18 Zwischenablage und Drag&Drop

Im Grunde genommen sind die Zwischenablage und Drag&Drop nichts weiter als zwei Gesichter ein und derselben Technologie, die auf den Namen OLE hört. Beide Mechanismen ermöglichen den Datentransfer von einem *Quellobjekt* zu einem *Zielobjekt* (im Falle der Zwischenablage sogar zu beliebig vielen Zielobjekten), wobei Quellobjekt und Zielobjekt auch unterschiedlichen Anwendungen angehören dürfen.

Dass der Datentransfer auch die Prozessgrenze und womöglich sogar die Systemgrenze überschreiten kann, davon merken Sie als Programmierer nichts. Einzig der Handshake, das heißt die Verständigung über das gewünschte Datenformat (als Tribut an das Typsystem) und das Angeben einer Operation sind vielleicht etwas ungewohnt.

18.1 Zwischenablage

Die Zwischenablage ist in der .NET-Klassenhierarchie über die direkt von `System.Object` abstammende, versiegelte und nicht instanziierbare `Clipboard`-Klasse repräsentiert.

18.1.1 Steuerelemente mit eigener Zwischenablagenfunktionalität

Bei Experimenten mit der Zwischenablage sollten Sie im Hinterkopf behalten, dass viele Steuerelemente, beispielsweise die von der Klasse `TextBoxBase` abgeleiteten Steuerelemente `TextBox` und `RichTextBox`, bereits von sich aus eine Programmierschnittstelle oder zumindest eine Tastaturschnittstelle für die Zwischenablage mitbringen. Die Implementierung einer Menü- bzw. Symbolleistenschnittstelle für die Zwischenablagenfunktionalität kann daher vom Prinzip her direkt mit den entsprechenden Methoden arbeiten. Kommt als Quell- oder Zielobjekt nur ein Textfeld `textBox1` in Frage, könnte die Behandlung der Menübefehle schlicht so aussehen:

```
#if Variante1
private void menuCopy_Click(object sender, System.EventArgs e)
{
  textBox1.Copy();
}
```

```
private void menuCut_Click(object sender, System.EventArgs e)
{
  textBox1.Cut();
}
private void menuPaste_Click(object sender, System.EventArgs e)
{
  textBox1.Paste();
}
#endif
```

Kommen mehrere Textfelder als Quelle oder Ziel in Frage, müsste man das betroffene Steuerelement irgendwie ausfindig machen und beispielsweise nach dem folgenden Muster ansprechen:

```
#if Variante2
private void menuCopy_Click(object sender, System.EventArgs e)
{
  try
  {
    TextBoxBase tbb = (TextBoxBase)ActiveControl;
    tbb.Copy();
  }
  catch  {}
}
...
#endif
```

Der Königsweg ist allerdings die folgende Formulierung:

```
#if Variante3
private void menuCopy_Click(object sender, System.EventArgs e)
{
  SendKeys.Send("^C");
}
private void menuCut_Click(object sender, System.EventArgs e)
{
  SendKeys.Send("^X");
}
private void menuPaste_Click(object sender, System.EventArgs e)
{
  SendKeys.Send("^V");
}
#endif
```

Der Code schickt mittels `SendKeys.Send()` eine Tastatureingabe (Strg+C, Strg+X, Strg+V) an das Steuerelement, das den Fokus besitzt, und überlässt alles Weitere dem Steuerelementobjekt. (Die syntaktischen Details, wie das `string`-Argument für `Send()` aussehen muss, finden Sie in guter Darstellung in der Online-Hilfe). Wenn dieses Zwischenablagenfunktionalität implementiert, verarbeitet es das Ereignis, ansonsten eben nicht. Auf diese Weise können Sie alle Steuerelemente erreichen, die von sich aus mit der Zwischenablage zusammenarbeiten, beispielsweise also auch Listen.

Der soeben vorgestellte Code findet sich in dem Projekt *Zwischenablage*. Sie können damit – wohlgemerkt über die Menüschnittstelle – Text von Textfeld zu Textfeld sowohl innerhalb einer Anwendung als auch anwendungsübergreifend austauschen. Die dritte Variante bezieht auch das auf dem Formular befindliche `NumericUpDown`-Steuerelement mit ein.

Um zwischen den einzelnen Varianten umzuschalten, passen Sie die `#define`-Anweisung in der ersten Zeile des Codes an.

18.1.2 Die Zwischenablage direkt ansprechen

In nicht allen Fällen lässt sich die tatsächliche Interaktion mit der Zwischenablage an ein Steuerelement delegieren. Oft müssen spezifische Inhalte gewissermaßen »manuell« in die Zwischenablage einlagert bzw. daraus entnommen werden, beispielsweise, wenn es um die Implementierung der Zwischenablagenfunktionalität für eigene Steuerelement- oder Formularklassen geht – und erst recht, wenn komplexe Inhalte, etwa ein Datensatz, über mehrere Steuerelemente verteilt bzw. aufzuteilen sind.

Die Klasse `Clipboard` stellt nur zwei statische Methoden bereit, `SetDataObject()` und `GetDataObject()`, deren Bezeichner schon fast eine klarere Sprache sprechen als ihre Prototypen:

```
public static void SetDataObject(object data);
public static void SetDataObject(object data, bool copy);
public static IDataObject GetDataObject();
```

Tabelle 18.1: Methoden der Schnittstelle `DataObject`

Methode	**Bedeutung**
`object GetData()`	Liefert die Daten aus dem Objekt in dem gewünschten Datenformat bzw. Datentyp. Falls das Datenformat nicht verfügbar ist bzw. eine Konvertierung in den gewünschten Datentyp nicht möglich ist, ist der Rückgabewert `null`.
`bool GetDataPresent()`	Liefert eine Aussage darüber, ob die von dem Objekt repräsentierten Daten in dem gewünschten Datenformat bzw. Datentyp verfügbar sind.
`string[] GetFormats()`	Liefert eine Auflistung der Datenformate, in denen das Objekt den repräsentierten Wert liefern kann.
`void SetData()`	Lagert Daten (in Form eines Objekts) in das Objekt ein. Diese Methode existiert in vier überladenen Varianten. Die einparametrige Variante übernimmt das angegebene Objekt und benutzt dessen Typinformation. Die beiden zweiparametrigen Varianten ermöglichen zusätzlich die explizite Angabe eines Datenformats als `string`-Wert oder eines Datentyps als `Type`-Wert. Die dreiparametrige Variante erlaubt weiterhin die Angabe eines `bool`-Werts, der ausdrückt, ob eine Konvertierung in kompatible Formate zulässig ist.

Daten einlagern – SetDataObject()

Um etwas in die Zwischenablage hineinzustecken, rufen Sie die Methode `SetDataObject()` auf und übergeben dieser ein Objekt, das vom Prinzip her einen beliebigen (!) – also auch eigenen Datentyp – tragen darf, solange dieser serialisierbar ist. Wenn Sie wollen, dass dieses Objekt auch nach Beendigung Ihrer Anwendung noch erhalten bleibt, verwenden Sie die zweiparametrige Überladung und versorgen den zweiten Parameter mit `true`. (Vielleicht fühlen Sie sich nun an die Rückfragen einiger Programme wie Photoshop oder Corel Draw erinnert, die beim Schließen einen entsprechenden Dialog hervorbringen.) Ist der zweite Parameter `false` oder fehlt er, wird das in die Zwischenablage übertragene Objekt bei Programmende mit abgebaut – und die Zwischenablage geleert.

Das wirft natürlich sofort die Frage auf, ob die Zwischenablage nun eine Referenz oder einen Klon des Objekts speichert. Die Antwort dürfte nicht überraschen. Es ist beides möglich.

Die Zwischenablage hat somit zwei Operationsmodi:

- *frühe Wertbindung* – in diesem Modus (zweiter Parameter des `SetDataObject()`-Aufrufs ist `true`) generiert `SetDataObject()` einen echten Klon, so dass `GetDataObject()` das Objekt in dem Zustand liefert, wie es eingelagert wurde.
- *späte Wertbindung* – in diesem Modus (zweiter Parameter des `SetDataObject()`-Aufrufs fehlt oder ist `false`) lagert `SetDataObject()` bei einem Verweistyp nur eine Referenz auf das Objekt ein, so dass `GetDataObject()` das Objekt in seinem jeweils aktuellen Zustand liefert. Werttypen werden hingegen kopiert. (Natürlich ist `string` mal wieder die rühmliche Ausnahme, weil letztlich ja die Implementierung der Zuweisungsoperation für den jeweiligen Datentyp darüber entscheidet, ob ein Klon oder eine Referenz bei der Zuweisung herauskommt.)

Der Aufruf von `SetDataObject()` für Textfelder sieht so aus.

```
private void menuClone_Click(object sender, System.EventArgs e)
{
  menuClone.Checked = !menuClone.Checked;
}

#if Variante4
private void menuCopy_Click(object sender, System.EventArgs e)
{
  Control c = ActiveControl;
  Clipboard.SetDataObject(c.Text.ToUpper(), menuClone.Checked);
}
```

```
private void menuCut_Click(object sender, System.EventArgs e)
{
  Control c = ActiveControl;
  Clipboard.SetDataObject(c.Text.ToUpper(), menuClone.Checked);
  c.Text = "";
}
```

Die `Checked`-Eigenschaft des Menüeintrags `menuClone` gibt den Operationsmodus vor, der angesichts des eingelagerten `string`-Werts allerdings nur beim Programmende Wirkung entfaltet (String-Werte werden ja grundsätzlich kopiert). Da der Code mit der `Text`-Eigenschaft des Steuerelements mit dem Eingabefokus arbeitet, bleibt das `NumericUpDown`-Steuerelement außen vor.

GetDataObject()

Um etwas aus der Zwischenablage herauszuholen, rufen Sie die Methode `GetDataObject()` auf. Sie liefert eine Instanz der Schnittstelle `IDataObject`, die ihrerseits drei in diesem Zusammenhang wichtige Methoden bereitstellt (Tabelle 18.1). `GetFormats()` vermittelt einen Überblick über die verfügbaren Datenformate und Datentypen, wobei unter »Datenformat« eine Familie von Datentypen zu verstehen ist. Die Klasse `DataFormats` definiert eine Reihe von `string`-Konstanten für Datenformate, die das von der Zwischenablage standardmäßig für die Einlagerung von Objekten verwendete Container-Objekt des Typs `DataObject` von sich aus versteht.

Die direkt von `object` *abstammende Klasse* `DataObject` *implementiert die Schnittstelle* `IDataObject` *und steht vom Prinzip her als Basisklasse für die Implementierung eigener Datenformate und Containerkonzepte (beispielsweise für die komprimierte Serialisierung) zur Verfügung.*

Häufig anzutreffende `DataFormats`-Konstanten sind: `Text`, `FileDrop`, `Bitmap`, `Html`, `Rtf`, `Dib`, `Tiff`, `EnhancedMetaFile`, `CommaSeparatedValue`, `UniCodeText` und `WaveAudio`.

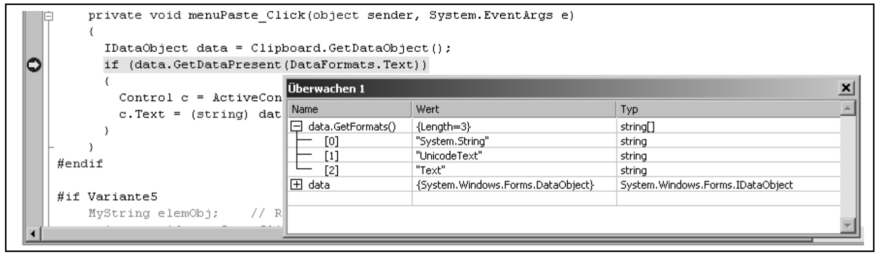

Abbildung 18.1: Ergebnis von `GetFormats()` für das Datenformat `Text`, nachdem zuvor ein String in die Zwischenablage kopiert wurde.

Damit Sie eine Vorstellung davon erhalten, was die `GetFormats()`-Methode liefert, setzen Sie auf die `if`-Anweisung des folgenden Codes einen Haltepunkt und sehen sich das `data`-Objekt in einem Überwachungsfenster des Debuggers einmal genauer an – freilich erst, nachdem Sie etwas in die Zwischenablage hingepackt haben (Abbildung 18.1 und Abbildung 18.2).

Abbildung 18.2: Ergebnis von `GetFormats()` für das Datenformat `FileDrop`, nachdem zuvor einige Dateien über den Windows-Explorer in die Zwischenablage kopiert wurden.

```
private void menuPaste_Click(object sender, System.EventArgs e)
{
  IDataObject data = Clipboard.GetDataObject();
  if (data.GetDataPresent(DataFormats.Text))
  {
    Control c = ActiveControl;
    c.Text = (string) data.GetData(DataFormats.Text);
  }
}
#endif
```

Nachdem ein Zielobjekt im Allgemeinen nicht wissen kann, welche Inhalte und Datenformate (bzw. Datentypen) das Quellobjekt in die Zwischenablage gepackt hat, muss es sich zuerst darüber informieren, inwieweit das erwartete Datenformat überhaupt verfügbar ist. Falls das Objekt ohnehin nur mit einem bestimmten Datenformat bzw. Datentyp etwas anfangen kann, können Sie sich auch den `GetFormats()`-Aufruf und die Auswertung des Ergebnisses sparen und stattdessen gleich mit der `GetDataPresent()`-Methode prüfen, ob das `IDataObject`-Objekt die Daten in dem gewünschten Datenformat oder Datentyp bereitstellen kann.

Referenztypen und eigene Datentypen

Wie bereits erwähnt, schluckt die Zwischenablage auch eigene Datentypen, sofern diese serialisierbar sind. Die folgende, fünfte Variante des Codes demonstriert die frühe und die späte Wertbindung für Referenztypen am Beispiel eines eigenen `class`-Datentyps. Anstatt der verwendeten Klasse `MyString`, deren Definition Sie gleich im nächsten Abschnitt finden, können Sie alles Mögliche in ein `DataObject`-Objekt packen, angefangen von `string`-Arrays bis hin zu Auflistungen beliebiger Objekte mit serialisierbaren Datentypen – auch wenn diese Datentypen ihrerseits wieder Auflistungen sind.

```
#using ClipboardClasses
#if Variante5
MyString elemObj;    // Referenztyp
private void menuCopy_Click(object sender, System.EventArgs e)
{
  if(elemObj == null)   // instanziiert?
  {
    elemObj = new MyString();
```

```
    menuCut.Text = "&Ref-Wert ändern";
  }
  elemObj.Elem = ActiveControl.Text;
Clipboard.SetDataObject(elemObj, menuClone.Checked);
}
// Achtung: veränderte Semantik. Ändert Wert des Reftyps elemObj,
// was ggf. den Inhalt der Zwischenablage ändert
private void menuCut_Click(object sender, System.EventArgs e)
{
  if (menuCut.Text == "&Ausschneiden")
  {
    elemObj = new MyString();
    menuCut.Text = "&Ref-Wert ändern"
    elemObj.Elem = ActiveControl.Text;
    Clipboard.SetDataObject(elemObj, menuClone.Checked);
    ActiveControl.Text = "";
  }
  else
    elemObj.Elem = ActiveControl.Text;
}
private void menuPaste_Click(object sender, System.EventArgs e)
{
  IDataObject data = Clipboard.GetDataObject();
  if (data.GetDataPresent(typeof(MyString)))
  {
    MyString ms = (MyString) data.GetData(typeof(MyString));
    ActiveControl.Text = ms.Elem;
    int ten = ms.Len;
  }
}
```

Eigene Datentypen anwendungsübergreifend verwenden

Für den anwendungsinternen Bedarf kann `MyString` vom Prinzip her auch ein privater Datentyp der Klasse `Form1` sein. Wenn Sie einen Datenaustausch mit anderen Anwendungen durchführen wollen, brauchen Sie auf jeden Fall eine eigene Assembly mit dem Ausgabetyp *Klassenbibliothek* (*.dll*). Dazu gehen Sie wie folgt vor:

1. Fügen Sie der Projektmappe (einer der Anwendungen) ein eigenständiges Projekt mit dem Typ *Klassenbibliothek* hinzu und definieren Sie darin alle Klassen, deren Objekte Sie anwendungsübergreifend per Zwischenablage (oder Drag&Drop) transferieren wollen. Kompilieren Sie das Projekt, um die Assembly der Klassenbibliothek zu erhalten.

2. Fügen Sie in jede Anwendung, die von diesen Datentypen Gebrauch macht, einen Verweis auf die Assembly ein. Eine Anleitung, wie Sie dies machen und welche Möglichkeiten Sie dabei haben (Stichworte: »starke Namen« und »Assembly Cache«), finden Sie beispielsweise im Abschnitt »Eine Steuerelementbibliothek in eine bestehende Projektmappe einfügen« (Seite 666).

Von nun an können Sie die Datentypen in allen Projekten verwenden und nach Herzenslust über die Zwischenablage transferieren. Um die importierten Datentypen ohne Namensraumzusätze ansprechen zu können, empfiehlt sich eine `using`-Direktive für den jeweiligen Namensraum. Hier die noch ausstehende Definition der Klasse `MyString`. Beachten Sie bitte das `Serializable`-Attribut:

```
using System;
namespace ClipboardClasses
{
  [Serializable]
  public class MyString
  {
    public string Elem = "";
    public int Len  // Eigenschaft
    {
      get { return Elem.Length;}
    }
  }
}
```

Vorteile der späten Wertbindung

Wenn Sie diesen Code testen oder auch nur durchdenken, achten Sie bitte darauf, dass der Menübefehl AUSSCHNEIDEN nach dem ersten Aufruf des KOPIEREN-Befehls nicht nur eine andere Beschriftung, sondern auch eine veränderte Semantik erhält. Trägt der Menübefehl KLON kein Häkchen, ändert der Code aufgrund der späten Wertbindung den »Inhalt« der Zwischenablage, indem er nur den Wert des ursprünglich eingelagerten Objekts manipuliert – und zwar *ohne* Aufruf der `SetDataObject()`-Methode.

Dieses Verfahren ist zu empfehlen, wenn große Datenmengen oder viele verschiedene Repräsentationen eines Objekts (dazu gleich noch mehr) bereitgestellt werden müssen. Die Eigenschaft `Len` der Beispielklasse `MyString` ist nicht nur der Form halber da: Wie sich mit einem darauf gesetzten Haltepunkt zeigen lässt, kommt der `get`-Accessor tatsächlich erst im Zuge von `menuPaste_Click()` zur Ausführung, also immer dann, wenn das Objekt aus der Zwischenablage kopiert und tatsächlich ein Zugriff auf die Eigenschaft erfolgt. In der Praxis heißt das, dass der Datentransfer auf die Daten und Datenformate beschränkt bleiben kann, die das Zielobjekt auch benötigt – und dass das Quellobjekt so lange als Server für die Zwischenablage zur Verfügung bleiben muss, bis diese einen anderen Inhalt erhält und die Referenz den Weg alles Irdischen geht. Nicht zuletzt auch aus diesem Grund fragen viele Anwendungen, ob der Inhalt der Zwischenablage nach Programmende erhalten bleiben soll und packen vor ihrem Ableben gegebenenfalls noch einen Klon hinein.

Mehrere Datenformate und Datentypen en bloc bereitstellen

Der Aufruf der SetDataObject()-Methode im bisher vorgestellten Beispielcode war noch nicht die ganze Wahrheit. Tatsächlich instanziiert die SetDataObject()-Methode ein neues DataObject-Objekt und packt das im ersten Parameter übergebene Objekt über einen Aufruf der IDataObject-Methode SetData() unter Angabe des entsprechenden Datenformats hinein – vorausgesetzt natürlich, dass dieses nicht bereits seinerseits schon ein DataObject-Objekt ist.

Wie Sie sicher schon vermutet haben, liegt hier der Ansatzpunkt für die Bereitstellung unterschiedlicher Datenformate: Anstatt wie bisher nur ein Objekt per SetDataObject() in die Zwischenablage einzulagern, haben Sie auch die Möglichkeit, das Bündel selbst zu schnüren, indem Sie der SetDataObject()-Methode ein zuvor fertig bepacktes Container-Objekt des Typs DataObject übergeben.

Die sechste Variante des Beispielcodes demonstriert diesen Weg anhand der beiden PictureBox-Steuerelemente des Formulars, über deren Sinn und Zweck sich die vorangehenden Abschnitte fröhlich ausgeschwiegen haben. Die Hilfsmethode packDataObject() packt zwei Objekte, ein string-Objekt und das Image-Objekt des oberen PictureBox-Steuerelements zusammen in das bereitgestellte DataObject-Objekt, das danach via SetDataObject() in die Zwischenablage wandert.

```
#if Variante6
private void menuCopy_Click(object sender, System.EventArgs e)
{
  packClipboard();
}
private void packClipboard()
{
  DataObject data = new DataObject();
  Control c = ActiveControl;
  data.SetData(c.Text);
  data.SetData(pictureBox1.Image);
  Clipboard.SetDataObject(data, menuClone.Checked);
}
private void menuCut_Click(object sender, System.EventArgs e)
{
  packClipboard();
  ActiveControl.Text = "";
  pictureBox1.Image = null;
}
private void menuPaste_Click(object sender, System.EventArgs e)
{
  IDataObject data = Clipboard.GetDataObject();
  if (data.GetDataPresent(DataFormats.Text))
  {
    Control c = ActiveControl;
    c.Text = (string) data.GetData(DataFormats.Text);
  }
```

```
  if (data.GetDataPresent(DataFormats.Bitmap))
  {
    pictureBox2.Image = (Image) data.GetData(DataFormats.Bitmap);
  }
}
#endif
```

`menuPaste_Click()` holt die beiden Objekte nach Möglichkeit wieder heraus, steckt das Bild aber in die untere Picturebox. Obwohl die Zwischenablage bei diesem Experiment ein höchst spezielles Format enthält – nämlich die Kombination von Text und Grafik, kann der Text von jedem Client gelesen werden, der einen String erwartet, und das Bild von jedem Client, der mit Bitmaps zurechkommt. Umgekehrt kann die Demo-Anwendung auch jeden Text und jedes Bild anzeigen. Probieren Sie es doch einmal mit einem Screenshot des Anwendungsfensters ([Alt]+[PrintScreen]) oder des gesamten Bildschirms ([PrintScreen]).

Derartige Kombinationen sind alles andere als ungewöhnlich: Microsoft Excel stellt in die Zwischenablage kopierte Ausschnitte von Tabellenblättern beispielsweise unter anderem als Text sowie als (notwendigerweise statische) Bitmap vor – und natürlich im programmeigenen Format zum Einfügen in andere (Excel-)Tabellenblätter.

Bestehende Inhalte der Zwischenablage erhalten

Die allerwenigsten Anwendungen gehen sorgsam mit dem Inhalt der Zwischenablage um: Meistens reicht bereits ein versehentlicher Druck auf [Strg]+[C] um eine zuvor mühsam zusammengestellte Liste durch ein einzelnes Leerzeichen zu ersetzen. Anders gesagt: Ablageoperationen in die Zwischenablage geschehen meist im Blindflug – was aber gerade bei Anwendungen, die mit exotischen Datenformaten kommunizieren, nicht sein müsste.

Leider lässt sich der Inhalt der Zwischenablage nicht einfach nachträglich noch über die `SetData()`-Methode des von `GetDataObject()` gelieferten `IDataObject`-Schnittstellenobjekts erweitern, denn ihr Aufruf geht schlicht ins Leere. Der steinigere Weg, der eine Kopie der gesamten Zwischenablage in ein neues `DataObject`-Objekt vorsieht, ist so steinig auch wieder nicht. Zumindest beschwert sich das `DataObject`-Objekt nicht darüber, wenn ein Datenformat mehrfach eingelagert wird, was dann ja passieren kann. Es überschreibt den neuen Inhalt stoisch mit dem alten – und erspart damit die Filterung der Datenformate. (Es bleibt Ihnen natürlich freigestellt, sich um solche Filterungen zu kümmern, sie spart Ressourcen und Laufzeit.)

Die nicht überschreibende Version von `packClipboard()` sieht damit so aus:

```
#if Variante7
...
private void packClipboard()
{
  IDataObject idata = Clipboard.GetDataObject();
  DataObject data = new DataObject();
  foreach(string format in idata.GetFormats())
    data.SetData(format, false, idata.GetData(format));
  Control c = ActiveControl;
  if (c.Text != "")
    data.SetData(DataFormats.Text, c.Text);
  if (pictureBox1.Image != null)
    data.SetData(DataFormats.Bitmap, pictureBox1.Image);
  Clipboard.SetDataObject(data, menuClone.Checked);
}
...
#endif
```

Abbildung 18.3: Formular der Anwendung Zwischenablagen-Demo in der Variante 7

Übung

Implementieren Sie eine Variante 8 des Projekts, die das Datenformat `FileDrop` des Windows-Explorers unterstützt. Beginnen Sie mit der Implementierung für das Zielobjekt und ergänzen Sie dann die für das Quellobjekt (prüfen Sie auch die Korrektheit von Dateipfaden):

- Erstes Ziel ist, dass Sie eine Sammlung von Dateien, die Sie vom Explorer aus in die Zwischenablage kopieren, in das jeweiligen Textfeld einfügen können (nur Dateinamen samt Pfad).
- Zweites Ziel ist, dass Sie eine bearbeitete (= teilweise gelöschte) Auswahl dieser Dateinamen (im gleichen Datenformat) von der Anwendung aus zurück in die Zwischenlage kopieren.

Wenn Sie alles richtig gemacht haben, müssten Sie wiederum im Explorer per BEARBEITEN/KOPIEREN Kopien nur der ausgewählten Dateien anfertigen können – am besten in ein leeres Verzeichnis.

18.2 Drag&Drop

Wenn Sie den vorigen Abschnitt über die Zwischenablage gelesen haben, sind Sie gut vorbereitet auf Drag&Drop. Falls nicht, sollten Sie dies, wenn nicht zuerst, so auf jeden Fall begleitend tun. Er vermittelt den grundlegenden Zusammenhang zwischen den Klassen `DataObject` und `DataFormats` und stellt auch die Techniken der *frühen* und der *späten Wertbindung* vor. Diesen Hintergrund brauchen Sie für ein tiefer gehendes Verständnis von Drag&Drop auf jeden Fall.

Der Rest ist schnell erzählt und betrifft im Wesentlichen den (bei der Zwischenablage nur mittelbar vorhandenen) »Handshake« zwischen Quellobjekt und Zielobjekt sowie das visuelle Feedback für den Benutzer.

18.2.1 Ablauf der Operation

Die Rollenverteilung ist klar: Das Quellobjekt initiiert einen Ziehvorgang (Drag) – meist als Reaktion auf ein `MouseDown`-Ereignis – durch einen Aufruf seiner Methode `DoDragDrop()`. (Dieser Aufruf ist synchron und kehrt erst zurück, wenn die Operation abgeschlossen ist.) Dabei übergibt es ein Objekt eines beliebigen serialisierbaren Datentyps oder aber ein fertig gepacktes `DataObject`-Objekt sowie einen Bitvektor des `enum`-Typs `DragDropEffects`, der die zulässigen Operationen ausdrückt. Wie das »Packen« im Einzelnen vor sich geht und welche Möglichkeiten Sie dabei haben, auch exotische Datenformate zu übertragen, was gerade bei Drag&Drop ein häufiger Anwendungsfall ist, finden Sie im vorigen Abschnitt »Zwischenablage« ausführlichst beschrieben. Der Bitvektor ist meist eine Oder-Kombination der vom Explorer her bekannten Operationen `Copy`, `Link` und `Move`. Darüber hinaus gibt es noch die selten benutzte Operation `Scroll`. Die `DragDropEffects`-Aufzählung enthält zudem noch die Werte `All` und `None`, deren Bedeutung sich über die Namensgebung erschließt.

```
// in der Klasse des Quellobjekts
private void textBoxSource_MouseDown(object sender, MouseEventArgs e)
{
  // Containerobjekt leitet DD ein!
  if (DoDragDrop(textBoxSource.Text, DragDropEffects.All) ==
      DragDropEffects.Move)
    textBoxSource.Text = "";
}
```

Einladung zum Drag&Drop

Da während des Ziehvorgangs noch nicht klar ist, welches Objekt tatsächlich das spätere Zielobjekt sein wird, benachrichtigt das System jedes als OLE-Client in Frage kommende Objekt, wenn die Maus dessen Fensterbereich betritt (`DragEnter`), überstreicht (`DragOver`) und verlässt (`DragLeave`). Ein Objekt kommt nur dann als Zielobjekt in Frage, wenn seine `AllowDrop`-Eigenschaft `true` ist.

```
// in der Klasse des Zielobjekts
private void InitializeComponent()
{
  ...
  this.textBoxTarget.AllowDrop = true;
  this.textBoxTarget.DragOver +=
    new System.Windows.Forms.DragEventHandler(textBoxTarget_DragOver);
  this.textBoxTarget.DragDrop +=
    new System.Windows.Forms.DragEventHandler(textBoxTarget_DragDrop);
  this.textBoxTarget.DragEnter +=
    new System.Windows.Forms.DragEventHandler(textBoxTarget_DragEnter);
 ...
}
```

Kommunikation mit einem potenziellen Zielobjekt

Ein potenziell aufnahmebereiter Empfänger informiert sich dann beim `DataObject`-Objekt, in welchen Datenformaten dieses die Daten liefern kann und signalisiert – sofern ein verwertbares Format dabei ist – seine Bereitschaft, indem er seinerseits die von ihm unterstützten Operationen kundtut. Dazu weist er in Antwort auf eintreffende `DragEnter`- und `DragOver`-Ereignisse sowie – im Allgemeinen – unter Auswertung des Tastaturzustands `KeyState` der `Effect`-Eigenschaft des Ereignisobjekts die anvisierten Operation als `DragDropEffects`-Bitmaske zu. (Zur Analyse des Tastaturzustands gleich mehr.)

```
// in der Klasse des Zielobjekts
private void textBoxTarget_DragOver(object sender, DragEventArgs e)
{
  switch(e.KeyState)
  {
    case 0x05:  // Umschalt + Linke Maustaste
    case 0x09:  // Strg + Linke Maustaste
    case 0x0d:  // Umschalt + Strg Linke Maustaste
    case 0x21:  // Alt + Linke Maustaste
    case 0x25:  // Alt + Umschalt + Linke Maustaste
    case 0x29:  // Alt + Strg + Linke Maustaste
      e.Effect = DragDropEffects.Move;
      break;
    default:    // keine Taste und Alt + Strg + Umschalt
      e.Effect = DragDropEffects.Copy;
      break;
  }
}
```

Findet eine DragOver-Behandlung statt, ist *keine* DragEnter-Behandlung erforderlich und umgekehrt. Sie können aber auch für beide Ereignisse dieselbe Behandlungsmethode registrieren. Damit das DragDrop-Ereignis beim Lösen der Maustaste generiert wird, ist nur wichtig, dass die zuletzt aufgerufene DragXxx-Behandlungsmethode eine Operation ungleich None gesetzt hat. Welche Operation das Zielobjekt schlussendlich bei der DragDrop-Behandlung anfordert, hat damit übrigens nichts zu tun. Die gesetzte Operation wirkt sich nur darauf aus, welches visuelle Feedback (Cursorform) der Benutzer erhält.

Feedback an den Benutzer

Im Zuge der DragOver-Behandlung analysiert das Zielobjekt den Tastaturzustand und ordnet diesem die Operation zu, die es im Falle eines DragDrop-Ereignisses ausführen würde – oder eben keine: der Vorgabewert für Effect ist None. Wie der angeführte Beispielcode zeigt, erfordert die Auswertung der KeyState-Eigenschaft ein für die Programmierung mit .NET höchst selten anzutreffendes Verfahren. Mangels Konstanten müssen die Bitmasken für den Tastaturzustand als Zahlenwert zusammengestellt werden. Warum es keine KeyStates-Aufzählung gibt, die diese Bit-Gymnastik erspart, ist unverständlich. Nehmen Sie es als Rückblick in Zeiten »längst überholter« Programmierpraktiken und achten Sie einfach beim nächsten Komponenten-Update darauf, ob Microsoft darin Versäumtes nicht vielleicht nachgeholt hat.

Tabelle 18.2: Bitmasken für die Auswertung der KeyState-Eigenschaft

Bitmaske	Tastatur-/Mauszustand
0x01	Linke Maustaste gedrückt
0x02	Rechte Maustaste gedrückt
0x04	⇧ gedrückt
0x08	Strg gedrückt
0x10	Mittlere Maustaste gedrückt
0x20	Alt gedrückt

Es steht Ihnen frei, die Masken selbst zu definieren und den Code etwas sprechender zu gestalten:

```
// in der Klasse des Zielobjekts
[Flags]
enum KeyStates {None, MouseLeft, MouseRight, Shift = 4, Control = 8,
MouseMiddle = 16, Alt = 32};
```

```
private void textBoxTarget_DragOver(object sender,
System.Windows.Forms.DragEventArgs e)
{
  switch((KeyStates)e.KeyState & ~KeyStates.MouseLeft)
  {
    case KeyStates.Shift:
    case KeyStates.Control:
    case KeyStates.Shift | KeyStates.Control:
    case KeyStates.Alt:
    case KeyStates.Shift | KeyStates.Alt: Maustaste
    case KeyStates.Alt | KeyStates.Control:
      e.Effect = DragDropEffects.Move;
      break;
    default:
      e.Effect = DragDropEffects.Copy;
      break;
  }
}
```

GiveFeedBack-Ereignis

Das `DataObject`-Objekt kombiniert die vom Quellobjekt und Zielobjekt gesetzten Bitvektoren für die Operationen über ein bitweises AND und gibt, indem es das `GiveFeedback`-Ereignis signalisiert, dem Quellobjekt im Gegenzug die Möglichkeit, eine eigene Cursorform zu setzen, um dem Benutzer die jeweils ausgewählte Operation zu visualisieren. Falls das Quellobjekt keine Behandlungsroutine dafür registriert oder die Eigenschaft `UseDefaultCursors` auf `true` setzt, setzt das `DataObject`-Objekt von sich aus die standardmäßigen Cursorformen des Systems. Der folgende Code zeigt eine mögliche Implementierung der `GiveFeedback`-Behandlung:

```
// in der Klasse des Quellobjekts
private void textBoxSource_GiveFeedback(object sender,
  System.Windows.Forms.GiveFeedbackEventArgs e)
{
  e.UseDefaultCursors = true;
#if OwnCursors
  e.UseDefaultCursors = false;
  switch(e.Effect)
  {
    case DragDropEffects.Copy:
      Cursor = cur[0];
      break;
    case DragDropEffects.Move:
      Cursor = cur[1];
      break;
    case DragDropEffects.Link:
      Cursor = cur[3];
      break;
    default:
      Cursor = cur[2];
      break;
  }
#endif
```

Vielleicht werden Sie sich bei dem Code fragen, wo denn nun die unmittelbar vor dem Ablegen des Objekts eingestellte Cursor-Form wieder auf den Standardwert zurückgesetzt wird. Der `default`-Zweig kann diese Aufgabe nicht übernehmen, da es keinen Effekt gibt, der dies ausdrücken würde. Außerdem tritt das `GiveFeedback`-Ereignis im Übrigen nach dem Ablegen des Objekts auch nicht mehr auf. Die Antwort ist ein echtes Aha-Erlebnis und bedarf wohl keines weiteren Kommentars:

```
// in der Klasse des Quellobjekts
private void textBoxSource_MouseDown(object sender,
  System.Windows.Forms.MouseEventArgs e)
{
  // Containerobjekt leitet DD ein!
  if (DoDragDrop(textBoxSource.Text, DragDropEffects.All) ==
      DragDropEffects.Move)
    textBoxSource.Text = ""; // wird erst nach Abschluss der gesamten
  Cursor = Cursors.Default;  // DD-Operation ausgeführt
}
```

Cursorformen als Ressourcen einlesen

Das `cur`-Array initialisiert der Konstruktor – durch Einlesen von eingebetteten Ressourcen. Hierbei handelt es sich um ein gängiges Verfahren, das Sie für alle möglichen Ressourcen der Typen *.bmp, .gif, .jpg, .jpe, .jpeg, .ico* und *.cur* anwenden können. Es hat den Vorteil, dass Sie – mit Blick auf eine etwaige Auslieferung – für Ihre Anwendung nicht etliche externe Dateien bereitstellen, verwalten und zusammenhalten müssen, zumal die Anwendung vielleicht ja nicht mehr funktioniert, wenn versehentlich eine davon gelöscht, verschoben, verändert oder auch nur umbenannt wurde.

Die notwendigen Schritte zum Einbetten von Ressourcen der genannten Typen sind:

1. Fügen Sie die Dateien über den Menübefehl PROJEKT/VORHANDENES ELEMENT HINZUFÜGEN in das Projekt ein. Visual Studio erzeugt dabei eine Kopie aller Dateien im Projektverzeichnis. (Sie verlieren also nicht die Originaldatei, wenn Sie später wieder welche davon löschen.)
2. Setzen Sie im EIGENSCHAFTEN-Fenster die `BuildAction`-Eigenschaft aller Ressourcedateien auf *Eingebettete Ressource.*
3. Lesen Sie die Ressourcen nach dem folgenden Muster ein, wobei Sie peinlichst auf die Schreibweise des Dateinamens achten müssen, da C# auch hier penibel zwischen Groß- und Kleinschreibung unterscheidet.

   ```
   cur = new Cursor[4];
   cur[0] = new Cursor(typeof(Form1),"H_CROSS.CUR");
   ...
   ```

   Die hier verwendete Überladung des Konstruktors existiert analog auch für die Datentypen `Bitmap` und `Icon`. Sie erwartet im ersten Parameter die

Typinformation einer beliebigen (!) Klasse, die dem Standardnamensraum der die Ressource bereitstellenden Assembly angehört. Dieser Namensraum lässt sich im EIGENSCHAFTEN-Dialog über die Eigenschaft `Standardnamespace` des jeweiligen Projekts setzen, dem die Ressource hinzugefügt wurde und muss gegebenenfalls nachträglich noch angepasst werden, wenn Sie den für das Codegerüst automatisch generierten Namensraum nicht beibehalten haben. Kurzum, er sollte mit dem Namensraum der im ersten Parameter genannten Klasse übereinstimmen, andernfalls erhalten Sie beim Versuch, die Ressourcen einzulesen, kuriose Ausnahmen mit wenig aussagekräftigen Fehlermeldungen.

QueryContinueDrag

Das Ereignis `QueryContinueDrag` gehört auch noch zum Feedbacksystem, wird aber im Allgemeinen nur selten behandelt. Tatsächlich wird es vom Formular gar nicht erst an das Quellobjekt weitergeleitet, sondern muss auf Formularebene (bzw. Container-Ebene) behandelt werden. Eine installierte Behandlungsroutine kommt in regelmäßigen Zeitabständen, etwa alle 50 Millisekunden zum Aufruf und kann den Vorgang von sich aus abbrechen, indem sie die `Action`-Eigenschaft des Ereignisobjekts auf den Wert `DragAction.Cancel` setzt. Das Ereignisobjekt transportiert zudem noch die Eigenschaften `KeyState` und `EscapePressed`. Letztere ist Grundlage für das Standardverhalten – und sollte auch für die Behandlungsmethode ein Kriterium für den sofortigen Abbruch sein:

```
// in der Klasse des Quellobjekts
private void textBoxSource_QueryContinueDrag(object sender,
  System.Windows.Forms.QueryContinueDragEventArgs e)
{
  if(e.EscapePressed)
  {
    e.Action = DragAction.Cancel;
    System.Console.WriteLine(
      "{0} Drag-Vorgang auf Benutzerwunsch abgebrochen", DateTime.Now);
  }
}
```

An weiteren Aktionen stehen noch `Continue` und `Drop` im Angebot. Beide werden allerdings nur in ganz speziellen Situationen, beispielsweise bei Implementierung von Eingabehilfen für Behinderte (Stichwort: Accessibility) oder beim Abspielen aufgezeichneter Vorgänge Anwendung finden.

Ablegen des Objekts

Sobald der Benutzer den gezogenen Inhalt auf einem empfangswilligen Zielobjekt ablegt, kann dieses damit beginnen, den Inhalt auszuwerten. Wie das im Einzelnen vor sich geht und welche Möglichkeiten Sie dabei haben, auch exotische Datenformate zu übertragen – was gerade bei Drag&Drop ja ein häufiger Anwendungsfall ist – finden Sie im vorigen Abschnitt »Zwischenablage« ausführlichst beschrieben.

Bei der DragDrop-Behandlung kann sich das Zielobjekt, wie schon angedeutet, über die während der DragOver- bzw. DragEnter-Behandlung zuletzt gesetzte Aktion hinwegsetzen und eine andere Aktion ausführen. Ob es sinnvoll ist, den Benutzer derartig zu verwirren, sei dahingestellt. Auf jeden Fall sollte das Zielobjekt bei einem Misslingen der Operation anzeigen, dass etwas schiefgegangen ist und die Operation None setzen. Das ist besonders wichtig, wenn der Benutzer eine Verschiebeoperation durchführen wollte und der Inhalt auf Seiten des Quellobjekt nun doch nicht gelöscht werden darf. Bei dem Datenformat DataFormats.Text mag dies vielleicht überflüssig erscheinen, bei anderen Datenformaten kann aber durchaus etwas schief gehen, und dies sollte das Quellobjekt in jedem Fall mitgeteilt bekommen.

```
private void textBoxTarget_DragDrop(object sender,
System.Windows.Forms.DragEventArgs e)
{
  if (e.Data.GetDataPresent(DataFormats.Text, true))
    try
    {
      textBoxTarget.Text = (string) e.Data.GetData(DataFormats.Text);
    }
  catch
    {
      e.Effect = DragDropEffects.None;
      return;
    }
}
```

Hier noch einmal der Code um den DoDragDrop()-Aufruf herum, der die Move-Operation abschließt:

```
// in der Klasse des Quellobjekts
private void textBoxSource_MouseDown(object sender, MouseEventArgs e)
{
  // Containerobjekt leitet DD ein!
  if (DoDragDrop(textBoxSource.Text, DragDropEffects.All) ==
      DragDropEffects.Move)
    textBoxSource.Text = "";
}
```

18.2.2 Zusammenfassung

Hier noch einmal ein schneller Überblick über die Abwicklung einer einfachen Drag&Drop-Operation ohne Schnörkel.

Aus Sicht des Quellobjekts

Grundsätzlich kann jedes Steuerelement eines Formulars und auch das Formular selbst die Rolle des Quellobjekts einnehmen. Die Aufgaben aufseiten des Quellobjekts sind:

1. *Einleiten der Drag&Drop-Operation* in Reaktion auf eine Benutzeraktion – im Einzelnen heißt das: Bereitstellen des zu übermittelnden Objekts und Aufruf der `DoDragDrop()`-Methode.
2. *Optional:* Für das Quellobjekt kann eine `GiveFeedback`-Behandlung erfolgen, wenn die Darstellung eigener Cursorformen erwünscht ist.
3. *Optional:* Für den Container (im Allgemeinen: das Formular) kann eine `QueryContinueDrag`-Behandlung erfolgen, wenn eine Ablaufkontrolle erwünscht ist – beispielsweise, um den Vorgang auch programmseitig abzubrechen.
4. *Löschen des übermittelten Objekts*, wenn eine `Move`-Operation ausgeführt wurde.

Aus Sicht des Zielobjekts

Damit ein Formular oder Steuerelement als Zielobjekt eines Drag&Drop-Vorgangs auftreten kann, müssen mindestens drei Voraussetzungen gegeben sein:

1. Die `AllowDrop`-Eigenschaft des Objekts muss `true` sein. Ist diese Eigenschaft `false`, erhält das Objekt keine `DragXxx`-Ereignisse und das Quellobjekt zeigt die Cursorform `Cursors.No` (Verbotsschild) an.
2. Für das Objekt muss eine Behandlungsmethode für das `DragEnter`- oder `DragDrop`-Ereignis registriert sein – wahlweise auch für beide, was jedoch wenig Sinn macht. Diese Methode muss eine Operation ungleich `None` setzen. Mit einer `DragEnter`-Behandlung kommen Sie aus, wenn das Zielobjekt ohnehin nur *eine* Operation (im Allgemeinen `Copy`) unterstützt, andernfalls muss eine `DragDrop`-Behandlung mit Unterscheidung des Tastatur- bzw. Mauszustands erfolgen, damit ein visuelles Feedback möglich ist.
3. Für das Objekt muss eine Behandlungsmethode für das `DragDrop`-Ereignis registriert sein, die das übermittelte `DataObject`-Objekt auswertet. Falls die Operation misslingt, sollte diese Methode die Operation `None` setzen.

Codebeispiel

Ein typisches Codebeispiel für eine einfache exotische Drag&Drop-Operation finden Sie ab Seite 630. Es demonstriert das Einfügen neuer Symbolschaltflächen in eine Symbolleiste per Drag&Drop.