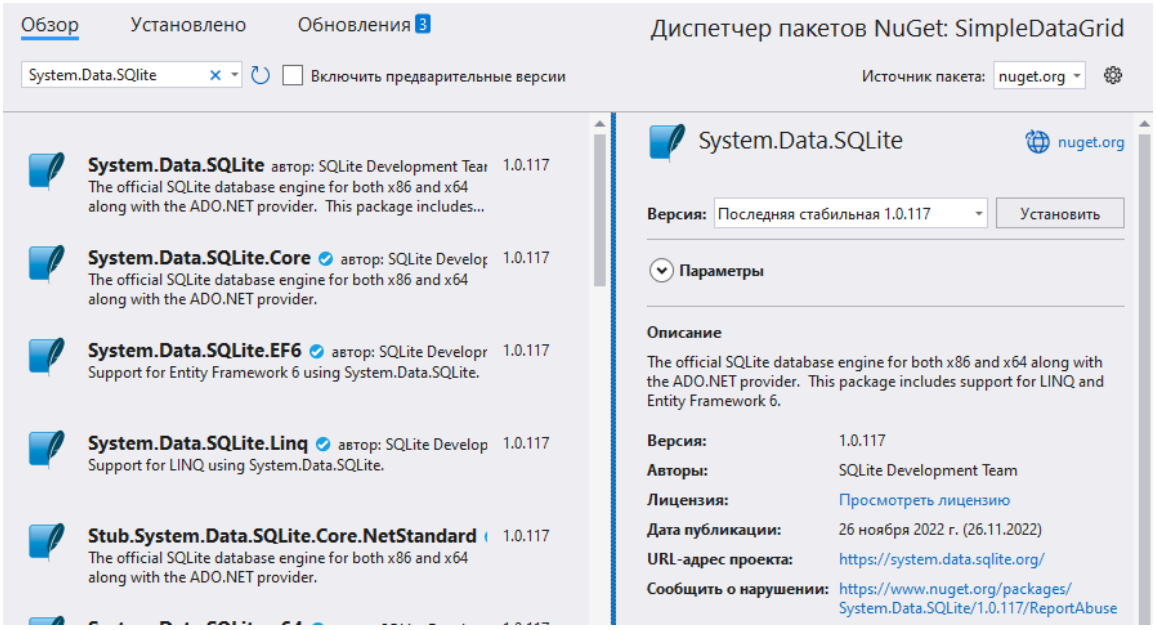


Рисунок 7.9 – Встановлення пакетів System.Data.SQlite

Для створення бази даних будемо використовувати один з безкоштовних програмних продуктів з відкритим вихідним кодом «DB Browser для SQLite». Завантажити останню версію програми можна за посиланням: https://github.com/sqlitebrowser/sqlitebrowser. Там же можна знайти вихідний код програми.

DB Browser для SQLite (DB4S) – це високоякісний візуальний інструмент із відкритим кодом для створення, проєктування та редагування файлів бази даних, сумісних із SQLite.

DB4S призначений для користувачів і розробників, які хочуть створювати, шукати та редагувати бази даних. DB4S використовує знайомий інтерфейс, подібний до електронної таблиці, тому не потрібно вивчати складні команди SQL.

Елементи керування та майстри, що доступні для користувачів:

– створення та стискання файлу бази даних;

– створення, визначення, зміни та видалення таблиць;

– створення, визначення та видалення індексів;

– перегляд, редагування, додавання та видалення записів;

– пошук записів;

– імпорт та експорт записів як тексту;

– імпорт та експорт таблиць із/до файлів CSV;

– імпорт та експорт баз даних із/до файлів дампа SQL;

– виконання SQL-запитів та перевірка результатів;

– перегляд журналу усіх команд SQL, що може бути опрацьовано програмою;

– будування простих графіків на основі даних таблиці або запиту.

Зовнішній вигляд інтерфейсу програми подано на рис. 7.10.

Для зберігання даних про персону створимо нову базу даних та відповідну таблицю. В якості назви бази даних обираємо «Employee.db».

Шлях до бази даних краще обрати в папці «…<Назва проєкту>\ bin\Debug\net6.0». Тоді до неї можна буде просто звернутись, маючи дані про папку запуску додатка.

Відповідно до класу person:

```
public class Person
{
```

```
        public int DepartmentNumber { get; set; }
        public string DeskLocation { get; set; }
        public int EmployeeNumber { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }
    }
```

створюємо структуру відповідної таблиці (табл. 7.1).

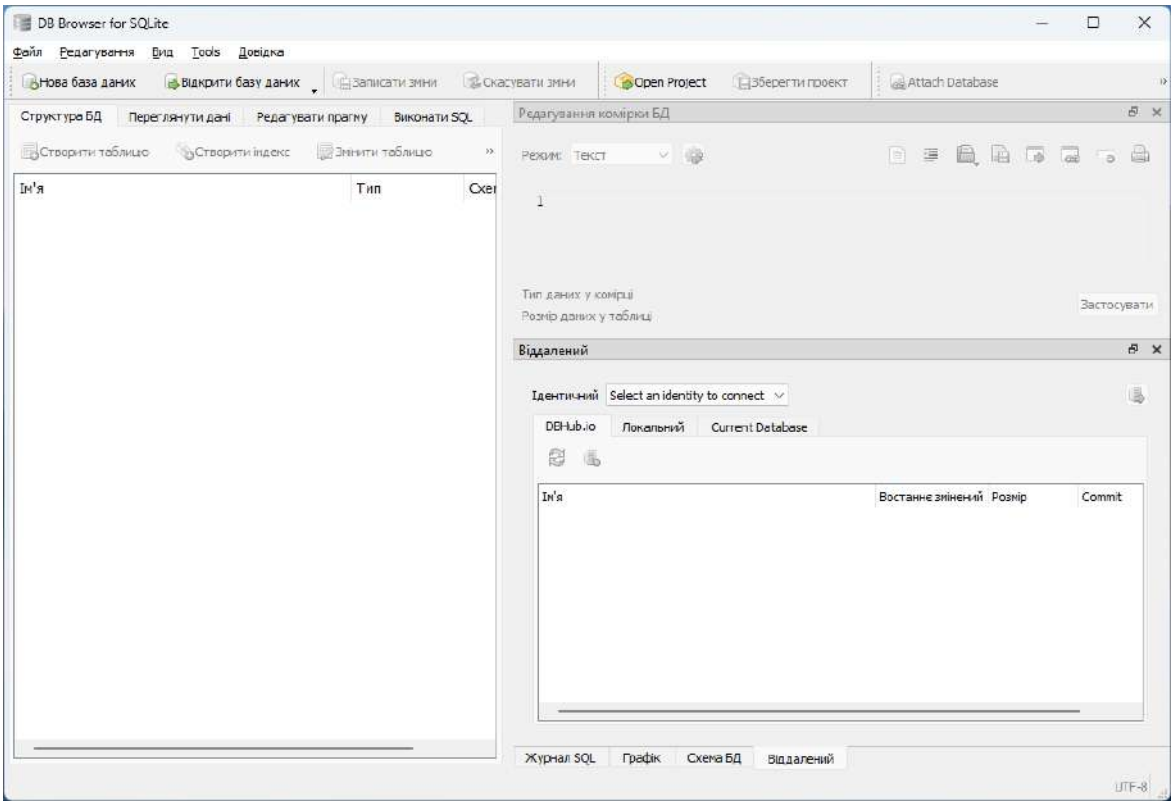


Рисунок 7.10 – Зовнішній вигляд інтерфейсу програми DB Browser для SQLite

Таблиця 7.1 – Структура таблиці «Person»

Назва поля	Призначення	Тип поля	Ключ
EmployeeNumber	Код робітника у базі даних	INTEGER	Первинний
FirstName	Ім’я робітника	TEXT	
LastName	Прізвище робітника	TEXT	
DepartmentNumber	Номер підрозділу	INTEGER	Зовнішній
DeskLocation	Ідентифікація робочого місця	TEXT	Зовнішній

Поле «EmployeeNumber» є первинним ключем, в якому всі дані унікальні. Це поле має бути цілим (Integer) із встановленої ознакою «Автоінкримент» та

«Первинний ключ». На рис. 7.11 подано приклад вікна створення таблиці Person. На даному рисунку можна бачити атрибути поля «EmployeeNumber» та обрані ознаки.

Поле «DepartmentNumber» є посиланням на таблицю зі списком підрозділів. Дане поле – це зовнішній ключ, який повинен мати тип Integer. В нашому простому прикладі ми не будемо посилатись на інші таблиці, тому значення в полі «DepartmentNumber» вводитиметься вручну.

Поле «DeskLocation» є ідентифікатором робочих місць. Воно також повинно бути зовнішнім ключем, але в нашому спрощеному прикладі містить абстрактні значення введені вручну.

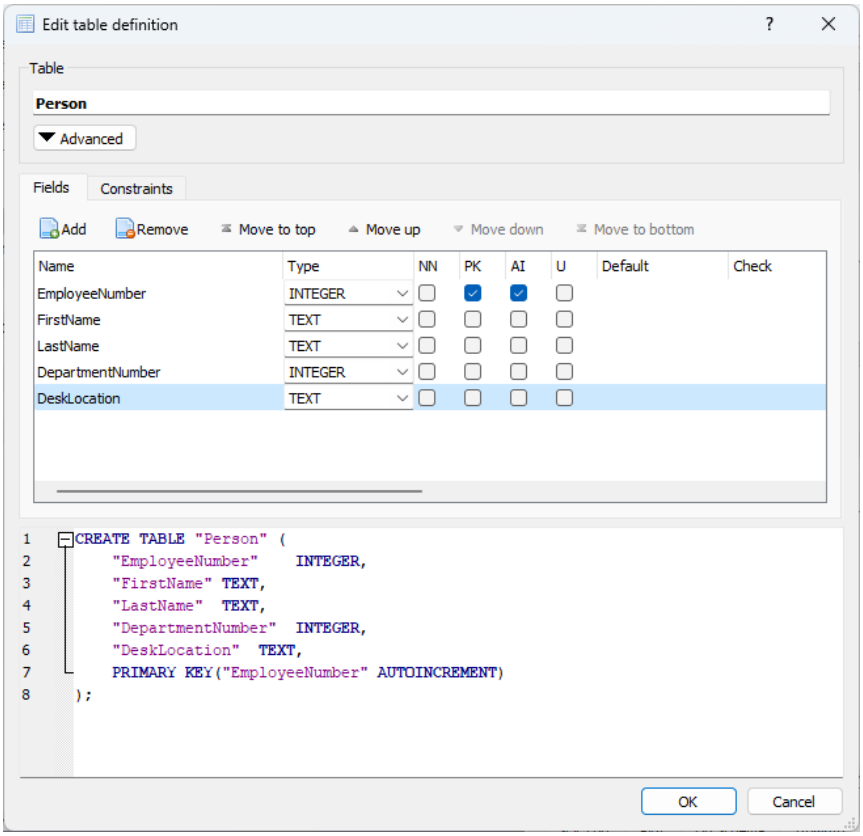


Рисунок 7.11 – Приклад вікна створення таблиці Person

Для підключення до бази даних в програмі необхідно створити змінну зі шляхом до файлу Employee.db. Це можна зробити за допомогою двох рядків коду:

```
public static String mPath = AppDomain.CurrentDomain.BaseDirectory;
public static String mDBPath = mPath + "Employee.db";
```

Перший рядок отримує посилання на базову робочу папку звідки завантажена наша програма. Інший рядок додає до базового шляху назву нашої бази даних. Ці два рядки необхідно вставити на початку файлу «MainWindow.axaml.cs» (рис. 7.13).

```
namespace SimpleDataGrid
{
    Ссылок: 2
    public partial class MainWindow : Window
    {
        public static String mPath = AppDomain.CurrentDomain.BaseDirectory;
        public static String mDBPath = mPath + "Employee.db";

        Ссылок: 1
        public People ThePeople { get; } = new People();

        Ссылок: 1
        public MainWindow()
        {
            InitializeComponent();
```

Рисунок 7.13 – Додавання рядків посилання на базу даних

Робота з базою даних включає в себе функції:

– отримання даних з таблиць у відповідності до поточної задачі;

– додавання нових даних;

– оновлення даних в таблиці;

– видалення даних.

Виконаємо реалізацію першої функції. Цю та інші функції будемо розміщувати в класі «People».

Функцію, що отримує дані з таблиці Person та заповнює відповідний список, назвемо «FillPeople». Результатом роботи функції буде заповнений список типу ObservableCollection<Person>.

На початку роботи функції очищуємо список від минулих записів:

```
this.Clear();
```

Наступним кроком відбувається підключення до бази даних. Для цього додамо наступні рядки:

```
string connString = String.Format("Data Source={0};
      New=False; Version=3", MainWindow.mDBPath);
SQLiteConnection sqlite_conn = new SQLiteConnection(connString);
sqlite_conn.Open();
```

Змінна «connString» отримує рядок, в якому містяться дані про підключення до бази даних:

– назву та шлях до бази даних;

– ознаку New, якій відповідає результат False, що означає, що не створюються нова база, а відкривається вже існуюча та попередні дані зберігаються;

– версія SQLite (3 версія).

Змінна «sqlite_conn» отримує посилання на рядок підключення, та за викликом методу «Open» намагається відкрити вказану базу даних. На даному етапі може виникнути помилка, яка пов’язана з відсутністю бази за вказаним шляхом, або помилка в назві файлу бази даних.

На наступному етапі створюється рядок SQL запиту для обирання потрібних записів з таблиці Person:

```
string sql = String.Format("Select * from Person order by LastName;");
```

Цей запит покаже всі рядки з таблиці Person та відсортує їх за полем LastName, тобто за прізвищем.

Після створення SQL-запита, передаємо його в зміну, що відображає команду SQLite – sqlite_cmd типу SQLiteCommand:

```
SQLiteCommand sqlite_cmd = new SQLiteCommand(sql, sqlite_conn);
```

Команда може виконуватись без повернення результату ExecuteNonQuery() та с поверненням – ExecuteReader(). В даному випадку використовуємо останній варіант, тому що нам необхідно отримати список співробітників з бази даних:

```
SQLiteDataReader reader = (SQLiteDataReader)sqlite_cmd.ExecuteReader();
```

Результат, що повертається, привласнюється змінній reader типу SQLiteDataReader. Клас SQLiteDataReader надає методи для читання результату команди, виконаної в базі даних SQLite.

В результаті позитивного виконання SQL запиту змінна reader буде вказувати на перший рядок отриманих даних. Для читання всіх даних необхідно

в циклі викликати reader.Read(), поки не буде отримано значення False, що означає, що список даних закінчився.

В нашій програмі це реалізовано наступним чином:

```
while (reader.Read())
{
      .     .     .
}
```

В кожній ітерації циклу можна отримати значення з будь-якого поля таблиці. Для цього необхідно звернутись до нього за ім’ям та вказати тип перетворення результату, наприклад:

```
int EmployeeNumber = Convert.ToInt32(reader["EmployeeNumber"]);
```

В даному прикладі звертаємось до поля EmployeeNumber, перетворюємо отримане значення в ціле число та записуємо в відповідну змінну EmployeeNumber.

Для читання всіх полів та додавання отриманого результату до списку персон використовується наступний код:

```
while (reader.Read())
{
      int EmployeeNumber = Convert.ToInt32(reader["EmployeeNumber"]);
      string firstName = Convert.ToString(reader["firstName"]);
      string lastName = Convert.ToString(reader["lastName"]);
      int departmentNumber = Convert.ToInt32(reader["departmentNumber"]);
      string deskLocation = Convert.ToString(reader["deskLocation"]);
      AddPerson(departmentNumber, EmployeeNumber, deskLocation, firstName,
lastName);
}
```

Рядок:

```
AddPerson(departmentNumber, EmployeeNumber, deskLocation, firstName,
lastName);
```

передає отримані дані в метод AddPerson, що додає їх до загального списку персон.

Після завершення операції читання з бази даних, необхідно закрити рідер та завершити підключення:

```
reader.Close();
sqlite_conn.Close();
```

Повий код функції FillPeople() наведено нижче:

```
public ObservableCollection<Person> FillPeople()
{
   this.Clear();

   string connString = String.Format(
      "Data Source={0};New=False;Version=3", MainWindow.mDBPath);
   SQLiteConnection sqlite_conn = new SQLiteConnection(connString);
   sqlite_conn.Open();

   string sql = String.Format("Select * from Person order by LastName;");

   SQLiteCommand sqlite_cmd = new SQLiteCommand(sql, sqlite_conn);
   try
   {
      SQLiteDataReader reader =
        (SQLiteDataReader)sqlite_cmd.ExecuteReader();
      while (reader.Read())
      {
         int EmployeeNumber = Convert.ToInt32(reader["EmployeeNumber"]);
         string firstName = Convert.ToString(reader["firstName"]);
         string lastName = Convert.ToString(reader["lastName"]);
         int departmentNumber =
            Convert.ToInt32(reader["departmentNumber"]);
         string deskLocation = Convert.ToString(reader["deskLocation"]);

         AddPerson(departmentNumber, EmployeeNumber,
            deskLocation, firstName, lastName);
         }
      reader.Close();
      sqlite_conn.Close();
   }
   catch (Exception ex) { }
   return this;
}
```

Створену функцію необхідно додати до конструктора класу People, щоб, кожен раз при створенні нового екземпляру класу, автоматично виконувалась функція читання інформації з бази даних:

```
public People()
{
      FillPeople();
}
```

Результат першого запуску програми подано на рис. 7.14.

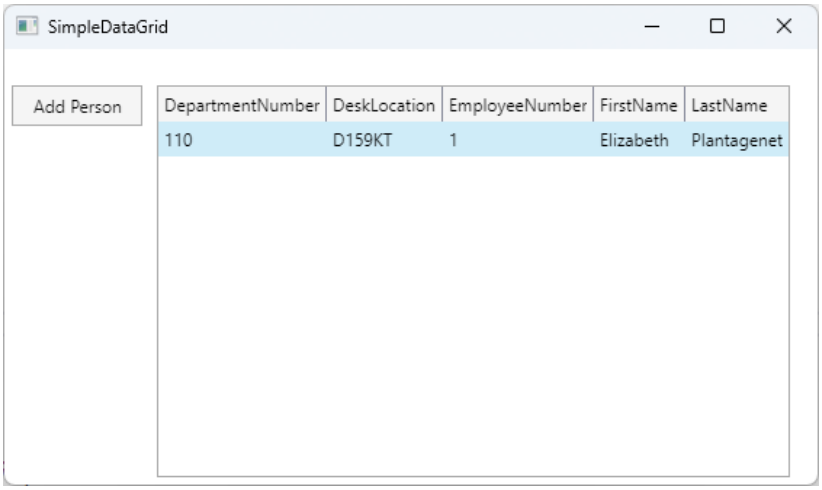


Рисунок 7.14 – Відображення інформації з таблиці Person

На даному рисунку можна бачити відображення одного рядку даних, що був попередньо доданий до відповідної таблиці за допомогою редактора бази даних DB4S.

7.2.2 Реалізація функції додавання інформації до бази даних

Для того, що працювала кнопка AddPerson необхідно внести зміни в метод обробки натискання на цю кнопку. Потрібно видалити два рядки:

```
//Person p = addPerson.itemPerson
//ThePeople.Add(p);
```

та додати новий:

```
ThePeople.FillPeople();
```

Приклад модифікації методу обробки натискання на кнопку AddPerson поданий на рис. 7.15.

```
async private void AddPerson_Click(object? sender, RoutedEventArgs e)
{
    AddPerson addPerson = new AddPerson();
    await addPerson.ShowDialog(this);
    if (addPerson.resultDialog == true)
    {
        //Person p = addPerson.itemPerson
        //ThePeople.Add(p);
        ThePeople.FillPeople();
    }
}
```

Рисунок 7.15 – Приклад модифікації методу обробки натискання на кнопку AddPerson

Додавання нової інформації в базу даних відбуватиметься за допомогою SQL-запиту. Для цього необхідно створити допоміжний клас, в якому будемо розміщувати загальні методи для роботи з базою даних. Цей клас необхідно створити, додаючи до проєкту, як новий елемент, за допомогою правою кнопки миші. На рис. 7.16 подано вікно додавання нового класу.

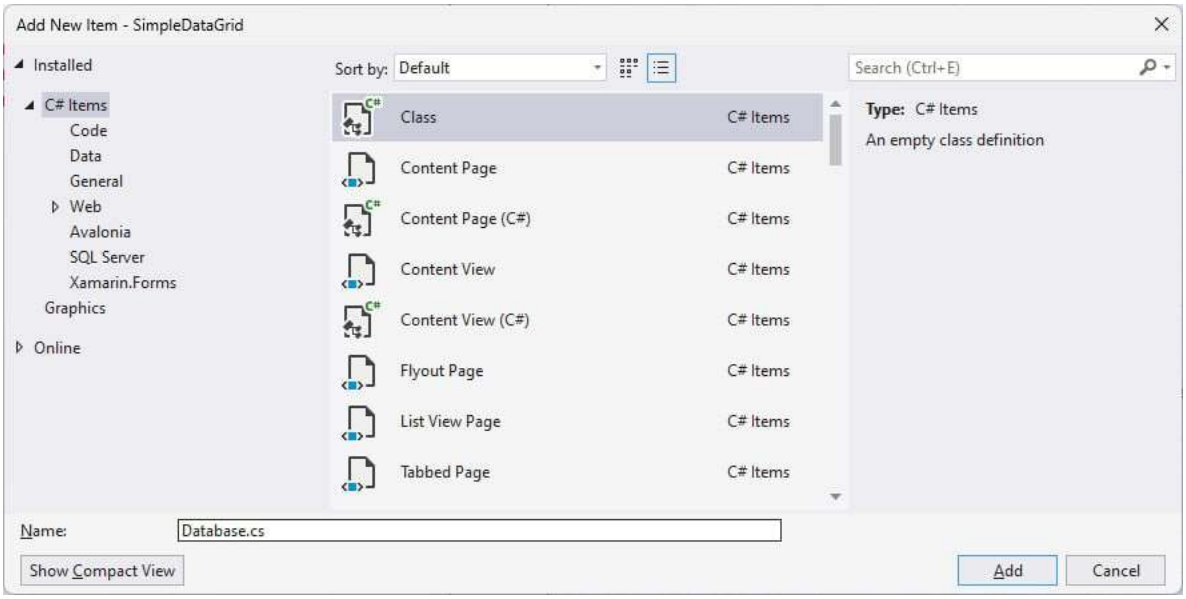


Рисунок 7.16 – Вікно додавання нового класу

Після створення класу «Database», дерево проєкту матиме вигляд, як на рис. 7.17.

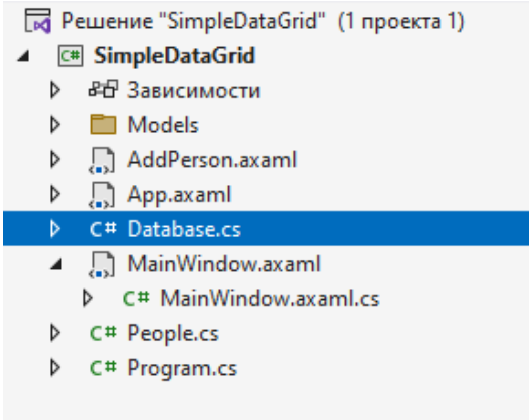


Рисунок 7.17 – Дерево проєкту після додавання класу «Database»

В новому класі Database створимо функцію Exec_SQL. Дана функція буде виконувати прості запити до бази даних та повертати результат виконання запиту у вигляді булевої змінної (true або false).

Код функції виконання SQL-запиту наступний:

```
public static bool Exec_SQL(string sql)
{
   bool result = false;

   string connString = String.Format(
      "Data Source={0};New=False;Version=3", MainWindow.mDBPath);
   SQLiteConnection sqlite_conn = new SQLiteConnection(connString);
   sqlite_conn.Open();

   SQLiteCommand sqlite_cmd = new SQLiteCommand(sql, sqlite_conn);
   try
   {
      sqlite_cmd.ExecuteNonQuery();
      result = true;
   }
   catch (Exception ex)
   {
      result = false;
   }

   sqlite_conn.Close();
   return result;
}
```

Дана функція отримує в якості вхідного параметра текстовий рядок – SQL-запит, що необхідно виконати. Підключення до бази даних виконується таким самим методом, що було описано раніше. На відміну від задачі отримання після виконання запиту набору рядків, в нашому випадку ми нічого не очікуємо, тому використовуємо команду ExecuteNonQuery():

```
sqlite_cmd.ExecuteNonQuery();
```

Після успішного виконання операції, в змінну result запишеться результат true, при виникненні помилки – false.

Для додавання в базу даних нового рядка інформації про співробітника створюється SQL-запит «Insert», до якого передаються параметри, що беруться з екранної форми, яку заповнює користувач. Нижче наведено приклад формування SQL-запиту:

```
string sql = string.Format("Insert into Person (
  DepartmentNumber, DeskLocation, FirstName, LastName) " +
  "Values (‘{0}’, ‘{1}’, ‘{2}’, ‘{3}’);",
DepartmentNumber, DeskLocation, FirstName, LastName);
resultDialog = Database.Exec_SQL(sql);
```

Повний код функції обробки натискання на кнопку «Зберегти» поданий нижче:

```
private void SaveButton_Click(object? sender, RoutedEventArgs e)
{
  int departmentNumber = 0;
  bool saccessParseDN =
  int.TryParse(this.FindControl<TextBox>("tDepartmentNumber").Text,
  out departmentNumber);
  int DepartmentNumber = departmentNumber;

  string DeskLocation = this.FindControl<TextBox>("tDeskLocation").Text;
  string FirstName = this.FindControl<TextBox>("tFirstName").Text;
  string LastName = this.FindControl<TextBox>("tLastName").Text;

  string sql = string.Format("Insert into Person (
    DepartmentNumber, EmployeeNumber, DeskLocation, FirstName, LastName)"
    +
```

```
        "Values (‘{0}’, ‘{1}’, ‘{2}’, ‘{3}’, ‘{4}’);",
        DepartmentNumber, EmployeeNumber, DeskLocation, FirstName, LastName);
      resultDialog = Database.Exec_SQL(sql);

      Close();
    }
```

В результаті роботи даної функції отримуємо можливість додавання до бази даних нової інформації про співробітника (рис. 7.18).

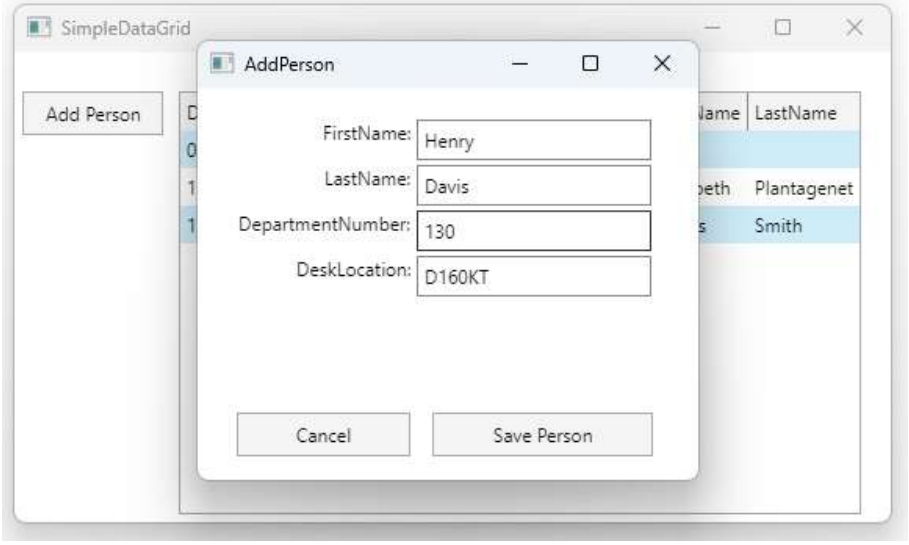


Рисунок 7.18 – Приклад вікна додавання інформації про співробітника

На даному рисунку можна бачити приклад вікна заповнення інформації про співробітника. Заповнюються всі вказані в моделі даних поля. Після натискання на кнопку «Save Person» вся введена інформація потрапляє до бази даних.

З поданого прикладу можна бачити, що поле EmployeeNumber не заповнюється та відсутнє в запиті про додавання інформації в базу даних. Це пов’язано з тим, що дане поле позначене як таке, що автоматично змінює наповнення на одиницю з кожним додаванням нового запису. Таким чином, СУБД самостійно слідкує, щоб в полі EmployeeNumber завжди був унікальний номер.

На рис. 7.19 подано приклад виведення інформації з бази даних. З даного прикладу можна бачити, що додані дані збереглися та виводяться при виконанні відповідного SQL-запиту.

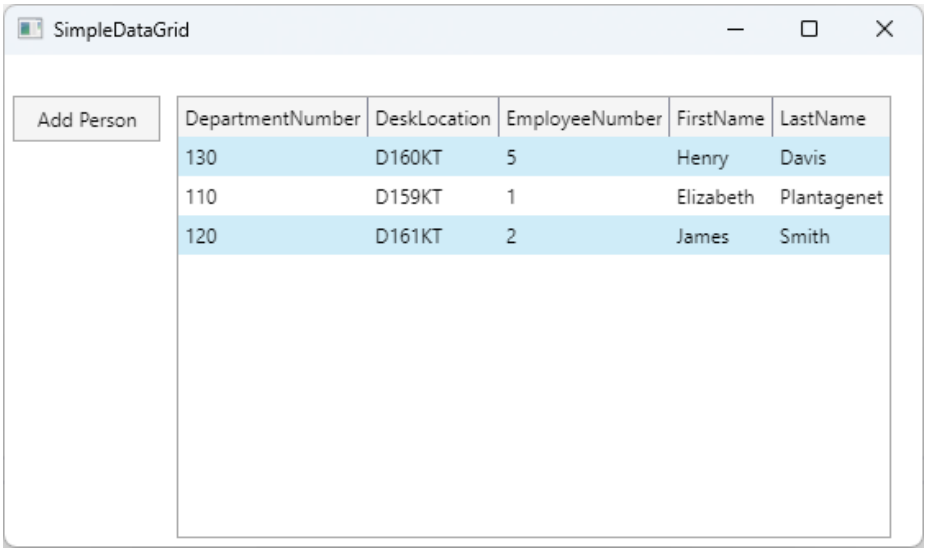


Рисунок 7.19 – Приклад виведення інформації з бази даних

7.2.3 Реалізація функції редагування виділеної інформації в базі даних

Для виконання дій, що пов’язані з вже доданою інформацією до бази даних, по-перше, необхідно реалізувати в програмі можливість вибору потрібного рядку в таблиці.

В віджеті DataGrid є можливість обробити подію SelectionChanged. Ця подія виникає кожного разу, коли користувач вказує курсором на потрібний йому рядок в таблиці на екрані комп’ютера.

Щоб обробити цю подію в коді програми, в конструкторі головного вікна необхідно додати такі рядки:

```
var gDataGrid = this.FindControl<DataGrid>("gDataGrid");
gDataGrid.SelectionChanged += mSelectionChanged;
```

Перший рядок виконує пошук компоненти DataGrid та присвоює знайдений результат змінній gDataGrid. Пошук відбувається за ім’ям компоненти «gDataGrid». Це ім’я необхідно створити в коді файлу «MainWindow.axaml» в тегах, що описують відповідний компонент:

```
<DataGrid x:Name="gDataGrid"
      AutoGenerateColumns="True"
      VerticalAlignment="Stretch"
      Items="{Binding $parent[Window].ThePeople}"/>
```

Другий рядок доданого коду створює посилання на функцію «mSelectionChanged». Цю функцію необхідно створити:

```
private void mSelectionChanged(object? sender,
      SelectionChangedEventArgs e)
{
      DataGrid grid = (DataGrid)sender;
      dynamic selected_row = grid.SelectedItem;
      SelectPerson = selected_row;
}
```

В даній функції ми отримуємо посилання на рядок, який було виділено користувачем, та зберігаємо його в змінну «selected_row».

Далі змінну «selected_row» присвоюємо глобальній змінній «SelectPerson» для подальшого використання, наприклад, для редагування:

```
SelectPerson = selected_row;
```

Щоб реалізувати функцію редагування виділеної інформації в базі даних, необхідно додати до головної форми кнопку «Edit Person»:

```
<Button x:Name="bEditPerson"
      Margin="0,10,0,0"
      Height="28">
      Edit Person
</Button>
```

Додана кнопка має назву «bEditPerson». Прив’язка коду до неї відбувається в конструкторі головної форми:

```
var bEditPerson = this.FindControl<Button>("bEditPerson");
bAddPerson.Click += EditPerson_Click;
```

Функція, що обробляє натискання на кнопку «bEditPerson», має назву «EditPerson_Click». Її завдання – передати у вікно «AddPerson» код обраного співробітника. Це робиться для того, щоб в даному вікні вивести користувачеві ту інформацію, що наразі міститься в базі даних, та дати йому змогу змінити потрібні поля.

Крім того, необхідно передати у вікно редагування ознаку режиму, в якому воно буде знаходитись: в режимі додавання (insert) або в режимі редагування (edit). За замовчуванням, вікно буде знаходитись в режимі додавання (файл «AddPerson.axaml.cs»):

```
public string mode = "insert";
```

Для передавання даних між вікнами програми можна скористатись глобальними полями. В даному випадку для передавання інформації про обраного співробітника створимо глобальну змінну:

```
public Person SelectPerson = new Person();
```

Змінемо код, який відповідає натисканню на кнопку, на наступне:

```
private void SaveButton_Click(object? sender, RoutedEventArgs e)
{
   int departmentNumber = 0;
   bool saccessParseDN =
     int.TryParse(this.FindControl<TextBox>("tDepartmentNumber").Text,
     out departmentNumber);
   int DepartmentNumber = departmentNumber;

   string DeskLocation = this.FindControl<TextBox>("tDeskLocation").Text;
   string FirstName = this.FindControl<TextBox>("tFirstName").Text;
   string LastName = this.FindControl<TextBox>("tLastName").Text;

   string sql = "";
   if (mode == "insert")
   {
      sql = string.Format("Insert into Person (DepartmentNumber,
      DeskLocation, FirstName, LastName) " +
      "Values (‘{0}’, ‘{1}’, ‘{2}’, ‘{3}’);",
      DepartmentNumber, DeskLocation, FirstName, LastName);
   }
   else
   {
      sql = string.Format("Update Person SET DepartmentNumber = ‘{0}’, " +
      "DeskLocation = ‘{1}’, " +
      "FirstName = ‘{2}’, " +
```

```
            "LastName = ‘{3}’ " +
            "Where EmployeeNumber = ‘{4}’;",
            DepartmentNumber, DeskLocation, FirstName,
            LastName, SelectPerson.EmployeeNumber);
        }
        resultDialog = Database.Exec_SQL(sql);

        Close();
    }
```

Також необхідно додати нову функцію «AddPerson_Activated» для вмісту рядків коду, що виконують початкове заповнення полів форми інформацією в режимі «edit»:

```
private void AddPerson_Activated(object? sender, EventArgs e)
{
    if (mode == "edit")
    {
        if (SelectPerson != null)
        {
            this.FindControl<TextBox>("tDepartmentNumber").Text =
                SelectPerson.DepartmentNumber.ToString();
            this.FindControl<TextBox>("tDeskLocation").Text =
                SelectPerson.DeskLocation.ToString();
            this.FindControl<TextBox>("tFirstName").Text =
                SelectPerson.FirstName.ToString();
            this.FindControl<TextBox>("tLastName").Text =
                SelectPerson.LastName.ToString();
        }
    }
}
```

На рис. 7.20 подано приклад роботи доданої функції редагування обраного запису.

7.2.4 Реалізація функції видалення інформації з бази даних

Для видалення обраної інформації потрібно створити відповідну кнопку на головному екрані програми:

```
        <Button x:Name="bDeletePerson"
```

```
            Margin="0,10,0,0"
            Height="28">
            Delete Person
        </Button>
```

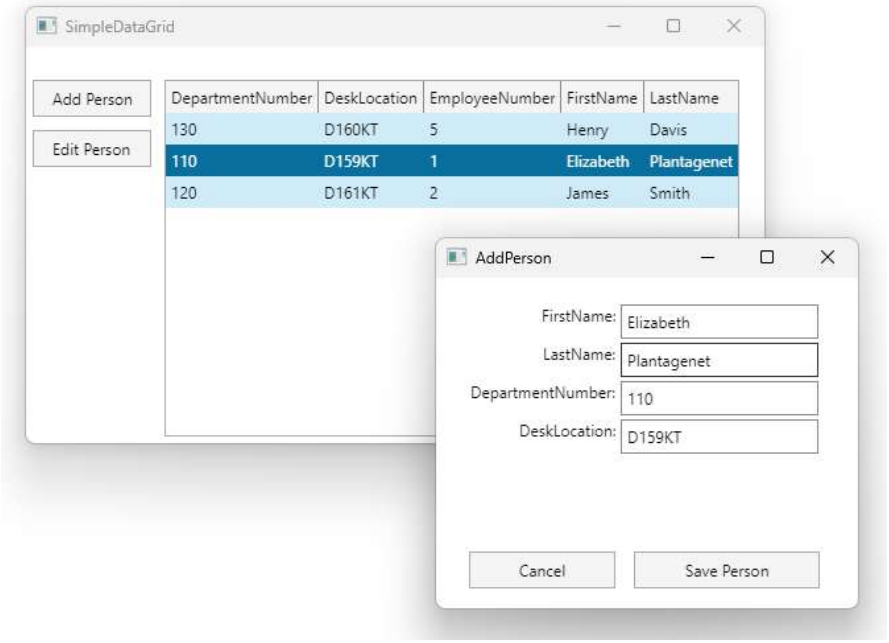


Рисунок 7.20 – Приклад роботи доданої функції редагування обраного запису

В файлі «MainWindow.axaml.cs» зробимо прив’язку до нової кнопки та викличемо функцію-обробник натискання на кнопку «Delete Person»:

```
var bDeletePerson = this.FindControl<Button>("bDeletePerson");
bDeletePerson.Click += DeletePerson_Click;
```

Функція видалення має наступний вміст:

```
private void DeletePerson_Click(object? sender, RoutedEventArgs e)
{
      if (SelectPerson != null)
      {
            string sql = String.Format("Delete from Person where "+
                  "EmployeeNumber = ‘{0}’", SelectPerson.EmployeeNumber);
            Database.Exec_SQL(sql);
            SelectPerson = null;
            ThePeople.FillPeople();
      }
}
```

В даній функції спочатку створюється SQL-запит для видалення інформації з БД за номером первинного ключа. Далі викликається функція Exec_SQL із допоміжного класу Database. Після видалення даних, очищається змінна «SelectPerson» та заново виводиться на екран оновлений список працівників з бази даних.

Повністю код прикладу подано в лістингу «Listing_7-3».

7.3 Контрольні запитання та завдання

1. Що таке Avalonia DataGrid і для чого він використовується?
2. Як створити діалог користувача для додавання даних у колекцію в Avalonia?
3. Які кроки необхідно виконати для створення та підключення до бази даних у Avalonia?
4. Як реалізувати функцію додавання інформації до бази даних у Avalonia?
5. Як реалізувати функцію редагування виділеної інформації в базі даних у Avalonia?
6. Як реалізувати функцію видалення інформації з бази даних у Avalonia?
7. Які основні властивості та методи DataGrid ви знаєте в Avalonia?
8. Як налаштувати DataGrid для відображення даних з бази даних у Avalonia?
9. Які підходи можна використовувати для забезпечення інтерактивності DataGrid у Avalonia?

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ

1. Невлюдов І. Ш. Застосування цифрових двійників технічних засобів автоматизації для розроблення програмно-технічних комплексів АСУ ТП : Навчальний посібник / І. Ш. Невлюдов, С. П. Новоселов, О. В. Сичова. – Харків: Видавництво Іванченка І. С., 2023. – 267 с. ISBN 978-617-8059-95-8, DOI: 10.30837/978-617-8059-95-8.

2. Невлюдов І. Ш. Технологія програмування промислових контролерів в інтегрованому середовищі CODESYS : навч. посіб. / І. Ш. Невлюдов, С. П. Новоселов, О. В. Сичова ; М-во освіти і науки України, Харків. нац. ун-т радіоелектроніки. – Харків : ХНУРЕ, 2019. – 264 с. : іл. –ISBN 978-966-659-265-4. DOI: 10.30837/978-966-659-265-4.

3. Alessandro Del Sole. Xamarin with Visual Studio: Launch your mobile development career by creating Android and iOS applications using .NET and C# (English Edition). – BPB Publications, 2022. – 468 p. – ISBN 9355511876, 9789355511874.

4. Щербаков О.В. Основи об’єктно-орієнтованого програмування [Електронний ресурс] : навчальний посібник / О.В. Щербаков, Ю.Е. Парфьонов, В.М. Федорченко. – Харків : ХНЕУ ім. С. Кузнеця, 2019. – 237 с. – ISBN 978-966-676-759-5.

5. Model-View-ViewModel [Електронний ресурс] // Вікіпедія – Режим доступу: https://uk.wikipedia.org/wiki/Model-View-ViewModel. – Станом на 17.04.2024. – Назва з екрану.

6. How-To Guides [Електронний ресурс] // Avalonia – Режим доступу: https://docs.avaloniaui.net/docs/guides/. – Станом на 17.04.2024. – Назва з екрану.

7. Gidon - Avalonia based MVVM Plugin IoC Container [Електронний ресурс] // Code Project. For those who code. – Режим доступу: https://www.codeproject.com/Articles/5325733/Gidon-Avalonia-based-MVVM-Plugin-IoC-Container. – Станом на 17.04.2024. – Назва з екрану.

8. Коноваленко І.В., Марущак П.О., Савків В.Б. Програмування мовою C# 7.0.Тернопіль: Тернопільський національний технічний унверситет ім. Івана Пулюя, 2017. – 300 с. – ISBN 978-966-305-085-0.

9. Alignment, Margins and Padding [Електронний ресурс] // Avalonia – Режим доступу: https://docs.avaloniaui.net/docs/layout/alignment-margins-and-padding. – Станом на 17.04.2024. – Назва з екрану.

10. Data Binding [Електронний ресурс] // Avalonia – Режим доступу: https://docs.avaloniaui.net/docs/data-binding. – Станом на 17.04.2024. – Назва з екрану.

11. How To Bind from Code [Електронний ресурс] // Avalonia – Режим доступу: https://docs.avaloniaui.net/docs/data-binding/binding-from-code. – Станом на 17.04.2024. – Назва з екрану.

12. Avalonia UI [Електронний ресурс] // Youtube – Режим доступу: https://www.youtube.com/@avaloniaui/featured. – Станом на 17.04.2024. – Назва з екрану.

13. Avalonia FAQ [Електронний ресурс] // Avalonia – Режим доступу: https://avaloniaui.net/FAQ. – Станом на 17.04.2024. – Назва з екрану.

14. Toggle Button [Електронний ресурс] // Avalonia – Режим доступу: https://docs.avaloniaui.net/docs/controls/togglebutton. – Станом на 17.04.2024. – Назва з екрану.

15. Corner Radius Struct [Електронний ресурс] // Avalonia – Режим доступу: http://reference.avaloniaui.net/api/Avalonia/CornerRadius/. – Станом на 17.04.2024. – Назва з екрану.

16. Inherit styles in avalonia [Електронний ресурс] // Code Project. For those who code. – Режим доступу: https://www.codeproject.com/Questions/5327287/Inherit-styles-in-avalonia. – Станом на 17.04.2024. – Назва з екрану.

17. Multiplatform Avalonia .NET Framework Programming Advanced Concepts in Easy Samples [Електронний ресурс] // Code Project. For those who code. – Режим доступу: https://www.codeproject.com/Articles/5317055/Multiplatform-Avalonia-NET-Framework-Programming-A. – Станом на 17.04.2024. – Назва з екрану.

18. Avalonia UI [Електронний ресурс] // Let’s build from here The world’s leading AI-powered developer platform. – Режим доступу: https://github.com/AvaloniaUI/. – Станом на 17.04.2024. – Назва з екрану.

19. Multiplatform UI Coding with AvaloniaUI in Easy Samples. Part 1 - AvaloniaUI Building Blocks [Електронний ресурс] // CodeProject. For those who code. – Режим доступу: https://www.codeproject.com/Articles/5308645/Multiplatform-UI-Coding-with-AvaloniaUI-in-Easy-Sa. – Станом на 11.06.2024. – Назва з екрану.

20. Multiplatform Avalonia .NET Framework Programming Basic Concepts in Easy Samples [Електронний ресурс] // CodeProject. For those who code. – Режим доступу: https://www.codeproject.com/Articles/5311995/Multiplatform-Avalonia-NET-Framework-Programming-B. – Станом на 11.06.2024. – Назва з екрану.

21. Multiplatform XAML/C# Miracle Package: Avalonia. Comparing Avalonia to WinUI based Solutions [Електронний ресурс] // CodeProject. For those who code. – Режим доступу: https://www.codeproject.com/Articles/5366945/Multiplatform-XAML-Csharp-Miracle-Package-Avalonia. – Станом на 11.06.2024. – Назва з екрану.

22. Multiplatform Avalonia .NET Framework Programming Advanced Concepts in Easy Samples [Електронний ресурс] // CodeProject. For those who code. – Режим доступу: https://www.codeproject.com/Articles/5317055/Multiplatform-Avalonia-NET-Framework-Programming-A. – Станом на 11.06.2024. – Назва з екрану.

Рекомендуємо інші наші книги:

Застосування цифрових двійників технічних засобів автоматизації для розроблення програмно-технічних комплексів АСУ ТП : Навчальний посібник / І.Ш. Невлюдов, С.П. Новоселов, О.В. Сичова. – Харків: Видавництво Іванченка І. С., 2023. – 267 с.

ISBN 978-617-8059-95-8
DOI: 10.30837/978-617-8059-95-8

У навчальному посібнику розглядаються питання практичного застосовування віртуальних макетів технічних засобів автоматизації, які використовуються для розроблення програмно-технічних комплексів сучасних АСУ ТП. Наведено приклади найпоширеніших технічних засобів автоматизації технологічних процесів і реалізації керування ними за допомогою релейно-контактної логіки.

Технології Інтернету речей в управлінні пристроями на мікроконтролерах: Навчальний посібник [Електронний ресурс] / І.Ш. Невлюдов, В.А. Андрусевич, С.П. Новоселов, О.Г. Резніченко. – Електронне видання. – Харків: ХНУРЕ, 2023. – 214 с. – pdf 2,88 Mb.

ISBN 978-966-659-364-4
DOI: 10.30837/978-966-659-364-4

У навчальному посібнику подані відомості про сучасну концепцію Інтернету речей, що дозволяє здійснювати передачу та обмін даними між фізичним світом і комп'ютерними системами в автоматичному режимі, за допомогою використання стандартних протоколів зв'язку. На реальних прикладах розглянуто процес підключення, моніторингу датчиків, віддаленого управління інтелектуальними виконавчими пристроями через Інтернет.

Навчальне видання

НОВОСЕЛОВ Сергій Павлович

СИЧОВА Оксана Володимирівна

ОСНОВИ РОЗРОБКИ КРОСПЛАТФОРМНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ НА AVALONIA

Навчальний посібник

Підписано до друку 12.06.2024. Формат 60x84/16
Умовн. друк. арк. 15,5. Наклад 200 прим. Зам. № 12-06.

Видавництво і друк ФОП Іванченко І. С.
пр. Тракторобудівників, 89-а/62, м. Харків, 61135
тел.: +38(050/093)4024350
Свідоцтво про внесення суб'єкта видавничої справи до державного реєстру видавців, виготівників та розповсюджувачів видавничої продукції ДК № 4388 від 15.08.2012 р.
www.monograf.com.ua