



1.	Introducció	2
2.	El patró de disseny MVVM	2
2.1.	Exemple de MVVM i Data Binding.	3
2.2.	Altre exemple amb una classe més complexa, amb col·leccions.	6
2.3.	L'ara MVVM amb base de dades.....	9
3.	Imatges.....	15
3.1.	Tipus d'imatges.....	15
3.2.	Formats d'imatge.....	16
3.3.	Resolució i profunditat del color	17
3.4.	Grandària i compressió d'imatges.....	19
4.	Programari per a la gestió de recursos gràfics.....	19
4.1.	Programari de visualització d'imatges	20
4.2.	Programari d'edició d'imatges.....	21
4.3.	Programari de creació d'imatges.....	21
4.4.	Programari: logotipus i icones	22
5.	Les imatges i la llei de propietat intel·lectual	23
5.1.	Drets de la propietat intel·lectual.....	23
5.2.	Drets d'autor	23
5.3.	Llicències	24
5.4.	Registre de contingut.....	25
6.	Formes i dibuix bàsic amb WPF	26
6.1.	Objectes Shape.	26
6.2.	Traçats i geometries.	28
	PathGeometry i PathSegments	28
	Sintaxi abreujada de XAML	30
6.3.	Pintar formes.	31
6.4.	Transformar formes.	32
7.	Generació de Gràfics	33
8.	Tipus de Gràfics.....	33
9.	WPF Toolkit Charting	34
10.	Llibreria Live Charts	36
10.1.	LiveChart amb base de dades.....	37

1. Introducció

En aquest punt farem una introducció al patró de disseny MVVM. Aquest patró és el més utilitzat en projectes WPF, Xamarin i eines per a desenvolupar aplicacions multiplataforma.

També farem una incursió en els elements gràfics de les aplicacions, tant imatges o elements simples com gràfics de dades.

2. El patró de disseny MVVM

Tots els desenvolupadors de programari estem en sempre buscant el codi perfecte. La veritat és que mai és possible aconseguir-ho, ja que per moltes raons, a vegades externes, a vegades per la nostra limitació de coneixement o simplement perquè la nostra manera de codificar sempre està en constant evolució (o almenys deuria), és impossible obtindre-ho.

D'igual manera, una altra cosa que és certa és que sempre podem intentar-ho, sempre podem tractar d'aprendre més, d'usar tècniques perquè siga més escalable, més fàcil de llegir, més eficient, etc. Una d'aquestes tantes maneres de fer millor el nostre codi és utilitzar patrons de disseny.

Però que és un patró de disseny?

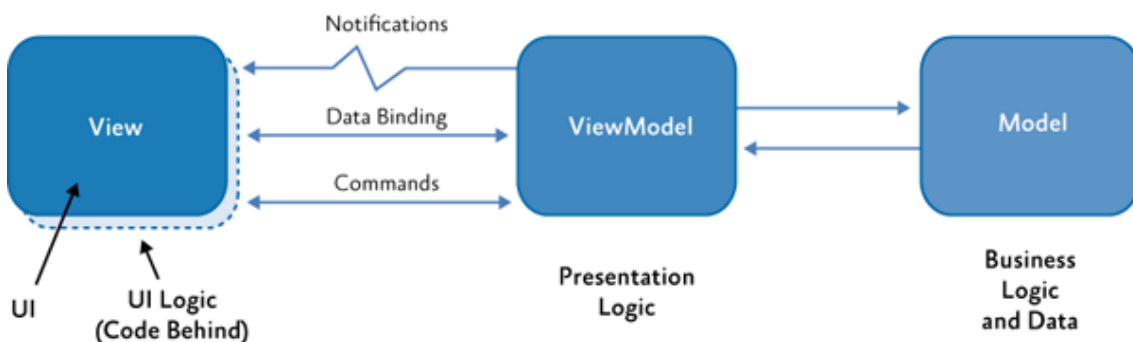
Per definició, els patrons de disseny són:

“Tècniques per a resoldre problemes comuns en el desenvolupament de programari i altres àmbits referents al disseny d'interacció o interfícies.”

Existeixen molts patrons de disseny, alguns es pot utilitzar combinats entre si i el seu ús dependrà del problema que vulguem resoldre amb el nostre programari.

Fins ara havíem utilitzat el clàssic patró de disseny MVC, on es separa el Model (classes), la Vista (fitxer xaml) i el controlador (codi C# associat al fitxer XAML).

Molt bé, doncs el model MVVM dona un pas més enllà, integrant el controlador dins del model (Business Logic) i creant una capa adient anomenada ViewModel.



La idea principal és que automàticament les dades del model és mostren directament en la vista mitjançant el **Data Binding** i si volem complicar-nos més gestionar els esdeveniments sense codi amb els Commands, encara que aquesta última part carregar el sistema i el codi és torna bastant farragós.

Des de el punt de vista del nostre cas (l'estudi de les interfícies) ens interessa implementar el patró MVVM en els nostres projectes d'una forma més lleugera. El nostre objectiu ha de ser que el Data Binding ens proporcione una sincronització entre el nostre model i la nostra vista.

Per a aplicar aquest model MVVM lleuger hem de seguir una sèrie de passos perquè no ens falle la sincronització i el Data Binding siga possible. L'ordre dels mateixos no és important, però hem de fer-los tots per a evitar problemes.

També dir-vos que aquest patró te els seus defensors i detractors, simplement és un patró diferent i una forma diferent de treballar. Comencem:

2.1. Exemple de MVVM i Data Binding.

Partirem d'un projecte amb una classe i una finestra:

Aquesta serà la classe:

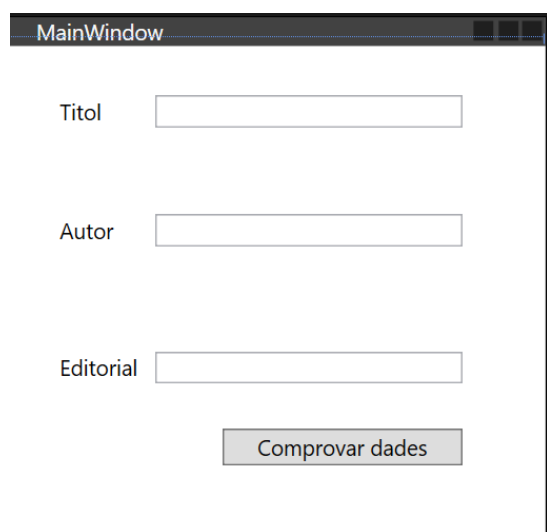
```
public class llibre
{
    private string titol;

    private string autor;

    private string editorial;

}
```

I aquesta seria la finestra per a omplir les dades d'un objecte de la classe:



The image shows a screenshot of a Windows application window titled "MainWindow". Inside the window, there are three text input fields arranged vertically. The first field is labeled "Titol", the second "Autor", and the third "Editorial". Below these fields is a button labeled "Comprovar dades". The window has a standard Windows title bar with minimize, maximize, and close buttons.

El botó "Comprovar dades" l'utilitzarem per verificar que el binding ambdues sentits funciona.

Aquesta classe bàsica hem de modificar-la d'aquesta forma:

- Primer afegirem altra classe dins del mateix fitxer. Serà aquesta:

```
1 referencia
public class MVBase : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;

    3 referencias
    protected void OnPropertyChanged(string propertyName)
    {
        PropertyChangedEventHandler handler = PropertyChanged;
        if (handler != null)
        {
            handler(this, new PropertyChangedEventArgs(propertyName));
        }
    }
}
```

Aquesta classe ha d'heretar de la classe **INotifyPropertyChanged** que és la que fa realitat la notificació dels canvis als objectes de classe als controls on estiguin enllaçats (per exemple amb un **binding**).

També ha d'implementar el mètode **OnPropertyChanged** que propagarà la notificació o esdeveniment de modificació de propietat amb el nom de la propietat afectada.

Aquesta classe podem prendre-la com una classe base i que las classes del nostre projecte MVVM hereten d'ella (és el cas d'aquest exemple) o bé que classe principal (en aquest cas llibre) herete directament de **INotifyPropertyChanged** i incloga el codi que hi ha dins d'aquesta classe.

- A continuació modificarem la classe d'aquesta forma:

```
2 referencias
public class Llibre : MVBase
{
    private string titol;

    2 referencias
    public string Titol
    {
        get { return titol; }
        set
        {
            titol = value;
            OnPropertyChanged("Titol");
        }
    }

    private string autor;

    2 referencias
    public string Autor
    {
        get { return autor; }
        set
        {
            autor = value;
            OnPropertyChanged("Autor");
        }
    }

    private string editorial;

    2 referencias
    public string Editorial
    {
        get { return editorial; }
        set
        {
            editorial = value;
            OnPropertyChanged("Editorial");
        }
    }
}
```

Observa com cada set de cada propietat es llança el mètode **OnPropertyChanged** de classe base amb el nom de la propietat modificada.

Observa també que aquesta classe hereta de la classe base abans creada.

Tornem a la finestra, el codi C# de la finestra seria aquest:

```
public partial class MainWindow : Window
{
    llibre millibre = new llibre();

    0 referencias
    public MainWindow()
    {
        InitializeComponent();

        millibre.Titol = "L'institut";
        millibre.Autor = "Stephen King";
        millibre.Editorial = "Andana Llibres";

        this.DataContext = millibre;
    }

    1 referencia
    private void Button_Click(object sender, RoutedEventArgs e)
    {
        MessageBox.Show(millibre.Titol + " -- " + millibre.Autor + " -- " + millibre.Editorial);
    }
}
```

Repasem un poc aquest codi:

- Primer creem un objecte de la classe llibre.
- Després, al constructor de la finestra, assignem contingut a les propietats de l'objecte **millibre** i l'indiquem que l'univers o context de dades és precisament l'objecte **millibre** (podria ser qualsevol objecte, inclús la pròpia finestra).
- Per últim, al mètode que és llançat al pulsar el botó "Comprovar dades" verifiquem que al canviar el contingut dels textbox (és on carregarem les propietats de l'objecte) és modifiquen automàticament les propietats de l'objecte **millibre**.

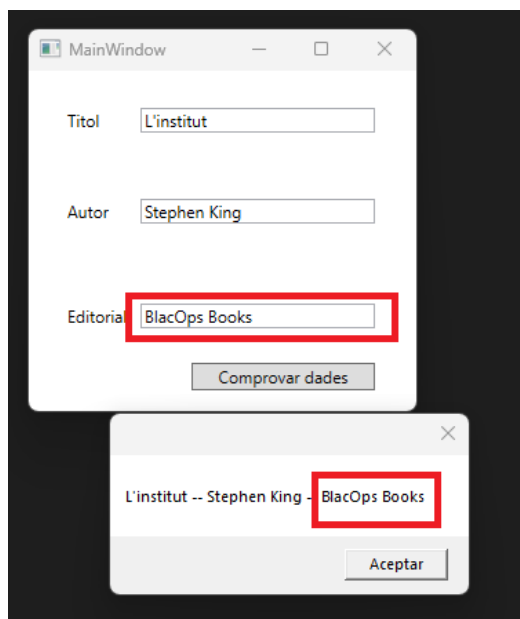
El codi XAML corresponent a la finestra seria aquest:

```
<Window x:Class="Ud6_MVVM.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:Ud6_MVVM"
        mc:Ignorable="d"
        Title="MainWindow" Height="286" Width="297">

    <Grid>
        <Label Content="Titol" HorizontalAlignment="Left" Margin="23,23,0,0" VerticalAlignment="Top"/>
        <Label Content="Autor" HorizontalAlignment="Left" Margin="23,89,0,0" VerticalAlignment="Top"/>
        <Label Content="Editorial" HorizontalAlignment="Left" Margin="23,165,0,0" VerticalAlignment="Top"/>
        <TextBox x:Name="txtTitol" HorizontalAlignment="Left" Margin="81,27,0,0" TextWrapping="Wrap"
            Text="{Binding Titol, Mode=TwoWay}" VerticalAlignment="Top" Width="170"/>
        <TextBox x:Name="txtAutor" HorizontalAlignment="Left" Margin="81,93,0,0" TextWrapping="Wrap"
            Text="{Binding Autor, Mode=TwoWay}" VerticalAlignment="Top" Width="170"/>
        <TextBox x:Name="txtEditorial" HorizontalAlignment="Left" Margin="81,169,0,0" TextWrapping="Wrap"
            Text="{Binding Editorial, Mode=TwoWay}" VerticalAlignment="Top" Width="170"/>
        <Button Content="Comprovar dades" HorizontalAlignment="Left" Margin="118,212,0,0"
            VerticalAlignment="Top" Click="Button_Click" Width="133"/>
    </Grid>
</Window>
```

Observem els Textbox. Cada textbox te enllaçada (amb un **binding**) la propietat text a una propietat de l'objecte. A més està enllaçada en mode TwoWay, és a dir els canvis es produiran ambdues sentits, de l'objecte cap al textbox i del textbox cap a l'objecte.

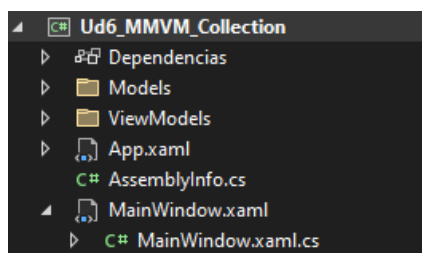
I si modifiquem qualsevol Textbox i posem el botó, el MessageBox ens mostra que l'objecte miLlibre també s'ha modificat:



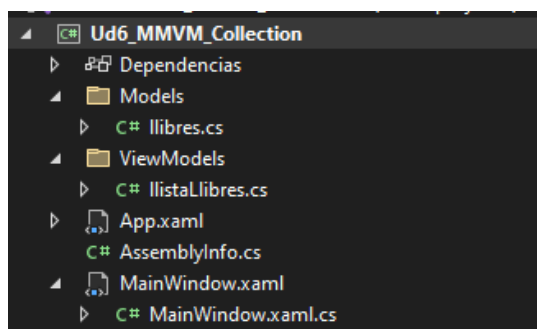
2.2. Altre exemple amb una classe més complexa, amb col·leccions.

El següent nivell ja seria un model MVVM molt més diferenciat (en aquest cada part del patró és clarament un fitxer) i amb una col·lecció d'objectes.

Comencem creant el projecte WPF com sempre, però aquesta vegada crearem dues carpetes per a emmagatzemar el model i la vista-model (viewmodel). D'aquesta forma també veurem com treballar amb fitxers amb classes que estan dins d'una carpeta. A aquestes carpetes les anomenarem **Models** i **ViewModels**:



Dins d'aquesta carpeta afegirem dues classes, **llibres.cs** i **llistaLlibres.cs**:



Comencem amb el fitxer de la classe llibres (**llibres.cs**), el seu contingut seria aquest:

```
public class Llibre : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;

    3 referencias
    protected void OnPropertyChanged(string propertyName)
    {
        PropertyChangedEventHandler handler = PropertyChanged;
        if (handler != null)
        {
            handler(this, new PropertyChangedEventArgs(propertyName));
        }
    }

    private string titol;

    4 referencias
    public string Titol
    {
        get { return titol; }
        set
        {
            titol = value;
            OnPropertyChanged("Titol");
        }
    }

    private string autor;

    4 referencias
    public string Autor
    {
        get { return autor; }
        set
        {
            autor = value;
            OnPropertyChanged("Autor");
        }
    }

    private string editorial;

    4 referencias
    public string Editorial
    {
        get { return editorial; }
        set
        {
            editorial = value;
            OnPropertyChanged("Editorial");
        }
    }
}
```

Com pots observar en aquesta ocasió ho fem de forma diferent, la classe hereta directament de la classe **INotifyPropertyChanged**, no tenim una classe base (és altra forma completament vàlida de plantejar MVVM). La resta de classe és exactament igual que l'exemple anterior.

Anem ara amb **llistaLlibres.cs**:

```
2 referencias
public class LlistaLlibres : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;

    1 referencia
    protected void OnPropertyChanged(string propertyName)
    {
        PropertyChangedEventHandler handler = PropertyChanged;
        if (handler != null)
        {
            handler(this, new PropertyChangedEventArgs(propertyName));
        }
    }

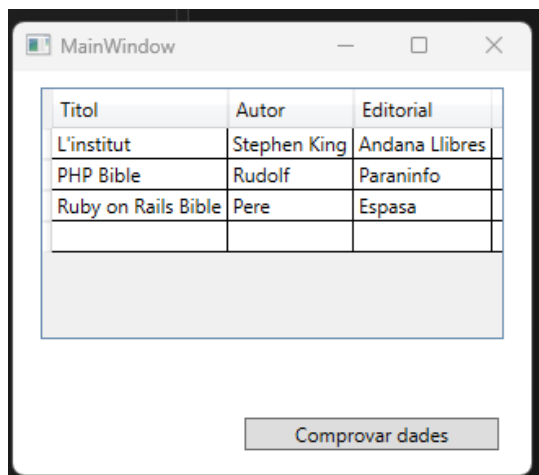
    private ObservableCollection<Llibre> _llistaLlibres = new ObservableCollection<Llibre>();

    4 referencias
    public ObservableCollection<Llibre> LlistaLlibres
    {
        get { return _llistaLlibres; }
        set
        {
            _llistaLlibres = value;
            OnPropertyChanged("LlistaLlibres");
        }
    }
}
```


Aquesta classe també ha d'heretar de la classe **INotifyPropertyChanged** i tindre a dins el mètode **OnPropertyChanged**, exactament igual que la classe llibre, ja que la llista també ha de notificar els canvis als controls enllaçats.

Per últim creem el constructor amb la llista d'objectes tipus llibre i el set i get corresponents. Observa que també és crida al mètode **OnPropertyChanged** quan és modifica la llista.

I per últim anem a per la finestra, que per aquest exemple seria aquesta:



En la que tenim la llista llibre dins d'un Datagrid i un botó amb el que comprovarem que les modificacions es fan ambdues sentits.

Mirem primer el codi C# de la finestra:

```
public partial class MainWindow : Window
{
    llibre millibre = new llibre();
    llibre millibre2 = new llibre();
    llibre millibre3 = new llibre();
    llista llista = new llista();

    0 referencias
    public MainWindow()
    {
        InitializeComponent();

        millibre.Titol = "L'institut";
        millibre.Autor = "Stephen King";
        millibre.Editorial = "Andana Llibres";
        llista.LlistaLlibres.Add(millibre);
        millibre2.Titol = "PHP Bible";
        millibre2.Autor = "Rudolf";
        millibre2.Editorial = "Paraninfo";
        llista.LlistaLlibres.Add(millibre2);
        millibre3.Titol = "Ruby on Rails Bible";
        millibre3.Autor = "Pere";
        millibre3.Editorial = "Espasa";
        llista.LlistaLlibres.Add(millibre3);

        dgLlibres.DataContext = millista;
    }

    1 referencia
    private void Button_Click(object sender, RoutedEventArgs e)
    {
        foreach (var llibre in millista.LlistaLlibres)
        {
            MessageBox.Show(llibre.Titol + " -- " + llibre.Autor + " -- " + llibre.Editorial);
        }
    }
}
```

Observa com instanciem 3 objectes de tipus llibre, i durant la carrega de la finestra (al constructor) omplim les seues propietats i els afegim a la llista **millista**.

A continuació indiquem al datagrid (**dgllibres**) que el seu univers de dades o context és la llista de llibres (**miLlista**).

Per acabar al pulsar el botó recorrem la llista de llibres per a comprovar si els canvis podríem haver fet al datagrid es reflecteixen a la llista **miLlista**.

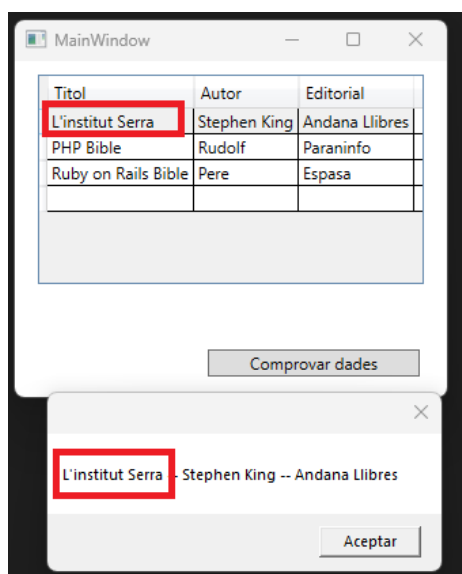
El contingut XAML de la finestra seria aquest:

```
Window x:Class="Ud6_MMVM_Collection.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:Ud6_MMVM_Collection"
mc:Ignorable="d"
Title="MainWindow" Height="286" Width="334">

    <Grid>
        <Button Content="Comprovar dades" HorizontalAlignment="Left" Margin="142,212,0,0" VerticalAlignment="Top" Click="Button_Click" Width="156"/>
        <DataGrid x:Name="dgllibres" ItemsSource="{Binding LlistaLlibres, Mode=TwoWay}" Margin="17,10,17,83"/>
    </Grid>
</Window>
```

Observa el binding associat a la propietat **ItemSource** del **Datagrid** i amb el **Mode TwoWay** (igual que a l'exemple anterior).

I ara si després de carregar el datagrid fem un xicotet canvi a la primera filera, observarem que aquest canvi també es reflecteix a la llista de llibres:



2.3. I ara MVVM amb base de dades.

Ara pasem a altre nivell que és un patró MVVM amb accés a base de dades. Amb les bases de dades poden entrar al patró MVVM altre element que encara no hem vist, els objectes **Commands**.

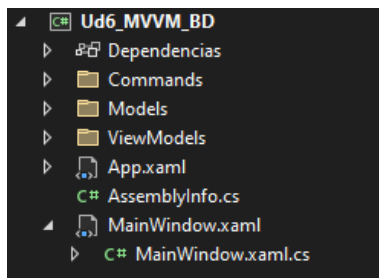
L'objecte **command** pretén reduir el codi C# associat als esdeveniments dels controls, per exemple el **Click** d'un botó. Per contra el que fa és embrutar un poc el codi XAML, ja que a dins de l'etiqueta **Button** (per exemple) cal afegir una propietat **Command** i altra **CommandParameter** com a mínim, a més d'afegir més complexitat a l'aplicació (per a aplicacions senzilles aquest model no val la pena).

Una forma d'aplicar un Command en un botó podria ser:

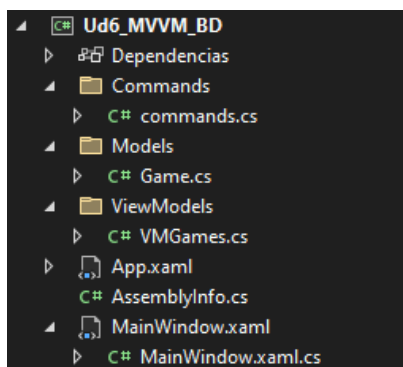
```
<Button Content="Create data" Command="{Binding Cmd}" CommandParameter="Create" Margin="5"/>
```

El que fem en aquesta etiqueta és enllaçar un objecte tipus ICommand (Cmd) amb el paràmetre create (més avant el veurem amb més detall).

Comencem a crear aquest projecte d'exemple (en aquest cas una llista de videojocs). El primer que hem fer en aquesta ocasió és crear 3 carpetes: **Models**, **ViewModels** i **Commands**:



Dins de cada carpeta crearem una classe: **Commands.cs**, **Game.cs** i **VMGame.cs**



Comencem per la més nova, la classe **commands.cs**:

```
3 referencias
public class RelayCommand : ICommand
{
    private readonly Predicate<object> _canExecute;
    private readonly Action<object> _action;
    1 referencia
    public RelayCommand(Action<object> action) { _action = action; _canExecute = null; }
    0 referencias
    public RelayCommand(Action<object> action, Predicate<object> canExecute) { _action = action; _canExecute = canExecute; }
    0 referencias
    public void Execute(object o) => _action(o);
    0 referencias
    public bool CanExecute(object o) => _canExecute == null ? true : _canExecute(o);
    public event EventHandler CanExecuteChanged
    {
        add { CommandManager.RequerySuggested += value; }
        remove { CommandManager.RequerySuggested -= value; }
    }
}
```

El nom de la classe és **RelayCommand**, però pot ser qualsevol. El requisit és que herete de la classe **ICommand**.

A continuació te dues objectes com a propietats per a limitar les accions o commands a realitzar i si poden executar-les.

Després tenim el constructor, escrit de dues formes diferents (al projecte únicament utilitzarem la primera) i la definicions d'alguns mètodes.

No cal estudiar molt aquesta classe, únicament hem de conèixer que hem de tindre-la al projecte quan vullgam utilitzar **Commands** al nostre patró MVVM.

Continuarem amb el model, **Game.cs**:

```
0 referencias
public class Game
{
    0 referencias
    public int GameID { get; set; }
    0 referencias
    public string Name { get; set; }
}
```

És una classe molt senzilla, ja que en aquest cas la complexitat estarà al ViewModel.

Vegem ara el **ViewModel** (o vista-model), **VMGames.cs**:

```
public class ViewModelGames : INotifyPropertyChanged
{
    private CollectionViewSource cvs = new CollectionViewSource();
    1 referencia
    public ICollectionView GameCollection { get => cvs.View; }

    0 referencias
    public ICommand Cmd { get => new RelayCommand(CmdExec); }

    public DataSet ds = new DataSet();
    1 referencia
    private void CmdExec(object parameter)
    {
        switch (parameter.ToString())
        {
            case "Create":
                CreateData();
                break;
            case "Load":
                ds = LoadData();
                cvs.Source = ds.Tables["Game1"].DefaultView;
                OnPropertyChanged(nameof(GameCollection));
                break;
            case "Check":
                foreach (DataRow row in ds.Tables["Game1"].Rows)
                {
                    MessageBox.Show("ID: " + row[0].ToString() + " -- Game Name: " + row[1].ToString());
                }
                break;
            default:
                break;
        }
    }

    1 referencia
    private void CreateData() { ... }

    1 referencia
    private DataSet LoadData() { ... }

    #region INotifyPropertyChanged Members
    public event PropertyChangedEventHandler PropertyChanged;

    1 referencia
    protected void OnPropertyChanged(string propertyName)
    {
        PropertyChangedEventHandler handler = PropertyChanged;
        if (handler != null)
        {
            handler(this, new PropertyChangedEventArgs(propertyName));
        }
    }
    #endregion
}
```

El primer que podem veure és que la classe hereta de **INotifyPropertyChanged**, com és normal en aquest patró.

En aquesta classe també tenim alguns mètodes més per a invocar als mètodes de classe **command.cs**, anem un per un:

- Definicions i mètode **CmdExec**:

```
private CollectionViewSource cvs = new CollectionViewSource();
1 referencia
public ICollectionView GameCollection { get => cvs.View; }

0 referencias
public ICommand Cmd { get => new RelayCommand(CmdExec); }

public DataSet ds = new DataSet();
1 referencia
private void CmdExec(object parameter)
{
    switch (parameter.ToString())
    {
        case "Create":
            CreateData();
            break;
        case "Load":
            ds = LoadData();
            cvs.Source = ds.Tables["Game1"].DefaultView;
            OnPropertyChanged(nameof(GameCollection));
            break;
        case "Check":
            foreach (DataRow row in ds.Tables["Game1"].Rows)
            {
                MessageBox.Show("ID:" + row[0].ToString() + " -- Game Name: " + row[1].ToString());
            }
            break;
        default:
            break;
    }
}
```

En aquesta part creem la **col·lecció** de jocs, el **command** (Cmd) i el **DataSet** per a emmagatzemar les dades de la base de dades.

També tenim el mètode **CmdExec**, és el mètode que enllaçarem (binding) als controls (en aquesta ocasió un botó) per a que se execute. Dins d'aquest mètode examinem amb un switch el paràmetre i executem el mètode necessari en funció del paràmetre.

- Mètode CreateData:

```
1 referencia
private void CreateData()
{
    using (SqlConnection cn = new SqlConnection(@"Data Source=PHOBOS\SQLEXPRESS;Initial Catalog=FASTFOOD;User ID=sa;Password=saroot"))
    {
        cn.Open();
        using (SqlCommand cmd = new SqlCommand { Connection = cn })
        {
            // delete previous table in SQL Server 2016 and above
            cmd.CommandText = "DROP TABLE IF EXISTS Game1;";
            // delete previous table in versions
            //cmd.CommandText = "If OBJECT_ID('Game1', 'U') IS NOT NULL DROP TABLE IF EXISTS Table1;";
            cmd.ExecuteNonQuery();
            // Create Tables
            cmd.CommandText = "CREATE Table Game1([GameID] Integer Identity, [Name] nvarchar(50), CONSTRAINT [PK_Game1] PRIMARY KEY ([GameID]));";
            cmd.ExecuteNonQuery();
            // Insert data records
            cmd.CommandText = "INSERT INTO Game1([Name]) VALUES('DOOM');INSERT INTO Game1([Name]) VALUES('QUAKE');INSERT INTO Game1([Name]) VALUES('HERETIC)";
            cmd.ExecuteNonQuery();
        }
    }
}
```

Aquest mètode simplement es connecta a la base de dades, esborra la taula si existeix i la genera de nou (és un forma de no dependre de si la base de dades està creada o no).

- Mètode LoadData:

```
1 referencia
private DataSet LoadData()
{
    using (SqlConnection cn = new SqlConnection(@"Data Source=PHOBOS\SQLEXPRESS;Initial Catalog=FASTFOOD;User ID=sa;Password=saroot"))
    {
        cn.Open();
        //Preparem la SQL
        String sql = "select * from Game1";
        //Llançem la SQL amb un DataAdapter
        SqlDataAdapter dataAdapter = new SqlDataAdapter(sql, cn);
        //Creem i omplim un DataSet amb la info del DataAdapter
        DataSet ds = new DataSet();
        dataAdapter.Fill(ds, "Game1");
        return ds;
    }
}
```

Aquest mètode recupera les dades de la base de dades i els associa al DataSet, que mitjançant la col·lecció enllaçarem al DataGrid al codi XAML de la finestra.

- Regió INotifyPropertyChanged

```
#region INotifyPropertyChanged Members
public event PropertyChangedEventHandler PropertyChanged;

1 referencia
protected void OnPropertyChanged(string propertyName)
{
    PropertyChangedEventHandler handler = PropertyChanged;
    if (handler != null)
    {
        handler(this, new PropertyChangedEventArgs(propertyName));
    }
}
#endregion
```

Una regió simplement és una part de codi que podem plegar i desplegar per a veure més contingut d'altre codi en pantalla. Dins d'aquesta regió tenim la propietat **PropertyChanged** i el mètode **OnPropertyChanged** necessari en aquest patró.

Per acabar vorem el codi XAML de la vista, la finestra **MainWindows.xaml**:

```
Window x:Class="Ud6_MVVM_BD.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:Ud6_MVVM_BD.ViewModels"
mc:Ignorable="d"
Title="MainWindows" Height="450" Width="800">
    <Window.DataContext>
        <local:ViewModelGames/>
    </Window.DataContext>
    <StackPanel margin="0,0,0,200">
        <Button Content="Create data" Command="{Binding Cmd}" CommandParameter="Create" Margin="5"/>
        <Button Content="Load data" Command="{Binding Cmd}" CommandParameter="Load" Margin="5"/>
        <DataGrid x:Name="dgGames" AutoGenerateColumns="False" ItemsSource="{Binding GameCollection}" Margin="5">
            <DataGrid.Columns>
                <DataGridTextColumn Header="GameID" Binding="{Binding GameID, Mode=OneWay}" Width="SizeToHeader" />
                <DataGridTextColumn Header="Game Name" Binding="{Binding Name, Mode=OneWay}" Width="SizeToHeader" />
            </DataGrid.Columns>
        </DataGrid>
        <Button Content="Check data" Command="{Binding Cmd}" CommandParameter="Check" Margin="5"/>
    </StackPanel>
</Window>
```

En aquesta ocasió indiquem el **DataContext** en aquest fitxer, en aquest cas serà tota la classe **ViewModelGames**.

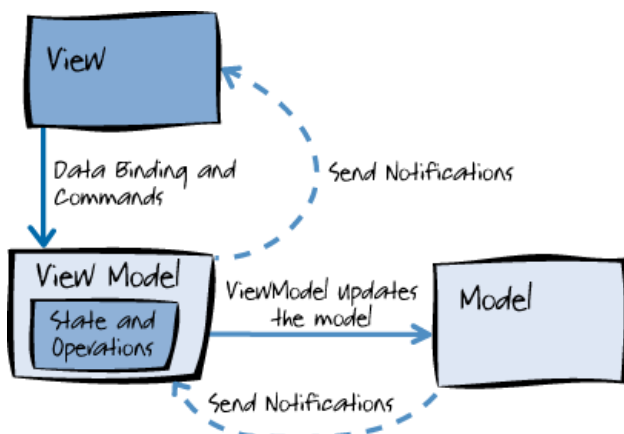
Observa com estan enllaçats els control buttons, amb un Binding al ICommand Cmd i al paràmetre l'acció a realitzar.

També enllacem el DataGrid amb la col·lecció (GameCollection) que ja està associada al DataSet al codi C# del ViewModel. I per acabar també enllacem les columnes del Datagrid als camps del DataSet.

Finalment si observem el codi C# d'aquesta finestra vegem que està net:

```
namespace Ud6_MVVM_BD
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    2 referencias
    public partial class MainWindow : Window
    {
        0 referencias
        public MainWindow()
        {
            InitializeComponent();
        }
    }
}
```

El patró aconseguit amb aquest exemple seria aquest:



Comentaris sobre aquest últim exemple:

El model Game.cs és prescindible (no se si s'heu donat compte), no s'utilitza a l'exemple. Simplement està per poder fer l'exemple més escalable, per exemple crean una col·lecció d'objectes de la classe Game.

Inclur Commands al disseny evita el codi associat a esdeveniments però enbruta el codi XAML i torna el disseny farragós per a aplicacions senzilles. No els recomane als projectes que fem en classe.

No es pretén que assimileu tot el codi d'aquest últim exemple, simplement heu de conèixer-lo i poder aplicar-lo amb altra base de dades copiant el codi reutilitzable (command.cs i alguns mètodes del ViewModel).

3. Imatges

L'ús d'imatges és molt important en el disseny gràfic de qualsevol mena d'interfície, ja que aquestes contribueixen favorablement a l'experiència de l'usuari, sempre que siguin de qualitat i s'adeqüen al contingut que s'està treballant. A més de complir uns certs requisits de qualitat i format, s'ha de tindre en compte l'autoria de les imatges, drets d'autor... Una de les característiques més importants que s'ha de tindre en compte és el format de les imatges, ja que d'aquesta manera es defineix la qualitat visual enfront del pes de les il·lustracions. En aquest primer apartat, ens centrarem en la imatge digital en els seus principals atributs.

Una imatge digital és una representació bidimensional d'una imatge utilitzant bits (uns i zeros). Depenent de si la resolució de la imatge és estàtica o dinàmica, pot tractar-se d'un gràfic rasteritzat o d'un gràfic vectorial.

3.1. Tipus d'imatges

Existeixen dues tipus d'imatges digitals, les vectorials i les de mapa de bits o mapes de bits, en funció de la mena d'aplicació en la qual es vaja a emprar es triarà un tipus o un altre. L'elecció es basa en diferents factors, com és el procés de creació de les imatges, ja que cadascuna d'elles requereix d'unes aplicacions i uns requisits determinats.

Les imatges de mapa de bits

Aquestes imatges, també dites de "raster", són aquelles formades per un conjunt de punts, anomenats píxels, on cadascun d'aquests punts conté un conjunt de valors que defineix un color uniforme. Per aquesta raó són indicades per a aquelles imatges en les quals és desitjable mostrar una gamma de colors molt àmplia i amb variacions precises de color i lluminositat.

La qualitat d'aquestes imatges depèn de la quantitat de píxels utilitzats en la seua representació.

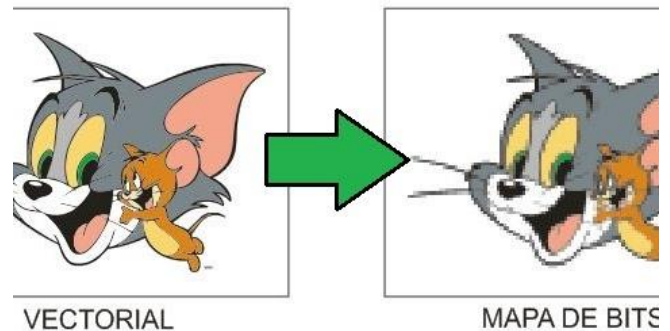
Una dels desavantatges principals de les imatges mapa de bits és que no permeten un canvi d'escala significatiu, ja que apareix l'anomenat pixelat. Per a crear o editar imatges existeixen multitud de programes, alguns de programari lliure com **Gimp**, i altres per als quals hauríem d'adquirir la seua llicència d'ús, com **Photoshop d'Adobe** o **Photopaint de Corel**.



Les imatges vectorials o de vector

Representen, a través de fórmules matemàtiques, entitats geomètriques simples (punts, segments, rectangles, cercles), els seus paràmetres principals: gruix, posició inicial, final, etc. El processador és l'encarregat de traduir aquesta informació matemàtica a la targeta gràfica. Quasi qualsevol tipus d'imatge pot obtindre's mitjançant la combinació d'aquestes

entitats geomètriques més senzilles. Una dels avantatges més evidents respecte a les mapes de bits és que **poden canviar d'escala sense perdre qualitat**.



3.2. Formats d'imatge

Com ja hem vist en l'apartat anterior, existeixen dues tipus d'imatges digitals, les quals presenten diverses diferències, entre elles el format en el qual han d'emmagatzemar-se per a la seua posterior reproducció. Aquest format apareix reflectit en la part del nom del fitxer coneguda com a extensió. L'elecció d'un tipus o un altre, es pot basar en tres factors importants:

1. El contingut de la imatge (foto, dibuix, logotip).
2. La qualitat que es desitja obtenir en funció del lloc i finalitat de la publicació (publicació en web, impressió).
3. La grandària que tindrà l'arxiu resultant.

Una de les principals decisions a l'hora d'incloure gràfics en qualsevol mena d'interfície és triar el format correcte per a cada tipus d'imatge de manera que s'aconsegueixca una correcta relació entre la qualitat visual de la mateixa i la seua grandària, és a dir, el seu pes. A continuació, es defineixen alguns dels formats d'imatge més comuns en l'actualitat:

- **BMP**. Format introduït per Microsoft i usat originàriament pel sistema operatiu Windows per a emmagatzemar les seues imatges.
- **GIF** (Graphic Image File Format, format d'intercanvi de gràfics). Format antic desenrotllat per CompuServe amb l'objectiu d'aconseguir arxius de grandària reduïda. No és adequat per a imatges fotogràfiques, atés que només permet 256 colors, però sí que és indicat per a una altra mena de representacions més senzilles, com ara els logotips. Si s'emmagatzema una imatge que té més d'aqueixos colors en format GIF, s'utilitza un algorisme que aproxima els colors de la imatge a una paleta limitada per 256 colors.

Per tant, GIF és una compressió d'imatges sense pèrdua només per a imatges de 256 colors o menys. Podem resumir les seues característiques com:

- Nombre de colors: de 2 a 256.
- Format de compressió sense pèrdua basat en l'algorisme LZW.
- Càrrega progressiva en el navegador.
- Permet l'animació simple.
- És el format més adequat per a aquelles imatges senzilles, de formes simples i en les quals no existeix un elevat nombre de colors.

- **JPEG** (Joint Photographic Experts Group). Es tracta d'un dels formats més utilitzats per a tractar fotografies digitals, gràcies a l'ampli ventall de colors que admet. És el format utilitzat en cambres fotogràfiques i escàners, per tant, és el més usat en pàgines web. A més, JPEG admet diferents nivells de compressió, d'aquesta manera aconsegueix modificar la grandària en funció del treball que es desitge, presentant com a contraprestació la disminució de la qualitat. Quant menor siga la compressió de la imatge, major serà la qualitat, però la grandària dels arxius serà major. Per contra, si s'utilitza un nivell de compressió major, aquesta produeix pèrdues i afecta a la qualitat d'imatge, per a dur a terme aquesta reducció de grandària, JPEG elimina la informació que l'ull humà no és capaç de distingir. Les característiques d'aquest format són:
 - Nombre de colors: 24 bits color o 8 bits B/N.
 - Format de compressió amb pèrdua.
 - No permet l'animació.
 - Per regla general, és el més indicat per a aquelles imatges que són fotografies.
- **PNG** (Portable Network Graphics). Format creat per a substituir les imatges de format GIF. Es tracta d'un sistema de compressió sense pèrdua, a més, permet una compressió reversible i per tant la imatge que es recupera és exacta a l'original. Les característiques d'aquest format són:
 - Color indexat fins a 256 colors i True-color fins a 48 bits per píxel.
 - Major compressió que el format GIF.
 - Compressió sense pèrdua.
 - No permet animació.
- **PSD**. És el format per defecte de l'editor d'imatges Adobe Photoshop i, per tant, és un format adequat per a editar imatges amb aquest programa i altres compatibles.
- **RAW**. Es tracta del format que ofereix la major qualitat fotogràfica, gràcies a aquesta mena de format, els píxels no es processen, és a dir, es mantenen tal com s'han pres, de tal forma que poden ser processats posteriorment per un programari específic conegut com a "revelador RAW".
- **TIFF** (Tagged Image File Format). Format utilitzat per a l'escanejat, l'edició i impressió d'imatges fotogràfiques. És compatible amb quasi tots els sistemes operatius i editors d'imatges.

3.3. Resolució i profunditat del color

Quan parlem de les imatges aquestes estan clarament vinculades a dos paràmetres, en primer lloc, la resolució, que determina el grau de detall de la imatge, i en segon lloc, la profunditat de color, la qual fa referència al nombre de bits utilitzat en cada píxel per a descriure un determinat color. Veurem a continuació tots dos paràmetres amb més atenció.

Resolució

La resolució consisteix en el grau de detall o qualitat d'una imatge digital. Aquest valor s'expressa en ppp (píxels per polzada) o en dpi (dots per inch). Quants més píxels continga una imatge per polzada lineal, major serà la seua qualitat. Per exemple, quan parlem de la resolució d'un monitor, estem fent referència al nombre de píxels per polzada que és capaç de mostrar. D'altra banda, en un mitjà d'impressió es parla del nombre de punts per polzada que es pot imprimir.

Profunditat de color

Una imatge en mapa de bits (mapa de bits) està formada per un conjunt de píxels, on cadascun d'ells presenta un determinat color, l'arxiu on està emmagatzemada la imatge, també contindrà la informació de color de cadascun dels píxels. El nombre de bits utilitzats per a descriure el color de cada píxel d'una imatge rep el nom de profunditat de color. Com més gran és la profunditat de color d'una imatge, més colors tindrà la paleta disponible i, per tant, la representació de la realitat podrà fer-se amb més matisos.

Si només es disposa d'1 bit per a descriure el color de cada píxel, aquest prendrà els valors 0 o 1, blanc i negre. Si disposem de 8 bits per a descriure el color de cada píxel, podrem triar entre 256 colors, ja que com vam veure en capítols anteriors $2^8 = 256$ colors. Aquesta profunditat de color és utilitzada per a les imatges en manera escala de grises, des del negre absolut (00000000), fins al blanc absolut (11111111), passant per totes les combinacions possibles de grisa.

A partir de 8 bits per a profunditat, també és possible assignar colors, en concret, 256. Entre aquestes possibles codificacions de color es troben el negre, el blanc, grisos i els colors més freqüents. En aquest cas, es crea una taula amb 256 colors, cadascuna de les combinacions possibles d'uns i zeros dels 8 bits és un índex que permet accedir a la taula i seleccionar un color, per aquesta raó, a les imatges de 8bits se les denomina, de color indexat. Per tant, com més gran siga el nombre de bits utilitzat, major serà la profunditat de color. En el quadre següent tens el càlcul dels colors disponibles per a cada profunditat:

Profunditat	Nombre de colors
1 bit	2
4 bits	16
8 bits	256
16 bits	65.536
32 bits	4.294.967.296

Per damunt de 16 bits de profunditat, la descripció del color es divideix en capes. Si la profunditat de color és de 16 bits, per exemple, es dediquen 4 bits (128 nivells) a cada capa. I si la profunditat és de 32 bits, cada capa utilitza 8 bits (256 nivells) per a ajustar el color.

3.4. Grandària i compressió d'imatges

Un dels factors més importants a l'hora de triar les imatges que formaran part del disseny d'una interfície, és la grandària d'arxiu d'imatge, ja que d'això dependrà la velocitat de la transferència. Si una imatge és massa pesada, és recomanable utilitzar formats amb compressió, com JPEG. Existeixen altres ocasions en les quals és desitjable que la grandària de la imatge siga elevada, la qual cosa suposarà una millor qualitat, és el cas de la impressió fotogràfica.

Per tant, un dels conceptes importants que tindre en compte és la grandària de l'arxiu, que consisteix en una xifra, normalment expressada en bits o bytes, i que quantifica la quantitat de memòria necessària per a emmagatzemar una imatge. Es defineix mitjançant l'expressió següent.

Grandària = $R^2 \cdot L \cdot A \cdot P$ on:

- R és la resolució
- L és la longitud d'imatge
- A és l'ample de la imatge
- P és la profunditat del color

Finalment, definirem en què consisteix la compressió d'imatges. Després d'obtenir la imatge, a través del canal oportú, aquesta s'emmagatzema en un fitxer, compost per un nom i la seua extensió, on es recull la informació de la imatge, és a dir, la informació de cadascun dels seus píxels, necessaris per a la representació d'aquesta.

Normalment, els arxius de tipus vector ocupen menys espai que els de tipus mapa de bits, per la qual cosa es fa recomanable la compressió d'aquests per a optimitzar la velocitat de processament, això va fer necessari el desenvolupament de tecnologies capaces de comprimir arxius gràfics, on el sistema de compressió utilitza un algorisme matemàtic propi per a reduir la quantitat de bits necessaris per a descriure la imatge, i marca l'arxiu resultant amb una extensió característica: bmp, wmf, jpg, gif, png, etc.

4. Programari per a la gestió de recursos gràfics

L'edició, visualització o creació de les imatges requereix d'un conjunt d'eines bàsiques, des de programari per a la visualització d'imatges, que ens permeten operacions senzilles com veure la imatge, ampliar unes certes zones o ajustar alguns paràmetres, com ara la lluentor o la saturació. Fins a operacions més complexes emprades per a modificar la imatge aplicant efectes, transparències o distorsions.

A continuació veurem en què consisteixen cadascun d'aqueixos tipus d'eines i alguns exemples d'aplicacions que podem trobar actualment en el mercat.

4.1. Programari de visualització d'imatges

És indispensable disposar d'un programari per a la visualització de les imatges. Algunes de les característiques més desitjables en aquesta mena d'aplicacions són:

- Que permeti aplicar i reduir la grandària de la imatge per a ser correctament visualitzada. D'aquesta manera es poden observar els detalls de les fotos en profunditat.
- Visualitzar totes les imatges emmagatzemades per a poder comparar i seleccionar aquella que s'adeqüe més a les nostres necessitats.
- Girar i girar les imatges, això és, canviar la seua orientació.
- Eliminar les imatges no desitjades.
- Copiar imatges.
- Consultar les propietats d'una imatge. Corba de color, grandària en píxels, etc.
- Imprimir, guardar i enviar per correu electrònic.

Existeixen multitud d'aplicacions per a aquest objectiu, disponibles per a tots els sistemes operatius Windows, Linux i Mac, com poden ser: **IrfanView**, **XnView** i **STDU Viewer**, per a Windows, **GwenView**, **Eye of GNOME** i **Feh**, per a Linux, **FFView**, **Xee** i **Photon**, per a Mac.

- **Xee** (programari per a **Mac**). Xee és un visor d'imatges per als sistemes operatius MacOS. Es tracta d'un programari molt lleuger que no ocupa massa memòria. És una de les millors opcions. Entre les seues característiques de funcionament, destaquen que permet navegar fàcilment a tot el contingut de carpetes i arxius, moure i copiar imatges ràpidament amb compatibilitat de molts formats d'arxiu. Compleix el seu propòsit de programari de visualització (fer zoom, girar, girar) encara que no incorpora massa funcionalitats extres. És gràcies a això que es converteix en un visor àgil que no necessita massa temps de processament.
- **IrfanView** (programari per a **Windows**). IrfanView, desenvolupat per Irfan Skiljan, va ser el primer visor gràfic de Windows a nivell mundial, es tracta d'un programari senzill d'utilitzar per a quasi qualsevol tipus d'usuari, és un programa totalment gratuït, potent i molt lleuger, que permet una visualització d'imatges àgil. A més, IrfanView busca crear característiques úniques, noves i interessants, a diferència d'alguns altres visors. Una de les seues característiques principals és que permet automatitzar tasques repetitives o complexes. Es tracta d'una de les millors opcions si es requereix modificar imatges sovint.
- **GwenView** (programari per a **Linux**). GwenView és un visor d'imatges i vídeos ràpid i fàcil d'usar, que proporciona dues maneres de funcionament: navegar i veure. El primer, permet la navegació per l'equip per a triar les imatges que es visualitzaran, les quals són mostrades com a miniatures, la qual cosa permet triar-les amb major precisió. La manera veure possibilita visualitzar les imatges d'una en una. La càrrega

d'imatges es possible gràcies la biblioteca Qt, per tant, GwenView admet tots els formats d'imatge que reconega aquesta biblioteca.

4.2. Programari d'edició d'imatges

Encara que no és obligatòria l'edició d'imatges, sobretot si aquestes han sigut preses amb cambres de qualitat o han sigut creades com a logos i icones, és recomanable que es duguen a terme uns certs retocs perquè es mantinga un estil comú en tots els elements que es disposen sobre una aplicació concreta.

En l'actualitat existeixen multitud d'aplicacions que permeten retocar imatges o crear-les com a combinació de vàries. A continuació, es mostren diversos programes d'edició que podem trobar en el mercat:

- **Microsoft Paint.** Una de les aplicacions més conegudes per a l'edició d'imatges és la que porta instal·lat el sistema operatiu Windows, Microsoft Paint. Es tracta d'un programari que s'utilitza per a editar imatges d'una forma senzilla si bé és cert que no permet fer grans canvis de disseny, per a retocs senzills, és una bona opció. Per a les versions de Windows superiors a Windows 8, existeix una nova versió de Paint, anomenada Fresh Paint, que incorpora funcionalitats extra.
- **Pinta.** És una aplicació similar a l'anterior, però en aquest cas es tracta de programari lliure. Està disponible per a Windows, Linux i Mac OS X. Aquest programari ofereix les funcionalitats bàsiques per a l'edició d'imatges, així com algunes funcionalitats extra respecte a Paint, com poden ser els efectes.
- **Gimp.** Programari multiplataforma que es troba disponible per a Windows, Linux i Mac OS X. Es tracta d'una de les eines lliures més avançades en l'actualitat per a l'edició d'imatges, arribant a ser comparada amb Adobe Photoshop, ja que incorpora múltiples funcionalitats professionals. En incorporar més funcions, la seua manera d'ús no és tan intuïtiu com els vistos anteriorment, però existeixen molts manuals i documentació disponible per a aprendre a utilitzar-ho.
- **Adobe Photoshop.** L'editor d'imatges i gràfics rasteritzats per excel·lència, desenvolupat per Adobe System Incorporated, és utilitzat per a l'edició de fotografies. La seua potència no sols ho ha convertit en l'editor d'imatges més conegut i utilitzat (sobretot de manera professional), sinó que s'utilitza per a elaborar un disseny des de zero. Una de les característiques més destacades són les capes en les quals se subdivideix la imatge, que permet aplicar diferents efectes, textos, marques i tractaments a cadascuna d'elles.

4.3. Programari de creació d'imatges

En el disseny gràfic, a més de l'edició d'imatges, també és important la creació d'elements que distingisquen la nostra imatge de marca, per exemple, l'elaboració del logotip de l'empresa, una paleta de colors identificativa, o el disseny d'icones o botons. Algunes eines utilitzades per a aquest objectiu són les següents:

- **iDraw.** Aquest programari incorpora multitud d'eines que permeten crear des d'il·lustracions tècniques fins a imatges, com si d'obres d'art es tractara. Està disponible per a Mac OS X. Igual que Photoshop, permet la creació de capes múltiples, d'altra banda, a través d'una eina, coneguda com a ploma, és possible crear formes personalitzades, i permet retocar punts de la imatge amb molta més precisió.
- **Bannershop GIF Animator** (<http://www.selteco.com>). Aquesta aplicació és utilitzada per a crear imatges en format GIF. Presenta un disseny senzill i intuïtiu que permet la creació d'aquests elements d'una forma àgil i dinàmica.

El programari per a la creació d'imatges s'ocupa sobretot del disseny dels elements visuals descrits en els apartats anteriors. Habitualment, les imatges i fotografies requereixen d'eines que permeten la seua edició i processament, amb l'objectiu de millorar la qualitat d'aquestes.

Algunes de les accions que es poden aplicar sobre les imatges són la modificació de la grandària o de la seua posició, l'esborrat d'elements no desitjats, el tractament de les propietats, i fins i tot l'aplicació de filtres i efectes. Un filtre és la transformació d'una imatge digital, mitjançant l'aplicació d'operadors matemàtics. És habitual utilitzar aquest tipus d'elements per a millorar la qualitat d'una fotografia quant al tractament d'aquesta. Per exemple, permet suavitzar o atenuar una imatge, realçar les vores o eliminar la distorsió o soroll, entre altres. Mentre que un efecte millora també les imatges, en aquest cas afegint-los elements externs, com ara ombres, reflexos, halos de color o bisells..

4.4. Programari: logotipus i icones

En primer lloc, parlem dels logos o logotips. Es tracta d'imatges que caracteritzen a una empresa o marca, distingint d'aquesta manera els seus productes o serveis de la resta. Per aquesta raó, realitzar un bon disseny del logotip, que resulte atractiu i fàcil d'identificar i associar a la imatge de marca és clau.

D'altra banda, es troben les icones gràcies als quals és possible representar una idea, funció o acció que expressada en text ocuparia més espai i fins i tot resultaria menys intuïtiva per a l'usuari.

Per exemple, en moltes ocasions pot resultar més intuïtiu incloure una icona amb la imatge d'una casa per a tornar a la pantalla inicial que escriure "Prem aquest botó per a anar a la pàgina en la qual comença tot...". Per exemple, al mercat disposem d'aquest programari:

IcoFx (<https://icofx.ro>). Editor d'icones gratuït, que permet la creació, extracció i edició d'icones. Està disponible per a sistemes sobre Windows, a partir de XP, i per a Mac OS X. A més de la creació d'icones des de zero, permet convertir imatges en icones. És desitjable que les icones d'una mateixa interfície d'aplicació presenten característiques similars, IcoFX permet crear llibreries d'icones.

5. Les imatges i la llei de propietat intel·lectual

Les imatges que s'utilitzen en qualsevol mena de disseny han de ser correctament obtingudes, és a dir, o bé són d'elaboració pròpia i tenim els seus drets d'ús, o si utilitzem unes altres han d'estar correctament referenciades. En aquest apartat ens centrarem en els punts relatius a aquest tema.

5.1. Drets de la propietat intel·lectual

Què és la propietat intel·lectual? És possible trobar diverses definicions sobre aquest controvertit concepte, que podem resumir "com el conjunt de drets sobre un contingut original que tenen els seus autors".

De forma més genèrica, una de les accepcions més dominant en l'actualitat en gran varietat de països és la definició recollida per la **OMPI** (Organització Mundial de la Propietat Intel·lectual), la qual defineix la propietat intel·lectual com:

"La propietat intel·lectual (P.I.) té a veure amb les creacions de la ment: les invencions, les obres literàries i artístiques, els símbols, els noms, les imatges i els dibuixos i models utilitzats en el comerç.

La propietat intel·lectual es divideix en dues categories: la propietat industrial, que inclou les invencions, patents, marques, dibuixos i models industrials i indicacions geogràfiques de procedència; i el dret d'autor, que abasta les obres literàries i artístiques, com ara les novel·les, els poemes i les obres de teatre, les pel·lícules, les obres musicals, les obres d'art, com ara els dibuixos, pintures, fotografies i escultures, i els dissenys arquitectònics. Els drets relacionats amb el dret d'autor són els anomenats drets connexos dels artistes intèrprets i executants sobre les seues interpretacions i execucions, els drets dels productors de fonogrames sobre els seus enregistraments i els drets dels organismes de radiodifusió sobre els seus programes de ràdio i de televisió."

5.2. Drets d'autor






En segon lloc, un altre dels termes importants a tindre enquesta són els drets d'autor, podem definir-los com el conjunt de normes i principis que regulen els drets dels autors, sobre qualsevol mena d'obra creada per aquests, és a dir, des que es crea una obra, l'autor posseeix plens drets sobre aquesta. Aquests drets estan constituïts per dues claus: drets morals i drets patrimonials.


- a) Drets morals o personals: els quals inclouen aspectes sobre el reconeixement de la condició d'autor de l'obra.
- b) Drets patrimonials: aquests són susceptibles de tindre un valor econòmic i solen estar associats al concepte anglosaxó de Copyright.

5.3. Llicències

Les llicències **Creative Commons** proporcionen uns certs drets d'ús baix determinades condicions, és a dir, no significa que no tinguin drets d'autor, sinó que a través d'unes determinades condicions d'ús, i de reconeixement d'autoria, és possible utilitzar segons quin contingut. En funció de l'elecció de la llicència per part de l'autor de l'obra, serà possible prohibir la reproducció i distribució de la totalitat o part d'aquesta sense l'autorització expressa de l'autor. És possible que l'autor decidisca posar a la disposició del públic la seua obra, aquesta haurà d'autoritzar-se explícitament per a cada ús que vaja a fer-se d'ella o s'estarà vulnerant la llei.

Sobre la base d'això podem distingir entre diferents tipus de llicències Creative Commons que es clasifiquen en quatre grans llicències: atribució de l'obra, no comercial, sense derivats i compartir igual. Les combinacions possibles de llicències i la seua funció es descriuen a continuació:

Tipus de llicència	Descripció	Imatge
Reconeixement CC By	Aquesta llicència permet a uns altres distribuir, barrejar, ajustar i construir a partir de la seua obra, fins i tot amb objectius comercials, sempre que li siga reconeguda l'autoria de la creació original.	
Reconeixement-CompartirIgual CC BY-SA	Aquesta llicència permet a altres modificar i desenvolupar sobre una obra, fins i tot per a propòsits comercials, sempre que t'atribuïsquen el crèdit i llicencien les seues noves obres sota idèntics termes. Qualsevol obra nova basada en la teua, ho serà sota la mateixa llicència, de manera que qualsevol obra derivada permetrà també el seu ús comercial.	
Reconeixement-SenseObraDerivada CC BY-ND	Aquesta llicència permet la redistribució, comercial i no comercial, sempre que l'obra no es modifique i es transmeta íntegrament, reconeixent la seua autoria.	
Reconeixement-NoComercial CC BY-NC	Aquesta llicència permet a uns altres ajustar i construir a partir de la seua obra amb eines no comercials, i encara que en les seues noves creacions hagen de reconèixer la seua autoria i no puguin ser utilitzades de manera comercial, no han d'estar sota una llicència amb els mateixos termes.	
Reconeixement-NoComercialCompartirIgual CC BY-NC-SA	Aquesta llicència permet a uns altres ajustar i construir a partir de la seua obra amb objectius no comercials, sempre que li reconeguen l'autoria i	

	les seues noves creacions estiguen sota una llicència amb els mateixos termes.	
Reconeixement- NoComercialSenseObraDerivada CC *BY-*NC-*ND	Aquesta llicència és la més restrictiva, només permet que uns altres puguin descarregar les obres i compartir-les amb altres persones, sempre que es reconega la seua autoria, però no es poden canviar de cap manera ni es poden utilitzar comercialment.	

5.4. Registre de contingut

El registre és un mitjà que garanteix la protecció dels drets de la propietat intel·lectual, la seua inscripció no és obligatòria. L'autor decideix quan i per què vol registrar una obra.

Aquest registre és una constància de l'autoria sobre una obra, no impedeix que aquestes siguin plagiades o es cometan altres infraccions sobre elles, hem de tindre en compte que el que és registrat és una obra i no una idea, per a aquestes últimes s'utilitzen les conegudes com a patents i marques. Existeixen diversos sistemes de propietat intel·lectual a través dels quals és possible dur a terme el registre d'una obra. Per al cas dels registres privats de la propietat intel·lectual, que faciliten la inscripció i publicació de l'autor i titulars dels drets, aquest procés es garanteix mitjançant un sistema d'empremta digital i time-stamping o segell de temps.

- **Registre Oficial de la Propietat Intel·lectual.** Es tracta d'un registre públic, que sol existir en tots els països. A Espanya, el Registre General de la Propietat Intel·lectual és únic en tot el territori nacional i és un mecanisme administratiu per a la protecció dels drets de propietat intel·lectual dels autors i altres titulars sobre les seues obres, actuacions o produccions.
- **Safe Creative.** Registre privat de la propietat intel·lectual. Es tracta d'un registre o depòsit d'obres de propietat intel·lectual en format digital. Com s'exposa des del seu lloc web, Safe Creative és una empresa que porta des de l'any 2007 oferint els sistemes tecnològics per a la generació i gestió d'evidències d'autoria i drets relacionats més innovadors, eficients i avançats. Compta amb l'aval de desenes de milers de creadors, empreses i institucions al voltant del món, s'ha convertit en interlocutor habitual i referència en relació amb polítiques i altres aspectes relacionats amb la propietat intel·lectual.
- **Re-Crea.** És un depòsit de creacions on l'usuari envia el seu document en línia a un servidor segur de la Cambra de Comerç de Barcelona i automàticament es genera un segell de temps i un certificat conforme el document ha sigut depositat en una hora i una data concreta. Re-Crea impedeix la manipulació, per qualsevol de les parts, del document depositat.
- **Registered Commons.** Depòsit de creacions. Aquest registre permet explotar econòmicament l'obra a través de la plataforma, per al que realitza venda de llicències.

6. Formes i dibuix bàsic amb WPF

En aquest punt veurem tècniques bàsiques sobre com dibuixar amb objectes **Shape**. Un **Shape** és un tipus de `UIElement` que li permet dibuixar una forma en la pantalla. Com es tracta d'elements de la interfície d'usuari, els objectes **Shape** es poden usar dins dels elements `Panel` i en la majoria dels controls.

WPF ofereix diverses capes d'accés a gràfics i serveis de representació. En la capa superior els objectes `Shape` són fàcils d'usar i proporcionen diverses característiques útils, com el disseny i la participació en el sistema d'esdeveniments de WPF.

6.1. Objectes Shape.

WPF proporciona un nombre d'objectes `Shape` llestos per utilitzar. Tots els objectes de forma hereten de la classe `Shape`. Els objectes de forma disponibles inclouen `Ellipse`, `Line`, `Path`, `Polygon`, `Polyline` i `Rectangle`. Els objectes `Shape` comparteixen les propietats comunes següents.

- `Stroke`: descriu com es pinta el contorn de la forma.
- `StrokeThickness`: descriu el gruix del contorn de la forma.
- `Fill`: descriu com es pinta l'interior de la forma.
- Propietats de dades per especificar coordenades i vèrtexs, mesurats en píxels independents del dispositiu.

Atès que deriven de `UIElement`, els objectes de forma poden utilitzar-se als panells i en la majoria dels controls. El panell `Canvas` és una opció particularment apropiada per crear dibuixos complexos perquè admet la posició absoluta dels seus objectes secundaris.

La classe `Line` us permet dibuixar una línia entre dos punts. A l'exemple següent es mostren diverses maneres d'especificar coordenades de línia i propietats de traç.

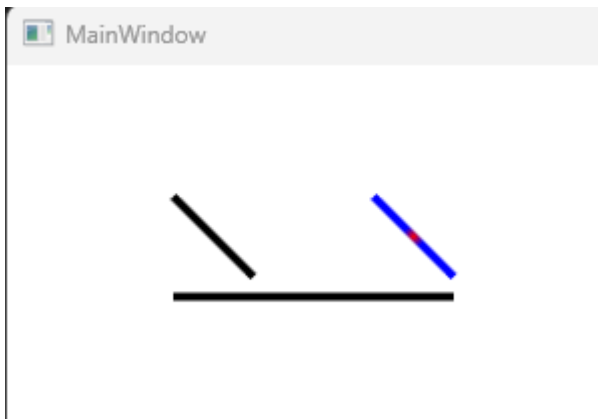
```
<Grid>
  <Canvas Height="300" Width="300">
    <!-- Draws a diagonal line from (10,10) to (50,50). -->
    <Line X1="10" Y1="10" X2="50" Y2="50" Stroke="Black" StrokeThickness="4" />

    <!-- Draws a diagonal line from (10,10) to (50,50) and moves it 100 pixels to the right. -->
    <Line X1="10" Y1="10" X2="50" Y2="50" StrokeThickness="4" Canvas.Left="100">
      <Line.Stroke>
        <RadialGradientBrush GradientOrigin="0.5,0.5" Center="0.5,0.5" RadiusX="0.5" RadiusY="0.5">
          <RadialGradientBrush.GradientStops>
            <GradientStop Color="Red" Offset="0" />
            <GradientStop Color="Blue" Offset="0.25" />
          </RadialGradientBrush.GradientStops>
        </RadialGradientBrush>
      </Line.Stroke>
    </Line>

    <!-- Draws a horizontal line from (10,60) to (150,60). -->
    <Line X1="10" Y1="60" X2="150" Y2="60" Stroke="Black" StrokeThickness="4"/>
  </Canvas>
</Grid>
```

Observa que aquests objectes dins d'un contenidor **Canvas** per separar-los de la resta de controls.

El resultat seria aquest:



També es podrien crear un objecte **Line** amb codi C#:

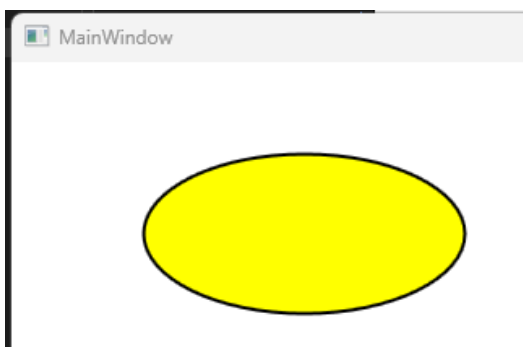
```
myLine = new Line();  
myLine.Stroke = System.Windows.Media.Brushes.LightSteelBlue;  
myLine.X1 = 1;  
myLine.X2 = 50;  
myLine.Y1 = 1;  
myLine.Y2 = 50;  
myLine.HorizontalAlignment = HorizontalAlignment.Left;  
myLine.VerticalAlignment = VerticalAlignment.Center;  
myLine.StrokeThickness = 2;  
myGrid.Children.Add(myLine);
```



Una altra forma freqüent és l' [El·lipse](#) . Creeu un [El·lipse](#) mitjançant la definició de les propietats de la forma [Width](#) i [Height](#) . Per dibuixar un cercle, especifiqueu un [Ellipse](#) amb valors [Width](#) i [Height](#) iguals. Per exemple podem dibuixar aquesta:

```
<Grid>  
  <Canvas Height="300" Width="300">  
    <Ellipse Fill="Yellow" Height="100" Width="200" StrokeThickness="2" Stroke="Black"/>  
  </Canvas>  
</Grid>
```

Que produiria aquesta el·lipse:



I també la podríem generar amb codi C#:

```
StackPanel myStackPanel = new StackPanel();

// Create a red Ellipse.
Ellipse myEllipse = new Ellipse();

// Create a SolidColorBrush with a red color to fill the
// Ellipse with.
SolidColorBrush mySolidColorBrush = new SolidColorBrush();

// Describes the brush's color using RGB values.
// Each value has a range of 0-255.
mySolidColorBrush.Color = Color.FromArgb(255, 255, 255, 0);
myEllipse.Fill = mySolidColorBrush;
myEllipse.StrokeThickness = 2;
myEllipse.Stroke = Brushes.Black;

// Set the width and height of the Ellipse.
myEllipse.Width = 200;
myEllipse.Height = 100;

// Add the Ellipse to the StackPanel.
myGrid.Children.Add(myEllipse);
```

6.2. Traçats i geometries.

La classe [Path](#) ens permet dibuixar corbes i formes complexes. Aquestes corbes i formes es descriuen mitjançant els objectes [Geometry](#). Per utilitzar un [Path](#), creeu un [Geometry](#) i utilitzeu - lo per establir la propietat [Path](#) de l' objecte [Data](#).

Hi ha una varietat d'objectes [Geometry](#) entre els quals pot triar. Les classes [LineGeometry](#), [RectangleGeometry](#) i [EllipseGeometry](#) descriuen formes relativament senzilles. Per crear formes més complexes o crear corbes, feu servir un [PathGeometry](#).

PathGeometry i PathSegments

Els objectes [PathGeometry](#) consten d'un o més objectes [PathFigure](#); cada [PathFigure](#) representa una «figura» o una forma diferent. Cada [PathFigure](#) consta, ell mateix, d'un o més objectes [PathSegment](#), cadascun dels quals representa una porció connectada de la figura o la forma. Els tipus de segments inclouen els següents: [LineSegment](#), [BezierSegment](#) i [ArcSegment](#).

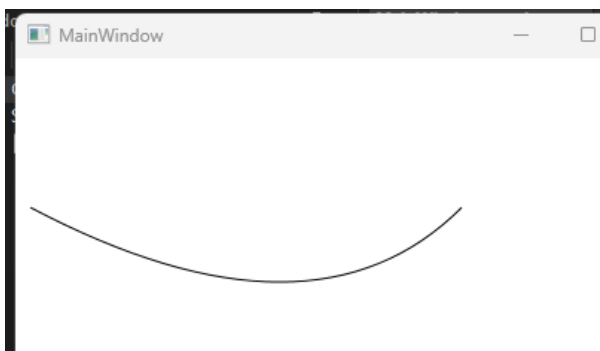
A l'exemple següent es fa servir un [Path](#) per dibuixar una corba de Bézier quadràtica.

```

<Grid>
  <Path Stroke="Black" StrokeThickness="1">
    <Path.Data>
      <PathGeometry>
        <PathGeometry.Figures>
          <PathFigureCollection>
            <PathFigure StartPoint="10,100">
              <PathFigure.Segments>
                <PathSegmentCollection>
                  <QuadraticBezierSegment Point1="200,200" Point2="300,100" />
                </PathSegmentCollection>
              </PathFigure.Segments>
            </PathFigure>
          </PathFigureCollection>
        </PathGeometry.Figures>
      </PathGeometry>
    </Path.Data>
  </Path>
</Grid>

```

I produiria aquesta corba:



També podríem dibuixar un arc, així:

```

<Path Stroke="Black" StrokeThickness="1">
  <Path.Data>
    <PathGeometry>
      <PathGeometry.Figures>
        <PathFigureCollection>
          <PathFigure StartPoint="10,100">
            <PathFigure.Segments>
              <PathSegmentCollection>
                <ArcSegment Size="100,50" RotationAngle="45" IsLargeArc="True" SweepDirection="CounterClockwise" Point="200,100" />
              </PathSegmentCollection>
            </PathFigure.Segments>
          </PathFigure>
        </PathFigureCollection>
      </PathGeometry.Figures>
    </PathGeometry>
  </Path.Data>
</Path>

```



Per a més informació sobre [PathGeometry](#) i altres classes [Geometry](#) , podeu consultar la [Informació general sobre geometria](#) .

Sintaxi abreujada de XAML

En llenguatge XAML també es pot fer servir una sintaxi abreujada especial per descriure un [Path](#) . A l'exemple següent, s'utilitza la sintaxi abreujada per dibuixar una forma complexa.

```
<Path Margin="10,40,40,10" Stroke="DarkGoldenRod"
      StrokeThickness="3" Data="M 100,200 C 100,25 400,350 400,175 H 280" />
```

La imatge generada seria aquesta:



La cadena d'atribut [Data](#) comença amb l'ordre **MoveTo**, indicada per **M**, que estableix un punt d'inici per a la trajectòria al sistema de coordenades del [Canvas](#) . Els paràmetres de dades [Path](#) distingeixen entre majúscules i minúscules. La lletra **M** majúscula indica una ubicació absoluta per al punt actiu nou. Una **m** minúscula indica coordenades relatives. El primer segment és una corba de Bézier cúbica que comença a (100,200) i acaba a (400,175), traçat amb els dos punts de control (100,25) i (400,350). Aquest segment s'indica amb l'ordre **C** de la cadena d'atribut [Data](#) . Un altre cop, la **C** majúscula indica una ruta d'accés absoluta; la minúscula indicaria una ruta d'accés relativa.

El segon segment comença amb una ordre **H** "lineet" horitzontal absolut, que especifica una línia traçada des del punt de connexió del subtraçat anterior (400,175) fins a un nou punt de connexió (280,175). Com que és una ordre «LineTo» horitzontal, el valor especificat és una coordenada x .

Si volem abreujar l'arc dibuixar a l'exemple anterior la sintaxi seria aquesta:

```
<Path Stroke="Black" StrokeThickness="1"
      Data="M 10,100 A 100,50 45 1 0 200,100" />
```

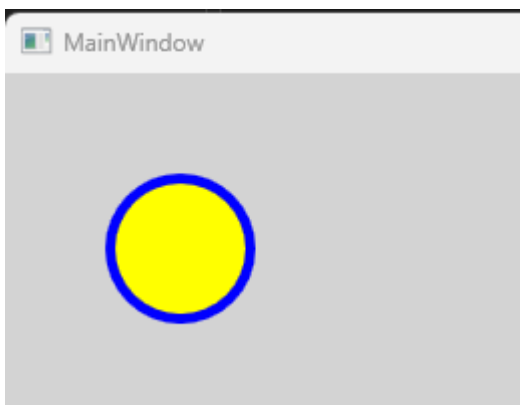
Per conèixer la sintaxi de traçat completa, consulteu la referència [Data](#) i [Crear una forma mitjançant un PathGeometry](#) .

6.3. Pintar formes.

Els objectes [Brush](#) s'usen per pintar les formes [Stroke](#) i [Fill](#) . A l'exemple següent s'especifiquen el traç i el farciment d'un [Ellipse](#) . Tingueu en compte que una entrada vàlida per a les propietats del pinzell pot ser una paraula clau o el valor de color en format hexadecimal. Per obtenir més informació sobre les paraules clau disponibles per al color, consulteu les propietats de la classe [Colors](#) a l'espai de noms [System.Windows.Media](#) .

```
<Canvas Background="■"LightGray">
  <Ellipse Canvas.Top="50" Canvas.Left="50" Fill="■"#FFFFFFF00"
    Height="75" Width="75" StrokeThickness="5" Stroke="■"#FF0000FF"/>
</Canvas>
```

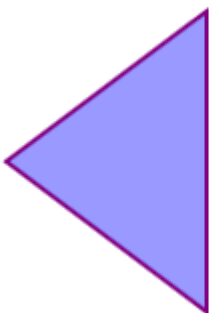
El resultat seria aquest



Altra alternativa, seria utilitzar la sintaxi de l'element de propietat per crear explícitament un objecte [SolidColorBrush](#) per pintar la forma amb un color sòlid:

```
<Polygon Points="300,200 400,125 400,275 300,200"
  Stroke="■"Purple" StrokeThickness="2">
  <Polygon.Fill>
    <SolidColorBrush Color="■"Blue" Opacity="0.4"/>
  </Polygon.Fill>
</Polygon>
```

El resultat d'aquest codi seria:



6.4. Transformar formes.

La classe [Transform](#) proporciona els mitjans per transformar formes en un pla bidimensional. Els diferents tipus de transformació inclouen rotació ([RotateTransform](#)), escalat ([ScaleTransform](#)), esbiaixat ([SkewTransform](#)) i translació ([TranslateTransform](#)).

Una transformació comuna per aplicar a una forma és la rotació. Per girar una forma, creeu un [RotateTransform](#) i especifiqueu el vostre [Angle](#) . Un [Angle](#) de 45 gira l'element 45 graus en el sentit de les agulles del rellotge; un angle de 90 gira l'element 90 graus en el sentit de les agulles del rellotge; i així successivament. Establiu les propietats [CenterX](#) i [CenterY](#) si voleu controlar el punt al voltant del qual gira l'element. Aquests valors de propietat s'expressen a l'espai de coordenades de l'element que es transformarà. [CenterX](#) i [CenterY](#) tenen valors predeterminats de zero. Finalment, apliqueu el [RotateTransform](#) a l'element. Si no voleu que la transformació afecti el disseny, establiu la propietat [RenderTransform](#) de la forma.

A l'exemple següent s'utilitza un [RotateTransform](#) per girar una forma 45 graus al voltant de la cantonada superior esquerra de la forma (0,0).

```
<Polyline Points="25,25 0,50 25,75 50,50 25,25 25,0" Stroke="Blue" StrokeThickness="10"
  Canvas.Left="75" Canvas.Top="50">
  <Polyline.RenderTransform>
    <RotateTransform CenterX="0" CenterY="0" Angle="45" />
  </Polyline.RenderTransform>
</Polyline>
```

Sense Transformació



Amb Transformació

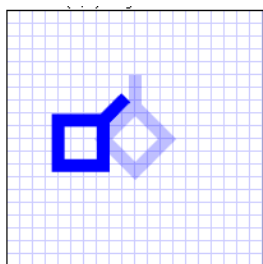


A l'exemple següent, en lloc de girar 45°, es gira al voltant del punt (25,50):

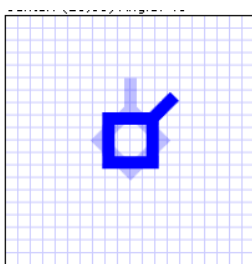
```
<Polyline Points="25,25 0,50 25,75 50,50 25,25 25,0" Stroke="Blue" StrokeThickness="10"
  Canvas.Left="75" Canvas.Top="50" RenderTransformOrigin="0.5,0.5">
  <Polyline.RenderTransform>
    <RotateTransform Angle="45" />
  </Polyline.RenderTransform>
</Polyline>
```

La diferència de rotació entre els dos exemples és aquesta:

Rotació amb angle 45°



Rotació al voltant del punt (25,50)



7. Generació de Gràfics

Els gràfics ens ajuden a visualitzar d'una forma més ràpida la informació a més de donar un plus a la nostra aplicació, per això resulta important incorporar-los en els nostres desenvolupaments.

També hem de tindre en compte que estan íntimament relacionats amb els informes (que veurem més avant), sobretot per a presentar resums relacionats amb l'informe.

En WPF podem trobar diferents generadors de gràfics (**charts**), que bé poden ser gratuïts o de pagament.

La informació se sol extreure de la base de dades mitjançant sentències **SELECT**. En els gràfics s'utilitzen molt les **funcions matemàtiques** de **SQL** i la funció **Group By**.

8. Tipus de Gràfics

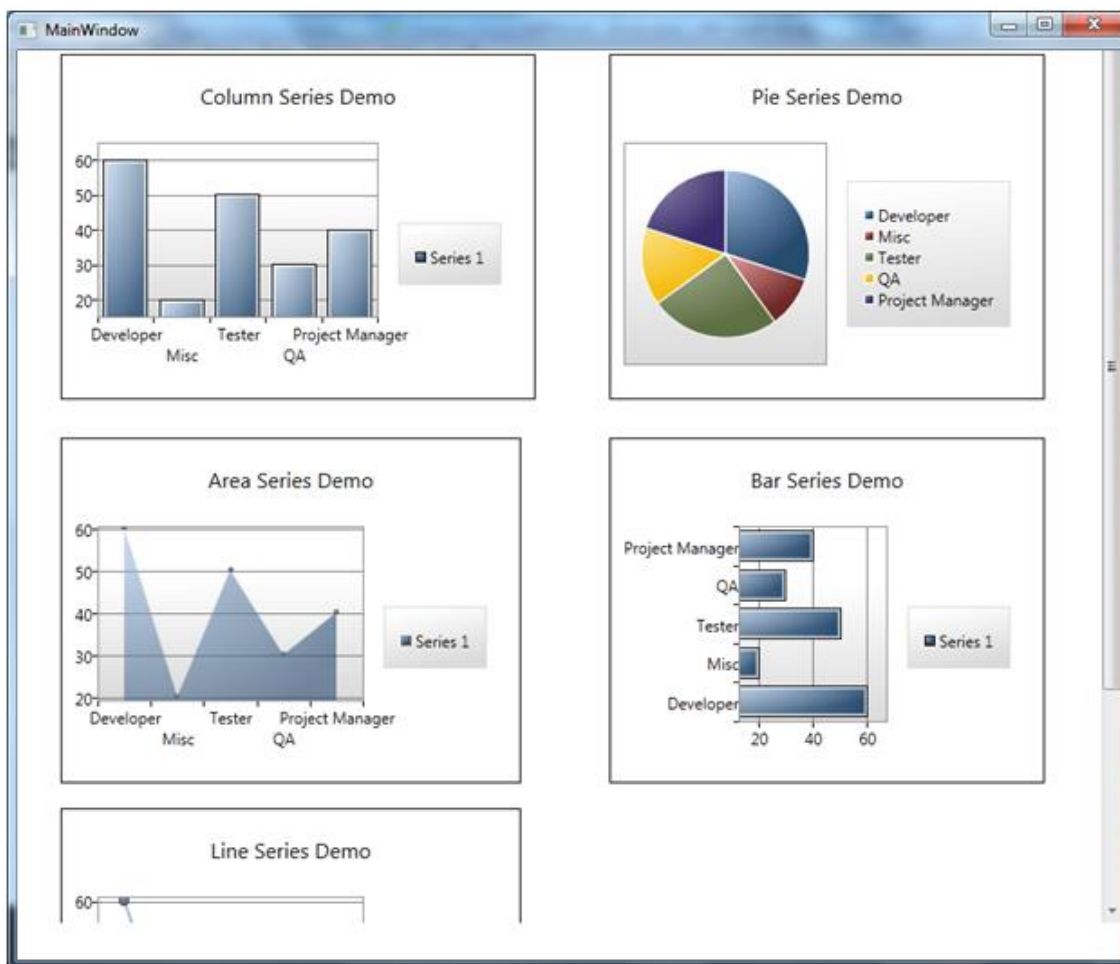
Els tipus de gràfics depenen de la llibreria utilitzada, normalment estan els típics com: **Pie Chart**, **Area Chart**, **Line Chart**, **Bar Chart**, ...

Als quals es poden afegir altres més específics com: **Bubble Chart**, **Gauge Chart**, **Dònut Chart**, **Plot Chart**, ...

A continuació es mostren un exemple d'alguns d'ells:



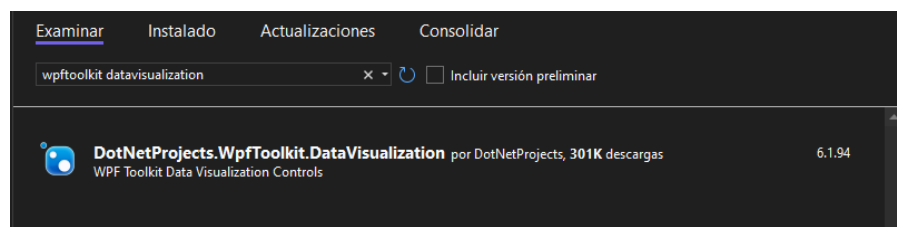
Aquest és un exemple d'una llibreria gratuïta de Charts per a WPF anomenada **Modern UI Charts**. Hi ha moltes més. Un altre exemple és **WPF Toolkit Charting**, la qual és una eina pròpia de Microsoft.



A continuació veurem diferents exemples d'algunes d'aquestes llibreries de charts, totes elles gratuïtes.

9. WPF Toolkit Charting

Per a poder utilitzar aquest tipus de gràfics primer hem d'instal·lar-lo en el nostre projecte des de **Nuget**. Hauriem d'instal·lar el paquet **System.Windows.Controls.DataVisualization.Toolkit** però VSC ja la discontinuat. En aquest cas utilitzarem un fork de la mateixa, que és aquesta:



Després de instal·lar-lo hem de referenciar-lo a la finestra XAML:

```
<Window x:Class="Ud6.Wpf.Toolkit.Charting.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:Ud6.Wpf.Toolkit.Charting"
        xmlns:chartingToolkit="clr-namespace:System.Windows.Controls.DataVisualization.Charting;assembly=DotNetProjects.DataVisualization.Toolkit"
        mc:Ignorable="d"
        Title="MainWindow" Height="450" Width="800">
    <Grid>
    </Grid>
</Window>
```

I ja podem treballar amb ell. Crearem un exemple amb gràfic de cada tipus al codi XAML:

```
<Grid>
  <ScrollViewer HorizontalScrollBarVisibility="Auto"
    VerticalScrollBarVisibility="Auto" Margin="0,-28,0,28">
    <Grid Height="921">
      <chartingToolkit:Chart Height="262" HorizontalAlignment="Left"
        Margin="33,0,0,620" Name="columnChart" Title="Column Series Demo"
        VerticalAlignment="Bottom" Width="360">
        <chartingToolkit:ColumnSeries DependentValuePath="Value"
          IndependentValuePath="Key" ItemsSource="{Binding}" />
      </chartingToolkit:Chart>
      <chartingToolkit:Chart Name="pieChart" Title="Pie Series Demo"
        VerticalAlignment="Top" Margin="449,39,43,0" Height="262">
        <chartingToolkit:PieSeries DependentValuePath="Value"
          IndependentValuePath="Key" ItemsSource="{Binding}"
          IsSelectionEnabled="True" />
      </chartingToolkit:Chart>
      <chartingToolkit:Chart Name="areaChart" Title="Area Series Demo"
        VerticalAlignment="Top" Margin="33,330,440,0" Height="262">
        <chartingToolkit:AreaSeries DependentValuePath="Value"
          IndependentValuePath="Key" ItemsSource="{Binding}"
          IsSelectionEnabled="True"/>
      </chartingToolkit:Chart>
      <chartingToolkit:Chart Name="barChart" Title="Bar Series Demo"
        VerticalAlignment="Top" Margin="449,330,43,0" Height="262">
        <chartingToolkit:BarSeries DependentValuePath="Value"
          IndependentValuePath="Key" ItemsSource="{Binding}"
          IsSelectionEnabled="True"/>
      </chartingToolkit:Chart>
      <chartingToolkit:Chart Name="lineChart" Title="Line Series Demo"
        VerticalAlignment="Top" Margin="33,611,440,0" Height="254">
        <chartingToolkit:LineSeries DependentValuePath="Value"
          IndependentValuePath="Key" ItemsSource="{Binding}"
          IsSelectionEnabled="True"/>
      </chartingToolkit:Chart>
    </Grid>
  </ScrollViewer>
</Grid>
```

Observa com per a cada gràfic crea una etiqueta **chartingToolkit:Chart** i a dins el tipus de gràfic que vol utilitzar, per exemple el **chartingToolkit:Series**.

També realitza un **binding** amb la propietat **ItemsSource**.

Al **codi C#** de la finestra trobem la resta de codi necessari amb les dades necessàries per als gràfics i el **DataContext** necessari:

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

    List<KeyValuePair<string, int>> valueList = new List<KeyValuePair<string, int>>();
    valueList.Add(new KeyValuePair<string, int>("Developer", 60));
    valueList.Add(new KeyValuePair<string, int>("Misc", 20));
    valueList.Add(new KeyValuePair<string, int>("Tester", 50));
    valueList.Add(new KeyValuePair<string, int>("QA", 30));
    valueList.Add(new KeyValuePair<string, int>("Project Manager", 40));

    //Setting data for column chart
    columnChart.DataContext = valueList;

    // Setting data for pie chart
    pieChart.DataContext = valueList;

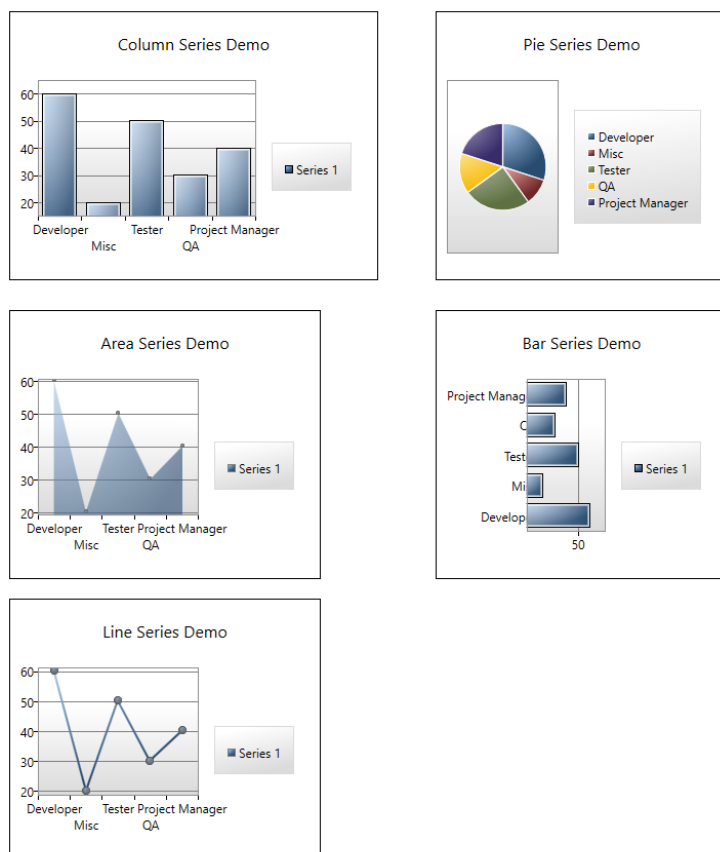
    //Setting data for area chart
    areaChart.DataContext = valueList;

    //Setting data for bar chart
    barChart.DataContext = valueList;

    //Setting data for line chart
    lineChart.DataContext = valueList;
}
```

En aquest cas crea una llista de parells de clau-valor i després l'assigna a cada **DataContext** de cada gràfic.

El resultat seria aquest:



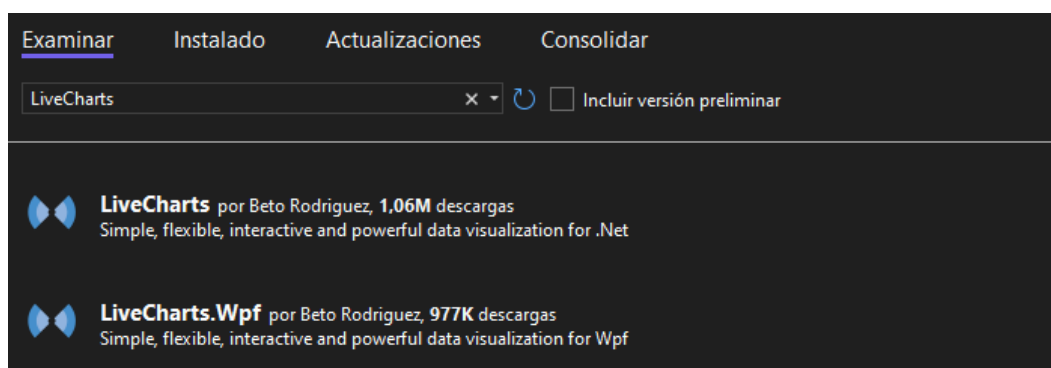
Es podria fer exactament el mateix amb un DataSet amb dades d'una base de dades com a origen de dades dels gràfics.

10. Llibreria Live Charts

Aquesta llibreria té molts tipus de gràfics i permet animacions i zoom, cosa que l'anterior no ho fa.

Té un aspecte modern i ofereix moltes possibilitats de configuració. Explicarem dos exemples perquè vegeu la manera de treballar.

Igual que amb un altre framework que no es troba per defecte en WPF, haurem d'instal·lar-ho des de Nuget. Concretament dos paquets: **LiveCharts** i **LiveCharts.WPF**.



També, igual que l'anterior, haurem de declarar-ho en el nostre fitxer XAML per a poder utilitzar-ho.

```
xmlns:lv="clr-namespace:LiveCharts.Wpf;assembly=LiveCharts.Wpf"
```

I després ja podrem començar a treballar amb els seus components, per exemple:

```
<lv:CartesianChart x:Name="lvGrups" />
```

10.1. Exemple de LiveChart amb base de dades.

Per a aquest exemple utilitzarem la classe connexió d'uns dels exemples de la unitat anterior:

```
3 referencias
internal class ConnexioBD
{
    private SqlConnection connexio;
    private string servidor;
    private string bd;
    private string usuari;
    private string contrasenya;

    // Constructor
    1 referencia
    public ConnexioBD()
    {
        Initialize();
    }

    // Inicialitzem valors
    1 referencia
    private void Initialize()
    {
        servidor = @"PHOBOS\SQLEXPRESS";
        bd = "CRUD";
        usuari = "sa";
        contrasenya = "saroot";
        string connectionString = "Data Source=" + servidor + ";Initial Catalog=" + bd + ";User ID=" + usuari + ";Password=" + contrasenya;
        connexio = new SqlConnection(connectionString);
    }
}
```

```
1 referencia
private bool ObrirConnexio()
{
    try
    {
        connexio.Open();
        return true;
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.ToString());
        return false;
    }
}

// Tanquem la connexió
1 referencia
private bool TancarConnexio()
{
    try
    {
        connexio.Close();
        return true;
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message);
        return false;
    }
}

1 referencia
public DataSet getProductes()
{
    DataSet ds = new DataSet();
    try
    {
        this.ObrirConnexio();
        SqlDataAdapter dataAdapter = new SqlDataAdapter("SELECT Nom, Stock FROM PRODUCTES", this.connexio);
        dataAdapter.Fill(ds, "productes");
        this.TancarConnexio();
        return ds;
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message);
        return ds;
    }
}
```

Dins del codi XAML de la finestra principal afegirem un gràfic LiveChart:

```
Window x:Class="Ud6_Wpf_LiveCharts.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:Ud6_Wpf_LiveCharts"
xmlns:lv="clr-namespace:LiveCharts.Wpf;assembly=LiveCharts.Wpf"
mc:Ignorable="d"
Title="MainWindow" Height="447" Width="880">
<Grid>
<lv:PieChart x:Name="graf1" LegendLocation="Bottom" >
</lv:PieChart>
</Grid>
</Window>
```

I al codi C# de la finestra cridarem als mètodes de la classe connexió per a obtenir les dades i crearem l'objecte series necessari per associar o enllaçar amb el gràfic:

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

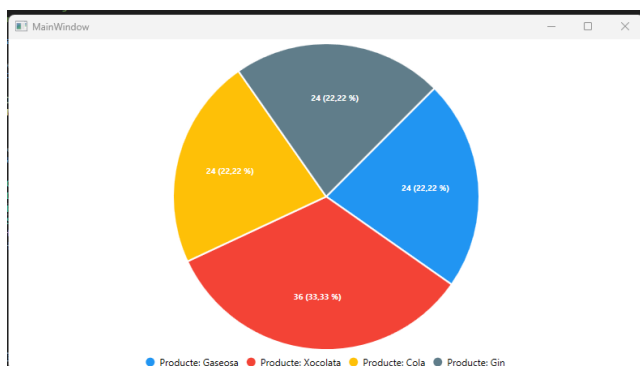
1 referencia
private void MostrarProductos()
{
    ConnexionBD cnn = new ConnexionBD();
    DataSet ds = cnn.getProductos();
    Func<ChartPoint, string> LabelPoint = chartPoint => string.Format("{0} ({1:P})", chartPoint.Y, chartPoint.Participation);
    SeriesCollection series = new LiveCharts.SeriesCollection();
    foreach (DataRow dr in ds.Tables["productos"].Rows)
    {
        PieSeries ps = new PieSeries
        {
            Title = "Producto: " + dr[0],
            Values = new ChartValues<double> { double.Parse(dr[1].ToString()) },
            DataLabels = true,
            LabelPoint = LabelPoint
        };
        series.Add(ps);
    }
    graf1.Series = series;
}
```

Fixem-nos en coses interessants del codi:

- LabelPoint: l'utilitzem per a que genere el percentatge dins de cada tros del gràfic.
- Series: Son les dades que associarem al gràfic.

Al bucle es recorre el DataTable i es genera un objecte PieSeries (necessari per el tipus de gràfic que anem a mostra). Per cada fila que insertem al PieSeries l'indiquem el títol, els valors, si mostra les etiquetes i el percentatge (LabelPoint).

El resultat final seria aquest:



Al següent enllaç podeu trobar exemples i tutorials <https://lvcharts.net/>