

# Und nun zu den Nachrichten

.NET  
Beginners

## Nachrichtenverarbeitung unter .NET

von Dirk Frischalowski

Die Nachrichtenverarbeitung über Ereignisse innerhalb von Windows Forms erfolgt so gut wie transparent. Wer als Entwickler bereit ist, etwas tiefer in die Ereignisverarbeitung einzutauchen, entdeckt zahlreiche neue Lösungsmöglichkeiten für alltägliche Probleme. Welche der Lösungen für welches Problem geeignet ist, wann Sie welche Lösung einsetzen können und welche Vorgänge unter der Oberfläche ablaufen, soll in diesem Artikel gezeigt werden.

Die Verarbeitung von Nachrichten und Ereignissen kann auf vielfältige Weise erfolgen. Soll ein Programm nur auf Ereignisse reagieren, ist die Vorgehensweise vergleichsweise einfach. Wie sieht es aber aus, wenn ein Programm eigene Ereignisse bereitstellen möchte oder es zur Konfiguration eines Kontrollelements keine Standardeigenschaft gibt oder Sie das Verhalten eines Kontrollelements beeinflussen möchten? Für alle diese Aufgaben müssen Sie tiefer in die Verarbeitung von Ereignissen und Nachrichten einsteigen.

Gleich zu Beginn soll angemerkt werden, dass die in diesem Artikel beschrie-

benen Techniken spezifisch für Windows sind. Der Artikel beginnt mit einem kurzen Überblick über das Auslösen und Abfangen von Ereignissen in .NET – dies ist prinzipiell in der Konsole wie auch in grafischen Anwendungen möglich. Daran anschließend wird das (vom .NET Framework unabhängige) Windows-Nachrichtensystem vorgestellt und verschiedene Formen des Subclassing betrachtet, die es erlauben, Nachrichten, die an ein Fenster gesendet werden, zu filtern. Zum Abschluss werden spezielle Formen der Nachrichtenverarbeitung in .NET und die Interprozesskommunikation über Nachrichten behandelt.

### Ereignisse á la .NET

Zur Ereignisbehandlung gehören immer mindestens zwei: Der Sender und der Empfänger. Wenn eine Windows-Forms-Anwendung auf das Klicken eines Button reagiert, ist sie der Empfänger. Doch woher

„weiß“ der Button, dass er einen Mausklick erhalten hat und wie wird diese Information weitergeleitet? Zur Veranschaulichung soll folgendes Beispiel dienen: Eine Klasse soll Log-Einträge von verschiedenen Objekten entgegennehmen und diese Informationen an alle registrierten Ereignisempfänger weiterleiten. Es können folgende Teilnehmer identifiziert werden: Ein Objekt *Auslöser* erzeugt einen Log-Eintrag. Es sendet diese Information an eine zentrale Stelle, den *Verwalter*. Weiterhin existieren bereits Objekte, welche Log-Einträge verarbeiten, die *Empfänger*. Der Empfänger reagiert beispielsweise mit einer Fehlermeldung oder schreibt den Eintrag in eine Datei. Diese Objekte registrieren sich beim *Verwalter* als Ereignisempfänger (Abbildung 1). Die Aktionen 1 und 2 müssen dabei nicht in dieser Reihenfolge auftreten. Es können jederzeit weitere Auslöser und Empfänger hinzukommen bzw. entfallen.

#### Inhalt

Dieser Artikel stellt das Prinzip der Nachrichtenverarbeitung bei Windows Forms vor. Der Schwerpunkt liegt dabei auf dem Erzeugen von Ereignissen, dem Reagieren auf System- und Benutzernachrichten sowie dem Subclassing

#### Zusammenfassung

Auch bei Windows Forms spielt die Nachrichtenverarbeitung eine Rolle, wenngleich sie über Klassen so gekapselt wird, dass die Entwickler nur in Ausnahmefällen mit ihr in Kontakt kommen

#### Quellcode

C#

Quellcode: [www.dotnet-magazin.de](http://www.dotnet-magazin.de)

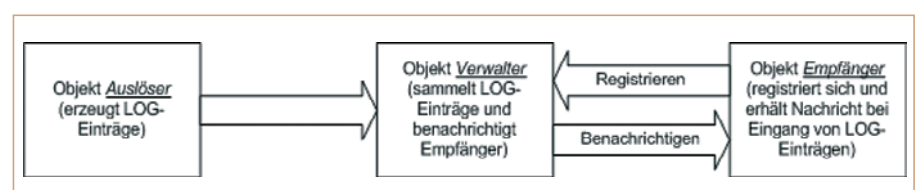


Abb. 1: Vereinfachte Ereignisverarbeitung in .NET

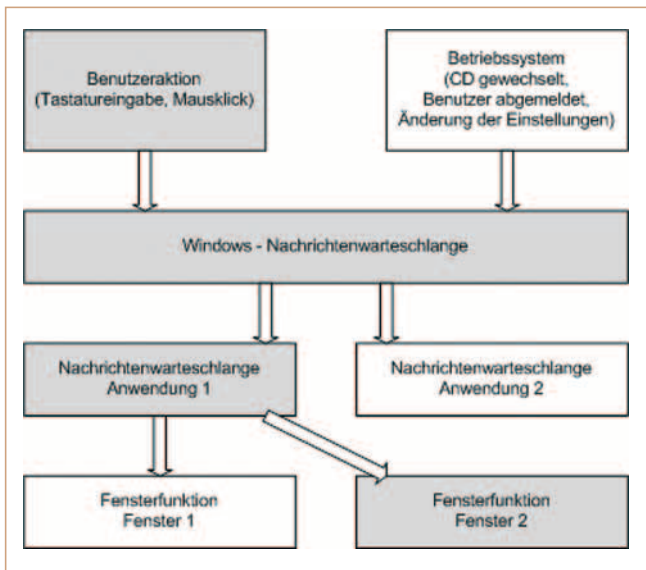


Abb. 2: Der Nachrichtenfluss unter Windows

In .NET wird dazu noch eine Klasse deklariert, die eine Methode enthält, welche beim Auftreten eines Ereignisses, zum Beispiel dem Erzeugen eines Log-Eintrages, aufgerufen wird. Diese Methode kann die Informationen zum Ereignis verarbeiten und ruft dann alle registrierten Empfänger auf. Die Empfänger (eine Methode des Empfängerobjekts) werden durch Delegates verwaltet. Bei dem Methodenaufruf des Empfängers wird weiterhin ein Objekt übergeben, das Informationen zum eingetretenen Ereignis enthält. Eine Realisierung finden Sie im mitgelieferten Beispiel im Projekt *EreignisDefinieren.csproj*. Ereignisse sind vereinfacht gesagt also nichts weiter als Methodenaufrufe.

### Nachrichtenverarbeitung

Windows ist ein auf Nachrichten basierendes Betriebssystem. Soll zum Beispiel eine Anwendung beendet werden, erzeugt Windows eine Nachricht, in diesem Fall `WM_QUIT`, und sendet diese an das Anwendungsfenster. Jede grafische Anwendung unterhält einen Haupt-Thread mit einer eigenen Nachrichtenschlange, in der die Verarbeitung der Nachrichten erfolgt. Die Nachrichtenschleife wird bei grafi-

schen Anwendungen durch den Aufruf von *Application.Run()* eingerichtet. Die Konsolenanwendungen besitzen eine solche Nachrichtenschleife nicht und können aus diesem Grund standardmäßig keine Windows-Nachrichten verarbeiten. Weiterhin besitzt jedes Fenster eine *Fensterfunktion*, die alle Nachrichten verarbeitet, die an das Fenster gerichtet sind. Klickt ein Anwender auf eine Schaltfläche, erzeugt der Maustreiber eine Information für das Betriebssystem und stellt die Nachricht in die Windows-Nachrichtenschleife. Windows verarbeitet die Nachrichten, identifiziert die Anwendung, für die der Mausklick bestimmt war, und stellt die Nachricht in die Nachrichtenschleife der betreffenden Anwendung. Diese Nachricht beinhaltet eine Information zum Typ der Nachricht (linke Maustaste betätigt) und noch weitere Informationen (Koordinaten). Von dort wird sie an die Fensterfunktion der Schaltfläche weitergereicht (Abbildung 2).

Durch das .NET Framework werden viele Nachrichten, die an eine Fensterfunktion gesendet werden, bereits vorverarbeitet. In Abbildung 1 ist der Auslöser zum Beispiel die Nachricht „Maustaste geklickt“. Eine Instanz, die eine Nachricht verarbeitet, liest diese Information und erzeugt daraus ein Ereignis, auf das ein Programm bequem reagieren kann. Mehrere Mausklicks oder Nachrichten zum Neuzeichnen von Fenstern (`WM_PAINT`) können aus Geschwindigkeitsgründen zusammengefasst werden. Weiterhin können Nachrichten auch direkt

in eine Nachrichtenwarteschlange von einer Fensterfunktion eingereicht werden und damit die anwendungsweite Nachrichtenschleife umgehen. Ziel dieser Funktionen ist es, sofort zur Ausführung zu kommen. Sie selbst können auch Nachrichten versenden, zum Beispiel um ein Kontrollelement ganz speziell zu konfigurieren. Ein Fenster (jedes sichtbare Kontrollelement ist in der Regel ein eigenes Fenster) wird dabei über ein Handle, einen eindeutigen Zahlenwert, identifiziert. So greifen Sie auf das Handle einer TextBox über *textBox.Handle* zu. Das mitgelieferte Beispiel *SpezialEdit.csproj* sendet beispielsweise die Nachricht `EM_SETCUEBANNER` an ein Eingabefeld (Listing 1). Unter Windows XP bewirkt dies, dass bei einem leeren Eingabefeld ein grauer Hinweistext angezeigt wird (Abbildung 3).

Wie komfortabel die Unterstützung für XP-Styles ist, hängt von der .NET-Version ab. Während unter dem kommenden .NET 2.0 die Unterstützung durch jedes Steuerelement fest eingebaut ist, ist bei 1.1 lediglich ein Aufruf von *Application.EnableVisualStyles()* vor *InitializeComponent* erforderlich. Unter .NET 1.0 war alles noch ein wenig komplizierter – neben einer zusätzlichen Manifestdatei für die Bindung an die richtige Version von *Comctl32.dll* im Verzeichnis der Anwendung, war auf ein Aufruf des *SendMessage*-API erforderlich. Für den Einsatz der API-Funktion muss der Namespace *System.Runtime.InteropServices* eingebunden werden. Anschließend wird nun *SendMessage()* über *DllImport* importiert. Der Aufruf von *SendMessage* erfolgt in der *Click*-Prozedur des Button, wobei dieser Funktion das Fenster-Handle, die Nachricht und als letzter Parameter

#### Listing 1

```

.. \SpezialEdit\MainForm.cs
using System.Runtime.InteropServices;
...
[DllImport("user32.dll", EntryPoint="SendMessage",
    CharSet=CharSet.Unicode)]
static extern int SendMessage(IntPtr hwnd, Int32 wParam,
    Int32 lParam, string lParam);
private const int ECM_FIRST = 0x1500;
private const int EM_SETCUEBANNER = ECM_FIRST + 1;
...
String s = "Geben Sie was ein...";
SendMessage(textBox1.Handle, EM_SETCUEBANNER, 0, s);
  
```

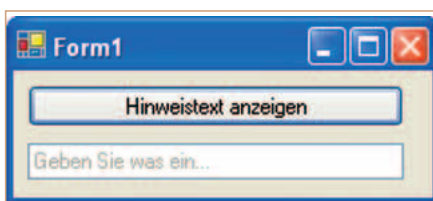


Abb. 3: Hinweistext unter Windows XP

der zu verwendende Hinweistext übergeben wird. Die Werte der beiden letzten Parameter hängen immer von der verwendeten Nachricht ab (Listing 1).

Um in Ihren Anwendungen spezielle Features von Windows zu nutzen, kommen Sie also um eine Nachrichtenverarbeitung, die über die Möglichkeiten von .NET hinausgeht, nicht herum. Die möglichen Nachrichten sind vom verwendeten Betriebssystem abhängig.

### Subclassing

Wie zu erwarten hat das Subclassing in der Nachrichtenverarbeitung nichts mit dem Ableiten von Klassen zu tun. Vielmehr geht es darum, in den Standardweg der Nachrichtenverarbeitung einzugreifen. Dies kann durch das Überschreiben der Methode *WndProc()*, dem „Verbiegen“ der Fensterfunktion oder das Verwenden spezieller Nachrichtenfilter erfolgen. Jede Methode hat Ihre Vor- und Nachteile und eignet sich nur in einem speziellen Umfeld. Für alle Anwendungen benötigen Sie wie im vorigen Beispiel die numerischen Codes der Nachrichten. Sie finden Sie in Visual Studio bei installiertem Visual C++ unter `[LW]:\Programme\Microsoft Visual Studio .NET 2003\VC\PlatformSDK\Include` oder im Internet nach dem Laden und Installieren des Core SDK [1] im Verzeichnis `[LW]:\Programme\Microsoft SDK\include` in den Dateien *CommCtrl.h* (allg. Kontrollelemente) und *WinUser.h* (Systemnachrichten). Nachdem der Wert bekannt ist, erstellen Sie eine entsprechende Konstante im Code und verwenden diese dann später zur Identifizierung der Nachricht. Das Klicken einer Maustaste wird beispielsweise über die Konstanten `WM_LBUTTONDOWN` sowie `WM_RBUTTONDOWN` definiert:

```
const int WM_LBUTTONDOWN = 0x0201;
const int WM_RBUTTONDOWN = 0x0204;
```

Subclassing dient damit der Filterung von bereits vordefinierten Nachrichten des Betriebssystems. Aber auch das Verarbeiten eigener Nachrichten ist dadurch möglich. Die einfachste und bekannteste Methode des Subclassing ist das Überschreiben der Methode *WndProc()* der Klasse *Control*. Damit besitzt jede visuelle Komponente, zum Beispiel ein Formular, diese Metho-

de, da die korrespondierenden Klassen direkt oder indirekt von *Control* abgeleitet sind. Dies bedeutet auch, dass jedes Kontrollelement als eigenes Fenster verwaltet wird. Die Fensterfunktion verarbeitet eine große Anzahl an Nachrichten, sodass es nicht ratsam ist, diese beispiels-

## Subclassing kennt viele Varianten

weise mit *MessageBox.Show()* einzeln anzuzeigen. Weil jede Mausbewegung und jedes Neuzeichnen des Fensters ein Ereignis erzeugt, erhalten Sie eine große Zahl an Nachrichten (das Neuzeichnen ist z.B. immer nach dem Anzeigen der *MessageBox* notwendig). In der Regel werden nur einige Nachrichten gefiltert – danach muss entschieden werden, ob diese an die übergeordnete Fensterfunktion weitergereicht oder als verarbeitet markiert werden sollen. Im letzten Fall müssen Sie sehr sorgsam vorgehen, da bestimmte Nachrichten unbedingt weitergegeben werden müssen, um eine korrekte Funktion Ihrer Anwendung zu gewährleisten. Die Weiterleitung erfolgt durch einen Ausruf der Methode *WndProc()* der Basisklasse und der Übergabe der aktuellen Nachricht (Listing 2). Beachten Sie bitte, dass nur Ereignisse abgefangen werden, die an das Formular gerichtet sind. Wenn sie beispielsweise in die angezeigte Textbox des Beispiels klicken, wird keine Nachricht an das Formular erzeugt.

Nachteilig an dieser Lösung ist, dass Sie nur für ein einzelnes Fenster die Methode *WndProc()* überschreiben können.

### Listing 2

```
..\Subclassing\MainForm.cs
protected override void WndProc(ref Message m)
{
    switch(m.Msg)
    {
        case WM_LBUTTONDOWN: textBox1.AppendText
            ("WndProc: LBUTTONDOWN\r\n"); break;
        case WM_RBUTTONDOWN: textBox1.AppendText
            ("WndProc: RBUTTONDOWN\r\n"); break;
    }
    base.WndProc(ref m);
}
```

Außerdem muss zum Überschreiben von *WndProc()* bei Steuerelementen erst ein neues Steuerelement abgeleitet werden. Die nächste Variante ist da dann schon etwas fortschrittlicher, denn hier verlagern Sie den Code zum Auswerten der Nachrichten in eine eigene Klasse. Diese muss von *System.Windows.Forms.NativeWindow* abgeleitet werden und die Methode *WndProc()* überschreiben. Danach wird dann zum Beispiel im Konstruktor eines Formulars ein Objekt dieser Klasse erzeugt und über die Methode *AssignHandle()* das Handle des betreffenden Formulars zugewiesen. Sollten Sie auf Elemente des Formulars zugreifen wollen, dann können Sie diese wie in Listing 3 dem Konstruktor der betreffenden Klasse übergeben.

Diese Methode besitzt den den Nachteil, dass Sie innerhalb der Methode *WndProc()* nicht direkt auf die Elemente der verschiedenen Formulare zugreifen können und somit Referenzen auf diese Elemente im Konstruktor (oder auf einem anderen Weg) bereitstellen müssen. Da-

### Listing 3

```
..\Subclassing\MainForm.cs
public class MyMsgWindow: NativeWindow
{
    const int WM_LBUTTONDOWN = 0x0201;
    const int WM_RBUTTONDOWN = 0x0204;
    private TextBox tbAusgabe;
    public MyMsgWindow(TextBox tb)
    {
        tbAusgabe = tb;
    }
    protected override void WndProc(ref Message m)
    {
        switch(m.Msg)
        {
            case WM_LBUTTONDOWN: tbAusgabe.AppendText
                ("LBUTTONDOWN"); break;
            case WM_RBUTTONDOWN: tbAusgabe.AppendText
                ("RBUTTONDOWN"); break;
        }
        base.WndProc(ref m);
    }
    ...
    public Form1()
    {
        InitializeComponent();
        MyMsgWindow mmw = new MyMsgWindow(textBox1);
        mmw.AssignHandle(this.Handle);
    }
}
```

gegen können Sie die Nachrichtenverarbeitung einfacher aktivieren und deaktivieren (*ReleaseHandle()*). Eine letzte Variante wird durch das Interface *IMessageFilter* bereitgestellt. Darüber verarbeiten Sie nicht nur die Nachrichten eines Fensters, sondern sämtliche Nachrichten, die in die Nachrichtenschleife der Anwendung (bzw. eines Threads) gelangt sind, dort bereits verarbeitet, aber noch nicht an die betreffenden Steuerelemente wei-

tergeleitet wurden. Über den Wert *HWnd* können Sie prüfen, für welches Fenster die Nachricht bestimmt war. Liefert die einzige Methode des Interfaces *PreFilterMessage()* den Wert *true*, gilt die Nachricht als verarbeitet. Durch die Rückgabe von *false* wird die Nachricht zur Verarbeitung weitergegeben (Listing 4).

Der Filter wird über die Methode *AddMessageFilter()* des *Application*-Objekts registriert und über die Methode *RemoveMessageFilter()* wieder entfernt. Die Besonderheit dieser Filtermethode ist, dass eventuell nicht alle Nachrichten diesen Weg nehmen. Wird die Nachricht über *SendMessage()* verschickt, wird diese auf dem direktem Weg in die Nachrichtenschlange der Fensterfunktion eingereiht. Ein Aufruf von *PostMessage()* fügt dagegen die Nachricht in die Nachrichtenschlange des Thread ein (beide aus der Win32-API). Da alle Nachrichten für die Anwendung nun auch diese Methode passieren müssen, kann sie sich als Performanzbremse erweisen. Führen Sie deshalb keine zeitkritischen Anweisungen in dieser Methode aus.

### Nachrichten im verwalteten Code kapseln

Eine Nachricht wird in .NET durch ein Objekt vom Typ der Struktur *Message* im Namespace *System.Windows.Forms* gekapselt. Die Eigenschaften erlauben den Zugriff auf das Fenster, das diese Nachricht erhält (Eigenschaft *hWnd*), den Nachrichtentyp (*Msg*), zwei zusätzliche Parameter *wParam* und *lParam* (die beispielsweise die Koordinaten eines Mausklicks enthalten können) sowie die Eigenschaft *Result*, die den Rückgabewert nach der Verarbeitung der Nachricht beinhaltet. Ein *Message*-Objekt wird aus Geschwindigkeitsgründen allerdings durch die Methode *Create()* und nicht über *New* erzeugt. Die Methode *Create()* greift in diesem Fall auf einen Pool von bereits erzeugten Strukturen zurück:

```
Message m = Message.Create(this.Handle,
    WM_LBUTTONDOWN, IntPtr.Zero, IntPtr.Zero);
```

Ein Beispiel zum Versenden von Nachrichten über *SendMessage()* und *PostMessage()* können Sie bei den Beispielprojekten im Verzeichnis *..\Subclassing\MainForm.cs* finden.

### Tastaturnachrichten filtern

Speziell zum Filtern von Tastatureingaben gibt es nun mehrere Möglichkeiten: Im einfachsten Fall reagieren Sie auf die Ereignisse *OnKeyDown*, *OnKeyPress* und *OnKeyUp*, die in der angegebenen Reihenfolge beim Betätigen einer Taste ausgelöst werden. *OnKeyPress* wird allerdings nur dann ausgelöst, wenn es sich auch um drückbare Tasten handelt. Eine weitere Alternative bietet die Methode *PreProcessMessage()*. Diese wird immer dann aufgerufen, wenn es sich um ein Ereignis handelt, das eine Eingabe repräsentiert, also zum Beispiel eine Tastatureingabe. Die Nachrichten werden dabei verarbeitet, bevor sie an die Methode *WndProc()* weitergeleitet werden. Damit diese aufgerufen wird, muss sie für das entsprechende Steuerelement überschrieben werden. Im Falle einer *TextBox* müssen Sie also eine neue Klasse ableiten. Auf diese Weise erreichen Sie, dass Sie die entsprechenden Anweisungen nur ein einziges Mal in der neuen Komponente angeben (oder ändern) müssen. So können Sie zum Beispiel eine spezielle *TextBox* erzeugen, die nur Leerzeichen als Eingabe zulässt. Beachten Sie, dass eventuell noch zusätzliche Anweisungen zum Verarbeiten von Tastenkombinationen notwendig sind (z.B. Umschalt + “). In Listing 5 wird dazu eine neue Klasse *NumberTextBox* deklariert und die Methode *PreProcessMessage()* überschrieben. Der Tastencode wird in der Eigenschaft *WParam* der Nachricht übergeben. Liefern Sie den Wert *true* zurück, gilt die Verarbeitung als erledigt. Auf diese Weise filtern Sie die Eingaben. Für alle anderen Nachrichten rufen Sie die Methode *PreProcessMessage()* der Basisklasse für die Standardverarbeitung auf.

Die Klasse *Control* besitzt noch weitere geschützte Methoden, die überschrieben werden können, um Tastaturereignisse zu verarbeiten. Über *ProcessCmdKey()* verarbeiten Sie Befehlstasten, mit *ProcessKeyEventArgs()* ganz allgemeine Tasteneingaben und mit *ProcessKeyPreview()* können Sie die Verarbeitung von Tasten durch übergeordnete Komponenten steuern.

### Systemnachrichten verarbeiten

Für einige Systemnachrichten bietet die Klasse *Microsoft.Win32.SystemEvents*

#### Listing 4

```
..\Subclassing\MainForm.cs
public class MyMsgFilter: IMessageFilter
{
    const int WM_LBUTTONDOWN = 0x0201;
    const int WM_RBUTTONDOWN = 0x0204;
    private TextBox tbAusgabe;
    public MyMsgFilter(TextBox tb)
    {
        tbAusgabe = tb;
    }
    public bool PreFilterMessage(ref Message m)
    {
        switch(m.Msg)
        {
            case WM_LBUTTONDOWN: tbAusgabe.AppendText(
                "LBUTTONDOWN"); break;
            case WM_RBUTTONDOWN: tbAusgabe.AppendText(
                "RBUTTONDOWN"); break;
        }
        return false;
    }
    ...
    public Form1()
    {
        InitializeComponent();
        Application.AddMessageFilter(new MyMsgFilter(
            textBox1));
    }
}
```

#### Listing 5

```
..\FilterKeys\MainForm.cs
public class NumberTextBox : TextBox
{
    public override bool PreProcessMessage(ref Message m)
    {
        const int WM_KEYDOWN = 0x0100;
        if(m.Msg == WM_KEYDOWN)
            return !(((int)m.WParam >= 48) && ((int)m.
                WParam <= 57));
        return base.PreProcessMessage(ref m);
    }
}
```

bereits vorgefertigte Ereignisse an. So können Sie beispielsweise auf das Abmelden eines Nutzers oder das Ändern von Systemeinstellungen reagieren. Die Verwendung der Klasse ist mit der Ereignisbehandlung eines Fensters identisch. Je nach Ereignistyp stehen Ihnen über den zweiten Parameter mehr oder weniger umfangreiche Reaktionsmöglichkeiten zur Verfügung. So können Sie beispielsweise im Ereignis *SessionEnding* über die Eigenschaft *Cancel* des Parameters vom Typ *SessionEndingEventArgs* den Abmeldevorgang auch abbrechen (ohne Garantie). Wenn sich die Benutzereinstellungen geändert haben, erhalten Sie über die Eigenschaft *Categorie* vom Typ *UserPreferenceCategory* (z.B. *Color*, *General*, *Locale*) eine allgemeine Information, in welchem Bereich sich etwas geändert hat (Listing 6). Den geänderten Wert können Sie aber nur durch manuelles Auslesen und Vergleichen herausbekommen.

### Nachrichten-Log

Wenn ein Programm auf Nachrichten nur lesend zugreifen können soll, bietet sich ein Überschreiben der Methode *OnNotifyMessage()* an. Dadurch wird nämlich sichergestellt, dass keine Nachricht verändert werden kann. Auch der Aufwand für die Implementierung ist geringer, da weder eine Methode der Basisklasse aufgerufen (diese enthält keine Implementierung), noch eine Nachricht weitergereicht werden muss. Die Methode *OnNotifyMessage()* wird allerdings erst dann aufgerufen, wenn zusätzlich über die Methode *setStyle()* des betreffenden Kontrollelements das Format-Bit *EnableNotifyMessage* gesetzt wird. Ein Programm kann damit also Nachrichten von beliebigen Steuerelementen (von denen zuvor entsprechende Klassen abgeleitet wurden) auslesen. In Listing 7 wird im Konstruktor der Formalklasse das Format-Bit gesetzt. Die überschriebene Methode *OnNotifyMessage()* reagiert auf das Klicken mit der linken und rechten Maustaste. Als Parameter wird an *OnNotifyMessage()* wie üblich ein Objekt vom Typ *Message* übergeben.

### Anwendungskommunikation mit Nachrichten

Es ist interessant, dass es unter Windows im Grunde keinen echten Standard für

die Kommunikation zwischen Anwendungen gibt. COM-Automatisierung hat sich nie wirklich durchgesetzt, Named Pipes sind C++- Programmierern vorbehalten und „Shared Memory“ zu nutzen, setzt ebenfalls gute Systemkenntnisse voraus. Eine Alternative, die sich auch von Visual Basic aus relativ einfach nutzen lässt, ist das Versenden von Nachrichten via *SendMessage()* und *PostMessage()*. Voraussetzung hierfür ist, dass das Fenster-Handle des Zielfensters bekannt ist. Üblicherweise wird dies mit der Funktion *FindWindow()* in allen Hauptfenstern gesucht. Dieser Funktion wird der Klassenname der Fensterklasse und der Name des Fensters (entspricht dem Fenstertitel) übergeben. Ist der Rückgabewert größer als 0, wurde mindestens ein passendes Fenster ermittelt. Anhand von der Funktion *FindWindowEx()* können Sie ein Fenster auch in den Kindfenstern der Anwendungen suchen. Wenn Sie ein Fen-

#### Listing 6

```
.. \FilterSysMsgs \ MainForm.cs
using Microsoft.Win32;
...
SystemEvents.UserPreferenceChanged +=
    new UserPreferenceChangedEventHandler
        (this.SystemMsgHandler2);
...
private void SystemMsgHandler2(object sender,
    UserPreferenceChangedEventArgs e)
{
    textBox1.Text = e.Category.ToString();
}
```

#### Listing 7

```
.. \NotifyMsg \ MainForm.cs
public Form1()
{
    InitializeComponent();
    this.SetStyle(ControlStyles.EnableNotifyMessage,
        true);
}
protected override void OnNotifyMessage(Message m)
{
    const int WM_LBUTTONDOWN = 0x0201;
    const int WM_RBUTTONDOWN = 0x0204;

    if((m.Msg == WM_RBUTTONDOWN) | (m.Msg == WM_
        LBUTTONDOWN))
        textBox1.Text = textBox1.Text + "\r\n" + m.Msg.
            ToString();
}
```

# Sie suchen die wirklich wichtigen Informationen rund um .NET?

## Wir haben Sie!

### Ihre Abo-Vorteile:

- Sie erhalten den 128 MB dot.net-**USB-Stick**\*
- Sie sparen rund 10% gegenüber dem Einzelverkauf
- Sie erhalten das **Riesen-Poster** „.NET Framework 2.0“
- Mit der **Jahres-CD** erhalten Sie alle Ausgaben des vergangenen Jahres gratis
- Sie verpassen keine Ausgabe
- Jede Ausgabe inklusive **Heft-CD** mit vielen Tools

Weitere Informationen und Bestellung unter [www.dotnet-magazin.de](http://www.dotnet-magazin.de)

ster nicht sicher über die angegebenen Methoden identifizieren können, kann über die Konstante `HWND_BROADCAST (0xFFFF)` auch eine Nachricht an alle Top-Level-Fenster geschickt werden.

Nachdem nun der Empfänger einer Nachricht feststeht, muss noch die Nachricht selbst versendet werden. Einerseits können Sie selbst einen Wert für die Nachricht vergeben. Windows sieht dazu Werte vor, die größer als der Wert der Konstanten `WM_USER (0x0400)` sind. Jetzt kann es aber passieren, dass auch ein anderes Programm diesen Wert für die Identifikation einer Nachricht verwendet – daher ist die Verwendung der Funktion `RegisterWindowMessage()` sinnvoll. Ihr wird eine Zeichenkette übergeben und daraufhin liefert Windows eine bisher noch unverbrauchte ID im Bereich von `0xC000` bis `0xFFFF` zurück. Ruft eine andere Anwendung diese Funktion mit derselben Zeichenkette ein weiteres Mal auf, wird die gleiche Konstante zurückgegeben. Auf diese Weise können zwei oder mehrere Anwendungen sicherstellen, dass dieselbe Konstante zum Versenden einer Nachricht genutzt wird. Die Zeichenkette kann dazu entsprechend lang und individuell gewählt werden.

Hauptproblem bei dieser Art der Kommunikation ist das Auffinden des korrekten Fensters über die Funktion `FindWindow()`. Als erster Parameter muss hier der Name der Fensterklasse übergeben werden. Wenn Sie mit dem Win32-SDK programmieren, vergeben Sie selbst diesen Namen bzw. der Name ist eindeutig identifizierbar. So lassen sich andere Anwendungen wie der Windows Explorer oder Word eindeutig auffinden. Unter .NET

wird der Name der Fensterklasse aber automatisch generiert, zum Beispiel `WindowsForms10.Window.8.app4`. Dieser kann sich dann noch nach erneutem Kompilieren ändern. Es kann also nur der zweite Parameter von `FindWindow()`, der Fenstertitel, zu einer Identifikation des Fensters genutzt werden.

```
Int32 hWnd = FindWindow(null, "Empfaenger");
if (hWnd != 0)
    ... // Fenster gefunden
```

Als Alternative bietet sich bei häufigem Nachrichtenaustausch die Wahl einer anderen Methode an, zum Beispiel eine ge-

## Hooks agieren systemweit

meinsam genutzte temporäre Datei für große Datenmengen. Sie können auch eine separate DLL für die Win32-Kommunikation erstellen. Dadurch entfallen möglicherweise die notwendigen Imports für die Funktionen des Win32-SDK (Listing 8). Eine andere Variante wird in der Dokumentation der Klasse `NativeWindow` gezeigt. Hier wird ein Fenster mit einem festen und selbst definierten Klassennamen erzeugt. Über `FindWindowEx()` kann dann dieses Fenster sicher identifiziert werden.

### Hooks

Hooks sind ein Konzept, nicht nur anwendungs-, sondern sogar systemweit bestimmte Nachrichten abzufangen und zu verarbeiten. Auf diese Weise kann ein Programm beispielsweise sämtliche Tastatur-

eingaben in allen laufenden Anwendungen kontrollieren oder nur aufzeichnen. Das .NET Framework unterstützt selbst keine systemweiten Hooks, sodass Sie auf die Methoden des Win32-SDK zurückgreifen müssen. Dies liegt unter anderem daran, dass dazu eine DLL benötigt wird, die von einem anderen Prozess aufgerufen werden muss und dazu eine spezielle Form von Funktionszeigern benötigt. Diese wird entsprechend den Anforderungen des Win32-SDK erzeugt und kann dann von einer .NET-Anwendung verwendet werden. Es gibt verschiedene Formen von Hooks. So können Sie alle Nachrichten an die Fensterfunktionen abfangen bevor bzw. nachdem Sie verarbeitet wurden. Weiterhin können Maus-, Tastatur- oder Shell-Nachrichten gefiltert werden. Weitere Informationen finden Sie unter [2].

### Fazit

Das .NET Framework bietet zahlreiche Lösungsmöglichkeiten zur Nachrichtenverarbeitung. Allerdings muss dazu häufig auf das Win32-SDK ausgewichen werden, was den resultierenden Programmcode nun nicht mehr plattformunabhängig macht und auch entsprechende Kenntnisse voraussetzt. Alle Varianten haben gemeinsam, dass sie sehr sorgfältig implementiert werden müssen, um die Ausführung einer Anwendung nicht unnötig zu bremsen bzw. den Ablauf nicht durch eine fehlerhafte Filterung zu beeinträchtigen. ●

*Dirk Frischalowski arbeitet selbstständig im Bereich der Anwendungsentwicklung mit C#, Delphi und Java. Seine Schwerpunkte liegen auf dem Schreiben von Artikeln und Dokumentationen, Vorträgen auf verschiedenen Veranstaltungen sowie dem Durchführen von Schulungen.*

### ● Links & Literatur

- [1] [www.microsoft.com/msdownload/platformsdk/sdkupdate/](http://www.microsoft.com/msdownload/platformsdk/sdkupdate/)
- [2] [www.codeproject.com/csharp/GlobalSystemHook.aspx](http://www.codeproject.com/csharp/GlobalSystemHook.aspx)  
[support.microsoft.com/default.aspx?scid=kb;EN-US;318804](http://support.microsoft.com/default.aspx?scid=kb;EN-US;318804)
- [3] Ereignisse in C#: [msdn.microsoft.com/library/DEU/csref/html/vcwlkEventsTutorial.asp](http://msdn.microsoft.com/library/DEU/csref/html/vcwlkEventsTutorial.asp)
- [4] Windows-Nachrichtenverarbeitung: [pronix.linuxdelta.de/C/win32/win32\\_2.shtml](http://pronix.linuxdelta.de/C/win32/win32_2.shtml)

### Listing 8

#### Importieren von Funktionen des Win32-SDK

```
[DllImport("user32.dll", EntryPoint="SendMessage")]
private static extern int SendMessage(int hwnd, int wMsg, int wParam, int lParam);
[DllImport("user32.dll", EntryPoint="SendMessage", CharSet=CharSet.Unicode)]
private static extern int SendMessage(IntPtr hwnd, Int32 wMsg, Int32 wParam, string lp);
[DllImport("user32.dll", EntryPoint="PostMessage")]
private static extern int PostMessage(int hwnd, int wMsg, int wParam, int lParam);
[DllImport("user32", EntryPoint="RegisterWindowMessage")]
public static extern int RegisterWindowMessage(string lpString);
[DllImport("user32.dll", EntryPoint="FindWindow")]
private static extern int FindWindow(string className, string windowName);
```