# Fachhochschule Dortmund

## Studiengang Elektrotechnik

## Studienrichtung Elektrische Energie- und Umwelttechnik


# Ingenieurarbeit

zum Thema

**Grundlage der .NET-Technologie**

und

**beispielhafte Anwendung für Client/Server-Kommunikation logischer Knoten nach IEC 61850 in verteilten Stationsleitsystemen**

von


**Jan Henning Arph**


**Dortmund, August 2002**

# 1. Einleitung

## 1.1 Verteilte Anwendungen

Neben dem rein objektorientierten Entwicklungskonzept, setzte sich Anfang der 90er Jahre das Konzept der Softwarekomponenten zunehmend durch. Die Idee ist, dass ein Programm nicht aus einer einzelnen zusammenhängenden Codeeinheit besteht, sondern aus vielen kleinen, nach Möglichkeit in sich logisch abgeschlossenen Softwarefragmenten zusammengesetzt wird, vergleichbar einem Baukastensystem. Die Idee der Komponentenarchitektur setzte sich im wesentlichen auf Grund der flexibleren Implementierungsmöglichkeiten durch. Auch die Wartbarkeit der aus kleinen Komponenten zusammengesetzten Programme wurde durch diese Infrastruktur enorm gesteigert und trug zu dem Erfolg der Komponentenmodelle bei. Im Laufe der Zeit und der zunehmenden Verfügbarkeit von Netzwerkarchitekturen wurde das Konzept der Komponentenmodelle dahingehend erweitert, dass sich einzelne Komponenten nicht mehr ausschließlich auf lokalen Systemen beschränken mussten, sondern eine Verteilung der Softwareeinheiten auf verschiedene Maschinen möglich wurde.
Damit solch ein Modell, sei es für lokale als auch für verteilte Komponenten ausgelegt, jedoch funktioniert, müssen die einzelnen Komponenten eine einheitliche Sprache sprechen, damit sie sich untereinander unmissverständlich austauschen können. Zum Zweck der Umsetzung und Implementierung von verteilten Anwendungen stehen heute verschiedene Binärstandards, wir etwa DCOM oder CORBA zur Verfügung, die für eine Sprachintegrität der Komponenten sorgen.
Das nachfolgende Schema, zeigt die clientseitige Anforderung zur Erzeugung eines fernen Objekts innerhalb einer DCOM Architektur.

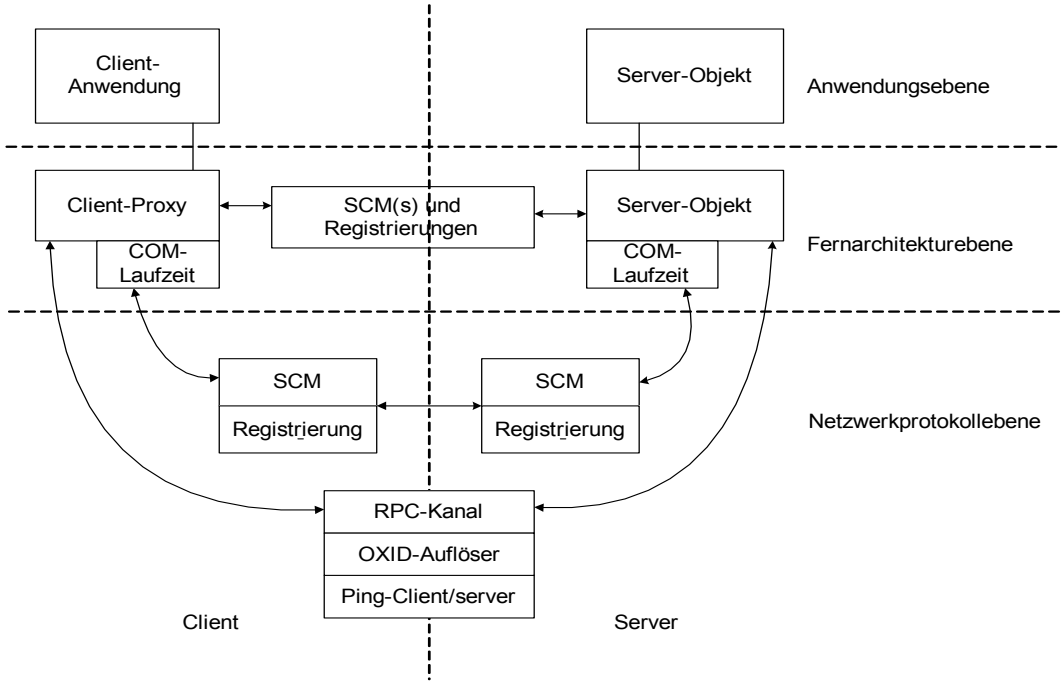



Abb. 1: Objekterzeugung innerhalb einer DCOM-Architektur

Um anschließend transparent auf das ferne Objekt zugreifen zu können, wird der clientseitigen Anwendung ein so genanntes Proxy (Stellvertreter)-Objekt zur Verfügung gestellt. Um ein fernes Objekt erzeugen zu können, wird zunächst dem lokalen Service Control Manager (SCM) die nötige Information übermittelt. Der lokale SCM wendet sich mit den ihm zuvor übergebenen Informationen betreffend das ferne System an selbiges und kontaktiert den dortigen SCM. Der ferne SCM aktiviert das ferne Objekt und liefert eine Klassenobjektreferenz des entsprechend fernen Objektes an das lokale System

zurück. Diese Klassenobjektreferenz wird an den lokalen Proxy übergeben. Das lokale Objekt kann nun mittels dieser Objektreferenz über das Proxyobjekt Instanzen des fernen Objektes erzeugen. Mittels des Weges über das Proxy kann der Client nun direkt mit dem fernen Objekt interagieren. Die Kommunikation zwischen beiden Objekten erfolgt über ein erweitertes DCE RCP Netzwerkprotokoll. Während der Kommunikation zweier verteilter Objektinstanzen regelt DCOM den Informationsaustausch über so genannte ORPC-Pakete (Object Remote Procedure Call). Soll eine Methode eines fernen Objektes aufgerufen werden schickt der Client, nachdem zuvor einige DCE RPC Verhandlungen und Sicherheitsmaßnahmen abgewickelt wurden, eine Anfrage an das ferne Objekt um selbiges, falls dies noch nicht der Fall ist, zu aktivieren. Das ferne System antwortet mit einem so genannten ObjRef-Paket, welches die spezielle Instanz des fernen Objekts lokalisiert. Mittels dieses ObjRef-Paketes, erhält das clientseitige Objekt eine eindeutige Objektkennung (OID), sowie eine Art ID für Threadkontextinformationen (OXID) und einen Zeiger auf die Schnittstelle eines speziellen sich in Ausführung befindenden Objekt (IPID). Diese Informationen benötigt der Client für weitere Methodenaufrufe. Um nun eine Objektmethode diese fernen Objekte aufzurufen, muss der Client eine Nachricht in Form eines ORPCTHIS-Paketes an das entsprechende ferne Objekt senden. Empfängt das ferne System dieses Paket führt es die entsprechende Methode aus und sendet das Ergebnis mittels eines ORPCTHAT-Paketes an den entsprechenden Aufrufer zurück. DCOM stellt von sich aus sicher, dass bei der Kommunikation zwischen verteilten Objekten alle beteiligten Komponenten verfügbar und aktiv sind. Dazu wird von allen beteiligten Clientobjekten verlangt, in einem gewissen zeitlichen Intervall über Pingnachrichten im ständigen Kontakt zum jeweiligen fernen Objekt zu stehen. Bleibt eine entsprechende Pingnachricht aus, sorgt DCOM für die notwendige Zerstörung des Objektes und die entsprechende Speicherbereinigung. Durch diesen ständigen Informationsfluss zwischen den beteiligten Komponenten, auch wenn diese nur temporär miteinander kommunizieren, wird verhindert, dass auf Grund endlicher Netzwerkressourcen über eine gewisse Anzahl an Clients hinaus gegangen wird. Was die Übertragungsgeschwindigkeit von DCOM anbelangt, so ist die Aktivierung eines Objektes der inperformanteste Vorgang der Kommunikation. Zunächst erhält der Client auf dem Weg über seinen lokalen SCM von dem fernen SCM eine Klassenobjektreferenz des fernen Objekts. Mit Hilfe dieser Referenz und seines Proxies erzeugt der Client eine oder mehrer Instanzen des fernen Objektes. Hierfür wird der SCM dann nicht mehr benötigt und der Client kann direkt mit dem fernen Objekt interagieren. Auf Grund seiner Speicherbereinigungsarchitektur und dem verbindungsbedingten Overhead gilt DCOM zwar als sehr sichere jedoch schlecht skalierbare Fernarchitektur.
Der große Vorteil solcher Infrastrukturen ist es, dass Softwarekomponenten als „Blackbox" eingesetzt werden können, ohne dass Informationen darüber wie die Implementierung der Komponente im Detail aussieht, geschweige denn mit welchen Programmiersprachen sie entwickelt wurde, vorliegen müssen.
Diese Modelle haben sich auch in der Leit- und Automatisierungstechnik weitgehend durchgesetzt und Standards wie etwa OPC hervorgebracht.

## 1.2 Internet und .NET

Mit der zunehmenden Marktdurchdringung des Internets wurde der Wunsch nach einer Portierung des Komponentenmodellkonzeptes für verteilte Anwendungen auf die Infrastruktur des weltumspannenden Internets immer größer. Größtes Problem bei einem solchen Vorhaben ist es, dass die bestehenden Komponenetenmodelle auf Grund ihrer firmenspezifischen, proprietären Binärstandards eine Homogenität des Gesamtsystems zwingend voraussetzen. Da das Internet jedoch alles andere als homogen ist, können die vorhandenen Systeme nicht eins zu eins auf die Struktur des Internet übertragen werden. Für die Realisierung verteilter Komponentenmodelle im Internet werden offenen Kommunikations- und Typsysteme benötigt, die nach Möglichkeit von übergeordneten und unabhängigen Institutionen wie etwa dem W3C-Konsortium standardisiert werden. Diesem Wunsch rechnungtragend wurde unter der Federführung von führenden Firmen

der Informationstechnologiebranche das Simple Object Access Protocol (SOAP) entwickelt. Dieses Protokoll setzt auf bestehende allgemein anerkannte und offene Normen wie http und XML auf. Mittels dieses Protokolls ist es nun möglich, einzelne Komponenten über die Infrastruktur des Internets miteinander kommunizieren zu lassen. Dabei spielt es keine Rolle mehr auf welchem System die einzelne Komponente läuft, entscheidend ist nur noch, dass das jeweilige System das Protokoll SOAP unterstützt. Das SOAP-Protokoll hat die Aufgabe den Aufbau der Nachrichten, die in XML codiert sind, zu definieren und die Aufrufreihenfolge zwischen den Komponenten festzulegen. Mit dieser Technologie können Softwarekomponenten Methoden für spezielle Aufgaben dem Internet zugänglich machen bzw. Methoden anderer Komponenten nutzen. Solch eine Komponente, die die eigenen Methoden dem Internet offen legt, sprich einen Service anbietet, wird im neuen .NET Konzept WebService genannt. Hierfür, sowie für andere Implementierungsaufgaben, wie etwa die klassische Windowsprogrammierung, stellt das .NET-Framework, beschrieben in Abschnitt 2, ein umfassendes Entwicklungskonzeptbereit. Als zentrales Netzwerkprotokoll für verteilte Anwendungsarchitekturen setzt .NET auf den offenen Standard SOAP. Hier sei noch ein mal herausgestellt, dass es sich bei SOAP lediglich um ein Netzwerkprotokoll und nicht um eine komplette Fernarchitektur, wie etwa DCOM handelt. Das nachfolgende Diagramm zeigt den prinzipiellen Aufbau eines SOAP-Paketes.

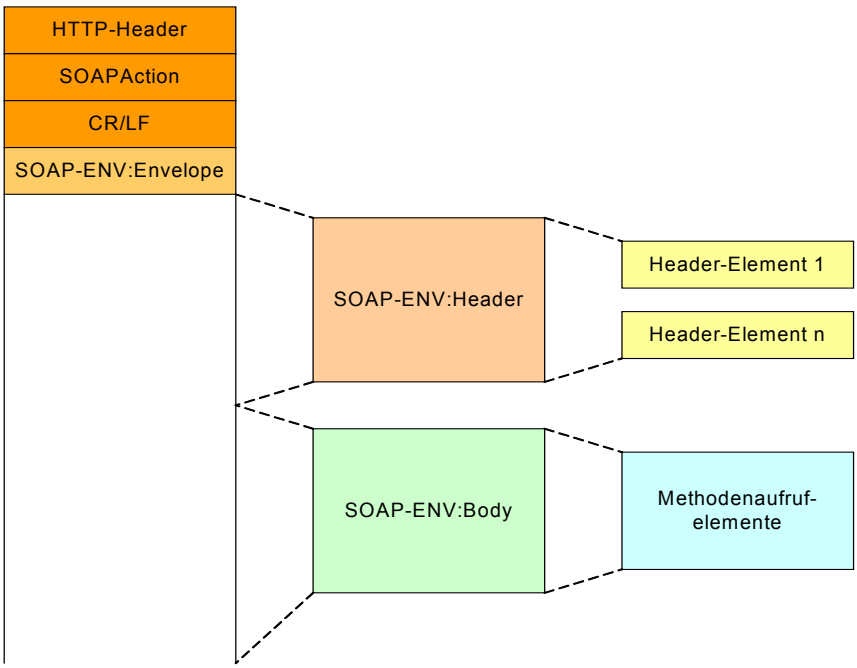



Abb. 2: Kommunikation mittels SOAP-Protokoll

In diesem Schema wurde http als Transportprotokoll vorausgesetzt. Die eigentliche Nutzlast des Paketes ist in einen SOAP-Umschlag (Envelope) eingeschlossen, welcher wiederum selbst als Teil der http-Nutzlast angesehen werden kann. Der SOAP-Umschlag an sich besteht aus dem SOAP-Header und dem darauf folgendem SOAP-Body. In dem optionalen SOAP-Header, der direkt nach dem einleitenden SOAP-Umschlag Tag steht, können Metainformationen über den entsprechenden Methodenaufruf enthalten sein. Innerhalb des SOAP-Body Tags sind die eigentlichen Methodenaufrufparamenter in serialisierter Form enthalten. Hierbei werden die Daten durch eine Kombination aus XML-Elementen und –Attributen serialisiert.

Nachfolgend ist ein SOAP-Protokollausschnitt einer Anfrage sowie die darauf folgende Antwort eines .NET-WebServices dargestellt.

Anfrage:

```
POST /WebService1/Service1.asmx HTTP/1.1
Host: localhost
Content-Type: text/xml; charset=utf-8
Content-Length: length
SOAPAction: "http://tempuri.org/HelloWorld"

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xmlns:xsd="http://www.w3.org/2001/XMLSchema"
               xmlns:soap="http://schemas.xmlsoap.org/
              soap/envelope/">



  <soap:Body>
    <HelloWorld xmlns="http://tempuri.org/" />
  </soap:Body>
</soap:Envelope>
```

Antwort:

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: length

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xmlns:xsd="http://www.w3.org/2001/XMLSchema"
               xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <HelloWorldResponse xmlns="http://tempuri.org/">
      <HelloWorldResult>string</HelloWorldResult>
    </HelloWorldResponse>
  </soap:Body>
</soap:Envelope>
```

Innerhalb des `<HelloWorldResult>` Tags wird der Rückgabeparameter des Services übermittelt.

## 1.3 IEC 61850-7 und das Substation Object Model (SOM)

Durch den Einsatz der digitalen Stationsleittechnik ab den 1980er Jahren wurden analoge Schnittstellen der Geräte der konventionellen Stationsleittechnik durch serielle Schnittstellen digitaler Gräte ersetzt. Die einfach zu beschreibenden Schnittstellen der konventionellen Stationsleittechnik bestehen im Wesentlichen aus normierten Strom- und Spannungsschnittstellen. Dies bedeutet, dass, um einzelne Geräte mit einander zu verbinden, deren Kontakte durch über Koppelperipherie und Rangierverteiler mit einander verdrahtet werden. Sind die Schnittstellen der konventionellen Stationsleittechnik für den Betreiber oder den Wartungsbeauftragten einer Anlage noch transparent, so dass er ohne größere Aufwendungen Geräte verschiedener Hersteller durch einfache Verdrahtung miteinander verbinden kann, so geht durch den Einsatz von seriellen Schnittstellen die einfache Kombination von Geräten unterschiedlicher Hersteller verloren. Damit jedoch verschiedene digitale Geräte Daten untereinander austauschen können wurde mit der Normenreihe IEC 60870-5 eine Protokollsyntax etabliert, die die Abbildung von typischen Informationsdatentypen der Stationsleittechnik auf die Protokollebene der physikalischen Schnittstelle von einzelnen digitalen Geräten ermöglicht. Dabei lassen die Syntaxbeschreibungen Spielraum für Hersteller-/Betreiberspezifische Protokollprofile. Dies führte dazu, dass Geräte verschiedener Hersteller meist mit unterschiedlichen Protokollprofilen betrieben werden und ein Zusammenwirken unterschiedlicher Geräte nur durch eine aufwendige Profilanpassung bzw. Konvertierung erreicht werden kann. Zudem ist in der Normenreihe IEC 60870-5 kein erweiterungsfähiges und allgemeingültiges Regelwerk für die inhaltliche Bedeutung der Information, die zwischen Geräten in Form eines Bitstroms ausgetauscht werden, festgelegt.
Damit bleibt, selbst durch die Möglichkeit von Profilfestlegungen, die inhaltliche Auszeichnung und damit die Grundlage der Erstellungen eines Datenmodels, wie beispielsweise die Abbildung einer Leistungsschalter-Störmeldung auf eine bestimmte Bitkombination, projekt- oder herstellerspezifisch.
Unter anderem wurde zu dem Zweck, ein genormtes Rahmenwerk für die inhaltliche Bewertung einer Nachricht zwischen verschiedenen Geräten der Stationsleittechnik zu definieren, die Normenreihe IEC 61850 entwickelt. Grundlage der Normenreihe IEC 61850 ist es, ein grundlegendes Daten- und Dienstmodell für ein Kommunikationssystem, das eine Interoperabilität zwischen Funktionen und Geräten verschiedener Hersteller gewährleistet, bereitzustellen. Um dies zu erreichen beschreibt die Normenreihe IEC 61850 in ihrem Teil 7 eine Protkollsyntax unabhängige Darstellungsform für die Datenmodellierung. Dies bedeutet jedoch, damit ein Datenmodell, das mit den Regeln von IEC 61850-7 konform geht, zur realen Anwendung gelangt, auf einen bestimmten Protokollstack gemappt werden muss. Somit können durchaus zukünftige Anwendungen, die auf der Normenreihe IEC 61850 basiere, also ein einheitliches Datenmodell darstellen, sich in der Abbildung auf einen Protokollstack voneinander unterscheiden.

Mit der Schnittstellenspezifikation SOM (Substation Object Model) kann ein auf IEC 61850-7 basierendes Datenmodel mit dynamischen Kardinalitäten mittels der .NET-Remotingtechnologie auf bestehende Protokolle der allgemeinen Informationstechnologie, wie etwa TCP, HTTP oder SOAP gemappt werden. Da das SOM eine möglichst große Skalierbarkeit innerhalb einer verteilten Anwendungsarchitektur ermöglichen soll, ist das Objektmodell des SOM's aus abstrakten Klassen, sprich Schnittstellen implementiert. Schnittstellen ermöglichen es, dass verschiedene Komponenten, die in einer verteilten Anwendung miteinander kommunizieren sollen, die konkreten Klassen der jeweilig anderen Komponente zur Zeit der Kompilierung noch nicht kennen müssen, lediglich muss zu diesem Zeitpunkt die Schnittstelle bekannt sein, über die die Komponenten zur Laufzeit miteinander kommunizieren sollen.

Die Version des eingesetzten SOM's beinhaltet die gesamte Schnittstellendefinition für die Implementierung der Containerklassen:

- Server,
- LogicalDevices,
- LogicalNodes,
- DataObjects, sowie
- DataAttributes.

Des Weiteren die Schnittstellendefinitionen der in den aufgelisteten Containerklassen zu verwaltenden Objekten der Klassen:

- LogicalDevice,
- LogicalNode,
- DataObject und
- DataAttribut.

Auch liegen die Schnittstellendefinitionen für einige ausgewählte DataObjects und DataAttributs vor.

DataObject:

- Pos

DataAttribut:

- stVal
- ctlVal

Die Implementierung der Schnittstellendefinition des SOM's wird im Abschnitt 3 dieser Arbeit beschrieben.
Um die Realisierung eines Datenmodells innerhalb einer verteilten Anwendung, basierend auf der SOM-Schnittstelle und der .NET-Remotingtechnologie zu zeigen, dient der Abschnitt 4 und 5 der vorliegenden Arbeit.

# 2. Die .NET-Technologie von Microsoft

## 2.1 Die .NET-Laufzeitumgebung (Common Language Runtime CLR)

Eines der fundamentalen Konzepte der .NET Plattform verwirklicht die universelle Laufzeitumgebung (Common Language Runtime CLR), die für alle .NET Programme zuständig ist. Mit der Einführung der gemeinsamen Laufzeitumgebung und damit der Einführung einer neuen Technologie, drängt sich die Frage auf, warum ein solcher Schritt notwendig ist und welche Vorteile er für die zukünftige Softwareentwicklung bietet. Die der Common Language Runtime zugrunde liegende Idee ist keine neue. Anfang der 90er Jahre führte Microsoft das Component Object Model (COM) ein. Dies sollte dazu dienen, die Kommunikation und die Integration zwischen einzelnen Softwarekomponenten zu vereinheitlichen. Diese Vereinheitlichung der Kommunikation zwischen Komponenten erfolgt im binären Format, damit eine Programmiersprachenunabhängigkeit erreicht wird. Zu diesem Zweck muss COM ein eigenes Typsystem bereitstellen. Damit jedoch jetzt eine Programmiersprache interoperabel mit COM ist, muss sie neben dem eigenen Typsystem ein zusätzliches Typsystem, nämlich das von COM implementieren. Durch diese Architektur, sprich der Verwendung einer Zwischenschicht, wird der Implementierungsaufwand immens gesteigert. Im Wesentlichen treten für die jeweiligen Programmiersprachen somit die drei folgenden Probleme auf, um Softwarekomponenten interoperabel mit COM zu gestalten

- sie benötigt einen Layer, der das Typsystem von COM implementiert,
- mittels des vorhandenen Layers muss in der Entwicklungsphase die Konvertierung zwischen dem spracheigenen und dem COM Typsystem realisiert werden,
- neben der Konvertierung der Typen muss ein zusätzlicher Implementierungsaufwand für die entsprechenden Aufrufkonventionen vom COM betreiben werden.

Mit der Zeit und der zunehmenden Verfügbarkeit von Netzwerkstrukturen wurde COM um die Möglichkeit der Fernaufrufe (Microsoft Transaction Server MTS, Distributed COM DCOM) erweitert. Mit der Einführung des Betriebssystems Windows 2000 wurden diese Dienste unter dem Sammelbegriff COM+ zusammengefasst. Jedoch fordert auch weiterhin diese Architektur einen Typlayer für die jeweilige Programmiersprache, welcher das Typsystem von COM implementiert. So muss weiterhin jede einzelne Sprache, die COM unterstützen will an selbiges angepasst werden. Mit der Einführung der Common Language Runtime (CLR) wird nun auch das grundlegende Typsystem integriert. Somit stellt die CLR ein einheitliches Integrationsmodel zur Softwareentwicklung zur Verfügung und macht diese einfacher und konsistenter.

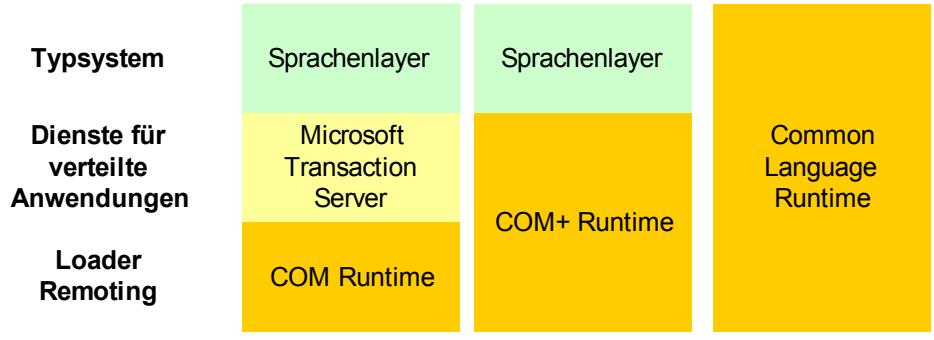



Abb. 3: Vergleich von COM, COM+ und CLR

Die Common Language Runtime ermöglicht eine einheitliche Ausführung aller .NET Applikationen unabhängig davon mit welchen Entwicklungssprachen diese erstellt wurden. So ist es möglich ohne größeren Aufwand, d.h. die Verwendung einer kommunikationsvereinheitlichenden Zwischenschicht wie etwa COM, in der Entwicklungsphase einer Anwendung die für die jeweilige Problemstellung geeignete Sprache einzusetzen. Dieses wird dadurch erreicht, dass die Kompilierung eines .NET Programms nicht direkt zu Maschinencode, sondern zu einem so genannten verwaltetem Zwischencode führt. Ähnlich wie bei dem Konzept der virtuellen Maschine bei Java wird dieser Zwischencode erst zur Laufzeit vom so genannten Just-In-Time-Compiler in Maschinencode übersetzt. Grundsätzlich erzeugt jeder .NET Compiler, sei es der VB.NET-C# oder einem anderer .NET-Compiler, verwalteten Zwischencode (managed Code) in IL (intermediate Language) Syntax.

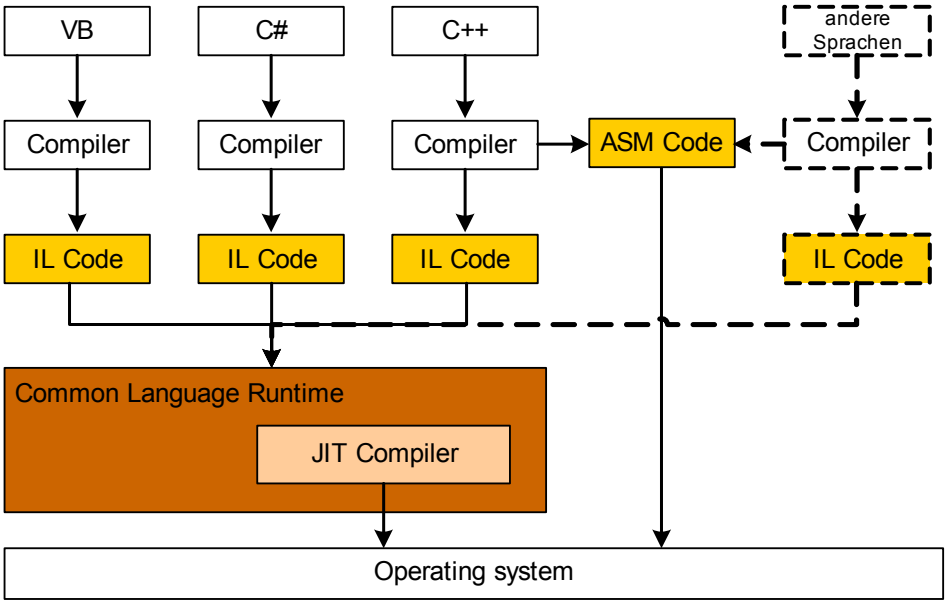



Abb. 4: Die Rolle der Common Language Runtime im Gesamtsystem

Gleichzeitig wird durch diese Architektur eine Systemunabhängigkeit erreicht, die nur noch die Existenz eines für den jeweiligen Prozessor ausgelegten Just-In-Time-Compilers fordert.
Neben diesem grundlegenden Konzept der CLR ermöglicht diese eine weiter Vielzahl von Optimierungen während der Softwareentwicklung.
Die CLR macht eine einfachere, schnellere und sicherere Softwareentwicklung möglich, in dem sie vor allem den Entwickler von unangenehmen Routineimplementierungen befreit. So unterstützt die CLR einen großen Satz an Systemfunktionalität, wie etwa die automatische Speicherverwaltung (Garbage Collection) und stellt dem Entwickler diese in einer standardisierten Form zur Verfügung oder verbirgt diese sogar vollständig vor ihm. Damit bleibt in der Entwicklungsphase mehr Zeit für die eigentlich zu implementierende Thematik.

Im nachfolgenden Diagramm sind die wesentlichen Komponenten der CLR dargestellt.

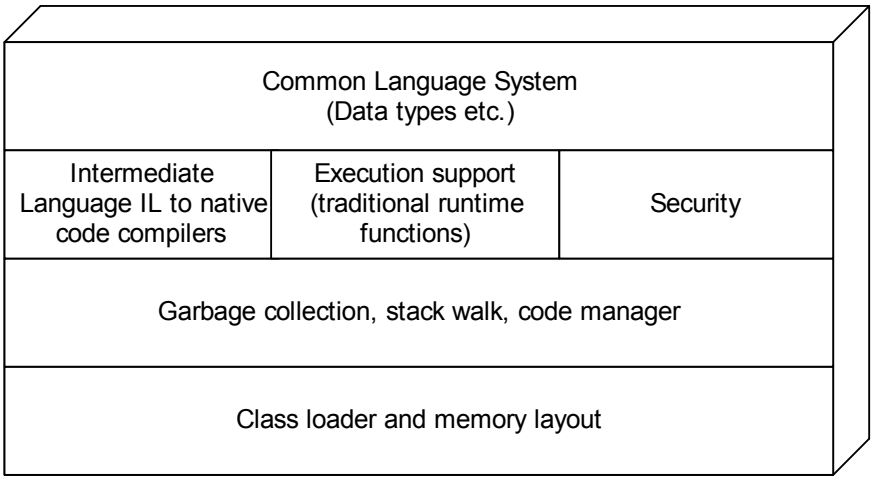



Abb. 5: Die wesentlichen Komponenten der Common Language Runtime

## 2.2 Das gemeinsame Typsystem (Common Type System CTS)

Neben dem Ziel des .NET Frameworks, mittels eines einheitlichen Zwischencodes eine gewisse Plattformunabhängigkeit zu erreichen, ist es ein weiteres Ziel, Sprachneutralität zu bieten. Die Probleme, die bei der Implementierung einer Softwareapplikation, bestehend aus verschiedenen Einzelkomponenten welche wiederum mit verschiedenen Programmiersprachen, wie etwa C++ oder Visual Basic erstellt wurden, gründen im wesentlichen darauf, eine reibungslose Kommunikation zwischen diesen Komponenten zu realisieren. Sicherlich bieten heutige Technologien für dieses Problem der Sprachkonsistenz geeignete Lösungen an. Hierfür stehen verschiedene horizontale Zwischenschicht-Architekturen, wie etwa COM/DCOM oder CORBA zur Verfügung. Ein entscheidender Grund für die Kommunikationsprobleme zwischen unterschiedlichen Komponenten besteht darin, dass die jeweils verwendeten Programmiersprachen die verschiedensten Darstellungsformen für ihre jeweiligen Datentypen verwenden. So wird z.B. der Datentyp Integer in der Sprache C++ anders dargestellt, als in der Sprache Visual Basic. Wollen zwei Komponenten untereinander Informationen in Form einer Integerzahl austauschen, so müssen sie sich auf eine einheitliche Darstellungsform des Typs einigen. Hier liefert COM eine gemeinsame Grundlage, um einen Datentyp definiert darzustellen. Wird via COM (also in Interface Definition Language IDL definiert) ein Parameter als Integer (int) übergeben, so soll es sich um einen vorzeichenbehafteten 32 Bit-Ganzzahlenwert handeln.

| | C++ | VB |
|---|---|---|
| Name | int | Integer |
| Größe in Bit | 32 | 16 |

Ein C++ Programm verwendet diesen Wert direkt als *int*, ein Visual Basic Programm muss den Wert schon als *Long* interpretieren, weil ein *Integer* in VB nur 16 bit groß ist. Handelt es sich bei diesem Beispiel noch um ein überschaubares Szenario, wird es bei der einheitlichen Darstellung von Datumswerten schon wesentlich komplexer. Die Implementierung einer Komponente, die einem solchen Kommunikationsstandard, wie etwa COM genügt, ist jedoch ohne geeignete Entwicklungstools nur mit größtem Aufwand zu realisieren. Um den Implementierungsaufwand zu minimieren, liegt es nahe ein einheitliches Typsystem in jeder einzelnen Programmiersprache zu verwenden. Zu diesem Zweck stellt das .NET Framework mit der Laufzeitumgebung das Konzept des Common Type System (CTS) zur Verfügung. Das CTS bietet eine Fülle von Datentypen, die aus jeder .NET Programmiersprache heraus verwendet werden können. Das bedeutet, dass der jeweilige Sprachcompiler kein eigenes Typsystem mehr mit sich bringen muss, sondern kann sich des Typsystems der Common Language Runtime bedienen. Das Typsystem wandert sozusagen vom Compiler zur Laufzeitumgebung und muss somit nicht mehr Bestandteil einer Programmiersprache sein. Auf Grund dessen, dass alle Sprachen auf dem gleichen Typsystem aufbauen, bedeutet dies, dass allgemein Typen eindeutiger werden, da diese nicht mehr unterschiedlich repräsentiert werden. Konvertierungen und Anpassungen an eine Zwischenschicht, wie etwa COM, sind somit bei der Kommunikation von Komponenten unterschiedlicher Sprachen nicht mehr notwendig, da sie von Hause aus interoperabel miteinander sind.

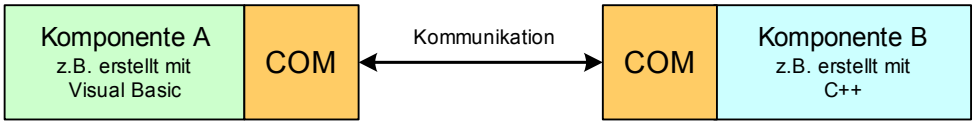



Abb. 6: Die Kommunikation mittels COM als horizontale Middelware

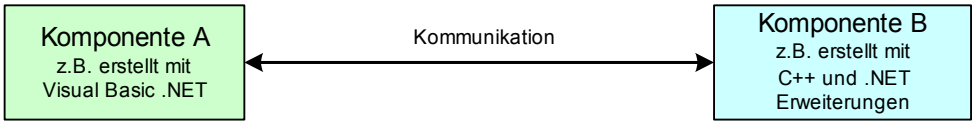



Abb. 7: Die direkte Kommunikation mittels eines von beiden gemeinsam benutzen Typsystems

Wie die Abb. 6 und Abb. 7 zeigen, wird unter COM für jede Sprache ein eigener Layer benötigt, der die zu implementierende Komponente interoperabel mit COM macht. Mit der Verwendung eines einheitlichen Typsystems, wie es das .NET Framework zur Verfügung stellt, ist dieser Schritt nicht mehr notwendig. Komponenten können direkt miteinander interagieren. Somit wird der Entwickler von zeitraubenden Routineimplementationen, wie Konvertierungsmethoden und die Einhaltung von Aufrufkonventionen, entlastet. Zudem ist das Verhalten der Programmiersprache aus Sicht des Anwendungsentwicklers transparent gestaltet, da er diese wie gewohnt verwenden kann. Das Mapping erfolgt im Hintergrund beim Übersetzen durch den jeweiligen Sprachcompiler, der entsprechenden IL Code erzeugt.

Die .NET Plattform erhebt den Anspruch, vollständig objektorientierte Techniken zu unterstützen. Dies zeigt sich auch bei der Betrachtung des Common Type System. Hier wird deutlich, dass das Common Type System keine einfachen Datentypen zur Verfügung stellt. Alle Typen sind Objekte und werden von einem grundsätzlichen Typ, dem **System.Object,** abgeleitet.

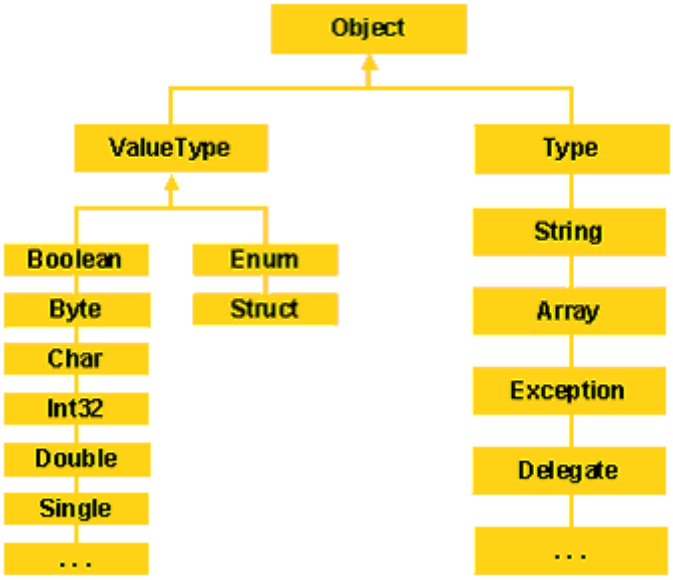
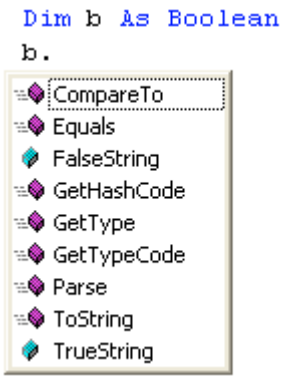



Abb. 8: Ausgewählte Typen im Namespace „System"

Da in der Regel Objekte jedoch auf dem Heap abgelegt werden und dies im Vergleich zu auf dem Stack abgelegten Typen ein Nachteil in der Performance mit sich bringt, werden die Datentypen des CTS in zwei Kategorien unterteilt.


ValueTypes zeichnen sich durch folgende Eigenschaften aus:

- sie werden auf dem Stack angelegt,
- sie enthalten Daten,
- sie können nicht dem Wert null annehmen,
- sie repräsentieren im Wesentlichen primitive Datentypen wie int, Aufzählungen, und Strukturen,
- sie besitzen auch Eigenschaften und Methoden
  z.B. der Typ Boolean:




Im Gegensatz dazu gilt für ReferenceTypes:

- sie werden auf dem Heap angelegt,
- sie enthalten Referenzen auf Objekte,
- sie können den Wert null annehmen,
- sie repräsentieren im Wesentlichen folgende Typen: Zeichenketten, Klassen und Felder.

Wie oben erwähnt werden alle ValueTypes auf dem Stack verwaltet. Wie aber kann jetzt ein ValueType eine Objektmethode aufrufen, wenn doch alle Typen von System.Object abgeleitet werden?
Sobald ein ValueType eine Objektmethode aufruft, legt die Runtime automatisch ein temporäres Objekt auf dem Heap an und kopiert den Wert des ValueTypes dort hinein. Dann erfolgt der Methodenaufruf. Nach der Ausführung des Methodenaufrufs wird das Objekt wieder entfernt und man arbeitet mit dem Value Type weiter.
Diese Technik wird mit den Begriffen Boxing und Unboxing bezeichnet. Mit Boxing wird das Konvertieren eines ReferenceType in einen ValueType bezeichnet, mit Unboxing der umgekehrte Vorgang. Im Hinblick auf die Performance ist diese Vorgehensweise ein guter Kompromiss. Es ist nur dann ein echtes Objekt auf dem Heap vorhanden, wenn es wirklich gebraucht wird. Für den Entwickler ist dies vollkommen transparent. Aus seiner Sicht sind alle Typen Objekte und die Runtime erledigt "den Rest" hinter den Kulissen.

## 2.3 Die .NET Klassenbibliothek

Um dem .Net Paradigma einer einfacheren, schnelleren und leichteren Programmentwicklung gerecht zu werden, stellt das .Net Framework eine sprachübergreifende gemeinsame Klassenbibliothek, das .NET Classframework zur Verfügung. Das .NET Classframework ist dabei nicht zu vergleichen mit herkömmlichen COM-Klassenbibliotheken. Das .NET Classframwork ist eine gänzlich objektorientierte, hierarchische und einheitliche Klassenbibliothek, die aus den unterschiedlichsten .NET Sprachen heraus verwenden werden kann. Das .NET Classframework stellt eine Vielzahl von Klassen, Schnittstellen und Strukturen bereit, die grob in folgende Kategorien unterteilt werden können:

- Datenzugriffskomponenten (ADO.NET),
- Komponenten für die Entwicklung von Webapplikationen (ASP.NET),
- Komponenten für die Windowsapplikation (WinForm),
- Komponenten für die Netzwerkkommunikation mittels unterschiedlicher Protokolle,
- Komponenten für den Zugriff auf Metainformationen von Assemblies,
- Komponenten zur Verwaltung und Manipulation von Ressourcen,
- Komponenten für die Threadentwicklung.

Mit der Verwendung des .NET Frameworks werden sprachspezifische Bibliotheken, wie etwa die Visual Basic Runtime (VBRun) oder die Microsoft Foundation Class (MFC) überflüssig. Da die gemeinsame universelle Klassenbibliothek sprachübergreifend eingesetzt werden kann. Wie die .NET Common Language Runtime eine multi-Sprach-Entwicklung durch die Bereitstellung einer gemeinsamen Laufzeitumgebung ermöglicht unterstützt das .NET Classframework dies indem es ein gemeinsames Sprach-API (Application Programming Interface) darstellt. Die Verwendung einer solchen gemeinsamen Klassenbibliothek erleichtert dem Entwickler die Arbeit, da er nun nicht mehr für jede einzusetzende Sprache die dazugehörige sprachspezifische Bibliothek und ihr Objektmodell beherrschen muss. Bekannt und vertraut muss nur noch mit einer einzigen Klassenbibliothek sein. Das .NET Classframework ermöglicht auch die sprachübergreifende Interoperabilität zwischen .NET Sprachen, indem es einen Standard von Klassen, Schnittstellen und Strukturen bereitstellt, die in jeder Programmiersprache, die die .NET Common Language Runtime unterstützt, eingesetzt werden können. Das .NET Classframework ersetzt die vielen existierenden COM Bibliotheken, indem es eine unzählige Anzahl an vergleichbaren verwalteten Klassen, Schnittstellen und Strukturen, innerhalb eines einfachen, hierarchischen Objektmodells zur Verfügung stellt. Damit es innerhalb dieses Objektmodells nicht zu Nameskonflikten kommt, organisiert das .NET Classframework diese Typen in so genannten Namespaces. Dabei ist ein Namespace eine einfache logische Gruppierung von aufgabenspezifisch zusammengehörigen Klassen, Schnittstellen und Strukturen. Des weiteren kann ein Namespace selber wiederum andere Namespaces enthalten. Damit wird es möglich, die Architektur der Klassenbibliothek hierarchisch zu gestalten. Innerhalb der .Net Classframeworkarchitektur stellt das ‚System' Namespace das Wurzelelement dar. Dieses enthält die grundlegenden Datentypen, wie etwa Object, Int32 oder String, sowie grundlegende Klassen, wie Console, Convert oder Math. Des weitern enthält das Wurzelelement „System" 25 weitere untergeordnete Namespaces, sowie etwa System.IO, welches für den Zugriff auf das Dateisystem Funktionalitäten logisch zusammenfasst, oder etwa den Namespace System.Data, der ein Vielzahl von Klassen für den Datenbankzugriff beinhaltet (ADO.NET).

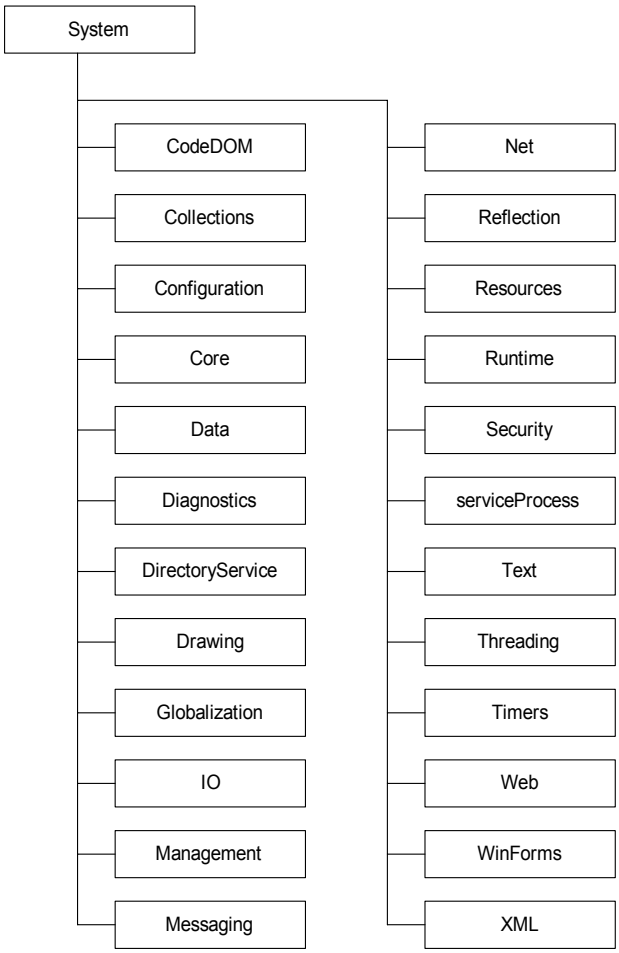



Abb. 9: Namespace ‚System'

Die .NET Klassenbibliothek stellt eine vollständig objekorientierte und erweiterbare Klassenbibliothek dar. Die Klassenbibliothek wird erweiterbar dadurch, dass sie vollständig auf der CLR aufsetzt, welche die Vorteile von modernen objektorientierten Konzepten, wie etwa Vererbung, Polymorphismus sowie Überladung von Funktionssignaturen, unterstützt.
Die Common Language Runtime und die Regeln des Common Type Systems erfordern es, dass alle verwalteten Klassen direkt oder indirekt von der Basisklasse System.Object abgeleitet werden. Diese Forderung führt dazu, dass jede Klasse ein gewisses Minimum von Methoden und Funktionen unterstützt, genau die Schnittstellen, die die Klasse System.Object implementiert, wie Abb.10 am Beispiel zeigt.

```
Public Class Class1



End Class

Public Module Module1
    Private Sub test()
        Dim c As New Class1()
        c.
        GetType

    End Sub

End Module
```

Abb. 10: eine von System.Object abgeleitete Klasse

## 2.4 Die Intermediate Language IL

Eine wichtige Frage bezüglich des Einsatzes eines Softwareentwicklungssystems für eine bestimmte zu erstellende Applikation, ist die der Ausführungsgeschwindigkeit (Performance) des zu erstellenden Codes. Dabei ist jedoch dabei der Begriff der Performance nicht als statische Größe zu bewerten, sondern vom Einsatzzweck aus der jeweiligen Anwendung relativ zu betrachten. So muss z.B. eine Anwendung, die ein grafisches Interface bereitstellt, in der Regel so schnell sein, das der Benutzer nicht auf lange, nervenaufreibende Verzögerungen im Bezug auf die Interaktion mit der Anwendung warten muss. Solch ein Beispiel stellt im Allgemeinen für heutige moderne Entwicklungssysteme keine nennenswerten Probleme bezüglich ihrer Performanceleistung dar. Bei der Entwicklung einer solcher Applikation spielt somit die Frage nach der Performance des Codes eines Softwareentwicklungssystems nur noch eine untergeordnete Rolle. Es treten andere wichtige Entscheidungskriterien für eine einzusetzende Entwicklungsplattform in den Fordergrund, z.B. eine gute Bedienbarkeit der Entwicklungsumgebung oder aber ein großer Funktionsumfang des Entwicklungssystems. Alles also Fragen und Kriterien die eine ökonomisch, wirtschaftliche Softwareentwicklung ermöglichen.
Anders dahingegen sieht es bei Anwendungen aus, die rechenintensive Operationen bewerkstelligen müssen. Um ein möglichst gutes, also performantes Resultat erzielen zu können, muss ein Softwareentwicklungssystem gewählt werden, welches entsprechend performanten Code liefert.
Wie also verhält sich nun .NET im Bezug auf seine Ausführungsgeschwindigkeit? Die Philosophie der .NET-Plattform ist es, dass die vorhandenen .NET-Compiler bestehenden Sourcecode nicht direkt in Maschinencode, sondern in eine Zwischensprache, die Microsoft Intermediate Language MSIL übersetzen. Diese Technik erinnert sehr stark an Java Bytecode oder etwa VB P-Code, was natürlich im Kontext der Performancediskussion den Schluss zulässt, MSIL Code sei ausgeprägt inperformant. Jedoch wird im Gegensatz zu dem erwähnten Java Bytecode und VB P-Code, welche während ihrer Ausführung von einem Interpreter interpretiert werden, der IL Code nicht interpretiert, sondern in Maschinencode kompiliert. Die Kompilierung findet jedoch erst auf dem jeweiligen Zielsystem der Applikation statt. In der Regel kommt hierfür auf dem Zielsystem ein so genannter Just-In-Time-Compiler(Jitter) zum Einsatz. Die Performance der Anwendung hängt also in erster Linie von dem eingesetzten Just-In-Time-Compiler ab. Die vorhandenen JIT-Compiler sind auf Grund ihrer Arbeitsweise sehr leistungsstark. Sie kompilieren den IL-Code einer Anwendung nicht am Stück, sondern immer nur sehr kleine Fragmente, im Normalfall einzelne Methoden vor ihrer Ausführung. Einfache Performancetests ergeben, das IL-Code generiert vom C#-Compiler maximal 50% langsamer ist, als C++ Maschinencode. Heutiger VB6 Maschinecode dagegen sogar um bis zu 300% langsamer als entsprechender IL-Code.
Zusammenfassend kann man sagen, dass .NET Sprachencode zwar durchaus langsamer als nativer C++ Code ist, jedoch fallen die Unterschiede hierbei sehr gering aus, und erreichen im Vergleich zu VB6 Code sogar eine wesentlich höhere Performance. Darüber hinaus sorgt die gemeinsame IL-Code Erzeugung und JIT-Compilation dafür, dass es nur unwesentliche Unterschiede zwischen der Performance von C# und VB.NET gibt. Die IL bringt also unterschiedliche Sprachdialekte im Bezug auf ihre Performance näher zusammen. Dies ist natürlich nicht der Grund für den Einsatz einer Zwischensprache innerhalb des .NET Konzepts.
Einer der Gründe für die Verwendung des Prinzips der Zwischensprache ist im Bereich der Realisierung einer Plattformunabhängigkeit von Software zu suchen. Bis vor nicht all zu langer Zeit fand die Softwareentwicklung sehr nahe an gegebenen, homogenen Hardwaresystemen statt. Applikationen wurden in erster Linie für ein bestimmtes Zielsystem entworfen, sprich für einen bestimmten Prozessor oder ein Betriebssystem. Software konnte ohne weiteres nach erfolgreicher Entwicklungsphase in den nativen Maschinencode des jeweiligen bekannten Zielsystems kompiliert werden.
Mit der zunehmenden Verbreitung des Internets kann nun die Frage nach dem Zielsystem, auf welchem die Anwendung einmal zur Ausführung kommt, nicht mehr ohne

weiters beantwortet werden. Die für die Ausführung von Maschinencode so wichtige Homogenität ist auf Seiten der Hardware nicht mehr gegeben. Um jedoch die korrekte Ausführung einer Anwendung zu gewährleisten, wird die benötigte Homogenität in einer abstraktern, hardwareunabhängigeren Ebene, in der der Zwischensprache, realisiert. Mit der Verwendung einer solchen Zwischenschicht zwischen Hardware und Software ist es jetzt nur noch notwendig, dass auf dem applikationsausführendem Zielsystem ein entsprechender Compiler vorhanden sein muss. Im Hinblick auf Marktentwicklungen, wie etwa Handheld PC's oder andere SmartDevices, wird die sich entwickelnde zukünftige Hardwarelandschaft immer heterogener. Somit sichert der Einsatz des Konzeptes der Zwischensprache die Zukunftsfähigkeit von heute zu entwickelnden Softwareprojekten. Da es für den Softwareentwickler zunehmend undurchsichtiger wird, auf welchem Zielsystem einmal seine Applikation laufen wird, ist es also obsolet den entwickelten Sourcecode direkt in Maschinencode zu übersetzen.
Ein wesentliches Prinzip von IL ist es also, prozessorunabhängig zu sein, damit im heterogenen Hardwareumfeld Software allgemein zuverlässig eingesetzt werden kann.

## 2.5 Das Remotingframework

Moderne Softwarearchitekturen benötigen für die Realisierung und Bereitstellung von Komponentenmodellen entsprechende Mechanismen, um eine Kommunikation zwischen den einzelnen Komponenten zu ermöglichen. Da einzelne Softwarekomponenten nicht unbedingt im selben Prozessraum, in verteilten Architekturen ja noch nicht mal auf dem gleichen System laufen, sind an die Kommunikationsmechanismen spezielle Anforderungen geknüpft. Damit ist es jedoch meist nicht auszuschließen, dass die Bedienung und Verwendung einer entsprechenden Kommunikation kompliziert wird. Bestehende Komponentenmodelle, wie DCOM oder CORBA kapseln diese Kommunikationsmechanismen nahezu vollständig vor dem Anwender. Neben dieser guten Benutzbarkeit für den Anwender wurden die Schwächen solcher Architekturen im Kapitel 1.2 erläutert. Sie sind mit ihren Binärstandart stark an homogene Systeme gebunden und machen es dadurch enorm aufwändig, die Systemgrenzen zu überschreiten.
Der Wunsch des Entwicklers ist es daher, Mechanismen an die Hand zu bekommen, die einerseits leicht zu bedienen sind und auf der anderen Seite die Implementierung systemübergreifender, verteilter Anwendungen ermöglichen.
Diesen Anforderungen wird das .NET Remoting Framework ansatzweise gerecht. Es stellt dem Entwickler eine umfassende Klassenbibliothek zur Verfügung, mit deren Hilfe sich verteilte Anwendungsszenarien entwickeln lassen. Um eine Systemunabhängigkeit erreichen zu können setzt das .NET Remoting Framework auf verschiedene Protokollstacks, wie etwa TCP oder HTTP, sowie SOAP.
Die Einfachheit der Anwendung der Kommunikationsmechanismen, wie sie z.B. von DCOM her bekann ist, ist zum Teil einer besseren Skalierbarkeit gewichen. So kann z.B. der zu verwendende Protokollstack von außen mittels Konfigurations-Datei zur Laufzeit einer Anwendung getauscht werden.
Die bessere Skalierbarkeit macht es jedoch erforderlich, einen höheren Implementierungsaufwand zu betreiben. So muss eine Server-Komponente, bevor sie ihren Serverdienst aufnehmen kann, im .NET Remoting angemeldet werden. Diese Anmeldung geschieht innerhalb eines Hostprozesses. Wird der Hostprozess beendet, so erlischt auch die Registrierung der Server-Komponente im .NET Remoting. Bei der Registrierung der Servers wird zudem dessen Eigenschaft festgelegt. Das .NET Remoting unterscheidet zwischen drei verschiedenen Eigenschaften eines Server-Objekts, die das Verhalten des Objektes innerhalb des .NET Remotings bestimmt.

Single Call Objekte

Single Call Objekte zeichnen sich durch ihre Statuslosigkeit aus. Ruft ein Client eine Methode eines solch konfigurierten Objektes auf, existiert diese lediglich für die Zeit der Methodenverarbeitung. Beim Aufruf der Methode wird es entsprechen instanziert, die Methode abgearbeitet und anschließend wieder zerstört. Etwaig gesetzte Attribute des Objekts bleiben somit nach Methodenaufruf nicht gespeichert und gehen verloren. Single Call Objekte eignen sich daher besonders für statuslose Applikationen. Mit ihnen lassen sich einfach ferne Funktionsaufrufe implementieren, so z.B. Konvertierungsfunktionen, die aus einer Anzahl übergebender Parameter einen bestimmten Rückgabewert berechnen und als Ergebnis zur Verfügung stellen.

Singleton Objekte

Singleton Objekte haben die Eigenschaft, singulär im gesamten .NET Remoting aufzutreten. Eine entsprechende Instanz eines als Singleton konfigurierten Objektes existiert lediglich einmal. Ist ein solches Objekt an irgendeiner Stelle instanziert worden, kann, solange diese Instanz besteht, kein weiteres Objekt instanziert werden. Stati

bleiben, wie bei gewöhnlichen Objekten während der gesamten Objektexistenz erhalten.

CAO Client Activated Objekt

Das Konzept der CAO´s ist dem der ActiveX-EXE Komponenten der COM/DCOM Umgebung vergleichbar. Der Client fordert eine Objektinstanz, dadurch wird im Hostprozess der Server-Komponente ein neues Objekt instanziert und eine entsprechende Referenz an den Client übermittelt.
Für jedes clientseitig angeforderte Objekt wird ein entsprechendes Objekt im Hostprozess instanziert.
Mit dieser Architektur lassen sich auch Datenpools schaffen, über die die einzelnen Objekte Daten gemeinsam nutzen können und so allen vorhandenen Clients ein konsistentes Datenmodell liefern können.

Da der SOM-Server den Status eines singulären Systems bereitstellen soll, somit nach Möglichkeit keine Objektkopie an verschiedenen Orten erstellt werden sollte, um Inkonsistenten entgegen zu wirken, fällt die Wahl der Konfigurationskategorie für die im Projekt verwendeten Remotingobjekte eindeutig für Objekte der Singelton-Kategorie aus. Ein solches, mit Singleton-Objekten realisiertes Szenario gewährleistet zu jedem Zeitpunkt die Datenkonsitenz für jeden Client des SOM-Servers.

Damit ein Objekt remotingfähig wird muss dieses von der Klasse **MarshalByRefObject** abgeleitet werden, somit ist das .NET Remoting Framework in der Lage, ein entsprechendes Objekt zu serialisieren.

```
public class SOMServer : System.MarshalByRefObject
{
}
```

Das Objekt muss innerhalb eines Hostprozesses ausgeführt werden. Dieser Prozess muss dafür sorgen, dass das Objekt im .Net Remoting Framework registriert wird.
Die Registrierung umfasst das Reservieren eines Ports für die Kommunikation zwischen Objekt und entsprechenden Clients des Objekts sowie die Bekanntgabe des Aufenthaltsortes des Objekts.

```
ChannelServices.RegisterChannel(new HttpChannel(1234));
SOMServer SOMServer = new SOMServer();
// .  SOM Server konfigurieren, z.B. logische Geräte laden etc.
// .
// .
RemotingServices.Marshal(SOMServer, "SOMServer.soap");
```

Im ersten Schritt werden für die Hostanwendung ein entsprechendes Protokoll sowie ein Port gewählt und registriert. Im angeführten Beispiel wurde http als zu verwendendes Protokoll und der Port 1234 gewählt.
Nachfolgend wird ein entsprechend von MarshalByRefObject abgeleitetes Objekt instanziert, hier ein Objekt der Klasse SOMServer.
Im dritten Schritt wird mit der Methode Marshal der Klasse RemotingServices das von MarshalByRefObject abgeleitetes Objekt im Remoting angemeldet, sowie die entsprechende Aufruf-URI für dieses Objekt festgelegt.

Über den bekannten Kanal und die entsprechenden URI kann nun auf Clientseite ein Verweis auf das Server-Objekt, welches in der Hostanwendung läuft, erzeugt werden.

```
Dim SOMServer As SOM.IServer

SOMServer = Activator.GetObject(GetType(SOM.IServer),
 "http://localhost/SOMServer.soap")
```

Die Methode GetObject der Klasse Activator liefert den Verweis auf das Server-Objekt. Als ersten Parameter empfängt die Methode den Datentyp des Objektes, auf den der Verweis zeigen soll, der zweite Parameter gibt die Lokation des Remote-Objektes, nebst zu verwendendem Übertragungsprotokoll an. Der erste Teil der Zeichenkette gibt das Protokoll an, hier 'http:', gefolgt von dem Rechner, der das Remote-Objekt ausführet. Das Schlüsselwort 'localhost' steht für die lokale Maschine, d.h. Server und Client befinden sich in dem Fall auf dem gleichen System. Der letzte Teil spezifiziert das entsprechende Remote-Objekt. Dies entspricht der Zeichenkette, die der Methode 'Marshal' der Klasse RemotingServices als zweiter Parameter bei der Anmeldung der Objektes übergeben wurde.

Die verwendeten statischen Klassen befinden sich in den folgenden Namespaces:

| Klasse | Namespace |
|---|---|
| ChannelServices | System.Runtime.Remoting.Channels |
| RemotingServices | System.Runtime.Remoting |
| Activator | System |

Um die Server/Client-Kommunikation zwischen zwei Objekten auf entfernten Rechnern zu ermöglichen, müssen die Sicherheitsrichtlinien des .Net Remoting Frameworks richtig eingestellt werden.

Auf Clientseite muss für die entsprechende Netzwerkzone das Vertrauenslevel für Assemblies auf 'Full Trust' gesetzt werden.

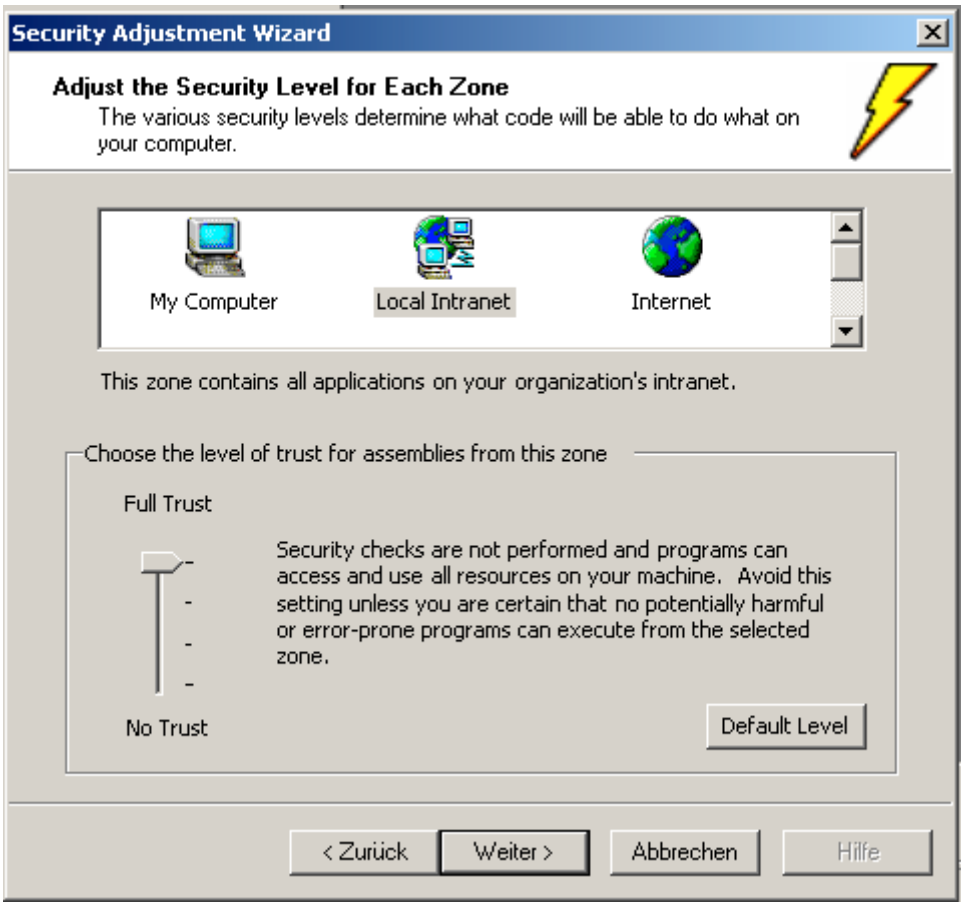
Die Einstellung kann wie folgt beschrieben vorgenommen werden:

Den Microsoft .NET Framework Wizards aufrufen, der sich in der Systemsteuerung/Verwaltung befindet.

Adjust .net security aufrufen


Anschließend die entsprechende Option wählen, ob die Einstellungen für die gesamte Maschine oder aber nur für den angemeldeten Benutzer gelten sollen.

Das Security Level für den auszuwählenden Bereich auf Full Trust setzen.




Nach der Eingabebestätigung mit 'weiter' werden die Security Levels für alle Bereiche noch einmal zusammengefasst dargestellt.

## 2.6 .NET Assembly

Ein Assembly stellt im .NET Kontext eine ausführbare Einheit dar, vergleichbar mit einer klassischen COM-Komponente. Jedoch beschreibt ein Assembly lediglich eine logische Zusammenstellung, d.h. ein Assembly muss nicht zwingend eine einzelne Datei sein, sondern kann auch aus mehreren Dateien bestehen. Neben dem eigentlichen ausführbaren Programmcode, der in IL Notation (Intermediate Language) im Assembly enthalten ist, stellt das Assembly weitere Daten zur Verfügung, s.g. Metadaten. Diese Metadaten haben die Aufgabe, die innerhalb des Programms verwendeten Typen und Klassensignaturen, sowie Referenzen zu anderen Assemblies zu beschreiben. Durch diesen strukturellen Aufbau der Assemblies, d.h. dass jedes Assembly neben dem Programmcode auch Informationen über selbigen mit sich führt, werden die Assemblies befähigt, sich selbst zu beschreiben. Das bedeutet, dass zur Installation einer Programmkomponente auf einem Zielsystem keine Informationen über die jeweiligen Komponenten an zentraler Stelle, z.B. der Registry, hinterlegt werden müssen, da genau diese Informationen von jeder Komponenten eigenständig mit sich geführt werden.
Dies bedeutet aber gleichzeitig auch, dass es Möglichkeiten und Wege geben muss, die es ermöglichen diese Informationen aus einem Assembly zu extrahieren.
Zu diesem Zwecke stellt das .NET Framework die Reflection Klassenbibliothek zur Verfügung. Mittels dieser Klassen ist es möglich, detaillierte Informationen über die in einem Assembly enthaltenen Typen und Klassen nebst deren vollständigen Signaturen zur Laufzeit zu ermitteln.

An einem Planspiel soll die Notwendigkeit des Einsatzes der Reflection Klassen innerhalb der SOM Server Implementation verdeutlicht werden.

In einen sich im Einsatz befindenden SOM Server, soll ein weiterer logischer Knoten, sprich eine Gerätefunktionalität hinzugefügt werden, deren Implementierung zur Projektierungsphase des SOM Servers noch nicht zur Verfügung stand. Der neue logische Knoten wird in einer .NET DLL bereitgestellt und entspricht dem Standard durch die Implementierung der Schnittstelle ILogicalNode des SOM´s. Um nun die Funktionalität des logischen Knotens nutzen zu können, muss dieser instanziert werden. Dafür ist der Klassenname des logischen Knotens notwendig, der jedoch zur Projektierungsphase des Servers noch nicht bekannt war. Mit Einsatz der Reflectionklassen lassen sich jetzt zur Laufzeit jedoch sämtliche Informationen, somit auch der Klassenname des neuen logischen Knotens aus der DLL ermitteln.

Der folgende Codeausschnitt verdeutlicht den Zugriff auf die in einem Assembly enthaltenen Typen.
Zunächst wird eine angegebene .NET DLL geladen um anschließend durch die im Assembly enthaltene Typen mit einer `For Each`-Schleife durch zu iterieren.
Zudem wird noch jeder einzelne Typ durchleuchtet, ob er die Schnittstelle SOM.ILogicalNode implementiert hat.

```
Private m_Assembly As System.Reflection.Assembly
Private m_NewLN As SOM.ILogicalNode

Private Sub cmdSearch_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles cmdSearch.Click

        Dim types As Type()
        Dim t As Type
        Dim inter As Type

        Me.OpenFileDialog1.Filter = "dll|*.dll"

        If Me.OpenFileDialog1.ShowDialog() = DialogResult.OK Then
            m_Assembly =
Reflection.Assembly.LoadFrom(Me.OpenFileDialog1.FileName)
            Me.txtAssembly.Text = Me.OpenFileDialog1.FileName
            ' get type names from the assembly.
            types = m_Assembly.GetTypes

            Me.cbxAllTypes.Items.Clear()
            For Each t In types

                For Each inter In t.GetInterfaces()
                    If inter.FullName = "SOM.ILogicalNode" Then
                        Me.cbxAllTypes.Items.Add(t.FullName)
                    End If
                Next

            Next t

        End If
End Sub
```

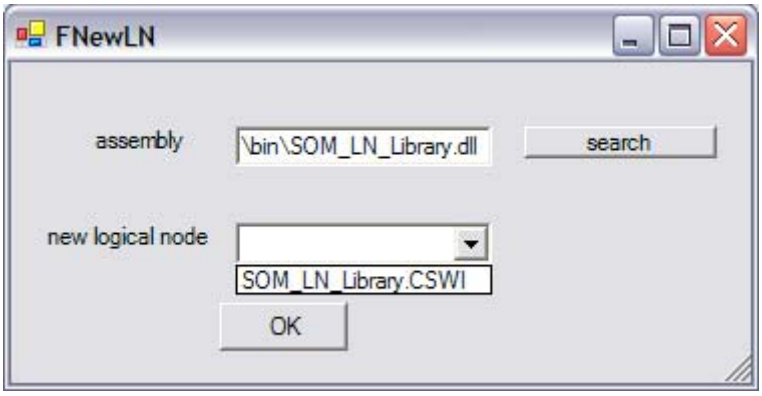



Abb. 11: Beispiel-Anwendung mit .NET-Assembly-Daten

Das Resultat dieser Routine könnte sich visuell wie in Abb. 11 oder ähnlich darstellen. In der dargestellten ComboBox sind alle in der SOM_LN_Library.dll enthaltenen Typen, die die Schnittstelle SOM.ILogicalNode implementieren, aufgelistet. In diesem exemplarischen Fall lediglich ein Typ, die Klasse CSWI.

# 3. Visual Studio .NET

## 3.1 Allgemeines

Mit der Einführung eines gänzlich neuen Softwareentwicklungskonzeptes liefert Microsoft auch eine neue Entwicklungsumgebung. Visual Studio .NET ist die konsequente Zusammenführung bestehender Entwicklungsumgebungen, wie die von Visual Basic, die von Visual C++, sowie das zur Web-Entwicklung zur Verfügung stehende Visual InterDev. Da Visual Studio .NET eine einheitliche Benutzerschnittstelle für alle Programmiersprachen sowie für Web-Projekte bereit stellt, bleibt es nicht aus, dass es in seiner Bedienung im Vergleich zu anderen bestehenden Entwicklungsumgebungen komplexer wirkt.
Nach dem Start der Entwicklungsumgebung präsentiert diese sich mit einem zentralen webbasierten Browserfenster, der Startseite.

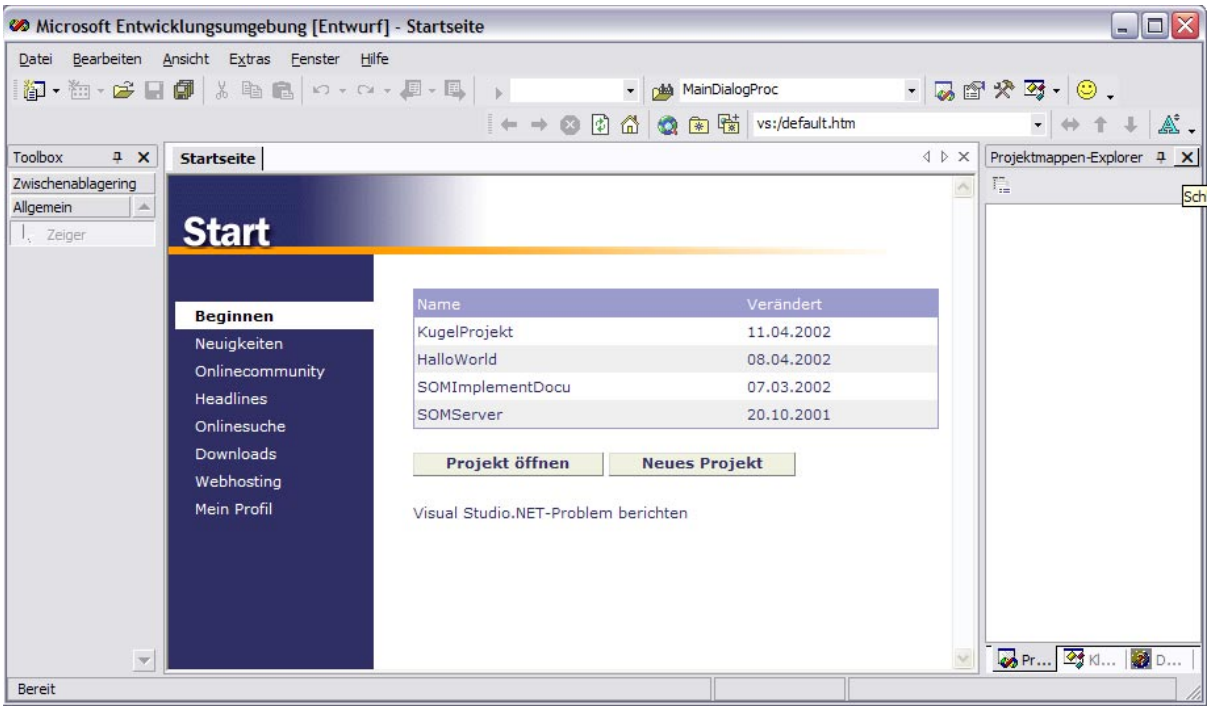



Abb. 12: Startseite der .NET-Entwicklungsumgebung

Im linken Frame der Seite sind einige Links aufgelistet, die im Kontext von Visual Studio .NET stehen. Unter dem Link ‚Mein Profil' lässt sich zum Beispiel die Entwicklungsumgebung an die persönlichen Bedürfnisse anpassen. Im Hauptframe der Seite werden standardmäßig die zuletzt verwendeten Projekte aufgelistet. Diese können von der Seite aus direkt gestartet werden. Auch können von hieraus neue Projekte angelegt werden.

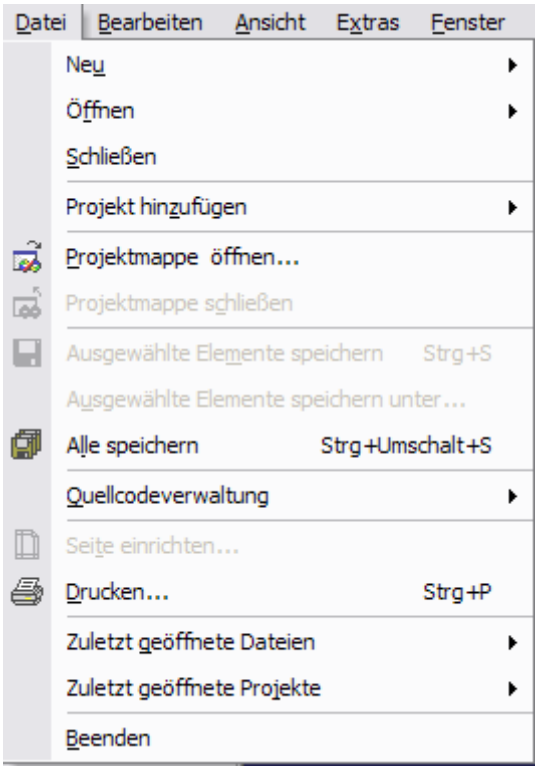



Abb. 13: Datei-Menü

Über den Menüpunkt **Datei** lassen sich die aus der Office-Welt bekannten Prozeduren zur Verwaltung der Programmdaten aufrufen. Hier können Projekte oder einzelnen Dateien neu angelegt, sowie bereits bestehende Daten zur weiteren Bearbeitung aufgerufen werden. Zudem kann die Druckausgabe von Daten gesteuert werden sowie auch die gesamte Entwicklungsumgebung geschlossen werden.

Der Menüpunkt **Bearbeiten** stellt Funktionen zur Verwendung der Zwischenablage bereit. Über diese können Objekte kopiert, gelöscht oder zuvor entfernte Objekte wieder eingefügt werden. Des Weiteren bietet das Menü Zugang zu einer Suchfunktion, um innerhalb von Quellcode nach bestimmten Passagen zu suchen.

Über den Menüpunkt **Ansicht** lassen sich alle Tool-Fenster der Entwicklungsumgebung ein- oder ausblenden. Visual Studio .NET stellt eine Reihe von Tool-Fenstern für die verschiedensten Aufgaben bereit. Dadurch wird erreicht, dass nahezu alle Komponenten eines Projektes sowie diese selbst visuell verwaltet werden können.

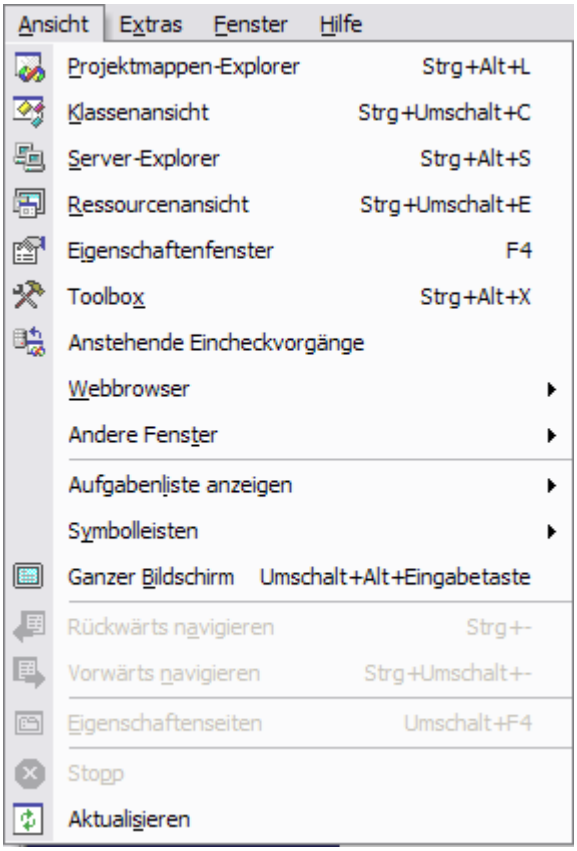



Abb. 14: Ansicht-Menü

## 3.2 Die Fenster der Entwicklungsumgebung

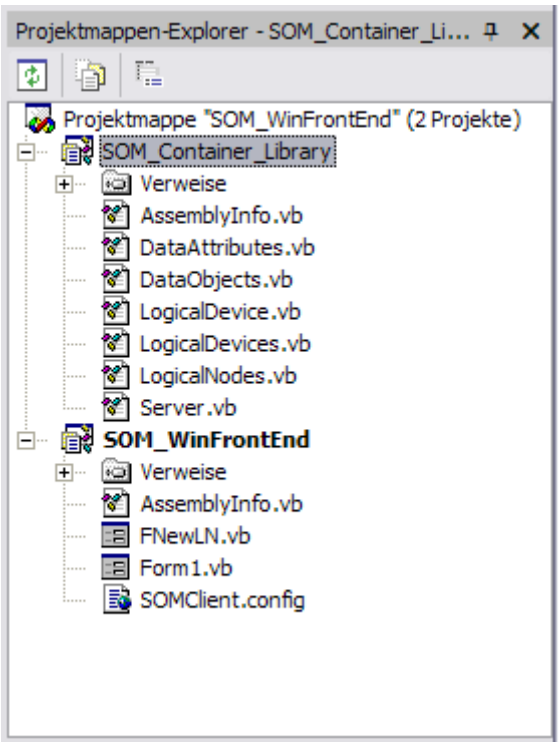



Abb. 15: Der Projektmappen-Explorer

Der **Projektmappen-Explorer** verwaltet die physikalischen Einheiten eines Projektes, sprich die dem Projekt zugehörigen Dateien, sowie Verweise auf andere Komponenten. Dabei ist eine Projektmappe jedoch nicht auf die Aufnahme eines einzigen Projektes festgelegt, es können innerhalb einer Mappe mehrere Projekte verwaltet werden.
Die Steuerung der Verwaltung der Projektmappe erfolgt entweder über den Menüpunkt Datei oder aber über das PullDown-Menü des Projektmappen-Explorers selbst. Dieses wird mittels rechter Maustaste geöffnet.
Hier können dann neue oder bereits vorhandenen Dateien dem Projekt hinzugefügt werden oder aber Dateien aus dem Projekt ausgeschlossen werden.

Für die objektorientierte Softwareentwicklung ist die **Klassenansicht** ein überaus hilfreiches Verwaltungs- und Übersichtstool. Die Klassenansicht stellt in einem Softwareprojekt alle vorhandenen Klassen nebst ihren Methoden, Attributen und Schnittstellen dar. Dabei unterstützt die Klassenansicht die visuelle Ansicht der einzelnen Methoden oder Attribute mittels verschiedener Icons.

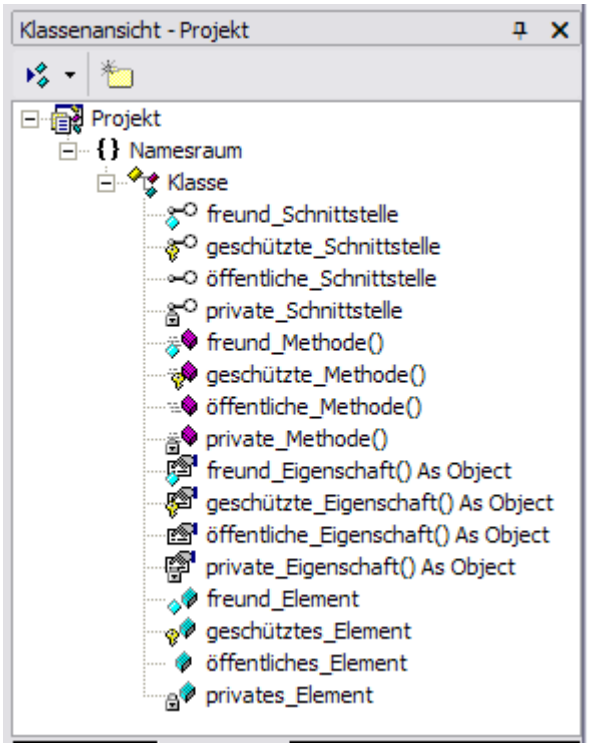



Abb. 16: Klassenansicht

Der **Server-Explorer** ist ein gänzlich neues Feature der Entwicklungsumgebung. Dieser bietet die Möglichkeit des graphischen Zugriffs auf verschiedene vorhandene Serverkomponenten, wie etwa den Zugriff auf einen installierten SQL Server. Auch lassen sich hier Datenbankverbindungen erstellen, die dann per Drag&Drop an das Projekt gebunden werden können.

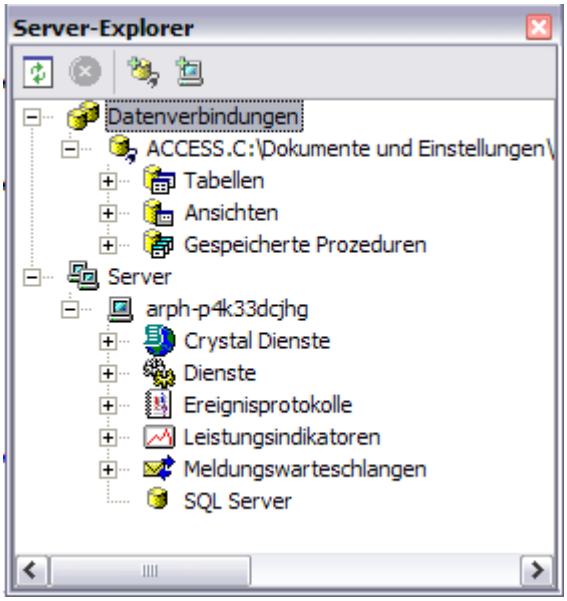



Abb. 17: Server-Explorer

Mittels des **Eigenschaftsfensters** lassen sich Eigenschaften von dem im Formulardesigner gerade selektierten Objekt einstellen. Hiermit lassen sich einem Objekt Defaultwerte, die beim Starten der Applikation gesetzt werden, einstellen. In der Regel handelt es sich dabei um Eigenschaften, wie etwa die Farbe oder die Größe eines Objektes.

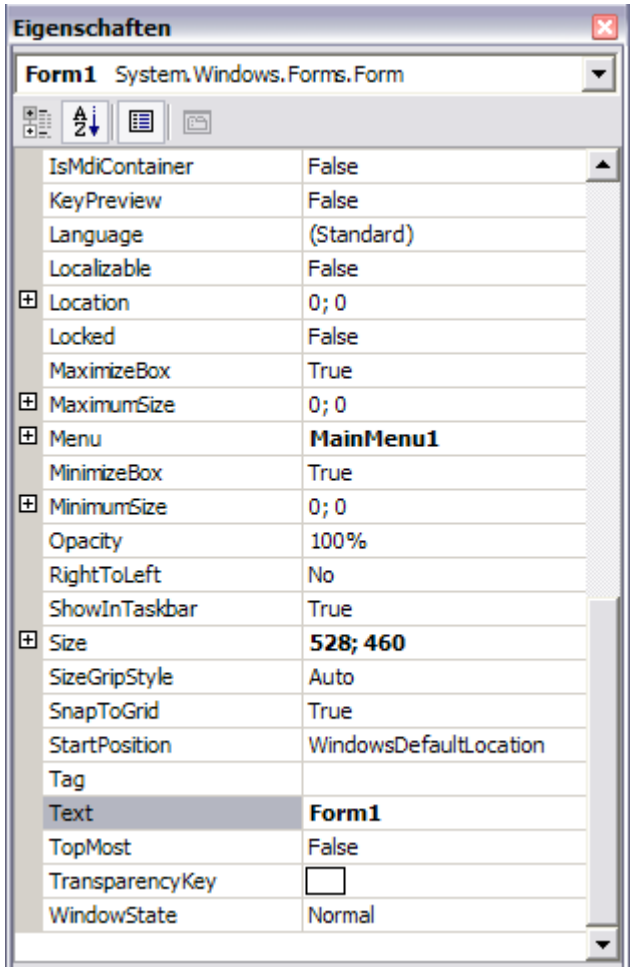



Abb. 18: Eigenschaftsfenster

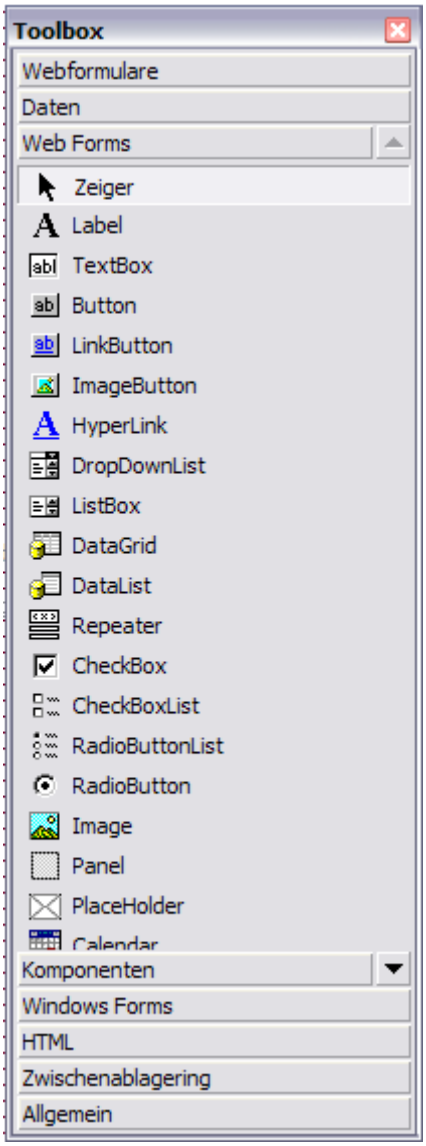



Abb. 19: Toolbox

Die **Toolbox** beinhaltet unter anderem eine Liste ausgewählter verfügbarer Komponenten. Diese Liste kann um weitere vorhandene Komponenten erweitert werden. Die einzelnen Komponenten werden für das visuelle Design eines Formulares verwendet. Nach der Selektierung einer Komponente kann mittels der Maus diese auf einem Formular im Formulardesigner platziert werden. Neben der dargestellten Toolbox für die Entwicklung von WebFormularen existiert natürlich auch eine Toolbox für die WindowFormular Entwicklung.

Bei allen vorhanden Fenster der Entwicklungsumgebung handelt es sich um sogenante Docking Windows. Diese bieten die Möglichkeit, jedes einzelne Fenster frei zu arrangieren. Bei den Docking Windows handelt es sich nicht um MDI-Fenster, die an das Hauptfenster gebunden sind und somit auch nur innerhalb dessen platziert werden können, sondern um Fenster die auch ausserhalb des Entwicklungsfensters platziert werden können. Auch können mehrer Fenster an ein und dieselbe Stelle auf dem Bildschirm positioniert werden.

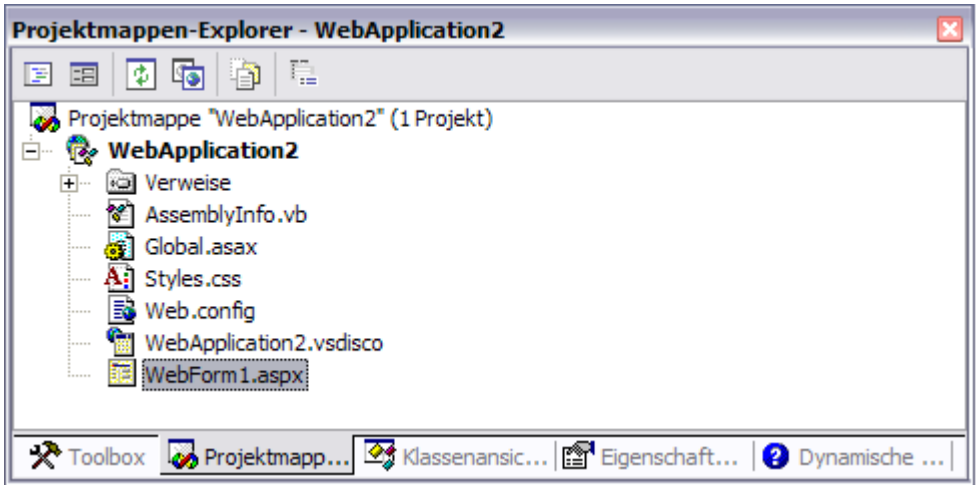


Abb. 20: Fensterarangement

In diesem Fall kann mittels Tabstrip zwischen den einzelnen Fenstern navigiert werden und das jeweils benötigte in den Vordergrund geholt werden.

## 3.3 Die Projekttypen

Nach dem Klick auf den Button Neues Projekt auf der Startseite oder des entsprechenden Menüpunktes im Menü **Datei** erscheint ein Dialog, der die Auswahl des Projekttyps sowie die Eingabe verschiedener Projektparameter verlangt.

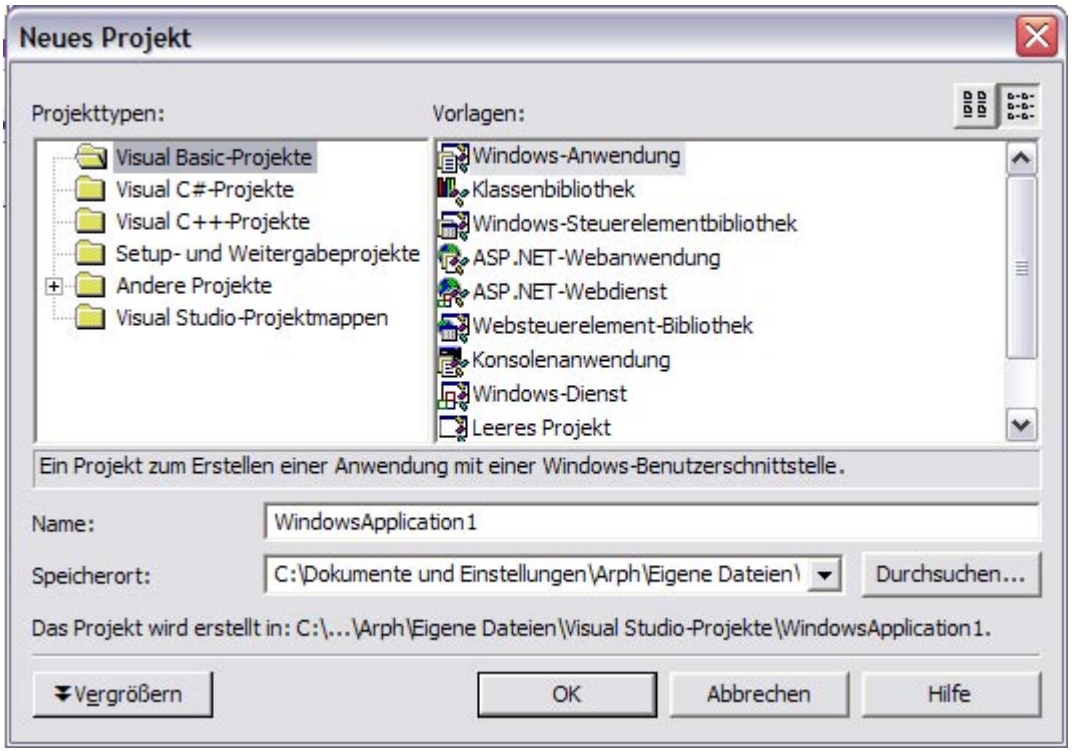
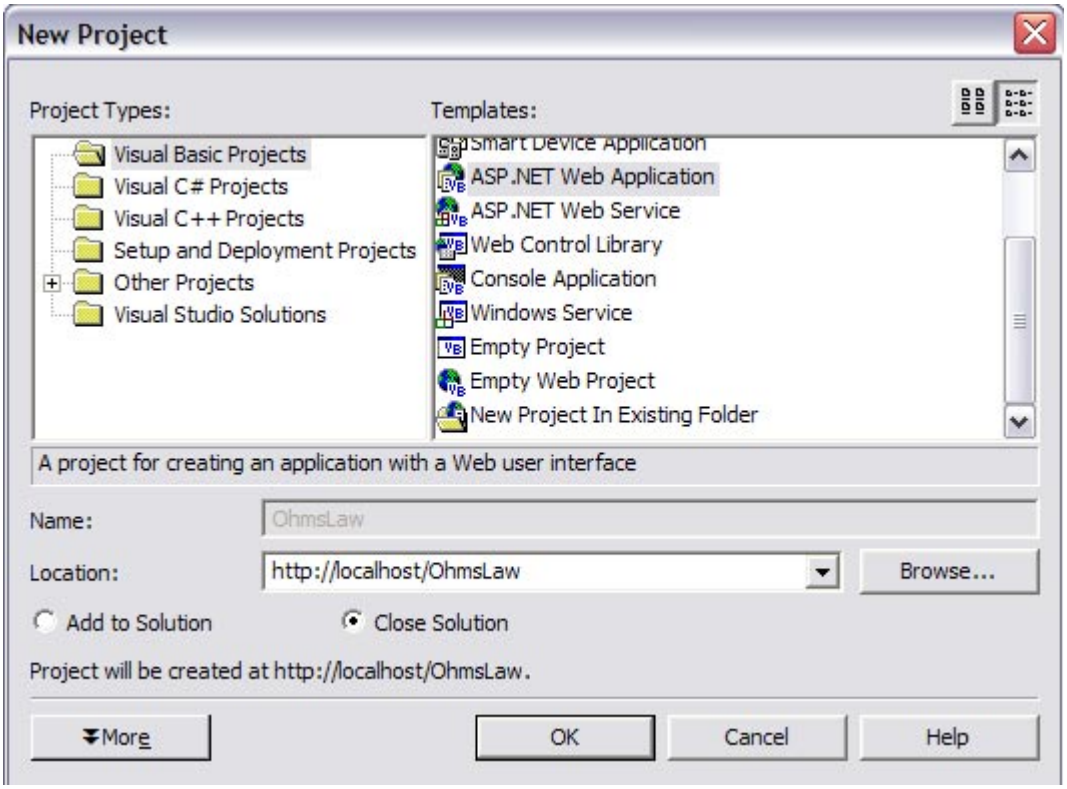


Abb. 21: Projekterstellung

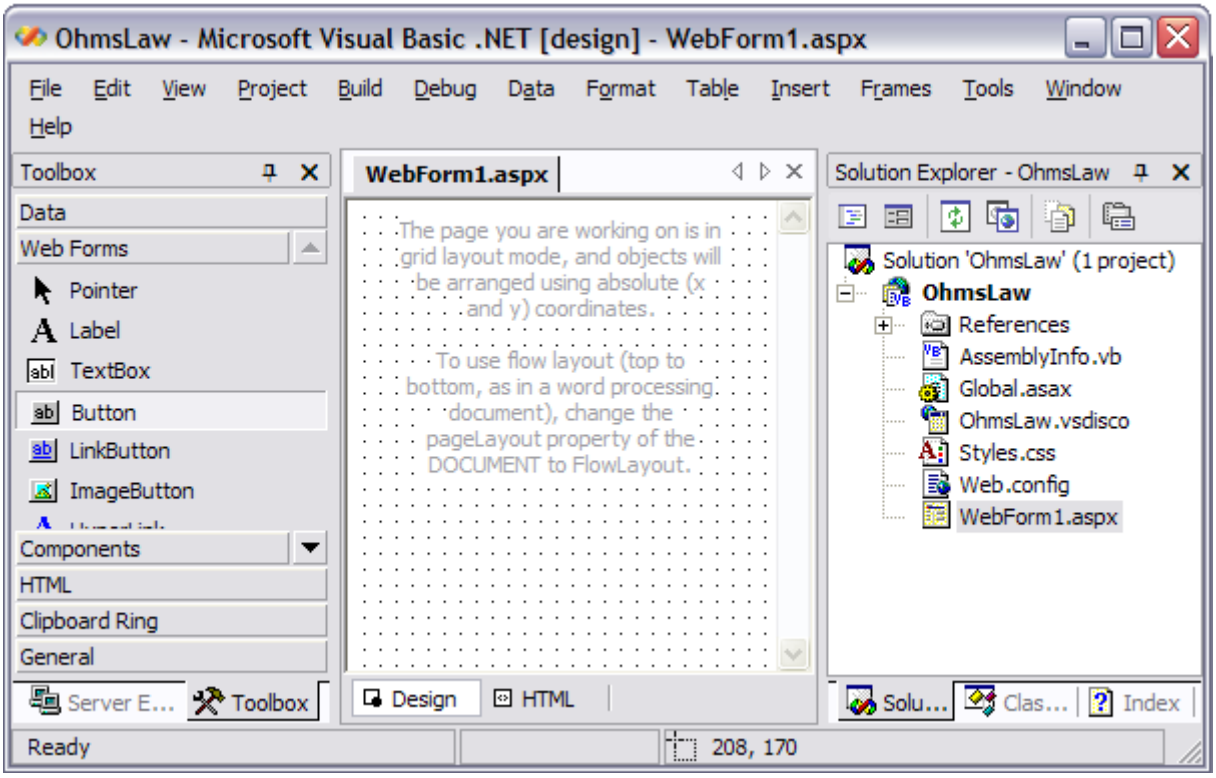
Hier kann zwischen den installierten Projekttypen gewählt werden. Standardmäßig sind im Visual Studio .NET Umfang Visual Basic, C# (sprich: C sharp) und C++ enthalten. Es stehen jedoch weitere Projekttypen zur Verwendung bereit, wie etwa J#, auch wird über die Einführung von Sprachen wie COBOL u.a., die das .NET-Framework unterstützen, gesprochen.
Neben dem Projekttyp muss hier die entsprechende Vorlage ausgewählt werden. So stellt VB z.B. die Vorlagen zur Entwicklung von Windows-Anwendungen oder Klassenbibliotheken und Steuerelementen, aber auch solche für die Implementierung von Webanwendungen zur Verfügung. Nach der Wahl des Projektnamens sowie dessen Speicherorts und der Bestätigung mittels des OK-Buttons, wird das neue Projekt angelegt. Hierzu werden alle notwendigen Dateien erzeugt und Standardwerte gesetzt.

Um den Projekttyp ASP.NET-Webanwendung etwas näher zu betrachten und die Verwendung der einzelnen vorgestellten Tools zu verdeutlichen, wird im weiteren das Anlegen einer Beispiel-ASP.NET Anwendung dargestellt. Aufgabe der Anwendung soll es sein, mittels eines HTML-Frontend einen Dialog zur Darstellung des Ohm'schen Gesetzes zu verwirklichen.

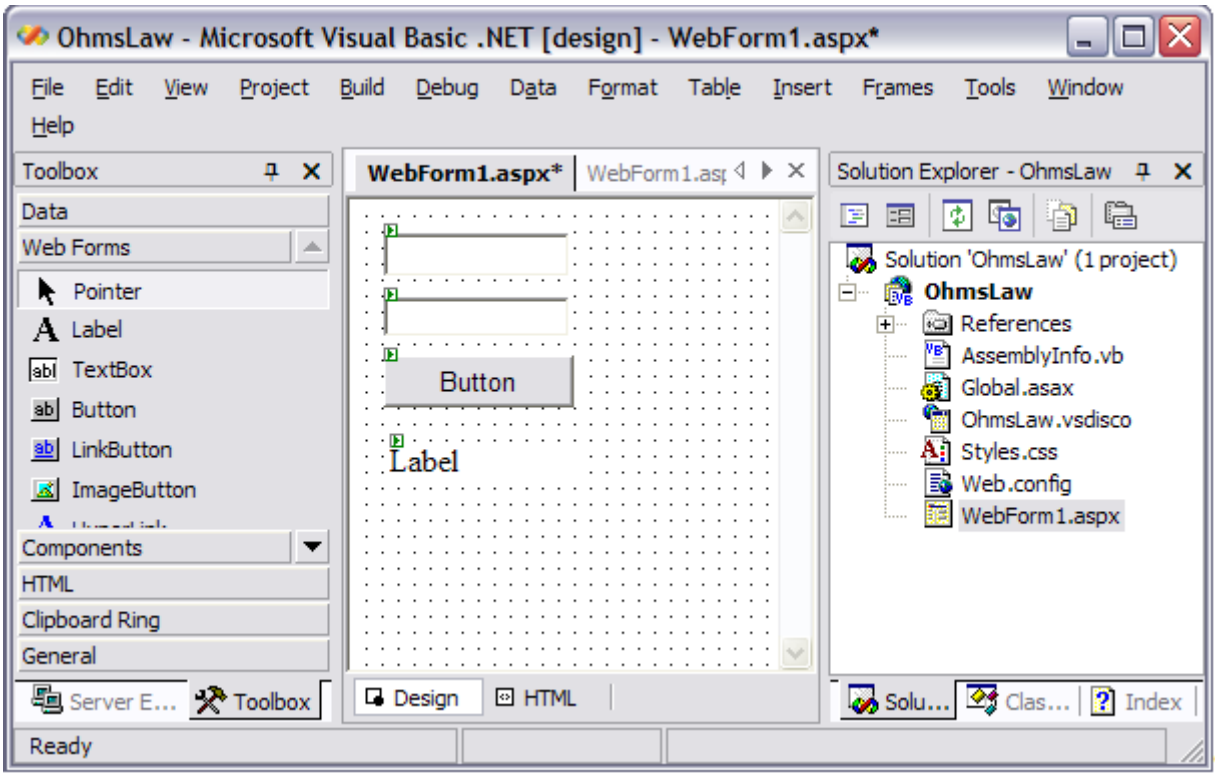
Nach dem Start der .NET Entwicklungsumgebung wird ein neues Projekt vom Typ ASP.NET-Webanwendung gewählt und diesem der Name „OhmsLaw" gegeben.



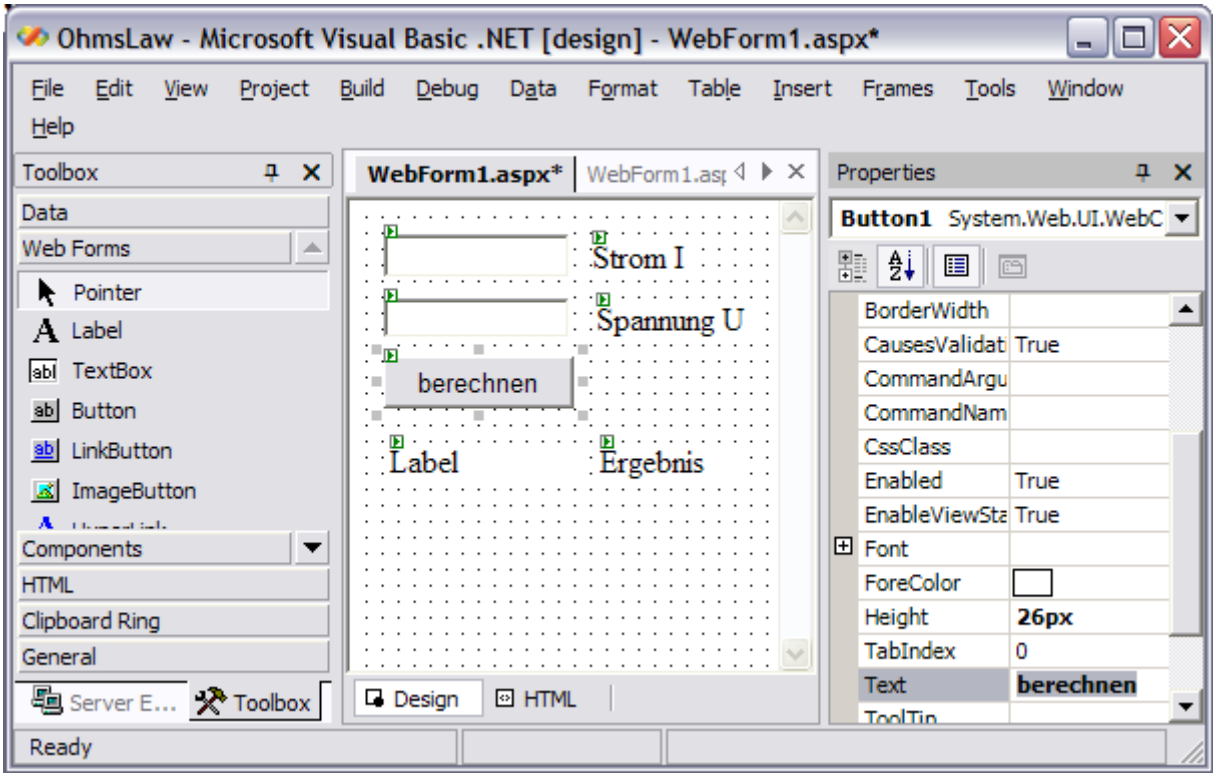

Mit der Bestätigung des Buttons „OK" wird im Standardverzeichnis des lokalen Webservers ein neues Web mit dem Namen „OhmsLaw" angelegt und dort die Projektdatei und andere dem Projekt angehörende Dateien gespeichert. So auch das standardmäßig angelegte Webformular WebForm1.aspx.



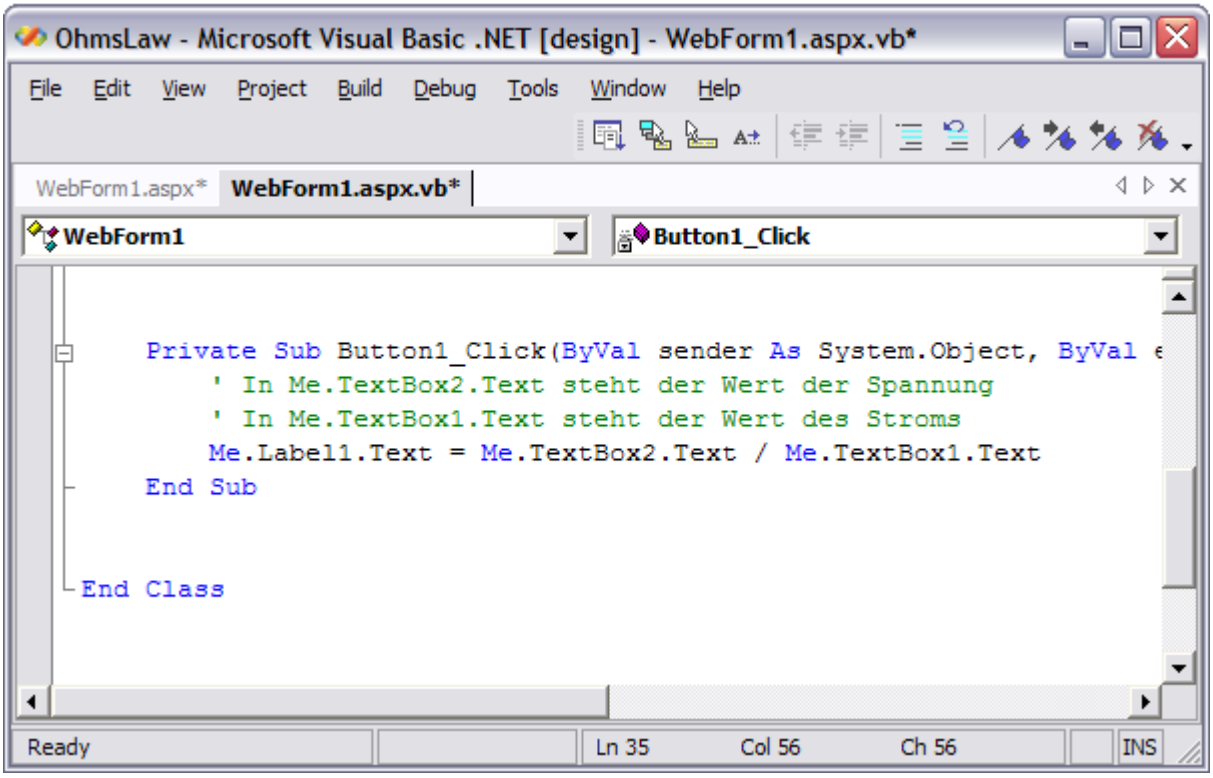
Auf das Webformular können jetzt mit Hilfe der ToolBox per Drag&Drop verschiedene graphische Komponenten platziert werden. Für die Umsetzung des Ohm'schen Gesetzes werden zunächst zwei TextBox-Elemente, sowie ein Button und ein Label auf das Webformular gelegt.



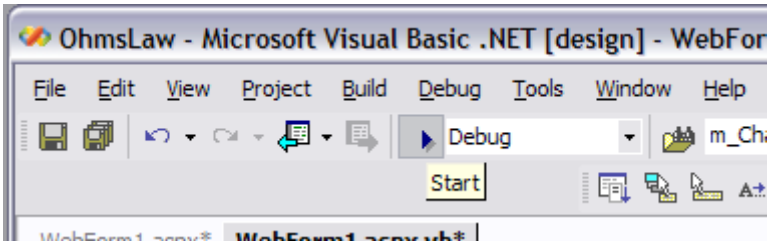

Des Weiteren kann mit zusätzlichen Label-Elementen eine Beschriftung der einzelnen Elemente vorgenommen werden.
Mit Hilfe des Eigenschaftsfensters wird nun die Text-Eigenschaft des Buttons auf den Wert „berechnen" gesetzt.



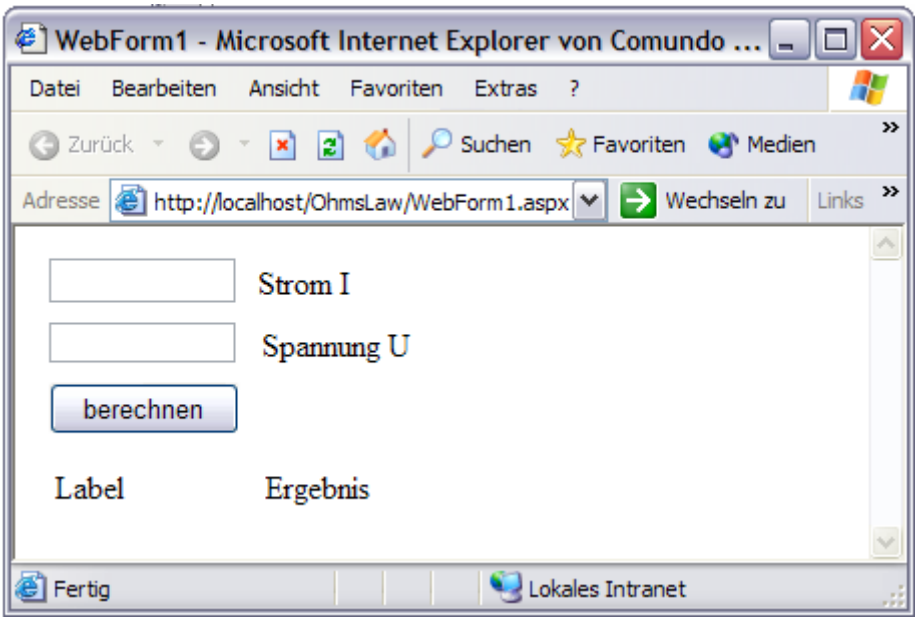
Nachdem das Design der Anwendung abgeschlossen ist, erfolgt nun die Dynamisierung des Formulars mittels Programmierung. Die Anwendung soll sich so verhalten, dass nach dem Klick auf den Button „berechnen" aus den zuvor angegebenen Werten für Strom und Spannung der entsprechende Widerstand berechnet wird und im Label1 angezeigt werden soll. Dies kann in der Ereignisbehandlungsroutine des Klickereignisses des Buttons realisiert werden. Um diese Ereignisbehandlungsroutine zu erzeugen ist einfach per Doppelklick auf den Button zu klicken. Daraufhin wechselt die Entwicklungsumgebung selbständig in die Programmcode-Ansicht des Webformulars. Innerhalb der Ereignisbehandlungsroutine des Klickereignisses gilt es jetzt den Widerstandswert aus Strom und Spannung zu berechnen und ihn im Label1 anzuzeigen.




Damit ist die Implementierung der Anwendung abgeschlossen und sie kann das erste Mal ausgeführt werden. Dies kann aus der Entwicklungsumgebung heraus erfolgen in dem der Button „Start" in der ToolBar „Standard" gedrückt wird.

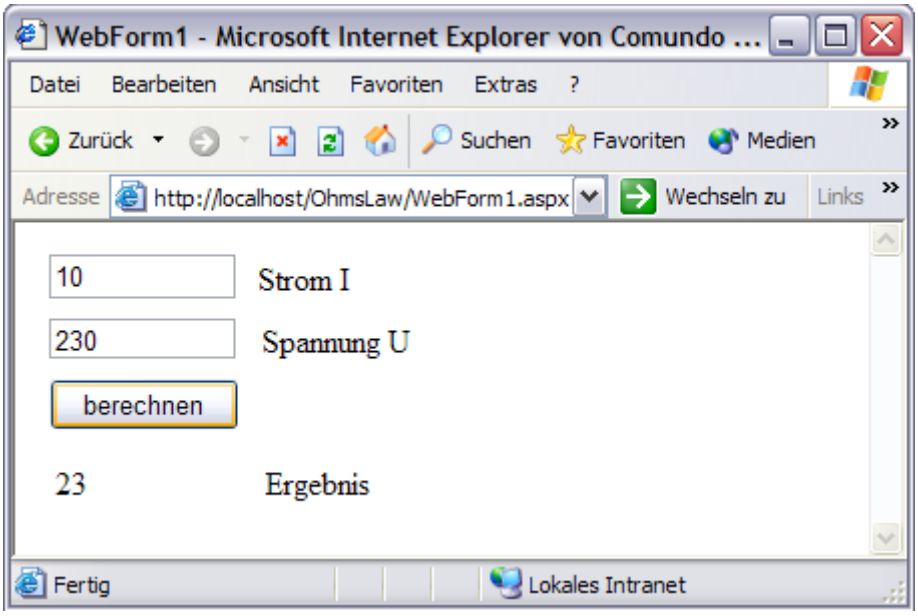



Zunächst wird der Quellcode der Applikation kompiliert. Danach ruft die Entwicklungsumgebung, da es sich um eine ASP.NET-Webanwendung handelt, den Standardbrowser des Rechnersystems auf, in diesem Fall den IE und übergibt diesem die Aufruf-URI der Webanwendung „OhmsLaw", d.h. also die URI: http://localhost/OhmsLaw/WebForm1.aspx.



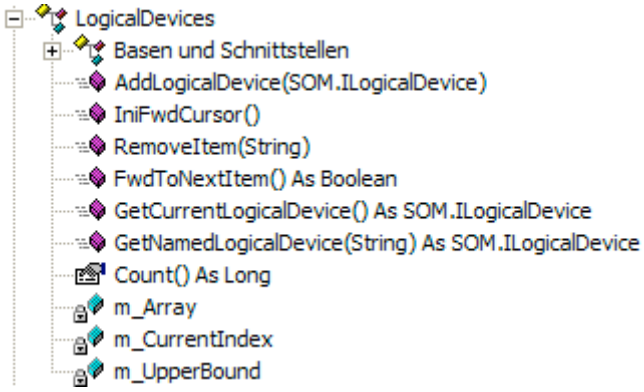

So stellt sich jetzt die Webanwendung im Browser da. Da die Webanwendung an den aufrufenden Browser reines HTML sendet, würde sich die Darstellung in einem anderen Webbrowser ähnlich gestallten, die Funktionalität auf jeden Fall die gleiche sein.

Jetzt können beispielhaft zwei Werte eingegeben und das Ergebnis berechnet werden.



## 3.4 Die Container-Klassen

In modernen, hierarchischen Objektmodellen spielen die sogenanten Containerklassen eine zentrale Rolle. Ihre Aufgabe ist es, Objekte gleichen Typs innerhalb des Objektmodells zu verwalten. Um mehrere Objekte innerhalb einer Containerklasse komfortabel verwalten zu können, sollte die entsprechende Containerklasse über bestimmte Methoden und Eigenschaften, die die Verwaltung der Objekte ermöglichen, implementieren.
In der Regel handelt es sich hierbei um Methoden, die das Hinzufügen eines neuen Objektes in den Container sowie das Entfernen eines bestehenden Objektes aus dem Container regeln. Hilfreich für die Anwendung von Containerklassen sind zudem Eigenschaften, die die Anzahl der vorhandenen Objekte innerhalb des Containers anzeigen, sowie es ermöglichen, auf ein spezielles indiziertes Objekt im Container zuzugreifen.
Des Weiteren ist es zweckmäßig eine Methode bereit zu stellen, die es ermöglicht durch die vorhandenen Objekte innerhalb des Containers durch zu iterieren.
In der Objekthierarchie des SOM-Schnittstellenmodells stellt die Klasse, die die Schnittstelle SOM.IServer implementiert, das Wurzelelement des Objektmodels dar. Diese Klasse stellt lediglich eine schreibgeschützte Eigenschaft zu Verfügung, die es ermöglicht auf den untergeordneten Container LogiclDevices, der wiederum die Schnittstelle SOM.ILogicalDevices implementiert, zu zugreifen. Der Container LogicalDevices implementiert die folgenden Methoden und Eigenschaften zur Verwaltung der einzelnen Objekte, die die Schnittstelle SOM.ILogicalDevice implementieren.




Innerhalb des Containers LogicalDevices werden die einzelnen Objekte, die die Schnittstelle SOM.ILogicalDevice implementieren, innerhalb eines privaten Array vom Typ SOM.ILogicalDevice gespeichert.
Des Weiteren sind die privaten Elemente zur Speicherung der Arrayobergrenze sowie die Speicherung des Indexes des aktuell selektierten Objektes vorhanden.

```
    Private m_Array As som.ILogicalDevice()
    Private m_UpperBound As Long
    Private m_CurrentIndex As Long
```

Mittels der Methode `AddLogicalDevice` lässt sich dem Container ein neues Objekt hinzufügen. Die Methode erhöht das private Array um ein weiteres Element, ohne die bereits vorhandenen Objekt zu löschen, und fügt im obersten Element des Arrays das zuvor übergebende Objekt hinzu. Anschließend wird die private Variable für die Obergrenze des Arrays inkrementiert.

```
    Public Sub AddLogicalDevice(ByVal LogicalDevice As _
        SOM.ILogicalDevice) Implements SOM.ILogicalDevices.AddLogicalDevice

        ReDim Preserve m_Array(m_UpperBound)
        m_Array(m_UpperBound) = LogicalDevice
        m_UpperBound = m_UpperBound + 1
    End Sub
```

Die schreibgeschützte Eigenschaft `Count` gibt den Wert der Arrayobergrenze zurück:

```
    Public ReadOnly Property Count() As Long Implements _
        SOM.ILogicalDevices.Count

        Get
            Count = m_UpperBound
        End Get
    End Property
```

Die Methode `GetCurrentLogicalDevice` liefert das aktuell indizierte Objekt aus dem Container:

```
    Public Function GetCurrentLogicalDevice() As SOM.ILogicalDevice _
        Implements SOM.ILogicalDevices.GetCurrentLogicalDevice

        GetCurrentLogicalDevice = m_Array(m_CurrentIndex)
    End Function
```

Die Methode `GetNamedLogicalDevice` liefert das Objekt mit dem übergebenen Namen aus dem Container. Dafür muss der gesamte Container durchiteriert werden, um das entsprechenden Objekt zu lokalisieren:

```
    Public Function GetNamedLogicalDevice(ByVal LogicalDeviceName As _
        String) As SOM.ILogicalDevice Implements _
        SOM.ILogicalDevices.GetNamedLogicalDevice

        Dim i As Long
        For i = 0 To m_UpperBound
            If m_Array(i).LogicaDeviceName = LogicalDeviceName Then
                GetNamedLogicalDevice = m_Array(i)
                Exit Function
            End If
        Next
    End Function
```

Die Methode `RemoveItem` entfernt bei Übereinstimmung mit dem übergebenen Name das entsprechende Objekt aus dem Container:

```
    Public Sub RemoveItem(ByVal SOMObjectName As String) Implements _
        SOM.ILogicalDevices.RemoveItem

        Dim i As Long
        Dim a As Long
        For i = 0 To m_UpperBound - 1
            If m_Array(i).LogicaDeviceName = SOMObjectName Then
                For a = i To m_UpperBound - 2
                    m_Array(a) = m_Array(a + 1)
                Next
                m_UpperBound = m_UpperBound - 1
                ReDim Preserve m_Array(m_UpperBound)
                Exit For
            End If
        Next
    End Sub
```

Die Methode `FwdToNextItem` erhöht den privaten Indexzeiger:

```
    Public Function FwdToNextItem() As Boolean Implements _
        SOM.IFwdCursor.FwdToNextItem

        If m_CurrentIndex < m_UpperBound - 1 Then
            m_CurrentIndex = m_CurrentIndex + 1
            FwdToNextItem = True
        Else : FwdToNextItem = False
        End If
    End Function
```

Die Methode `IniFwdCursor` setzt den privaten Indexzeiger zurück:

```
    Public Sub IniFwdCursor() Implements SOM.IFwdCursor.IniFwdCursor
        m_CurrentIndex = -1
    End Sub
```

Diesem Schema folgend sind alle weitern Containerklassen der SOM Implementierung ausgeprägt. Um einmal die Vorteile einer Schnittstellenvererbung zu verdeutlichen, arbeitet die Containerklasse LogicalNodes intern nicht, wie oben dargestellt, mit einem Array, sondern einer so genannten HashTable.

```
    Private m_HashTabel As New Hashtable()
    Private m_Enumerator As IDictionaryEnumerator
```

```
    Public Sub AddLogicalNode(ByVal LogicalNode As SOM.ILogicalNode) _
        Implements SOM.ILogicalNodes.AddLogicalNode

        m_HashTabel.Add(LogicalNode.LogicalNodeName, LogicalNode)
    End Sub
```

```
    Public ReadOnly Property Count() As Long Implements _
        SOM.ILogicalNodes.Count

        Get
            Count = m_HashTabel.Count
        End Get
    End Property
```

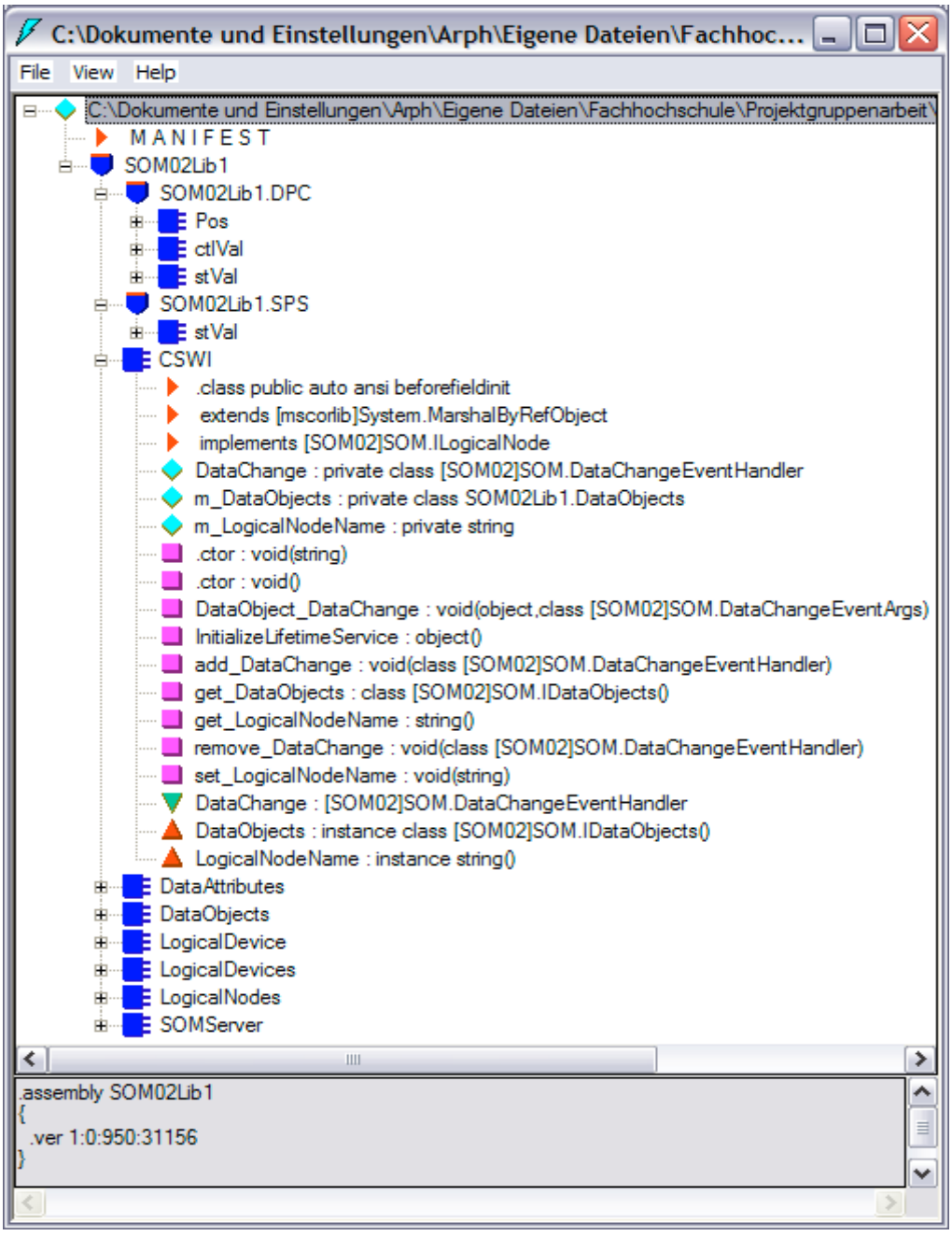
```
    Public Function GetNamedLogicalNode(ByVal LogicalNodeName As String) _
        As SOM.ILogicalNode Implements OM.ILogicalNodes.GetNamedLogicalNode

        GetNamedLogicalNode = m_HashTabel.Item(LogicalNodeName)
    End Function

    Public Function GetCurrentLogicalNode() As SOM.ILogicalNode _
        Implements SOM.ILogicalNodes.GetCurrentLogicalNode

        GetCurrentLogicalNode = m_Enumerator.Value
    End Function

    Public Sub RemoveItem(ByVal SOMObjectName As String) _
        Implements SOM.ILogicalNodes.RemoveItem

        m_HashTabel.Remove(SOMObjectName)
    End Sub

    Public Function FwdToNextItem() As Boolean _
        Implements SOM.IFwdCursor.FwdToNextItem

        FwdToNextItem = m_Enumerator.MoveNext()
    End Function


    Public Sub IniFwdCursor() Implements SOM.IFwdCursor.IniFwdCursor
        m_Enumerator = m_HashTabel.GetEnumerator
    End Sub
```

Obwohl also die Klasse LogicalDevices intern gänzlich anders arbeitet als die Klasse LogicalNodes weisen sie nach außen hin gleiches Verhalten auf. Es wird deutlich, dass durch eine Schnittstellenvererbung dem Entwickler die Freiheit überlassen wird, ein ihm am geeigneten erscheinendes internes Konzept der Klasse zu implementieren. Da das .NET-Framework mit seiner Klassenbibliothek bereits grundlegende Containerklassen, wie etwa die HashTable zur Verfügung stellt, wird deutlich, dass bei der Verwendung solcher Bausteine der Implementierungsaufwand erheblich verringert werden kann. Im Gegensatz zu der Klasse LogicalDevices muss hier intern keine separate Variable für die Arrayobergrenze verwaltet werden, und auch die Dynamisierung des Arrays selber ist nicht mehr notwendig. Die HashTable verfügt von sich aus über Methoden und Eigenschaften, wie sie für Containerklassen notwendig sind. Die Klasse LogicalNodes bedient sich somit der vom .NET-Framework mit gelieferten HashTable Funktionalität. Man spricht in solch einem Fall von Kapselung. Die HashTable wird von der Klasse LogicalNodes gekapselt.

# 4. Implementierung einer Server/Client-Anwendung auf Grundlage der SOM Schnittstelle

## 4.1 Allgemeines

An Hand der folgenden Server/Client Anwendung, soll die Kommunikation mittels des .NET Remothing-Frameworks, anhand einer beispielhaften Implementierung der SOM Schnittstelle verdeutlicht werden. Die gesamte verteilte Server/Client-Anwendung besteht im wesentlichen aus drei einzelnen Softwarekomponenten. Zum einen aus einer .NET DLL (SOM02Lib1.dll), welche die einzelnen Klassen, die die SOM Schnittstelle implementieren, beinhaltet. Dieses sind die gesamten Containerklassen, die die SOM Schnittstelle fordert, sowie einige beispielhaft implementierte Klassen, die die Schnittstelle `ILogicalNode`, `IDataObject` und einige Klassen, die die Schnittstelle `IDataAttribut` implementieren.



Mit Hilfe des .NET Disassemblers **ildasm.exe** ist es möglich den Inhalt einer .NET Codekomponente zu betrachten. Im obigen Bild ist die .NET-DLL SOM02Lib1.dll bzw. das gleichnamige Assembly dargestellt. Im übergeordneten Namespace SOM02Lib1 befinden sich zwei weiter Namespaces DPC (double point controllable) und SPS (singel point status). Der Namespace DPC beinhalten drei Klassen die alle die SOM Schnittstelle IDataObject implementieren. Ebenso implementiert die Klasse stVal im Namespace SPS die Schnittstelle IDataObject des SOM's. Die vollständig expandiert dargestellt Klasse CSWI implementiert die Schnittstelle ILogicalNode. Des Weitern enthält das Assembly alle, für das SOM Objektmodell notwendigen Containerklassen: DataAttributes, DAtaObjects, LogicalDevices usw. Damit alle Klassen zur Kommunikation mittels Remoting imstande sind und somit in verteilten Anwendungen eingesetzt werden können, sind sie alle von der Klasse MarshalByRefObject abgeleitet. Dies ist auch im Disassembler durch das Schlüsselwort extends zu erkennen.

Um eine remotingfähige Applikation zu realisieren, müssen die entsprechenden Objekte im .NET Remoting zur Laufzeit angemeldet werden. Bei dem hier vorgestellten Server/Client-System müssen jedoch nicht alle Objekte im Remoting angemeldet werden, sondern auf Grund des hierarchischen Objektmodells des SOM's genügt es, das Wurzelobjekt der Objekthierarchie, d.h. ein Objekt der Klasse SOMServer, die die Schnittstelle IServer implementiert, im Remoting anzumelden. Wie bereits aus dem Kapitel „Das Remotingframework" hervorgeht, benötigt ein Objekt, das fernbedienbar gemacht werden soll eine so genannte Hostanwendung. Diese ist ein ausführbares Programm, welches das Objekt im .NET Remoting anmeldet. Da das Wurzelelement SOMServer jedoch in einer DLL und nicht in einer ausführbaren Programmeinheit (EXE) implementiert ist, wird eine weitere Softwarekomponente benötigt.

## 4.2 Die Hostanwendung (Server)

In der vorliegenden Implementierung handelt es sich um eine .NET Consolen Anwendung. Wird dieses Programm gestartet, so registriert diese zunächst einen Kommunikationskanal, über welche später die Nachrichten zwischen Server und Client ausgetauscht werden.

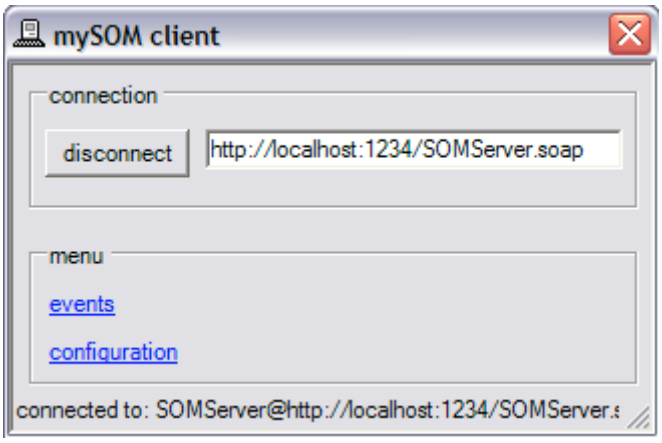
```
Imports System.Runtime.Remoting
Imports System.Runtime.Remoting.Channels
Imports System.Runtime.Remoting.Channels.http

Module Module1
    Sub Main()
        Dim SOMServer As SOM02Lib1.SOMServer

        ChannelServices.RegisterChannel(New HttpChannel(1234))
        SOMServer = New SOM02Lib1.SOMServer()
        SOMServer.LogicalDevices.AddNewLogicalDevice("MyDefLD")

        SOMServer.LogicalDevices.GetNamedLogicalDevice("MyDefLD"). _
                        LogicalNodes.AddNewLogicalNode _
                        ("SOM02Lib1.dll", "SOM02Lib1.CSWI", "CSWI1")

        SOMServer.LogicalDevices.GetNamedLogicalDevice("MyDefLD"). _
                LogicalNodes.GetNamedLogicalNode("CSWI1"). _
                DataObjects.AddNewDataObject _
                ("SOM02Lib1.dll", "SOM02Lib1.DPC.Pos", "Pos1")

        SOMServer.LogicalDevices.GetNamedLogicalDevice("MyDefLD"). _
                LogicalNodes.GetNamedLogicalNode("CSWI1"). _
                DataObjects.GetNamedDataObject("Pos1"). _
                DataAttributes.AddNewDataAttribute _
                ("SOM02Lib1.dll", "SOM02Lib1.DPC.stVal", "stVal")
```

```
        SOMServer.LogicalDevices.GetNamedLogicalDevice("MyDefLD"). _
                LogicalNodes.GetNamedLogicalNode("CSWI1"). _
                DataObjects.GetNamedDataObject("Pos1"). _
                DataAttributes.AddNewDataAttribute _
               ("SOM02Lib1.dll", "SOM02Lib1.DPC.ctlVal", "ctlVal")

        SOMServer.LogicalDevices.GetNamedLogicalDevice("MyDefLD"). _
                LogicalNodes.GetNamedLogicalNode("CSWI1"). _
                DataObjects.GetNamedDataObject("Pos1"). _
                DataAttributes.AddNewDataAttribute _
                ("SOM02Lib1.dll", "SOM02Lib1.SPS.stVal", "Toggle")

        RemotingServices.Marshal(SOMServer, "SOMServer.soap")

        System.Console.WriteLine("*********************************")
        System.Console.WriteLine("  SOMServer - Startup: {0}", _
                System.DateTime.Now.ToString())
        System.Console.WriteLine("*********************************")
        System.Console.WriteLine("")
        System.Console.ReadLine()
    End Sub
End Module
```

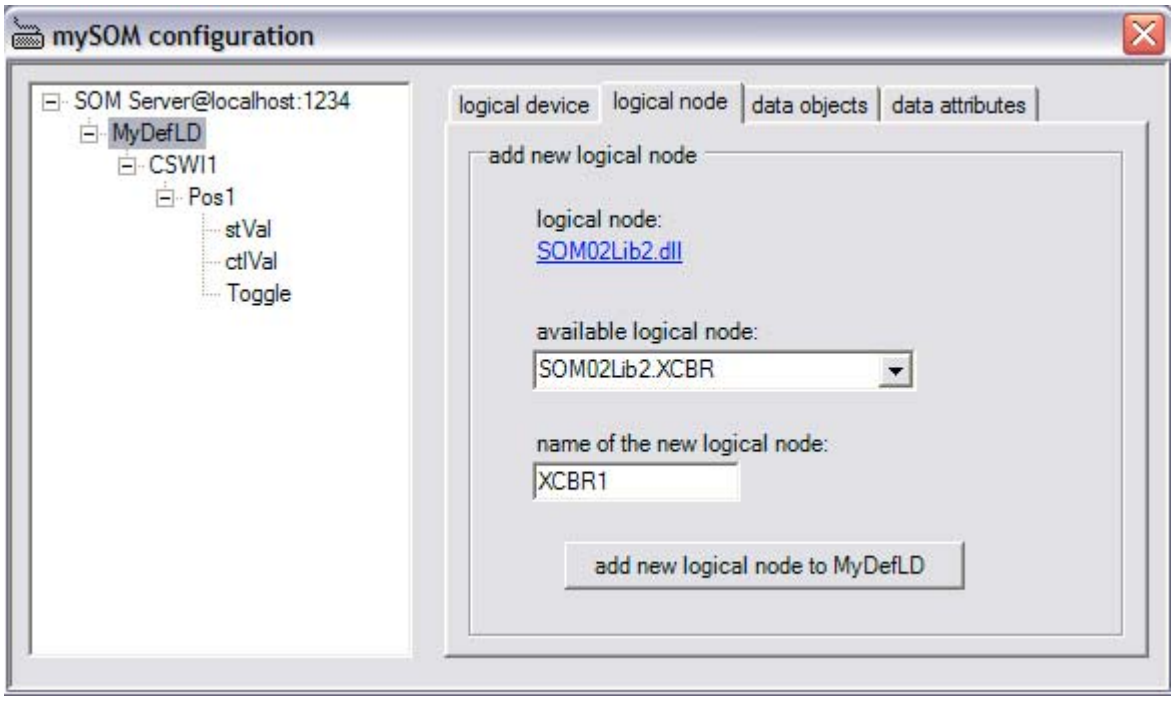
Für den Kanal wird der Port 1234 und als Kommunikationsprotokoll das HTTP-Protokoll gewählt. Mit diesem Schritt ist für die Hostanwendung ein Kanal registriert und reserviert. Er kann nun von keinem weiteren Programm registriert werden und steht somit exklusiv für diese Anwendung zur Verfügung. Im nächsten Schritt wird ein Objekt der Klasse SOMServer instanziert. Des Weiteren wird eine beispielhafte Konfiguration des SOMServers geladen. Hierfür werden im SOMServer ein neues Objekt LogicalDevice mit dem Namen „myDefLD", diesem wiederum ein LogicalNode mit dem Namen „CSWI1" hinzugefügt. Desweitern dem CSWI1 ein DataObjeckt mit dem Namen „Pos1" und diesem wiederum ein DataAttríbute mit dem Namen stVal, sowie ctVal und Toggle hinzugefügt. Nachdem das SOMServer Objekt beispielhaft konfiguriert ist, wird es mit einer ensprechenden URI (unified resource identifier) im .NET Remoting angemeldet. Danach steht es allen Clients, die über den HTTP-Port 1234 auf dieses zugreifen wollen, zur Verfügung. Da ein entsprechend am .NET Remoting angemeldetes Objekt nur solange registriert bleibt, solange die Anwendung, die es registriert hat, ausgeführt wird, wird die Hostanwendung nicht terminiert, sondern wartet mit dem Befehl `System.Console.ReadLine` auf eine Tastatureingabe, um den Server zu beenden.

## 4.2 Der Client

Im Gegensatz zu der relative kleinen Hostanwendung des Servers, welcher als Consolen Anwendung implementiert ist, ist der etwas komplexere Client als .NET Windows Anwendung realisiert. Wird der SOM Client ausgeführt, präsentiert er sich zunächst in einem einfachen Dialog, welcher dazu auffordert den Client mit einem sich bereits in Ausführung befindendem SOMServer zu verbinden. Dabei ist in der TextBox der entsprechende Rechnername, auf dem der SOMServer ausgeführt wird, inklusive der Portnummer des Kanals, an welchem der SOMServer registriert ist, anzugeben.

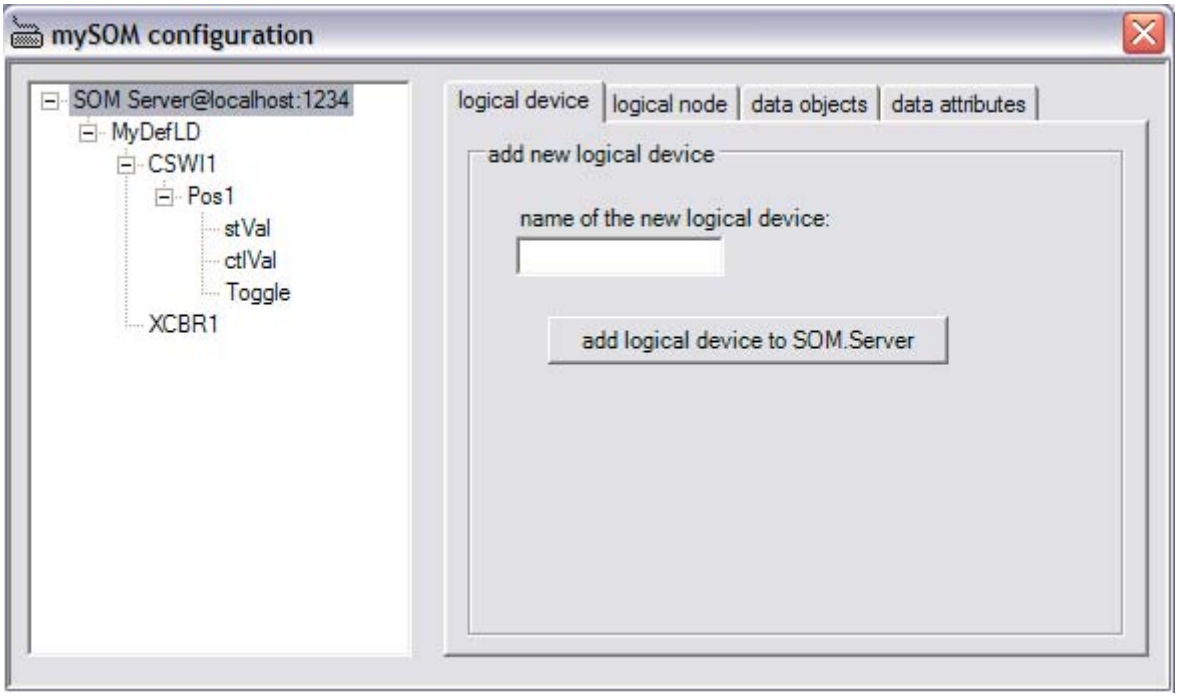



Über den Button „connect to" kann nun die Verbindung zu dem SOMServer hergestellt werden. Wurde die Verbindung erfolgreich hergestellt so stehen weitere Menüpunkte den SOMServer betreffend zur Verfügung. Der über den Menüpunkt „configuration" aufzuschaltende Dialog zeigt die gesamte Modellkonfiguration des SOMServers mittels eines TreeViews an. Des Weiteren bietet dieser Dialog die Möglichkeit, neue Objekte in die bestehende Objektmodellkonfiguration einzupflegen. So kann z.B. unter dem bestehenden LogicalDevice mit dem Namen „myDefLD" ein weiterer logischer Knoten hinzugefügt werden. Über den LinkButton „logical node" kann die entsprechende DLL ausgewählt werden, die den hinzuzufügenden logischen Knoten enthält. Wurde eine DLL ausgewählt die eine oder mehrere Klassen, welche die Schnittstelle ILogicalNode implementieren, ausgewählt, so erscheinen in der ComboBox die Namen dieser Klassen.



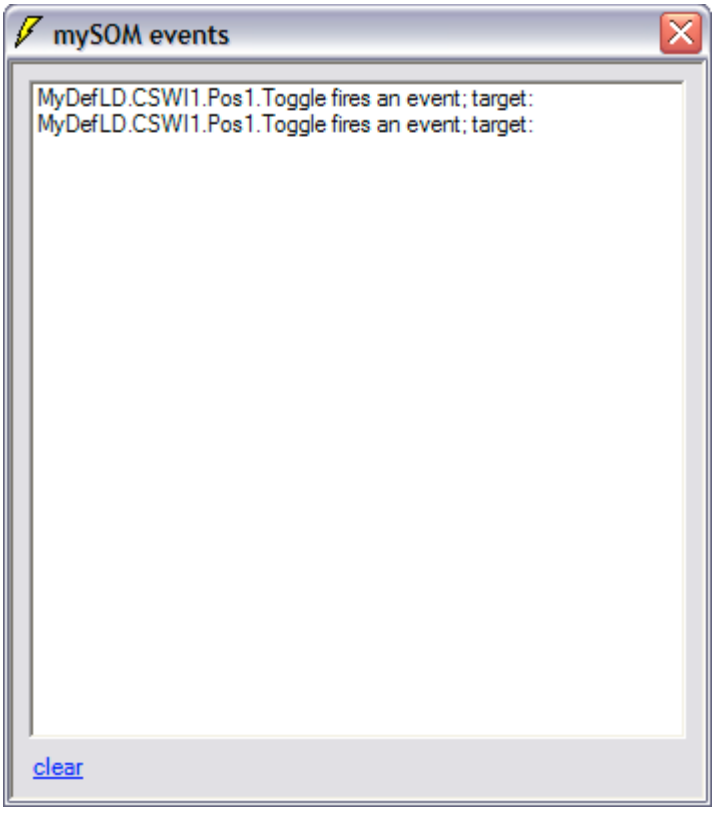

In diesem Fall enthält die DLL SOM02Lib2.dll eine Klasse mit Namen XCBR, die die Schnittstelle ILogicalNode implementiert. Dem neu hinzuzufügen Objekt kann ein beliebiger Name vergeben werden.

Mit dem Button „add logical node“ wird der neue logische Konten instanziert und in die Objektmodellkonfiguration des laufenden SOMServers hinzugefügt.

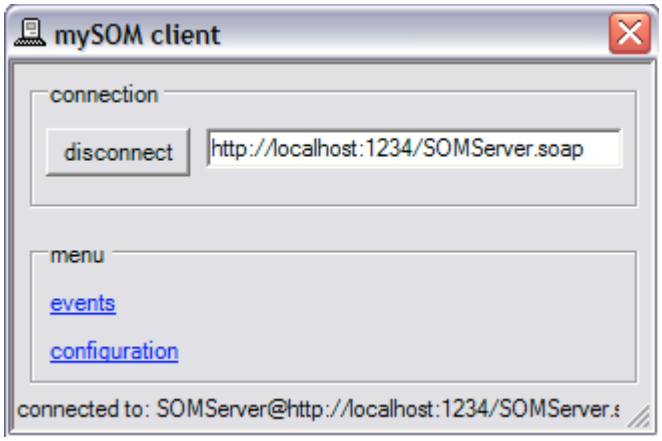



Um zu einem bestehenden logischen Knoten weitere DataObjects, bzw. zu bestehenden DataObjects weitere DataAttributes hinzuzufügen, kann wie beschreiben unter der jeweiligen Registerkarte vorgegangen werden.

Über den Menüeintrag „events“ können die auftretenden Ereignisse, die von einem existierenden Objekt innerhalb eines SOMServers gefeuert werden, empfangen und dargestellt werden.



## 4.4 Die Codierung




Mit dem Klick auf den Button „connect to" wird die Methode

```
Connect(ByVal SOMServerURIHostname As String)
```

aufgerufen. Als Parameter empfängt diese den kompletten Aufruf-URI des SOMServers (Inhalt der TextBox). Mittels der Methode `GetObject` der statischen Klasse Activator wird nun über die mit übergebenen URI ein Verweis auf den laufenden SOMServer erzeugt und in der Variable `g_server` gespeichert. Mit der Ausführung dieses Codes ist die Verbindung zum SOMServer hergestellt.

```
Public Sub Connect(ByVal SOMServerURI As String)
   g_server = Activator.GetObject(GetType(SOM.IServer), SOMServerURI)
.
.
.
End Sub
```

Der SOMServer kann jetzt im Client wie ein lokal existierendes Objekt benutz werden. So z.B. ein TreeView gefüllt werden der die gesamte Konfiguration des SOMServers darstellt.
Die Methode `FillTree` realisiert genau diese. Als erstes wird der eventuell bereits gefüllte TreeView gelöscht. Weiter wird durch die gesamte SOMServer-Konfiguration durchiteriert. Dem Wurzelelement des TreeViews wird die gesamte Aufruf-URI des SOMServers zu gewiesen und damit angezeigt. Danach wird der Iterierungscourser des Container LogicalDevices des SOMServers initialisiert, sprich auf das erste sich im Container befindende Element gesetzt und mit einer `While`-Schleife anschließend mit der Methode `FwdToNextItem()` durch den gesamten Container durchiteriert. Bei jedem Schleifendurchlauf wird der Name des aktuellen LogicalDevice-Objekt einem neuen TreeView-Knoten übergeben und dem TreeView hinzugefügt. Um das hierarchische Objektmodell des SOMServers im TreeView ebenfalls hierarchisch darzustellen umschließt die äußere `While`-Schleife, die den Container LogicalDevices durchläuft weiter `While`-Schleifen, die die jeweils untergeordneten Container (LogicalNodes, DataObjects,usw.) durchlaufen.

```
    Private Sub FillTree()
        Dim LDs As SOM.ILogicalDevices
        Dim LNs As SOM.ILogicalNodes
        Dim DOs As SOM.IDataObjects
        Dim DAs As SOM.IDataAttributes
        Dim Ld As SOM.ILogicalDevice
        Dim ln As SOM.ILogicalNode
        Dim Dob As SOM.IDataObject
        Dim da As SOM.IDataAttribute

        Dim RootNode As TreeNode
        Dim LDNode As TreeNode
        Dim LNNode As TreeNode
        Dim DONode As TreeNode
        Dim DANode As TreeNode

        Me.TreeView1.Nodes.Clear()
```

```
        If Not g_server Is Nothing Then
            LDs = g_server.LogicalDevices
            LDs.IniFwdCursor()
            RootNode = Me.TreeView1.Nodes.Add("SOM Server@" & Me.m_host)
            RootNode.Tag = SERVER
            While (LDs.FwdToNextItem)
                Ld = LDs.GetCurrentLogicalDevice
                LDNode = RootNode.Nodes.Add(Ld.LogicalDeviceName)
                LDNode.Tag = LOGICAL_DEVICE

                LNs = Ld.LogicalNodes
                LNs.IniFwdCursor()
                While (LNs.FwdToNextItem)
                    ln = LNs.GetCurrentLogicalNode
                    LNNode = LDNode.Nodes.Add(ln.LogicalNodeName)
                    LNNode.Tag = LOGICAL_NODE

                    DOs = ln.DataObjects
                    DOs.IniFwdCursor()
                    While (DOs.FwdToNextItem)
                        Dob = DOs.GetCurrentDataObject
                        DONode = LNNode.Nodes.Add(Dob.DataObjectName)
                        DONode.Tag = DATA_OBJECT

                        DAs = Dob.DataAttributes
                        DAs.IniFwdCursor()
                        While (DAs.FwdToNextItem)
                            da = DAs.GetCurrentDataAttribute
                            DANode = DONode.Nodes.Add(da.DataAttributeName)
                            DANode.Tag = DATA_ATTRIBUT


                        End While
                    End While
                End While
            End While
            RootNode.Expand()
        End If
    End Sub
```

# 5. Implementierung zweier beispielhaft ausgewählter logischer Knoten XCBR und CSWI

## 5.1 Beschreibung der logischen Knoten

Bei den beispielhaft ausgewählten logischen Knoten handelt es sich mit dem CSWI um eine logische Steuereinheit des Leistungsschalters XCBR. Ziel der Anwendung ist es, die beiden logischen Knoten, die jeweils in einem eigenen Hostprozess in einem SOMServer laufen, über das .NET Remoting miteinander kommunizieren zu lassen. Die beiden logischen Knoten CSWI und XCBR sind gemeinsam in der .NET DLL mySOMLogicalNodes.dll implementiert. Des Weiteren enthält diese DLL DataObject- sowie DataAttribute-Klassen für die logischen Knoten XCBR und CSWI.

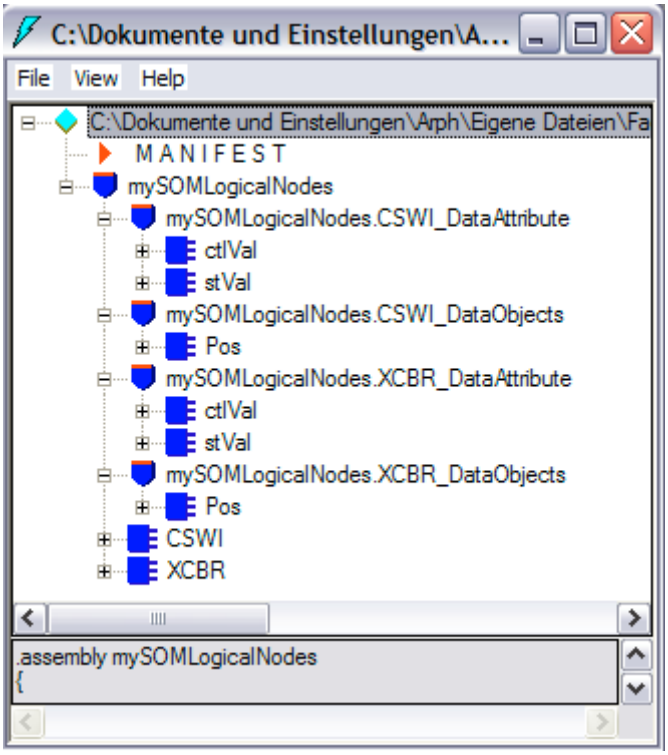


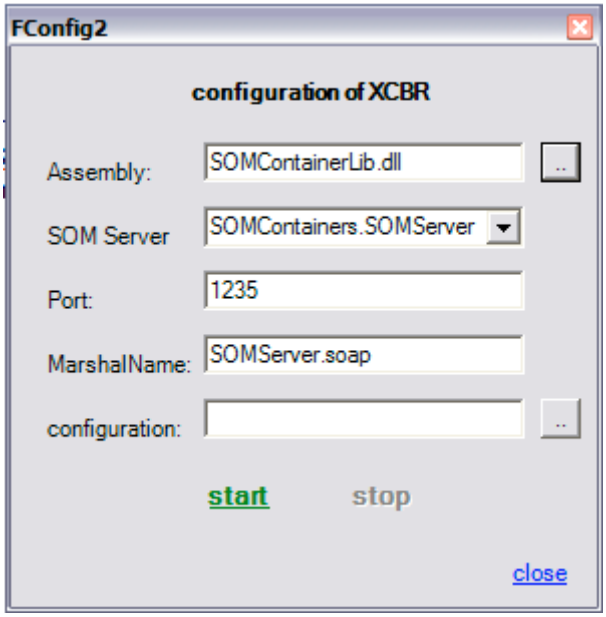

Abb. 22: Ansicht der DLL mySOMLogicalNodes.dll im .NET Disassembler

Bei der Implementierung der logischen Knoten XCBR und CSWI ist die vollständige Unterstützung der Norm IEC61850 nicht gegeben. Es sind lediglich einzelne DataObjects und DataAttributes implementiert, um den Einsatz und die Verwendung der verwendeten .NET Softwaretechnologie im Bezug auf das SOM Objektmodell zu verdeutlichen.

Der logische Knoten XCBR, der einen Leistungsschalter repräsentiert, wird in der .NET Windows-Anwendung myVirtualXCBR innerhalb eines SOMServers gehostet. Nachdem die Anwendung myVirtualXCBR gestartet wurde, präsentiert sie sich in folgendem Fensterdialog, in dessem linken oberen Bereich die grundlegenden Konfigurationen für den Betrieb der Anwendung einzustellen sind.


Abb. 23: Die .NET Windows-Anwendung myVirtaulXCBR

Zunächst muss der XCBR als logischer Knoten eines SOMServers im .NET Remoting angemeldet werden. Dies geschieht über den Menüeintrag server configuration. In dem neu aufgeblendeten Fenster muss zunächst eine .NET DLL, sprich ein Assembly ausgewählt werden, welches eine Klasse, die die Schnittstelle IServer des SOM's implementiert enthält. Zu diesem Zweck steht die DLL SOMContainerLib.dll zur Verfügung. Nachdem diese ausgewählt wurde erscheint in der ComboBox SOMServer eine Liste aller im entsprechend ausgewählten Assembly enthaltenen Klassennamen, die die Schnittestelle IServer implementieren. In dem Fall der DLL SOMContainerLib.dll ist es lediglich eine Klasse, die Klasse SOMServer. Des Weiteren ist in diesem Dialog der Port einzustellen, über welchen die Kommunikation erfolgen soll. Hierfür ist standardmäßig die Portnummer 1235 angegeben. Weiterhin kann der Name der Aufruf-URI des Servers eingestellt werden. Hierfür ist defaultmäßig der Wert SOMServer.saop eingestellt. Über den LinkButton „start" wird der SOMServer gestartet, sprich am .NET Remoting angemeldet. Dieser wird beim Start so konfiguriert, dass bereits ein LogicalDevice, sowie der logische Knoten XCBR, ein DataObject Pos und zwei DataAttributes stVal (status value) und ctlVal(control value) vorhanden sind. Diese Konfiguration des SOMServers kann über den Menüpunkt own configuration angezeigt werden. Des Weiteren ermöglicht der Dialog, ähnlich wie bei der zuvor vorgestellten Server/Client Implementierung eine Erweiterung der bestehenden Server Objektmodellkonfiguration, durch hinzufügen neue Objekte. Mit



einem SOMClient, wie etwa dem zuvor vorgestellten SOMCLient.exe, kann jetzt zu dem laufenden SOMServer, der den XCBR hostet, Verbindung aufgenommen werden. Um jetzt jedoch eine Verbindung zwischen dem Leistungsschalter XCBR und seinem Steuergerät CSWI herzustellen, muss zunächst auch der CSWI genauso wie der XCBR in einem SOMServer ausgeführt werden. Der logische Knoten CSWI wird innerhalb der .NET Windows-Anwendung myVirtualCSWI in einem weiteren SOMServer ausgeführt. Die Anwendung myVirtualCSWI stellt sich genauso dar, wie die beschriebene Anwendung myVirtualXCBR. Über den Menüeintrag server configuration wird der Dialog zur Konfiguration des SOMServers, der den CSWI hostet, eingeblendet. Genauso wie bei dem XCBR muss hier zunächst eine DLL ausgewählt werden, die einen SOMServer beinhaltet, auch hier steht die DLL SOMContainerLib.dll zur Verfügung. Des weiteren muss auch bei diesem Server ein Kommunikationsport gewählt werden. Dieser darf natürlich der gleiche sein wie zuvor bei dem SOMServer, der den XCBR beinhaltet. Eingestellt ist hier der Port 1234. Mit dem default eingestellten MarshalNamen des Servers SOMServer.soap kann der SOMServer gestartet werden. Ebenso wie bei dem SOMServer, der den XCBR beinhaltet, wird der SOMServer des CSWI beim Starten mit einem LogicalDevice, dem CSWI als logischen Knoten, einem DataObject Pos, sowie zwei DataAttributes stVal und ctlVal gestartet. Äquivalent zur Anwendung des XCBR's kann über den Menüeintrag own configuration die Konfiguration des SOMServers des CSWI angezeigt werden.

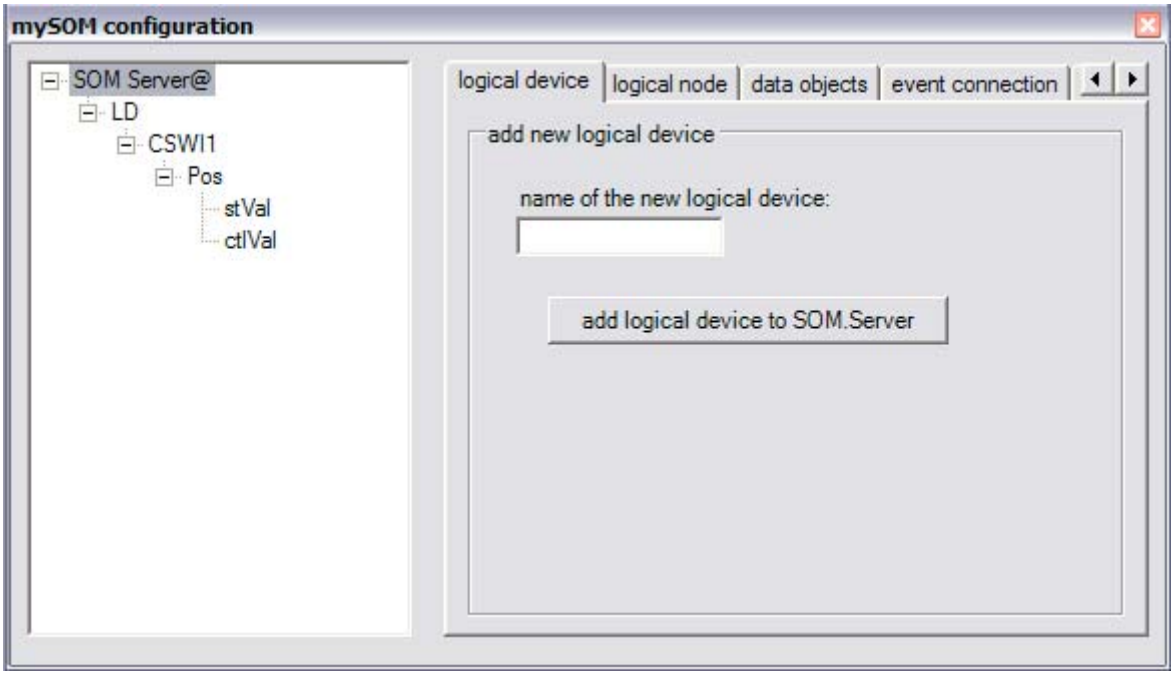
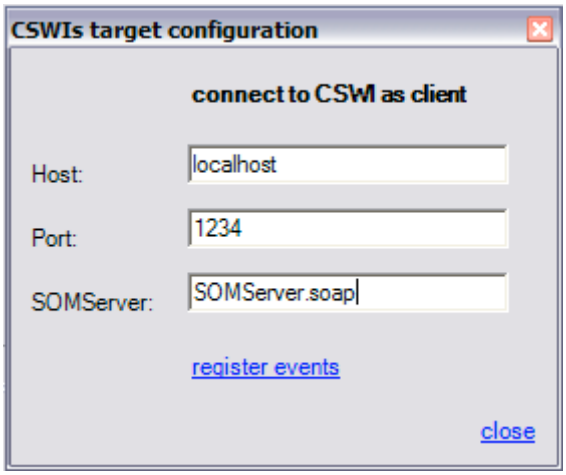


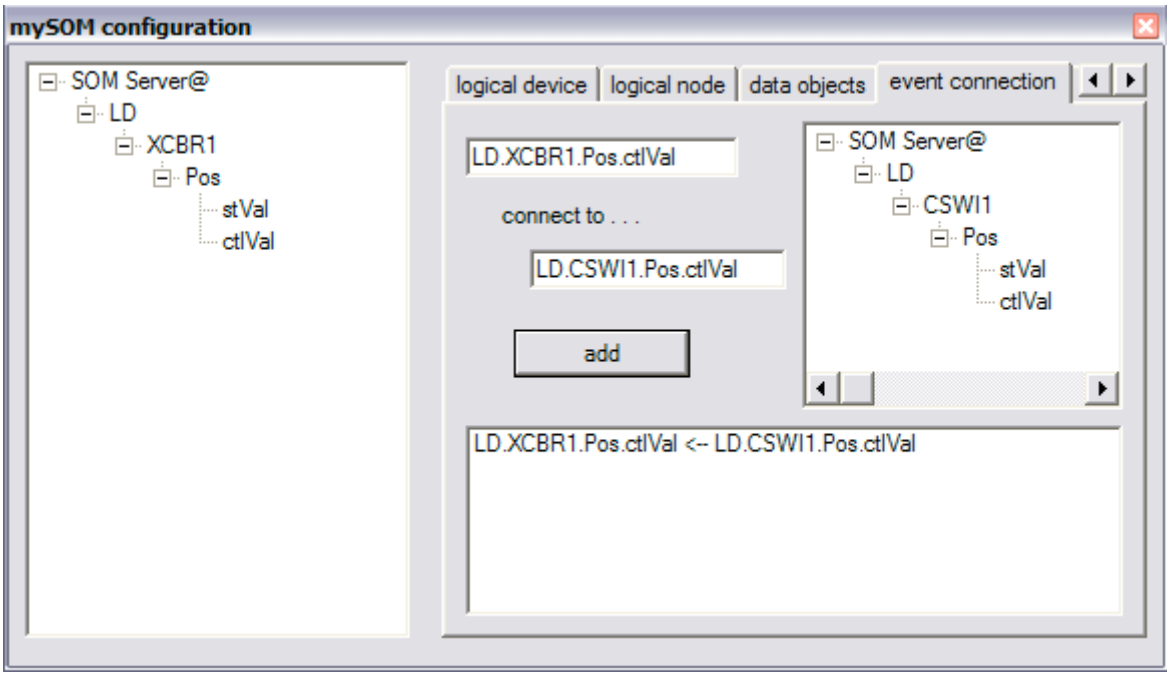

Abb. 24: Die Startkonfiguration des SOMServers, der den CSWI enthält.

Zum jetzigen Zeitpunkt laufen zwei SOMServer parallel, lediglich auf unterschiedlichen Ports. Jedoch ist jetzt noch keine Kommunikationsverbindung zwischen beiden Servern hergestellt. Um eine Verbindung zwischen beiden Servern herzustellen, muss eine der beiden Hostanwendungen, z.B. die des XCBR's, als Client des anderen SOMServers konfiguriert werden. Über den Menüeintrag client configuration der Hostanwendung des XCBR's wird der Dialog der Clientkonfiguration aufgeschaltet.

In der TextBox Host ist der Rechnername des Systems anzugeben, auf welchem der SOMServer des CSWI läuft. In diesem Fall, da beide Server auf der gleichen Maschine laufen, ist der Name des Rechners „localhost". In der TextBox Port ist die Portnummer, an welchem der SOMServer des CSWI's angemeldet ist, anzugeben. In der gewählten Konfiguration ist dies der Port 1234. In der TextBox SOMServer ist der MarshalName des SOMServers einzustellen. In diesem Falle SOMServer.soap. Über den LinkButton register events werden vom CSWI gefeuerte Ereignisse im Client registriert. Jetzt arbeitet die Hostanwendung des SOMServers des XCBR gleichzeitig als Client des SOMServers des CSWI's und ist in der Lage, auftretende Ereignisse des SOMservers des CSWI's zu empfangen und auszuwerten.




Um nun das Verhalten des XCBR's auf entsprechende Ereignisse des CSWI's einzustellen, ist über den Menüeintrag own configuration der entsprechende Dialog einzublenden. Eine mögliche Konfiguration könnte so aussehen, dass wenn der CSWI ein Ereignis feuert, dass angibt das sein DataAttribute ctlVal geändert wurde, der entsprechende Wert des DataAttribute ctlVal des XCBR's mit dem des CSWI gleichgesetzt wird. Dafür sind die beiden ctlVal's des CSWI's und des XCBR's zu verbinden. Um dies zu erreichen, wird das ctlVal DataAttribute des XCBR's im TreeView selektiert. Daraufhin erscheint in der rechten Hälfte ein weiterer TreeView, der die Objektkonfiguration des SOMServers des CSWI's darstellt.




Zusätzlich wird in der oberen TextBox der vollständige Name des DataAttributes ctlVal des XCBR's eingeblendet. Jetzt muss der ctlVal des CSWI's selektiert werden, dieser wird in der unteren TextBox angezeigt. Um die beiden DataAttributes miteinander zu verbinden wird die Verbindung mit dem Button „add" bestädigt und in der ListBox angezeigt. Dies bedeutet, dass jetzt eine Änderung von ctlVal des CSWI's eine Änderung des ctlVal des XCBR's bewirkt. Die Hostanwendung des CSWI's ist so implementiert, dass der ctlVal des CSWI's über die beiden Buttons „on" und „off" gesetzt werden kann. Auf

Grund der bestehenden Verbindung zwischen XCBR und CSWI wird nun eine Änderung des ctlVal des CSWI im XCBR sichtbar gemacht. Der logische Knoten XCBR ist so implementiert, dass wenn sein DataAttribute ctlVal gesetzt wird daraufhin sein stVal (status value) auf der Wert seines ctlval gesetzt wird. Weiterhin ist die Hostanwendung des SOMServers des XCBR's so implementiert. dass der Wert des DataAttribute stVal des XCBR's die graphische Anzeige eines Leistungsschalters steuert. D.h. ist der stVal des XCBR „false" ist der Schalter geschlossen und umgekehrt. Zusätzlich kann über die Nahsteuerung über die beiden Buttons „on" und „off" der ctlVal des XCBR direkt gesetzt werden. Damit die Steuereinheit des XCBR's, der CSWI erfährt, ob die Schalthandlung ausgeführt wurde, sprich eine Rückmeldung vom Leistungsschalter XCBR zu erhalten, kann die Konfiguration der Anwendung so erweitert werden, dass zusätzlich die Hostanwendung des CSWI's sich als Client des SOMServers der den XCBR's beinhaltet, verhält. Dafür ist der entsprechende Dialog über den Menüpunkt client configuration der Hostanwendung des CSWI's aufzurufen. Hier muss analog wie bei der Hostanwendung des XCBR's, der entsprechende Rechnernamens des SOMServers des XCBR's, d.h. in diesem Fall „localhost" gewählt werden, sowie die Portnummer, auf welchem der SOMServer des XCBR's registriert ist, hier 1235. Des Weiteren muss auch der MarshalName des SOMServers angegeben werden, hier also „SOMServer.soap". Wie bei der Hostanwendung des XCBR's werden die eintreffenden Ereignisse über den LinkButton „register events" registriert.
Um jetzt auf ein Ereignis, wie z.B. eine Änderung des ctlVal des XCBR's, zu reagieren, muss unter dem Menüpunkt own configuration im linken TreeView der die Objektkonfigurations des SOMServers des CSWI's anzeigt der stVal mit dem stVal des XCBR's verbunden werden. Wird in der jetzt bestehenden Konfiguration der ctlVal des CSWI's mittels der Buttons „on" oder „off" geändert, so empfängt zunächst der XCBR das entsprechende Ereignis und setzt seinen ctlVal, woraufhin auf Grund der Implementierung der stVal des XCBR's gesetzt wird und ein Ereignis feuert, das wiederum die Hostanwendung des CSWI empfängt und daraufhin den stVal des CSWI's setzt.
Auch die Hostanwendung des CSWI's ist so implementiert, dass der Wert des stVal des CSWI die graphische Anzeige der Schalterstellung steuert.

In der folgenden Graphik ist der vereinfachte schematische Kommunikationsweg dargestellt.
Ganz rechts erfolgt durch einen Methodenaufruf die Änderung des ctlVal des CSWI's.

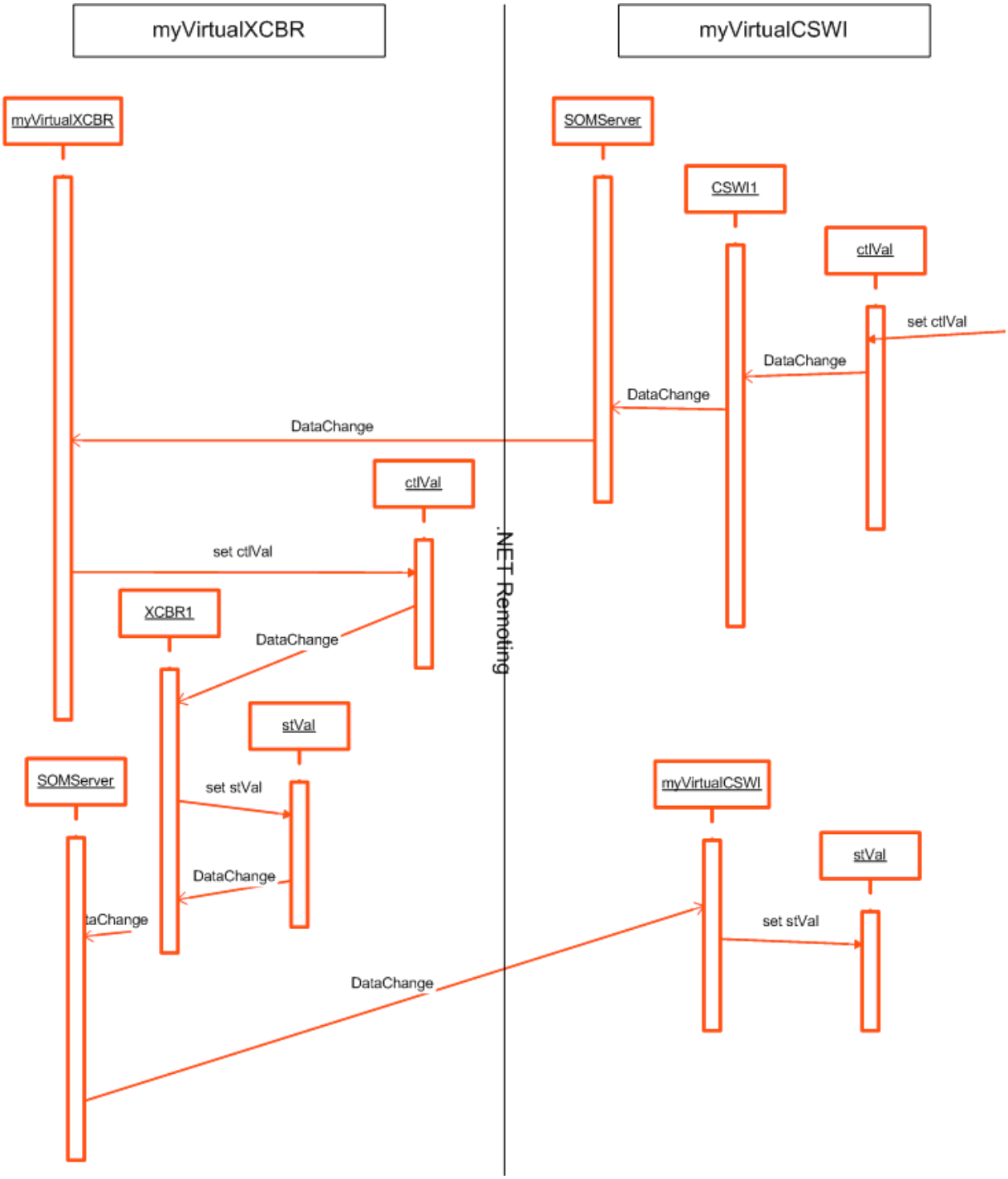



Abb. 25: Ereignispfad, ausgelöst durch den ctlVal des CSWI's

## 5.2 Die Codierung

Nach der Einstellung der Parameter des SOMServers und der Bestätigung durch den LinkButton „start" wird aus dem angegebenen Assembly, d.h. aus der DLL SOMContainerLib.dll ein Objekt der Klasse SOMServer instanziert und der Verweis darauf in der Variablen g_server gehalten. Danach wird die Methode `LoadSOMServerConfig()`, die im Modul MGlobal implementiert ist, aufgerufen:

```
Dim Ass As System.Reflection.Assembly
Ass = Reflection.Assembly.LoadFrom(Me.OpenFileDialog1.FileName)
g_server = Ass.CreateInstance(Me.ComboBox1.Text)
LoadSOMServerConfig()
    If StartServer(Me.txtPort.Text, Me.txtSOMServer.Text) Then
        Me.LinkLabel1.Enabled = False
        Me.LinkLabel2.Enabled = True
        Me.Hide()
    End If
```

Hier wird lediglich die default Objektkonfiguration des SOMServers geladen.

```
Public Sub LoadSOMServerConfig()
    Dim ld As SOM.ILogicalDevice
    Dim ln As SOM.ILogicalNode
    Dim dob As SOM.IDataObject
    Dim da As SOM.IDataAttribute

    If Not g_server Is Nothing Then
        g_server.LogicalDevices.AddNewLogicalDevice("LD")
        ld = g_server.LogicalDevices.GetNamedLogicalDevice("LD")
        ln = ld.LogicalNodes.AddNewLogicalNode("mySOMLogicalNodes.dll",
             "mySOMLogicalNodes.XCBR", "XCBR1")
        dob = ln.DataObjects.AddNewDataObject("mySOMLogicalNodes.dll",
             "mySOMLogicalNodes.XCBR_DataObjects.Pos", "Pos")
        dob.DataAttributes.AddNewDataAttribute("mySOMLogicalNodes.dll",
             "mySOMLogicalNodes.XCBR_DataAttribute.stVal", "stVal")
        dob.DataAttributes.AddNewDataAttribute("mySOMLogicalNodes.dll",
             "mySOMLogicalNodes.XCBR_DataAttribute.ctlVal", "ctlVal")
    End If
End Sub
```

Anschließend wird die Methode `StartServer` aufgerufen. Die Methode `StartServer` registriert den SOMServer auf dem entsprechenden Port mit dem übergebenen MarshalNamen.

```
Public Function StartServer(ByVal Port As Integer, ByVal MarshalName As
                                                    String) As Boolean

    Static first As Boolean = True

    MyPort = Port
    MyMarshalName = MarshalName

    m_Channel = New Http.HttpChannel(Port)
    If first Then RemotingServices.Marshal(g_server, MarshalName) :
                                                        first = False
    g_ServerActivity.Text = SERVER_IS_RUNNING
    Return True
End Function
```

Um die Hostanwendung des einen SOMServers gleichzeitig als Client eines anderen SOMSevers zu verwenden wird mit dem Klick auf den LinkButton „register events" des Dialogs des Menüeintrages client configuration die Methode `GetObject` des Objekts `Activator` aufgerufen.

```
        g_remoteserver = Activator.GetObject(GetType(SOM.IServer),
                  "http://" & Me.txtHost.Text & ":" & Me.txtPort.Text & "/"
                  & Me.txtSOMServer.Text)
```

Anschließend wird die Methode `RegisterEvents()` aufgerufen. Diese ist im Modul MGlobal implementiert.

```
Public Sub RegisterEvents()
  SEW = New SOM.ServerEventWrapper()
  d2 = New SOM.DataChangeEventHandler(AddressOf SEW.Server_DataChance)
  AddHandler g_remoteserver.DataChange, d2
End Sub
```

Innerhalb der Methode `RegisterEvents()` wird zunächst das Objekt SEW, welches zuvor außerhalb der Methode mit dem Schlüsselwort `WithEvents` deklariert wurde, neu instanziert. Weiter wird eine neue Instanz eines Objektes der Klasse SOM.DataChangeEventHandler, wobei dem Konstruktorparameter die entsprechende Adresse der Ereignisbehandlungsroutine des Objekts SEW mit übergeben wird. Diese Ereignisbehandlungsroutine des Objekts SEW ist ebenfalls im Modul MGlobal definiert. Mit der Funktion `AddHandler` wird nun dem Ereignis `DataChange` des Objekts `g_remoteserver`, welches den Verweis auf den entfernten SOMServer hält, der zuvor erzeugte Delegat `d2` angehängt. Damit ist der Clint nun in der Lage, Ereignisse des entfernten SOPMServers zu empfangen und in der Ereignisbehandlungsroutine des Objekts SEW zu bearbeiten.

**Verknüpfung von DataAttributes**

Um zwei DataAttributes, z.B. das DataAttribut ctlVal des CSWI mit dem ctlVal des XCBR, zu verbinden, wird in der Behandlungsroutine des Button „add" der vollständige Name des Zielobjektes, sowie der vollständige Name des Quellobjektes in einer Variablen der Struktur `EventConnection` gespeichert und in der Collection `g_EventConnections` abgelegt.

```
Public Structure EventConnection
    Dim Source As String
    Dim Target As String
End Structure

Private Sub btnConnectAttributes_Click(ByVal sender As System.Object,
        ByVal e As System.EventArgs) Handles btnConnectAttributes.Click

    Dim nc As EventConnection
    nc.Target = Me.txtAttribute1.Text
    nc.Source = Me.txtAttribute2.Text
    g_EventConnection.Add(nc)
End Sub
```

Feuert der Server jetzt ein Ereignis wird diese im Client in der Ereignisbehandlungsroutine

```
Private Sub SEW_DataChange(ByVal Sender As Object, ByVal Args As
                    SOM.DataChangeEventArgs) Handles SEW.DataChange
```

empfangen. Aus dem Parameter `Args` lässt sich der genaue Name des Herkunftsobjekts dieses Ereignisses aus dem SOMServer ermitteln. Dieser wird in der Variable `source` gespeichert. Daraufhin wird durch die gesamte `g_EventConnection` durchiteriert und das Source Element jeder EventConnection Variable aus der Collection mit der empfangenen Quelle in der Variable `source` gespeichert, verglichen. Stimmen diese miteinander überein, so wird mit der Methode GetSOMObject aus dem eigenen SOMServer das entsprechende Zielobjekt ermittelt und diesem der Wert des `Args.ctlVal` übergeben.

```
Private Sub SEW_DataChange(ByVal Sender As Object, ByVal Args As
        SOM.DataChangeEventArgs) Handles SEW.DataChange
    Dim source As String
    Dim nc As EventConnection

    source = Args.LogicalDeviceName & "." & Args.LogicalNodeName & "."
              & Args.DataObjectName & "." & Args.DataAttributeName

    For Each nc In g_EventConnection
        If nc.Source = source Then
            Dim o As Object
            o = GetSOMObject(g_server, nc.Target)

            If TypeOf o Is SOM.DPC.IctlVal Then
                Dim ctlVal As SOM.DPC.IctlVal
                ctlVal = o
                ctlVal.DataAttributeValue = Args.ctlVal
            End If
        End If
    Next
End Sub
```

Somit wird der Wert des Quellobjektes dem Zielobjekt zugewiesen.

# 6. Zusammenfassung und Ausblick

Die vorliegende Arbeit teilt sich in zwei Themenbereiche. Im erste Teil werden die Grundlagen der .NET-Technologie der Firma Microsoft dargestellt. Des Weiteren wird eine einführende Darstellung in die .NET-Entwicklungsumgebung Visual Studio .NET gegeben. Im Hinblick auf den zweiten Teil der Arbeit, wird neben den grundlegenden Komponenten des .NET-Frameworks, wie Laufzeitumgebung (CLR), gemeinsames Typsystem (CTS) und Klassenbibliothek, verstärkt auf weitere, spezielle Komponenten des .NET-Frameworks, wie das .NET-Remotingframework und das .NET-Reflectionframework eingegangen. Da die Implementierungen im zweiten Teil der Arbeit verstärkt mit Funktionalitäten aus dem .NET-Remoting- und .NET.Reflectionframework arbeiten, werden deren Aufgabe und Verwendbarkeit hier dargestellt.
Der zweite Teil der Arbeit stellt die Implementierung einer Server/Client-Anwendung innerhalb einer verteilten Architektur, basieren auf der Schnittstellendefinition des SOM's, anhand zweier Anwendungen, dar. Ziel der Anwendungen ist es, auf der Normenreihe IEC 61850 basierend, eine Server/Client-Kommunikation zu realisieren. Die Konformität mit IEC 61850 wird durch die Implementierung der Schnittstellendefinition SOM erreicht.

Die vorliegende Arbeit hat gezeigt, dass die Implementierung einer Server/Client-Kommunikation mittels des .NET-Remotingframeworks und basierend auf der Schnittstellendefinition SOM realisierbar ist. Die vorliegenden Applikationen, die die Server/Client-Kommunikation veranschaulichen, stellen lediglich eine beispielhafte Anwendung dar. Zudem stellen die ausgewählten logische Knoten CSWI und XCBR nur einzelne beispielhaft ausgewählte DataAttributes, wie ctlVal und stVal. Die Anwendungen erheben nicht den Anspruch vollständig mit IEC 61850 konform zu gehen, sondern beschränken sich auf eine kleine eingeschränkte Auswahl an Diensten und Eigenschaften, die es ermöglichen anhand von IEC 61850 eine Server/Client-Kommunikation in verteilten Netzwerken zu realisieren und darzustellen. Die Anwendungen erheben den Anspruch, die Machbarkeit einer solchen Kommunikation, die konform zu IEC 61850 ist, zu zeigen.

Weiterführend und aufbauend auf dieser Arbeit ist es denkbar, weitere logische Knoten zu implementieren um eine umfangreichere Konfiguration eines Stationsleitsystems innerhalb einer verteilten Anwendungsarchitektur zu realisieren.