

## Sommario

Introduzione a Visual Studio. ....	3
Come decidere il progetto da far partire: .....	3
Introduzione al codice. ....	3
Introduzione all'Interfaccia Utente .....	4
Esempi di comandi per costruirsi un'interfaccia utente.....	4
Regole per creare un'interfaccia .....	4
Design e linguaggio di Markup .....	4
Esempio somma tra due numeri .....	7
Verifiche posticipata e anticipata .....	7
PRIMO ESEMPIO .....	7
SECONDO ESEMPIO .....	8
Separare l'interfaccia dall'elaborazione .....	9
Gestione strutturata degli errori tramite le exception.....	13
Classe System.Exception .....	14
Esempio 2 con metodo ToString() .....	15
Esempio Classe Persona .....	15
Eccezioni: esercizi. ....	18
DTO Data Transfer Object .....	20
Application programming interface .....	30
Interfacce e UML .....	31
ESERCITAZIONE API .....	35
AddressBookManagment.NET .....	38
Progettazione UML.....	38
Implementazione classi .....	38
Proprietà automatiche. ....	39
Serializzare e de-serializzare .....	42
Passaggio per riferimento degli argomenti: parametri ref e out. ....	44
Il concetto di uguaglianza.....	45
Model View e UI .....	47
Model-View .....	49
Observer Pattern .....	54
Simple Binding.....	60
CSV: .....	64
XML: .....	64
JSON: .....	64

Model View View Model .....	70
Creazione di un file XML.....	77
Leggere da XML .....	77
Lettura da XML utilizzando le classi.....	79
Scrivere su XML .....	80
Campi Mandatory.....	82
Esercitazione: Hello MVVM!.....	84

## Introduzione a Visual Studio.

BLANCK SOLUTION (Name e Location)

View → SOLUTION EXPLORER

TASTO DESTRO su SolutionNomeAssegnato → ADD NEW PROJECT

Per Interfaccia Utente →      Selezionare      WINDOWS FORM APPLICATION (Nominare il progetto)  
Oppure ConsoleApplication

Per Elaborazione Dati → ADD NEW PROJECT

→ CLASS LIBRARY (Nominare la libreria)  
→ OPPURE CLASS LIBRARY (PORTABLE FOR..)

Per utilizzare un componente (far riferimento ad un componente...)

TASTO DESTRO SU APPLICAZIONE WINDOWS

ADD → REFERENCE → PROJECT (Trovera l'elenco dei progetti presenti es. Libreria).

(→ Using NomeLibreria;)

Namespace: dare il nome ad una collezione di funzioni.

## Come decidere il progetto da far partire:

ESECUZIONE: TASTO DESTRO SUL PROGETTO DA AVVIARE (APPLICAZIONE WINDOWS O CONSOLE) → SET AS START UP PROJECT

SET AS START UP PROJECTs → Per avviare più progetti contemporaneamente.

### Per l'esecuzione:

Prima si verifica che non vi siano errori → BUILD

Debug → Trovare gli errori.

## Introduzione al codice.

Console.Read() → Aspetta per leggere cosa scrivo da tastiera.

Console.ReadLine() → Aspetta per leggere cosa scrivo da tastiera, permette di anche a capo

Console.ReadKey() → Aspetta che premo un tasto.

Console.Write() → Scrive

Console.WriteLine() → Scrive e va a capo

**Segnaposti** Console.WriteLine("Ciao {0}{1}", nome, cognome);

**Proprietà:** definisce aspetto

## Introduzione all'Interfaccia Utente

Aprire soluzione

Add New Project-> Class library (Date un nome attinente al progetto)

Add New Project

selezionare **WPF Application** (post windows XP), o Windows Form (pre Windows XP) o

Asp.Net Web Application per siti web.

## Controls: Componenti di un'interfaccia utente

View -> **Toolbox**

All WPF Controls

Common WPF Controls

**Ricordare:** Set as startup project per decidere quale progetto far partire.

## Esempi di comandi per costruirsi un'interfaccia utente

TextBox -> Crea una casella di testo

Button-> Crea un bottone

ListBox-> Per inserire più valori in una lista

## Regole per creare un'interfaccia

Un Form contiene dei controlli in grado di reagire a delle azioni dell'utente: questo significa **Generare un evento**. Qualunque operazione all'interno di una interfaccia utente solleva un evento. Alcuni eventi non vanno gestiti (es. eventi Click su una Label).

Gestire un evento significa scrivere un metodo che dice cosa fare.

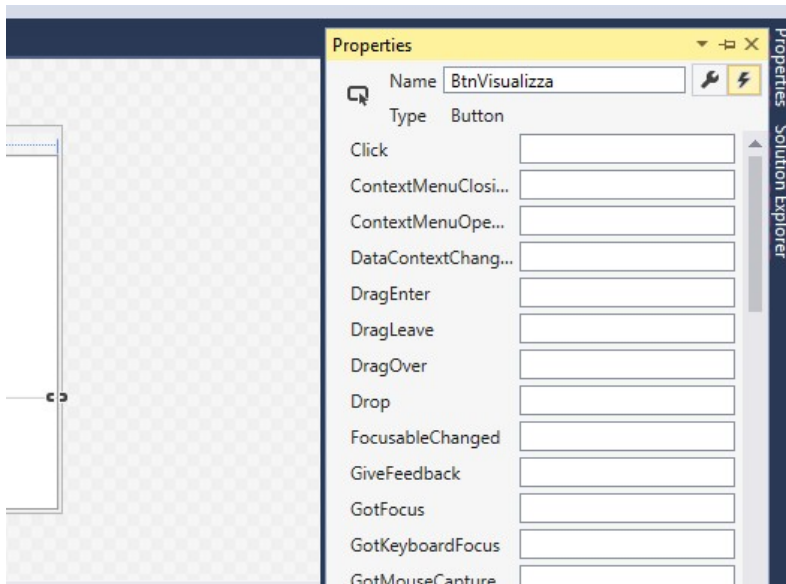
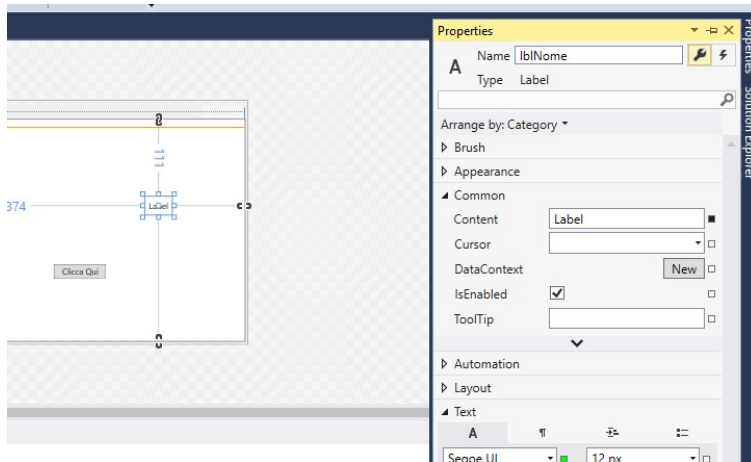
## Design e linguaggio di Markup

Le modifiche possono essere effettuate in due modalità:

Attraverso il linguaggio

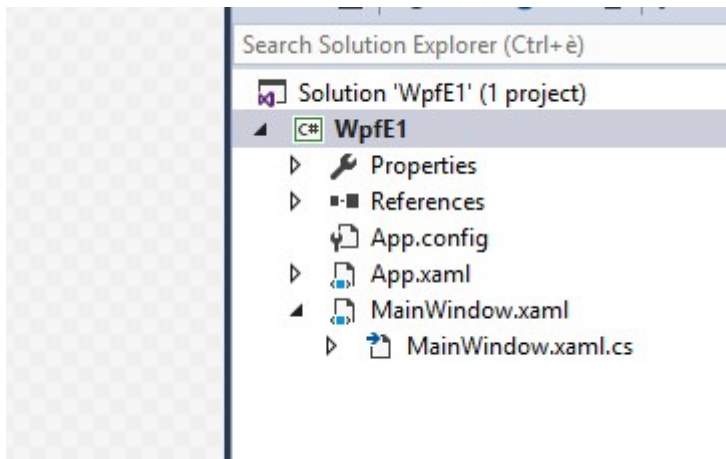


Attraverso il designer



Regola: Dare un nome pertinente al bottone che descrive che cosa fa.

Ad esempio un bottone: BtnVisualizzaCalcoli.



Aprire File xaml.cs

Doppio Click sul Bottone

Nel Markup Cambia il codice, `Click="BtnNome_Click"/>` associa l'evento Click a questo gestore di eventi

```
<Button x:Name="BtnNome" Content="Clicca Qui" HorizontalAlignment="Left"
Margin="242,209,0,0" VerticalAlignment="Top" Width="75" Click="BtnNome_Click"/>
```

```
public partial class MainWindow : Window
{
    //Metodo Costruttore
    public MainWindow()
    {
        InitializeComponent();
    }

    //Facendo doppio clic sul pulsante Bottone viene generato automaticamente un metodo
    //BtnNome_Click nome attinente al Bottone che gestisce L'evento click
    // I parametri 1. Sender cioè il mittente, chi ha scatenato l'evento;
    //2. La distinzione dell'evento scatenato es. Click destro....
    private void BtnNome_Click(object sender, RoutedEventArgs e)
    {
        string nome;
        nome = txtNome.Text;
        lblNome.Content = nome;
    }
}
```

## Esempio somma tra due numeri

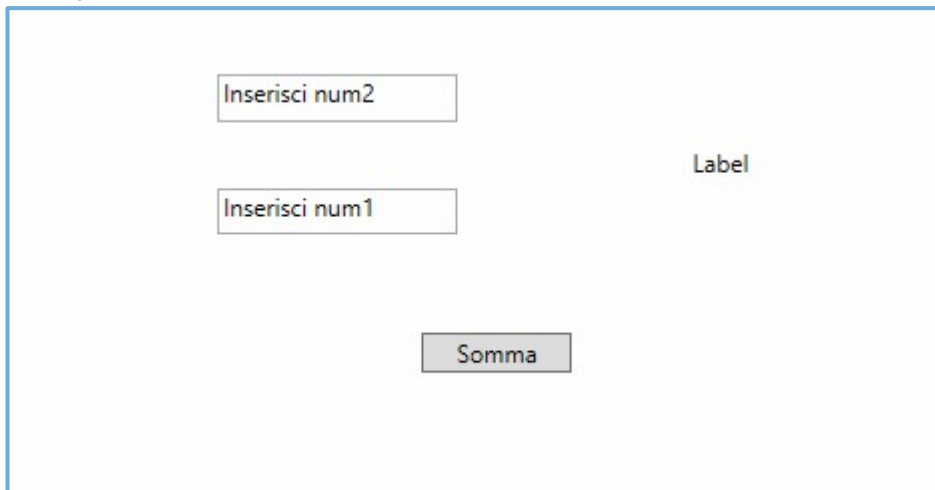


Diagram illustrating the UI layout for the sum example:

- Input field: Inserisci num2
- Input field: Inserisci num1
- Label
- Button: Somma

```
private void BtnSomma_Click(object sender, RoutedEventArgs e)
{
    int num1;
    int num2;
    int somma = 0;
    num1 = Convert.ToInt32(txtNum1.Text);
    num2 = Convert.ToInt32(txtNum2.Text);
    somma = num1 + num2;
    lblRisultato.Content = somma;
}
```

## Verifiche posticipata e anticipata

### PRIMO ESEMPIO

```
private void BtnSomma_Click(object sender, RoutedEventArgs e)
{
    string strnum1=txtNum1.Text;
    string strnum2=txtNum2.Text;

    if(strnum1 != "" && strnum2!="")//verifica posticipata
    {
        int somma = 0;
        int num1;
        int num2;
        num1 = Convert.ToInt32(txtNum1.Text);
        num2 = Convert.ToInt32(txtNum2.Text);
        somma = num1 + num2;
        lblRisultato.Content = somma;
    }
    else
    {
        MessageBox.Show("Mancano i numeri", "Errore", MessageBoxButton.OK,
        MessageBoxImage.Information);
    }
}
```

## SECONDO ESEMPIO

```
private void Somma_con_Eccezione_Click(object sender, RoutedEventArgs e)
{
    string strnum1 = txtNum1.Text;
    string strnum2 = txtNum2.Text;

    if (strnum1 != "" && strnum2 != "")//verifica posticipata
    {
        try//verifica posticipata, prova ad effettuare la conversione, se tutto ok
        visualizza risultato
        {
            int somma = 0;
            int num1;
            int num2;
            num1 = Convert.ToInt32(txtNum1.Text);
            num2 = Convert.ToInt32(txtNum2.Text);
            somma = num1 + num2;
            lblRisultato.Content = somma;
        }
        catch (Exception) //cattura il problema, il programma salta e manda un
        messaggio di errore
        {
            MessageBox.Show("I numeri non sono nel formato giusto", "Errore",
            MessageBoxButton.OK, MessageBoxImage.Information);
        }
    }

    else
    {
        MessageBox.Show("Mancano i numeri", "Errore", MessageBoxButton.OK,
        MessageBoxImage.Information);
    }
}
```

Problema: Abbiamo lo stesso messaggio di errore per problemi diversi.



## Separare l'interfaccia dall'elaborazione

Primo step: Notiamo che elaborazione e interfaccia (istruzioni in giallo) non sono separati.

Se riusciamo a separare elaborazione ed interfaccia è possibile “ riusare” la parte dell’elaborazione con più interfacce grafiche differenti (IOS, Android, Windows..)

```
private void Somma_con_Eccezione_Click(object sender, RoutedEventArgs e)
{
    string strnum1 = txtNum1.Text;
    string strnum2 = txtNum2.Text;
    double somma = 0;
    double num1;
    double num2;

    if (strnum1 != "" && strnum2 != "")
    {
        Try
        {
            num1 = Convert.ToInt32(strnum1);
            num2 = Convert.ToInt32(strnum2);
            somma = num1 + num2;
            lblRisultato.Content = somma;
        }
        catch (Exception)
        {
            MessageBox.Show("I numeri non sono nel formato giusto", "Errore",
            MessageBoxButton.OK, MessageBoxImage.Information);
        }
    }

    else
    {
        MessageBox.Show("Mancano i numeri", "Errore", MessageBoxButton.OK,
        MessageBoxImage.Information);
    }
}
```

Vediamo come procedere.

In questa porzione di codice dove mi accorgo se il calcolo è andato a buon fine?

```
double somma = 0;
double num1;
double num2;
num1 = Convert.ToInt32(strnum1);
num2 = Convert.ToInt32(strnum2);
somma = num1 + num2;
```

Ovviamente rilevo l'esecuzione "corretta" se il contenuto della variabile somma è un numero.

Pertanto posso controllare se al termine dell'elaborazione se il risultato è effettivamente un numero.

Non posso, quindi, in fase di dichiarazione, inizializzare a zero la variabile somma, ma devo "dire" che la variabile somma è del tipo "Not a Number".

```
double somma = double.NaN;
double num1;
double num2;
```

Pertanto, al termine dell'istruzione `somma = num1 + num2`, potrò verificare se il contenuto della variabile è un numero oppure no con la seguente istruzione `if (!double.IsNaN(somma))`

```
num1 = Convert.ToInt32(strnum1);
num2 = Convert.ToInt32(strnum2);
somma = num1 + num2;
if (!double.IsNaN(somma))
{
    lblRisultato.Content = somma;
}
```

A questo punto ho gli ingredienti per separare elaborazione e interfaccia

Creo un classe Operazioni con i metodi che riguardano l'elaborazione.

All'interno del metodo inserisco tutte le istruzioni che non coinvolgono l'interfaccia (vedi codice iniziale a pagina 1, non sottolineato in giallo).

```
private void Somma_con_Eccezione_Click(object sender, RoutedEventArgs e)
{
    string strnum1 = txtNum1.Text;
    string strnum2 = txtNum2.Text;
    int somma = 0;
    int num1;
    int num2;
    if (strnum1 != "" && strnum2 != "")
    {
        Try
        {
            num1 = Convert.ToInt32(strnum1);
            num2 = Convert.ToInt32(strnum2);
            somma = num1 + num2;
            lblRisultato.Content = somma;
        }
        catch (Exception)
        {
            MessageBox.Show("I numeri non sono nel formato giusto", "Errore", MessageBoxButton.OK,
            MessageBoxImage.Information);
        }
    }
    else
    {
        MessageBox.Show("Mancano i numeri", "Errore", MessageBoxButton.OK,
        MessageBoxImage.Information);
    }
}
```

Il metodo restituisce un double (somma) e ha come parametri di ingresso le due stringhe:

**strnum1 ed strnum2**

```
public static double Calcolo(string strnum1, string strnum2)
{
    double somma = double.NaN;
    double num1;
    double num2;
    if (strnum1 != "" && strnum2 != "")
    {
        try
        {
            num1 = Convert.ToInt32(strnum1);
            num2 = Convert.ToInt32(strnum2);
            somma = num1 + num2;
        }

        catch (Exception)
        { }
    }

    return somma;
}
```

Cosa cambia nel codice di partenza?

Si mantengono le istruzioni relative all'interfaccia grafica e basta invocare il metodo "Calcolo" della classe Operazioni.

```
somma = Operazioni.Calcolo(strnum1, strnum2);
```

```
private void Calcolo_Click(object sender, RoutedEventArgs e)
{
    string strnum1 = txtNum1.Text;
    string strnum2 = txtNum2.Text;
    double somma;

    somma = Operazioni.Calcolo(strnum1, strnum2);
    if (!double.IsNaN(somma))
    {
        lblRisultato.Content = somma;
    }
    else
    {
        MessageBox.Show("I numeri non sono nel formato giusto", "Errore",
        MessageBoxButton.OK, MessageBoxImage.Information);
    }
}
```

```

namespace Library_Calcolatrice
{
    public class Operazioni
    {
        public static double Somma(string strnum1 ,string strnum2 )
        {
            double somma = double.NaN;
            double num1;
            double num2;

            if (strnum1 != "" && strnum2 != "")//verifica posticipata
            {
                try//verifica posticipata, prova ad effettuare la conversione, se tutto ok
                visualizza risultato
                {
                    num1 = Convert.ToInt32(strnum1);
                    num2 = Convert.ToInt32(strnum2);
                    somma = num1 + num2;

                }
                catch (Exception) //cattura il problema, il programma salta e manda un
                messaggio di errore
                {
                    //da definire
                }
            }
            return somma;
        }
    }
}

```

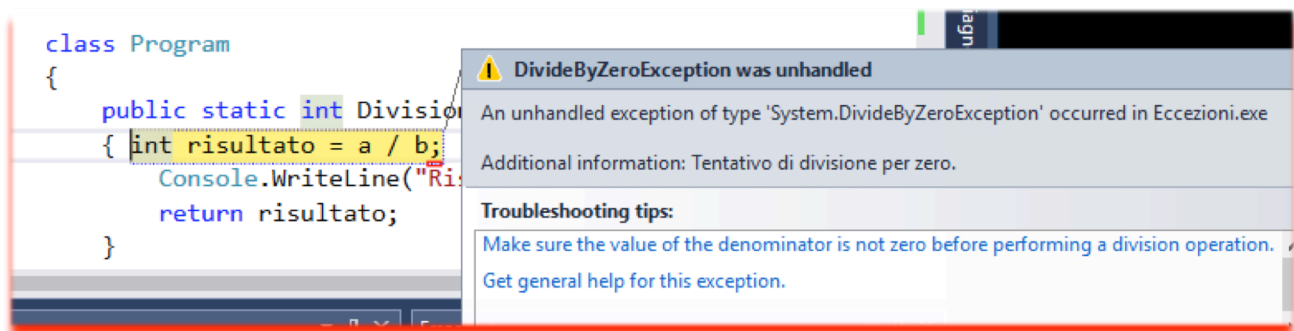
## Gestione strutturata degli errori tramite le exception.

Per comprendere i vantaggi di una gestione strutturata degli errori osserviamo il seguente codice:

```
class Program
{
    public static int Divisione(int a, int b)
    { int risultato = a / b;
      Console.WriteLine("Risultato calcolato con successo");
      return risultato;
    }

    static void Main(string[] args)
    {
        int ris = Divisione(6, 0);
        Console.WriteLine("Il risultato è {0}", ris);
    }
}
```

La sua esecuzione viene interrotta e, al contesto di runtime viene associato un particolare oggetto di tipo `DivideByZeroException`, rappresentativo della tipologia di errore che si è verificata.



Quando un metodo solleva un eccezione il CLR (Common Language Runtime ) interrompe il flusso logico di esecuzione, per passare alla ricerca di un eventuale gestore, cioè di un blocco di codice opportunamente marcato come in grado di prenderla in carico. Se la ricerca non è eseguita con successo, il gestore viene eseguito. In caso contrario, invece, l'eccezione viene marcata come non eseguita e provoca la chiusura dell'applicazione.

Tutte le volte che si ha a che fare con un codice che potenzialmente può sollevare un'eccezione e vogliamo essere in grado di gestire questa eventualità possiamo avvalerci del costrutto `try{ } catch { }`, grazie al quale, quando il codice all'interno del blocco `try` genera un'eccezione, ne viene interrotta l'esecuzione e il controllo viene passato al blocco `catch` che corrisponde al tipo dell'eccezione sollevata.

```
class Program
{
    public static int Divisione(int a, int b)
    { int risultato = a / b;
      Console.WriteLine("Risultato calcolato con successo");
      return risultato }
}
```

```

static void Main(string[] args)
{
    try {
        int result = Divisione(6, 0);
        Console.WriteLine("Risultato calcolato con successo");
    }
    catch (Exception e)
    { Console.WriteLine("si è verificato un errore");
      Console.ReadLine();
    }
}
}

```

Il risultato quindi è che l'eccezione viene gestita e l'applicazione non termina più in maniera anomala.

In generale un blocco `try{ } catch { }`, può contenere molteplici clausole `catch`, in modo da prevedere diversi gestori per specifiche tipologie di eccezione.

```

static void Main(string[] args)
{
    try {
        int result = Divisione(6, 0);
        Console.WriteLine("Risultato calcolato con successo");
    }
    catch (DivideByZeroException)
    { Console.WriteLine("si è verificato un errore");
    }
    catch (OutOfMemoryException)
    {
        Console.WriteLine("Memoria Terminata");
    }
    catch (Exception)
    {
        Console.WriteLine("Si è verificato un errore generico");
    }
    Console.ReadLine();
}

```

Il risultato è il medesimo se si indica come eccezione da gestire `DivideByZeroException` o `ArithmeticException`, da cui essa deriva, mentre l'applicazione tornerebbe a chiudersi con un errore se il gestore fosse relativo a un tipo differente come, per esempio, `OutOfMemoryException`

Quindi quando si utilizza questo approccio occorre stare attenti all'ordine secondo cui vengono disposti i blocchi `catch`, dato che vengono valutati in sequenza dal CLR, finché non ne viene trovato uno adatto.

Se mettessi in prima posizione `catch (Exception)` essendo in grado di gestire qualsiasi eccezione impedirebbe di fatto l'esecuzione di quelli più specifici

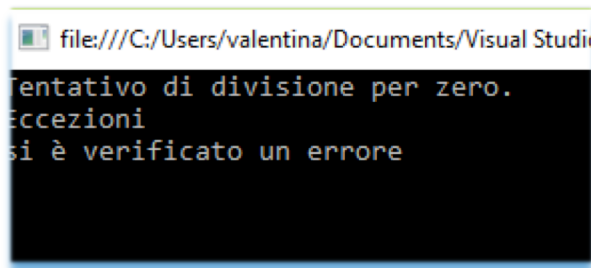
### Classe `System.Exception`

Al verificarsi di un'eccezione, il CLR associa al contesto di esecuzione un'istanza di un particolare oggetto il cui tipo è (o deriva da) `Exception`. La classe `Exception` è capostipite di una numerosa

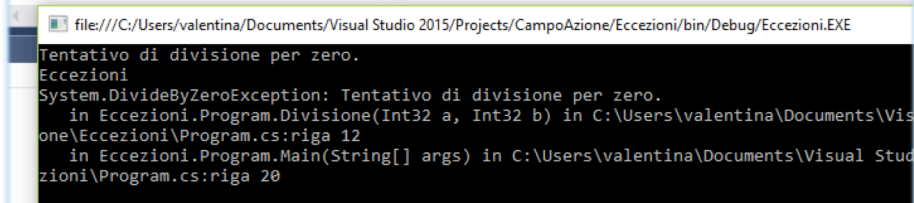
`ToString()`: Metodo che produce una stringa contenente la natura dell'eccezione.

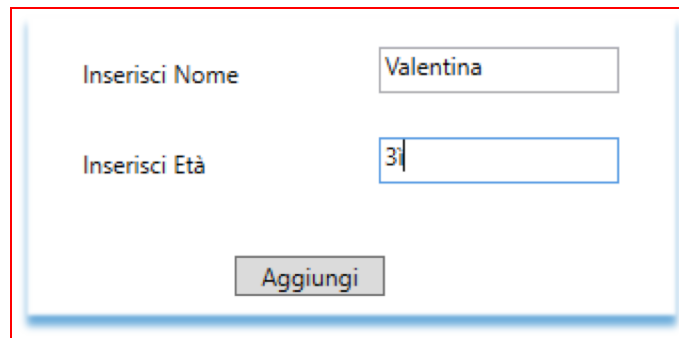
```
catch (DivideByZeroException e)
{
    Console.WriteLine(e.Message);
    Console.WriteLine(e.Source);

    Console.WriteLine("si è verificato un errore");
}
```

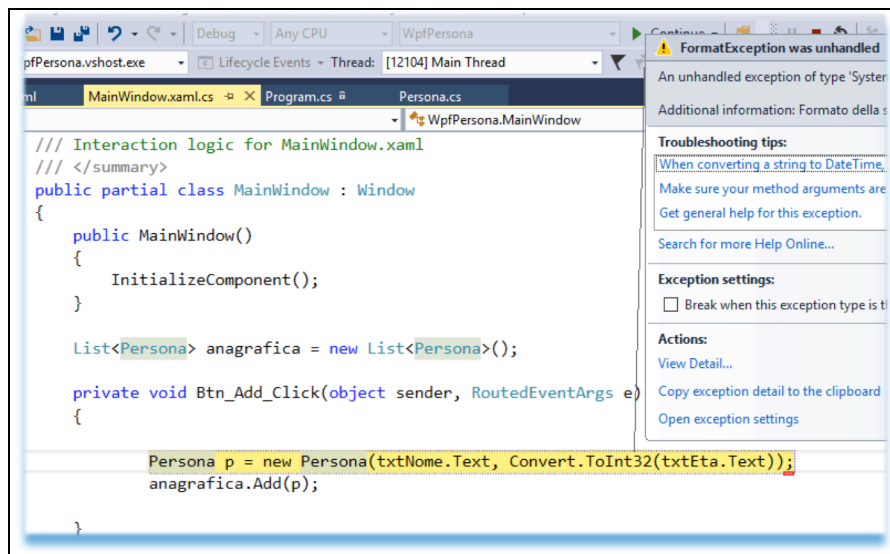


```
static void Main(string[] args)
{
    try {
        int result = Divisione(6, 0);
        Console.WriteLine("Risultato calcolato con successo");
    }
    catch (DivideByZeroException e)
    {
        Console.WriteLine(e.Message);
        Console.WriteLine(e.Source);
        string a = e.ToString();
        Console.WriteLine(a);
    }
}
```





Al Click del pulsante aggiungi, con un codice che non prevede la gestione delle eccezioni, il programma si blocca:



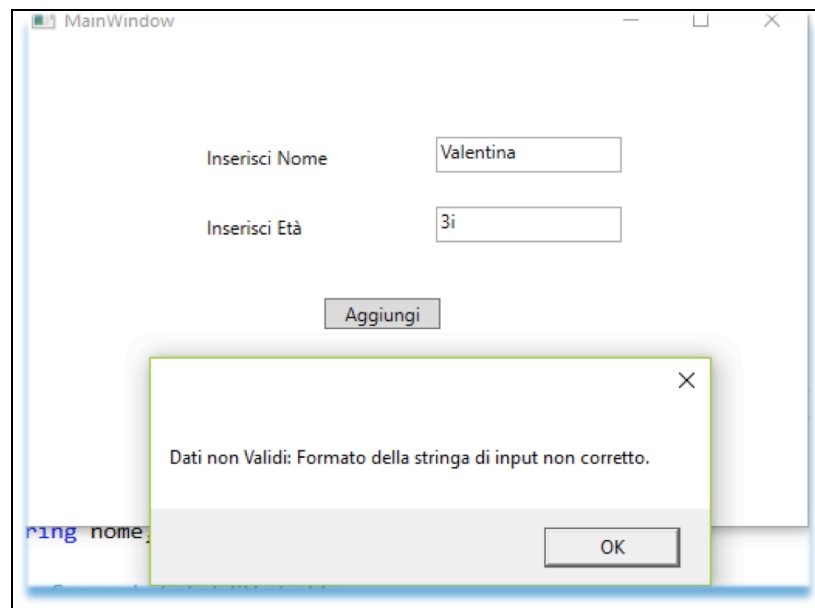
E' necessario quindi catturare l'eccezione:

```
List<Persona> anagrafica = new List<Persona>();

private void Btn_Add_Click(object sender, RoutedEventArgs e)
{
    try
    {
        //Persona p = new Persona(txtNome.Text, Convert.ToInt32(txtEta.Text));
        //anagrafica.Add(p);
        AggiungiPersona(txtNome.Text, txtEta.Text);
    }
    catch (FormatException ex)
    {
        MessageBox.Show("Dati non Validi" + ex.Message);
    }
}
```

```
private void AggiungiPersona(string nome, string eta)
{
    Persona p = new Persona(nome, Convert.ToInt32(eta));
    anagrafica.Add(p);
}
```





## Eccezioni: esercizi.

Spiegare perché i seguenti frammenti di codice sollevano un'eccezione e di che tipo.

Provare il codice in Visual Studio, consegnare la correzione e la relazione il 27 ottobre.

Catturare l'eccezione con il costrutto:

```
try
{
    // codice che si vuole proteggere dalle eccezioni
}
catch(tipo-eccezione nome-variabile_opz)
{
    // codice che gestisce l'eccezione
}
```

### Esempio 1

```
class Program
{
    static int[] vettore;
    static void Main(string[] args)
    {
        Array.Clear(vettore, 0, 10);
    }
}
```

### Esempio 2

```
int[] v1 = { 1, 2, 3, 4 };
int[] v2 = new int[v1.Length];
Array.Copy(v1, v2, -1);
```

### Esempio 3

```
int[] v1 = { 1, 2, 3, 4 };
int[] v2 = new int[v1.Length];
Array.Copy(v1, v2, 5);
```

### Esempio 4

```
double d2 = double.Parse("1000,,0");
double d1 = double.Parse("1000");
bool b1 = bool.Parse("false");
```

```
bool b2 = bool.Parse("false");
```

### Esempio 5

```
object[] vettoreGenerico = { "Fermi", 10, "Einstein" };  
    Array.Sort(vettoreGenerico);
```

### Esempio 6

```
int[] vettore = { 1, 2, 3, 4 };  
    int somma = 0;  
    for (int i = 0; i <= vettore.Length; i++)  
    { somma += vettore[i]; }  
    Console.WriteLine(somma);
```

## DTO Data Transfer Object

Esistono tre tipologie di classi:

- Classi che fanno riferimento ad oggetti del mondo reale, con attributi, proprietà, costruttori, metodi
- Classi statiche
- DTO: Data Transfer Object, una classe contenitore di valore.

Nella nuova versione dell'esercizio Calcolatrice in cui separo elaborazione con interfaccia non ho ancora ben gestito i messaggi di errore (Se separo interfaccia ed elaborazione nella parte elaborazione non posso scrivere MessageBox...!).

Il problema nasce dal fatto che il metodo somma nella classe operazione restituisce solo un risultato e non i messaggi di errore con la relativa tipologia.

Si utilizza un nuovo tipo di oggetto: DTO Data Transfer Object. Si tratta di oggetti che servono a trasferire dati. Sono contenitori di valori.

Creo una classe DTO che contiene gli elementi che dovrebbe restituire il metodo.

La classe permette di far comunicare l'elaborazione con l'interfaccia.

```
public class Risultato
{
    public string Messaggio;
    public double Ris;
    public bool IsError;
}
```

**NB. La definizione si chiama classe, quando la uso si chiama oggetto!**

Ecco come cambia ora il codice del metodo Somma nella classe Operazioni

```
public static Risultato Somma(string strnum1, string strnum2)
//invece di restituire un double vorrei restituire due "cose" es. un double e una stringa
//il metodo restituisce un oggetto Risultato

{
    double somma = 0;
    double num1;
    double num2;

//creo l'oggetto risposta
    Risultato risposta = new Risultato();
    if (strnum1 != "" && strnum2 != "")
    {
        try
        {
            num1 = Convert.ToInt32(strnum1);
            num2 = Convert.ToInt32(strnum2);
            somma = num1 + num2;
        }
    }
}
```

```

//se tutto va bene!
        risposta.Messaggio = "ok";
        risposta.Valore = somma;
        risposta.IsError = false;
    }
    catch (Exception) //cattura il problema, il programma salta e manda un
messaggio di errore
    {
        somma = Double.NaN;
        risposta.Messaggio = "Conversione errata";
        risposta.Valore = somma;
        risposta.IsError = true;
    }
}
else
{
    somma = Double.NaN;
    risposta.Messaggio = "Mancano i numeri";
    risposta.Valore = somma;
    risposta.IsError = true;
}

//restituisce "risposta"
// faccio la return di risposta
return risposta;
}

```

Passiamo alla parte relativa all'interfaccia.

Ora la variabile che ottiene il contenuto dell'Operazione somma sarà di tipo risultato.

```

private void Somma_con_Eccezione_Click(object sender, RoutedEventArgs e)
{
    string strnum1 = txtNum1.Text;
    string strnum2 = txtNum2.Text;
    Risultato risposta=Operazioni.Somma(strnum1, strnum2);

    if (!risposta.IsError)
    {
        lblRisultato.Content = risposta.Valore; }

    Else
// adesso posso far visualizzare il messaggio giusto contenuto in risposta.messaggio
    {
        MessageBox.Show(risposta.Messaggio, "Errore", MessageBoxButton.OK,
        MessageBoxImage.Information); }
}

```

## Esercizio 1

Quali attributi possiede la classe e quali proprietà?

Come si utilizzano le proprietà? Che funzione hanno?

Che cosa cambia se elimino `set { num1 = value; }`?

E se elimino `get { return _valore; } }`?

```
public class Calcolatrice
{
    private int _operazione;
    public int Operazione
    {
        get { return _operazione; }
        set { _operazione = value; }
    }
    private double _num1;
    public double Num1
    {
        get { return _num1; }
        set { _num1 = value; }
    }
    private double _num2;
    public double Num2
    {
        get { return _num2; }
        set { _num2 = value; }
    }

    private double _valore;
    public double Valore
    {
        get { return _valore; }
        set { _valore = value; }
    }

    private bool _isError;
    public bool IsError
    {
        get { return _isError; }
        set { _isError = value; }
    }

    private string _messaggio;
    public string Messaggio
    {
        get { return _messaggio; }
        set { _messaggio = value; }
    }
}
```

## Esercizio 2

Che cosa rappresenta la seguente porzione di codice?

Descrivi il significato di ciascuna riga.

```
public Calcolatrice()
{
    Valore = double.NaN;
    Messaggio = "Numeri mancanti";
    IsError = true;
}
```

## Esercizio 3

Che cosa rappresenta la seguente porzione di codice?

Che relazione ha con l'esercizio 2?

```
public partial class MainWindow : Window
{
    private Calcolatrice calc;
    public MainWindow()
    {
        InitializeComponent();
        calc = new Calcolatrice();
    }
}
```

## Esercizio 4

La seguente porzione di codice fa parte dell'elaborazione o dell'interfaccia? Perché?

Quando viene eseguita?

```
private void btnSomma_Click(object sender, RoutedEventArgs e)
{
    calc.InseriscePrimoOperando(txtNumero.Text);
    calc.InserisceOperazione(1);
    if (calc.IsError)
        MessageBox.Show(calc.Messaggio, "Errore", MessageBoxButton.OK,
        MessageBoxImage.Error);
    txtNumero.Focus();
    txtNumero.Select(0, txtNumero.Text.Length);
}
```

## Esercizio 5

Nella classe Calcolatrice sono presenti i seguenti metodi.

Perché? Come sono in relazione con l'esercizio 4?

```
public void InserisceOperazione(int operazione)
{
    Operazione = operazione;
    Valore = double.NaN;
    Messaggio = "OK";
    IsError = false;
}
```

```

public void InseriscePrimoOperando(string strOperando)
{
    Valore = double.NaN;

    if (strOperando != "")
    {
        try
        {
            Num1 = Convert.ToDouble(strOperando);
            Messaggio = "OK";
            IsError = false;
        }
        catch (Exception)
        {
            Messaggio = "Conversione errata";
            IsError = true;
        }
    }
    else
    {
        Messaggio = "Numeri mancanti";
        IsError = true;
    }
}

```

## Esercizio 6

Riproduci sul tuo quaderno l'interfaccia

Numero

+

-

\*

/

Clear

=

e spiega passo passo, seguendo il codice dell'esercizio 4 e 5, cosa accade ad ogni azione dell'utente.



## Esercizio 7

In riferimento all'esercizio 6 cosa accade quando l'utente preme il pulsante =?

Dopo aver risposto alla domanda precedente spiega il codice? Fa parte dell'interfaccia o dell'elaborazione? Perché?

```
private void btnUgual_Click(object sender, RoutedEventArgs e)
{
    calc.InserisceSecondoOperando(txtNumero.Text);
    if (calc.IsError)
        MessageBox.Show(calc.Messaggio, "Errore", MessageBoxButton.OK,
        MessageBoxImage.Error);
    else
    {
        calc.EsegueOperazione();
        if (!calc.IsError)
            txtNumero.Text = calc.Valore.ToString();
        else
            MessageBox.Show(calc.Messaggio, "Errore", MessageBoxButton.OK,
            MessageBoxImage.Error);
    }
    txtNumero.Focus();
    txtNumero.Select(0, txtNumero.Text.Length);
}
```

## Esercizio 8.

Che cosa rappresenta il seguente codice? Fa parte dell'elaborazione o dell'interfaccia?

Che relazione ha con l'esercizio precedente?

Quali proprietà utilizzano?

```
public void InserisceSecondoOperando(string strOperando)
{
    Valore = double.NaN;

    if (strOperando != "")
    {
        try
        {
            Num2 = Convert.ToDouble(strOperando);
            Messaggio = "OK";
            IsError = false;
        }
        catch (Exception)
        {
            Messaggio = "Conversione errata";
            IsError = true;
        }
    }
    else
    {
        Messaggio = "Numeri mancanti";
        IsError = true;
    }
}
```

```

public void EsegueOperazione()
{
    if (Operazione == 1)
        Somma();
    if (Operazione == 2)
        Sottrazione();
    if (Operazione == 3)
        Prodotto();
    if (Operazione == 4)
        Divisione();
}

```

## Esercizio 9

Potresti prevedere l'uso di una classe DTO?

Come?

Spiega quali modifiche andrebbero fatte al codice?

Soluzione:

```

namespace Elaborazione
{
    public class Risultato
    {
        public bool IsError;
        public string Messaggio;
        public double Valore;
    }
}

```

```

namespace Elaborazione
{
    public class Calcolatrice
    {
        private Risultato _risposta;
        public Risultato Risposta
        {
            get { return _risposta; }
        }

        private double _num1;
        private double _num2;

        private int _operazione;
        public int Operazione
        {
            get { return _operazione; }
        }

        public Calcolatrice()
        {
            _num1 = double.NaN;
            _num2 = double.NaN;
            _risposta = new Risultato();
        }
    }
}

```

```

}

public void Clear()
{
    _num1 = double.NaN;
    _num2 = double.NaN;
    _operazione = 0;
}

public double ImpostaNumero1(string num1)
{
    if (num1!="")
    {
        try
        {
            _num1 = Convert.ToDouble(num1);

            _risposta.Messaggio = "OK";
            _risposta.IsError = false;
        }
        catch (Exception)
        {
            _risposta.Messaggio = "Conversione errata";
            _risposta.IsError = true;
        }
    }
    else
    {
        _risposta.Messaggio = "Numero mancante";
        _risposta.IsError = true;
    }
    return _num1;
}

public double ImpostaNumero2(string num2)
{
    if (num2 != "")
    {
        try
        {
            _num2 = Convert.ToDouble(num2);

            _risposta.Messaggio = "OK";
            _risposta.IsError = false;
        }
        catch (Exception)
        {
            _risposta.Messaggio = "Conversione errata";
            _risposta.IsError = true;
        }
    }
    else
    {
        _risposta.Messaggio = "Numero mancante";
        _risposta.IsError = true;
    }
    return _num2;
}

```

```

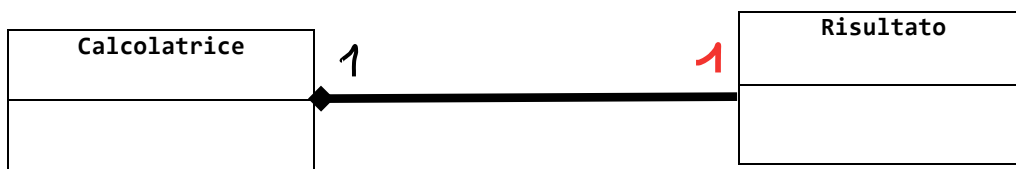
public void ImpostaOperazione(int op)
{

    if (_risposta.IsError== false && op==1)
    { Somma(); }

}

public void Somma()
{
    if (!_risposta.IsError)
    {
        _risposta.Valore = _num1 + _num2;
        _risposta.IsError = false;
        _risposta.Messaggio = "OK";
    }
}
}

```



La relazione tra le due classi prende il nome di composizione.

Rappresenta una relazione del tipo tutto(calcolatrice)-parte(risultato).

Quando viene creato il tutto vengono creati anche i suoi componenti, quando viene cancellato il tutto, viene cancellato anche il suo componente.

La molteplicità (numero di istanze dell'oggetto che sono presenti nella classe) è sempre 1 o 0..1

Una relazione di questo tipo si legge: "Risultato è parte di Calcolatrice".

Quando un componente ha molteplicità con limite inferiore 1 allora deve essere creato al momento della creazione dell'oggetto composto e quindi all'interno del costruttore.

```

public Calcolatrice()
{
    _num1 = double.NaN;
    _num2 = double.NaN;
    _risposta = new Risultato();
}

```

```

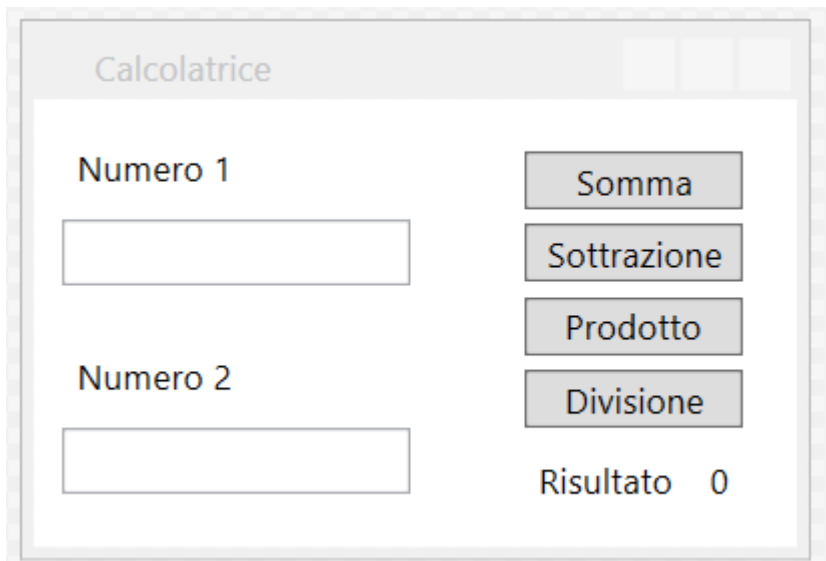
private Risultato _risposta;
public Risultato Risposta
{
    get { return _risposta; }
}

```

Se invece la molteplicità del componente ha molteplicità inferiore pari a 0 (0..) allora si può creare il componente in qualsiasi momento dopo la creazione dell'oggetto composto (con un metodo ad esempio).

```
        public void CreaComponente()  
{  
    _nomeComponente = new NomeClasseComponente();  
}
```

## Application programming interface



Definizione generica di interfaccia: "come relazionarsi, interagire con un oggetto".

In informatica:

Interfaccia utente (User Interface -Figura sopra-) di un'applicazione: come l'utente interagisce con l'applicazione. Il modo come un utente usa un'applicazione.

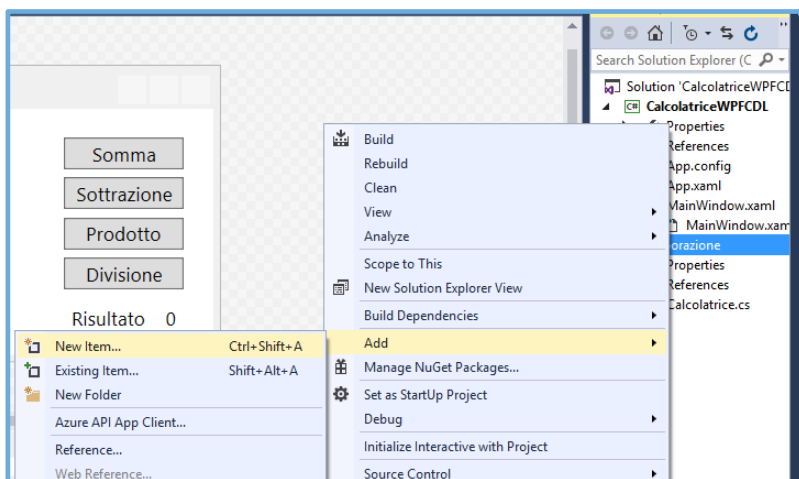
Obiettivo: capire come l'interfaccia utente interagisce con l'elaborazione.

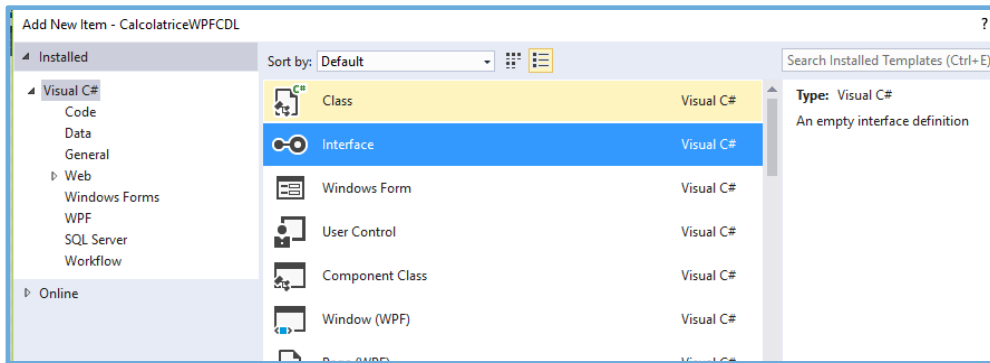
API- Interfaccia di programmazione di un'applicazione- Insieme di procedure disponibili al programmatore. I metodi e le proprietà di una classe rappresentano l'interfaccia di programmazione dell'applicazione. Tutto ciò che è **pubblico** è per definizione un'API. Il codice è l'API.

API mette a disposizione oggetti Device Independent, indipendenti dal device.

Creiamo un'API.

All'interno del progetto libreria.





Come nominarla

Se è un'interfaccia per la classe: **INomeClasse.cs** (es. ICalcolatrice.cs)

Oppure assegno il nome della funzionalità.

Osserviamo il codice:

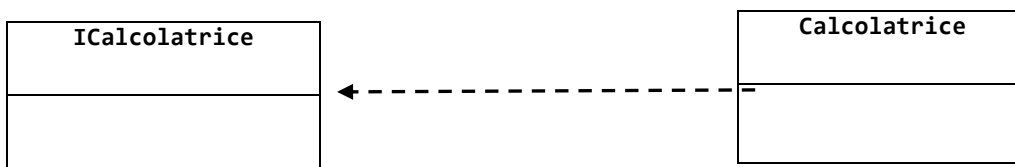
```
public interface ICalcolatrice
{
}
```

Non compare class ma interface. Ovviamente deve essere pubblica.

## Interfacce e UML

Nel diagramma UML:

Calcolatrice usa/dipende ICalcolatrice



## Prima proposta di soluzione

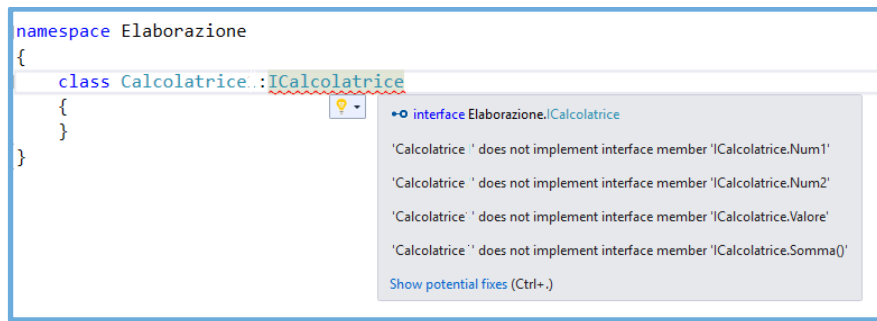
```
public interface ICalcolatrice
{
    double Num1 {set;}
    double Num2 { set; }
    double Ris { get;}
    void Somma();
}
```

Si scrive cosa fare e non come fare!

Le interfacce devono essere implementate da una classe.

Si tratta di una relazione di dipendenza.

La classe usa una interfaccia, “implementa un’interfaccia”.



```
class Calcolatrice:ICalcolatrice
{
    private double _num1;
    public double Num1
    {
        set { _num1 = value; }
    }
    private double _num2;
    public double Num2
    {
        set { _num2 = value; }
    }

    private double _ris;
    public double Ris
    {
        get { return _ris; }
    }

    public void Somma()
    { _ris = _num1 + _num2; }

    //il costruttore va scritto nella classe e non nella interfaccia
    public Calcolatrice()
    { }
}
```

Ecco il codice relativo all’interfaccia grafica (UI)

```
private void btnSomma_Click(object sender, RoutedEventArgs e)
{
    Calcolatrice Calc = new Calcolatrice();
    //controlli posticipati lasciati allo studente.
    double n1 = Convert.ToDouble(txtNumero1.Text);
    double n2 = Convert.ToInt32(txtNumero2.Text);
    Calc.Num1=n1;
    Calc.Num2= n2;
    Calc.Somma();
    lblRisultato.Content = Calc.Ris;
}
```



## Seconda Proposta di soluzione

```
public interface ICalcolatrice
{
    void ImpostaNumeri(double n1, double n2);
    double Ris { get; }
    void Somma();
}
```

```
public class Calcolatrice : ICalcolatrice
{
    private double _ris;
    public double Ris
    {
        get { return _ris; }
    }

    private double _n1;
    private double _n2;

    public void ImpostaNumeri(double n1, double n2)
    {
        _n1 = n1;
        _n2 = n2;
    }
    public void Somma()
    { _ris = _n1 + _n2; }

    //il costruttore va scritto nella classe e non nella interfaccia
    public Calcolatrice() { }
}
```

Ecco il codice relativo all'interfaccia grafica (UI):

```
public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
    }
    private void btnSomma_Click(object sender, RoutedEventArgs e)
    {
        Calcolatrice Calc = new Calcolatrice();
        //controlli posticipati lasciati allo studente.

        double n1 = Convert.ToDouble(txtNumero1.Text);
        double n2 = Convert.ToDouble(txtNumero2.Text);
        Calc.ImpostaNumeri(n1, n2);
        Calc.Somma();
        lblRisultato.Content = Calc.Ris;
    }
}
```

### Terza Proposta di soluzione

```
public interface ICalcolatrice
{

    double Ris { get; }
    void Somma(double n1, double n2);

}
```

```
public class Calcolatrice : ICalcolatrice
{
private double _ris;
    public double Ris
    {
        get { return _ris; }
    }

    public void Somma(double n1, double n2)
    { _ris = n1 + n2; }
}

//il costruttore va scritto nella classe e non nella interfaccia
public Calcolatrice()
{ }
```

Ecco il codice relativo all'interfaccia grafica (UI):

```
public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
    }

    private void btnSomma_Click(object sender, RoutedEventArgs e)
    {
        Calcolatrice1 Calc = new Calcolatrice1();
        double n1 = Convert.ToDouble(txtNumero1.Text);
        double n2 = Convert.ToInt32(txtNumero2.Text);
        //Controlli posticipati lasciati allo studente
        Calc.Somma(n1,n2);
        lblRisultato.Content = Calc.Ris;
    }

    .....
}
```

## ESERCITAZIONE API

Obiettivi:

- Creare una libreria compito
- Creare l'API ICompito
- Creare la classe Compito

### Primo metodo.

Creazione dell'API

```
public interface ICompito
{
    double Numero1 { get; set; }
    double Numero2 { get; set; }
    void Moltiplica();
    void Dividi();
}
```

Implementazione della classe:

```
public class Compito : ICompito
{
    private double _numero1;
    public double Numero1
    {
        get
        {
            return _numero1;
        }
        set
        {
            _numero1 = value;
        }
    }

    private double _numero2;
    public double Numero2
    {
        get
        {
            return _numero2;
        }
        set
        {
            _numero2 = value;
        }
    }
    public void Dividi()
    {
        _numero1 = _numero2 / 2;
    }
}
```

```

    public void Moltiplica()
    {
        _numero2 = _numero1 * 4;
    }
}

```

Interfaccia Grafica

```

private void btnDividePerDue_Click(object sender, RoutedEventArgs e)
{
    Compito c = new Compito();
    c.Numero2 = Convert.ToDouble(txtNumero2.Text);
    c.Dividi();
    txtNumero1.Text = c.Numero1.ToString();
}

private void btnMoltiplicaPerQuattro_Click(object sender, RoutedEventArgs e)
{
    Compito c = new Compito();
    c.Numero1 = Convert.ToDouble(txtNumero1.Text);
    c.Moltiplica();
    txtNumero2.Text = c.Numero2.ToString();
}

```

**Secondo metodo.**

Nell'Interfaccia prevedo un unico metodo che in base al parametro passato esegue l'operazione prodotto o divisione.

```

public interface ICompito
{
    double Numero1 { get; set; }
    double Numero2 { get; set; }
    void Elabora(string operazione);
}

```

Si passa all'implementazione della classe.

Osservare il codice del metodo Elabora.

```

public class Compito : ICompito
{
    private double _numero1;
    public double Numero1
    {
        get
        {
            return _numero1;
        }

        set
        {
            _numero1 = value;
        }
    }
}

```

```

private double _numero2;
public double Numero2
{
    get
    {
        return _numero2;
    }

    set
    {
        _numero2 = value;
    }
}

public void Elabora(string operazione)
{
    if (operazione == "Prodotto")
        _numero2 = _numero1 * 4;
    else
        _numero1 = _numero2 / 2;
}
}

```

Interfaccia grafica:

```

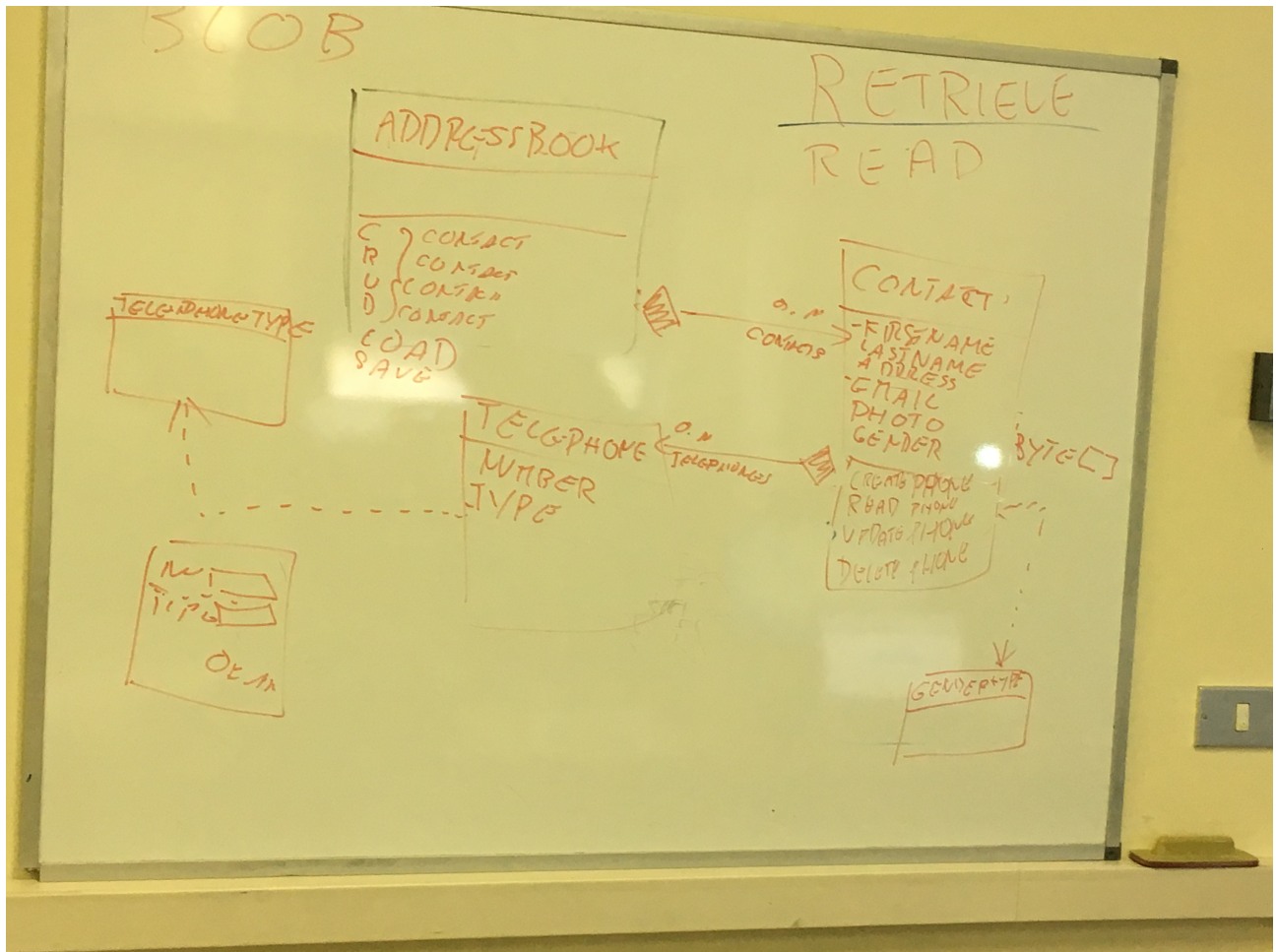
private void btnDividePerDue_Click(object sender, RoutedEventArgs e)
{
    Compito c = new Compito();
    c.Numero2 = Convert.ToDouble(txtNumero2.Text);
    c.Elabora("Divisione");
    //Osservare che invece di invocare il metodo c.Dividi(); si invoca il metodo Elabora
    //passando come parametro la stringa "prodotto"
    txtNumero1.Text = c.Numero1.ToString();
}

private void btnMoltiplicaPerQuattro_Click(object sender, RoutedEventArgs e)
{
    Compito c = new Compito();
    c.Numero1 = Convert.ToDouble(txtNumero1.Text);
    c.Elabora("Prodotto");
    txtNumero2.Text = c.Numero2.ToString();
}

```

## AddressBookManagement.NET

### Progettazione UML



### Implementazione classi

Nell'implementazione delle classi di parte sempre dalla classe figlio

```
//Quando il dominio dei valori è limitato è "best practice" usare un enum
public enum TelephoneType
{
    Mobile,
    Home,
    Fax
}
```

Pertanto l'attributo type, inizialmente inserito nella classe Telephone diventa un enumeratore. E la relazione tra la classe Telephone ed TelephoneType è di tipo Dipendenza.

Infatti notiamo che, nella classe Telephone, l'attributo Type non è di tipo stringa ma di tipo TelephoneType.

```

public class Telephone:ITelephone
{
    //esiste un metodo più semplice: syntactic sugar
    public string Number { get; set; }
    public TelephoneType Type { get; set; }

    //Il costruttore con due parametri verrà utilizzato nella classe Contact per creare un
    oggetto di Telephone

    public Telephone(string number, TelephoneType type)
    {
        Number = number;
        Type = type;
    }
}

```

### Proprietà automatiche.

Le proprietà sono membri che permettono di accedere in lettura e in scrittura ai campi di una classe. In genere, una proprietà definisce una coppia di metodi (uno per la lettura e uno per la scrittura) e a questi blocchi associa un nome identificativo. La dichiarazione di una proprietà viene effettuata specificando due blocchi di codice contrassegnati dalle parole chiave `get` e `set` (rispettivamente lettura e scrittura). Generalmente le proprietà vengono associate ad un campo privato, detto *backing field*, e che contiene effettivamente il valore. Così è possibile passare una logica ai due metodi legati al caricamento e salvataggio dei dati (chiamati anche *getter* e *setter*).

```

string _number;
public string Number
{
    get { return _number; }
    set { _number = value; }
}

```

In C# è possibile definire le proprietà in modo alternativo attraverso le proprietà automatiche. Tali proprietà rappresentano una modalità di dichiarazione molto compatta che ci evita di dover dichiarare il campo privato, reso accessibile tramite la proprietà, e di dover scrivere il codice che implementa gli accessor per la scrittura e per la lettura in modo esplicito. In questo tipo di dichiarazione è il compilatore che crea in modo totalmente trasparente per il programmatore sia il campo privato, sia il contenuto degli accessori delle proprietà. Si osservi inoltre come il codice sia maggiormente compatto ed essenziale, di conseguenza si ha un miglioramento della leggibilità.

Pertanto il codice si riduce a `public string Number { get; set; }`

La classe implementa l'interfaccia `ITelephone`. Si ricorda che l'interfaccia è pubblica, e racchiude tutti i metodi pubblici della classe escluso il costruttore

```

public interface ITelephone
{
    string Number { get; set; }
    TelephoneType Type { get; set; }
}

```

Si procede con IContact e Contact

```
public interface IContact
{
    string FirstName { get; set; }
    string LastName { get; set; }
    string Email { get; set; }
    string Address { get; set; }
    //Anche il genere è un enumeratore, trattandosi di un insieme limitato di valori
    //FirstName, LastName e gli altri campi non possono essere enumeratori in quanto il loro
    //dominio di valori è ILLIMITATO
    GenderType Gender { get; set; }

    //Un'immagine è un vettore di byte (BLOB
    byte[] Photo { get; set; }
}
```

Aggiungiamo dunque al progetto un altro item (class) che chiameremo GenderType e sarà un enumeratore in relazione di dipendenza con Contact.

```
//il tipo booleano per va usato solo in situazioni vero/falso
//il genere in quale caso è vero o falso?
//In caso di domini limitati si usa un enum
public enum GenderType
{
    Male,
    Female
}
```

Procediamo con l'implementazione della classe.

```
public class Contact:IContact
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Email { get; set; }
    public string Address { get; set; }
    public GenderType Gender { get; set; }
    public byte[] Photo { get; set; }

    //Per mettere in relazione la classe Contact con Telephone attenzione alla molteplicità. Se
    //la molteplicità è a molti usare lista

    public List<Telephone> Telephones { get; set; }

    public Contact()
    { Telephones = new List<Telephone>(); }
}
```

In composizione il contatto si occupa di creare aggiungere, togliere numeri di telefono.

Il contatto ha quindi potere di "vita di morte" sul numero di telefono, pertanto crea e distrugge l'oggetto telefono.



```

        public void CreatePhone(string number, TelephoneType type)
        { //è necessario creare l'oggetto telefono, utilizzando il costruttore vuoto, di
default
            Telephone newPhone = new Telephone();
            newPhone.Number = number;
            newPhone.Type = type;
//una volta creato l'oggetto lo aggiungo alla lista Telephones
            Telephones.Add(newPhone);
        }

```

Il metodo CreatePhone si potrebbe riscrivere meglio utilizzando il costruttore creato nella classe Telephone.

```

Public class Telephone{
.....

    public Telephone(string number, TelephoneType type)
    {
        Number = number;
        Type = type;
    }
}

```

Pertanto il codice si riscrive in:

```

        public void CreatePhone(string number, TelephoneType type)
        {
//in questo caso utilizzo il costruttore creato nella classe Telephone con parametri number
e type
        Telephone newPhone = new Telephone(number, type);
//Una volta creato l'oggetto si aggiunge alla lista Telephones
        Telephones.Add(newPhone);
        }

```

Si procede con la classe AddressBook e la sua interfaccia IAddressBook

```

public interface IAddressBook
{
    List<Contact> Contacts { get; set; }

    void CreateConctat(string firstname, string lastname, string mail, string address,
GenderType gender, byte[] photo);
}

```

AddressBook è sempre in composizione con Contact. La relazione è uno a molti. Pertanto in AddressBook è necessario inserire un campo lista di tipo Contact che si chiamerà al plurale Contacts. Prima di procedere con la creazione dell'oggetto di contact è necessario rivedere anche in Contact i costruttori.

**Esercizio: terminare AddressBook.**

## Serializzare e de-serializzare

Le famose “best practices” del .NET ci consigliano anche di dare la possibilità al codice consumer di creare un oggetto a partire da una stringa.

Per poter fare ciò ci servirà un meccanismo per “serializzare” gli attributi di una classe in una stringa e di conseguenza un sistema per “de-serializzare” la stringa negli attributi della classe.

Serializzare significa trasformare informazioni in una stringa. Esportare dati.

Il metodo ToString() è un metodo per serializzare.

```
public override string ToString()
{
    string tmp="";
    tmp = String.Format("{0},{1}", Number, Type);
    return tmp;
}
```

Obbligatorio: Ogni classe che manipola dati deve avere il metodo ToString().

Il separatore "{0},{1}" è importante, nel nostro caso è la virgola.

Si sceglie il separatore in base a quello che non viene usato dalla variabile. Si sceglie il separatore in base al tipo di dato. Normalmente la virgola, punto e virgola sono i separatori migliori (il punto, il segno meno, non possono essere buoni separatori perché vengono utilizzati nei calcoli matematici) .

```
tmp = String.Format("{0},{1}", Number, Type);
```

De-serializzare: ricavare informazioni da una stringa.

Per deserializzare e quindi costruire un oggetto a partire da una stringa, abbiamo bisogno di un altro costruttore che accetti questo parametro.

Parse significa analizzare qualcosa e creare un contenuto.

Per deserializzare ho bisogno di un costruttore che accetta in ingresso una stringa

Il programmatore crea il serializzatore, e il programmatore in base al serializzatore crea il deserializzatore.

```
public Telephone(string str)
{
    string[] parts = str.Split(',');
    Number = parts[0];
    Type = (TelephoneType)Enum.Parse(typeof(TelephoneType), parts[1], false);
}
```

Vediamo in dettaglio il codice.

Il metodo Split richiede il separatore cioè il carattere per decidere come separare

Dichiaro un vettore parts:

```
string[] parts = str.Split(',');
```

Ogni elemento del vettore conterrà una delle parti separate in base al separatore, che nel nostro caso è la virgola

```
Number = parts[0];
```

Finché si ha una stringa non si hanno problemi, ma Type è un enumeratore e non una stringa

Andrebbe fatta una conversione, invece di utilizzare “ConvertTo” esiste un metodo “Parse” che prende una stringa e la trasforma in enumeratore

Poiché nell'interfaccia sono tutte stringhe è preferibile utilizzare il metodo “Parse”

Attenzione nell'enumeratore perchè abbiamo tante possibilità, già nel nostro esercizio ce ne sono due differenti.

```
Type = (TelephoneType)Enum.Parse(typeof(TelephoneType),parts[1],false);
```

(TelephoneType) indica il tipo di dato da trasformare, parts[1] indica la parte del vettore in cui verrà memorizzato, false indica che la tipologia di telefono va bene scritta in tutti i modi (es.tutto minuscolo, tutto maiucolo...).

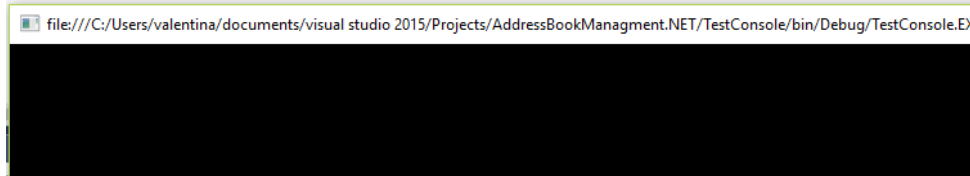
Se metto true la tipologia deve essere scritta esattamente nello stesso modo come è stata scritta al momento della creazione dell'enumeratore.

Con questo metodo si riesce a separare la stringa nelle sue singole parti, poi si prendono i "pezzi" risultanti e si sono inizializzano gli attributi.

Purtroppo questo costruttore si comporta a dovere solamente in presenza di una stringa "well-formed" cioè contenente dati completamente corretti e nell'ordine corretto.

Se scriviamo questo frammento di codice per provare il costruttore, non si ha output:

```
string stringa = "075, fax";  
//attenzione a come lo scrivo,  
//si aspetta esattamente come è scritto l'enumeratore  
//se scrivo in minuscolo non lo accetta  
Telephone tel2 = new Telephone(stringa);  
Console.WriteLine(tel2.ToString());
```



Infatti l'enumeratore è Fax e non fax e pertanto

```
Type = (TelephoneType)Enum.Parse(typeof(TelephoneType),parts[1],false);
```

solleva un'eccezione (false).

E' necessario pertanto utilizzare altri due metodi statici.

Serve, quindi un meccanismo che contemporaneamente ci consenta di controllare che una stringa sia "well-formed" e ci eviti di duplicare il codice.

La soluzione a questo problema sta nella creazione di due metodi che possiedono quasi tutti gli oggetti semplici del .NET: il metodo **TryParse** ed il metodo **Parse**.

Questi due metodi, provvedono all'analisi della stringa ed alla successiva creazione dell'oggetto e devono essere, ovviamente, dei metodi statici di classe.

Il metodo TryParse(), all'interno di un costrutto try..catch, "prova" a costruire un oggetto a partire da una stringa.

In caso di successo verrà restituito il valore booleano true e l'oggetto costruito utilizzando il parametro passato come out, altrimenti verrà restituito false e null.

```
public static bool TryParse(string str, out Telephone newtel)  
{  
    bool creata = false;  
    try  
    {  
        string[] parts = str.Split(',');  
        string numero = parts[0];  
        TelephoneType type = (TelephoneType)Enum.Parse(typeof(TelephoneType),  
parts[1], false);
```

```

        newtel = new Telephone(numero,type);
        creata = true;
    }
    catch (Exception)
    {
        newtel = null;
    }
    return creata;
}

```

Il metodo Parse(), utilizzando il metodo precedente solleva un'eccezione se la costruzione fallisce, restituendo invece l'oggetto creato se tutto va a buon fine.

```

public static Telephone Parse (string str)
{
    Telephone nuovoPhone = null;
    bool creata = Telephone.TryParse(str, out nuovoPhone);
    if (!creata)
        throw new FormatException("Formato Nave non corretto");
    return nuovoPhone;
}

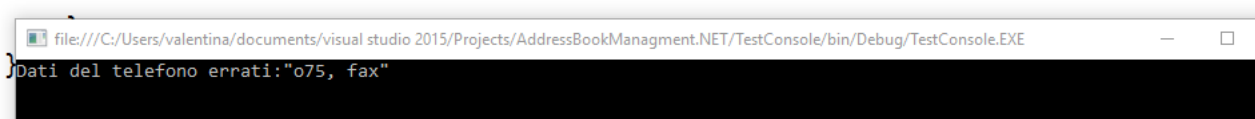
```

Testiamo in Console:

```

string stringa = "o75, fax";
Telephone tel2 = null;
bool ok = Telephone.TryParse(stringa, out tel2);
if (!ok)
    Console.WriteLine("Dati del telefono errati:\"{0}\"", stringa);
else
    Console.WriteLine("Dati del telefono: {0}", tel2.ToString());
Console.Read();

```



## Passaggio per riferimento degli argomenti: parametri ref e out.

Osserviamo che i parametri del metodo Tryparse hanno una nuova sintassi:

```

public static bool TryParse(string str, out Telephone newtel)

```

Al terzo anno abbiamo visto il passaggio per riferimento e per valore.

Ricordiamo che i parametri ref (reference, riferimento) consentono al metodo chiamante e al metodo chiamato di condividere la stessa variabile, pur facendo riferimento a due oggetti apparentemente distinti. Definire un parametro come ref fa sì che qualsiasi modifica effettuata su di esso si rifletta sull'argomento corrispondente. Un parametro ref rappresenta dunque un alias dell'argomento: è come se tutte le istruzioni che coinvolgono il parametro, in realtà coinvolgessero l'argomento.

Nel passaggio per valore invece argomento e parametro rappresentano a tutti gli effetti variabili.

La possibilità di utilizzare parametri e/o valore di ritorno estende notevolmente le potenzialità applicative dei metodi, consentendo l'utilizzo di un modello di comunicazione più sicuro, potente e generalizzabile. Tale modello, però, non copre completamente le esigenze della programmazione con i metodi, poiché presenta un limite importante: funziona in una sola direzione soltanto. Tramite i parametri d'ingresso il metodo chiamante passa dei valori al metodo chiamato. Tramite il valore di ritorno accade l'inverso. In alcuni scenari, serve un modello di comunicazione più potente, che possa funzionare in entrambe le direzioni distinte.

I parametri out (output, uscita) consentono al metodo chiamato di utilizzare i parametri come mezzo per comunicare dei risultati al metodo chiamante. I parametri out usano lo stesso modello di comunicazione dei parametri ref: tra argomenti e parametri avviene un passaggio per riferimento e dunque un parametro out rappresenta un alias dell'argomento out corrispondente.

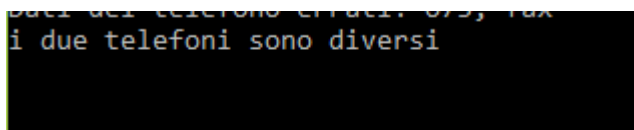
Nonostante parametri out e ref utilizzino lo stesso meccanismo di passaggio, rivestono un ruolo distinto nella comunicazione tra metodo chiamante e metodo chiamato. I parametri ref consentono di condividere una variabile tra i due metodi, i parametri out consentono al metodo chiamato di comunicare dei dati al metodo chiamante: nel secondo caso la comunicazione si svolge in una sola direzione. I parametri out, dunque, svolgono la funzione inversa dei parametri di ingresso.

## Il concetto di uguaglianza

Scriviamo in TestConsole il seguente codice:

```
Telephone n3 = new Telephone("333", TelephoneType.Fax);
Telephone n4 = new Telephone("333", TelephoneType.Fax);
if (n3 == n4)
    Console.WriteLine("i due telefoni sono uguali");
else
    Console.WriteLine("i due telefoni sono diversi");
```

Quale è il risultato atteso?



```
data del telefono errati: 0/3, fax
i due telefoni sono diversi
```

Per il .NET i due oggetti sono effettivamente diversi in quanto la verifica dell'uguaglianza non avviene sul contenuto ma sul puntatore. Il .NET non utilizza l'operatore di uguaglianza per confrontare due oggetti. Pertanto è necessario riscrivere il metodo Equals() con tutta la logica necessaria per stabilire se due oggetti sono uguali.

Il metodo Equals() riceve in ingresso un parametro di tipo Object e restituisce un valore booleano risultato del confronto l'oggetto corrente e quello passato.

Questo metodo, inoltre, non deve mai generare un'eccezione e deve sempre restituire false se l'argomento è un riferimento ad un oggetto null o è di un tipo diverso dal tipo corrente.

```
public override bool Equals(object obj)
{
    if (obj == null)
        return false;
    Telephone n = obj as Telephone;
    if (n == null)
        return false;
    return (Number == n.Number && Type == n.Type);
}
```

```
}
```

Una volta sovraccaricato il metodo Equals() è necessario anche ridefinire i metodi che seguono:

```
public override int GetHashCode()
{
    int hashCode = Number.GetHashCode() ^ Type.GetHashCode();
    return hashCode;
}
```

Affinché due oggetti siano considerati uguali è necessario che restituiscano anche lo stesso codice hash.

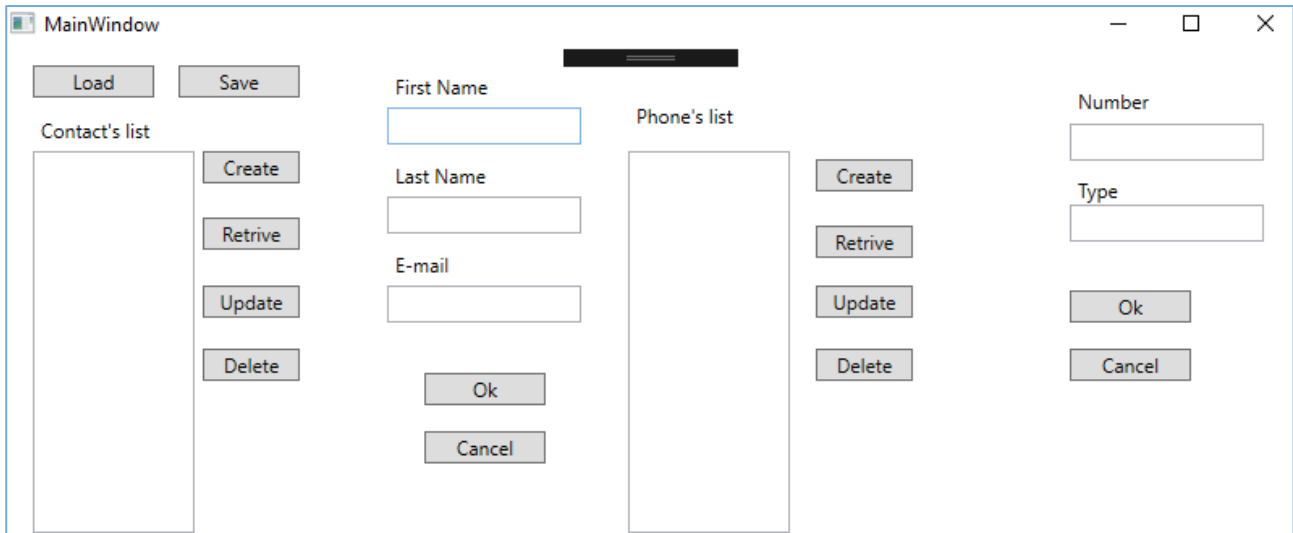
Il valore restituito del metodo GetHashCode non deve cambiare durante l'intero ciclo di vita dell'oggetto e per fare ciò, si può restituire un valore costante combinando i codici hash di due o più campi immutabili.

A questo punto è necessario ridefinire l'operatore di uguaglianza e di disuguaglianza.

```
public static bool operator ==(Telephone t1, Telephone t2)
{
    return t1.Equals(t2);
}
```

```
public static bool operator !=(Telephone t1, Telephone t2)
{
    return !(t1.Equals(t2));
}
```

## Model View e UI



Dall'osservazione della struttura dell'interfaccia analizziamo le variabili necessarie:

```
//queste sono le variabili che servono alla nostra interfaccia
private List<Contact> Contacts;
private Contact currentContact;
private List<Telephone> Telephones;
private Telephone currentTelephone;
```

In informatica il concetto di contenitore in cui depositare le informazioni si chiama Repository. Si tratta di una classe in cui si scrive e si leggono dati. Il metodo LoadContact() è pubblico e restituisce una lista di contatti

```
public class AddressBookRepository
{
```

```
    public List<Contact> LoadContact()
    {
        List<Contact> resp = new List<Contact>();
        Contact c1 = new Contact()
        { FirstName = "Stefano",
          LastName = "Del Furia",
          Email = "delfo@" };

        c1.CreatePhone("075", TelephoneType.Home);
        c1.Telephones.Add(new Telephone("333", TelephoneType.Mobile));
        resp.Add(c1);
    }
```

Vediamo i metodi:

```
private void btnLoad_Click(object sender, RoutedEventArgs e)
{
    //dichiaro un oggetti di tipo Repository
    AddressBookRepository repo = new AddressBookRepository();
```

```

        Contacts = repo.LoadContact();

        lboContacts.ItemsSource = Contacts;
        lboContacts.DisplayMemberPath = "FirstName";
    }

```

Se nella lista non si trova il contenuto vuol dire che si è dimenticato qualcosa. Ricordare di scrivere il ToString() nelle classi.

Nella classe Contact deve esserci il ToString()

```

public override string ToString()
{
    return string.Format("{0}-{1}", FirstName, LastName);
}

```

Vediamo il metodo retrieve:

```

private void btnRetrieveContact_Click(object sender, RoutedEventArgs e)
{
    //prendiamo la posizione del contatto di selezionato
    //select index indica l'elemento selezionato
    // Ricordarsi che si devono fare dei controlli sulla lista
    int pos = lboContacts.SelectedIndex;
    if (pos != -1)
    {
        currentContact = Contacts[pos];
        txtFirstName.Text = currentContact.FirstName;
        txtLastName.Text = currentContact.LastName;
        txtEmail.Text = currentContact.Email;

        // devo mostrarre anche la lista dei telefoni
        Telephones = currentContact.Telephones;
        lboTelephones.ItemsSource = Telephones;
        lboTelephones.DisplayMemberPath = "Number";
    }
    else
        MessageBox.Show("Errore");
}

```

```

private void btnRetrievePhone_Click(object sender, RoutedEventArgs e)
{
    //posto scelto
    int pos = lboTelephones.SelectedIndex;
    if (pos != -1)
    {
        currentTelephone = Telephones[pos];
        txtNumber.Text = currentTelephone.Number;
        //se nn metto il ToString ho un errore
        txtType.Text = currentTelephone.Type.ToString();
    }

    else
        MessageBox.Show("Errore");
}

```



## Model-View

Con questo metodo la finestra dei contatti si carica automaticamente. Capiremo presto che il metodo Load diventerà inutile.

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
    AddressBookRepository repo = new AddressBookRepository();
    contacts = repo.LoadContact();
    lboContacts.ItemsSource = contacts;
}
```

Ricordare di scrivere l'evento anche nello XAML

```
<Window x:Class="AddressApp.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:AddressApp"
        mc:Ignorable="d"
        Title="MainWindow" Height="323" Width="230" Loaded="Window_Loaded" >
```

Dalla lista visualizzata dovrò visualizzare i dati di un contatto. Pertanto devo creare un oggetto "currentContact" di tipo Contact

```
public partial class MainWindow : Window
{
    private List<Contact> contacts;

    private Contact currentContact;
```

```
private void btnRetrieveContact_Click(object sender, RoutedEventArgs e)
{
    //recupero la posizione del contatto selezionato
    int pos = lboContacts.SelectedIndex;
    //se effettivamente ho selezionato un contatto (la posizione è diversa da -1)
    if (pos != -1)
        //nell'oggetto currentContact assegno i dati contenuti nel contatto
    selezionato
    { currentContact = contacts[pos];
      //voglio cambiare schermata
```

```

        //per passare alla schermata ContactDetail creo l'oggetto
        //ma in ContactDeteil deve essere presente un costruttore che accetta come
parametro un contatto
        ContactDetail det=new ContactDetail(currentContact)
        det.ShowDialog();
    }
}

```

Pertanto andiamo in contactDetail

```

public partial class ContactDetail : Window
{
    // anche in questa classe ho bisogno dell'oggetto contatto e dell'oggetto telefono
    private Telephone _currentTelephone;
    private Contact _currentContact;

    //passiamo al costruttore il parametro di ingresso.
    public ContactDetail(Contact currentContact)
    {
        InitializeComponent();
        _currentContact = currentContact;
    }
}

```

Ora in MainWindow è tutto ok!

Si riesce a passare da una finestra all'altra! Ma ancora non visualizzo i dati del contatto.

Ripetiamo la procedura per creare l'evento Loaded

```

<Window x:Class="AddressApp.ContactDetail"
    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:AddressApp"
    mc:Ignorable="d"
    Title="ContactDetail" Height="400" Width="411" Loaded="Window_Loaded">

```

```

//ora occorre caricare i dati del contatto predentemente selezionato!
//andiamo a popolare textBox e listBox

private void Window_Loaded(object sender, RoutedEventArgs e)
{
    txtFirstName.Text = _currentContact.FirstName;
    txtLastName.Text = _currentContact.LastName;
    txtEmail.Text = _currentContact.Email;
    lboTelephones.ItemsSource = _currentContact.Telephones;
}

```

```
//ora quando si seleziona un numero di telefono devo poter vederne i dettagli nella una
//terza Window che
//abbiamo nominato PhoneDetail
private void btnRetrievePhone_Click(object sender, RoutedEventArgs e)
{
    //recupero la posizione del contatto selezionato
    int pos = lboTelephones.SelectedIndex;
    //se effettivamente ho selezionato un contatto (la posizione è diversa da -1)
    if (pos != -1)
    //nell'oggetto currentContact assegno i dati contenuti nel contatto selezionato
    {
        //prima recupero il telefono selezionato del contatto
        _currentTelephone = _currentContact.Telephones[pos];
        //posso cambiare finestra
        //e per passare alla schermata PhoneDetail creo l'oggetto
        //ma in PhoneDetail deve essere presente un costruttore che accetta come
parametro un telefono
        PhoneDetail det = new PhoneDetail(_currentTelephone);
        det.ShowDialog();
    }
}
```

Andiamo quindi in PhoneDetail

```
public partial class PhoneDetail : Window
{
    private Telephone _currentTelephone;
    public PhoneDetail(Telephone currentTelephone)
    {
        InitializeComponent();
        _currentTelephone = currentTelephone;
    }
}
```

A questo punto, quando si apre la terza finestra occorre ripopolare tutti i campi relativi al telefono selezionato.

Quindi in PhoneDetail occorre ricorrere nuovamente all'evento OnLoaded

```
<Window x:Class="AddressApp.PhoneDetail"
    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:AddressApp"
    mc:Ignorable="d"
    Title="PhoneDetail" Height="300" Width="182" Loaded="Window_Loaded">

    private void Window_Loaded(object sender, RoutedEventArgs e)
    {
        txtNumber.Text = _currentTelephone.Number;
    }
}
```

```
txtType.Text = _currentTelephone.Type.ToString();  
}
```

Ora procediamo con l'aggiornamneto.

Nella classe ContactDetail scriviamo il metodo Update

```
private void btnUpdateContact_Click(object sender, RoutedEventArgs e)  
{  
    _currentContact.FirstName = txtFirstName.Text;  
    _currentContact.LastName = txtLastName.Text;  
}
```

Se torniamo alla schermata principale però non si notano gli aggiornamenti.

Questo perché la listbox non è stata ripopolata.

Pertanto è necessario nel metodo btnRetrieve, cancellare il contenuto della listbox e ripopolarla di nuovo

Ecco quindi il metodo completo

```
private void btnRetrieveContact_Click(object sender, RoutedEventArgs e)  
{  
    //recupero la posizione del contatto selezionato  
    int pos = lboContacts.SelectedIndex;  
    //se effettivamente ho selezionato un contatto (la posizione è diversa da -1)  
    if (pos != -1)  
        //nell'oggetto currenctContat assegno i dati contenuti nel contatto  
        selezionato  
        { currentContact = contacts[pos];  
          //voglio cambiare schermata  
          //per passare alla schermata ContactDetail creo l'oggetto  
          //ma in ContactDeteil deve essere presente un costruttore che accetta come  
          parametro un contatto  
          ContactDetail det = new ContactDetail(currentContact);  
          det.ShowDialog();  
          lboContacts.ItemsSource = null;  
          lboContacts.ItemsSource = contacts;  
        }  
}
```

Analogo nel metodo Retrieve di ContactsDetails

```
private void btnRetrievePhone_Click(object sender, RoutedEventArgs e)  
{  
    //recupero la posizione del contatto selezionato  
    int pos = lboTelephones.SelectedIndex;  
    //se effettivamente ho selezionato un contatto (la posizione è diversa da -1)  
    if (pos != -1)  
        //nell'oggetto currenctContat assegno i dati contenuti nel contatto selezionato  
        {  
            //in questo modo recupero il telefono selezionato del contatto  
            _currentTelephone = _currentContact.Telephones[pos];  
            //voglio cambiare schermata  
        }
```

```

        //per passsare alla schermata PhoneDetail creo l'oggetto
        //ma in PhoneDetail deve essere presente un costruttore che accetta come
parametro un telefono
        PhoneDetail det = new PhoneDetail(_currentTelephone);
        det.ShowDialog();

        lboTelephones.ItemsSource = null;
        lboTelephones.ItemsSource = _currentContact.Telephones;
    }

```

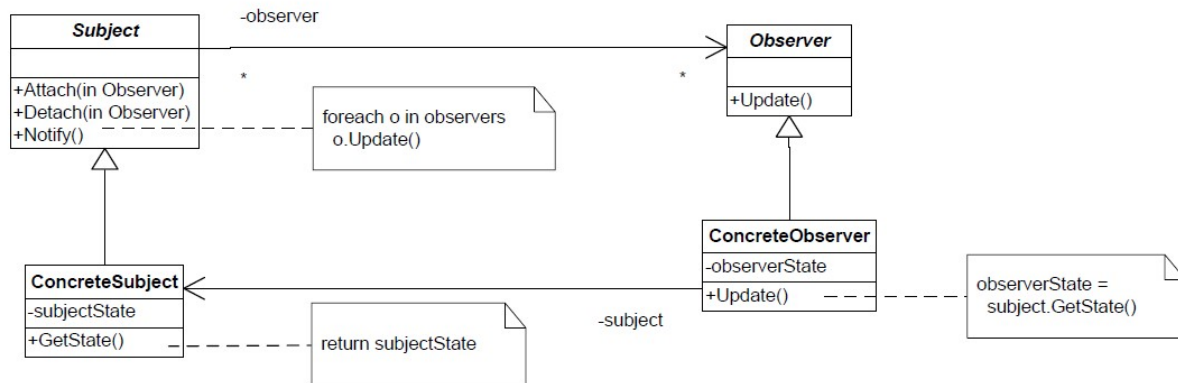
```

private void btnUpdatePhone_Click(object sender, RoutedEventArgs e)
{
    _currentTelephone.Number = txtNumber.Text;
    _currentTelephone.Type = (TelephoneType)Enum.Parse(typeof(TelephoneType),
txtType.Text);
}

```

## Observer Pattern

Definisce una dipendenza uno-a-molti tra gli oggetti in maniera tale che quando un oggetto cambia di stato tutte le sue dipendenze sono avvisate ed aggiornate automaticamente



Nella prima implementazione abbiamo ignorato la relazione di generalizzazione tra le due classi implementando il progetto considerando un'unica classe.

```

public class Centrale
{
    public Centrale()
    {
        Navi = new List<INave>();
    }
    private List<INave> Navi;

    public void Attach(INave nave)
    {
        Navi.Add(nave);
    }
    public void Detach(INave
nave)
    {
        Navi.Remove(nave);
    }

    public void Notify()
    {
        for (int i = 0; i < Navi.Count; i++)
        {
            Navi[i].Update(Message);
        }
    }

    private string _message;

    public string Message
    {
        get { return _message; }
set
        {
            _message = value;
            Notify();
        }
    }
}

```

Andiamo a questo punto ad approfondire il concetto di generalizzazione.

La relazione di generalizzazione rappresenta uno degli elementi fondamentali della programmazione ad oggetti: l'ereditarietà.

Caratteristica degli oggetti reali di un problema è la “somiglianza”: gli oggetti possono condividere alcune caratteristiche generali e differenziarsi per altre.

L'alunno provi a scrivere le somiglianze tra i mezzi di trasporto!

Un'automobile + un tipo di mezzo di trasporto, ma lo è anche un motociclo, una bicicletta, un tir, un furgone...

La relazione “un tipo di” rappresenta la relazione di ereditarietà. Stabilisce che un determinato soggetto eredita le caratteristiche di un secondo soggetto. Riprendiamo l'analisi del pattern.

<pre> classDiagram     class Subject {         +Attach(in Observer)         +Detach(in Observer)         +Notify()     }     class ConcreteSubject {         -subjectState         +GetState()     }     Subject &lt; -- ConcreteSubject </pre>	<p>ConcreteSubject è un tipo di Subject. La centrale operativa è un tipo di centrale! Le classi e/o gli oggetti che partecipano a questo pattern sono:</p> <ul style="list-style-type: none"> <li>• Subject (<b>Centrale</b>) <ol style="list-style-type: none"> <li>1. Conosce gli “osservatori”. Qualunque numero di oggetti Observer può osservare un Subject</li> <li>2. fornisce dei metodi per attaccare o staccare un oggetto Observer</li> </ol> </li> <li>• ConcreteSubject (<b>CentraleOperativa</b>) <ol style="list-style-type: none"> <li>1. eredita tutte le funzionalità di Subject</li> <li>2. Memorizza lo stato che interessa al ConcreteObserver 3. invia una notifica ai suoi Observers quando questo stato cambia (GetState)</li> </ol> </li> </ul>
<pre> classDiagram     class Observer {         +Update()     }     class ConcreteObserver {         -observerState         +Update()     }     Observer &lt; -- ConcreteObserver     note for ConcreteObserver "observerState = subject.GetState()" </pre>	<p>Anche tra Observer e ConcreteObserver vi è una relazione di generalizzazione.</p> <p>Observer (<b>INave</b>)</p> <ol style="list-style-type: none"> <li>1. definisce una <b>interfaccia</b> di aggiornamento per oggetti che devono essere notificati del cambio in un Subject.</li> </ol> <p>ConcreteObserver (<b>Nave</b>)</p> <ol style="list-style-type: none"> <li>2. mantiene un riferimento ad un oggetto di tipo ConcreteSubject</li> <li>1. memorizza uno stato che deve essere consistente con quello dei Subject</li> <li>2. implementa l’Observer (osservare il commento che indica come implementare il metodo Update)</li> </ol>

Già sappiamo scrivere la seconda relazione di generalizzazione (relazione tra interfaccia e classe).

```

public interface INave
{
    void Update(string msg, string NomeCentrale);
}

```

```

public class Nave : INave
{
    public Nave(string name)
    {
        _name = name;
    }
    private string _name;
}

```



```

    public void Update(string msg, string nomeCentrale)
    {
        Console.WriteLine("Ricevuto:{0} da {1}", msg, nomeCentrale);
    }
}

```

E' possibile ereditare una classe da un'altra specificando il nome della seconda nell'intestazione della prima dopo il simbolo due-punti.

```

public class ClasseDerivata : ClasseBase
{
    }

```

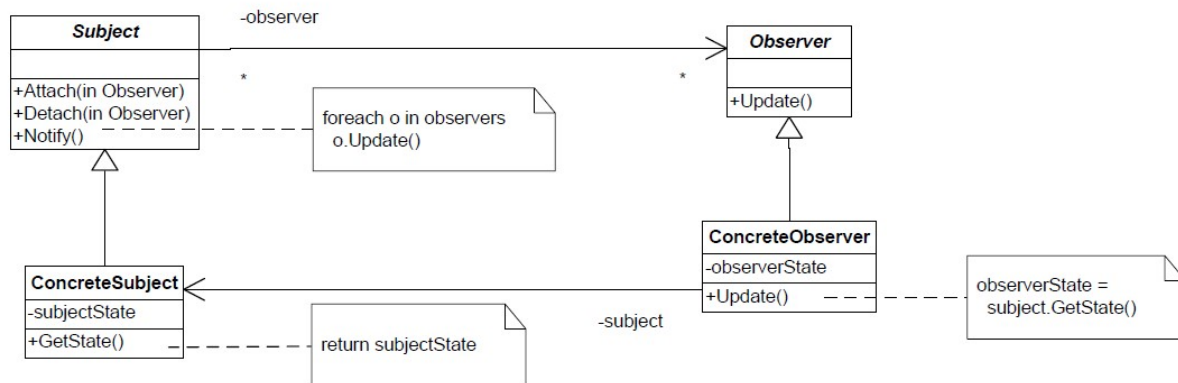
La classe base viene specificata in quello che si chiama "lista base" o "lista di derivazione". Si usa la parola lista in quanto, come abbiamo già visto, oltre al nome della classe possono comparire uno più nomi di interfacce.

```

public class Nave : INave

```

Completiamo dunque il codice del pattern implementando l'ereditarietà tra la classe Centrale e CentraleOperativa.



Osservando l'UML viene inserita nella classe CentraleOperativa cancellandola da Centrale.

```

public class CentraleOperativa : Centrale
{
    public CentraleOperativa(string nome) : base()
    {
        Nome = nome;
    }

    private string _message;

    public string Message
    {
        get { return _message; }
set
        {
            _message = value;
            Notify(Message);
        }
    }
}

```

In centrale quindi modificheremo solo il metodo Notify passandogli il parametro message.

```

public class Centrale
{
    public string Nome { get; set; }

    public Centrale()
    {
        Navi = new List<INave>();
    }

    protected List<INave> Navi ;

    public void Attach(INave nave)
    {
        Navi.Add(nave);
    }

    public void Detach(INave nave)
    {
        Navi.Remove(nave);
    }

    public void Notify(string
Message)
    {
        for (int i = 0; i < Navi.Count; i++)
        {
            Navi[i].Update(Message, Nome);
        }
    }
}

```

Andiamo adesso ad analizzare il costruttore della CentraleOperativa. Osserviamo una nuova parola chiave

```
public CentraleOperativa(string nome) : base()
{
    Nome = nome;
}
```

I costruttori della classe base non vengono ereditati dalle classi che derivano da esse. Pertanto nella classe ereditata vanno comunque ridefiniti costruttori propri. Per far riferimento ad un costruttore già scritto nella classe base va utilizzata la parola chiave base.

In questo modo posso inizializzare i membri ereditati (nel nostro caso la lista Navi) e definire, se esistono nuovi campi membro (nome della centrale operativa).

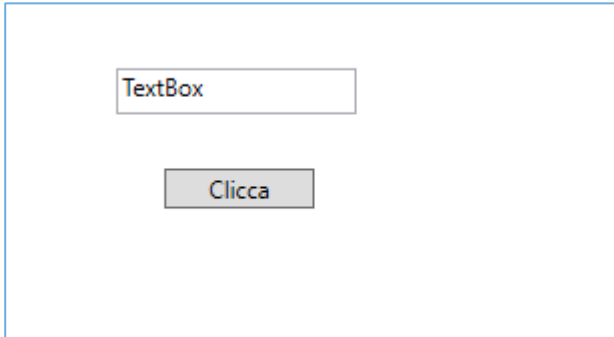
Scorrendo il codice della classe Centrale troviamo una seconda parola chiave protected

```
protected List<INave> Navi ;
```

Gli attributi di una classe sono di norma privati. Ma se restano tali la classe ereditata non può accedervi. Ma non possiamo nemmeno renderli pubblici in quanto si violerebbe il principio dell'information hiding. Per questo motivo esiste un altro livello di protezione per gli attributi, che li mantiene inaccessibili al codice consumer, ma accessibili alle classi derivate "protected".

## Simple Binding

Consideriamo la seguente semplice applicazione.



Fino ad oggi abbiamo agito nel seguente modo: si scrive nella TextBox, si clicca sul Bottone ed il valore viene visualizzato nel contenuto della label.

Il tutto avviene grazie alle proprietà scritte all'interno della classe su cui si vuol agire. Nel nostro caso consideriamo la semplice classe Persona.

```
public class Persona
{
    public string FirstName{get;set;}
}
```

Nella parte relativa all'interfaccia si procede con lo scrivere il seguente codice:

```
public partial class MainWindow : Window
{
    Persona p;
    public MainWindow()
    {
        InitializeComponent();
        p = new Persona();
    }

    private void btnClick_Click(object sender, RoutedEventArgs e)
    {
        p.FirstName = txtName.Text;
        lblname.Content = p.FirstName;
    }
}
```

Vediamo come cambia il codice con il sistema di notifiche.

Intanto è necessario implementare le notifiche all'interno della classe.

```
namespace LibraryPersonaVale
{
    public class Persona: INotifyPropertyChanged
    {
        public string
    }
}
```

```
public class Persona: INotifyPropertyChanged
{
    public string
}
```

L'interfaccia implementata porterà ad avere il seguente codice:

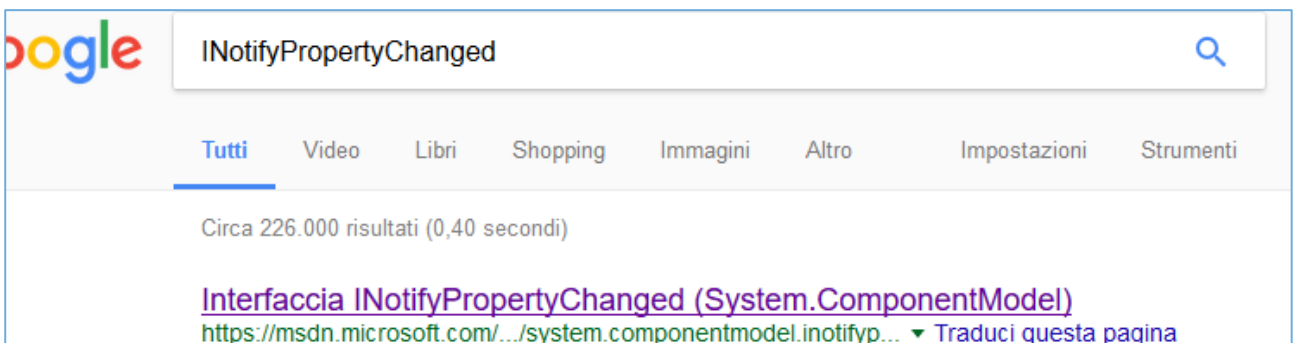
```
public class Persona: INotifyPropertyChanged
{
    public string FirstName{get;set;}

    public event PropertyChangedEventHandler PropertyChanged;
}
```

A questo punto, per implementare il sistema di notifiche è necessario un metodo Notify (ricordare il pattern Observer) e la modifica delle proprietà.

Pertanto la proprietà automatica va riscritta!

Il codice per implementare le notifiche si trova facilmente anche su internet, non è necessario ricordarselo a memoria!



```
string _firstName;
public string FirstName
{
    get { return _firstName; }
    set
    {
        _firstName = value;
        OnPropertyChanged("FirstName");
    }
}
```

```

//public void Notify(string propertyName)
public void OnPropertyChanged(string propertyName)
{
    if (PropertyChanged != null)
    {
        PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
    }
}

```

A questo punto vediamo cosa cambia nel codice relativo all'interfaccia!

```

Persona p;
public MainWindow()
{
    InitializeComponent();
    //non abbiamo più un oggetto Persona, ma un contesto di dati,
    //cioè tutti quei dati che servono alla finestra
    p = new Persona();
    p.FirstName = "Stefano";
    DataContext = p;
}

```

Eliminiamo il bottone e il codice relativo al bottone e effettuiamo la terza modifica : il binding. Tale modifica avviene nello xaml.

Per collegare FirstName al contenuto della TextBox effettuiamo la seguente modifica:

```

<Grid>
    <TextBox x:Name="txtName" Text="{Binding FirstName}" />
    <Label x:Name="lblname" Content="{Binding FirstName}" />

```

Ora torniamo alla nostra addressBook!

Occorre effettuare all'interno delle classi tutte le modifiche effettuate nell'esempio della classe persona.

Per quanto riguarda le collezioni esiste un meccanismo automatico che implementa le modifiche ObservableCollection.

Ecco alcune modifiche per la classe Telephone:

```

public class Contact : IContact, INotifyPropertyChanged
{
    private string _firstName;

    public string FirstName
    {
        get { return _firstName; }
        set
        {
            _firstName = value;
            OnPropertyChanged("FirstName");
        }
    }
}

```

```

    }

    private string _lastName;
    public string LastName
    {
        get { return _lastName; }
        set
        {
            _lastName = value;
            OnPropertyChanged("LastName");
        }
    }

public ObservableCollection<Telephone> Telephones { get; set; }
public Contact()
{
    Telephones = new ObservableCollection<Telephone>();
}

    private byte[] _photo;

    public byte[] Photo
    {
        get { return _photo; }
        set
        {
            _photo = value;
            OnPropertyChanged("Photo");
        }
    }

public void OnPropertyChanged(string propertyName)
{
    if (PropertyChanged != null)
    {
        PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
    }
}

```

## I Formati

### CSV:

Separo i dati con la virgola.

Valentina, Falucca, Panicale

Stefano, Del Furia, Città di Castello

Il separatore non è detto che sia la virgola. Potrei utilizzare anche lo spazio come separatore, ma non saprei distinguere dove inizia e dove finisce il dato. Es. Città di castello

Il separatore si sceglie in base al contesto di utilizzo. Generalmente la virgola è la più utilizzata.

Problema: devo sapere come è fatto il file di testo. Devo sapere l'ordine (es. nome, cognome, indirizzo). Se inverte un'informazione non ho modo di comunicare la variazione.

### XML:

Oltre al dato fornisce anche la descrizione del dato.

Posso stabilire una gerarchia del file XML.

```
<utenti>
  <utente anni="16">
    <nome>Luca</nome>
    <cognome>Cicci</cognome>
    <indirizzo>Milano</indirizzo>
  </utente>
  <utente anni="54">
    <nome>Max</nome>
    <cognome>Rossi</cognome>
    <indirizzo>Roma</indirizzo>
  </utente>
</utenti>
```

### JSON:

Mentre XML vuole il tag di apertura e di chiusura, cioè è più verboso.

Nel formato JSON la fine di un'informazione coincide con l'inizio di quella di un'altra.

E' una notazione troppo compatta, ottima per la comunicazione tra macchine. La lettura del formato è più difficoltoso per l'essere umano. Infatti Visual Studio legge in XML.

```
{
  "type": "menu",
  "value": "File",
}
```



```

        "items": [
            {"value": "New", "action": "CreateNewDoc"},
            {"value": "Open", "action": "OpenDoc"},
            {"value": "Close", "action": "CloseDoc"}
        ]
    }
}

```

Applicazione su Address Book

Parse e TryParse: meccanismo con cui a partire da una stringa recupero i dati.

Ogni classe DEVE avere questi metodi.

Ad esempio, nella classe Telefono.

```

public static bool TryParse(string str, out Telephone newPhone)
{
    bool creata = false;
    try
    {
        string[] parts = str.Split(',');
        string number = parts[0];
        TelephoneType t = (TelephoneType) Enum.Parse(typeof(TelephoneType),
parts[1], false);
        newPhone = new Telephone(number, t);
        creata = true;
    }
    catch (Exception)
    {
        newPhone = null;
    }
    return creata;
}

public static Telephone Parse(string str)
{
    Telephone newPhone = null;
    bool created = Telephone.TryParse(str, out newPhone);
    if (!created)
        throw new FormatException("Invalid Telephone format");
    return newPhone;
}
}

```

To String: Stabilisce il formato

```

public override string ToString()
{
    string tmp = "";
    tmp = String.Format("{0},{1}", Number, Type);
    return tmp;
}

```

E nella classe Contact

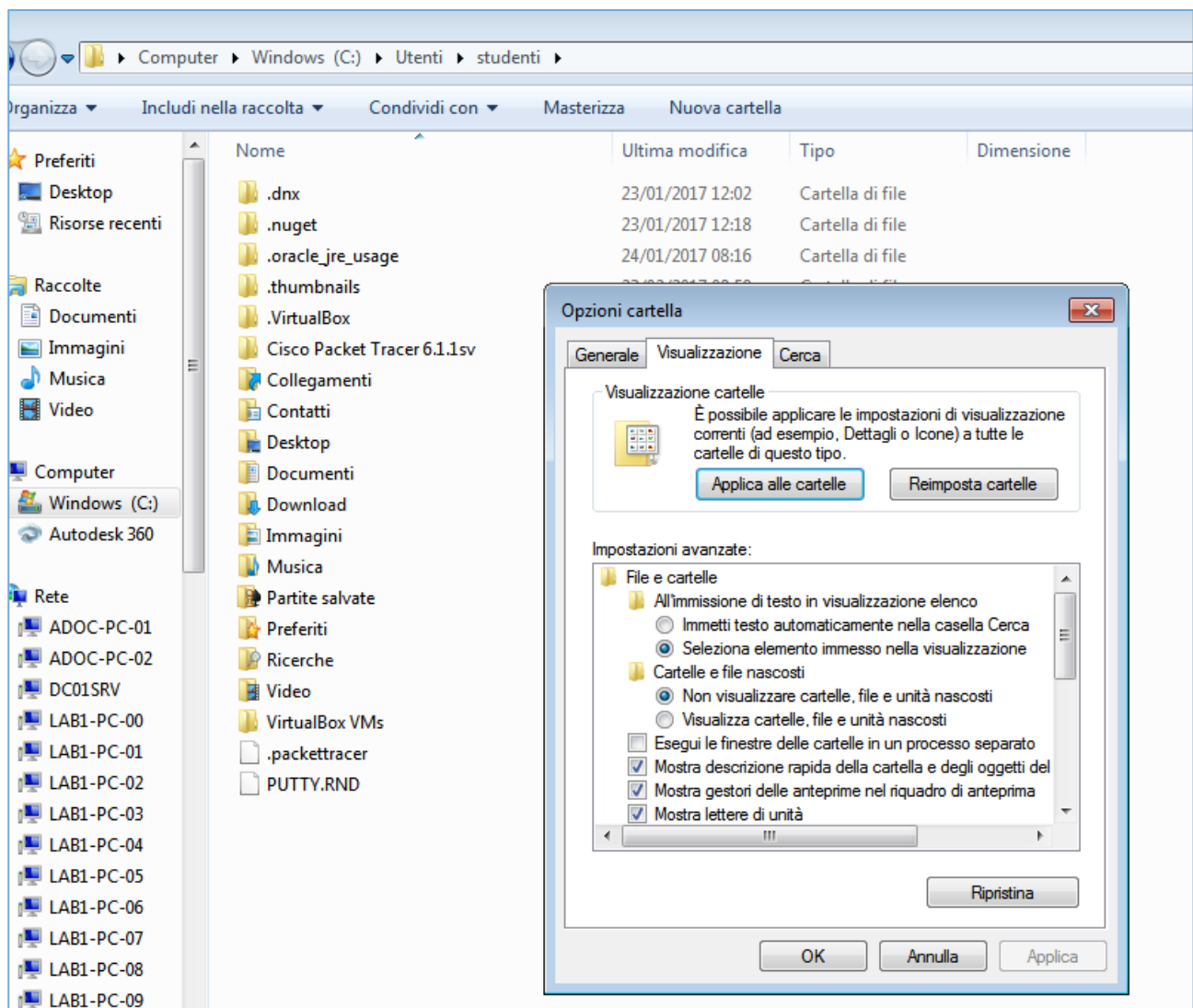
```
public override string ToString()
{
    string resp = string.Format("{0},{1},{2}", FirstName, LastName, Email);
    return resp;
}

public static bool TryParse(string str, out Contact newContact)
{
    bool creata = false;
    try
    {
        string[] parti = str.Split(',');
        string nome = parti[0];
        string cognome = parti[1];
        string mail = parti[2];
        newContact = new Contact(nome, cognome, mail);
        creata = true;
    }
    catch (Exception)
    {
        newContact = null;
    }
    return creata;
}

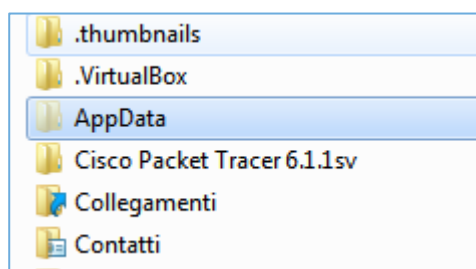
public static Contact Parse(string str)
{
    Contact newContact = null;
    bool creata = Contact.TryParse(str, out newContact);
    if (!creata)
        throw new FormatException("Formato Contact non corretto");
    return newContact;
}
```

Quando si caricano i dati da dove prendono le informazioni quindi?

Ogni sistema operativo ha un posto predefinito dove mettere i dati (zona della memoria interna del dispositivo utilizzata per i dati dell'utente e le informazioni degli applicativi usati dall'utente.)



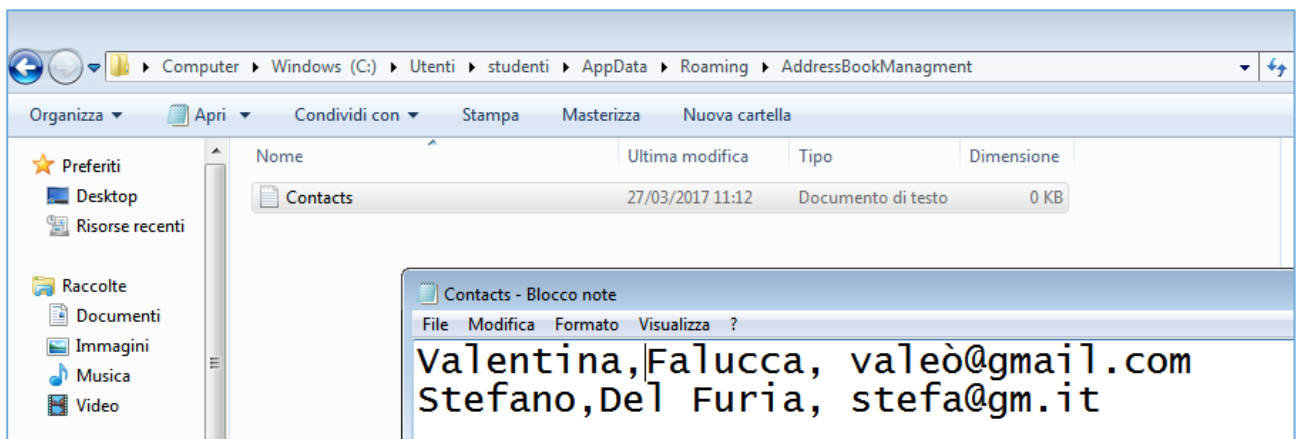
Cliccando su "Visualizza cartelle, file ed unità nascoste" avremo la cartella AppData



Accedo a Roaming

Nome	Ultima
Local	27/03/
LocalLow	23/03/
Roaming	24/03/

E creo la cartella AddressBookManagment con al suo interno un file .txt, ad esempio "Contacts.txt"



**Compito:** E se la cartella non è presente?

Adesso vediamo come assegnare il percorso al nostro progetto AddressBookManagment.

```
public class AddressBookRepository
{
    private string _path;
    public AddressBookRepository()
    {
        //_path = Environment.GetFolderPath(Environment.SpecialFolder.ApplicationData);
        //_path = _path + "AddressBookManagement";
        //_path = _path + "Contacts.txt";

        _path =
Path.Combine(Environment.GetFolderPath(Environment.SpecialFolder.ApplicationData),
                "AddressBookManagement",
                "Contacts.txt");
    }

    public void SaveContacts(List<Contact> contacts)
    {
        StreamWriter sw = new StreamWriter(_path);
        for (int i = 0; i < contacts.Count; i++)
        {
            sw.WriteLine(contacts[i].ToString());
        }
        sw.Close();
    }
    public List<Contact> LoadContact()
    {
        List<Contact> resp = new List<Contact>();
        StreamReader sr = new StreamReader(_path);
        string linea = sr.ReadLine();
        while (linea != null)
        {
            resp.Add(Contact.Parse(linea));
            linea = sr.ReadLine();
        }
        sr.Close();

        return resp;
    }
}
```

## Model View View Model

Riprendiamo il progetto calcolatrice.

Attualmente il progetto ha:

**Model** cioè la classe, l'elaborazione.

Rappresenta il dominio del problema

```
public class Calcolatrice
{
    private double _num1;
    public double Num1
    {
        get { return _num1; }
        set { _num1 = value; }
    }
}
```

//altro codice

Tutte le classi devono implementare  
INotifyPropertyChanged

**View**, cioè la User Interface

```
public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
    }

    private void btnSomma_Click(object sender, RoutedEventArgs e)
    {
        Calcolatrice calc = new
        Calcolatrice(txtNumero1.Text,
        txtNumero2.Text);
        calc.Somma();
        if (!calc.IsError)
            lblRisultato.Content =
        calc.Valore;
        else
        MessageBox.Show(calc.Messaggio, "Errore",
        MessageBoxButton.OK, MessageBoxImage.Error);
    }
}
```



Introduciamo un intermediario tra la vista e il modello



ViewModel: è specifico per la view, contiene solo di dati e metodi che servono all'interfaccia.

E' responsabile per la gestione della logica della vista. In genere, il *view model* interagisce con il modello invocando metodi nelle classi del modello. Il *view model* fornisce quindi i dati dal modello in una forma che la vista può usare facilmente. Rende così la vista completamente INDIPENDENTE dall'elaborazione.

Il punto di contatto è ViewModel.

Modifichiamo la classe calcolatrice. Ogni classe deve implementare INotifyPropertyChanged

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

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

Ogni classe deve avere una modifica nelle sue proprietà.

```
private double _num1;
public double Num1
{
    get { return _num1; }
    set { _num1 = value;
        OnPropertyChanged("Num1");
    }
}

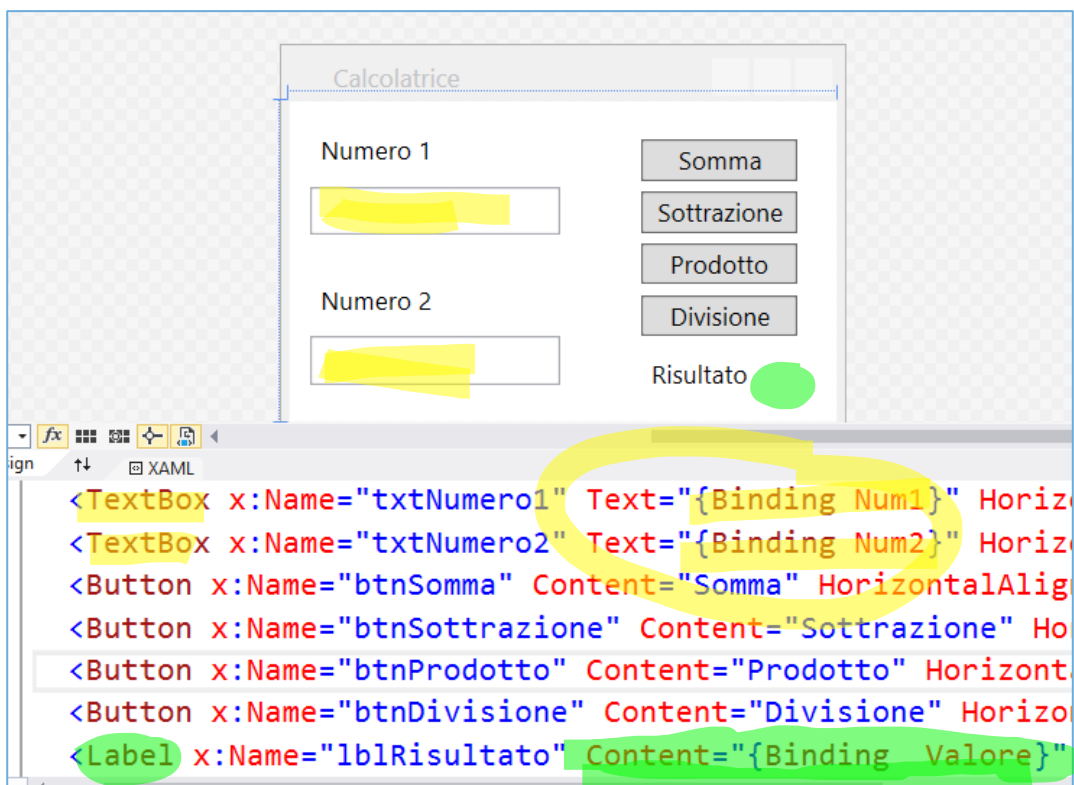
private double _num2;
public double Num2
{
    get { return _num2; }
    set { _num2 = value;
        OnPropertyChanged("Num2");
    }
}

private double _valore;
public double Valore
{
    get { return _valore; }
    set { _valore = value;
        OnPropertyChanged("Valore");
    }
}
```

Passiamo alla View. Creiamo in contesto di dati

```
public partial class MainWindow : Window
{
    Calcolatrice calc;
    public MainWindow()
    {
        InitializeComponent();
        calc = new Calcolatrice();
        DataContext = calc;
    }
}
```

E passiamo ad effettuare il binding nello XAML alle due TextBox e alla label che visualizza il risultato.



Ci manca da collegare i metodi degli oggetti, cioè collegare i click dei comandi.

Inseriamo il ModelView che farà da intermediario tra la View ed il Model ed al cui interno verranno trasferiti i comandi. I comandi sono collegati alle azioni della UI.

Ogni schermata ragiona sui suoi dati. Per il principio della responsabilità singola.

Aggiungiamo un nuovo progetto: Add-NewProject-ClassLibrary.

Nel progetto WPF dovrò aggiungere la reference

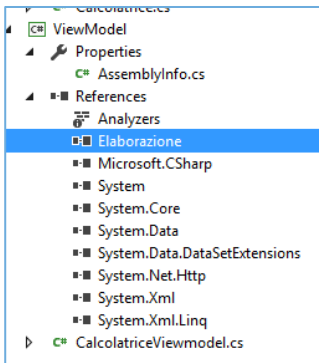
```
using System.Windows.Shapes;
using Elaborazione;
using ViewModel;
```

Arriveremo ad eliminare nel progetto WPF la reference ad Elaborazione!

Infatti la ViewModel avrà il riferimento all'elaborazione

```
using System.Threading.Tasks;
using Elaborazione;
```





Ad un'azione corrisponde un comando.

Per collegare i metodi all'interfaccia grafica abbiamo bisogno di ICommand.

L'interfaccia ICommand prevede un gestore degli eventi e possiede due metodi:

CanExecute: Decide se il comando può essere eseguito

Execute: Definisce il metodo da chiamare quando il comando è invocato

Quindi per ogni comando presente nell'interfaccia andiamo a creare una classe che implementerà ICommand.

Vediamo ad esempio per la Somma

```
{
    public class ComandoSomma : ICommand
    {
        private CalcolatriceViewModel _calcolatriceViewModel;

        public ComandoSomma(CalcolatriceViewModel calcolatriceViewModel)
        {
            _calcolatriceViewModel = calcolatriceViewModel;
        }
        public event EventHandler CanExecuteChanged;

        public bool CanExecute(object parameter)
        {
            return true;
        }
        public void Execute(object parameter)
        {
            _calcolatriceViewModel.Somma();
        }
    }
}
```

Vediamo come usare l'ereditarietà.

Creiamo una classe padre comando

```
public class Comando : ICommand
{
    public event EventHandler CanExecuteChanged;

    protected CalcolatriceViewmodel _calcolatriceViewModel;
    public Comando(CalcolatriceViewmodel calcolatriceViewModel)
    {
        _calcolatriceViewModel = calcolatriceViewModel;
    }

    public virtual bool CanExecute(object parameter)
    {
        throw new NotImplementedException();
    }

    public virtual void Execute(object parameter)
    {
        throw new NotImplementedException();
    }
}
```

Specializziamo per ogni comando.

```
namespace ViewModel
{
    public class ComandoSomma : Comando
    {
        public ComandoSomma(CalcolatriceViewmodel calcolatriceViewModel):base(calcolatriceViewModel)
        {
        }

        public override bool CanExecute(object parameter)
        {
            return true;
        }
        public override void Execute(object parameter)
        {
            _calcolatriceViewModel.Somma();
        }
    }
}
```

Procediamo con i comandi della calcolatrice.

Il comando Sottrazione è un tipo di Comando.

Pertanto:

```

public class ComandoSottrazione:Comando
{

    public ComandoSottrazione(CalcolatriceViewmodel
calcolatriceViewModel):base(calcolatriceViewModel)
    {

    }

    public override bool CanExecute(object parameter)
    {
        return true;
    }
    public override void Execute(object parameter)
    {
        _calcolatriceViewModel.Sottrazione();
    }
}

```

La classe ViewModel avrà il metodo Somma e Differenza che prima erano presenti nella View.

Avrà un riferimento alla classe Calcolatrice e ad ICommand.

```

public class CalcolatriceViewmodel
{

    public Calcolatrice Calcolatrice { get; set; }
    public CalcolatriceViewmodel()
    { Calcolatrice = new Calcolatrice(); }

    public void Somma()
    {
        Calcolatrice.Somma();
    }

    public void Sottrazione()
    { Calcolatrice.Sottrazione(); }

    public ICommand ComandoSomma
    {
        get { return new ComandoSomma(this); }
    }

    public ICommand ComandoSottrazione
    {
        get { return new ComandoSottrazione(this); }
    }
}

```

Tornando alla View ecco come si riduce il codice!

```

public partial class MainWindow : Window
{
    private CalcolatriceViewmodel calcolatrice;
    public MainWindow()
    {

```

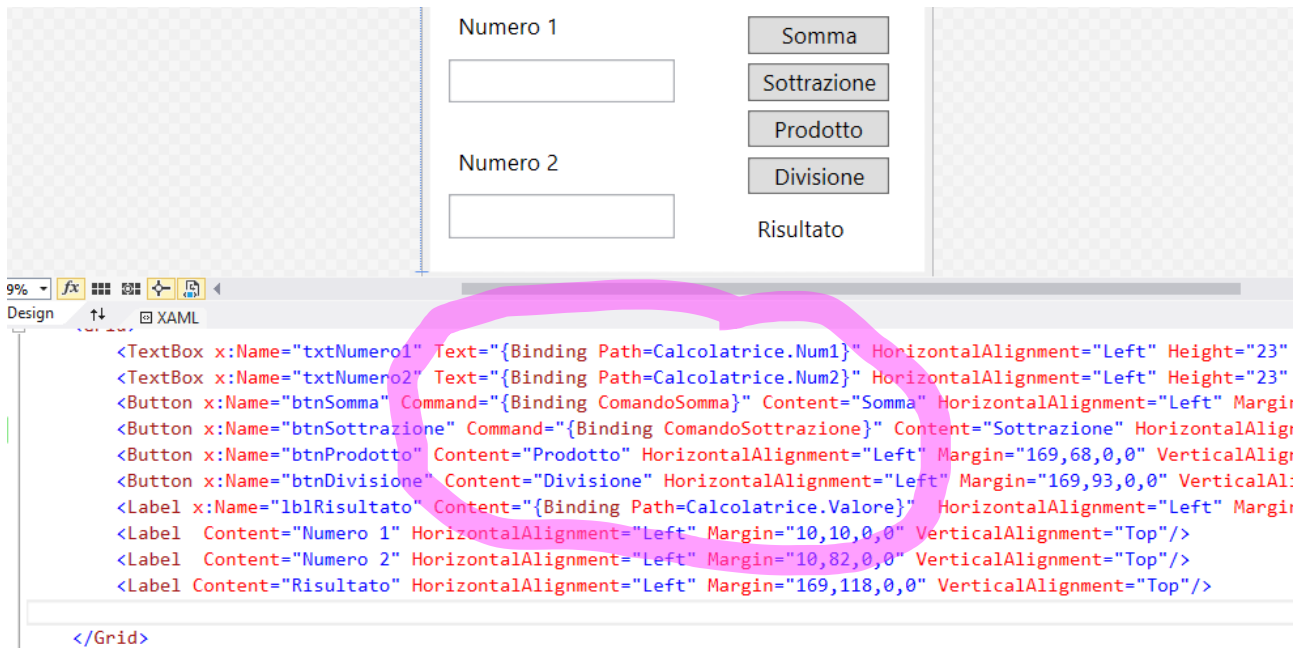
```

InitializeComponent();
calcolatrice = new CalcolatriceViewmodel();

DataContext = calcolatrice;
}

```

Ed infine vediamo come si modifica il codice XAML



## Creazione di un file XML

```
<?xml version="1.0" encoding="utf-8" ?>
<AddressBook>
  <Contact gender="Male">
    <FirstName>Stefano</FirstName>
    <LastName>Del Furia</LastName>
    <Email>delfo@</Email>
    <Address>p. matteotti</Address>
    <Telephones>
      <Telephone type="Home">
        <Number>0756789</Number>
      </Telephone>
      <Telephone type="Mobile">
        <Number>3345678</Number>
      </Telephone>
    </Telephones>
  </Contact>
  <Contact>
  </Contact>
</AddressBook>
```

Se AddressBook avesse avuto una sua variabile si aggiungeva il tag <Contacts> i cui figli sono i singoli contatti <Contact>.

Se il dato è fondamentale si crea l'elemento. Se il dato è accessorio si usa l'attributo.

### Leggere da XML

A questo punto è necessario leggere il file XML.

Ritorniamo alla nostra AddressBookRepository.

Nella classe XmlDocument esiste un metodo Load che carica un file XML. Per prima cosa è necessario dire dove andare a leggere.

Si consiglia di salvare il file XML nella cartella temp.

```
string path = @"c:\temp\AddressBook.xml";
```

A questo punto è possibile invocare il metodo Load che restituisce un oggetto document.

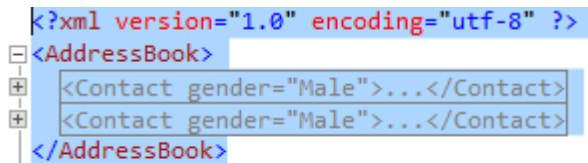
```
XmlDocument xmlDoc = XmlDocument.Load(path);
```

In realtà il file XML è stato letto tutto, ma va visto con un insieme di parti:

Una parte è AddressBook:

```
1  <?xml version="1.0" encoding="utf-8" ?>
2  <AddressBook>...</AddressBook>
31
```

Un'altra parte è Contact, un'altra ancora Telephone.



```
<?xml version="1.0" encoding="utf-8" ?>
<AddressBook>
  <Contact gender="Male">...</Contact>
  <Contact gender="Male">...</Contact>
</AddressBook>
```

In una prima fase è necessario selezionare l'elemento: devo scegliere l'elemento del document XML

```
XElement xmlAddressBook = xmlDoc.Element("AddressBook");
```

Successivamente è necessario selezionare l'elemento di AddressBook per poi leggerlo.

```
var xmlContacts = xmlAddressBook.Elements("Contact");
```

Si osservi che la variabile xmlContacts è di tipo var, non esplicitata. Sarà il compilatore a determinarne il tipo XElement.

Con questa istruzione ho preso in considerazione tutti i contatti tutti; ma è necessario leggerne uno per volta. E quindi è necessario un ciclo. Si leggono gli elementi e gli attributi uno ad uno e successivamente si crea l'oggetto ct.

Attenzione all'enumeratore perché è necessario utilizzare il metodo Parse.

```
foreach (var xmlContact in xmlContacts)
{
    XElement xmlFirstName = xmlContact.Element("FirstName");
    XElement xmlLastName = xmlContact.Element("LastName");
    XAttribute attrGender = xmlContact.Attribute("gender");

    Contact ct = new Contact();
    ct.FirstName = xmlFirstName.Value;
    ct.LastName = xmlLastName.Value;
    ct.Gender = (GenderType)Enum.Parse(typeof(GenderType), attrGender.Value);

    resp.Add(ct);
}
```

Il codice completo del metodo LoadContact() che restituisce una lista di contatti.

```
public class AddressBookRepository
{
    public List<Contact> LoadContact()
    {
        List<Contact> resp = new List<Contact>();

        string path = @"c:\temp\AddressBook.xml";
        XDocument xmlDoc = XDocument.Load(path);

        XElement xmlAddressBook = xmlDoc.Element("AddressBook");

        var xmlContacts = xmlAddressBook.Elements("Contact");

        //XElement xmlContact = xmlAddressBook.Element("Contact");

        foreach (var xmlContact in xmlContacts)
        {
            XElement xmlFirstName = xmlContact.Element("FirstName");
            XElement xmlLastName = xmlContact.Element("LastName");
            XAttribute attrGender = xmlContact.Attribute("gender");
            XElement xmlTelephones = xmlContact.Element("Telephones");
```

```

        XElement xmlTelephone = xmlTelephones.Element("Telephone");

        Contact ct = new Contact();
        ct.FirstName = xmlFirstName.Value;
        ct.LastName = xmlLastName.Value;
        ct.Gender = (GenderType)Enum.Parse(typeof(GenderType), attrGender.Value);

        resp.Add(ct);
    }

    return resp;
}

```

## Lettura da XML utilizzando le classi

Rivediamo come migliorare il codice di LoadContact utilizzando le classi dominio del problema.

All'interno delle classi inseriamo il metodo FromXML.

Partiamo dalla classe Telefono

```

public static Telephone FromXML(XElement xml)
{
    XElement xmlNumber = xml.Element("Number");
    XAttribute attrType = xml.Attribute("type");
    TelephoneType type = (TelephoneType)Enum.Parse(typeof(TelephoneType),
attrType.Value);
    string number = xmlNumber.Value;
    Telephone nuovoTelefono = new Telephone(number, type);
    return nuovoTelefono;
}

```

Successivamente si procede con la classe Contact.

```

public static Contact FromXML(XElement xml)
{
    XElement xmlFirstName = xml.Element("FirstName");
    string firstName = xmlFirstName.Value;
    XElement xmlLastName = xml.Element("LastName");
    string lastName = xmlLastName.Value;
    // leggo tutti gli altri elementi
    var xmlTelephones = xml.Elements("Telephone");
    List<Telephone> telephones = new List<Telephone>();
    foreach (var xmlTelephone in xmlTelephones)
    {
        Telephone tel = Telephone.FromXML(xmlTelephone);
        telephones.Add(tel);
    }

    Contact nuovoContatto = new Contact();
    nuovoContatto.FirstName = firstName;
    nuovoContatto.LastName = lastName;
    nuovoContatto.Telephones = new ObservableCollection<Telephone>(telephones);

    return nuovoContatto;
}

```

Ed infine modifichiamo la classe AddressBook

```
public static AddressBook FromXML(XElement xml)
{
    AddressBook nuovoAddressBook = new AddressBook();
    var xmlContacts = xml.Elements("Contact");
    List<Contact> contacts = new List<Contact>();
    foreach (var xmlContact in xmlContacts)
    {
        Contact contact = Contact.FromXML(xmlContact);
        contacts.Add(contact);
    }
    nuovoAddressBook.Contacts = new ObservableCollection<Contact>(contacts);
    return nuovoAddressBook;
}
```

In AddressBookRepository, il metodo LoadContact() si limiterà ad invocare il metodo FromXML di AddressBook.

```
public class AddressBookRepository
{
    public ObservableCollection<Contact> LoadContact()
    {
        ObservableCollection<Contact> resp = new ObservableCollection<Contact>();
        string path = @"c:\temp\addressbook.xml";

        XDocument doc = XDocument.Load(path);
        XElement xmlAddressBook = doc.Element("AddressBook");

        AddressBook nuovoAddressBook = AddressBook.FromXML(xmlAddressBook);

        return nuovoAddressBook.Contacts;
    }
}
```

## Scrivere su XML

Si parte dalla classe più in fondo al modello UML

Consideriamo la classe Telephone di AddressBook

```
public XElement ToXML()
{
    //un elemento XML ha un tag, nel costruttore deo per forza avere un parametro
    //Costruisco l'elemento in base a come deve essere fatto
    // questa riga di codice crea un elemento nuovo
    XElement nuovoTelephone = new XElement("Telephone");
    // ora è necessario riempirlo
    //creo un elemento per il numero
    XElement XmlNumber = new XElement("Number", Number);
    //creo un elemento per il type
    XAttribute XmlType = new XAttribute("Type", Type);
    // a questo punto bisogna "incastrarli"
    //numero diventa figlio di telefono
    nuovoTelephone.Add(XmlNumber);
    nuovoTelephone.Add(XmlType);
    return nuovoTelephone;
}
```



Procedo con La classe Contact

```
public XElement ToXML()
{
    XElement xmlContact = new XElement("Contact");
    XElement XmlFirstName = new XElement("FirstName", FirstName);
    XElement XmlLastName = new XElement("LastName", LastName);
    xmlContact.Add(XmlFirstName);
    xmlContact.Add(XmlLastName);
    //vediamo come gestire i Telefoni
    //per ogni telefono, crea l'XML da telefono e lo aggiunge ai contatti
    foreach(var telephone in Telephones)
    {
        //Invoca il metodo ToXML della classe Telephone
        XElement xmlTelephone=telephone.ToXML();
        xmlContact.Add(xmlTelephone);
    }
    return xmlContact;
}
```

Ed infine creiamo il codice sulla classe AddressBook

```
public XElement ToXML()
{
    XElement xmlAddressBook= new XElement("AddressBook");
    //AddressBook contiene solo una collezione di contatti
    foreach (var contact in Contacts)
    {
        //Invoca il metodo ToXML della classe Telephone
        XElement xmlContact = contact.ToXML();
        xmlContact.Add(xmlContact);
    }
    return xmlAddressBook;
}
```

Passiamo in AddressBookRepository.

Devo scrivere in AddressBook i contatti

```
public void SaveContact(ObservableCollection<Contact> contacts)
{
    AddressBook addressBook = new AddressBook();
    addressBook.Contacts = contacts;

    //stabilisco il path
    string path = @"c:\temp\addressbook.xml";
    //invoco il metodo ToXML di addressBook
    XElement xmlAddressBook = addressBook.ToXML();

    xmlAddressBook.Save(path);
}
```

## Campi Mandatory

Alcuni attributi non sono obbligatori.

Vediamo come si può fare il controllo dell'obbligatorietà.

```
public static Contact FromXML(XElement xml)
{
    //Per il controllo dell'opzionalità il codice va rifattorizzato (cambiato di
posto)
    //prima vanno dichiarate le variabili
    string firstName = "";
    string lastName = "";
    string address = "";

    // successivamente vanno assegnate
    XElement xmlFirstName = xml.Element("FirstName");
    if(xmlFirstName!=null)
        firstName = xmlFirstName.Value;

    XElement xmlLastName = xml.Element("LastName");
    lastName = xmlLastName.Value;
    // leggo tutti gli altri elementi
    var xmlTelephones = xml.Elements("Telephone");

    //nelle liste il discorso dell'opzionalità può essere evitato perchè il ciclo
foreach garantisce il controllo
    List<Telephone> telephones = new List<Telephone>();
    foreach (var xmlTelephone in xmlTelephones)
    {
        Telephone tel = Telephone.FromXML(xmlTelephone);
        telephones.Add(tel);
    }

    Contact nuovoContatto = new Contact();
    nuovoContatto.FirstName = firstName;
    nuovoContatto.LastName = lastName;
    nuovoContatto.Telephones = new ObservableCollection<Telephone>(telephones);

    return nuovoContatto;
}
```

```
public XElement ToXML()
{
    //creo gli elementi
    XElement xmlContact = new XElement("Contact");
    XElement xmlFirstName = new XElement("FirstName", FirstName);
    XElement xmlLastName = new XElement("LastName", LastName);

    //Controllo dell'opzionalità
    //Prima scelta progettuale: creo tag vuoto
    // Seconda scelta progettuale

    xmlContact.Add(xmlFirstName);
    xmlContact.Add(xmlLastName);
}
```

```

//if (Address != null && Address!="")
if (!string.IsNullOrEmpty(Address))
{ XElement xmlAddress = new XElement("Address", Address);
  xmlContact.Add(xmlAddress);
}

if (!string.IsNullOrEmpty(Email))
{
  XElement xmlEmail = new XElement("Email", Email);
  xmlContact.Add(xmlEmail);
}

//Controllo opzionalità nelle liste: il foreach salta
foreach (var telephone in Telephones)
{
  XElement xmlTelephone = telephone.ToXML();
  xmlContact.Add(xmlTelephone);
}
return xmlContact;
}
}

```

## Esercitazione: Hello MVVM!

MVVM è il più famoso design pattern per lo sviluppo di applicazioni cross-platform attraverso Xamarin.

Il model rappresentano i dati e la logica business.

La view rappresenta la User Interface (bottoni, text-box...)

Il ModelView rappresenta l'intermediario tra View e Model.

In aggiunta ai tre layers occorre inserire il "binding layer" che connette il ViewModel alla View.

**Problema: Creare un'applicazione che calcola la radice quadrata di un numero.**

**Passo 0: Creare una BlankSolution: SquareRootCalculator**

**Passo 1: Creazione del Model.**

Aggiungiamo alla solution una libreria denominata ModelCalc al cui interno creeremo la classe SquareRootCalculator.

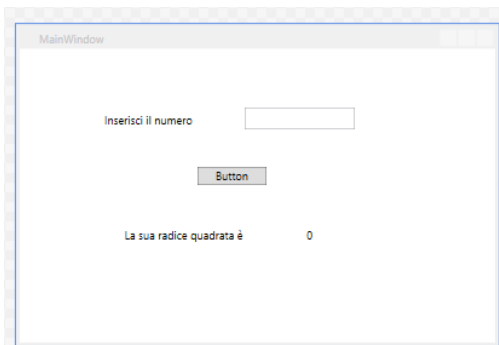
Andiamo a scrivere una possibile implementazione della classe SquareRootCalculator.

```
namespace Model
{
    public class SquareRootCalc
    {
        public double Number { get; set; }
        public double Result { get; private set; }

        public void Sqrt()
        { Result = Math.Sqrt(Number); }
    }
}
```

**Passo2: Creazione della View.**

Aggiungiamo un progetto WPF denominato WiewCalc.



**Passo 3: Creazione del ViewModel**

Aggiungiamo alla libreria un nuovo progetto denominato ViewModelCalc. Siccome il View Model fa da intermediario occorre inserire le Reference agli altri progetti.

In particolare, nella View si aggiunge il riferimento al ViewModel.

```
using System.Windows.Shapes;
using ViewModelCalc;

namespace ViewCalc
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
```

E nel ViewModel si aggiunge un riferimento al Model.

```
using ModelCalc;

namespace ViewModelCalc
{
    class ViewModelCalc
    {
    }
}
```

Creiamo l'oggetto SquareRoot di tipo SquareRootCalc (utilizzando la classe SquareRootCalc presente nel Model) nel ViewModelCalc, e quindi nel ModelView.

Il ModelView invocherà il metodo presente nella classe. Ricordiamo che nelle precedenti esperienze di laboratorio il metodo veniva invocato dall'interfaccia grafica nel MainWindow del progetto WPF.

```
namespace ViewModelCalc
{
    class ViewModelCalc
    {
        public SquareRootCalc SquareRoot { get; set; }

        public ViewModelCalc()
        { SquareRoot = new SquareRootCalc(); }

        public void Sqrt()
        { SquareRoot.Sqrt(); }

    }
}
```

Modifichiamo la classe del Model implementando l'interfaccia INotifyPropertyChanged. In questo modo sarà possibile implementare il pattern ObserverPattern. Il pattern è indispensabile per poter notificare all'interno della classe se vi sono modifiche agli attributi dell'oggetto.

```

namespace ModelCalc
{
    public class SquareRootCalc : INotifyPropertyChanged

    {
        double _number;
        public double Number
        {
            get { return _number; }
            set
            {
                _number = value;
                OnPropertyChanged("Number");
            }
        }

        double _result;
        public double Result
        {
            get { return _result; }
            private set
            {
                _result = value;
                OnPropertyChanged("Result");
            }
        }

        public event PropertyChangedEventHandler PropertyChanged;

        public void OnPropertyChanged(string propertyName)
        {
            if (PropertyChanged != null)
            {
                PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
            }
        }

        public void Sqrt()
        {
            Result = Math.Sqrt(Number);
        }
    }
}

```

Torniamo nel ModelView e creiamo una classe per il comando che effettua la radice quadrata. Ogni pulsante dell'interfaccia rappresenta un comando che va implementato come classe nel ModelView.

Negli esercizi precedenti abbiamo visto come i comandi potrebbero implementare una struttura gerarchica in quanto utilizzando tutti la stessa interfaccia ICommand hanno porzioni di codice ripetute.

```

namespace ViewModelCalc
{
    class CommandSqrt : ICommand
    {
        private ViewModelCalc _sqrtRootCalc;
        public CommandSqrt(ViewModelCalc sqrtRootCalc)
        {
            _sqrtRootCalc = sqrtRootCalc;
        }

        public event EventHandler CanExecuteChanged;

        public bool CanExecute(object parameter)
        {

```

```

        return true;
    }

    public void Execute(object parameter)
    {
        _sqrtRootCalc.Sqrt();
    }
}

```

A questo punto inseriamo il riferimento del comando alla classe ModelViewCalc del ModelView.

```

namespace ViewModelCalc
{
    class ViewModelCalc
    {
        public SquareRootCalc SquareRoot { get; set; }

        public ViewModelCalc()
        { SquareRoot = new SquareRootCalc(); }

        public void Sqrt()
        { SquareRoot.Sqrt(); }

        public ICommand CommandSqrt
        { get { return new CommandSqrt(this); } }

    }
}

```

Passiamo all'interfaccia, cioè alla View. Andiamo a creare i dati di contesto

```

namespace ViewCalc
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        private ViewModelRootCalc calcroot;
        public MainWindow()
        {
            InitializeComponent();
            calcroot = new ViewModelRootCalc();
            DataContext = calcroot;
        }
    }
}

```

Ed ora effettuiamo il binding allo XAML

```

<Grid>
    <Label x:Name="label" Content="Inserisci il numero" HorizontalAlignment="Left"
Margin="87,64,0,0" VerticalAlignment="Top"/>
    <Label x:Name="label_Copy" Content="La sua radice quadrata è"
HorizontalAlignment="Left" Margin="109,189,0,0" VerticalAlignment="Top" Width="193"/>
    <Button x:Name="BtnSqrt" Content="Square Root" Command="{Binding CommandSqrt}"
HorizontalAlignment="Left" Margin="193,128,0,0" VerticalAlignment="Top" Width="75"/>

```

```
<TextBox x:Name="TxtNumber" HorizontalAlignment="Left" Height="23"
Margin="245,64,0,0" TextWrapping="Wrap" Text="{Binding Path=SquareRoot.Number}"
VerticalAlignment="Top" Width="120"/>
<Label x:Name="LblRisultato" Content="{Binding Path=SquareRoot.Result}"
HorizontalAlignment="Left" Margin="307,189,0,0" VerticalAlignment="Top"/>

</Grid>
```

L'esecuzione andrà a buon fine.