

.NET Collections for Data Bound Controls

(with FREE Bonus Sections)

by Rob Bracken

Amaze yourself by creating collections of your own objects that you can use with .NET grids, lists and combo boxes! Marvel as you use them with “foreach”! Astound your friends by filtering out unsightly objects! I presented this topic at the June meeting, but I’ve produced this article to get the information to those of you who were unable to attend. I’ve also added a couple of Bonus Sections that weren’t in the presentation, and supplied some sample source code.

All but the simplest business models contain collections of business objects. There’s always more than one way to build these collections – the challenge is to save myself some work by building a collection that can be used with other parts of the .NET framework. In particular, I want to load it into standard .NET controls. To demonstrate what’s involved, I’ll develop a *csCustomerList* collection that contains a list of *csCustomer* objects. Here’s the definition of the *csCustomer* class (Listing 1), note that I also defined an *ICustomer* interface to make it easier to experiment with both Delphi and C# implementations:

```
public class csCustomer: ICustomer
{
    private IList _owner;

    private string _title;
    private string _firstName;
    private string _lastName;

    public csCustomer( IList p_owner )
    {
        if ( p_owner != null )
        {
            _owner = p_owner;
        }
    }

    public override string ToString()
    {
        return ( ( ICustomer )this ).FullName;
    }

    public string FirstName
    {
        get { return _firstName; }
        set { _firstName = value; }
    }

    public string LastName
    {
        get { return _lastName; }
        set { _lastName = value; }
    }

    public string Title
    {
        get { return _title; }
        set { _title = value; }
    }

    public string FullName
    {
        get
        {
            // Put the Title, First name & Last name together
            return _title + ' ' + _firstName + ' ' + _lastName;
        }
    }
}
```

Listing 1 – the *csCustomer* class

Grids!

Love 'em or hate 'em, no self-respecting demo program would be complete without a grid – the most complex control for displaying and editing collections. If my collection works correctly with a grid, it'll work with other list controls, too.

The help for a .NET *DataGrid* control tells me it will accept a variety of data sources. The one I'm interested in is "Any component that implements the *IList* interface". So what's this *IList* interface? Again, checking the help I find (not surprisingly) that it contains methods for managing lists such as *Add*, *Clear*, *Insert*, *Remove* and *RemoveAt*. It also lists some classes that implement it, such as *Array*, *ArrayList* and *CollectionBase*. Any list will need methods like these, so it looks like a good place to start.

The simplest way to provide a collection for a *DataGrid* is to load my objects into an *Array* or *ArrayList*. I can assign the *ArrayList* to the grid's *DataSource* property to display it. *ArrayLists* are not strongly typed, however, and it would be possible to load a different type of object into it by mistake. This would generate a run-time error when the program tried to use it as the original object type. It's better to create a strongly typed collection class for my objects. The framework contains the *CollectionBase* class to use as a base for strongly typed collections. It contains an *ArrayList* to hold all the objects, but lets you wrap the *IList* methods inside strongly typed method calls to make sure you can only put in objects of the correct type. For example, the strongly typed *Add* method will look like Listing 2,

```
public class csCustomerList: CollectionBase
{
    public int Add( ICustomer p_newCustomer )
    {
        return List.Add( p_newCustomer );
    }
}
```

Listing 2 – an Add method in a collection that descends from *CollectionBase*

where *List* is the collection, returned as an *IList*. (There is also an *InnerList* property that gives direct access to the *ArrayList* inside the *CollectionBase*, but – as you'll see later – it's better to use *List* here.) You can see that it takes a *csCustomer* as its argument, to make sure that the inner *ArrayList* will only contain *csCustomer* objects. I need to implement *Add*, *Contains*, *IndexOf*, *Insert* and *Remove*. The advantage of this is that if I try to put the wrong type of object into the list, the compiler will give me an error.

This is fine if I'm using an object that I've declared as a *csCustomerList*. If I'm using the *IList* interface, however, all its methods take *TObject* parameters, so if you create a *csCustomerList* and put it into an *IList* variable, you can put any old object you like into the list and sit back while the customer support team tears their hair out! One way to avoid this is to re-implement the *IList* interface in the collection, checking that all objects are the right type. This involves writing new versions of all the members of *IList* - *Add*, *Contains*, *IndexOf*, *Insert*, *Remove*, *RemoveAt*, *get_FixedSize*, *get_IsReadOnly*, *get_Item* and *set_Item* – such as the new *Add* method shown in Listing 3.

```
( csCustomerList )
int IList.Add(object value)
{
    if ( value is ICustomer )
    {
        int result = this.InnerList.Add( value );
        OnListChanged( this, new ListChangedEventArgs( ListChangedType.ItemAdded, result ) );
        return result;
    }
    else
    {
        throw new ArgumentException( "You may only add an ICustomer to a csCustomerList" );
    }
}
```

Listing 3 – an example of type-checking in the Add method of an *IList* interface

Fortunately there's an easier way. *CollectionBase* already implements the methods of *IList*, and it calls some protected methods as it modifies the list. The one I'm most interested in here is the *OnValidate* method that's called from *Add*, *Insert*, *Remove*, *RemoveAt* and *set_Item*. If I override *OnValidate* and make it test the object I'm about to add, remove or set I can throw an exception if the object is the wrong type. This is a lot easier than rewriting all the *IList* methods and gives the same result (Listing 4).

```

( csCustomerList )
protected override void OnValidate(object value)
{
    if ( ! ( value is ICustomer ) )
    {
        throw new ArgumentException( "You may only put an ICustomer in a csCustomerList" );
    }
}

```

Listing 4 – adding type checking to a class that descends from *CollectionBase*

If I create one of these collections and put some objects into it I can assign it to the *DataSource* property of a *DataGrid* to display it (Listing 5).

```

public void CreateCollection()
{
    csCustomerList l_customerList = new csCustomerList();
    LoadCollection; // Put some Customers into the
                   csCustomerList
    MyDataGrid.DataSource = null; // Clear existing data source
    MyDataGrid.DataSource = l_customerList; // Display the list of Customers
}

```

Listing 5 – display a collection in a *DataGrid*

(Note – if you want a property of a Delphi object to display in a *DataGrid*, it must be *published*. C# properties must be *public*). Once I've got the *DataGrid* to display the collection, I can modify what's displayed in the usual way with the *TableStyles* property. In this case, the *MappingName* for the table style is "csCustomerList".

Sheep from Goats

Great! I've got a collection that I can give to a standard *DataGrid*. My next goal is to be able to sort the elements in the collection. There's no *Sort* method in the *IList* interface, but the *ArrayList* class has an overloaded *Sort* method. As my collection is a wrapped *ArrayList* (courtesy of *CollectionBase*), I'll use its *Sort* method to sort the collection.

The first version of *Sort* takes no parameters. Help says it "Sorts the elements in the entire *ArrayList* using the *IComparable* implementation of each element". The Help for *IComparable* shows that it only contains one method – *CompareTo* – that returns zero, a positive integer or a negative integer, depending on whether the object that contains the *CompareTo* method is the same as, greater than or less than the object being compared. For example, suppose I want to test whether a string is longer than 255 characters. I could write a function like that in Listing 6.

```

public bool StringIsTooLong( string p_string )
{
    if ( p_string.Length.CompareTo( 255 ) == 0 )
        return false; // String is 255 characters long
    if ( p_string.Length.CompareTo( 255 ) < 0 )
        return false; // String is less than 255 characters long
    if ( p_string.Length.CompareTo( 255 ) > 0 )
        return true; // String is more than 255 characters long
}

```

Listing 6 – using *CompareTo* to test the length of a string against 255

So, I need to decide how to compare two Customers. I'll do this by sticking the Last Name, the First Name and the Title together (in that order) and comparing the resulting strings (Listing 7). This method goes in the *csCustomer* class, which now has to declare that it supports *IComparable*.

```

public class csCustomer: ICustomer, IComparable
{
    int IComparable.CompareTo( object p_customer )
    {
        string l_mySortName = _lastName + _firstName + _title;
        string l_objectsSortName = ( ( ICustomer ) p_customer ).LastName +
            ( ( ICustomer ) p_customer ).FirstName +
            ( ( ICustomer ) p_customer ).Title;
        return l_mySortName.CompareTo( l_objectsSortName );
    }
}

```

Listing 7 – comparing a *csCustomer* with another (the *IComparable* interface)

All I have to do now is add a *Sort* method to my collection which calls the *ArrayList's Sort* method (Listing 8).

```
( csCustomerList )
public void Sort()
{
    this.InnerList.Sort();
}
```

Listing 8 – sorting the contents

This works, but it's not very flexible and it doesn't allow for different results from different human languages. It's time to look at the other versions of the *Sort* method, both of which take an *IComparer* parameter. Looking at Help again, it tells me that the *IComparer* interface contains one function – *Compare* – that takes two objects as its parameters and returns an integer. I can write comparer classes that implement *Compare* differently, depending on how I want to sort the Customers. Suppose I want to sort them by their Last Name. The comparer would look like Listing 9. If I want to take account of different human languages, I can pass a *CultureInfo* to the comparer in the constructor and change the comparison code to use it.

```
public class csCompareCustomerByLastName: IComparer
{
    int IComparer.Compare( object p_customer1, object p_customer2 )
    {
        if ( ( p_customer1 != null ) && ( p_customer1 is ICustomer ) && ( p_customer2 != null )
            && ( p_customer2 is ICustomer ) )
        {
            return ( ( ICustomer ) p_customer1 ).LastName.CompareTo( ( ( ICustomer ) p_customer2
                ).LastName );
        }
        else
        {
            throw new ArgumentException( "Incorrect arguments passed to Compare" );
        }
    }
}
```

Listing 9 – a Comparer object to compare Customers by Last Name

The framework contains a *Comparer* class that will compare two objects, but I can't use that here because it will use the *CompareTo* method of *csCustomer* which will always compare them the same way.

I need to add another, overloaded *Sort* method to my collection to use a comparer (Listing 10). For completeness, I've added a *Reverse* method to reverse the order of the Customers in the list. (*ArrayList* does this for me.)

```
{csCustomerList}
public void Sort( IComparer p_comparer )
{
    this.InnerList.Sort( p_comparer );
    OnListChanged( this, new ListChangedEventArgs( ListChangedType.Reset, 0 ) );
}

public void Reverse()
{
    this.InnerList.Reverse();
    OnListChanged( this, new ListChangedEventArgs( ListChangedType.Reset, 0 ) );
}

public void SortByLastName()
{
    csCompareCustomerByLastName l_comparer = new csCompareCustomerByLastName();
    Sort( l_comparer );
}
```

Listing 10 – using a comparer to sort by Last Name (with example)

Here's an opportunity to be creative. I can design comparers and *CompareTo* methods that are event-driven, or use *Reflection* to sort by different properties. I'll leave it to you to work out the details.

foreach his own

I can now display and edit my collection in a grid and sort it by different properties. I now want to be able to use it with "*foreach*" – available in C# and Visual Basic (and soon Delphi, I hope). The .NET framework uses the *Iterator* pattern here, but it calls the iterator an enumerator, so the interfaces I need to implement are *IEnumerator* and *IEnumerable* rather than *Iterator* and *IIterable* (or perhaps *IIrritable* if you've had a bad day).

The *IEnumerable* interface is the one my collection has to implement. It contains one method – *GetEnumerator* – that returns an *IEnumerator* object. Fortunately, the *CollectionBase* class that is the base class for my collection already implements this, returning an enumerator for the contained *ArrayList*. So my collection will already work with *foreach*, without me having to do anything. What I also want to do with the iterator, however, is to filter the list of Customers. I can use this feature to – for instance - search for all the customers whose last name begins with “S”. To do this I’ll have to use a custom enumerator and declare that my collection supports the *IEnumerable* interface. This means I have to add the *GetEnumerator* method to my collection (Listing 11). Note that it has to be *public*, or *foreach* will use the enumerator from the base class, rather than this one, because it’s declared *public* in the base class.

```
public class csCustomerList: CollectionBase, IEnumerable
{
    public new IEnumerator GetEnumerator()
    {
        return new csCustomerListEnumerator( this, LastNameBeginsWith );
    }
}
```

Listing 11 – implementing the IEnumerable interface

Of course, I also have to write the custom enumerator. It will need to implement the *IEnumerator* interface, which has two methods, one property and a constructor. You can see a typical, very simple enumerator in Listing 12. If I use this enumerator outside of the automatic “*foreach*” statement, I have to create one, passing in the list I want to enumerate and then keep calling *MoveNext* until it returns *False*. Each time I call *MoveNext*, the *Current* property gives me the element from the list. The *Reset* method sets everything back to the start so I can repeat the enumeration.

```
public class csCustomerListEnumerator: IEnumerator
{
    int _position;
    csCustomerList _customers;

    public csCustomerListEnumerator( csCustomerList p_customers )
    {
        _customers = p_customers;
        _position = -1;
    }

    object IEnumerator.Current
    {
        get { return _customers[ _position ]; }
    }

    bool IEnumerator.MoveNext()
    {
        if ( _position < ( _customers.Count - 1 ) )
        {
            _position++;
            return true;
        }
        else
        {
            return false;
        }
    }

    void IEnumerator.Reset()
    {
        _position = -1;
    }
}
```

Listing 12 – a simple Enumerator

If I want to filter the list, I can modify the *MoveNext* method so it skips any objects in the list that don’t meet the filtering criteria. I want to test the Last Name of each *csCustomer* object to see if it starts with a supplied string. What I’ll do is add a *LastNameBeginsWith* property to my collection and pass it to the enumerator in the constructor. Then I’ll change the *MoveNext* method to look like Listing 13.

Developer-friendly web hosting and dedicated servers

with great support

Are you fed up with explaining what you need to your web host? Wish they understood about .NET, ISAPI, CGI and XML? Wish you could get some real technical support?

At TDMWeb we know about software development: we also publish *The Delphi Magazine*, and deal with developers all day long!

So talk to us and you're talking with friends. We even have a special Windows Developer package designed just for the development and testing of web applications.

From simple shared hosting to your own dedicated server (complete with our Total Management service if you need it), we can do it all.

- Windows and Linux hosting and dedicated servers.
- ISAPI, ASP, ASP.NET, CGI, PHP, Perl, SSI, MySQL, Access, SQL Server, SOAP, XML, XSLT and more.
- Spam and virus protection.
- Excellent prices, great support.

Full package details at www.TDMWeb.com
Call +44 (0)870 740 7610 or Email info@tdmweb.com

www.tdmweb.com 

VS.NETcodePrint is an add-in to Microsoft® Visual Studio .NET that enables you to produce professional style printouts of the source code of Visual Basic .NET and Visual C# .NET applications. VS.NETcodePrint professional styled color coded printouts are ideal for:



- Learning programming using Visual Basic .NET and Visual C# .NET
- Documenting Visual Basic .NET and Visual C# Projects
- Debugging and supporting complex systems developed using Visual Basic .NET and Visual C# .NET
- Code inspections
- Submitting presentations to tutors and/or clients

You can preview the output on screen before printing or exporting to RTF, HTML and PDF formats. The output is fully customizable enabling professional looking color coded output to be generated which can be exported to files with Rich Text Format (RTF), HTML and Adobe Portable Document Format (PDF) formats.

STARPRINT
2000

```

bool IEnumerator.MoveNext()
{
    // Test to see if there is another element of the list after the current one
    if ( _position < ( _customers.Count - 1 ) )
    {
        int l_oldPosition = _position;
        _position = _position + 1;
        // Do filtering, if required
        if ( _lastNameBeginsWith.Length > 0 )
        {
            // Check the start of the Last Name against the comparison string (case
            // insensitive)
            // If they don't match, move to the next Customer, until we reach the end of the
            // list
            while ( ( _position < _customers.Count ) &&
                ( string.Compare( ( ( ICustomer ) ( ( IList )_customers )[_position ]
                ).LastName, 0, _lastNameBeginsWith, 0, _lastNameBeginsWith.Length, true ) != 0 )
            )
            {
                _position = _position + 1;
            }
        }
        bool l_result = _position < _customers.Count;
        if ( ! l_result )
        {
            // If there isn't another Customer that meets the filtering criteria,
            // restore the old position
            _position = l_oldPosition;
        }
        return l_result;
    }
    else
    {
        return false;
    }
}

```

Listing 13 – a filtering *MoveNext* method

Now, if I pass a string in the constructor, the enumerator will skip any Customers whose Last Name doesn't begin with that string. There's another opportunity for creativity here – *MoveNext* could call an event handler to test whether the Customer meets the filtering criteria, or could use *Reflection*. Note that if I pass a blank string to the constructor, the enumerator will return all the Customers. A word of warning – be sure to clear any filtering straightaway if you're using the same list in a grid, otherwise it'll get thoroughly confused. It's a good idea to make a filtered copy of the list if you want to use it for any editing.

More automation

My collection's doing all the things I want it to, but there's still a problem. When I attach it to a *DataGrid*, it displays happily and I can use the grid to edit the Customers. If I add a Customer, or sort the list, however, the grid doesn't change unless I set its *DataSource* to *nil* (or *null*) and reassign the collection. The .NET framework contains the necessary infrastructure to deal with this deficiency. Surprise, surprise, it involves another interface – *IBindingList*. This interface contains nine properties, six methods and an event. That's quite a lot, but fortunately, I don't have to worry about most of it. The only items I need to implement are the *ListChanged* event property and the *get_SupportsChangeNotification* method, for the *SupportsChangeNotification* property. I can write one-line implementations of all the other methods and property getters (property getters will return *False* or raise a *NotSupportedException*, methods will either be empty or raise a *NotSupportedException*). I also have to call the event handler (if any) when the list changes. I can use more of the events in the *CollectionBase* class for this, specifically *OnClearComplete*, *OnInsertComplete*, *OnRemoveComplete* and *OnSetComplete*, as well as my *Sort* methods. Brilliant! The grid updates when I add, delete or change a Customer.

There's one final bit of functionality I want to add to my collection. I want to be able to use a grid to add Customers to the list. To make this work, I need to make the *IBindingList* properties *AllowEdit*, *AllowNew* and *AllowRemove* return *True* and I need to implement the *AddNew* method to create a new Customer and add it to the list. The complete *IBindingList*, the overridden event routines and the modified *Sort* methods are in Listing 15.

RAIZE SOFTWARE

CodeSite

When your code isn't doing what you expect, find out why with CodeSite



- New for CodeSite 3...**
- CodeSite Projects
 - Automatic Message Filtering
 - Automatic Views
 - XML Support
 - .NET Support
 - Custom Views
 - Enhanced Thread Support
 - Plus much more...

CodeSite helps developers locate problems in their code by enabling them to send detailed information from within their application code to a specialized receiver.

Send Any Information

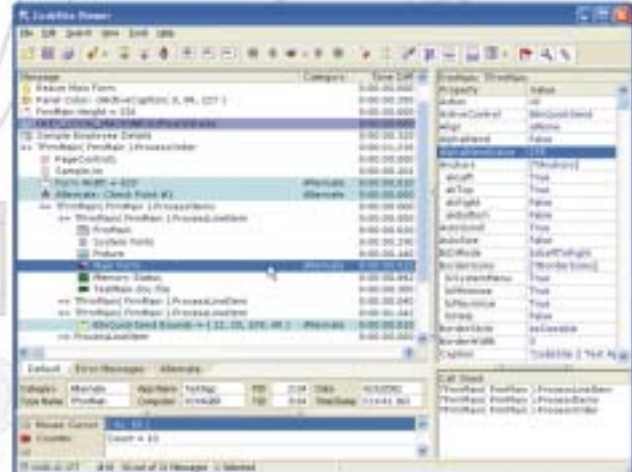
CodeSite supports sending snapshots of all kinds of data, including all standard types, objects, string lists, variants, bitmaps, files, memory statistics, registry entries, custom data, and much more.

Receive the Messages

CodeSite messages can be sent to a variety of destinations including the CodeSite Viewer, a log file, a remote computer, and even a web server.

Analyze the Messages

The CodeSite Viewer (screen shot) provides a powerful and highly-flexible environment for analyzing CodeSite messages.



Raize DropMaster

OLE Drag-and-Drop made easy

Simply Powerful

DropMaster encapsulates the complexity of OLE inter-application drag-and-drop in 4 easy-to-use components.



Drag Sources

Easily support dragging text, images, or custom data formats to other applications.

Drop Targets

Accept plain text, rich text, file lists, URL links, images, and custom data formats from other applications that supports OLE drag and drop.

Customize the Process

Special events are provided so that the drag and drop process can be customized.

Real World Examples

DropMaster comes with more than 30 example projects that demonstrate the extreme power and flexibility of these components.

Examples Include:

- Accept messages dragged from Outlook, Outlook Express, Netscape, and Eudora.
- Dragging URLs from an application to a browser or the desktop.
- Dragging multiple custom formats
- Dragging and creating multiple files
- Accepting OLE object links
- Dragging JPEG images
- Dragging custom content to Explorer folders or the desktop

Raize Components

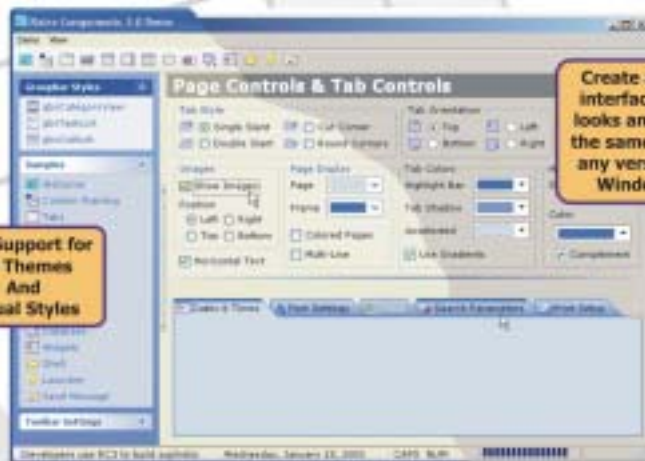
Build sophisticated user interfaces in less time with less effort

User Interface Design System
Raize Components includes **more than 120** general-purpose native VCL controls for use in Delphi and C++Builder.

Powerful and Easy to Use
RC3 also includes **more than 100** component designers focused on simplifying user interface development.

Component Innovations
Raize Components also features one-step installation, automatic help integration, and dynamic component registration.

Source Code Included
Complete source code for all components, packages, and design editors is provided at no additional charge.



Full Support for XP Themes And Visual Styles

Create a user interface that looks and feels the same under any version of Windows.

www.raize.com

```

public class csCustomerList: CollectionBase, IBindingList, IEnumerable
{
    // IBindingList members
    public event System.ComponentModel.ListChangedEventHandler ListChanged;

    public void AddIndex(PropertyDescriptor property) { // Do nothing }

    public bool AllowNew
    {
        get { return true; }
    }

    public void ApplySort(PropertyDescriptor property,
        System.ComponentModel.ListSortDirection direction)
    {
        throw new NotSupportedException();
    }

    public PropertyDescriptor SortProperty
    {
        get { throw new NotSupportedException(); }
    }

    public int Find(PropertyDescriptor property, object key)
    {
        throw new NotSupportedException();
    }

    public bool SupportsSorting
    {
        get { return false; }
    }

    public bool IsSorted
    {
        get { return false; }
    }

    public bool AllowRemove
    {
        get { return true; }
    }

    public bool SupportsSearching
    {
        get { return false; }
    }

    public System.ComponentModel.ListSortDirection SortDirection
    {
        get { throw new NotSupportedException(); }
    }

    public bool SupportsChangeNotification
    {
        get { return true; }
    }

    public void RemoveSort(){ throw new NotSupportedException(); }

    public object AddNew()
    {
        csCustomer l_newCustomer = new csCustomer( this );
        ( ( IList )this ).Add( l_newCustomer );
        return l_newCustomer;
    }

    public bool AllowEdit
    {
        get { return true; }
    }

    public void RemoveIndex(PropertyDescriptor property)
    {

```

```

    // Do nothing
}

// Helper function for event handling
private void OnListChanged( object p_sender, ListChangedEventArgs p_eventArgs )
{
    if ( ListChanged != null )
    {
        ListChanged( p_sender, p_eventArgs );
    }
}

// Overridden CollectionBase events
protected override void OnValidate(object value)
{
    if ( ! ( value is ICustomer ) )
    {
        throw new ArgumentException( "You may only put an ICustomer in a csCustomerList" );
    }
}

protected override void OnClearComplete()
{
    OnListChanged( this, new ListChangedEventArgs( ListChangedType.Reset, 0 ) );
}

protected override void OnInsertComplete(int index, object value)
{
    OnListChanged( this, new ListChangedEventArgs( ListChangedType.ItemAdded, index ) );
}

protected override void OnRemoveComplete(int index, object value)
{
    OnListChanged( this, new ListChangedEventArgs( ListChangedType.ItemDeleted, index ) );
}

protected override void OnSetComplete(int index, object oldValue, object newValue)
{
    OnListChanged( this, new ListChangedEventArgs( ListChangedType.ItemChanged, index ) );
}

// Modified Sort methods
public void Sort()
{
    this.InnerList.Sort();
    OnListChanged( this, new ListChangedEventArgs( ListChangedType.Reset, 0 ) );
}

public void Reverse()
{
    this.InnerList.Reverse();
    OnListChanged( this, new ListChangedEventArgs( ListChangedType.Reset, 0 ) );
}

public void Sort( IComparer p_comparer )
{
    this.InnerList.Sort( p_comparer );
    OnListChanged( this, new ListChangedEventArgs( ListChangedType.Reset, 0 ) );
}

```

Listing 14 – implement most of the *IBindingList* interface, with events, for editing in grids

I also need to make the *csCustomer* class implement the *IEditableObject* interface (it's the last one, I promise!). *IEditableObject* has three methods – *BeginEdit*, *CancelEdit* and *EndEdit*. When you add a new record to the end of a *DataGrid*, the grid calls the *AddNew* method of the *IBindingList* and then calls *BeginEdit* on the resulting object (in this case a new *csCustomer*). When you've finished editing the new object, the grid calls *EndEdit*. If, however, you decide you didn't want a new object you press the *Escape* key, the grid calls the *CancelEdit* method and everything should return to its previous, peaceful state. The *AddNew* method, however, has already added the new object to the list, so I have to put code in the *CancelEdit* method that removes it from the list if it's just been added. I set a flag in the constructor that tells me I've just created the *Customer* and clear the flag in the *EndEdit* method (Listing 15). A consequence of this is that I have to call *EndEdit* on the *Customer* if I create it in code, rather than letting the grid create it, otherwise if I navigate to the new *Customer* and press *Escape* it will disappear!

```

public class csCustomer: ICustomer, IComparable, IEditableObject
{
    private IList _owner;
    private bool _newCustomer;
    private csCustomer _backupData;
    private bool _editing;

    // IEditableObject members
    void IEditableObject.BeginEdit()
    {
        if ( !_editing )
        {
            if ( _backupData == null )
            {
                _backupData = new csCustomer( null );
            }
            _backupData._title = _title;
            _backupData._firstName = _firstName;
            _backupData._lastName = _lastName;
            _editing = true;
        }
    }

    void IEditableObject.CancelEdit()
    {
        if ( _editing )
        {
            if ( _backupData != null )
            {
                _title = _backupData._title;
                _firstName = _backupData._firstName;
                _lastName = _backupData._lastName;
            }
            if ( _newCustomer )
            {
                _owner.Remove( this );
            }
            _editing = false;
        }
    }

    void IEditableObject.EndEdit()
    {
        _newCustomer = false;
        _editing = false;
        if ( IsChanged() )
        {
            // Trigger the OnSetComplete event to update controls
            int l_position = _owner.IndexOf( this );
            _owner[ l_position ] = this;
        }
    }
}

```

Listing 15 – implementing the *IEditableObject* interface

Making life easier

Phew! I've achieved my goal of a collection that I can display in grids, list boxes, combo boxes, etc. I needed to do quite a bit of work, however, to get there. If I have a lot of collections to write, I'd soon get tired of cutting & pasting all that code, not to mention any issues with maintainability. That indicates to me that I can make a base collection class that will do a lot of the work, reducing the number of methods and properties I have to write. I'm also keen to try out the new *Generics* feature of .NET version 2. I'm sure it should be possible to write a generic collection class with only one set of code. This article is far too long already, however, so I'll leave that for a future edition.

Summary

I began with the aim of writing a collection of objects that I can use with standard .NET controls. I wrote a *csCustomer* class and a *csCustomerList* class, both implementing .NET interfaces. The end result is a collection that can be assigned to the *DataSource* property of various list controls, that can be edited with a standard *DataGrid* and notifies the controls about changes so they update themselves automatically.

The interfaces I implemented are:

- IList* – provides methods to manage the list and the objects inside it. Objects that implement this interface can be displayed by standard .NET controls. Implemented by the collection.
- IComparable* – provides a method to compare an object with another one of the same type. Used by the *ArrayList Sort* method. Implemented by the class that the collection contains.
- IComparer* – provides a method that compares two objects of the same type. Used by the *ArrayList Sort* method. Implemented by a helper class.
- IEnumerable* – provides a factory method that returns an enumerator for this object. Used by *foreach*. Implemented by the collection.
- IEnumerator* – provides methods to enumerate (aka. iterate) a list. Can also be used for list filtering. Used by *foreach*. Implemented by a separate class.
- IBindingList* – provides more methods to manage a list. Used by *DataGrid* to edit the list and its elements. Implemented by the collection.
- IEditableObject* – provides commit and rollback to object editing. Used by *DataGrid* when editing elements in a list. Implemented by the class that the collection contains.

Other interfaces to look at are:

- IComponent* – provides a method and a property for use with components. Used by Visual Studio to add the class to the toolbox and allow dropping it onto forms. Can be implemented by both the collection and the class it contains.
- IListSource* – provides methods that allow the .NET framework to find out about the list and the objects it contains. Used by grids at design time. Implemented by the collection.
- ITypedList* – provides methods to get details of the properties of the objects in the list. Used by grids at design time. Implemented by the collection.

To learn more about iterators and factories, read *Design Patterns* by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, published by Addison Wesley. There have also been a number of articles in the pages of this esteemed magazine, *The Delphi Magazine* and others.

Craig Murphy asked that we should mention useful tools. I found *Reflector for .NET* extremely useful for this article. It takes a .NET assembly, shows you what's in it and will put the code out as IL, C#, VB.NET or Delphi (yes, Delphi!). It's free and you can get it from <http://www.aisto.com/roeder/dotnet>.

The Source Code

There's some sample source with this article [see our downloads page on <http://www.richplum.co.uk/downloads/default.asp> . Ed], both Delphi and C#. They're not the same, because I started to write in Delphi, but when I added *IBindingList*, Delphi insisted that I had to implement *IList* and *ICollection* as well, even though they're implemented in the base class.



Both versions of the code include a form with a grid and a list box. You can see that the list box and the grid keep themselves synchronised, so moving to another Customer on the grid moves to that Customer on the list box as well. The C# version also has an enumeration list box that is filled with a *foreach* loop. Put one or more characters in the text box to see only Customers whose Last Name begins with what you've typed.

Rob Bracken is a freelance Delphi and C# developer based near Bristol, U.K. You can contact him at bug@brackensoftware.co.uk.

the silly bit the silly

IMPORTANT: This email is intended for the use of the individual addressee(s) named above and may contain information that is confidential, privileged or unsuitable for overly sensitive persons with low self-esteem, no sense of humour or irrational beliefs. If you are not the intended recipient, any dissemination, distribution or copying of this email is not authorised (either explicitly or implicitly) and constitutes an irritating social faux pas. Unless the word canonicalisation has been used in its correct context somewhere other than in this warning, it does not have any legal or grammatical use and may be ignored. No children or animals were harmed in the transmission of this email, although ghastly neighbours' cats are living on borrowed time. Those with an overwhelming fear of the unknown will be gratified to learn that there is no hidden message to be revealed by reading this warning backwards.