Esame Microsoft 70-258: Windows-Based

Riccardo Pietrucci

31 marzo 2008

Indice

1 Controlli

1.1 Introduzione

Tutti controlli derivano dalla classe base *Control*. Il designer fornisce un ausilio per il posizionamento, tramite le *snaplines*, e uno wizard per l'impostazione delle proprietà, tramite gli *smart tags*. L'elenco dei controlli inseriti e il loro posizionamento nell'albero dei controlli si può osservare nel *Document Outline*.

Molte proprietà sono comuni a quelle web (semmai il contrario) tranne:

Anchor determina come il controllo mantiene le distanze rispetto al controllo padre che lo contiene a seguito di ridimensionamenti dello stesso padre. Si dichiarano quali angoli manterranno una distanza costante, a prescindere da eventuali ridimensionamenti del padre. Quindi, se impostiamo *Right,Left* allora a seguito di ridimensionamenti il controllo si allargherà o restringerà nella larghezza, mentre l'altezza rimarrà identica.

Dock determina come il controllo è attaccato ai bordi del controllo padre che lo contiene. Un controllo può ad esempio rimanere attaccato al top di un controllo e a seguito di ridimensionamenti rimarrà costantemente nel top.

ContainFocus indica se il controllo o uno dei suo figli ha il focus

Location indica la localizzazione dell'angolo in alto a destra del controllo

Region imposta la regione associata con il controllo

Molti controlli prevedono un menu contestuale nella proprietà *ContextMenuStrip*. Tutti i controlli prevedono degli eventi per il mouse e la pressione dei tasti.

1.2 Container Control per il Layout

I *container control* sono contenitori di altri controlli che consentono di visualizzare e manipolare i controlli contenuti in modo consistente. Cambiamenti al container causano cambiamenti ai controlli contenuti: impostare BackColor nel container influisce tutti i controlli contenuti. Ogni controllo prevede una proprietà *Controls* (pattern Composite) ed è necessario inizializzare tutte le proprietà *prima* di aggiungere il controllo creato alla lista dei Controls.

Il controllo *GroupBox* consente di suddividere logicamente una form in sezioni e viene spesso utilizzato per distinguere i vari gruppi di RadioButtons. La proprietà *Text* consente di impostare il nome della sezione. Il controllo non prevede alcuno scrolling.

Il controllo *Panel* crea una sottosezione nella form che consente di contenere altri controlli. Il controllo prevede scrolling a seconda della proprietà *AutoScroll* e può prevedere bordo a seconda della proprietà *BorderStyle*.

Il controllo *FlowLayoutPanel* estende Panel e consente di riposizionare dinamicamente i controlli contenuti a seguito di ridimensionamenti del contenitore. La direzione del flusso viene definita da *FlowDirection*: LeftToRight (default), RightToLeft, TopDown, BottomUp. Una volta raggiunta la fine del flusso (riga o colonna) si va a capo se vale *WrapContents*. Se invece non vale bisogna impostare manualmente i controlli in cui è concesso spezzare il flusso. Si usa per questo il metodo *SetFlowBreak* dove si passa il controllo che consente il break.

Il controllo *TableLayoutPanel* estende Panel è una tabella che fornisce celle adibite a contenere controlli. Per layout complessi si inseriscono nelle celle altri contenitori, che a loro volta conterranno i controlli. Lo stile è influenzato dalle proprietà *CellBorderStyle*, *ColumnStyle*, *RowStyle*, il cui *SizeType* può essere *Absolute* o *Relative*. Per aggiungere controlli si utilizza il metodo *Add* in Controls. Se si supera il limite di celle che la tabella può contenere il comportamento dipende dal valore di *GrowStyle*:

AddRows aggiunge righe per consentire l'inserimento

AddColumns aggiunge colonne per consentire l'inserimento

FixedSize non consente di aggiungere nulla: se si supera il limite stabilito solleva un'eccezione

Si può aggiungere un controllo specificando la cella interessata: se già occupata viene inserita alla prima cella disponibile, seguendo sempre il comportamento GrowStyle.

Il controllo *TabControl* estende Control e contiene vari controlli *TabPage* scambiando tramite tab. Lo stile è impostabile da *Appearance* e *Alignment* e *MultiLine*.

Il controllo *SplitContainer* eredita *ContainerControl* e consente di dividere due generici Panel tramite uno *Splitter*. I due Panel sono esposti tramite le proprietà *Panel1* e *Panel2*. Lo stile è impostabile tramite *Orientation*, *FixedPanel*, *IsSplitterFixed*, *SplitterDistance*, *SplitterWidth*, *Panel1Collapsed*, *Panel2Collapsed*, *Panel1MinSize*, *Panel2MinSize*.

1.3 Controlli di Command e di Testo

Il controllo *Button* consente di eseguire comandi, il cui stile è controllabile tramite *FlatStyle*. Se il comando prevede prima una conferma si può impostare un *DialogResult* al pulsante.

Il controllo *Label* consente di visualizzare informazioni testuali di sola lettura, frequentemente usate per coniugare informazioni testuali a controlli *TextBox*. Possono essere utilizzati per definire *access key* verso altri controlli. Per definirla è sufficiente specificare un testo che contenga un simbolo di ampersand, ad esempio: *"&First Name"* e impostare la proprietà *UseMnemonic* a true (di default comunque abilitato).

Il controllo *LinkLabel* consente di creare link simili a quelli del web, in cui si possono specificare informazioni di stile come nel web, tra cui la proprietà *LinkBehaviour* per specificare cosa succede al passaggio del mouse.

Il controllo *TextBox* consente di ricevere testo dall'utente e mostrare lo stesso all'utente. Se *MultiLine* è true allora si comporta come una TextArea web, e le singole linee inserite possono essere restituite come un array di stringhe tramite la proprietà *Lines*. Consente autocompletamento da una sorgente di dati (*AutoCompleteSource*), il che può essere suggerito o automaticamente concatenato a ciò che si sta scrivendo (*AutoCompleteMode*).

Il controllo *MaskedTextBox* è un TextBox modificato che consente di definire un pattern di input tramite la proprietà *Mask*. Una mask potrebbe essere: *"__/__/__"*, dove il carattere **_** è un carattere di *prompt*, mentre i caratteri fissi (**/**) sono i caratteri *literal*. Tale controllo prevede un *MaskedTextProvider* che fornisce il motore di parsing dei caratteri speciali che definiscono la mask[1]:

0 [0-9] (obbligatorio da 0 a 9)

9 [0-9]? (opzionale da 0 a 9)

[0-9 +-]?

L [A-Za-z]

? [A-Za-z]?

& .

C .?

A,a [A-Za-z0-9]?

. separatore decimale

, separatore migliaia

: separatore orario

/ separatore data

$ simbolo valuta

< converte a minuscolo

> converte a maiuscolo

| disabilita una precedente conversione a minuscolo/maiuscolo

**** carattere di escape

Tutti gli altri appaiono fissi nella maschera e non possono essere spostati o eliminati

I caratteri literal sono fissi e possono essere saltati automaticamente specificando *SkipLiterals*. *RejectImputOnFirstFailure* controlla come il testo incollato viene trattato: se fallisce viene completamente annullato o altrimenti vengono considerati solo i caratteri che soddisfano qualche criterio. I caratteri di prompt possono essere controllati tramite *ResetOnSpace*, *ResetOnPrompt* e *AllowPromptAsInput*. Per controllare il formato del testo viene copiato e incollato (che può essere diverso da quanto visualizza il controllo) si specifica il *CutCopyMaskFormat* a un valore tra *ExcludePromptAndLiterals*, *IncludeLiterals*, *IncludePrompt*, *IncludePromptAndLiterals*.

1.4 Controlli per contenere strutture dati

I controlli di Lista contengono una collezione di *Items* da mostrare che, anche se sono spesso stringhe, non è necessario che lo siano. Items rappresenta una collezione di oggetti generici. Spesso supportano il binding a una base di dati tramite le proprietà *DataSource*, in cui si agganciano tramite reflection i *DisplayMember* e *ValueMember*.

Il controllo *ListBox* è il più semplice controllo per mostrare liste[2]. *SelectionMode* consente di determina il comportamento di selezione, a scelta tra: *Single*, *MultiSimple* e *MultiExtend* (prevede uso combinato di Shift e Ctrl).

Il controllo *ComboBox* è simile al controllo ListBox a selezione singola, ma fornisce inoltre lo spazio all'utente per inserire manualmente un elemento, mantenendo allo stesso tempo

[1] viene riportato la corrispondente espressione regolare. Se specificato il simbolo '.' si intende qualunque carattere, a meno che non sia stata impostata a true la proprietà *AsciiOnly*, nel qual caso per '.' si intende solo [A-Za-z]

[2] coincide con un box in cui compare una lista di elementi selezionabili

la possibilità di sceglierne uno dalla lista[3]. Il comportamento è definito da *DropDownStyle*, impostabile a *Simple* (stile del ListBox a selezione singola), *DropDown* (default), *DropDownList* (senza immissione di dati manuali).

Il controllo *CheckedListBox* è una ListBox con a fianco dei check che consentono di effettuare selezioni multiple senza dover ricorrere a combinazioni di tasti alt e ctrl, come invece avviene nelle ListBox.

Il controllo *ListView* consente di mostrare liste allo stesso modo din *Windows Explorer*. Prevede quindi una proprietà *View* che consente di stabilire il modo di visualizzazione come in Windows Explorer (Icons, List, Details). Le icone possono essere raggruppate tramite *Groups*, allo stesso modo in cui Risorse del Computer comprende elementi raggruppati come file, removable device, hard disk, network, other. Gli elementi della lista (Items) sono una collezione di *ListViewItems* che contengono il gruppo di riferimento e i *SubItems* da visualizzare in caso di visualizzazione *Details*. L'immagine può essere impostata sia piccola che grande tramite *LargeImageList* o *SmallImageList*.

Il controllo *TreeView* consente di mostrare elementi in modo gerarchico e si comporta allo stesso modo del TreeView web.

Il controllo *NumericUpDown* consente di far ispezionare e selezionare all'utente un range di numeri. In modo simile, il controllo *DomainUpDown* consente di far ispezionare e selezionare all'utente un range di valori descrittivi.

1.5 Controlli per impostare valori

Il controllo *CheckBox* consente di abilitare o disabilitare un'opzione. Il valore assunto si può analizzare tramite *Checked* o *CheckState*, che può assumere valore *Checked*, *Unchecked* e *Indeterminate*, quest'ultimo presente se specificato *ThreeState*. I valori assunti non dovrebbero avere ripercussioni su altri CheckBox: per questo esiste il *RadioButton*, che consente invece una selezione mutuamente esclusiva. Se esistono differenti gruppi di esclusività si possono raggruppare i RadioButton in contenitori *GroupBox*.

Il controllo *TrackBar* è uno slider che consente di selezionare un valore a scelta tra un range di valori trascinando uno slider con il mouse.

Il controllo *DateTimePicker* consente di scegliere visualmente una data o un orario tramite un wizard. Il controllo *MonthCalendar* consente invece di selezionare una data nell'ambito di un calendario.

Il controllo *PictureBox* viene utilizzato per visualizzare le immagini da un file di risorsa, dal web o dal disco. L'organizzazione delle immagini possono essere gestite grazie al componente *ImageList*, che non è un controllo e non prevede interfaccia utente, ma viene utilizzato in altri controlli.

1.6 Controlli WebBrowser e NotifyIcon

Il controllo *WebBrowser* consente di utilizzare il browser IE all'interno di un'applicazione windows. Diventa utile in situazioni di help online, in cui il metodo più importante è *Navigate*.

Il componente *NotifyIcon* non è un controllo: consente di inserire un'icona nella system tray e di inserire menu contestuali tramite *ContextMenuStrip* ed eventuali messaggi informativi tramite *BaloonTipText* e il metodo *ShowBaloonTip*.

[3]coincide con una dropdown in cui è possibile digitare

1.7 Controlli finalizzati ad assistere l'utente

Alcuni controlli nascono con il solo scopo di migliorare l'usabilità e l'utilità dell'applicazione.

Il controllo *PropertyGrid* è una versione configurabile del Properties Window di Visual Studio, che scandisce tramite reflection il *SelectedObject* passato a run-time. L'approccio è bidirezionale: modifiche nei valori della grid vanno a modificare istantaneamente le corrispondenti proprietà dell'oggetto. Le proprietà possono essere ordinate secondo quanto espresso da PropertySort (per categoria, alfabeticamente, ...).

Il controllo *ProgressBar* visualizza una progress che va da *Minimum* a *Maximum*. Il metodo *PerformStep* avanza del valore *Step*, mentre *Increment* avanza del valore specificato. Il valore corrente è ottenibile tramite *Value*.

Il controllo *ToolTip* consente di mostrare aiuto contestuale ai controlli. Si può mostrare sottoforma di fumetto se *IsBaloon* è true. Il valore può essere impostato tramite *SetTooltip* che richiede in input il controllo interessato e il testo da associare. Il comportamento di visualizzazione varia a seconda delle temporizzazioni[4]:

InitialDelay tempo che intercorre tra il passaggio del mouse e la visualizzazione del tooltip (*AutomaticDelay*)

AutoPopDelay tempo in cui rimane visualizzato il tooltip ($AutomaticDelay \cdot 5$)

ReshowDelay tempo che intercorre tra un controllo su cui si sta visualizzando il tooltip e un controllo adiacente di cui si richiede il corrispondente tooltip ($\frac{AutomaticDelay}{5}$)

Il controllo *ErrorProvider* fornisce feedback all'utente nel caso in cui un particolare controllo ha generato una condizione di errore. Sono normalmente associati a controlli di validazione e mostrano un tooltip di errore. Il metodo che imposta questo è *SetError* e si comporta come il precedente SetTooltip. Per fare in modo che messaggi di errore scompaiano basta reinvocare il metodo con un messaggio di errore vuoto. Possono anche essere impostati con un DataSource in cui vengono notificati errori tramite, ad esempio, *DataRow.SetColumnError*.

Il controllo *HelpProvider* fornisce una documentazione sottoforma di file .htm o .chm alla pressione del tasto F1. La proprietà HelpNamespace consente di impostare il percorso di questi file. La visualizzazione dell'help (sommario, indice, ricerca, ...) è controllabile con il metodo *SetHelpNavigator*.

La classe *SoundPlayer* richiede nella sua costruzione il path dell'audio da riprodurre. Il metodo *Play* riproduce l'audio.

Il componente *Timer* consente di scatenare eventi *Tick* a intervalli regolari, definiti dalla proprietà *Interval*. La proprietà *Enabled* o i metodi *Start* e *Stop* consentono di abilitare o disabilitare lo scatenarsi di tali eventi.

I controlli *HScrollBar* e *VScrollBar* sono *ScrollBar* utilizzate in quei controlli che non hanno le scroll native, come il PictureBox. Si possono impostare proprietà come *SmallChange*, *LargeChange*, *Maximum*, *Minimum*.

Le impostazioni possono essere definite tramite il *Settings Editor* e specificare se un utente è in grado di modificare o meno tali proprietà a run-time definendo se lo *scope* della proprietà è *User* o *Application*

[4]tra parentesi è riportato come i valori vengono impostati nel caso si imposti *AutomaticDelay*

2 Menu

2.1 Tool Strip

Il controllo *ToolStrip* o *StatusStrip* consente di creare toolbar che hanno lo stesso stile di Office o di Internet Explorer. Contiene una collezione di elementi *ToolStripItem*. Gli elementi ToolStripItems possono a loro volta essere dei[5] Label, Button, *Separator*, ComboBox, Textbox, *ProgressBar*, *DropDownButton*, *SplitButton* (combina il Button con il DropDownButton fornendo il pulsante stile ControlPanel nel menu, dove un click espande il menu, mentre il doppio click apre il ControlPanel).

Vari ToolStrip sono inseriti nel *ToolStripContainer*, che consente il rafting (raggruppamento verticale/orizzontale). Infatti, il container contiene cinque pannelli: quattro *ToolStripPanel* e un *ContentPanel*.

La classe *ToolStripManager* consente di fondere due ToolStrip in un unico solo a seconda del comportamento definito in *ToolStripItem.MergeAction*, a scelta tra *Append*, *Insert*, *MatchOnly*, *Remove*, *Replace*. Ciò che avviene è che il ToolStrip di destinazione viene fuso in quello sorgente a seconda del match dei controlli, verificato tramite le proprietà *Text* dei controlli.

2.2 Menu Strip

Il controllo *MenuStrip* eredita da ToolStrip ma contiene una collezione di *ToolStripMenuItem* e possono essere aggiunti ricorsivamente, cosa che non era ammessa precedentemente. I menu item possono avere dei check, delle access key e delle *shortcut key* che, a differenza delle access key, possono essere invocate da qualunque parte del programma, indipendentemente dal fatto che siano o meno visibili.

Gli elememti dei menu possono essere spostati semplicemente aggiungendoli a un nuovo menu: questo scatena un automatica rimozione dal menu di origine. Un menu può essere disabilitato (*Enable*) o nascosto (*Visible*). Si noti che la shortcut viene disabilitata solo se Enable è false: se invece Visible è false non implica che sia anche disabilitata la shortcut.

L'intero menu può essere sostituito a runtime semplicemente eliminando il vecchio MenuStrip e aggiungendo il nuovo MenuStrip.

Il controllo *ContextMenuStrip* è come i MenuStrip ma rappresenta menu contestuali. Ogni controllo prevede infatti una proprietà ContextMenuStrip e appare alla pressione del pulsante destro sul controllo.

Il controllo *StatusStrip* è una sottoclasse di ToolStrip e consente di mostrare informazioni sullo stato. Tra i ToolStripItem che possono essere agganciati c'è il controllo *ToolStripStatusLabel* per mostrare info testuali e *ToolStripProgressBar* per mostrare una ProgressBar formato mignon.

3 Funzionalità

3.1 Stampa

Ogni *PrintDocument* contiene un istanza di *PrinterSettings*, che può essere modificata dai componenti *PrintDialog* e *PageSetupDialog*.

Il componente *PrintDialog* rappresenta la dialog di stampa classica, dove è possibile selezionare la stampante di lavoro, quali pagine stampare e se stampare su file. Le impostazioni vengono quindi salvate nel PrinterSettings a cui si riferisce. Alla pressione del tasto ok viene quindi normalmente eseguito il metodo PrintDocument.*Print*.

Il componente *PageSetupDialog* rappresenta la dialog corrispondente al Page Setup. Possiede un'istanza di PrintDocument e consente quindi di apportare modifiche al PrinterSettings che tale istanza a sua volta contiene. Normalmente consente solo di modificare le PrinterSettings, ma a seguito del pulsante ok non dovrebbe corrispondere la vera e propria stampa. Consente quindi di impostare il layout di stampa, come i margini, l'orientamento della pagina e il formato della carta.

Il metodo PrintDocument.Print stampa il PrintDocument sulla stampante specificata in PrinterSettings. Se viene utilizzato il componente *PrintPreviewDialog* allora l'output verrà redirezionato sullo schermo per mostrarne l'anterprima di stampa. Il componente consente di modificare il numero di pagine mostrate, modificare lo zoom e procedere quindi alla stampa vera e propria. Notare che eventuali modifiche nella resa di stampa non vengono salvate nel corrispondente PrinterSettings e vengono quindi perse alla chiusura della dialog. Più precisamente, tali informazioni vengono memorizzate nel *PrintPreviewComponent* che contiene, che in caso di preview customizzata può essere esteso per fornire funzionalità aggiuntive.

Il metodo PrintDocument.Print scatena uno o più eventi *PrintPage*. Gestire questo evento è necessario per stampare qualcosa, altrimenti non avverrà alcuna stampa. L'argomento *PrintPageEventArgs* consente di accedere a varie proprietà utili:

Margini la pagina intera è rappresentata da *PageBounds*, all'interno della quale vi è l'area di stampa *MarginBounds*. Nelle zone PageBounds che non coincidono con MarginBounds troviamo elementi come l'intestazione e il piè di pagina

Numero Pagine il gestore dell'evento PrintPage consente di stampare solo una pagina. Nel caso la stampa continui il gestore dell'evento prima di uscire dal metodo deve impostare *HasMorePages* a true. Il gestore dell'evento che si troverà a rendere l'ultima pagina[6] deve impostare invece HasMorePages a false

Cancellazione stampa l'esecuzione può essere interrotta tramite la proprietà *Cancel*, che deve essere analizzata dal gestore di PrintPage

Contenuto da stampare la proprietà *Graphics* è un oggetto che consente di svolgere tutti i possibili lavori di rendering richiesti, come DrawPath, DrawImage, DrawString

Al termine della stampa PrintDoucemnt scatena l'evento *EndPrint*, utile per messaggi informativi.

La stampa può essere vincolata a sicurezza tramite *PrintingPermission* che può assumere valori *NoPrinting* (nessuna stampa), *SafePrinting* (sol tramite dialog), *DefaultPrinting* (programmatico per la default, dialog per tutte), *AllPrinting* (programmatico e dialog per tutte).

[5] ogni nome contiene il prefisso *ToolStrip*

[6] si deve quindi fornire un modo per tracciare il numero di pagina

3.2 Drag and Drop

Il drag-and-drop è simile al cut-and-paste ma le azioni da eseguire sul controllo target devono essere descritte esplicitamente. L'operazione di *Drag-And-Drop*:

1. nel source viene inizializzata invocando, all'interno del gestore dell'evento Control.*MouseDown*[7], il metodo Control.*DoDragDrop* che copia i dati interessati dal controllo source a un *DataObject*
2. nel source opzionalmente viene gestito l'evento *GiveFeedback* per dare un feedback all'utente mediante cambio della forma del puntatore impostando un effetto a scelta tra l'enumerato *DragDropEffects*
3. nel source opzionamente viene gestito l'evento *QueryContinueDrag* per determinare se continuare o moeno l'operazione di drag
4. nel target viene gestito l'evento Control.*DragEnter*, il cui *DragEnterEventArgs* può essere utilizzato per ottenere il DataObject popolato dal source. Se il DataObject corrisponde a un tipo compatibile allora si imposta l'*Effect* corrispondente per dare un feedback positivo (o negativo) all'utente
5. nel target viene gestito l'evento *DragDrop* che esegue le operazioni necessarie con il DataObject ricevuto

Il DataObject espone metodi come *GetDataPresent* per determinare il tipo di dati presenti, e *GetData* per ottenere una *copia* del dato specificato.

Riassumendo, affinché il drag-and-drop possa avvenire è necessario che:

- il target control abbia la proprietà *AllowDrop* impostata a true
- il target control deve accettare dati compatibili con quelli del source control

3.3 Clipboard

La clipboard è la versione semplificata del drag-and-drop: possiede un metodo *Clipboard.GetDataObject* che restituisce, come il drag-and-drop, un DataObject.

3.4 Globalizzazione e Localizzazione

Per creare applicazioni che possano essere correttamente lette anche da popoli con lingue differenti è necessario poter impostare:

Globalizzazione formattazione degli elementi (valuta, simbolo decimale, ...). Coincide con *Thread.CurrentCulture*

Localizzazione valore testuale (ciao, hello, ...). Coincide con *Thread.CurrentUICulture*

Si possono creare versioni localizzate di un'applicazione impostando nel designer la proprietà *Language*. Questa restituirà la form con i valori testuali per quella specifica language, solo se la form ha impostata a true *Localizable*.

Alcune culture potrebbero impostare a true la proprietà *RightToLeft* che provoca la modifica dell'intero layout di una form.

[7] nel caso di TreeView si considera l'evento *ItemDrag*

3.5 Applicazioni MDI (Multiple Document Interface)

Quando una form deve coordinare più finestre aperte allora si ha un'*applicazione MDI* (*Multiple Document Interface*).

La form MDI padre (quella con Form.*IsMDIContainer* a true) organizza ed ordina tutti i documenti figli attivi (implementazione del design pattern *View Handler*). I vari *Form*[8] vengono creati dal padre e lo stesso padre imposta la proprietà Form.*MdiParent* a sé stesso. La form padre può ottenere la form attiva tramite la proprietà *ActiveMDIChild*. Le finestre aperte si possono riarrangiare tramite il metodo LayoutMdi, che può assumere uno dei valori dell'enumerato *MdiLayout* (Cascade, Tile, ...). Per visualizzare la lista dei child presenti, si può creare un MenuItem e impostare a true *MenuItem.MdiWindowListItem*: popolerà automaticamente il menu con i form figli.

3.6 Accessibilità

Risulta possibile sviluppare applicazioni accessibili solo se si inizia fin dal principio a pensarle in questi termini. Bisogna quindi evitare l'uso di sfondi o font customizzati ma utilizzare i font o colori di sistema, in modo che l'utente possa configurarli in caso di problemi di ipovedenza. Il namespace System.Drawing espone i *SystemBrushes*, *SystemColors*, *SystemFonts*, *SystemIcons* e *SystemPens*. Bisogna fornire un accesso alle funzionalità anche da tastiera, abilitando l'uso del tab mediante l'impostazione del *TabOrder*. Bisogna evitare utilizzi funzionali del suono: dovrebbe avere utilizzo solo ornamentale. Nel caso in cui abbia invece un uso bisogna combinare una notifica visiva all'evento, come un flash visivo invece di un beep. Tutti i controlli contengono inoltre delle proprietà per ulteriori ausilii: *AccessibleName*, *AccessibleDescription*, *AccessibleRole*, *AccessibilityObject*, *AccessibleDefaultActionDescription*.

3.7 Multi-tasking

Il multitasking può essere abilitato attraverso le tecniche convenzionali di async delegate e Thread. Oltre a queste, il componente *BackgroundWorker* consente di eseguire con maggiore semplicità le operazioni di un aync delegate:

- il metodo background corrisponde al gestore dell'evento *DoWork*. L'eventuale risultato deve essere memorizzato in *DoWorkEventArgs.Result*.
- il metodo BeginInvoke corrisponde a *RunWorkerAsync* (con o senza parametri)
- il metodo EndInvoke corrisponde al gestore dell'evento *RunWorkerCompleted*, dai cui argomenti si può ottenere il RunWorkerCompletedEventArgs.*Result*

Oltre a implementare le funzionalità di base, BackgroundWorker aggiunge funzionalità utili:

- **Stato di esecuzione del metodo**: la proprietà *IsBusy* consente di stabilire se il BackgroundWorker è attualmente in esecuzione. *Dovrebbe sempre essere invocato prima di eseguire RunWorkerAsync*

[8] spesso marcati come ChildForm al solo scopo di identificarli, ma non cambia nulla

- **Cancellazione dell'esecuzione del metodo**: se è verificata la proprietà *WorkerSupportsCancellation*, a seguito dell'esecuzione del metodo *CancelAsync* (normalmente provocato dal click di un Button) viene impostata la proprietà *CancellationPending*. Questa proprietà deve essere verificata attivamente dal nostro metodo in background (*polling*) e il metodo deve uscire se tale proprietà viene rilevata. Nel gestore di *RunWorkerCompleted* dovremo quindi verificare la proprietà RunWorkerCompletedEventArgs.*Cancelled* per determinare la causa della terminazione del metodo

- **Stato di avanzamento del metodo**: il metodo *ReportProgress* consente di riportare lo stato di avanzamento in percentuale. Tale metodo scatena l'evento *ProgressChanged*, normalmente gestito per aggiornare una ProgressBar.

I controlli in Windows Form sono *associati a un thread specifico e non sono thread-safe*. Pertanto, se si chiama il metodo di un controllo da un thread diverso, è necessario utilizzare il metodo *Control.Invoke(Delegate)* del controllo per effettuare il marshalling della chiamata al thread proprietario del controllo. La proprietà *InvokeRequired* può essere utilizzata per determinare se è necessario chiamare un metodo Invoke, che può essere utile se non si conosce il thread proprietario di un controllo. Ad esempio:

```
public delegate void SetTextDelegate(string t);
public void SetText(string t)
{
   if (textBox1.InvokeRequired)
   {
      SetTextDelegate del = new SetTextDelegate(SetText);
      this.Invoke(del, new object[]{t});
   }
   else
   {
      textBox1.Text = t;
   }
}
```

Se ad esempio textBox1 vive nel thread t_1 e lo invochiamo dal thread t_2 allora il metodo SetText chiama il metodo Invoke, che a sua volta richiama sé stesso. Alla riesecuzione di SetText il contesto è stavolta quello di t_1: i parametri sono stati trasferiti da t_2 a t_1 tramite marshal/unmarshal automatico. Viene quindi eseguita la seconda parte del metodo ed applicato realmente il testo desiderato a textBox1.

4 Persistenza dei dati

La persistenza viene gestita tramite le classi di ADO.NET, divise in due categorie: *connesse* e *disconnesse*

4.1 Classi ADO.NET disconnesse

Le classi disconnesse consentono di assegnare valori ad un'applicazione web senza essere necessariamente connessi. La classe fondamentale è la *DataTable*, composta da *DataColumn* (metadati o schema) e *DataRow* (dati) e *PrimaryKey* (riferimento a n DataColumn). I DataColumn consentono di impostare varie proprietà come[9]: *DataType* (String), *MaxLength* (-1), *Unique* (false), *AllowDBNull* (true), *Caption* (ColumnName).

Ogni DataRow contiene tre *DataRowVersion*: *Original*, *Current* e *Proposed*. Durante la modifica si opera con la Proposed e al termine di tale modifica (metodo *AcceptChanges*) Proposed passa a Current, e Current passa a Original. In questo modo è possibile ottenere la copia precedente di tale riga per supportare un semplice undo, limitato però a un singolo elemento di cronologia. Con l'undo (metodo *RejectChanges*) il rollback ripristina Original a Current, ma viene completamente perso il precedente Original (non è possibile eseguire un redo). Riassumendo, il RejectChanges consente di ripristinare lo stato assunto da AcceptChanges a un solo livello e non è previsto un redo. Inoltre, la versione *Default* restituisce Proposed se esiste, altrimenti Current. Nel caso in cui venga richiesto una versione inesistente viene lanciata un'eccezione (meglio verificare *HasVersion*).

Le DataRow possono avere uno dei seguenti *RowState*:

Detached riga creata ma non aggiunta

Added riga aggiunta

Unchanged riga senza alcun cambiamento dall'ultima esecuzione di AcceptChanges

Modified riga modificata dall'ultima esecuzione di AcceptChanges

è possibile impostare esplicitamente il RowState tramite i metodi di DataRow *SetAdded* e *SetModified*, utile per memorizzare i dati in un supporto differente da quello di default.

Per aggiungere righe si usa in modo appropriato[10] DataTable.*Add*, per importarle da un'altra DataTable si usa *ImportRow* e per aggiornarle si usa DataTable.*Load*, in cui si può specificare una delle seguenti *LoadOption*:

OverwriteRow sovrascrive le DataRowVersion *Original* e *Current*. Cambia sempre il RowState a *Unchanged*

PreserveCurrentValues (default) sovrascrive le DataRowVersion *Original*, ma lascia inalterata la Current. Cambia sempre il RowState a *Unchanged*

UpdateCurrentValues sovrascrive le DataRowVersion *Current*, ma lascia inalterata la Original. Il RowState cambia nel seguente modo:

- le righe aggiunte hanno *Added*
- le righe con stato Unchanged mantengono *Unchanged* se current è uguale a original, altrimenti viene impostato *Modified*

Per creare una copia completa (schema e dati) di un DataTable si usa il suo metodo *Copy*, utile ad esempio per supportare modifiche che prevedono un abort: un modo semplice è copiare la DataTable e reimpostarla al momento dell'apply. Per copiare solo lo schema ma non i dati si usa invece *Clone*.

DataView consente di visualizzare una DataTable impostando inoltre le proprietà *Sort*[11], RowFilter[12], RowStateFilter.

Il *DataSet* è la visuale relazionale dei dati. Contiene un'insieme di DataTable e di *DataRelation* tra varie DataTable e di creare quindi vincoli di integrità referenziale tra le DataTable collegate. Per creare id univoci si utilizza la classe *Guid*, che è

[9] tra parentesi il suo default

[10] i dati immessi devono soddisfare i vincoli di metadati specificati in DataColumn

[11] prevede una stringa nel formato, ad esempio: "LastName ASC, Salary DESC"

[12] coincide con la clausola WHERE dell'sql, senza il WHERE

una chiave surrogata: non ha senso modificare il valore assunto, ma serve solo a definire l'univocità della chiave e a collegare varie tabelle. Guid consente qiundi di simplificare il join di tabelle nel caso particolare di chiavi multiple. Senza l'uso di Guid, nel contesto di chiavi multiple si richierebbe di andare incontro a problemi di aggiiornamento ricorsivo.

Normalmente è buona pratica inserire versioni fortemente tipizzate degli oggetti DataSet, cioè sostituire espressioni come: footnotesize

```
DataTable companyTable = salesData.Tables["Company"];
```

con una classe che estende DataSet, che contiene inoltre:

```
DataTable companyTable = vendorData.Company;
```

Si può generare automaticamente la versione fortemente tipizzata utilizzando il DataSet Editor per modificare il file XSD che descrive il DataSet.

Il DataRelation consente di ottenere le righe che costituiscono il risultato di uno join. In particolare:

- **join in relazioni (1,N)**: metodo DataRow.*GetChildRows(DataRelation)*
- **join in relazioni (N,1)**: metodo DataRow.*GetParentRow(DataRelation)*

Si può specificare il comportamento in caso di update o delete che coinvolgono vincoli di integrità referenziale, impostando la proprietà DataRelation.*ForeignKeyContraint*:

Cascade il delete/update si ripercuote anche nella DataRow referenziata

None solleva una *InvalidContraintException*

SetDefault imposta al valore di default del tipo della DataRow referenziata (0, "", ...)

SetNull imposta a null il valore della DataRow referenziata

Si può serializzare e deserializzare il contenuto di un DataTable o di un DataSet in un file xml (*WriteXml*, *ReadXml*) o in uno stream (BinaryFormatter[13]). Gli spazi nei nomi sono convertiti nei caratteri "_x0020". Si può impostare il tipo di serializzazione xml tramite la proprietà DataColumn.*MappingType*: *Attribute*, *Element*, *Hidden* (utili nelle colonne di tipo Expression), *SimpleContent*. Si può inoltre scegliere il formato di esportazione di DataRelation.*Nested*. Nel metodo WriteXml è possibile specificare il *XmlWriteMode.WriteSchema* per memorizzare, oltre ai dati, anche lo schema associato. Viceversa, è possibile esportare solo lo schema tramite *WriteXmlSchema*. Tra i formati disponibili, si può salvare anche nel formato *XmlWriteMode.DiffGram* per salvare anche le versioni associate (Current, Original, ...). Il metodo ReadXml consente di leggere in vari modi:

Auto sceglie il miglior metodo

DiffGram applica i cambiamenti nel DataSet

Fragment analizza il caso di multipli nodi radice

IgnoreSchema ignora lo schema eventualmente memorizzato e disponibile

InferSchema estende il DataSet corrente con le informazioni dello schema

InferTypedSchema prova a determinare lo schema basandosi sui dati

ReadSchema legge lo schema. Se una DataTable già esiste viene lanciata un'eccezione

Bisogna scegliere accuratamente il formato: il formato binario, nonostante abbia dimensioni ridotte, contiene un maggior overhead di 20kb rispetto al formato xml. Quindi, nel caso di piccoli DataSet conviene il formato xml, altrimenti quello binario.

E' possibile combinare i dati di più DataSet tramite il metodo DataTable.*Merge*, che modifica il DataSet correntemente in uso e che prevede un *MissingSchemaAction*:

Add aggiunge le necessarie DataTable e DataColumn

AddWithPrimaryKey come Add, ma aggiunge anche le PrimaryKey

Error errore nel caso in cui è necessario un Add. Quindi se il merge viene applicato a una colonna inesistente non la aggiunge ma solleva un errore

Ignore ignora dati che risiedono in DataColumn che non trovano corrispondenza

Quando si utilizza il Merge è necessario che ogni DataTable abbia definita una chiave primaria, altrimenti non si è in grado di identificare le righe e verranno sempre aggiunte

4.2 Classi ADO.NET connesse

Le *classi connesse* o *provider class* rappresentano il bridge che consente di trasferire dati tra classi disconnesse e un data store. Il framework .NET contiene i seguenti provider:

OleDb accesso generico ai dati verso SQL Server 6.5, SyBase, DB2/400 e Microsoft Access

Odbc accesso generico ai dati quando non sono disponibili provider più recenti

SQL Server simile a OleDb, ma consente di accedere a SQL Server 7.0 e superiori

Oracle accesso per Oracle 8i e superiori

... accesso a ulteriori provider specializzati scaricabili da internet, come DB2 e MySql

Per utilizzare le classi connesse è possibile creare esplicitamente le classi specializzate le quali, anche se abilitano codice specializzato, richiedono però una conoscenza esplicita di tali classi da parte dell'utilizzatore. Per ovviare a questo problema si utilizzano le classi concrete *DbProviderFactory* (OracleClientFactory, OdbcFactory, ...), creabili tramite *GetFactory* e che consentono metodi *CreateConnection*, *CreateCommand*, *CreateParameter*, *CreateDataSourceEnumerator*, *CreateDataAdapter*. Ogni classe concreta è implementata come Singleton e l'elenco delle DbProviderFactory disponibili è accessibile tramite *DbProviderFactories.GetFactoryClasses()* e specificabile nelle config, nell'elemento `<system.data> <DbProviderFactories>`. Ogni classe

[13] necessario impostare anche la proprietà DataSet.*RemotingFormat* a Binary

connessa solleva eccezioni specializzate, che derivano da *DbException*.

Una *DbConnection* deve essere creata, aperta e chiusa. Per creare una connessione è necessaria una connection string valida, in cui si può specificare:

File Name il percorso fisico di un file che contiene la connection string

Driver, Provider il driver ODBC da utilizzare (anche text, excel, ...)

Server il nome del server a cui connettersi

UID nome utente

PWD, Password password utente

DBQ il path fisico verso il data source

AttachDbFilename, Extended Properties, Initial File Name percorso del file (spesso *.mdf*) nel filesystem locale contenente il database. Per la definizione del path viene supportata la keyword *—DataDirectory—* che punta alla cartella *Data* dell'applicazione. Ottiene o crea il file di log `<database>_log.ldf`.

Database, Initial Catalog il database a cui connettersi

Data Source, addr, address, network address, server il nome del server a cui connettersi, locale o remoto. Se remoto prevede un indirizzo IP nella forma *IP,PORT*. Per specificare un DBQ si deve anteporre il carattere '.
', nella forma .
PATH.

Connect Timeout (15) quanti secondi attendere per stabilire la connessione

Trusted_Connection se la sicurezza è basata sull'account di dominio dell'utente collegato

Persist Security Info se restituire l'intera connection string, compresa quindi delle informazioni di sicurezza

Integrated Security, trusted_connection (false) se stabilire una connessione sicura al SQL Server. Valori possibili: true, false e sspi

MultipleActiveResultSets (false) abilita MARS: consente di eseguire comandi multipli senza che sia necessario chiudere la connessione

Asynchronous Processing, async (false) abilita APM nei metodi di esecuzione delle query (BeginExecuteReader, EndExecuteReader, ...)

Pooling (true) abilita il connection pool, che consente di mantenere un pool di connessioni attive in modo da ridurre l'overhead richiesto per creare una connessione e impedire fenomeni di starvation. Risulta particolarmente utile distribuire il pool in un cluster di server, ma in questo caso devono valere le seguenti regole:

1. la connection string deve essere esattamente la stessa tra i vari server
2. l'uid deve essere lo stesso
3. l'id del processo deve essere lo stesso

Il pool viene creato e mantenuto nella stessa macchina che ha creato la DbConnection. Gradualmente viene adattato il numero di connessioni aperte, rilasciando le inattive dopo un tempo casuale compreso tra 4 e 8 minuti

Min Pool Size (0) minimo pool da mantenere. Utile inserire un numero basso, come 5, per garantire risposte rapide anche dopo lunghi periodi di inattività

Max Pool Size (100) massimo pool da mantenere, la 101esima connessione attende in coda di inserirsi nel pool fino a un timeout stabilito

Connection Reset (true) se, al momento della rimozione di una connessione dal pool, inserire informazioni di reset sulla connessione

Load Balancing Timeout, Connection Lifetime (0) utile per bilanciare il carico tra un server online e un server appena avviato

Enlist se inserire automaticamente la connessione nel contesto di transazione corrente del thread

Tramite `aspnet_regiis.exe` si può cifrare (`-pef`) e decifrare ((`-pdf`)) il contenuto di una connection string che usa le librerie DPAPI o RSA.

Una volta ottenuta la DbConnection si devono impostare le proprietà *CommandType* e *CommandText*, passando eventualmente una lista di *DbParameter* per la stored procedure, le quali richiedono le proprietà *ParameterName* e *Value*. Una volta impostato il comando si può eseguire in tre modi:

ExecuteNonQuery non aspetto un risultato (come in un update o delete). Restituisce il numero di tuple condizionate dall'operazione

ExecuteScalar mi aspetto un singolo risultato (voglio il numero di dipendenti)

ExecuteReader restituisce un *DbDataReader*, che rappresenta un cursore unidirezionale lato server di sola lettura.

ExecuteReader diventa quindi un metodo adatto a popolare dati ad esempio in una GridView, ma non per modificare dati letti e rispedirli al server. Per popolare una GridView si deve prima eseguire il metodo *DataTable.Load*, che richiede un DbDataReader, ed utilizzare la classe disconnesse DataTable per popolare GridView impostando la proprietà DataSource ed eseguendo il DataBind. Il metodo Load prevede una *LoadOption*, come già analizzato:

OverwriteChanges come OverwriteRow

PreserveChanges come PreserveCurrentValues

Upsert come UpdateCurrentValues

Multiple Active Result Sets MARS consente di eseguire comandi molteplici sulla stessa connessione, senza che sia necessario chiudere la connessione, consentendo di creare comandi annidati. Non aggiunge potere espressivo[14] ma in alcune situazioni facilita lo sviluppo, anche se peggiora le prestazioni.

[14] con due select annidati in realtà eseguo una join

Una DbConnection può essere monitorata sottoscrivendo l'evento DbConnection.*InfoMessage*.

Una DbConnection può creare *DbTransaction* tramite DbConnection.*BeginTransaction*, a cui deve seguire un DbTransaction.*Commit* o DbTransaction.*Rollback*.

Per leggere/scrivere dati molto grandi (*Binary Large OBject — BLOB*) è necessario trasferirli in blocchi, in modo da non esaurire la RAM del computer. Per questo, in fase di lettura si può trattare il DbDataReader come un flusso unidirezionale in cui è possibile leggere solo colonna per colonna e scrivere a blocchi il contenuto, ad esempio, in un FileStream. Si può abilitare questo comportamento passando l'enumerato *CommandBehavior.SequentialAccess* a DbCommand.ExecuteReader. Viceversa, in fase di scrittura è possibile scrivere nel database grazie alla seguente sintassi SQL:

1. ottengo il puntatore alla cella da scrivere

   ```
   SELECT TEXTPTR(Immagine) FROM Persona WHERE Matricola = '252762'
   ```

2. utilizzando il puntatore ottenuto, scrivo man mano a blocchi

   ```
   UPDATETEXT Persona.Immagine ImmaginePtr  0 null fiow3ef1ji5oejf
   UPDATETEXT Persona.Immagine ImmaginePtr 32 null sdchduivhuiwhvv
   UPDATETEXT Persona.Immagine ImmaginePtr 64 null qjiorhviuercqoi
   ```

   dove i parametri sono:

   (a) puntatore correntemente in uso. Non c'è quindi bisogno di specificare i vincoli di update, poiché la sola presenza del buffer indica con precisione cosa scrivere

   (b) offset: da dove iniziare la scrittura

   (c) byte da cancellare

   (d) i dati concreti che devono essere inviati

4.3 Bridge classi disconnesse – classi connesse

Per copiare i dati da una classe connessa a una classe disconnessa si utilizzano i metodi di *DbDataAdapter* (SqlDataAdapter, OracleDataAdapter, ...) tramite il metodo *Fill*, che richiede che sia stata impostata la proprietà *SelectCommand* a un oggetto *DbCommand* valido, che consente di impostare quale particolare insieme di tuple selezionare per popolare la classe disconnessa.

Viceversa, per copiare dati da una classe disconnessa a una classe connessa si utilizzano la funzione *SqlBulkCopy.WriteToServer*, impostando prima la proprietà SqlBulkCopy.*DestinationTableName*. Per trasferire solo i cambiamenti si usa di nuovo DbDataAdapter, ma stavolta il metodo *Update*, che richiede che siano state impostate le proprietà (in teoria opzionali) *InsertCommand*, *UpdateCommand* e *DeleteCommand*. In realtà questi metodi sono deducibili a partire dalla proprietà (obbligatoria) SelectCommand: infatti dopo la sua esecuzione è possibile ottenere il DbDataReader.*GetSchemaTable()*, che restituisce la DataTable contenente lo schema di colonne previsto e la chiave primaria prevista (una riga per ogni colonna: una metatable). A partire da queste informazione è possibile determinare a quale schema di dati applicare i metodi insert, delete e update senza che sia necessario definirli esplicitamente. Infatti i metodi generati sono definibili come:

```
// INSERT
INSERT INTO %TABLE_NAME%
    [%COLUMN_1%], [%COLUMN_2%]
VALUES
    @%ARG_1%, @%ARG_2%

// UPDATE
UPDATE %TABLE_NAME%
SET
    [%COLUMN_1%] = @%ARG_1%
    [%COLUMN_2%] = @%ARG_2%
WHERE
    (([%COLUMN_PK%] = @%ARG_PK%))

// DELETE
DELETE FROM
    %TABLE_NAME%
WHERE
    (([%COLUMN_PK%] = @%ARG_PK%))
```

Questo approccio garantisce migliore sicurezza rispetto a scrivere stored procedure apposite, poiché elimina completamente rischi di attacchi SQL Injection. E' possibile trasferire l'update a blocchi, invece che tupla per tupla utilizzando la proprietà UpdateBatchSize. L'utilizzo complessivo è il seguente:

```
DataSet pubsDataSet = new DataSet("Pubs");

string myProvider = ConfigurationManager.
          ConnectionStrings["PubsData"].ProviderName;
string myConnectionString = ConfigurationManager.
          ConnectionStrings["PubsData"].ConnectionString;
DbProviderFactory fact =
          DbProviderFactories.GetFactory(myProvider);

using(DbConnection conn = fact.CreateConnection()) {
    conn.ConnectionString = myConnectionString;
    conn.Open();

    DbCommand sel = conn.CreateCommand();
    sel.CommandType = CommandType.Text;
    sel.CommantText = "SELECT * FROM Publishers";

    DbDataAdapter da = fact.CreateDataAdapter();
    da.SelectCommand = sel;
    da.UpdateBatchSize = 3; // 3 tuple per volta

    da.Update(pubsDataSet, "publishers"); // ERRORE!

    DbCommandBuilder bldr = fact.CreateCommandBuilder();
    bldr.DataAdapter = da;
    da.InsertCommand = bldr.GetInsertCommand();
    da.UpdateCommand = bldr.GetUpdateCommand();
    da.DeleteCommand = bldr.GetDeleteCommand();

    da.Update(pubsDataSet, "publishers"); // OK!
}
```

Si può utilizzare il DbDataAdapter concreto *OleDbDataAdapter*, che consente di leggere record legacy ADO in un DataSet. Diventa così possibile visualizzare in componenti GUI dati legacy.

4.4 Dati in XML

Anche se esistono le classi connesse e disconnesse, il formato XML viene trattato in modo speciale per la sua portabilità. Il W3C ha elaborato una interfaccia standard di programmazione denominata *DOM* (*Document Object Model*), in grado di eseguire accesso casuale ai dati e modifiche di documenti xml. Le funzioni xml derivano dal namespace System.Xml ed estese da System.Data. Esistono diverse classi per gestire documenti xml, ognuna con vari gradi di funzionalità. Per la lettura di un file xml da disco e la creazione della corrispondente versione xml a oggetti esistono le classi:

Xml[Data Document] rappresenta la versione ad oggetti di un xml e implementa le specifiche DOM di livello 2. XmlDataDocument è una sua estensione che rappresenta anche dati relazionali. Si possono creare elementi (*CreateElement*),

attributi dell'elemento correntemente in uso (CreateAttribute) e aggiungerli all'oggetto (*AppendChild*). Nell'header è necessario inoltre aggiungere un elemento di intestazione, ottenibile tramite *CreateXmlDeclaration*. Si può caricare tale struttura in memoria tramite il metodo XmlDocument.*Load*. Gli elementi e gli attributi sono accessibili in modo programmatico tramite gli iteratori. E' possibile ricercare un *XmlNode* tramite *GetElementByID*, *GetElementByTagName*, *SelectSingleNode* e *SelectNodes*, in cui gli ultimi due consentono di passare una query XPath. Mentre il primo metodo richiede un DTD per determinare quali elementi sono ID, gli altri due non lo richiedono poiché eseguono la query su qualunque tipo di elemento.

XPathNavigator consente una navigazione efficiente di un XmlDocument. Dato quindi un XmlDocument, creato tramite il Load, è possibile creare un XPathNavigator tramite XmlDocument.CreateNavigator. La navigazione avviene tramite metodi come *MoveToRoot*, *MoveTo[First—Next]Attribute*, *MoveTo[FirstChild—Next]*, *HasAttributes*, *HasChildren*. Anche se funziona con XmlDocument, dovrebbe se possibile essere utilizzato con XmlPathDocument, specificatamente pensato per XPathNavigator. Il metodo *Select* consente di eseguire unaquery XPath e di restituire un *XPathNodeIterator*, che consente di iterare tra elementi XPathNavigator (e non XmlNode, come in XmlDocument). Il motivo per utilizzare XPathNavigator rispetto a XmlDocument stà nella possibilità di ordinare l'output sulla versione compilata della query XPath. Si può compilare una query XPath tramite *XPathExpression.Compile*, utilizzare XPathExpression.*AddSort* e passare tale oggetto al metodo XPathNavigator.Select.

XmlReader unidirezionalmente legge e verifica l'xml, ma non genera la corrispondente versione DOM. Consente di passare un *XmlReaderSettings* per far eseguire la validazione, da passare al metodo *XmlReader.Create*. La validazione avviene al momento del XmlReader.*Load*. Alla fine vuole un *Close*. Diventa la soluzione ideale quando c'è la possibilità che l'informazione che si vuole trovare è verso l'inizio del file xml, in quanto fornisce un accesso forward e privo di cache, opposto a quello random degli altri metodi, che prevede un caching. Prevede metodi di scansione molto simili all'XPathNavigator. XmlReader è una classe astratta: implementazioni concrete sono *XmlTextReader* o *XmlValidatingReader*. Infine, è possibile ottenere la propria implementazione di XmlReader semplicemente estendendo XmlReader

Quando si lavora con oggetti XML risulta quindi più semplice lavorare con i metodi DOM forniti dalla classe XmlDocument, ma ci sono limiti nelle capacità di ricerca che possono implicare un intera scansione dell'albero per raggiungere l'output desiderato. D'altra parte, XPathNavigator è meno intuitivo ma consente query più sofisticate e diventa la soluzione ideale per contesti di utilizzo xml complessi. Viceversa, per la scrittura di un oggetto xml nel corrispondente file xml esistono le classi:

XmlWriter scrive l'xml secondo lo standard XML 1.0, ma senza tradurre da una versione DOM. Utilizza XmlConvert per convertire tra i data type CLR e i data type XSD e tradurre spazi in `_x0020_`. Consente di impostare lo stile e altre caratteristiche. Al termine della scrittura segue un *Close* per non perdere dati. Implementazioni specifiche: *XmlTextWriter*, ...

Xml[Data Document] consente di scrivere (*Save*) XmlDocument in un file xml. Tipicamente questo avviene dopo un Load, dopo eventuali *RemoveChild* o *InsertAfter*.

4.5 Controllo DataGridView

Il controllo *DataGridView* consente di visualizzare un DataTable o un DataSet impostando la proprietà DataSource e il DataMember da visualizzare.

Le colonne riflettono il tipo ed esistono 6 tipi predefiniti di colonne: *DataGridViewTextBoxColumn*, *DataGridViewCheckBoxColumn*, *DataGridViewImageColumn*, *DataGridViewButtonColumn*, *DataGridViewComboBoxColumn*, *DataGridViewLinkColumn*. Oltre ai tipi predefiniti è possibile creare un tipo customizzato di colonna derivando da *DataGridViewColumn*. Le colonne possono essere aggiunte in *Columns* e si specifica prima dell'inserimento il *Name*, *HeaderText* e il *ValueType*.

La cella correntemente cliccata è la *CurrentCell*. L'evento *CellValidating* consente di validare l'input e cancellare l'edit in caso di errori e di impostare l'*ErrorText* della riga. L'evento *CellPainting* consente di personalizzare la visualizzazione delle celle.

5 Personalizzazione

5.1 User Control

I *controlli compositi o user-control* sono il tipo più semplice di controlli customizzati e consentono di raccogliere vari controlli in una singola unità funzionale. I controlli compositi ereditano dalla classe *UserControl* e prevedono il supporto del designer per trascinare i vari elementi.

Si può configurare il controllo in modo da avere un background trasparente in due modi:

- nel caso di un semplice controllo, impostando a *Color.Transparent* la proprietà BackColor
- nel caso di un controllo che si comporta come una finestra all'interno di una form impostando a BackColor lo stesso colore assunto di trasparenza assunto dalla form, indicato da *TransparencyKey*.

Si può fornire un'immagine per il controllo nella toolbox impostando la proprietà *ToolboxBitmapAttribute*, specificando un path verso un'immagine o un tipo di un controllo già esistente o una risorsa inglobata nell'assembly specificando sé stesso come tipo e specificando il path verso l'immagine.

5.2 Custom Control

I *custom control* consentono maggiore personalizzazione al costo di una maggiore complessità per la mancanza del supporto del designer. I custom control ereditano dalla classe *Control*[15] e comprendono funzionalità di base come la rilevazione

[15] a differenza degli user control che ereditano da UserControl

di eventi del mouse. Per personalizzare la visualizzazione basta sovrascrivere il metodo *OnPaint* manipolando l'oggetto *Graphics* restituito tramite i *PaintEventArgs* e disegnare una figura che rientri nel box che va dalle coordinate *(0,0)* a *(Control.Width,Control.Height)*.

5.2.1 Grafica

Per creare grafica la classe System.Drawing.*Graphics* fornisce metodi per disegnare per un supporto grafico. La classe *Pen* consente di disegnare linee e curve nei metodi Draw* di Graphics mentre le classi che derivano dalla classe astratta *Brush* consentono di riempire una forma geometrica nei metodi Fill* di Graphics. Per visualizzare le immagini create in una GUI è necessario utilizzare il componente *PictureBox*. Le struct più utilizzate sono *Color*, *Point*, *Rectangle* e *Size*.

Per creare un'istanza di Graphics è possibile utilizzare il metodo CreateGraphics nei componenti GUI o crearla a partire da una classe *Image*, di cui un'implementazione concreta è la classe *Bitmap*[16], che contiene inoltre i due metodi GetPixel e SetPixel. La classe Image contiene metodi statici di utilità per la creazione di immagini e il salvataggio. Nel metodo Image.Save consente anche di specificare l'ImageFormat (di default jpg).

La classe *SystemIcons* consente di ottenere le icone di sistema, utili per dialog di errore o di warning o in altri contesti.

E' possibile inserire testo in una *Graphics* creando un oggetto *Font* specificandone la *FontFamily* e la dimensione e successivamente invocando il metodo di Graphics *DrawString* che accetta un font e opzionalmente un'istanza Brush e uno *StringFormat* per controllarne la formattazione.

Si noti che il sistema di coordinate viewport (quello utilizzato dal namespace Graphics) ha origine in alto a sinistra. L'ascissa si comporta nel modo usuale: valori maggiori considerano valori più a destra, mentre l'ordinata si comporta nel modo contrario di come siamo abituati ad usarla: valori maggiori considerano valori più in basso (essendo l'origine in alto).

Si noti infine che si dovrebbe sempre richiamare la *Dispose* degli oggetti Pen, Brush e Graphics, pertanto è sempre opportuno utilizzare tali oggetti tramite lo *using*.

5.2.2 Estendere controlli di default

I Custom Control non devono necessariamente ereditare solo da Control, ma è possibile ereditare anche da un controllo esistente, come Button, estendendo le funzionalità del controllo. Se i metodi vengono sovrascritti richiamando anche il metodo padre si stanno estendendo le funzionalità, mentre se non viene richiamato il metodo padre si stanno completamente ridefinendo.

Le *DialogForm* sono frequentemente estese per fornire una dialog customizzata per l'inserimento di valori utente e per questo la sua estensione è supportata anche dal designer mediante dei template pronti all'uso. Le dialog possono essere non modali se visualizzate con il metodo *Show* o modali se visualizzate con il metodo *ShowDialog*. Quest'ultima restituisce una *DialogResult* per analizzare lo stato di uscita della dialog (Cancel, Ok). Al momento della show è possibile passare la form padre, riottenibile in seguito analizzando la proprietà *ParentForm*.

[16]Bitmap è un oggetto utilizzato per operare con immagini definite dai dati pixel, non c'entra niente con l'estensione BMP: si può scrivere infatti `Bitmap b = new Bitmap(foo.jpg)`