

Ressourcen und Globalisierung

Mit .NET auf Weltreise

Das .NET Framework ermöglicht die komfortable Globalisierung von Anwendungen. Wer bei der Internationalisierung seiner Applikationen aber nicht in einer Sackgasse enden will, muss die Zusammenhänge kennen. dotnetpro zeigt, wie Sie Ihre Anwendungen erfolgreich auf große Fahrt schicken.

Im ersten Teil der Serie hat dotnetpro die Grundlagen zum Thema Ressourcen in .NET gezeigt. In diesem zweiten Artikel geht es darum, wie die beiden Themen Ressourcen und Globalisierung auf technischer Ebene zusammenspielen, welche Fallstricke es in diesem Bereich gibt und wo die Grenzen des .NET Frameworks liegen. Zunächst sollen einige Begriffe eingeführt und Hintergründe geklärt werden.

Globalisierung oder Internationalisierung bedeutet, eine Anwendung funktional auf unterschiedliche Sprachen und Regionen vorzubereiten. Lokalisierung ist die Entwicklung einer auf eine Sprache

oder Region abgestimmten Version einer Anwendung.

Globalisierung ist dabei mehr als „nur“ das Übersetzen der Masken und anderer Texte. Eine globalisierte Anwendung muss mit unterschiedlich formatierten Daten umgehen können. Dazu zählen unter anderem unterschiedliche Zahlenformate (Dezimal- und Tausendertrennzeichen) oder unterschiedliche Datumsformate, im Extremfall sogar unterschiedliche Kalender, Schrift, die von rechts nach links geschrieben wird und andere Dinge. All dies wird in diesem Artikel jedoch nicht vertieft. Der Fokus liegt hier allein auf dem Umgang mit Ressourcen. Die Dokumentation bietet zum Thema Globalisierung weitergehende Informationen. Das .NET Framework nutzt Ressourcen, um die in einer Applikation enthaltenen Forms in unterschiedlichen Sprachen anbieten zu können.

Der grobe Mechanismus, den die .NET Framework Class Library (FCL) implementiert, ist folgender: Texte und andere sprachabhängige Informationen werden in eigene lokalisierte Ressourcen ausgelagert, stehen also nicht mehr im eigentlichen Quelltext. Diese lokalisierten Ressourcen werden in eigenen Assemblies bereitgestellt, die parallel zur eigentlichen Assembly existieren, darüber hinaus jedoch keinen Code beinhalten. Die Klasse *ResourceManager* wählt zur Laufzeit die Ressourcen aus, die zur aktuellen Sprache beziehungsweise Kultur passen. Allein das Bereitstellen einer Assembly mit den lokalisierten Informationen ist damit bereits ausreichend, um von der Anwendung genutzt zu werden.

Die Klasse *CultureInfo*

In der FCL sammeln sich die für Globalisierung wichtigen Klassen im Namespace *System.Globalization*. Dreh- und Angelpunkt ist die Klasse *System.Globaliza-*

tion.CultureInfo. In dieser Klasse sind alle Informationen zu einer bestimmten Kultur versammelt. Enthalten sind Informationen zur Region und der Sprache, zum Formatieren von Zahlen (etwa Geldbeträgen) und Daten, zum Sortieren von Texten bis hin zum verwendeten Kalender.

Kulturen werden in einer einfachen, dreistufigen Hierarchie organisiert, wobei die verschiedenen Stufen unterschiedliche Konkretisierungen darstellen:

- Auf der untersten Hierarchiestufe befinden sich die spezifischen Kulturen (specific culture). Diese sind vollständig mit Sprache und regionalen Eigenschaften versehen, wie zum Beispiel dem Datumsformat, dem Kalender oder dem Währungssymbol. Beispiele sind *französisch/Frankreich (fr-FR)* oder *französisch/Kanada (fr-CA)*.
- Den spezifischen Kulturen übergeordnet sind die neutralen Kulturen (neutral cultures). Diese beschreiben die Sprache, sind aber keiner Region zugeordnet. Beispiele sind *französisch (fr)* oder *deutsch (de)*.
- Oberste Stufe schließlich ist die invariante Kultur (culture independent), die keiner Sprache oder Region zugeordnet ist. Als invariante Kultur, auch Standardkultur oder fallback culture genannt, wird üblicherweise die Kultur verwendet, in der eine Anwendung ohnehin erstellt wurde und die man nicht explizit auswählen muss.

Diese Abstufung findet sich ebenfalls in den Bezeichnungen der Kulturen wieder, die diversen RFCs und ISO-Standards folgen. Das übliche Schema für spezifische Kulturen ist ein zweibuchstabiger Sprachcode in kleinen Buchstaben gefolgt von einem Bindestrich und einem zweibuchstabigen Code für Land oder Region in Großbuchstaben. Bei neutralen Kulturen fällt der zweite Teil weg und die

Auf einen Blick

Autor



Als Manager Technology der SDX Software Development Experts AG verantwortet **Alexander Jung** die Evaluation und Implementierung neuer Technologien. Er ist seit vielen Jahren für Enterprise-Kunden als Architekt und Consultant in komplexen Software-Entwicklungsprojekten tätig. Sie erreichen ihn unter alexander.jung@sdx-ag.de.

dotnetpro.code
A0407Ressourcen



Sprachen C#

Technik Ressourcen zur Lokalisierung verwenden

Voraussetzungen
Visual Studio .NET

Serie

1. Ressourcen in .NET
2. Ressourcen und Globalisierung
3. Globalisierung in ASP.NET

invariante Kultur wird mit einem leeren Text repräsentiert. Weitere Hinweise zu den angesprochenen Standards und einigen Ausnahmen beziehungsweise Erweiterungen des beschriebenen Schemas zum Beispiel zur Wahl der Schrift in einigen osteuropäischen Ländern finden sich in der Dokumentation der Klasse *CultureInfo* [1].

Die übergeordnete Kultur erhält man über *CultureInfo.Parent*. Benötigt man eine spezifische Kultur, so hilft *CultureInfo.CreateSpecificCulture* weiter. Diese Methode liefert für eine neutrale Kultur die spezifische Kultur der dominierenden Region zurück, zum Beispiel *Frankreich* für *Französisch*. Das ist zugegebenermaßen nicht mehr als gut geraten, aber unter bestimmten Umständen wird einfach eine spezifische Kultur benötigt.

Die aktuelle Kultur einer Anwendung wird anhand der Betriebssystemeinstellungen vorgelegt. Verwaltet wird sie zu jedem einzelnen Thread. Über *Thread.CurrentCulture* kann man die Kultur für Datenformatierungen einstellen, mit *Thread.CurrentUICulture* die für das Laden der Ressourcen verwendete Kultur. *CurrentUICulture* kann mit neutralen Kulturen umgehen, für *CurentCulture* muss jedoch eine spezifische oder die invariante Kultur verwendet werden. Der Grund dafür ist, dass *DateTimeFormatInfo* und *NumberFormatInfo* nicht für neutrale Kulturen erzeugt werden können. So stellen Sie beispielsweise Französisch als aktuelle Kultur ein:

```
// => fr-FR
System.Threading.Thread.CurrentThread.
CurrentCulture=
CultureInfo.CreateSpecificCulture("fr");
```

Das Beispielprogramm zu diesem Artikel nutzt dies, um einen in der *app.config* vorhandenen Eintrag als Voreinstellung der Kultur vorzunehmen, bevor die Masken erzeugt werden. Listing 1 zeigt die Vorgehensweise.

Für den Fall, dass die fest verdrahtete Liste der vom .NET Framework unterstützten Kulturen nicht ausreichend ist hat man zudem die Möglichkeit, eigene Kulturen (custom cultures) zu erzeugen. Denkbar wären zum Beispiel hessisch oder plattdeutsch lokalisierte Versionen. Diese unterscheiden sich von den anderen Kulturen nur durch ihren Bezeichner und die Werte der Properties, einen prinzipiellen Unterschied gibt es für die FCL nicht. Allerdings gibt es Fallstricke im Zu-

Listing 1

Aktuelle Kultur voreinstellen.

```
static void Main()
{
    string StartupCulture =
        System.Configuration.
        ConfigurationSettings.AppSettings
        ["StartupCulture"];
    CultureInfo ci = new
        CultureInfo(StartupCulture);
    System.Threading.Thread.
        CurrentThread.CurrentUICulture= ci;

    Application.Run(new Form1());
}
```

sammenhang mit Visual Studio .NET, die den Umgang mit diesen Kulturen deutlich erschweren; mehr dazu weiter unten. Ein Beispiel zum Thema custom cultures findet sich in [2].

Lokalisierte Ressourcen bereitstellen

Lokalisierte Ressourcen werden in eigenen Assemblies bereitgestellt, so genannten Satelliten-Assemblies (satellite assemblies). Satelliten-Assemblies enthalten per definitionem keinen Code, sondern ausschließlich Manifestressourcen. Sie müssen einer bestimmten Namenskonvention folgen und in bestimmten Verzeichnissen stehen, damit sie zur Laufzeit gefunden werden können. Satelliten-Assemblies sind der eigentlichen Assembly zugeordnet, der Main Assembly, die auch den Code enthält. Außerdem wird empfohlen, in dieser Main Assembly einen vollständigen Satz Ressourcen bereitzustellen, die der neutralen Kultur zugeordnet sind.

Es gilt folgende einfache Konvention: Der Name der Satelliten-Assembly für eine Kultur entspricht dem Schema *<AssemblyName>.resources.dll*, wobei *<AssemblyName>* der Name der Main Assembly ist. Abgelegt werden diese Satelliten-Assemblies in Unterverzeichnissen mit dem Namen der jeweiligen Kultur. Dabei wird kein Unterschied zwischen spezifischen und neutralen Kulturen gemacht. Insbesondere werden Unterverzeichnisse für spezifische Kulturen nicht den Verzeichnissen für neutrale Kulturen untergeordnet. Abbildung 1 zeigt eine Dateistruktur mit einer lokalisierten Satelliten-Assembly.

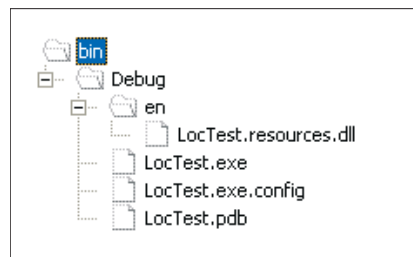


Abbildung 1 Satelliten-Assemblies im Dateisystem.

Daneben gibt es eine weitere Konvention, der die in den Satelliten-Assemblies enthaltenen Manifestressourcen vom Typ **.resources* folgen müssen. Das in Teil 1 behandelte Namensschema muss hier etwas erweitert werden: Im Namen der Manifestressourcen muss der Name der Kultur ergänzt werden. Aus *<name>.resources* in der Main Assembly wird *<name>.<kultur>.resources* für die lokalisierte Variante in der Satelliten-Assembly *<kultur>\assembly.resources.dll*. Für eigene Manifestressourcen gilt dies hingegen nicht. Diese haben in der Satelliten-Assembly den gleichen Namen wie in der Main Assembly. Abbildung 2 zeigt dies auf.

Unterstützung in Visual Studio .NET

Visual Studio .NET erleichtert die Einhaltung der Namenskonventionen deutlich. Hat man im Projekt eine Datei *<Dateiname>.<ext>* als *embedded resource* angeben, so wird diese im Allgemeinen unter ihrem Namen, gebildet nach dem bekannten Schema *<DefaultNamespace>.<Unterverzeichnisse>.<Dateiname>.<ext>*, als Manifestressource abgelegt.

Folgt der Dateiname jedoch dem Schema *<Dateiname1>.<kultur>.<ext>*, so landet die Datei nicht in der Main Assembly. Stattdessen legt Visual Studio .NET automatisch eine Satelliten-Assembly für die angegebene Kultur an und fügt die Datei dort als Manifestressource ein. Zudem berücksichtigt Visual Studio .NET den angesprochenen Unterschied zwischen **.resources*- und sonstigen Manifestressourcen. Bei Ersteren wird der Name der Datei beibehalten, einschließlich des Anteils *<kultur>*. Bei den anderen Manifestressourcen wird der Anteil *<kultur>* im Namen der Manifestressource in den Satelliten-Assemblies entfernt.

Die Datei *Form1.fr-FR.resources* wird also in der Satelliten-Assembly im Unterverzeichnis *fr-FR* landen und dort unter dem Namen *LocTest.Form1.fr-FR.resour-*

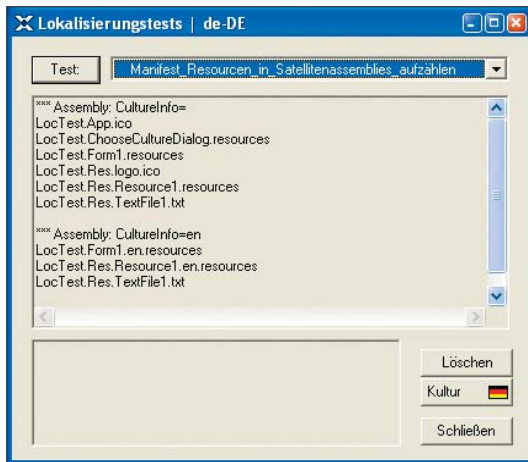


Abbildung 2 Manifestressourcen in Satelliten-Assemblies aufzählen.

ces abgelegt. Die Datei *Textdatei.fr-FR.txt* wird ebenfalls in dieser Satelliten-Assembly abgelegt, jedoch wird der Name dort nicht *LocTest.Textdatei.fr-FR.txt* lauten, sondern *LocTest.Textdatei.txt*. Auch dies ist in Abbildung 2 gut erkennbar.

Dieses Verhalten macht das Laden von Manifestressourcen aus beliebigen Satelliten-Assemblies einfacher. *resources*-Dateien werden im Regelfall über *ResourceManager* bearbeitet, sodass man sich nicht selbst um die Namensunterschiede kümmern muss. Von Ausnahmefällen abgesehen, wird einem dieser Unterschied also nicht wirklich Probleme bereiten. Ein Grund, warum *resources*-Dateien die Kultur als redundante Information im Namen haben, ist jedoch nicht erkennbar.

Die gerade beschriebenen Vorgänge funktionieren in Visual Studio .NET für die vom .NET Framework von Hause aus unterstützten Kulturen, nicht jedoch für die oben angesprochene Möglichkeit von custom cultures. Sollte der Anteil *<kultur>* kein von *CultureInfo* unmittelbar unterstützter Bezeichner einer Kultur sein, so wird die Kultur ganz normal als Namensteil behandelt und natürlich wird auch keine Satelliten-Assembly erzeugt. Damit hilft Visual Studio .NET im Zusammenhang mit custom cultures nicht weiter, sondern ist sogar hinderlich.

Lokalisierte Ressourcen zur Laufzeit verwenden

Der Zugriff auf lokalisierte Ressourcen erfolgt über die Klasse *ResourceManager*. Beim Laden einer Ressource folgt *ResourceManager* einem als resource fallback process bezeichneten Vorgang. Dieser Vorgang läuft so ab:

Wenn ein *ResourceManager* mit Verweis auf einen Datentypen oder einen Bezeichner und eine Assembly angelegt wird, dann verwendet er nicht nur die Main Assembly selbst, sondern auch deren Satelliten-Assemblies. Wird nun eine Ressource abgerufen, so beginnt der *ResourceManager* seine Suche mit der aktuell eingestellten spezifischen Kultur. Findet er zu dieser keine passende Ressource – das heißt keine passende Satelliten-Assembly oder in dieser keine passende Manifestressource oder in dieser keine entsprechende Ressource –, sucht er

erneut, diesmal mit der übergeordneten neutralen Kultur. Schlägt auch das fehl, wird mit der Standard- (oder Fallback-) Kultur in der Main Assembly gesucht. War auch das erfolglos, bricht der Vorgang ab. Entweder wurde überhaupt keine passende Manifestressource (die *resources*-Datei) gefunden, dann ist das Ergebnis eine *System.Resources.MissingManifestResourceException*. Oder es wurde lediglich die Ressource (der einzelne Wert innerhalb der *resources*-Datei) nicht gefunden, dann ist das Ergebnis *null*.

In diesem Verfahren liegt auch die Empfehlung begründet, in der Main Assembly einen vollständigen Satz an Ressourcen bereitzustellen.

Der resource fallback process beginnt mit einer spezifischen Kultur. Diese kann der jeweiligen Methode (zum Beispiel *ResourceManager.GetObject()*) als Argument mitgeteilt werden. Unterlässt man das, wird die aktuelle Kultur (*Thread.CurrentUICulture*) verwendet.

Damit wird die Bereitstellung einer neuen lokalisierten Ressource in Form einer neuen Satelliten-Assembly denkbar einfach: Der Code in Listing 2 geht davon aus, dass in *res/resource1.resx* ein String mit Namen *test* abgelegt ist.

Angenommen, der abgelegte String hätte in der Main Assembly den Wert *this is a test*, dann wäre genau das der Wert, den der String *s* in Listing 2 hätte.

Zusätzlich sei angenommen, dass die lokalen Ländereinstellungen deutsch/Deutschland (de-DE) vorsehen.

Nun erzeugt man eine *resources*-Datei mit einem String mit Namen *test* und dem Wert *dies ist ein test* und packt diese Datei unter dem Namen *LocTest.res.Resource1.de.resources* in die Satelliten-As-

Listing 2

Die Klasse *ResourceManager* verwenden.

```
ResourceManager rm = new ResourceManager  
    ("LocTest.Res.Resource1",  
    GetType().Assembly);  
string s = rm.GetString("test");
```

sembly – also eine DLL mit Namen *LocTest.resources.dll*. Allein das Kopieren dieser Datei in das passende Unterverzeichnis *de* reicht aus, um dafür zu sorgen, dass der String *s* bei erneutem Aufruf nicht mehr den Wert *this is a test* hat, sondern über den neuen deutschen Text verfügt. Der *ResourceManager* hat in beiden Fällen zunächst nach einer Satelliten-Assembly für die spezifische Kultur *de-DE* gesucht, danach für die neutrale Kultur *de*. Durch die Bereitstellung der Satelliten-Assembly wird er nun plötzlich fündig.

Damit stellt der resource fallback process sicher, dass bei Bedarf nachträglich weitere lokalisierte Satelliten-Assemblies bereitgestellt werden können, ohne dass der Code der eigentlichen Anwendung angefasst werden müsste.

Man kann den Suchprozess optimieren, indem man den *ResourceManager* mithilfe des Attributes *NeutralResourcesLanguageAttribute* in der Assembly darüber informiert, welche Kultur als neutrale Kultur verwendet wird. Der vollständige Vorgang schließt darüber hinaus noch die Suche im Global Assembly Cache (GAC) ein und wird in [3] beschrieben.

Der Vollständigkeit halber sollte noch das Attribut *SatelliteContractVersionAttribute* (ebenfalls für die Assembly) erwähnt werden. Mit diesem kann die Versionierung der Main Assembly von der der Satelliten-Assemblies entkoppelt werden. Auf diese Weise kann Funktionalität unabhängig von der Präsentation versioniert werden.

Eine vielleicht triviale Anmerkung: Wenn ein Objekt einmal seinen Weg aus den Ressourcen in die Anwendung gefunden hat, wird es sich nicht mehr von selbst ändern. Das nachträgliche Umändern der Kultur hat auf dieses Objekt keinerlei Einfluss. Wenn eine Anwendung also die Möglichkeit bietet, zur Laufzeit die Kultur zu wechseln, dann muss sie auch selbst dafür sorgen, dass beim Wechsel alle kulturspezifischen Texte

und Objekte neu zugewiesen werden. Den Anwender aufzufordern, die Anwendung neu zu starten, dürfte hier die einfachste Variante sein.

Lokalisierung von Windows Forms

Nachdem nun die Mechanismen des .NET Frameworks zur Lokalisierung bekannt sind, können diese im Zusammenspiel mit den entsprechenden Werkzeugen genutzt werden, um lokalisierte Anwendungen zu erstellen. Zunächst die Betrachtung für Windows Forms.

Wie bereits in Teil 1 erwähnt, verwenden die Designer von Visual Studio .NET *resx*-Dateien, um dort Informationen abzulegen, die sich schlecht im Code darstellen lassen. Ordnet man etwa einem Button ein Bild zu, so landet das Bild in der *resx*-Datei. Setzt man hingegen einen Text ein, so wird entsprechender Code generiert. Listing 3 zeigt den entsprechenden vom Designer generierten Code.

Listing 3

Nicht lokalisierter Code.

```
this.button1.Image = ((System.Drawing.Image)
(resources.GetObject("button1.Image")));
this.button1.Size = new
    System.Drawing.Size(48, 23);
this.button1.Text = "go";
```

Will man eine solche Form lokalisieren, so sollte mindestens der Text ebenfalls den Ressourcen entnommen werden, andernfalls müsste der Quelltext angepasst werden. Wenn man den im Beispiel *go* benannten Button in der deutschen Variante mit *Beginnen* übersetzt, wird ziemlich schnell deutlich, dass auch Größe und Position der Lokalisierung unterliegen sollten. Das ist aber noch nicht alles: Der generierte Code weist dem Button nicht explizit eine *Font* zu, da diese Angabe nicht vom Default abweicht, der in der Liste der Eigenschaften mit fett formatierten Werten dargestellt wird.

Für Sprachen, die eine andere Schrift benötigen, wird dies jedoch notwendig. Man muss also sogar Dinge berücksichtigen, die im generierten Code zunächst gar nicht auftauchen. Erst wenn das in ausreichendem Maße berücksichtigt ist, kann man an den eigentlichen Vorgang der Lokalisierung denken.

Listing 4

Vom Designer für lokalisierbare Forms generierter Code.

```
[...]
this.button1.BackgroundImage = ((System.Drawing.Image)(resources.GetObject
("button1.BackgroundImage")));
this.button1.Font = ((System.Drawing.Font)(resources.GetObject("button1.Font")));
this.button1.Image = ((System.Drawing.Image)(resources.GetObject("button1.Image")));
this.button1.ImageAlign = ((System.Drawing.ContentAlignment)
(resources.GetObject("button1.ImageAlign")));
this.button1.Location = ((System.Drawing.Point)(resources.GetObject("button1.Location"));
this.button1.Name = "button1";
this.button1.Size = ((System.Drawing.Size)(resources.GetObject("button1.Size")));
this.button1.TabIndex = ((int)(resources.GetObject("button1.TabIndex")));
this.button1.Text = resources.GetString("button1.Text");
```

Der Form-Designer von Visual Studio .NET trägt dem Rechnung, indem er bei Forms zwei künstliche Properties – *Localizable* und *Language* – ergänzt. „Künstlich“ deshalb, weil sie in den Eigenschaften genauso aufgelistet werden, wie dies bei den Properties der Klasse der Fall ist, ohne dass aber die Klasse *Form* die beiden Properties kennen würde. Es handelt sich um ebensolche Metainformationen über die Form wie das Sperren der Controls gegen Verschieben (*Kontextmenü/Lock Controls*). Diese landen in der *resx*-Datei und werden lediglich vom Designer ausgewertet.

Zur Lokalisierung muss man zunächst *Localizable* auf *True* setzen. Dies weist den Designer an, die Code-Generierung umzustellen. Statt nur „unhandliche“ Teile auszulagern, landen nun nahezu alle Properties der Controls in den Ressourcen, unabhängig davon, ob sie vom Standardwert abweichen oder nicht. Ausschnittsweise sieht das für den Button so aus, wie es Listing 4 darstellt.

Tatsächlich werden nicht alle Properties dieser Vorgehensweise unterworfen. Wie üblich kann man über ein Attribut zur Property steuern, ob die Property lokali-

Windows Forms Resource Editor

Wenn Visual Studio .NET nicht zur Verfügung steht, muss man deshalb nicht gleich auf Lokalisierung verzichten. Im .NET Framework SDK findet sich die Anwendung Windows Forms Resource Editor (*WinRes.exe*). Diese bietet eine dem Designer von Visual Studio .NET nachempfundene Oberfläche zur Bearbeitung von Forms, greift dabei aber ausschließlich auf Informationen aus der Ressource-Datei zurück. Dies setzt aber voraus, dass alle dazu notwendigen Informationen in dieser Ressource-Datei abgelegt sind – bis hin zu den Klassennamen der Controls. Auch kann *WinRes.exe* lediglich Controls manipulieren – Löschen und Erzeugen ist nicht möglich. Damit eignet sich *WinRes.exe* ausschließlich zu Lokalisierung von Forms – zu deren Erstellung wird nach wie vor ein anderes Werkzeug benötigt.

Der Einsatz von *WinRes.exe* kann auch für den Anwender von Visual Studio .NET sinnvoll sein, insbesondere dann, wenn der Quelltext nicht zur Verfügung gestellt werden soll. Allerdings gibt es Unterschiede in der Arbeitsweise. Im Gegensatz zu Visual Studio .NET speichert *WinRes.exe* alle Ressourcen in einer lokalisierten Version ab, da es diese zum Aufbau der Forms benötigt. In Visual Studio .NET lokalisierte Ressource-Dateien beinhalten hingegen nur die Abweichungen dieser Version von den Ressourcen der neutralen Kultur. So steht dann auch in [5] der Hinweis:

*“Before you begin to localize an application’s Windows Forms forms, you should decide whether you want to use Visual Studio .NET or Winres.exe as the localization tool. The localized *.resx files produced by each tool are not interchangeable. Therefore, after you have started to localize with one tool, you cannot switch to the other tool.”*

Listing 5

Lokalisierbarkeit über ein Attribut steuern.

```
public class LocTextBox : TextBox {
    [...]
    private string _lokalisierbarerText;
    private string _nichtLokalisierbarerText;
    [Localizable(true)]
    public string LokalisierbarerText {
        get { return _lokalisierbarerText; }
        set { _lokalisierbarerText= value; }
    }
    public string NichtLokalisierbarerText {
        get { return _nichtLokalisierbarerText; }
        set { _nichtLokalisierbarerText= value; }
    }
}

[...]
this.textBox1.LokalisierbarerText =
resources.GetString("textBox1.LokalisierbarerText");
this.textBox1.NichtLokalisierbarerText = "nicht lokalisiert";

[...]
<data name="textBox1.LokalisierbarerText">
    <value>lokalisiert</value>
</data>
```

sierbar sein soll oder nicht. In diesem Fall ist es das *System.ComponentModel.LocalizableAttribute*. Listing 5 zeigt die Verwendung in einer Klasse *LocTextBox* sowie die Auswirkungen in der Methode *InitializeComponent()* der Form und in ihrer *resx*-Datei. Das Beispiel zeigt auch, dass eine Property bei Abwesenheit des Attributes nicht lokalisiert wird.

Mit der Property *Language* schließlich kann man einstellen, welche lokalisierte Version der Form man gerade bearbeitet, also letztlich in welcher *resx*-Datei die Informationen abgestellt werden. So lassen sich die unterschiedlich lokalisierten Versionen direkt in Visual Studio .NET erstellen, wie es Abbildung 3 darstellt. Wenn man eine lokalisierte Version in Visual

Studio .NET bearbeitet, werden Ressourcen dort erst dann angelegt, wenn sie von der neutralen Ressource abweichen. Legt man ein neues Control an, während man noch die lokalisierte Version bearbeitet, meldet sich Visual Studio .NET mit einem Warnhinweis, den Abbildung 4 zeigt.

Visual Studio .NET speichert die Daten also sowohl in der gerade eingestellten Sprache als auch in den Ressourcen zur neutralen Kultur. Es stellt auf diese Weise sicher, dass in der Main Assembly immer ein vollständiger Satz Ressourcen vorhanden ist.

Noch einmal deutlich herausgestellt: *Localizable* und *Language* müssen nicht gesetzt werden, um Lokalisierung technisch möglich zu machen. Diese Funktio-

nalität ist im .NET Framework fest verankert und muss nicht extra aktiviert werden. Diese beiden Einstellungen sorgen lediglich dafür, dass der Designer bei der Code-Generierung die zu lokalisierenden Informationen auch an der richtigen Stelle ablegt. Nun sollen die bisher noch nicht abgedeckten Fälle untersucht werden: Wie sieht es mit normalen Manifestressourcen aus, und wie kann man den Resource-fallback-process-Mechanismus umgehen?

Lokalisierung von Manifestressourcen

Der beschriebene resource fallback process und das ganze Konzept der Lokalisierung hängen im .NET Framework ausschließlich an der Klasse *ResourceManager*. Diese wiederum arbeitet ausschließlich mit Ressourcen, also den Inhalten der *resources*-Dateien. Wenn man es mit anderen Manifestressourcen zu tun hat, bekommt man das Problem, dass die Methoden der Assembly keine Suche nach Satelliten-Assemblies durchführen:

- Beim Zugriff auf Manifestressourcen über *Assembly.GetManifestResourceStream()* hat man keine Unterstützung für lokalisierte Versionen. Diese Methode ist notwendig für alle Manifestressourcen außer *resources*-Dateien.
- Über *Assembly.GetSatelliteAssembly()* kann man sich gezielt eine Satelliten-Assembly geben lassen. Allerdings löst die Methode eine Exception aus, wenn die Satelliten-Assembly zur gewünschten Kultur nicht existiert. Eine Suche nach der Satelliten-Assembly der übergeordneten Kultur findet nicht statt.

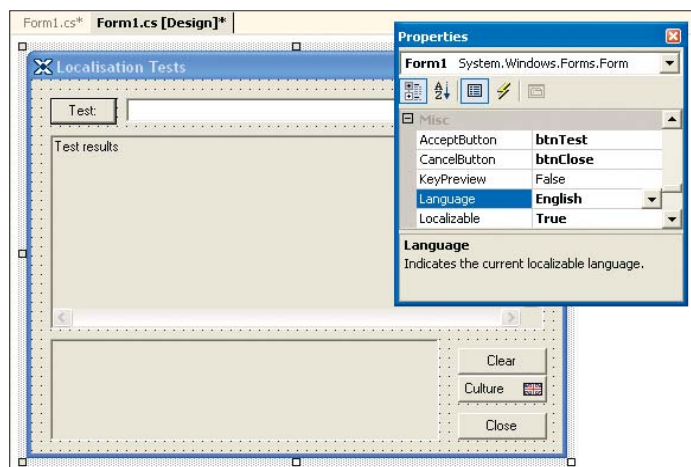


Abbildung 3 Die Form in englischer (neutraler) Darstellung.

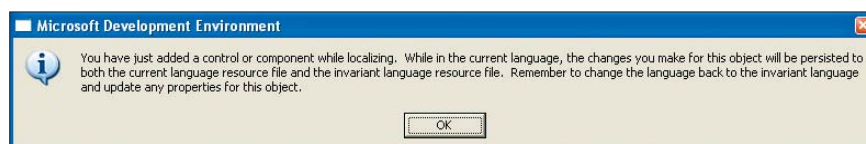


Abbildung 4 Vorsicht beim Hinzufügen von Controls während der Bearbeitung einer bestimmten Kultur.

Listing 6

Einen eigenen resource fallback process implementieren.

```
Stream GetManifestResourceStream(Assembly assembly, string name,
    CultureInfo ci) {
    Assembly a= FindLocalizedManifestResource(assembly, name, ci);
    return a.GetManifestResourceStream(name);
}

Assembly FindLocalizedManifestResource(Assembly assembly, string name,
    CultureInfo ci) {
    Assembly a= null;
    while (true) {
        try {
            if (ci==CultureInfo.InvariantCulture)
                a= assembly;
            else
                a= assembly.GetSatelliteAssembly(ci);
            string[] names= a.GetManifestResourceNames();
            if (Array.IndexOf(names, name)>=0)
                return a;
        } catch (Exception ex) {
            // GetSatelliteAssembly versagt
            System.Diagnostics.Trace.WriteLine(ex);
        }
        // oberste ebene erreicht?
        if (ci==CultureInfo.InvariantCulture)
            return null;
        // übergeordnete CulturInfo ...
        ci= ci.Parent;
    }
}

private string GetTestContent() {
    // erzeugt CultureInfo für => fr-FR
    CultureInfo ci= CultureInfo.CreateSpecificCulture("fr");
    Stream strm= GetManifestResourceStream(GetType().Assembly,
        "LocTest.Res.TextFile1.txt", ci);
    StreamReader sr= new StreamReader(strm);
    string s2= sr.ReadToEnd().Trim();
    sr.Close();
    return s2;
}
```

- Der Konstruktor von Icons, der einen Typ übernimmt (für Bitmaps gilt das analog) stellt lediglich eine Abkürzung für `type.Assembly.GetManifestResourceStream(...)` dar. Auch hier wird gar nicht erst nach einer Satelliten-Assembly gesucht.

Will man in diesen Fällen auf den resource fallback process nicht verzichten, muss man wohl oder übel selbst Hand anlegen. Zum einen gibt es die im ersten Teil beschriebene Möglichkeit, externe Dateien in *resources*-Dateien einzubetten und somit auf Manifestressourcen ganz zu verzichten. Zum anderen kann man den resource fallback process natürlich auch nachprogrammieren. Von Fehlerbehandlung und Optimierungen abgesehen, könnte das etwa so aussehen wie in Listing 6 dargestellt. Die erwähnten Konstruktoren bei *Icon* und *Bitmap* können damit zwar nicht verwendet werden. Da Satelliten-Assemblies keinen Code enthalten dürfen, können sie keine Typen exportieren, was hier aber notwendig wäre. Aber diese Klassen stellen ebenfalls Konstruktoren bereit, die einen *Stream* akzeptieren. Auf die Exceptions der Methode `Assembly.GetSatelliteAssembly()` zu reagieren, ist nicht unbedingt optimal. Es gibt allerdings keine legale Möglichkeit, sich die vorhandenen Satelliten-Assemblies aufzählen zu lassen. Man könnte sich allenfalls auf die Konventionen verlassen und das Dateisystem selbst absuchen.

Lokalisierten Zugriff verhindern

Bleibe noch der umgekehrte Fall: Man hat eine *resources*-Datei, möchte aber verhindern, dass der resource fallback process greift, und stattdessen gezielt Informationen einer ganz bestimmten Kultur laden, ohne *CurrentUICulture* zu ändern.

Zwar kann der Methode *ResourceManager.GetObject()* ein *CultureInfo* übergeben werden, dieses stellt aber nur den Startpunkt der Suche dar. Wird die Ressource nicht gefunden, so greift der resource fallback process mit der übergeordneten Kultur.

Die Lösung liegt in Objekten vom Typ *ResourceSet*, die von der Methode *ResourceManager.GetResourceSet()* geliefert werden. Ein solches *ResourceSet* bietet wie *ResourceManager* die Methoden *GetObject()* und *GetString()* an, bietet jedoch nur den Zugriff auf genau die der eigenen Kultur zugeordneten Ressourcen, ein *fallback* wird nicht durchgeführt.

Man könnte auf diese Art einen Dialog zur Auswahl der Kultur anbieten und in diesem die dynamisch geladenen, in den jeweiligen Ressourcen hinterlegten Flaggsymbole anzeigen.

Wertung

Es gibt Stimmen, die die Herangehensweise kritisieren, zum Beispiel [4]. Man muss aber anerkennen, dass im .NET Framework eine durchgängige Unterstüt-

zung für die Lokalisierung von Forms gegeben ist. Und falls die vorhandenen Werkzeuge nicht ausreichend sind, können sicher andere, spezialisiertere Anwendungen integriert werden – schließlich handelt es sich bei den *resx*-Dateien um ein denkbar einfaches XML-Format. |||||

[1] CultureInfo Class .NET Framework

Class Library

msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfSystemGlobalizationCultureInfoClassTopic.asp

[2] SDK, Sample CustomCulture

C:\Programme\Microsoft Visual Studio .NET 2003\SDK\v1.1\Samples\Technologies\Localization\CustomCulture

[3] Packaging and Deploying Resources,

.NET Framework Developer's Guide, msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconpackagingdeployingresources.asp

[4] .NET GUI BLISS – Streamline Your Code and Simplify Localization Using an XML-Based GUI Language Parser

Paul DiLascia, msdn.microsoft.com/msdnmag/issues/02/11/NETGUIBliss/default.aspx

[5] Windows Forms Resource Editor (Winres.exe), .NET Framework Tools

msdn.microsoft.com/library/default.asp?url=/library/en-us/cptools/html/cpgrfwindowsformsresourceeditor-winresexe.asp