20. LA PROGRAMMAZIONE AD OGGETTI

Premessa

All'inizio del nostro studio della programmazione il nostro **obiettivo primario** è stato quello di **iniziare a programmare (**ossia a **codificare)**.
Per fare ciò abbiamo studiato i costrutti base della **programmazione strutturata** *(istruzioni di sequenza, selezione ed iterazione).*
Abbiamo studiato i concetti di **variabili semplici e strutturate** *(array e record*) e siamo giunti alla conclusione *(secondo la ben nota equazione dell'informatico svizzero Niklaus Wirth che riassume efficacemente il paradigma della programmazione procedurale di tipo imperativo)* che per un qualunque programma vale la seguente legge

PROGRAMMA = ALGORITMO + STRUTTURE DATI

In pratica finora ci siamo occupati più degli algoritmi che delle strutture dati che in molti casi altro non erano che *strutture dati al servizio degli algoritmi* e non *strutture dati ottenute dall'osservazione della realtà* che volevamo rappresentare.

La "rivoluzione" informatica che propone la **programmazione orientata agli oggetti o object-oriented programming o O-O-P** nata intorno al 1970 è proprio questa: *creare dei programmi in grado di riflettere l'ordine delle cose del mondo reale che vogliono rappresentare.*

In sintesi:
- la programmazione strutturata nasce come suddivisione di un intero programma in un vasto numero di sottoprogrammi (funzioni e/o procedure) sulle quali si pone la nostra attenzione
- la programmazione ad oggetti sposta la sua attenzione verso l'**oggetto** come entità indipendente, costituito da dati e procedure, mediante le quali può dialogare con altri oggetti

Astrazione, classi ed oggetti

Con la programmazione ad oggetti è possibile modellare la realtà di interesse in un sistema software in modo più naturale e vicino all'uomo.

Per giungere alla definizione di un oggetto è necessario l'utilizzo dell'astrazione

Def: L'**astrazione** è un procedimento mentale che permette di evidenziare alcune proprietà, ritenute significative, relative ad un determinato sistema osservato escludendone altre considerate non rilevanti per la sua comprensione.

Appare evidente che l'utilizzo dell'astrazione permette la creazione di classi

Def: Una **classe** è un modello astratto generico per una famiglia di oggetti con caratteristiche comuni.
Una **oggetto** (o *istanza*) è una rappresentazione concreta e specifica di una *classe.*

Def: Per **processo di istanziazione** si intende la possibilità che una classe permetta la generazione di oggetti (o istanze) in modo che ciascuno possa contenere informazioni specifiche che lo differenziano dagli altri.

Proprietà (o attributi) e Metodi

Un **oggetto** rappresenta un elemento software che contiene un insieme di procedure e dati collegati. Le procedure si chiamano **metodi** ed i dati si chiamano **proprietà (o attributi)**.

Una classe quindi si compone generalmente di due parti: quella per la definizione delle **proprietà** (o **attributi)** e quella per la definizione dei **metodi**.

N.B. *Lo* ***pseudocodice*** *che utilizzeremo per la programmazione ad oggetti è molto vicino a quello che abbiamo utilizzato per la programmazione imperativa. Alcune differenze anche sostanziali verranno illustrate di volta in volta all'occorrenza.*

CLASSE <NomeClasse>
INIZIO

[<ListaProprietà> **]**
[<ListaMetodi> **]**

FINE

dove **<ListaProprietà>** può essere:
<NomeProprietà**1**> **:** <TipoProprietà**1**>
……..
<NomeProprietà**M**> **:** <TipoProprietà**M**>

e dove **<Lista Metodi>** può essere:
<NomeMetodo**1**> (**[** <ListaParametri> **]**) **[:**<TipoValoreRitorno> **]**
INIZIO
…
FINE
……
<NomeMetodo**P**> (**[** <ListaParametri> **]**) **[:**<TipoValoreRitorno> **]**
INIZIO
…
FINE

Creazione di un'istanza di una classe (oggetto) e metodi costruttori

Nella programmazione ad oggetti per creare un oggetto si utilizzano due fasi distinte:

- FASE 1: **dichiarazione** dell'oggetto
- FASE 2: **allocazione** dell'oggetto

In pseudocodifica: dichiarazione dell'oggetto

<NomeOggetto> **:** <NomeClasse>

N.B. Un oggetto dichiarato non può essere ancora utilizzato perché per farlo occorre allocarlo in memoria.

Allocare in memoria un'istanza di una certa classe significa riservare un'*area di memoria* (RAM,) delle dimensioni opportune per accogliere tutte le *variabili istanza* dell'oggetto stesso.

In pseudocodifica: allocazione dell'oggetto

<NomeOggetto> ← <NomeMetodoCostruttore> (**[** <ListaParametri> **]**)

dove:
- **<NomeOggetto>**: è il nome che vogliamo dare all'oggetto della classe;

- <**NomeMetodoCostruttore>** : è un *metodo* (funzionalità) speciale (sempre presente anche se non compare esplicitamente nella definizione di *classe)* detto **costruttore di classe** in grado, dopo la sua invocazione di creare un'oggetto ossia di allocare lo spazio di memoria necessario a contenerlo nella RAM e provvedere a tutte le inizializzazioni delle sue proprietà**.**

Per convenzione il nome del metodo costruttore deve essere lo stesso di quello della *classe*;

- **<ListaParametri>**: contiene l'elenco (eventualmente vuoto) dei parametri necessari al metodo costruttore per creare l'istanza.

In pratica **i metodi costruttori** inizializzanzo un nuovo oggetto con le sue variabili, creano eventualmente altri oggetti necessari all'oggetto originale ed, in generale. compiono tutte le operazioni necessarie per la *nascita* dell'oggetto in questione.

N.B. Se il costruttore di classe non è esplicitamente presente, i valori iniziali delle proprietà dei suoi oggetti saranno definiti *per default* dal sistema.

Interazione tra oggetti

Dopo aver creato le **classi** (con *proprietà e metodi)* e gli **oggetti** (*istanze della classe*) che ci occorrono, dobbiamo vedere come tali oggetti comunicano tra di loro per realizzare un algoritmo.

Accesso ai metodi:

Gli oggetti comunicano mediante **scambio di messaggi**.
Un messaggio parte da un oggetto **mittente** e giunge ad un oggetto **destinatario** il quale, all'atto della ricezione, *attiva il comportamento* richiesto dal *mittente*.

Un messaggio è quindi logicamente costituito da:
- il **nome** dell'*oggetto destinatario*;
- il **nome** del *metodo dell'oggetto destinatario* che si vuole attivare;
- un **elenco** di *parametri attuali* che vengono passati al metodo dell'oggetto destinatario.

In pseudocodifica: accesso ai metodi

<NomeOggetto> **.** <NomeMetodo> ([<ListaParametriAttuali) >])

Accesso alle proprietà (o attributi):

Per accedere direttamente ad una proprietà (o attributo) basterà usare la stessa notazione usata per i metodi ovvero:

In pseudocodifica: accesso alle proprietà

<NomeOggetto> **.** <NomeProprietà>

Per accedere meglio ad una proprietà di un oggetto si può far ricorso ad un metodo ad hoc che possa effettuare **modifiche della proprietà**.

Bisognerà aggiungere un nuovo metodo per ogni proprietà di cui si voglia modificare il valore detto **metodo Set( )**, ed un nuovo metodo per ogni proprietà di cui si voglia leggere il valore detto **metodo Get( ).**

In pseudocodifica: sintassi metodi Set ( ) e Get ( )

Set <NomeProprietà> ( )
Get <NomeProprietà> ( )

N.B. L'utilizzo dei metodi Set ( ) e Get ( ) permette di gestire meglio l'*accesso ad una proprietà* riuscendo ad implementare anche i controlli necessari ad evitare le controindicazioni possibili che l'*accesso diretto* porta con se.

I metodi distruttori

I **metodi distruttori** o **conclusivi** sono il contrario dei *metodi costruttori*: un *metodo costruttore* alloca ed inizializza un oggetto mentre un *metodo distruttore* viene richiamato per svolgere le operazioni finali e deallocare l'oggetto ossia *liberare la memoria occupata*.

Questa memoria può essere deallocata in **modo esplicito** *dal programmatore* attraverso opportune primitive del linguaggio OOP oppure **in modo implicito,** per ragioni di efficienza, *dal sistema operativo*.

La programmazione ad oggetti: I CONCETTI CHIAVE

La programmazione ad oggetti è in generale caratterizzata dalla presenza di quattro elementi fondamentali

1) INCAPSULAMENTO ed information hiding

Abbiamo visto che due oggetti dialogano tra loro tramite uno *scambio di messaggi* ed abbiamo supposto finora che tutti i metodi e gli attributi di un oggetto possano essere visibili agli altri oggetti.
In realtà ogni oggetto mittente *non è obbligato a conoscere tutti i metodi dell'oggetto destinatario* poiché sono sufficienti solo quelli che il destinatario ritiene opportuno.

Con **interfaccia con l'esterno** o semplicemente **interfaccia** indichiamo la lista di proprietà e metodi (per i metodi solo la **segnatura** ossia il *nome*, l'eventuale *tipo* del valore di ritorno e la *lista ordinata* con i *nomi* ed il *tipo* dei suoi *parametri*) che l'oggetto rende noti all'esterno e che sono utilizzabili per interagire con esso.

In questo modo *l'oggetto chiamante* conosce solo il modo di interagire con *l'oggetto chiamato* anche **se ignora completamente** *i dettagli implementativi* di quest'ultimo.

Per definire l'interfaccia con l'esterno occorre introdurre le *clausole di visibilità*:
PUBBLICO e **PRIVATO**.

La clausola **PUBBLICO** *rende visibile all'esterno* le segnature dei metodi e le proprietà che precede.
La clausola **PRIVATO** invece *nasconde all'esterno* le segnature dei metodi e le proprietà che precede.
L'implementazione dei metodi (sia pubblici sia privati) non è comunque visibile all'esterno.

Quanto appena detto prende il nome di **information hiding** ossia **mascheramento delle informazioni**.

Per **information hiding (o mascheramento delle informazioni)** si intende la capacità di un oggetto di nascondere i dettagli implementativi di un suo metodo rendendo nota all'esterno solo la sua

Il principale vantaggio dell'*information hiding* per un oggetto consiste nel poter modificare l'implementazione *in modo trasparente* ossia senza dover informare tutti gli oggetti che interagiscono con l'oggetto stesso.
Questo è uno dei modi più corretti di gestire la manutenzione del software.
Un *nuovo programmatore* può intervenire sull'implementazione di una classe senza causare gli spiacevoli "effetti collaterali" che puntualmente si verificano nella programmazione imperativa.

Per **incapsulamento** si intende la stretta unione di *dati* e *metodi* di una *classe*.

L'incapsulamento rappresenta il punto di arrivo di un'evoluzione cominciata negli anni '60 con la nascita dell'idea di modulo e di information hiding e continuata negli anni '70 con l'introduzione del concetto di tipo di dato astratto.
In base al principio di incapsulamento, un oggetto è una**"scatola nera"** o **black box** che racchiude una struttura dati (*stato dell'oggetto*). Questa struttura non può essere manipolata direttamente ma solo attraverso i *metodi* dell'oggetto.

L'incapsulamento introduce una distinzione fondamentale fra *interfaccia* e *implementazione* aumentando la capacità di un sistema software di resistere ai cambiamenti (*resilience*).
E' possibile cambiare completamente la struttura interna di un oggetto senza influenzare il resto del sistema: basta conservare l'interfaccia.
E' in pratica **il principio dei compartimenti stagni.**

2) ASTRAZIONE

Realizzare un programma con *un linguaggio orientato agli oggetti* vuol dire individuare le **classi** che occorrono e metterle insieme per permettere l'interazione tra oggetti di quelle classi.
Per *individuare una classe* occorre mettere in atto un **processo di astrazione.**
Tale processo prende il nome di **astrazione per classificazione**: in pratica si parte da oggetti reali e si cercano le caratteristiche ed i comportamenti comuni.

> L'**astrazione per classificazione** è un procedimento mentale che permette di definire una classe a partire da un insieme di oggetti di cui si individuano le proprietà comuni.

L'astrazione è dunque un concetto intrinseco nelle classi: infatti è possibile che più oggetti posseggano lo stesso tipo di metodi ed anche gli stessi tipi di proprietà, sebbene con valori diversi.

Esempio Tante automobili della stessa marca e modello ma di colore diverso sono tutte oggetti appartenenti alla stessa classe "Automobile".

Nel momento in cui si genera un oggetto si dice che si è ottenuta un'istanza della classe.
Le istanze ottenute da una stessa classe possono differire per il valore delle proprietà ma tutti i metodi sono gli stessi.
Un'istanza di una classe è qualcosa di concreto, esistente nella realtà mentre la classe cui appartiene è qualcosa di astratto che ne riassume le caratteristiche generali.

3) EREDITARIETA': overriding ed overloading

E' senza dubbio la caratteristica più importante della programmazione ad oggetti.

Abbiamo già detto che alla base di tale programmazione ci sono gli oggetti che vengono creati a partire da classi precedentemente realizzate.

Molto spesso non è indispensabile o utile creare una classe dal nulla, ma è possibile utilizzare una classe già esistenti dalla quale partire per la sua creazione, specificando solo le differenze con quest'ultima.

Nella programmazione ad oggetti è possibile fare questo utilizzando il concetto di **ereditarietà**.

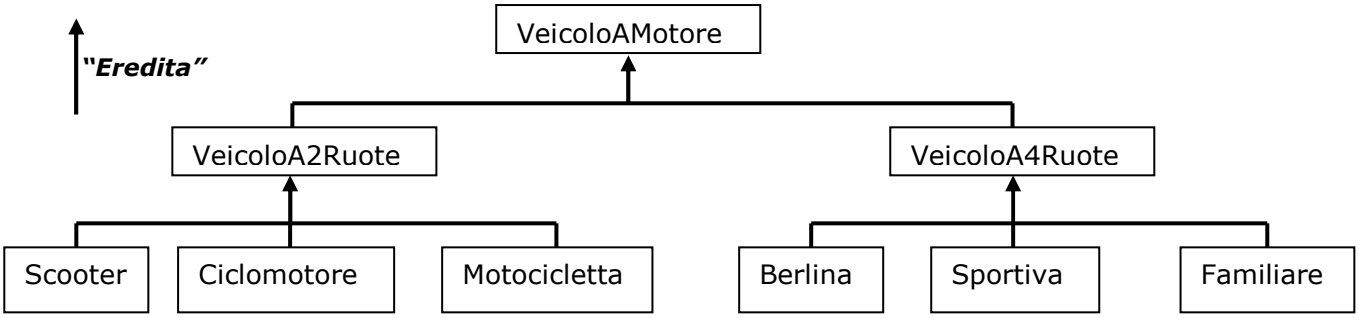
> L'**ereditarietà** è un meccanismo che, in fase di definizione di classe, permette di specificare solo le differenze rispetto ad una classe già esistente, detta **superclasse**.
> Tutte le altre caratteristiche ed i comportamenti della classe chiamata **sottoclasse** che si sta definendo **saranno gli stessi della superclasse**.

Con questo concetto di ereditarietà si introduce quello di **gerarchia delle classi**.

Ad ogni classe possono essere associate una o più classi che la *precedono* immediatamente sopra nella gerarchia (le sue **superclassi**).

Ad ogni classe possono essere associate una o più classi che la *seguono* immediatamente sotto nella gerarchia (le sue **sottoclassi**).

Esempio: Utilizziamo la seguente rappresentazione grafica per evidenziare la gerarchia tra le classi.



La classe *Motocicletta* è una **sottoclasse** della classe *VeicoloA2Ruote*; la classe *VeicoloA2Ruote* è **superclasse** della classe *Motocicletta* e **sottoclasse** della classe *VeicoloAMotore*.

Una classe **eredita** dalle classi superiori nella gerarchia tutti i loro *metodi* e le loro *proprietà*

Se la superclasse definisce già un comportamento o una proprietà richiesto dalla classe non occorre ridefinire quest'ultimo o copiarne il codice, in quanto l**a classe ottiene automaticamente il comportamento o la proprietà della sua superclasse** e così via.

La **classe** diventa una *combinazione* dei *comportamenti* e delle *proprietà* di **tutte le superclassi** che la *precedono* nella **gerarchia delle classi**.

Quando una **sottoclasse** possiede un **metodo** con la **stessa segnatura** della **superclasse** ma con ***codice*** (implementazione) *differente* allora non viene più ereditato il metodo della superclasse.
Si dice che la sottoclasse *espressamente* **ridefinisce** (**overriding**) il metodo della superclasse.

Quando una **sottoclasse** possiede un **metodo** con lo **stesso nome** della **superclasse** ma con ***numero e/o tipo di parametri*** *differente* allora non viene più ereditato il metodo della superclasse.
Questa situazione è nota come **overloading dei metodi**.

Quando una **sottoclasse** aggiunge nuovi metodi ed attributi non presenti nella superclasse si dice che la sottoclasse *estende metodi ed attributi.*

Una nuova classe può dunque differenziarsi dalla superclasse o **per ridefinizione** (overriding e/o overloading) **o per estensione.**

In sintesi una classe può:
- **ereditare** metodi ed attributi dalla superclasse (*ereditarietà*);
- **estendere** la superclasse **aggiungendo** nuovi metodi ed attributi (*estensione*);
- **ridefinire** metodi ed attributi (*overriding*);
- **ridefinire** metodi (*overloading*);

4) POLIMORFISMO

Il **polimorfismo** è la capacità espressa dai *metodi ridefiniti* di assumere forme (*implementazioni*) diverse all'interno di una gerarchia di classi o all'interno di una stessa classe.

In altre parole il **polimorfismo** indica la possibilità dei metodi di possedere diverse implementazioni.

Quando un oggetto richiama un metodo di un altro oggetto appartenente ad una certa classe, esso verrà cercato dapprima in quella stessa classe. Se viene trovato (*stessa segnatura della chiamata*) sarà eseguito, altrimenti verrà ricercato risalendo nell'albero della gerarchia di classe (*tra le sue superclassi*).

Binding statico e binding dinamico

Il concetto di **polimorfismo** è strettamente collegato a quello di **binding dinamico** o **associazione posticipata** o ancora **collegamento ritardato** (**late binding**).

Nella **programmazione imperativa** quando il compilatore incontra una chiamata ad un sottoprogramma (funzione o procedura) con i relativi parametri attuali, realizza un legame tra tali parametri ed il sottoprogramma che deve essere chiamato. (**binding statico o early binding**)
Quindi il legame tra parametri e sottoprogramma è già noto e perfettamente specificato **a tempo di compilazione.**
Il **compilatore** nei linguaggi imperativi ad ogni chiamata di sottoprogramma fissa quale parte del codice deve essere eseguita e su quali dati farlo.
Si realizza un **legame statico** o **binding statico** o **associazione anticipata** tra il dato (parametro attuale) ed il sottoprogramma chiamato.

Nella **programmazione ad oggetti** proprio a causa del polimorfismo vi è **l'impossibilità** di stabilire **a tempo di compilazione** il legame tra la *chiamata* di un *metodo* e la sua *definizione*
Il legame tra nome del metodo ed il suo codice sarà *posticipato* **a tempo di esecuzione** e non potrà essere risolto *a tempo di compilazione*.

VISUAL BASIC

Visual Basic della piattaforma .NET framework di Microsoft (VB NET) è un **linguaggio di programmazione pseudo compilato**

Riassumendo in sintesi:

- linguaggio compilato (es. C, C++, FORTRAN, PASCAL, etc.): è un linguaggio di programmazione di alto livello per il quale la trasformazione dal codice sorgente al codice macchina (programma eseguibile) avviene mediante programmi, chiamati *compilatori*, che traducono tutte le istruzioni del programma sorgente in codice binario in un'unica volta.
Quindi traduzione ed esecuzione sono fatte in tempi differenti;

- linguaggio interpretato (es BASIC, JAVASCRIPT, etc.): è un linguaggio di programmazione di alto livello per il quale la trasformazione dal codice sorgente al codice macchina eseguibile avviene mediante programmi, chiamati *interpreti*, che traducono ed eseguono le istruzioni del programma sorgente una alla volta.
Quindi traduzione ed esecuzione procedono contemporaneamente una di seguito all'altro alternandosi.

- linguaggio pseudocompilato (es JAVA, VB NET, etc.) : è un linguaggio di programmazione di alto livello per il quale il programma sorgente viene dapprima compilato in codice intermedio simile all'assembler e successivamente viene eseguito da un interprete. Siamo quindi in una situazione intermedia tra la compilazione e l'interpretazione o meglio metà lavoro viene svolto dal compilatore e metà dall'interprete.

Visual Basic è anche un linguaggio di **programmazione ad eventi o *even-driven***

La **programmazione a eventi** è un paradigma di programmazione dell'informatica. Mentre in un programma tradizionale l'esecuzione delle istruzioni segue percorsi fissi, che si ramificano soltanto in punti ben determinati predefiniti dal programmatore, nei programmi scritti utilizzando la tecnica *a eventi* il flusso del programma è largamente determinato dal verificarsi di eventi esterni (quali ad esempio la pressione di un tasto della tastiera oppure un click del mouse oppure ancora l'apertura di una finestra etc.).

I programmi che utilizzano la programmazione a eventi sono composti tipicamente da diversi brevi sotto-programmi (routine) , chiamati **gestori degli eventi** (*event handlers*), che sono eseguiti in risposta agli eventi esterni, e da un **dispatcher**, che effettua materialmente la chiamata, spesso utilizzando una *coda degli eventi* che contiene l'elenco degli eventi già verificatisi, ma non ancora "processati".

In molti casi i gestori degli eventi possono, al loro interno, innescare (*"trigger"*) altri eventi, producendo una cascata di eventi.

VISUAL BASIC e la programmazione ad oggetti

Un programma Visual Basic è formato da un insieme di classi.

In questo linguaggio di programmazione i due tipi di classi più comuni sono i **Form** e le **Classi utente.**
La principale differenza tra le due tipologie è che rispetto ad una Classe utente i Form contengono già incorporata un'interfaccia grafica utente

L'**interfaccia grafica utente (GUI o** *Graphic User Interface***)** consente all'utente di interagire con il computer manipolando graficamente degli oggetti in maniera molto accattivante ed user-friendly, al contrario di quanto avviene sulla cosiddetta *riga di comando* di una **CLI** (*Command Line Interface*), in cui l'esecuzione del programma viene guidata da istruzioni o comandi impartiti dall'utente tramite tastiera.

L'elemento visuale principale di un'applicazione Visual Basic di tipo Windows Form (applicazione con interfaccia Windows) è la **finestra** (o **Form**) sulla quale possiamo disporre i **controlli grafici** ai quali è associato il codice scritto in Visual Basic.
Il codice Visual Basic associato ad un controllo viene eseguito quando si verifica un **evento**.

I principali elementi sui quali poggia la programmazione visuale sono quelli illustrati di seguito:

Oggetti: rappresentano gli oggetti della programmazione visuale come ad esempio le finestre (o Form) ed i controlli grafici (caselle di testo, pulsanti, caselle combinate, etc.).
Ciascun oggetto è identificato da un nome (proprietà *name*) ed è l'istanza di una classe.

Proprietà: ciascun oggetto possiede alcune proprietà che ne specificano le caratteristiche (nome, dimensioni, colore, posizione nel Form, etc.) che possono essere impostate o direttamente nella sezione apposita prevista oppure utilizzando nel codice la dot notation.

Eventi: ciascun oggetto riconosce alcune azioni specifiche prodotte o dall'utente o dal sistema perativo, chiamate eventi, in risposta alle quali il programmatore può predisporre la relativa routine (sottoprogramma) di evento che verrà eseguita al verificarsi di quello specifico evento.

Metodi: rappresentano le azioni eseguibili sugli oggetti. Si tratta di Function o Sub associate ai controlli e possono essere eseguite secondo la dot notation.

Speciale laboratorio

VISUAL BASIC 2008 Express Edition : c

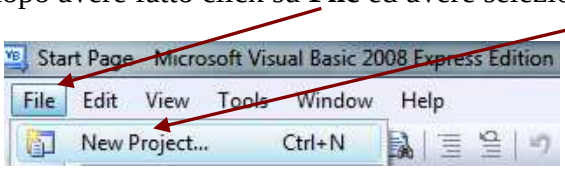
a) Creare la prima applicazione Windows Form

Ora impareremo come realizzare un programma che consente di utilizzare i controlli tipici di Windows, quali, per esempio, i pulsanti, le caselle di tetso, etc.

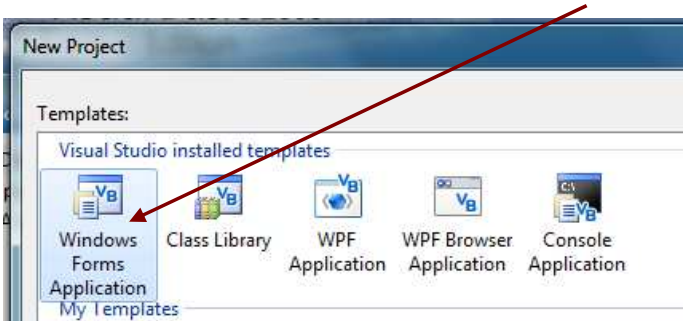
Per creare un nuovo progetto di modello Applicazione Windows Form dobbiamo:

1) Prima di tutto aprire l'ambiente di sviluppo Visual Basic 2008 Express Edition facendo doppio click sull'icona relativa

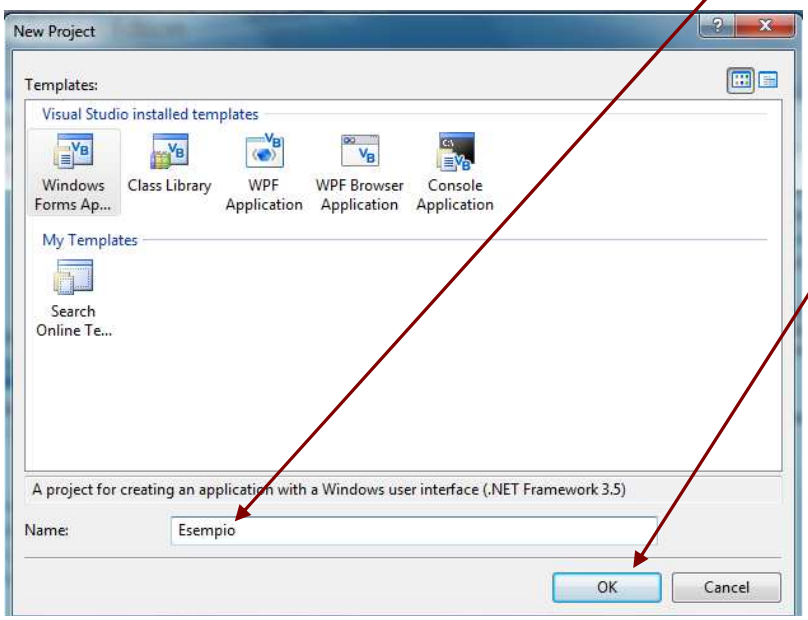
poi dopo avere fatto click su **File** ed avere selezionato **Nuovo Progetto…**



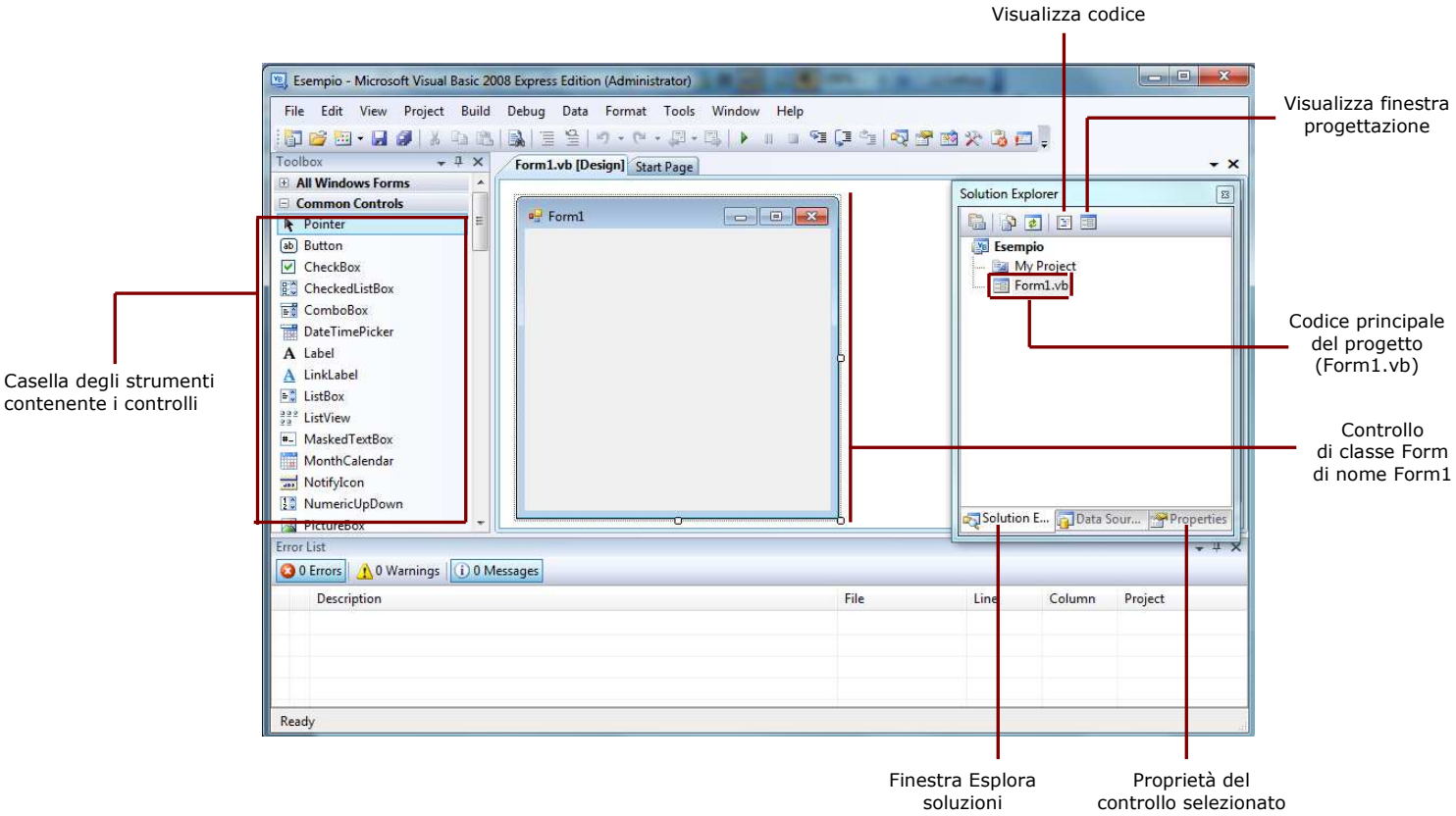
scegliere la prima icona, denominata **Applicazione Windows Form**



2) Dopo avere digitato il nome del progetto (in questo caso *Esempio*) e cliccato su OK

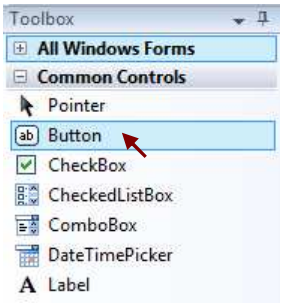


si ottiene la schermata iniziale dell'IDE suddivisa in diverse sezioni, come mostrato dalla figura seguente



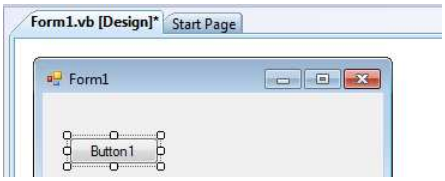
La finestra *Esplora soluzioni* elenca tutti i Form, i moduli e le classi del progetto selezionabili con il mouse.
Attraverso le icone *Visualizza finestra progettazione* e *Visualizza Codice* della finestra *Esplora soluzioni* si passa dalla visualizzazione del "disegnatore" del Form al codice Visual Basic associato e viceversa.

3) Passiamo all'inserimento di un controllo all'interno del Form1.
Proviamo ad immettere un pulsante (controllo *Button*).
Per far ciò occorre selezionare il controllo desiderato con il mouse, come mostrato a lato, all'interno della casella degli strumenti



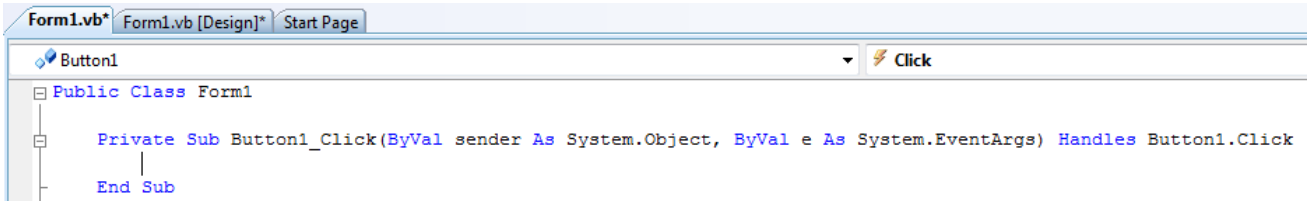
4) Per disegnare un nuovo pulsante nel Form, dopo averlo selezionato nella casella degli strumenti, occorre trascinare il mouse fino a raggiungere la dimensione desiderata

5) A questo punto una volta lasciato il mouse, appare il disegno che raffigura il controllo, in questo caso un pulsante di classe *Button* di nome *Button1*

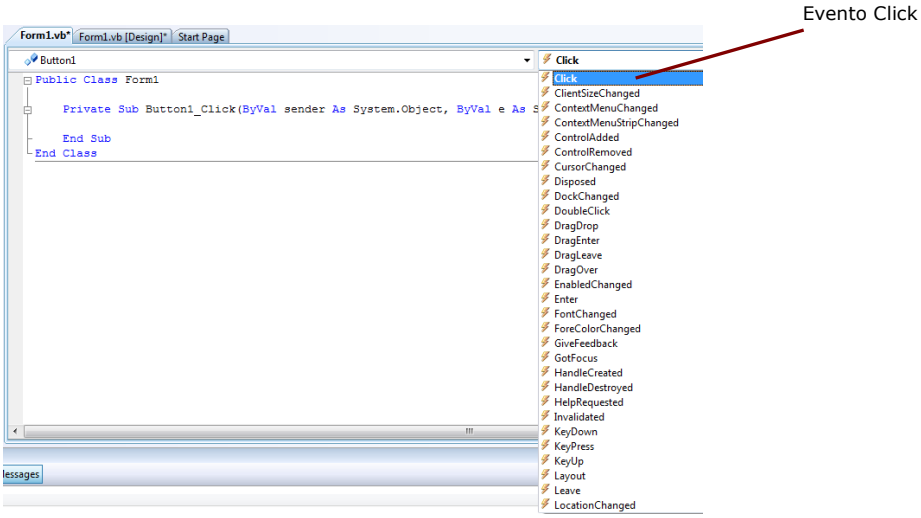


6) E' possibile associare del codice Visual Basic al controllo facendo doppio click sul pulsante per creare una routine di risposta al click

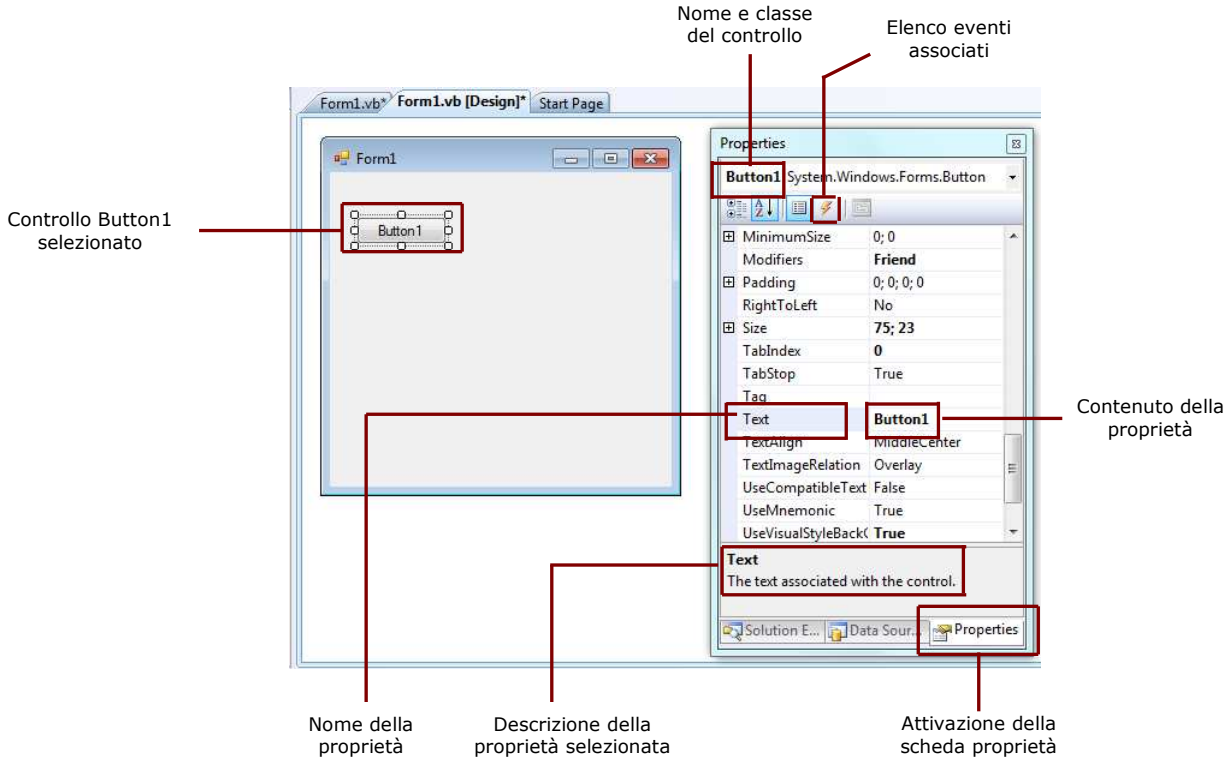
7) Si apre automaticamente una nuova Sub con lo stesso nome del controllo e dell'evento separato dal carattere underscore (*Button1_Click*)



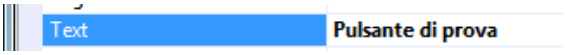
8) Nella finestra del codice (Form1.vb) sono presenti due caselle a tendina.
La prima a sinistra, contiene l'elenco degli oggetti presenti nel progetto;
la seconda sulla destra contiene l'elenco degli eventi attivabile da quel particolare oggetto selezionato nella casella associato.
In questo caso si può notare che l'evento di default è *Click*.



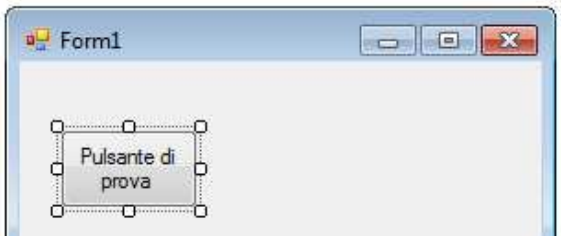
9) Un oggetto possiede **proprietà** che possono essere modificate attraverso la finestra *Proprietà* come mostrato nella figura seguente



10) Si può notare che tra le proprietà vi è il testo mostrato sopra il pulsante (proprietà *Text*): per modificarlo è sufficiente fare click all'interno della casella di testo posta accanto (contenuto della proprietà *Text*), assegnandogli il testo seguente



11) Il pulsante ora ha cambiato aspetto



N.B. Occorre non confondere la proprietà *Text* con la proprietà *Name*: sul pulsante appare il testo "Pulsante di prova", tuttavia il nome dell'oggetto è rimasto *Button1* ed è proprio questo il nome che occorre utilizzare per identificarlo nel codice.

12) E' possibile ora passare alla scrittura del codice da eseguire al click sul pulsante.
Basta fare doppio click sul pulsante oppure selezionare l'evento *Click* dal menù a tendina posto in alto.
si ottiene La seguente procedura all'interno della quale è possibile scrivere il seguente codice da eseguire (*MsgBox*)

```
Public Class Form1

    Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button1.Click
        MsgBox("Hai premuto un pulsante")
    End Sub
End Class
```

13) Una volta eseguito il programma si ottiene la visualizzazione del Form1: il click sul pulsante Button1 determina l'attivazione della finestra che segue:

b) L'output tramite tastiera: sintassi completa dell'istruzione MsgBox