C# API für Windows 7

Übersicht

Mit der Einführung von Windows 7 hat Microsoft 36 neue und erweiterte API Funktionen bereitgestellt, die Entwickler auch in eigenen Anwendungen verwenden können. Zwar sind Eigenschaften wie Hyper-V oder Multitouch nicht umbedingt für jede Anwendung geeignet, aber es gibt andere, kleine Erweiterungen, mit denen bestehende Programme aufgewertet werden können.
Der Schwerpunkt dieser Lektion liegt daher in den verbesserten Möglichkeiten der neuen Taskbar, da mit ihnen die Bedienung nahezu jeder Anwendung verbessert werden kann. Ausserdem erfahren Sie wissenswertes über die neuen standardisierten Dialogfenster, die Ihnen Windows jetzt zur Verfügung stellt.

Die Themen im einzelnen:

API Code Pack
Überprüfung der Plattform
Overlay Icons
Progress Bar
Application ID
TaskDialog
TaskDialog mit Timer
TaskDialog mit Auswahl
CommonFileDialog
Jumplist
ThumbnailClip
Download

API Code Pack

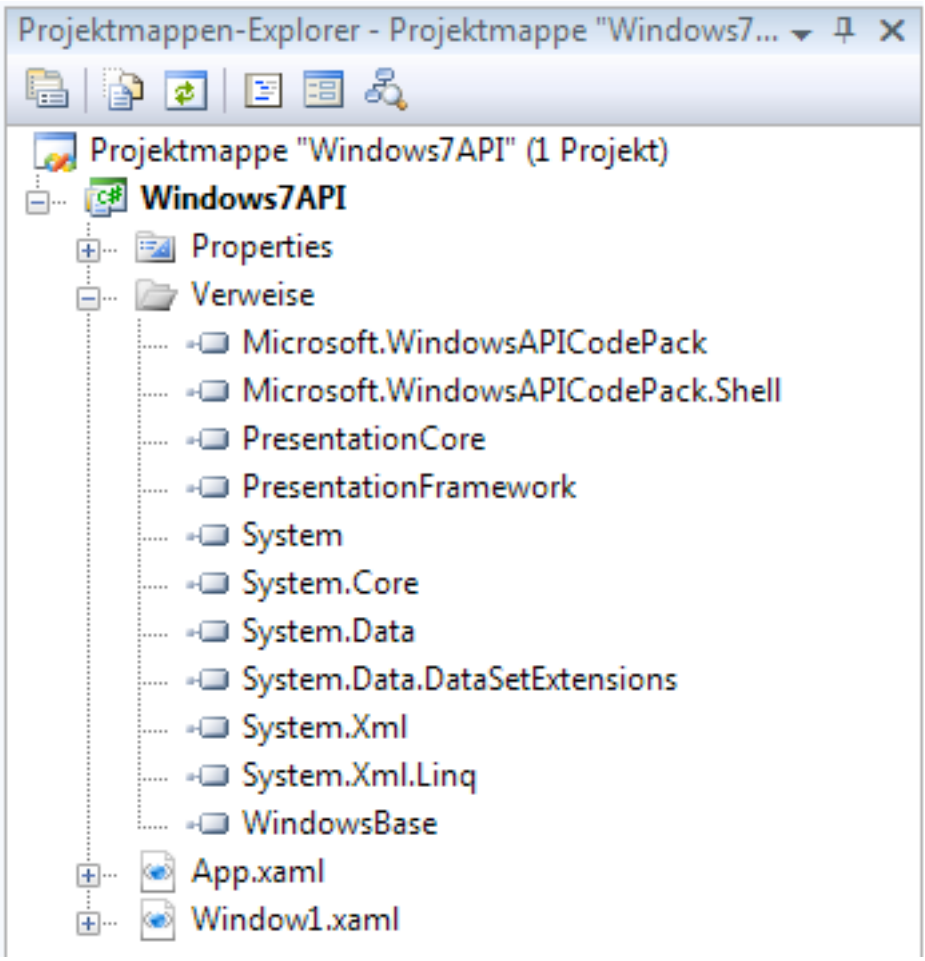
Leider sind die neuen API Funktionen nicht über Objekte und Methoden in der aktuellen Version des .NET Framework abgebildet. Um aber nicht umständlich Bibliotheken des Betriebssystems importieren zu müssen, gibt es von Microsoft das Windows API Code Pack als kostenlosen Download. Dabei handelt es sich um Managed Code Bibliotheken, die problemlos in bestehende .NET-Anwendungen eingefügt werden können.

Windows® API Code Pack for Microsoft® .NET Framework

http://code.msdn.microsoft.com/WindowsAPICodePack

Der Download enthält neben einigen Beispielen auch den C# Sourcecode des API Code Pack. Um an verwendbare dll-Dateien zu kommen, muss der Code zuvor kompiliert werden. Das sollte aber kein Problem darstellen, da er in Form eines Visual Studio Projekt vorliegt. Nach Abschluss erhält man die Microsoft.WindowsAPICodePack.dll und die Microsoft.WindowsAPICodePack.Shell.dll ,auf die man dann auch in eigenen Projekten verweisen kann.
Natürlich kann das API Code Pack auch mit Visual Basic.NET Projekten verwendet werden, die folgenden Beispiele sind allerdings in C# programmiert. Wie Sie auf der folgenden Abbildung erkennen können, handelt es sich um ein WPF-Projekt mit dem Namen Win7API, dem die beiden Verweise auf die Bibliotheken schon hinzugefügt

wurden. Die Verwendung in Windows Presentation Foundation Anwendungen ist nicht immer unkompliziert, teilweise gibt es Namenskonflikte bei Klassenbezeichnungen und andere Probleme, die ein wenig mehr Arbeit erfordern. Sie können die Bibliotheken aber natürlich auch in Windows Forms Anwendungen verwenden.



Wie bei allen Verweisen muss nachträglich der Namespace geladen, wenn man auf die neuen Methoden und Objekte zugreifen will. Das bedeutet in diesem Fall:

```
using Microsoft.WindowsAPICodePack;
using Microsoft.WindowsAPICodePack.Taskbar;
```

Möchte man nur die Funktionen der Taskbar benutzen, reicht die eine zugehörige using Direktive aus. Auch ist dann nur ein Verweis auf die Microsoft.WindowsAPICodePack.Shell.dll nötig. Für die Verwendung der neue Dialogfenster, die ebenfalls auf den folgenden Seiten vorgestellt werden, benötigt man wiederum einen zusätzlichen Namespace.

```
using Microsoft.WindowsAPICodePack.Dialogs;
```

Inwieweit sich die einzelnen Anweisungen im API Code Pack ändern werden, wenn die Funktionen ihrer Weg ins kommende .NET Framework gefunden haben, bleibt abzuwarten. Bis es so weit ist, bleibt das API Code Pack eine wertvolle Hilfe.

Überprüfung der Plattform

Natürlich muss vor der Verwendung der neuen API Funktionen überprüft werden, ob das verwendete Betriebssystem diese auch unterstützt. Als Entwickler weiss man leider nie im Voraus auf welchem Betriebssystem eine Anwendung letztendlich ausgeführt wird. Die Überprüfung ist zum Glück recht einfach und erfordert nur wenige Zeilen Programmcode.

```
if(!TaskbarManager.IsPlatformSupported)
{
    MessageBox.Show("Diese Anwendung benötig Windows 7.");
}
```

Ob man in solch einem Fall das Programm beendet oder die neuen Funktionen nur nicht zu Verfügung stellt, sollte jeder Entwickler für sich entscheiden. Noch ist Windows 7 nicht weit verbreitet und die neuen Taskbar Funktionen auch nicht zwingend erforderlich. Wahrscheinlich wäre es daher besser, alle erweiterten Funktionen in eine Klasse zu kapseln und nur dann freizugeben, wenn die Plattform es erlaubt. So kann der Benutzer weiterhin mit der Anwendung arbeiten und kommt automatisch in den Genuss der neuen Funktionen, wenn er Update des Betriebssystems durchführt.

Der Start einer WPF-Anwendung könnte beispielsweise so aussehen:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Windows;
using Microsoft.WindowsAPICodePack.Taskbar;

namespace Windows7API
{
        public class WPFApplication : Application
        {
                [STAThread]
                public static void Main()
                {
                        WPFApplication myApp = new WPFApplication();

                        MainWindow myWindow = new MainWindow();

                        if (!TaskbarManager.IsPlatformSupported)
                        {

                                MessageBox.Show("Diese Anwendung benötig Windows 7");
                                myApp.Shutdown(0);
                        }
                        myApp.ShutdownMode = ShutdownMode.OnMainWindowClose;
                        myApp.Run(myWindow);
                }
        }
}
```

Als Alternative, auch unabhängig vom API Code Pack, lässt sich die Version des Betriebssystems auf folgende Art ermitteln:

```
Environment.OSVersion.Version.Major >= 6;
```

Overlay Icons

Ein neues nützliches Feature von Windows 7 und dem Code Pack API ist die Möglichkeit, Overlay Icons in der Taskbar anzuzeigen. Hierbei wird das normale Programmsysbol um ein weiteres kleines Symbol erweitert. So kann der Benutzer jederzeit von der Anwendung Informationen erhalten, selbst dann, wenn alle Fenster minimiert sind. Prinzipiell eignen sich alle Symbole auch als Overlay Icon. Sie werden automatisch positioniert und skaliert, so das man sich darüber keine Gedanken machen muss.

Auch die Overlay Icons werden wieder durch das Objekt TaskbarManager gesteuert. In der folgenden Anweisung wird ein Icon aus den Resourcen des Programms verwendet.

```
TaskbarManager.Instance.SetOverlayIcon(Properties.Resources.favourite, "Favourite");
```

Die Icons werden tatsächlich eingeblendet, das heisst, sie werden langsam sichtbar. Genau so verhält es sich, wenn man das Overlay Icon wieder entfernen möchte. Das Symbol verschwindet langsam.

```
TaskbarManager.Instance.SetOverlayIcon(null, "");
```

Doch Vorsicht! Overlay Icons werden nur dargestellt, wenn Taskbar große Icons anzeigt. Das ist zwar die Standarteinstellung von Windows 7, kann aber durch den Anwender verändert werden. Überprüfen Sie daher diese Einstellung, wenn Sie keine Overlay Icons sehen.

Man sollte den Anwender aber nicht mit Overlay Icons überfordern und ständig andere Symbole anzeigen. Auch sollte man keine blinkenden Symbole programmieren, obwohl das sicherlich mit geringem Aufwand möglich wäre. Die Overlay Icons dienen in erster Linie dazu, längerfristige Statusmeldungen anzuzeigen. Zum Beispiel die erfolgreiche Verbindung zu einer Datenbank oder eine umfangreiche Aufgabe welche die Anwendung eine Zeit lang beschäftigen wird.

Progress Bar

Fortschrittsbalken sind seit je her ein beliebtes Mittel, um Anwender über den Status eines länger dauernden Vorganges zu informieren. Leider erhält der Anwender selten eine

Mitteilung nachdem dieser Vorgang abgeschlossen wurde und ihm fehlen sämtliche Informationen, sollte er das Programmfenster minimieren oder zu einer anderen Anwendung wechseln.
Mit Windows 7 gehören auch diese Probleme der Vergangenheit an, denn jetzt kann eine Anwendung auch in der Taskbar einen Fortschrittsbalken anzeigen. Dabei gibt es sogar unterschiedlichen Möglichkeiten der Konfiguration.

Die einfachste Verwendung eines Fortschrittsbalken ist der kontinuierliche Anstieg von einem minimalen auf einen maximalen Wert. Wie auch bei einem normalen Progress Bar, wird das Taskbar-Symbol komplett ausgefüll,t wenn der maximale Wert erreicht ist.

```
int minValue = 0;
int maxValue = 100;

for (int value = minValue; value < maxValue; value++)
{
        TaskbarManager.Instance.SetProgressValue(value, maxValue);

        // Pause von 10 ms.
        System.Threading.Thread.Sleep(10);
}
```

Zusätzlich zum regulären grünen Fortschrittsbalken lassen sich durch weitere Farben zusätzliche Informationen an den Anwender übermitteln.

Einen gelben Fortschrittsbalken erreicht man durch den Aufruf von SetProgressState mit dem Parameter TaskbarProgressBarState.Paused.

```
TaskbarManager.Instance.SetProgressState(TaskbarProgressBarState.Paused);
```

Benötigen Sie indes einen roten Fortschrittsbalken, geht das mit dem folgen Befehl.

```
TaskbarManager.Instance.SetProgressState(TaskbarProgressBarState.Error);
```

Als vierte Variation haben Sie die Möglichkeit, den Progress Bar im Status Indeterminate darzustellen. Jetzt können keine konkreten Werte mehr angezeigt werden, Sie erhalten stattdessen eine kontinuierliche Animation in der Progress Bar in Form eines wandernden Farbfeldes.

```
TaskbarManager.Instance.SetProgressState(TaskbarProgressBarState.Indeterminate);
```

Zurücksetzten lässt sich der Progress Bar mit der Anweisung:

```
TaskbarManager.Instance.SetProgressState(TaskbarProgressBarState.NoProgress);
```

ApplicationID

Die Application ID gibt dem Entwickler die Möglichkeit die zusätzlichen Fenster einer Anwendung in der Taskbar zu gruppieren. Normalerweise werden alle weiteren Fenster, die ein Programm vielleicht öffnet auch dieser Anwendung zugeordnet. Hat ein Programm beispielsweise mehrere geöffnete Fenster, wird das auch in der TaskBar deutlich.

Dabei handelt es sich um normale Fenster, die auf ganz normale Art und Weise erzeugt und angezeigt werden. In diesem Beispiel wird eine Instanz der Klasse ChildWindow erzeugt und angezeigt.

```
ChildWindow childWin = new ChildWindow();
childWin.Show();
```

Möchten man aber, dass die Kindfenster einer separaten Gruppe, und nicht dem Hauptfenster untergeordnet angezeigt werden, muss man mit der Application ID arbeiten. Dabei ist es vollkommen, egal wie diese ID aussieht, solange sich die IDs vom Haupt - und Kindfenster unterscheiden.

Die Zuweisung der Application ID erfolgt am besten im Konstruktor des Fensters. Direkt nach InitializeComponent.

Die Zuweisung für das Hauptfenster könnte so aussehen:

```
TaskbarManager.Instance.ApplicationId = "MyCompany.MyApplication";
```

Für die Kindfenster würde folgende Anweisung ausreichen:

```
TaskbarManager.Instance.ApplicationId = "MyCompany.MyApplication.ChildWindow";
```

Nach dieser minimalen Erweiterung im Programmcode werden die Fenster der Anwendung in der Taskbar jetzt in zwei Gruppen aufgeteilt. Die folgende Abbildung zeigt wie das aussehen könnte.

TaskDialog

Als Programmierer ist man meistens froh, wenn man eine Anwendung nicht von Grund auf selbst entwickeln muss, sondern stattdessen auf vorgefertigte Bauteile aus dem Windows-Baukasten zugreifen kann. Eine nützliche Gruppe dieser Bausteine sind die Windowseigenen Dialogfenster wie die MessageBox oder der FileOpenDialog. Diese existieren auch im .NET Framework, um den Entwicklern das Leben einfacher zu machen und standardisierte Fenster für bestimmte Aufgaben anzubieten. Auch für den Benutzer sind solche Fenster von Vorteil, da er sich nicht für jedes Programm an unterschiedliche Fenster gewöhnen muss.
Obwohl es die bekannten Dialoge auch weiterhin, gibt sind sie doch in die Jahre gekommen. Windows 7 bietet moderner Varianten der Dialogfenster an, die zwar die gleichen Aufgaben erfüllen, aber vom Design besser aussehen und teilweise auch mehr Möglichkeiten bieten. An dieser Stelle werden Sie jetzt einige, aber bei weitem nicht alle, neuen Dialoge kennenlernen.

Auch die neuen Dialogfenster sind zwar Teil der erweitern WindowsAPI, aber noch nicht von .NET Anwendungen ansprechbar. Auch hier schlägt das API Code Pack eine Brücke.

Um die erweitern Fenster zu benutzen, benötigen Sie jetzt zwingend auch den Verweis auf die Anfangs erwähnte Microsoft.WindowsAPICodePack.dll. Ausserdem sollten Sie folgende Namespace Deklaration hinzufügen.

```
using Microsoft.WindowsAPICodePack.Dialogs;
```

Jetzt können Sie in einen TaskDialog instanzieren und konfigurieren.

```
TaskDialog td = new TaskDialog();
td.Icon = TaskDialogStandardIcon.Error;
td.Caption = "Die Überschrift";

td.InstructionText = "Wichtige Mitteilung";

td.StandardButtons = TaskDialogStandardButtons.Yes |
TaskDialogStandardButtons.Cancel;

TaskDialogResult res = td.Show();
```

Achtung! Eventuell bekommen Sie bei der Ausführung eine Fehlermeldung, die besagt, dass Sie nicht die Version 6 der Common Controls Bibliothek geladen haben.

TaskDialog feature needs to load version 6 of comctl32.dll but a different version is current loaded in memory.

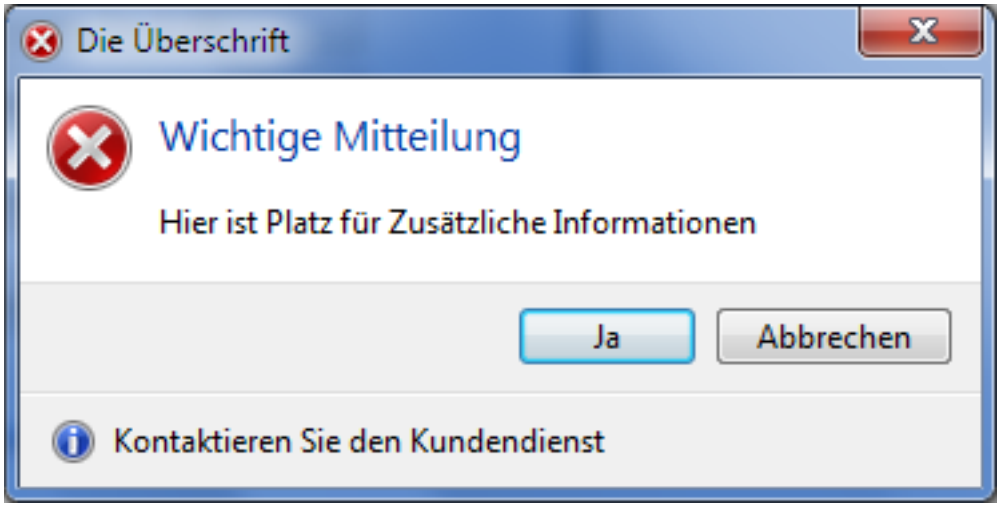
Diese Meldung bezieht sich auf die verwendet Version der Windows Steuerelemente Bibliothek. Die neuste Version wird nämlich nicht automatisch verwendet.
Alles, was man tun muss, ist die Anwendung anzuweisen, diese Bibliothek zu laden. Falls noch nicht vorhanden, fügen Sie die dem Projekt eine Manifestdatei hinzu, und erweitern Sie das Manifest um die folgende Abhängigkeit.

```
<dependency>
	<dependentAssembly>
		<assemblyIdentity
			type="win32"
			name="Microsoft.Windows.Common-Controls"
			version="6.0.0.0"
			processorArchitecture="*"
			publicKeyToken="6595b64144ccf1df"
			language="*"/>
	</dependentAssembly>
</dependency>
```

Falls Sie weitere Informationen benötigen, finden Sie auch Manifeste in den Beispielen und Projekten des API Code Pack. Dann steht der Verwendung des TaskDialogs nichts mehr im Weg.

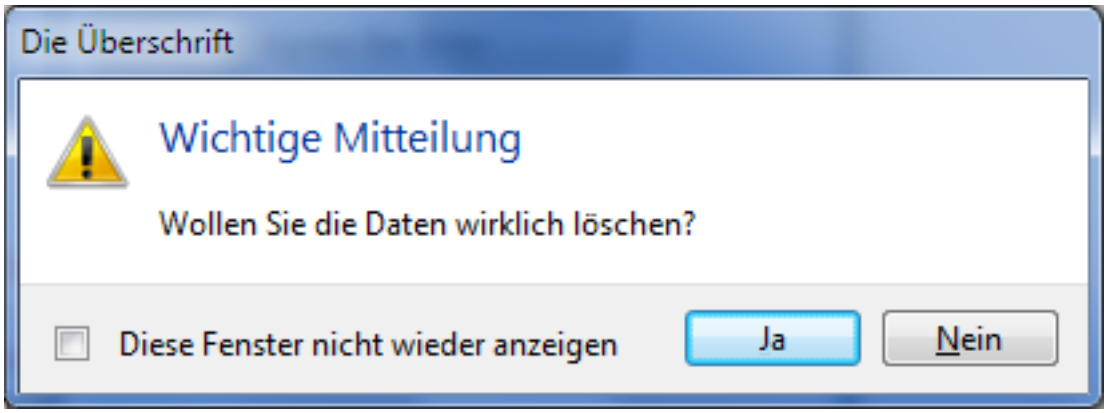
Durch erweiterte Konfiguration können in diesem Dialogfenster sogar noch mehr Informationen angezeigt werden.

```
TaskDialog td = new TaskDialog();
td.Icon = TaskDialogStandardIcon.Error;
td.Caption = "Die Überschrift";
td.Text = "Hier ist Platz für Zusätzliche Informationen";
td.InstructionText = "Wichtige Mitteilung";
td.FooterText = "Kontaktieren Sie den Kundendienst";
td.FooterIcon =
TaskDialogStandardIcon.Information;
td.StandardButtons =
TaskDialogStandardButtons.Yes | TaskDialogStandardButtons.Cancel;
```



Ein typischer Anwendungsfall ist es, den Benutzer zu fragen, ob er Meldungen dieser Art in Zukunft nicht mehr sehen möchte. Mit dem TaskDialog ist das sehr leicht möglich, denn er erlaubt es, eine CheckBox mit anzuzeigen.

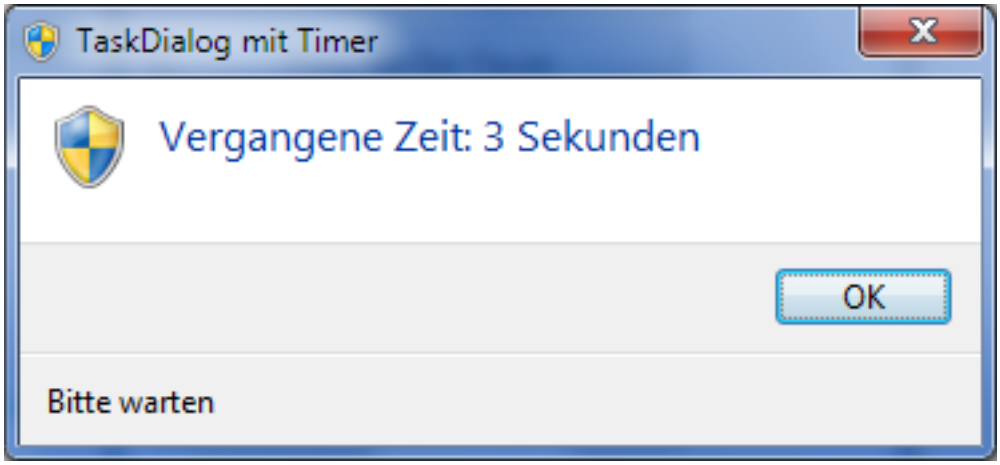
```
TaskDialog td = new TaskDialog();
td.Icon = TaskDialogStandardIcon.Warning;
td.Caption = "Die Überschrift";
td.Text = "Wollen Sie die Daten wirklich löschen?";
td.InstructionText = "Wichtige Mitteilung";
td.FooterCheckBoxText = "Diese Fenster nicht wieder anzeigen";
td.FooterIcon = TaskDialogStandardIcon.Information;
td.StandardButtons =
TaskDialogStandardButtons.Yes | TaskDialogStandardButtons.No;

TaskDialogResult res = td.Show();

if (td.FooterCheckBoxChecked == true)
{

}
```



TaskDialog mit Timer

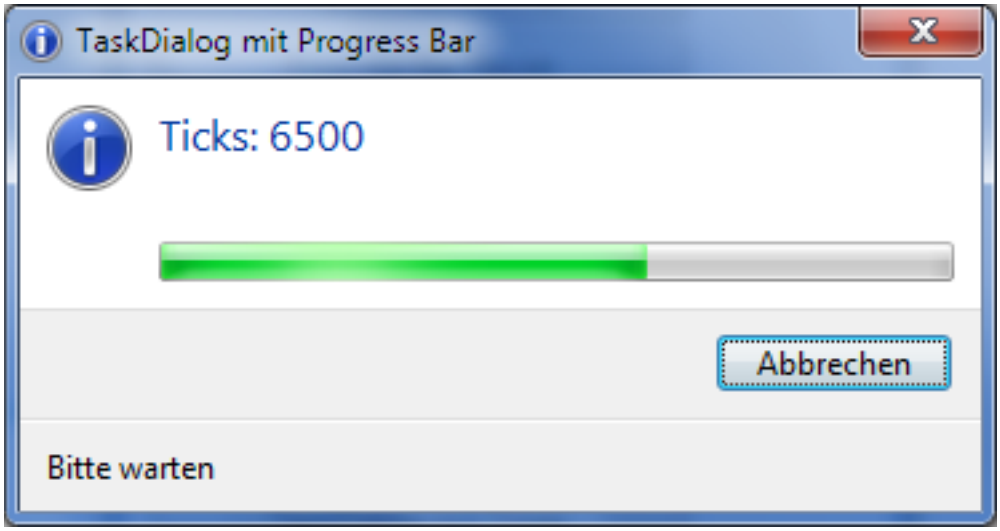
Es war für Entwickler schon immer ein Problem, dass sich der Inhalt eines Benachrichtigungsfenstern nicht anpassen lies und man von Klassen wie MessageBox nicht ableiten könnte. Benötigte man zusätzliche Elemente, wie vielleicht einen Fortschrittsbalken oder einen Countdown in einem Mitteilungsfenster, musste man selbst programmieren und ein von Grund auf eigenes und neues Fenster konstruieren. Mit dem neuen TaskDialog ist zumindest das jetzt nicht mehr nötig, denn durch einen eingebauten Timer können Events ausgelöst werden, mit denen dann auch der Inhalt des TaskDialog-Fensters selbst manipuliert werden kann. Es ist sogar möglich, dass sich das Fenster nach eine genau definierten Zeit selbstständig schliesst und dabei einen zuvor festgelegten TaskDialogResult zurückgibt. Es große Erleichterung für viele Routineaufgaben im Programmieralltag.

```
private void btnTaskDialogTimer_Click(object sender, RoutedEventArgs e)
{
        TaskDialog td = new TaskDialog();
        td.Icon = TaskDialogStandardIcon.Warning;
        td.Tick +=  new EventHandler<TaskDialogTickEventArgs>(td_Tick);
        td.InstructionText = "";
        td.Icon = TaskDialogStandardIcon.Shield;
        td.Caption = "TaskDialog mit Timer";
        td.FooterText = "Bitte warten";
        td.Cancelable = true;
        td.Show();
}

void td_Tick(object sender, TaskDialogTickEventArgs e)
{
        TaskDialog td = sender as TaskDialog;
        int seconds = e.Ticks / 1000;
        td.InstructionText =
        string.Format("Vergangene Zeit: {0} Sekunden",seconds);

        if (seconds == 5)
        {
                td.Close(
                TaskDialogResult.Cancel);
        }
}
```



Möchte man im TaskDialog zusätzliche einen Fortschrittsbalken anzeigen, ist auch das mit geringem Aufwand möglich.

```
int  maxValueTicks = 10000;

private void btnTaskDialogProgressTimer_Click(object sender, RoutedEventArgs e)
{
        TaskDialog tdprogress = new TaskDialog();
        tdprogress.Icon = TaskDialogStandardIcon.Information;
        tdprogress.FooterText = "Bitte warten";
        tdprogress.Caption = "TaskDialog mit Progress Bar";
        tdprogress.Tick +=
        new EventHandler<TaskDialogTickEventArgs>(tdprogress_Tick);
        tdprogress.Cancelable = true;
        tdprogress.StandardButtons = TaskDialogStandardButtons.Cancel;
        TaskDialogProgressBar progressBar
        = new TaskDialogProgressBar(0, maxValueTicks, 0);
        tdprogress.ProgressBar = progressBar;
        tdprogress.Show();
}

void tdprogress_Tick(object sender, TaskDialogTickEventArgs e)
{
        TaskDialog td = sender as TaskDialog;
        if (maxValueTicks >= e.Ticks)
        {
                td.InstructionText =
                string.Format("Ticks: {0}", e.Ticks);
                td.ProgressBar.Value = e.Ticks;
        }
        else
        {
                td.InstructionText =
                "Vorgang abgeschlossen.";
                td.FooterText = "";
                td.ProgressBar.Value = maxValueTicks;
        }
}
```

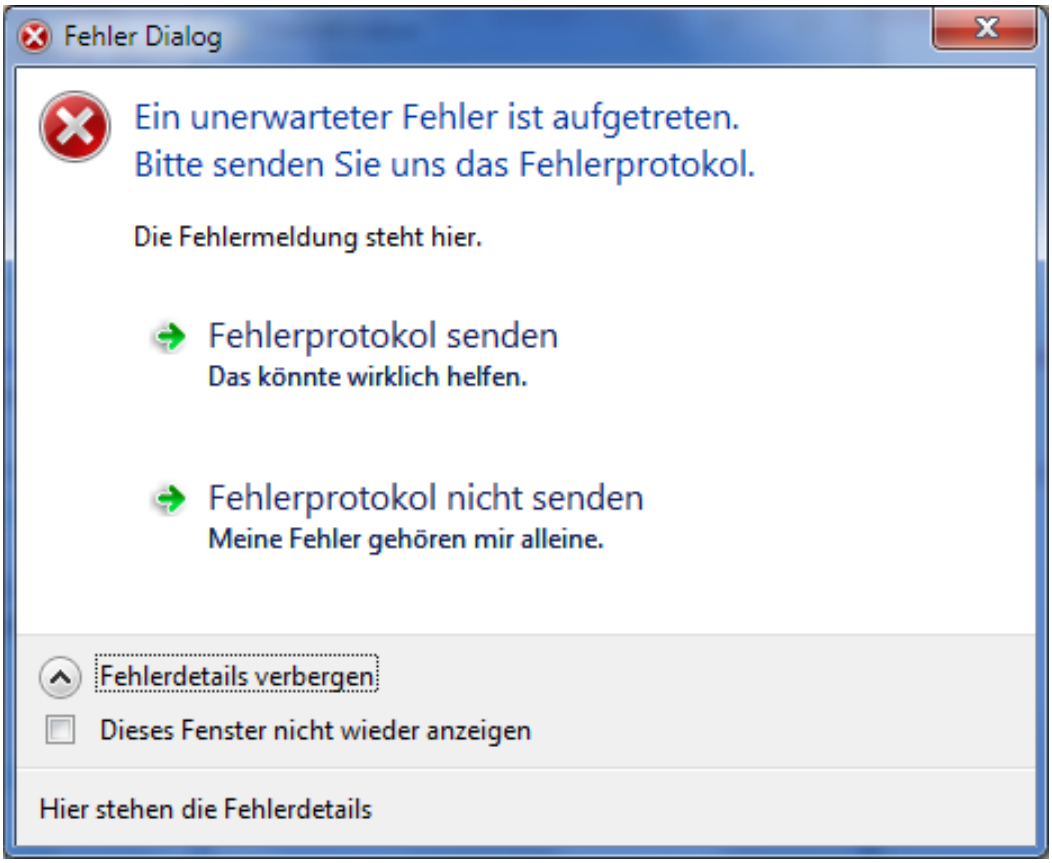


TaskDialog mit Auswahl

Eine weitere Anwendungsmöglichkeit des TaskDialog ist die Anzeige von Auswahlmöglichkeiten, wie sie zum Beispiel im Falle eines Programmfehlers nützlich sein kann. In einem konkreten Beispiel könnte ein Mitteilungsfenster dann wie folgt aussehen.

```
private void btnTaskDialogError_Click(object sender, RoutedEventArgs e)
{
        TaskDialog tdErrorDialog = new TaskDialog();
        tdErrorDialog.DetailsExpanded = false;
        tdErrorDialog.Cancelable = true;
        tdErrorDialog.Icon = TaskDialogStandardIcon.Error;
        tdErrorDialog.Caption ="Fehler Dialog";
        tdErrorDialog.InstructionText =
        "Ein unerwarteter Fehler ist aufgetreten.\nBitte senden Sie uns das
        Fehlerprotokol.";

        tdErrorDialog.Text = "Die Fehlermeldung steht hier.";
        tdErrorDialog.DetailsExpandedLabel ="Fehlerdetails verbergen";
        tdErrorDialog.DetailsCollapsedLabel = "Fehlerdetails anzeigen";
        tdErrorDialog.DetailsExpandedText = "Hier stehen die Fehlerdetails";
        tdErrorDialog.FooterCheckBoxText ="Dieses Fenster nicht wieder anzeigen";
        tdErrorDialog.FooterCheckBoxChecked = false;
        tdErrorDialog.ExpansionMode =
        TaskDialogExpandedDetailsLocation.ExpandFooter;
        TaskDialogCommandLink sendBtn =
         new TaskDialogCommandLink
        ("sendButton",
         "Fehlerprotokol senden\nDas knnte wirklich helfen.");
        sendBtn.Click += new EventHandler(sendBtn_Click);

        TaskDialogCommandLink dontSendBtn = new TaskDialogCommandLink
        ("dontSendButton",
        "Fehlerprotokol nicht senden\nMeine Fehler gehren mir alleine.");

        dontSendBtn.Click += new EventHandler(dontSendBtn_Click);
        tdErrorDialog.Controls.Add(sendBtn);
```

```
        tdErrorDialog.Controls.Add(dontSendBtn);
        tdErrorDialog.Show();
}
```



Der gezeigte Code erzeugt zwei TaskDialogCommandLink, die dem Fenster als Steuerel emente angehängt werden. Verbunden damit sind zwei Events, die aufgerufen werden, wenn der Anwender eine der beiden Auswahlmöglichkeiten anklickt.

Die Auswertung der Ereignisse ist dann wieder Aufgabe des Entwicklers.

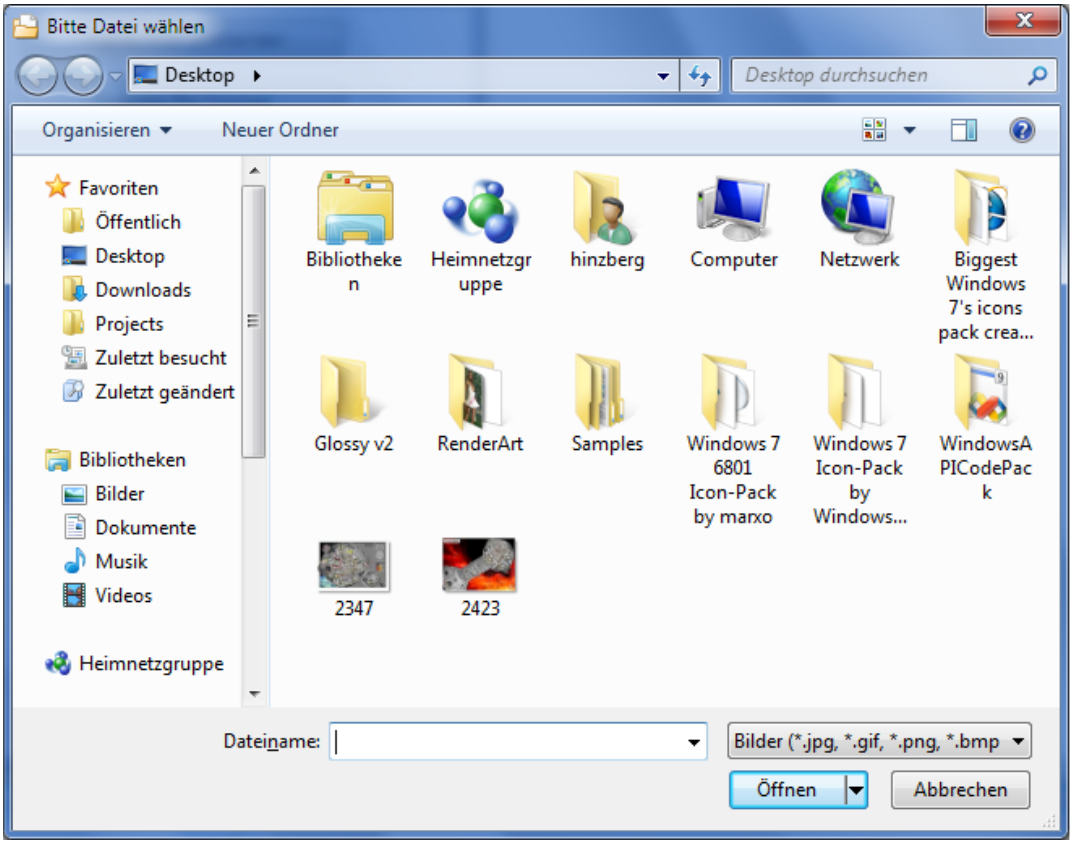
```
void dontSendBtn_Click(object sender, EventArgs e)
{
        TaskDialog td = sender as TaskDialog;
        td.Close( TaskDialogResult.Cancel);
}

void sendBtn_Click(object sender, EventArgs e)
{
        // Etwas sinnvolles tun.
}
```

CommonFileDialog

Ebenfalls eine Überarbeitung erfuhren die Dialogfenster zur Auswahl von Dateien. Diese heissen jetzt CommonOpenFileDialog und CommonSaveFileDialog. Während Sie in der Verwendung unverändert blieben, hat sich das Design in der Seitenleiste deutlich geändert und die Bibliotheken und Favoriten stehen jetzt direkt zu Verfügung.

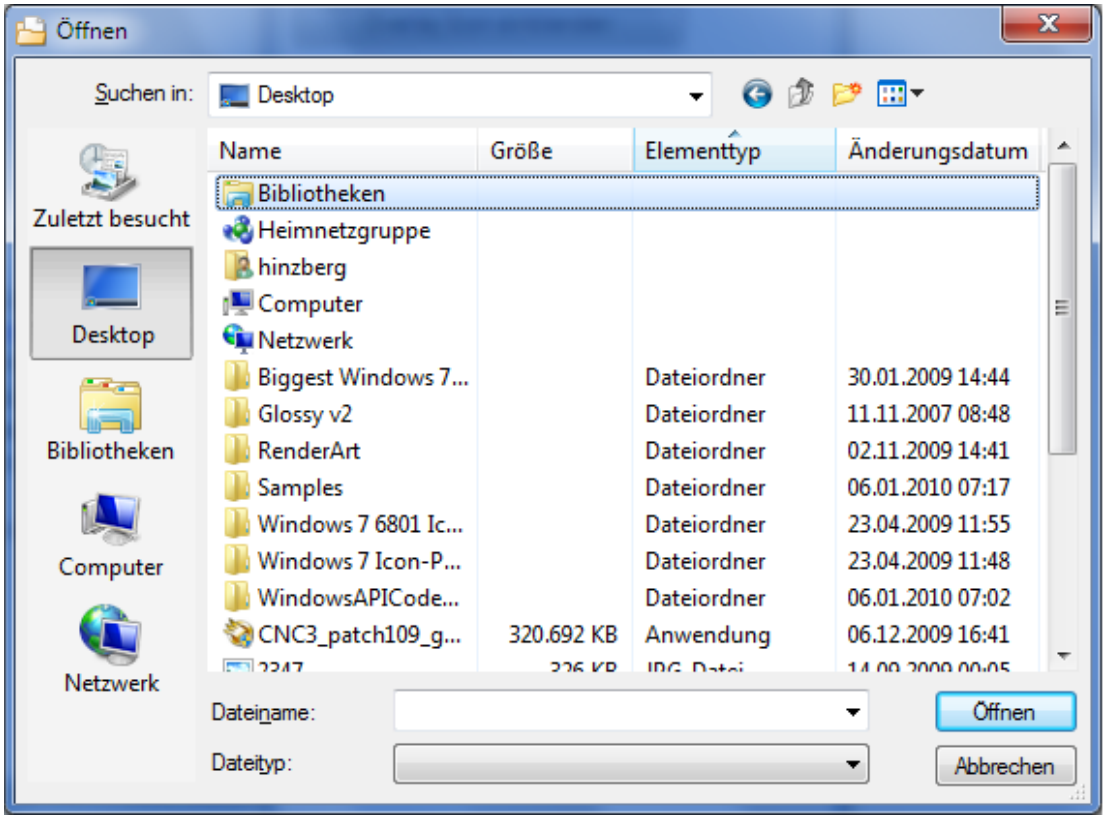
Der folgende Codeausschnitt soll nur als kleine Anregung dienen, die Möglichkeiten dieser Dialoge sind sehr komplex und die gezeigten Parameter sind bei weitem nicht alle Eigenschaften, die man konfigurieren kann.

```
CommonOpenFileDialog COpenFileDialog
= new CommonOpenFileDialog("Bitte Datei whlen");

COpenFileDialog.Filters.Add(
new CommonFileDialogFilter("Bilder", "*.jpg,*.gif,*.png,*.bmp"));

COpenFileDialog.Multiselect = false;
COpenFileDialog.EnsureFileExists = true;
COpenFileDialog.IsFolderPicker = false;
COpenFileDialog.InitialDirectory = KnownFolders.Desktop.Path;

if (COpenFileDialog.ShowDialog() == CommonFileDialogResult.OK)
{
    // ...
}
```



Im Vergleich dazu eine Abbildung des OpenFileDialog aus dem Win32 Namespace, wie es ihn bisher gab und aus Gründen der Kompatibilität auch weiterhin geben wird.

```
Microsoft.Win32.OpenFileDialog OpenFileDia = new Microsoft.Win32.OpenFileDialog();
OpenFileDia.ShowDialog();
```



Ebenfalls eine Neuerung ist die Klasse KnowFolders. Dahinter versteckt sich eine Auflistung einer auf Windowssystemen bekannter, und oft verwendeter Ordner, wie Desktop, Documents, Downloads, Computer und viele mehr.

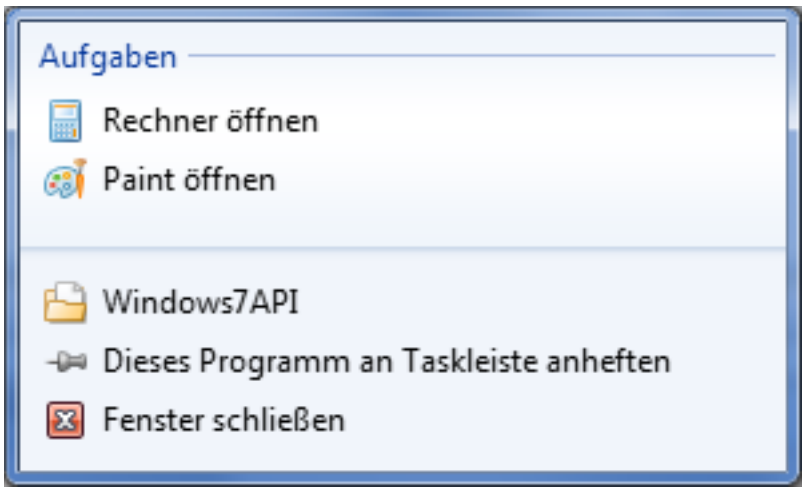
Jumplist

Die Jumplist ist ebenfalls ein neues Feature, das mit Windows 7 eingeführt wurde. Unter einer Jumplist versteht man ein Kontextmenü, das eingeblendet wird, wenn der Anwender mit der rechten Maustaste auf ein TaskBar Symbol klickt. Die Jumplist soll den Start einer Anwendung beschleunigen, indem sie zum Beispiel eine Liste der zuletzt verwendeten Dateien anzeigt oder eine Anwendung mit bestimmten Parametern startet.
Es ist aber genau so gut möglich, über die Jumplist einer Anwendung auch andere Programme zu starten. Alles, was der Entwickler tun muss, ist eine neue JumpList zu erzeugen und dieser Liste Objekte vom Typ JumpListLink hinzuzufügen. Mit etwas Geschick kann man sich das anzuzeigende Icon direkt aus der Anwendung selbst holen.

```
JumpList list = JumpList.CreateJumpList();
list.ClearAllUserTasks();

string systemFolder = Environment.GetFolderPath(Environment.SpecialFolder.System);

list.AddUserTasks(
        new JumpListLink(System.IO.Path.Combine(systemFolder, "calc.exe"),
        "Rechner öffnen")
{
        IconReference =
        new IconReference(System.IO.Path.Combine(systemFolder, "calc.exe"), 0)
});
```

```
list.AddUserTasks(
        new JumpListLink(System.IO.Path.Combine(systemFolder, "mspaint.exe"),
        "Paint öffnen")
{
        IconReference =
        new IconReference(System.IO.Path.Combine(systemFolder, "mspaint.exe"), 0)
});
list.Refresh();
```

Es ist wichtig, die Jumplist erst dann zu erzeugen, wenn die Anwendung schon ein geladenes Fenster hat. In einer WPF Anwendung bietet sich dafür die Methode Window_Loaded an. Versucht man hingegen die JumpList schon im Konstruktor des Fensters zu erstellen, ist das wenig erfolgreich und führt zu einem Programmabbruch.



Ein auf diese Art erstellte Jumplist bleibt erhalten, nachdem die Anwendung beendet wurde. Sie können das sehr leicht selbst ausprobieren, indem Sie die von Visual Studio erstellte Anwendung an der TaskBar anheften und anschliessend beenden. Wenn Sie nun auf mit der rechten Maustaste auf das Symbol klicken, wird die Jumplist wieder dargestellt. Wählen Sie jetzt den Eintrag "Paint öffnen", wird das Programm Paint gestartet. Die zur Jumplist gehörende Anwendung bleibt allerdings weiterhin geschlossen. Man könnte also auch über Jumplisten Schnellstart Funktionen für Programme verwirklichen, indem man eine Anwendung entwickelt, die nichts anderes tut, als eine Jumplist zur Verfügung zu stellen.

Etwas ungewöhnlich sind in diesem Beispiel auch die Zuweisung der Symbole zu den Einträgen. Diese werden nicht wie gewohnt als Icon, sondern als IconReference erzeugt, denn sie werden tatsächlich von den anderen Anwendung geholt. Möchte man das Symbol des eigenen Programms anzeigen, ist ein etwas anderer Weg nötig.

```
IconReference =
new IconReference(Assembly.GetEntryAssembly().Location, 0);
```

In der Regel will man mit einer Jumplist aber nicht fremde, sondern die eigene Anwendung mit bestimmten Parametern starten. Dafür ist es zunächst nötig, einen JumpListLink zu erzeugen, der auf die eigenen Anwendung verweist.

```
string myExecutablePath = Assembly.GetEntryAssembly().Location;
JumpListLink myJumpLink = new JumpListLink(myExecutablePath, "Befehl A ausfhren");

myJumpLink.IconReference =
new IconReference(Assembly.GetEntryAssembly().Location, 0);
myJumpLink.Arguments ="Command-A";
list.AddUserTasks(myJumpLink);
list.Refresh();
```

Wird später auf diesen JumpListLink geklickt, startet das Programm mit dem Commandozeilenparameter "Command-A". Diesen Parameter kann man in der main-Methode der Anwendung auswerten und dann entsprechend darauf reagieren.

```
[STAThread]
public static void Main()
{
	string[] args = Environment.GetCommandLineArgs();
	if (args.Count() > 1)
	{
		if (args[1] == "Command-A")
		{
			MessageBox.Show("Command-A");
		}
	}
//...
```

Aber Achtung! Wird bei einer schon laufenden Anwendung erneut auf diesen Link geklickt, ist es wahrscheinlich, dass sich eine weitere Instanz der Anwendung öffnet. Das ist nicht immer im Sinne des Entwicklers und mit einem einfachen Mutex kann verhindert werden, dass mehrere Instanzen einer Anwendung gleichzeitig ausgeführt werden können. Wichtig ist, dass der Mutex statisch und ausserhalb der Main-Methode angelegt wird.

```
static Mutex mutex = new Mutex(false, "Win7ApiDemo");

[STAThread]
public static void Main()
{
	if (!mutex.WaitOne(TimeSpan.FromMilliseconds(500), false))
	{
		MessageBox.Show("Ein andere Instanz läuft schon");
	return;
}

//...
```

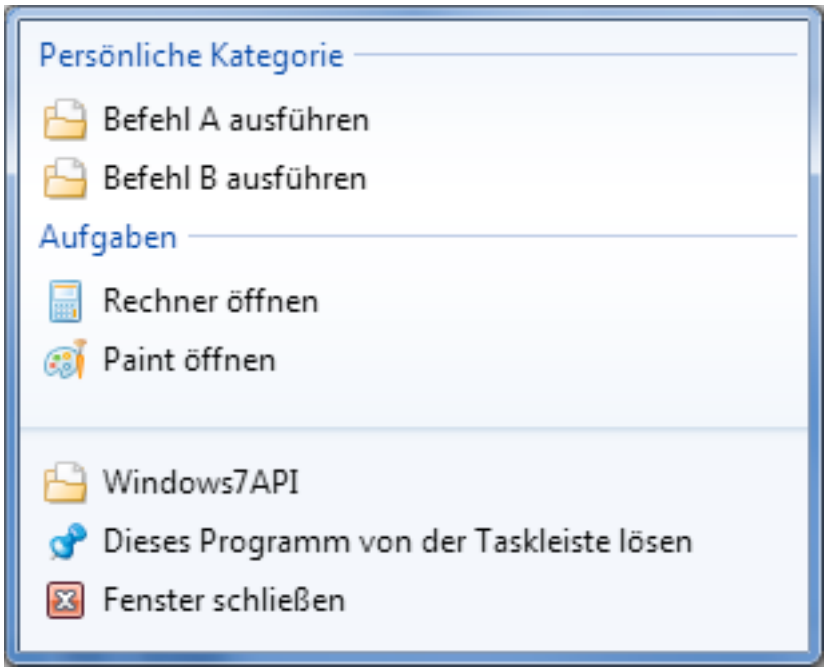
Es ist durchaus möglich mit einer Jumplist auch Funktionen in einer schon gestarteten Anwendung auszuführen. Dazu muss aber zuerst ermittelt werden, ob schon eine Instanz dieser Anwendung läuft, und die Anweisungen werden dann entsprechend durchgereicht. Zwei Verfahren wie so etwas funktionieren könnte finden Sie unter den folgenden Links.

http://www.codeproject.com/KB/statusbar/Clipz.aspx

http://www.developer.com/net/article.php/10916_3850661_2/Creating-Windows-7-Jump-Lists-With-The-API-Code-Pack-and-Visual-Studio-2008.htm

Um etwas mehr Ordnung in die Jumplist zu bekommen ist es möglich, eigene Kategorien zu erzeugen und dort dann JumpListItem einzufügen.

```
string myExecutablePath = Assembly.GetEntryAssembly().Location;

JumpListCustomCategory personalCategory
= new JumpListCustomCategory("Persnliche Kategorie");
list.AddCustomCategories(personalCategory);

JumpListLink myJumpLinkA =
new JumpListLink(myExecutablePath, "Befehl A ausfhren");
myJumpLinkA.IconReference =
new IconReference(Assembly.GetEntryAssembly().Location, 0);
myJumpLinkA.Arguments ="Command-A";
personalCategory.AddJumpListItems(myJumpLinkA);

JumpListLink myJumpLinkB = new JumpListLink(myExecutablePath, "Befehl B ausfhren");
myJumpLinkB.IconReference =
new IconReference(Assembly.GetEntryAssembly().Location, 0);
myJumpLinkB.Arguments ="Command-B";
personalCategory.AddJumpListItems(myJumpLinkB);

list.Refresh();
```



Eine weitere nützliche Eigenschaft einer Jumplist ist die Möglichkeit, die zuletzt verwendeten Dateien der Anwendung anzuzeigen. Auf diesem Wege kann das Programm gestartet werden und erhält gleichzeitig Informationen darüber, welche Datei es

anschliessend laden soll. Diese Liste wird natürlich nicht automatisch verwaltet, sondern ist Aufgabe des Entwicklers.
Um das ganze noch weiter zu verkomplizieren, funktioniert es nicht mit beliebigen Dateien, sondern nur mit solchen Typen, die auch für die Anwendung registriert wurden. Mehr Informationen über den etwas komplexen Vorgang der Registrierung finden Sie auch in der Beispielen des Windows 7 API Code Pack. Die von Microsoft zur Verfügung gestellten Klassen erledigen diese Aufgabe perfekt.
Sind alle Hürden genommen, können Dateien mit nur eine einzelnen Anweisung der Jumplist hinzugefügt werden.

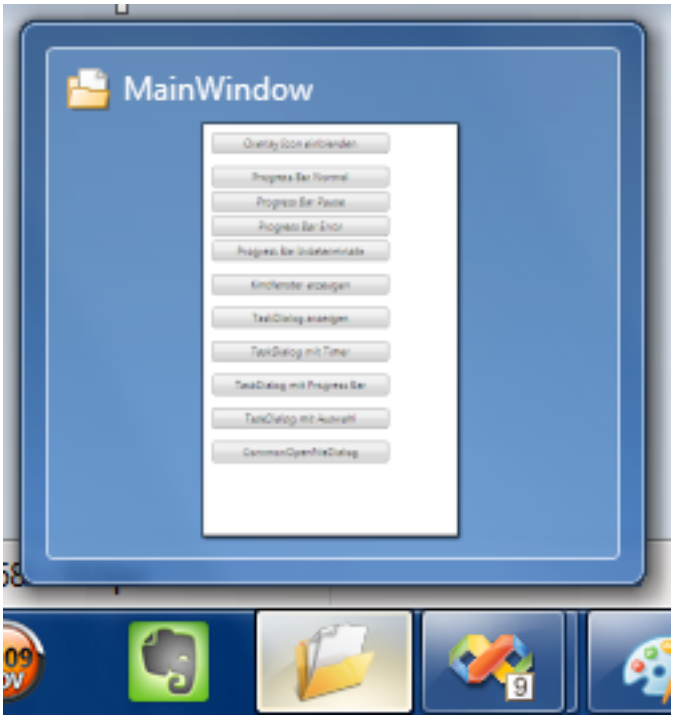
```
list.AddToRecent("c:\test1.txt");
```

Wird später auf diesen Eintrag geklickt, findet sich die zu ladenden Datei wieder in der Commandozeilenparametern.

Falls Ihnen dieser Vorgang zu kompliziert erscheint, können Sie sicherlich im einer persönlichen Kategorie das gleiche Resultat mit weniger Aufwand erreichen.

ThumbnailClip

Bewegt man den Mauspfeil über das Taskbar Symbol einer laufenden Anwendung, öffnet sich in einen Fenster eine verkleinerte Vorschau des aktuellen Programmfensters.
Diese, auch als Thumbnail bezeichnete Vorschau, muss aber nicht zwangsläufig immer das komplette Fenster anzeigen.

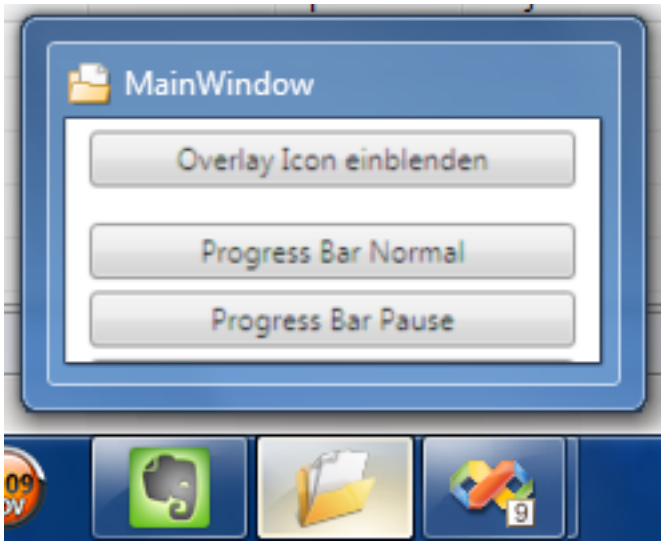


Man kann den ThumbnailClip der Windows 7 Taskbar auch so konfigurieren, dass er nur einen Teil des Fensterinhaltes anzeigt.
In einer WPF Anwendung gestaltet sich das ein wenig kompliziert, da man zur Konfiguration des ThumbnailClip ein Fensterhandle benötigt, das es in Anwendungen dieses Typs nicht so ohne weiteres gibt. Mit Hilfe des WindowInteropHelper ist es aber dennoch möglich, das Fensterhandle zu ermitteln, indem man den folgenden Code dem Fenster hinzufügt.

```
private IntPtr handle;

public IntPtr Handle
{
        get
        {
                if (this.handle == IntPtr.Zero)
                {
                        System.Windows.Interop.
                        WindowInteropHelper interopHelper =
                        new System.Windows.Interop.WindowInteropHelper(this);
                        this.handle = interopHelper.Handle;
                }
        return this.handle;
        }
}
```

Jetzt gilt es gewünschten Bildausschnitt zu finden, und der ThumbnailClip ist konfiguriert.

```
System.Drawing.
Rectangle clippingRectangle = new System.Drawing.Rectangle(

new System.Drawing.Point(0, 5),

new System.Drawing.Size(220, 100));

TaskbarManager.Instance.TabbedThumbnail.SetThumbnailClip(this.Handle,
clippingRectangle);
```



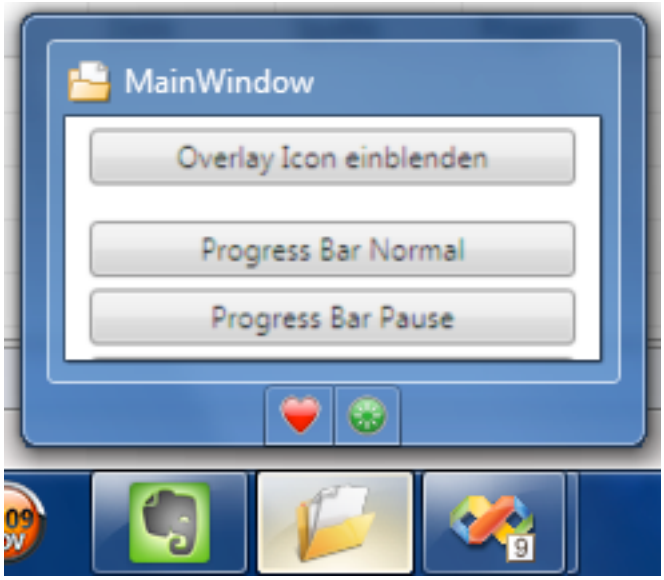
Möchte man im TumbnailClip ein bestimmtes Steuerelement anzeigen, muss man diese Postionen des Objektes nicht in mühevoller Kleinarbeit selbst ermitteln, sondern kann dies im Programmcode errechnen lassen. Wieder sind für eine WPF-Anwendung ein paar mehr Schritt notwendig, da Werte in andere Formate konvertiert werden müssen.
Im folgenden Code wir die Position der Schaltfläche btnOverlay relativ zum Fenster (this) ermittelt und ans TumbnailClip verwendet.

```
Point relativePoint
        = btnOverlay.TransformToAncestor(this).Transform(new Point(0, 0));
System.Drawing.Point drawingPoint
        = new System.Drawing.Point((int)relativePoint.X, (int)relativePoint.Y);
System.Drawing.Size drawingSize
        = new System.Drawing.Size((int)btnOverlay.ActualWidth, (int)
        btnOverlay.ActualHeight);
System.Drawing.Rectangle clippingRectangle
        = new System.Drawing.Rectangle(drawingPoint, drawingSize);

TaskbarManager.Instance.TabbedThumbnail.SetThumbnailClip(this.Handle,
clippingRectangle);
```

Der angezeigte Bildausschnitt ist immer identisch mit dem zugehörigen Inhalt des Fensters. Findet im Bildausschnitt eine Animation statt, sieht man diese auch im ThumbnailClip.

Als Erweiterung zum ThumbnailClip können im Thumbnailfenster zusätzliche Button angezeigt werden. Diese Steuerelemente vom Typ ThumbnailToolbarButton bieten nicht viele Konfigurationsmöglichkeiten, können aber ein Icon darstellen.

```
ThumbnailToolbarButton thumbnailButton1
        = new ThumbnailToolbarButton(Properties.Resources.favourite, "Favorite");
thumbnailButton1.Click
        += new EventHandler<ThumbnailButtonClickedEventArgs>
        (thumbnailButton1_Click);

ThumbnailToolbarButton thumbnailButton2
        = new ThumbnailToolbarButton(Properties.Resources.restart, "Restart");
thumbnailButton2.Click
        += new EventHandler<ThumbnailButtonClickedEventArgs>
        (thumbnailButton2_Click);

TaskbarManager.Instance.ThumbnailToolbars.AddButtons(this.Handle, thumbnailButton1,
thumbnailButton2);
```



Im Gegensatz zu den JumpListItem sind die ThumbnailToolbarButton auch für die Bedienung der laufenden Anwendung geeignet und können durch einen Click-Event direkt im Programmcode Methoden aufrufen.

Downloads

Das Windows API Code Pack inklusive der Beispiele von Microsoft finden Sie hier:

http://code.msdn.microsoft.com/WindowsAPICodePack

Ein Tutorial von Holger Hinzberg
http://www.hinzberg.net