HANSER

Leseprobe

Johann Weiher

EPLAN Electric P8 automatisieren

Grundlagen und Beispiele zum Erstellen von Scripten in Visual C#

ISBN: 978-3-446-42715-0

Weitere Informationen oder Bestellungen unter

http://www.hanser.de/978-3-446-42715-0

sowie im Buchhandel.

1 Einführung

Was ist Scripting?

Scripting bezeichnet die Möglichkeit, einzelne Befehle bzw. Programmcode in EPLAN auszuführen. Dies geschieht über die sogenannte API (application programming interface, dt. Programmierschnittstelle). Hinter der EPLAN-API verbergen sich alle Funktionen, welche in der Plattform (Electric P8, Fluid, PPE, Pro-Panel) vorhanden sind. Diese Programme bauen alle auf dem gleichen Programmcode auf und sind dadurch untereinander kompatibel. EPLAN Cabinet ist hierbei eine Ausnahme, denn es bietet nur eine eingeschränkte Schnittstelle. In den verschiedenen Applikationen sind ähnliche bzw. gleiche Funktionen enthalten, z. B. kann man in Fluid und Electric P8 Beschriftungen erzeugen. Einziger Unterschied ist der Inhalt.

Diese Abläufe werden in EPLAN *Actions* genannt. Ihnen ist ein eigenes Kapitel gewidmet, da es mehrere Wege gibt, solche Actions auszuführen. Das Wort Scripting bezieht sich meistens nur auf Scripte, welche Sie ab der Version *Compact* nutzen können. Um weitere Befehle oder Funktionen ausführen zu können, benötigen Sie das API-Modul von EPLAN. In diesem Buch gehe ich aber ausschließlich auf den Standardumfang der Compact-Version ein.

Was sind Scripte?

Scripte sind kleine Programmcodes. Diese können in zwei Programmiersprachen erstellt werden:

- Microsoft C# (C-Sharp)
- Microsoft VB.NET (Visual-Basic.NET)

Ich werde in den folgenden Kapiteln nur Beispiele in C# bereitstellen, da EPLAN mit dieser Sprache fertigen Code generiert und dadurch eine optimale Vorlage liefert. Ein Script ist nicht alleine ausführbar. Es muss in Verbindung mit EPLAN gestartet werden. Die verschiedenen Möglichkeiten werden in einem eigenen Kapitel erklärt.

Was können Scripte?

Vieles, aber nicht alles. EPLAN stellt eine Reihe von Befehlen bereit, schränkt diese aber auf einen überschaubaren Bereich ein. Dadurch wird dem Anwender der Einstieg enorm erleichtert. Auf diese Weise wird auch sichergestellt, dass keine ungewollten Aktionen, z. B. auf das Projekt, ausgeführt werden.

Vielleicht kennen Sie die immer wiederkehrende Aufgabe, Beschriftungen auszugeben. Je Projekt sind mehrere Exporte nötig, und jedes Mal muss das Beschriftungsschema neu ausgewählt, zusätzlich der Ordner benannt und ein Dateiname vergeben werden. Mit einem Script können Sie all diese Arbeitsschritte zusammenfassen und z. B. auf einen Menüpunkt legen. Sie können auch mehrere Beschriftungen nacheinander über diese Funktion erzeugen. Auch der PDF-Export kann automatisiert werden. Möchten Sie zur Änderungsverfolgung z. B. automatisch beim Schließen des Projektes das PDF erzeugen? Kein Problem mit einem Script. Sie wollen Schnittstellen schaffen, Informationen in EPLAN aus z. B. Ihrem ERP zu nutzen? Kein Problem, über die Import-Funktion geht das auf Knopfdruck. Auch die Weitergabe von Informationen aus EPLAN in andere Systeme klappt problemlos. Oft muss zwischen verschiedenen Einstellungen hin- und hergewechselt werden. Das Suchen in den unzähligen Settings in EPLAN ist mühselig. Schreiben Sie ein Script für Ihre Konfigurationen, und erledigen Sie das unter der Projektierung.

Eine kleine Auflistung der Möglichkeiten, welche mit Scripten realisiert werden können:

- Beschriftungen automatisieren
- PDF-Export
- Backup
- eigene Menüs/Toolbars erstellen
- grafische Formulare mit z. B. Buttons, Checkboxen, Auswahldialogen
- Eigenschaften verändern
  - Projekteigenschaften
  - Seiteneigenschaften
- Einstellungen
  - Lesen
  - Schreiben

... und das sind noch lange nicht alle Funktionen. Durch das Erweitern des Programmcodes können mehr Funktionen hinzugefügt werden.

Was kann das API-Modul im Vergleich zum Scripting?

Um den Unterschied etwas deutlicher zu machen, anbei eine kleine Auflistung der wichtigsten Merkmale des API-Moduls in EPLAN:

- Zugriff auf das komplette EPLAN-Datenmodell
- einfacheres Lesen von Objekten
- Zugriff auf mehr Objekte
- direkter Zugriff auf Projekteigenschaften/Projekteinstellungen
- Lesen/Schreiben von Daten der Artikeldatenbank
- mehr verfügbare Actions
- Online-Debugging

2 Scriptfunktionen

2.1 Attribute

Ein Script kann, wie wir schon im ersten Kapitel erfahren haben, verschiedenste Funktionen ausführen. Darum müssen Sie im Programmcode hinterlegen, welche Funktion das Script bereitstellt. Es können auch mehrere Actions in einem Script vorhanden sein. Sie können z. B. eine Beschriftung automatisch erzeugen lassen. Außerdem kann für diese Action ein eigener Menüpunkt in der Menüleiste erzeugt werden. Dies ist alles innerhalb einer Datei möglich. In diesem Fall macht das auch Sinn, da der Menüpunkt nicht ohne die dazugehörige Action ausgeführt werden kann. Es macht aber keinen Sinn, alle Actions in eine Datei zu packen, da es so schnell unübersichtlich wird.

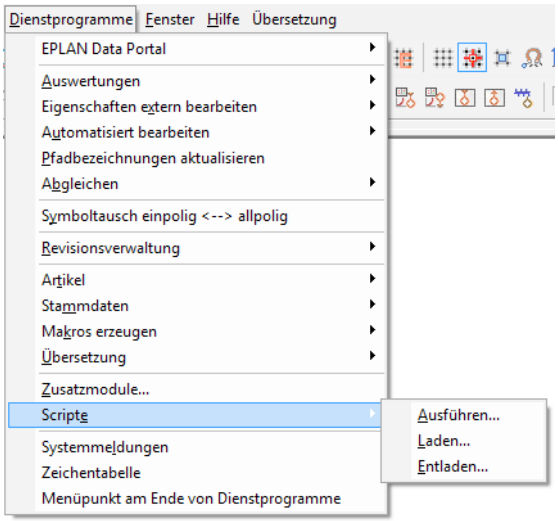


Bild 2.1
Scripte ausführen/laden/entladen

Für unser erstes Beispiel verwenden wir das Attribut *[Start]*. Das ist der schnellste Weg, mehrere Scripte zu testen. Oftmals stellt sich die Frage, wann ein Script geladen und was ausgeführt werden muss. Über die Attribute können Sie das schnell und einfach herausfinden. Anbei eine Auflistung:

- Ausführen:
  - *[Start]*
- Laden:
  - *[DeclareAction]*
  - *[DeclareEventHandler]*
  - *[DeclareMenu]*
  - *[DeclareRegister]*
  - *[DeclareUnregister]*

2.1.1 Start

Das Attribut *[Start]* wird meistens verwendet, wenn man ein Script nur einmal oder eher selten benötigt. Sie müssen bei jedem Starten der Action über die Menüleiste gehen, um das Script auszuführen.

In der Entwicklungsumgebung wechseln Sie in das Codefenster und betrachten die ersten Zeilen. Führen Sie hierfür einen Doppelklick auf die Datei  *01_Start.cs* aus.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

Mit dem Befehl *using* beschreiben Sie, welche Verweise verwendet werden sollen. In einer sogenannten Using-Direktive werden bestimmte Programmbefehle gespeichert und können, sofern sie geladen sind, abgerufen werden. Für das Scripting in EPLAN sind folgende Verweise erforderlich:

- Eplan.EplApi.ApplicationFramework
- Eplan.EplApi.Base
- Eplan.EplApi.Scripting

Zusätzlich benötigen wir *System.Windows.Forms*, um unsere Messagebox anzeigen zu lassen.

Es werden nicht immer alle Verweise benötigt. Dennoch ist es ratsam, eine Vorlage zu erstellen, welche alle für Sie wichtigen Using-Direktiven enthält. Später können Sie nicht benötigte Elemente entfernen.

Ergänzen Sie nun den Code um folgende Zeilen:

```
using Eplan.EplApi.ApplicationFramework;
using Eplan.EplApi.Base;
using Eplan.EplApi.Scripting;
```

Beim Tippen des Codes erscheint ein kleiner Dialog mit verschiedenen Begriffen:

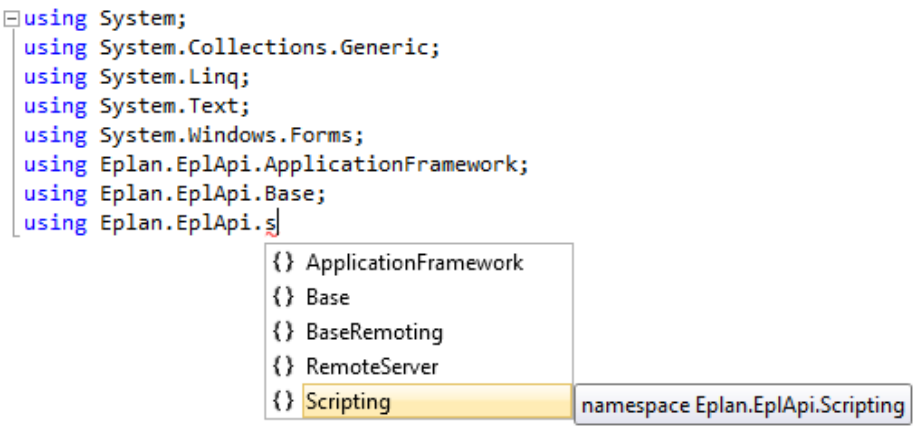


Bild 2.2
IntelliSense

Diesen Dialog nennt man *IntelliSense*. Es ist eine Möglichkeit zur Wortvervollständigung einzelner Begriffe in der Programmierung. Das von Microsoft entwickelte Hilfsmittel ist in fast allen Entwicklungsumgebungen für sämtliche Programmiersprachen vorhanden. Wenn Sie einen Begriff eingeben, werden übereinstimmende Ergebnisse angezeigt. Sie können nun mit den Pfeiltasten auf der Tastatur den gewünschten Begriff auswählen. Um das Getippte zu vervollständigen, drücken Sie auf die **Tab**-Taste. Ist der *IntelliSense* nicht sichtbar, kann der Dialog mit dem Tastatur-Shortcut **Strg** + **Leertaste** wieder eingeblendet werden.

Eine Code-Folge wird in C# immer mit einem Semikolon (;) beendet. Wer schon einmal in Visual Basic programmiert hat, weiß, dass es dort der Zeilenumbruch ist. In C# können Sie beliebig viele Zeilenumbrüche im Code einfügen.

Den Aufbau eines Programmcodes in C# können Sie sich vorstellen wie eine Zwiebel. Es gibt verschiedene Schichten, welche mit geschweiften Klammern eingegrenzt werden. Der Aufbau ist nicht so kompliziert, dass die Zwiebel Sie zum Weinen bringen wird. Die erste Schale wird in einem Script nicht benötigt. Darum können Sie den Namespace getrost löschen. Achten Sie darauf, dass Sie beide Klammern aus dem Code entfernen. EPLAN lädt den Inhalt des Scriptes in einen fest definierten Namespace.

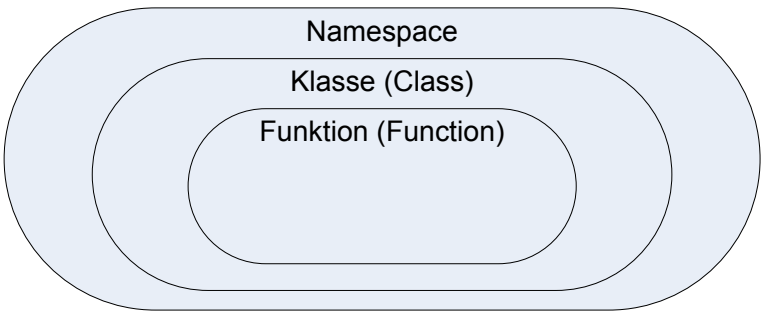


Bild 2.3
Aufbau des Programmcodes in Visual C#

Die „äußere Schale“ ist die Klasse (Class):

```
public class Class
{
}
```

In einer Klasse muss mindestens eine Funktion vorhanden sein. Ergänzen Sie den Programmcode um folgende Zeilen:

```
public void Function()
{
}
```

In der Funktion steht nun der eigentliche Programmcode (Was soll EPLAN tun?).

Wir lassen eine Messagebox anzeigen, um festzustellen, ob unser Script ausgeführt wurde. Der Aufbau des Befehls in kurzen Worten:

- Was soll ich machen → *MessageBox*
- Was soll ich damit anstellen → *Show* (anzeigen)
- Was soll ich anzeigen → *Ich kann Scripten!*

Die einzelnen Klassen und Methoden werden durch einen Punkt getrennt. Stellen Sie sich eine Treppe vor. Auf jeder Ebene befinden sich andere Räume mit unterschiedlichen Funktionen. Zu Beginn findet man oft nicht sofort die gewünschte Funktion, aber mit der Zeit versteht man die Struktur des .NET-Frameworks.

Beispielaufbau eines Methodenaufrufes:

```
Ebene0.Ebene1.Ebene2.Methode();
```

Nehmen wir als Beispiel eine Aktivität an einem Ort, müsste dies wie folgt angegeben werden:

```
München.Theresienwiese.Oktoberfest.Achterbahnfahren();
```

Text steht im Programmcode immer in Hochkommas ("Anführungszeichen"):

```
München.Bahnhof.Schaffner.Ruft("Alles einsteigen!");
```

Kommentare können mit zwei Schrägstrichen eingefügt werden:

```
// Ich bin ein Kommentar
```

Dadurch weiß das Programm, dass es sich um einen Kommentar handelt, der nicht abgearbeitet werden muss.

In Visual Studio gibt es dafür schon fertige Buttons. Ist diese Toolbar nicht sichtbar, können Sie diese, wie in Bild 2.4 dargestellt, einblenden (es wird die komplette Zeile, welche gerade aktiv ist, ein- bzw. auskommentiert).

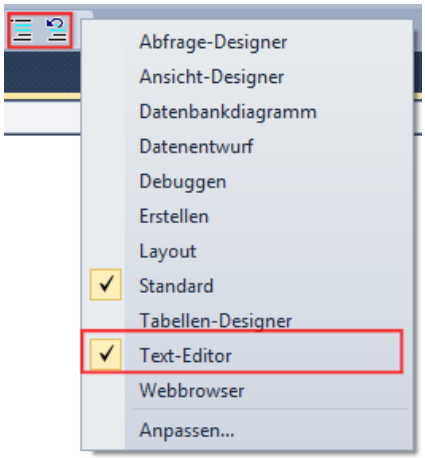


Bild 2.4
Toolbar *Text-Editor* einblenden

Um eine Messagebox anzuzeigen, ist folgender Code notwendig:

```
MessageBox.Show("Ich kann Scripten!"); // Kommentar
```

Eine Funktion wird mit *return* (zurückkehren) abgeschlossen. Dies ist nicht zwingend notwendig, wenn jeder Teil des Programmcodes Werte zurückgibt.

```
return;
```

Nicht verwendete Verweise sollten Sie entfernen, um das Starten des Scriptes zu beschleunigen. Durch eine Funktion in Visual Studio geschieht dies automatisch: *rechte Maustaste > Using-Direktiven organisieren > Entfernen und Sortieren*.

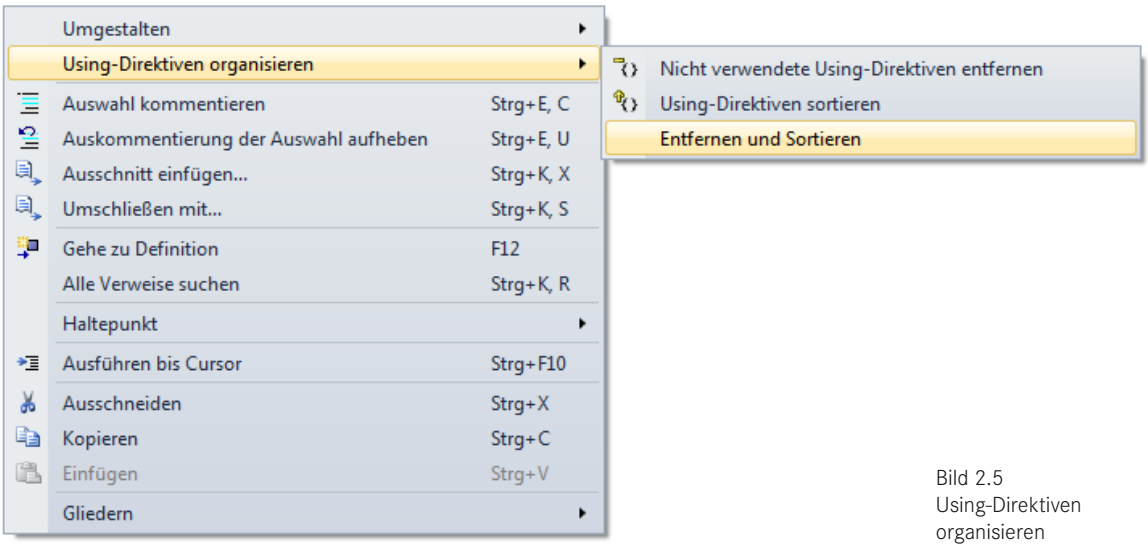


Bild 2.5
Using-Direktiven organisieren

Nun setzen wir unsere „Schalen“ zusammen. Das komplette Beispiel ist immer am Ende einer Lektion zu finden:

```
using System.Windows.Forms;
using Eplan.EplApi.Scripting;

public class Class
{
    [Start]
    public void Function()
    {
        MessageBox.Show("Ich kann Scripten!"); // Kommentar

        return;
    }
}
```

Wechseln Sie nun nach EPLAN und testen Sie das Script über *Dienstprogramme* > *Scripte* > *Ausführen...*. Im darauffolgenden Dialog wählen Sie das Script aus. Die Scriptdatei befindet sich im Ordner Ihres Visual-Studio Projektes.

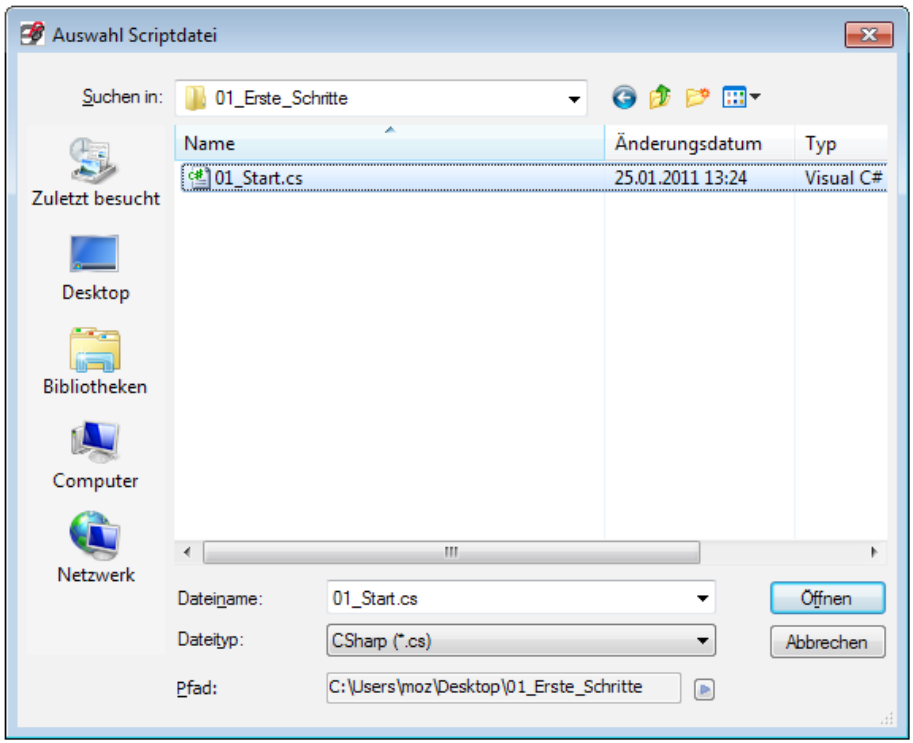


Bild 2.6
Script ausführen

Wurde der Programmcode fehlerfrei ausgeführt, sehen Sie folgenden Dialog:

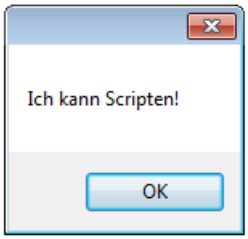


Bild 2.7
Messagebox

Beim Schreiben des Programmcodes können mehrere Fehler passieren. Kommentieren Sie testweise die Initialisierung der Using-Direktiven aus:

```
//using System.Windows.Forms;
//using Eplan.EplApi.Scripting;
```

Vielleicht ist Ihnen beim Auskommentieren der Using-Anweisungen aufgefallen, dass im Fehlerfenster von Visual Studio einige Fehler hinzugekommen sind. Das resultiert aus der Tatsache, dass die Entwicklungsumgebung die Ausdrücke aufgrund fehlender Verweise nicht finden kann. Diese Fehlerliste ist ähnlich wie die Meldungsverwaltung von EPLAN aufgebaut. Hier wird ebenfalls unterschieden zwischen:

- Fehler
- Warnungen
- Meldungen

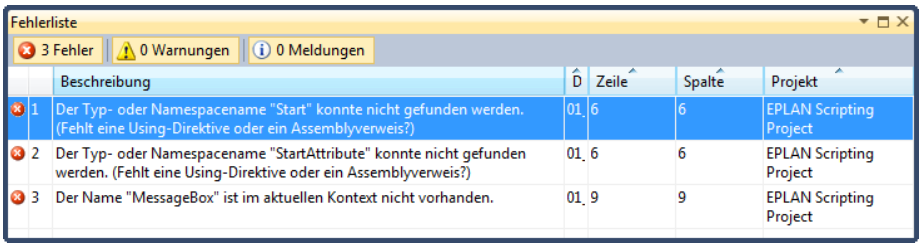


Bild 2.8
Fehlerliste

Ist dieses Fenster nicht sichtbar, können Sie es über *Ansicht* > *weitere Fenster* > *Fehlerliste* einblenden.

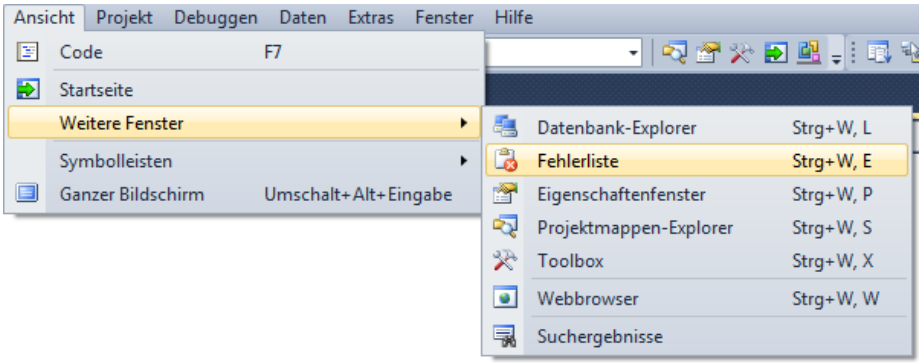


Bild 2.9
Fehlerliste einblenden

Fehler: Using-Direktive

Es kann sein, dass Fehler in Visual-Studio nicht auftreten, während sie beim Ausführen/Starten des Scriptes in EPLAN gefunden werden. Ist dies der Fall, werden diese in den Systemmeldungen angezeigt.

Ab Version 2.0 beachtet EPLAN nicht, welche Using-Direktiven intern schon bekannt waren. Ist ein anderer Fehler im Script vorhanden, werden diese (fehlerhafterweise) auch angezeigt, können aber guten Gewissens ignoriert werden. In Version 1.9 müssen Using-Direktiven immer auskommentiert werden.

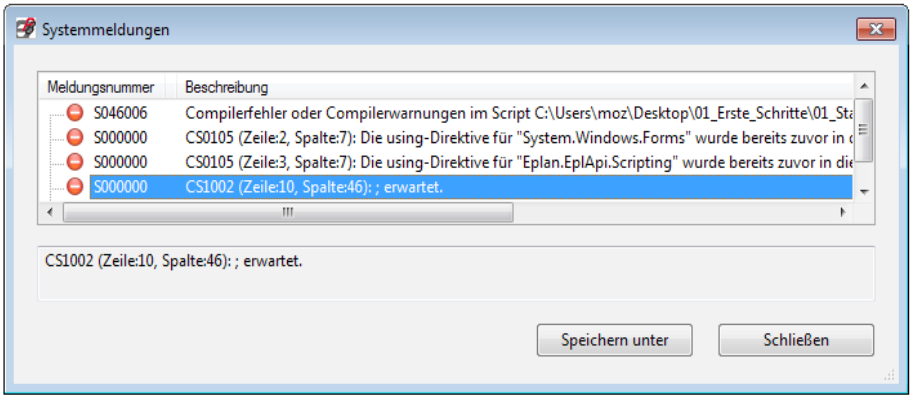


Bild 2.10
Fehler: Using-Direktive wurde bereits zuvor verwendet.

In den Systemmeldungen werden vier Fehler angezeigt:

1. Compilerfehler: Information, dass das Script einen Fehler verursacht hat
2. Die Using Direktive wurde bereits zuvor in diesem Namespace verwendet.
3. Die Using Direktive wurde bereits zuvor in diesem Namespace verwendet.
4. Semikolon erwartet: Dies ist der eigentliche Fehler, welchen Sie als einzigen im Script korrigieren müssen. Danach wird das Script korrekt ausgeführt.

Um eventuell auftretende Fehler schnell zu finden, können Sie in Visual Studio auch die Zeilennummern einblenden.

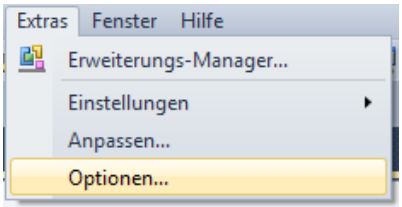


Bild 2.11
Optionen in Visual Studio

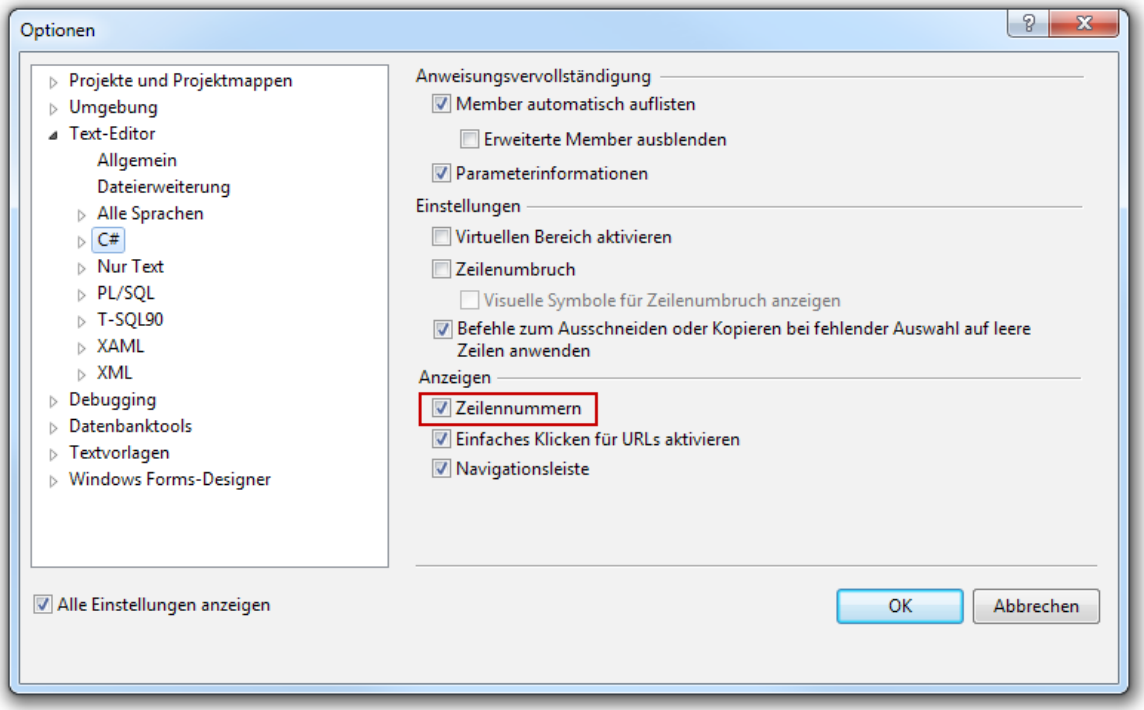


Bild 2.12 Anzeigen der Zeilennummern aktivieren

Semikolon erwartet

Gerade am Anfang vergisst man öfters mal, den Code mit dem Semikolon abzuschließen. Wenn Sie die Fehlerliste eingeblendet haben, können Sie dies schon vor dem Ausführen in EPLAN sehen. Im Code-Fenster wird die Stelle rot unterstrichen.

Bild 2.13
Visual Studio: Semikolon erwartet

Aber auch EPLAN überprüft den Code, bevor dieser geladen oder ausgeführt wird.

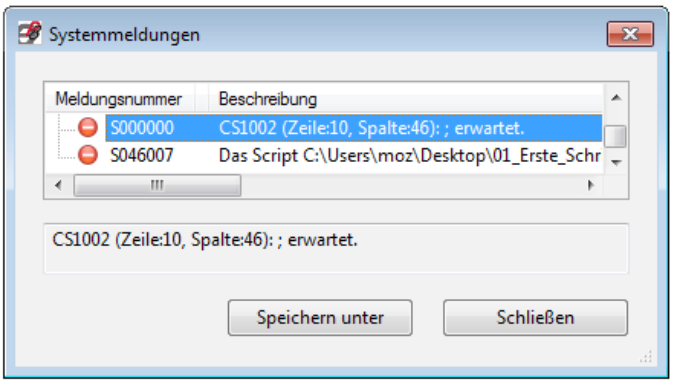
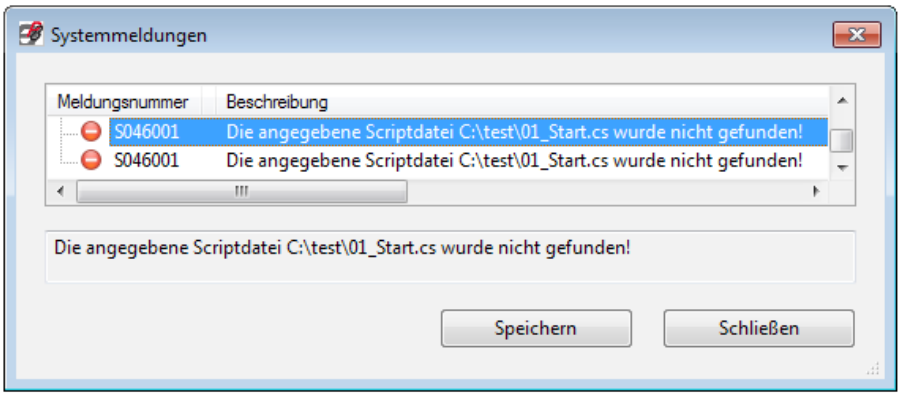


Bild 2.14
EPLAN: Semikolon erwartet

Scriptdatei nicht gefunden

Wenn Sie auf Dateiebene den Dateinamen eines Scripts ändern oder es löschen, weiß EPLAN davon nichts. Darum bekommen Sie in diesem Fall auch einen Fehler in den Systemmeldungen angezeigt. Bei jedem Start von EPLAN werden die Scripte überprüft. Das Script muss entladen und neu geladen werden. Dies ist nur notwendig, wenn ein Script geladen wurde (wie im nächsten Abschnitt).



2.1.2 DeclareAction

Wollen Sie eine Action häufiger verwenden, macht es Sinn, diese dauerhaft zu laden. Kopieren Sie das Script *01_Start.cs* über **Strg** + **C** oder per Kontextmenü.

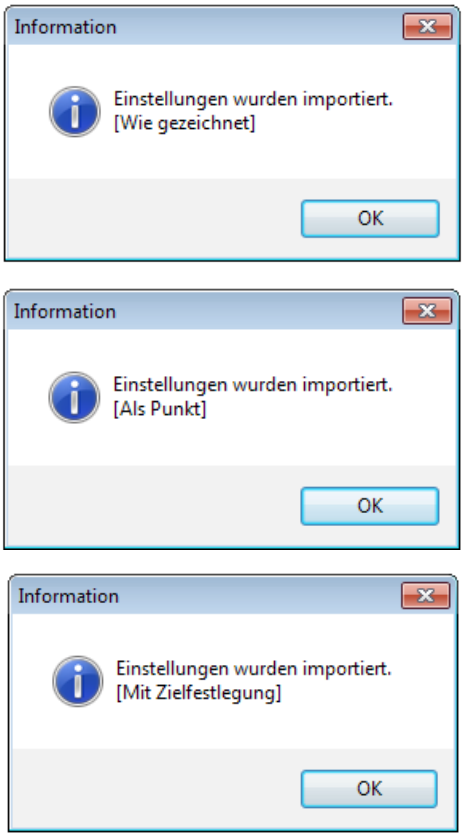


Bild 2.87
Modale Dialoge zur Anzeige von Einstellungen

2.6 Menüs

Wie bereits am Anfang des Buches beschrieben, können in EPLAN eigene Menüs erzeugt werden. Es gibt mehrere Arten von Menütypen, welche unterschiedliche Funktionen haben:

- Menüpunkt in Dienstprogrammen
- Menüpunkt in einem bestehenden Menü
- Hauptmenü mit einem Menüpunkt
- Popup-Menü

Jedem neu hinzugefügten Menüpunkt muss eine Action zugewiesen werden. Dies kann eine EPLAN- oder eine eigene Action sein. In den folgenden Beispielen werden wir immer die Action mit Namen *MenuAction* verwenden. Es darf kein anderes Script mit dieser Action vorhanden sein, wenn das Script in EPLAN geladen wird.

Wird ein Menü-Script entladen, wird das Menü nicht immer entfernt. Auch bei Änderungen werden diese oft nicht übernommen, oder das Menü enthält doppelte Einträge. Tritt eines dieser Probleme auf, müssen Sie EPLAN neu starten.

2.6.1 Menüpunkt in Dienstprogramme

Erstellen Sie ein neues Script mit dem Namen  *01_Menüpunkt_in_Dienstprogramme.cs* und zusätzlich einen neuen Ordner für das Kapitel Menüs ( *06_Menüs*). Als Attribute benötigen wir *[DeclareAction("MenuAction")]* für die Action und *[DeclareMenu]* für den dazugehörigen Menüpunkt.

Unsere Action soll nur eine Messagebox mit dem Text *Action wurde ausgeführt!* anzeigen, um die Funktionalität zu testen. Erstellen Sie die zwei folgenden Funktionen:

- *ActionFunction()*: Hier wird die Action für das Menü bereitgestellt.
- *MenuFunction()*: Hier wird die Action aufgerufen.

```
[DeclareAction("MenuAction")]
public void ActionFunction()
{
  MessageBox.Show("Action wurde ausgeführt!");

  return;
}
```

In der neuen Funktion *MenuFunction()* erstellen wir nun ein Objekt vom Typ *Menu*, welches im EPLAN-Namespace zu finden ist:

```
Eplan.EplApi.Gui.Menu oMenu = new Eplan.EplApi.Gui.Menu();
```

Es ist (ähnlich wie bei Settings) nötig, den kompletten Pfad zum Objekt zu hinterlegen, da es z. B. in *System.Windows.Forms* auch ein Objekt mit *Menu* gibt.

Um einen einfachen Menüpunkt unter dem Dienstprogramm zu erstellen, nutzen wir die Methode *AddMenuItem()*. Dadurch wird das Menü *Dienstprogramme* durch den angegeben Menüpunkt ergänzt. Die Methode besitzt folgende Eigenschaften:

- Name Menüpunkt
- Name Action

Als Actionnamen tragen Sie die zuvor initialisierte *MenuAction* ein:

```
oMenu.AddMenuItem(
  "Menüpunkt am Ende von Dienstprogramme", // Name: Menüpunkt
  "MenuAction" // Name: Action
  );
```

Sie können diesen Programmcode auch in einer Zeile schreiben. Wegen der besseren Lesbarkeit habe ich den Code hier aufgeteilt.

Da bei späteren Methoden nicht immer klar ist, für was eine Überladung zuständig ist, füge ich die Beschreibung immer als Kommentar hinzu.

Wichtig bei Menü-Scripten ist es, wie schon beschrieben, diese zu laden. Haben Sie alles richtig gemacht, erscheint nun ein neuer Menüpunkt unter *Dienstprogramme*.

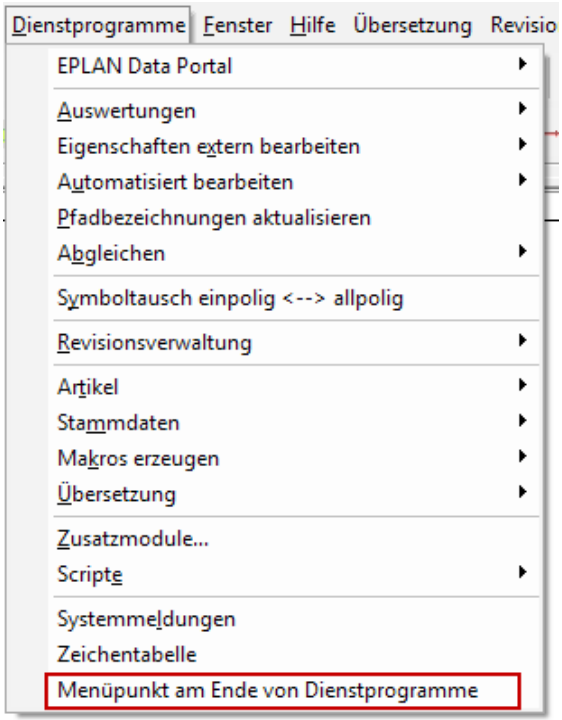


Bild 2.88
Neuer Menüpunkt unter Dienstprogramme

Das komplette Script sieht wie folgt aus:

```
using System.Windows.Forms;
using Eplan.EplApi.Scripting;

public class Class
{

    [DeclareAction("MenuAction")]
    public void ActionFunction()
    {
        MessageBox.Show("Action wurde ausgeführt!");

        return;
    }

    [DeclareMenu]
    public void MenuFunction()
    {
        Eplan.EplApi.Gui.Menu oMenu = new Eplan.EplApi.Gui.Menu();
```

```
            oMenu.AddMenuItem(
                "Menüpunkt am Ende von Dienstprogramme", // Name: Menüpunkt
                "MenuAction" // Name: Action
                );

            return;
        }
}
```

2.6.2 Bestehendes Menü erweitern

Für die zweite Variante des Menü-Objektes kopieren Sie das erste Script und vergeben den Namen *02_Bestehendes_Menü_erweitern.cs*. Lassen Sie die erste Funktion unberührt, und verändern Sie nur die Methode des Menü-Objektes.

Damit ein bestehendes Menü erweitert werden kann, steht die Methode *AddMenuItem()* bereit. Folgende Überladungen stehen zur Verfügung:

- Name Menüpunkt
- Name Action
- Statustext
- Menü-ID
- hinter/vor Menüpunkt

Der Statustext wird in der Fußzeile von EPLAN angezeigt und kann eine längere Beschreibung enthalten:

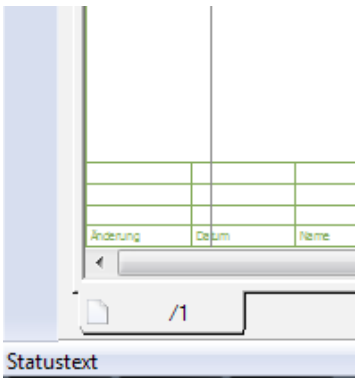
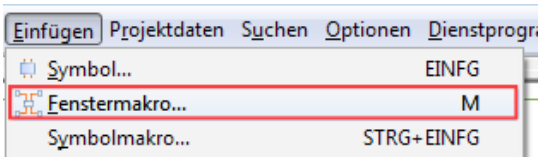


Bild 2.89
Statustext

Die Menü-ID (*Integer*) dient als Referenz-Menüpunkt, um die Position festzulegen. Über den Diagnose-Dialog (**Strg** + **^**) können wir nach dem Aufrufen des Menüpunktes die Menü-ID herausfinden. Führen Sie dazu zuerst *Einfügen* > *Fenstermakro* aus und öffnen dann den Diagnose-Dialog.



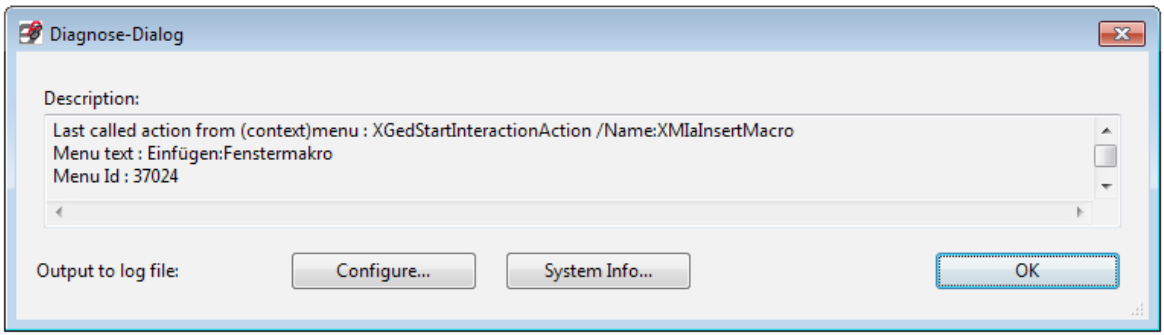


Bild 2.90 Menü-ID über den Diagnose-Dialog herausfinden

Mit der letzten Eigenschaft können Sie festlegen, ob der neue Menüpunkt vor oder hinter dem Referenz-Menüpunkt eingefügt werden soll:

- 1: hinter Menüpunkt
- 0: vor Menüpunkt

Die beiden letzten Überladungen „Separator davor/dahinter“ zeigen eine Trennlinie im Menü. Dieser Boolesche Wert hat die bekannten Zustände *true* oder *false*.

Ändern Sie das Script wie folgt:

```
using System.Windows.Forms;
using Eplan.EplApi.Scripting;

public class Class
{

    [DeclareAction("MenuAction")]
    public void ActionFunction()
    {
        MessageBox.Show("Action wurde ausgeführt!");

        return;
    }

    [DeclareMenu]
    public void MenuFunction()
    {
        Eplan.EplApi.Gui.Menu oMenu = new Eplan.EplApi.Gui.Menu();

        oMenu.AddMenuItem(
            "Bestehendes Menü erweitern", // Name: Menüpunkt
```

```
            "MenuAction", // Name: Action
            "Statustext", // Statustext
            37024, // Menü-ID: Einfügen/Fenstermakro...
            1, // 1 = hinter Menüpunkt, 0 = vor Menüpunkt
            false, // Separator davor anzeigen
            false // Separator dahinter anzeigen
            );

        return;
    }
}
```

Stellen Sie sicher, dass kein Script geladen ist, und laden Sie die neue Datei. Nach dem Eintrag *Einfügen* > *Fenstermakro* erscheint ein neuer Untermenüpunkt:

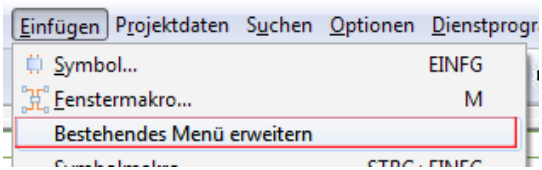


Bild 2.91
Bestehendes Menü erweitern

2.6.3 Hauptmenü mit einem Untermenüpunkt

Kopieren Sie das Script erneut und vergeben den Namen *03_Hauptmenü_mit_Untermenüpunkt.cs*. Die erste Funktion lassen Sie unverändert, da die Action erneut ausgeführt wird. In diesem Beispiel erzeugen Sie ein neues Menü in der EPLAN-Oberfläche. Zusätzlich müssen Sie diesem Menü einen Menüpunkt hinzufügen.

Dafür verwenden Sie die Methode *AddMainMenu()*. Das Objekt hat folgende Eigenschaften:

- Name Menü
- neben Menüpunkt
- Name Menüpunkt
- Name Action
- Statustext
- hinter/vor Menüpunkt

Mit „neben Menüpunkt“ geben Sie an, an welcher Stelle das Menü eingefügt wird. Zusätzlich können Sie festlegen, ob das Menü davor oder dahinter platziert werden soll:

- hinter Menüpunkt: 1
- vor Menüpunkt: 0

Möchten Sie z. B. an erster Stelle das Menü erzeugen, müssten Sie „Projekt“ und „0“ angeben.

Es ist aber ratsam, die Reihenfolge der Grundeinstellung von EPLAN nicht zu verändern. Sonst finden sich andere User meist nicht so schnell zurecht. Aus diesem Grund fügen wir neue Menüs immer hinter dem Menü *Hilfe* ein.

```
oMenu.AddMainMenu(
  "Menü 1", // Name: Menü
  "Hilfe", // neben Menüpunkt
  "Hauptmenü mit einem Menüpunkt", // Name: Menüpunkt
  "MenuAction", // Name: Action
  "Statustext", // Statustext
  1 // 1 = hinter Menüpunkt, 0 = vor Menüpunkt
  );
```

Stellen Sie sicher, dass kein Script geladen ist, und laden Sie die neue Datei. Nun erscheint ein neuer Menüpunkt mit dem Namen *Menü 1* auf der Oberfläche:

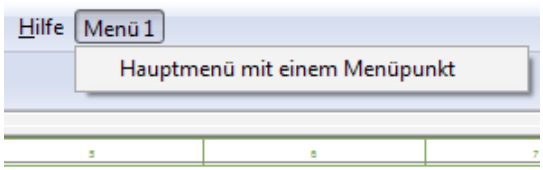


Bild 2.92
Eigenes Hauptmenü

Das komplette Script sieht so aus:

```
using System.Windows.Forms;
using Eplan.EplApi.Scripting;

public class Class
{

    [DeclareAction("MenuAction")]
    public void ActionFunction()
    {
        MessageBox.Show("Action wurde ausgeführt!");

        return;
    }

    [DeclareMenu]
    public void MenuFunction()
    {
        Eplan.EplApi.Gui.Menu oMenu = new Eplan.EplApi.Gui.Menu();

        oMenu.AddMainMenu(
            "Menü 1", // Name: Menü
            "Hilfe", // neben Menüpunkt
            "Hauptmenü mit einem Menüpunkt", // Name: Menüpunkt
            "MenuAction", // Name: Action
            "Statustext", // Statustext
            1 // 1 = hinter Menüpunkt, 0 = vor Menüpunkt
            );

        return;
    }
}
```

```
    pbr.PerformStep();
    pbr.Value = 0;

    this.Close();

    return;
}
```

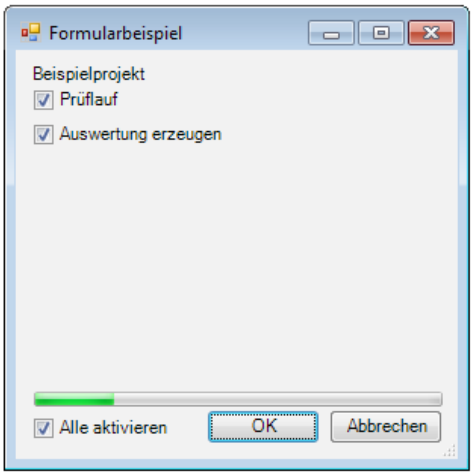


Bild 2.122
Progressbar

2.8.7 Mauszeiger ändern

Im Scripting werden oft Actions ausgeführt, die EPLAN etwas beschäftigen, aber dem User darüber keine Rückmeldung geben. Hier ist es oft zu aufwändig, extra eine Progressbar dafür anzulegen. Für diesen Anwendungsfall eignet sich das Ändern des Mauszeigers (engl. *Cursor*). Wenn der Anwender eine Sanduhr sieht, ist diesem bewusst, dass etwas berechnet/ausgeführt wird.

Erstellen Sie ein neues Script mit dem Attribut *[Start]*. Als Namen vergeben Sie *03_Cursor.cs*. Dies wird eine normale Scriptdatei ohne Formular.

Der gerade sichtbare Cursor ist immer das Objekt *Cursor.Current*. Diesem Objekt können Sie verschiedene Darstellungsarten zuweisen. In unserem Beispiel verwenden wir zwei Arten:

- *AppStarting*: Wird angezeigt, wenn ein Programm gestartet wird.
- *WaitCursor*: Wird angezeigt, wenn das Programm etwas berechnet/ausführt.

Um den Mauszeiger länger sichtbar zu machen, fügen Sie wieder eine Wartezeit ein (drei Sekunden):

```
using Eplan.EplApi.Base;
using Eplan.EplApi.Scripting;
using System.Threading;
using System.Windows.Forms;

public class Class
{
    [Start]
    public void Function()
    {
        Cursor.Current = Cursors.AppStarting;
        Thread.Sleep(3000);
        Cursor.Current = Cursors.WaitCursor;
        Thread.Sleep(3000);

        return;
    }
}
```

Folgende Darstellungsarten des Mauszeigers gibt es unter .NET:

AppStarting	
Arrow / Default	
Hand	
Help	
No	
SizeAll	
WaitCursor	

2.8.8 ListView

Erstellen Sie ein neues Script aus der Vorlage *01_Vorlage.cs* mit dem Namen *04_Projektsuche.cs*, und passen Sie die Objektnamen ähnlich wie im Beispiel in Abschnitt 2.8.2, „Button“, an.

Fügen Sie folgende Steuerelemente zum Formular hinzu:

- Button: *btnOK*
- Button: *btnCancel*
- Button: *btnSearch*

- Button: *btnOpenProject*
- TextBox: *txtSearch*
- StatusStrip: *st*
- ToolStripStatusLabel: *lbl*
- ListView: *liviResult*
- Spalte: *Projektname*
  - Spalte: Pfad
  - Spalte: Erweiterung

Um das Objekt *ToolStripStatusLabel* einzufügen, muss zuerst ein *StatusStrip* platziert werden. Dieses Objekt finden Sie unter *Menüs & Symbolleisten*.

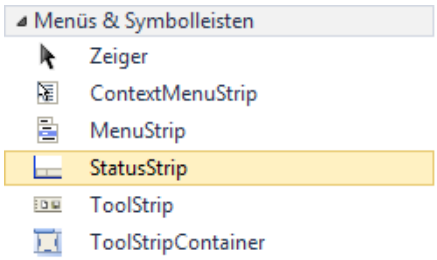


Bild 2.123
Toolbox: *StatusStrip*

Erstellen Sie nun *ToolStripStatusLabel* über das Symbol *Neu* am linken unteren Rand des *StatusStrips*.

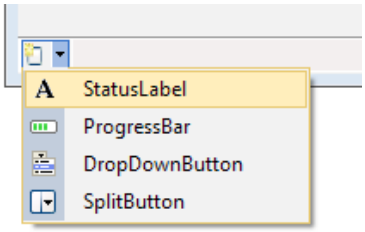


Bild 2.124
StatusLabel erzeugen

Um Spalten zu einem ListView hinzuzufügen, markieren Sie das Objekt im Designer. Nun erscheint ein kleiner Pfeil am oberen rechten Rand.

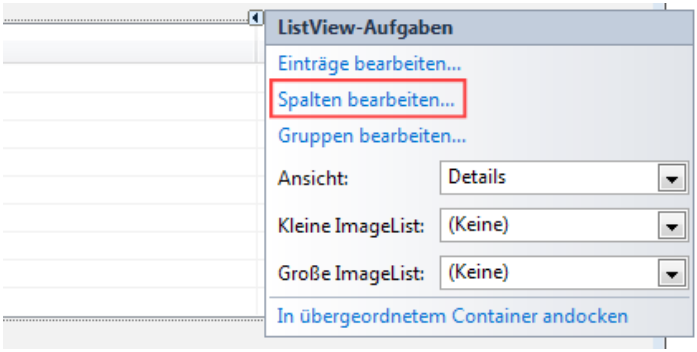


Bild 2.125
Spalten zum ListView hinzufügen

Wichtig ist, dass dem Steuerelement *ListView* folgende Eigenschaften zugewiesen werden:

- **FullRowSelect**: *True*
- **HideSelection**: *False*
- **Sorting**: *Ascending*
- **View**: *Details*

Das Script dient dazu, den EPLAN-Projekte-Order nach Projekten zu durchsuchen und ggf. zu öffnen. Voraussetzung hierfür ist, dass alle Projektdateien (*.elk) in einem Ordner sind.

In Bild 2.126 sehen Sie einen Beispielaufbau des Formulars.

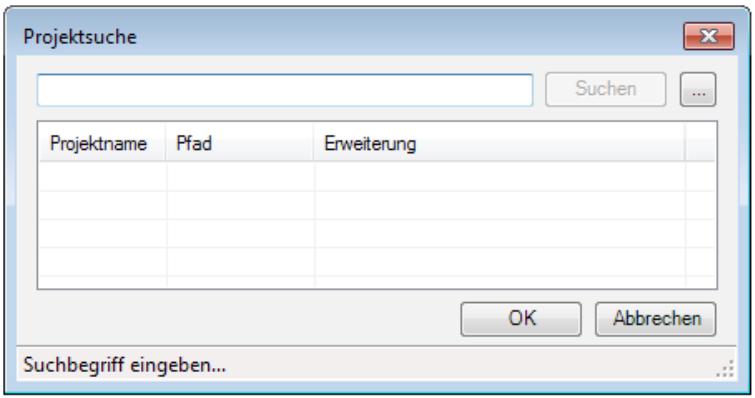


Bild 2.126
Dialog für die Projektsuche

Wechseln Sie nun in die Codeansicht. Als Erstes wird über *Menü* > *Projekt* > *Projektsuche...* ein neuer Menüeintrag erzeugt.

```
[DeclareMenu]
public void MenuFunction()
{
  Eplan.EplApi.Gui.Menu oMenu = new Eplan.EplApi.Gui.Menu();
  oMenu.AddMenuItem
       (
       "Projektsuche...",
       "SearchProjects",
       "Projekt(e) suchen und öffnen",
       35002,
       1,
       false,
       false
       );

  return;
}
```

Nun erzeugen wir eine Variable, welche in **jeder** Funktion verfügbar ist. Hierfür stellen wir der Variablen-Deklaration ein *public* zur Verfügung. Solch eine Initialisierung muss immer in der Klasse (Class) geschehen, da hier übergeordnete Elemente zugewiesen werden.