

# Xamarin: How to survive Code Sharing

Francesco Bonacci

Xamarin Certified Mobile Developer  
Consultant at 

[francesco.bonacci@outlook.com](mailto:francesco.bonacci@outlook.com)

Twitter: [@francedot](https://twitter.com/francedot)

[github.com/francedot](https://github.com/francedot)

# Agenda

1. PCL vs Shared Project vs .NET Standard
2. MVVM
3. Major Pattern in Xamarin
4. Prism



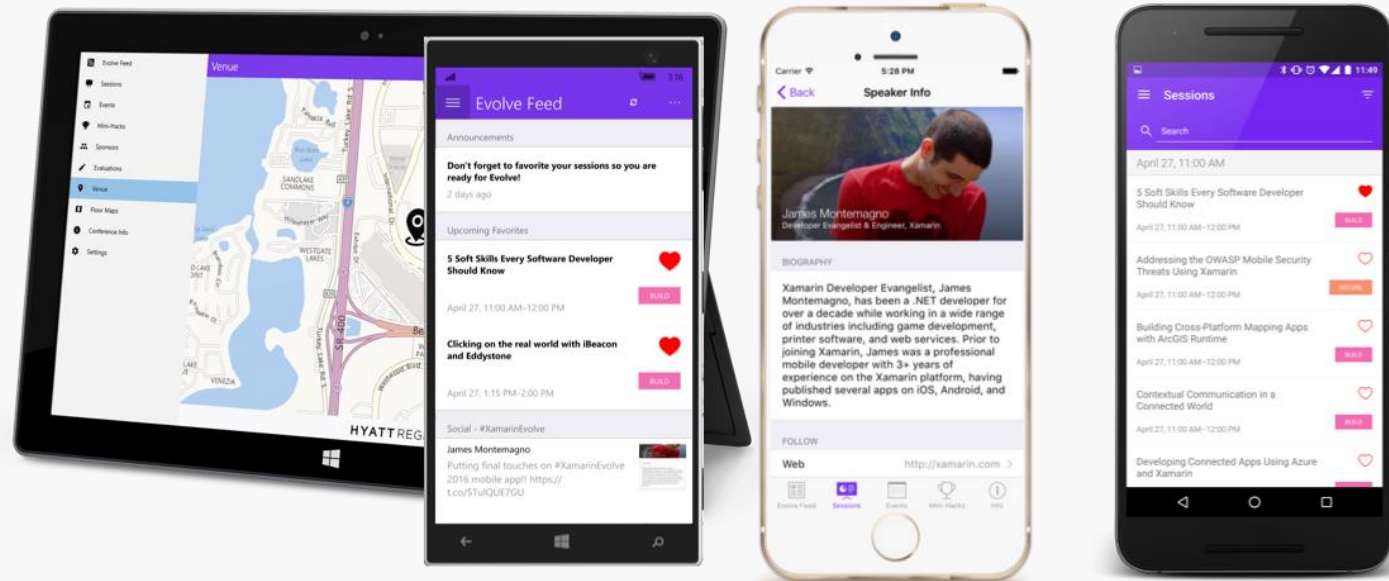
# Code Sharing

# Code Sharing in Xamarin

Uno tra le principali utilità di Xamarin è la possibilità di condividere porzioni di codice tra piattaforme diverse

Non sempre facile o possibile fattorizzare ad un'unica codebase

[github.com/xamarin/hq/app-evolve](https://github.com/xamarin/hq/app-evolve)



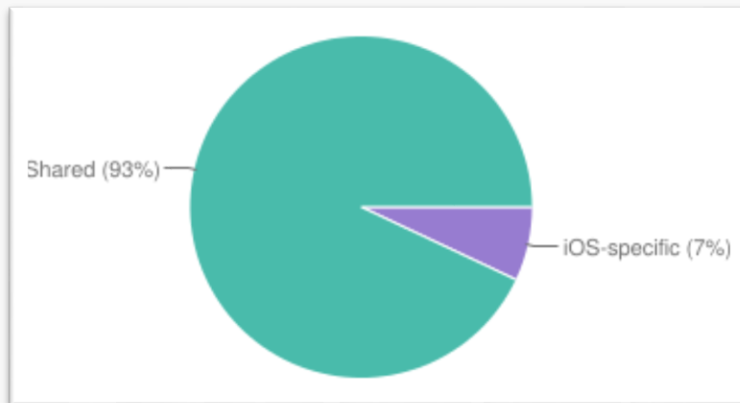
# Code Sharing - Percentuali

In quanto **native**, applicazioni Xamarin mantengono quasi sempre una porzione di codice platform-specific

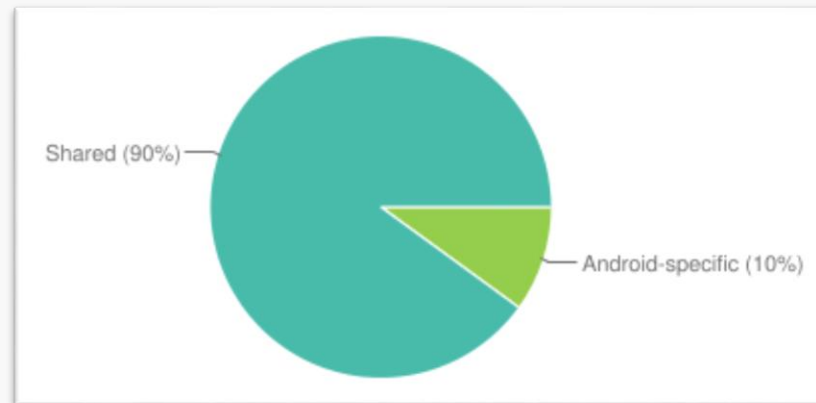
Cross  
Platform

Statistiche prese dall'app  
Xamarin Evolve 2016

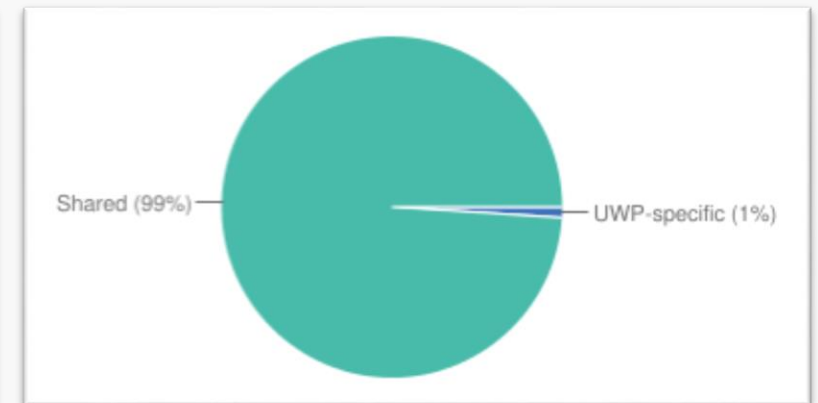
iOS



Android



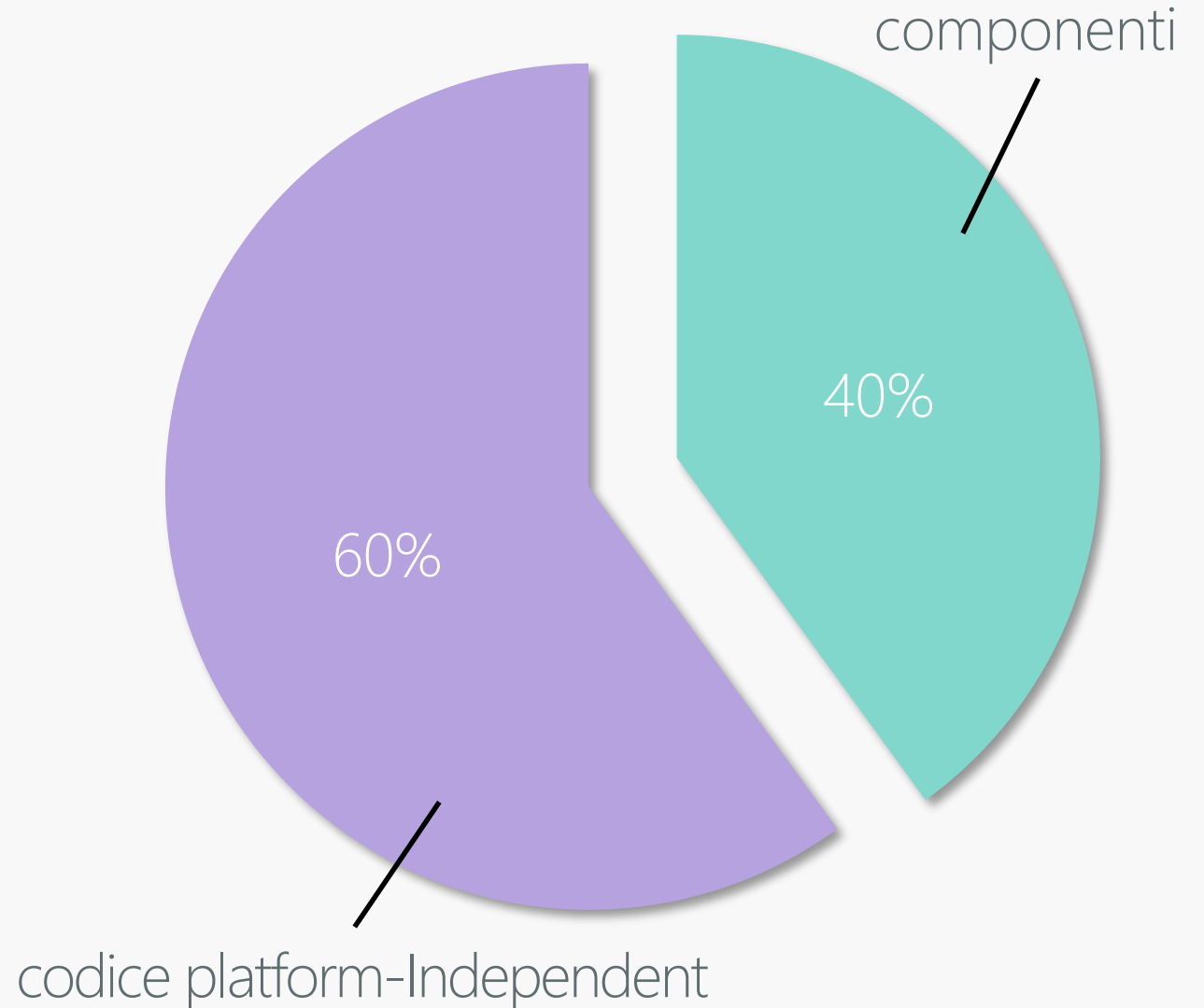
UWP



# Code Sharing - Percentuali

La porzione di codice condivisibile tra diverse piattaforme è suddivisa in:

- Componenti Riutilizzabili
- Codice Platform-Independent



# Code Sharing - Xamarin.Forms

Il framework Xamarin.Forms costituisce un esempio di riutilizzo e condivisione di componenti



Xamarin.Forms

## Shared C# Mobile

UI + Application Logic  
(PCL o Shared Project)

# Code Sharing - Accesso ai Dati

Offline Database:

- SQLite
- Realm

Storing e Sincronizzazione con il cloud:

- Azure Mobile Services (e.g. Easy Tables)
- Amazon
- Dropbox





# Code Sharing - Web Services

Utilizzo della classe platform-independent **HttpClient** per chiamate a servizi RESTful e parsing con:

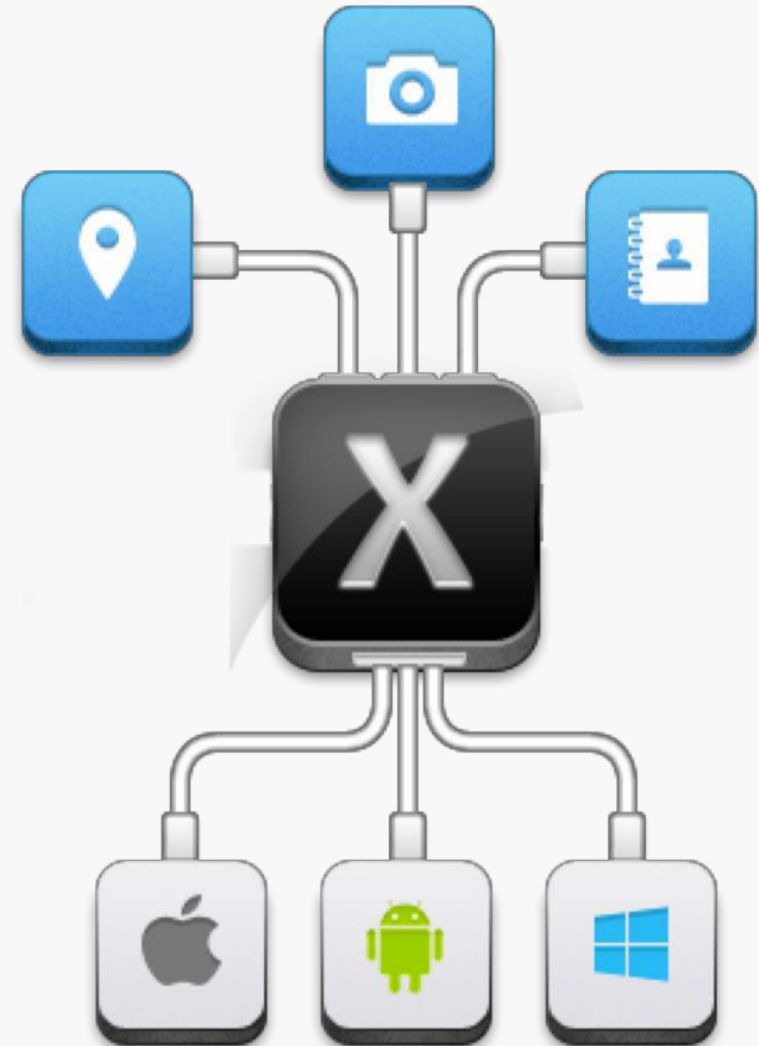
- **System.Xml / System.Json**
- LINQ to XML
- Json.NET



# Code Sharing - Librerie Xamarin

API cross-platform per funzionalità e servizi comuni:

- Xamarin.Social
- Xamarin.Auth
- Xamarin.Mobile



# Esempio MediaPlayer

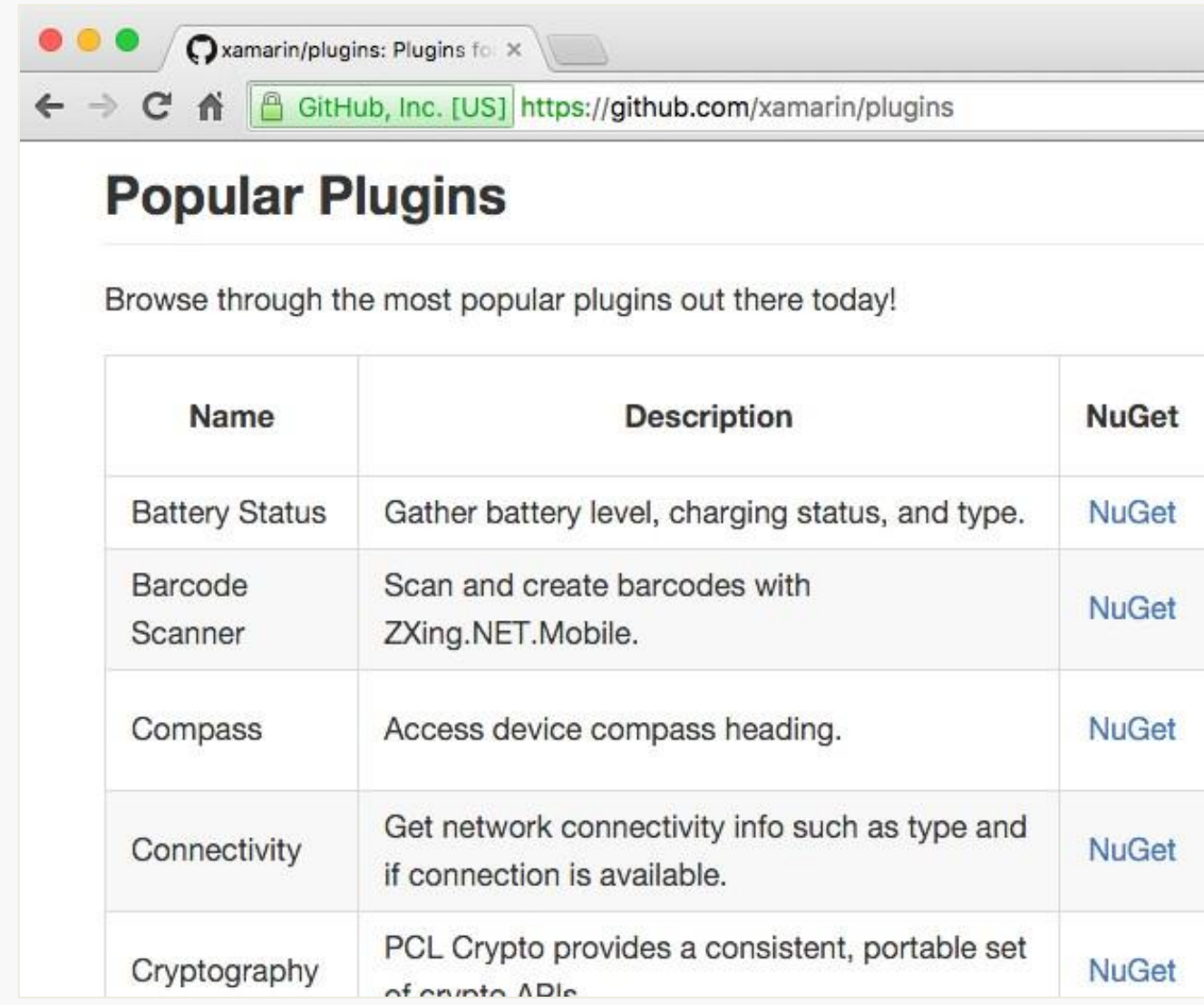
La libreria **Xamarin.Mobile** consente l'accesso ad alcune funzionalità del sistema operativo specifico quali la Camera

```
async void OnTakePicture(object sender, EventArgs e)
{
    var picker = new MediaPlayer();
    if (picker.IsCameraAvailable) {
        MediaFile photo = await picker.TakePhotoAsync(
            new StoreCameraMediaOptions {
                Name = "photo.jpg",
                DefaultCamera = CameraDevice.Rear
            });
        string filePath = photo.Path;
    }
}
```

# Altri Componenti

Oltre ai componenti integrati nel framework, sono disponibili ulteriori **Plugin** open-source sviluppati e mantenuti dalla community Xamarin

[github.com/xamarin/plugins](https://github.com/xamarin/plugins)

A screenshot of a web browser showing the GitHub page for Xamarin plugins. The browser's address bar displays the URL 'https://github.com/xamarin/plugins'. The page has a title 'Popular Plugins' and a subtitle 'Browse through the most popular plugins out there today!'. Below this is a table with three columns: 'Name', 'Description', and 'NuGet'. The table lists five plugins: 'Battery Status', 'Barcode Scanner', 'Compass', 'Connectivity', and 'Cryptography'. Each row provides a brief description of the plugin's functionality and a link to its NuGet package.

Popular Plugins		
Browse through the most popular plugins out there today!		
Name	Description	NuGet
Battery Status	Gather battery level, charging status, and type.	<a href="#">NuGet</a>
Barcode Scanner	Scan and create barcodes with ZXing.NET.Mobile.	<a href="#">NuGet</a>
Compass	Access device compass heading.	<a href="#">NuGet</a>
Connectivity	Get network connectivity info such as type and if connection is available.	<a href="#">NuGet</a>
Cryptography	PCL Crypto provides a consistent, portable set of crypto APIs.	<a href="#">NuGet</a>

# NuGet

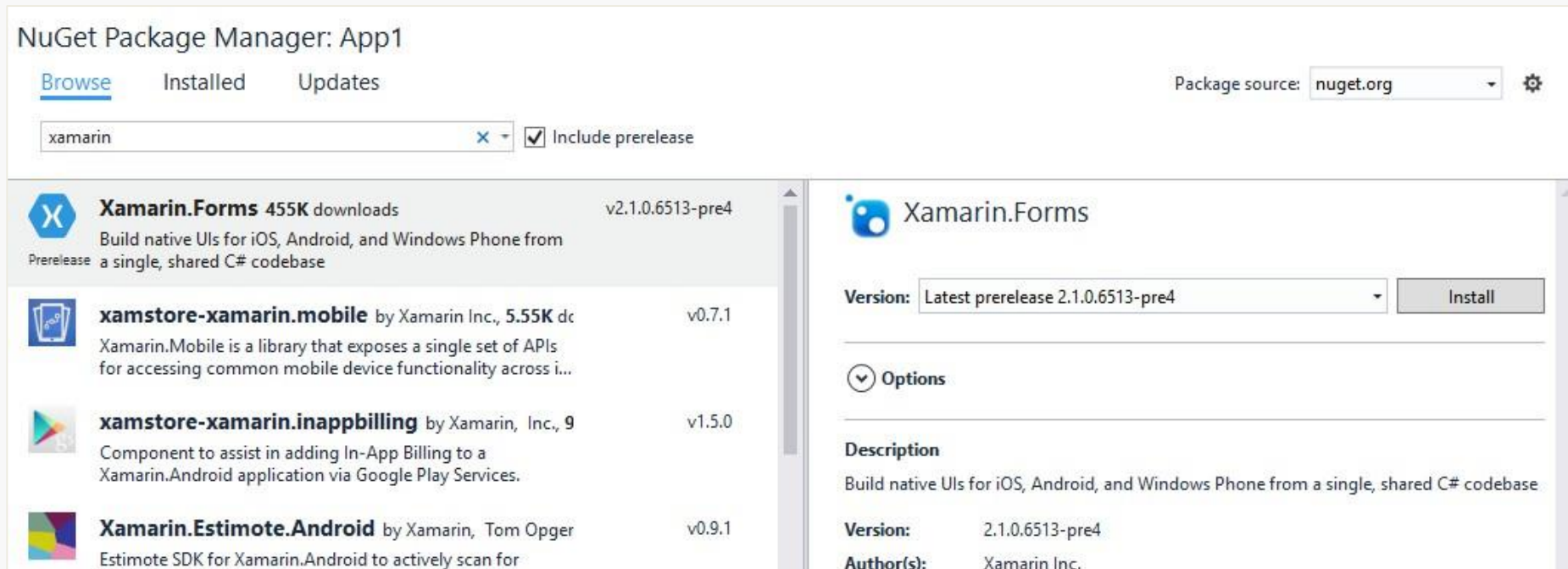
NuGet è il package manager di default in ambiente .NET e permette di gestire, installare ed aggiornare direttamente dall'IDE gli stessi componenti resi disponibili dalla community Xamarin



[www.nuget.org](http://www.nuget.org)

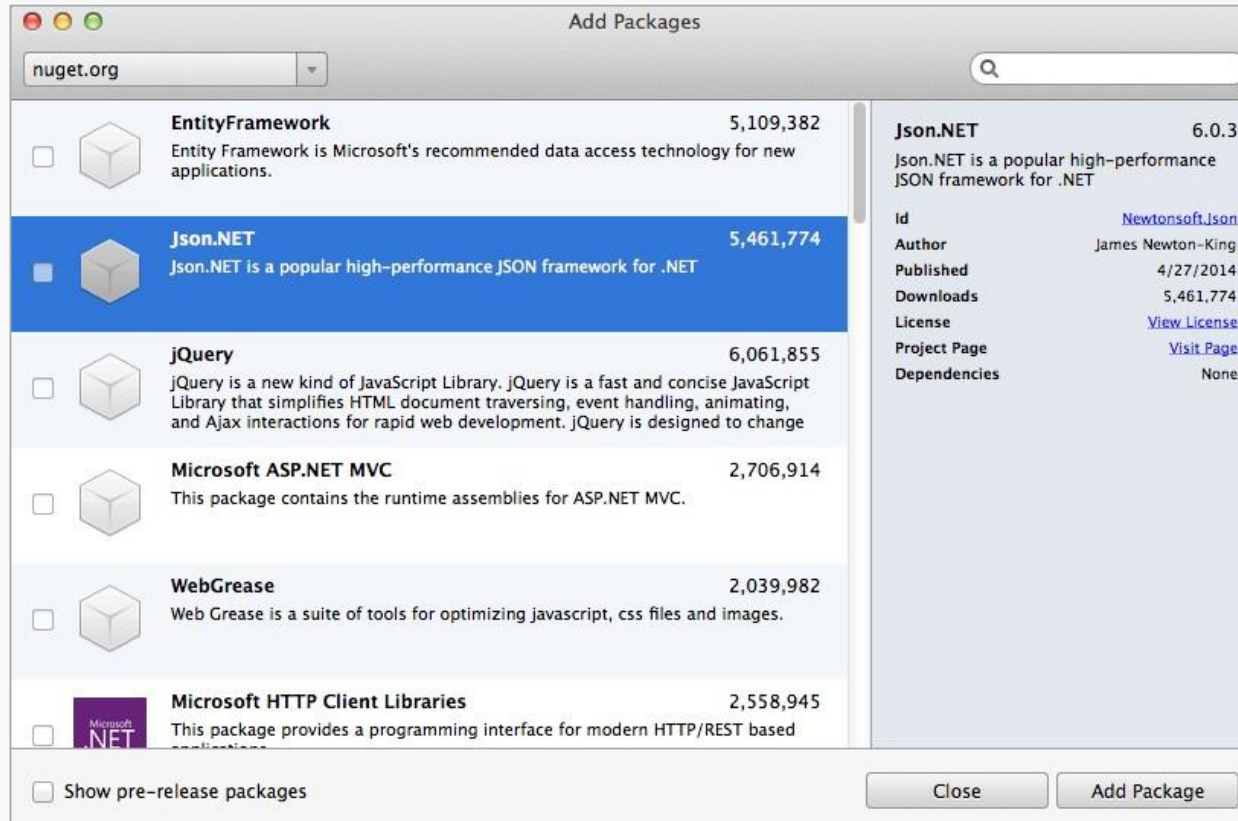
# NuGet - Visual Studio

È possibile aggiungere pacchetti NuGet da Visual Studio facendo click con il tasto destro sul progetto e selezionando Manage NuGet Packages



# NuGet - Visual Studio for Mac

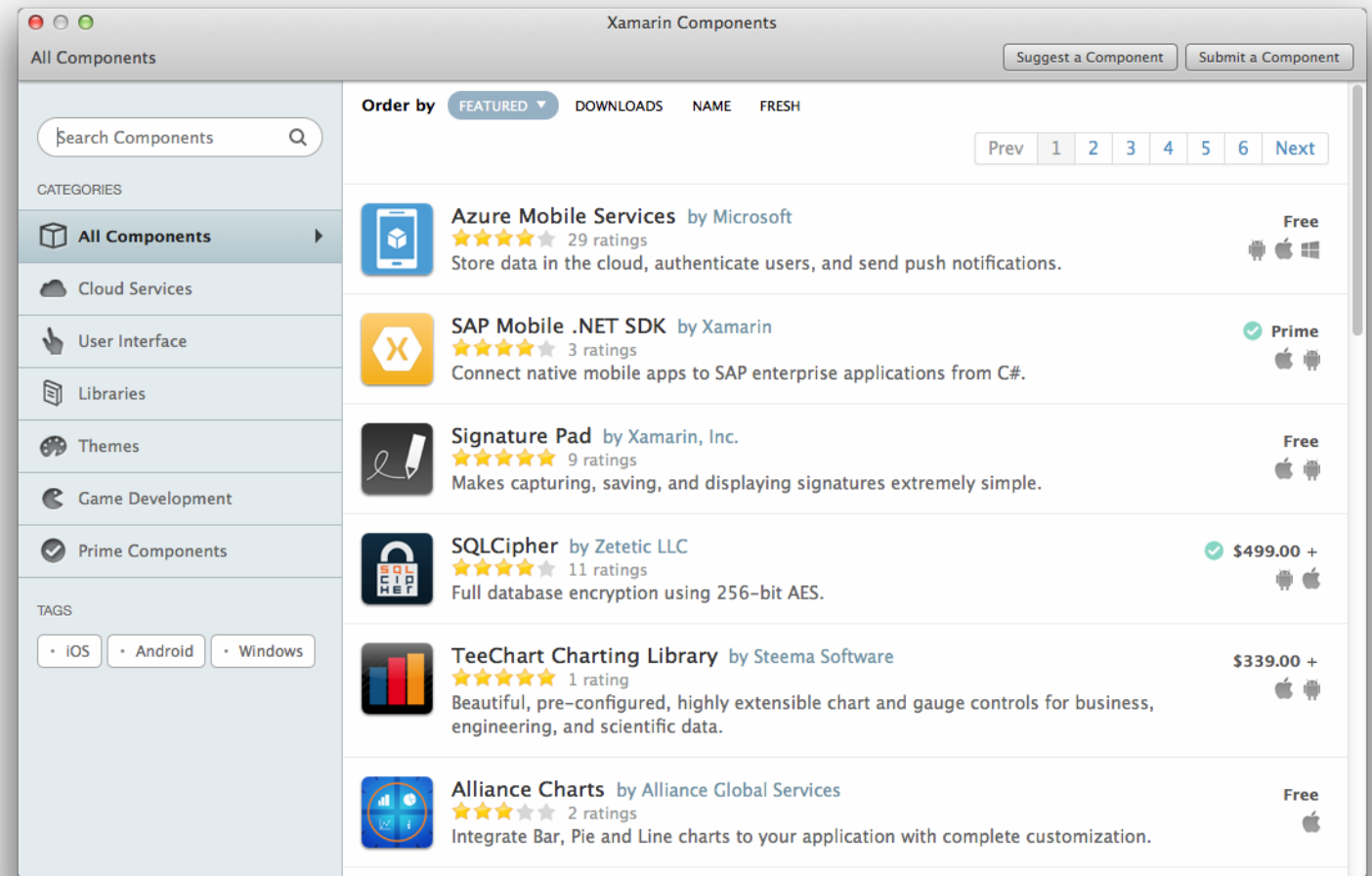
È possibile aggiungere pacchetti NuGet da Visual Studio for Mac facendo click con il tasto destro sul progetto e selezionando Add > Add NuGet Packages



# Xamarin Component Store

Lo Xamarin Component Store ospita ulteriori componenti riutilizzabili

Sono installati nella cartella **Components** del singolo progetto Xamarin



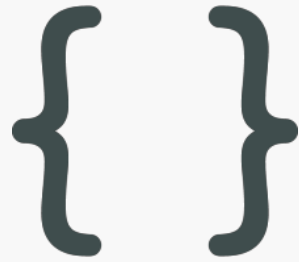


# Codice Condivisibile

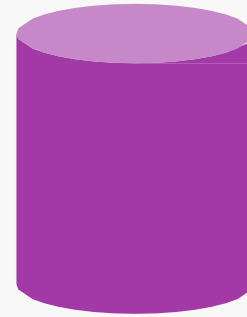
Una sezione di codice può essere condivisa tra diverse piattaforme se non dipende da specifiche funzionalità presenti solo in alcune di queste



Interrogazione di un  
web service



Parsificazione  
di un dato  
(XML, JSON...)



Accesso a  
un Database



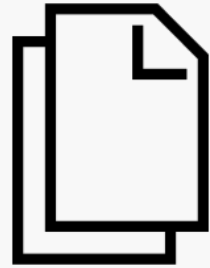
Esecuzione di logica  
computazionale

# Codice Non Condivisibile

Se una sezione di codice dipende da API presenti solo su una piattaforma (platform-specific API), questa deve essere **isolata** o, o la sua chiamata **astratta** in modo da poter essere usata solo dove disponibile



Informazioni  
di Sistema



Accesso al File  
System



Informazioni Personali  
(Contatti, Messaggi...)



Uso di periferiche  
esterne

# Tipi di Progetto

Esistono due tipologie di progetto adatte a condividere codice tra progetti Xamarin specifici



Shared Project



Portable Class  
Library

# Shared Project

# Prima di SP – File Linking

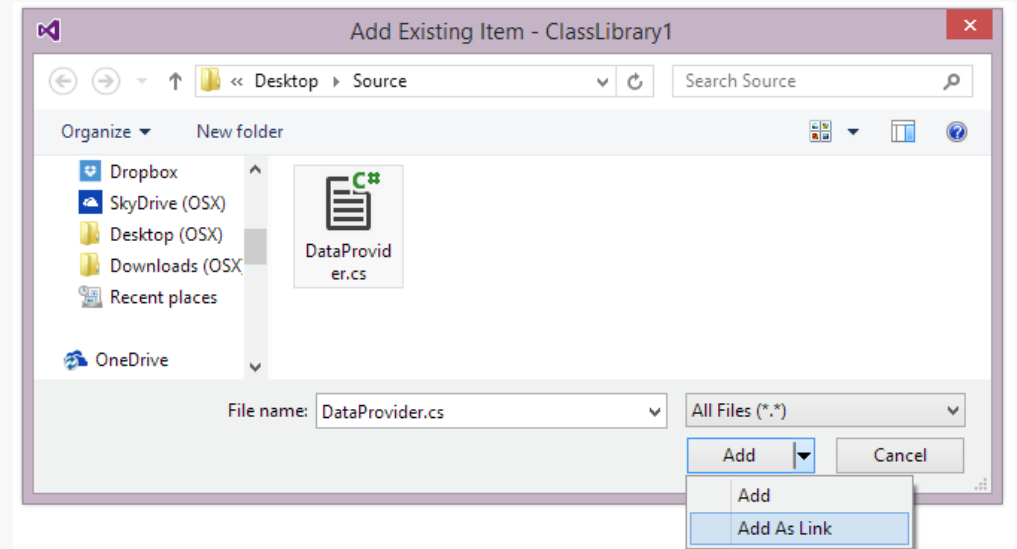
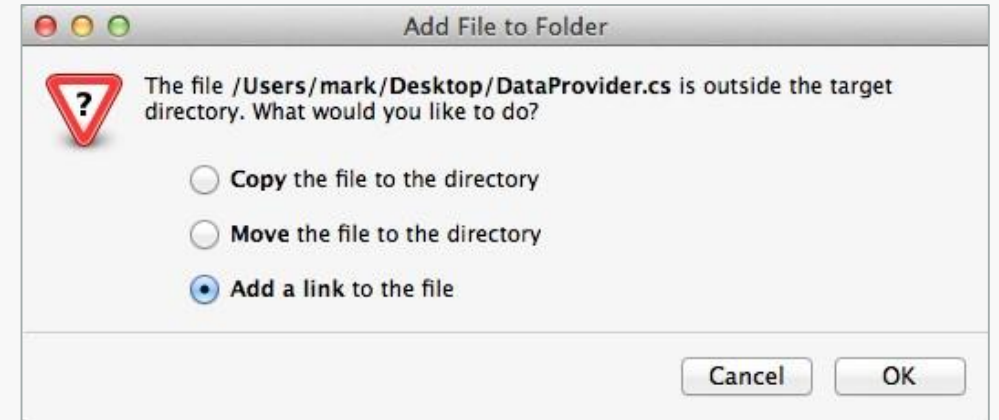
Il **File Linking** consente di condividere singoli file tra un progetto e l'altro

## Vantaggi:

- Unica copia dei file sorgente
- Semplicità di utilizzo - direttive al preprocessore per isolare codice platform-specific

## Svantaggi:

- Refactoring e Navigazione limitate
- Testing difficile



# Shared Project

Gli Shared Project permettono di condividere più file sorgente ed asset sotto forma di progetto condiviso

## Vantaggi

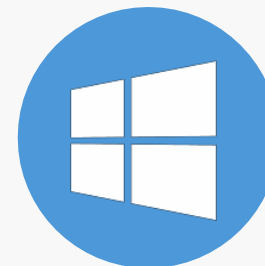
- Gli stessi del File Linking



- Compilato assieme al progetto che lo riferisce – nessuna dll generata
- Refactoring e Navigazione sempre possibili



SP con Sorgenti ed Asset



# Packaging

Un SP è descritto dal file .shproj

Shared.shproj

NoteManager.cs

NoteItem.cs

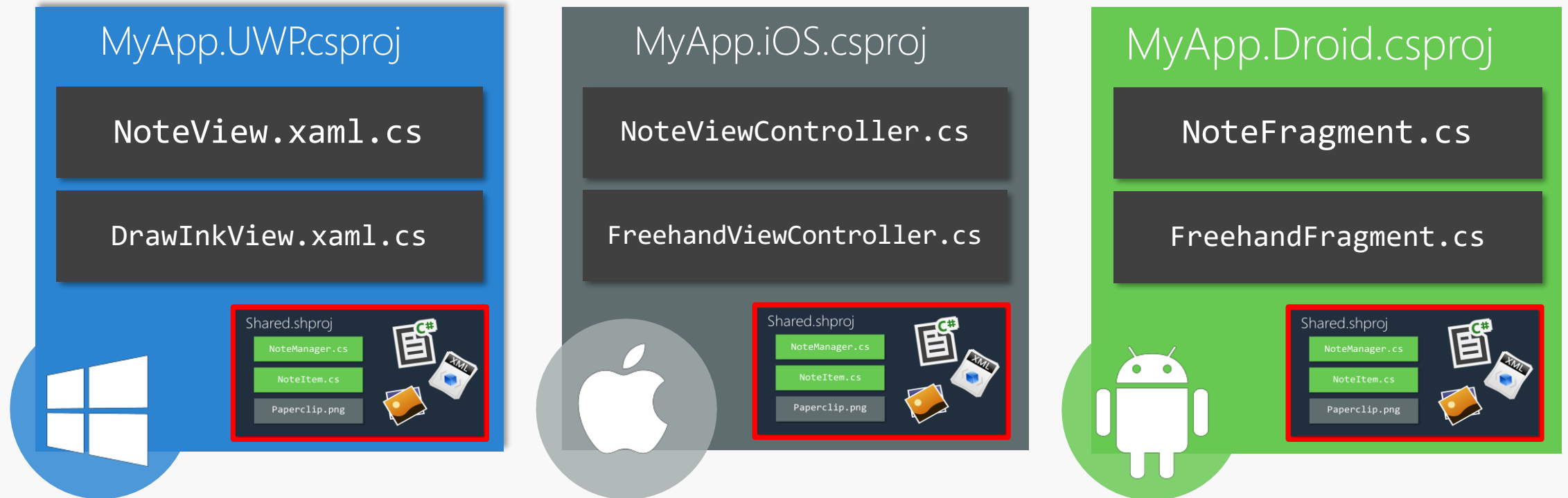
Paperclip.png



Così come per tutti i progetti, all'interno di un SP è possibile specificare il **build type** dei file inclusi (e.g. Compile, None, ecc.) ma un SP non genera di per sé alcun assembly

# SP – Compile Time

Aggiungendo un riferimento ad uno Shared Project, durante il processo di compilazione del progetto specifico vengono aggiunti tutti i file contenuti nell'SP, dopodiché viene compilato un unico progetto





# Strategie Platform-Specific

Esistono diverse strategie per gestire casi in cui si voglia differenziare l'implementazione di una funzionalità platform-specific in uno Shared Project (o in file linkati)



# Compilazione Condizionale

È la strategia più semplice per isolare codice platform-specific

Simboli al preprocessore:

```
#if MOBILE____  
#if ANDROID____  
#if IOS____  
#if WINDOWS_PHONE  
#if SILVERLIGHT  
#else //UWP
```

```
public static string DatabaseFilePath {  
    get {  
        var filename = "HRdb.db3";  
        #if WINDOWS_PHONE  
            var path = filename;  
        #elif __ANDROID__  
            var path = Path.Combine(  
                Environment.GetFolderPath(  
                    Environment.SpecialFolder.Personal),  
                filename);  
        #elif __IOS__  
            string documentsPath = Environment.GetFolderPath(  
                Environment.SpecialFolder.Personal);  
            var path = Path.Combine(  
                documentsPath,  
                "..", "Library",  
                filename);  
        #endif  
        return path;  
    }  
}
```

# Class Mirroring

Consente di implementare una dipendenza da codice platform-specific

N.B. Tutte le piattaforme devono definire un implementazione

```
public class NoteManager
{
    void CloudBackupComplete()
    {
        Alert.Show("Success!",
            "Notes have been backed up.");
    }
}
```

Shared Project

```
class Alert
{
    internal static void Show(string title,
                             string message) {
        new UIAlertView(title, message, null, "OK")
            .Show();
    }
}
```

AnyNote.iOS

```
class Alert
{
    internal static void Show(string title,
                             string message) {
        new AlertDialog.Builder(Application.Context)
            .SetTitle(title)
            .SetMessage(message)
    }
}
```

AnyNote.Droid

# Classi Parziali

Le classi parziali permettono di suddividere l'implementazione di una classe su più file sorgente

Utilizzate ad esempio per associare markup a codice (e.g. xaml.cs, designer.cs)

In Xamarin utilizzate per separare la definizione tra lo Shared Project e il progetto specifico

```
partial class NoteManager
{
    void OnDeleteNote() {
        if (ShowAlert("Warning!", "...")) {
            ...
        }
    }
}
```

Shared Project



```
partial class NoteManager
{
    bool ShowAlert(
        string title, string msg) {
        ...
    }
}
```

AnyNote.iOS

# Metodi Parziali

Utilizzati per rendere opzionale l'implementazione di un metodo

Se non si fornisce l'implementazione, il codice relativo alla chiamata del metodo non viene processato dal compilatore

```
partial class NoteManager
{
    partial void ShowPrintSettings();

    void PrintNote(NoteItem note) {
        ...
        ShowPrintSettings();
    }
}
```

Shared Project

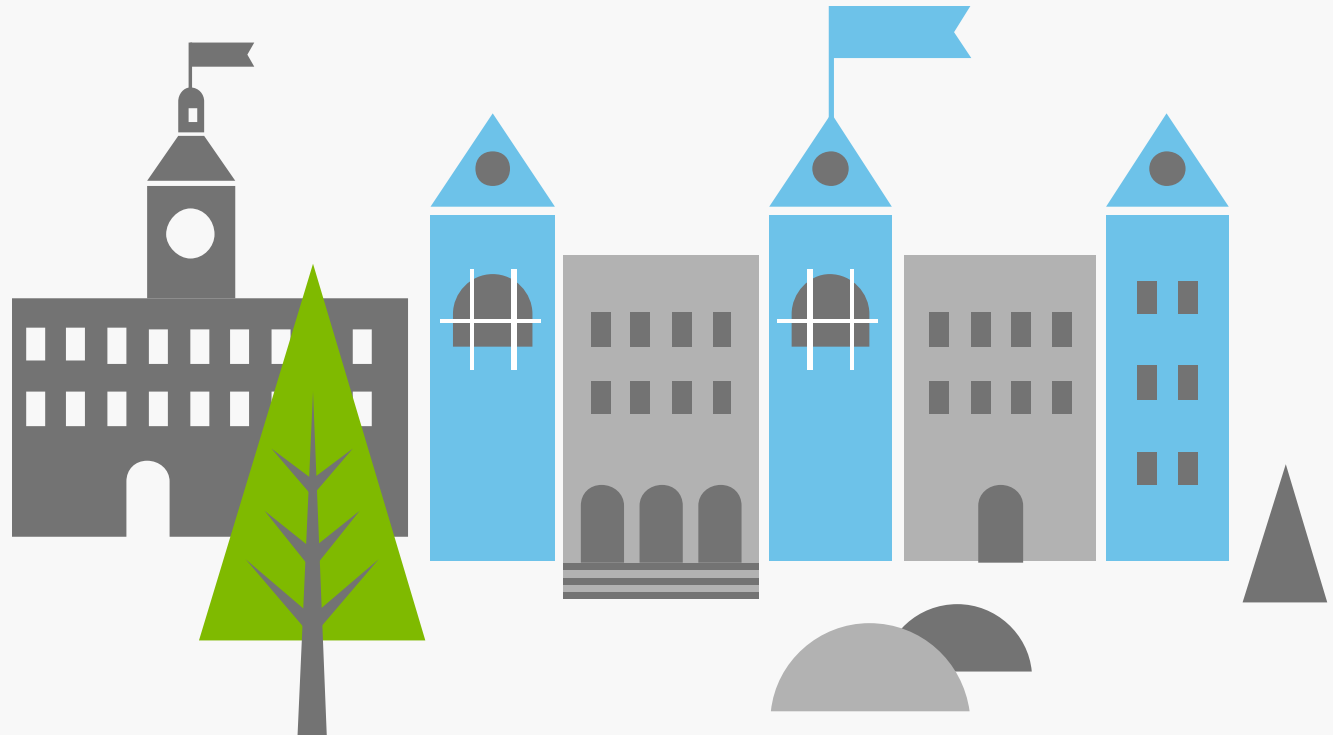
```
partial class NoteManager
{
    // No definition of method
}
```

NoteManager.iOS

# Code Sharing - SP

## Demo

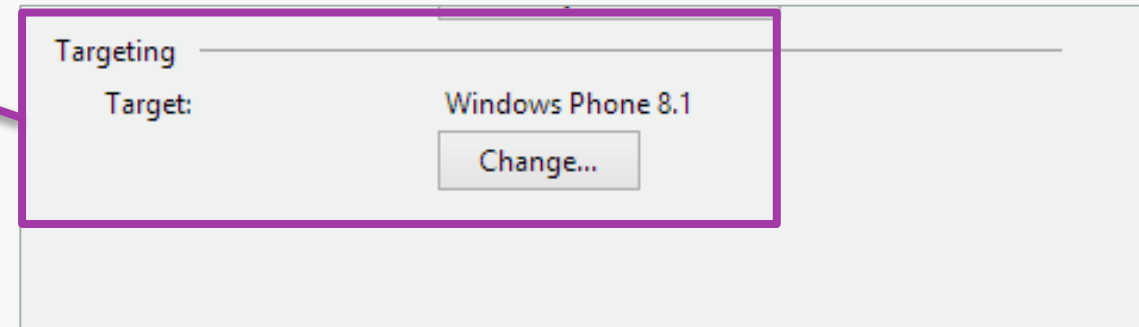
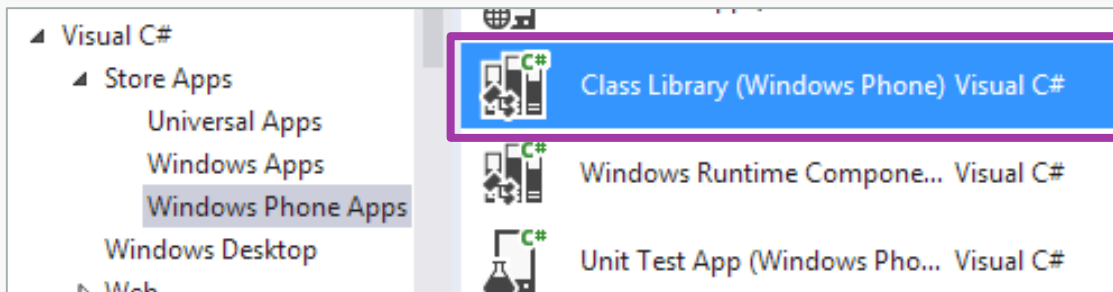
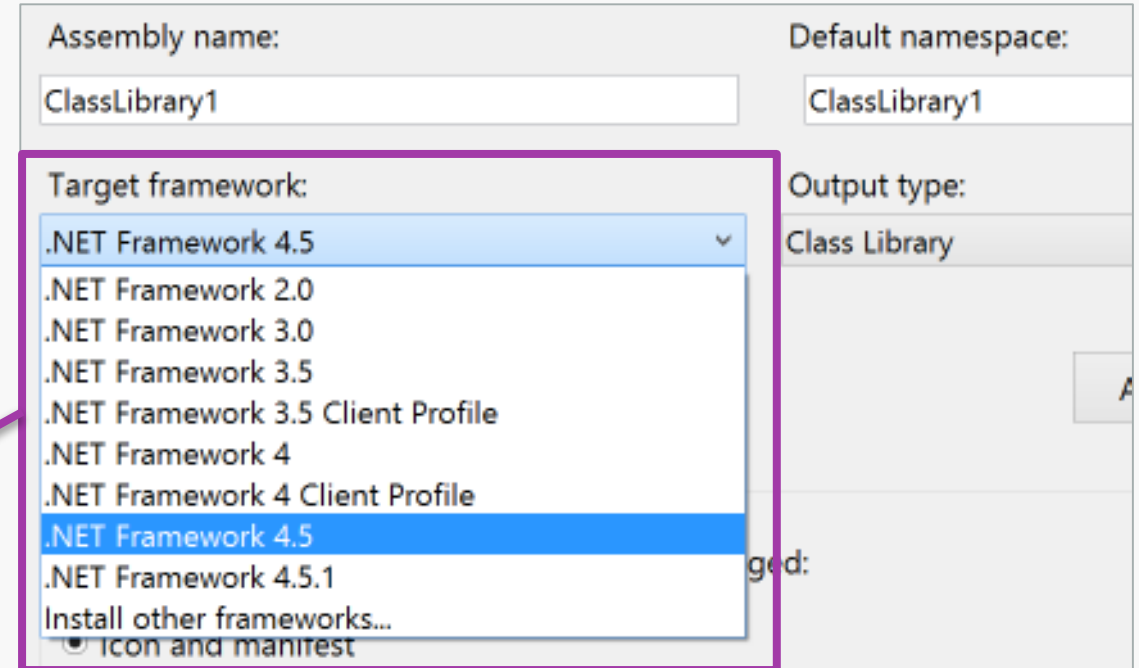
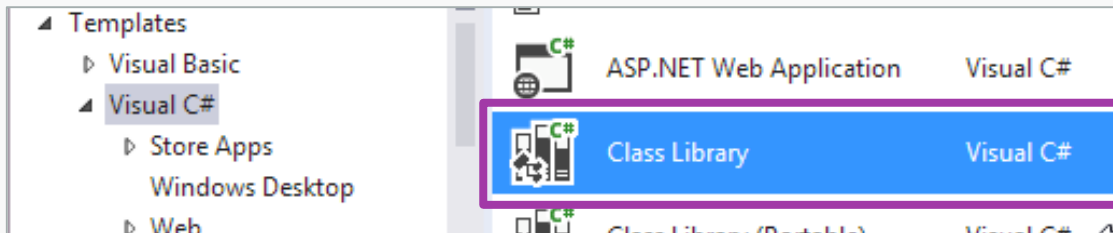
**GitHub Repo:** [bit.ly/2IKqMAI](https://bit.ly/2IKqMAI)



# Portable Class Library

# Prima di PCL - Class Library

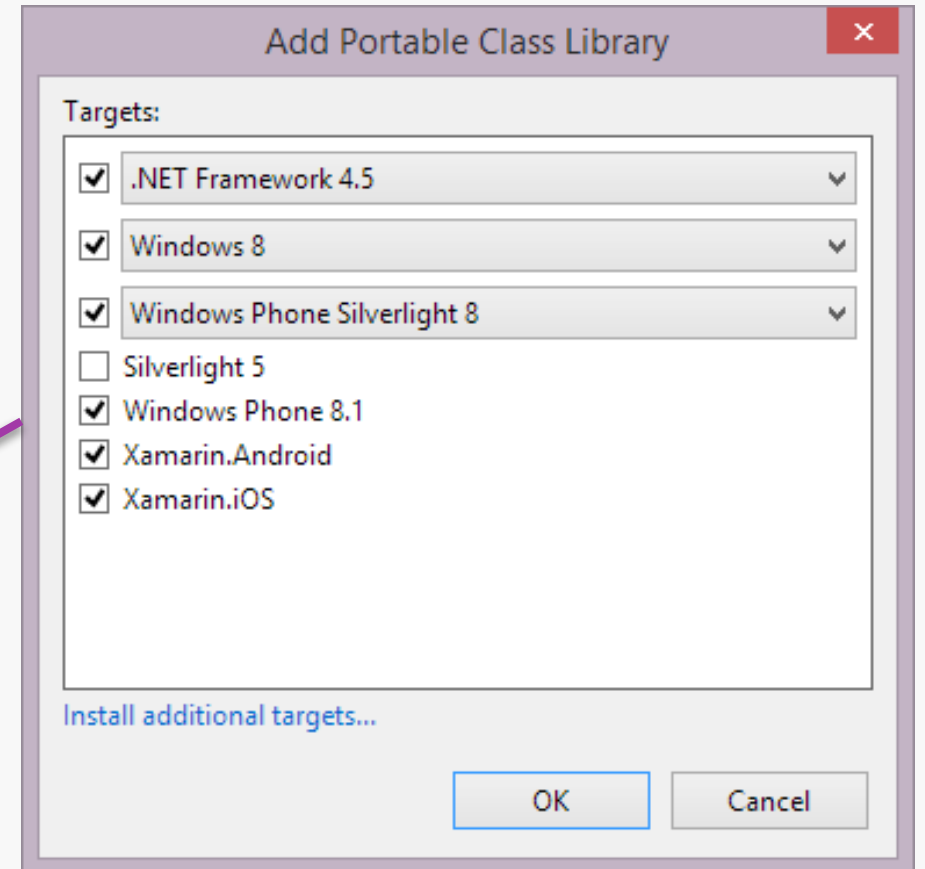
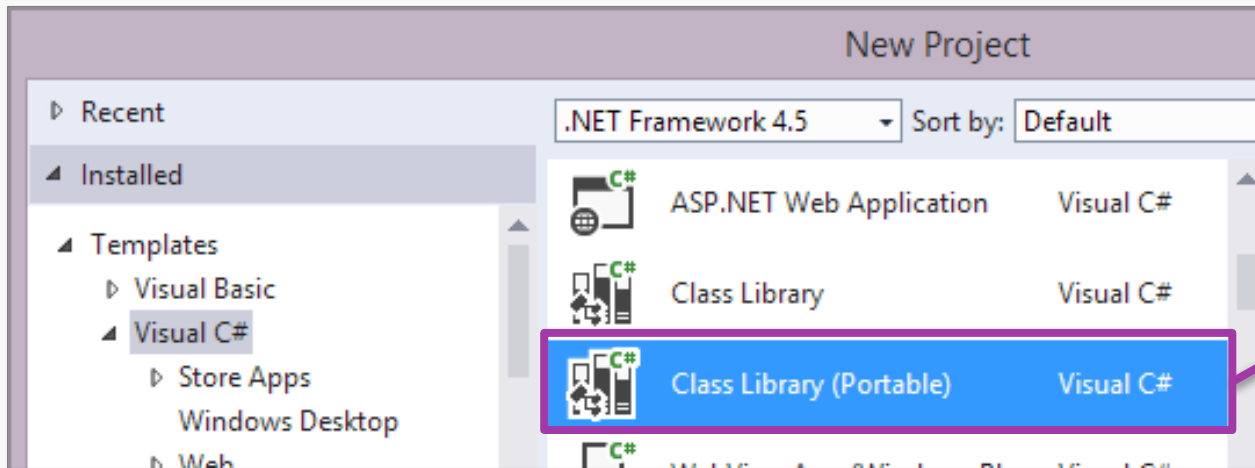
Progetti di tipo Class Library sono strettamente legati alla piattaforma specifica e al framework utilizzato





# Portable Class Library

Le Portable Class Library sono assembly che possono essere utilizzati in differenti tipologie di progetto .NET senza bisogno di ricompilare



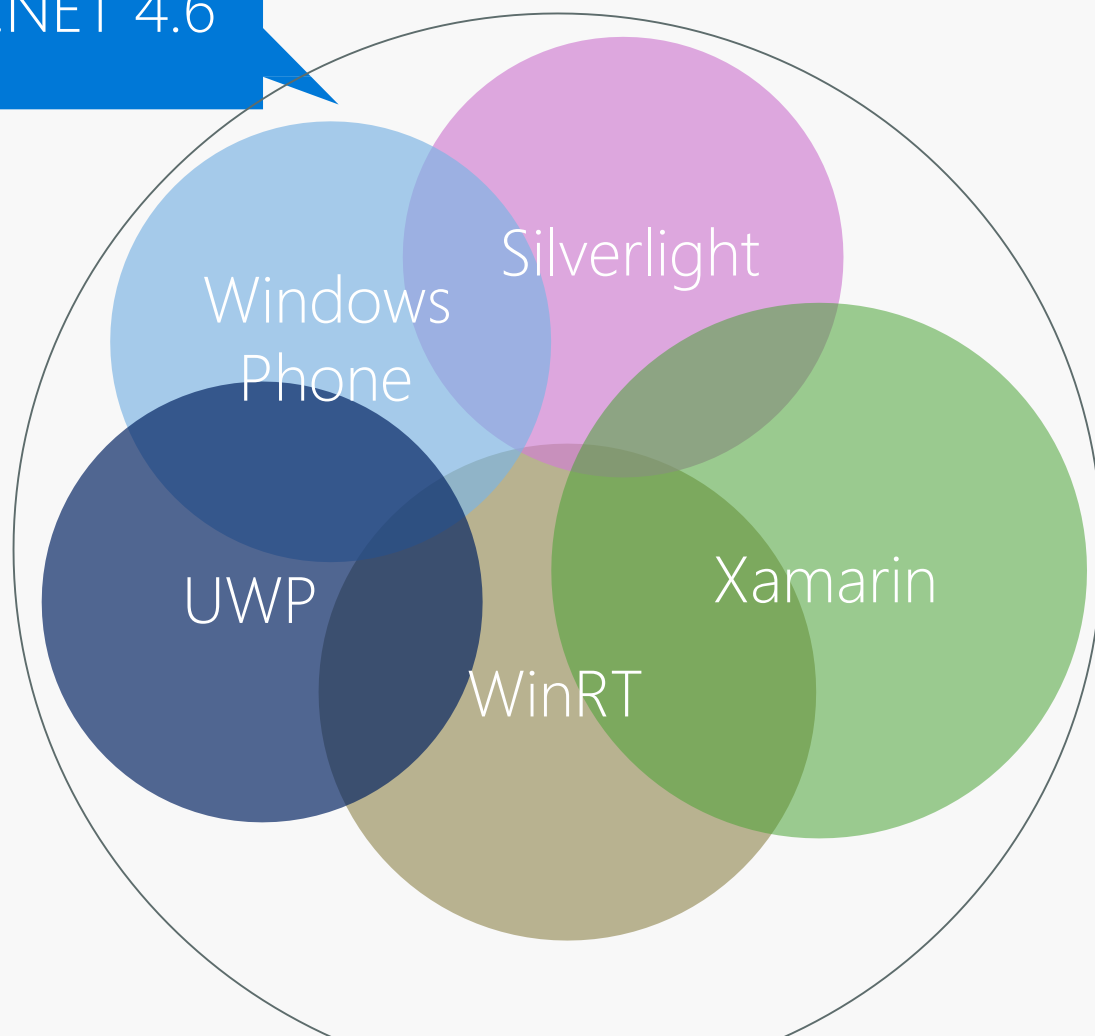
# Profili

Un profilo rappresenta uno **standard** o insieme di API relative alla Base Class Library (BCL) che la Class Library deve soddisfare per ognuna delle piattaforme

Feature	.NET Framework	Windows Store	Silverlight	Windows Phone (SL)	Windows Phone (Store)	Xamarin
Core Libraries	✓	✓	✓	✓	✓	✓
LINQ	✓	✓	✓	✓	✓	✓
IQueryable	✓	✓	✓	7.5+	✓	✓
Compression	4.5+	✓	✗	✗	✓	✓
Data Annotations	4.0.3+	✓	✓	✗	✗	✓
System.IO.File	✗	✗	✗	✗	✗	✗

# Configurazione di una PCL

.NET 4.6

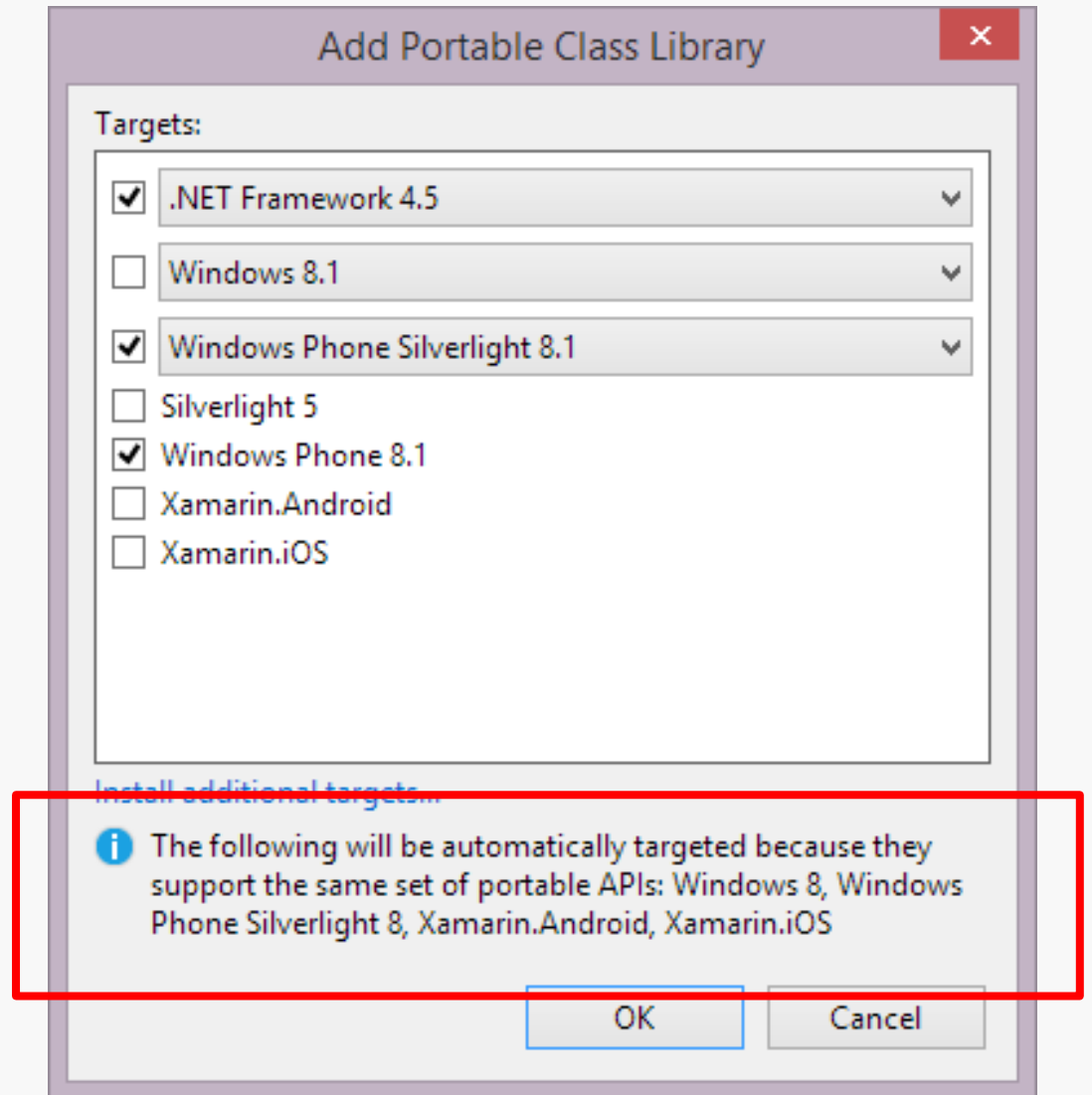


- Selezionare il set di piattaforme con cui si vuole rendere compatibile la libreria
- Una combinazione di piattaforme, assieme ad una versione del .NET framework, identifica un profilo
- Più piattaforme si scelgono, più si riduce il set di API disponibili

# Profili Mancanti

Non tutte le combinazioni sono disponibili in quanto Microsoft può ancora non aver rilasciato il profilo per una combinazione specifica

In tal caso, l'IDE consiglia la combinazione più "vicina"

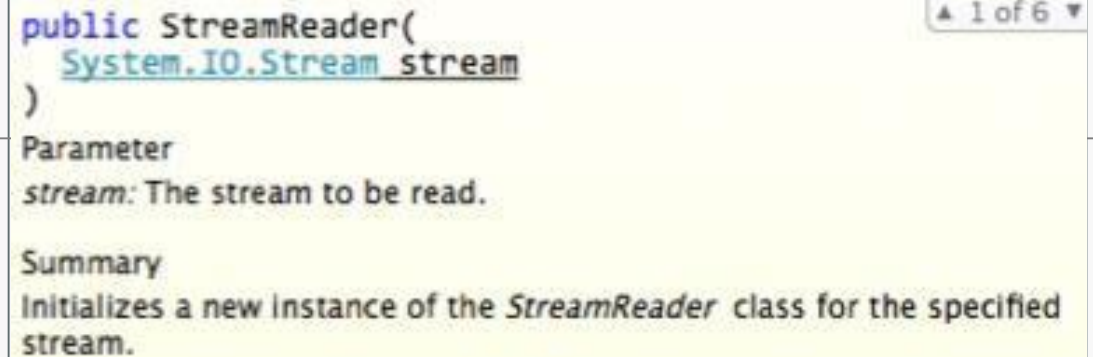


# Limiti di una PCL

Il numero di API disponibili in una PCL viene limitato inferiormente dalla piattaforma con meno API della BCL

```
partial class NoteManager
{
    void LoadNotes(string filename) {
        var reader = new System.IO.StreamReader(filename);
    }
}
```

In questo caso, il profilo scelto non prevede costruttori per il tipo **StreamReader** con parametri di tipo stringa



public StreamReader(  
 System.IO.Stream stream  
)  
Parameter  
stream: The stream to be read.  
Summary  
Initializes a new instance of the *StreamReader* class for the specified stream.

# Strategie Platform-Specific

Esistono diverse strategie per passare dati da codice platform-specific e codice condiviso nella PCL e viceversa

Memorizzare dati platform-specific in proprietà della PCL

1. Chiamare le API nel progetto platform-specific
2. Memorizzare il risultato in proprietà pubbliche esposte dalla PCL

# Strategie Platform-Specific

Esistono diverse strategie per passare dati da codice platform-specific e codice condiviso nella PCL e viceversa

Memorizzare dati platform-specific in proprietà della PCL

Utilizzare il FS per passare tipi supportati alla PCL

1. Decidere il path e il nome del file di output
2. Aprire uno **Stream** di comunicazione e parsificare il risultato

# Strategie Platform-Specific

Esistono diverse strategie per passare dati da codice platform-specific e codice condiviso nella PCL e viceversa

Memorizzare dati platform-specific in proprietà della PCL

Utilizzare il ES per

1. Definire un tipo astratto, un'interfaccia o un evento
2. Fornire un implementazione per quell'astrazione nel codice platform-specific

Utilizzare astrazioni di più alto livello



# Callback

Le PCL possono esporre eventi o delegati in modo da scatenare meccanismi di notifica da codice platform-specific

```
public class Dialer
{
    public static
        Func<string,bool> MakeCallImpl;

    public bool MakeCall(string number) {
        if (MakeCallImpl(number)) {
            ...
        }
    }
}
```

```
Dialer.MakeCallImpl = number =>
{
    return UIApplication
        .SharedApplication
        .OpenUrl(new NSURL(
            "tel:" + number));
}
```

PCL



# Astrazioni della Piattaforma

Astrazioni più complesse possono essere descritte da classi astratte o interfacce nella PCL

```
public interface IDialer
{
    bool MakeCall(string number);
}
```

PCL

1. Nel codice condiviso della PCL si definisce l'interfaccia **IDialer** che espone le funzionalità richieste



PhoneDialerIOS



PhoneDialerDroid



PhoneDialerWin

2. Nel progetto specifico si implementa l'interfaccia usando le API Xamarin platform-specific

# Dependency Injection

È possibile “iniettare” implementazioni concrete delle astrazioni via costruttore, metodo o setter di una proprietà

```
Dialer.Instance = new Dialer(new iPhoneDialer());
```

OR

```
Dialer.Instance.Initialize(new AndroidDialer());
```

OR

```
Dialer.Instance.Platform = new WindowsDialer();
```

# Code Sharing - PCL

## Demo

**GitHub Repo:** [bit.ly/2mOyZDm](https://bit.ly/2mOyZDm)



# SP e PCL – Pro e Contro

## Shared Project

### PRO

Disponibili tutte le API

Può essere aggiunta direttamente logica platform-specific

Tutti I tipi di file possono essere condivisi

Dimensioni del pacchetto finale inferiori

### CONTRO

Può portare a codice spaghetti

Lo Unit Testing può risultare tedioso per via della compilazione condizionale

Deve essere distribuita sotto forma di sorgente

## Portable Class Library

### PRO

Incoraggia l'utilizzo di design pattern

Unit Testing separato dai progetti specifici

Può essere distribuita sotto forma di assembly

### CONTRO

Numeri di API limitate

Difficile condividere file di asset

Richiede più Lavoro per implementare codice platform-specific

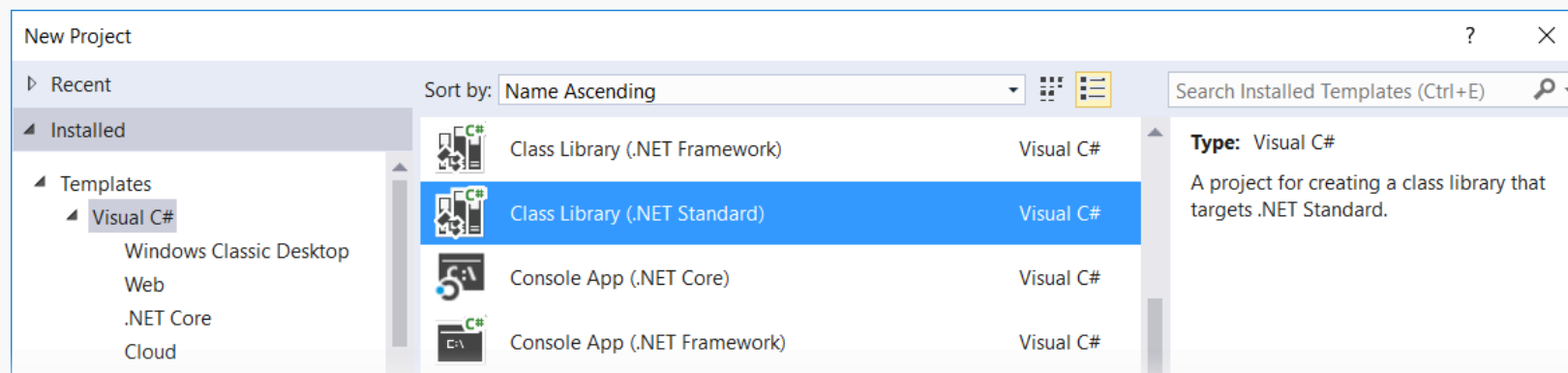
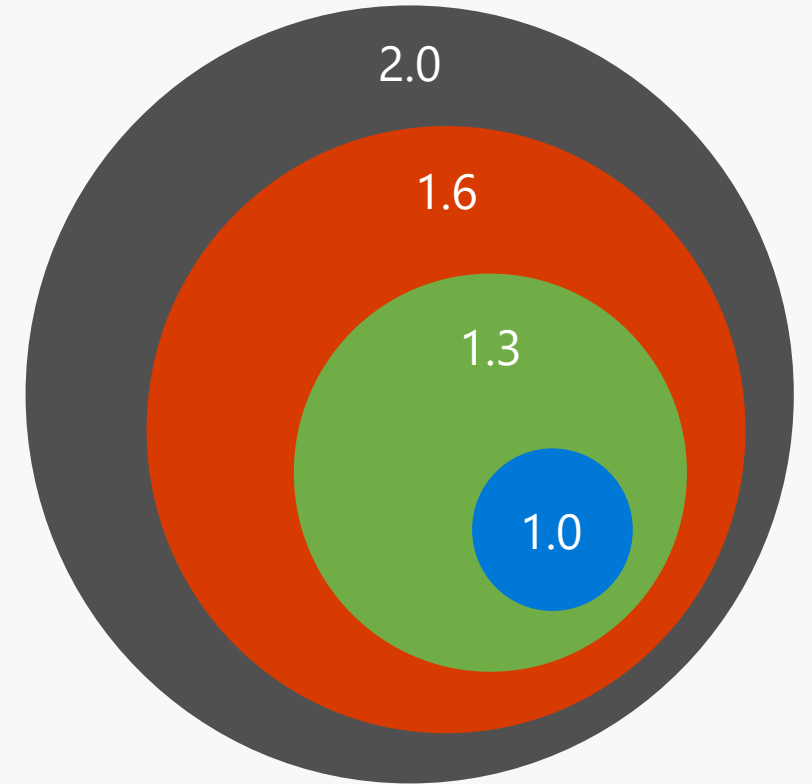
Uso limitato alle piattaforme target

# Oltre le PCL - .NET Standard Library

Rappresentano un set di API che tutte le piattaforme .NET (.NET Framework, .NET Core, Xamarin) devono implementare

Le API esposte non sono ricavate dall'intersezione tra piattaforme ma sono indipendenti da queste

Hanno versioni lineari – versioni maggiori incorporano le API offerte dalle versioni precedenti



# XAML Standard

Proposta di uno Standard per uniformare il nome di controlli e proprietà tra UWP e Xamarin.Forms

XAML attuale

```
<StackLayout Orientation="Horizontal"
              Background="SkyBlue">
  <Label Text="UserName: " />
  <Entry PlaceholderText="Enter your username" />
</StackLayout>
```



XAML Standard 1.0

```
<StackPanel Orientation="Horizontal"
             Background="SkyBlue">
  <TextBlock Text="UserName: " />
  <TextBox PlaceholderText="Enter your username" />
</StackLayout>
```

# XAML Standard

Proposta di uno Standard per uniformare il nome di controlli e proprietà tra UWP e Xamarin.Forms

The screenshot shows the GitHub repository page for **Microsoft / xaml-standard**. The repository has 92 watches, 575 stars, and 27 forks. The **Issues** tab is selected, showing 157 open issues. The filter bar shows `is:issue is:open`. The issue list includes:

Issue Title	Labels	Comments
<b>Discussion: XAML is more than XML. Code-behind needs consideration too</b> #193 opened 9 days ago by dotMorten		31
<b>Add MathMLView Control</b> #192 opened 11 days ago by insinfo	controls, proposal	6
<b>Add CommandParameter to Button</b> #191 opened 11 days ago by Daniel-Svensson		2
<b>Add Padding property to Label, TextBox, Button, etc.</b> #190 opened 12 days ago by dan-meier	proposal	6
<b>DataTemplate type</b> Needs more info		1

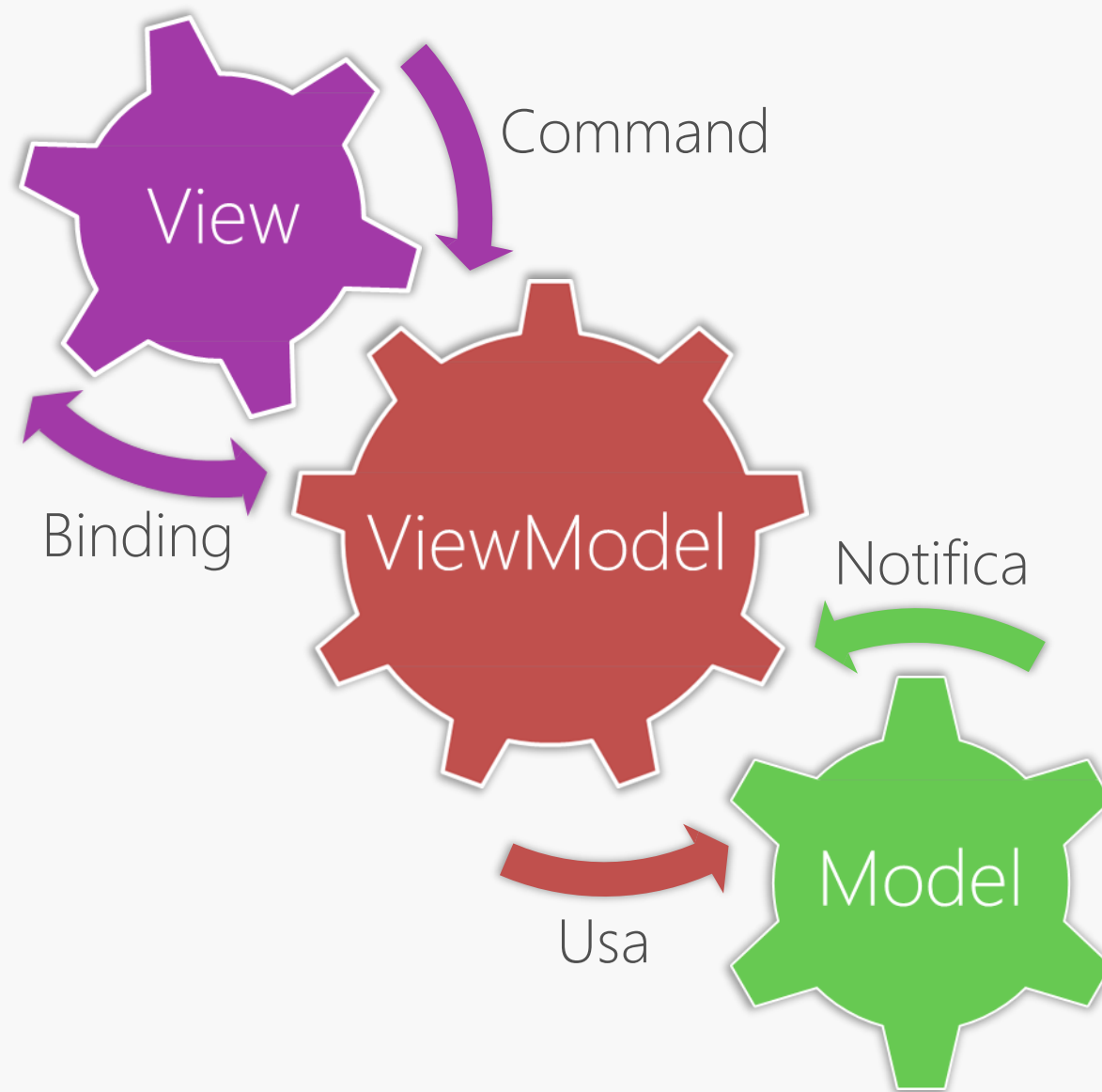


Model-View-ViewModel

# MVVM

**Model-View-ViewModel** (MVVM) è un design pattern architetturale (o di framework) atto a separare la logica dell'applicazione tra UI, dati (più comunemente modello) e comportamento

Il ViewModel prende il posto del controller di MVC gestendo l'interazione tra View e Model grazie all'engine di Data Binding



# Model

Il **Model** contiene il modello dei dati e la logica di business dell'applicazione (e.g. Persistenza, Validazione...)

Il Model fa parte dello Shared Code - pertanto non dovrebbe contenere funzionalità specifiche della piattaforma

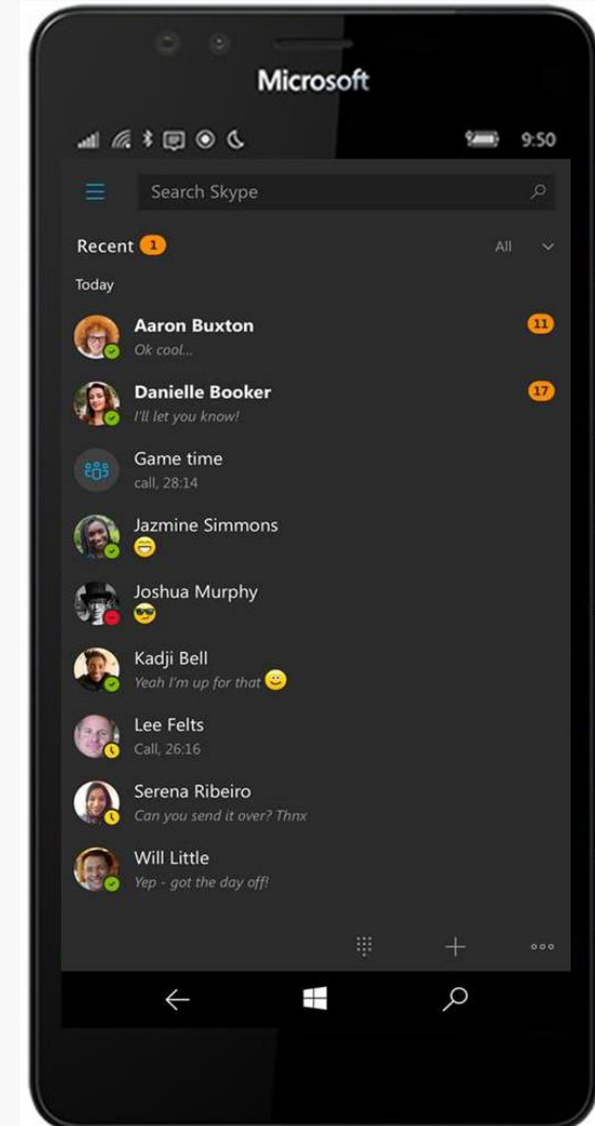
```
public class Profile {  
    public string Name { get; set; }  
    public string BackgroundImage { get; set; }  
    public string ProfileImage { get; set; }  
    public string Title { get; set; }  
    public string Description { get; set; }  
    public uint Likes { get; set; }  
    public uint Following { get; set; }  
    public uint Followers { get; set; }  
}
```

# View

La **View** presenta le informazioni all'utente utilizzando il look-and-feel nativo della piattaforma

Non dovrebbe contenere codice su cui effettuare unit testing

Proprietà visuali e animazioni devono essere gestite a questo livello



# ViewModel

Il **ViewModel** fornisce una rappresentazione view-centrica del **Modello** alla **View**

Esponde le  
proprietà  
sorgenti per il  
Binding

```
public class ProfileViewModel : INotifyPropertyChanged {  
    private readonly Profile _profile;  
    public ProfileViewModel(Profile profile) {  
        _profile = profile;  
    }  
    public string Name {  
        get => _profile.Name;  
        set {  
            if (_profile.Name == value)  
                return;  
            _profile.Name = value;  
            OnPropertyChanged();  
        }  
    }  
}
```

Incapsula il  
Modello

# ViewModel

Agisce da Adapter evitando di dover realizzare Converter per ogni Bindable Property

Formato della  
proprietà  
manipolato

```
public class ProfileViewModel : INotifyPropertyChanged
{
    // ...
    public string ProfileImagePath =>
        await DownloadImageInLocalFolderAsync(_profile.ProfileImage);

    private async Task<string> DownloadImageInLocalFolderAsync(
        string profileImage)
    {
        // Download Image and Return File Path asynchronously
    }
}
```

# ViewModel

E' il posto adatto per inserire logica aggiuntiva per la UI:

- Eseguire validazione dell'input prima di aggiornare il modello
- Controllare valori di stato della UI

```
partial class DownloaderViewModel {  
    private int _percentComplete;  
    public int PercentComplete {  
        get { return _percentComplete; }  
        set {  
            if (_percentComplete == value)  
                return;  
            _percentComplete = value;  
            OnPropertyChanged();  
        }  
    }  
}
```

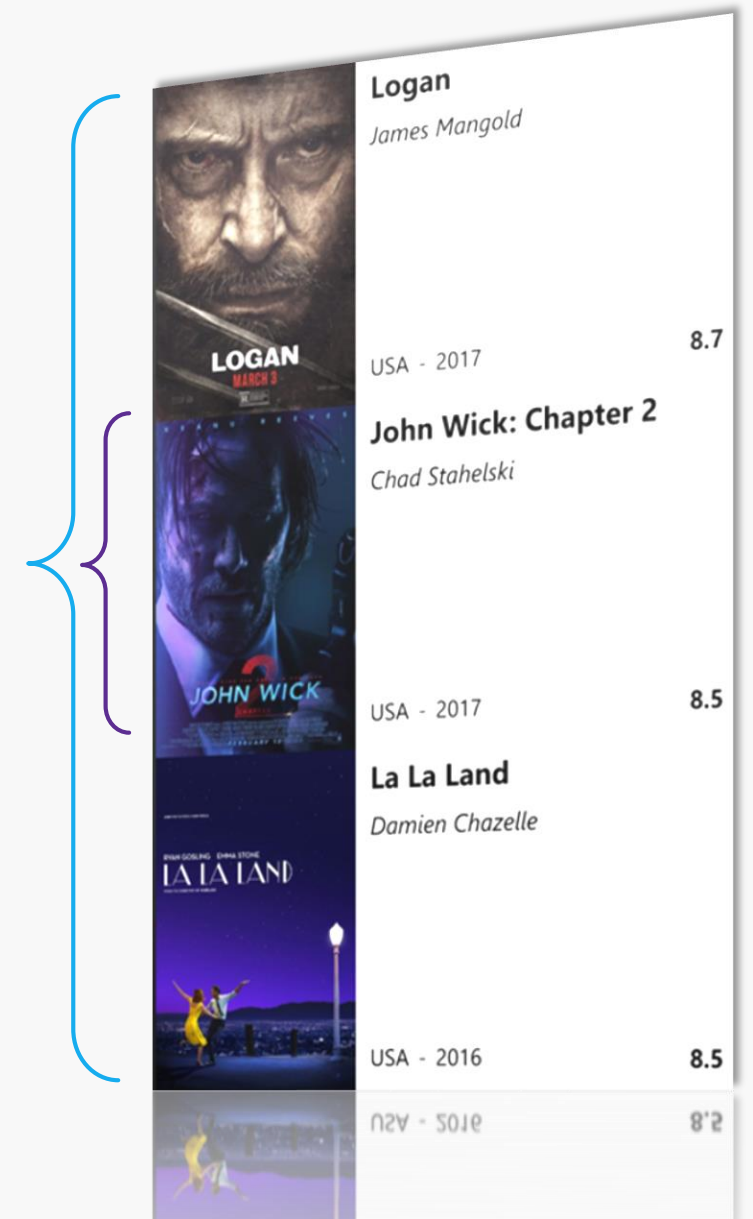
Modifica del valore di una proprietà per controllare un indicatore di caricamento

# ViewModel

Spesso un'applicazione possiede diversi ViewModel – uno per ogni entità per cui si prevede una rappresentazione della UI

Un ViewModel può essere condiviso tra più View (**ItemViewModel**)

In MVVM ad ogni **Page** è associato il corrispondente ViewModel (**PageViewModel**), che agisce da Controller per quella Page

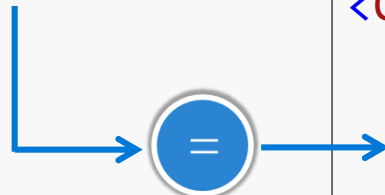




# View e ViewModel - Connessione

Un `PageViewModel` viene spesso utilizzato come **BindingContext** per l'intera pagina - si può specificare sia tramite C# che XAML

```
public partial class MainPage : ContentPage {  
    private readonly MainPageViewModel _viewModel  
    public MainPage()  
    {  
        BindingContext = _viewModel =  
            new MainPageViewModel();  
        InitializeComponent();  
    }  
    //...  
}
```



```
<ContentPage ...>  
    <ContentPage.BindingContext>  
        <viewModels:MainPageViewModel/>  
    </ContentPage.BindingContext>  
</ContentPage>
```

# MVVM - Pro e Contro

**MVVM** è adatto per framework che offrono un engine di Data Binding come Xamarin.Forms, WPF o UWP ed è l'approccio consigliato per applicazione complesse

## PRO

- Trae vantaggio dal Data Binding
- Permette di creare componenti loosely-coupled facilmente portabili e testabili
- Incoraggia l'utilizzo di design pattern permettendo di isolare View dalla logica di business
- Sostituisce l'utilizzo di Converter

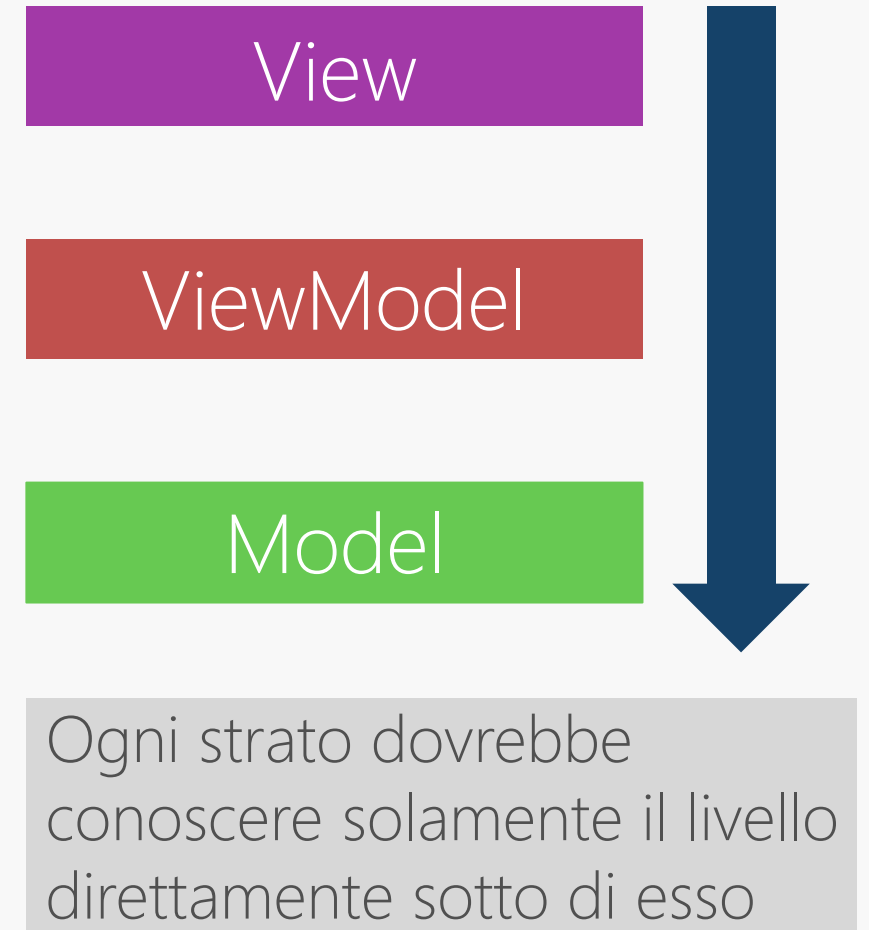
## CONTRO

- Richiede un minimo di infrastruttura
- Necessita di diversi strati – overhead per piccole applicazioni
- Binding più difficile da debuggare e poco performante per grandi data set

# View vs. ViewModel

Ogni ViewModel dovrebbe essere sviluppato per essere UI-agnostico così da essere facilmente portabile

Pertanto, un ViewModel non dovrebbe contenere nessun riferimento a tipi di Xamarin.Forms



# ViewModel e Platform-Specific

Si assuma di dover cambiare il colore di una **Label** Title della UI in base ad un requisito del Modello

```
public class ProfileViewModel {  
    // ...  
    public bool IsAdmin { get; set; }  
}
```



```
<Label Text="{Binding Title}" >  
    <Label.Triggers>  
        <DataTrigger TargetType="Label"  
            Binding="{Binding IsAdmin}"  
            Value="True">  
            <Setter Property="TextColor" set; set; }  
            Value="#0979C1" />  
        </DataTrigger>  
    </Label.Triggers>  
</Label>
```



# MVVM e Design Pattern

MVVM incoraggia l'Utilizzo di altri Design Pattern atti a creare componenti il più disaccoppiati possibile tramite astrazioni e meccanismi di notifica

Dependency  
Injection

Factory e  
Singleton

Command

Navigation

Alert e  
Prompt

Messaggi

# Messaging

Spesso più ViewModel possono avere la necessità di comunicare tra di loro senza per forza conoscersi a vicenda - Xamarin.Forms offre built-in un servizio di Messaging (non solo tra VM) chiamato **MessagingCenter**



3. Receive

# Publicazione di un messaggio

Il **Publisher** passa come argomento la chiave del messaggio ed il suo payload (opzionale)

**Send** accetta come parametri generici il tipo del mittente e quello del parametro passato



```
MessagingCenter.Send<MainViewModel, ItemViewModel>(
    this, "Item", selectedItem);
```

# Sottoscrizione di un messaggio

I **Subscriber** identificano il messaggio da ricevere dalla combinazione [tipo sender, tipo parametro, chiave] - **Subscribe** espone un Delegato come callback per la ricezione del messaggio

```
MessagingCenter.Subscribe<MainViewModel, ItemViewModel> (  
    this, "Item",  
    (mainVM, selectedItem) => {  
        // Action to run when "Item" is received  
        // from MainViewModel  
    });
```

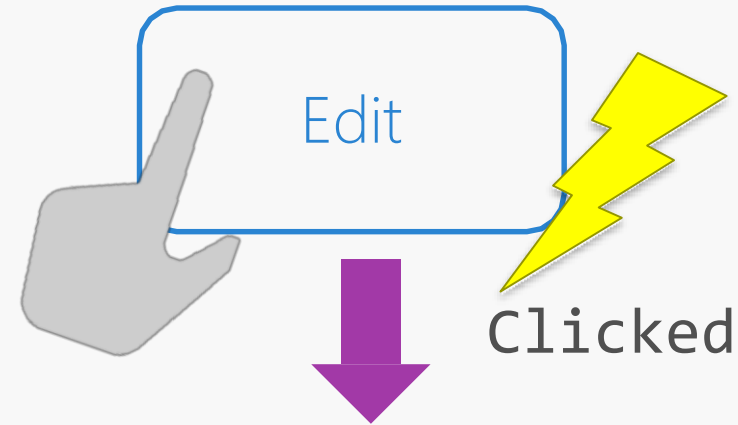


# Gestione degli Eventi

La UI solleva eventi per gestire l'interazione con l'utente

- **Clicked**
- **ItemTapped**
- ...

Un caveat di questo approccio è che gli eventi Xamarin.Forms possono essere gestiti solamente nel code-behind



```
public MainPage()
{
    ...
    Button editButton = ...;
    editButton.Clicked += OnClick;
}

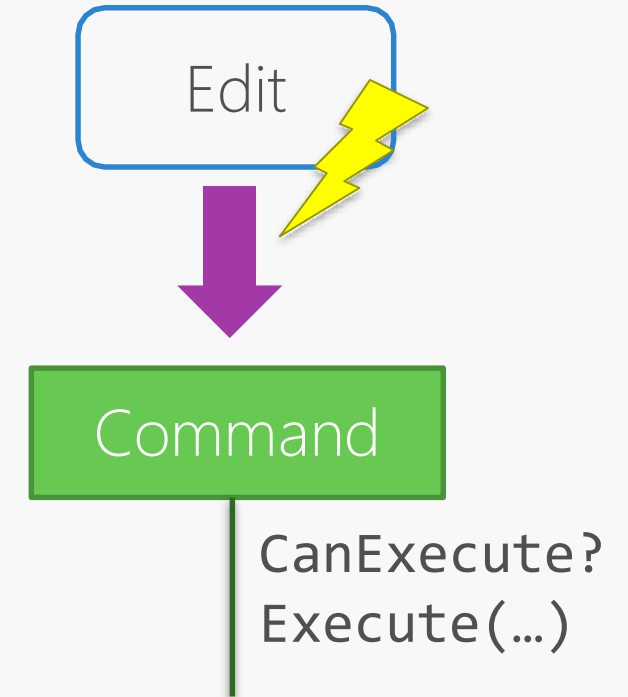
void OnClick (object sender, EventArgs e)
{
    ...
}
```

# Command

Microsoft ha definito l'interfaccia  **ICommand**  come astrazione per i normali Event Handler .NET

```
public interface ICommand
{
    bool CanExecute(object parameter);
    void Execute(object parameter);
    event EventHandler CanExecuteChanged;
}
```

Parametro opzionale (spesso **null**) per l'abilitazione/disabilitazione del comando



# Command in Xamarin.Forms

Alcuni controlli Xamarin.Forms supportano già l'utilizzo di Comandi per mezzo della proprietà **Command**, associata all'evento principale per quel controllo



Button



Menu



ToolbarItem



TextCell

# Command in Xamarin.Forms

Alcuni controlli Xamarin.Forms supportano già l'utilizzo di Comandi per mezzo della proprietà **Command**, associata all'evento principale per quel controllo

```
public ICommand LoginCommand { get; }
```

```
<Button Text="Login"  
        Command="{Binding LoginCommand}" />
```



E' possibile bindare la proprietà sorgente di tipo **ICommand** al target **Command** del controllo

# Command in Xamarin.Forms

Alcuni controlli Xamarin.Forms supportano già l'utilizzo di Comandi per mezzo della proprietà **Command**, associata all'evento principale per quel controllo


```
<Image Source="Logout.png">  
  <Image.GestureRecognizers>  
    <TapGestureRecognizer  
      Command="{Binding LogoutCommand}"  
      CommandParameter="{Binding .}" />  
  </Image.GestureRecognizers>  
</Image>
```

La proprietà **CommandParameter** accetta il parametro passato con il comando – in questo caso il **BindingContext** di **Image**

# Implementare Command

I **Command** devono essere esposti nel ViewModel come proprietà pubbliche – notificabili se si intende cambiare l'implementazione runtime

```
public class MainPageViewModel : INotifyPropertyChanged
{
    public ICommand LoginCommand { get; private set; }
    //...
    public MainPageViewModel()
    {
        LoginCommand = new MyCommand(this);
    }
    //...
}
```




```
public class MyCommand : ICommand
```

# Implementare Command

L'interfaccia  **ICommand**  espone tre campi da implementare

**CanExecute** viene  
chiamato per  
determinare se il  
comando è valido –  
se non valido, viene  
disabilitato  
(**IsEnabled=false**)  
il controllo  
corrispondente



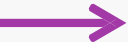
```
public interface ICommand
{
    bool CanExecute(object parameter);
    void Execute(object parameter);
    event EventHandler CanExecuteChanged;
}
```

# Implementare Command

L'interfaccia  **ICommand**  espone tre campi da implementare

**Execute** viene invocato per eseguire la logica associata al Command – eseguito subito dopo **CanExecute** solo se questo restituisce **true**

```
public interface ICommand
{
    bool CanExecute(object parameter);
    void Execute(object parameter);
    event EventHandler CanExecuteChanged;
}
```





# Implementare Command


L'interfaccia  **ICommand**  espone tre campi da implementare

L'evento

**CanExecuteChanged**

viene utilizzato per scatenare manualmente il controllo di validità del Command, in risposta al quale viene abilitato o disabilitato l'elemento

```
public interface ICommand
{
    bool CanExecute(object parameter);
    void Execute(object parameter);
    event EventHandler CanExecuteChanged;
}
```



# Implementare Command<T>

È possibile implementare una versione generica di Command che supporti il passaggio di parametri (**CommandParameter**) in **Execute** e **CanExecute**

```
public class Command<T> : ICommand
{
    Action<T> _function;
    public void Execute(object parameter) {
        _function.Invoke((T) parameter);
    }

    public bool CanExecute(object parameter) {...}
    public event EventHandler CanExecuteChanged;
}
```

# EventToCommand – Oltre le Azioni di Default

È possibile ricorrere a **Command** anche per azioni non di default

```
<ListView ItemsSource="{Binding Posts}">
  <ListView.Behaviors>
    <behaviors:EventToCommandBehavior EventName="ItemSelected"
                                     Command="{Binding NavigateCommand}"
                                     Converter="{StaticResource SelectedItemConverter}" />
  </ListView.Behaviors>
</ListView>
```

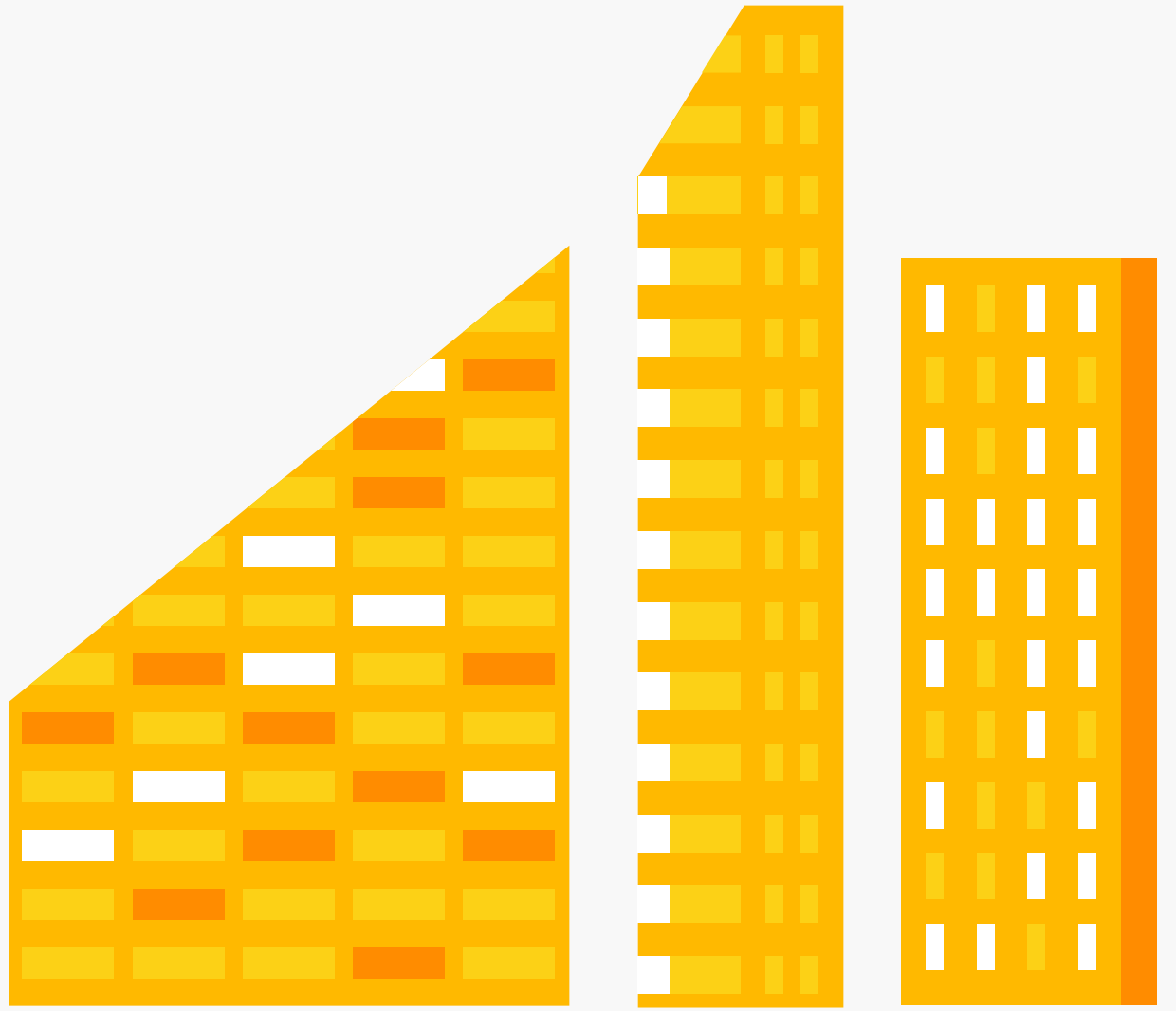
Diverse Implementazioni:

- [EventToCommandBehavior](#) di *David Britch*
- [Corcav.Behaviors](#) di *Corrado Cavalli*

# MVVM

## Demo

**GitHub Repo:** [bit.ly/2oPPsJ7](https://bit.ly/2oPPsJ7)



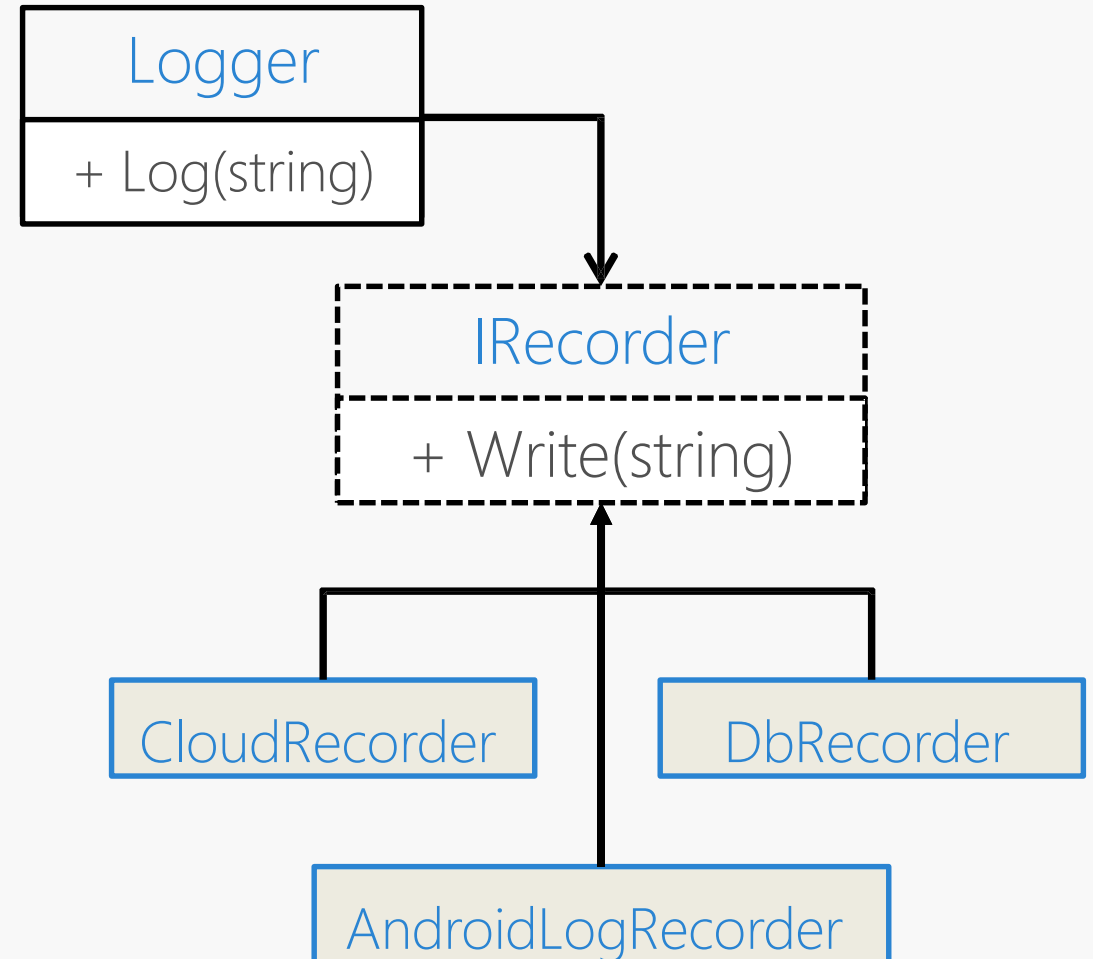
# Major Pattern in Xamarin

# Funzionalità Platform-Specific

Molti problemi comuni richiedono l'utilizzo di API platform-specific:

- Alert, Notification, Popup...
- Operazioni di I/O
- Esecuzione sullo UI Thread

Possibile utilizzare il Pattern Bridge per separare l'implementazione



# Astrazioni

Ogni piattaforma ha il proprio modo per riportare notifiche all'utente

Lo Shared Code utilizzerà l'astrazione

**IAAlertService**

Ogni piattaforma interessata deve implementare l'astrazione con le sue API specifiche

```
public interface IAAlertService
{
    bool Show(string title,
              string message,
              string yesButton,
              string noButton);
}
```

# Astrazioni - Utilizzo

Lo Shared Code deve essere consapevole esclusivamente dell-astrazione astratto

Necessario fornire  
un'implementazione  
di **IAlertService**

```
public class TerminatorViewModel
{
    ...
    public void TerminateJohnConner()
    {
        IAlertService alert = ??;
        if (!alert.Show("John Conner Located!",
            "Initiate termination sequence?",
            "Yes", "No")) { ... }
    }
}
```

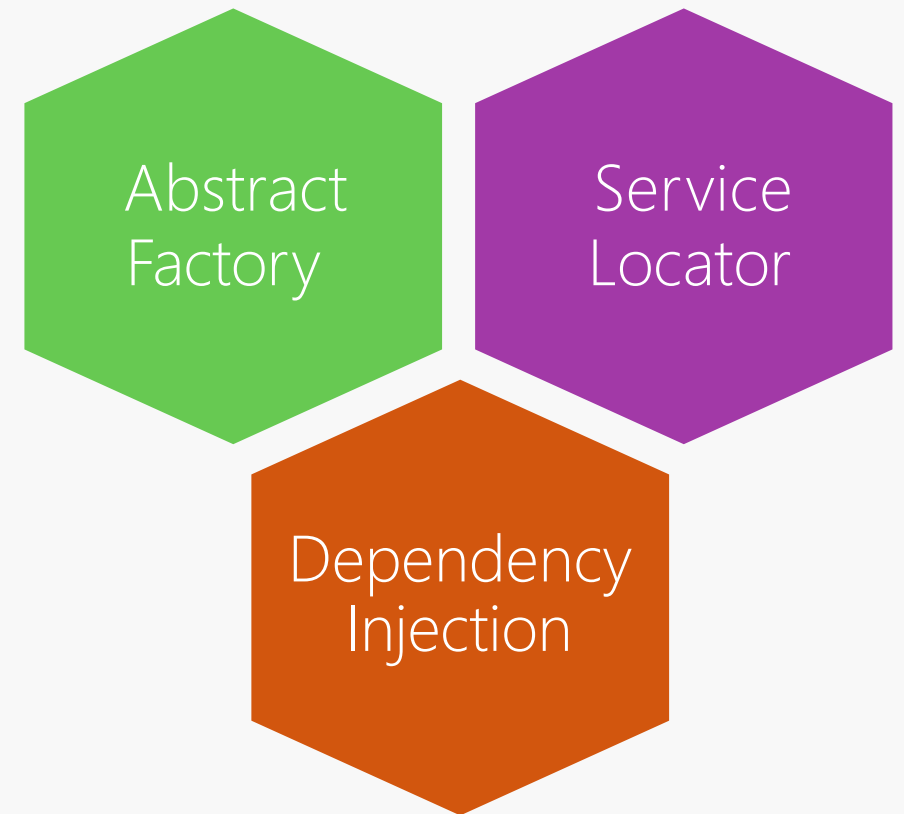


# Inversion of Control

Alcuni pattern possono essere utilizzati per "rompere" le dipendenze e disaccoppiare componenti

Noto anche come "Inversion of Control" (IoC)

In Xamarin utilizzabile per chiamare componenti platform-specific da Shared Code senza dipendere da questi



# Pattern Factory

Permette di localizzare le dipendenze attraverso servizi di Factory responsabili di creare le astrazioni



# Factory - Astrazione

Un delegato è esposto alle piattaforme specifiche che si preoccupano di fornire una propria implementazione di **AlertService**

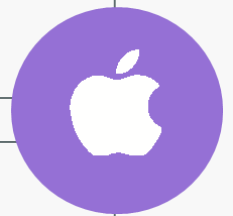
```
public abstract class AlertService {  
    // Factory Property  
    public static Func<AlertService> Create { get; set; }  
  
    public abstract bool Show(string title,  
                             string message,  
                             string yesButton,  
                             string noButton);  
}
```

# Factory - Implementazione

Ogni piattaforma implementa l'astrazione e assegna l'implementazione alla proprietà Factory (e.g. Create)

```
class AlertServiceiOS : AlertService
{
    public bool Show(string title, string message,
                    string yesButton, string noButton) { ... }
}
```


```
public override bool FinishedLaunching(...) {
    ...
    AlertService.Create = () => new AlertServiceiOS();
}
```



# Factory - Utilizzo

Il cliente che ha bisogno della particolare funzionalità nativa, delega alla Abstract Factory la creazione dell'oggetto da utilizzare

```
public void OnReceivedError(string errorMessage)
{
    var alertService = AlertService.Create();
    alertService.Show ("Error",
        $"Got error: {errorMessage}", "OK", null);
    ...
}
```



Il Cliente non si preoccupa dell'implementazione → Rovesciate le dipendenze (IoC)

# Abstract Factory - Pro e Contro

## PRO

- Possibilità di cambiare l'Implementazione runtime
- Facile da capire e usare
- Nasconde l'implementazione

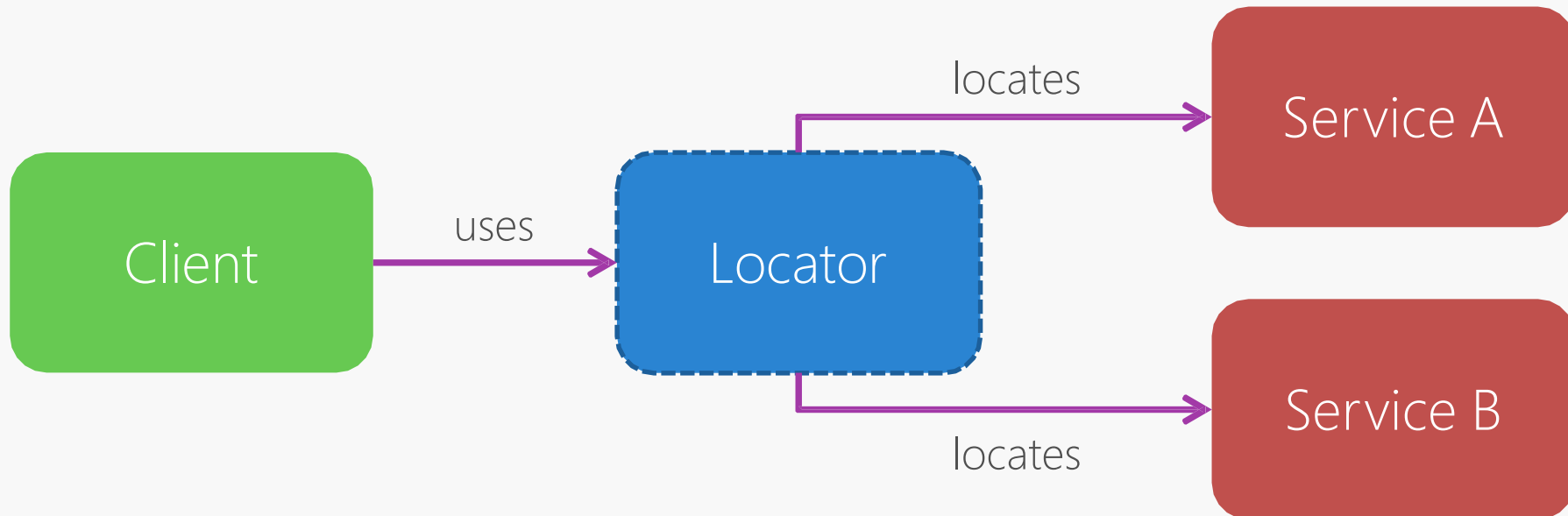
## CONTRO

- Richiede una Factory per ogni astrazione → Manutenibilità 😞
- Il Cliente deve dipendere dalla Factory
- Dipendenze iniettate prima della richiesta da parte del cliente

# Service Locator

Il pattern **Service Locator** richiede l'utilizzo di un Container che mappa le astrazioni nelle corrispondenti concretizzazioni registrate presso il Locator

Il Cliente usa il Locator per trovare le dipendenze



# Service Locator - Implementazione

Il **Service Locator** deve esporre metodi per registrare e risolvere (localizzare) le dipendenze

```
public sealed class ServiceLocator
{
    public static ServiceLocator Instance { get; set; }

    public void Add(Type contractType, object value);
    public void Add(Type contractType, Type serviceType);
    public object Resolve(Type contractType);
    public T Resolve<T>();
}
```



# Service Locator - Registrazione

Il codice Platform-Specific (e.g. l'entry point nativo) è responsabile di registrare i tipi concreti per le astrazioni

```
public partial class AppDelegate
{
    ...
    public override void FinishedLaunching(UIApplication application)
    {
        ...
        ServiceLocator.Instance.Add<IAlertService, MyAlertService>();
    }
}
```



# Service Locator - Utilizzo

Il cliente richiede l'astrazione e il Locator restituisce l'Implementazione registrata

```
public void DialNumber(string number)
{
    var alert = ServiceLocator.Instance.Resolve<IAlertService>();
    if (!alert.Show("Dial Number",
        "Are you sure you want to dial " + number,
        "Yes", "No")) { ... }
}
```

# Service Locator Comuni

Esistono diverse implementazioni di Service Locator già pronte:

- Common Service Locator  
[[commonservicelocator.codeplex.com](http://commonservicelocator.codeplex.com)]
- La maggior parte degli MVVM framework espone la propria Implementazione di Service Locator
- Xamarin.Forms **DependencyService**

# Service Locator - Pro e Contro

## PRO

- Dipendenze iniettate just-in-time alla richiesta del cliente
- Facile da capire e usare
- Utilizzabile da ogni cliente

## CONTRO

- Tutti i clienti devono avere accesso al Service Locator
- Più difficile identificare le dipendenze nel codice
- Più difficile scovare dipendenze non registrate

# Dependency Injection

Si delega il progetto platform-specific di "iniettare" la dipendenza passandola come parametro del costruttore o settando una proprietà

```
public class DataAccessLayer
```

```
{
```

```
    public DataAccessLayer(
```

```
        IDbRespository db,
```

```
        IAlertService alerts) { ... }
```

```
    public ILogger Logger { get; set; }
```

```
    ...
```

```
}
```

← Injection tramite  
costruttore

← Injection tramite  
proprietà

# Dependency Injection

Si può poi connettere manualmente il cliente alle dipendenze richieste iniettando le istanze concrete da progetto platform-specific

```
public DataAccessLayer CreateDataLayer()
{
    var dataAccessLayer = new DataAccessLayer(
        new SqliteRepository(), // IDbRepository
        new UWPAAlertService()); // IAlertService
    dataAccessLayer.Logger = new AzureLogger(); // ILogger

    return dataAccessLayer
}
```



# Inversion of Control (IoC) Container

Un **IoC Container** è un gestore di dipendenze utilizzato per:

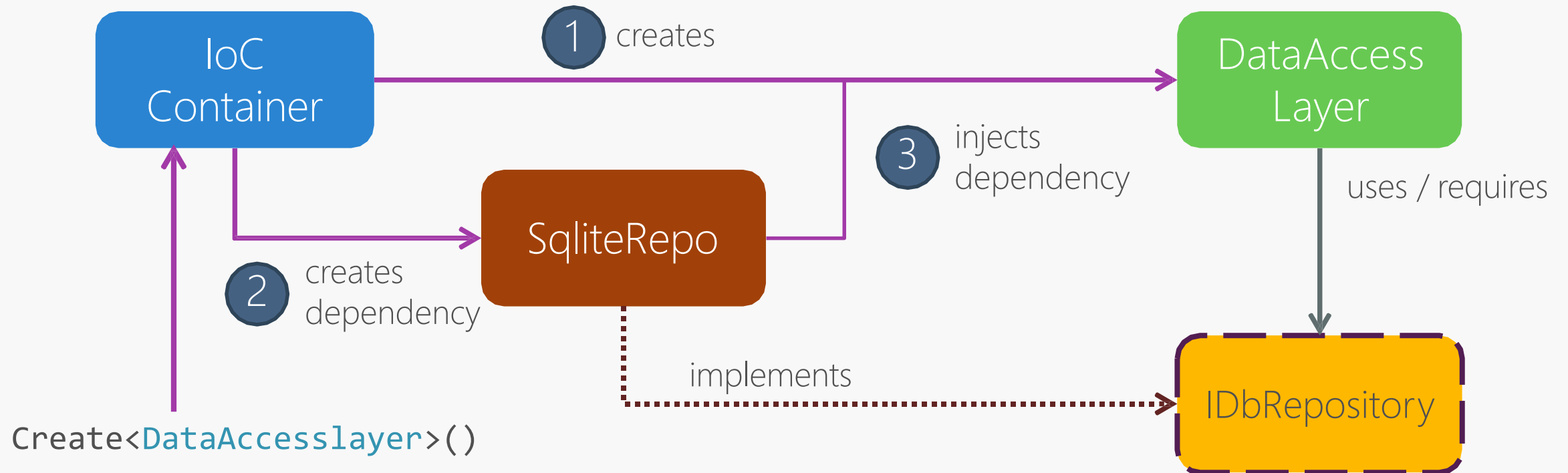
- Creare e ospitare le dipendenze (**Service Locator**)
- Iniettarle JIT quando necessario (**Dependency Injection**)
- Controllarne il ciclo di vita nell'applicazione

Fa da "registry" per  
dipendenze note

Crea gli oggetti e li  
"inietta" quando  
necessario

# DI con IoC Container

Si può automatizzare la DI utilizzando un Container che registra le dipendenze e ne istanzia i tipi concreti, assegnandoli automaticamente come parametri del costruttore o proprietà

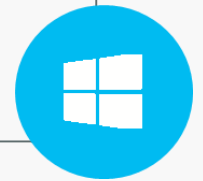




# DI Container - Esempio

Le dipendenze sono tipicamente registrate nel codice platform-specific

```
MyContainer container = new MyContainer();  
container.Register<IDbRepository, SqliteRepository>();  
container.Register<IAlertService, UWPAlertService>();  
container.Register<ILogger>(new AzureLogger(AzureToken));  
container.Register<MessageBus>(new MessageBus(this));
```



```
var dataLayer = container.Create<DataAccessLayer>();  
...
```



Si richiede al Container di creare **DataAccessLayer** – questo risolverà automaticamente **IDbRepository** e **IAlertService** da cui dipende

# DI con IoC Container - Pro e Contro

## PRO

- Riferimenti al Container non necessari nel cliente
- Facile identificare le dipendenze dato che queste vengono passate nel costruttore o assegnate a proprietà

## CONTRO

- Richiede un certo grado di architettura iniziale
- Implementazione basata su Reflection → Performance Issue

# IoC Containers

Esistono diverse implementazioni di Container:

- TinyIoC
- Ninject
- AutoFac
- Unity
- MvvmCross
- ...

# Prism

# Un po' di storia

**Prism** è un progetto nato per fornire uno standard per lo sviluppo di applicazioni che seguono il pattern MVVM

Precedentemente noto come Composite Application Library

Nel 2015 reso open-source assieme alle librerie Prism.WPF, Prism.Windows e **Prism.Forms**

[github.com/PrismLibrary](https://github.com/PrismLibrary)



# Panoramica

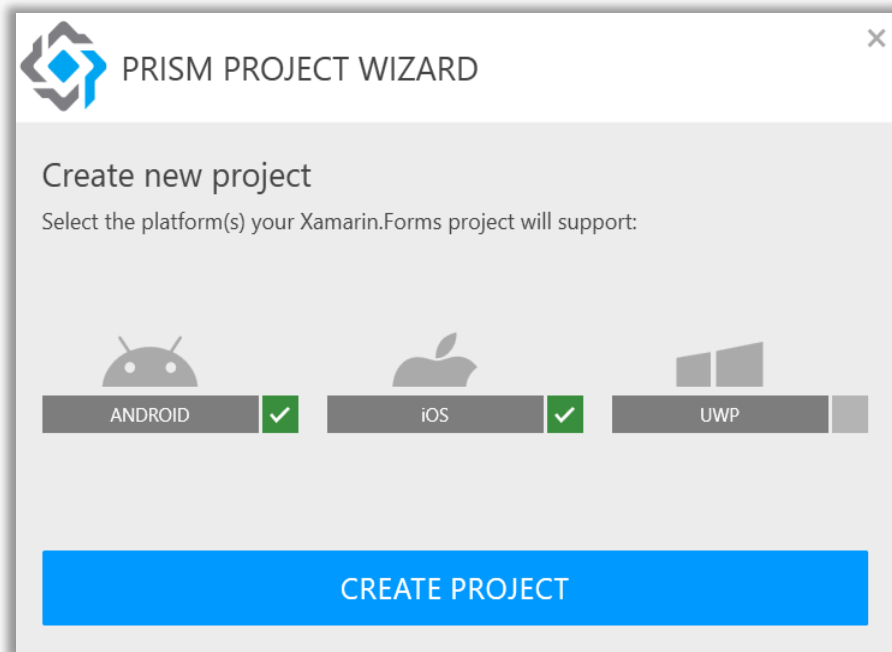
Tra le diverse funzionalità offerte da Prism:


- Supporto a MVVM
- Commanding
- Messaging
- Navigation Service
- Page Dialog Service
- Logging
- Dependency Injection
- Supporto a diversi IOC Container





# Installazione

- a. Per progetti esistenti:
- Installare da Nuget i pacchetti Prism.Core, Prism.Forms e Prism.Unity.Forms



 **Prism.Core** by Brian Lagunas, Brian Noyes  
Prism provides an implementation of a collection of design patterns that are helpful in writing well structured and maintainable applications.

 **Prism.Forms** by Brian Lagunas, Brian Noyes  
Prism for Xamarin.Forms helps you more easily design and build rich, flexible, and easy to maintain Xamarin.Forms applications.

 **Prism.Unity.Forms** by Brian Lagunas, Brian Noyes  
Unity extensions for Prism for Xamarin.Forms.

- b. Per nuovi progetti:
- Scaricare ed installare l'estensione per VS **Prism Template Pack**
  - Scegliere il Template di progetto **Prism App (Xamarin.Forms)**
  - Selezionare le piattaforme da supportare

# ViewModel

In Prism ogni ViewModel deve ereditare da **BindableBase**, classe astratta che semplifica l'implementazione di **INotifyPropertyChanged**

```
public abstract class BindableBase : INotifyPropertyChanged {
    public event PropertyChangedEventHandler PropertyChanged;
    protected virtual bool SetProperty<T>(ref T storage, T value,
                                           [CallerMemberName] string propertyName = null) {
        if (object.Equals((object) storage, (object) value))
            return false;
        storage = value;
        this.RaisePropertyChanged(propertyName);
        return true;
    }
    protected void RaisePropertyChanged([CallerMemberName] string propertyName = null) {
        this.OnPropertyChanged(propertyName);
    }
}
```



# AutoWiring del ViewModel

Prism adotta la seguente convenzione di Naming per ogni ViewModel:

View Name + "ViewModel" = ViewModel Name

Per ogni View Xamarin.Forms, è possibile delegare direttamente a Prism il collegamento al ViewModel corrispondente - **AutoWiring**

```
<ContentPage xmlns:prism="clr-namespace:Prism.Mvvm;assembly=Prism.Forms"
              prism:ViewModelLocator.AutowireViewModel="True"
              x:Class="MyReddit.Views.MainPage">
    ...
</ContentPage>
```

# PrismApplication

In Prism, la classe astratta **PrismApplication** prende il posto di Application come classe base e entry-point per l'applicazione Xamarin.Forms

Responsabile di:

- Inizializzare il Visual Tree Xamarin.Forms procedendo con la prima navigazione
- Gestire il Container utilizzato (Unity, Ninject, AutoFac...)
- Registrare i tipi necessari presso il Container
- Registrare le pagine dell'applicazione presso il NavigationService

# PrismApplication

In Prism, la classe astratta **PrismApplication** prende il posto di Application come classe base e entry-point per l'applicazione Xamarin.Forms

T rappresenta il  
Tipo del Container

Sostituisce a tutti  
gli effetti il  
costruttore della  
classe App

```
public abstract class PrismApplication<T> : Application
{
    public T Container { get; protected set; }

    protected INavigationService NavigationService { get; set; }

    protected abstract void OnInitialized();
    protected abstract void RegisterTypes();
    // ...
}
```

Registrazione delle pagine e dei tipi astratti da iniettare nei VM

# Unity Container

La classe **UnityContainer** espone i metodi e proprietà tipici di un IoC container, tra cui **Register** con supporto a **LifetimeManager** e **Resolve**

```
public class UnityContainer : IUnityContainer
{
    IEnumerable<ContainerRegistration> Registrations { get; }

    IUnityContainer RegisterType(Type from, Type to, string name,
                                LifetimeManager lifetime);
    IUnityContainer RegisterInstance(Type t, string name, object instance,
                                    LifetimeManager lifetime);

    object Resolve(Type t, string name, ...);
    IEnumerable<object> ResolveAll(Type t, ...);
    // ...
}
```

# Unity Container - LifetimeManager

**LifeTimeManager** permette di controllare come verrà iniettato l'istanza

Diverse possibilità:

- **TransientLifetimeManager**: Viene restituita una nuova istanza ad ogni Resolve, ResolveAll – modalità di default per **RegisterType**
- **ContainerControlledLifetimeManager**: Viene restituita la stessa istanza ad ogni Resolve, ResolveAll - modalità di default per **RegisterInstance**
- **HierarchicalLifetimeManager**: Come ContainerControlled ma istanze non condivise nella gerarchia di dipendenze

Altre modalità: [bit.ly/2rjGGQK](https://bit.ly/2rjGGQK)

# App Bootstrap

```
public partial class App : PrismApplication
{
    protected override async void OnInitialized()
    {
        InitializeComponent();
        await NavigationService.NavigateAsync("MainPage");
    }

    protected override void RegisterTypes()
    {
        Container.RegisterType<IWebApiSource, WeatherApiSource>(
            new ContainerControlledLifetimeManager());

        Container.RegisterTypeForNavigation<RootMasterDetailPage>();
    }
}
```

# ViewModel con DI

Prism automatizza l'injection nei ViewModel dei tipi registrati presso il Container – purchè questi ereditino da **BindableBase**

```
public class MainPageViewModel : BindableBase
{
    public MainPageViewModel(INavigationService navigationService,
                             IRedditApiSource redditApiSource)
    {
        _navigationService = navigationService;
        _redditApiSource = redditApiSource;
    }
}
```

# Navigation Service

Prism espone un servizio di navigazione tra pagine **INavigationService**

- Nessun riferimento esplicito tra View e ViewModel → Testabilità del VM
- Basato su URI assoluti e relativi che vengono mappati in percorsi di navigazione - **Deep Linking**

Occorre però registrare la pagina presso il Navigation Service:

```
// In App.cs
protected override void RegisterTypes()
{
    Container.RegisterTypeForNavigation<MainPage>();
}
```



# Navigation Service

Prism espone un servizio di navigazione tra pagine **INavigationService**

```
public interface INavigationService {  
    Task<bool> GoBackAsync(NavigationParameters parameters,  
                           bool? useModalNavigation, bool animated);  
    Task NavigateAsync(Uri uri, NavigationParameters parameters,  
                       bool? useModalNavigation, bool animated);  
    Task NavigateAsync(string name, NavigationParameters parameters,  
                       bool? useModalNavigation, bool animated);  
}
```

```
// In PageViewModel.cs
```

```
await NavigationService.NavigateAsync("SecondPage");  
  
await NavigationService.GoBackAsync();
```

# Eventi di Navigazione

Implementando l'interfaccia **INavigationAware**, è possibile rovesciare la gestione degli eventi OnAppearing e OnDisappearing dalla Pagina al PageViewModel corrispondente

```
public interface INavigatedAware : BindableBase, INavigationAware {
public class MainPageViewModel : BindableBase, INavigationAware {
{
    public void OnNavigatedFrom(NavigationParameters parameters) {
        void OnNavigatedFrom(NavigationParameters parameters);
        // Handle OnNavigatedFrom ~ OnDisappearing
    }
    public void OnNavigatedTo(NavigationParameters parameters) {
        void OnNavigatedTo(NavigationParameters parameters);
        // Handle OnNavigated ~ OnAppearing
    }
}

public void OnNavigatingTo(NavigationParameters parameters) {
    // Handle OnNavigating. e.g Load data asynchronously from DB or web service
}

public interface INavigationAware : INavigatedAware, INavigatingAware {
{
}
```

# Parametri di Navigazione

Prism supporta il passaggio di parametri di navigazione tra Pagine sfruttando la lista chiave-valore **NavigationParameters**

```
var navParams = new NavigationParameters
{
    { "param1", "prism" },
    { "param2", "mvvm" }
};

// Or using URI Syntax
var navParams = new NavigationParameters("param1=prism&param2=mvvm");

await NavigationService.NavigateAsync("SecondPage", navParams);
// Or using URI Syntax
await NavigationService.NavigateAsync("SecondPage?param1=prism&param2=mvvm");
```

# Parametri di Navigazione

Prism supporta il passaggio di parametri di navigazione tra Pagine sfruttando la lista chiave-valore **NavigationParameters**

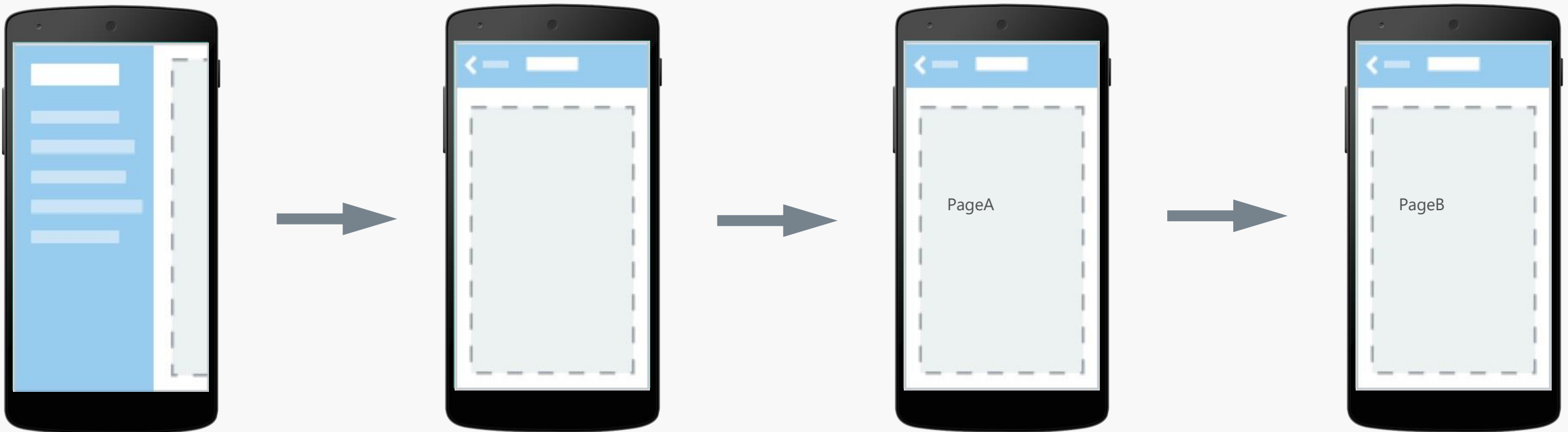
Nel PageViewModel Target, effettuare l'unmarshalling del parametro passato

```
// In Target Page ViewModel
public void OnNavigatingTo(NavigationParameters parameters)
{
    string param1;
    if (parameters.ContainsKey("param1"))
    {
        param1 = (string) parameters["param1"];
    }
}
```

# Deep Linking

Caratteristica del servizio di navigazione di Prism che permette di mappare URI in path di navigazione composti

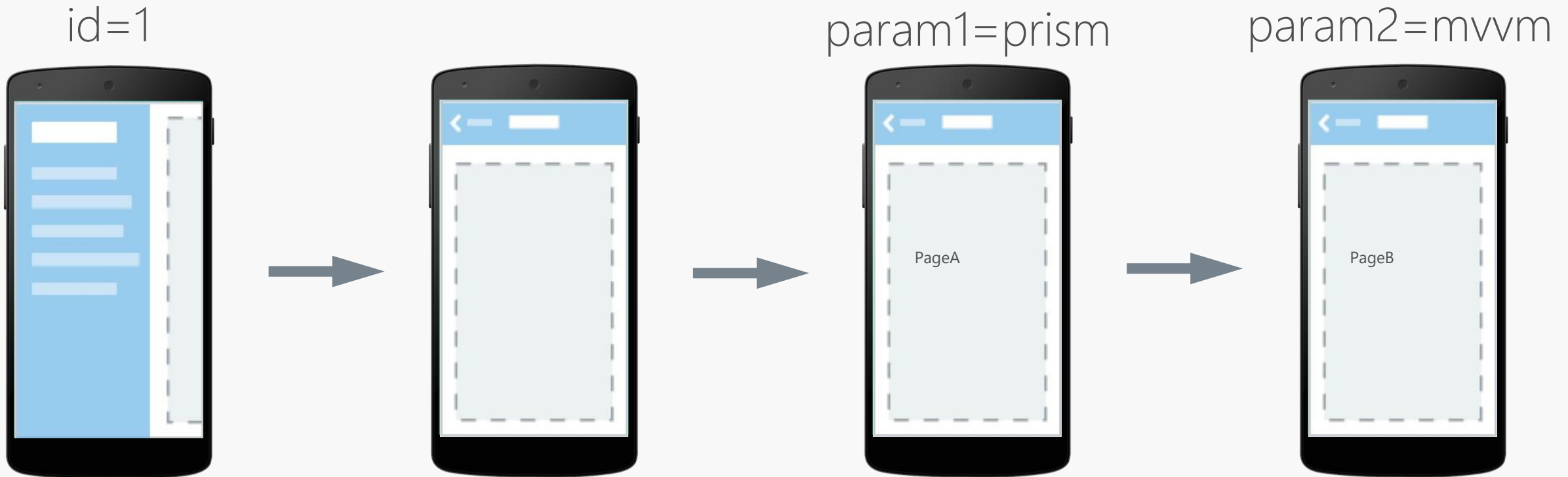
```
NavigateAsync("MasterDetailPage/NavigationPage/PageA/PageB")
```



# Deep Linking con Parametri

Caratteristica del servizio di navigazione di Prism che permette di mappare URI in path di navigazione composti

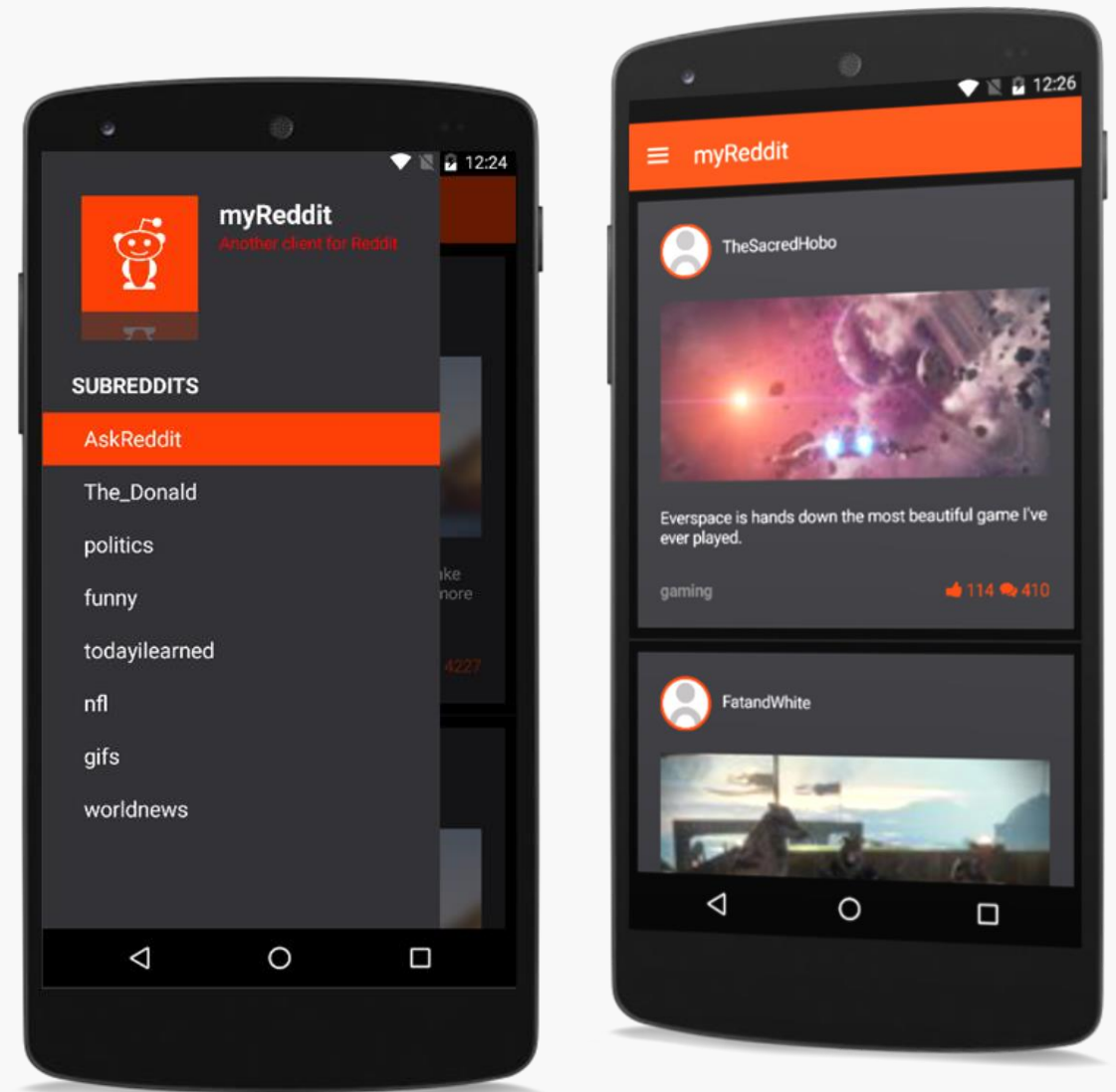
```
NavigateAsync("MasterDetailPage?id=1/NavigationPage/PageA?param1=prism/PageB?param2=mvvm")
```



# Prism

# Demo

**GitHub Repo:** [bit.ly/2qYXvD2](https://bit.ly/2qYXvD2)



FINE

Francesco Bonacci

[francesco.bonacci@outlook.com](mailto:francesco.bonacci@outlook.com)

Twitter: @francedot

[github.com/francedot](https://github.com/francedot)



# Xamarin per Principianti



Serie su Channel9: [channel9.msdn.com/Series/Xamarin-per-principianti](https://channel9.msdn.com/Series/Xamarin-per-principianti)

Repo: [github.com/francedot/xamarin-for-beginners](https://github.com/francedot/xamarin-for-beginners)