

Komponenten richtig einsetzen

Vielseitige Kleinteile

Komponenten machen das Leben leichter. Wer genau weiß, wie man sie richtig verwendet, profitiert am meisten von ihrem Einsatz.

Eigenschaften einer Komponente als expandierende Unterelemente anzeigen

Wenn die Eigenschaft einer Komponente ein benutzerdefinierter Typ ist, bietet es sich an, diesen im Eigenschaftsfenster als expandierenden Eintrag darzustellen. Auf diese Weise kann der Benutzer die Eigenschaften des Unterelements direkt manipulieren. Ein Beispiel:

```
using System;
using System.ComponentModel;
public class MyComponent :
    System.ComponentModel.Component {
    private Customer m_customer;
    public MyComponent(){
        m_customer = new Customer();
    }
    [DesignerSerializationVisibility(
        DesignerSerializationVisibility.Content)]
    public Customer Customer {
        get{return m_customer;}
        set{m_customer = value;}
    }
}
```

Damit für den Inhalt der *Customer*-Klasse später auch entsprechender Code in der *InitializeComponent*-Methode des Containers erzeugt wird, muss die Eigenschaft mit dem *DesignerSerializationVisibility*-Attribut ausgestattet werden und dies mit dem Wert *DesignerSerializationVisibility.Content* belegt werden.

Die *Customer*-Klasse sieht nun wie folgt aus:

```
[TypeConverter(typeof(CustomerConverter))]
public class Customer {
    private string m_firstName = string.Empty;
    private string m_lastName = string.Empty;
    public Customer() {
    }
    public string FirstName {
        get{return m_firstName;}
        set{m_firstName = value;}
    }
    public string LastName {
        get{return m_lastName;}
    }
}
```

```
set(m_lastName = value;)
}
}
```

Über das *TypeConverter*-Attribut der Klasse wird der Entwurfszeitinfrastruktur mitgeteilt, wie die Objekte dieses Typs im Eigenschaftsfenster dargestellt werden sollen und was beim Konvertieren zu beachten ist.

Die Implementierung von *CustomerConverter* sieht nun wie folgt aus:

```
internal class CustomerConverter :
    ExpandableObjectConverter {
    public override bool
        GetCreateInstanceSupported(
            ITypeDescriptorContext context) {
        return true;
    }
    public override object CreateInstance(
        ITypeDescriptorContext context,
        IDictionary propertyValues) {
        Customer cust = new Customer();
        cust.FirstName =
            (string)propertyValues["FirstName"];
        cust.LastName =
            (string)propertyValues["LastName"];
        return cust;
    }
}
```

Die Klasse leitet von *ExpandableObjectConverter* ab, einem *TypeConverter*, der im .NET Framework enthalten ist und dafür sorgt, dass die Objekte im Eigenschaftsfenster expandiert dargestellt werden. Hier muss nun lediglich durch Überschreiben der *GetCreateInstanceSupported*-Methode signalisiert werden, dass die Instanzierung der Objekte von Hand geschieht. In *CreateInstance* findet daraufhin die Objekterstellung statt. Hier werden die entsprechenden Eigenschaften aus der übergebenen *propertyValues*-Auflistung ausgelesen und der neuen Instanz zugewiesen.

Im Eingabefeld des Eigenschaftsfensters erscheint nun als Text der Typname. Soll hier ein anderer Text angezeigt werden, so ist die *ToString*-Methode der *Customer*-Klasse zu überschreiben. Dies

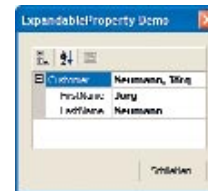


Abbildung 1 Die Customer-Eigenschaft wird im PropertyGrid als expandierendes Objekt angezeigt.

könnte beispielsweise so aussehen:

```
public override string ToString() {
    if (m_firstName.Length > 0 &&
        m_lastName.Length > 0)
        return m_lastName + ", " + m_firstName;
    return "(Kein)";
}
```

Das Ergebnis der Arbeit sehen Sie in **Abbildung 1**.

Handelt es sich bei der Eigenschaft um einen Komponententyp, bietet es sich an, im Eigenschaftsfenster zusätzlich eine *ComboBox* anzuzeigen, aus der der Benutzer eine vorhandene Komponenteninstanz auswählen kann. Hierfür ist die Eigenschaft mit dem *TypeConverter ComponentConverter* zu versehen. Auf die Anlage eines *TypeConverters* kann in diesem Fall verzichtet werden. (jn)

► **Beispielprojekt** ExpandablePropertyDemo.zip

Eigenschaften einer Komponente zur Entwurfszeit verborgen

Möchte man verhindern, dass eine Eigenschaft zur Entwurfszeit im Eigenschaftsfenster erscheint, so ist die Eigenschaft mit dem *Browsable*-Attribut auszustatten und das Attribut auf *false* zu setzen.

Soll darüber hinaus auch verhindert werden, dass in der *InitializeComponent*-Methode Code für die Eigenschaft erzeugt wird, so ist zusätzlich die Deklaration des *DesignerSerializationVisibility*-

Attribute-Attributs nötig, das den Wert *DesignerSerializationVisibility.Hidden* bekommt. Beispiel:

```
[Browsable(false),
DesignerSerializationVisibilityAttribute(DesignerSerializationVisibility.Hidden)]
public string MyProperty {
...
}
```

Darüber hinaus ist auch ein Verbergen der Eigenschaft in der Code-Ansicht möglich. Hierfür ist das *EditorBrowsable*-Attribut verantwortlich.

```
[EditorBrowsable(EditorBrowsableState.Never)]
```

Dies verhindert, dass die Eigenschaft beim Codieren in der IntelliSense-Liste angezeigt wird. (jn)

Komponenten zur Entwurfszeit verbergen

Um zu verhindern, dass eine Komponente in die Toolbox aufgenommen werden kann, ist diese mit dem *ToolboxItem*-Attribut auszustatten und dieses auf *false* zu setzen.

Außerdem kann verhindert werden, dass eine Komponente im Komponentenbereich des *Windows.Forms*-Designers erscheint. Hierfür ist das *DesignTimeVisible*-Attribut zuständig. Beispiel:

```
[ToolboxItem(false),
DesignTimeVisible(false)]
public class MyComponent : Component {
...
}
(jn)
```

Komponente mit einem Symbol ausstatten

Möchte man seine Komponente mit einem eigenen Symbol ausstatten, das in der Toolbox erscheint, so ist über der Klasse das *ToolboxBitmap*-Attribut zu deklarieren. Beispiel:

```
[System.Drawing.ToolboxBitmapAttribute(
typeof(MyComponent), "MyComponent.ico")]
public class MyComponent : Component {
...
}
```

Diesem Attribut ist der Typ der Komponente sowie der Name der Icon-Datei zu übergeben. Zu beachten ist, dass sich die Datei im gleichen Verzeichnis wie die



Abbildung 2 Das Bearbeiten der Elemente im Collection Editor.

Komponente befindet und diese ins Projekt aufgenommen wurde. Hierbei ist die Build-Aktion der Datei auf *Eingebettete Ressource* zu setzen. (jn)

Collection-Eigenschaften bearbeiten

Wenn einer Eigenschaft eine Liste von Werten zugewiesen werden soll, so bietet es sich an, hierfür eine eigene *Collection*-Klasse zu erstellen und mit einem Editor zu verknüpfen, der die Bearbeitung zur Entwurfszeit ermöglicht, wie es Abbildung 2 beispielhaft zeigt.

Das folgende Beispiel erstellt eine von *CollectionBase* abgeleitete Klasse, die *Parameter*-Objekte verwaltet. Sie können Listing 1 als Vorlage für Ihre eigene *Collection* verwenden, indem Sie einfach *Parameter* durch den Namen Ihres Objekts ersetzen.

Ein *Parameter*-Objekt könnte nun beispielsweise so aussehen:

```
using System;
using System.ComponentModel;
public class Parameter : Component {
private string m_parameterName =
string.Empty;
public Parameter() {
}
public string ParameterName {
get{return m_parameterName;}
set{m_parameterName = value;}
}
}
```

Die Ableitung von *Component* ist nicht zwingend erforderlich. Andernfalls ist jedoch eine Verknüpfung mit einer eigenen *TypeConverter*-Klasse nötig.

Was nun noch fehlt, ist die Komponente, die die *Collection* als Eigenschaft definiert. Hier findet nun auch die Verknüpfung mit dem Editor über das gleichnamige Attribut statt. Zusätzlich ist das *DesignerSerializationVisibility*-Attri-

Listing 1

Eine einfache Collection-Klasse.

```
using System;
using System.Collections;
public class ParameterCollection :
CollectionBase {
public ParameterCollection() : base() {
}

public Parameter this[int index] {
get{return (Parameter)List[index];}
}
public Parameter this[string parameterName]
{
get {
for (int i = 0; i < List.Count; i++)
{
if (((Parameter)List[i]).
ParameterName == parameterName)
return (Parameter)List[i];
}
return null;
}
}
public IList Values{
get {return List;}
}
public int Add(Parameter value) {
return List.Add(value);
}
public bool Contains(Parameter value) {
return List.Contains(value);
}
public void Remove(Parameter value) {
List.Remove(value);
}
}
}
```

but zu setzen, damit die erstellten Objekte auch in den Code serialisiert werden.

```
using System;
using System.ComponentModel;
using System.Drawing.Design;
public class MyComponent : System.ComponentModel.
Component {
private ParameterCollection m_parameters =
new ParameterCollection();
public MyComponent() {
}
[Editor(typeof(System.ComponentModel.Design.Co
llectionEditor),typeof(UITypeEditor)),
DesignerSerializationVisibility(
DesignerSerializationVisibility.Content)]
public ParameterCollection Parameters {
get{return m_parameters;}
set{m_parameters = value;}
}135
}
(jn)
```

► Beispielprojekt *CollectionDemo.zip*