Sommario

Introduzione

Questa parte ha lo scopo di introdurre lo studente alla realizzazione di programmi dotati di interfaccia grafica, definiti Applicazioni Windows poiché usufruiscono delle funzionalità grafiche del sistema operativo Windows (diversamente dalle Applicazioni Console, che fanno uso di una semplice interfaccia a caratteri)

Il testo non rappresenta affatto una guida sulla programmazione windows; l'argomento in sé è semplicemente troppo vasto per essere affrontato in un volume introduttivo. Per questo motivo, fatta salva la presentazione dei concetti e degli elementi di base necessari per la realizzazione di applicazioni grafiche, sono stati presi in esame soltanto alcuni degli argomenti inerenti la programmazione windows, lasciando allo studente la responsabilità di un loro approfondimento.

All'inizio si introduce il termine **interfaccia utente** e si definiscono le caratteristiche che differenziano un tipo di interfaccia da un altro. Quindi vengono presentati due tipici modelli di interfaccia utente, gli unici presi in considerazione in tutto il corso di studi. Infine, viene esaminato il modello di interfaccia usato dalle Applicazioni Console, in modo da poterlo usare come termine di paragone nella successiva analisi del modello grafico impiegato dalle Applicazioni Windows.

Vengono prima introdotti gli aspetti che caratterizzano una Applicazione Windows, sia nella gestione del video che nella comunicazione con l'utente, nonché i costituenti fondamentali di un'interfaccia grafica, e cioè i controlli. Quindi, prendendo come riferimento un semplice programma dotato di interfaccia grafica, viene affrontato il compito di costruirne uno che produca lo stesso funzionamento.

Proseguendo viene presentato un "mini" reference sugli elementi di base che caratterizzano una interfaccia grafica. Vene introdotta la classe `Control`, che espone proprietà, metodi ed eventi comuni a tutti i controlli, dei quali sono presi in considerazioni quelli fondamentali: `Form`, `Label`, `Button`, `TextBox`. Vengono inoltre trattate le *message dialog* e gli eventi sollevati dal mouse e dalla tastiera. Viene infine introdotto il concetto di verifica della consistenza dell'interfaccia e della validità dai dati inseriti dall'utente.

La scrittura di applicazioni Windows sofisticate viene resa possibile mediante l'introduzione di altri controlli come – `ListBox`, `ComboBox`, `RadioButton`, `CheckBox`, `Panel`, `GroupBox`, `PictureBox` – i quali consentono un'interazione con l'utente più sofisticata.

Si passa poi ad analizzare la creazione di un'applicazione grafica dimostrativa, che ha lo scopo di tradurre in pratica gli argomenti introdotti nei capitoli precedenti. Viene inoltre approfondito il problema della verifica della «consistenza» dell'interfaccia, prendendo come spunto l'applicazione di esempio precedentemente realizzata.

Successivamente si passa all'approfondimento di alcuni argomenti avanzati concernenti la realizzazione di Applicazioni Windows. Occorre dire, comunque, che dato il taglio introduttivo del libro si è sorvolato su molti elementi importanti inerenti le interfacce grafiche, peraltro pienamente supportati dal ricco modello ad oggetti fornito da .NET Framework.

Per prima cosa si affronta l'argomento delle finestre di dialogo, o form modali. Dopo una breve introduzione teorica, viene spiegato come realizzare finestre di dialogo che consentano all'utente di fornire dei dati al programma. Quindi vengono iontrodotte le «finestre di dialogo comuni», e cioè classi fornite da .NET che implementano le finestre di dialogo comunemente usate in tutte le applicazioni windows.

Si prosegue con uno spazio è dedicato alle funzionalità di disegno fornite da .NET. Queste sono implementate da uno strato software che ricade sotto il nome di GDI+: *Graphics Device Interface, plus*. Inoltre il capitolo tratta gli argomenti di base e cioè i tipi e i metodi da utilizzare per eseguire il disegno di figure geometriche, immagini e stringhe di testo.

Per ultimo si affronta il tema della gestione del rendering di un controllo, e cioè del codice di disegno che viene eseguito ogni qual volta l'aspetto visuale del controllo necessita di essere

aggiornato. Il tema viene trattato da due punti di vista distinti: la scrittura di codice *consumer* che interviene sul rendering del controllo; la realizzazione di un nuovo tipo controllo, basato su un tipo esistente, che fornisce una propria gestione della fase di rendering.

10

Interfaccia utente: concetti di base

1 Interfaccia utente

Con il termine **interfaccia utente** viene designata quella parte del programma che gestisce la comunicazione con l'utente, consentendo a questo di immettere le informazioni da elaborare e di ricevere il risultato di tale elaborazione. In maggior dettaglio, il termine interfaccia denota due significati distinti, anche se collegati, in base alla prospettiva dalla quale lo si considera:

- dal punto di vista dell'utente, l'interfaccia è rappresentata dalla modalità di rappresentazione su video dei dati e dalla modalità di acquisizione degli stessi;
- dal punto di vista del programmatore, l'interfaccia è rappresentata della parte di codice che svolge la funzione di ricevere le azioni dell'utente e di rispondere ad esse, visualizzando su video.

Detto questo, ogni programma può dunque intendersi logicamente suddiviso in due parti. la parte **interfaccia**, che svolge la funzione di comunicare con l'utente e dunque di ricevere i dati da elaborare e di comunicare i risultati della elaborazione. La parte **elaborazione**, con la quale si identifica il codice demandato alla vera e propria elaborazione dei dati.

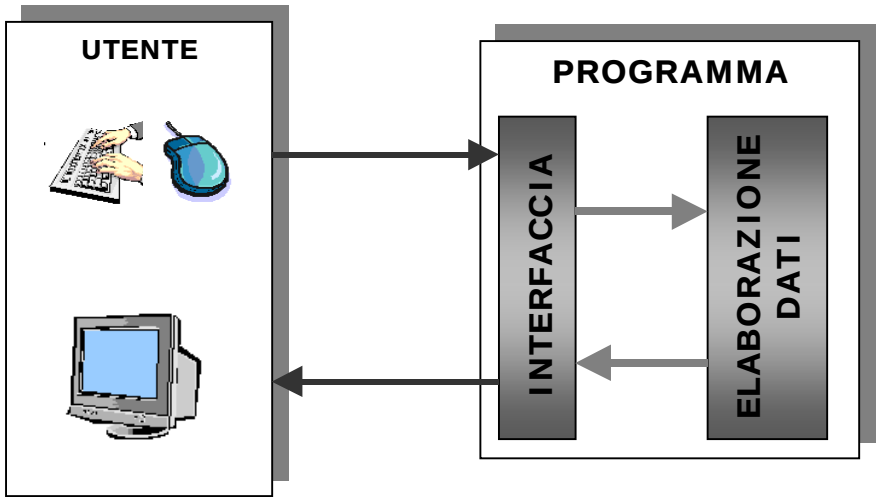


Figura 10-1 Rappresentazione schematica generale di un programma.

Va sottolineato che una simile suddivisione dev'essere intesa in senso concettuale; in altre parole, perlomeno in programmi realistici, non esiste una demarcazione netta tra le istruzioni che appartengono alla parte interfaccia e quelle dedicate all'elaborazione, e nemmeno sarebbe utile una cosa del genere. L'importante è comprendere che ogni programma deve soddisfare due tipi di compiti logicamente distinti: ricevere e rispondere alle azioni dell'utente: e ciò è carico della parte chiamata interfaccia; elaborare i dati: e ciò è a carico della parte chiamata elaborazione.

1.1 Caratteristiche di una interfaccia utente

Sono due gli aspetti che caratterizzano l'interfaccia di un programma:

1) le gestione del video, e dunque la forma con la quale le informazioni vengono visualizzate;
2) il modello di comunicazione con l'utente, e dunque il modo in cui il programma riceve le azioni e i dati dall'utente.

Per ognuno di questi aspetti esistono fondamentalmente due possibilità.

Gestione del video

La gestione del video, e cioè l'apparenza dell'interfaccia, può essere di tipo **a caratteri** (o **testuale**) o **grafica**.

Nel primo caso, il video viene trattato come una matrice di caratteri, il cui numero di righe e di colonne dipende da alcune impostazioni del sistema operativo. Il singolo elemento visualizzabile è il carattere, e dunque l'interfaccia può rappresentare solo informazioni testuali; sono escluse forme geometriche, bordi, immagini, ombreggiature, animazioni, eccetera.

Nel secondo caso il video viene visto come una matrice di punti, chiamati **pixel**, idealmente molto simile a un grafico cartesiano, dotata degli assi delle ascisse e delle ordinate. Poiché il numero di pixel è molto superiore a quello dei caratteri disponibili in un interfaccia a caratteri (minimo 480 pixel in ordinata e 640 in ascissa), un'interfaccia grafica consente la visualizzazione di elementi grafici, ombreggiature, animazioni, eccetera.

Modello di comunicazione con l'utente

La comunicazione con l'utente può essere di tipo **ingresso/uscita** o **guidata dagli eventi** (*event-driven*).

La forma di comunicazione ingresso/uscita presuppone innanzitutto l'esistenza di metodi specializzati per l'acquisizione (ingresso) e per la visualizzazione (uscita) dei dati. Una seconda caratteristica è data dal fatto che esiste una chiara corrispondenza – una sequenzialità – tra l'ordine di esecuzione di tali metodi e l'effetto che essi producono dal punto di vista dell'utente. In altre parole, il comportamento dell'interfaccia dal punto di vista dell'utente – la sequenza di visualizzazione e di richiesta dei dati – è rigidamente codificato nelle istruzioni che implementano l'interfaccia stessa. Infine, un modello di comunicazione ingresso/uscita non gestisce il mouse.

Il modello di comunicazione guidato dagli eventi assume una forma completamente diversa. Benché esistano ovviamente dei metodi per l'acquisizione dei dati e la loro visualizzazione, la comunicazione con l'utente non avviene direttamente mediante essi ma attraverso degli **oggetti visuali** e gli **eventi** che tali oggetti sono in grado di gestire. Dunque, l'interazione con l'utente avviene attraverso singole azioni – la pressione di un tasto, il clic o lo spostamento del mouse; azioni dirette sempre verso un determinato oggetto visuale, che può essere o meno è in grado di riceverle.

Infine, gli oggetti visuali che costituiscono l'interfaccia determinano il tipo di azioni che l'utente può dirigere verso di essi, ma non il loro ordine di esecuzione, come avviene per un modello di comunicazione ingresso/uscita.

1.2 Modelli di interfaccia utente

Considerato che un'interfaccia utente è caratterizzata da due aspetti: gestione del video e modello di comunicazione con l'utente; e che per ognuno di essi esistono due forme distinte, ne consegue che sono possibili quattro modelli di interfaccia utente:

1) **a caratteri | ingresso/uscita;**
2) a caratteri | guidata dagli eventi;
3) grafica | ingresso/uscita;
4) **grafica | guidata dagli eventi.**

Storicamente parlando, ognuno di questi quattro modelli ha avuto il suo periodo di grande diffusione. Attualmente (e senz'altro anche per il prossimo futuro), il modello dominante è l'ultimo, utilizzato dalle Applicazioni Windows. Il primo modello viene comunque tuttora impiegato, soprattutto in programmi che non richiedono una vera e propria interazione con l'utente se non l'acquisizione iniziale di alcuni dati e l'eventuale successiva visualizzazione dei risultati.

E' questo il modello utilizzato dalle Applicazioni Console.

2 Interfaccia utente delle Applicazioni Console

Il modello offerto dalle Applicazioni Console è quello della "telescrivente". In esso il video è considerato alla stessa stregua di un modulo continuo cartaceo, del tutto simile a quelli utilizzati nelle tradizionali stampanti ad aghi o a margherita. Il programma, esattamente come il carrello di una stampante o di una macchina da scrivere, scrive sullo schermo da sinistra a destra e dall'alto verso il basso. In questa analogia, il ruolo del carrello è svolto dal **cursore testo**, rappresentato normalmente da una linea orizzontale lampeggiante.

Durante la visualizzazione, i caratteri vengono visualizzati sullo schermo a partire dalla posizione attuale del cursore testo, che si sposta verso destra dopo ogni carattere visualizzato. Se il cursore testo si trova sul limite destro della finestra, il successivo carattere viene visualizzato a partire dall'inizio della riga successiva. Se il cursore testo si trova nell'ultima riga visibile della finestra, il successivo ritorno a capo determinerà lo scorrimento verso l'alto di tutte le righe precedentemente visualizzate, per fare posto alla visualizzazione della nuova riga.

L'intera gestione dell'interfaccia delle Applicazioni Console è implementata dalla classe `Console`. Qui ci limiteremo a considerare le funzioni di base fornite dalla classe e cioè i metodi di visualizzazione e di acquisizione dati:

- `WriteLine()` e `Write()` per la visualizzazione;
- `ReadLine()` e `Read()` per l'acquisizione, di cui sarà trattato soltanto il primo.

> Quella descritta è una versione "povera" del modello di interfaccia effettivamente implementato dalla classe `Console`. Questa offre anche la gestione dei colori, il posizionamento assoluto del curore testo, ed altre funzionalità ancora.

2.1 Modello di comunicazione delle «Applicazioni Console»

Com'è già stato sottolineato, il modello di comunicazione ingresso/uscita implementato nelle Applicazioni Console fa sì che l'ordine con il quale vengono visualizzati e richiesti i dati, equivalga all'ordine in cui sono eseguiti i metodi `Write()`, `WriteLine()` e `ReadLine()`. Ed entro certi limiti, l'ordine in cui tali metodi sono eseguiti dipende dall'ordine in cui sono scritti nel codice sorgente.

Un esempio aiuterà a comprendere meglio il concetto. Il seguente programma risolve il problema del calcolo della soluzione di un'equazione di 1° grado, dati i coefficienti A e B.

```
class Program
{
    static void Main()
    {
        double a, b, x;
        string tmp;
        Console.Write("Immetti il valore del coefficiente A:");
        tmp = Console.ReadLine();
        a = Convert.ToDouble(tmp);

        Console.Write("Immetti il valore del coefficiente B:");
        tmp = Console.ReadLine();
        b = Convert.ToDouble(tmp);

        if (a != 0)
        {
            x = -b / a;
            Console.WriteLine("La soluzione è: {0}", x);
        }
        else       // a è uguale a zero
        {
            if (b != 0)
                Console.WriteLine("Non esiste nessuna soluzione");
            else
                Console.WriteLine("Esistono infinite soluzioni");
        }
    }
}
```

In grigio sono evidenziate le istruzioni che appartengono alla parte interfaccia del programma[31]. Il loro ordine all'interno del codice sorgente determina anche l'ordine con il quale sono richiesti i dati, visualizzati i messaggi informativi e visualizzati i risultati. L'esecuzione del programma produce dunque la sequenza predefinita di eventi:

1) viene richiesto il coefficiente `a`;
2) viene richiesto il coefficiente `b`;
3) se esiste viene calcolata e visualizzata la soluzione dell'equazione;
4) altrimenti viene visualizzato un appropriato messaggio informativo.

Tale sequenza è strutturata nel programma, e cioè: non esiste alcun modo per l'utente di inserire prima il valore di `b` e poi quello di `a`, oppure: di ricevere i risultati prima di aver inserito i valori dei coefficienti.

[31] Benché non gestiscano la comunicazione con l'utente, vengono considerate facenti della parte interfaccia anche le due invocazioni ai metodi `ToDouble()`, poiché rientrano nell'insieme di istruzioni il cui scopo è di acquisire i coefficienti `a` e `b`.

Segue l'output prodotto dal programma, ipotizzando il valore `2` per il coefficiente `a` e `-1` per il coefficiente `b`:

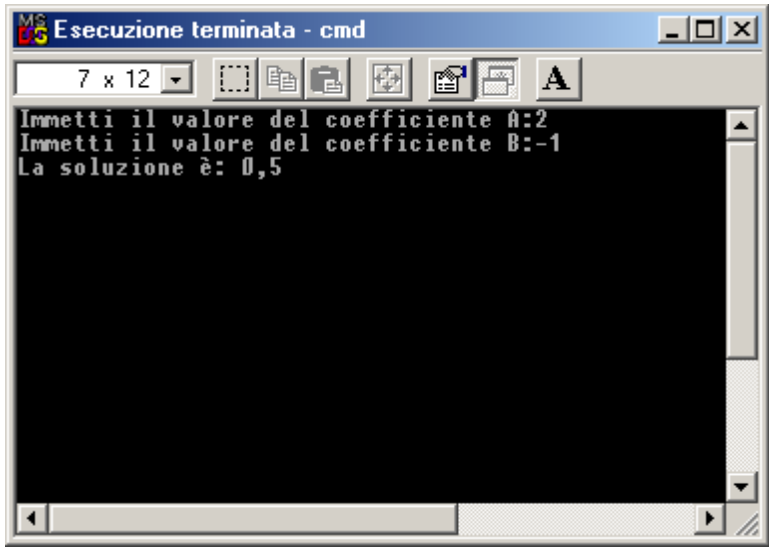


Figura 10-2 Output prodotto dall'esecuzione di PrimoGradoCon.

La sequenzialità nella richiesta dei dati è anche garantita dal fatto che l'invocazione del metodo `ReadLine()` determina la sospensione dell'esecuzione del programma, esecuzione che riprende soltanto dopo che l'utente ha premuto il tasto INVIO. In altre parole, finché l'utente non ha inserito entrambi i coefficienti, il programma non può procedere al calcolo della soluzione dell'equazione.

La rigida sequenzialità nella richiesta e nella visualizzazione dei dati ha importanti implicazioni: il programmatore è in grado, semplicemente agendo sulla struttura del programma, di imporre un preciso ordine all'interazione con l'utente. Ciò consente ad esempio di implementare con estrema facilità il tipico modello di funzionamento di una Applicazione Console, impiegato nell'esempio precedente e schematizzato nella figura sottostante:

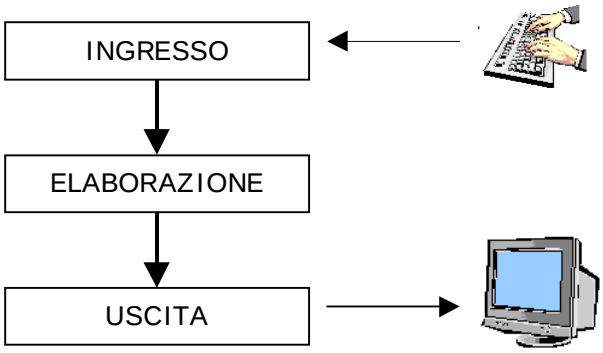


Figura 10-3 Schema di funzionamento di una tipica Applicazione Console.

Il modello è semplice perché è basato sulla struttura del programma:

- prima le istruzioni che richiedono i dati;
- poi le istruzioni che li elaborano;
- infine le istruzioni che visualizzano i risultati.

Una simile architettura garantisce che le variabili che memorizzano i dati non saranno elaborate finché l'utente non ha inserito i dati stessi. Garantisce anche che i risultati non saranno visualizzati finché non è stata eseguita la parte elaborazione. Entrambe le garanzie dipendono implicitamente

dall’organizzazione del programma, e l’unico modo perché siano violate (ad esempio che siano visualizzati dei dati prima che siano stati eseguiti i calcoli appropriati) è che il programma sia stato strutturato in modo errato.

11

Introduzione alle Applicazioni Windows

1 Struttura di una interfaccia grafica

Il modello di interfaccia **grafica | guidata dagli eventi** (d'ora in avanti semplicemente: interfaccia grafica) presenta un'architettura molto diversa da quello impiegato dalle Applicazioni Console e schematizzato in Figura 10-1. Alla base di una interfaccia grafica vi sono quattro elementi fondamentali:

- la finestra grafica – **form** – associata all'applicazione;
- gli oggetti visuali, definiti **controlli**, contenuti nella finestra grafica;
- gli **eventi** generati dai controlli;
- i metodi che gestiscono gli eventi, o **gestori di eventi**.

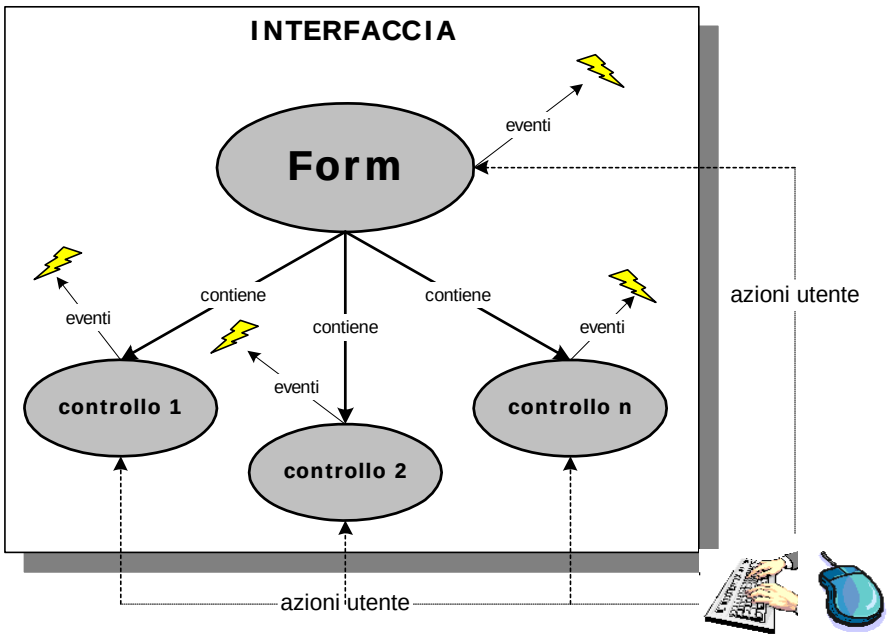


Figura 11-1 Rappresentazione schematica di una interfaccia grafica.

In questo modello, il dialogo tra l'utente e il programma non avviene mediante metodi di ingresso e di uscita dei dati, ma attraverso i controlli che costituiscono l'interfaccia. Essi sono di vari tipi, in base alla natura della comunicazione che svolgono con l'utente: acquisire una sequenza di caratteri, rispondere al clic del mouse, visualizzare un immagine, un'animazione, una serie di valori in forma di lista o di tabella, eccetera.

I controlli rispondono alle azioni dell'utente mettendo in atto dei comportamenti predefiniti e generando uno o più eventi. Ad esempio, se l'utente clicca su un bottone questo si abbassa e si alza e contemporaneamente genera l'evento `Click`. Come vedremo più avanti, a un evento può essere

associato un metodo scritto dal programmatore, chiamato **gestore di evento**. Nel momento in cui il controllo genera un evento, il gestore di evento ad esso associato viene eseguito automaticamente.

1.1 Introduzioni ai controlli

I controlli sono oggetti visuali, e cioè oggetti in grado di visualizzare se stessi sullo schermo. In effetti non esistono metodi specifici per la visualizzazione dei controlli, essi lo fanno da sé; tutto ciò che deve fare il programmatore è:

creare il controllo;

definirne l'aspetto, impostando opportunamente le sue **proprietà**;

associare dei metodi a gli eventi del controllo che si desidera gestire (i gestori di eventi);

aggiungere il controllo all'interfaccia.

Dopodiché esso sarà sempre visibile e in grado di ricevere le azioni dell'utente, almeno fin quando non sarà eliminato dall'interfaccia oppure temporaneamente reso invisibile o disabilitato.

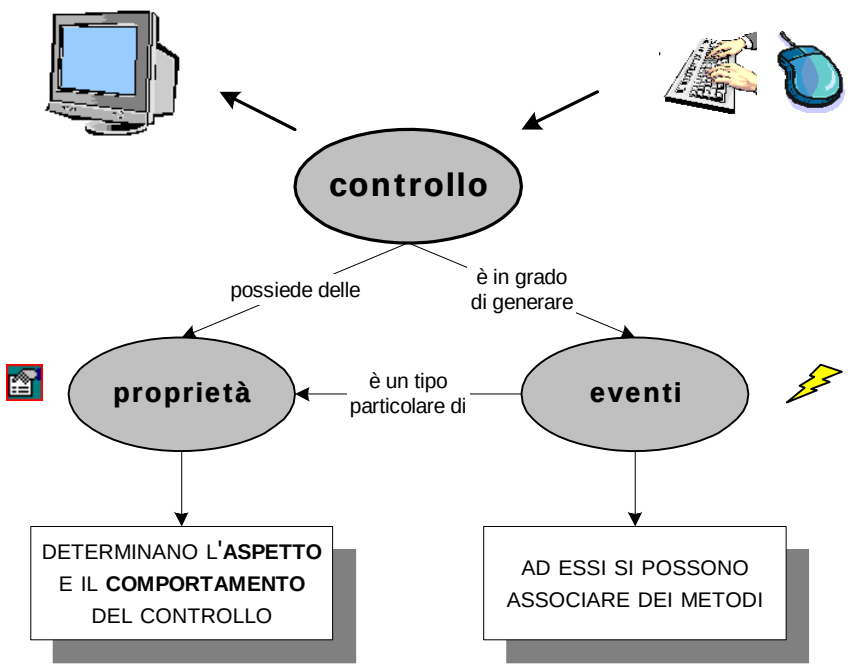


Figura 11-2 Rappresentazione schematica del rapporto tra controllo, proprietà ed eventi.

Dichiarare e creare il controllo

I controlli sono oggetti e dunque è possibile accedere ad essi attraverso delle variabili, ma soltanto dopo che sono stati creati. Ogni controllo appartiene a un determinato tipo e pertanto la dichiarazione della variabile controllo assume l'identica sintassi di qualsiasi altra dichiarazione:

```
tipo nome-controllo
```

Ad esempio, per dichiarare un bottone, e cioè un controllo della classe `Button`, si scrive:

```
Button b;
```

dove `b` è il nome del il bottone. La creazione del controllo avviene mediante l'operatore `new`:

```
nome-controllo = new tipo();
```

Ad esempio, per creare un bottone si scrive:

```
b = new Button();
```

Definire l'aspetto del controllo

L'aspetto di un controllo è rappresentato dalle sue caratteristiche visuali, come altezza, larghezza, colore, eccetera. Queste vengono definite assegnando gli opportuni valori alle sue **proprietà**. Per modificare il valore di una proprietà si usa la sintassi:

```
nome-controllo.nome-proprietà = valore
```

Ad esempio, per definire il testo di un bottone si imposta il valore della proprietà `Text`, che è di tipo stringa:

```
b.Text = "Clicca qui!";
```

Per impostare invece la posizione verticale del controllo al valore 100, occorre modificare la proprietà `Top`:

```
b.Top = 100;
```

Associare dei metodi agli eventi del controllo

Ogni controllo definisce un certo numero di eventi di varia natura, molti dei quali sono generati in risposta alle azioni dell'utente. Se ad esempio si desidera che in risposta al clic su un bottone il programma svolga un determinato compito, è necessario scrivere un metodo che implementi il compito e associarlo all'evento `Click` del bottone. Il metodo diventa dunque il gestore dell'evento `Click` di quel bottone.

L'associazione di un metodo a un determinato evento sarà descritta più avanti.

Aggiungere il controllo all'interfaccia

Creare un controllo e definirne l'aspetto non lo rende operativo, né visibile sullo schermo; occorre prima aggiungerlo all'interfaccia e cioè alla finestra all'interno della quale viene eseguito il programma. La finestra stessa è a sua volta un controllo, appartenente al tipo `Form`.

1.2 Form: il controllo principale

Ogni Applicazione Windows presenta almeno un controllo di tipo `Form`; esso è l'elemento principale dell'interfaccia, poiché è il primo ad essere creato, rappresenta la finestra associata all'applicazione e fa da contenitore a tutti gli altri controlli. Aggiungere un controllo all'interfaccia significa dunque aggiungerlo al form. Questo mantiene internamente una collezione di controlli chiamata `Controls`, la quale espone un metodo chiamato `Add()` per l'inserimento dei controlli.

Ad esempio, per aggiungere il bottone `b` al form si scrive:

```
Controls.Add(b);
```

Solo dopo che è stato inserito nella collezione `Controls`, il bottone appartiene realmente all'interfaccia ed è in grado di ricevere le azioni dell'utente.

1.3 Controlli di base di un'interfaccia: Button, TextBox, Label

Per introdurre i controlli d'uso più comune di un interfaccia grafica si farà riferimento a un programma di esempio, che risolve l'equazione di 1° grado, la cui interfaccia è rappresentata nella figura sottostante.

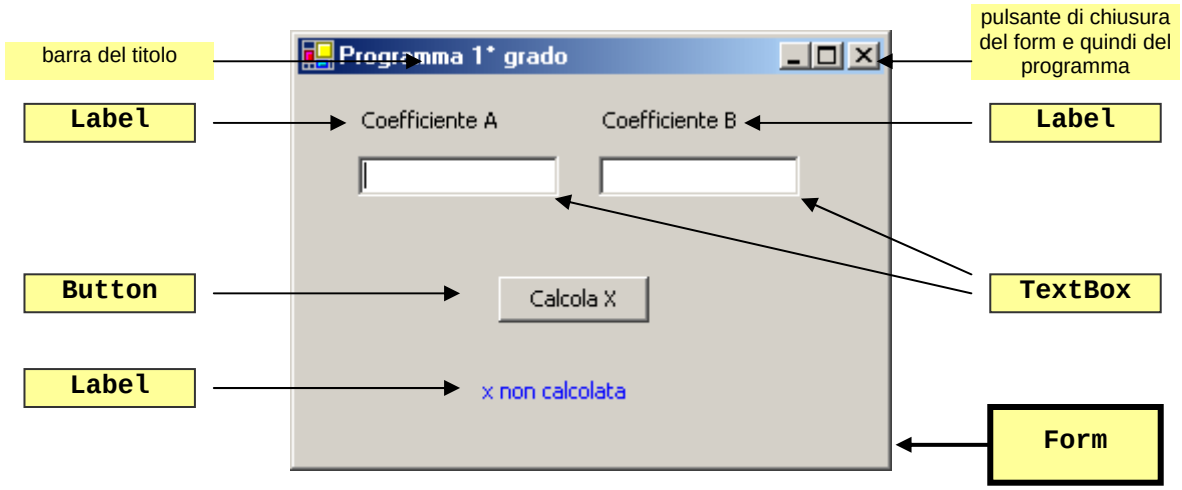


Figura 11-3 Interfaccia del programma PrimoGradoWin.

L'interfaccia del programma contiene sei controlli, oltre al form.

Controllo TextBox

I controlli di tipo `TextBox` (casella di testo) svolgono l'analoga funzione del metodo `ReadLine()` e cioè l'acquisizione di stringhe di caratteri. L'utente può digitare caratteri in un `TextBox` soltanto se possiede il **fuoco** (dentro di esso lampeggia il cursore testo, chiamato anche **caret**). Un modo molto semplice per selezionare un `TextBox` è cliccare al suo interno.

Controllo Label

I controlli di tipo `Label` (etichetta) svolgono l'analoga funzione del metodo `WriteLine()` e cioè la visualizzazione di stringhe di caratteri; per questo motivo, di solito, non devono ricevere le azioni dell'utente. Un uso tipico delle etichette è quello di qualificare i controlli `TextBox`, informando così l'utente cosa gli si richiede di inserire.

Il programma PrimoGradoWin utilizza tre etichette, di cui una per visualizzare la soluzione dell'equazione (che in figura si presuppone non ancora calcolata).

Controllo Button

I controlli di tipo `Button` (bottone) vengono impiegati per dare il via a qualche forma di elaborazione, risultato che viene ottenuto associando un metodo all'evento `Click`.

Nel programma PrimoGradoWin, il calcolo della soluzione dell'equazione, che rappresenta la parte elaborazione del programma, viene eseguito in risposta al clic dell'utente sul bottone.

1.4 Analisi del funzionamento del programma

PrimoGradoWin risolve lo stesso problema del suo omologo con interfaccia a caratteri e possiede la stessa parte elaborazione, ma si interfaccia con l'utente in modo completamente diverso. Il modello di comunicazione impiegato non presuppone un sequenza predefinita di azioni da parte dell'utente e di elaborazioni da parte del programma. L'utente è infatti libero di:

- cliccare sul bottone “Calcola X”, determinando così il calcolo della soluzione, senza aver inserito alcun valore in uno dei due `TextBox` o in entrambi;
- inserire prima il valore del coefficiente B e poi quello del coefficiente A, o viceversa;
- inserire i due valori dei coefficienti, ma non eseguire il calcolo della soluzione, cliccando invece sul pulsante di chiusura del form;
- calcolare la soluzione di quante equazioni desidera, modificando ogni volta i valori dei due coefficienti e cliccando nuovamente sul bottone “Calcola X”.

La parte elaborazione del programma viene eseguita se e quando l’utente clicca sul bottone. Nell’analogo programma con interfaccia a caratteri, invece, l’elaborazione è garantita dalla struttura del programma: il flusso di esecuzione, per raggiungere l’ultima istruzione del metodo `Main()` *deve per forza passare per la parte elaborazione!*

Anche in un’Applicazione Windows, dunque, l’elaborazione avviene in risposta alle azioni dell’utente, ma queste ultime non sono predeterminate dalla struttura del programma. Ciò, come vedremo, impone alcuni accorgimenti nella progettazione dell’interfaccia.

2 Struttura di una Applicazione Windows

Tutte le Applicazioni Windows condividono una struttura generale identica, che le distingue dalle Applicazioni Console. Questa non è completamente rigida e può differenziarsi in base alle scelte del programmatore e, soprattutto, del programma di sviluppo (IDE) utilizzato per scrivere il codice.

In questo testo viene fatto riferimento alla struttura generata dal software Visual C# Express, peraltro identica a quella adottata dal fratello maggior Visual Studio. Entrambi sono ambienti di sviluppo RAD (Rapid Development Environment) e mediante il cosiddetto **designer** consentono al programmatore di realizzare in modo visuale l’interfaccia, producendo automaticamente il codice necessario.

Esamineremo la struttura prodotta da Visual C# Express a fine capitolo e la utilizzeremo nel resto del volume. Nei successivi paragrafi, invece, adotteremo una versione semplificata allo scopo di chiarire il modello di funzionamento di un’interfaccia grafica.

2.1 Scheletro di una Applicazione Windows

Prendendo come riferimento il programma PrimoGradoWin, sarà costruita una Applicazione Windows che presenti una interfaccia del tutto analoga, partendo da uno “scheletro” minimale, ma funzionante, e aggiungendo via via i vari elementi dell’interfaccia. Il nome dell’applicazione è MioPrimoGrado. Ecco lo scheletro iniziale del programma.

```
using System;
using System.Windows.Forms;
using System.Drawing;

class MainForm: Form
{
    public MainForm()
    {
    }
```

```
    public static void Main()
    {
        Application.Run( new MainForm() );
    }
}
```

Esempio 11-1 Scheletro di base di una Applicazione Windows.

L'esecuzione di tale programma produce un form vuoto:

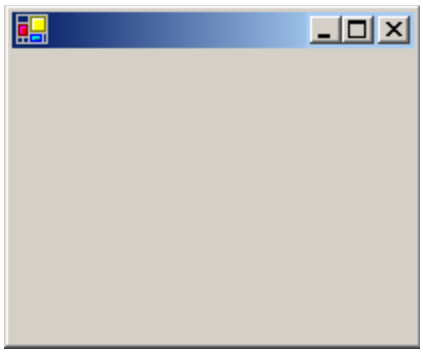


Figura 11-4 Output prodotto dall'esecuzione di una Applicazione Windows vuota.

Segue una breve introduzione ai singoli elementi del programma.

```
using System.Windows.Forms;
```

Il *namespace* `System.Windows.Forms` contiene le classi alle quali appartengono i controlli `Form`, `TextBox`, `Label`, `Button`, eccetera.

```
using System.Drawing;
```

Il *namespace* `System.Drawing` definisce tipi di dati e metodi spesso impiegati nelle Applicazioni Windows.

```
class MainForm: Form { }
```

`MainForm` è il nome della classe principale del programma, e può essere scelto dal programmatore. La notazione "**`: Form`**" (simbolo due-punti seguito dal nome `Form`) indica che la classe `MainForm` deriva dalla classe `Form`. Il concetto di derivazione di una classe rappresenta un concetto centrale della programmazione object oriented e per quanto attiene le Applicazioni Windows sarà affrontato nel prossimo capitolo; per il momento ne forniremo una breve spiegazione. Derivare una classe da un'altra significa:

definire un nuovo tipo di dato[32] che mantiene, ed eventualmente estende, le caratteristiche del tipo originale da cui deriva.

Derivare una classe dalla classe `Form` significa dunque definire un nuovo tipo di form che presenta tutte le caratteristiche del tipo originale, più eventualmente altre che è il programmatore a definire.

```
public MainForm() { }
```

`MainForm()` rappresenta il **costruttore** della classe `MainForm`. Un costruttore è un metodo molto speciale, che ha lo stesso nome della classe alla quale appartiene e:

[32] I termini classe e tipo sono sostanzialmente equivalenti.

viene invocato automaticamente nel momento in cui viene creato un oggetto di quella classe.

Le istruzioni contenute in `MainForm()` vengono dunque eseguite automaticamente ogni qual volta viene creato un oggetto della classe.

> Si sorvoli sulla parola chiave `public`, la quale non è comunque una prerogativa del costruttore `MainForm()`, in quanto può essere applicata a qualsiasi metodo e campo di una classe, come dimostra la sua applicazione al metodo `Main()`.

```
Application.Run( new MainForm() );
```

Per renderla meglio comprensibile, questa istruzione può essere scomposta in due istruzioni distinte che producono lo stesso effetto:

```
MainForm formPrincipale = new MainForm();
Application.Run(formPrincipale);
```

che è quello di creare un oggetto della classe `MainForm` e di passarlo come argomento al metodo `Run()` della classe `Application`. Il metodo `Run()` svolge la funzione di dare l'avvio vero e proprio al programma; esso visualizza infatti il form sullo schermo, consentendo dall'utente di cominciare a interagire con il programma. Prima di questo, però, il form principale dev'essere creato. Ciò viene fatto, come per ogni oggetto, attraverso l'operatore `new`.

La chiusura del `form` principale, che avviene in risposta al clic sul pulsante di chiusura, determina anche la fine dell'esecuzione del metodo `Run()` e dunque dell'intero programma, poiché l'invocazione di tale metodo è l'unica istruzione contenuta nel metodo `Main()`.

2.2 Conclusioni

Lo scheletro di una Applicazione Windows presenta tre sostanziali novità rispetto alle Applicazioni Console:

- la classe principale deriva dalla classe `Form`;
- la classe principale definisce un metodo speciale, chiamato costruttore, che viene invocato automaticamente quando viene creato il `form`;
- il metodo `Main()` contiene una sola istruzione, nella quale viene creato un form, il cui riferimento viene passato come argomento al metodo `Run()` della classe `Application`. Di fatto, è tale metodo che determina l'esecuzione vera e propria del programma.

Non è necessario (e sarebbe impossibile) comprendere a fondo gli elementi presentati; ciò che importa è comprendere che una Applicazione Windows, al di là della sua complessità, presenta comunque una struttura di fondo equivalente a quella appena introdotta.

3 Costruire l'interfaccia

Il precedente programma funziona, ma non è in grado di svolgere alcun compito, poiché presenta un'interfaccia completamente vuota. Occorre popolarla dei controlli appropriati.

3.1 Dichiarare i controlli

Perché un controllo faccia parte dell'interfaccia occorre che sia dichiarata una variabile del tipo appropriato, sia creato il controllo e sia aggiunto al form. L'interfaccia del programma PrimoGradoWin presenta sei controlli e quindi richiede la dichiarazione di sei variabili, le quali dovranno essere dichiarate come come campi di classe.

Segue lo scheletro del programma con l'aggiunta delle necessarie dichiarazioni (d'ora in avanti saranno omesse le direttive `using`):

```
class MainForm: Form
{
    Button btnCalcolaX; // bottone che determina il calcolo della soluzione X
    Label lblA;         // etichetta informativa associata al TextBox txtA
    Label lblB;         // etichetta informativa associata al TextBox txtA
    Label lblX;         // etichetta che visualizza la soluzione X
    TextBox txtA;       // TextBox per l'acquisizione del coefficiente A
    TextBox txtB;       // TextBox per l'acquisizione del coefficiente B

    public MainForm()
    {
    }

    public static void Main()
    {
        Application.Run( new MainForm() );
    }
}
```

> Nota bene. I nomi dei controlli presentano un suffisso che ne ricorda il tipo: **`lbl`** per `Label`; **`btn`** per `Button`; ecc. E' una pratica molto comune quella di utilizzare dei suffissi nei nomi degli oggetti visuali (e non solo) messi a disposizione da .NET.
>
> Inoltre, le dichiarazioni non sono precedute dalla parola chiave `static`. Si rimanda al volume sulla programmazione orientata agli oggetti per una spiegazione sul significato della parola chiave `static`.

3.2 Creazione, impostazione e inserimento dei controlli all'interfaccia

Nel momento in cui il form viene visualizzato l'interfaccia dev'essere già completa; dunque, il luogo più ovvio nel quale collocare le istruzioni che creano, impostano e inseriscono i controlli è il costruttore. Sempre nel costruttore viene impostato il testo che fa da titolo al programma, testo che compare nella barra del titolo del form.

Allo scopo di procedere in modo graduale, segue l'implementazione del solo codice che definisce il testo della barra del titolo e crea e imposta la `Label` informativa e il `TextBox` relativi al coefficiente A.

```
public MainForm()
{
    Text = "Programma 1° grado";    // imposta il testo della «barra del titolo»

    // crea e imposta le proprietà dell'etichetta lblA
    lblA = new Label();
```

```
    lblA.Text = "Coefficiente A";
    lblA.Top = 15;
    lblA.Left = 30;

    Controls.Add(lblA);       // aggiunge lblA al form

    // crea e imposta le proprietà del TextBox txtA
    txtA = new TextBox();
    txtA.Top = 40;
    txtA.Left = 30;

    Controls.Add(txtA);       // aggiunge txtA al form

...
}
```

L'esecuzione del programma produce adesso il seguente output:

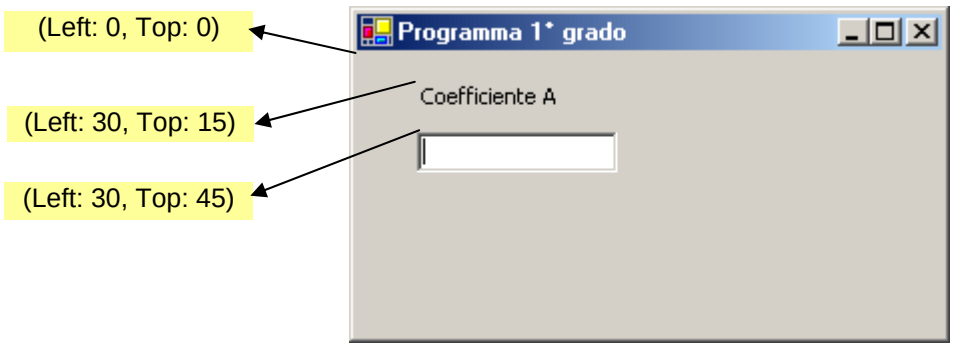


Figura 11-5 Output prodotto dal programma.

I due controlli, `lblA` e `txtA`, vengono creati e inseriti nel form in posizioni dalle coordinate ben precise, espresse in pixel. Esistono vari modi per impostare le coordinate di un controllo; quello usato nel codice imposta le proprietà `Left` e `Top`, rispettivamente l'ascissa e l'ordinata in un sistema di coordinate che ha la propria origine nell'angolo in alto a sinistra dell'area grigia del form. Infine, alla proprietà `Text` del controllo `lblA` viene assegnato l'appropriato testo informativo.

Fin d'ora è importante notare che:

- alcune proprietà, come ad esempio `Left`, `Top` e `Text`, sono possedute da più tipi di controlli;
- l'accesso alle proprietà di un controllo si ottiene sempre attraverso la notazione:

  `nome-controllo.nome-proprietà`

- quando il nome del controllo non viene specificato, la proprietà si intende appartenere al form. Infatti, l'istruzione:

  `Text = "Programma 1° grado";`

- determina l'assegnazione del costante stringa `"Programma 1° grado"` alla proprietà `Text` del form, il che equivale a impostare il testo della barra del titolo.

Completamento dell'interfaccia

I passi descritti devono essere ripetuti anche per gli altri controlli. Segue il frammento di codice da aggiungere al costruttore per il completamento dell'interfaccia.

```
...
// crea e imposta le proprietà dell'etichetta lblB
lblB = new Label();
lblB.Text = "Coefficiente B";
lblB.Top = 15;
lblB.Left = 150;
Controls.Add(lblB);

// crea e imposta le proprietà del TextBox txtB
txtB = new TextBox();
txtB.Top = 40;
txtB.Left = 150;
Controls.Add(txtB);

// crea e imposta le proprietà del bottone btnCalcolaX
btnCalcola = new Button();
btnCalcola.Text = "Calcola X";
btnCalcola.Top = 100;
btnCalcola.Left = 100;
Controls.Add(btnCalcolaX);

// crea e imposta le proprietà dell'etichetta lblX
lblX = new Label();
lblX.Text = "x non calcolata";
lblX.Top = 150;
lblX.Left = 90;
lblX.ForeColor = Color.Blue;
lblX.AutoSize = true;
Controls.Add(lblX);
...
```

Non c'è molto da commentare, se non le due istruzioni evidenziate in grigio, nelle quali vengono introdotte due nuove proprietà: `ForeColor` e `AutoSize`. La prima (che sta per *foreground color*) definisce il colore del testo di un controllo. Esiste un elenco di identificatori predefiniti per i colori, appartenenti al tipo `Color`.

> .NET definisce un insieme di proprietà statiche che identificano alcuni colori di base. Per accedere ad esse è necessario premettere il nome del tipo `Color`. Ad esempio: `Color.Red`, `Color.Green`, `Color.White`, eccetera.

La proprietà `AutoSize` fa sì che le dimensioni dell'etichetta varino automaticamente in relazione alla grandezza del testo da visualizzare.

Con l'aggiunta del precedente codice, l'esecuzione del programma produce un form identico a quello rappresentato in Figura 11-3. L'interfaccia è completa e funzionante, in grado di ricevere le

azioni dell'utente. Resta il fatto che non è stata ancora programmata alcuna risposta a tali azioni, nella fattispecie all'unica azione che il programma deve elaborare e cioè il clic sul bottone `btnCalcolaX`.

Occorre dunque "collegare" la parte interfaccia con la parte elaborazione del programma, ancora da scrivere. Ciò lo si ottiene scrivendo un gestore dell'evento `Click` sollevato dal bottone `btnCalcolaX`.

4 Eventi e gestori di eventi

Il concetto di evento acquista significati distinti, anche se collegati , in base alla prospettiva dalla quale si considera tale termine. Esiste infatti l'evento inteso come "qualcosa che accade", ed è il significato comune e generico che viene attribuito ad esso normalmente. Esiste poi l'evento inteso come elemento del linguaggio e si riferisce a "una notifica di qualcosa che è accaduto" o che "sta per accadere" all'interno del programma in esecuzione.

Nelle prossime righe faremo riferimento alla prima definizione di evento con il termine **evento reale**, mentre faremo riferimento alla seconda con il termine **evento C#**.

4.1 Meccanismo di generazione degli eventi

Per evento reale si intende qualcosa che accade ed è potenzialmente in grado di influenzare lo stato di esecuzione del programma. Ad esempio, la pressione di un tasto, della tastiera o del mouse, è un evento reale, come lo è lo spostamento del mouse. Ma gli eventi reali non corrispondono necessariamente in modo diretto alle azioni dell'utente. La visualizzazione del form (apertura) corrisponde a un evento reale che si verifica dopo che l'utente ha eseguito il programma. Analogamente, la chiusura del form è un evento reale che segue il clic dell'utente sul pulsante di chiusura.

Un evento reale è dunque qualcosa che viene rilevato dall'interfaccia, più precisamente da uno o più controlli. Se si considerano solo gli eventi reali che corrispondono direttamente alle azioni dell'utente, essi possono essere così schematizzati:

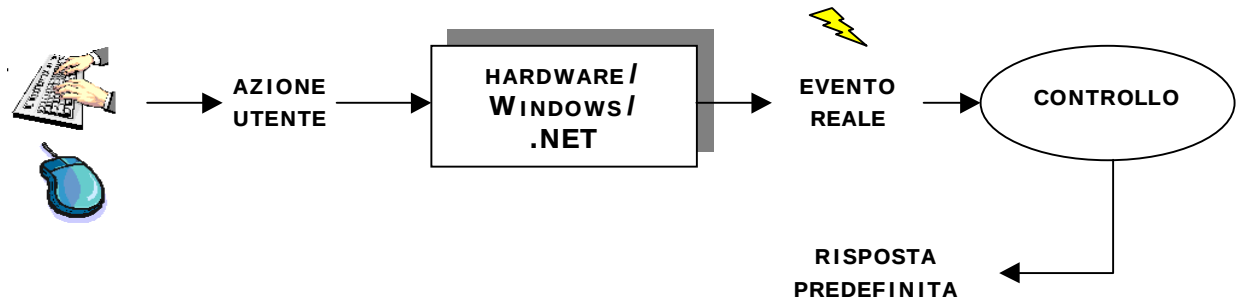


Figura 11-6 Schematizzazione di un evento reale.

L'azione dell'utente viene elaborata dall'hardware, da Windows e da .NET per essere tradotta in un evento reale, ricevuto dal controllo verso il quale è diretta.

I controlli mettono in atto dei comportamenti predefiniti in risposta agli eventi reali, o semplicemente non rispondono affatto. Se ad esempio si clicca in un punto qualsiasi del form non si ottiene nessuna forma di risposta; la stessa azione diretta verso un bottone produce l'effetto ottico di abbassamento e successivo innalzamento del bottone. Se si preme una lettera mentre il cursore testo si trova in un `TextBox`, la lettera viene inserita nel testo. Se si clicca sul pulsante di chiusura del form, questo risponde chiudendosi; la stessa risposta viene ottenuta se l'utente digita la combinazione `ALT-F4`.

In conclusione, ad ogni evento reale corrisponde una risposta – anche nulla – da parte dei controlli che lo ricevono; tale risposta è chiamata predefinita poiché i controlli sono in grado di metterla in atto senza che il programmatore debba scrivere alcuna riga di codice.

Mentre un «evento reale» è *ricevuto*, un evento C# è *generato* dai controlli; il termine tecnico è **sollevato**[33], o **notificato**. Lo scopo degli eventi C# è quello di consentire al programmatore di scrivere le risposte del programma agli eventi reali, in aggiunta, o in alternativa, alle risposte predefinite messe in atto dai controlli.

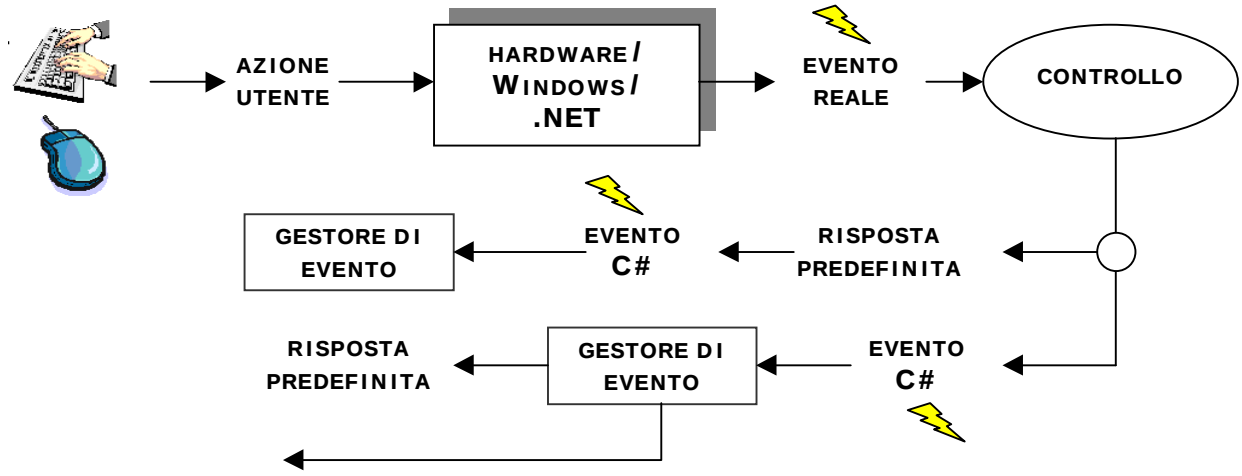


Figura 11-7 Schematizzazione di un «evento reale» e conseguente evento C#.

Lo schema mostra che dopo aver ricevuto l'evento reale, il controllo può:

- mettere comunque in atto la risposta predefinita e poi sollevare l'evento C#;
- sollevare l'evento C# prima di mettere in atto la risposta predefinita;

E' importante comprendere che un evento C# è soltanto qualcosa di potenziale, che di per sé non produce alcunché. Un evento C# rappresenta un meccanismo che consente al programmatore di "agganciare" il proprio codice agli eventi reali, ma solo se lo desidera. Un evento C# rappresenta dunque un meccanismo di notifica: il controllo notifica che si è verificato un evento reale e delega il gestore di evento per mettere in atto la risposta stabilita dal programmatore. Come mostra lo schema, in base al tipo di evento C#, il gestore di evento è in grado di influenzare, o addirittura cancellare, la risposta predefinita del controllo.

Ciò detto, d'ora in avanti, se non diversamente specificato, con il termine evento sarà designato un evento C#.

4.2 Gestire gli eventi di un programma

Per realizzare un programma equivalente a PrimoGradoWin l'unico evento da gestire è l'evento `Click` sollevato dal bottone `btnCalcolaX`. In risposta a tale evento occorre:

1) acquisire i coefficienti A e B, inseriti nei due `TextBox`;
2) se possibile, calcolare la soluzione X;
3) visualizzare la soluzione X, o l'appropriato testo informativo se X non è calcolabile, impostando il testo dell'etichetta `lblX`.

[33] Con il termine sollevare non si intende qui alzare, portare in altro, ma innescare, dare l'avvio.

E' necessario scrivere un metodo che implementi i compiti sopra elencati e associare tale metodo all'evento `Click` di `btnCalcolaX`. Dopo che il metodo è stato attaccato all'evento, il clic del mouse sopra il bottone ne determina automaticamente l'esecuzione.

4.3 Prototipo di un gestore di evento

Un gestore di evento è un metodo che viene eseguito automaticamente in risposta all'evento al quale è attaccato; per questo motivo il suo prototipo deve rispettare la seguente forma:[34]

```
void nome-metodo (object sender, Classe-informazioni-evento e)
```

Il metodo deve cioè definire due parametri e non ritornare alcun valore.

Parametri di un gestore di evento

`object sender`.

Il parametro `sender` (che sta per **mandante**) rappresenta un riferimento al controllo che ha sollevato l'evento. Per il momento tale parametro può essere tranquillamente ignorato.

`tipo-informazioni-evento` `e`.

Sappiamo che un evento rappresenta la risposta di un controllo a un evento reale che ha ricevuto. In alcuni casi, la semplice notifica è sufficiente. E' questo il caso dell'evento `Click`: non importa conoscere altre informazioni se non quella che l'utente ha cliccato con il mouse sul controllo. In altre situazioni, comunque, all'evento si accompagnano necessariamente informazioni aggiuntive. Ad esempio, se, mentre il cursore si trova su un `TextBox`, l'utente preme un tasto, viene generato l'evento `KeyPress`. Tale evento non implica la sola notifica che un tasto è stato premuto, ma anche l'informazione *di quale tasto è stato premuto*.

Ebbene, le informazioni che qualificano ulteriormente l'evento sono contenute nel parametro `e`, il cui tipo dipende dal tipo dell'evento gestito. Nel caso in cui l'evento sia di sola notifica, non necessiti cioè di informazioni aggiuntive, il tipo in questione è `EventArgs`. In questo caso il parametro `e` può essere ignorato.

4.4 Gestire l'evento Click del bottone

Segue l'implementazione del metodo che gestisce l'evento `Click` sul bottone `btnCalcolaX`.

```
void btnCalcola_Click(object sender, EventArgs e)
{
    // acquisisce i coefficienti dai due TextBox
    double a = Convert.ToDouble(txtA.Text);
    double b = Convert.ToDouble(txtB.Text);
    double x;
    if (a != 0)
    {
        x = -b / a;
        lblX.Text = "La soluzione è: " + x;
    }
```

[34] I nomi «`sender`» ed «`e`» sono arbitrari (nomi diversi andrebbero altrettanto bene) ma non casuali. Se un gestore di evento viene creato attraverso i comandi disponibili in Visual Studio.NET, è con questi identificatori che vengono denominati i due parametri.

```
    else       // a è uguale a zero
    {
        if (b != 0)
            lblX.Text = "Non esiste nessuna soluzione";
        else
            lblX.Text = "Esistono infinite soluzioni";
    }
}
```

Rispetto alla versione console del programma, la parte elaborazione è rimasta praticamente immutata; le uniche modifiche riguardano le istruzioni (evidenziate in grigio) attraverso le quali vengono acquisiti i dati e visualizzati i risultati.

Nota bene. Il nome del metodo, `btnCalcolaX_Click()`, è frutto di una scelta arbitraria ma non casuale. Esso informa che il metodo gestisce l'evento `Click` sollevato dal bottone `btnCalcolaX`. La forma utilizzata per nominare il metodo segue lo stile impiegato da Visual Studio:

nome-controllo_nome-evento

4.5 Attaccare un gestore di evento a un evento

L'interfaccia è completa, la parte elaborazione anche, ora non resta che collegarle insieme. Ciò si ottiene fornendo al metodo `btnCalcolaX_Click()` la "delega" per gestire l'evento `Click` sul bottone. L'istruzione necessaria è la seguente:

```
btnCalcolaX.Click += btnCalcolaX_Click;
```

Essa produce il risultato di delegare il metodo `btnCalcolaX_Click()` a rispondere all'evento `Click` sollevato dal controllo `btnCalcolaX`.

In realtà, quella presentata è una forma compatta dell'istruzione completa:

```
btnCalcolaX.Click += new EventHandler(btnCalcolaX_Click);
```

dove `EventHandler` è il tipo di delega, più precisamente, in inglese, **delegate**. Infatti, eventi di natura diversa richiedono delegate di tipo diverso. `EventHandler` è tipo di delegate appropriato per la gestione di eventi di sola di sola notifica, ma tipi di eventi specializzati nella rappresentazione di derminati eventi reali richiedono altri tipi di delegate. Ad esempio, l'evento `KeyPress` rappresenta la pressione di un tasto e richiede un delegate di tipo `KeyPressEventHandler`.

La precedente istruzione, sia nella forma compatta che nella forma completa produce un risultato che può essere così schematizzato:

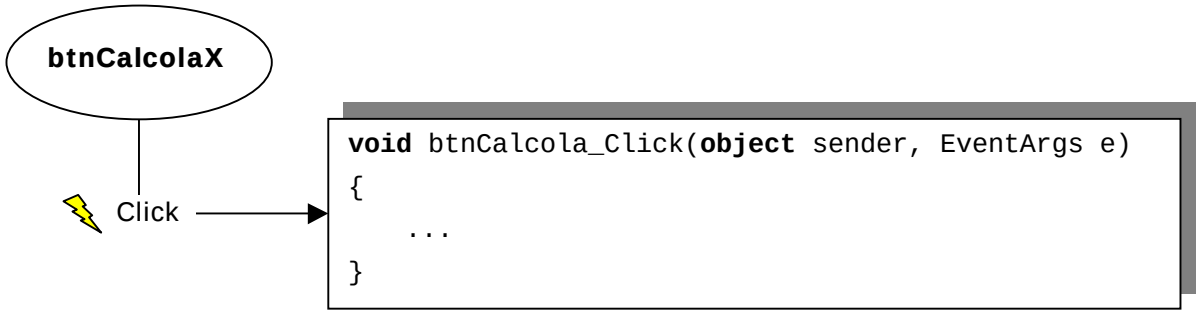


Figura 11-8 Schema associazione tra metodo ed evento.

E' importante comprendere che nell'istruzione:

controllo.evento += **new** *tipo-delegate*(*nome-metodo*);

vi dev'essere corrispondenza tra tipo di evento, tipo di delegate e prototipo del metodo, altrimenti il compilatore segnalerà un errore. D'altra parte, il linguaggio consente di utilizzare la forma compatta, desumendo da solo il tipo di delegate necessario, il quale può dunque essere omesso.

[MioPrimoGrado.cs»**]**

5 Applicazioni Windows con Visual C# Express

Nei paragrafi precedenti abbiamo mostrato la struttura generale di una Applicazione Windows ed esempio introduttivo del codice necessario per costruire l'interfaccia. Quest'ultimo implica la dichiarazione dei controlli, la loro creazione, l'impostazione del loro aspetto e infine l'associazione dei metodi agli eventi che si intendono gestire. Con Visual C# Exspress, il processo appena descritto può essere svolto automaticamente dall'IDE, senza che il programmatore debba scrivere una sola riga di codice.

L'analisi delle funzionalità di Visual C# Espress va oltre lo scopo di questo testo; qui ci limiteremo a esaminare la struttura del codice prodotta dal suddetto IDE e le funzionalità di base del designer.

5.1 Struttura di un progetto "Applicazione Windows"

Dopo aver creato un nuovo progetto di tipo Applicazione Windows, l'IDE si presenta nel seguento modo:

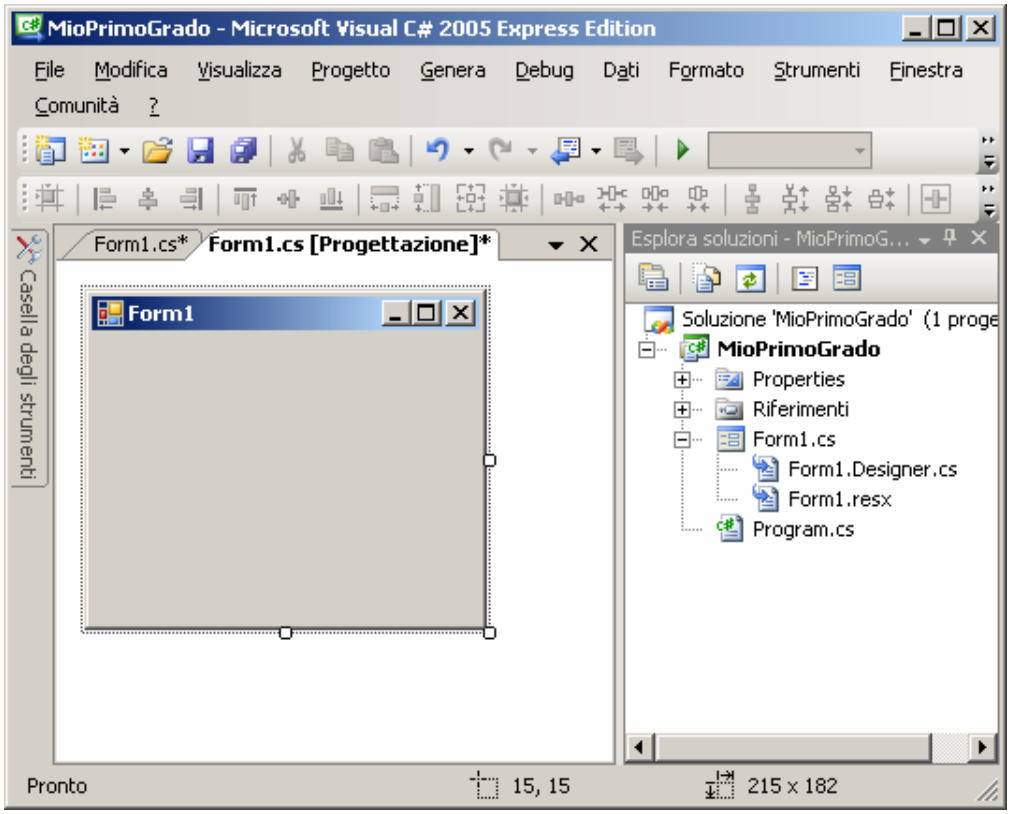


Figura 11-9 IDE di Visual C# Express dopo la creazione di un'Applicazione Windows.

Il progetto è composto da diversi file, mostrati nel pannello "Esplora soluzioni", tre dei quali contengono il codice sorgente, codice che negli esempi prevenenti era riunito in un unico file.

File “Program.cs”

Questo file definisce la classe `Program`. Analogamente alle Applicazioni Console, questa contiene il metodo `Main()`, all’interno del quale è collocato il codice di avvio dell’applicazione. Segue il contenuto del file, privo dei commenti:

```
using System;
using System.Collections.Generic;
using System.Windows.Forms;

namespace MioPrimoGrado
{
    static class Program
    {
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new Form1());
        }
    }
}
```

Si sorvoli sugli altri elementi del codice; l’unico importante è l’istruzione evidenziata, che avvia l’applicazione dopo aver creato il form principale.

Il file Program.cs è importante per l’applicazione ma praticamente irrilevante per il programmatore, e pertanto può essere tranquillamente ignorato.

Form1.cs

Form1.cs contiene il codice scritto dal programmatore. E’ qui che il programmatore scrive i metodi di gestione degli eventi e, in generale, tutta il codice necessario per soddisfare i requisiti del programma[35]. Segue il contenuto completo del file:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace MioPrimoGrado
{
    public partial class Form1: Form
```

[35] Naturalmente questo discorso vale in relazione alle semplici applicazioni che stiamo considerando. Le applicazioni realistiche sono di solito composte da più form e la logica applicativa è in genere suddivisa in svariati file.

```
    {
        public Form1()
        {
            InitializeComponent();
        }
    }
}
```

Due sono le righe di codice interessanti. La prima è l'intestazioned ella classe `Form1`, che identifica il tipo del form principale. Si noti l'uso della parola chiave `partial`: essa sta ad indicare che parte del codice appartenente alla classe `Form1` risiede in un altro file. Ciò è dimostrato dalla seconda riga evidenziata, nella quale viene invocato il metodo `InitializeComponent()`. Tale metodo appartiene sempre alla classe `Form1`, ma è definito in altro file.

Form1.Designer.cs

Questo file contiene tutto il codice necessario al funzionamento dell'interfaccia. Rientrano in questo codice le dichiarazione dei controlli, la loro creazione, l'impostazione delle loro proprietà visuali e le istruzioni che associano i metodi scritti dal programmatore agli eventi sollevati dai controlli.

Segue il contenuto del file, dal quale sono stati omessi gli elementi non significativi:

```
namespace MioPrimoGrado
{
    partial class MainForm
    {
        private System.ComponentModel.IContainer components = null;
        protected override void Dispose(bool disposing)
        {
            ...
        }

        private void InitializeComponent()
        {
            this.SuspendLayout();

            this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
            this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
            this.ClientSize = new System.Drawing.Size(207, 155);
            this.Name = "Form1";
            this.Text = "Form1";

            this.ResumeLayout(false);

        }
    }
}
```

Da notare innanzitutto l'uso della parola `partial` nell'intestazione della classe `Form1`: essa indica che il codice appartenente alla classe completa quello definito in file Form1.cs.

Il file è gestito automaticamente dall'IDE, più precisamente dal **designer**, e dunque non dovrebbe mai essere modificato dal programmatore. L'elemento fondamentale è rappresentato dal metodo `InitializeComponent()`: contiene il codice necessario al funzionamento dell'interfaccia e viene costantemente aggiornato dal designer ogniqualvolta il programmatore aggiunge e rimuove controlli o ne modifica l'aspetto.

5.2 Realizzare l'interfaccia con il designer

Il designer svolge la tipica funzione di uno strumento RAD: consente al programmatore di progettare in modo visuale una determinata funzionalità dell'applicazione (l'interfaccia, in questo caso), generando automaticamente il codice necessario per implementarla.

Il designer fornisce tre strumenti:

- una superfice di disegno che rappresenta il form (finestra progettazione form);
- una "tavolozza" dei controlli disponibili (casella degli strumenti);
- un editor delle proprietà dei controlli.

Dopo aver selezionato un controllo nella tavolozza è possibile disegnarlo sul form con il mouse, stabilendo sia la posizione che le dimensioni. Successivamente, mediante l'editor delle proprietà, è possibile sia definire l'aspetto visuale del controllo che stabilire quali eventi si intendono gestire.

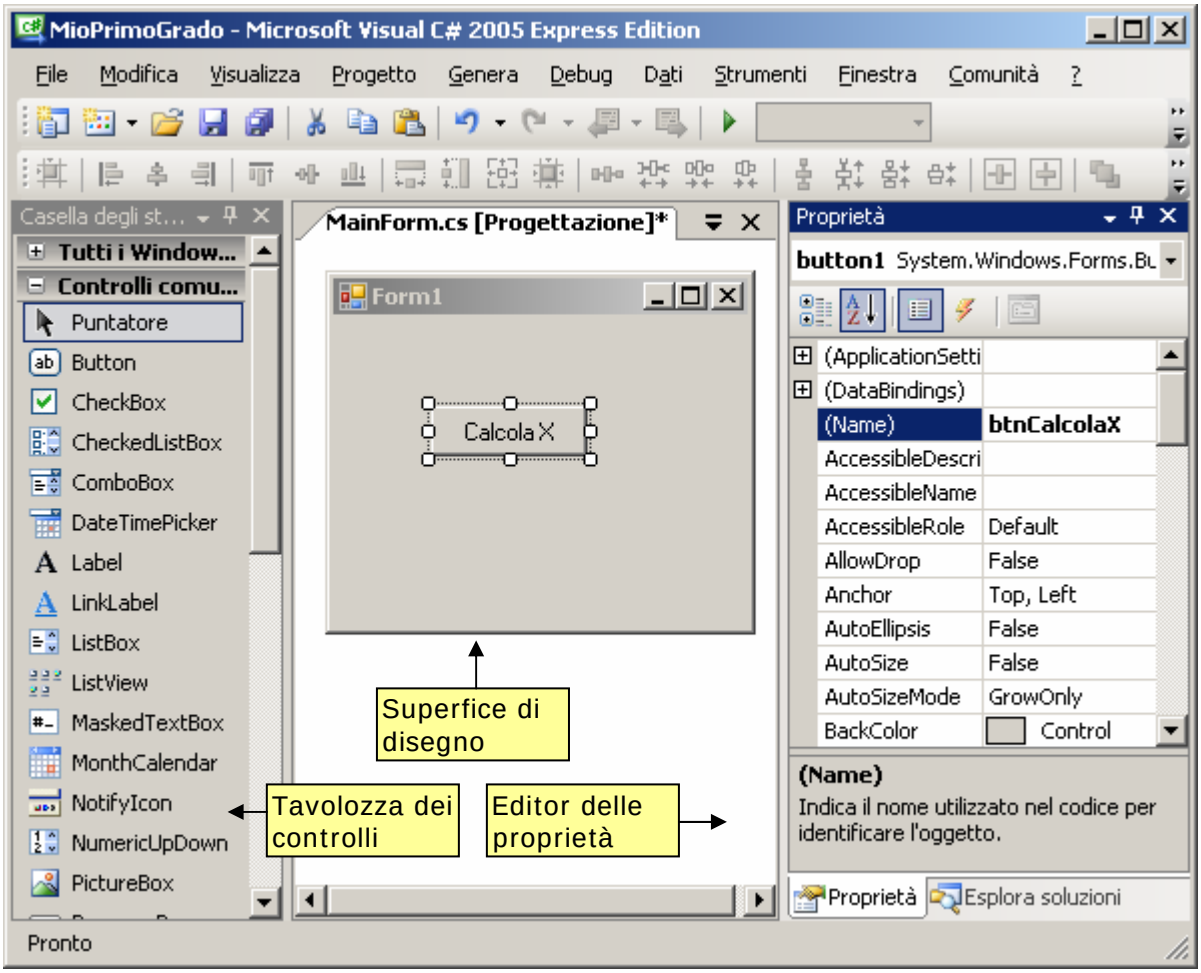


Figura 11-10 I tre strumenti del designer.

La figura mostra gli strumenti del designer e ipotizza che sia già stato aggiunto un `Button` al form; del bottone sono state impostate le proprietà `Name` e `Text`.

L'operazione di disegno del bottone viene tradotta nelle istruzioni equivalenti, le quali sono collocate all'interno del metodo `InitializeComponent()`:

```
private void InitializeComponent()
{
    this.btnCalcolaX = new System.Windows.Forms.Button();
    this.SuspendLayout();
    //
    // btnCalcolaX
    //
    this.btnCalcolaX.Location = new System.Drawing.Point(49, 45);
    this.btnCalcolaX.Name = "btnCalcolaX";
    this.btnCalcolaX.Size = new System.Drawing.Size(78, 25);
    this.btnCalcolaX.TabIndex = 0;
    this.btnCalcolaX.Text = "Calcola X";
    this.btnCalcolaX.UseVisualStyleBackColor = true;
...
}
```

Naturalmente successive modifiche alle proprietà del bottone determinano l'aggiornamento automatico del file Form.Designer.cs e di `InitializeComponent()`.

Dato che è il designer a gestire in automatico il file Form.Designer.cs, il programmatore non dovrebbe mai intervenire su di esso. La modifica manuale di questo file oltre che inutile potrebbe anche essere dannosa, a tal punto da corromperne la struttura e impedire al designer di svolgere correttamente il proprio lavoro.

In conclusione, il designer automatizza e velocizza un processo di scrittura del codice che in assenza di un ambiente di sviluppo RAD sarebbe il programmatore a dover scrivere.

5.3 Gestire un evento

L'IDE fornisce il proprio supporto anche nella gestione degli eventi. Dopo aver selezionato un controllo, mediante l'editor delle proprietà è possibile accedere all'elenco degli eventi che questo è in grado di generare.

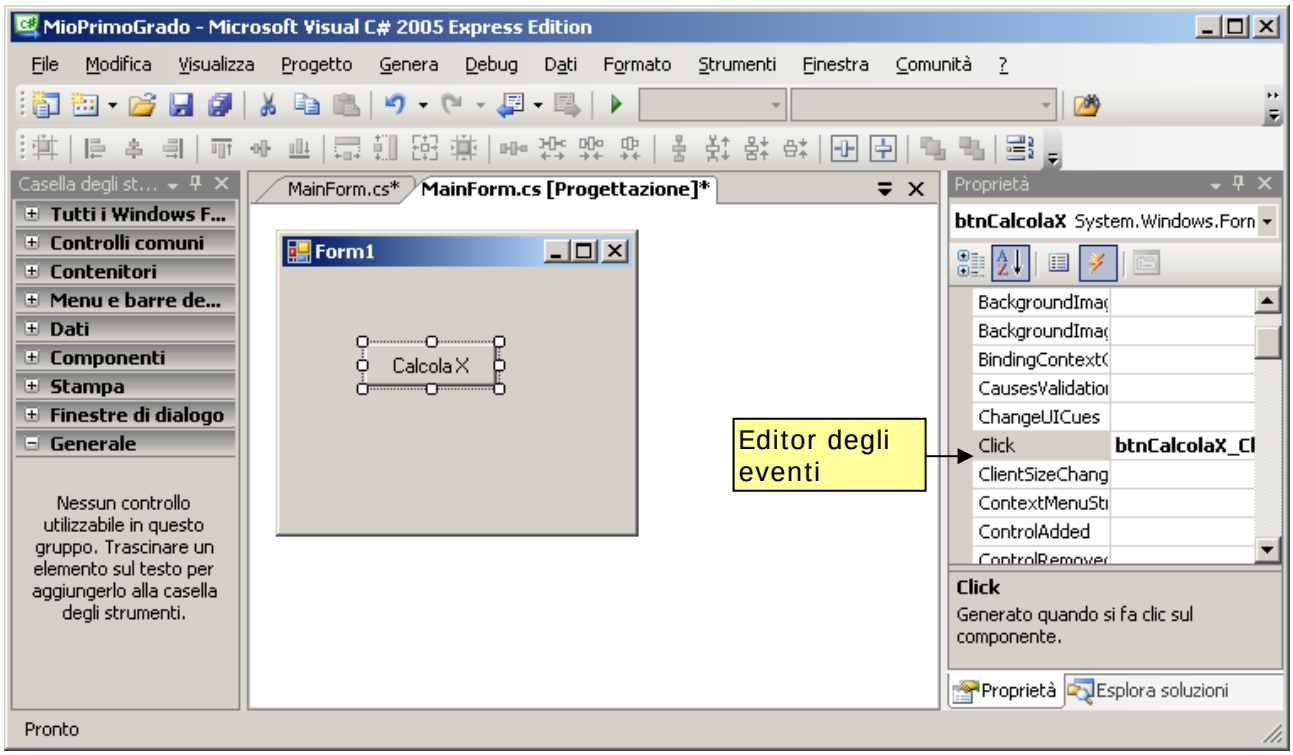


Figura 11-11 Gestione degli eventi mediante l'editor delle proprietà.

Per gestire un evento è sufficiente eseguire un doppio clic su di esso; dopodiché, l'IDE:

- genera automaticamente lo scheletro del gestore di evento, scrivendo il codice nel file Form.cs;
- collega il gestore di evento all'evento, scrivendo il codice nel file Form.Designer.cs

Nel caso del dell'evento `Click` del bottone `btnCalcolaX` viene generato il seguente codice nel file Form.Designer.cs:

```
private void InitializeComponent()
{
    ...
    this.btnCalcolaX.Location = new System.Drawing.Point(49, 45);
    ...
    this.btnCalcolaX.Click += new System.EventHandler(this.btnCalcolaX_Click);
    ...

}
```

Nel file Form.cs, il quale viene automaticamente portato in primo piano nell'editor:

```
private void btnCalcolaX_Click(object sender, EventArgs e)
{

}
```

A questo punto termina il lavoro dell'IDE e inizia quello del programmatore, il quale dovrà scrivere la logica di gestione dell'evento.

5.4 Eventi predefiniti

Ogni controllo definisce un **evento predefinito**; questo rappresenta l'evento gestito automaticamente dal designer quando l'utente esegue un doppo clic sul controllo. Ciò consente di velocizzare le operazioni di gestione degli eventi, evitando di utilizzare l'editor degli eventi.

Ad esempio, poiché `Click` è l'evento predefinito dei controlli `Button`, per gestirlo non è necessario utilizzare l'editor degli eventi, come mostrato nella figura precedente, ma è sufficiente eseguire un doppio clic sul bottone.

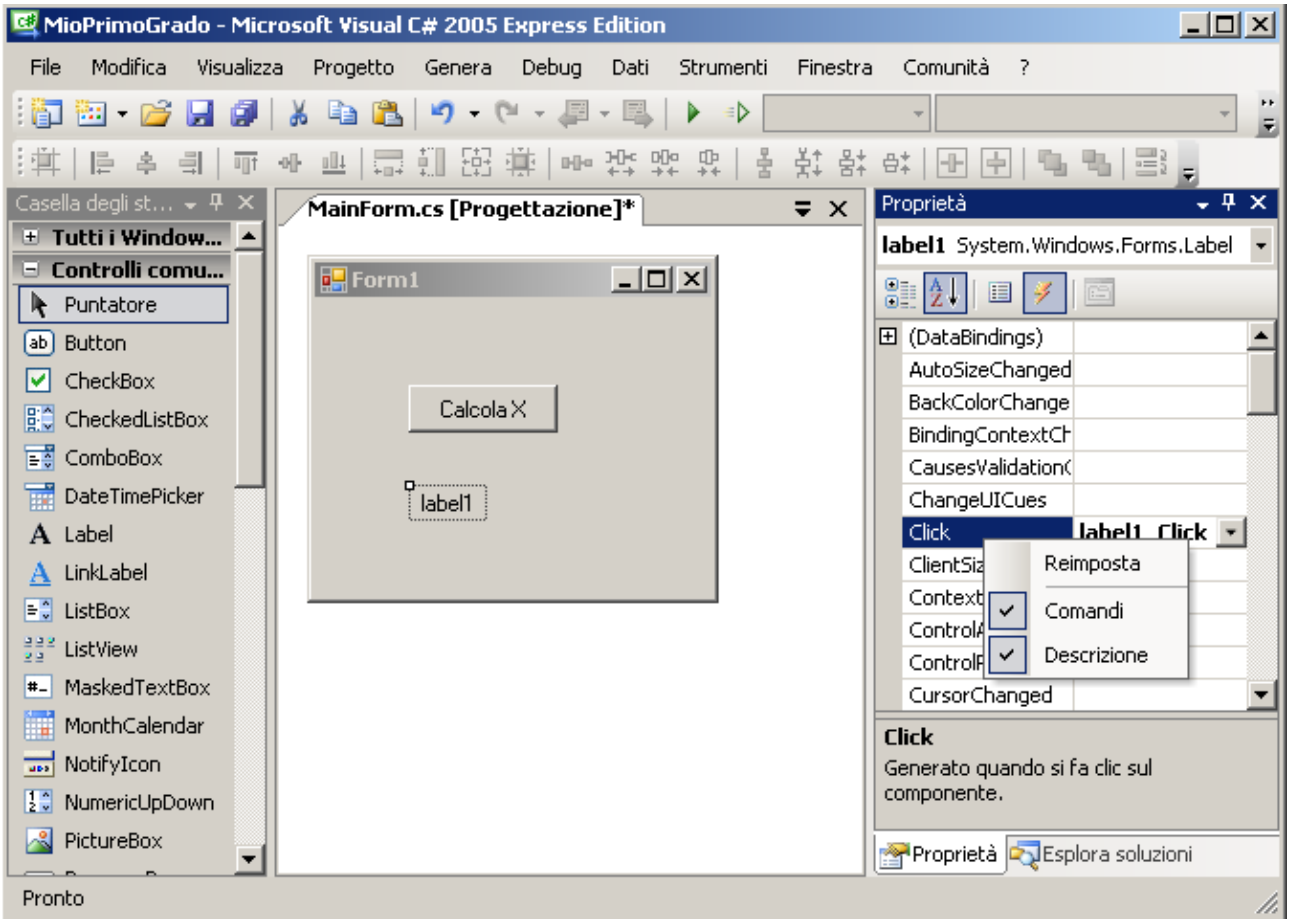
5.5 Rimuovere la gestione di un evento

A volte può capitare di voler annullare la gestione di un evento, perché non necessaria, oppure perché dovuta ad un errore.

Ad esempio, si supponga di aver creato una `Label` mediante il designer e di aver, per errore, eseguito un doppio clic su di essa. Come risposta, l'IDE genera automaticamente il codice di gestione dell'evento predefinito delle `Label`, che è l'evento `Click`.

Per rimuovere la gestione dell'evento è opportuno affidarsi all'IDE. Occorre innanziutto selezionare l'etichetta, quindi, nell'editor degli eventi, accedere all'evento erroneamente generato, cliccando con il pulsante destro del mouse. Dopodiché occorre selezionare il comando "Reimposta" del menù di scelta rapida. (Vedi figura a pagina successiva.)

La precedente operazione rimuove l'istruzione di associazione dell'evento al relativo medito di gestione, collocata nel metodo `InitializeComponent()` del file Form1.Designer.cs[36]. Infine resta da cancellare lo scheletro del gestore di evento, operazione che deve essere eseguita manualmente.



[36] L'operazione potrebbe essere eseguita manualmente accedendo direttamente al file Form1.Designer.cs, ma è altamente sconsigliato, poiché, per errore, si rischia di compromettere l'integrità del file.

Figura 11-12 Rimozione della gestione di un evento mediante l'editor degli eventi.

5.6 Modificare il nome del form

Nel generare lo scheletro dell'applicazione, l'IDE assegna dei nomi predefiniti sia ai file che ai vari elementi del codice sorgente. A questo proposito al form principale (e unico form dell'applicazione negli esempi considerati) prende viene assegnato il nome "Form1". E' possibile, e consigliabile, scegliere un nome più significativo, ad esempio "MainForm".

Per farlo è sufficiente accedere al pannello Esplora soluzioni e modificare il nome del file "Form1.cs" in "MainForm.cs"; dopodiché sarà l'IDE a modificare automaticamente il nome di tutti gli elementi connessi, come il file Form1.Designer.cs, la class Form1, ecc.

5.7 Conclusioni

Le funzionalità di progettazione visuale di Visual C# Express e prodotti analoghi semplificano e velocizzano il processo di costruzione dell'interfaccia, evitando un lavoro di codifica ripetitivo e noioso. Ma tutto ciò riguarda la fase di costruzione iniziale dell'interfaccia, e come questa debba presentarsi all'avvio dell'applicazione; ogni successiva modifica sull'aspetto o sul comportamento dei controlli deve essere codificata manualmente dal programmatore. Per questo motivo è importante che quest'ultimo conosca il codice necessario per impostare le caratteristiche visuali dei controlli e la gestione dei loro eventi.

12

Elementi base di un’interfaccia grafica

1 La classe Control

L’elemento centrale dell’interfaccia di una Applicazione Windows è la classe `Control`; ciò perché tutti i controlli, `TextBox`, `Button`, `Label`, `Form`, ecc, prendono la maggior parte delle loro caratteristiche da questa classe.

Conoscere e comprendere le caratteristiche della classe `Control` significa dunque conoscere e comprendere le basi di funzionamento di tutti i controlli.

1.1 Derivazione e gerarchia di classi

La derivazione è un concetto object oriented ed è trattato in modo completo nel volvolume dedicato alla programmazione ad oggetti. Qui ne forniremo una breve introduzione per quanto riguarda l’ambito della realizzazione di interfacce grafiche e utilizzando un’analogia di facile comprensione.

Si consideri la tassonomia del mondo animale, più precisamente quella parte che ci interessa da vicino in quanto appartenenti alla specie umana:

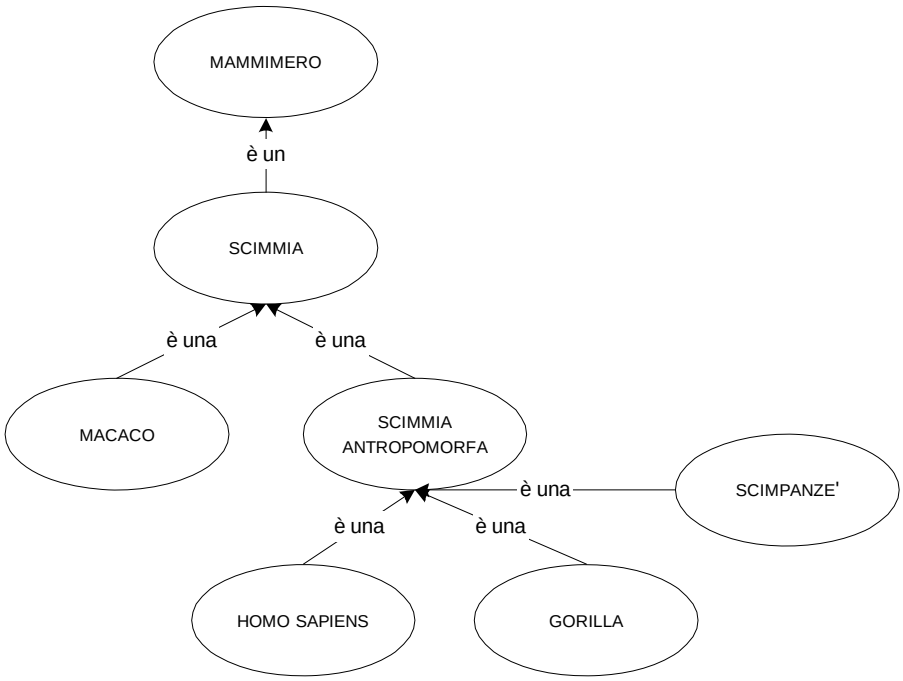


Figura 12-1 Rappresentazione schematica delle relazioni tra alcune specie e classi animali.

I soggetti che costituiscono lo schema sono legati da una speciale **relazione di parentela** designata dal termine “è un” (o “è una”). Questa relazione possiede un verso che determina il modo in cui dev’essere letta. E dunque, ad esempio: una scimmia “è un” mammifero, ma non vale il contrario, un mammifero non è una scimmia. Nello schema ci sono inoltre animali che non possiedono una

relazione tra loro: il macaco non è una scimmia antropomorfa; l'homo sapiens non è un gorilla o uno scimpanzé, ecc.

Il significato della relazione di parentela "è un" è abbastanza intuitivo. Affermando che una scimmia è un mammifero si sostiene che tutte le scimmie condividono le caratteristiche biologiche di qualsiasi mammifero – quattro arti, sangue caldo, gestazione dei piccoli, ecc – e ne introducono di nuove, che fanno parte dell'essere scimmia.

E ancora, affermare che una scimmia antropomorfa è una scimmia significa sostenere che essa condivide tutte le caratteristiche delle scimmie, e quindi anche tutte le caratteristiche dei mammiferi. Infine, un homo sapiens è una scimmia antropomorfa e dunque condivide tutte le caratteristiche delle scimmie antropomorfe, e quindi anche delle scimmie e dei mammiferi.

Designando i termini mammifero, scimmia, homo sapiens, eccetera, come "tipi di animali", si può sostenere che il tipo homo sapiens deriva (è una) dal tipo scimmia antropomorfa, che deriva dal tipo scimmia, che deriva dal tipo mammifero. In termini invertiti, si può sostenere anche che mammifero è il **tipo base** (da cui deriva) di scimmia, che è il tipo base di scimmia antropomorfa, che il tipo base di homo sapiens.

Lo schema mostra una gerarchia di tipi di animali. Alla base della gerarchia sta il mammifero, e cioè il tipo di animale da cui tutti gli altri derivano; esso definisce delle caratteristiche biologiche di base (che caratterizzano appunto l'essere un mammifero) le quali sono condivise da tutti gli altri tipi di animali. Ogni tipo di animale introduce nuove caratteristiche rispetto al tipo di animale da cui deriva direttamente, ma mantiene ovviamente le caratteristiche del tipo base.

1.2 Gerarchia dei controlli di una interfaccia grafica

I tipi di controlli di un'interfaccia grafica sono organizzati in una gerarchia analoga. Alla base della gerarchia sta il tipo `Control`.

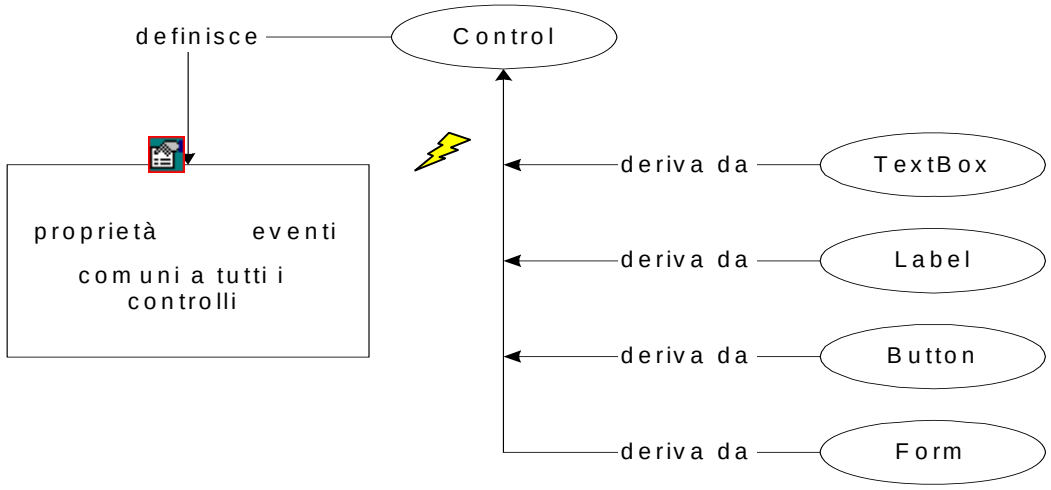


Figura 12-2 Schema delle relazioni tra i controlli di un'interfaccia grafica.

> Lo schema mostra un sotto insieme dei controlli disponibili, e cioè quelli introdotti nel capitolo precedente. Inoltre, lo schema non mostra i tipi di controlli intermedi. Infatti, `TextBox`, `Button`, `Form` e `Label`, non derivano direttamente da `Control`.

La classe `Control`, esattamente come il tipo mammifero, è ciò che si definisce un **tipo astratto**; essa definisce delle proprietà e degli eventi comuni a tutti i controlli, ma in pratica nessuna interfaccia presenta mai un oggetto del tipo `Control`, ma soltanto oggetti il cui tipo deriva da esso, come `Label`, `Button`, `TextBox`, ecc. Analogamente, non esiste un animale reale che sia semplicemente un mammifero, poiché il termine mammifero non designa una specie ma definisce un'insieme di caratteristiche comuni a varie specie.

1.3 Proprietà di base della classe Control

La classe `Control` definisce un notevole numero di proprietà, le quali possono essere impostate attraverso l'editor delle proprietà. Segue un elenco delle proprietà più comunemente utilizzate:

Tabella 12-1 Proprietà di base della classe Control.

PROPRIETÀ	DESCRIZIONE
ForeColor, BackColor `Color`	Definiscono rispettivamente il colore di testo e dello sfondo. Per sfondo si intende l'area occupata dal controllo.
BorderStyle `BorderStyle`	Definisce l'aspetto del bordo che delimita l'area del controllo. I possibili valori sono: `None`: non viene visualizzato alcun bordo; `FixedSingle`: il bordo è rappresentato da un linea sottile nera; `Fixed3D`: il bordo riproduce l'effetto visuale «bassorielievo».
BackgroundImage `Image`	Definisce l'immagine con la quale riempire l'area del controllo. Un modo per assegnare una valore alla proprietà e quello di specificare il file che contiene l'immagine mediante il metodo `FromFile` della classe `Image`; ad esempio: *`controllo`*`.BackgroundImage = Image.FromFile("icona.bmp");`
Enabled `bool`	Indica se il controllo è abilitato (`true`) o non abilitato (false) a ricevere gli «eventi reali».
Focused `bool`	Indica se il controllo detiene il fuoco. (Vedi Paragrafo 1.6.)
Left, Top `int`	Definiscono la posizione, ascissa (`Left`) e ordinata (`Top`), del controllo all'interno del `form`. I valori si intendono in pixel.
Width, Height `int`	Definiscono rispettivamente la dimensione orizzontale (`Width`) e verticale (`Height`) del controllo. I valori si intendono in pixel.
Location `Point`	Definisce la posizione del controllo. (Cioè le stesse informazioni delle proprietà `Left` e `Top`.)
Size `Size`	Definisce le dimensioni del controllo. (Cioè le stesse informazioni delle proprietà `Width` e `Height`.)
Text `string`	Definisce la stringa di testo visualizzata. Ogni controllo fornisce un uso personalizzato di questa proprietà.
TabIndex `int`	Definisce l'ordine di tabulazione rispetto agli altri controlli; se non impostato riflette l'ordine di inserimento degli stessi nel `form`.
TabStop `bool`	Definisce se il controllo può essere raggiunto mediante il tasto di tabulazione (`true`) oppure no (`false`).

Visible	
	Indica se il controllo è visibile (`true`) o non visibile (`false`).
`bool`	

1.4 Posizione e dimensioni dei controlli

La posizione e le dimensioni di un controllo sono definite in relazione alla **client area** del form che lo contiene. In figura, questa è circoscritta dal bordo evidenziato.

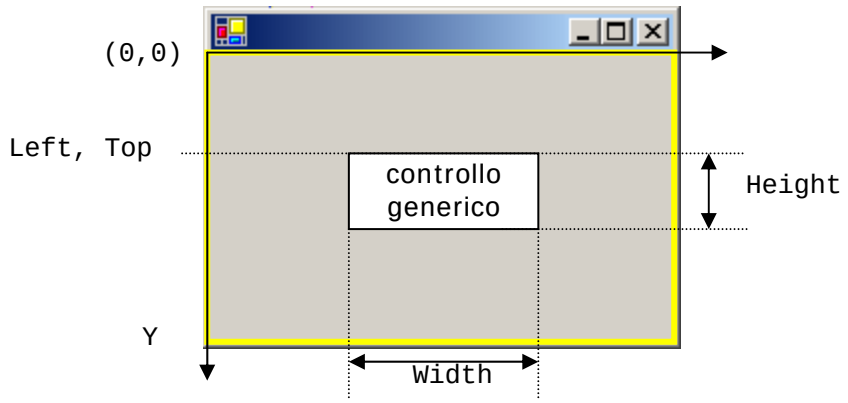


Figura 12-3 Rappresentazione della *client area* di un form.

La client area è rappresentata da un piano cartesiano la cui origine si trova nell'angolo in alto a sinistra. All'interno di essa ogni controllo occupa una posizione e una zona rettangolare – **area controllo** – di dimensioni ben precise, entrambe misurate in pixel[37]. La parte di controllo che eventualmente eccede le dimensioni della *client area* del form non viene visualizzata.

La posizione di un controllo nella client area del form può essere espressa in due modi:

- attraverso la coppia di proprietà `Left` (ascissa) e `Top` (ordinata);
- attraverso la proprietà `Location`, che appartiene a un particolare tipo di dato: il `Point`.

In entrambi casi, la posizione si riferisce all'angolo in altro a sinistra dell'area controllo.

Quando un controllo viene creato, sia la posizione che le dimensioni vengono automaticamente impostate a dei valori predefiniti: `Left` e `Top` sempre a zero; `Width` e `Height` in base al tipo di controllo. Di norma, il programmatore reimposta la posizione del controllo in fase di costruzione dell'interfaccia, ma nulla impedisce di modificarla in qualsiasi punto del programma.

Ad esempio, il seguente codice:

```
lblEsempio.Left -= lblEsempio.Width / 2;
lblEsempio.Top -= lblEsempio.Height / 2;
lblEsempio.Width += lblEsempio.Width *2;
lblEsempio.Height += lblEsempio.Height * 2;
```

raddoppia le dimensioni del controllo sia in orizzontale che in verticale, quadruplicando l'area occupata, ma senza spostare il punto centrale di tale area.

Proprietà `Location` e tipo `Point`

La proprietà `Location` memorizza la posizione del controllo sotto forma di una coppia di coordinate, `X` e,`Y`, e può essere manipolata come un unico valore, il cui tipo è `Point`. Il contenuto di `Location` riflette i valori di `Top` e `Left`, e viceversa.

[37] Pixel sta per «*picture element*» e può essere tradotto in «punto».

Ad esempio, il seguente codice (il bottone `b` si suppone già creato):

```
b.Left = 100;
b.Top = 150;
```

equivale all'istruzione:

```
b.Location = new Point(100, 150);
```

Proprietà `Size` e tipo `Size`

Analogamente alla proprietà `Location`, la proprietà `Size` memorizza le dimensioni del controllo sotto forma di una coppia di valori, `Width` e `Height`, e può essere manipolata come un unico oggetto di tipo è `Size`. Il valore contenuto in `Size` riflette i valori delle proprietà `Width` e `Height`, e viceversa.

Ad esempio, il seguente codice:

```
b.Width = 50;
b.Height = 25;
```

equivale all'istruzione:

```
b.Size = new Size(50, 25);
```

1.5 Metodi della classe Control

Control definisce un numeroso set di metodi allo scopo di implementare le funzionalità di base, comunu a tutti i controlli. Qui ne elencheremo un ridottissimo sottoinsieme.

Tabella 12-2 Metodi della classe Control

PROTOTIPO / DESCRIZIONE
`void BringToFront()`
Porta il controllo in primo piano rispetto a tutti gli altri controlli del form. E' utile quando due o più controlli occupano la stessa area del form e dunque sono sovrapposti e si desidera che uno di essi appaia davanti agli altri.
`bool Focus()`
Seleziona il controllo rendendolo attivo. Ritorna `true` se l'operazione ha avuto successo, `false` altrimenti. L'operazione potrebbe fallire per, ad esempio, il controllo è disabilitato. (Vedi il paragrafo successivo).
`void SendToBack()`
Porta il controllo in secondo piano. Esegue l'operazioone opposta del metodo `BringToFront()`.
`void Select()`
Seleziona il controllo, rendendolo attivo. Diversamente dal metodo `Focus()` non indica se l'operazione ha avuto successo o meno.

1.6 Concetti di “fuoco” e di controllo selezionato

Le idee di **fuoco** e di **controllo selezionato** sono collegate. Il tutto si riduce a rispondere dalla seguente domanda: in un dato momento quale controllo, tra tutti quelli inseriti nel form, riceve gli eventi provenienti dalla tastiera? La risposta è: al controllo attualmente selezionato, ovvero al controllo che in quel momento detiene il fuoco[38].

Acquisire il fuoco e perdere il fuoco

Perché l’interfaccia sia effettivamente funzionale, deve essere possibile spostare il fuoco da un controllo all’altro. Ciò può avvenire sia come risposta alle azioni dell’utente, sia come effetto dell’esecuzione dei metodi `Focus()` e `Select()`.

L’utente può spostare il fuoco da un controllo all’altro:

1) premendo il tasto di tabulazione (TAB e SHIFT TAB): il fuoco si sposta al controllo successivo (o precedente, nel caso di SHIFT TAB) nell’ordine di tabulazione, il quale dipende, per ogni controllo, dal valore della proprietà `TabIndex`.

2) cliccando sul controllo con il mouse.

Tutto ciò funziona soltanto se il controllo è del tipo che può ricevere il fuoco è visibile (la proprietà `Visible` è `true`) ed è attivo (la proprietà `Enabled` è `true`).

> L’uso dei tasti di tabulazione richiede un’ulteriore condizione: la proprietà `TabStop` del controllo dev’essere impostata a `true`, altrimenti esso potrà ricevere il fuoco soltanto attraverso il mouse. (Il valore predefinito di tale proprietà è per l’appunto `true`.)

A livello di codice è possibile dare il fuoco ad un controllo richiamando i metodi `Select()` o `Focus()`.

Ogni controllo, sia quando riceve il fuoco sia quando lo perde, solleva due eventi di sola notifica: `GotFoucs` ed `Enter` dopo che ha ricevuto il fuoco; `LostFocus` e `Leave` dopo che lo ha perso. Attaccando ad essi dei gestori di eventi è possibile eseguire determinati compiti dopo che il controllo ha ricevuto, o perduto, il fuoco.

Apparenza di un controllo selezionato

Poiché l’utente deve sapere quale controllo è selezionato in un dato momento, di norma ogni controllo modifica il proprio aspetto in base al fatto che possieda il fuoco o meno. Ad esempio, un `Button` selezionato mostra un bordo tratteggiato che circoscrive il testo, mentre un `TextBox` mostra il cursore testo.

2 Eventi di base della classe Control

Segue una lista di alcuni degli eventi definiti dalla classe `Control`.

[38] In realtà un controllo può essere nello stato «selezionato» pur senza avere il fuoco, ma ciò soltanto quando il form che contiene il controllo in questione non è attivo.

2.1 Eventi di tastiera

Tabella 12-3 Eventi prodotti dalla tastiera.

EVENTO	DESCRIZIONE
KeyDown	Un tasto qualsiasi viene premuto. (Compresi i tasti SHIFT, CAPS, CTRL, ALT GR).
KeyUp	Un tasto qualsiasi viene rilasciato. (Compresi i tasti SHIFT, CAPS, CTRL e CTRL, ALT GR)
KeyPress	Un tasto viene premuto e rilasciato. (Questo evento è sollevato in risposta a un sotto insieme dei tasti disponibili, dal quale sono esclusi i tasti funzione (F1, F2, ect), i tasti freccia, i tasti INS e CANC, SHIFT, CAPS, CTRL, ALT GR, e altri.

2.2 Eventi prodotti dal mouse

Tabella 12-4 Eventi prodotti dal mouse

EVENTO	DESCRIZIONE
Click	Clic con il pulsante sinistro del mouse. (Oppure quello destro in base alle impostazione del mouse.)
DoubleClick	Doppio clic con il pulsante sinistro del mouse. (Oppure quello destro in base alle impostazione del mouse.)
MouseHover	Il puntatore del mouse si trova sopra il controllo. L'evento viene sollevato in continuazione ad intervalli regolari finché il puntatore non esce dal controllo.
MouseEnter	Il puntatore del mouse "entra" nell'area del controllo.
MouseLeave	Il puntatore del mouse "lascia" l'area del controllo.
MouseDown	Abbassamento di un pulsante qualsiasi del mouse.
MouseMove	Spostamento del puntatore del mouse sopra l'area del controllo.
MouseUp	Rilascio di un pulsante qualsiasi del mouse. Nel caso del pulsante sinistro, il rilascio viene rilevato anche se il pulsante non si trova più sopra l'area del controllo quando ciò avviene. Non è così nel caso del pulsante destro.

2.3 Classi KeyEventArgs» e KeyPressEventArgs

Gli eventi `KeyDown`, `KeyUp` producono informazioni supplementari sul tasto premuto e sullo stato della tastiera, memorizzate in un parametro di tipo `KeyEventArgs`.

Tabella 12-5 Proprietà principali della classe KeyEvenArgs.

PROPRIETÀ	DESCRIZIONE
Alt, Control, Shift bool	Indicano se i tasti omonimi (ALT, CTRL, SHIFT) sono premuti (true) oppure no (false).
Handled bool	Se impostata al valore true, comunica al controllo che l'evento è stato gestito e che dunque non è più necessaria alcuna forma di elaborazione sul tasto premuto. (In pratica, inibisce la risposta predefinita prodotta dal controllo.)
KeyCode Keys	Contiene il codice del tasto premuto. I codici dei tasti sono definiti mediante delle costanti appartenenti al tipo Keys. Ad esempio: Keys.Enter, Keys.Del, Keys.End rappresentano rispettivamente i tasti ENTER, DEL, END.

Il tipo di delegate dell'evento KeyPress richiede che il secondo parametro del gestore di evento sia di tipo KeyPressEventArgs. Questa classe espone le sole proprietà Handled e KeyChar; la seconda, di tipo char, contiene il codice Unicode del tasto premuto.

Si gestisce l'evento KeyDown (o KeyUp) quando è necessario un controllo sofisticato della tastiera; più semplice e più comune è la gestione dell'evento KeyPress. Un esempio sulla gestione di tale evento sarà mostrato più avanti.

2.4 Classe MouseEventArgs

Gli eventi MouseDown, MouseUp, MouseMove producono informazioni supplementari sullo stato del mouse, memorizzate in un parametro di tipo MouseEventArgs.

Tabella 12-6 Proprietà principali della classe MouseEventArgs.

PROPRIETÀ	DESCRIZIONE
Button MouseButton	Indica quale pulsante del mouse è stato premuto o rilasciato. I pulsanti sono definiti mediante le costanti: Left, Middle, Right del tipo enumeratore MouseButton.
X, Y int	Definiscono le coordinate (ascissa e ordinata) del puntatore del mouse. Le coordinate sono relative all'angolo in alto a sinistra dell'area controllo.

2.5 Commento sugli eventi «MouseXXX» e «KeyXXX»

Tali eventi hanno una corrispondenza diretta con le azioni dell'utente (*user generated events*) e forniscono informazioni dettagliate sullo stato di tali azioni. Di solito non è necessario interagire con l'utente fino a questo livello. Ad esempio, nel gestire gli eventi relativi a un Button, poco importa conoscere le coordinate del puntatore del mouse quando l'utente clicca su di esso, ciò che serve è semplicemente poter rispondere al clic dell'utente e per questo basta l'evento Click.

Analogamente, i TextBox dialogano in modo molto sofisticato con l'utente, gestendo automaticamente i tasti INS, DEL, HOME, ecc, oltre al mouse naturalmente, senza che al programmatore sia richiesto di scrivere una sola riga di codice. E di norma non c'è alcun bisogno

che esso intervenga in questa gestione, benché gli eventi `KeyPress`, `KeyDown` e `KeyUp` gli consentano di farlo.

2.6 Altri eventi della classe Control

Gli eventi sotto elencati non hanno una corrispondenza diretta con azioni dell'utente.

Tabella 12-7 Altri eventi della classe Control.

EVENTO	DESCRIZIONE
`Enter`	Il controllo diventa selezionato.
`Leave`	Il controllo perde la selezione.
`GotFocus`	Il controllo riceve il fuoco.
`LostFocus`	Il controllo perde il fuoco.
`Validating`	Il controllo è in fase di validazione. Gestendo questo evento è possibile impedire che il controllo, ad esempio un `TextBox`, perda il fuoco almeno fin quando non contiene un valore appropriato. (I tipi `CancelEventArgs` e `CancelEventHandler` sono definiti nel namespace: `System.ComponentModel`)
`Validated`	Questo evento viene di norma gestito in combinazione con l'evento `Validating`. `Validated` viene sollevato se la fase di validazione dà esito positivo.

Gli eventi `GotFocus` ed `Enter` da una parte, e `LostFocus` e `Leave` dall'altra non sono dei duplicati, poiché vengono sollevati in momenti diversi durante l'acquisizione e la perdita del fuoco. Inoltre il loro funzionamento è diverso nel caso di applicazioni da più form. Per il momento, comunque, possono essere considerati eventi dal funzionamento equivalente.

3 Visualizzare messaggi: classe MessageBox

In molte situazioni si presenta la necessità di comunicare all'utente lo stato di una certa elaborazione, oppure di chiedere una conferma prima di procedere o meno all'esecuzione di una determinata operazione. Un esempio, che riguarda i programmi di video scrittura, è la richiesta di conferma per il salvataggio del documento, richiesta avanzata all'utente quando questo tenta di chiudere il programma senza aver prima salvato il documento:

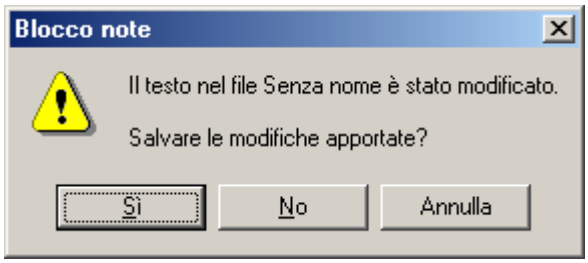


Figura 12-4 Finestra di dialogo di salvataggio visualizzata da Blocco note.

L'esempio mostra un particolare tipo di form, chiamato **message dialog**, che chiede all'utente di selezionare una fra tre possibili alternative. La principale caratteristica di una message dialog è che assume il pieno controllo sulle azioni dell'utente; in altre parole, finché la finestra non viene chiusa non è possibile accedere al resto dell'interfaccia.

Le message dialog sono solitamente impiegate per servire i seguenti scopi:

1) visualizzare un messaggio informativo;

1) visualizzare un messaggio di errore;

visualizzare un messaggio di avvertimento o di richiesta di conferma (vedi esempio).

Una message dialog viene visualizzata mediante l'esecuzione del metodo `Show()` della classe `MessageBox`. L'invocazione del metodo può assumere varie forme, in base al numero di elementi con i quali si intende caratterizzare la finestra. Il suo prototipo è[39]*:*

```
DialogResult Show(string testo,
                  string titolo_opz,
                  MessageBoxButtons buttons_opz,
                  MessageBoxIcon icon_opz);
```

dove:

`testo` : rappresenta il messaggio da visualizzare;
`titolo` : rappresenta il testo da visualizzare sulla barra del titolo della message dialog;
`buttons` : indica quali bottoni visualizzare;
`icon` : indica con quale icona caratterizzare la message dialog.

Come si vede dal prototipo, solo il parametro `testo` è obbligatorio; gli altri hanno lo scopo di caratterizzare l'apparenza e il comportamento della finestra.

Il metodo `Show()` produce un valore di ritorno di tipo `DialogResult` che rappresenta la risposta dell'utente, e cioè il bottone cliccato. Esso è definito dalle seguenti costanti:

Tabella 12-8 Elenco dei possibili valori prodotti del metodo Show().

RIPOSTA UTENTE	DESCRIZIONE
Abort	E' stato cliccato il bottone **Termina**.
Cancel	E' stato cliccato il bottone **Annulla**.
Ignore	E' stato cliccato il bottone **Ignora**.
No	E' stato cliccato il bottone **No**.
OK	E' stato cliccato il bottone **Ok**.
Retry	E' stato cliccato il bottone **Riprova**.
Yes	E' stato cliccato il bottone **Si**.

3.1 *Message dialog* informative

Una message dialog informativa si limita a comunicare un certo messaggio all'utente, senza chiedere a quest'ultimo di compiere alcuna scelta. Nella sua forma più semplice appare così:

[39] In realtà esiste una ulteriore versione del metodo che accetta un quinto argomento.

Figura 12-5 Esempio di message dialog informativa.

Tale risultato si ottiene mediante l'invocazione:

```
MessageBox.Show("Questo è un messaggio informativo");
```

Come si nota, il valore ritornato dal metodo `Show()` viene ignorato ed è naturale che sia così, poiché, per definizione, l'unica azione concessa all'utente è quella di cliccare sul bottone OK.

La precedente dialog è piuttosto avara di contenuti; occorre senz'altro specificare un testo da visualizzare sulla barra del titolo, che può qualificare il tipo di messaggio, oppure semplicemente riportare il nome del programma. Ciò si ottiene specificando come secondo argomento il testo da visualizzare sulla barra del titolo:

```
MessageBox.Show("Questo è un messaggio informativo", "Questo è il titolo");
```

Infine, anche se non strettamente necessario, può essere appropriato visualizzare un'icona che mostra la natura informativa della dialog e ottenere un risultato simile al seguente:

Figura 12-6 Esempio di message dialog informativa con icona Information.

Ciò richiede di specificare altri due parametri, poiché è impossibile visualizzare una *message dialog* con icona senza specificare anche quali bottoni visualizzare:

```
MessageBox.Show("Questo è un messaggio informativo",
                "Questo è il titolo",
                MessageBoxButtons.OK,
                MessageBoxIcon.Information);
```

Il testo in grassetto rappresenta il parametro che caratterizza l'icona visualizzata; i possibili valori appartengono all'enumeratore `MessageBoxIcon` e sono rappresentati nella tabella a pagina successiva.

Message dialog di errore

Una message dialog di errore comunica uno stato errato del programma. Di norma ciò avviene aggiungendo l'icona appropriata al messaggio informativo. Ciò si ottiene specificando MessageBoxIcon.Error come ultimo parametro. Ad esempio:

```
MessageBox.Show("Messaggio di errore!",
                "Questo è il titolo",
                MessageBoxButtons.OK,
                MessageBoxIcon.Error);
```

Tabella 12-9 Elenco dei possibili valori del parametro icon.

COSTANTE	ICONA
Asterisk	
Error	
Exclamation	
Hand	
Information	
None	Nessuna
Question	
Stop	
Warning	

3.2 Elaborare la risposta dell'utente

Con le message dialog informative la comunicazione avviene dal programma verso l'utente; quest'ultimo può solo confermare di aver "ricevuto" il messaggio. Diverso è il caso delle message dialog di avvertimento, domanda o conferma. Con esse si chiede all'utente di selezionare una tra più scelte possibili, rappresentate dai bottoni visualizzati. In questo caso il programma dovrà elaborare in qualche modo la risposta dell'utente.

> Ciò non rappresenta un requisito del linguaggio. D'altra parte è illogico visualizzare una message dialog che presenta i bottoni OK e Annulla e non verificare quale tra le due scelte ha effettuato l'utente.

Il seguente frammento di codice visualizza una message dialog di conferma ed elabora successivamente il valore ritornato dal metodo `Show()`, che rappresenta la risposta dell'utente:

```
DialogResult scelta;
scelta = MessageBox.Show("Confermi l'operazione?","Salvataggio dati cliente ",
                         MessageBoxButtons.YesNo,
                         MessageBoxIcon.Warning);
if (scelta == DialogResult.Yes)
{
    ...// effettua l'operazione di cui è stata richiesta la conferma
}
```

La figura mostra la message dialog visualizzata:

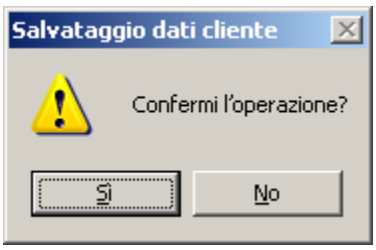


Figura 12-7 Esempio di message dialog di richiesta conferma.

In questo caso, il metodo `Show()` ritorna un codice che rappresenta la risposta fornita dall'utente; codice che nell'istruzione precedente viene assegnato alla variabile `scelta`. Ovviamente, il tipo e il numero dei bottoni visualizzati determina il tipo di risposta che può fornire l'utente.

Segue l'elenco dei valori che può assumere il parametro `buttons`, i cui possibili valori appartengono all'enumeratore `MessageBoxButtons`:

Tabella 12-10 Elenco dei possibili valori del parametro buttons.

COSTANTE	DESCRIZIONE
`AbortRetryIgnore`	Sono visualizzati i bottoni **Termina**, **Riprova**, **Ignora**.
`OK`	E' visualizzato il bottone **OK**.
`OKCancel`	Sono visualizzati i bottoni **OK**, **Annulla**.
`RetryCancel`	Sono visualizzati i bottoni **Riprova**, **Annulla**.
`YesNo`	Sono visualizzati i bottoni **Si**, **No**.
`YesNoCancel`	Sono visualizzati i bottoni **Si**, **No**, **Annulla**.

4 Controlli Form, Label, TextBox, Button

I controlli di tipo `Form`, `Label`, `TextBox` e `Button` ereditano le caratteristiche – proprietà, metodi ed eventi – dalla classe `Control`, e ne aggiungono di nuove. Ognuno di essi, essendo stato realizzato per svolgere particolari funzioni all'interno dell'interfaccia, specializza alcune delle proprietà ereditate, oppure le nasconde addirittura.

Segue una breve introduzione per ognuno di essi.

4.1 Classe Form

Rispetto agli altri tipi di controllo, la classe `Form` rientra in una categoria a parte, poiché di fatto non esiste un'interfaccia grafica senza almeno un form. La classe espone un notevole numero di proprietà ed eventi; di entrambi ne sarà esaminato soltanto un piccolo sotto insieme.

Proprietà

Tabella 12-11 Proprietà principali della classe Form.

PROPRIETÀ	DESCRIZIONE
`ClientSize` `Size`	Memorizza le dimensioni, in pixel, della *client area* del form.
`ForeColor` `Color`	Impostando questa proprietà viene modificato il `ForeColor` di tutti controlli contenuti nel form per i quali non è stata impostata la proprietà omologa.
`FormBorderStyle` `FormBorderStyle`	Stile del bordo. Questa proprietà influenza sia l'apparenza del bordo che la possibilità di poter dimensionare il form. I possibili valori sono: `None`, `FixedSingle`, `Fixed3D`, `FixedDialog`, `Sizable`, `FixedToolWindow`, `SizableToolWindow`.

PROPRIETÀ	DESCRIZIONE
KeyPreview `bool`	se `true`, fa sì che il form riceva gli eventi di tastiera prima del controllo che possiede il fuoco.
MinimumSize **MaximumSize** `Size`	Dimensioni minime e massime che può assumere il form.
StartPosition `FormStartPosition`	Posizione iniziale del form sullo schermo. I possibili valori sono: `Manual`, `CenterScreen`, `WindowsDefaultLocation`, `WindowsDefaultBounds`, `CenterParent`

Eventi

Tabella 12-12 Eventi della classe Form.

EVENTO	DESCRIZIONE
Activated	Viene sollevato dopo che il form è diventato attivo. Un form si dice attivo quando assume il controllo sulle azioni dell'utente. Un form attivo mostra una barra del titolo colorata (la barra del titolo di un form inattivo è grigia).
Deactivate	Viene sollevato prima che il form diventi inattivo.
FormClosing	Viene sollevato mentre è in atto un tentativo di chiusura del form. Impostando a `true` la proprietà `Cancel` del parametro informazioni evento il tentativo di chiusura viene abortito.
FormClosed	Viene sollevato dopo che il form è stato chiuso. (Ma prima che scompaia dallo schermo)
Load	Viene sollevato dopo che il form è stato costruito, ma prima che sia visualizzato. Rappresenta l'evento predefinito.

Questi eventi sono contestuali ai vari cambiamenti di stato di un form. Per quanto riguarda il form principale:

- viene creato una sola volta, contestualmente all'esecuzione del programma (evento `Load`);
- viene chiuso una sola volta, contestualmente alla fine del programma (eventi `FormClosing` e `FormClosed`);
- come ogni altro form, può diventare attivo e non attivo un numero arbitrario di volte (eventi `Activated` e `Deactivate`);

Di seguito viene fornito un esempio di gestione dell'evento `Closing`, attraverso il quale il programmatore può intervenire sulla sequenza di chiusura di un form. L'esempio riporta il metodo di gestione dellp'evento, collocato in MainForm.cs; ovviamente, l'evento è stato collegato mediante l'editor degli eventi, che produce l'istruzione appropriata nel file MainForm.Designer.cs.

```
public partial class MainForm : Form
{
    public MainForm()
    {
```

```
        InitializeComponent();
    }
    private void MainForm_FormClosing(object sender, FormClosingEventArgs e)
    {
        DialogResult s;
        s = MessageBox.Show("Confermi chiusura del programma?",
                            "Gestione clienti",MessageBoxButtons.YesNoCancel);
        if (s != DialogResult.Yes)        // chiusura non confermata
            e.Cancel = true;
    }
}
```

Mediante una message dialog viene chiesto all'utente se intende confermare la chiusura del form (e quindi del programma). Se l'utente non conferma viene impostata a `true` la proprietà `Cancel` del parametro e. La procedura di chiusura viene dunque abortita.

4.2 Classe Label

L'uso più comune delle `Label` è quello di testo informativo, con lo scopo di qualificare gli altri controlli, soprattutto `TextBox`, specificando la natura delle informazioni che essi visualizzano e/o consentono di inserire. D'altra parte nulla impedisce di impiegarle direttamente per l'output dei dati, come è stato fatto nel programma MioPrimoGrado.

Proprietà

Tabella 12-13 Proprietà della classe Label.

PROPRIETÀ	DESCRIZIONE
`AutoSize` `bool`	Se `true` questa proprietà fa sì che le dimensioni dell'etichetta varino in modo automatico in relazione allo spazio occupato dal testo da visualizzare. Questo dipende oltre che dalla lunghezza del testo anche dalle caratteristiche del Font impostato.
`TextAlign` `ContentAlignment`	Consente di impostare l'allineamento del testo rispetto all'area dell'etichetta. (solo se `AutoSize` è `false`). .
`Text` `string`	Testo visualizzato. Se contiene il carattere &, il carattere che lo segue viene interpretato come **tasto di accesso** o **acceleratore**, e cioè un tasto che premuto in combinazione con ALT determina la selezione del controllo associato alla etichetta. (Ciò vale soltanto se la proprietà `UseMnemonic` è impostata a `true`, a cioè al suo valore di default.)

Una `Label` può essere associata al controllo che la segue nell'ordine di tabulazione, tipicamente un TextBox (proprietà `TabIndex`). Se l'utente digita la combinazione "ALT + acceleratore", il controllo associato viene automaticamente selezionato. Poiché il controllo `txtProva1` è stato aggiunto al form immediatamente dopo `lblProva1`, il primo viene associato al secondo; di conseguenza, la combinazione di tasti `ALT+T` (dove `T` è il tasto di accesso definito nel testo dell'etichetta) determina la selezione di `txtProva1`, ovunque si trovi il fuoco in quel momento.

Eventi

Di norma non c'è alcun bisogno di gestire gli eventi sollevati da un'etichetta.

4.3 Classe TextBox

Il ruolo svolto dai controlli `TextBox` è quello consentire l'inserimento dei dati, benché nulla impedisca di utilizzarli a semplice scopo di visualizzazione.

Proprietà

Tabella 12-14 Proprietà della classe TextBox.

PROPRIETÀ	DESCRIZIONE
`CharacterCasing` `CharacterCasing`	Mediante questa proprietà è possibile forzare il case dei caratteri digitati dall'utente, trasformandoli in maiuscolo, minuscolo oppure lasciandoli inalterati. I possibili valori sono: `Normal`, `Upper`, `Lower`.
`MaxLength` `int`	Stabilisce il numero massimo di caratteri che l'utente può inserire.
`MultiLine` `bool`	Definisce la natura, singola linea o multilinea, del `TextBox`. Un `TextBox` multilinea consente la visualizzazione e l'inserimento di più righe di testo.
`Lines` `string[]`	Consente, nel caso di un TextBox multilinea (vedi prop. `MultiLine`) di accedere al testo contenuto nel controllo come ad un vettore di stringhe.
`ReadOnly` `bool`	Impostando a `true` questa proprietà si rende il `TextBox` di sola lettura, impedendo all'utente di modificarne il contenuto.
`TextAlign` `HorizontalAlignment`	Consente di specificare l'allineamento orizzontale del testo. I possibili valori sono: `Center`, `Left`, `Right`.
`WordWrap` `bool`	Questa proprietà funziona solo per i `TextBox` multilinea (vedi proprietà `MultiLine`). Impostata a `true` fa sì che il controllo esegua un ritorno a capo automatico quando il testo digitato dall'utente raggiunge il limite destro.

Eventi

L'evento più importante della classe `TextBox` è `TextChanged`, il quale viene sollevato in risposta a una qualsiasi modifica del testo contenuto nel `TextBox` (inserimento, cancellazione, modifica dei caratteri contenuti nella proprietà `Text`).

La gestione delle evento `TextChanged` è utile quando lo stato di alcuni controlli deve dipendere dal contenuto di uno o più `TextBox`. Un esempio di gestione di tale evento sarà presentato più avanti.

Metodi

Tra i molti metodi definiti dalla classe TextBox, uno che può essere particolarmente utile è il metodo `SelectAll()`. Questo metodo consente di selezionare tutto il testo contenuto in un `TextBox`. Poiché il metodo funziona soltanto se il TextBox è selezionato, di norma viene impiegato insieme al metodo `Select()`.

4.4 Classe Button

Il ruolo svolto dai controlli `Button` è del tutto evidente: essi consentono all'utente di confermare e/o dare l'avvio a determinate elaborazioni; ciò si ottiene gestendo l'evento `Click` sollevato dal controllo. Nella sua forma di base, l'aspetto di un `Button` è caratterizzato semplicemente da un testo che ne precisa la funzione.

Proprietà

Tabella 12-15 Proprietà della classe Button.

PROPRIETÀ	DESCRIZIONE
`FlatStyle` `FlatStyle`	Definisce l'apparenza del bottone.
`Image` `Image`	Consente di caratterizzare il bottone con un'immagine. Un modo per definire il contenuto di tale proprietà e quello di specificare il file che contiene l'immagine mediante il metodo `FromFile` della classe `Image`: `btnEsempio.Image = Image.FromFile("nomeicona.bmp");`
`ImageAlign` `ContentAlignment`	Consente di impostare l'allineamento dell'immagine rispetto all'area del controllo. (Per i possibili valori, vedi proprietà `TextAlign` della classe `Label`.)
`Text` `string`	Testo visualizzato. Se il testo contiene il carattere &, il carattere che lo segue viene interpretato come **acceleratore** e cioè un tasto che premuto in combinazione con ALT produce l'evento `Click`.
`TextAlign` `ContentAlignment`	Consente di impostare l'allineamento del testo rispetto all'area del controllo. (Per i possibili valori, vedi proprietà `TextAlign` della classe `Label`.)

Eventi

Nella maggior parte dei casi l'unico evento da gestire è l'evento `Click`.

Metodi

La classe Button definisce un metodo, `PerformClick()`, che consente di simulare il clic dell'utente sul bottone. In pratica, eseguendo tale metodo viene eseguito il gestore di evento associato all'evento `Click` del bottone.

5 "Consistenza" dell'interfaccia

Il programma MioPrimoGrado presenta dei limiti nella gestione dell'interazione con l'utente; di questi saranno affrontati i seguenti due:

- nulla impedisce all'utente di eseguire il calcolo della soluzione dell'equazione senza aver prima inserito i valori dei due coefficienti;

- nulla impedisce all'utente di inserire caratteri qualsiasi, dunque anche lettere, nei `TextBox` relativi ai due coefficienti.

Entrambe le questioni riguardano il problema della **consistenza** dell'interfaccia. Brevemente:

un'interfaccia è consistente quando è in grado di garantire che le azioni e i dati inseriti dall'utente siano coerenti con la natura delle elaborazioni effettuate dal programma.

Di norma, un'interfaccia consistente la si progetta e non la si rende tale dopo averla realizzata, e impiegando proprietà ed eventi specifici, ma a questo livello ciò che ci interessa realmente è cominciare a mettere in pratica alcuni degli elementi acquisiti nei paragrafi precedenti allo scopo di rendere il funzionamento di MioPrimoGrado più vicino a quello di un programma realistico.

Data la precedente definizione, l'interfaccia di MioPrimoGrado non è consistente, poiché:

1) consente l'esecuzione del calcolo dell'equazione senza che siano stati inseriti i coefficienti;

consente l'esecuzione del calcolo dell'equazione a prescindere dal fatto che i valori inseriti siano effettivamente di natura numerica.

5.1 Verifica dell'esistenza dei dati

Un requisito fondamentale di qualsiasi interfaccia è quello di garantire che i dati siano stati effettivamente inseriti prima che il programma svolga qualsiasi elaborazione su di essi. Il problema può essere affrontato in due modi distinti, implementando cioè una forma di verifica "anticipata", oppure "posticipata". (In alcuni casi è appropriato implementarle entrambe).

Verifica posticipata dell'esistenza dei dati

Questa verifica è implementata controllando, all'interno del metodo che gestisce l'elaborazione dei dati, se questi esistono davvero. Se la verifica dà esito negativo, l'elaborazione viene abortita e l'utente viene informato che i dati non rispettano i requisiti richiesti dal programma.

```
void btnCalcola_Click(object sender, EventArgs e)
{
    // verifica dell'esistenza dei dati
    if (txtA.Text == "" || txtB.Text == "")
    {
        MessageBox.Show("Uno o entrambi i coefficienti non sono stati inseriti",
                        "MioPrimoGrado",
                        MessageBoxButtons.OK, MessageBoxIcon.Error);
        return;
    }
    // acquisisce i coefficienti dai due TextBox
    ...
}
```

L'implementazione si spiega da sé. Se uno o entrambi i `TextBox` sono vuoti viene visualizzata una message dialog di errore e successivamente viene terminato il metodo.

5.2 Verifica anticipata dell'esistenza dei dati

Questa forma di verifica sfrutta la natura ad eventi delle interfacce grafiche. Lo scopo è quello di impedire all'utente di dare l'avvio all'elaborazione fintantoché tutti i dati non sono stati inseriti; ciò si ottiene abilitando o disabilitando il bottone `btnCalcolaX`. Viene dunque a crearsi una relazione tra lo stato del bottone – abilitato o non abilitato – e il contenuto dei due `TextBox`, che può essere espressa così: `btnCalcolaX` è abilitato solo se `txtA` e `txtB` contengono i dati; altrimenti è disabilitato.

Una simile relazione può essere tradotta in pratica gestendo l'evento `TextChanged` sollevato dai due `TextBox`. Tale evento viene sollevato ogni qual volta viene modificato il contenuto del `TextBox`. Quando avviene una tale modifica, di qualunque natura essa sia, è sufficiente verificare se il contenuto di entrambi i `TextBox` sia o no diverso da stringa nulla. In caso positivo la proprietà `Enabled` di `btnCalcolaX` viene impostata a `true`, altrimenti viene impostata a `false`.

Naturalmente, lo stesso gestore di evento dev'essere attaccato a entrambi i `TextBox`, in modo che qualsiasi modifica venga effettuata ad uno qualsiasi di essi determini l'immediata impostazione di `btnCalcolaX.Enabled`.

Associare lo stesso metodo agli eventi di due o più controlli

Per gestire con lo stesso metodo gli eventi prodotti da due (o più) controlli vi sono varie tecniche. La più semplice consiste nel generare il gestore di evento del primo controllo, usando l'editor degli eventi o eseguendo un doppio clic (se si tratta dell'evento predefinito). Ad esempio, facendo doppio clic sul controllo `txtB` viene generato il gestore dell'eventi `TextChanged`:

```
void txtA_TextChanged(object sender, EventArgs e)
{
}
```

Successivamente si seleziona l'altro controllo e, dall'editor degli eventi, si associa l'evento `TextChanged` al metodo precedentemente generato selezionandolo tra i gestori di eventi disponibili:

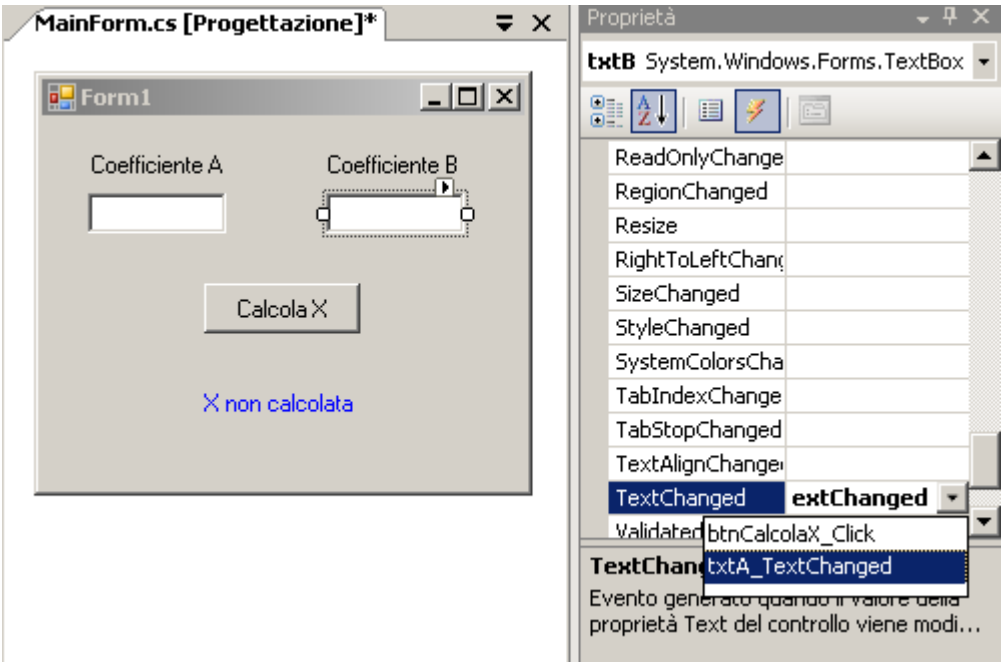


Figura 12-8 Associazione di un evento ad un metodo già esistente.

La possibilità di associare un evento ad un metodo già creato consente di creare manualmente il metodo (eventualmente anche completo di codice) e soltanto successivamente di associarlo all'evento o agli eventi desiderati. In questo senso, l'editor di eventi è abbastanza intelligente da elencare soltanto i gestori di eventi compatibili con il tipo di evento selezionato.

Impostare lo stato di attivazione del bottone

Il codice attivazione/disattivazione del bottone è composto da una sola istruzione:

```
void txtA_TextChanged(object sender, EventArgs e)
{
    btnCalcolaX.Enabled = (txtA.Text != "" && txtB.Text != "");
}
```

L'istruzione assegna alla proprietà booleana enabled il risultato di un'espressione condizionale che risulta vera soltanto se sia il primo che il secondo `TextBox` contengono un valore.

Impostare lo stato iniziale del bottone

La verifica anticipata non è ancora completa. Infatti, subito dopo il lancio del programma i due `TextBox` sono vuoti, ma il bottone è egualmente abilitato. Ciò avviene perché finché l'utente non modifica il contenuto dei `TextBox`, l'evento `TextChanged` non viene sollevato, il gestore dell'evento non viene eseguito e di conseguenza lo stato del bottone non viene aggiornato. Per risolvere questo problema è sufficiente impostare il bottone su uno stato iniziale disattivato.

Ciò può essere fatto sia mediante l'editor delle proprietà, sia scrivendo manualmente il codice necessario e collocandolo nel costruttore del form, dopo l'invocazione al metodo `InitializeComponent()`.

```
public MainForm()
{
    InitializeComponent();
    btnCalcolaX.Enabled = false;
}
```

5.3 Verifica della natura e del valore dei dati

L'avere inserito "qualcosa" da parte dell'utente non è un requisito sufficiente perché possa essere eseguita la parte elaborazione del programma; occorre anche che quel "qualcosa" sia coerente con la natura della elaborazione effettuata. In questo senso, con il termine *verifica della natura* ci si riferisce a una verifica sul tipo dei dati inseriti: valori numerici, numeri telefonici, nominativi, codici fiscali, eccetera; con il termine *verifica del valore* ci si riferisce invece a una verifica sull'appartenenza dei dati a un insieme di valori ammissibili. Sarà presa in considerazione soltanto la prima modalità di verifica.

Verifica posticipata della natura dei dati

Per verificare che il contenuto dei due `TextBox` sia effettivamente di natura numerica è possibile utilizzare il metodo TryParse(), il quale ritorna false se la conversione del valore inserito non ha avuto successo.

```
void btnCalcola_Click(object sender, EventArgs e)
{
```

```
    double a;
    double b;
    if (double.TryParse(txtA.Text, out a) == false ||
        double.TryParse(txtA.Text, out a) == false)
    {
        MessageBox.Show("Formato non valido di uno o entrambi i coefficienti",
                        "MioPrimoGrado",
                        MessageBoxButtons.OK, MessageBoxIcon.Error);
        return;
    }
}
```

Da notare che la verifica del corretto formato dei dati implica anche la verifica della loro esistenza, poiché una stringa vuota non è convertibile in un valore numerico.

5.4 Verifica anticipata sulla natura dei dati

Si possono usare diverse tecniche, una delle quali usa l'evento `Validating`, creato appositamente per la validazione del contenuto dei controlli. Qui ne proponiamo un'altra, però, che sfrutta l'evento `KeyPress` con lo scopo di impedire all'utente di inserire caratteri che non siano coerenti con la natura dei dati, nella fattispecie i caratteri non appartenenti all'insieme "0-9, punto-decimale, segno-meno".

> In realtà questa non rappresenta una garanzia molto robusta, poiché non impedisce l'inserimento di valori del tipo "0…10", "---100", "1-1-2.22", ecc. Ma in questa fase lo scopo non è quello di raggiungere un alto grado di sofisticazione nella progettazione di interfacce, ma di fornire degli esempi che aiutino a comprendere il meccanismo di gestione degli eventi.

Segue l'implementazione del gestore di evento:

```
void txtAeB_KeyPress(object sender, KeyPressEventArgs e)
{
    char ch = e.KeyChar;
    if (ch == (char)Keys.Back) // tasto BACKSPACE
        return;

    if (char.IsDigit(ch) == false && ch != ',' && ch != '-')
        e.Handled = true;
}
```

Nel gestore di evento, dopo aver assegnato alla variabile `ch` il codice del tasto premuto (proprietà `KeyChar` del parametro informazioni evento), viene verificato se questo corrisponde al tasto BACKSPACE, rappresentato dalla costante `Keys.Back`: in questo caso, infatti, il metodo deve lasciare che il controllo elabori il tasto. In tutti gli altri casi viene verificato se il carattere è coerente con un valore numerico; in caso contrario, la proprietà `Handled` del parametro informazioni evento viene impostata a `true`, in pratica inducendo il controllo a scartare il carattere.

Ci sono due osservazioni da fare:

- per verificare se un carattere appartiene all'insieme `'0'` – `'9'` viene fatto uso del metodo statico `IsDigit()`, appartenente al `char`. Questo ritorna `true` se l'argomento rientra nell'intervallo `'0'` – `'9'`, altrimenti ritorna `false`.

- il punto decimale è rappresentato dal carattere virgola. Ciò dipende dalla cultura impostata a livello di sistema operativo. In Italia per il punto decimale viene impiegata la virgola, mentre il punto viene utilizzato come separatore delle migliaia.

Anche in questo caso, occorre attaccare il gestore di evento ad entrambi i `TextBox`:

Attaccare il gestore all'evento KeyPress dei due controlli

Anche in questo caso, lo stesso gestore è associato all'evento `KeyPress` di due controlli. Guardando il codice si nota che il nome del metodo non rispetta la forma consueta; infatti è stato generato con una tecnica leggermente diversa da quelle viste finora.

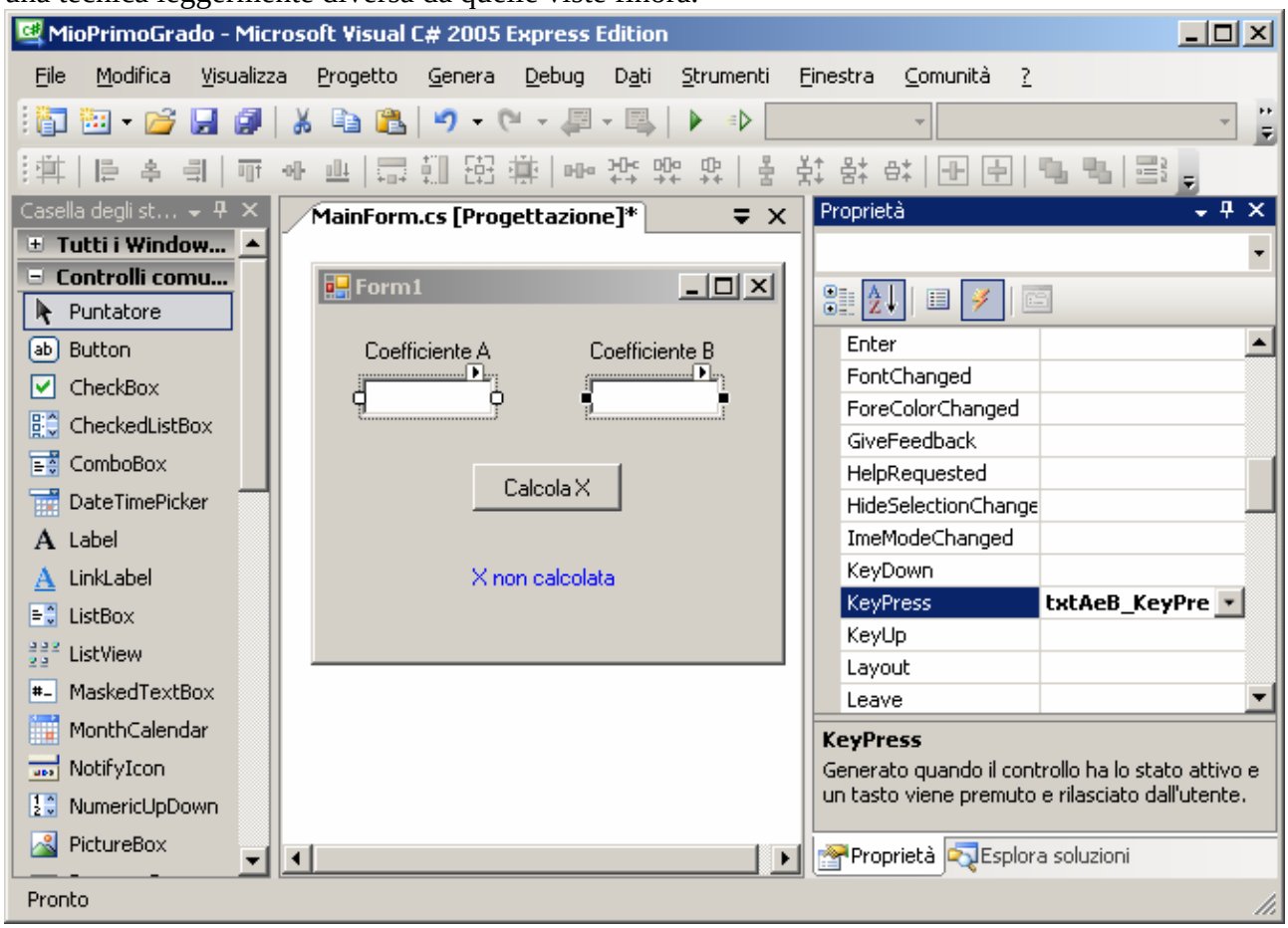


Figura 12-9 Gestione dell'evento KeyPress di due TextBox.

Prima di tutto, prima di generare il gestore, sono stati selezionati entrambi i controlli, in modo da eseguire l'operazione una sola volta. Seconda cosa, è stato scelto un nome del metodo personalizzato, scrivendolo sulla casella adiacente a quella contenente l'evento.

13

Migliorare la comunicazione con l'utente

1 Premessa

Nonostante la potenza delle interfacce grafiche, l'impiego dei soli controlli `Label` e `TextBox` per la visualizzazione e l'acquisizione dei dati non consente di implementare un modello di comunicazione particolarmente più sofisticato rispetto a quello che si può ottenere attraverso l'interfaccia di un'Applicazione Console. Ciò dipende dalla natura dei suddetti controlli, ognuno dei quali consente di visualizzare e acquisire un solo valore per volta. Ciò che è necessario per aumentare la qualità della comunicazione con l'utente è la possibilità di:

- gestire collezioni di dati, offrendo all'utente la possibilità di scegliere uno tra un insieme di valori e/o di aggiungere e togliere valori alla collezione;
- offrire all'utente la possibilità di selezionare una o più tra un insieme di alternative;
- visualizzare elementi grafici (immagini, disegni, sfondi) per migliorare l'estetica e la comunicatività dell'interfaccia.

2 Gestire collezioni di dati

Due tipi di controllo, `ListBox` e `ComboBox`, consentono di gestire collezioni di dati unidimensionali. Con il termine gestire si intende:

- visualizzare una lista di valori;
- aggiungere e togliere un valore o un insieme di valori;
- cancellare tutti i valori della lista;
- popolare la lista di valori mediante l'assegnazione di una collezione, ad esempio un vettore;
- eseguire ricerche sugli elementi;
- mantenere la lista ordinata;

Entrambi i tipi di controllo condividono queste e altre caratteristiche. Il tipo `ComboBox`, inoltre, aggiunge le funzionalità di un `TextBox` e dunque combina le caratteristiche di due tipi di controlli (`ComboBox` sta appunto per "casella combinata").

3 Classe ListBox

I controlli `ListBox` hanno la caratteristica di mantenere internamente una collezione dinamica di elementi, ai quali si può accedere usando la stessa sintassi impiegata per i vettori. L'uso più comune prevede che l'utente selezioni con il mouse uno o più elementi dalla lista, determinando quindi le successive elaborazioni del programma.

Proprietà

Tabella 13-1 Proprietà della classe ListBox.

PROPRIETÀ	DESCRIZIONE
DataSource `object`	Mediante questa proprietà è possibilepopolare il `ListBox` con i valori di una collezione.
Items `object[]`	`Items` è la proprietà attraverso la quale si può accedere agli elementi della lista. Essa espone a sua volta proprietà e metodi attraverso i quali aggiungere e togliere elementi, conoscere il numero degli elementi, eccetera: `Items.Count`: memorizza il numero degli elementi della lista; `Items.Add(object)`: aggiunge un elemento alla lista; `Items.AddRange(object[])`: aggiunge una sequenza di elementi; `Items.RemoveAt(int)`: rimuove un elemento dalla lista; `Items.Clear()`: rimuove tutti gli elementi dalla lista.
SelectedIndex `int`	Indice dell'elemento correntemente selezionato. (Tale elemento viene di norma visualizzato in bianco su blu.) Se è non selezionato alcun elemento, `SelectedIndex` vale –1.
SelectedItem `object`	Riferimento all'elemento correntemente selezionato. Il riferimento è di tipo `object`.
SelectionMode `SelectionMode`	Definisce il tipo di selezione ammissibile: `None`: non è possibile selezionare alcun elemento; `One`: è possibile selezionare un solo elemento per volta; `MultiSimple`: è possibile selezionare più elementi; `MultiExtended`: è possibile selezionare più elementi; l'utente può usare i tasti CTRL e SHIFT e tasti freccia per eseguire la selezione.
Sorted `bool`	Indica se la lista debba o meno essere mantenuta ordinata. Il valore di default è `false` (lista non ordinata).
Text `string`	Vedi più avanti il paragrafo "Uso della proprietà Text"

Eventi

Del nutrito insieme di eventi pubblicato dalla classe `ListBox` vengono comunemente gestiti quelli che rispondono alla selezione di un elemento da parte dell'utente.

Tabella 13-2 Eventi della classe ListBox.

EVENTO	DESCRIZIONE
Click **DoubleClick**	Entrambi vengono sollevati dopo che l'utente ha cliccato (o eseguito un doppio clic) sull'area del `ListBox`. Se il clic (o il doppio clic) avviene su un elemento questo diventa selezionato prima che l'evento sia sollevato.
SelectedIndexChanged	Viene sollevato dopo che è stato selezionato un elemento diverso da quello corrente.

3.1 Popolare un ListBox

Con il verbo "popolare" si intende aggiungere gli elementi alla lista mantenuta dal controllo. Nel suo impiego più comune un `ListBox` viene inizialmente popolato con una lista di dati dei quali l'utente potrà in seguito selezionare quello desiderato in relazione a una determinata operazione. E' possibile popolare un `ListBox` attraverso due modalità distinte:

- assegnando una collezione alla proprietà `DataSource`;
- aggiungendo elementi attraverso i metodi `Items.Add()` e `Items.AddRange()`.

3.2 Popolare un ListBox attraverso l metodi Add() e AddRange()

Il metodo `Add()` della proprietà `Items` consente di aggiungere un elemento in coda al `ListBox`. Ad esempio, il seguente codice aggiunge ad esso tre elementi al `ListBox lboNomiFamosi`:

```
lboNomiFamosi.Items.Add("Einstein");
lboNomiFamosi.Items.Add("Turing");
lboNomiFamosi.Items.Add("Newton");
```

L'argomento del metodo `Add()` può essere di qualsiasi tipo e ciò implica che la lista può essere composta da elementi di tipo diverso[40]:

```
lboNomiFamosi.Items.Add("Mussolini");     // aggiunge una stringa
lboNomiFamosi.Items.Add(200.2);           // aggiunge un double
```

Il funzionamento del metodo `AddRange()` è analogo a quello di `Add()` con la differenza che il primo è in grado di aggiungere più elementi contemporaneamente. Ad esempio:

```
string[] filosofi = {"Socrate", "Platone", "Kant", "Popper"};
lboNomiFamosi.Items.AddRange(filosofi);
```

Un limite di questo tecnica è dovuto al fatto che gli elementi del vettore devono essere di tipo riferimento e dunque non possono essere `double`, `int`, `char`, ecc. E' dunque sbagliato scrivere:

```
int[] altezze = {180, 190, 170);
lboAltezze.Items.AddRange(altezze);
```

Per questo motivo, se si desidera popolare il `ListBox` con un vettore di elementi di tipo valore occorre iterare attraverso il vettore e aggiungere ogni elemento con `Add()`:

```
int[] altezze = {180, 190, 170};
foreach(int altezza in altezze)
    lboAltezze.Items.Add(altezza);
```

[40] Ovviamente l'esempio mostrato non rappresenta un caso comune.

3.3 Popolare un ListBox mediante la proprietà DataSource

Per popolare un ListBox con questa tecnica è sufficiente assegnare la collezione alla proprietà DataSource; ciò gli eventuali elementi esistenti siano rimossi e siano aggiunti tutti gli elementi della collezione. Ad esempio, il seguente codice popola un `ListBox` con un vettore di stringhe:

```
string[] fisici = {"Einstein", "Fermi", "Hawking", "Bohr"};
...
lboNomiFamosi.DataSource = fisici;
```

Dopo la precedente assegnazione *è proibito aggiungere, rimuovere o modificare elementi*. Per rendere questo possibile è necessario prima assegnare `null` alla proprietà `DataSource`. Ad esempio:

```
lboNomiFamosi.DataSource = null;          // ora posso modificare la lista
...
lboNomiFamosi.Items[0] = "Plank";         // modifica del primo elemento
```

E' importante comprendere che assegnare `null` a `DataSource` non determina la cancellazione degli elementi esistenti, ma soltanto la possibilità che questi possono essere modificati, aggiunti o eliminati.

Un altro aspetto importante riguarda la possibilità di modificare gli elementi del `ListBox` agendo sulla collezione usata per popolarlo. Semplicemente, ciò non produce alcun effetto, come dimostra il seguente codice:

```
string[] nomiFisici = {"Einstein", "Fermi", "Hawking", " Bohr "};
...
lboNomiFamosi.DataSource = nomiFisici;
...
nomiFisici[0] = "Plank";      // Il ListBox resta invariato;
```

E nemmeno il codice che segue produce effetti sul `ListBox`:

```
string[] fisici = {"Einstein", "Fermi", "Hawking", "Bohr"};
...
lboNomiFamosi.DataSource = fisici;
...
fisici[0] = "Plank"; // modifica del primo elemento del vettore
lboNomiFamosi.DataSource = fisici;
```

Infatti, il `ListBox` viene popolato soltanto se a `DataSource` viene assegnato un riferimento diverso da quello attualmente memorizzato. E' questo il caso in cui cui viene assegnato un nuovo vettore:

```
lboNomiFamosi.DataSource = fisici;
...
string[] matematici = {"Gauss", "Poincarè", "Hilbert"};
lboNomiFamosi.DataSource = matematici;   // ListBox ripopolato
```

D'altra parte è possibile ripopolare un `ListBox` usando lo stesso vettore (modificato o meno che sia) purché prima di farlo venga assegnato `null` a `DataSource`:

```
string[] fisici = {"Einstein", "Fermi", "Hawking", "Bohr"};
...
lboNomiFamosi.DataSource = fisici;
...
```

```
fisici[0] = "Plank"; // modifica del primo elemento del vettore
lboNomiFamosi.DataSource = null;
lboNomiFamosi.DataSource = fisici; // ListBox ripopolato
```

Infine, diversamente dal metodo `AddRange()`, attraverso la proprietà `DataSource` è possibile popolare un `ListBox` anche con vettori i cui elementi siano di tipo valore:

```
int[] altezze = {180, 190, 170};
lboAltezze.DataSource = altezze;          // ok
```

3.4 Accesso agli elementi di un ListBox

Gli elementi di un `ListBox` sono memorizzati in una collezione di `object`; ciò fa sì che questo possa essere popolato con valori di qualsiasi natura. D'altra parte, quando si accede ad essi occorre di norma applicare l'operatore di cast per convertirli nel tipo appropriato.

Ad esempio, il seguente codice scandisce gli elementi di un `ListBox` e li copia in un vettore di stringhe (il `ListBox` si suppone essere già stato popolato in precedenza):

```
string[] grandiPoeti = new string[lboNomiFamosi.Items.Count];
for (int i = 0; i < lboNomiFamosi.Items.Count; i++)
    grandiPoeti[i] = (string) lboNomiFamosi.Items[i]; // uso del cast
```

Ovviamente, tutto ciò funziona se gli elementi del `ListBox` sono effettivamente delle stringhe. Non è questo il caso del seguente codice:

```
int[] altezze = {180, 190, 170};
lboNomiFamosi.DataSource = altezze;
...
string[] grandiPoeti = new string[lboNomiFamosi.Items.Count];
for (int i = 0; i < lboNomiFamosi.Items.Count; i++)
    grandiPoeti[i] = (string) lboNomiFamosi.Items[i]; // uso errato del cast
```

In questo caso il `ListBox` contiene una collezione di interi e dunque il tentativo di cast al tipo `string` produce un errore di esecuzione.

Per inciso, in questo caso avremmo comunque potuto ottenere un vettore di stringhe invocando il metodo `ToString()` per ogni elemento, come nel seguente codice:

```
int[] altezze = {180, 190, 170};
lboNomiFamosi.DataSource = altezze;
...
string[] grandiPoeti = new string[lboNomiFamosi.Items.Count];
for (int i = 0; i < lboNomiFamosi.Items.Count; i++)
    grandiPoeti[i] = lboNomiFamosi.Items[i].ToString()
```

3.5 Uso della proprietà Text

Anche il controllo `ListBox` definisce la proprietà `Text`, mediante la quale è possibile ottenere una rappresentazione stringa dell'elemento selezionato. In alcune situazioni, dunque, può essere usata come sostituto della proprietà `SelectedItem`. Ad esempio, il seguente gestore di evento :

```
void lboNomiFamosi_SelectedIndexChanged(object sender, EventArgs e)
{
    MessageBox.Show(lboNomiFamosi.Text);
```

```
}
```

mostra con una message dialog l'elemento selezionato.

La proprietà `Text` può essere usata anche per modificare l'elemento attualmente selezionato. Ciò si ottiene assegnando alla proprietà il valore in formato stringa dell'elemento che si vuole selezionare. Ad esempio, si supponga di aver popolato il `ListBox` con il seguente codice:

```
string[] fisici = {"Einstein", "Fermi", "Hawking", "Bohr"};
lboNomiFamosi.DataSource = fisici;
```

E si supponga inoltre che `"Einstein"` sia l'elemento selezionato. Assegnando alla proprietà `Text` il valore `"Fermi"`:

```
lboNomiFamosi.Text = "Fermi";
```

si ottiene il risultato di selezionare il secondo elemento del `ListBox`.

Nel caso in cui nessun elemento del `ListBox` abbia una rappresentazione stringa equivalente a quella assegnata alla proprietà `Text`, l'elemento selezionato resta quello attuale. Ad esempio, la seguente istruzione:

```
lboNomiFamosi.Text = "Plank";
```

lascia la situazione invariata, poiché non esiste un elemento di valore `"Plank"` nel ListBox.

3.6 Accesso all'elemento selezionato

Il seguente esempio mostra come gestire l'evento `SelectedIndexChanged` allo scopo di visualizzare l'lemento correntemente selezionato.

```
public partial MainForm: Form
{
    string[] grandiPoeti = {"ShakeSpeare", "Dante", "Keats", "Leopardi"}

    public MainForm()
    {
        InitializeComponent();
        lboGrandiPoeti.DataSource = grandiPoeti;  // popola il ListBox
    }

    void lboGrandiPoeti_SelectedIndexChanged(object sender, EventArgs e)
    {
        if (lboGrandiPoeti.SelectedIndex != -1)
            lblPoetaSelezionato.Text = lboGrandiPoeti.Text;
        else
            lblPoetaSelezionato.Text = "Nessun elemento selezionato";
    }
}
```

Vi sono alcune osservazioni da fare.

Prima di ogni altra cosa viene verificato che vi sia un elemento selezionato, testando il valore della proprietà `SelectedIndex`. E' una verifica fondamentale, poiché non esiste nessuna garanzia in tal senso. A tal proposito, potrebbe non esistere un selemento selezionato per i seguenti motivi:

- il ListBox è vuoto;
- l'utente non ancora selezionato nessun elemento;
- il ListBox è stato appena popolato (o ripopolato);
- l'elemento correntemente selezionato è stato appena rimosso.

Un secondo aspetto riguarda la tecnica di accesso all'elemento selezionato. Nell'esempio è stata utilizzata la proprietà `Text`. L'uso di `SelectedItem` sarebbe andato altrettanto bene, anche se trattandosi di elemento di tipo stringa, avrebbe complicato un po' il codice:

```
lblPoetaSelezionato.Text = (string) lboGrandiPoeti.SelectedItem;
```

4 Classe ComboBox

Un `ComboBox` unisce le caratteristiche di un `ListBox` a quelle di un `TextBox`. Il controllo interagisce con l'utente sia attraverso il mouse (caratteristica tipica del `ListBox`) sia attraverso la tastiera (caratteristica tipica del `TextBox`). Tipicamente, l'utente può:

- cliccare sulla freccia per visualizzare la lista degli elementi (**elenco a discesa**) e quindi selezionare un elemento; l'elemento selezionato viene visualizzato nella casella di testo;
- digitare all'interno della casella di testo.

Proprietà

Di seguito sono elencare alcune proprietà che caratterizzano il `ComboBox` rispetto al `ListBox`.

Tabella 13-3 Proprietà della classe ComboBox.

PROPRIETÀ	DESCRIZIONE
`DropDownStyle` `ComboBoxStyle`	Questa proprietà caratterizza il comportamento del `ComboBox` e dunque la particolare funzione per il quale viene inserito nell'interfaccia. I possibili valori sono: `Simple`: L'utente può digitare nella casella oppure selezionare un elemento dall'elenco a discesa, il quale è sempre visibile. (Dunque non è presente la freccia per visualizzarlo). `DropDown`: L'utente può digitare nella casella oppure selezionare un elemento dall'elenco a discesa, il quale viene visualizzato cliccando sulla freccia. `DropDownList`: L'utente non può digitare nella casella e dunque può soltanto selezionare un elemento dall'elenco a discesa, il quale viene visualizzato cliccando sulla freccia.
`MaxDropDownItems` `int`	Massimo numero di elementi visualizzati contemporaneamente nell'elenco a discesa.
`MaxTextLength`	Entrambe svolgono la stessa funzione delle proprietà omologhe esposte dalla classe `TextBox`.

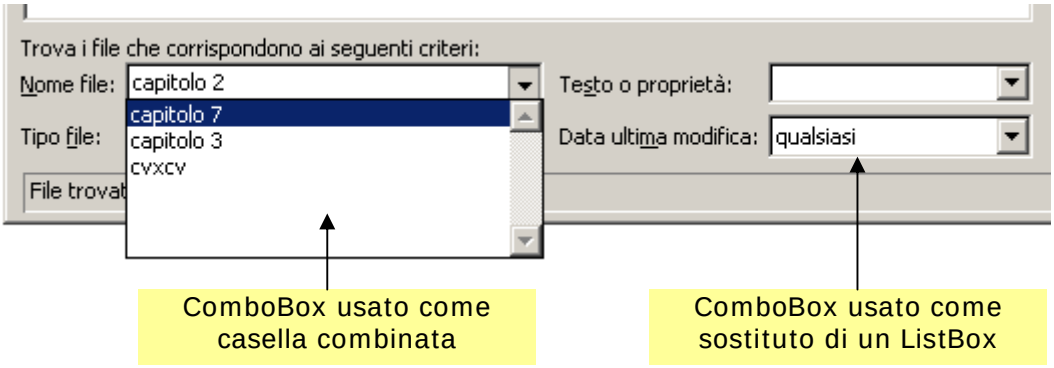
Eventi

Gli eventi pubblicati dalla classe `ComboBox` sono sostanzialmente gli stessi delle classi `ListBox` e `TextBox`. Quelli gestiti più comunemente sono: `TextChanged`, `Click`, `DoubleClick`, `SelectedIndexChanged`.

4.1 Uso del controllo ComboBox

Un primo impiego dei `ComboBox` è quello di sostituto dei `ListBox` laddove è richiesta la possibilità di selezionare un elemento da una lista ma lo spazio nel form non è sufficiente perché la lista sia completamente visibile. L'impiego classico prevede che l'utente possa inserire un valore da tastiera oppure selezionarne uno tra quelli disponibili nella lista. In ogni caso, si può accedere all'elemento selezionato, o al testo digitato, attraverso le proprietà `Text` e `SelectedItem`, analogamente a quanto avviene con i `TextBox`.

Figura 13-1 Esempio d'impiego di controlli ComboBox nella dialog di apertura file in MS Word.



5 RadioButton e CheckBox

I controlli di tipo `RadioButton` e `CheckBox` arricchiscono ulteriormente il modello di comunicazione con l'utente, consentendo a quest'ultimo di selezionare una o più tra un insieme di opzioni disponibili.

5.1 Classe «RadioButton»

Come un'interrutore, che può trovarsi nello stato acceso o spento, analogamente, un `RadioButton` può assumere due soli stati: *cheked* o *uncheked*. Un `RadioButton` funziona sempre in collaborazione con altri `RadioButton`, insieme ai quali definisce la lista dei possibili stati che può assumere un determinato valore; in ogni momento uno solo tra i `RadioButton` può essere *checked*, determinando dunque lo stato del valore in questione.

> Per questo motivo, benché non rappresenti in sé un errore formale, inserire nel form un solo `RadioButton` è semplicemente privo di senso.

Proprietà

La proprietà più importante di un `RadioButton` è la proprietà `Checked`, che ne memorizza lo stato: **marcato** (`true`), **non marcato** (`false`).

Eventi

Di norma non è necessario gestire gli eventi sollevati dai `RadioButton`, poiché essi elaborano autonomamente le azioni dell'utente, impostando in modo appropriato il proprio stato in relazione allo stato degli altri `RadioButton`. E' invece necessario gestire gli eventi `Click` o `CheckedChanged` se si desidera che lo stato di uno o più controlli venga aggiornato immediatamente in relazione al cambiamento dello stato del `RadioButton` in questione.

5.2 Uso del controllo RadioButton

Un insieme di `RadioButton` consente di definire la lista dei possibili valori di un'opzione, dei quali soltanto uno può essere selezionato in un dato momento. Diversamente dalla lista di elementi mantenuta da un `ListBox` o un `ComboBox`, il numero dei valori è fisso e corrisponde al numero di `RadioButton`.[41]

Il programma che segue ipotizza la richiesta di dati per la prenotazione di un volo aereo. Nella fattispecie viene chiesto all'utente di specificare il nome, mediante un `TextBox`, e la classe – prima classe o classe economica –, mediante due `RadioButton`. Il programma si limita a elaborare lo stato dei RadioButton, senza gestire i loro eventi.

```
public partial class MainForm: Form
{
    public MainForm()
    {
        InitializeComponent();
    }

    void btnConferma_Click (object sender, EventArgs e)
    {
        string s;
        if (rbuPrimaClasse.Checked == true)
            s = " -- Prima classe: confermata";
        else
            s = " -- Classe economica: confermata";
        MessageBox.Show("Cliente: " + txtNome.Text + s, "Conferma prenotazione");

    }
}
```

L'esecuzione del programma e produce il seguente l'output

[41] Nulla impedisce di aggiungere (o togliere) RadioButton in risposta a una qualche elaborazione o alle azioni dell'utente. Di fatto, questo rappresenta un impiego del tutto inappropriato dei RadioButton, che si prestano invece alla gestione di un elenco fisso di opzioni.

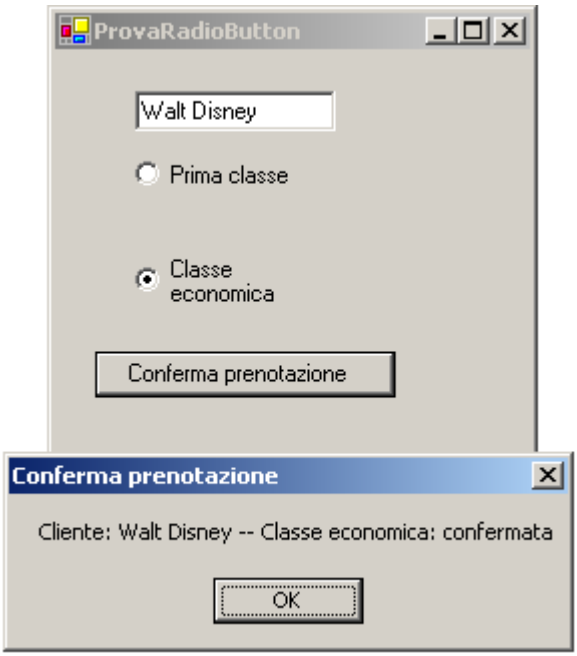


Figura 13-2 Output prodotto dal programma.

5.3 Controllo «CheckBox»

Anche un `CheckBox` svolge la funzione di un interruttore, il quale può trovarsi soltanto nello stato marcato o non marcato.

> In realtà, per i CheckBox è previsto un terzo stato, **indeterminato**.

Diversamente dai `RadioButton`, un `CheckBox` non agisce in collaborazione con altri `CheckBox`; esso, infatti, definisce lo stato, *checked* o *unchecked*, di una opzione che è indipendente da eventuali altre opzioni presenti nell'interfaccia.

Proprietà

Analogamente al controllo `RadioButton`, la classe `CheckBox` definisce la proprietà `Checked`, attraverso la quale è possibile conoscere e impostare il suo stato: marcato (`true`), non marcato (`false`).

Eventi

Analogamente al controllo `RadioButton`, di norma non è necessario gestire gli eventi sollevati da un `CheckBox`. E' comunque appropriato gestire gli eventi `Click` o `CheckedChanged` se si desidera che lo stato di uno o più controlli venga aggiornato immediatamente in relazione al cambiamento dello stato del `CheckBox` in questione.

5.4 Uso del controllo «CheckBox»

Laddove un insieme di `RadioButton` definisce i possibili stati ammissibili per un determinato valore, ogni `CheckBox` definisce lo stato di un valore a sé stante, che può essere.

Il programma che segue consente all'utente di modificare il valore di due proprietà di un `TextBox` – `Visible` e `ReadOnly` – impostando lo stato dei due `CheckBox` corrispondenti. Per fare in modo l'aggiornamento delle due proprietà direttamente sia collegato al cambiamento dello stato dei due `CheckBox`, il programma gestisce l'evento `CheckedChanged` di entrambi i `CheckBox`.

```
public partial class MainForm : Form
{
    public MainForm()
    {
        InitializeComponent();
    }

    void chkEnabled_CheckedChanged (object sender, EventArgs e)
    {
        txtNome.Enabled = chkEnabled.Checked;
    }

    void chkVisible_CheckedChanged (object sender, EventArgs e)
    {
        txtNome.Visible = chkVisible.Checked;
    }
}
```

Sotto è mostrato l'output del programma.

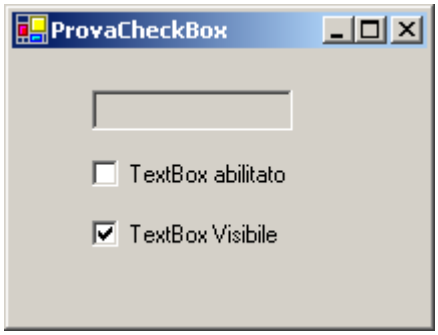


Figura 13-3 Output prodotto dal programma.

6 Controlli "container"

Esistono dei controlli il cui ruolo non è quello di interagire con l'utente ma di fungere da contenitori per gli altri controlli. Come contenitori, entrambi svolgono una funzione analoga a quella di un form, cioè "ospitano" altri controlli e possono influenzare il loro aspetto e il loro comportamento.

Esistono svariate ragion per raggruppare dei controlli inserendoli in un controllo container:

- creare, mediante un riquadro o un particolare colore di sfondo, delle aree visivamente separate dal resto dell'interfaccia;
- facilitare la gestione di controlli logicamente correlati tra loro. Ad esempio, impostando a `false` la proprietà `Enabled` di un controllo container vengono automaticamente disabilitati tutti i controlli in esso contenuti;
- creare insiemi di `RadioButton` funzionalmente indipendenti tra loro;

- rendere possibile il ridemnsionamento di una o più aree del form e la realita disposizione dei controlli in esse contenuti.

I controlli container sono `FlowLayoutPanel`, `GroupBox`, `Panel`, `SplitContainer`, `TabControl`, `TableLayoutPanel`. Di questi prenderemo in considerazione i controlli `Panel` e `GroupBox`, i quali hanno modalità di impiego analoghe, anche se presentano alcune differenze nell'aspetto e nel comportamento.

6.1 Classe GroupBox

Il controllo `GroupBox` viene comunemente impiegato come contenitore per `RadioButton`, consentendo così di gestire insiemi di `RadioButton` indipendenti tra loro. Infatti, all'interno di un controllo contenitore, ad esempio un form, un solo `RadioButton` per volta può trovarsi nello stato marcato; dunque, se è necessario gestire due valori distinti, i cui stati sono rappresentati attraverso due insiemi di `RadioButton`, è necessario collocare tali insiemi in contenitori a loro volta distinti.

Il programma che segue, attraverso due `GroupBox`, gestisce due insiemi di `RadioButton` che determinano il case e l'allineamento del testo di un `TextBox`.

```
public partial class MainForm: Form
{
    public MainForm()
    {
        InitializeComponent();
    }

    void Case_CheckedChanged (object sender, EventArgs e)
    {
        if (rbuMaiusc.Checked == true)
            txtEsempio.CharacterCasing = CharacterCasing.Upper;
        else
            txtEsempio.CharacterCasing = CharacterCasing.Lower;
    }

    void Align_CheckedChanged (object sender, EventArgs e)
    {
        if (rbuSinistra.Checked == true)
            txtEsempio.TextAlign = HorizontalAlignment.Left;
        else
            txtEsempio.CharacterCasing = HorizontalAlignment.Left;
    }

}
```

Di seguito è mostrato l'output del programma.

Figura 13-4 Output del programma.

Ci sono alcuni commenti da fare. Innanzitutto gli eventi `Checked` dei controlli `rbuMaiusc`, `rbuMinusc` da una parte e `rbuSinistra`, `rbuDestra` dall'altra sono gestiti mediante gli stessi gestori di evento. Seconda cosa, i quattro RadioButton sono inseriti all'interno dei due GroupBox; in caso contrario non potrebbero svolgere correttamente la loro funzione. Ciò è dimostrato dal seguente frammento di codice, collocato nel metodo InitializeComponent():

```
private void InitializeComponent()
{
    ...
    this.grpCase.Controls.Add(this.rbuMinusc);
    this.grpCase.Controls.Add(this.rbuMaiusc);
    ...
    this.grpAlign.Controls.Add(this.rbuDestra);
    this.grpAlign.Controls.Add(this.rbuSinistra);
    ...

}
```

Come si vede, il controllo GroupBox, come qualsiasi controllo container, definisce la proprietà `Controls`, attraverso la quale ospitare i controlli figli.

6.2 Classe Panel

Il `Panel` è un container generico, utilizzato normalmente per facilitare la disposizione e l'aggiornamento di gruppi di controlli logicamente correlati. D'altra parte, il Panel può essere utile anche nei casi in cui un'area del form è troppo piccola per i controlli che deve ospitare. Il `Panel` è infatti dotato di barre due di scorrimento, una verticale l'altra orizzontale, che gli consentono di rappresentare un'area più grande di quella effettivamente visualizzabile. Per abilitare il `Panel` all'impiego delle barre di scorrimento è necessario impostare a `true` la proprietà `AutoScroll`.

Il programma di esempio che segue non sfrutta comunque questa caratteristica; esso mostra come utilizzare un `Panel` per abilitare o disabilitare un gruppo di controlli in relazione allo stato di un `CheckBox`.

```
public partial class MainForm : Form
{
    public MainForm()
    {
        InitializeComponent();
```

```
    }

    void chkLaureato_CheckedChanged (object sender, EventArgs e)
    {
        pnlLaurea.Enabled = chkLaureato.Checked;
    }

}
```

Di seguito è mostrato l'output del programma:

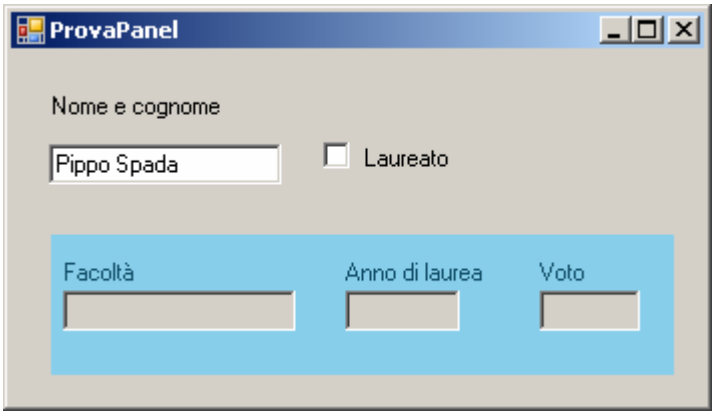


Figura 13-5 Output del programma.

Il programma abilita l'inserimento dei dati universitari soltanto se l'utente si dichiara laureato. L'uso del `Panel` semplifica l'operazione, riducendola a una sola istruzione, invece delle tre che sarebbero necessarie.

7 Gestire le immagini

Le immagini sono parte integrante delle interfacce moderne. Esse possono svolgere svariati ruoli:

- come informazioni da rappresentare ed elaborare; si pensi a un archivio fotografico, alle foto dei dipendenti di un archivio aziendale, all'oggetto di elaborazione di un programma di fotoritocco, ecc;
- per caratterizzare i controlli dell'interfaccia: le icone che caratterizzano i bottoni, i menù, le message dialog, ecc;
- come elemento dell'interfaccia che risponde alle azioni dell'utente;
- come elemento decorativo;

Esistono svariati controlli che consentono di visualizzare immagini, la maggior parte dei quali semplicemente come elemento che caratterizza l'aspetto di un controllo, ma che non ne influenza le funzionalità. Esiste invece un particolare controllo, il `PictureBox`, il cui solo scopo è appunto quello di visualizzare un'immgine.

7.1 Classe PictureBox

Il controllo `PictureBox` è fondamentalmente caratterizzato da due proprietà:

- l'immagine da visualizzare;
- la modalità di visualizzazione dell'immagine.

Proprietà

Tabella 13-4 Proprietà della classe PictureBox.

PROPRIETÀ	DESCRIZIONE
`Image` `Image`	Definisce l'immagine da visualizzare. Un modo per impostare il valore della proprietà e quello di specificare il file che contiene l'immagine mediante il metodo `FromFile` della classe `Image`: `picEsempio.Image = Image.FromFile("nomeicona.bmp");`
`SizeMode` `PictureBoxSizeMode`	Definisce il modo in cui l'immagine viene visualizzata: `Normal`: l'immagine è visualizzata a partire dall'angolo in alto a sinistra dell'area del controllo. La parte che eccede tale area viene tagliata; `StretchImage`: l'immagine viene ridimensionata per coprire per intero l'area del controllo; `AutoSize`: il controllo modifica le proprie dimensioni in relazione a quelle dell'immagine; `CenterImage`: l'immagine è centrata rispetto all'area del controllo. Le parti di immagine che eventualmente eccedono tale area vengono tagliate.

Eventi

Di norma non è necessario gestire gli eventi sollevati da questo controllo. In alcune situazioni, il `PictureBox` può essere utilizzato come un bottone, nel qual cosa occorre gestire l'evento `Click`.

7.2 Uso del controllo PictureBox

Nel programma che segue, due `PictureBox` visualizzano la stessa immagine, usando diverse modalità di rappresentazione.

```
public partial class MainForm : Form
{
    public MainForm()
    {
        InitializeComponent();

        picA.Image = Image.FromFile(@"immagini\atleta.jpg");
        picA.SizeMode = PictureBoxSizeMode.StretchImage;

        picB.Image = Image.FromFile(@"immagini\atleta.jpg");
        picB.SizeMode = PictureBoxSizeMode.CenterImage;
    }

    void btnScambia_Click(object sender, EventArgs e)
    {
```

```
        if (picA.SizeMode == PictureBoxSizeMode.StretchImage)
            picA.SizeMode = PictureBoxSizeMode.CenterImage;
        else
            picA.SizeMode = PictureBoxSizeMode.StretchImage;

        if (picB.SizeMode == PictureBoxSizeMode.StretchImage)
            picB.SizeMode = PictureBoxSizeMode.CenterImage;
        else
            picB.SizeMode = PictureBoxSizeMode.StretchImage;
    }

}
```

Di seguito viene mostrato l'ouput del programma.

Figura 13-6 Output del programma.

Nota bene: le due proprietà principali sono state impostate nel costruttore del form e non mediante il designer, come consuetudine.

Allo scopo di rendere evidente la differenza di rappresentazione dell'immagine in relazione al valore della proprietà `SizeMode`, il programma, mediante il bottone `btnScambia`, consente all'utente di scambiare tra loro le modalità di visualizzazione impiegate dai due `PictureBox`. In figura è mostrato l'output del programma prima e dopo che l'utente ha cliccato sul bottone:

8 Menu

Uno degli elementi che ricorrono sistematicamente nella maggior parte delle Applicazioni Windows è il sistema a menù, attraverso il quale è di possibile accedere a tutte le funzionalità dell'applicazione. In .NET esistono due tipi di menu:

- il **sistema dei menù principale**, gestito attraverso la classe `MenuStrip` e visualizzato sotto il titolo. Di norma esiste uno solo sistema dei menù principale per ogni applicazione.

- il **menu contestuale** o di **scelta rapida**, gestito attraverso la classe `ContextMenuStrip`. I menù di scelta rapida sono associati ai controlli e sono accessibili mediante il clic sul pulsante destro del mouse.

In questo paragrafo ci occuperemo del menù principale.

8.1 Disegno del sistema di menu dell'applicazione

Il menù principale di una applicazione è strutturato in un sistema di menu e sotto menù di natura gerarchica. Alla base del sistema sta la **barra dei menù**, normalmente posizionata sotto la barra del titolo della finestra; essa contiene le voci che identificano i menù principali della gerarchia. Questi definiscono una o più elementi, ognuno dei quali può rappresentare un comando oppure un sottomenù, e così via.

Segue uno screen shot del programma MS Word che mostra il sistema dei menù.

Figura 13-7 Screen shot che mostra il sistema di menù di Microsoft Word.

La realizzazione di un sistema di menù è enormemente facilitata dal designer di menu disponibile in Visual C# Express e non richiede la scrittura di codice da parte del programmatore.

Il designer di menù diventa disponibile dopo aver trascinato il controllo `MenuStrip` sul form. A questo punto, il designer propone una vista wysiwyg del sistema di menù che, unita all'editor delle proprietà, consente di impostarne e modificarne tutte le caratteristiche: nome degli elementi, testo visualizzato, tipo dell'elemento, tasti acceleratori, icone, ecc.

La figura a pagina successiva mostra un sistema di menù in fase di costruzione. E' stato creato il menù "File", il quale contiene già due voci: "Nuovo" e "Apri". Digitando alla destra della voce "File" è possibile aggiungere un nuovo menù alla barra; mentre digitando nella casella sotto "Apri" si aggiunge un nuovo elemento al menù File.

Alternativamente, è possibile gestire la costruzione di un menù selezionando il comando "Modifica elementi" dello smart tag associato all'oggetto. Ciò rende disponibile l'editor di menù (mostrato nella seconda figura a pagina successiva), il quale fornisce le stesse funzionalità del designer e incorpora l'elenco delle proprietà dell'elemento di menù correntemente selezionato.

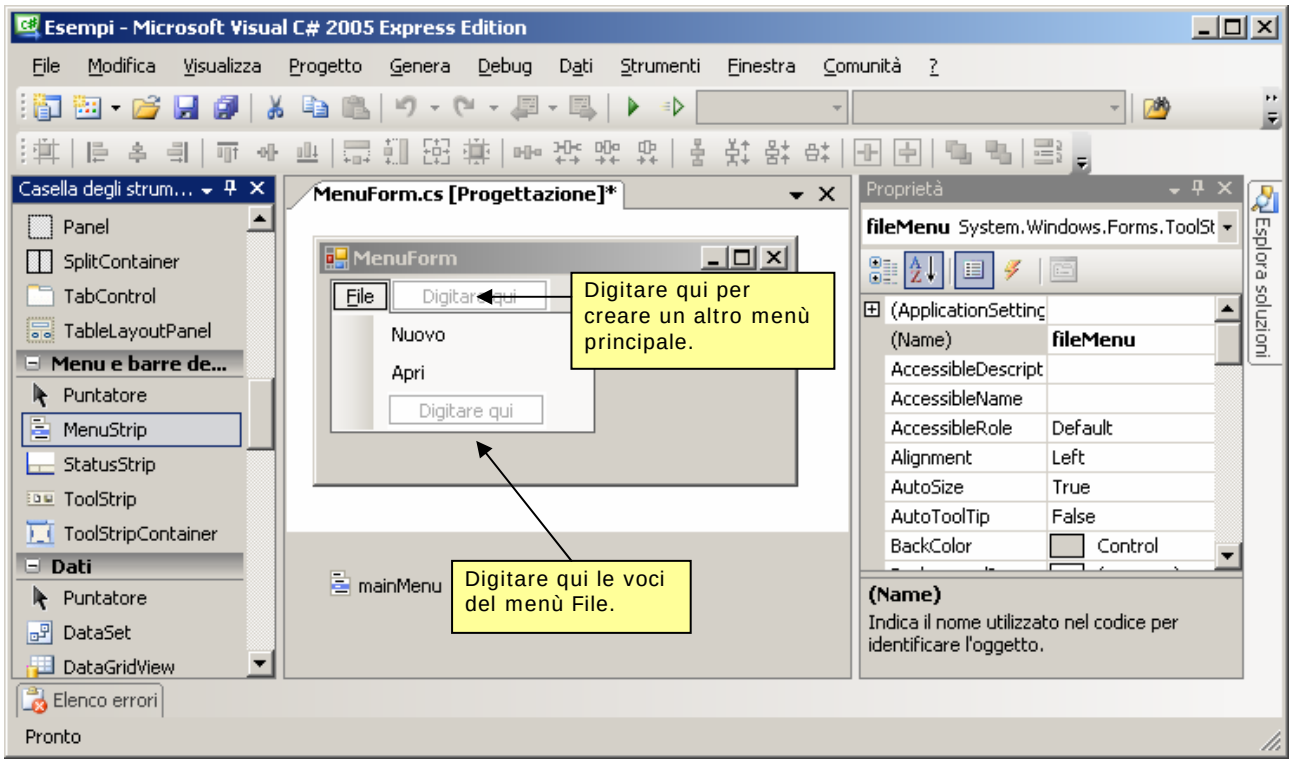


Figura 13-8 Designer di menù.

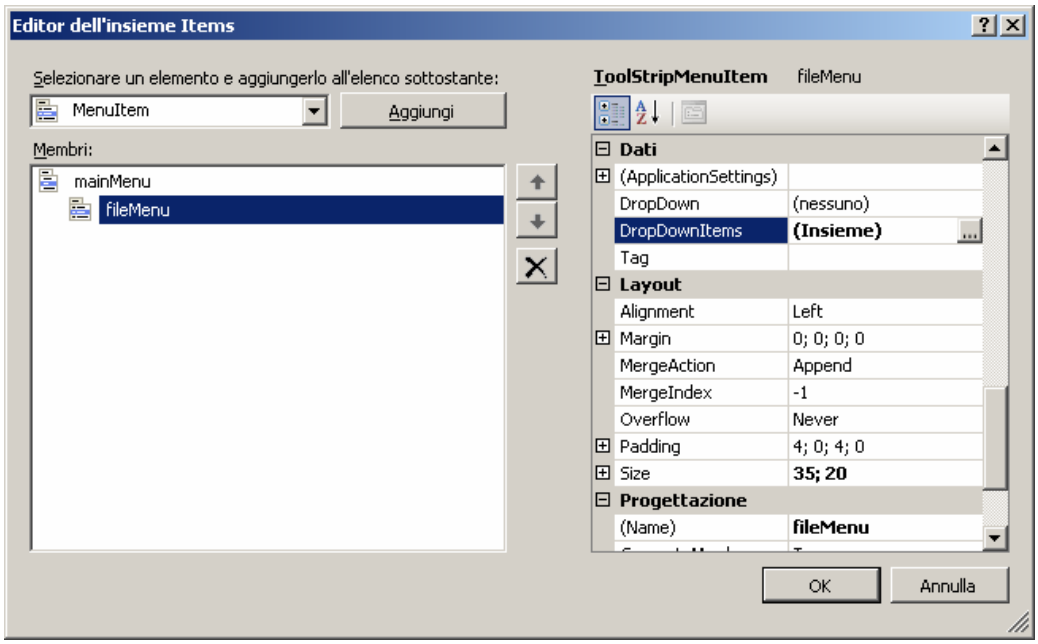


Figura 13-9 Editor di menù.

8.2 Gestire gli eventi di menù

La gestione degli eventi di menù è perfettamente analoga a quella degli altri controlli. Eseguendo doppio click sull'elemento di menù che si intende gestire, oppure selezionando l'evento `Click` dall'editor delle proprietà, viene generato il gestore evento corrispondente.

Ad esempio, ecco il codice generato in risposta al doppio clic sull'elemento `menuApri`:

```
private void menuApri_Click(object sender, EventArgs e)
{
}
```

Ovviamente i menù, come qualsiasi altro controllo, sono in grado di generare un gran numero di eventi e non soltanto l'evento `Click`, anche se è questo il più importante e comunemente gestito.

8.3 Centralizzare la gestione degli elementi appartenenti allo stesso menù

Anche nel caso dei menù è possibile associare un evento ad un metodo già realizzato in precedenza. Sostanzialmente, ciò può servire a due scopi:

1) gestire mediante lo stesso metodo gli eventi generati da un elemento di menù e un controllo.

2) gestire mediante lo stesso metodo gli eventi generati da due o più elementi di menù.

La prima tecnica può essere utile quando una determinata funzionalità dell'applicazione deve essere accessibile sia mediante menù che mediante un controllo. D'altra parte, un approccio migliore è probabilmente quello di scrivere un normale metodo che gestisce la funzionalità e di invocare il metodo nei gestori di evento dell'elemento di menù e del controllo.

Il secondo punto riguarda la scelta di raggruppare all'interno di un unico gestore il codice relativo a tutti gli elementi di un determinato menù.

A tal proposito, consideriamo nuovamente i due elementi del menù "File". Si può decidere di gestirli medianti gestori di evento separati, oppure di associare i loro eventi allo stesso gestore, che fa capo al menù "File":

```
private void menuFile_Click(object sender, EventArgs e)
{
    // qui vengono gestiti gli eventi Click degli elementi menuAPri e menuNuovo
}
```

Ovviamente, per poter elaborare l'evento è necessario prima di tutto stabilire quale elemento di menù lo ha sollevato. A questo scopo esiste il parametro `sender`, il quale referenzia l'oggetto che ha sollevato l'evento. E' dunque possibile scrivere il seguente codice:

```
private void menuFile_Click(object sender, EventArgs e)
{
    if (sender == menuNuovo)
        Nuovo();
    else if (sender == menuApri)
        Apri();
    //... gestione degli altri eventuali elementi del menu "File"
}
```

Il codice presuppone che ogni funzionalità sia gestita da un metodo, un approccio senz'altro ragionevole.

Il vantaggio di questa modalità di gestione degli eventi consiste nella drastica riduzione del numero di gestori necessari. Infatti, la maggior parte dei software realistici definiscono decine di elementi di menù, i quali, però, sono generalmente raggruppati in pochi menù.

Per contro, si complica il codice dei gestori, i quali sono chiamati a gestire gli eventi di più oggetti elementi di menù.

14

Esempio di una interfaccia completa

1 Definizione del problema

L'obiettivo di questo capitolo è di applicare ad un determinato problema alcune delle conoscenze apprese nei capitoli precedenti. A questo scopo viene mostrata la realizzazione di un programma che presenta la sola parte interfaccia.

1.1 Testo del problema

Si richiede la realizzazione di un programma che consenta la compilazione di un curriculum personale. Il curriculum è composto dai seguenti dati:

1) nome
2) cognome;
3) residenza;
4) sesso;
5) titolo di studio conseguito:
   "Nessuno", "Licenza elementare", "Licenza media", "Diploma superiore", "Laurea";
6) corso frequentato (se il titolo di studio conseguito è "Laurea");
7) esperienze lavorative precedenti.

Il programma dovrà implementare un'interfaccia che consenta all'utente di inserire i dati richiesti e di confermare l'avvenuto inserimento mediante il clic su un bottone. Mediante un secondo bottone l'utente potrà azzerare tutti i dati inseriti, riportando il contenuto e lo stato dei controlli alle loro condizioni iniziali.

[Curricolo.cs**]**

2 Progettazione dell'interfaccia

2.1 Progettazione del layout

Si decide di suddividere l'interfaccia in due aree distinte; la prima riservata ai dati personali: nome e cognome, residenza, sesso; la seconda riservata a tutti gli altri dati. Segue uno schema che mostra il layout generale (disposizione) dell'interfaccia:

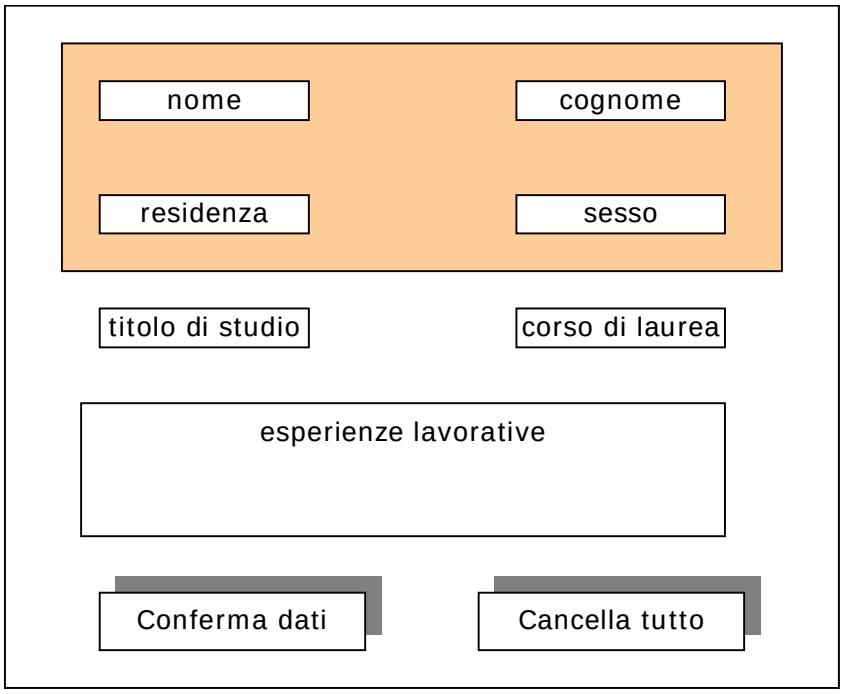


Figura 14-1 Layout generale dell'interfaccia.

Lo schema mostra una visione generale di quella che dovrà essere l'interfaccia, ma non specifica l'esatta natura dei controlli e le eventuali verifiche di consistenza sui dati.

2.2 Scelta dei controlli

In questa fase si decide quali controlli utilizzare per l'acquisizione dei dati. Alcune scelte sono del tutto ovvie, altre sono in parte arbitrarie.

Tabella 14-1 Lista dei controlli utilizzati per l'acquisizione dei dati.

DATO	CONTROLLO	NOME CONTROLLO
nome	TextBox	txtNome
cognome	TextBox	txtCognome
Residenza	TextBox	txtResidenza
Sesso	RadioButton	rbuMaschile rbuFemminile
Titolo di studio	ComboBox	cboTitolo
Corso di laurea	TextBox	txtCorso
Esperienze lavorative	TextBox (multilinea)	txtEsperienze

I due bottoni, il primo per la conferma dei dati, il secondo l'azzeramento dei dati inseriti, sono chiamati rispettivamente: `btnConferma` e `btnCancella`. Vi sono inoltre altri tre controlli:

- `pnlDatiPersonali`: un `Panel` il cui scopo è quello di evidenziare mediante un appropriato colore di sfondo l'area del form dedicata ai dati personali;

- `grpSesso`: `GroupBox` che fa da contenitore ai due `RadioButton` per la selezione del sesso; esattamente come il controllo precedente, anche `grpSesso` riveste un ruolo puramente estetico e non partecipa in alcun modo al dialogo con l'utente;
- `lblCorso`: `Label` il cui testo qualifica il controllo `txtCorso`. I possibili valori della proprietà `Text` dell'etichetta possono essere: "`Laureato in`", "`Laureata in`" o stringa vuota, in base al tipo di selezione fatta dall'utente in relazione al sesso.

3 Layout dell'interfaccia

La figura seguente mostra l'apparenza dell'interfaccia dopo che sono stati inseriti tutti i controlli:

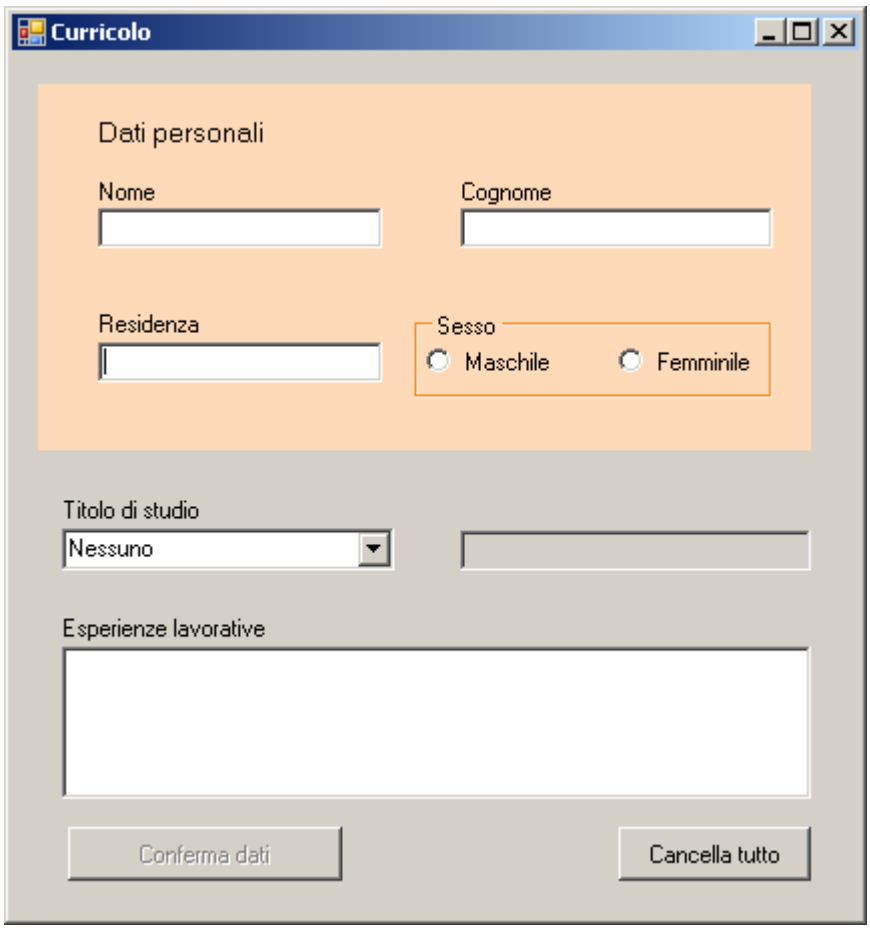


Figura 14-2 Apparenza dell'interfaccia.

4 Inizializzazione dell'interfaccia

Per garantire che l'interfaccia abbia sempre un aspetto consistente è opportuno stabilire, per ogni controllo, uno stato iniziale ben definito, sia relativamente al suo aspetto che al suo contenuto. In entrambi i casi si può scegliere di utilizzare l'editor delle proprietà oppure di collocare il codice di inizializzazione in un metodo, che sarà invocato all'avvio del programma (o entrambe le teniche).

L'uso di un metodo è particolarmente indicato soprattutto per quelle proprietà soggette a variare durante l'esecuzione del programma, infatti:

1) collocare il codice di inizializzazione in un solo punto ne facilita la manutenzione e l'aggiornamento;

utilizzare un metodo rende estremamente semplice reinizializzare l'interfaccia in risposta alle azioni dell'utente.

Decidiamo pertanto di implementare il codice di inizializzazione mediante il metodo `InizializzaControlli()`. Segue il codice del metodo:

```
void InizializzaControlli()
{
    txtNome.Text = "";
    txtCognome.Text = "";
    txtResidenza.Text = "";
    cboTitolo.Text = "Nessuno";
    txtCorso.Text = "";
    txtEsperienze.Text = "";
    rbuMaschile.Checked = false;
    rbuFemminile.Checked = false;
}
```

4.1 Gestione dell'evento Load del form

Per garantire che il metodo `InizializzaControlli()` sia eseguito all'avvio dell'applicazione, l'istruzione di invocazione può essere collocata nel costruttore del form, subito dopo la chiamata a `InitializeComponent()`. Ma esiste un'alternativa migliore, che è di quella di collocare la chiamata nel gestore dell'evento `Load` del form. Tale evento viene sollevato dopo che il form è stato caricato in memoria e l'interfaccia completamente costruita, ma non ancora visualizzata.

Poiché `Load` è l'evento predefinito, per gestirlo basta eseguire un doppio clic su una zona qualsiasi del form:

```
private void MainForm _Load(object sender, EventArgs e)
{
    InizializzaControlli();
}
```

4.2 Inizializzare l'elenco dei titoli di studio

Nell'inserimento del titolo di studio, il programma, mediante un `ComboBox`, propone all'utente un elenco di titoli all'interno del quale scegliere. Poiché tale elenco non è soggetto a variare durante l'esecuzione del programma, l'inizializzazione del `ComboBox` può essere effettuata mediante l'editor delle proprietà, inserendo direttamente i valori richiesti nella proprietà `Items`. Ciò nonostante, resta valida l'alternativa di inizializzare l'elenco mediante codice scritto dal programmatore.

A questo proposito è stato definito un vettore di stringhe contenente l'elenco dei titoli:

```
string[] titoliDiStudio =
       {"Nessuno",
        "Licenza elementare",
        "Licenza media",
```

```
        "Diploma superiore",
       "Laurea"};
```

Per popolare il ComboBox con il suddetto elenco è sufficiente aggiungere un'istruzione nel gestore dell'evento Load:

```
private void MainForm _Load(object sender, EventArgs e)
{
    cboTitoli.DataSource = titoliDiStudio;
    InizializzaControlli();
}
```

A questo proposito c'è una considerazione da fare. Mentre l'impostazione del contenuto iniziale di `cboTitolo` viene eseguita, come per tutti gli altri controlli, all'interno di `InizializzaControlli`, il popolamento della lista è stato collocato fuori dal metodo. Si tratta di una scelta coerente, poiché l'elemento selezionato di `cboTitolo` può essere reinizializzato in risposta alle azioni dell'utente, mentre l'elenco dei titoli di studio non varia mai durante l'esecuzione del programma e dunque può essere impostato una volta per tutte.

5 Rendere l'interfaccia consistente

Intervenire sulla consistenza dell'interfaccia rappresenta un'operazione soggetta alle scelte del programmatore, sia in relazione a quali verifiche si decide di implementare, sia in relazione al modo di implementarle. In questo senso, comunque, va la regole generale che un'interfaccia non dovrebbe soltanto garantire la coerenza dei dati in relazione alle elaborazioni effettuate su di essi, ma anche:

- la coerenza dell'aspetto e dello stato dei controlli, quando questo dipende dal contenuto e/o dallo stato di altri controlli;
- che le azioni dell'utente siano dirette in modo appropriato, impedendo a quest'ultimo di agire in modo incoerente in relazione allo stato e/o al contenuto di determinati controlli.

Dunque, un'interfaccia può essere consistente per quanto riguarda la verifica dei dati, garantendo che questi siano comunque coerenti con le future elaborazioni, ma inconsistente per quanto riguarda l'aspetto e il comportamento, consentendo all'utente azioni inappropriate e comunicando con il medesimo in modo incoerente e ambiguo. Un'interfaccia davvero robusta deve curare entrambi gli aspetti.

Nel programma Curricolo si è deciso di considerare tre aspetti:

- garantire l'esistenza dei dati personali (nome, cognome, residenza, sesso);
- consentire l'accesso al controllo `txtCorso` soltanto se il titolo di studio selezionato è "`Laurea`";
- modificare il testo informativo (l'etichetta) associato del controllo `txtCorso` in relazione al fatto che l'utente si dichiari maschio o femmina e che abbiamo specificato come titolo di studio la laurea.

5.1 Garantire l'esistenza dei dati personali

In questo caso, a scopo dimostrativo, è stato deciso di implementare due forme diverse di verifica, una anticipata, per i dati nome, cognome e residenza, l'altra, posticipata, per il sesso.

Verifica anticipata di nome, cognome e residenza

Questa verifica condiziona lo stato di attivazione del bottone `btnConferma` all'effettiva esistenza dei dati in questione. Ciò si ottiene gestendo l'evento `TextChanged` dei controlli che contengono tali dati.

```
void DatiPersonali_TextChanged (object sender, EventArgs e)
{
       btnConferma.Enabled =(txtNome.Text != "" &&
                             txtCognome.Text != "" &&
                             txtResidenza.Text != "");
}
```

Ovviamente, il gestore di evento dev'essere attaccato a tutti e tre i controlli:

Verifica posticipata del sesso

In questo caso la verifica avviene dopo che è stato cliccato su `btnConferma` e quindi all'interno del gestore dell'evento `Click` di tale bottone:

```
void btnConferma_Click(object sender, EventArgs e)
{
    bool sesso = (rbuMaschile.Checked == true || rbuFemminile.Checked == true);
    if (sesso == false)
    {
        MessageBox.Show("Dati incompleti: specificare il sesso",
                        "Curricolo", MessageBoxButtons.OK,
                        MessageBoxIcon.Error);
        return;
    }

    MessageBox.Show("Dati confermati",  "Programma Curricolo",
                    MessageBoxButtons.OK, MessageBoxIcon.Information);
}
```

La verifica si traduce nel testare se l'uno o l'altro dei due `RadioButton` è marcato. Se nessuno dei due lo è (variabile `sesso == false`) viene visualizzato un messaggio di errore che comunica all'utente di inserire il dato mancante.

5.2 Condizionare l'accesso a txtCorso

Questa forma di verifica fa sì che il `TextBox` relativo al corso di laurea sia accessibile soltanto se l'utente dichiara di essere laureato. Ciò si ottiene gestendo l'evento `SelectedIndexChanged` di `cboTitolo` e impostando lo stato `Enabled` in relazione al contenuto della proprietà `Text` del `ComboBox`:

```
void cboTitolo_SelectedIndexChanged (object sender, EventArgs e)
{
    txtCorso.Enabled = (cboTitolo.Text == "Laurea");
}
```

Da notare che il gestore di evento si limita a modificare lo stato di abilitazione del controllo `txtCorso` e non il suo contenuto. In altre parole, l'utente potrebbe dichiarare di essere laureato, digitare il corso di laurea nel controllo `txtCorso` e successivamente modificare il titolo di studio, selezionando ad esempio "Scuola superiore". In questo caso `txtCorso` verrebbe disabilitato ma manterrebbe il proprio contenuto, precedentemente inserito.

A questo proposito, una ulteriore verifica sulla consistenza dei dati sarebbe appropriata.

5.3 Modificare l'etichetta associata al controllo txtCorso

Questa forma di verifica è stata implementata a scopo dimostrativo, poiché un'etichetta del tipo "Laureato/a in" sarebbe stata sufficiente.

Per far sì che il testo dell'etichetta `lblCorso` sia coerente con il sesso dichiarato dall'utente, occorre gestire l'evento `CheckedChanged` sollevato dai due controlli:

```
void Sesso_CheckedChanged (object sender, EventArgs e)
{
    if (cboTitolo.Text != "Laurea")
    {
        lblCorso.Text = ""
        return;
    }

    if (rbuMaschile.Checked == true)
        lblCorso.Text = "Laureato in:";
    else
    if (rbuFemminile.Checked == true)
        lblCorso.Text = "Laureata in:"
    else
        lblCorso.Text = ""
}
```

Il gestore di evento dev'essere attaccato ad entrambi i `RadioButton`.

Gestire i cambiamenti dei RadioButton non è sufficiente, poiché il testo dell'etichetta deve essere coerente anche con il titolo di studio: occorre dunque aggiornare il gestore dell'evento `SelectedIndexChanged` del ComboBox.

```
void cboTitolo_SelectedIndexChanged (object sender, EventArgs e)
{
    txtCorso.Enabled = (cboTitolo.Text == "Laurea");

    if (cboTitolo.Text != "Laurea")
    {
        lblCorso.Text = ""
        return;
    }

    if (rbuMaschile.Checked == true)
        lblCorso.Text = "Laureato in:";
```

```
    else
    if (rbuFemminile.Checked == true)
        lblCorso.Text = "Laureata in:"
    else
        lblCorso.Text = ""
}
```

Adesso la gestione del'etichetta è completa, ma il codice è stato duplicato in due metodi. In questi casi la soluzione è collocare il codice duplicato in un terzo metodo e invocare quest'ultimo in entrambi i gestori di eventi.

Affronteremo il problema più avanti.

6 Stato iniziale del controlli

Il codice implementato finora garantisce un comportamento consistente dell'interfaccia ma trascura un aspetto molto importante: garantire la consistenza dello stato iniziale dei controlli. Il problema sta nel fatto che l'esecuzione dei gestori di eventi che gestiscono il cambiamento di stato e di contenuto dei controlli non è garantita se non in risposta alle azioni dell'utente. Ciò fa sì che all'atto della esecuzione del programma alcuni controlli possano trovarsi in uno stato inconsistente poiché non sono ancora stati eseguiti i metodi che ne gestiscono il cambiamento di stato.

Ad esempio, il testo dell'etichetta `lblCorso` dovrebbe valere inizialmente stringa vuota, ma non è certo che sia così[12], poiché prima che l'utente compia un'azione non è garantita l'esecuzione del gestore di evento `Sesso_CheckedChanged`, il cui scopo è appunto quello di impostare in modo coerente il testo dell'etichetta. Quanto detto vale anche per altri controlli.

Di questo problema esistono due soluzioni: la prima del tutto ovvia, ma poco soddisfacente; la seconda meno intuitiva, ma che garantisce in qualsiasi momento la consistenza dell'interfaccia.

6.1 Impostare manualmente lo stato iniziale dei controlli

Ciò si traduce nello scrivere, all'interno del costruttore o del gestore di evento `MainForm_Load`, il codice che imposta lo stato iniziale dei controlli dell'interfaccia. Ad esempio

```
void MainForm_Load (object sender, EventArgs e)
{
    cboTitolo.DataSource = titoliDiStudio;
    InizializzaCampi();

    btnConferma.Enabled = false;
    lblCorso.Text = "";
    txtCorso.Enabled = false;
}
```

Il codice funziona, ma presenta dei problemi potenziali. Infatti, eventuali modifiche nelle regole di gestione della consistenza dell'interfaccia non si rifletterebbero nello stato iniziale dei controlli. Ad esempio, attualmente il testo iniziale dell'etichetta `lblCorso` è stringa vuota, e ciò è coerente con lo stato iniziale dei due `RadioButton` `rbuMaschile` e `rbuFemminile`, per entrambi falso. In altre parole, finché l'utente non dichiara il proprio sesso, il testo dell'etichetta non può che essere

[12] Non è certo poiché dipende dall'ordine in cui vengono eseguite le istruzioni che impostano lo stato dei controlli.

indefinito, e cioè vuoto. Si supponga però di decidere che il valore predefinito per il sesso sia “Maschile” e che il valore predefinito per il titolo di studio sia “Laurea”; per coerenza, il valore iniziale di `lblCorso` dovrebbe diventare “Laureato in”. Data la semplicità del programma, modificare il codice di inizializzazione collocato in MainForm_Load non sarebbe un grosso problema, ma si immagini un’interfaccia con una ventina di controlli: qualsiasi modifica nelle regole di gestione di consistenza avrebbe il potere di rendere lo stato iniziale dell’interfaccia inconsistente.

6.2 Scrivere un metodo che aggiorni lo stato dei controlli

Questa soluzione non solo risolve il problema di impostare in modo consistente lo stato iniziale dei controlli, ma semplifica anche il codice presente nei gestori di eventi. L’obiettivo è quello di centralizzare in un solo metodo tutto il codice che svolge il compito di aggiornare lo stato dei controlli. Ogni qual volta è necessario garantire uno stato consistente dell’intera interfaccia è sufficiente invocare il metodo in questione:

```
void AggiornaStatoControlli ()
{
    btnConferma.Enabled =(txtNome.Text != "" &&
                          txtCognome.Text != "" &&
                          txtResidenza.Text != "");

    txtCorso.Enabled = (cboTitolo.Text == "Laurea");

    txtCorso.Enabled = (cboTitolo.Text == "Laurea");
    lblCorso.Text = "";
    if (cboTitolo.Text == "Laurea")
    {
        if (rbuMaschile.Checked == true)
            lblCorso.Text = "Laureato in:";
        else if (rbuFemminile.Checked == true)
                lblCorso.Text = "Laureata in:";
    }
}
```

Il metodo `AggiornaStatoControlli()` contiene tutto il codice di gestione di consistenza dell’interfaccia scritto in precedenza e disperso nei vari gestori di eventi. Il metodo dev’essere invocato all’inizio del programma, dopo che è stato caricato il form, e in qualunque gestore di evento che risponda al cambiamento dello stato di uno dei controlli dell’interfaccia:

```
void DatiPersonali_TextChanged(object sender, EventArgs e)
{
    AggiornaStatoControlli();
}
...
void Sesso_CheckedChanged(object sender, EventArgs e)
{
    AggiornaStatoControlli();
}
void cboTitolo_SelectedIndexChanged(object sender, EventArgs e)
```

```
{
    AggiornaStatoControlli();
}
void MainForm_Load (object sender, EventArgs e)
{
    InizializzaCampi();
    cboTitolo.DataSource = titoliDiStudio;
    AggiornaStatoControlli();
}
```

Vantaggi e svantaggi di questo approccio

Ci sono i seguenti vantaggi:

- l'esecuzione di `AggiornaStatoControlli()` garantisce la consistenza dell'interfaccia; la sua esecuzione all'inizio del programma garantisce dunque anche uno stato iniziale coerente di tutti i controlli;
- il codice di gestione della consistenza viene centralizzato in un unico punto e dunque può essere facilmente modificato o esteso;
- viene eliminato ogni eventuale codice duplicato;
- nei gestori di eventi, il codice dei gestori che risponde ai cambiamenti di stato si riduce alla semplice invocazione di un metodo.

Questo approccio comporta comunque un problema, di norma trascurabile, ma in alcuni casi di una certa rilevanza:

- il cambiamento in uno qualsiasi dei controlli determina l'esecuzione dell'intero codice che gestisce la consistenza dell'interfaccia, e dunque anche di quel codice che non ha nessuna relazione con il controllo che ha subito il cambiamento.

Esiste dunque un problema di inefficienza, trascurabile nella maggior parte dei casi, ma rilevante se il codice di gestione della consistenza è piuttosto corposo oppure deve compiere elaborazioni sofisticate e dunque potenzialmente rilevanti in termini di costo computazionale.

15

Finestre di dialogo

1 Finestre di dialogo (o *dialog*): Form **modali**

Una finestra di dialogo[13] o *dialog*, è un form che consente al programma di interagire con l'utente, ad esempio per:

- richiedere dei dati;
- eseguire un comando che richiede determinate informazioni prima di essere lanciato;
- impostare delle opzioni di funzionamento del programma;
- visualizzare un messaggio informativo (o di conferma o di errore) personalizzato, eccetera.

Esempi tipici di finestre di dialogo sono quelle aperte in risposta ai comandi «File | Apri…», «Formato | Carattere …». Ognuna di esse funziona secondo uno schema prestabilito:

2) la *dialog* viene aperta e diventa automaticamente l'unica finestra dell'applicazione in grado di rispondere alle azioni dell'utente;
3) l'utente interagisce con la finestra, di norma inserendo e/o modificando dei dati, fintantoché decide di chiuderla;
4) la chiusura della *dialog* può avvenire di norma in base a due modalità:
   - di «conferma», generalmente mediante il bottone «Ok»;
   - di «annullamento», generalmente mediante il bottone «Annulla», oppure cliccando sul pulsante di chiusura della finestra;
5) il controllo dell'esecuzione ritorna al form che era attivo al momento dell'apertura della finestra di dialogo.

[13] La definizione completa sarebbe «Finestra di dialogo modale». Abbiamo deciso di tralasciare l'aggettivo «modale» poiché si intende sottinteso.

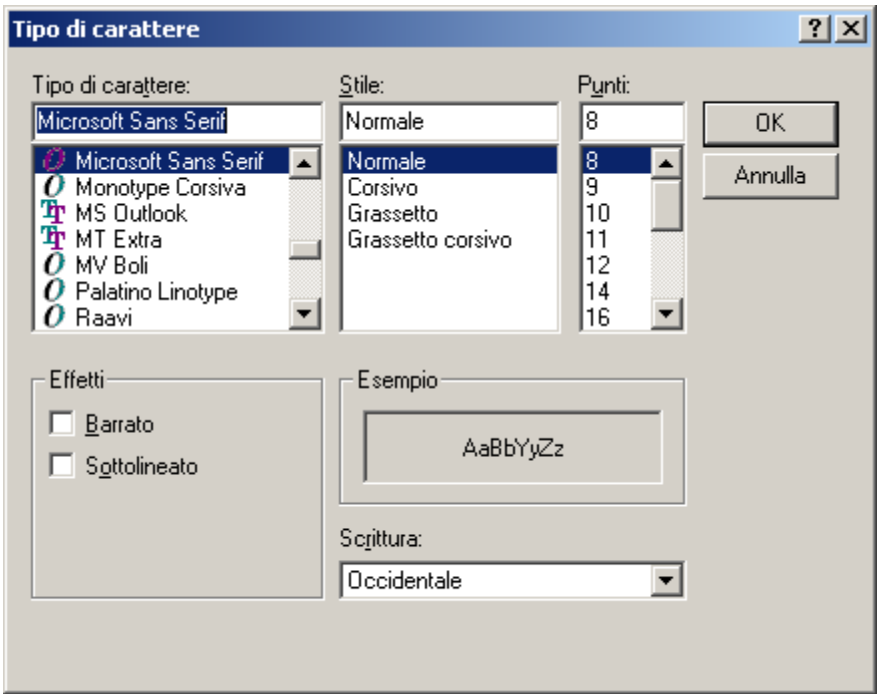


Figura 15-1 Finestra di dialogo standard (Tipo Carattere)

La modalità di chiusura della finestra di dialogo è determinante, poiché:

- in caso di conferma, i dati e le modifiche effettuate dall'utente si intendono confermate. Il codice che ha eseguito la finestra provvederà a elaborare i dati;
- in caso di annullamento, qualsiasi inserimento o modifica prodotti dall'utente vengono semplicemente scartati.

1.1 Form modali

Una finestra di dialogo è realizzata mediante un form «modale». L'aggettivo «modale» indica che il form, una volta visualizzato, assume il controllo delle azioni dell'utente fintantoché non viene chiuso. Mentre è aperto un form modale, gli altri form dell'applicazione non sono in grado di rispondere alle azioni dell'utente. Di contro, un form «non modale» può rilasciare il controllo a un altro form pur restando aperto.

Non c'è nulla che differenzi un form modale da uno che non lo è (entrambi gli oggetti appartengono alla classe `Form`) se non nelle modalità di visualizzazione e di chiusura.

1.2 Visualizzare un Form come modale

Il fatto che un sia modale o meno dipende dal metodo utilizzato per visualizzarlo. Esistono infatti i metodi `Show()` e `ShowDialog()`: il primo visualizza il form come non modale, il secondo lo visualizza come modale.

Come primo esempio consideriamo il seguente programma, il quale definisce due form: `Form1` e `FormModale`. Il `Form1` contiene semplicemente un bottone; cliccando su di esso viene visualizzato il form `FormModale`.

```
void btnDialog_Click(object sender, EventArgs e)
{
  FormModale dialogo;
  dialogo = new FormModale();       // il form viene creato
```

```
  dialogo.ShowDialog();             // il form viene visualizzato
    // il codice che segue la precedente istruzione sarà eseguito
    // soltanto dopo che il form modale è stato chiuso!
}
```

Le tre istruzioni evidenziate sono quelle che ci interessano. La prima dichiara una variabile di tipo `FormModale`. La seconda determina la creazione del form. L'esecuzione di questa istruzione non produce affatto la sua visualizzazione: il form esiste in memoria ma non è ancora visibile. Infine, l'invocazione del metodo `ShowDialog()` rende il form visibile e gli passa il controllo del programma.

L'esecuzione di «`dialogo.ShowDialog();`» fa sì che tutte le azioni dell'utente siano ricevute dal secondo form. In `Form1` l'esecuzione resta sospesa sull'istruzione suddetta, fintantoché il form modale non viene chiuso. Quando ciò avviene, l'esecuzione riprende dall'istruzione successiva della chiamata a `ShowDialog()`.

1.3 Chiudere un Form modale

In generale, il processo di chiusura di un form sottintende due questioni: da una parte l'azione o le azioni dell'utente che determinano la chiusura, dall'altra il codice che produce tale processo e gli eventi che esso solleva. Per quanto riguarda un form modale esistono ulteriori due aspetti:

- la modalità di chiusura del form, conferma o annullamento;
- la comunicazione della modalità di chiusura al codice chiamante.

Azioni dell'utente che determinano la chiusura del Form

Oltre agli altri controlli, un form modale contiene di norma due `Button` il cui scopo è quello di consentire all'utente di chiuderlo, confermando o annullando i dati inseriti e le modifiche effettuate. I loro nomi dipendono ovviamente dalle scelte del programmatore e dalla natura del form, ma nella maggior parte dei casi sono per convenzione «OK» e «Annulla».

Un secondo meccanismo è il pulsante di chiusura: per convenzione, cliccare su di esso equivale a cliccare sul bottone «Annulla», e cioè chiudere il form con la modalità annullamento.

E' possibile chiudere il form anche attraverso la tastiera. La combinazione predefinita `ALT+F4` funziona con qualsiasi form e lo chiude con modalità annullamento. Esistono inoltre meccanismi implementabili dal programmatore (e convenzionalmente adottati da tutti i programmi). Uno di questi consente di associare ai due bottoni di chiusura i tasti `INVIO` ed `ESC`, il che permette all'utente di chiudere il form in modalità conferma premendo `INVIO` e in modalità annullamento premendo `ESC`.

Chiusura del Form mediante i bottoni «OK» e «Annulla»

Alcuni dei meccanismi di chiusura precedentemente menzionati (combinazione `ALT+F4` e clic sul pulsante di chiusura) sono predefiniti e dunque non richiedono alcun intervento da parte del programmatore; gli altri devono essere implementati. Per il momento tratteremo soltanto i bottoni «OK» e «Annulla».

E' possibile chiudere un form modale assegnando un valore alla proprietà `DialogResult` dello stesso. Tra quelli ammessi, i soli valori che ci interessano sono `DialogResult.OK` e `DialogResult.Cancel`, che determinano rispettivamente la chiusura con conferma e quella con annullamento. Dunque, per chiudere il form in risposta al clic sul bottone «OK» è sufficiente gestire l'evento corrispondente. Discorso analogo vale per il bottone «Annulla»:

```
void btnOk_Click(object sender, EventArgs e)
{
    DialogResult = DialogResult.OK;       // chiusura con conferma
}
void btnAnnulla_Click(object sender, EventArgs e)
{
    DialogResult = DialogResult.Cancel;   // chiusura con annullamento
}
```

Alternativamente, e senza gestire alcun evento, è possibile ottenere lo stesso risultato associando ai due bottoni il valore da assegnare alla proprietà `DialogResult` nel momento in cui sono cliccati. Questo viene fatto assegnando il valore opportuno alla proprietà `DialogResult` esposta dai due bottoni (da non confondere con la proprietà `DialogResult` esposta dal form). Ciò è possibile mediante le seguenti istruzioni, da collocare nel costruttore del form:

```
btnOk.DialogResult = DialogResult.Ok;
btnAnnulla.DialogResult = DialogResult.Cancel;
```

Nel resto del capitolo prenderemo in considerazione soltanto la modalità tramite gestione eventi.

Comunicazione al codice chiamante della modalità di chiusura

La modalità di chiusura del form viene comunicata al codice chiamante attraverso il valore di ritorno del metodo `ShowDialog()`, che equivale al valore assegnato alla proprietà `DialogResult` del form. Nel primo esempio del capitolo questo valore viene ignorato, ma ciò non è di norma corretto in una applicazione reale. Ecco una nuova versione del gestore di evento che visualizza il form modale:

```
void btnDialog_Click(object sender, EventArgs e)
{
    FormModale dialogo = new FormModale();
    DialogResult cmd = dialogo.ShowDialog();
    if (cmd == DialogResult.OK)
    {
        // ... acquisisce ed elabora i dati inseriti nel form
    }
    ...
}
```

Il codice è molto simile a quello che verifica il valore di ritorno del metodo `Show()` della classe `MessageBox`: solo se l'utente ha confermato i dati inseriti, questi vengono elaborati, altrimenti vengono semplicemente scartati.

Eventi sollevati nel processo di chiusura del Form

Il processo di chiusura di un form (modale o non modale) solleva due eventi. `Closing` e `Closed`, il primo durante il processo, il secondo dopo che il form è stato praticamente chiuso.

2 Scambiare dati tra codice chiamante e finestra di dialogo

La verifica della modalità di chiusura di una finestra di dialogo fa parte di un problema più generale che riguarda la comunicazione dei dati tra il form modale e il codice che lo esegue. Ad esempio, riconsideriamo la finestra di dialogo «Tipo di carattere ». Dopo che l'utente ha chiuso cliccando sul bottone «Ok» il controllo ritorna al codice che aveva visualizzato la finestra; ebbene, come fa questo codice ad accedere alle informazioni sul font impostate dall'utente?

Il problema dello scambio dei dati tra form modale e codice chiamante può essere affrontato con vari approcci, che dipendono dalla natura del form e dei dati da scambiare, dall'architettura dell'applicazione e da scelte arbitrarie. Qui prenderemo in considerazione l'approccio più semplice.

2.1 Acquisire i dati attraverso campi pubblici definiti dal Form modale

Consideriamo la seguente applicazione dimostrativa. Il form principale contiene una etichetta della quale si vuole rendere possibile modificare il font. Esiste un bottone, «Preferenze», attraverso il quale accedere a un form modale che mediante due `TextBox` richiede il nome e la dimensione del font da applicare all'etichetta.

Dunque, devono essere elaborate due informazioni, una stringa e un `float` (`float` è il tipo utilizzato per la dimensione nel costruttore di un oggetto `Font`). Per rendere questi dati accessibili al codice chiamante, utilizziamo due campi pubblici definiti nel form modale.

```
public partial class Form1: Form
{
    // . . .

    void btnPreferenze_Click(object sender, EventArgs e)
    {
        FormModale preferenze = new FormModale();
        DialogResult cmd = preferenze.ShowDialog();
        if (cmd == DialogResult.OK)
        {
            lblTest.Font = new Font(preferenze.NomeFont, preferenze.DimFont);
        }
    }
}
// definizione del form che fungerà da form modale

public partial class FormModale: Form
{
    //. . .
    // dati da rendere accessibili al codice chiamante
    public string NomeFont;
    public float DimFont;


    void btnOk_Click(object sender, EventArgs e)
    {
        // memorizza i dati digitati dall'utente
        NomeFont = txtNomeFont.Text;
        DimFont = float.Parse(txtDimFont.Text);
```

```
        DialogResult = DialogResult.OK;         // chiusura con conferma
    }
    void btnAnnulla_Click(object sender, EventArgs e)
    {
        DialogResult = DialogResult.Cancel;     // chiusura con annullamento
    }
}
```

I tre frammenti di codice evidenziati sono quelli che ci interessano. Nel primo:

```
lblTest.Font = new Font(preferenze.NomeFont, preferenze.DimFont);
```

viene assegnato il nuovo font all'etichetta sulla base dei dati inseriti dall'utente, memorizzati nei campi `NomeFont` e `DimFont` dell'oggetto form `preferenze`. Tali campi sono dichiarati nel secondo frammento evidenziato:

```
public string NomeFont;
public float DimFont;
```

Infine, il terzo frammento di codice:

```
NomeFont = txtNomeFont.Text;
DimFont = float.Parse(txtDimFont.Text);
```

presente nel gestore di evento associato al bottone «OK» del form modale, memorizza nei due campi pubblici il contenuto dei due `TextBox` utilizzati per acquisire i dati dall'utente.

In figura è mostrato il programma in esecuzione. Il form principale è mostrato due volte, durante e dopo l'esecuzione del form modale «Preferenze».

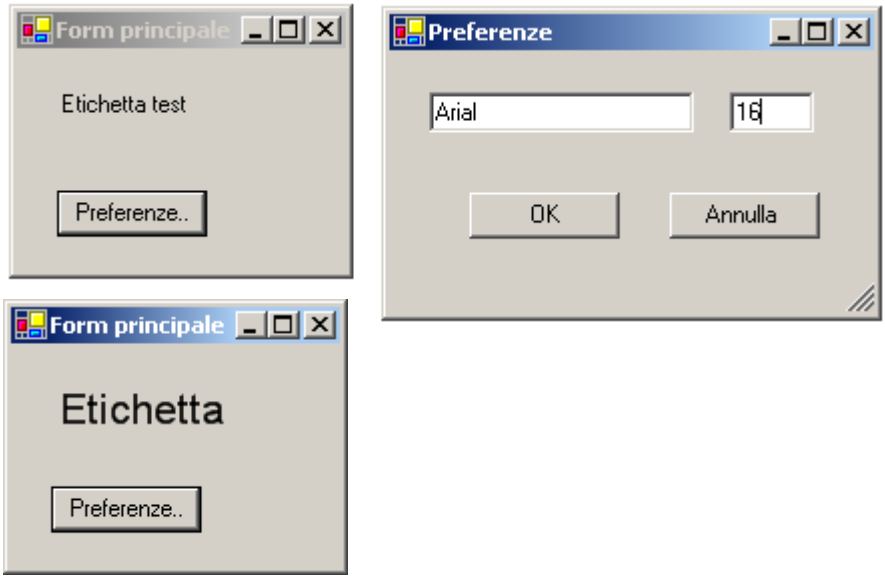


Figura 15-2 Output del programma

Considerazioni sul modello di scambio dei dati

L'approccio utilizzato si basa sui seguenti presupposti:

- il codice chiamante non “conosce” niente del form modale, se non i due campi pubblici che rappresentano i dati da acquisire;
- il form modale non ha alcun bisogno di accedere ai campi o ai controlli del form principale (o in generale del form che contiene il codice chiamante).

Un simile approccio presenta il vantaggio di rendere il codice chiamante indipendente dalla struttura della finestra di dialogo e dunque anche dalla modalità utilizzata per acquisire i dati dall’utente. Per comprendere il concetto ipotizziamo di modificare il form modale in modo che l’inserimento della dimensione del font avvenga attraverso un `ListBox` che permette la selezione da un elenco predefinito di valori.

Nella classe `FormModale`, dopo aver sostituito il controllo `txtDimFont` con il controllo `lboDimFont` (sia in fase di dichiarazione che di costruzione), è sufficiente effettuare due semplici modifiche. Aggiungere la seguente istruzione al costruttore:

```
public FormModale()
{
    // ... istruzioni di costruzione del form

    // popola il list box
    float[] dimFont = {8, 9, 10, 12, 14, 16};
    lboDimFont.DataSource = dimFont;
}
```

Sostituire la variabile `lboDimFont` a `txtDimFont` nel gestore di evento associato al bottone «OK»:

```
void btnOk_Click(object sender, EventArgs e)
{
    // memorizza i dati digitati dall’utente
    NomeFont = txtNomeFont.Text;
    DimFont = (float) lboDimFont.SelectedItem;

    DialogResult = DialogResult.OK;       // chiusura con conferma
}
```

La cosa notevole è che non occorre modificare in alcun modo il codice chiamante, poiché esso fa riferimento unicamente ai due campi pubblici e dunque non dipende da eventuali modifiche all’interfaccia grafica della finestra di dialogo.

2.2 Fornire dei valori iniziali ai campi pubblici del Form modale

Il modello di scambio fin qui realizzato funziona in una sola direzione. Esso prevede che il codice chiamante, dopo aver eseguito la finestra di dialogo, acceda ai campi pubblici contenenti i dati, ma non prevede che i due campi suddetti abbiano un valore iniziale diverso da quello predefinito. D’altra parte, una finestra di dialogo dovrebbe sempre fornire dei valori iniziali per i dati, in modo che l’utente non sia obbligato a inserire tutti i dati richiesti, ma possa semplicemente modificarne alcuni e confermare gli altri. Tali valori possono essere:

- predefiniti e sempre gli stessi;
- i valori inseriti l’ultima volta che l’utente ha aperto e confermato la finestra di dialogo;
- i valori attualmente utilizzati dall’applicazione (che equivalgono a quelli del punto precedente se la finestra di dialogo rappresenta l’unico mezzo per modificarli).

La scelta migliore è senz'altro l'ultima ed è universalmente adottata da tutte le applicazioni. Esistono vari modi per metterla in atto, noi ne adotteremo uno che sia coerente con la scelta di rendere le variabili del form principale inaccessibili al form modale.

Detto ciò, per fornire dei valori iniziali ai campi della finestra di dialogo occorre che:

- il codice chiamante, dopo aver creato il form modale, ma prima di visualizzarlo, imposti i valori iniziali dei suoi campi pubblici;
- il form modale, prima di diventare visibile, inizializzi i controlli della propria interfaccia utilizzando i valori dei campi pubblici.

La prima fase non richiede spiegazione, e nel nostro caso consiste in due semplici assegnazioni prima dell'invocazione del metodo `ShowDialog()`:

```
public partial class Form1: Form
{
    // ...
    void btnPreferenze_Click(object sender, EventArgs e)
    {
        FormModale preferenze = new FormModale();
        preferenze.NomeFont = lblTest.Font.Name;
        preferenze.DimFont = lblTest.Font.Size;
        DialogResult cmd = preferenze.ShowDialog();
        if (cmd == DialogResult.OK)
        {
            lblTest.Font = new Font(preferenze.NomeFont, preferenze.DimFont);
        }
    }
}
```

La seconda fase richiede la gestione dell'evento `Load` del form modale. Tale evento viene sollevato prima che il form venga visualizzato per la prima volta (o venga visualizzato nuovamente dopo che era stato chiuso).

```
public partial class FormModale: Form
{
    // ...
    public FormModale()
    {
        // codice necessario alla costruzione del form
        Load += new EventHandler(FormModale_Load);
    }
    ...
    void FormModale_Load(object sender, EventArgs e)
    {
        txtNomeFont.Text = NomeFont;
        txtDimFont.Text = DimFont.ToString();
    }
}
```

Occorre precisare che l'impiego di campi pubblici per la condivisione di dati tra due oggetti non è l'approccio più corretto; al loro posto sarebbe opportuno usare delle proprietà, che qui abbiamo evitato per alleggerire il codice sorgente. Ciò detto, anche sostituendo le proprietà ai campi pubblici, il modello di scambio dei dati presentato resta perfettamente valido.

3 Creare e distruggere una finestra di dialogo

Nel codice chiamante scritto finora:

```
void btnPreferenze_Click(object sender, EventArgs e)
{
    FormModale preferenze = new FormModale();
    ...
    DialogResult cmd = preferenze.ShowDialog();
    ...
}
```

la finestra di dialogo viene ricreata ogni volta e referenziata attraverso una variabile locale. Premettendo subito che questo codice è incompleto, ci si può domandare: è per forza necessario ricreare ogni volta la finestra? La risposta è no: si può utilizzare una variabile globale, creare la finestra una sola volta nel costruttore del form principale e quindi visualizzarla quando richiesto:

```
FormModale preferenze;

public Form1()
{
    // ... codice necessario per la costruzione del form
    preferenze = new FormModale();  // il form viene creato una sola volta

}


void btnPreferenze_Click(object sender, EventArgs e)
{
    ...
    DialogResult cmd = preferenze.ShowDialog();
    ...
}
```

Questo approccio è reso possibile dal fatto che la chiusura di un form modale, diversamente da quanto accade per un form non modale, non ne determina la distruzione, e cioè il rilascio delle risorse grafiche impiegate. Il form, in realtà, viene semplicemente nascosto.

Ciò detto, tale approccio non è l'ideale; esso implica infatti il mantenimento in memoria di un controllo che utilizza notevoli risorse grafiche del sistema, nonostante esso venga usato occasionalmente (o addirittura mai) dall'utente. In un programma realistico, che quasi certamente richiede più finestre di dialogo, sarebbe uno spreco di risorse inaccettabile. (Per inciso, Microsoft Word supera agilmente le 50 finestre di dialogo).

3.1 Distruggere (rilasciare le risorse grafiche di) un Form modale

L'approccio di ricreare ogni volta il form modale è dunque quello corretto, ma così come è stato realizzato è incompleto. Esso non prevede infatti la distruzione esplicita del form modale, processo che consente al sistema operativo di recuperare le risorse grafiche impegnate. Questo risultato si ottiene invocando il metodo `Dispose()` del form:

```
void btnPreferenze_Click(object sender, EventArgs e)
{
    FormModale preferenze = new FormModale();
    ...
    DialogResult cmd = preferenze.ShowDialog();
    ...
    preferenze.Dispose();     // rilascia le risorse grafiche del form
}
```

> E' importante comprendere che l'invocazione del metodo `Dispose()` riguarda l'aspetto dell'efficienza del programma e non influisce sul suo corretto funzionamento. Se non viene invocato `Dispose()`, dopo che è terminato il gestore di evento, il form continua a impegnare le risorse grafiche del sistema finché il *Garbage Collector* non decide di distruggerlo, recuperare la memoria e, conseguentemente, rilasciare al sistema operativo le risorse grafiche.

4 Verificare la validità dei dati prima di chiudere il Form modale

Un'applicazione ben progettata prevede sempre una verifica sulla validità dei dati prima che essi vengano elaborati. Ciò riguarda ovviamente anche i dati inseriti in una finestra di dialogo. In questo senso nasce la questione: quando effettuare la verifica dei dati, prima o dopo aver chiuso il form modale?

Laddove possibile (e non sempre lo è), la scelta migliore è quella di verificare la validità dei dati all'interno del form modale ed eventualmente di abortire il processo di chiusura con conferma se questi non risultano validi. Ciò consente all'utente di correggere i dati non validi senza doverli inserire tutti di nuovo..

Esistono fondamentalmente due modi per farlo.

4.1 Evento «Closing»: abortire il processo di chiusura del Form

Innanzitutto, non ci occuperemo qui della modalità di verifica, ma soltanto di come arrestare il processo di chiusura del form in modo che l'utente non possa confermare dei dati non validi.

Per fare questo si può gestire l'evento `Closing` che, come suggerisce anche il nome, è sollevato durante il processo di chiusura. Il parametro informazioni evento associato a `Closing` è di tipo `CancelEventArgs`; esso espone la proprietà `Cancel`, mediante la quale è possibile decidere se interrompere il processo di chiusura (`Cancel = true`), oppure lasciare che si completi (`Cancel = false`). Il valore predefinito della proprietà è `false`.

Ecco un possibile modo per gestire l'evento:

```
public partial class FormModale
{
    ...
    void FormModale_Closing(object sender, CancelEventArgs e)
```

```
    {
        if (DialogResult == DialogResult.OK && SeOkDati() == false)
            e.Cancel = true;
    }
}
```

Il codice precedente presuppone che esista un metodo `SeOkDati()` che ritorna `true` se i dati del form sono validi, `false` in caso contrario. Da notare che se il form non è stato chiuso con modalità conferma non viene prodotta (né testata) alcuna forma di verifica.

Ovviamente, il gestore deve essere attaccato all'evento corrispondente. Il tipo di delega è `CancelEventHandler`[14]:

```
public FormModale()
{
    // ... codice necessario per la costruzione del form
    Closing += new CancelEventHandler(FormModale_Closing);
}
```

Questo approccio presenta comunque qualche problema, poiché l'evento `Closing` viene sollevato dopo che è stato eseguito il gestore d'evento del bottone «OK» e dunque dopo che i valori dei controlli sono stati assegnati ai campi pubblici che memorizzano i dati da ritornare al codice chiamante. In altre parole, per alcuni tipi di verifica (ad esempio la verifica del formato numerico di un valore) potrebbe essere già troppo tardi.

Esistono vari modi per superare questo problema. Uno, che prenderemo in esame, non presuppone la gestione dell'evento `Closing`.

4.2 Verifica dei dati nel gestore di evento del bottone «OK»

Il modo più semplice per abortire il processo di chiusura è non cominciarlo nemmeno. Per fare questo è sufficiente eseguire la verifica all'interno del gestore d'evento del bottone «OK» e confermare la chiusura soltanto se l'esito è positivo:

```
public partial class FormModale: Form
{
    //...
    void btnOk_Click(object sender, EventArgs e)
    {
        if (SeOkDati(ref NomeFont, ref DimFont) == true)
                DialogResult = DialogResult.OK;        // chiusura con conferma
    }
}
```

Nell'esempio si presuppone che il metodo `SeOkDati()` svolga il duplice ruolo di verificare la validità dei valori digitati dall'utente e, in caso positivo, di memorizzarli nei campi pubblici corrispondenti. Il valore di verifica ritornato viene testato per decidere se chiudere il form o meno.

> Le tecniche presentate sono molto semplici e non richiedono una profonda conoscenza del *framework*. In questo senso, .NET fornisce eventi e meccanismi specializzati nella validazione dati. Esiste inoltre il controllo `ErrorProvider` che consente di gestire in modo semplice ma sofisticato la segnalazione all'utente dei controlli che contengono dati non validi.

[14] I tipi `CancelEventArgs` e `CancelEventHandler` sono definiti nel *namespace* `System.ComponentModel`.

5 Standardizzare aspetto e comportamento della finestra di dialogo

A meno che non si debba rispondere ad esigenze particolari, l'interfaccia della nostra applicazione dovrebbe sempre seguire un modello coerente, sia nell'aspetto che nella modalità di interazione con l'utente; modello che in genere dovrebbe rispecchiare quello adottato dal sistema operativo. Ciò dovrebbe valere anche per le finestre di dialogo. Esse dovrebbero dunque rispettare i seguenti requisiti di base:

- essere visualizzate al centro dello schermo;
- non essere ridimensionabili, (a meno che ciò non rappresenti una caratteristica esplicita della finestra, com'è ad esempio per finestra di dialogo comune `OpenFileDialog`, mostrata più avanti nel testo);
- non avere i pulsanti «Ingrandisci» e «riduci a icona», e dunque avere il solo pulsante di chiusura (ed eventualmente il pulsante di aiuto, se questo è fornito);
- avere i tasti INVIO ed ESC associati rispettivamente ai bottoni «OK» e «Annulla», in modo che l'utente possa usare la tastiera per confermare o annullare la *dialog*.

La classe `Form` espone proprietà apposite per fornire alle finestra di dialogo queste caratteristiche. Se si dispone di Visual Studio.NET sarà sufficiente impostarle mediante la «Finestra delle proprietà», altrimenti sarà necessario scrivere il codice appropriato nel costruttore del form modale, cosa che faremo modificando la classe `FormModale` precedentemente realizzata.

```
public partial class FormModale: Form
{
    //... dichiarazione dei controlli e dei campi pubblici
    public FormModale()
    {
        // ... codice necessario per la costruzione del form

        // posiziona il form al centro dello schermo
        StartPosition = FormStartPosition.CenterScreen;

        // elimina i pulsanti «Ingrandisci» e «Riduci a icona»
        MinimizeBox = false;
        MaximizeBox = false;

        // fornisce al form un bordo non ridimensionabile
        FormBorderStyle = FormBorderStyle.FixedDialog;

        // associa i tasti INVIO e ESC ai bottoni «OK» e Annulla
        AcceptButton = btnOk;
        CancelButton = btnAnnulla;
    }
}
```

6 Finestre di dialogo comuni: (*Common dialog*)

La maggior parte delle applicazioni deve svolgere delle tipiche attività di interazione con l'utente nelle quali viene richiesto ad esempio:

- il nome di un file (per aprirlo o per salvarlo);
- un colore, un font,
- di impostare delle opzioni di stampa, eccetera.

Poiché sono attività comuni a molti tipi di applicazioni, .NET fornisce delle finestre di dialogo preconfezionate che svolgono in pratica tutto il lavoro e che entro certi limiti possono essere personalizzate per la specifica applicazione. Sono le cosiddette «Finestre di dialogo comuni, tra le quali: `OpenFileDialog`, `SaveFileDialog`, `FontDialog`, `ColorDialog`, `PrintDialog`, eccetera.

A scopo dimostrativo sarà mostrato un esempio d'uso per due di esse.

6.1 Richiedere all'utente il nome di un file: «OpenFileDialog»

Chiedere all'utente il percorso e il nome di un file è probabilmente una delle attività più comuni di una applicazione. La classe `OpenFileDialog` implementa una finestra di dialogo d'uso generale che in molti casi è sufficiente allo scopo.

Segue un elenco delle proprietà di base, che consentono di scambiare i dati con la *dialog* nonché di personalizzarne il comportamento.

Tabella 15-1 proprietà di base della classe `OpenFileDialog`

PROPRIETÀ - TIPO	DESCRIZIONE
`CheckFileExist` `bool`	Se `true`, fa sì che la *dialog* verifichi l'effettiva esistenza del file inserito dall'utente. Il valore predefinito è true.
`DefaultExt` `string`	Consente di impostare l'estensione predefinita, e cioè l'estensione aggiunta al nome del file nel caso in cui l'utente non la specifichi eplicitamente. (L'estensione si intende priva del punto: ad esempio "DOC", "CS", "EXE", eccetera. Il valore predefinito è stringa vuota.
`FileName` `string`	Contiene il nome del file inserito dall'utente.
`InitialDirectory` `string`	Consente di impostare la cartella iniziale visualizzata nella finestra di dialogo. Il valore predefinito è stringa vuota.
`Filter` `string`	Consente di impostare le opzioni di filtro da utilizzare nella casella «Tipo file». Ciò permette di visualizzare soltanto quei file il cui nome corrisponde a un certo modello (pattern); ad esempio file con estensione "DOC", piuttosto che "BMP", o "JPG", eccetera. La stringa può essere composta da più sezioni, ognuna delle quali può contenere uno o più filtri. Ogni filtro è composto da una parte descrittiva, che è quella che compare nella casella «Tipo file» e dal filtro vero e proprio. Ad esempio, la seguente stringa contiene due filtri, uno per i file con estensione "TXT", l'altro per file qualsiasi: `"File di testo (*.txt)|*.txt|Tutti i file (*.*)|*.*"`

L'applicazione che segue dimostra l'impiego della finestra di dialogo. In risposta al clic sul bottone «ApriFile» viene aperta una `OpenFileDialog`. Successivamente, mediante una *message dialog* sarà visualizzata la risposta dell'utente. Prima di visualizzare la *dialog* viene impostato un filtro che consente all'utente di scegliere se visualizzare tutti i file o soltanto quelli con estensione "DOC".

```
public partial class Form1: Form
{
    void btnApriFile_Click(object sender, EventArgs e)
    {
        OpenFileDialog dgApri = new OpenFileDialog();
        dgApri.Filter = "Documento (*.doc)|*.doc|Tutti i file (*.*)|*.*";
        dgApri.CheckFileExists = false;
        DialogResult cmd = dgApri.ShowDialog();
        if (cmd != DialogResult.OK)
        {
            MessageBox.Show("L'utente ha annullato la dialog");
        }
        else
        {
            MessageBox.Show("E' stato selezionato il nome: " + dgApri.FileName);
        }
    }
}
```

Ed ecco come appare la finestra di dialogo:

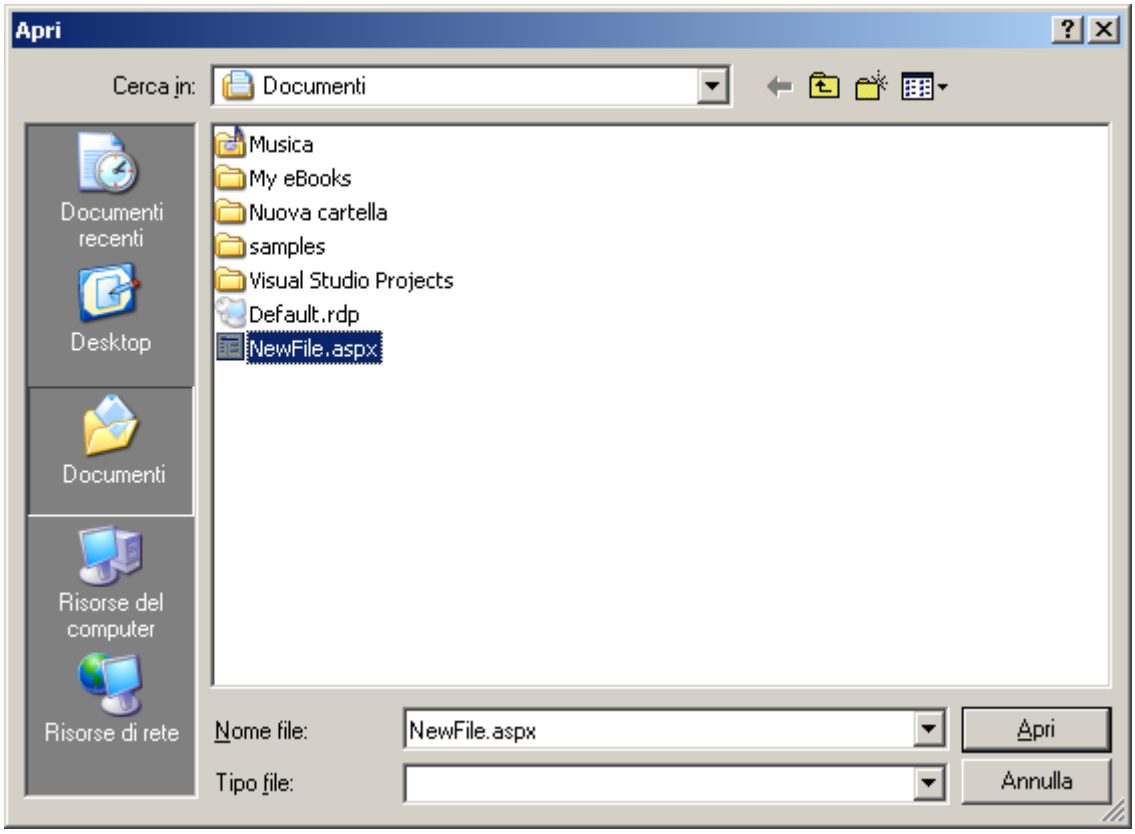


Figura 15-3 Output del programma

La *dialog* contiene tutti gli elementi di base che caratterizzano una tipica finestra di dialogo per l'apertura file. La proprietà più importante è `FileName`, attraverso la quale è possibile ottenere (ma

anche impostare prima di eseguire la *dialog*) il nome del file selezionato dall'utente. Da notare che esso è completo di percorso e di estensione, anche nel caso in cui non venga specificata dall'utente.

6.2 Richiedere all'utente di scegliere un font: «FontDialog»

La finestra di dialogo prodotta nei paragrafi precedenti, che richiede all'utente di inserire il nome e la dimensione di un font è piuttosto povera e poco funzionale. In realtà non esiste alcun bisogno di realizzarne una più sofisticata, poiché questa esiste già, ed è la `FontDialog`. Essa consente di selezionare tutte le caratteristiche di un font e crea automaticamente un oggetto font al quale il codice chiamante può accedere attraverso la proprietà `Font`.

Anche la classe `FontDialog`, come `OpenFileDialog`, espone numerose proprietà che consentono al programmatore di personalizzare la finestra di dialogo. In questa sede ci limiteremo a considerarne soltanto cinque.

Tabella 15-2 proprietà di base della classe `FontDialog`

PROPRIETÀ - TIPO	DESCRIZIONE
`Color` `Color`	Consente di ottenere il colore del tipo di carattere selezionato.
`Font` `Font`	Consente di ottenere il font selezionato.
`MaxSize` `int`	Dimensione massima ammessa (espressa in punti) del font[15].
`MinSize` `int`	Dimensione minima ammessa (espressa in punti) del font(*).
`ShowColor` `bool`	Se `true` fa sì che la finestra di dialogo mostri all'utente un elenco di colori da selezionare.

L'applicazione che segue dimostra l'impiego della finestra `FontDialog`. In risposta al clic sul bottone «Font» viene aperta una `FontDialog`. Nel caso l'utente confermi la *dialog*, viene utilizzata la proprietà `Font` per impostare il font di una etichetta, altrimenti viene visualizzato un messaggio informativo.

```
public partial class Form1: Form
{
    // ...
    void btnFont_Click(object sender, EventArgs e)
    {
        FontDialog dgFont = new FontDialog();
        dgFont.ShowColor = true;
        dgFont.MinSize = 10;
        dgFont.MaxSize = 22;
        DialogResult cmd = dgFont.ShowDialog();
        if (cmd != DialogResult.OK)
        {
```

[15] Un punto equivale a 1/72 di pollice

```
            MessageBox.Show("L’utente ha annullato la dialog");
        }
        else
        {
            lblTest.Font = dgFont.Font;
            lblTest.ForeColor = dgFont.Color;
        }
    }
}
```

GDI+: Funzionalità grafice del .NET

1 Introduzione al «GDI+»

Il termine «GDI+» (*Graphics Device Interface*, *plus*) designa lo strato di software demandato al disegno (rendering) di forme geometriche, immagini e testi che formano i controlli o in generale che rappresentano il contenuto di un documento. GDI+ comprende un insieme di tipi, oggetti e funzioni che consentono di:

- disegnare forme geometriche, rette, curve, rettangoli, ellissi, poligoni, eccetera, sia tracciando il solo contorno, sia riempiendo l'interno della figura;
- gestire l'aspetto del colore e dello stile, sia per quanto riguarda i contorni che i riempimenti;
- scrivere testi, con la possibilità di intervenire sul font utilizzato, sull'orientamento e sull'allineamento del testo, eccetera;
- disegnare e manipolare immagini.

Tali funzionalità sono definite nei seguenti *namespaces*:

Tabella 16-1 *Namespace* che definiscono le funzionalità di GDI+oprietà di base della classe `FontDialog`

NAMESPACE	DESCRIZIONE
`System.Drawing`	Definisce le funzionalità grafiche di base di GDI+, rappresentate dagli oggetti `Pen`, `Font`, `Brush`, `Graphics`, eccetera.
`System.Drawing.Drawing2D`	Definisce le funzionalità grafiche avanzate; come ad esempio realizzazione di gradienti (sfumature), trasformazioni geometriche, eccetera.
`System.Drawing.Imaging`	Definisce le funzionalità grafiche avanzate per la manipolazione delle immagini.
`System.Drawing.Printing`	Definisce le funzionalità necessarie di rendere, figure, testo e immagini nella stampante, per personalizzare il lavoro di stampa, eccetera.
`System.Drawing.Text`	Definisce le funzionalità per la gestione della collezione di font installati sul sistema.

Nel resto del capitolo saranno prese in esame le funzionalità di base, definite nel *namespace* `System.Drawing`.

1.1 Oggetto «Graphics»: accesso alla superficie di disegno

L'elemento centrale delle operazioni di disegno è un oggetto di tipo `Graphics`. Esso incapsula una superficie di disegno, associata a un controllo, a una pagina di stampa o a un'immagine, e i metodi per disegnare su di essa.

Un oggetto `Graphics` non è esiste di per sé; esso dev'essere creato ogni qual volta è necessario eseguire delle operazioni di disegno, e ciò viene fatto sempre in riferimento a un determinato oggetto (controllo, pagina di stampa, immagine). Esso rappresenta dunque un riferimento alla

superficie dell'oggetto attraverso il quale è stato creato, ed è in grado di disegnare soltanto all'interno di essa.

Per il momento prenderemo in considerazione soltanto oggetti `Graphics` ottenuti attraverso un controllo dell'interfaccia, ad esempio un `Panel`.

1.2 «*Drawing*» e «*painting*»

Drawing e *painting* sono due termini convenzionali, utili per precisare il contesto all'interno del quale avvengono le operazioni di disegno.

Per capire il concetto consideriamo il seguente esempio. Immaginiamo di avere un form contenente un bottone e un'immagine, visualizzata mediante un controllo `PictureBox`. Si supponga inoltre che associato all'evento `Click` del bottone vi sia un gestore d'evento che, dopo aver ottenuto un oggetto `Graphics` dell'area del form, disegni su di esso un cerchio. (E vedremo più avanti come si fa). Ora, si supponga di ridurre a icona il form e di ripristinarlo subito dopo. Ebbene, sia il bottone che l'immagine sono rimasti al loro posto, mentre il cerchio è scomparso. Ovviamente viene disegnato di nuovo se si clicca sul bottone.

La questione sta in questi termini: tutto ciò che viene disegnato mediante un oggetto `Graphics` non ha in sé la caratteristica della persistenza. Qualsiasi evento che "sporchi" l'area nel quale è stato prodotto il disegno, "sporca" anche il disegno stesso, cancellandolo in tutto o in parte. Perché allora sia il bottone che l'immagine vengono ripristinati insieme al form? Perché entrambi eseguono automaticamente il codice necessario al loro disegno in risposta all'evento `Paint` generato da Windows. Windows "sa" quando una certa area del form ha bisogno di essere nuovamente visualizzata (il termine tecnico è «rinfrescata»), ed a questo scopo solleva l'evento `Paint` relativo all'area in questione. Tutti i controlli che risiedono in tutto o in parte in quest'area rispondono automaticamente eseguendo il proprio codice di disegno. Ebbene, le operazioni di disegno in sé, eseguite in risposta a un evento qualsiasi, ricadono nell'ambito del *drawing*. Le operazioni di disegno eseguite in risposta a un evento `Paint` ricadono nell'ambito del *painting*, ed hanno lo scopo di rendere persistenti gli oggetti disegnati, poiché vengono eseguite ogni volta che l'area del controllo necessita di essere rinfrescata.

Nel proseguo del capitolo ci concentreremo sull'aspetto relativo al *drawing*. Nel capitolo successivo vedremo come la gestione dell'evento `Paint` non solo consente di rendere persistenti gli oggetti disegnati, ma anche di personalizzare il rendering dei controlli, o addirittura di realizzarne di propri.

2 Disegno di figure geometriche

Le funzionalità di disegno geometrico sono senz'altro alla base di GDI+. Oltre alle figure geometriche fondamentali, linea, rettangolo (quadrato), ellisse (cerchio), esistono metodi per disegnare curve, poligoni, torte, nonché figure composite, costruite sulla base delle forme suddette.

Ogni operazione di disegno avviene nell'ambito del sistema di coordinate dell'oggetto `Graphics`. L'angolo in alto a sinistra, di coordinate (0, 0), rappresenta l'origine, quello in basso a destra il punto di massime coordinate, equivalenti alle dimensioni dell'area di disegno. Le coordinate possono essere espresse in sei unità di misura diverse, delle quali prenderemo in considerazione soltanto quella predefinita e cioè il pixel.

L'altro aspetto fondamentale che caratterizza ogni operazione è lo strumento grafico impiegato per eseguirla. Ne esistono due, `Pen` e `Brush`, (penna e pennello) utilizzati rispettivamente per disegnare i contorni e gli interni delle figure. In questo senso esistono due classi di operazioni:

- le operazioni «*draw*», e cioè di tracciatura, che richiedono un oggetto `Pen`;

- le operazioni «*fill*»,e cioè di riempimento, che richiedono un oggetto `Brush`.

Si immagini ora di aver già ottenuto un oggetto `Graphics` relativamente a un controllo dell'interfaccia, ad esempio un `Panel`; abbiamo cioè cominciato una sessione di disegno in risposta a un certo evento. Abbiamo a disposizione:

- un'area di disegno, rappresentata dall'oggetto `Graphics`;
- delle penne predefinite, caratterizzate da un colore predefinito e da uno spessore di un pixel; ognuna di esse è rappresentata da una proprietà della classe `Pens` che ha un nome di colore;
- dei pennelli predefiniti, di tipo «a tinta unita» e caratterizzati da un colore predefinito; essi colorano in modo uniforme la figura geometrica alla quale sono applicati; ognuno di essi è rappresentato da una proprietà della classe `Brushes` che un nome di colore;
- la possibilità di creare penne e pennelli personalizzati;

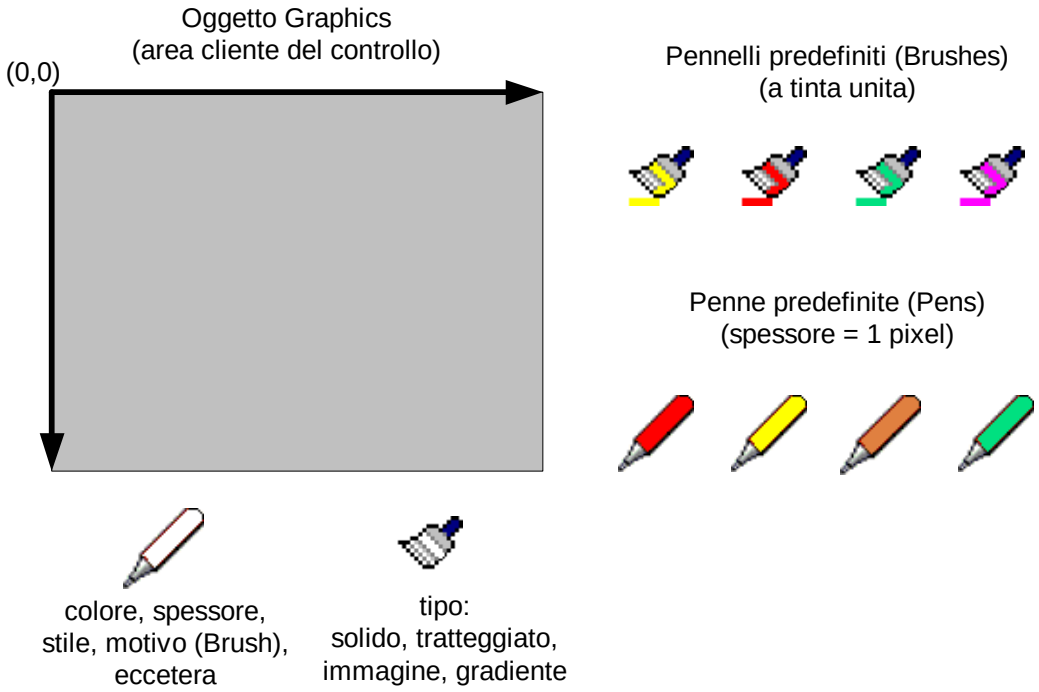


Figura 16-1 Rappresentazione schematica di una sessione di disegno

Ogni operazione di disegno geometrico richiede di specificare uno strumento, `Pen` o `Brush`, in base alla natura dell'operazione, *draw* o *fill*. L'oggetto può essere uno di quelli predefiniti, oppure può essere creato per l'occasione, e riutilizzato quante volte si vuole.

Vediamo un breve esempio. L'interfaccia del programma che segue è composta da un oggetto di tipo `Panel` chiamato `pnlGrafico` ed un bottone chiamato `btnDisegna`. In risposta al clic sul bottone, nell'area cliente del pannello vengono disegnati un quadrato circoscritto a un cerchio, il primo comprensivo delle due diagonali.

```
public partial class Form1: Form
{
    // . . .
    void btnDisegna_Click(object sender, EventArgs e)
    {
```

```
        Graphics g = pnlGrafico.CreateGraphics();
        Pen penQuadrato = new Pen(Color.Red, 3);
        g.DrawRectangle(penQuadrato, 25, 25, 100, 100);
        g.FillEllipse(Brushes.Yellow, 25, 25, 100, 100);
        g.DrawLine(Pens.Blue, 25, 25, 125, 125);
        g.DrawLine(Pens.Green, 125, 25, 25, 125);
    }
}
```

Di seguito è mostrato l'output prodotto.

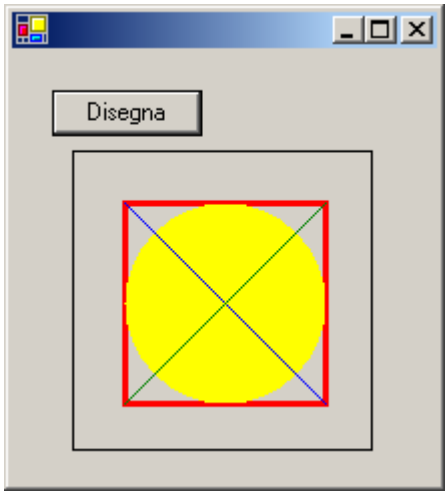


Figura 16-2 Output del programma

Esaminiamo brevemente le istruzioni. Nella prima:

```
Graphics g = pnlGrafico.CreateGraphics();
```

mediante l'invocazione del metodo `CreateGraphics()` si ottiene un oggetto di tipo `Graphics` associato all'area cliente del pannello `pnlGrafico`. Il riferimento all'oggetto è memorizzato nella variabile g, utilizzata per le successive operazioni di disegno. L'istruzione:

```
Pen penQuadrato = new Pen(Color.Red, 3);
```

crea una penna di colore rosso dallo spessore di 3 pixel e l'assegna alla variabile `penQuadrato`. L'istruzione:

```
g.DrawRectangle(penQuadrato, 25, 25, 100, 100);
```

disegna un rettangolo utilizzando la penna precedentemente creata. I primi due valori (25, 25) rappresentano le coordinate (X, Y) dell'angolo in alto a sinistra del rettangolo. La seconda coppia ne rappresenta le dimensioni (larghezza, altezza). Le istruzioni:

```
g.DrawLine(Pens.Blue, 25, 25, 125, 125);
g.DrawLine(Pens.Green, 125, 25, 25, 125);
```

disegnano le due diagonali del quadrato, utilizzando due penne predefinite. In questo caso le due coppie di valori specificate in entrambe le chiamate al metodo `DrawLine()` rappresentano le coordinate (X, Y) dei vertici della linea. Infine, l'istruzione:

```
g.FillEllipse(Brushes.Yellow, 25, 25, 100, 100);
```

disegna l'interno di una ellisse mediante un pennello predefinito. Posizione e dimensioni dell'ellisse sono espresse specificando posizione e dimensioni del rettangolo che la circoscrive; nella fattispecie rappresentato dal quadrato precedentemente disegnato.

2.1 Strumenti di disegno: oggetti «Pen» e «Brush»

Come abbiamo già detto, sia `Pen` che `Brush` possono essere ottenuti da un insieme di oggetti predefiniti già disponibili, oppure creati per l'occasione. Nel secondo caso è possibile specificarne le caratteristiche sia in fase di costruzione, sia dopo averli creati.

Al di là della loro natura, esiste una sostanziale differenza tra penne e pennelli. Infatti, c'è un solo tipo di penna, rappresentato dalla classe `Pen`, attraverso il quale è possibile creare oggetti con caratteristiche diverse. Esistono invece più tipi di pennello, che si differenziano dalla modalità di riempimento prodotta, tutti che derivano dal tipo generico `Brush`.

Tipi di pennello disponibili

Esistono cinque tipi di pennello, dei quali gli ultimi tre sono definiti nel *namespace* `System.Drawing.Drawing2D`:

Tabella 16-2 Tipi di pennello

TIPO	DESCRIZIONE
`SolidBrush`	Pennello a tinta unita, caratterizzato dal solo colore di riempimento.
`TextureBrush`	Pennello definito attraverso un'immagine.
`HatchBrush`	Pennello con stile tratteggiato. E' caratterizzato dallo stile di tratteggio, da un colore di primo piano e uno di sfondo (utilizzato negli spazi fra i tratteggi).
`LinearGradientBrush`	Pennello definito attraverso una sfumatura lineare, e cioè un cambiamento graduale, in un sola direzione, da un colore iniziale e uno finale.
`PathGradientBrush`	Pennello definito attraverso una insieme di sfumature, lineari e non.

A titolo di esempio mostriamo come creare un `HatchBrush` per disegnare un cerchio con stile di riempimento a righe diagonali gialle su sfondo verde (è necessario definire il *namespace* `System.Drawing.Drawing2D`:

```
HatchBrush hb = new HatchBrush(HatchStyle.BackwardDiagonal,
                               Color.Yellow, Color.Green);
g.FillEllipse(hb, 135, 25, 100, 100);
```

Il primo parametro, `HatchStyle.BackwardDiagonal`, definisce lo stile del tratteggio (diagonali retroverse). Tutti gli stili, ne esistono più di 50, appartengono al tipo enumeratore `HatchStyle`.

Tipo «Pen»

Il tipo `Pen` definisce delle proprietà attraverso le quali personalizzare lo stile usato per tracciare, ottenendo risultati anche molto sofisticati. Segue un elenco delle proprietà di base, per alcune delle quali il tipo è definito nel *namespace* `System.Drawing.Drawing2D`:

Tabella 16-3 Proprietà di base del tipo `Pen`

PROPRIETÀ - TIPO	DESCRIZIONE
`Brush` `Brush`	Consente di utilizzare un pennello per definire lo stile della penna.
`Color` `Color`	Colore della penna.
`DashStyle` `DashStyle`	Consente di definire un stile tratteggiato. Possibili valori sono: `Custom`: personalizzato; `Dash`: trattino; `DashDot`: trattino-punto; `DashDotDot`: trattino-punto-punto; `Dot`: punto; `Solid`: continuo.
`StartCap` e **`EndCap`** `LineCap`	Consente di impostare il tipo di inizio e di terminazione di una linea. Tra i possibili valori ci sono: `ArrowAnchor`: freccia; `DiamondAnchor`: rombo; `RoundAnchor`: cerchio; `SquareAnchor`: quadrato.
`Width` `int`	Larghezza del tratto.

A titolo di esempio mostriamo come disegnare un linea orizzontale tratteggiata che inizi con un quadrato e termini con una freccia:

```
Pen p = new Pen(Color.Blue);
p.StartCap = LineCap.SquareAnchor;
p.EndCap = LineCap.ArrowAnchor;
p.DashStyle = DashStyle.Dash;
g.DrawLine (p, 20, 20, 120, 20);
```

2.2 Metodi di disegno geometrico

Il tipo `Graphics` definisce un numero elevato di proprietà e metodi di disegno. Oltre a quelli già visti nei paragrafi precedenti, vi sono metodi per disegnare sequenze di linee e di rettangoli, curve e figure composite chiamate «percorsi».

Tabella 16-4 Metodi di disegno geometrico della classe `Graphics`

NOME	DESCRIZIONE
`DrawArc()`	Disegna un arco di ellisse. Richiede le dimensioni dell'ellisse, l'angolo iniziale in gradi (a partire dall'asse X) e la misura in gradi dell'arco.
`DrawBezier()` e **`DrawBeziers`**	Il primo disegna una curva spline di Bézier. Il secondo disegna una serie di curve spline di Bézier.
`DrawCurve()` e **`DrawClosedCurve()`**	Disegnano una curva spline.
`DrawEllipse`	Disegna un'ellisse circoscritto a un rettangolo. Richiede le coordinate e le dimensioni del rettangolo.

DrawLine() e **DrawLines**	IL primo disegna una linea, e richiede le coordinate dei due vertici. Il secondo disegna una serie di linee, le cui coordinate dei vertici sono memorizzare in un vettore di `Point`. (`Point[]`)
DrawPath()	Disegna un «percorso» e cioè una figura composita di tipo `GraphicsPath`.
DrawPie()	Disegna una torta (sezione di ellisse). Richiede le dimensioni dell'ellisse, l'angolo in gradi iniziale (a partire dall'asse X) e la misura in gradi della sezione.
DrawPoligon()	Disegna un poligono. Le coordinate dei vertici del poligono sono memorizzati in un vettore di `Point`. (`Point[]`)
DrawRectangle() e **DrawRectangles()**	Il primo disegna un rettangolo. Il secondo disegna una serie di rettangoli, le cui coordinate sono memorizzate in un vettore di `Rectangle`. (`Rectangle[]`)
FillClosedCurve	Riempie l'area interna di una curva spline.
FillEllipse	Riempie l'area interna di una ellisse.
FillPath	Riempie l'area interna di un oggetto `GraphicsPath`.
FillPie()	Riempie l'area interna di una torta.
FillPolygon()	Riempie l'area interna di un poligono.
FillRectangle() e **FillRectangles()**	Il primo riempie l'area interna di un rettangolo. Il secondo riempie l'area interna di una serie di rettangoli.
FillRegion()	Riempie l'area interna di una «regione», e cioè di un oggetto di tipo `Region`.

I metodi elencati sono forniti in più versioni, ognuna delle quali ammette un diverso modo per specificare le coordinate dell'oggetto da disegnare, singolarmente oppure attraverso oggetti `Point` e `Rectangle` (o `PointF` e `RectangleF`). Considerato che il tipo `Point` è stato brevemente introdotto nel terzo capitolo, qui prenderemo in esame il tipo `Rectangle`.

2.3 Tipi «Rectangle» e «RectangleF»

Un oggetto di tipo `Rectangle`, o `RectangleF`, memorizza la posizione e le dimensioni di un'area rettangolare. La differenza tra i due tipi consiste nel fatto che un oggetto di tipo `RectangleF`, consente di memorizzate coordinate `float`, in modo da poter gestire unità di misura diverse dal pixel. Per il resto i due sono tipi omologhi, e dunque qui faremo riferimento al solo tipo `Rectangle`.

Un oggetto `Rectangle` viene definito dalle coordinate dell'angolo superiore sinistro e dalle dimensioni, larghezza e altezza. Esso espone proprietà e metodi che consentono di impostare, conoscere e manipolare l'area rappresentata, nonché di combinare o confrontare due oggetti `Rectangle`. In figura sono indicate le proprietà sulla base della loro relazione con l'area rappresentata; come si vede alcune di esse esprimono lo stesso valore, se pur con un nome o in una forma diversa:

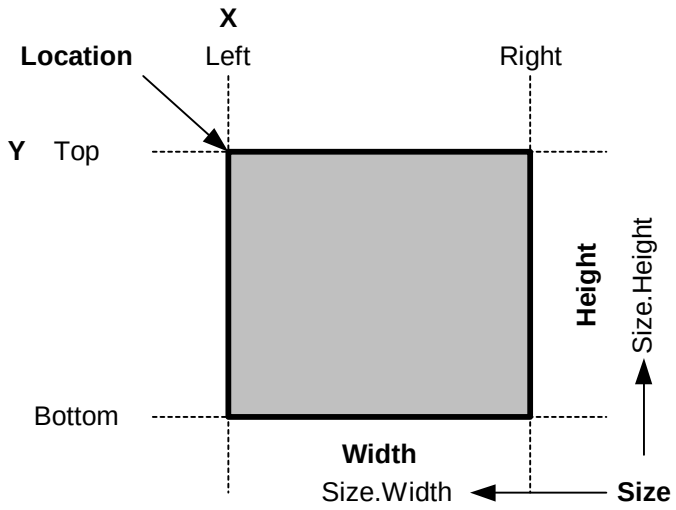


Figura 16-3 Proprietà del tipo `Rectangle`[16]

Segue l'elenco dei metodi principali (non sono riportati i metodi di conversione da un oggetto `RectangleF` e un oggetto `Rectangle`):

Tabella 16-5 Metodi del tipo `Rectangle`

NOME	DESCRIZIONE
`Contains()`	Ritorna `true` se il punto o il rettangolo specificato come argomento è contenuto nell'oggetto.
`FromLTRB()`	Metodo statico che ritorna un oggetto equivalente alle coordinate `Left`, `Top`, `Right` e `Bottom` specificate. (Tali coordinate sono anche chiamate: «posizioni di bordo»).
`Inflate()`	Ingrandisce (o rimpicciolisce) l'oggetto sia in larghezza che in altezza in base agli argomenti specificati. Ad esempio, supposto che esista già l'oggetto rect: `rect.Inflate(10, 5);` aumenta la larghezza e l'altezza dell'oggetto rispettivamente di 10 e 5. (Esiste una versione statica del metodo.)
`Intersect()`	Esiste in due versioni. Nella prima accetta come argomento un `Rectangle` e sostituisce all'oggetto l'intersezione tra esso e l'argomento specificato. La seconda, statica, accetta due oggetti `Rectangle` e ne restituisce l'intersezione.
`IntersectWith()`	Ritorna `true` se l'oggetto interseca il `Rectangle` passato come argomento, `false` altrimenti.
`Offset()`	Modifica la posizione dell'oggetto aggiungendo le coordinate specificate come argomento (espresse mediante una coppia di valori o mediante un `Point`). Ad esempio, supposto che esista già l'oggetto rect: `rect.Offset(-5, 10);` modifica le coordinate (X,Y) dell'oggetto diminuendo la X di 5 e aumentando la Y di 10.
`Union()`	Metodo statico che accetta due oggetti `Rectangle` e ne ritorna uno che rappresenta la loro unione (delle loro aree).

Agli oggetti di tipo `Rectangle` sono inoltre applicabili gli operatori "==" e "`!=`", che verificano se due oggetti hanno posizione e dimensioni uguali (o diverse).

[16] Le proprietà scritte in neretto sono di tipo get e set, le altre di tipo get (sola lettura)

Anche senza considerare sofisticate operazioni di manipolazione delle aree di disegno (necessarie in operazioni di rendering realistiche), gli oggetti `Rectangle` sono utili poiché consentono di memorizzare le coordinate di un'area in un'unica variabile, che può essere utilizzata con più metodi disegno. E' questo il caso dell'esempio di disegno di figure geometriche fatto all'inizio del paragrafo precedente, nel quale il quadrato e il cerchio sottendono la stessa area rettangolare. Ecco come può essere riscritto il gestore di evento `btnDisegna_Click()`:

```
void btnDisegna_Click(object sender, EventArgs e)
{
    Graphics g = pnlGrafico.CreateGraphics();
    Pen penQuadrato = new Pen(Color.Red, 3);
    Rectangle r = new Rectangle(25, 25, 100, 100);
    g.DrawRectangle(penQuadrato, r);
    g.FillEllipse(Brushes.Yellow, r);
    g.DrawLine(Pens.Blue, 25, 25, 125, 125);
    g.DrawLine(Pens.Green, 125, 25, 25, 125);
}
```

Inoltre, una volta definito il rettangolo di disegno, diventa semplice disegnare il cerchio senza che il contorno si sovrapponga parzialmente a quello del quadrato. E' sufficiente diminuire l'area del rettangolo mediante il metodo `Inflate()` prima di usarlo per disegnare il cerchio:

```
void btnDisegna_Click(object sender, EventArgs e)
{
    Graphics g = pnlGrafico.CreateGraphics();
    Pen penQuadrato = new Pen(Color.Red, 3);
    Rectangle r = new Rectangle(25, 25, 100, 100);
    g.DrawRectangle(penQuadrato, r);
    r.Inflate(-1,-1);
    g.FillEllipse(Brushes.Yellow, r);
    g.DrawLine(Pens.Blue, 25, 25, 125, 125);
    g.DrawLine(Pens.Green, 125, 25, 25, 125);
}
```

3 Esempio di una applicazione di disegno geometrico

Come esempio di disegno geometrico un po' più concreto affrontiamo il problema della visualizzazione del grafico di una funzione. I punti della funzione devono essere stati calcolati in precedenza e memorizzati in una matrice di `double` di due colonne, la prima per le ascisse, la seconda per le ordinate. L'obiettivo è quello realizzare una classe che incapsuli:

- una superficie di disegno;
- il codice per convertire le coordinate reali (X, Y) dei punti della funzione nelle corrispondenti coordinate intere relative all'area di disegno;
- il codice per disegnare il grafico mediante una serie di linee che uniscono i punti precedentemente calcolati.

3.1 Conversione da un sistema di coordinate reali a un sistema di coordinate intere

Ecco i termini del problema. Siano dati un sistema cartesiano in coordinate reali qualsiasi (d'ora in avanti «sistema reale») e un sistema cartesiano in coordinate intere (d'ora in avanti «sistema intero») la cui origine, di coordinate (0, 0) si trova nell'angolo in alto a sinistra:

trovare le funzioni di conversione delle coordinate (X,Y) dal primo al secondo sistema.

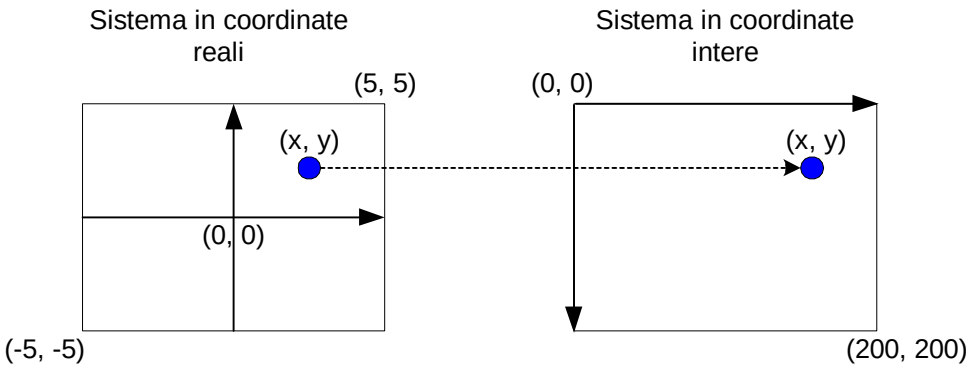


Figura 16-4 Rappresentazione dei sistemi di coordinate reale ed intero[17]

In figura, la freccia sulla linea tratteggiata mostra la direzione delle conversioni da effettuare. Le coordinate minime e massime del sistema reale state messe a titolo di esempio e potrebbero essere valori qualsiasi; ciò vale anche per le coordinate massime del sistema intero (quelle minime sono sempre zero).

Parametri che definiscono i due sistemi

Il sistema reale è definito da due coppie di valori reali, che rappresentano le coordinate minime e massime del sistema; stabiliamo di chiamarle: `minX`, `minY`, `maxX`, `maxY`.

Il sistema intero è definito da una coppia di valori che rappresentano le sue dimensioni (poiché l'origine è sempre 0, 0); stabiliamo di rappresentare tale coppia tramite un oggetto di tipo `Size`, che chiameremo `size`.

Calcolare i fattori di scala

Calcolare i fattori di scala significa calcolare quante unità intere (pixel, nel nostro caso) corrispondono ad una unità reale. Ciò si ottiene dividendo le dimensioni del sistema intero per le dimensioni del sistema reale. Occorre trovare due fattori di scala, uno per l'asse delle ascisse, l'altro per l'asse delle ordinate. Ecco le formule:

```
scalaX = size.Width / (maxX - minX);
scalaY = size.Height / (maxY - minY);
```

Funzioni di conversione

Per funzione di conversione si intende una funzione che data una coordinata nel sistema reale produca la coordinata corrispondente nel sistema intero. Ciò si ottiene semplicemente calcolando la distanza dall'origine della coordinata reale e moltiplicando tale risultato per il fattore di scala. Ecco la funzione di conversione per le ascisse:

[17] Le proprietà scritte in neretto sono di tipo get e set, le altre di tipo get (sola lettura)

```
xIntero = (xReale – minX) * scalaX;
```

Il valore risultante dev'essere ovviamente convertito nel tipo `int`.

La conversione delle coordinate Y richiede un'operazione aggiuntiva, in quanto l'asse delle ordinate del sistema intero ha un verso opposto rispetto a quello del sistema reale. Poiché desideriamo che la rappresentazione del grafico sia quella consueta (con lo zero in basso), occorre, dopo aver ottenuto la coordinata intera, calcolare la sua speculare rispetto all'asse X. Ciò si ottiene semplicemente sottraendo la coordinata alla dimensione in Y:

```
yIntero = size.Height - (yReale – minY) * scalaY;
```

Anche in questo caso occorre convertire il risultato nel tipo `int`.

3.2 Scheletro della classe «Grafico»

La classe, di nome `Grafico`, deriva dal tipo `Panel`. L'interfaccia pubblica definisce:

- un costruttore che accetta come argomenti le dimensioni del sistema reale;
- il metodo `Disegna()`, che richiede come argomento la matrice contenente le coordinate dei punti e visualizza il grafico come insieme di linee che uniscono i punti;
- la proprietà a sola scrittura `Penna`, che consente al codice *consumer* di impostare la penna da usare per il disegno; (altrimenti sarà usata una penna di un pixel di colore equivalente alla proprietà `ForeColor` dell'oggetto).

Per uso interno, `Grafico` implementa due metodi necessari per la conversione delle coordinate dal sistema reale al sistema intero.

Segue lo scheletro della classe. Il costruttore e la proprietà `Penna` sono già implementati:

```
class Grafico: Panel
{
    double scalaX, scalaY;
    Pen penna;
    double minX, minY, maxX, maxY;

    public Grafico(double minX, double minY, double maxX, double maxY)
    {
        this.minX = minX;
        this.minY = minY;
        this.maxX = maxX;
        this.maxY = maxY;
    }

    public Pen Penna
    {
        set { penna = value; }
    }

    public void Disegna(double[,] punti) {...}

    Point XYtoP(double x, double y) {...}
```

```
    Point[] ConvertiCoordinate(double[,] punti) {...}

}
```

Così come è stata progettata, la classe `Grafico` non brilla di certo per eleganza e funzionalità. D'altra parte, ciò che ci interessa qui è semplicemente mostrare l'uso di alcune delle funzionalità di disegno geometrico di GDI+.

Implementazione dei metodi di conversione delle coordinate

L'aspetto matematico è già stato analizzato e dunque non c'è bisogno di particolari commenti. Il primo metodo accetta due coordinate reali e ritorna un valore `Point` corrispondente. Il secondo metodo calcola i fattori di scala e quindi richiama il primo per ogni coppia di coordinate reali, ritornando un vettore di `Point`.

```
Point XYtoP(double x, double y)
{
    int xIntero = (int) ((x – minX) * scalaX);
    int yIntero = ClientSize.Height - (int)((y - minY) * scalaY);
    return new Point(xIntero, yIntero);
}


Point[] ConvertiCoordinate(double[,] punti)
{
    scalaX = ClientSize.Width / (maxX - minX);
    scalaY = ClientSize.Height / (maxY - minY);
    Point[] puntiInteri = new Point[punti.GetLength(0)];
    for(int i = 0; i < puntiInteri. Length; i++)
        puntiInteri[i] = XYtoP(punti[i, 0], punti[i, 1]);
    return puntiInteri;
}
```

3.3 Metodo «Disegna()»

L'implementazione del metodo `Disegna()` è piuttosto semplice. Come prima cosa le coordinate reali vengono convertite nelle coordinate intere corrispondenti. Quindi si ottiene un oggetto `Graphics` relativo all'oggetto mediante il metodo `CreateGraphics()`. Viene ripulita l'area di disegno mediante il metodo `Clear()` dell'oggetto `Graphics` utilizzando il colore memorizzato in `BackColor`. Successivamente viene verificata l'esistenza di una penna, se così non è ne viene creata una con il colore memorizzato nella proprietà `ForeColor`. Infine viene eseguito il codice di disegno; esso fa uso del metodo `DrawLines()`.

```
public void Disegna(double[,] punti)
{
    Penna pennaGrafico;
    Point[] puntiInteri = ConvertiCoordinate(punti);
    Graphics g = CreateGraphics();
    g.Clear(BackColor);                          // cancella l'area di disegno
```

```
    if (penna == null)
        pennaGrafico = new Pen(ForeColor, 1);
    else
        pennaGrafico = penna;
    g.DrawLines(pennaGrafico, puntiInteri);                // disegna il grafico
}
```

Il metodo `DrawLines()` unisce con una linea ogni punto del vettore al precedente, creando un insieme di "spezzate". Di norma, nel caso di una funzione continua, per ottenere un grafico rappresentativo è opportuno calcolare un elevato numero di punti; altrimenti, alternativamente a `DrawLines()` è possibile usare `DrawCurve()`, il quale unisce i punti introducendo artificialmente delle curvature, evitando così degli angoli netti nella congiunzione tra due linee.

Segue un esempio di codice *consumer* che usa la classe, visualizzando il grafico della funzione «seno»:

```
public partial class Form1: Form
{
    // . . .
    const double minX = -5;
    const double minY = -2;
    const double maxX = 5;
    const double maxY = 2;

    Grafico pnlGrafico;
    Button btnDisegna;
    void btnDisegna_Click(object sender, EventArgs e)
    {
        double dx = (maxX - minX) / pnlGrafico.ClientSize.Width;
        double[,] punti = new double[pnlGrafico.ClientSize.Width, 2];
        double x = -5;
        for (int i = 0; i <punti.GetLength(0); i++)
        {
            double y = Math.Sin(x);
            punti[i,0] = x;
            punti[i,1] = y;
            x += dx;
        }
        pnlGrafico.Penna = new Pen(Color.Red, 2);
        pnlGrafico.Disegna(punti);
    }
}
```

Nell'esempio si è presupposto che le dimensioni del sistema reale siano predefinite e costanti. Da notare che il numero dei punti calcolati equivale alla larghezza della *client area* del pannello. E' questa una scelta appropriata, poiché questo valore rappresenta il massimo numero di punti significativi graficamente rappresentabili (infatti, in larghezza non esistono abbastanza pixel per rappresentare un numero di punti superiore).

Di seguito è mostrato l'output dell'applicazione.

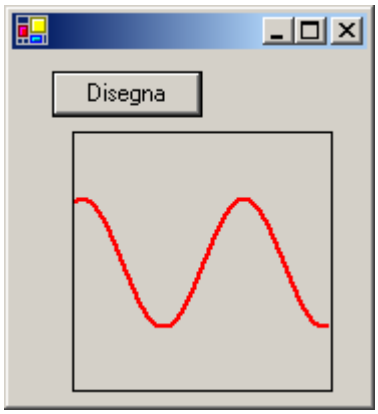


Figura 16-5 Output del programma

4 Disegnare immagini

La relazione tra immagini – memorizzate in oggetti `Bitmap`, `Metafile`, `Icon` – e oggetti di tipo `Graphics` si presenta sotto due prospettive diverse. La prima vede l'immagine come un oggetto da disegnare sulla superficie rappresentata dall'oggetto `Graphics`, e in questo senso non esiste una sostanziale differenza tra il disegno di immagini e quello di figure geometriche. La seconda prospettiva, opposta, vede l'immagine come la superficie sulla quale disegnare attraverso un oggetto `Graphics`.

In questa sede prenderemo in esame soltanto la prima.

4.1 Metodi di rendering delle immagini

La classe `Graphics` espone quattro metodi per il disegno di immagini.

Tabella 16-6 Metodi di disegno di immagini

NOME	DESCRIZIONE
`DrawIcon()`	Disegna l'icona specificata. Una versione del metodo richiede un oggetto `Rectangle` per indicare la posizione e le dimensioni dell'icona; questa viene adattata all'oggetto in questione.
`DrawIconUnstretched()`	Disegna l'icona specificata. Il metodo richiede un oggetto `Rectangle` per indicare la posizione e le dimensioni dell'icona; se questa ha dimensione maggiori del rettangolo, la parte eccedente viene tagliata.
`DrawImage()`	Disegna l'immagine. Il metodo è fornito in un numero elevato di versioni. Alcune di esse richiedono di specificare soltanto la posizione, nel qual caso l'immagine viene riprodotta nelle sue dimensioni originali. Altre richiedono di specificare un rettangolo o una struttura a parallelogramma; in questo caso l'immagine viene adattata.
`DrawImageUnscaled()`	Disegna l'immagine specificata. Questa viene sempre visualizzata nelle sue dimensioni originali. La versione del metodo che richiede un rettangolo per specificare posizione e dimensioni taglia l'eventuale parte eccedente dell'immagine.

Esistono due coppie di metodi distinte, perché il tipo `Icon`, diversamente dai tipi `Bitmap` e `Metafile`, non deriva dal tipo astratto `Image`. Dunque, quando si desidera visualizzare un'icona è necessario usare i metodi `DrawIconXXX()`, i quali accettano come primo argomento un oggetto di tipo `Icon`. In tutti gli altri casi è necessario utilizzare i metodi `DrawImageXXX()`, che accettano come primo argomento un oggetto di tipo `Image`, e dunque anche `Bitmap` e `Metafile`

4.2 Tipi «Icon», «Image», «Bitmap» e «Metafile»

Segue una breve introduzione ai tipi impiegati per la gestione di immagini e icone. Di questi, i tipi `Bitmap` e `Metafile` derivano dal tipo `Image`, e quindi ereditano da esso proprietà e metodi.

Tutti e quattro i tipi espongono le proprietà a sola lettura `Width`, `Height` e `Size`, le quali memorizzano le dimensioni (in pixel) dell'immagine.

Tipo «Icon»

Gli oggetti di tipo `Icon` sono in grado di memorizzare icone nel formato Windows; a livello di file queste hanno l'estensione "ICO". Il tipo `Icon` espone il metodo `ToBitmap()`, il quale crea e ritorna un oggetto `Bitmap` equivalente all'icona.

Tipo «Image»

`Image` è una classe astratta utilizzata come tipo base per manipolare immagini di qualsiasi formato: "BMP", "JPG", "GIF", "GIF animate", "PNG", "WMF", "EMF", "TIFF", eccetera. `Image` definisce la maggior parte delle proprietà e dei metodi necessari per eseguire le operazioni più comuni sulle immagini. Tra questi, di impiego molto comune è il metodo statico `FromFile()`, che legge un'immagine dal file specificato come argomento, interpretandone automaticamente il formato.

Una caratteristica importante degli oggetti `Image` è data dal fatto che è possibile ottenere da essi un oggetto `Graphics` attraverso il metodo `Graphis.FromImage()`. Ciò consente in pratica di disegnare su un'immagine come se si trattasse dell'area di un controllo.

In quanto tipo astratto non è possibile creare un oggetto di tipo `Image`. `Image` è di norma il tipo della variabile, alla quale viene assegnato un oggetto di tipo `Bitmap` o `Metafile`. Ciò consente di realizzare metodi generici, (i metodi `DrawImageXXX()`), i quali definiscono in parametro `Image` che richiede come argomento un oggetto `Metafile` o `Bitmap`.

Tipo «Bitmap»

Il tipo `Bitmap` consente la manipolazione di immagini di qualsiasi formato; dunque, eccetto che per file icona o immagini in formato *Windows Metafile* ("WMF" o "EMF"), è sempre `Bitmap` il tipo dell'oggetto che memorizza l'immagine.

Tra i metodi specificatamente definiti da `Bitmap` (e dunque non esposti da `Image` e `Metafile`) vi sono `MakeTransparent()`, `GetPixel()` e `SetPixel()`. Il primo consente di stabilire quale, tra tutti i colori utilizzati nella bitmap, dev'essere considerato trasparente. Tutti i pixel della bitmap che hanno il colore in questione non vengono disegnati, lasciando invariato lo sfondo che si trova sotto l'immagine.

`GetPixel()` accetta come argomenti le coordinate (X, Y) di un pixel e ne ritorna il colore. `SetPixel()` accetta come argomenti le coordinate (X, Y) di un pixel e un colore, modificando il pixel in questione in accordo al colore specificato.

Tipo «Metafile»

Un oggetto di tipo `Metafile` consente di manipolare immagini in formato vettoriale *Windows Metafile* (WMF e EMF), e cioè immagini definite non da un insieme di pixel ma da operazioni di disegno, geometrico e non.

Oltre che per memorizzare immagini caricate da file "WMF" e "EMF", un oggetto `Metafile` consente di produrre delle immagini, specificando le operazioni di disegno attraverso le quali devono essere costruite.

4.3 Riempire con un'immagine lo sfondo di un controllo

Il seguente esempio mostra come utilizzare un'immagine per riempire lo sfondo di un `Panel`. Il riempimento può avvenire nella forma «adattata» e «ripetuta». Nel primo caso, l'immagine viene visualizzata una sola volta e adattata alle dimensioni del controllo. Nel secondo caso l'immagine viene visualizzata nelle dimensioni originali tante volte quante sono necessarie per coprire tutta l'area cliente del controllo.

```
public partial class Form1: Form
{
    // . .
    Panel pnlArea;
    Button btnDisegna;
    RadioButton rbuAdatta, rbuRipeti;

    void DisegnaImgRipetuta(Graphics g, Image img, Size area)
    {
        Size sizeImg = img.Size;
        for(int y = 0; y < area.Height; y += sizeImg.Height)
            for (int x = 0; x < area.Width; x += sizeImg.Width)
                g.DrawImage(img, x, y);
    }

    void btnDisegna_Click(object sender, EventArgs e)
    {
        Bitmap bmp = new Bitmap("palla.bmp");
        Graphics g = pnlArea.CreateGraphics();
        g.Clear(pnlArea.BackColor);
        if (rbuAdatta.Checked)
            g.DrawImage(bmp, pnlArea.ClientRectangle);
        else
            DisegnaImgRipetuta(g, bmp, pnlArea.ClientSize);
    }
}
```

In base alla modalità di rendering scelta dall'utente, l'immagine viene adattata alla *client area* del pannello:

```
g.DrawImage(bmp, pnlArea.ClientRectangle);
```

oppure visualizzata ripetutamente senza alterarne le dimensioni originali. Ciò viene fatto dal metodo `DisegnaImgRipetuta()`, il quale richiede come argomenti l'oggetto `Graphics`, l'immagine e le dimensioni della *client area* del pannello. Internamente, il metodo utilizza due variabili, `x` e `y`, per memorizzare la posizione dell'immagine. Questa viene fatta variare in ragione delle sue dimensioni, in modo che tutta l'area del controllo venga "coperta".

Di seguito è mosrato l'output del programma:

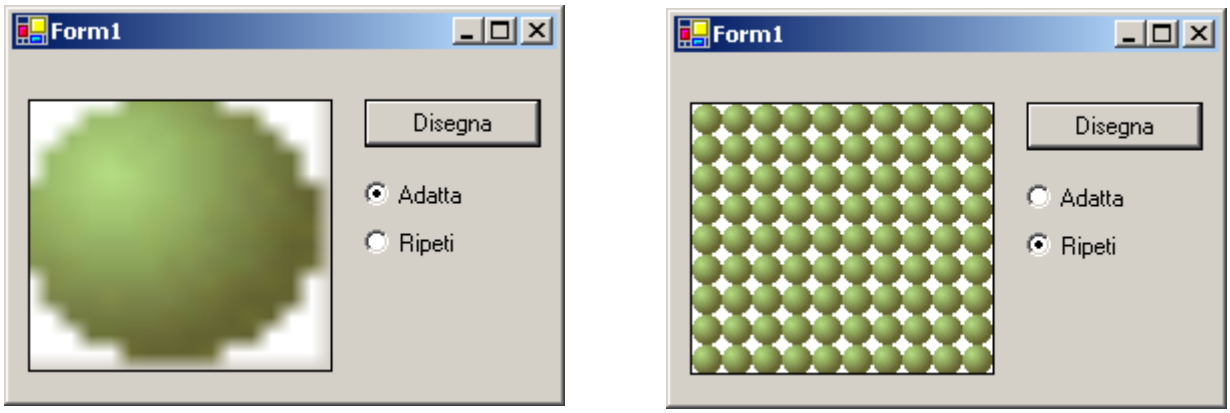


Figura 16-6 Output del programma

Rendere lo sfondo trasparente

L'immagine usata nell'esempio contiene uno sfondo bianco che viene ovviamente renderizzato insieme al resto. Ebbene, è possibile renderlo trasparente utilizzando il metodo `MakeTrasparent()`. Se è conosciuto, si può specificare direttamente il colore:

```
void btnDisegna_Click(object sender, EventArgs e)
{
    Bitmap bmp = new Bitmap("palla.bmp");
    bmp.MakeTransparent(Color.White);
    ...
}
```

Oppure, una volta stabilito di usare il colore di un determinato pixel dell'immagine come colore di sfondo, si può prima ottenere il colore in questione mediante il metodo `GetPixel()`:

```
void btnDisegna_Click(object sender, EventArgs e)
{
    Bitmap bmp = new Bitmap("palla.bmp");
    Color coloreSfondo = bmp.GetPixel(0, 0);    // pixel in alto a sinistra
    bmp.MakeTransparent(coloreSfondo);
    ...
}
```

Ecco l'output del programma modificato:

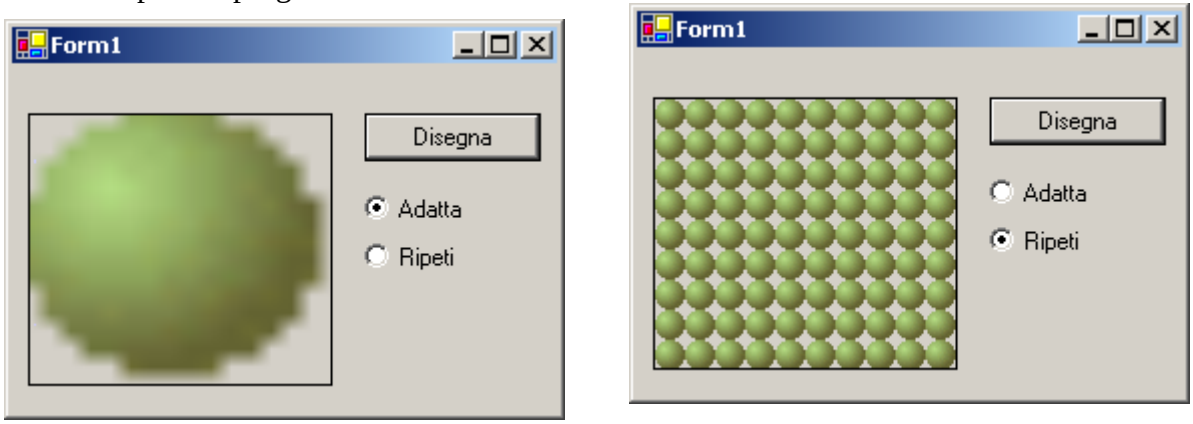


Figura 16-7 Output del programma

5 Effettuare il rendering del testo

Per il rendering del testo esiste il metodo `DrawString()`, caratterizzato da tre parametri di base:

- la stringa di testo da visualizzare;
- il font utilizzato;
- il pennello utilizzato per disegnare i caratteri;

A questi si aggiunge opzionalmente la possibilità di formattare il testo all'interno di un rettangolo, chiamato «rettangolo di layout».

Il metodo è fornito in sei versioni (riconducibili a tre), queste si differenziano per il modo in cui vengono specificate le coordinate e per la possibilità di caratterizzare l'allineamento e altri aspetti del testo visualizzato.

> Nota bene: in tutte le versioni, i parametri che rappresentano dimensioni e coordinate sono dichiarati come valori reali: `float`, `PointF`, `RectangleF`. Poiché esistono però delle conversioni implicite da `int` a `float`, da `Point` a `PointF` e da `Rectangle` a `RectangleF`, è perfettamente lecito specificare valori interi (`int`, `Point`, `Rectangle`), come abbiamo fatto finora.

5.1 Anatomia di un font

Il termine «font», che può essere tradotto in «tipo di carattere», designa un insieme di caratteristiche coerenti (applicate a tutti i caratteri) che influiscono sull'aspetto del testo visualizzato, tra le quali la forma e lo stile. La forma si riconduce alla «famiglia» del font (*Font family*) e fornisce l'impronta visuale ai caratteri, rendendoli esteticamente diversi dai caratteri visualizzati mediante un font appartenente a un'altra famiglia. Esistono ad esempio le famiglie Times Roman, Arial, Courier, eccetera.

Lo stile caratterizza un font all'interno di una famiglia, modificando l'aspetto dei caratteri ma sempre nell'ambito di una determina impronta visuale. Esistono tre stili fondamentali: «normale», «grassetto» (neretto), «corsivo» (italico), ai quali si aggiungono «sottolineato» e «barrato». Gli stili possono essere combinati per ottenere effetti compositi (grassetto-corsivo, ad esempio). In .NET, uno stile è rappresentato da un valore di tipo `FontStyle`:

Tabella 16-7 Stile Font: tipo enumeratore `FontStyle`

STILE	DESCRIZIONE
`FontStyle.Bold`	Grassetto.
`FontStyle.Italic`	Corsivo
`FontStyle.Regular`	Normale
`FontStyle.Strikeout`	Barrato.
`FontStyle.Underline`	Sottolineato.

Nell'ambito di un certo font, un testo è sempre caratterizzato da una certa dimensione, generalmente espressa in punti (un punto equivale a 1/72 di pollice). A parità di dimensione, testi visualizzati in font diversi occupano spazi diversi, sia in orizzontale che in verticale. Poiché la dimensione può

essere anche un valore frazionario (10,5 ad esempio), il parametro corrispondente dichiarato dal metodo `DrawString()` è di tipo `float`.

La classe `Font` espone numerose proprietà, ma a sola lettura. Ciò significa che occorre fornire tutte le caratteristiche desiderate in fase di creazione del font, poiché successivamente non è più possibile modificarle. A questo scopo il costruttore della classe è fornito in 13 versioni.

5.2 Visualizzare il testo specificando la sola posizione

L'esempio seguente mostra come visualizzare una stringa di testo utilizzando la forma più semplice del metodo `DrawString()`, che richiede la sola posizione; questa si riferisce all'angolo in altro a sinistra di un rettangolo virtuale che contiene la stringa da visualizzare.

```
public partial class Form1: Form
{
    // . . .
    Panel pnlTesto;
    Button btnScrivi;

    void btnScrivi_Click(object sender, EventArgs e)
    {
        Graphics g = pnlTesto.CreateGraphics();
        g.Clear(panTesto.BackColor);
        Font fontTimes = new Font("Times New Roman", 12);
        g.DrawString("Times New Roman: nomale; 12pt", fontTimes, Brushes.Yellow,
                     0, 0);

        Font fontTimes2 = new Font(fontTimes, FontStyle.Bold);
        g.DrawString("Times New Roman: grassetto; 12 pt", fontTimes2,
                     Brushes.Yellow, 0, 16);

        Font fontArial = new Font("Arial", 18, FontStyle.Italic,
                                  GraphicsUnit.Pixel);
        g.DrawString("Arial: corsivo; 18px", fontArial, Brushes.Blue,
                      new Point(0, 42));

        Font fontArial2 = new Font("Arial", 14,
                                   FontStyle.Bold | FontStyle.Underline);
        g.DrawString("Arial: grassetto/sottolineato; 14pt", fontArial2,
                      Brushes.Blue, new Point(0, 68));
    }
}
```

Di seguito è mostrato l'output del programma:

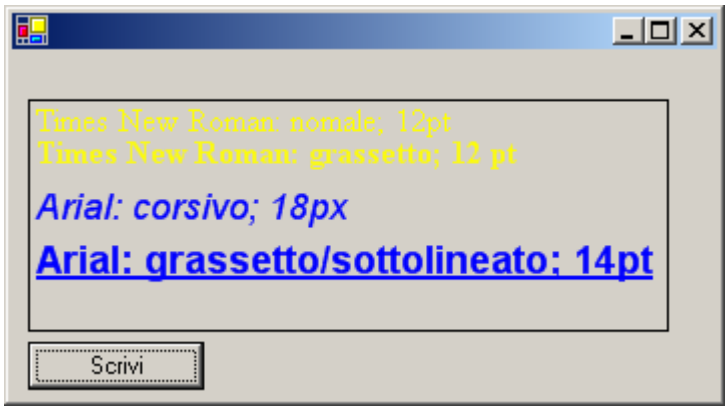


Figura 16-8 Output del programma

5.3 Visualizzare il testo in un rettangolo di layout

Il metodo `DrawString()` fornisce la possibilità di specificare un rettangolo – rettangolo di layout – all'interno del quale visualizzare il testo. Di norma si fornisce un rettangolo di layout per:

- limitare l'area di visualizzazione del testo, ottenendo il risultato di tagliare il testo che cade al di fuori di essa, oppure (in orizzontale) un'interruzione di riga automatica con ritorno a capo del testo che supera il margine destro del rettangolo;
- allineare il testo relativamente al rettangolo, in orizzontale e/o in verticale.

Anche se non espressamente richiesto, il rettangolo di layout viene di norma accompagnato da un oggetto di tipo `StringFormat`, il quale consente di intervenire sulla modalità di formattazione del testo, ad esempio per:

- abilitare o meno il ritorno a capo automatico;
- non considerare gli eventuali spazi finali di ogni riga;
- allineare il testo, orizzontalmente e/o verticalmente;
- abilitare o meno la ricerca di un font alternativo nel caso in cui il font specificato non sia installato nel computer, eccetera;

L'esempio che segue mostra come visualizzare del testo centrato sia orizzontalmente che verticalmente nell'area cliente di un pannello.

```
public partial class Form1
{
    // . . .
    Panel pnlTesto;
    Button btnScrivi;

    void btnScrivi_Click(object sender, EventArgs e)
    {
        Graphics g = pnlTesto.CreateGraphics();
        g.Clear(panTesto.BackColor);

        StringFormat sf = new StringFormat();
```

```
        sf.Alignment = StringAlignment.Center;     // allineamento orizzontale
        sf.LineAlignment = StringAlignment.Center; // allineamento verticale

        Rectangle layout = panTesto.ClientRectangle;

        Font font = new Font("Verdana", 12F, FontStyle.Bold | FontStyle.Italic);
        string testo = "La donzelletta vien dalla campagna in sul calar del
                        sole...";
        g.DrawString(testo, font, Brushes.Black, layout, sf);
    }
}
```

Di seguito è mostrato l'output del programma.

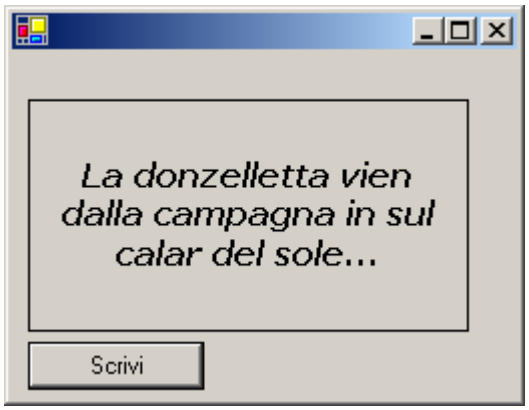


Figura 16-9 Output del programma

5.4 Misurare l'area occupata da un testo

Negli esempi proposti finora non ci siamo preoccupati di conoscere l'area occupata dal testo visualizzato, ma in molti casi questa informazione è fondamentale. Si pensi ad un programma di videoscrittura. Per determinare la posizione di ogni riga di testo da visualizzare è necessario conoscere l'altezza occupata da tutte le righe che la precedono; altezza che può essere la stessa per tutte (caso più semplice) oppure potenzialmente diversa per ognuna. Questo problema è tipico di ogni controllo che deve visualizzare un testo suddiviso in righe o paragrafi.

A questo scopo la classe `Graphics` espone il metodo `MeasureString()`, che mediante un oggetto di tipo `SizeF` ritorna l'occupazione orizzontale e verticale del testo specificato come primo argomento, sulla base del font specificato come secondo argomento.

Testo e font rappresentano i due argomenti di base, ma il metodo `MeasureString()` è fornito in sette versioni, che consentono di calcolare l'occupazione del testo in varie circostanze e modalità di formattazione, oltre che di ottenere informazioni accessorie, quali il numero di caratteri e di righe che compongono il testo.

5.5 Visualizzare un elenco di stringhe

Esistono svariate modalità di impiego del metodo `MeasureString()`, che dipendono dal tipo di rendering da effettuare, nella sostanza riconducibili a due:

- misurare l'occupazione di un testo campione relativamente a un determinato font e a una determinata dimensione. In questo caso si presuppone la necessità di una sola misura, che sarà utilizzata come campione per il rendering di tutto il testo;

- misurare ogni volta il testo visualizzato. Ciò presuppone che ogni stringa visualizzata possa occupare un'area diversa.

L'esempio che segue affronta uno scenario del secondo tipo. Esso visualizza un elenco di stringhe al quale è associato un elenco di bitmap. Le stringhe, di lunghezza diversa, vengono visualizzate a destra dell'immagine corrispondente all'interno di un rettangolo di layout la cui larghezza equivale alla larghezza dell'area cliente meno la larghezza dell'immagine. Poiché le stringhe non entrano nel rettangolo di layout, vengono spezzate in più righe, il numero delle quali dipende dalla lunghezza della stringa.

In conclusione, ogni stringa occupa un'area di altezza diversa; altezza che dev'essere calcolata per sapere in quale posizione visualizzare la stringa successiva. Nota bene: per evidenziare la differenza di altezza delle aree occupate dal testo, ogni immagine viene adattata a un rettangolo la cui altezza corrisponde a quella della stringa corrispondente.

```
public partial class Form1: Form
{
    // . . .
    Panel panTesto;
    Button btnScrivi;

    string[] testo =
    {
        "L'icona rappresenta una faccia felice",
        "L'icona rappresenta una faccia con espressione neutra",
        "L'icona rappresenta una faccia con espressione chiaramente corrucciata"
    };

    string[] nomiIcone =
    {
        "felice.bmp",
        "neutra.bmp",
        "corrucciata.bmp"
    };


    void btnScrivi_Click(object sender, EventArgs e)
    {
        Graphics g = panTesto.CreateGraphics();
        g.Clear(panTesto.BackColor);
        Font font = new Font("Verdana", 10F, FontStyle.Bold);
        StringFormat sf = new StringFormat();
        sf.Alignment = StringAlignment.Center;
        Size areaCliente = panTesto.ClientSize;
        int gap = 6;
        float altezza = 0;

        for (int i = 0; i < testo.Length && altezza < areaCliente.Height; i++)
        {
```

```
            Bitmap icona = new Bitmap(nomiIcone[i]);
            icona.MakeTransparent(Color.White);
            int maxLarghezzaTesto = areaCliente.Width - icona.Width;

            SizeF size = g.MeasureString(testo[i], font, maxLarghezzaTesto);

            RectangleF layoutIcona = new RectangleF(0, altezza, icona.Width,
                                                    size.Height);
            g.DrawImage(icona, layoutIcona);

            RectangleF layout = new RectangleF(icona.Width, altezza,
                                               maxLarghezzaTesto, size.Height);
            g.DrawString(testo[i], font, Brushes.Blue, layout, sf);

            altezza += size.Height + gap;
        }
    }
}
```

Di seguito è mostrato l'output del programma:

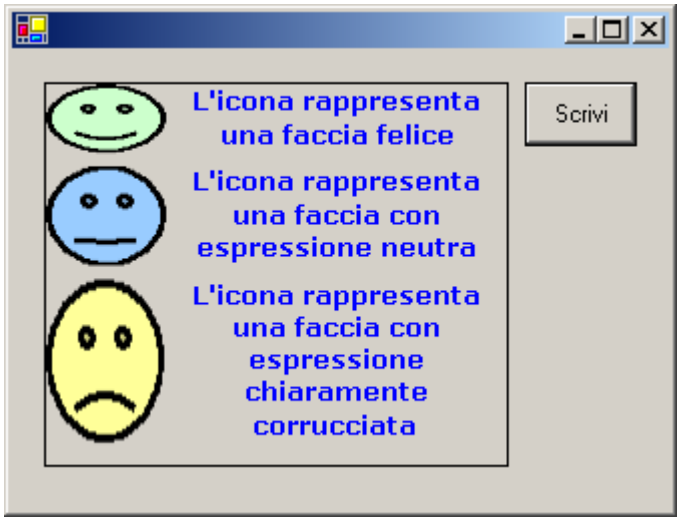


Figura 16-10 Output del programma

Nota bene: nel misurare il testo viene fornita la larghezza massima ammessa, equivalente a quella del rettangolo di layout usato per visualizzarlo. Se tale argomento fosse omesso, `MeasureString()` non considererebbe le interruzioni di riga necessarie per formattare il testo nello spazio disponibile e dunque ritornerebbe delle dimensioni incoerenti con il tipo di rendering che vogliamo ottenere.

Altra cosa degna di nota è il tipo, `float`, della variabili `altezza` e del rettangolo di layout. Ciò dipende dal fatto che l'oggetto ritornato da `MeasureString()` è di tipo `SizeF` e dunque tutte le variabili che dipendono da esso sono di tipo `float`.

6 Rilasciare le risorse grafiche

Tutti gli oggetti di GDI+, `Graphics`, `Bitmap`, `Pen`, eccetera, impegnano delle risorse grafiche del sistema operativo. Tali risorse non sono infinite e dunque dovrebbero essere rilasciate appena possibile. Quando un oggetto GDI+ non è più necessario si divrebbe dunque invocare il metodo `Dispose()`, il quale rilascia le risorse grafiche impegnate dall'oggetto.

Ecco un frammento di codice ripreso da un esempio precedente. Le ultime due istruzioni invocano il metodo Dispose() per gli oggetti di tipo `Graphics` e `Bitmap`:

```
void btnDisegna_Click(object sender, EventArgs e)
{
    Bitmap bmp = new Bitmap("palla.bmp");
    Graphics g = pnlArea.CreateGraphics();
    g.Clear(pnlArea.BackColor);
    if (rbuAdatta.Checked)
        g.DrawImage(bmp, pnlArea.ClientRectangle);
    else
        DisegnaImgRipetuta(g, bmp, pnlArea.ClientSize);
    g.Dispose();
    bmp.Dispose();
}
```

E' importante comprendere che l'invocazione del metodo `Dispose()` non riguarda il corretto funzionamento del programma (questo funzionerebbe comunque) ma l'uso efficiente delle risorse grafiche disponibili, aspetto che non dovrebbe mai essere sottovalutato in applicazioni realistiche. Nei esempi presentati abbiamo omesso le chiamate a `Dispose()` per alleggerire il codice e focalizzare l'attenzione sulle caratteristiche principali delle classi utilizzate.

Gestire il rendering dei controlli

1 Rendering di un controllo e richiesta di «*painting*»

La fase di rendering di un controllo consiste nella riproduzione delle sue caratteristiche visuali sullo schermo e rappresenta la risposta a una richiesta di *painting* generata da Windows, da altri controlli, oppure dal controllo stesso.

Una richiesta di *painting* può generarsi in svariate circostante:

- un'altra finestra si sovrappone temporaneamente al form, nascondendo il controllo, il quale necessita dunque di essere ridisegnato;
- il controllo viene ridimensionato;
- il controllo è reso visibile dopo che era stato precedentemente nascosto;
- viene modificato lo stato del controllo, ad esempio un suo attributo visuale, come il colore di sfondo o lo stile del bordo,
- viene modificata la sorgente dei dati che popola il controllo, ad esempio viene assegnato un nuovo valore alla proprietà `DataSource` di un `ListBox` o un `ComboBox`.

In questi casi viene prodotta una sequenza di eventi e azioni che si traduce nella generazione di una richiesta di *painting* al controllo in questione. Questo risponde eseguendo il proprio codice di rendering, il quale varia da un tipo di controllo all'altro e può a sua volta generare altri eventi e coinvolgere altri controlli.

1.1 Intervenire sul rendering di un controllo

La sequenza di eventi sopra menzionata è predeterminata e non richiede alcun intervento del programmatore; un controllo "sa" come disegnarsi sullo schermo. D'altra parte, molti tipi di controlli danno al programmatore la possibilità di intervenire su questo processo, consentendogli di scrivere del codice di rendering che integri o sostituisca quello predefinito.

Infatti, alla richiesta di *painting* essi sollevano un evento al quale il programmatore può attaccare un gestore esattamente come accade nella gestione di qualsiasi evento. D'altra parte esiste anche la possibilità di definire un controllo ex-novo, scrivendo per intero il proprio codice di rendering, che sarà eseguito in risposta alla richiesta di *painting*.

Nel proseguo del capitolo prenderemo in esame entrambi gli scenari.

2 Gestire l'evento «Paint» di un controllo

Molti tipi di controlli espongono un evento, `Paint`, che viene sollevato in risposta a una richiesta di *painting*. E' dunque possibile intervenire sul rendering di un controllo attaccando un metodo all'evento `Paint` dello stesso.

Ecco ciò avviene: la richiesta di *painting*, comunque essa sia generata, si traduce nella creazione di un oggetto di tipo `PaintEventArgs` e nell'invocazione del metodo virtuale `OnPaint()`, al quale viene passato come argomento l'oggetto in questione. Il metodo `OnPaint()` verifica se all'evento `Paint` del controllo è attaccato un gestore di evento; in caso affermativo questo viene eseguito e riceve come argomento l'oggetto `PaintEventArgs`. Questo contiene un riferimento all'oggetto `Graphics` necessario per disegnare sull'area occupata dal controllo.

Lo schema sottostante riassume la situazione:

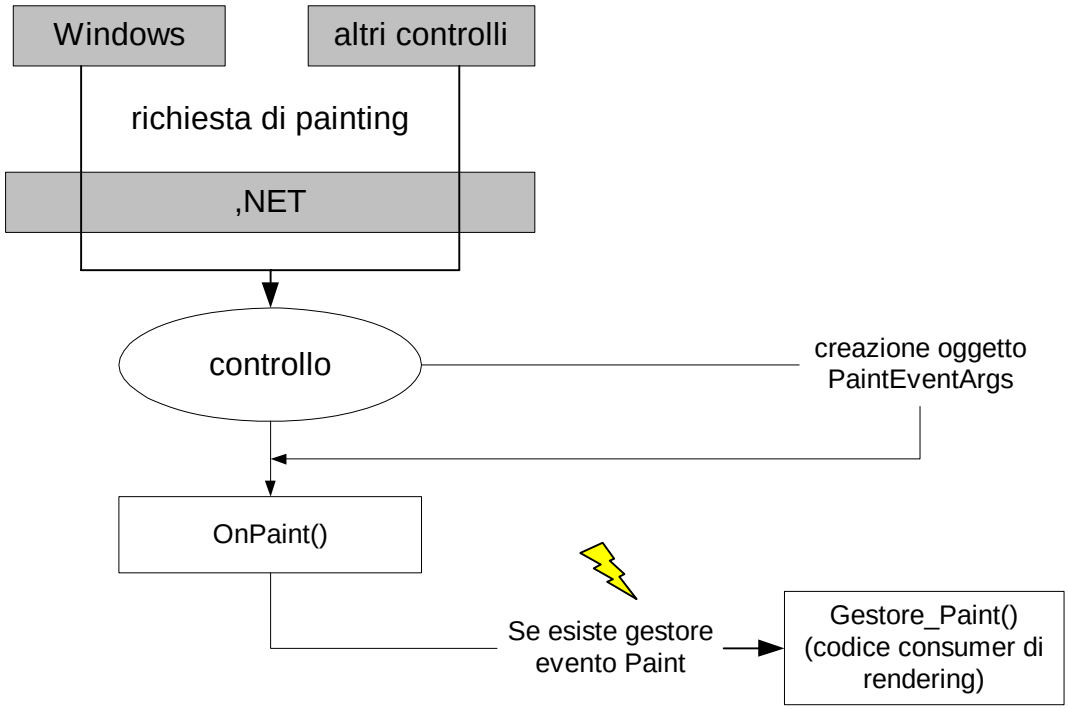


Figura 17-1 Schema di risposta ad una richiesta di *painting*

Il tutto avviene automaticamente; ciò che deve fare il programmatore è semplicemente attaccare il gestore all'evento e utilizzare l'oggetto `Graphics` per disegnare sull'area cliente del controllo.

2.1 Esempio di gestione dell'evento «Paint»

L'esempio seguente mostra come gestire l'evento `Paint` sollevato da un `Panel` in modo che esso visualizzi una tra le bitmap memorizzate in un vettore, il quale rappresenta un ipotetico elenco delle foto degli utenti che possono utilizzare il programma. La variabile `idUtente` funge da indice del vettore e quindi è utilizzata per accedere all'immagine da visualizzare.

```
public partial class Form1: Form
{
    // . . .
    Panel pnlUtente;
    public Form1()
    {
        // codice necessario per costruire il form
        pnlUtente.Paint += new PaintEventHandler(this.panUtente_Paint);
        fotoUtenti[0] = new Bitmap("felice.bmp");
        fotoUtenti[1] = new Bitmap("neutra.bmp");
        fotoUtenti[2] = new Bitmap("corrucciata.bmp");
```

```
        idUtente = 0;
    }

    Bitmap[] fotoUtenti = new Bitmap[3];
    int idUtente;

    void panUtente_Paint(object sender, PaintEventArgs e)
    {
        Graphics g = e.Graphics;
        Bitmap foto = fotoUtenti[idUtente];
        g.DrawImage(foto, pnlUtente.ClientRectangle);
    }
}
```

Esaminando il codice si vede che non esiste alcun meccanismo esplicito che provochi l'esecuzione del metodo `panTesto_Paint()`. Questo viene invocato automaticamente all'avvio del programma, quando il form principale, e tutti i controlli che questo contiene, riceve la richiesta di *painting*.

Un secondo aspetto riguarda l'oggetto `Graphics`. Questo non viene creato esplicitamente mediante il metodo `CreateGraphics()`, ma ottenuto dal parametro informazioni evento `e`. In altre parole, l'oggetto `Graphics` esiste già, ed è associato all'area cliente del controllo.

Infine la cosa più importante, che può essere verificata eseguendo delle azioni che sporchino in tutto o in parte il pannello: l'immagine è persistente, poiché viene ridisegnata ogni volta che l'area occupata dal pannello necessita di essere rinfrescata.

2.2 «Invalidare» l'area occupata da un controllo

Al di là della sua natura dimostrativa, il programma precedente è comunque incompleto. Per comprenderlo basta modificare il valore della variabile `idUtente`, ad esempio in risposta al clic su un bottone (aggiungiamo anche un'etichetta per riflettere visivamente questo cambiamento):

```
void btnCambiaUtente_Click(object sender, EventArgs e)
{
    if (idUtente == 2)
        idUtente = 0;
    else
        idUtente++;
    lblUtente.Text = idUtente.ToString();
}
```

L'identificatore utente cambia, ma l'immagine visualizzata no. Il perché è ovvio: non esiste alcun meccanismo automatico che al variare della variabile `idUtente` invii una richiesta di *painting* al pannello. Ciò dev'essere fatto mediante l'invocazione del metodo `Invalidate()`:

```
void btnCambiaUtente_Click(object sender, EventArgs e)
{
    if (idUtente == 2)
        idUtente = 0;
    else
        idUtente++;
    lblUtente.Text = idUtente.ToString();
```

```
    pnlUtente.Invalidate();
}
```

Il metodo `Invalidate()`, come suggerisce il nome, rende “non valida” l’area occupata dal controllo, informandolo che necessita di essere ridisegnato. `Invalidate()` invia dunque una richiesta di *painting* al controllo, il quale risponde come di consueto eseguendo il proprio codice di rendering e sollevando l’evento `Paint`.

Di seguito è mostrato l’output del programma:

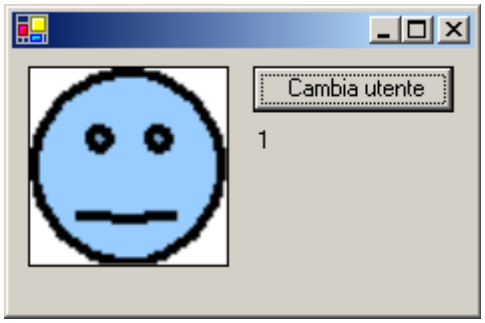


Figura 17-2 Output del programma

2.3 Ottimizzare il rendering di un controllo

Ci limitiamo a introdurre un aspetto che dovrebbe essere tenuto in debito conto nelle applicazioni realistiche, quello dell’ottimizzazione del codice di rendering. Benché le operazioni di disegno su video siano senz’altro più veloci rispetto ad altre eseguite normalmente da una applicazione (accedere al disco e generare pagine di stampa, ad esempio), occorre essere consapevoli che un controllo può ricevere molte richieste di *painting* in un solo secondo, e che rispondere ad esse può diventare oneroso in termini computazionali. Ciò è tanto più vero quanto è più complesso l’aspetto visuale del controllo.

In molti casi, d’altronde, soltanto una parte dell’area occupata dal controllo viene sporcata, ad esempio quando una finestra si sovrappone parzialmente al form dell’applicazione; in questo caso è possibile velocizzare l’operazione di rendering eseguendo soltanto il codice che visualizza la parte che necessita di essere rinfrescata.

A questo scopo, il parametro informazioni evento associato a una richiesta di *painting* espone la proprietà `ClipRectangle`; essa contiene coordinate e dimensioni dell’area da rinfrescare, che può essere anche molto minore dell’area totale del controllo. In alcune situazioni questa informazione può essere sfruttata per ottimizzare la fase di rendering, facendo sì che venga eseguito soltanto il codice strettamente necessario, invece di ridisegnare completamente il controllo.

Collegato a questo aspetto c’è l’uso del metodo `Invalidate()`, introdotto nel paragrafo precedente Esso è infatti disponibile in più versioni, alcune delle quali consentono di indicare una specifica area da invalidare, invece che l’area totale del controllo, come fa la versione senza parametri. Ed è questa l’area che viene memorizzata nella proprietà `ClipRectangle` del parametro informazioni evento e che può essere utilizzare per velocizzare il *rendering*.

Poiché l’ottimizzazione del codice di *rendering* dipende dalla natura e dall’aspetto visuale del controllo, e dunque varia da controllo a controllo, e poiché richiede spesso un livello di programmazione sofisticato, nel resto del capitolo non prenderemo in considerazione questo aspetto.

3 Gestire l'evento «DrawItem» dei controlli «ListBox» e «ComboBox»

Ad una richiesta di *painting*, non tutti i tipi controlli rispondono secondo lo schema presentato in Figura 11-1. Alcuni – `TextBox` ad esempio – non rispondono affatto con l'esecuzione del metodo `OnPaint()` e dunque neanche sollevano l'evento `Paint` o un evento equivalente al quale il programmatore possa attaccare un proprio gestore. Nel loro caso non è dunque possibile intervenire sulla fase di rendering.

Altri tipi di controlli, invece, pur non sollevando l'evento `Paint`, consentono comunque al programmatore di attaccare il proprio codice di rendering. E' questo il caso dei controlli `ListBox` e `ComboBox`, `CheckedListBox`, `StatusBar`, `TabControl`, e altri. Di seguito prenderemo in considerazione i primi due.

Entrambi, in risposta a una richiesta di *painting* relativa a una certa area del controllo, eseguono il metodo `OnDrawItem()` per ognuno degli elementi che cadono nell'area. Tale metodo solleva l'evento `DrawItem` corrispondente e dunque esegue un eventuale gestore ad esso attaccato.

Occorre precisare, comunque, che perché `OnDrawItem()` sia eseguito e l'evento `DrawItem` sollevato è necessario che la proprietà `DrawMode` del controllo sia impostata al valore `OwnerDrawFixed` oppure al valore `OwnerDrawVariable`. Ciò informa il controllo che sarà il codice *consumer* a prendersi la responsabilità di visualizzarne gli elementi.

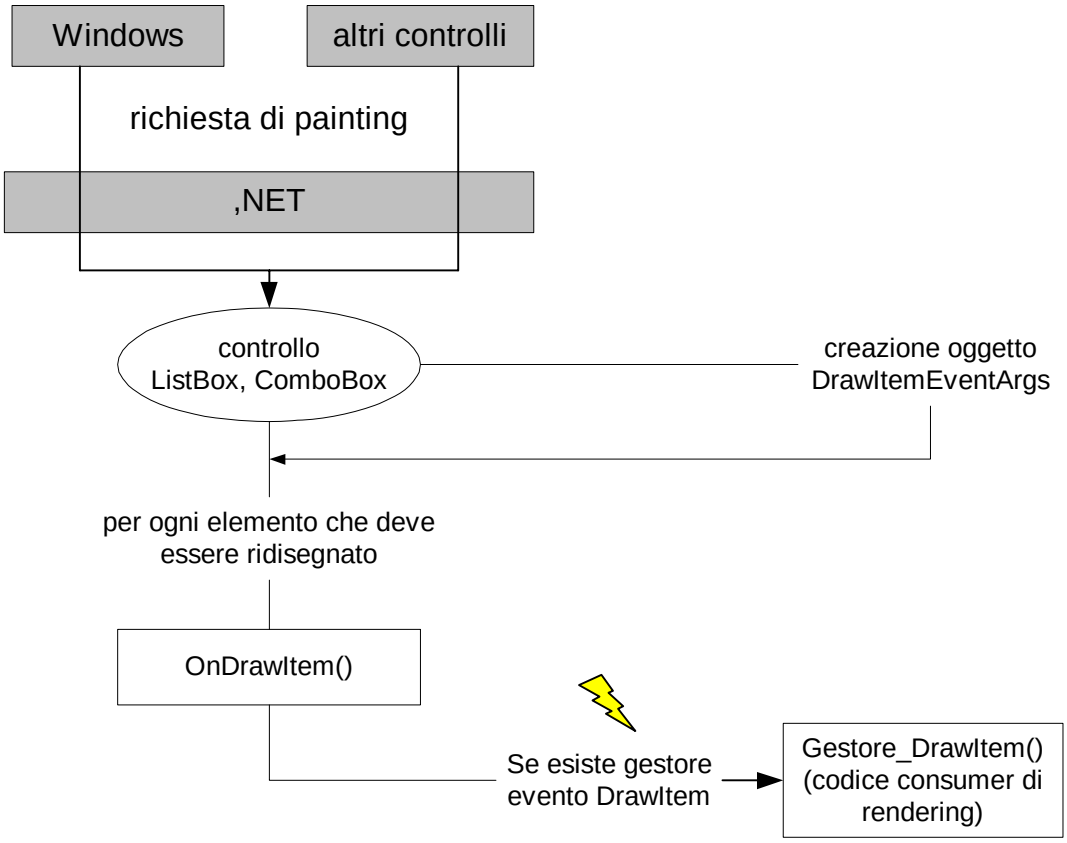


Figura 17-3 Schema di risposta ad una richiesta di *painting* dei controlli `ListBox` e `ComboBox`

All'evento `DrawItem`, prodotto per ogni elemento, è associato un parametro informazioni evento di tipo `DrawItemEventArgs`, il quale contiene le informazioni necessarie per disegnare l'elemento.

3.1 Tipo «DrawItemEventArgs»

`DrawItemEventArgs` è più complesso del tipo `PaintEventArgs` associato all'evento `Paint` e dunque necessita di qualche spiegazione. La tabella mostrata di seguito elenca le proprietà dell'oggetto, che contengono le informazioni necessarie per disegnare un elemento. Da precisare che nulla impedisce al codice *consumer* di ignorare alcune o addirittura tutte le proprietà elencate, benché di norma non sia consigliabile farlo. Quelle essenziali sono `Index` e `Bounds`, oltre ovviamente alla proprietà `Graphics`. Il parametro informazioni evento espone inoltre due metodi utili, anche se non essenziali, per il disegno dell'elemento: `DrawBackGround()` e `DrawFocusRectangle()`. In seguito considereremo soltanto il primo, il quale disegna lo sfondo appropriato in base allo stato dell'elemento.

Tabella 17-1 proprietà del tipo `DrawItemEventArgs`

PROPRIETÀ - TIPO	DESCRIZIONE
`BackColor` e **`ForeColor`** `Color`	Colori di sfondo e di primo piano dell'elemento da disegnare. Vengono impostati automaticamente in base allo stato dell'elemento (vedi proprietà `State`)
`Bounds` `Rectangle`	Rettangolo che rappresenta i limiti dell'elemento.
`Font` `Font`	Tipo di carattere assegnato all'elemento (di solito equivalente alla proprietà `Font` del controllo).
`Graphics` `Graphics`	Oggetto `Graphics` sul quale disegnare l'elemento. L'oggetto fa riferimento all'intera area del controllo e quindi è necessario utilizzare la proprietà `Bounds` per disegnarlo alle giuste coordinate.
`Index` `int`	Indice dell'elemento.
`State` `DrawItemState`	Stato dell'elemento. Il tipo enumeratore `DrawItemState` definisce svariate costanti, utilizzate anche da altri tipi di controlli; in questa sede ci interessa la sola costante `Selected`: essa indica che l'elemento è selezionato.

3.2 Esempio di gestione dell'evento «DrawItem»

Segue un esempio che mostra come gestire il rendering di un `ListBox`, visualizzando un'immagine a sinistra del testo rappresentato da ogni elemento. Il `ListBox` memorizza l'elenco delle nazioni dove si svolgono (o si sono svolti) i gran premi di Formula 1. Una variabile intera, `gpDisputati`, memorizza il numero dei gran premi già disputati. Accanto ad essi, nel `ListBox` sarà visualizzata un'immagine che rappresenta la bandiera a scacchi. Per quei gran premi il cui indice è superiore a `gpDisputati`, invece, viene visualizzata un'immagine che rappresenta la bandiera verde.

Nel costruttore del form principale, oltre a caricare in memoria le due bitmap che rappresentano le bandiere, vengono impostate opportunamente le proprietà `ItemHeight` e `DrawMode` del `ListBox`. Infine, mediante il bottone `btnNuovoGP` è possibile incrementare il valore di `gpDisputati`, simulando la conclusione di un nuovo gran premio.

```
public partial class Form1: Form
{
    // . . .
    ListBox lboGP;
```

```
    Button btnNuovoGP;

    public Form1()
    {
        // codice necessario per costruire il form
        bmpSvolto = new Bitmap("scacchi.bmp");
        bmpDaCorrere = new Bitmap("verde.bmp");
        lboGP.DrawItem += new DrawItemEventHandler(lboGP_DrawItem);
        lboGP.DrawMode = DrawMode.OwnerDrawFixed;
        lboGP.ItemHeight = bmpSvolto.Height + 2;
        lboGP.DataSource = granPremi;
    }

    string[] granPremi =
    {
        "Australia",
        "Malesia",
        "Brasile",
        "Spagna"
    };

    int gpDisputati;
    Bitmap bmpSvolto;
    Bitmap bmpDaCorrere;

void lboGP_DrawItem(object sender, DrawItemEventArgs e)
{
        Image bandiera;
        Graphics g = e.Graphics;
        e.DrawBackground();
        if (e.Index < gpDisputati)
            bandiera = bmpSvolto;
        else
            bandiera = bmpDaCorrere;
        g.DrawImage(bandiera, e.Bounds.Left, e.Bounds.Top);

        int left = e.Bounds.Left + bandiera.Width;
        Color c = e.ForeColor;
        if ((e.State & DrawItemState.Selected) != 0)
            c = Color.Red;
        string item = lboGP.Items[e.Index].ToString();
        g.DrawString(item, e.Font, new SolidBrush(c), left, e.Bounds.Top);
    }
```

```
    void btnNuovoGp_Click(object sender, EventArgs e)
    {
        if (gpDisputati < granPremi.Length)
            gpDisputati++;
        lboGP.Invalidate();
    }
}
```

Di seguito è mostrato l'output del programma dopo che sono stati svolti due gran premi:

Figura 17-4 Output del programma

Commentiamo il codice di rendering. Mediante il metodo `DrawBackGround()` viene innanzitutto disegnato lo sfondo dell'elemento, il quale è influenzato dalle proprietà `BackColor` e `BackgroundImage`. Quindi viene verificata quale immagine visualizzare, confrontando l'indice dell'elemento (`e.Index`) con la variabile `gpDisputati`. Se il primo è minore della seconda, il gran premio è stato già disputato e dunque è necessario visualizzare la bandiera a scacchi, memorizzata nella bitmap `bmpSvolto`, altrimenti sarà visualizzata la bandiera verde. Da notare che viene utilizzata la proprietà `Bounds` per specificare la posizione dell'immagine:

```
g.DrawImage(bandiera, e.Bounds.Left, e.Bounds.Top);
```

Successivamente viene calcolata la posizione orizzontale della stringa contenente il nome della nazione. Quindi viene stabilito il colore da utilizzare per visualizzarla:

```
Color c = e.ForeColor;
if ((e.State & DrawItemState.Selected) != 0)
    c = Color.Red;
```

La `if()` è stata aggiunta per mostrare l'uso della proprietà `State`, la quale consente di ignorare la gestione predefinita del colore associato all'elemento selezionato, (e cioè `Color.HighLightText`), per assegnare un colore personalizzato, in questo caso il rosso.

Infine viene ottenuta la stringa e successivamente visualizzata.

4 Controlli realizzati dal programmatore (*User's controls*)

Personalizzare il rendering di un controllo, attaccando il proprio codice all'evento di disegno – `Paint` o `DrawItem` – che questo genera in risposta a una richiesta di *painting* va bene come

soluzione ad hoc, da adottare in una determinata applicazione. Riconsideriamo l'ultimo esempio e ipotizziamo di avere svariate applicazioni che richiedono un tipo di `ListBox` in grado di visualizzare un'immagine adiacente al testo. Nulla ci impedisce di impiegare ogni volta un `ListBox` standard e attaccare all'evento `DrawItem` un codice di rendering personalizzato, ma probabilmente la cosa migliore sarebbe avere a disposizione un tipo di `ListBox` che faccia tutto da sé, magari con alcune proprietà aggiuntive che ne estendano le funzionalità. Ciò può essere fatto scrivendo una nuova classe che derivi da `ListBox` e che aggiunga al tipo base il codice di rendering e le funzionalità desiderate.

La realizzazione di un nuovo tipo di controllo può rispondere a diverse esigenze:

- estendere le funzionalità di un controllo esistente;
- creare nuovi tipi di oggetti visuali, attualmente non disponibili, e renderli fruibili da tutte le applicazioni;
- semplificare il codice dell'applicazione, demandando ai soli controlli la responsabilità di gestione dell'interfaccia utente;

In base a queste il programmatore può:

- derivare il nuovo tipo da un tipo esistente, – `ListBox`, `Label`, `Panel`, o un controllo scritto da terze parti – il quale incapsula la maggior parte delle funzionalità desiderate, limitandosi a modificare o estendere soltanto alcune di esse;
- creare il controllo ex novo, derivandolo dal tipo base `Control` e fornendolo di tutte le caratteristiche richieste;

.NET fornisce una classe particolare, `UserControl`, appositamente realizzata per fungere da tipo base per la realizzazione di nuovi controlli. Questi non sono da intendersi come controlli d'uso generale, ma nascono soprattutto per condividere determinate funzionalità nell'ambito di un insieme di applicazioni e/o di un'organizzazione che produce software applicativo.

4.1 Requisiti di base di un controllo

Realizzare un nuovo tipo di controllo non implica necessariamente o soltanto la scrittura del codice di rendering, anche se è soprattutto questo l'aspetto che prenderemo in considerazione; vi sono delle scelte progettuali da fare e determinati requisiti da rispettare. Tra questi, ne esistono tre molto importanti:

- le funzioni del controllo non devono accedere a variabili definite nel codice *consumer*. Ciò vale quindi anche per il codice di rendering, il quale deve riflettere lo stato del controllo senza dipendere da uno o più oggetti esterni;
- il controllo deve prevedere uno stato e un aspetto iniziali predefiniti. Ne consegue che il controllo deve definire un costruttore di default;
- ogni cambiamento di stato che implica un cambiamento nell'aspetto del controllo deve provocare l'esecuzione del codice di rendering.

Sull'ultimo punto è opportuna una estensione. Un codice ben progettato dovrebbe sempre garantire la coerenza tra lo stato del controllo e il suo aspetto visuale. Laddove questo non fosse possibile, il codice dovrebbe o rifiutare il cambiamento di stato o sollevare un'eccezione e dunque evitare che il controllo mostri un aspetto incoerente con il proprio stato interno.

4.2 Collocazione del codice di rendering scritto dal programmatore

Un aspetto fondamentale nella realizzazione di un nuovo tipo di controllo (in generale di una nuova classe) è la ridefinizione delle funzioni virtuali, operazione che consente di modificare il comportamento fornito dal tipo base. Ciò vale anche per il codice di rendering.

Per fornire un rendering personalizzato occorre dunque ridefinire il metodo di disegno che risponde alle richieste di *painting*. Questo è normalmente il metodo `OnPaint()`, oppure il metodo `OnDrawItem()` per i controlli `ListBox`, `ComboBox`, eccetera. Entrambi sono dichiarati virtuali appunto per poter essere ridefiniti nelle classi derivate.

Nei paragrafi successivi mostreremo due esempi di realizzazione di un nuovo tipo di controllo. Il primo riprende l’esempio precedente e mostra come estendere il tipo `ListBox` in modo che possa visualizzare delle immagini adiacenti al testo. Il secondo esempio affronta nuovamente il problema della visualizzazione di un grafico in coordinate reali, presentato nel capitolo precedente. Allo scopo viene realizzato il controllo `Grafico`, che incorpora il codice di conversione e di rendering necessario.

5 Controllo «GPListBox»

Il progetto rappresenta una rivisitazione di quello precedente, nel quale un `ListBox` era utilizzato per mostrare l’elenco dei gran premi del mondiale di Formula 1. La classe prodotta, `GPListBox`, così com’è realizzata non presenta caratteristiche d’impiego generali, poiché è adattata sulla base di un problema specifico, ma rappresenta comunque un utile esempio di progettazione.

5.1 Caratteristiche del controllo «GPListBox»

Il controllo `GPListBox` estende il funzionamento del tipo base `ListBox` offrendo in più:

- la proprietà `GpDisputati`, di tipo `int`, attraverso la quale è possibile aggiornare il numero di gran premi disputati;
- le proprietà `BandieraScacchi` e `BandieraVerde`, di tipo `Image`, attraverso le quali è possibile definire le immagini visualizzate accanto al testo;

`GPListBox` ridefinisce inoltre il metodo `OnDrawItem()`, all’interno del quale è collocato il codice di *rendering*.

5.2 Scheletro della classe «GPListBox»

Segue lo scheletro della classe, che implementa le dichiarazioni delle variabili, il costruttore e le proprietà sopra elencate. Del metodo `OnDrawItem()` viene fornito il solo prototipo:

```
public class GPListBox: ListBox
{
    int gpDisputati;
    Image bandieraScacchi;
    Image bandieraVerde;

    public GPListBox()
    {
        DrawMode = DrawMode.OwnerDrawFixed;
```

```
    }

    public int GpDisputati
    {
        get {return gpDisputati;}
        set
        {
            if (value == gpDisputati)
                return;
            gpDisputati = value;
            Invalidate();
        }
    }

    public Image BandieraScacchi
    {
        get {return bandieraScacchi;}
        set
        {
            if (value == bandieraScacchi)
                return;
            bandieraScacchi = value;
            Invalidate();
        }
    }

    public Image BandieraVerde
    {
        get {return bandieraVerde;}
        set
        {
            if (value == bandieraVerde)
                return;
            bandieraVerde = value;
            Invalidate();
        }
    }

    protected override void OnDrawItem(DrawItemEventArgs e) { ... }
}
```

Degna di nota è l’implementazione delle proprietà. Ne consideriamo una, poiché seguono tutte le stesso modello:

```
public Image BandieraScacchi
{
```

```
    get {return bandieraScacchi;}
    set
    {
        if (value == bandieraScacchi)
            return;
        bandieraScacchi = value;
        Invalidate();
    }
}
```

Come si vede, dopo l’assegnazione al campo `bandieraScacchi` viene invocato il metodo `Invalidate()`, che forza il rendering del controllo. Ciò vale per tutte le proprietà che presuppongo una modifica dello stato che a sua volta influenza l’aspetto del controllo. L’istruzione `if()` non rappresenta invece un requisito, ma rientra in una prassi consolidata. Infatti, poiché l’assegnazione di un valore alla proprietà provoca il ridisegno del controllo (operazione costosa in termini computazionali), è opportuno prima verificare che il nuovo valore sia effettivamente diverso dal valore attuale.

Infine un’ultima nota, che riguarda il costruttore. Poiché i valori predefiniti dei campi della classe sono in questo caso accettabili, l’unica istruzione contenuta nel costruttore serve a impostare la modalità di disegno del controllo a `DrawMode.OwnerDrawFixed`; in caso contrario il metodo `OnDrawItem()` non sarebbe eseguito, e sarebbe dunque ignorato il nuovo codice di rendering.

5.3 Codice di rendering della classe «GPListBox»: metodo «OnDrawItem»

Il codice di rendering ricalca quello dell’esempio precedente, con qualche modifica:

```
protected override void OnDrawItem(DrawItemEventArgs e)
{
    Image bandiera;
    Graphics g = e.Graphics;
    e.DrawBackground();

    if (e.Index < gpDisputati)
        bandiera = bandieraScacchi;
    else
        bandiera = bandieraVerde;

    int left = 0;
    if (bandiera != null)
    {
        g.DrawImage(bandiera, e.Bounds.Left, e.Bounds.Top);
        Rectangle r = new Rectangle(e.Bounds.Left, e.Bounds.Top, bandiera.Width
                                    e.Bounds.Height-2);
        g.DrawImage(bandiera, r);
        left = e.Bounds.Left + bandiera.Width;
    }

    Color c = e.ForeColor;
```

```
    string item = Items[e.Index].ToString();
    g.DrawString(item, e.Font, new SolidBrush(c), left, e.Bounds.Top);
    base.OnDrawItem(e);
}
```

L'ultima istruzione:

```
base.OnDrawItem(e);
```

rappresenta una novità. Essa invoca il metodo sovrascritto `OnDrawItem()`, definito nella classe base. E' norma che un metodo ridefinito invochi il metodo corrispondente della classe base, in modo da estendere il funzionamento predefinito della classe e non sostituirlo. In questo caso, omettere l'invocazione del metodo base non pregiudicherebbe il funzionamento del controllo, ma renderebbe impossibile attaccare un eventuale codice *consumer* di rendering, poiché è proprio il metodo `OnDrawItem()` definito da `ListBox` a verificare se esiste un gestore attaccato all'evento `DrawItem`.

> E' la documentazione della classe, fornita insieme a .NET SDK, a suggerire se un metodo base debba essere invocato o meno, e quali sono le conseguenze nel caso ciò non avvenga.

Le altre novità sostanziali di `OnDrawItem()` rispetto al codice di *rendering* dell'esempio precedente sono:

- viene verificata l'effettiva esistenza di un oggetto immagine associato alle proprietà `BandieraVerde` e `BandieraScacchi`; in caso contrario, viene visualizzato il solo testo;
- l'immagine viene dimensionata relativamente a un rettangolo di layout, la cui altezza è uguale all'altezza memorizzata in `ItemHeight – 2`;
- la gestione del colore associato al testo è quella standard.

5.4 Applicazione di esempio

Come applicazione di test per la classe `GPListBox` va benissimo quella utilizzata nell'esempio precedente. Sono da modificare soltanto la dichiarazione della variabile `lboGP`, il costruttore del form principale e il gestore dell'evento `Click` del bottone:

```
public partial class Form1: Form
{
    GPListBox lboGP;
    Button btnNuovoGP;

    public Form1()
    {
        // codice necessario per costruire il form
        lboGP.DataSource = granPremi;
        lboGP.BandieraScacchi = new Bitmap("svolto.bmp");
        lboGP.BandieraVerde = new Bitmap("dacorrere.bmp");
        lboGP.ItemHeight = lboGP.BandieraScacchi.Height + 2;
    }
    void btnNuovoGP_Click(object sender, EventArgs e)
    {
```

```
        if (lboGP.GpDisputati < lboGP.Items.Count)
            lboGP.GpDisputati++;
    }
}
```

6 Controllo «Grafico»

Nel capitolo precedente è stata realizzata una classe – `Grafico` – che nella forma può essere considerata un tipo di controllo, poiché deriva dalla classe `Panel`, ma che nella sostanza non lo è affatto. Infatti, così come è stato definita, `Grafico` non rispetta nessuno dei requisiti fondamentali di un controllo, alla base dei quali c'è la persistenza su video e cioè la capacità di rispondere alle richieste di *painting*. Alla luce di quanto appreso nel paragrafo precedente, per trasformare la classe `Grafico` in un vero e proprio controllo occorrono delle modifiche nella sua architettura. E cioè:

- un oggetto `Grafico` deve fornire uno stato iniziale predefinito. La classe deve dunque fornire un costruttore di default. Ciò significa inoltre che il rendering dell'oggetto dev'essere possibile anche in assenza di dati;
- la fase di rendering, eseguita in risposta alle richieste di *painting*, dev'essere separata dalla fase di acquisizione e conversione delle coordinate reali;
- a un oggetto di tipo `Grafico` dev'essere possibile fornire separatamente (e dunque in tempi diversi) le coordinate reali del grafico e le sue dimensioni;

Segue l'elenco delle modifiche prodotte rispetto al progetto precedente:

- viene eliminato il metodo `Disegna()`, la cui funzione di rendering viene svolta dal metodo `OnPaint()`;
- viene fornito il solo costruttore di default, e dunque eliminato il costruttore che accettava le coordinate minime e massime del sistema reale (valori ora acquisibili mediante un metodo);
- viene fornito un metodo – `ImpostaLimiti()` – per l'acquisizione delle coordinate minime e massime del sistema reale;
- viene fornita una proprietà – `Punti` – per l'acquisizione delle coordinate reali del grafico (matrice `double` di "n" righe per due colonne);
- la proprietà `Penna` viene modificata in accordo al modello utilizzato nella classe `GPListBox`.

6.1 Scheletro della classe «Grafico»

Segue lo scheletro della classe, che implementa le dichiarazioni delle variabili, il costruttore, proprietà e metodo di accesso ai dati. Degli altri metodi viene fornito il solo prototipo:

```
class Grafico: Panel
{
    Pen penna;
    double scalaX, scalaY;
    double minX, minY, maxX, maxY;
    double[,] punti;
```

```
    Point[] puntiInteri;

    public Grafico()
    {
        minX = -5;
        minY = -5;
        maxX = 5;
        maxY = 5;
    }

    public void ImpostaLimiti(double minX, double minY, double maxX,
                              double maxY)
    {
        this.minX = minX;
        this.minY = minY;
        this.maxX = maxX;
        this.maxY = maxY;
        puntiInteri = ConvertiCoordinate(punti);
        Invalidate();
    }

    public Pen Penna
    {
        get { return penna; }
        set
        {
            if (penna == value)
                return;
            penna = value;
            Invalidate();
        }
    }

    public double[,] Punti
    {
        get {return punti;}
        set
        {
            if (punti == value)
                return;
            punti = value;
            puntiInteri = ConvertiCoordinate(punti);
            Invalidate();
        }
```

```
    }

    protected override void OnPaint(PaintEventArgs e) { ...}

    Point XYtoP(double x, double y) { ... }

    Point[] ConvertiCoordinate(double[,] punti) { ... }
}
```

Analogamente a quanto avviene nella classe `GPListBox`, anche in questo caso qualsiasi modifica allo stato dell'oggetto implica l'invio di una richiesta di *painting* mediante l'invocazione di `Invalidate()`. Degno di nota è il costruttore, il quale assegna dei valori predefiniti alle coordinate minime e massime del sistema reale. E' questa una prassi abbastanza comune ed ha lo scopo di fornire all'oggetto uno stato iniziale consistente. (Rappresenta comunque una scelta arbitraria; lasciare le quattro variabili ai loro valori di default, e cioè zero, sarebbe stata una scelta altrettanto valida.)

6.2 Codice di rendering e metodo «ConvertiCoordinate()»

Il vecchio progetto della classe `Grafico` presupponeva che le operazioni di acquisizione e conversione delle coordinate e di successivo disegno del grafico fossero eseguite in rigida sequenza, poiché erano implementate nel metodo `Disegna()`.

La nuova classe `Grafico` funziona in modo diverso. Acquisizione coordinate, conversione coordinate, definizione limiti del sistema reale e rendering sono tutte operazioni che possono avvenire le une indipendentemente dalle altre. Ciò richiede una certa accortezza nella progettazione delle funzioni membro, che tradotto in pratica significa:

- l'operazione di conversione non deve presupporre che le coordinate siano già state acquisite;
- l'operazione di rendering non deve presupporre che le coordinate siano già state acquisite e convertite;
- l'operazione di impostazione dei limiti del sistema reale non deve presupporre che le coordinate siano già state acquisite.

Tutto questo conduce alle seguenti implementazioni dei metodi `ConvertiCoordinate()` e `OnPaint()`:

```
Point[] ConvertiCoordinate(double[,] punti)
    {
        if (punti == null)     // verifica che esistano le coordinate reali
            return null;
        scalaX = ClientSize.Width / (maxX - minX);
        scalaY = ClientSize.Height / (maxY - minY);
        Point[] puntiInteri = new Point[punti.GetLength(0)];
        for(int i = 0; i < puntiInteri. Length; i++)
            puntiInteri[i] = XYtoP(punti[i, 0], punti[i, 1]);
        return puntiInteri;
    }

    protected override void OnPaint(PaintEventArgs e)
```

```
    {
        if (puntiInteri == null)          // verifica che esistano le coordinate
            return;                       // convertite
        Graphics g = e.Graphics;

        if (penna == null)
            penna = new Pen(ForeColor, 1);
        g.DrawLines(penna, puntiInteri);
        base.OnPaint(e);
    }
```

Adesso gli oggetti di tipo `Grafico` si comportano come un vero controllo, in grado di rispondere alle richieste di *painting* e di aggiornare il proprio aspetto in relazione ad un cambiamento di stato.

6.3 Applicazione di esempio

L'applicazione che segue crea un oggetto di tipo `Grafico` e dopo aver impostato i limiti del sistema reale assegna le coordinate dei punti della funzione seno calcolate nell'intervallo del sistema. Tutto ciò viene fatto nel costruttore; il form contiene inoltre due bottoni, mediante i quali modificare la penna utilizzate per disegnare il grafico e le dimensioni del sistema reale.

```
public partial class Form1: Form
{
//. . .
    Grafico grafico;
    Button btnGrafico1, btnGrafico2;

    public Form1()
    {
        // codice necessario per costruire il form
        grafico.ImpostaLimiti(-5, -2, 5, 2);
        grafico.Punti = CalcolaPunti();
    }

    double[,] CalcolaPunti()
    {
        double dx = 10.0 / grafico.ClientSize.Width;
        double[,] punti = new double[grafico.ClientSize.Width, 2];
        double x = -5;
        for (int i = 0; i < punti.GetLength(0); i++)
        {
            double y = Math.Sin(x);
            punti[i,0] = x;
            punti[i,1] = y;

            x += dx;
        }
```

```
        return punti;
    }

    void btnGrafico1_Click(object sender, EventArgs e)
    {
        grafico.Penna = new Pen(Color.Red, 2);
    }

    void btnGrafico1_Click(object sender, EventArgs e)
    {
        grafico.ImpostaLimiti(-5, -1, 5, 1);
        grafico.Penna = new Pen(Color.Blue, 4);
    }

}
```

Di seguito è mostrato l'output programma, dopo l'avvio, dopo il clic sul primo bottone e infine dopo il clic sul secondo bottone:

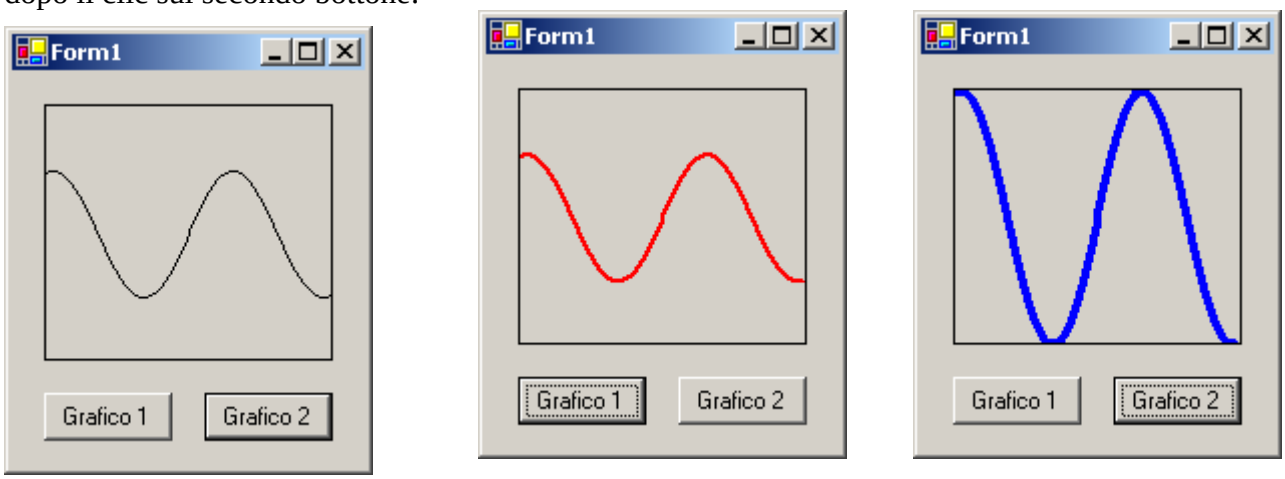


Figura 17-5 Output del programma

6.4 Gestione del ridimensionamento (*resizing*) del controllo

Esiste ancora un aspetto che non abbiamo preso in considerazione ed è quello della gestione dell'evento di ridimensionamento (*resizing*) del controllo. Tale evento viene sollevato dopo che le dimensioni del controllo sono state modificate.

La gestione di questo evento è necessaria quando lo stato del controllo dipende direttamente o indirettamente dalle sue dimensioni. E' questo il caso del controllo `Grafico`, poiché i due fattori di scala – `scalaX` e `scalaY` – sono calcolati sulla base della larghezza e dell'altezza del controllo. Se queste ultime variano, i fattori devono ovviamente essere nuovamente calcolati. Ma poiché il grafico stesso, e cioè l'aspetto del controllo, dipende dai fattori si scala, è ovvio che dopo averli aggiornati è necessario riconvertire i punti dal sistema reale al sistema intero ed eseguire nuovamente il codice di *rendering*.

La gestione del *resizing* viene fatta ridefinendo il metodo `OnResize()`. Essa consiste semplicemente nell'invocare il metodo `ConvertiCoordinate()` e invalidare il controllo:

```
protected override void OnResize(EventArgs e)
{
    puntiInteri = ConvertiCoordinate(punti);
    Invalidate();
    base.OnResize(e);
}
```

Un modo semplice per testare la gestione del *resizing* è ancorare un controllo di tipo `Grafico` ai quattro lati del form, in modo che qualsiasi modifica alle dimensioni di questo si rifletta automaticamente sulle dimensioni del primo. Ciò si ottiene impostando la la proprietà `Anchor`:

```
grafico.Anchor = AnchorStyles.Top | AnchorStyles.Bottom | AnchorStyles.Left |
                 AnchorStyles.Right;
```