Einstieg in die Knotenentwicklung

1 Voraussetzungen

Dieser Einstieg soll die Grundlagen in der Entwicklung einer Knotenbibliothek für das visuelle Programmiersystem DynamicNodes vermitteln. Es werden Programmierkenntnisse in C# vorausgesetzt. Der Umgang mit der Entwicklungsumgebung wird erläutert.

1.1 Entwicklungsumgebung

Für die ersten Schritte beim Entwickeln von einer Knotenbibliothek sind nur wenige Voraussetzungen zu erfüllen. Neben einer funktionstüchtigen *DynamicNodes-Installation* ist lediglich eine *Entwicklungsumgebung für eine .NET-Sprache* erforderlich. Die in dieser Anleitung enthaltenen Beispiele sind in C# verfasst und es wird der Umgang mit de kostenlosen *Microsoft Visual C# Express*[1] erklärt.

1.2 Optionales Debug-Paket

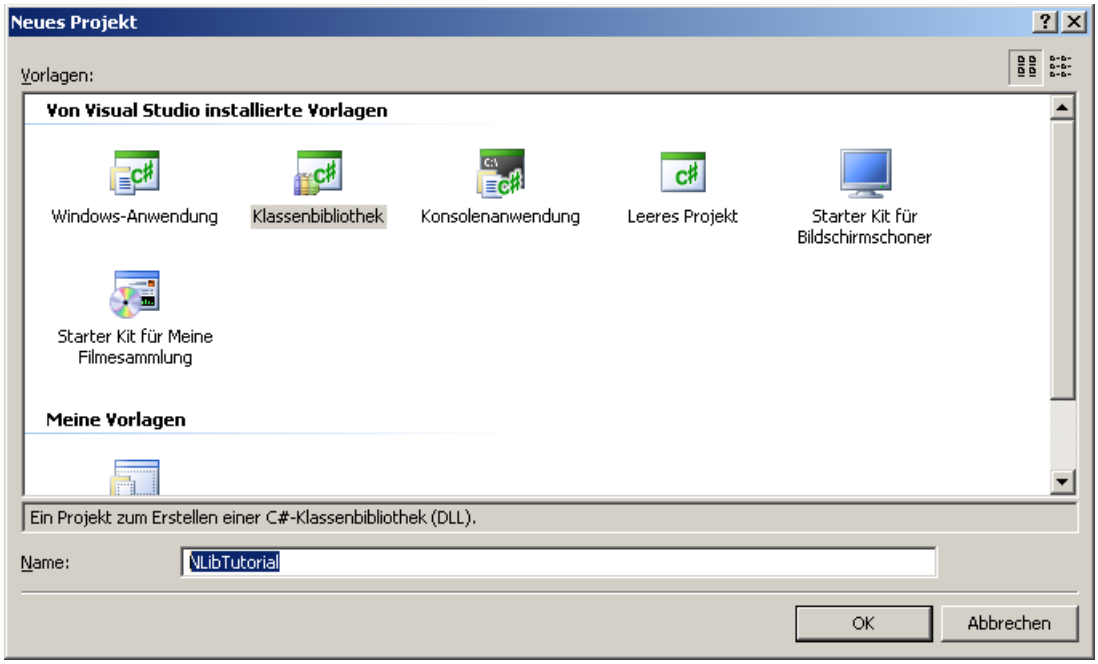
Falls die Möglichkeit besteht, ist es empfehlenswert, anstelle der Standard-Installation von DynamicNodes, das Debug-Paket zu verwenden. Das Debug-Paket ist eine ZIP-Datei, welche sich von der Homepage herunterladen lässt und in beliebiges Verzeichnis entpackt werden kann. Es enthält einen Ordner mit einer kompletten DynamicNodes-Installation, deren Programme und Programmbibliotheken im Debug-Modus kompiliert wurden. Zusätzlich enthält es die Programm-Debug-Datenbanken mit der Endung pdb. Wenn anstelle der Standard-Installation dieses Paket verwendet wird, ist das Debuggen von Knotenbibliotheken komfortabler.

[1] http://msdn.microsoft.com/vstudio/express/downloads/

2 Das erste Projekt

2.1 Projekt anlegen

Eine Knotenbibliothek ist technisch gesehen ein .NET-Assembly in Form einer DLL. Um eine Knotenbibliothek zu entwickeln, ist der Anfang demzufolge ein DLL-Projekt. Unter Microsoft Visual C# Express wird ein solches Projekt beispielsweise mit *Datei → Neues Projekt → Klassenbibliothek* angelegt.



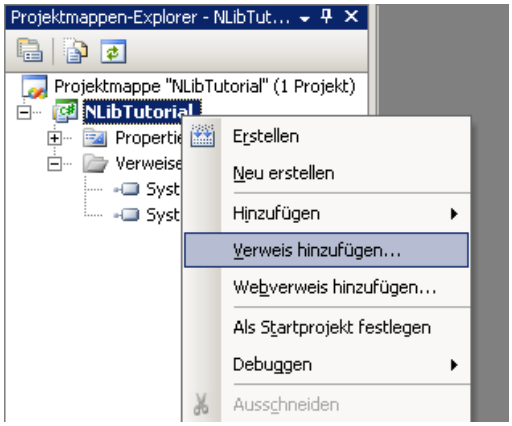
Als Projekt-Name wird für die Knotenbibliotheken nach Möglichkeit das Präfix `NLib` verwendet. So heißt das Projekt für die Beispiele in diesem Tutorial `NLibTutorial`, was auch der Name des Assemblies ist.

Die Quellcode Datei mit dem Namen `Class1.cs`, welche Visual C# Express automatisch anlegt und öffnet, kann geschlossen und gelöscht werden. Nun sollte das Projekt erst einmal gespeichert werden.

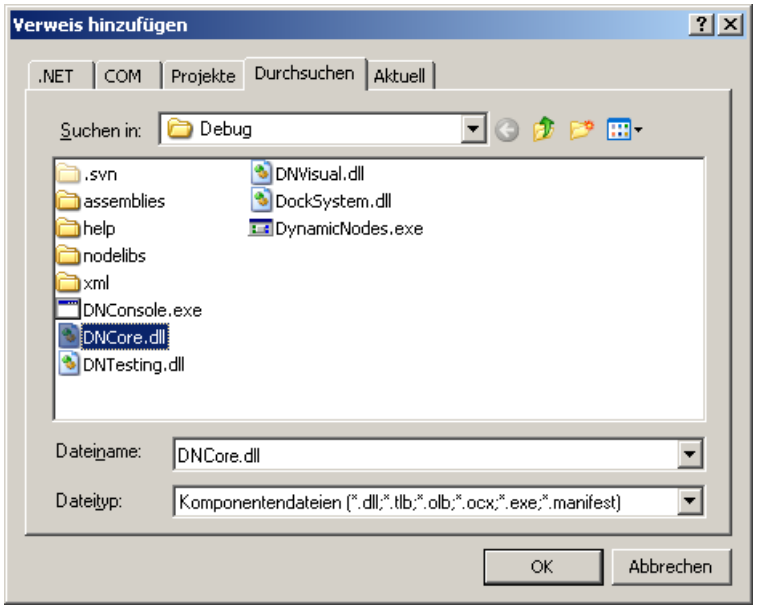
2.2 Verweise einrichten

Das Projekt benötigt mindestens Verweise auf die .NET-Framework-Assemblies `System` und `System.Xml`. Wenn ein neues Klassenbibliothek-Projekt mit Microsoft Visual C# Express erstellt wird, besitzt es automatisch Verweise auf diese beiden .NET-Framework-Assemblies. Es wird noch der Verweis auf das `DNCore`-Assembly benötigt, welches sich im Installationsverzeichnis von DynamicNodes (oder im Debug-Paket) befindet.

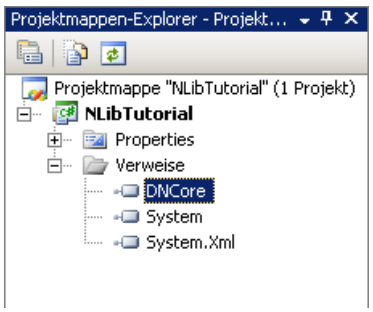
Der Verweis auf `DNCore` kann unter Microsoft Visual C# Express mit dem Dialog „Verweis hinzufügen“ erstellt werden. Mit einem Rechtsklick auf das Projekt im Projektmappen-Explorer wird ein Kontext-Menü geöffnet, dessen Eintrag „Verweis hinzufügen…“ den entsprechenden Dialog öffnet.



Unter der Registerkarte „Durchsuchen“ muss nur noch die `DNCore.dll` im Installationsverzeichnis (oder im Debug-Paket) von DynamicNodes ausgewählt werden.



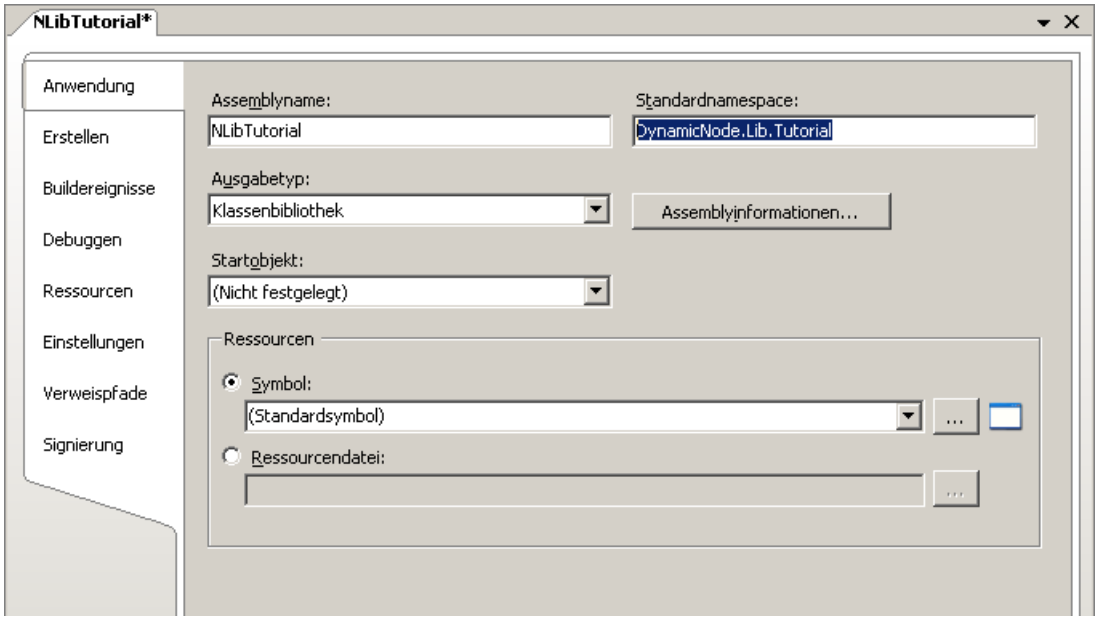
Der Projektmappen-Explorer sollte jetzt in etwa so aussehen:



2.3 Standard-Namensraum definieren

Für die Knotenbibliothek sollte ein eigener Namensraum definiert werden. Der Namensraum sollte von dem jeweiligen Knotenentwickler nach den üblichen Konventionen gewählt werden, so dass keine Namenskonflikte mit anderen Assemblies/Knotenbibliotheken auftreten können. Z.B. könnte Der Entwickler Hans Mustermann mit einer Homepage www.mustermann.de den Namensraum `de.mustermann.dnlib` verwenden. Der Namensraum `DynamicNode` ist für das DynamicNode-System und die Standard- Knotenbibliotheken reserviert. Der Namensraum für die Beispiele in diesem Tutorial ist `DynamicNode.Lib.Tutorial`.

In Microsoft Visual C# Express lässt sich für ein Projekt ein Standard-Namensraum festlegen, in welchem automatisch generierte Klassen erzeugt werden. Das Ist möglich auf der Eigenschaftsseite des Projektes. Dieser wird aufgerufen mit einem Doppelklick im Projektmappen-Explorer auf den Eintrag „Properties“. In der Rubrik „Anwendung“ befindet sich oben rechts das Feld „Standardnamespace“.

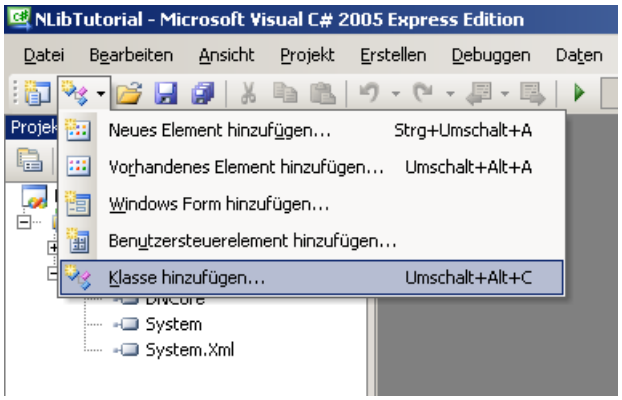


2.4 Der erste Knoten

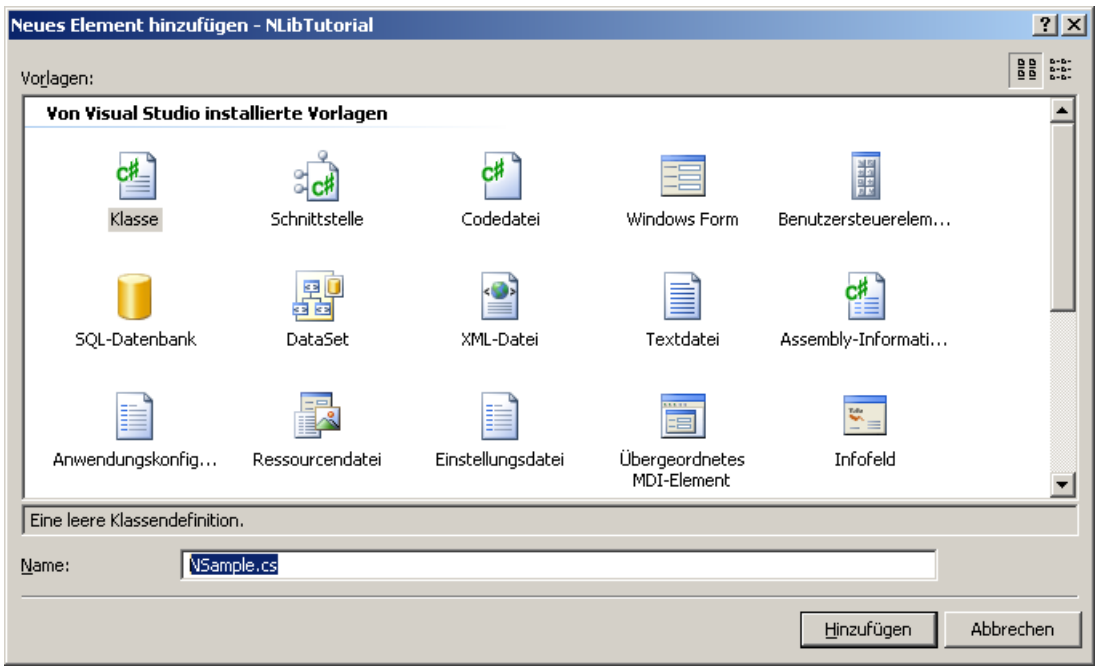
Nach dem das DLL-Projekt angelegt ist und die notwendigen Verweise besitzt, kann es mit dem ersten eigentlichen Knoten losgehen. Ein Knoten oder besser ein Knotentyp ist eine Klasse, welche die Schnittstelle `DynamicNode.Core.INode` implementiert. Diese Schnittstelle ist jedoch recht umfangreich und sollte nur von erfahrenen Entwicklern direkt implementiert werden denen die Standardimplementierung nicht ausreicht. Die Schnittstelle muss also nicht direkt implementiert werden, sondern es genügt die neue Klasse von der abstrakten Standardimplementierung `DynamicNode.Core.Node` abzuleiten.

Die einzige abstrakte Methode, welche von einer Unterklasse von `DynamicNode.Core.Node` implementiert werden muss, ist `public void Work()`.

Um eine Quellcode-Datei anzulegen, wo der oben aufgeführte Quellcode eingegeben werden kann, anzulegen, genügt ein Klick auf den Symbolleisten „Neues Element“ und in dem aufklappenden Menü auf den Eintrag „Klasse hinzufügen“.



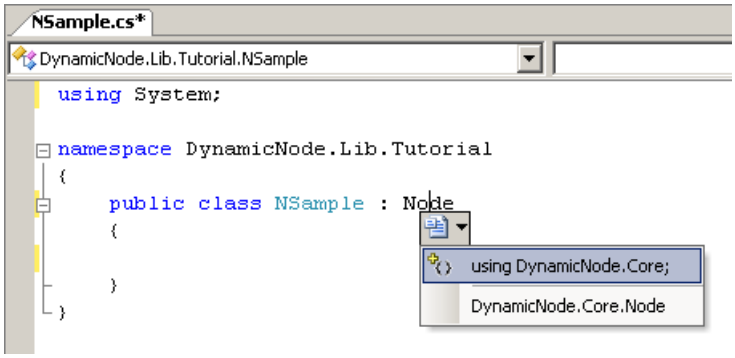
Für Klassennamen von Knoten gelten die allgemeinen Konventionen von Klassennamen in der C#-Programmierung. Als Besonderheit wird einem Knotennamen ein großes N vorangestellt, in diesem Beispiel also `NSample`.



Die automatisch geöffnete Quellcode-Datei kann mit wenigen Änderungen an das folgende Listing angepasst werden.

```
using System;
using DynamicNode.Core;

namespace DynamicNode.Lib.Tutorial
{
    public class NSample : Node
    {
        public override void Work()
        {

        }
    }
}
```

Es werden zunächst nur die absolut notwendigen `using`-Direktiven aufgeführt. Alle weiteren notwendigen `using`-Direktiven werden im Folgenden nicht immer erwähnt. Wenn für einen Klassennamen die `using`-Direktive in der Quellcode-Datei fehlt, bietet Microsoft Visual C# Express ein Quick-Fix-Menü an, um die Direktive zu erstellen. Das Quick-Fix-Menü kann mit dem Short-Cut *Alt + Umschalt + F10* aufgeklappt werden.



Dieser Quellcode würde nun bereits ausreichen, um einen lauffähigen Knoten zu kompilieren. In Microsoft Visual C# Express ist das über den Menüeintrag *Erstellen → Projektmappe erstellen* möglich. Später kann dieser Punkt übersprungen werden, denn Microsoft Visual C# Express kompiliert nach einer Änderung am Quellcode automatisch, wenn das Projekt ausgeführt wird.

Wenn keine Fehler oder Warnungen angezeigt wurden, ist die Klassenbibliothek erfolgreich in eine DLL-Datei kompiliert worden.

3 Starten/Debuggen

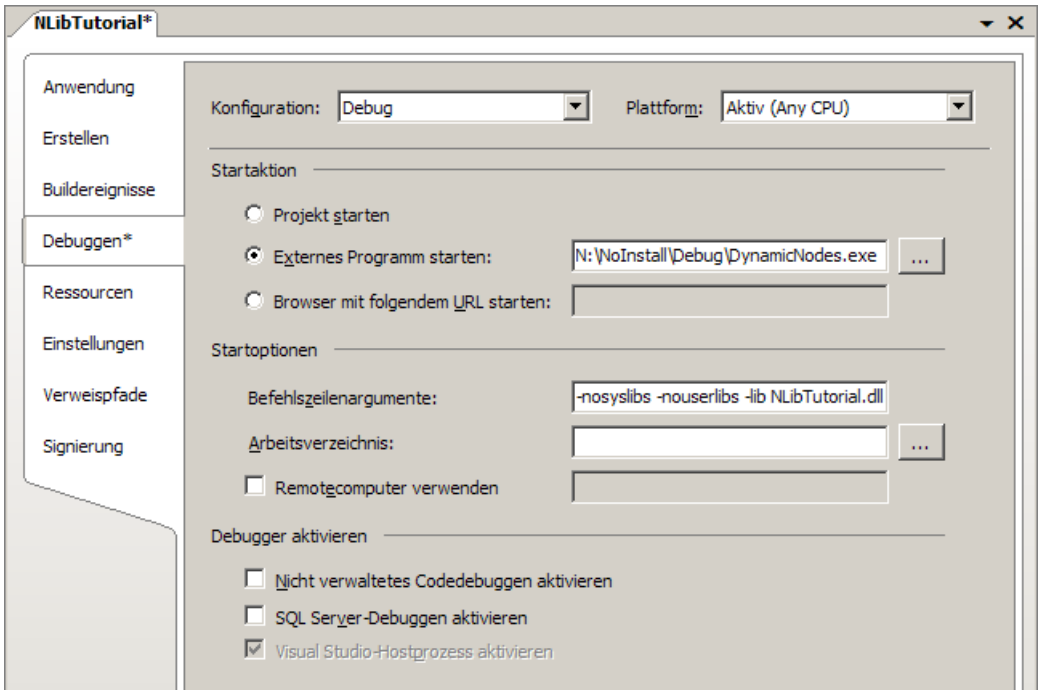
Um die Knotenbibliothek nun in DynamicNodes zu laden gibt es mehrere Möglichkeiten. DynamicNodes lädt Knotenbibliotheken prinzipiell von zwei Orten. Der erste ist das System-Bibliotheks-Verzeichnis `nodelibs` im Installationsverzeichnis von DynamicNodes. Der zweite ist das Benutzer-Bibliotheks-Verzeichnis, welches typischerweise unter `…\Eigene Dateien\DynamicNodes\nodelibes` (Windows XP) oder `…\Documents\DynamicNodes\nodelibs` (Windows Vista) zu finden ist.

Die kompilierte Knotenbibliothek könnte also in eines dieser Verzeichnisse kopiert werden. Jedoch ist das natürlich nach jeder Änderung notwendig, was keine flüssige Entwicklungsarbeit gestattet. DynamicNodes unterstützt aber eine Reihe von Befehlszeilenargumente die an dieser Stelle aushelfen. Das wichtigste ist `-lib "DLL-Datei"`, mit dem DynamicNodes explizit eine Knotenbibliothek zum Laden übergeben werden kann.

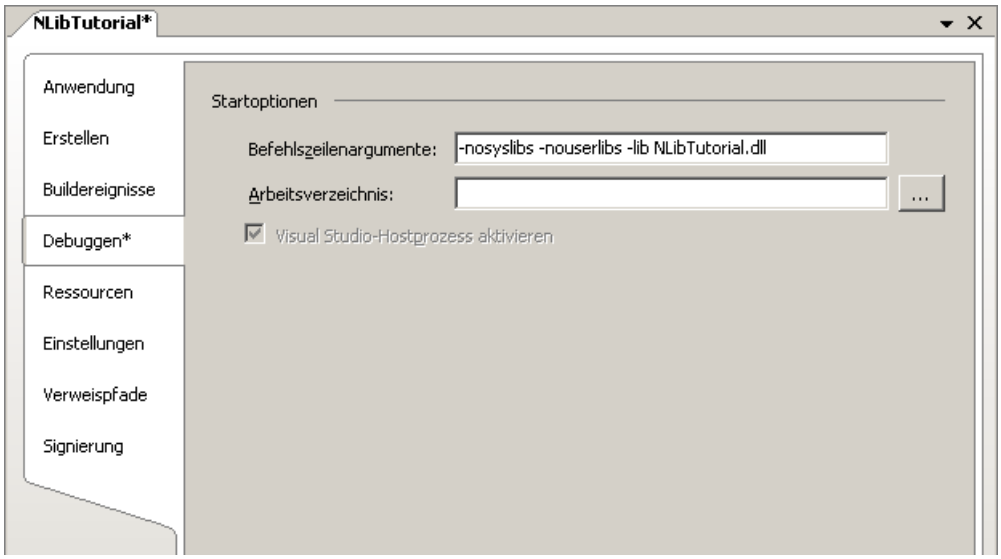
3.1 Hack Express ☺

Um die Knotenbibliothek zu debuggen, muss ja nun DynamicNodes, und nicht die kompilierte DLL-Datei gestartet werden. In dem kostenpflichtigen Microsoft Visual Studio kann für ein Projekt angegeben werden ob ein externes Programm ausgeführt werden soll, um das Projekt zu debuggen. In Microsoft Visual C# Express Ist dieses nette Feature leider nicht enthalten.

Die Eigenschaftsseite in Microsoft Visual Studio 2005:



Die Eigenschaftsseite in Microsoft Visual C# 2005 Express:



Versteckt sind viele Fähigkeiten von Microsoft Visual Studio jedoch auch in der Express-Variante verfügbar.

Zunächst muss die Projektmappe in Microsoft Visual C# Express geschlossen werden. Anschließend reicht es, im Projektverzeichnis eine Datei mit dem Namen `<Projektname>.csproj.user` und folgendem Inhalt anlegen. (Im Beispiel würde die Datei `NLibTutorial.csproj.user` heißen.) Dazu reicht ein gewöhnlicher Texteditor wie der Editor von Windows.

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <PropertyGroup>
    <StartAction>Program</StartAction>
    <StartProgram>D:\DynamicNodes\DebugPaket\DynamicNodes.exe</StartProgram>
  </PropertyGroup>
</Project>
```

Der fett gedruckte Pfad muss dabei an das Installationsverzeichnis von DynamicNodes oder den Pfad zum Debug-Paket angepasst werden.

Nun kann die Projektmappe wieder geöffnet werden. Ohne dass es sichtbar wird, ist nun für das Projekt eingestellt, dass für das Debugging die `DynamicNodes.exe` aufgerufen werden soll.

3.2 Befehlszeilenargumente

Es gibt eine Reihe nützliche Befehlszeilenargumente für das Debugging. Darunter sind `-nosyslibs`, `-nouserlibs`, `-nosplash` und `-lib "DLL-Datei"`. Das Argument `-nosyslibs` weist die Laufzeitumgebung von DynamicNodes an, keine Bibliotheken aus dem System-Bibliotheks-

Verzeichnis zu laden. Das Argument `-nouserlibs` hat die gleiche Wirkung für das Benutzer-Bibliotheks-Verzeichnis. Mit dem Argument `-nosplash`, kann verhindert werden, dass DynamicNodes beim Starten, den schönen Splash-Screen anzeigt, der beim Testen von Knoten(bibliotheken) doch ein wenig auf die Nerven geht. Und mit `-lib "DLL-Datei"` kann, wie oben erwähnt, eine Knotenbibliothek explizit an die Laufzeitumgebung übergeben werden.

Auf der Eigenschaftsseite des Projektes kann nun das Feld „Befehlszeilenargumente“ verwendet werden, um die Argumente für den Aufruf von DynamicNodes anzugeben.
An dieser Stelle ist es evtl. sinnvoll `-nosyslibs` und `-nouserlibs` einzutragen, falls die zu entwickelnde Knotenbibliothek zunächst isoliert getestet werden soll. Sind Knotentypen aus den installierten System- oder Benutzerbibliotheken für das Testen der zu entwickelnden Knoten erforderlich, sind die Argumente natürlich wegzulassen.
Mit dem Argument `-lib "Bibliotheksdatei"`, kann DynamicNodes nun explizit die zu entwickelnde Knotenbibliothek übergeben werden.

So könnten die Befehlszeilenargumente in diesem Beispiel für den Anfang wie folgt aussehen:

```
-nosyslibs -nouserlibs -lib NLibTutorial.dll
```

oder aber

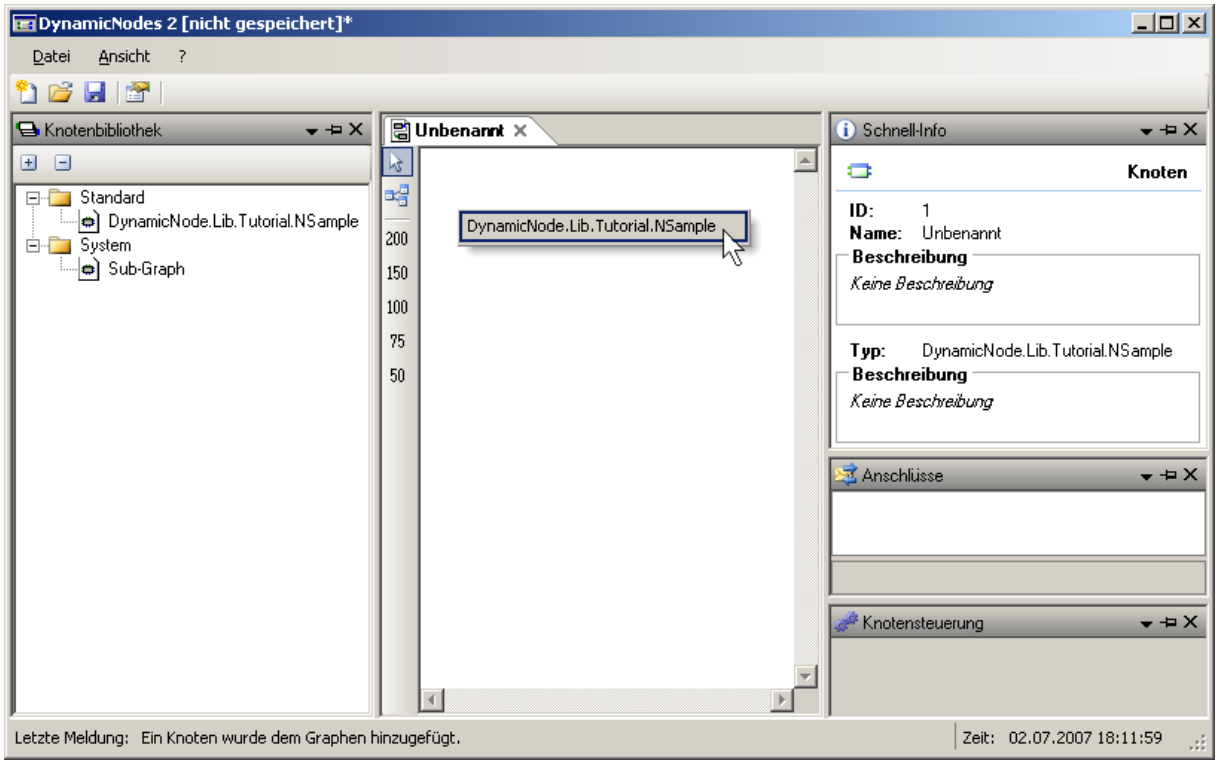
```
-lib "NLibTutorial.dll"-nosplash
```

Das Feld „Arbeitsverzeichnis“ sollte leer gelassen werden, weil nur so eine relative Pfadangabe der zu testenden Knotenbibliothek möglich ist.

3.3 Starten

Nun steht einem Starten von DynamicNodes aus der Entwicklungsumgebung heraus nichts mehr im Wege. Ein Klick auf den grünen Start-Pfeil in der Symbolleiste reicht aus.

Nach einem Start von DynamicNodes, ist der erste Knoten in der Ansicht „Knotenbibliothek“ unter der Gruppe „Standard“ zu finden. Von dort kann er per Drag&Drop auf die Arbeitsfläche gezogen und damit instanziiert werden.



3.4 Breakpoints und DN-Quellen

An dieser Stell noch zwei kurze Hinweise:

Wenn im Quelltext des Knotens Break-Points gesetzt werden, zeigt Microsoft Visual C# Express nun im Stack nicht nur die Aufrufe im Quellcode der Knotenbibliothek an, sondern zeigt die Aufrufe auch im Quellcode von DynamicNodes.

Wenn eine Ausnahme im Quellcodebereich von DynamicNodes auftritt, fragt die Entwicklungsumgebung nach den Quellen der entsprechenden Assembly. Dabei ist für ein Assembly das jeweilige Projektverzeichnis anzugeben. Falls der Quellcode von DynamicNodes verfügbar ist, kann die Entwicklungsumgebung so den genauen Ort der Ausnahme im Quelltext von DynamicNodes anzeigen.

4 Ein- und Ausgänge

Nun ist Knoten ohne Ein- und Ausgängen in den meisten Fällen recht nutzlos, also fügen wir dem Beispielknoten zwei Eingänge und einen Ausgang hinzu. Für den Anfang soll der Knoten eine Multiplikation von Fließkommazahlen ermöglichen.

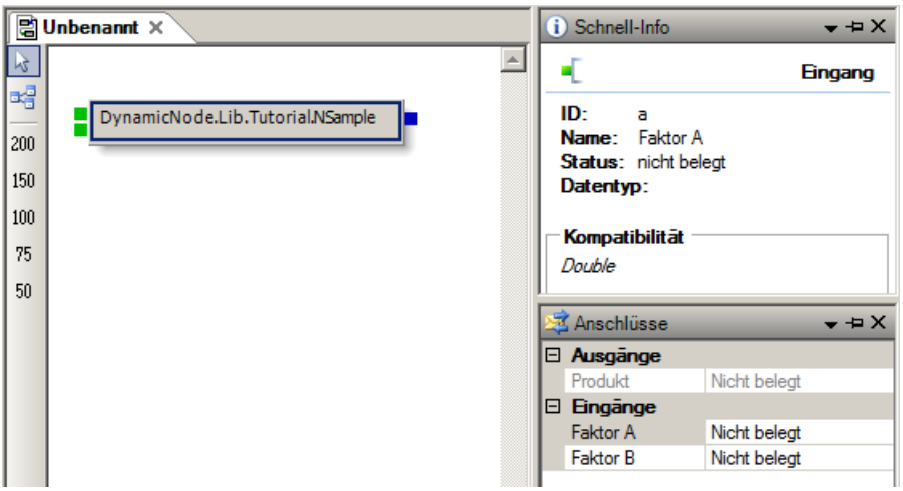
Durch einen Mechanismus zur automatischen Generierung von Ein- und Ausgängen mit Hilfe von Attributen, ist das eine Sache von wenigen Zeilen.

```
public class NSample : Node
{
    [InPortInfo("a", "Faktor A")]
    public double factor_a = 0.0;

    [InPortInfo("b", "Faktor B")]
    public double factor_b = 0.0;

    [OutPortInfo("p", "Produkt", typeof(double))]
    public OutPortHandler product;

    ...
}
```

Mit dem Attribut `InPortInfoAttribute` kann ein öffentliches Feld als Eingang markiert werden. Es erwartet eine ID und eine Beschriftung. Eingang, der mit diesem Attribut erzeugt wird besitzt keine Queue, und hat damit die Kapazität von 1.

Mit dem Attribut `OutPortInfoAttribute` kann ein Feld vom Type `OutPortHandler` als Ausgang markiert werden. Bei der Instanziierung, wird dem Feld im Standardkonstruktor von `Node` die Referenz auf eine Methode zugewiesen, welche den echten Ausgang aktualisiert. Das Feld hat also für den Knotenentwickler nie den Wert `null`.

Nach dem Starten und Instanziieren des Knotens sind die Anschlüsse sowohl auf der Arbeitsfläche, als auch in der Ansicht „Anschlüsse“ zu sehen.



5 Arbeit

Nun ist es an der Zeit, dem Knoten etwas zu tun zu geben. Mit einer Zeile, kann die A mit Leben gefüllt werden.

```
public override void Work()
{
    product(new Token(factor_a * factor_b));
}
```

Das Feld `product` vom Type `OutPortHandler` besitzt eine Referenz auf eine Methode, welche den Ausgang aktualisiert. Also lässt sich die in `product` referenzierte Methode mit der obigen Syntax aufrufen. `OutPortHandler` erwartet als einzigen Parameter die Struktur `Token`. Ein möglicher Konstruktor von `Token` wiederum, erwartet als Parameter ein `object`. Durch automatisches In-Boxing, wird das Produkt `factor_a * factor_b` vom Type `double` nach `object` konvertiert und als Markenwert in der Markenstruktur an den Ausgang weitergegeben. Der Markenstatus wird bei der Verwendung dieses Konstruktors automatisch auf `Valid` gesetzt.

Beim nächsten Neustart lassen sich in der Ansicht „Anschlüsse“ Eingabewerte per Tastatur eingeben. Daraufhin wird der Knoten aktiviert und der Ergebnis-Wert mit Hilfe der Arbeitsmethode aktualisiert.

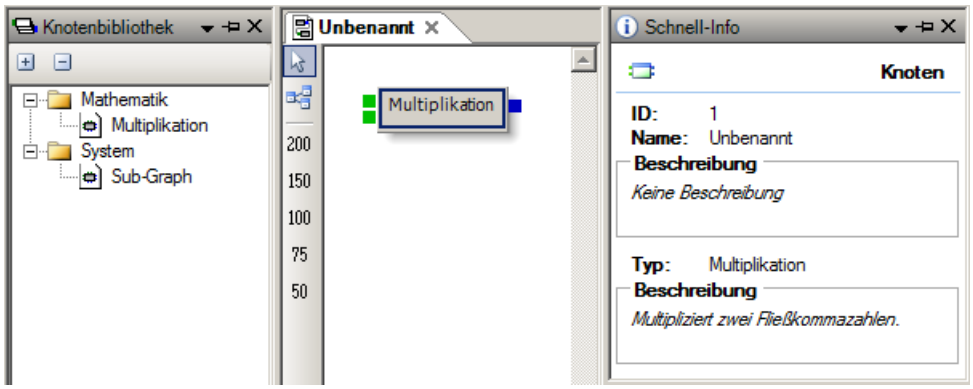
Fertig! – Na, ja es gibt natürlich noch mehr zu lernen. ☺

6 Metadaten

Der lange eindeutige Knotenname mit dem Namensraum ist recht lang und unschön. Um dem Benutzer an dieser Stelle mehr zu bieten, gibt es das Attribut `NodeInfoAttribute`. Damit können ein Name, eine thematische Gruppe und eine Kurzbeschreibung für den Knoten angegeben werden.

```
[NodeInfo("Multiplikation", "Mathematik",
    "Multipliziert zwei Fließkommazahlen.")]
public class NSample : Node
{
    ...
```

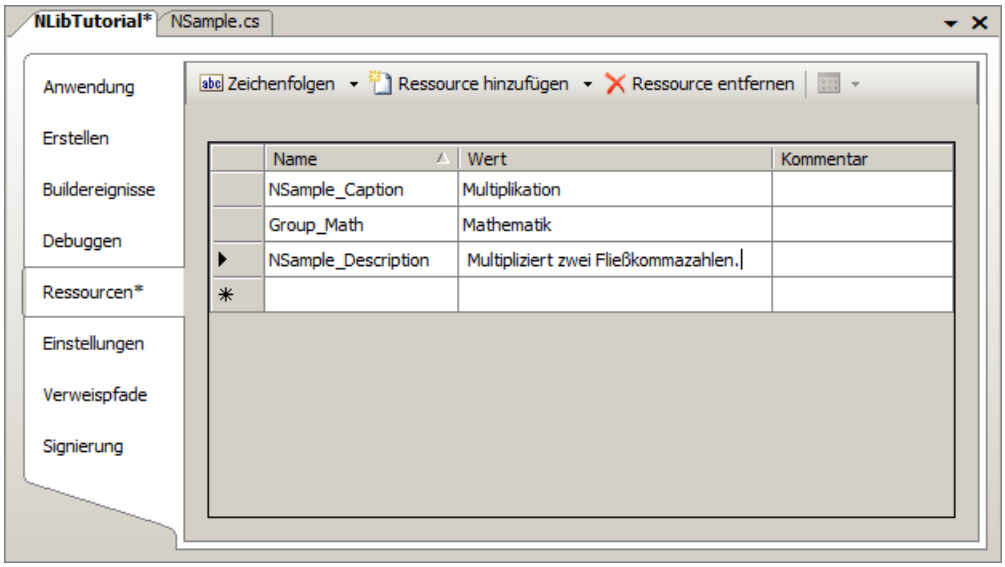
Damit sieht das Ganze schon wesentlich schöner aus:



Falls die Metadaten mit einer Ressourcen-Datei lokalisiert werden sollen, funktioniert diese Methode nicht, da in dem Konstruktor des Attributes nur Konstanten zugelassen sind. Aber auch dafür gibt es eine Lösung: Ein weiteres Attribut, welches die Angabe einer statischen Methode ermöglicht, welche dann wiederum die Metadaten liefert.

Eine Ressourcen-Datei lässt sich leicht über die Eigenschaftsseite des Projektes in der Rubrik „Ressourcen“ anlegen:

In dem Abschnitt „Zeichenfolgen“ der Ressourcen-Datei lassen sich nun die Werte eintragen:



Nun müssen diese Werte nur noch als Metadaten für den Beispielknoten geladen werden. Dazu wird eine statische Methode benötigt, welche eine Instanz von `NodeInfoAttribute` mit den Werten aus der Ressourcen-Datei erstellt und ein Verweis auf diese statische Methode als Lieferant der Metadaten.

```
[StaticNodeInfo(typeof(NSample), "GetNodeInfo")]
public class NSample : Node
{
    public static NodeInfoAttribute GetNodeInfo()
    {
        return new NodeInfoAttribute(
            Resources.NSample_Caption,
            Resources.Group_Math,
            Resources.NSample_Description);
    }
    ...
```

Das Attribut `StaticNodeInfoAttribute` erwartet einen Type und einen Methodennamen. Die angegebene Methode muss öffentlich sein, den Rückgabe-Type `NodeInfoAttribute` besitzen und eine leere Parameterliste erwarten.

Nach dem Starten hat sich scheinbar nichts geändert, aber wenn nun die Ressourcen-Datei lokalisiert wird, können die Metadaten in mehreren Sprachen angezeigt werden. Hilfe für Lokalisierung findet sich in der MSDN.