Л13. Шаблон проектування MVVM

Патерн проектування *— це типовий спосіб вирішення певної проблеми, що часто зустрічається при проектуванні архітектури програм. На відміну від готових функцій чи бібліотек, патерн не можна просто взяти й скопіювати в програму. Патерн являє собою не якийсь конкретний код, а загальний принцип вирішення певної проблеми, який майже завжди треба підлаштовувати для потреб тієї чи іншої програми.*

Патерни часто плутають з алгоритмами, адже обидва поняття описують типові рішення відомих проблем. Але якщо алгоритм — це чіткий набір дій, то ***патерн*** *— це високорівневий опис рішення, реалізація якого може відрізнятися у двох різних програмах.*

Якщо провести аналогії, то алгоритм — це кулінарний рецепт з чіткими кроками, а ***патерн*** *— інженерне креслення, на якому намальовано рішення без конкретних кроків його отримання.*

Описи патернів зазвичай дуже формальні й найчастіше складаються з таких пунктів:

- *проблема, яку вирішує патерн;*
- *мотивація щодо вирішення проблеми способом, який пропонує патерн;*
- *структура класів, складових рішення;*
- *приклад однією з мов програмування;*
- *особливості реалізації в різних контекстах;*
- *зв’язки з іншими патернами.*

Л13. Шаблон проектування MVVM

Навіщо ж знати патерни?.

Перевірені рішення. *Витрачається менше часу, використовуючи готові рішення, замість повторного винаходу велосипеда. До деяких рішень ви могли б дійти й самотужки, але багато які з них стануть для вас відкриттям.*

Стандартизація коду. *Робиться менше прорахунків при проектуванні, використовуючи типові уніфіковані рішення, оскільки всі приховані в них проблеми вже давно знайдено.*

Загальний словник програмістів. *Ви вимовляєте назву патерна, замість того, щоб годину пояснювати іншим програмістам, який крутий дизайн ви придумали і які класи для цього потрібні.*

Патерн MVVM *(Model - View - ViewModel) ґрунтується на розділенні функціональної частини програми на три ключові компоненти:*

View *- це графічний інтерфейс, тобто вікно, кнопки тощо;*

Model *- модель або дані, що використовуються у додатку;*

ViewModel *- проміжний шар між поданням та даними, що забезпечує їх взаємодію.*

Перевагою використання даного патерну є менша зв'язаність між компонентами та поділ відповідальності між ними. Тобто ***Model*** *відповідає за дані,* ***View*** *відповідає за графічний інтерфейс, а* ***ViewMode****l – за логіку програми.*

Л13. Шаблон проектування MVVM

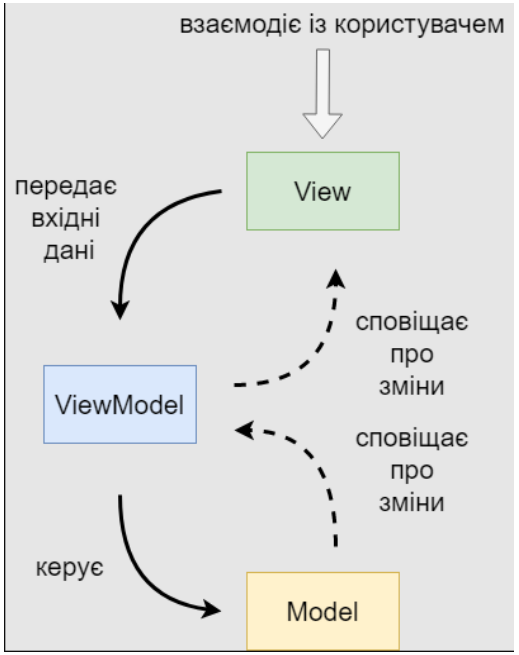


Рис.1. Діаграма взаємодії між компонентами шаблону MVVM

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. MVVM була створена з метою поділу праці дизайнера і програміста.

Архітектура MVVM вирішує цю проблему ясним поділом відповідальності:

1) Розробка користувацького інтерфейсу здійснюється дизайнером інтерфейсів за допомогою технології, більш-менш природної для такої роботи (XML).

2) Логіка користувацького інтерфейсу реалізується розробником як компонент ViewModel.

3) Функціональні зв'язки між користувацьким інтерфейсом 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).*

Л13. Шаблон проектування MVVM

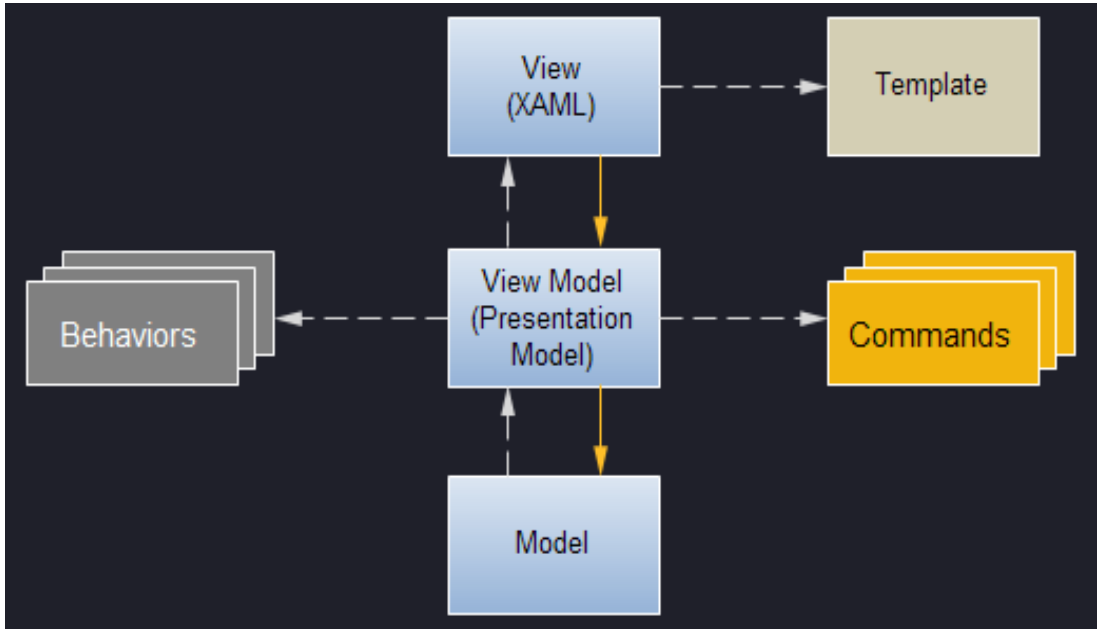


Рис. 2. Діаграма реалізації шаблону MVVM

Розглянемо найпростіший приклад. Визначимо клас даних чи моделі:

```
public class Phone
{
    public string Title { get; set; }
    public string Company { get; set; }
    public int Price { get; set; }
}
```

Л13. Шаблон проектування MVVM

Додамо в проект клас PhoneViewModel з таким вмістом:

```
using System.ComponentModel;
public class PhoneViewModel :INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;
    private Phone phone;

    public PhoneViewModel()
    {
        phone = new Phone();
    }

    public string Title
    {
        get { return phone.Title; }
        set
        {
            if (phone.Title != value)
            {
                phone.Title = value;
                OnPropertyChanged("Title");
            }
        }
    }
```

```
public string Company
{
    get { return phone.Company; }
    set
    {
        if (phone.Company != value)
        {
            phone.Company = value;
            OnPropertyChanged("Company");
        }
    }
}
```

Л13. Шаблон проектування MVVM

```
public int Price
{
    get { return phone.Price; }
    set
    {
        if (phone.Price != value)
        {
            phone.Price = value;
            OnPropertyChanged("Price");
        }
    }
}
```

```
    protected void OnPropertyChanged(string propName)
    {
        if (PropertyChanged != null)
            PropertyChanged(this, new PropertyChangedEventArgs(propName));
    }
}
```

*Це і буде компонент **ViewModel**, який пов'язує дані та візуальний інтерфейс. Фактично ViewModel представляє обгортку над класом Phone, визначаючи ті самі властивості. Для спрощення завдання сам об'єкт Phone створюється в конструкторі, хоча можна було отримання об'єкта з бази даних.*

*Важливо, що цей клас реалізує інтерфейс **INotifyPropertyChanged**, який дозволяє повідомляти систему про зміну його властивостей за допомогою події **PropertyChanged***

Л13. Шаблон проектування MVVM

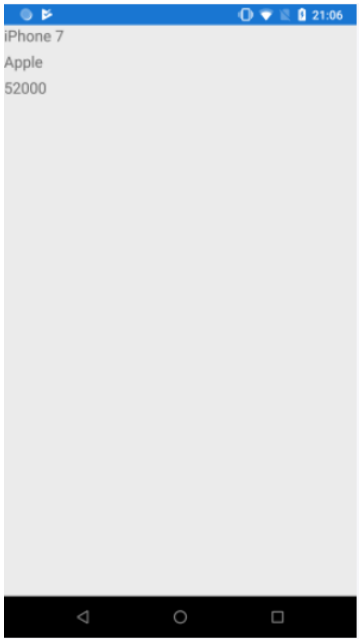
Створимо візуальну частину. Визначимо на головній сторінці MainPage.xaml такий вміст:

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="HelloApp.MainPage">
  <StackLayout>
    <Label Text="{Binding Title}" FontSize="Medium" />
    <Label Text="{Binding Company}" FontSize="Medium" />
    <Label Text="{Binding Price}" FontSize="Medium" />
  </StackLayout>
</ ContentPage >
```

Тут визначено прив'язку до властивості ViewModel

У конструкторі сторінки у файлі коду c# пропишемо як контекст даних для сторінки визначену раніше ViewModel

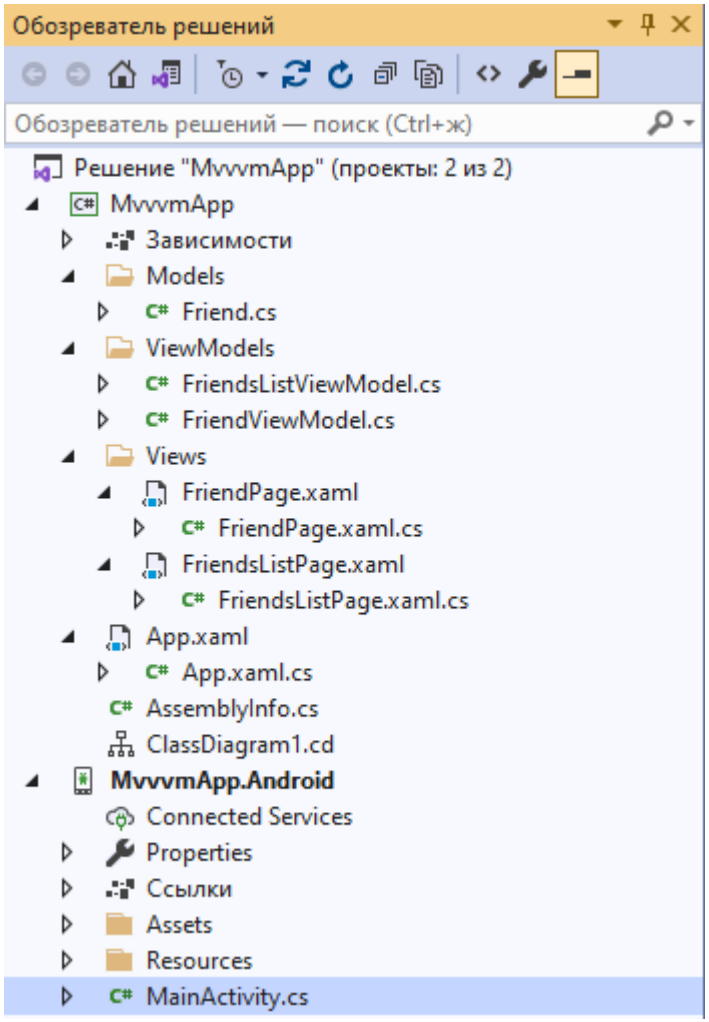
```
public partial class MainPage : ContentPage
{
    public MainPage()
    {
        InitializeComponent();
        this.BindingContext = new PhoneViewModel
        {
            Title = "iPhone 7",
            Company = "Apple",
            Price=52000
        };
    }
}
```



Л13. Приклад MVVM

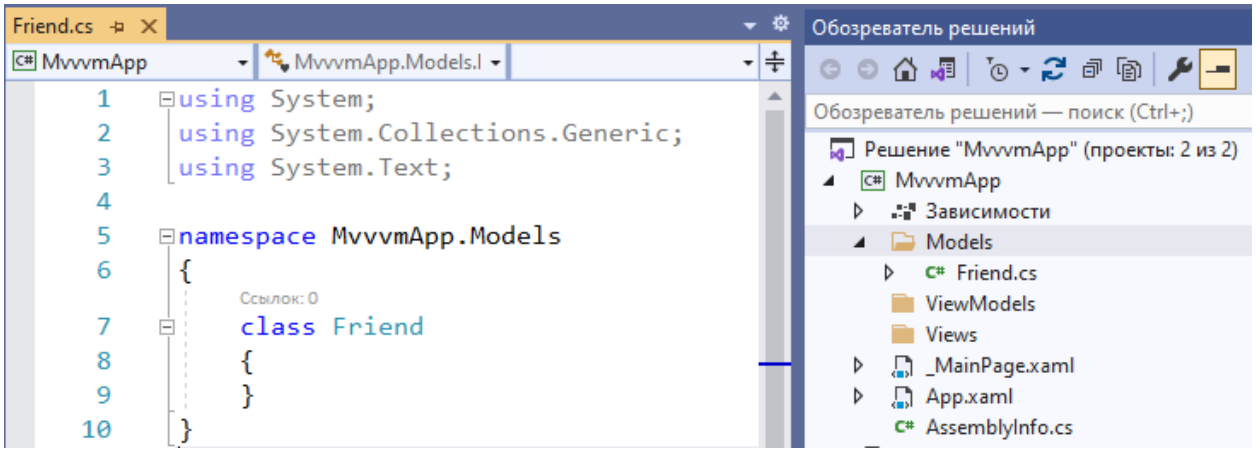
*Створимо нову програму **MvvmApp**, яка керує списком друзів (створює список друзів, додає їх, редагує список та видаляє зі списку).*

Додаємо до головного проекту три нові папки для кожного з компонентів патерну: Models, ViewModels, Views та видаляємо сторінку MainPage.xaml ().



Л13. Приклад MVVM

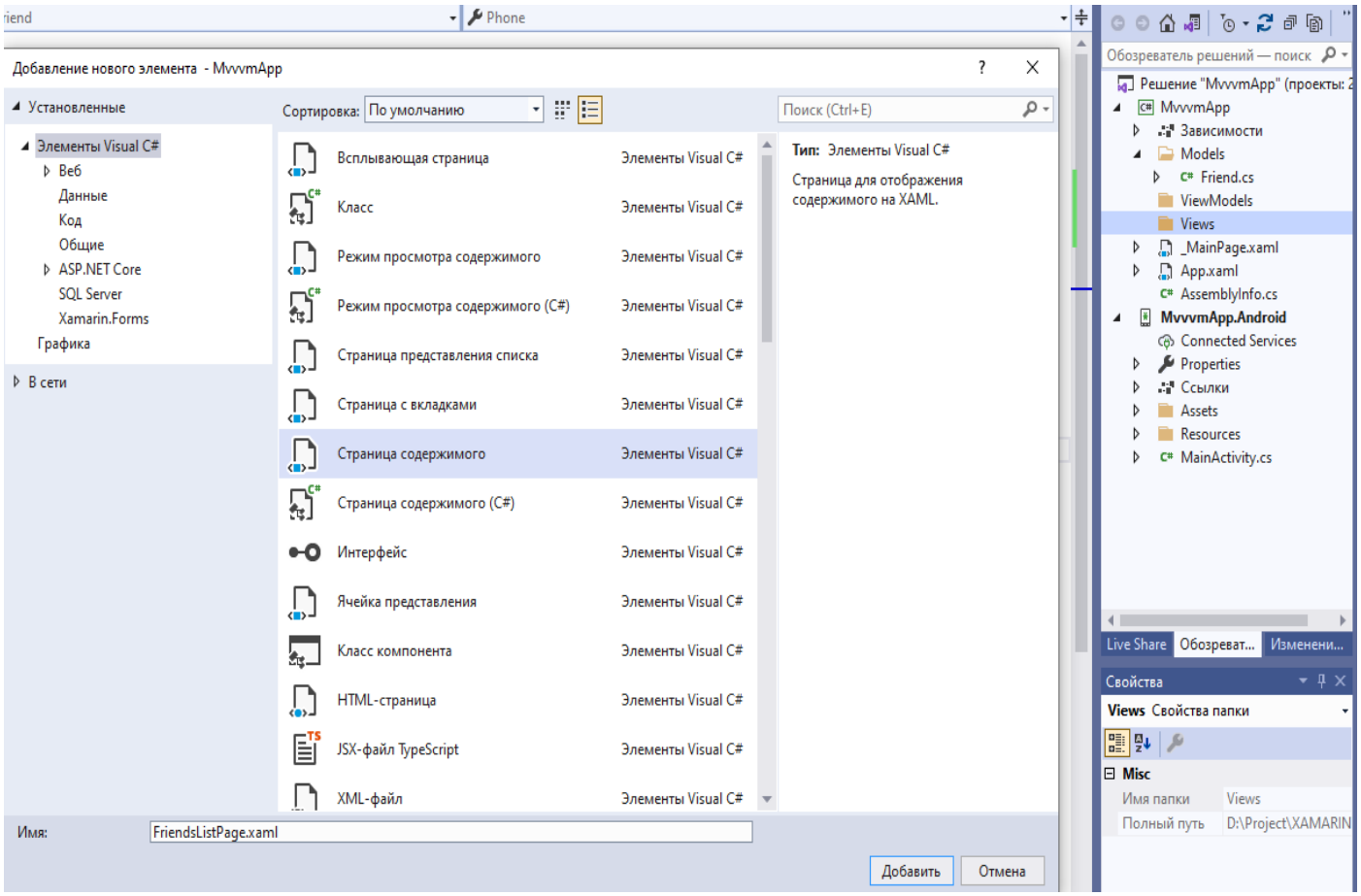
*До папки **Models**, яка представлятиме дані, додаємо **клас Friend***




```
namespace MvvvmApp.Models
{
    class Friend
    {
        public string Name { get; set; }
        public string Email { get; set; }
        public string Phone { get; set; }
    }
}
```

*Щоб працювати з даними цієї моделі додамо до папки Views дві сторінки XAML за типом Content Page : **FriendsListPage.xaml** (для виведення списку друзів) і **FriendPage.xaml** (для керування одним другом).*

Л13. Приклад MVVM



Додавання сторінок типу Content Page до папки Views

*Додаємо до папки **ViewModels** клас **FriendViewModel**, який є надбудовою над об'єктом **Friend**, та редагуємо його код:*

Л13. Приклад MVVM

*Додаємо до папки **ViewModels** клас **FriendViewModel**, який є надбудовою над об'єктом **Friend**, та редагуємо його код:*

```
using System.ComponentModel;
using MvvvmApp.Models;

namespace MvvvmApp.ViewModels
{
    public class FriendViewModel : INotifyPropertyChanged
    {
        public event PropertyChangedEventHandler PropertyChanged;
        // Подія PropertyChanged вказує, що властивості об’єкта змінилися
        FriendsListViewModel lvm;

       Friend Friend { get; set; }

        public FriendViewModel()
        {
            Friend = new Friend();
        }

        public FriendsListViewModel ListViewModel
        {
            get { return lvm; }
            set
            {
                if (lvm != value)
                {
                    lvm = value;
                    OnPropertyChanged("ListViewModel");
                }
            }
        }
```

Л13. Приклад MVVM

```
public string Name
{
    get { return Friend.Name; }
    set
    {
        if (Friend.Name != value)
        {
            Friend.Name = value;
            OnPropertyChanged("Name");
        }
    }
}
public string Email
{
    get { return Friend.Email; }
    set
    {
        if (Friend.Email != value)
        {
            Friend.Email = value;
            OnPropertyChanged("Email");
        }
    }
}
```

```
public string Phone
{
    get { return Friend.Phone; }
    set
    {
        if (Friend.Phone != value)
        {
            Friend.Phone = value;
            OnPropertyChanged("Phone");
        }
    }
}

public bool IsValid
{
    get
    {
        return ((!string.IsNullOrEmpty(Name.Trim())) ||
            (!string.IsNullOrEmpty(Phone.Trim())) ||
            (!string.IsNullOrEmpty(Email.Trim())));
    }
}
```

```
protected void OnPropertyChanged(string propName)
{
    if (PropertyChanged != null)
        PropertyChanged(this, new PropertyChangedEventArgs(propName));
}
```

Л13. Приклад MVVM

*Додаємо до папки **ViewModels** клас **FriendsListViewModel**, представлятиме список друзів й який буде використовуватись на сторінці FriendsListPage:*

```
using System.Collections.ObjectModel;
using System.Windows.Input;
using Xamarin.Forms;
using System.ComponentModel;
using MvvvmApp.Views;

namespace MvvvmApp.ViewModels
{
    public partial class FriendsListViewModel : INotifyPropertyChanged
    {
        public ObservableCollection<FriendViewModel> Friends { get; set; }

        public event PropertyChangedEventHandler PropertyChanged;

        public ICommand CreateFriendCommand { protected set; get; }
        public ICommand DeleteFriendCommand { protected set; get; }
        public ICommand SaveFriendCommand { protected set; get; }
        public ICommand BackCommand { protected set; get; }
        FriendViewModel selectedFriend;

        public INavigation Navigation { get; set; }

        public FriendsListViewModel()
        {
            Friends = new ObservableCollection<FriendViewModel>();
            CreateFriendCommand = new Command(CreateFriend);
            DeleteFriendCommand = new Command(DeleteFriend);
            SaveFriendCommand = new Command(SaveFriend);
            BackCommand = new Command(Back);
        }
```

Л13. Приклад MVVM

```
public FriendViewModel SelectedFriend
{
    get { return selectedFriend; }
    set
    {
        if (selectedFriend != value)
        {
            FriendViewModel tempFriend = value;
            selectedFriend = null;
            OnPropertyChanged("SelectedFriend");
            Navigation.PushAsync(new FriendPage(tempFriend));
        }
    }
}
protected void OnPropertyChanged(string propName)
{
    if (PropertyChanged != null)
        PropertyChanged(this, new PropertyChangedEventArgs(propName));
}

private void CreateFriend()
{
    Navigation.PushAsync(new FriendPage(new FriendViewModel()
      {
         ListViewModel = this }));
}
private void Back()
{
    Navigation.PopAsync();
}
```

Л13. Приклад MVVM

```
private void SaveFriend(object friendObject)
 {
     FriendViewModel friend = friendObject as FriendViewModel;
     if (friend != null && friend.IsValid && !Friends.Contains(friend))
     {
         Friends.Add(friend);
     }
     Back();
 }
 private void DeleteFriend(object friendObject)
 {
     FriendViewModel friend = friendObject as FriendViewModel;
     if (friend != null)
     {
         Friends.Remove(friend);
     }
     Back();
 }
```

Для збереження списку друзів, який виводиться на сторінку, у класі FriendsListViewModel визначено ***колекцію Friends****.*

Навігація також є частиною логіки програми, яка не є візуальною частиною, тому у коді визначено властивість ***Navigation****. Через цю властивість здійснюватиметься перехід до* ***FriendPage****. Для керування списком друзів у класі визначено чотири команди. Команда додавання нового друга - метод* ***CreateFriend()****, у якому здійснюється перехід до FriendPage. Конструктор FriendPage передає поточний об'єкт FriendViewModel, який далі створюється. По команді повернення назад виконується метод* ***Back()****, який здійснює перехід назад. Команда збереження об'єкта виконує метод* ***SaveFriend()****. У цьому методі новий об'єкт додається до колекції Friends. За командою видалення викликається метод* ***DeleteFriend*** *видалення об'єкта зі списку.*

Л13. Приклад MVVM

Змінимо код сторінок із папки Views. У коді C# сторінки ***FriendsListPage*** *створюється об'єкт* ***FriendsListViewModel****, який встановлюється як контекст сторінки.*

```
using Xamarin.Forms;
using Xamarin.Forms.Xaml;
using MvvvmApp.ViewModels;

namespace MvvvmApp.Views
{
    [XamlCompilation(XamlCompilationOptions.Compile)]
    public partial class FriendsListPage : ContentPage
    {
        public FriendsListPage()
        {
            InitializeComponent();
            BindingContext = new FriendsListViewModel() { Navigation = this.Navigation };
        }
    }
}
```

Л13. Приклад MVVM

У коді XAML у цій сторінці пропишемо вирази прив'язки до цього об'єкта:

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="MvvvmApp.Views.FriendsListPage" Title="Список друзів">
    <StackLayout>
        <Button Text="Додати" Command="{Binding CreateFriendCommand}" />
        <ListView x:Name="booksList" ItemsSource="{Binding Friends}"
              SelectedItem="{Binding SelectedFriend, Mode=TwoWay}" HasUnevenRows="True">
            <ListView.ItemTemplate>
                <DataTemplate>
                    <ViewCell>
                        <ViewCell.View>
                            <StackLayout>
                                <Label Text="{Binding Name}" FontSize="Medium" />
                                <Label Text="{Binding Email}" FontSize="Small" />
                                <Label Text="{Binding Phone}" FontSize="Small" />
                            </StackLayout>
                        </ViewCell.View>
                    </ViewCell>
                </DataTemplate>
            </ListView.ItemTemplate>
        </ListView>
    </StackLayout>
</ContentPage>
```

*Кнопка для додавання нового об'єкта не містить жодних обробників, тому що всю обробку буде виконувати команда **CreateFriendCommand**, яка визначена в **FriendsListViewModel**.*

Л13. Приклад MVVM

*Змінимо код c# на сторінці **FriendPage**:*

```
using Xamarin.Forms;
using Xamarin.Forms.Xaml;
using MvvvmApp.ViewModels;

namespace MvvvmApp.Views
{
    [XamlCompilation(XamlCompilationOptions.Compile)]
    public partial class FriendPage : ContentPage
    {
        public FriendViewModel ViewModel { get; private set; }
        public FriendPage(FriendViewModel vm)
        {
            InitializeComponent();
            ViewModel = vm;
            this.BindingContext = ViewModel;
        }
    }
}
```

Тепер сторінка як параметр у конструкторі приймає об'єкт FriendViewModel, встановлює його як контекст і також не містить жодної іншої логіки.

Л13. Приклад MVVM

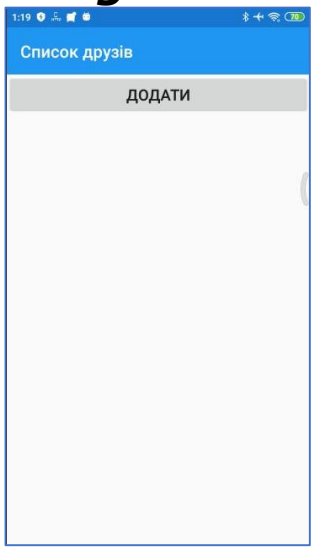
Змінимо код c# на сторінці ***FriendPage****:*

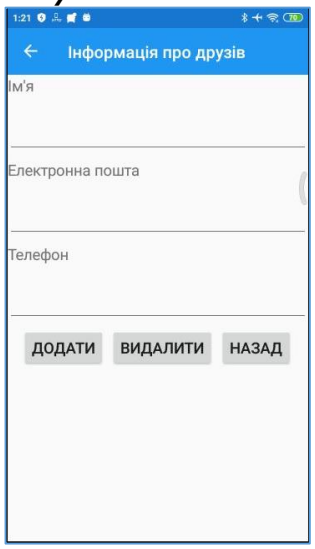
```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="MvvmApp.Views.FriendPage" Title="Інформація про друзів">
    <StackLayout>
        <StackLayout x:Name="friendStack">
            <Label Text="Ім'я" />
            <Entry Text="{Binding Name}" FontSize="Medium" />
            <Label Text="Електронна пошта" />
            <Entry Text="{Binding Email}" FontSize="Medium" />
            <Label Text="Телефон" />
            <Entry Text="{Binding Phone}" FontSize="Medium" />
        </StackLayout>
        <StackLayout Orientation="Horizontal" HorizontalOptions="CenterAndExpand">
            <Button Text="Додати" Command="{Binding ListViewModel.SaveFriendCommand}"
CommandParameter="{Binding}" />
            <Button Text="Видалити" Command="{Binding ListViewModel.DeleteFriendCommand}"
CommandParameter="{Binding}" />
            <Button Text="Назад" Command="{Binding Path=ListViewModel.BackCommand}" />
        </StackLayout>
    </StackLayout>
</ContentPage>
```

При натисканні на кнопки буде викликатись відповідна команда. Дві кнопки (Додати, Видалити) передають у команди параметр, який представляє ***прив'язаний об'єкт*** *- тобто той об'єкт FriendViewModel, який передано в конструктор й встановлений як контекст сторінки. Вираз* ***{Binding}*** *без вказівки якості об'єкта виконує прив'язку до об'єкта, який є контекстом для даного елемента управління. Таким чином, команди в FriendsListViewModel отримають об'єкт FriendViewModel, що додається або видаляється.*

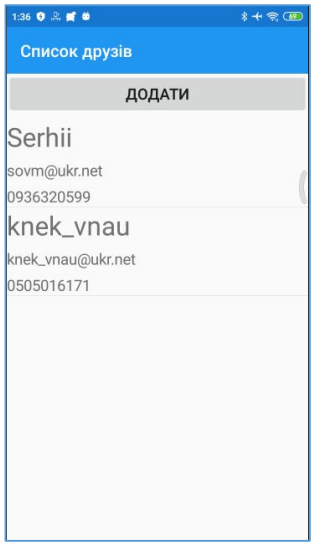
Л13. Приклад MVVM

*У класі App встановимо сторінку **FriendsListPage** як головну:*

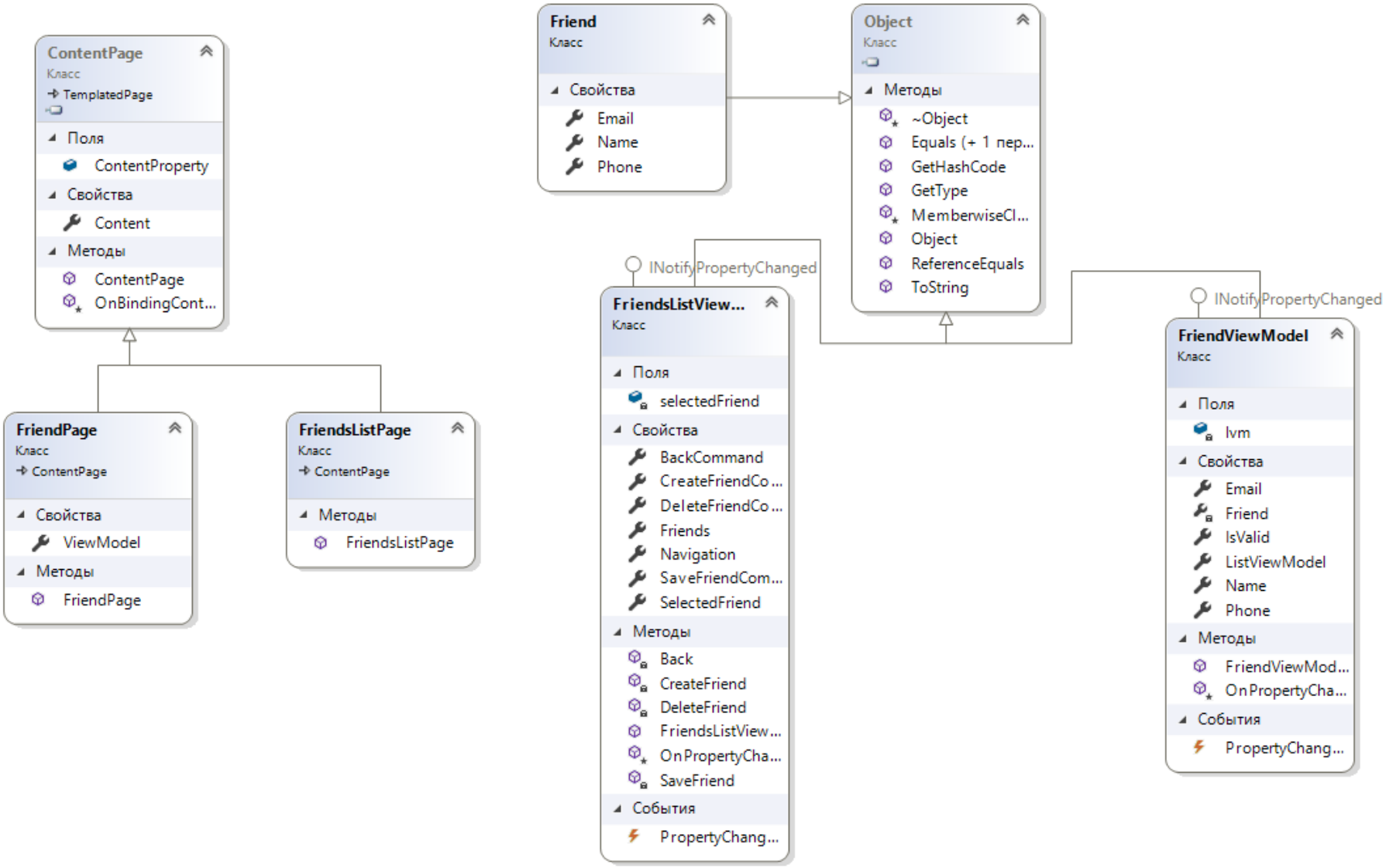





```
using System;
using Xamarin.Forms;
using Xamarin.Forms.Xaml;
using MvvvmApp.Views;

namespace MvvvmApp
{
    public partial class App : Application
    {
        public App()
        {
            InitializeComponent();
            MainPage = new NavigationPage(new FriendsListPage());
        }

        protected override void OnStart()
        {
        }

        protected override void OnSleep()
        {
        }

        protected override void OnResume()
        {
        }
    }
}
```



Л13. Приклад MVVM



Л13. Особливості тестування додатків на мобільних пристроях

Користувач мобільного пристрою очікує, що додатки, які встановлюються, прості, інтуїтивно зрозумілі, працюють завжди й всюди без збоїв. Якщо очікування не виправдовуються, то користувач попросту видаляє додаток та встановлює аналогічний додаток від іншого розробника, яких у сфері мобільних розробок завжди достатньо. Тому ***якість програми*** *є одним з головних факторів його популярності.*

Тестування додатків на мобільних пристроях в цілому відповідає загальним принципам тестування, однак, є ряд особливостей, які характерні саме для тестування мобільних додатків.

1) Розмір екрана та touch-інтерфейс:

- *розміри всіх елементів графічного інтерфейсу користувача;*
- *перевірка можливості використання усіх активних елементів (кнопок, посилань та ін.);*
- *швидкість відгуку активних елементів повинна бути досить високою;*
- *перевірка того, що багаторазове швидке натискання на кнопку не викличе екстрене завершення програми;*
- *підтримка мультитач – одночасне натискання кількох кнопок;*
- *підтримка горизонтального (landscape) та вертикального (portrait) положення;*
- *перевірка використання в додатку спеціальних жестів (double tap, swipe, pinch in/out та ін.).*

Л13. Особливості тестування додатків на мобільних пристроях

2) Ресурси телефону:

- *необхідно проконтролювати можливі витоки пам'яті. Часто це трапляється в додатках з вікнами, що містять велику кількість інформації. Також витоки пам'яті можуть бути присутніми під час тривалої роботи програми;*
- *перевірка обробки ситуацій нестачі пам'яті для функціонування операційної системи, під час роботи програми в активному та фоновому режимах;*
- *нестача місця для установки або роботи програми;*
- *установка на SD-карту;*
- *перевірка роботи батареї (акумулятора) пристрою при запущеному додатку, роботи у фоновому режимі, при включеному Wi-Fi, 4G Інтернеті, без підключення до мережі та ін.*

Л13. Особливості тестування додатків на мобільних пристроях

3) Різні роздільні здатності екрану та версії ОС:

- *необхідно перевірити роботу програми на пристроях з різними роздільними здатностями екрану. На екранах з високою роздільною здатністю елементи інтерфейсу та текст відображаються дрібніше, при роботі додатка на пристрої з екраном більш низькою роздільною здатністю – елементи інтерфейсу можуть стати занадто великими;*
- *необхідно переконатися, що додаток не може бути встановлений на непідтримуваному пристрої. При цьому обов'язково тестування програми на усіх заявлених підтримуваних пристроях.*

4) Реакція програми на зовнішні переривання:

- *вхідні та вихідні SMS та MMS;*
- *вхідні та вихідні дзвінки;*
- *нагадування, будильник та ін.;*
- *відключення та підключення SD-карти;*

Л13. Особливості тестування додатків на мобільних пристроях

4) Реакція програми на зовнішні переривання:

- *відключення та підключення Wi-Fi. Наприклад, додаток може екстрено завершувати свою роботу під час запуску авіарежиму при відключеному Wi-Fi. При втраті сигналу та одночасному виконанні операції може відображатися нескінченне завантаження програми, замість коректного повідомлення про відсутність інтернет-підключення;часто виникають проблеми з переходом в онлайн-режим після офлайн режиму;*
- *відключення/підключення мобільного пристрою до зарядного пристрою;*
- *робота з фізичною клавіатурою;*
- *робота в фоновому (background) режимі. Для відправки у фоновий режим – запускається додаток, а потім натискається кнопка home. В результаті при повторному запуску програми бувають помилки, екстрені завершення роботи, неправильне відображення останнього відкритого вікна. Також при тривалій бездіяльності додатка виникає екстрене завершення роботи або помилки;*
- *робота додатка після виходу зі сплячого режиму;*
- *сумісність з іншими додатками.*