HANSER

Leseprobe

Walter Doberenz, Thomas Gewinnus

Visual Basic 2015 – Grundlagen, Profiwissen und Rezepte

ISBN (Buch): 978-3-446-44380-8

ISBN (E-Book): 978-3-446-44605-2

Weitere Informationen oder Bestellungen unter

http://www.hanser-fachbuch.de/978-3-446-44380-8

sowie im Buchhandel.

Inhaltsverzeichnis

Teil II: Technologien

Anhang

Download-Kapitel

LINK: http://doko-buch.de

Teil IV: WPF-Anwendungen

Teil V: Windows Forms

Vorwort

Die Zeit, in der Visual Basic-Programmierer meinten, mit ein paar Klicks auf ein Formular und mit wenigen Zeilen Quellcode eine vollständige Applikation erschaffen zu können, ist zumindest seit Anbruch der .NET-Epoche endgültig vorbei. Vorbei ist aber auch die Zeit, in der mancher C-Programmierer mitleidig auf den VB-Kollegen herabblicken konnte. VB ist seit langem zu einem vollwertigen und professionellen Werkzeug zum Programmieren beliebiger Komponenten für das Microsoft .NET Framework geworden, beginnend bei Windows Forms- über WPF-, ASP.NET-Anwendungen bis hin zu systemnahen Applikationen.

Das vorliegende Buch zu Visual Basic 2015 soll ein faires Angebot sowohl für künftige als auch für fortgeschrittene VB-Programmierer sein. Seine Philosophie knüpft an die vielen anderen Titel an, die wir in den vergangenen siebzehn Jahren zu verschiedenen Programmiersprachen geschrieben haben:

- Programmieren lernt man nicht durch lineares Durcharbeiten eines Lehrbuchs, sondern nur durch unermüdliches Ausprobieren von Beispielen, verbunden mit ständigem Nachschlagen in der Referenz.
- Der Umfang einer modernen Sprache wie Visual Basic in Verbindung mit Visual Studio ist so gewaltig, dass ein seriöses Programmierbuch das Prinzip der Vollständigkeit aufgeben muss und nach dem Prinzip "so viel wie nötig" sich lediglich eine "Initialisierungsfunktion" auf die Fahnen schreiben kann.

Das ist auch der Grund, warum das vorliegende Buch keinen ausgesprochenen Lehrbuchcharakter trägt, sondern mehr ein mit sorgfältig gewählten Beispielen durchsetztes Nachschlagewerk der wichtigsten Elemente der .NET-Programmierung unter Visual Basic 2015 ist.

Gegenüber anderen Büchern zur gleichen oder ähnlichen Thematik nimmt unser Titel für sich in Anspruch, gleichzeitig Lehr- und Übungsbuch zu sein.

Zum Buchinhalt

Wie Sie bereits dem Buchtitel entnehmen können, wagt das vorliegende Werk den Spagat zwischen einem Grundlagen- und einem Profibuch. Sinn eines solchen Buches kann es nicht sein, eine umfassende Schritt-für-Schritt-Einführung in Visual Basic 2015 zu liefern oder all die Informatio-

nen noch einmal zur Verfügung zu stellen, die Sie in der Produktdokumentation (MSDN) ohnehin schon finden und von denen Sie in der Regel nur ein Mausklick oder die F1-Taste trennt.

- Für den *Einsteiger* wollen wir den einzig vernünftigen und gangbaren Weg beschreiten, nämlich nach dem Prinzip "so viel wie nötig" eine Schneise durch den Urwald der .NET-Programmierung mit Visual Basic 2015 schlagen, bis er eine Lichtung erreicht hat, die ihm erste Erfolgserlebnisse vermittelt.
- Für den *Profi* wollen wir in diesem Buch eine Vielzahl von Informationen und Know-how bereitstellen, wonach er bisher in den mitgelieferten Dokumentationen, im Internet bzw. in anderen Büchern vergeblich gesucht hat.

Die Kapitel des Buchs haben wir in fünf Themenkomplexen gruppiert:

1. Grundlagen der Programmierung mit VB.NET
2. Technologien
3. Windows Store Apps
4. WPF-Anwendungen
5. Windows Forms-Anwendungen

Die Kapitel innerhalb eines Teils bilden einerseits eine logische Aufeinanderfolge, können andererseits aber auch quergelesen werden. Im Praxisteil eines jeden Kapitels werden anhand realer Problemstellungen die behandelten Programmiertechniken im Zusammenhang demonstriert.

Im gedruckten Teil dieses Buchs finden Sie die ersten drei Themenkomplexe, denn bereits hier sind wir an die Grenze des drucktechnisch Machbaren gestoßen. Die übrigen zwei Themenkomplexe mussten wir als PDF auslagern, welche Sie sich kostenlos aus dem Internet herunterladen können.

Zu den Codebeispielen

Alle Beispieldaten dieses Buchs und die Kapitel des vierten und fünften Teils können Sie sich unter folgender Adresse herunterladen:

LINK: `http://www.doko-buch.de`

Beim Nachvollziehen der Buchbeispiele beachten Sie bitte Folgendes:

- Kopieren Sie die Buchbeispiele auf die Festplatte. Wenn Sie auf die Projektmappendatei (**.sln*) klicken, wird Visual Studio in der Regel automatisch geöffnet und das jeweilige Beispiel wird in die Entwicklungsumgebung geladen, wo Sie es z.B. mittels F5-Taste kompilieren und starten können.
- Einige wenige Datenbankprojekte verwenden absolute Pfadnamen, die Sie vor dem Kompilieren des Beispiels erst noch anpassen müssen.
- Für einige Beispiele sind ein installierter Microsoft SQL Server Express LocalDB sowie der Microsoft Internet Information Server (ASP.NET) erforderlich.

- Bei der Fehlermeldung "Der Microsoft.Jet.OLEDB.4.0-Provider ist nicht auf dem lokalen Computer registriert." müssen Sie als Zielplattform für das Projekt x86 wählen, da es sich bei OLEDB um einen 32-Bit-Treiber handelt.
- Um mit den WinRT-Projekten arbeiten zu können, müssen Sie Visual Studio 2015 unter Windows 8 bzw. 10 ausführen.
- Beachten Sie die zu einigen Beispielen beigefügten *Liesmich.txt*-Dateien, die Sie auf besondere Probleme hinweisen.

Nobody is perfect

Sie werden – trotz der rund 1900 Seiten – in diesem Buch nicht alles finden, was Visual Basic 2015 bzw. das .NET Framework 4.6 zu bieten haben. Manches ist sicher in einem anderen Spezialtitel noch besser oder ausführlicher beschrieben. Aber Sie halten mit unserem Buch einen optimalen und überschaubaren Breitband-Mix in den Händen, der sowohl vertikal vom Einsteiger bis zum Profi als auch horizontal von den einfachen Sprachelementen bis hin zu komplexen Anwendungen jedem etwas bietet, ohne dabei den Blick auf das Wesentliche im .NET-Dschungel zu verlieren.

Wenn Sie Vorschläge oder Fragen zum Buch haben, können Sie uns gern über unsere Website kontaktieren:

LINK: `http://www.doko-buch.de`

Wir hoffen, dass wir Ihnen mit diesem Buch einen nützlichen Begleiter bei der .NET-Programmierung zur Seite gestellt haben, der es verdient, seinen Platz nicht im Regal, sondern griffbereit neben dem Computer einzunehmen.

Walter Doberenz und Thomas Gewinnus *Wintersdorf/Frankfurt/O., im August 2015*

Kapitel 3

OOP-Konzepte

In .NET ist alles ein Objekt! Viele Entwickler – insbesondere wenn sie mit "altem" Code zu kämpfen haben – tun sich immer noch ziemlich schwer mit OOP, weil ihnen die Komplexität einer vollständigen Anwendung zu hoch erscheint.

Visual Basic erlaubt es Ihnen aber, bereits ohne fundierte OOP-Kenntnisse objektorientiert zu programmieren! Davon haben Sie bereits vor der Lektüre dieses Kapitels, mehr oder weniger unbewusst, Gebrauch gemacht: Sie haben Ereignisbehandlungsroutinen (Event-Handler) geschrieben und den Objekten der visuellen Benutzerschnittstelle (Form, Steuerelemente) Eigenschaften zugewiesen bzw. deren Methoden aufgerufen.

Die Entwicklungsumgebung von Visual Studio erlaubt objektorientiertes Programmieren bereits mit einem Minimum an Vorkenntnissen. Das vorliegende Kapitel will etwas tiefer in die OOP-Problematik eindringen und präsentiert Ihnen neben einigen grundlegenden Ausführungen die für den Einstieg wichtigsten objektspezifischen Features von Visual Basic im Überblick.

3.1 Strukturierter versus objektorientierter Entwurf

Im Unterschied zur objektorientierten ist die klassische strukturierte Programmierung ziemlich sprachunabhängig und hatte Zeit genug, um auch in den letzten Winkel der Programmierwelt vorzudringen.

Demgegenüber stand es um die Akzeptanz der objektorientierten Programmierung bis Anbruch des .NET-Zeitalters zu Beginn dieses Jahrtausends noch nicht zum Besten, das aber hat sich seitdem dramatisch geändert.

3.1.1 Was bedeutet strukturierte Programmierung?

Gern bezeichnet man die strukturierte Programmierung auch als Vorläufer der objektorientierten Programmierung, obwohl dieser Vergleich hinkt. Richtig ist, dass sowohl strukturierte als auch objektorientierte Programmierung fundamentale Denkmuster sind, die gleichberechtigt nebeneinander existieren.

Die Grundkonzepte der strukturierten Programmierung wurden beginnend mit dem Ende der Sechzigerjahre entwickelt und lassen sich mit folgenden Stichwörtern charakterisieren: hierarchische Programmorganisation, logische Programmeinheiten, zentrale Programmsteuerung, beschränkte Datenverfügbarkeit.

Ziel der strukturierten Programmierung ist es, Algorithmen so darzustellen, dass ihr Ablauf einfach zu erfassen und zu verändern ist.

Gegenstand der strukturierten Programmierung ist also die bestmögliche Anordnung von Code, um dessen Transparenz, Testbarkeit und Wiederverwendbarkeit zu maximieren.

Dass VB eine konsequent objektorientierte Sprache ist, bedeutet noch lange nicht, dass man damit nicht auch strukturiert programmieren könnte, im Gegenteil. Im Kapitel 2, wo sich alles um die grundlegenden sprachlichen Elemente von VB dreht, haben wir uns fast ausschließlich auf dem Boden der traditionellen strukturierten Programmierung bewegt und versucht, die OOP noch weitestgehend auszuklammern. So haben wir es größtenteils ignoriert, dass selbst die einfachen Datentypen Objekte sind, und haben z.B. anstatt mit Methoden mit Funktionen und Prozeduren und anstatt mit Klassen mit strukturierten Datentypen (*Structure*) gearbeitet. Tatsächlich können Sie aber mit OOP alles machen, was auch die strukturierte Programmierung erlaubt.

Anstatt globale Variablen in einem Modul zu deklarieren, können Sie statische Klasseneigenschaften verwenden.

Um fit für die aktuellen Herausforderungen zu sein, sollten Sie deshalb – wo immer es vertretbar ist – nach objektorientierten Lösungen streben.

3.1.2 Was heißt objektorientierte Programmierung?

Die objektorientierte Programmierung entfaltete auf breiter Basis erst seit Ende der 80er-Jahre mit dem Beginn des Windows-Zeitalters ihre Wirkung. Sehr bekannte Vertreter objektorientierter Sprachen sind C++, Java, Smalltalk und Borland Delphi – aber auch das alte Visual Basic war bereits in vielen wesentlichen Zügen objektorientiert aufgebaut.

Objektorientierte Programmierung ist ein Denkmuster, bei dem Programme als Menge von über Nachrichten kooperierenden Objekten organisiert werden und jedes Objekt Instanz einer Klasse ist.

Im Unterschied zur strukturierten Programmierung bedeutet "objektorientiert" also, dass Daten und Algorithmen nicht mehr nebeneinander existieren, sondern in Objekten zusammengefasst sind.

Während Module in der strukturierten Programmierung zwar auch Daten und Code zusammenfassen, stellen Klassen jetzt Vorlagen dar, von denen immer neue Kopien (Instanzen) angefertigt werden können. Diese Instanzen, d.h. die Objekte, kapseln den Zugriff auf die enthaltenen Daten hinter Schnittstellen (Interfaces).

Der große Vorteil der OOP ist ihre Ähnlichkeit mit den menschlichen Denkstrukturen. Dadurch wird vor allem dem Einsteiger, der bisher über keine bzw. wenig Programmiererfahrung verfügt, das Verständnis der OOP erleichtert.

HINWEIS: Die OOP verlangt eine Anpassung des Software-Entwicklungsprozesses und der eingesetzten Methoden an den Denkstil des Programmierers – nicht umgekehrt!

Die OOP ist eine der wenigen Fälle, in denen der Einsteiger gegenüber dem Profi zumindest einen kleinen Vorteil besitzt: Er ist noch nicht in der Denkweise klassischer Programmiersprachen gefangen, die dazu erziehen, in Abläufen zu denken, bei denen die in der realen Welt zu beobachtenden Abläufe Schritt für Schritt in Algorithmen umgesetzt werden, etwa um betriebliche Prozesse per Programm zu automatisieren.

Die OOP entspricht hingegen der üblichen menschlichen Denkweise, indem sie z.B. reale Objekte aus der abzubildenden Umwelt identifiziert und in ihrer Art beschreibt.

Das Konzept der objektorientierten Programmierung (OOP) überwindet den prozeduralen Ansatz der klassischen strukturellen Programmierung zugunsten einer realitätsnahen Modellierung.

3.2 Grundbegriffe der OOP

Bevor wir uns den Details zuwenden, sollen die wichtigsten Begriffe der objektorientierten Programmierung zunächst allgemein, d.h. ohne Bezug auf eine konkrete Programmiersprache, erörtert werden.

3.2.1 Objekt, Klasse, Instanz

Der Programmierer versteht unter einem *Objekt* die Zusammenfassung (Kapselung) von Daten und zugehörigen Funktionalitäten. Ein solches Softwareobjekt wird auch oft benutzt, um Dinge des täglichen Lebens für Zwecke der Datenverarbeitung abzubilden. Aber das ist nur ein Aspekt, denn Objekte sind ganz allgemein Dinge, die Sie in Ihrem Code beschreiben wollen, es sind Gruppen von Eigenschaften, Methoden und Ereignissen, die logisch zusammengehören. Als Programmierer arbeiten Sie mit einem Objekt, indem Sie dessen Eigenschaften und Methoden manipulieren und auf seine Ereignisse reagieren.

Eine *Klasse*[1] ist nicht mehr und nicht weniger als ein "Bauplan", auf dessen Grundlage die entsprechenden Objekte zur Programmlaufzeit erzeugt werden. Gewissermaßen als Vorlage (Prägestempel) für das Objekt legt die Klasse fest, wie das Objekt auszusehen hat und wie es sich verhalten soll. Es handelt sich bei einer Klasse also um eine reine Softwarekonstruktion, die Eigenschaften, Methoden und Ereignisse eines Objekts definiert, ohne das Objekt zu erzeugen.

HINWEIS: In VB haben Sie grundsätzlich die Möglichkeit, zwischen Klassen und Strukturen zu wählen. Letztere wurden bereits im Sprachkapitel (Abschnitt 2.7.2) einführend behandelt, bieten allerdings noch weitaus mehr Möglichkeiten, die fast an die von Klassen heranreichen. Wir aber wollen uns im vorliegenden Kapitel ausschließlich mit Klassen beschäftigen.

[1] Oft wird anstatt "Klasse" mit völlig gleichwertiger Bedeutung auch der Begriff "Objekttyp" (oder auch "Typ") verwendet.

Man erhält erst dann ein konkretes Objekt, wenn man eine *Instanz* einer Klasse bildet. Es lassen sich viele Objekte mit einer einzigen Klassendefinition erzeugen.

BEISPIEL 3.1: Objekt, Klasse, Instanz

Auf dem Montageband werden zahlreiche Auto-Objekte nach ein und denselben Konstruktionsvorschriften für die Klasse "Auto" gebaut. Diesen Vorgang könnte man auch als Bildung von Instanzen der Klasse "Auto" bezeichnen. Während die Klasse lediglich die Eigenschaft *Farbe* definiert, wird der konkrete Wert (rot, blau, grün ...) erst beim Erzeugen des Objekts (der Instanz) zugewiesen.

3.2.2 Kapselung und Wiederverwendbarkeit

Klassen realisieren das Prinzip der *Kapselung* von Objekten, das es ermöglicht, die Implementierung der Klasse (der Code im Inneren) von deren Schnittstelle bzw. Interface (die öffentlichen Eigenschaften, Methoden und Ereignisse) sauber zu trennen. Durch das Verbergen der inneren Struktur werden die internen Daten und einige verborgene Methoden geschützt, sind also von außen nicht zugänglich. Die Manipulation des Objekts kann lediglich über streng definierte, über die Schnittstelle zur Verfügung gestellte öffentliche Methoden erfolgen.

Klassen ermöglichen die *Wiederverwendbarkeit* von Code. Nachdem eine Klasse geschrieben wurde, können Sie diese an verschiedenen Stellen innerhalb einer Applikation verwenden. Klassen reduzieren somit den redundanten Code einer Anwendung, sie erleichtern außerdem die Wartung des Codes.

3.2.3 Vererbung und Polymorphie

Echte Vererbung (*Implementierungsvererbung*) ermöglicht es Klassen zu definieren, die von anderen Klassen abgeleitet werden, wobei nicht nur die Schnittstelle, sondern auch der dahinter liegende Code (die Implementierung) vom Nachkommen übernommen wird.

Da es nun möglich ist, die Implementierung einer Klasse für weitere Klassen als Grundlage zu verwenden, kann man Unterklassen bilden, die alle Eigenschaften und Methoden ihrer Oberklasse (auch oft als Superklasse bezeichnet) erben. Diese Unterklassen können zu den geerbten Eigenschaften neue hinzufügen oder Eigenschaften der Oberklasse verstecken, indem sie diese überschreiben.

Wird von einer solchen Unterklasse ein Objekt erzeugt (also eine Instanz der Unterklasse gebildet), dann dient für dieses Objekt sowohl die Ober- als auch die Unterklasse als "Bauplan".

Visual Basic unterstützt das Überschreiben (*Overriding*) von Methoden[1] der Oberklasse mit alternativen Methoden der Unterklasse.

[1] Nicht zu verwechseln mit dem Überladen (Overloading) von Methoden.

OOP macht es möglich, ein und dieselbe Methode für ganz verschiedene Objekte zu verwenden, man nennt dies dann *Polymorphie* (Vielgestaltigkeit). Jedes dieser Objekte kann die Ausführung unterschiedlich realisieren. Für das aufrufende Objekt bleibt der Vorgang trotzdem derselbe.

BEISPIEL 3.2: Vererbung

Die Methode "Beschleunigen" ist in einer "Fahrzeug"-Klasse definiert, welche an die Unterklassen "Auto" und "Fahrrad" vererbt. Es ist klar, dass diese Methoden in beiden Unterklassen überschrieben, d.h. völlig unterschiedlich implementiert werden müssen.

Als Polymorphie, die aufs Engste mit der Vererbung verknüpft ist, kann man also die Fähigkeit von Unterklassen bezeichnen, Eigenschaften und Methoden mit dem gleichen Namen, aber mit unterschiedlichen Implementierungen aufzurufen.

3.2.4 Sichtbarkeit von Klassen und ihren Mitgliedern

Um die Klasse bzw. ihre Mitglieder (Member, Elemente) gezielt zu verbergen oder offen zu legen, sollten Sie von den Zugriffsmodifizierern Gebrauch machen, die den Gültigkeitsbereich (bzw. die *Sichtbarkeit*) einschränken.

Klassen

Die folgende Tabelle zeigt die möglichen Einschränkungen bei der Sichtbarkeit von Klassen:

Modifizierer	Sichtbarkeit
Public	Unbeschränkt. Auch von anderen Assemblierungen aus können Objekte der Klasse erstellt werden.
Friend	Nur innerhalb des aktuellen Projekts. Außerhalb des Projekts ist kein Objekt dieser Klasse erstellbar. Gilt als Standard, falls kein Modifizierer vorangestellt wird.
Private	Nur innerhalb einer anderen Klasse.

Klassenmitglieder

Die folgende Tabelle zeigt die Zugriffsmöglichkeiten auf die Klassenmitglieder (Member).

Modifizierer	Sichtbarkeit
Public	Unbeschränkt.
Friend	Innerhalb der aktuellen Anwendung wie *Public*, sonst wie *Private*.
Protected	Wie *Private*, Mitglieder dürfen aber auch in abgeleiteten Klassen verwendet werden.
Protected Friend	Innerhalb der aktuellen Anwendung wie *Protected*, sonst wie *Private*.
Private	Nur innerhalb der Klasse.

Die Schlüsselwörter *Private* und *Public* definieren immer die beiden Extreme des Zugriffs. Betrachtet man die jeweiligen Klassen als allein stehend, so reichen diese beiden Zugriffsarten völlig aus. Mit solchen, quasi isolierten, Klassen lassen sich allerdings keine komplexeren Probleme lösen.

Um einzelne Klassen miteinander zu verbinden, verwenden Sie den mächtigen Mechanismus der Vererbung. In diesem Zusammenhang gewinnt die *Protected*-Deklaration wie folgt an Bedeutung:

- Da eine abgeleitete Klasse auf die *Protected*-Member zugreifen kann, sind diese Member für die abgeleitete Klasse quasi *Public*.
- Ist eine Klasse nicht von einer anderen abgeleitet, kann sie nicht auf deren *Protected*-Member zugreifen, da diese dann quasi *Private* sind.

Mehr zu diesem Thema finden Sie im Abschnitt 3.9 (Vererbung und Polymorphie).

3.2.5 Allgemeiner Aufbau einer Klasse

Bevor der Einsteiger seine erste Klasse schreibt, sollte er sich zunächst im einführenden Sprachkapitel mit den Strukturen (siehe Abschnitt 2.7.2) anfreunden, die in Aufbau und Anwendung starke Ähnlichkeiten zu Klassen aufweisen (der wesentliche Unterschied ist, dass Strukturen Wertetypen, Klassen hingegen Referenztypen sind). Auch im Aufbau von Funktionen bzw. Methoden sollte sich der Lernende auskennen (Abschnitt 2.8).

Im Unterschied zu einer Struktur (Schlüsselwort *Structure*) wird eine Klasse mit dem Schlüsselwort *Class* deklariert. Hier die (stark vereinfachte) Syntax:

SYNTAX:
```
Modifizierer Class Bezeichner
        ' ... Felder
        ' ... Konstruktoren
        ' ... Eigenschaften
        ' ... Methoden
        ' ... Ereignisse
    End Class
```

Im Klassenkörper haben es wir es mit "Klassenmitgliedern" (Member) wie Feldern, Konstruktoren, Eigenschaften, Methoden und Ereignissen zu tun, auf die wir noch detailliert zu sprechen kommen werden.

Die Definition der Klassenmitglieder bezeichnet man auch als *Implementation* der Klasse.

BEISPIEL 3.3: Eine einfache Klasse *CKunde* wird deklariert und implementiert.

VB
```
Public Class CKunde
   Private _anrede As String          '    Feld
   Private _name As String            '    dto.

   Public Sub New(anr As String, nam As String)    ' Konstruktor
      _anrede = anr
```

BEISPIEL 3.3: Eine einfache Klasse *CKunde* wird deklariert und implementiert.

```
      _name = nam
   End Sub
   Public Property name() As String                ' Eigenschaft
      Get
         Return _name
      End Get
      Set(value As String)
         _name = value
      End Set
   End Property

   Public Function adresse() As String             ' Methode
       Dim s As String = _anrede & " " & _name
       Return s
   End Function
End Class
```

Unsere Klasse verfügt damit über zwei Felder, eine Eigenschaft und eine Methode. Da die beiden Felder mit dem *Private*-Modifizierer deklariert wurden, sind sie von außen nicht sichtbar.

3.3 Ein Objekt erzeugen

Existiert eine Klasse, so steht dem Erzeugen von Objektvariablen nichts mehr im Weg. Eine Objektvariable ist ein Verweistyp, sie enthält also nicht das Objekt selbst, sondern stellt lediglich einen Zeiger (Adresse) auf den Speicherbereich des Objekts bereit. Es können sich also durchaus mehrere Objektvariablen auf ein und dasselbe Objekt beziehen. Wenn eine Objektvariable den Wert *Nothing* enthält, bedeutet das, dass sie momentan "ins Leere" zeigt, also kein Objekt referenziert.

Unter der Voraussetzung, dass eine gültige Klasse existiert, verläuft der Lebenszyklus eines Objekts in Ihrem Programm in folgenden Etappen:

- Referenzierung (eine Objektvariable wird deklariert, sie verweist momentan noch auf *Nothing*)
- Instanziierung (die Objektvariable zeigt jetzt auf einen konkreten Speicherplatzbereich)
- Initialisierung (die Datenfelder der Objektvariablen werden mit Anfangswerten gefüllt)
- Arbeiten mit dem Objekt (es wird auf Eigenschaften und Methoden des Objekts zugegriffen, Ereignisse werden ausgelöst)
- Zerstören des Objekts (das Objekt wird dereferenziert, der belegte Speicherplatz wird wieder freigegeben)

Werfen wir nun einen genaueren Blick auf die einzelnen Etappen.

3.3.1 Referenzieren und Instanziieren

Es stehen zwei Varianten zur Verfügung.

Variante 1 erfordert zwei Schritte:

SYNTAX:
```
Modifizierer myObject As Klasse
myObjekt = New Klasse(Parameter)
```

BEISPIEL 3.4: Ein Objekt *kunde1* wird referenziert und erzeugt.

VB
```
Private kunde1 As CKunde                    ' Referenzieren
kunde1 = New CKunde1()                      ' Erzeugen
```

In *Variante2*, der Kurzform, sind beide Schritte in einer Anweisung zusammengefasst, d.h., das Objekt wird zusammen mit seiner Deklaration erzeugt.

SYNTAX:
```
Modifizierer myObject = New Klasse()
```

BEISPIEL 3.5: Das Äquivalent zum Vorgängerbeispiel.

VB
```
Private kunde1 As New CKunde()
```

Dem Klassenbezeichner (*Klasse*) müsste genauer genommen noch der Name der Klassenbibliothek (bzw. Name des Projekts) vorangestellt werden, doch dies wird unter Visual Studio nicht erforderlich sein, da der entsprechende Namensraum (*Namespace*) bereits automatisch eingebunden wurde (*Imports*-Anweisung).

Obwohl die Kurzform sehr eindrucksvoll ist, können Sie hier keine Fehlerbehandlung (*Try...Catch*-Block) durchführen. Diese Einschränkung macht diese Art von Deklaration weniger nützlich.

Empfehlenswert ist also fast immer das getrennte Deklarieren und Erzeugen[1].

BEISPIEL 3.6: Eine mögliche Fehlerbehandlung

VB
```
Private kunde1 As CKunde
Try
   kunde1 = New CKunde()
Catch e As Exception
   MessageBox.Show(ex.Message)
End Try
```

[1] Aus Platz- und Bequemlichkeitsgründen halten sich auch die Autoren nicht immer an diese Empfehlung.

3.3.2 Klassische Initialisierung

Anstatt die Anfangswerte einzeln zuzuweisen, können Sie diese zusammen mit einem Konstruktor übergeben. Zunächst ein Beispiel ohne eigenen Konstruktor, wobei der parameterlose Standardkonstruktor zum Einsatz kommt.

BEISPIEL 3.7: Das Objekt *kunde1* wird erzeugt (Standardkonstruktor), zwei Eigenschaften werden einzeln zugewiesen.

VB
```
Dim kunde1 As New CKunde()
kunde1.anrede = "Frau"
kunde1.name = "Müller"
```

BEISPIEL 3.8: Das Objekt *kunde1* wird erzeugt und mit einem Konstruktor initialisiert.

VB
```
Dim kunde1 As New CKunde("Frau", "Müller")
```

HINWEIS: Weitere Einzelheiten entnehmen Sie dem Abschnitt 3.8.1.

3.3.3 Objekt-Initialisierer

Man kann ein Objekt auch dann erzeugen und seine Eigenschaften (keine privaten Felder!) initialisieren, wenn es dazu keinen Konstruktor gibt.

BEISPIEL 3.9: Das Vorgängerbeispiel mit Objekt-Initialisierer

VB
```
Private kunde1 As New CKunde With {.anrede = "Frau", .name = "Müller"}
```

HINWEIS: Mehr zu Objekt-Initialisierern siehe Abschnitt 3.8.2!

3.3.4 Arbeiten mit dem Objekt

Wie Sie bereits wissen, erfolgt der Zugriff auf Eigenschaften und Methoden eines Objekts, indem der Name des Objekts mit einem Punkt (.) vom Namen der Eigenschaft/Methode getrennt wird.

SYNTAX: *Objekt.Eigenschaft|Methode()*

BEISPIEL 3.10: Die Eigenschaft *Guthaben* des Objekts *kunde1* wird zugewiesen und die Methode *adresse* aufgerufen.

VB
```
kunde1.Guthaben = 10
Label1.Text = kunde1.adresse()
```

3.3.5 Zerstören des Objekts

Wenn Sie das Objekt nicht mehr brauchen, können Sie die Objektvariable auf *Nothing* setzen. Vorher können Sie (müssen aber nicht) die *Dispose*-Methode des Objekts aufrufen, vorausgesetzt, Sie haben Sie auch implementiert.

BEISPIEL 3.11: Der *kunde1* wird entfernt.

VB
```
kunde1.Dispose()
kunde1 = Nothing
```

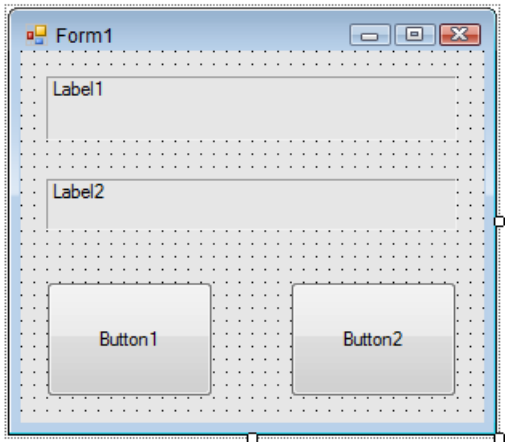
HINWEIS: Das Objekt wird allerdings erst dann zerstört, wenn der Garbage Collector festgestellt hat, dass es nicht länger benötigt wird.

3.4 OOP-Einführungsbeispiel

Raus aus dem muffigen Hörsaal, lassen Sie uns endlich einmal selbst eine einfache Klasse erstellen und beschnuppern!

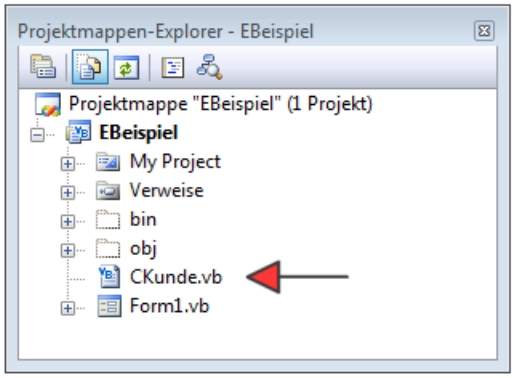
3.4.1 Vorbereitungen

- Öffnen Sie ein neues Projekt (z.B. mit dem Namen *Kunden*) als Windows Forms-Anwendung.
- Auf das Startformular (*Form1*) platzieren Sie zwei *Label*s und zwei *Button*s.



- Nachdem Sie den Menüpunkt *Projekt|Klasse hinzufügen...* gewählt haben, geben Sie im Dialogfenster den Namen *CKunde.vb* ein und klicken "Hinzufügen".

Der Projektmappen-Explorer zeigt jetzt die neue Klasse:



HINWEIS: Sie müssen eine Klasse nicht unbedingt in einem eigenen Klassenmodul definieren, Sie könnten die Klasse z.B. auch zum bereits vorhandenen Code des Formulars (*Form1.vb*) hinzufügen. Das Verwenden eigener Klassenmodule (idealerweise eins pro Klasse) steigert aber die Übersichtlichkeit des Programmcodes und erleichtert dessen Wiederverwendbarkeit.

3.4.2 Klasse definieren

Tragen Sie in den Klassenkörper die Implementierung der Klasse ein, sodass der komplette Code der Klasse schließlich folgendermaßen ausschaut:

```
Public Class CKunde
    Public Anrede As String             ' einfache Eigenschaften
    Public Name As String               '      dto.
    Public PLZ As Integer               '      dto.
    Public Ort As String                '      dto.
    Public Stammkunde As Boolean        '      dto.
    Public Guthaben As Decimal          '      dto.

    Public Function getAdresse() As String              ' erste Methode
        Dim s As String = Anrede & " " & Name & vbCrLf & PLZ.ToString & " " & Ort
        Return s
    End Function

    Public Sub addGuthaben(betrag As Decimal)     ' zweite Methode
        If Stammkunde Then Guthaben += betrag
    End Sub
End Class
```

Bemerkungen

- Die Klasse verfügt über sechs "einfache" Eigenschaften, und zwar sind das alle als *Public* deklarierten Variablen, die man auch als "öffentliche Felder" bezeichnet. Die Betonung liegt hier auf "einfach", da wir später noch lernen werden, wie man "richtige" Eigenschaften programmiert.
- Weiterhin verfügt die Klasse über zwei *Methoden* (eine Funktion und eine Prozedur). Die Funktion *getAdresse()* liefert als Rückgabewert die komplette Anschrift des Kunden.
- Die Prozedur (*Sub*) *addGuthaben()* hingegen liefert keinen Wert zurück, sie erhöht den Wert des *Guthaben*-Felds bei jedem Aufruf um einen bestimmten Betrag.

3.4.3 Objekt erzeugen und initialisieren

Wechseln Sie nun in das Code-Fenster von *Form1*.

Zu Beginn deklarieren Sie eine Objektvariable *kunde1*:

```
Private kunde1 As CKunde            ' Objekt referenzieren
```

Dem linken Button geben Sie die Beschriftung "Objekt erzeugen und initialisieren" und belegen sein *Click*-Ereignis wie folgt:

```
    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        kunde1 = New CKunde()          ' Objekt erzeugen (instanziieren)
        With Kunde1                    ' Objekt initialisieren:
            .Anrede = "Herr"
            .Name = "Müller"
            .PLZ = 12345
            .Ort = "Berlin"
            .Stammkunde = True
        End With
    End Sub
```

3.4.4 Objekt verwenden

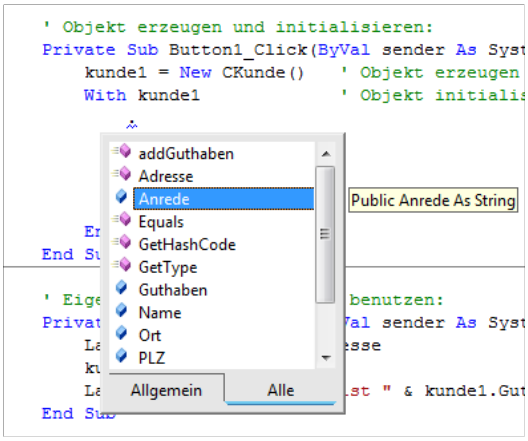
Hinterlegen Sie nun den rechten Button mit der Beschriftung "Eigenschaften und Methoden verwenden" wie folgt:

```
    Private Sub Button2_Click(sender As Object, e As EventArgs) Handles Button2.Click
        Label1.Text = Kunde1.getAdresse    ' erste Methode aufrufen
        Kunde1.addGuthaben(50D)            ' zweite Methode aufrufen
        Label2.Text = "Guthaben ist " & Kunde1.Guthaben.ToString("C")    ' Eigenschaft lesen
    End Sub
```

3.4.5 Unterstützung durch die IntelliSense

Sie haben beim Eintippen des Quelltextes (insbesondere im Code-Fenster von *Form1*) bereits gemerkt, dass Sie durch die IntelliSense von Visual Studio eifrigst unterstützt werden.

Die IntelliSense weist Sie z.B. auf die verfügbaren Klassenmitglieder (Eigenschaften und Methoden) hin und ergänzt den Quellcode automatisch, wenn Sie doppelt auf den gewünschten Eintrag klicken.

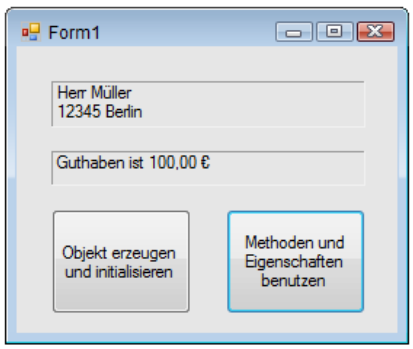


Falls das gewünschte Klassenmitglied nicht erscheint, müssen Sie sofort stutzig werden und es keinesfalls mit dem gewaltsamen Eintippen des Namens versuchen, denn dann gibt es wahrscheinlich einen Fehler beim Kompilieren. Überprüfen Sie stattdessen lieber nochmals die Klassendeklaration, z.B. ob vielleicht nicht doch der *Public*-Modifizierer vergessen wurde.

3.4.6 Objekt testen

Nun ist es endlich so weit, dass Sie Ihr erstes eigenes VB-Objekt vom Stapel lassen können. Unmittelbar nach Programmstart betätigen Sie den linken Button und danach den rechten. Durch mehrmaliges Klicken auf den zweiten Button wird sich das Guthaben des Kunden Müller in 50-€-Schritten erhöhen.

Falls Sie zu voreilig gewesen sind und unmittelbar nach Programmstart den zweiten statt den ersten Button gedrückt haben, stürzt Ihnen das Programm mit der Laufzeit-Fehlermeldung "Der Objektverweis wurde nicht auf eine Objektinstanz festgelegt." ab.



3.4.7 Warum unsere Klasse noch nicht optimal ist

Unsere Klasse funktioniert nach außen hin zwar ohne erkennbare Mängel, ist hinsichtlich ihrer inneren Konstruktion aber keinesfalls als optimal zu bezeichnen. Wir haben deshalb keinerlei Grund, uns zufrieden zurückzulehnen, denn das uns unter Visual Basic zur Verfügung stehende OOP-Instrumentarium wurde von uns bei weitem noch nicht ausgeschöpft.

- Beispielsweise haben wir nur "einfache" Eigenschaften, nämlich *Public*-Felder verwendet, was eigentlich eine schwere Sünde in den Augen der OOP-Puristen ist.
- Weiterhin war das Initialisieren der Eigenschaften über mehrere Codezeilen ziemlich mühselig (von einem hilfreichen Konstruktor haben wir noch keinerlei Gebrauch gemacht).
- Außerdem wird eine Klasse erst dann so richtig effektiv, wenn wir davon nicht nur eine, sondern mehrere Instanzen (sprich Objekte) ableiten. Diese wiederum kann man ziemlich elegant in so genannten Auflistungen (Collections) verwalten (siehe Kapitel 5).

Doch zur Beseitigung dieser und anderer Unzulänglichkeiten kommen wir erst später. Ein weiteres Problem, was uns unter den Nägeln brennt, können und wollen wir aber nicht weiter aufschieben und es gleich im folgenden Abschnitt behandeln.

3.5 Eigenschaften

Eigenschaften bestimmen die statischen Attribute eines Objekts, sie leiten sich von dessen *Zustand* ab, wie er in den Zustandsvariablen (Objektfeldern) gespeichert ist. Im Unterschied zu den Methoden, die von allen Instanzen der Klasse gemeinsam genutzt werden, sind die den Eigenschaften zugewiesenen Werte für alle Objekte einer Klasse meist unterschiedlich.

3.5.1 Eigenschaften kapseln

Von den im Objekt enthaltenen Feldern sind die *Public*-Felder als "einfache" Eigenschaften zu betrachten.

In unserem Beispiel hatten wir für die Klasse *CKunde* solche "einfachen" Eigenschaften als *Public*-Variable deklariert. Das allerdings ist nicht die "feine Art" der objektorientierten Programmierung, denn das Veröffentlichen von Feldern widerspricht dem hochgelobten Prinzip der Kapselung und erlaubt keinerlei Zugriffskontrolle wie z.B. Wertebereichsüberprüfung oder die Vergabe von Lese- und Schreibrechten.

Idealerweise sind deshalb in einem Objekt nur private Felder enthalten, und der Zugriff auf diese wird durch Accessoren (Zugriffsmethoden) gesteuert.

In diesem Sinn ist eine *Eigenschaft* gewissermaßen ein Mittelding zwischen Feld und Methode. Sie verwenden die Eigenschaft wie ein öffentliches Feld. Vom Compiler aber wird der Feldzugriff in den Aufruf von Accessoren – das sind spezielle Zugriffsmethoden auf private Felder – übersetzt. Doch schauen wir uns das Ganze lieber in der Praxis an.

Deklarieren von Eigenschaften

Eigenschaften werden ähnlich wie öffentliche Methoden deklariert. Innerhalb der Deklaration implementieren Sie für den Lesezugriff eine *Get*- und für den Schreibzugriff eine *Set*-Zugriffsmethode. Während die *Get*-Methode ihren Rückgabewert über *Return* liefert, erhält die *Set*-Methode den zu schreibenden Wert über den Parameter *value*.

```
SYNTAX:  [Public|Friend|Protected] Property Eigenschaftsname As Type
                Get
                     ' hier Lesezugriff (Wert=priv.Felder) implementieren
                     Return Wert
                End Get
                Set(value As Type)
                      ' hier Schreibzugriff (priv.Felder=value) implementieren
                 End Set
           End Property
```

Wir wollen nun unser Beispiel mit "echten" Eigenschaften ausstatten. Dazu werden zunächst die *Public*-Felder in *Private* verwandelt und durch Voranstellen von "_" umbenannt, um Namenskonflikte mit den gleichnamigen Eigenschafts-Deklarationen zu vermeiden.

```
Public Class CKunde
    Private _anrede As String        '    privates Feld
    Private _name As String          '      dto.
    ...
```

Der Schreibzugriff auf die Eigenschaft *Anrede* soll so kontrolliert werden, dass nur die Werte "Herr" oder "Frau" zulässig sind. Geben Sie die erste Zeile der *Property*-Deklaration ein, so generiert Visual Studio automatisch den kompletten Rahmencode:

```
Public Property Anrede As String
    Get

    End Get
    Set(value As String)

    End Set
End Property
```

Sie brauchen dann nur noch den Lese- und den Schreibzugriff zu implementieren, sodass die komplette Eigenschaftsdefinition schließlich folgendermaßen aussieht:

```
    Public Property Anrede() As String
        Get
            Return _anrede
        End Get
        Set(value As String)
            If (value = "Herr") Or (value = "Frau") Then
                _anrede = value
            Else
```

```
            MessageBox.Show("Die Anrede '" & value & "' ist nicht zulässig!",
                                            "Fehler bei der Eingabe!")
        End If
    End Set
End Property
```

Beim Implementieren der Eigenschaft *Name* machen wir es uns etwas einfacher. Hier soll uns die einfache Kapselung genügen (es gibt also keinerlei Zugriffskontrolle):

```
Public Property Name() As String
    Get
        Return _name
    End Get
    Set(value As String)
        _name = value
    End Set
End Property
```

Zugriff

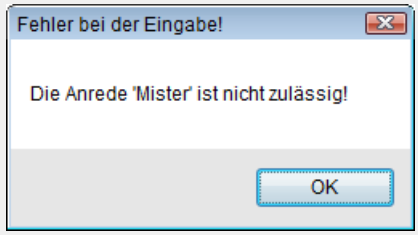
Wenn Sie ein Objekt verwenden, merken Sie auf Anhieb natürlich nicht, ob es noch über "einfache" oder schon über "richtige" Eigenschaften verfügt, es sei denn, die in die *Get*- bzw. *Set*-Methoden eingebauten Zugriffsbeschränkungen werden verletzt und Sie erhalten entsprechende Fehlermeldungen.

BEISPIEL 3.12: Sie wollen die Anrede "Mister" zuweisen, was zu einem Laufzeitfehler führt.

VB

```
Dim kunde1 As New CKunde()
kunde1.Anrede = "Mister"      ' Fehler!
```

Ergebnis



Bemerkung

- Beim Schreiben des Quellcodes in der Entwicklungsumgebung Visual Studio merken Sie den "feinen" Unterschied zwischen "einfachen" und "richtigen" Eigenschaften, denn die IntelliSense zeigt dafür unterschiedliche Symbole.
- In unserem Beispiel verhält sich nur die Eigenschaft *Anrede* "intelligent", d.h., sie unterliegt einer Zugriffskontrolle. Bei den übrigen Eigenschaften erfolgt lediglich eine 1:1-Zuordnung zu den privaten Feldern. Hier sollte man nicht "päpstlicher als der Papst" sein und es bei den ursprünglichen *Public*-Feldern belassen. Wir aber haben diesen (eigentlich sinnlosen) Aufwand bei der *Name*-Eigenschaft nur wegen des Lerneffekts betrieben.

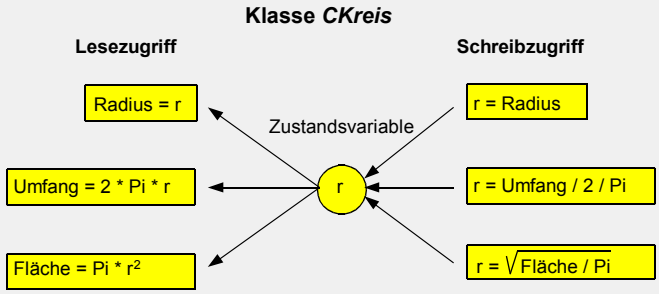
3.5.2 Eigenschaften mit Zugriffsmethoden kapseln

Mit Zugriffsmethoden lässt sich weit mehr anstellen, als nur den Zugriff auf private Felder der Klasse zu kontrollieren. So können z.B. innerhalb der Methode komplexe Berechnungen mit den Feldern (die man auch *Zustandsvariablen* nennt) und den übergebenen Parametern ausgeführt werden.

BEISPIEL 3.13: Eigenschaften mit Zugriffsmethoden kapseln

VB

Eine Klasse *CKreis* hat die Eigenschaften *Radius*, *Umfang* und *Fläche*. In der einzigen Zustandsvariablen *r* braucht aber nur der Radius abgespeichert zu werden, da sich die übrigen Eigenschaften aus *r* berechnen lassen (*Get* = Lesezugriff) bzw. umgekehrt (*Set* = Schreibzugriff).




```
Public Class CKreis
    Private r As Double                 ' die einzige Zustandsvariable
```

Die Eigenschaft *Radius*:

```
    Public Property Radius() As String
        Get
            Return r.ToString("#,#0.00")
        End Get
        Set(value As String)
            If value <> String.Empty Then
                r = CDbl(value)
            Else
                r = 0
            End If
        End Set
    End Property
```

Die Eigenschaft *Umfang*:

```
    Public Property Umfang() As String
        Get
            Return (2 * Math.PI * r).ToString("#,#0.00")
        End Get
        Set(value As String)
```

BEISPIEL 3.13: Eigenschaften mit Zugriffsmethoden kapseln

```
            If value <> String.Empty Then
                r = CDbl(value) / 2 / Math.PI
            Else
                r = 0
            End If
        End Set
    End Property
```

Die Eigenschaft *Fläche*:

```
    Public Property Fläche() As String
        Get
            Return (Math.PI * Math.Pow(r, 2)).ToString("#,#0.00")
        End Get
        Set(value As String)
            If value <> String.Empty Then
                r = Math.Sqrt(CDbl(value) / Math.PI)
            Else
                r = 0
            End If
        End Set
    End Property
End Class
```

Das komplette Programm finden Sie unter

► 3.12.1 Eigenschaften sinnvoll kapseln

3.5.3 Lese-/Schreibschutz für Eigenschaften

Es kommt häufig vor, dass bestimmte Eigenschaften nur gelesen oder nur geschrieben werden dürfen (*ReadOnly* bzw. *WriteOnly*). Um diese Art der Zugriffsbeschränkung zu realisieren, ist keinerlei Aufwand erforderlich – im Gegenteil:

HINWEIS: Um eine Eigenschaft allein für den Lese- bzw. Schreibzugriff zu deklarieren, lässt man einfach die *Get*- bzw. die *Set*-Zugriffsmethode weg.

BEISPIEL 3.14: Lese-/Schreibschutz für Eigenschaften

VB

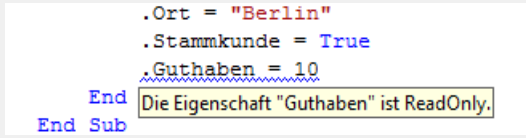
In unserer *CKunde*-Klasse soll das Guthaben für den direkten Schreibzugriff gesperrt werden. Das klingt logisch, da zur Erhöhung des Guthabens bereits die Methode *addGuthaben* existiert.

```
Public Class CKunde
    ...
     Public ReadOnly Property Guthaben() As Decimal
        Get
```

BEISPIEL 3.14: Lese-/Schreibschutz für Eigenschaften

```
            Return _guthaben
        End Get
    End Property
    ...
End Class
```

Bereits in der Entwicklungsumgebung von Visual Studio wird nun der Versuch abgewiesen, dieser Eigenschaft einen Wert zuzuweisen:



3.5.4 Statische Eigenschaften

Mitunter gibt es Eigenschaften, deren Werte für alle aus der Klasse instanziierten Objekte identisch sind und die deshalb nur einmal in der Klasse gespeichert zu werden brauchen.

HINWEIS: Statische Eigenschaften (*Klasseneigenschaften*) werden mit dem Schlüsselwort *Shared* deklariert.

Außer dem *Shared*-Schlüsselwort gibt es beim Deklarieren keine Unterschiede zu den normalen Instanzeneigenschaften.

Statische Eigenschaften können benutzt werden, ohne dass dazu eine Objektvariable deklariert und ein Objekt instanziiert werden muss! Es genügt das Voranstellen des Klassenbezeichners.

BEISPIEL 3.15: Die Klasse *CKunde* soll zusätzlich eine "einfache" Eigenschaft *Rabatt* bekommen, die für jedes Kundenobjekt immer den gleichen Wert hat.

VB
```
Public Class CKunde
    ...
    Public Shared Rabatt As Double
    ...
End Class
```

Der Zugriff ist sofort über den Klassenbezeichner möglich, ohne dass dazu eine Objektvariable erzeugt werden müsste.

BEISPIEL 3.16: Allen Kunden wird ein Rabatt von 15% zugewiesen.

VB
```
CKunde.Rabatt = 0.15
```

Vielen Umsteigern, die aus der strukturierten Programmierung kommen, bereitet es Schwierigkeiten, auf ihre globalen Variablen zu verzichten, mit denen sie Werte zwischen verschiedenen Programmmodulen ausgetauscht haben. Genau hier bieten sich statische Eigenschaften an, die z.B. in einer extra für derlei Zwecke angelegten Klasse *CAllerlei* abgelegt werden könnten.

3.5.5 Selbst implementierende Eigenschaften

Bei sehr einfachen Eigenschaften (vergleichbar mit denen, die Sie bislang "unsauber" als *Public*-Felder deklariert haben) können Sie seit VB 2010 so genannte *Auto-implemented Properties* verwenden. Der VB-Compiler generiert im Hintergrund für Sie die entsprechenden *Get*- und *Set*-Zugriffsmethoden und erzeugt außerdem ein privates Feld, um den Wert der Eigenschaft zu speichern.

Wie viel lästige Schreibarbeit Sie sparen können, soll das folgende Beispiel verdeutlichen.

BEISPIEL 3.17: Die folgende Deklaration einer selbst implementierenden Eigenschaft

VB
```
Property Ort As String = "München"
```

ist äquivalent zu

```
Private _Ort As String = "München"                  ' backing field
Property Ort As String
    Get
        Return _Ort
    End Get
    Set(value As String)

        _Ort = value
    End Set
End Property
```

Wie Sie sehen, kann der Eigenschaft auch ein Standardwert zugewiesen werden. Der Name des automatisch angelegten "backing fields" entspricht dem Namen der Eigenschaft mit vorangestelltem Unterstrich.

HINWEIS: Achten Sie auf Namenskonflikte, die entstehen können, wenn Sie eigene Felder zur Klasse hinzufügen, die auch mit einem Unterstrich (_) beginnen!

Komplette Eigenschaftsdeklarationen lassen sich in einer einzigen Zeile erledigen, wie es die folgenden Beispiele zeigen.

BEISPIEL 3.18: Einige selbst implementierende Eigenschaften

VB
```
Public Property FullName As String
Public Property FullName As String = "Max Muster"
Public Property ID As New Guid()
Public Property ErstesQuartal As New List(Of String) From {"Januar", "Februar", "März"}
```

Seit VB 2015 gibt es auch selbst implementierende ReadOnly-Eigenschaften. Wie bei normalen Auto-Properties können Sie jetzt auch Werte zu ReadOnly-Properties zuweisen und das auch im Konstruktor.

BEISPIEL 3.19: ReadOnly-Eigenschaften

VB
```
Class CAutor
   Public ReadOnly Property Tags As New List(Of String)
   Public ReadOnly Property Name As String = ""
   Public ReadOnly Property Datei As String

   Sub New(datei As String)
      Me.Datei = datei
   End Sub
End Class
```

3.6 Methoden

Methoden bestimmen die dynamischen Attribute eines Objekts, also sein Verhalten. Eine Methode ist eine Funktion, die im Körper der Klasse implementiert ist.

3.6.1 Öffentliche und private Methoden

Bereits im Kapitel 2 haben Sie gelernt, wie man Methoden programmiert. Jetzt wollen wir noch etwas nachhaken und den Fokus auf die Methoden richten, die in unseren selbst programmierten Klassen zum Einsatz kommen.

Genau wie das bei "richtigen" Eigenschaften der Fall ist, arbeiten in einer sauber programmierten Klasse alle Methoden ausschließlich mit privaten Feldern (Zustandsvariablen) zusammen.

HINWEIS: Wenn Sie eine Methode als *Private* deklarieren, ist sie nur innerhalb der Klasse sichtbar, und es handelt sich um keine Methode im eigentlichen Sinn der OOP, sondern eher um eine Funktion/Prozedur im herkömmlichen Sinn.

BEISPIEL 3.20: Die beiden öffentlichen Methoden *getAdresse()* und *addGuthaben()* arbeiten mit sechs privaten Feldern zusammen.

VB
```
Public Class CKunde
```

Private Variablen:

```
    Private _anrede As String
    Private _name As String
    Private _plz As Integer
    Private _ort As String
    Private _stammkunde As Boolean
```

BEISPIEL 3.20: Die beiden öffentlichen Methoden *getAdresse()* und *addGuthaben()* arbeiten mit sechs privaten Feldern zusammen.

```
    Private _guthaben As Decimal
```

Öffentliche Methoden:

```
Public Function getAdresse() As String
        Dim s As String = _anrede & " " & _name & vbCrLf & _plz.ToString & " " & _ort
        Return s
End Function

Public Sub addGuthaben(betrag As Decimal)
        If _stammkunde Then _guthaben += betrag
End Sub
```

Der Aufruf:

```
Private kunde1 As New CKunde()
...
Label1.Text = kunde1.getAdresse         ' erste Methode (Funktion) aufrufen
kunde1.addGuthaben(50)                  ' zweite Methode (Prozedur) aufrufen
```

3.6.2 Überladene Methoden

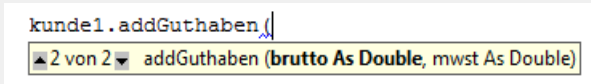
Innerhalb des Klassenkörpers dürfen zwei und mehr gleichnamige Methoden konfliktfrei nebeneinander existieren, wenn sie eine unterschiedliche Signatur (Reihenfolge und Datentyp der Übergabeparameter) besitzen.

BEISPIEL 3.21: Zwei überladene Versionen einer Methode in der Klasse *CKunde*, die erste hat nur den Nettobetrag als Parameter die zweite den Bruttobetrag und die Mehrwertsteuer.

VB

```
Public Class CKunde
    ...
    Public Sub addGuthaben(betrag As Decimal)
        If _Stammkunde Then _Guthaben += betrag
    End Sub

    Public Sub addGuthaben(brutto As Double, mwst As Double)
        If _Stammkunde Then _Guthaben += CDec(brutto / (1 + mwst))
    End Sub
    ...
End Class
```

Wenn Sie diese Methoden verwenden wollen, so fällt die Auswahl im Code-Fenster leicht:



3.6.3 Statische Methoden

Genauso wie s*tatischen Eigenschaften* können *statische Methoden* (auch als *Klassenmethoden* bezeichnet) ohne Verwendung eines Objekts aufgerufen werden. Statische Methoden werden ebenfalls mit dem *Shared*-Modifizierer gekennzeichnet und eignen sich z.B. gut für diverse Formelsammlungen (ähnlich *Math*-Klassenbibliothek). Auch können Sie damit auf private statische Klassenmitglieder zugreifen.

HINWEIS: Der Einsatz statischer Methoden für relativ einfache Aufgaben ist bequemer und ressourcenschonender als das Arbeiten mit Objekten, die Sie jedes Mal extra instanziieren müssten.

BEISPIEL 3.22: Wir bauen eine Klasse, in der wir wahllos einige von uns häufig benötigte Berechnungsformeln verpacken.

VB
```
Public Class MeineFormeln

    Public Shared Function kreisUmfang(radius As Double) As Double
        Return 2 * Math.PI * radius
    End Function

    Public Shared Function kugelVolumen(radius As Double) As Double
        Return 4 / 3 * Math.PI * Math.Pow(radius, 3)
    End Function

    Public Shared Function Netto(brutto As Decimal, mwst As Decimal) As Decimal
        Return brutto / (1 + mwst)
    End Function
    ...

End Class
```

Der Zugriff von außerhalb ist absolut problemlos, weil man sich nicht mehr um das lästige Instanziieren einer Objektvariablen kümmern muss.

HINWEIS: Leider kann bei Klassen die *With*-Anweisung nicht verwendet werden, da diese nur bei Objekten funktioniert.

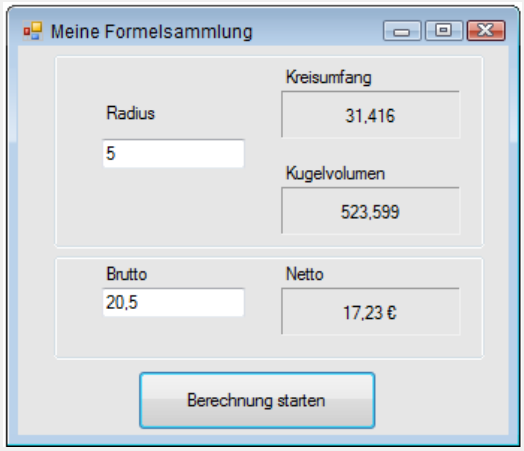
BEISPIEL 3.23: *(Fortsetzung)* Die statischen Methoden der Klasse *MeineFormeln* werden in einer Eingabemaske aufgerufen.

VB
```
Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
  Dim r As Double = Convert.ToDouble(TextBox1.Text)          ' Kreisradius konvertieren

  Label1.Text = MeineFormeln.kreisUmfang(r).ToString("0.000")
  Label2.Text = MeineFormeln.kugelVolumen(r).ToString("0.000")
```

BEISPIEL 3.23: ***(Fortsetzung)*** **Die statischen Methoden der Klasse *MeineFormeln* werden in einer Eingabemaske aufgerufen.**

```
    Dim b As Double = Convert.ToDouble(TextBox2.Text)          ' Brutto konvertieren
    Label3.Text = MeineFormeln.Netto(b, 0.19).ToString("C")
End Sub
```

Ergebnis



3.7 Ereignisse

Nachdem wir uns den Eigenschaften und Methoden von Objekten ausführlich gewidmet haben, wollen wir die Dritten im Bunde, die Ereignisse, nicht vergessen. Wie Sie bereits wissen, werden Ereignisse unter bestimmten Bedingungen vom Objekt ausgelöst und können dann in einer Ereignisbehandlungsroutine abgefangen und ausgewertet werden.

Allerdings bieten bei weitem nicht alle Klassen Ereignisse an, denn diese werden nur benötigt, wenn auf bestimmte Änderungen eines Objekts reagiert werden soll.

Nachdem wir mit dem Deklarieren von Eigenschaften und Methoden überhaupt keine Probleme hatten, hört aber bei Ereignissen der Spaß auf.

HINWEIS: Eine ausführliche Einführung in das .NET-Ereignismodell erhalten Sie im Kapitel 26 (Microsoft Event Pattern).

Im Folgenden werden deshalb nur die wichtigsten Grundlagen der Ereignismodellierung erläutert.

3.7.1 Ereignise deklarieren

Ein Ereignis fügen Sie der Klassendefinition über das *Event*-Schlüsselwort zu.

SYNTAX: `Public Event Ereignisname([Parameterdeklarationen])`

BEISPIEL 3.24: In der Klasse *CKunde* wird ein Ereignis mit dem Namen *GuthabenLeer* deklariert.

VB
```
Public Class CKunde
    Private _guthaben As Decimal

    Public Event GuthabenLeer(sender As Object, e As String)

    Public Sub New(betrag As Decimal)       ' Konstruktor
        ...
        _guthaben = betrag
    End Sub
    ...
```

3.7.2 Ereignis auslösen

Ausgelöst wird das Ereignis ebenfalls im Klassenkörper über das Schlüsselwort *RaiseEvent*.

SYNTAX: `RaiseEvent` *Ereignisname*([*Parameterwerte*])

BEISPIEL 3.25: *(Fortsetzung)* Das Ereignis *GuthabenLeer* "feuert" innerhalb der Methode *addGuthaben* dann, wenn das Guthaben den Wert von 10 € unterschreitet.

VB
```
    Public Sub addGuthaben(betrag As Decimal)
        _guthaben += betrag
        If _guthaben <= 10 Then
            Dim msg As String = "Das Guthaben beträgt nur noch" &
                                _guthaben.ToString("C") & "!"
            RaiseEvent GuthabenLeer(Me, msg)
        End If
    End Sub
    ...
End Class
```

Die Ereignisdefinition ist in diesem Beispiel bewusst einfach gehalten, um den Einsteiger nicht zu verschrecken. Normalerweise sollten Ereignisse immer zwei Parameter an die aufrufende Instanz übergeben: eine Referenz auf das Objekt, welches das Ereignis ausgelöst hat, und ein Objekt der *EvenArgs*- oder einer davon abgeleiteten Klasse (siehe 20.2.2). Auf Letzteres haben wir der Einfachheit wegen verzichtet und stattdessen einen einfachen Meldungsstring übergeben.

3.7.3 Ereignis auswerten

Um dem Compiler mitzuteilen, dass das Objekt über Ereignisse verfügt, müssen Sie bei der Referenzierung der Objektvariablen vor dem Objektbezeichner das Schlüsselwort *WithEvents* einfügen.

SYNTAX: Private|Public `WithEvents` *Objektname* As *Klassenname*

BEISPIEL 3.26: ***(Fortsetzung)*** **Wir verwenden die im Vorgängerbeispiel definierte Klasse *CKunde* in einem Formular *Form1* mit einem *Button***

VB

```
Public Class Form1
```

Der Einfachheit wegen erledigen wir die Referenzierung und die Instanziierung der Objektvariablen *kunde1* in einem Schritt und weisen dabei dem Kunden ein Anfangsguthaben von 100 Euro zu.

```
Private WithEvents kunde1 As New CKunde(100)
```

Der Eventhandler für das Ereignis *GuthabenLeer* des Kunden:

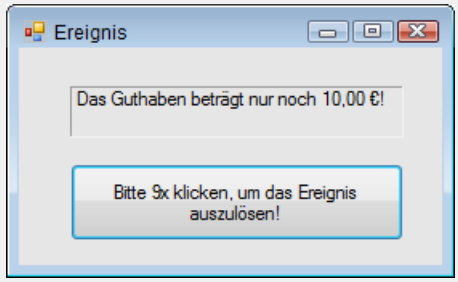
```
Private Sub kunde1_GuthabenLeer(sender As Object, e As String) Handles kunde1.GuthabenLeer
    Label1.Text = e.ToString
End Sub

End Class
```

Bei jedem Klick auf den *Button* wird das Guthaben des Kunden um 10 € verringert:

```
Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
    Kunde1.addGuthaben(-10)
End Sub
```

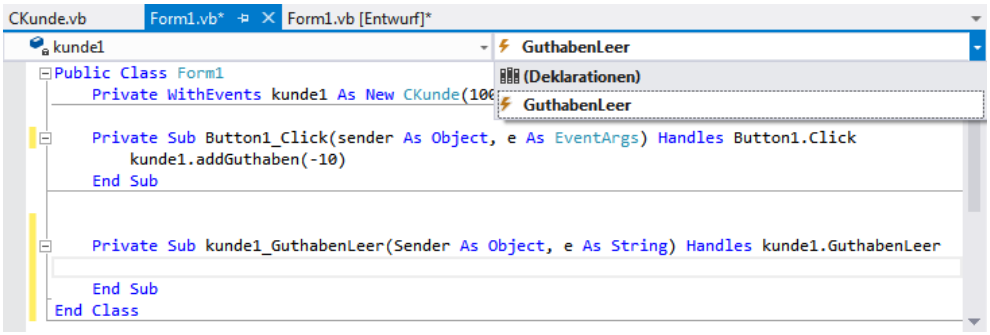
Ergebnis

Nachdem Sie den *Button* neunmal geklickt haben, feuert das Ereignis und im *Label* erscheint eine Meldung:



Visual Studio unterstützt Sie beim Erstellen des Rahmencodes der Event-Handler für selbst definierte Ereignisse natürlich genauso, wie Sie das z.B. für das *Click*-Ereignis eines *Button*s zur Genüge gewöhnt sind.

Nach dem Deklarieren der Objektvariablen wird in der linken oberen Klappbox das Objekt *kunde1* selektiert und in der rechten das Ereignis *GuthabenLeer*. Der Rahmencode des Eventhandlers wird automatisch generiert:



3.7.4 Benutzerdefinierte Ereignisse (Custom Events)

Meistens deklarieren Sie ein Ereignis mittels *Event*-Schlüsselwort unter Angabe eines Ereignisdelegaten, die Codegenerierung für die Ereignisverwaltung erfolgt automatisch.

BEISPIEL 3.27: Standardmäßige Ereignisdefinition in zwei Schritten

VB
```
Public Delegate Sub NumberChangedHandler(i As Integer)
Public Event NumberChanged As NumberChangedHandler
```

In der Regel ist diese Art der Deklaration auch völlig ausreichend, aber in einigen Fällen möchten Sie die Ereignisverwaltung doch lieber selbst in die Hand nehmen.

Benutzerdefinierte Ereignis-Accessoren (Custom Events) erlauben dem Programmierer die genaue Definition der Vorgänge, die beim Hinzufügen bzw. Entfernen eines Eventhandlers und beim Auslösen eines Events ablaufen sollen (siehe dazu auch die Ausführungen zum Microsoft Event Pattern im Kapitel 26).

Deklaration

Durch Deklarieren eines Custom Events wird dem Compiler mitgeteilt, dass er die Codegenerierung für die Ereignisverwaltung außer Kraft setzen soll, der Programmierer muss sich also nun selbst darum kümmern wie die Ereignishandler an-/abgemeldet und aufgerufen werden.

Ein Custom Event wird durch das der *Event*-Deklaration vorangestellte *Custom*-Schlüsselwort unter Angabe eines Ereignisdelegaten definiert.

BEISPIEL 3.28: Deklaration eines benutzerdefinierten Ereignisses *NumberChanged*

VB
```
Public Delegate Sub NumberChangedHandler(i As Integer)
Public Custom Event NumberChanged As NumberChangedHandler
```

Wenn Sie nach einer solchen Ereignisdeklaration die *Enter*-Taste drücken, erstellt Visual Studio automatisch den Rahmencode für die Ereignis-Acessoren (auf ähnliche Weise wie für Properties):

```
Public Custom Event NumberChanged As NumberChangedHandler
    AddHandler(value As NumberChangedHandler)

    End AddHandler

    RemoveHandler(value As NumberChangedHandler)

    End RemoveHandler

    RaiseEvent(i As Integer)

    End RaiseEvent
End Event
```

Wie Sie sehen, besteht die Deklaration eines Custom Events aus drei Sektionen, die unter folgenden Bedingungen abgearbeitet werden:

- *AddHandler*, wenn ein Ereignishandler entweder mittels *Handles*-Schlüsselwort oder *AddHandler*-Methode hinzugefügt wird.
- *RemoveHandler*, wenn ein Handler mittels *RemoveHandler*-Methode wieder entfernt wird.
- *RaiseEvent*, wenn ein Ereignis ausgelöst wird.

Sie müssen sich jetzt also selbst um das Hinzufügen und Entfernen der Handler zu bzw. aus ihrem Container (üblicherweise eine Collection), sowie um die Benachrichtigung der angeschlossenen Handler beim Auslösen des Ereignisses kümmern. Obwohl das mehr Codezeilen erfordert als eine standardmäßige Implementierung, haben Sie eine größere Flexibilität beim Programmieren.

Anwendung

Da das alles ziemlich verwirrend klingen mag, soll ein Beispiel Licht in die Dunkelheit bringen.

BEISPIEL 3.29: Benutzerdefinierte Ereignisse

VB

Eine Klasse *CCounter* erzeugt im Sekundentakt eine Zahl von 1 bis 10 und löst dabei das benutzerdefinierte Ereignis *NumberChanged* aus. Als Container für die angemeldeten Eventhandler dient eine (generische) *List*:

```
Public Class CCounter

        Public Delegate Sub NumberChangedHandler(i As Integer)
        Private handlers As New List(Of NumberChangedHandler)

        Public Custom Event NumberChanged As NumberChangedHandler
            AddHandler(value As NumberChangedHandler)
                If handlers.Count <= 3 Then handlers.Add(value)
            End AddHandler
```

BEISPIEL 3.29: Benutzerdefinierte Ereignisse

```
            RemoveHandler(value As NumberChangedHandler)
                handlers.Remove(value)
            End RemoveHandler

            RaiseEvent(i As Integer)
                If i > 30 Then
                    For Each handler As NumberChangedHandler In handlers
                        handler.Invoke(i)
                    Next
                End If
            End RaiseEvent
        End Event
```

Die Methode, in welcher das Ereignis ausgelöst wird:

```
        Public Sub DoCount()
            For i As Integer = 1 To 10
                System.Threading.Thread.Sleep(1000)
                RaiseEvent NumberChanged(i * 10)
            Next
        End Sub
    End Class
```

Ein Eventhandler zeigt die erzeugten Zahlen in einer *ListBox* an:

```
    Private Sub c_NumberChanged(i As Integer)
        ListBox1.Items.Add(i.ToString)
    End Sub
```

Schließlich der Test der Klasse *CCounter*:

```
    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
```

Instanziierung:

```
        Dim c As New CCounter
```

Eventhandler anmelden:

```
        AddHandler c.NumberChanged, AddressOf c_NumberChanged
```

Methode aufrufen, die das Ereignis auslöst:

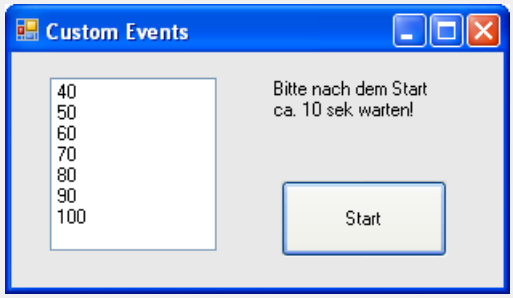
```
        c.DoCount()
```

Eventhandler abmelden:

```
        RemoveHandler c.NumberChanged, AddressOf c_NumberChanged
    End Sub
```

BEISPIEL 3.29: Benutzerdefinierte Ereignisse

Ergebnis



Wie Sie an diesem Beispiel sehen, ermöglichen Ihnen benutzerdefinierte Ereignisse eine freizügige Programmierung der Verwaltung Ihrer Ereignishandler. Im obigen Code haben wir als Container für die Ereignisdelegaten eine typisierte *List* gewählt, wir hätten aber auch z.B. eine (untypisierte) *ArrayList* nehmen können. In die *AddHandler*-Sektion wurde von uns eine Bedingung eingebaut, die die Anzahl der angeschlossenen Eventhandler auf 3 limitiert. Die *RaiseEvent*-Sektion bewirkt, dass die angeschlossenen Handler nur dann benachrichtigt werden, wenn die übergebene Zahl größer als 30 ist.

Ein nicht zu unterschätzender Vorteil von Custom Events ist auch das Vorhandensein eines zentralen Containers (im Beispiel die *handlers*-Liste), welcher alle angemeldeten Eventhandler innerhalb der Eventdeklaration kapselt.

Die doppelte Verwendung von *RaiseEvent* (innerhalb der *DoCount*-Methode und innerhalb der *Custom Event*-Ereignisdefinition) kann möglicherweise Verwirrung stiften. Deshalb sei hier nochmals auf den grundlegenden Unterschied hingewiesen: *RaiseEvent* in der *DoCount*-Methode löst das *NumberChanged*-Ereignis aus, die *RaiseEvent*-Sektion hingegen spezifiziert den beim Auslösen des Ereignisses auszuführenden "organisatorischen" Code, d.h., alle im Container (*List*) enthaltenen Eventhandler (Delegates) werden durchlaufen und aufgerufen.

3.8 Arbeiten mit Konstruktor und Destruktor

Eine "richtige" objektorientierte Sprache, zu der Visual Basic ja mittlerweile auch gehört, realisiert das Erzeugen und Entfernen von Objekten mit Hilfe von Konstruktoren und Destruktoren.

HINWEIS: Wenn Sie in einigen bisherigen Beispielen in den von Ihnen selbst entwickelten Klassen keinen eigenen Konstruktor definiert hatten, so wurde automatisch der von *System.Object* geerbte parameterlose *New*-Standardkonstruktor verwendet.

3.8.1 Der Konstruktor erzeugt das Objekt

Der Konstruktor ist gewissermaßen die Standardmethode der Klasse und kann in mehreren Überladungen vorhanden sein.

HINWEIS: Der Konstruktor ist immer eine *Sub* mit dem Namen *New()*.

```
SYNTAX:  Public Sub New([Parameterliste])
             ' Parameter den Feldern der Klasse zuweisen
             ' Initialisierungscode ausführen
         End Sub
```

Der Konstruktor wird automatisch bei der Instanziierung eines Objekts aufgerufen und dient vor allem dazu, den Feldern des neu erzeugten Objekts Anfangswerte zuzuweisen.

HINWEIS: Nachdem Sie einer Klasse einen oder mehrere Konstruktoren hinzugefügt haben, sind Sie auch zur Verwendung von mindestens einem davon verpflichtet. Die bisher gewohnte einfache Instanziierung von Objekten ist nicht mehr möglich, d.h., der von *System.Object* geerbte parameterlose Konstruktor steht nicht mehr zur Verfügung!

Deklaration

Einen Konstruktor fügen Sie dem Klassenkörper ähnlich wie eine *Set*-Eigenschaftsprozedur mit dem Namen *New* hinzu. Als Parameter übergeben Sie die Werte für die Felder, die Sie initialisieren möchten.

Wie bei jeder anderen Methode können Sie auch hier mehrere überladene Konstruktoren implementieren.

BEISPIEL 3.30: Unserer Klasse *CKunde* werden zwei überladene Konstruktoren hinzugefügt.

VB

```
Public Class CKunde
```

Die (natürlich privaten) Zustandsvariablen:

```
    Private _anrede As String
    Private _name As String
    Private _plz As Integer
    Private _ort As String
    Private _stammkunde As Boolean
    Private _guthaben As Decimal
```

Der erste Konstruktor initialisiert nur zwei private Variablen:

```
    Public Sub New(Anrede As String, Nachname As String)
        _anrede = Anrede
        _name = Nachname
    End Sub
```

Der zweite Konstruktor initialisiert alle Variablen der Klasse. Um den Code etwas zu kürzen, wird für die ersten beiden Variablen der erste Konstruktor bemüht:

```
    Public Sub New(Anrede As String, Nachname As String, PLZ As Integer,
                   Ort As String, Stammkunde As Boolean, Guthaben As Decimal)
        Me.New(Anrede, Nachname)
```

BEISPIEL 3.30: Unserer Klasse *CKunde* werden zwei überladene Konstruktoren hinzugefügt.

```
        _plz = PLZ
        _ort = Ort
        _stammkunde = Stammkunde
        _guthaben = Guthaben
    End Sub

End Class
```

Aufruf

Da wir der Klasse *CKunde* zwei Konstruktoren hinzugefügt haben, ist die bisher gewohnte parameterlose Instanziierung von Objekten mit dem *New()*-Standardkonstruktor nicht mehr möglich (es sei denn, Sie fügen selbst eine weitere Überladung hinzu, die keine Parameter entgegen nimmt)!

BEISPIEL 3.31: (Fortsetzung) Zwei Objekte der oben deklarierten Klasse *CKunde* werden erzeugt und mit Anfangswerten initialisiert. Für jedes Objekt wird ein anderer überladener Konstruktor verwendet.

VB
```
Try
      Dim kunde1 As New CKunde("Herr", "Müller")
      Dim kunde2 As New CKunde("Frau", "Hummel", 12345, "Berlin", True, 100)
      ' Dim kunde3 As New CKunde()             ' geht nicht mehr!!!
      MessageBox.Show("Objekte erfolgreich erzeugt!")
Catch ex As Exception
      MessageBox.Show("Fehler beim Erzeugen des Objekts!")
End Try
```

Sie sehen, dass das Initialisieren der Objekte viel einfacher geworden ist. Anstatt umständlich eine Eigenschaft nach der anderen zuzuweisen, geht das jetzt mit einer einzigen Anweisung.

3.8.2 Bequemer geht's mit einem Objekt-Initialisierer

Vor allem in Hinblick auf die in der neuen LINQ-Technologie erforderlichen anonymen Typen (siehe Kapitel 6) wurden so genannte Objekt-Initialisierer eingeführt. Damit können nun öffentliche Eigenschaften und Felder von Objekten ohne das explizite Vorhandensein des jeweiligen Konstruktors in beliebiger Reihenfolge initialisiert werden.

Der Objekt-Initialisierer erwartet das Schlüsselwort *With* unmittelbar nach dem Erzeugen des Objekts. Anschließend folgt die in geschweiften Klammern eingeschlossene Liste der zu initialisierenden Mitglieder.

HINWEIS: Einem Objektinitialisierer können nur Eigenschaften oder öffentliche Felder übergeben werden, das Initialisieren privater Felder, wie im obigen Konstruktor-Beispiel, ist nicht möglich!

BEISPIEL 3.32: Erzeugen einer Instanz der Klasse *CPerson*

VB
```
Public Class CPerson
   Public Name As String
   Public Strasse As String
   Public PLZ As Integer
   Public Ort As String
End Class
```

Der Objektinitialisierer:

```
Dim person1 As New CPerson With {.Name = "Müller", .Strasse = "Am Waldesrand 7",
                                 .PLZ = 12345, .Ort = "Musterhausen"}
```

BEISPIEL 3.33: Verschachtelte Objektinitialisierung beim Erzeugen einer Instanz der Klasse *Rectangle*

VB
```
Dim rect As New Rectangle With {.Location = New Point With {.X = 3, .Y = 7},
                            .Size = New Size With {.Width = 19, .Height = 34} }
```

BEISPIEL 3.34: Initialisieren einer Collection aus Objekten

VB
```
Dim personen() = {New CPerson With {.Name = "Müller", .Strasse = "Am Wald 7",
                                    .PLZ = 12345, .Ort = "Musterhausen"},
                  New CPerson With {.Name = "Meier", .Strasse = "Hauptstr. 2",
                                    .PLZ = 2344, .Ort = "Walldorf"},
                  New CPerson With {.Name = "Schulz", .Strasse = "Wiesenweg 5",
                                    .PLZ = 32111, .Ort = "Biesdorf"}}
```

3.8.3 Destruktor und Garbage Collector räumen auf

Das Pendant zum Konstruktor ist aus objektorientierter Sicht der Destruktor. Da der Lebenszyklus eines Objektes bekanntlich mit dessen Zerstörung und der Freigabe der belegten Speicherplatzressourcen endet, ist der Destruktor für das Erledigen von "Aufräumarbeiten" zuständig, kurz bevor das Objekt sein Leben aushaucht.

In .NET haben wir allerdings keine echten Destruktoren, da hier die endgültige Zerstörung eines Objekts nicht per Code, sondern automatisch vom Garbage Collector vorgenommen wird. Dieser durchstöbert willkürlich und in unregelmäßigen Zeitabständen den Heap nach Objekten, um diejenigen zu suchen, die nicht mehr referenziert werden.

An die Stelle eines echten Destruktors tritt ein Quasi-Destruktor. Das ist eine Finalisierungsmethode, die zu einem unbestimmbaren Zeitpunkt vom Garbage Collector aufgerufen wird, kurz bevor dieser das Objekt vernichtet.

Der *Public*-Zugriffsmodifizierer entfällt, da Sie selbst den Destruktor nicht aufrufen dürfen, auch Parameter dürfen nicht übergeben werden.

SYNTAX:
```
Protected Overrides Sub Finalize()
        ' hier Code für Aufräumarbeiten implementieren
End Sub
```

BEISPIEL 3.35: Destruktor und Garbage Collector

VB

Unsere Klasse *CKunde* erhält ein öffentliches statisches Feld, welches durch den Konstruktor inkrementiert und durch den Quasi-Destruktor dekrementiert werden soll. Wir beabsichtigen damit, die Anzahl der momentan instanziierten Klassen (sprich Anzahl der Kunden) abzufragen.

Der auf das Wesentliche reduzierte Code von *CKunde*:

```
Public Class CKunde
    Public Shared anzahl As Integer = 0
```

Konstruktor:

```
    Public Sub New()
        anzahl += 1
    End Sub
```

Quasi-Destruktor:

```
    Protected Overrides Sub Finalize()
        anzahl -= 1
        MyBase.Finalize()        ' Aufruf der Basisklassenmethode
    End Sub

End Class
```

Wir verwenden zum Testen der Klasse ein Windows-Formular mit zwei *Button*s, einer *Timer*-Komponente (*Interval* = 1000, *Enabled* = *True*) und einem *Label*.

Zum Code der Klasse *Form1* fügen Sie hinzu:

```
    Private kunde1 As CKunde            ' Objekt referenzieren
```

Objekt erzeugen:

```
    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        kunde1 = New CKunde
    End Sub
```

Objekt entfernen:

```
    Private Sub Button2_Click(sender As Object, e AsEventArgs) Handles Button2.Click
        kunde1 = Nothing                 ' Objekt dereferenzieren
    End Sub
```

Anzeige der im Speicher befindlichen Instanzen im Sekundentakt:

```
    Private Sub Timer1_Tick(sender As Object, e As EventArgs) Handles Timer1.Tick
        Label1.Text = CKunde.anzahl.ToString
    End Sub
```

BEISPIEL 3.35: Destruktor und Garbage Collector

Ergebnis

Beim Programmtest müssen Sie etwas Geduld aufbringen.

Nach dem Programmstart fügen Sie durch Klicken auf den linken *Button* ein Objekt *kunde1* hinzu, wonach sich die Anzeige von 0 auf 1 ändert. Anschließend klicken Sie auf den rechten Button, um das Objekt wieder zu entfernen.

Es kann ziemlich lange dauern, bis die Anzeige wieder auf 0 zurück geht, nämlich dann, wenn dem Garbage Collector gerade einmal wieder die Lust zum Aufräumen überkommt und er den Quasi-Destruktor aufruft[1].

Übrigens können Sie auch den linken Button mehrmals hintereinander klicken. Die Anzeige zählt zwar hoch, das aber täuscht, denn es bleibt bei nur einer Objektvariablen (*kunde1*). Allerdings wird Ressourcenverschwendung betrieben, denn dem Objekt wird immer wieder ein neuer Speicherbereich zugewiesen. Der vorher belegte Speicher liegt brach und wartet auf die Freigabe durch den Garbage Collector.

HINWEIS: Obiges Beispiel sollten Sie aufgrund seiner Unberechenbarkeit keinesfalls als Vorbild für ähnliche Zählaufgaben verwenden!

Da wegen der Unberechenbarkeit der Objektvernichtung der Umgang mit der *Finalize*-Methode ziemlich problematisch ist, sollten Sie für das definierte Freigeben von Objekten besser eine separate Methode verwenden (siehe *Dispose*-Methode).

3.8.4 Mit Using den Lebenszyklus des Objekts kapseln

Auch mit dem Schlüsselwort *Using* kann man selbst für das sichere Erzeugen und Vernichten von Objekten sorgen. Voraussetzung dafür ist, dass das Objekt die *IDisposable*-Schnittstelle implemen-

[1] Der Garbage Collector läuft in einem eigenen Thread, er wird nur dann aufgerufen, wenn sich die anderen Threads in einem sicheren Zustand befinden.

tiert. Hinter den Kulissen wird ein *Try-Finally*-Block um das entsprechende Objekt generiert und beim Beendigen für das Objekt *Dispose()* aufgerufen.

BEISPIEL 3.36: Sicheres Erzeugen und Freigeben von ADO.NET-Objekten (siehe 23.3.5)

VB
```
...
Using conn As New SqlConnection(connString)
   Using cmd As New SqlCommand(cmdString, conn)
      conn.Open()
      cmd.ExecuteNonQuery()
   End Using
End Using
...
```

Ein weiteres Beispiel für *Using* finden Sie im Praxisbeispiel

► 8.8.3 Ein Memory Mapped File verwenden

3.9 Vererbung und Polymorphie

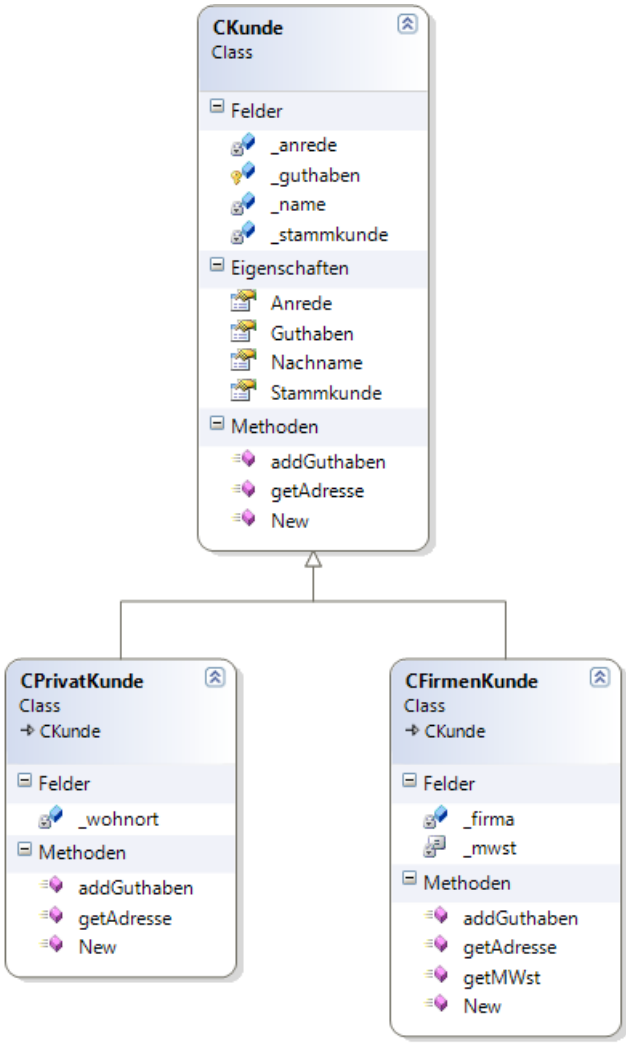
Ein zentrales OOP-Thema ist die *Vererbung*, die es ermöglicht, Klassen zu definieren, die von anderen Klassen abhängen. Eng mit der Vererbung verknüpft ist die *Polymorphie* (Vielgestaltigkeit). Man versteht darunter die Fähigkeit von Subklassen, die Methoden der Basisklasse mit unterschiedlichen Implementierungen zu verwenden. Visual Basic unterstützt sowohl Vererbung als auch polymorphes Verhalten, da das Überschreiben (*Overriding*) der Basisklassenmethoden mit alternativen Implementierungen erlaubt ist.

Durch Vererbung können Sie sich die Programmierarbeit wesentlich erleichtern, indem Sie spezialisierte Subklassen verwenden, die den Code zum großen Teil von einer allgemeinen Basisklasse erben. Die Subklassen heißen auch *abgeleitete Klassen, Kind-* oder *Unterklassen*, die Basisklasse wird auch als *Super-* oder *Elternklasse* bezeichnet. In den Subklassen können Sie bestimmte Funktionalitäten überschreiben, um spezielle Prozesse auszuführen.

Lassen Sie uns anhand eines kurzen und dennoch ausführlichen Beispiels die wichtigsten Vererbungstechniken demonstrieren!

3.9.1 Vererbungsbeziehungen im Klassendiagramm

Mittels *Unified Modeling Language* (UML) lassen sich Vererbungsbeziehungen zwischen verschiedenen Klassen grafisch darstellen.



Das obige, mit Visual Studio erzeugte, Klassendiagramm zeigt eine Basisklasse *CKunde*, von der die Klassen *CPrivatKunde* und *CFirmenKunde* "erben".

Die Basisklasse hat die Eigenschaften *Anrede*, *Nachname*, *StammKunde* (ja/nein) und *Guthaben* und die Methoden *getAdresse()* und *addGuthaben()* (das Guthaben ist hier als Bonus zu verstehen, der den Kunden in prozentualer Abhängigkeit von den getätigten Einkäufen gewährt wird). Die *New*-Methoden sind nichts weiter als die Konstruktoren der entsprechenden Klassen.

3.9.2 Überschreiben von Methoden (Method-Overriding)

Die Subklassen *CPrivatKunde* und *CFirmenKunde* können auf sämtliche Eigenschaften und Methoden der Basisklasse zugreifen und fügen selbst eigene Methoden (auch Eigenschaften wären natürlich möglich) hinzu.

Die "geerbten" Methoden *getAdresse* und *addGuthaben* tauchen allerdings nochmals in den beiden Subklassen auf, warum? In unserem Beispiel handelt es sich um so genannte *überschriebene Methoden*, d.h., Adresse und Guthaben sollen für Privatkunden auf andere Weise als für Firmenkunden ermittelt werden. Genaueres dazu erfahren Sie im nächsten Abschnitt.

HINWEIS: Verwechseln Sie das *Überschreiben* von Methoden nicht mit dem in 2.8.5 beschriebenen *Überladen* von Methoden. Beides hat nichts, aber auch gar nichts, miteinander zu tun!

3.9.3 Klassen implementieren

Vorbild für die drei zu implementierenden Klassen ist obiges Klassendiagramm.

Basisklasse CKunde

Die Deklaration entspricht (fast) der einer normalen Klasse. Dass es sich um eine Basisklasse handelt, erkennt man in unserem konkreten Fall eigentlich nur an dem *Protected*-Feld und an den *Overridable*-Methodendeklarationen[1].

```
Public Class CKunde
```

Die privaten Felder:

```
    Private _anrede As String
    Private _name As String
    Private _stammkunde As Boolean
```

Auf das folgende Feld soll auch eine Subklasse zugreifen können:

```
    Protected _guthaben As Decimal
```

Der eigene Konstruktor:

```
    Public Sub New(Anrede As String, Nachname As String)
        _anrede = Anrede
        _name = Nachname
    End Sub
```

Die Eigenschaften:

```
    Public WriteOnly Property Anrede()
        Set(value)
```

[1] Eigentlich hätten wir die Klasse auch noch als *MustInherit* deklarieren müssen (siehe dazu 3.6.6).

```
            _anrede = value
        End Set
    End Property

    Public WriteOnly Property Nachname()
        Set(value)
            _name = value
        End Set
    End Property

    Public Property Stammkunde() As Boolean
        Get
            Return _stammkunde
        End Get
        Set(value As Boolean)
            _stammkunde = value
        End Set
    End Property

    Public ReadOnly Property Guthaben() As Decimal
        Get
            Return _guthaben
        End Get
    End Property
```

Die folgenden beiden Methoden sollen von den Subklassen überschrieben werden können:

```
    Public Overridable Function getAdresse() As String
        Return _anrede & " " & _name
    End Function

    Public Overridable Sub addGuthaben(betrag As Decimal)
        If _stammkunde Then _guthaben += betrag
    End Sub

End Class
```

Subklassen

Die erste Methode in der Subklasse ist in der Regel der Konstruktor. Dieser Konstruktor benutzt das *MyBase*-Schlüsselwort, um den Konstruktor der Basisklasse aufzurufen. Falls aber die Basisklasse über keinen eigenen Konstruktor verfügt, wird der Standardkonstruktor automatisch aufgerufen, wenn ein Objekt aus einer Subklasse erzeugt wird.

Das *Overrides*-Schlüsselwort der beiden Funktionen bedeutet, dass hier die in der Basisklasse definierten Funktionen überschrieben werden. Das erlaubt der Subklasse, eine eigene Implementierung der Funktionen zu realisieren.

HINWEIS: Wenn Sie das *Overrides*-Schlüsselwort in der Subklasse vergessen wird angenommen, dass es sich um eine "Schattenfunktion" der originalen Funktion handelt. Eine solche Funktion hat denselben Namen wie das Original, überschreibt dieses aber nicht.

Der Code für die Subklasse *CPrivatKunde:*

```
Public Class CPrivatKunde
    Inherits CKunde
    Private _wohnort As String
```

Der Konstruktor ist notwendig, weil auch die Basisklasse einen eigenen Konstruktor verwendet:

```
    Public Sub New(Anr As String, Name As String, Ort As String)
        MyBase.New(Anr, Name)  ' Aufruf des Konstruktors der Basisklasse
        Me._wohnort = Ort      ' klassenspezifische Ergänzung
    End Sub
```

Die Methoden werden überschrieben:

```
    Public Overrides Function getAdresse() As String
        Return MyBase.Adresse & vbCrLf & _wohnort
    End Function

    Public Overrides Sub addGuthaben(geld As Decimal)
```

5% des Rechnungsbetrags werden als Guthaben angerechnet (Direktzugriff auf die *Protected*-Variable der Basisklasse *CKunde*):

```
        _guthaben += 0.05 * geld
     End Sub
End Class
```

Der Code für die Subklasse *CFirmenKunde* unterscheidet sich in einigen Details:

```
Public Class CFirmenKunde
    Inherits CKunde
    Private _firma As String
    Private Const _mwst As Double = 0.19
```

Konstruktor (notwendig, weil Basisklasse eigenen Konstruktor verwendet):

```
    Public Sub New(Anr As String, Name As String, Frm As String)
        MyBase.New(Anr, Name)           ' Aufruf des ererbten Konstruktors
        Me._firma = Frm
    End Sub
```

Überschreiben der überschreibbaren Methoden:

```
    Public Overrides Function getAdresse() As String
        Return MyBase.Adresse & vbCrLf & _firma
    End Function
```

```
    Public Overrides Sub addGuthaben(brutto As Decimal)
        Dim netto As Decimal = brutto / CDec((1 + _mwst))         ' Netto berechnen
        MyBase.addGuthaben(netto * 0.01D)        ' 1% als Guthaben angerechnet
    End Sub
```

Eine normale Methode:

```
    Public Function getMWst() As Double
        Return _mwst
    End Function

End Class
```

Subklasse CPrivatKunde

Diese Klasse erbt alle Eigenschaften und Methoden der Basisklasse, wird also sozusagen um deren Code "erweitert". Das *Overrides*-Schlüsselwort der beiden Methoden bedeutet, dass hier die in der Basisklasse als *Overridable* definierten Funktionen überschrieben werden. Das erlaubt der Subklasse, eine eigene Implementierung der Funktionen zu realisieren.

Der Code für die Subklasse *CPrivatKunde:*

```
Public Class CPrivatKunde
    Inherits CKunde                          ' erbt von der Basisklasse CKunde!
    Private _wohnort As String
```

Ein eigener Konstruktor ist notwendig, weil auch die Basisklasse einen eigenen Konstruktor verwendet:

```
    Public Sub New(Anr As String, Name As String, Ort As String)
        MyBase.New(Anr, Name)                 ' Aufruf des Konstruktors der Basisklasse
        Me._wohnort = Ort      ' klassenspezifische Ergänzung
    End Sub
```

Die Methode *getAdresse()* wird so überschrieben, dass zusätzlich zu Anrede und Name (von der Basisklasse geerbt) noch der Wohnort des Privatkunden angezeigt wird:

```
    Public Overrides Function getAdresse() As String
        Return MyBase.getAdresse & vbCrLf & _wohnort
    End Function
```

Die Methode *addGuthaben()* wird komplett neu überschrieben. Ohne Rücksicht auf die Zugehörigkeit zur Stammkundschaft werden jedem Privatkunden 5% vom Rechnungsbetrag als Bonusguthaben angerechnet:

```
    Public Overrides Sub addGuthaben(geld As Decimal)
```

Hier erfolgt ein Direktzugriff auf die *Protected*-Variable *_guthaben* der Basisklasse *CKunde*:

```
        _guthaben += 0.05 * geld
    End Sub

End Class
```

Subklasse CFirmenKunde

Der Code für die Subklasse *CFirmenKunde* unterscheidet sich in folgenden Details von der Klasse *CPrivatKunde*:

- Die Methode *getAdresse()* liefert statt des Wohnorts den Namen der Firma des Kunden.
- Die *addGuthaben()*-Methode berechnet zunächst den Nettobetrag und addiert davon 1% zum Bonusguthaben. Damit nur Stammkunden in den Genuss dieser Vergünstigung kommen, wird dazu die gleichnamige Methode der Basisklasse aufgerufen.
- Die neu hinzugekommene "stinknormale" Methode *getMWSt()* erlaubt einen Lesezugriff auf die Mehrwertsteuer-Konstante.

```
Public Class CFirmenKunde
    Inherits CKunde

    Private _firma As String
    Private Const _mwst As Double = 0.19
```

Auch hier ist ein Konstruktor notwendig (weil Basisklasse eigenen Konstruktor verwendet):

```
    Public Sub New(Anr As String, Name As String, Frm As String)
        MyBase.New(Anr, Name)           ' Aufruf des ererbten Konstruktors
        Me._firma = Frm
    End Sub
```

Überschreiben der überschreibbaren Methoden:

```
    Public Overrides Function getAdresse() As String
        Return MyBase.getAdresse & vbCrLf & _firma
    End Function

    Public Overrides Sub addGuthaben(brutto As Decimal)
        Dim netto As Decimal = brutto / CDec((1 + _mwst))  ' Netto berechnen
        MyBase.addGuthaben(netto * 0.01)                   ' 1% als Guthaben angerechnet
    End Sub
```

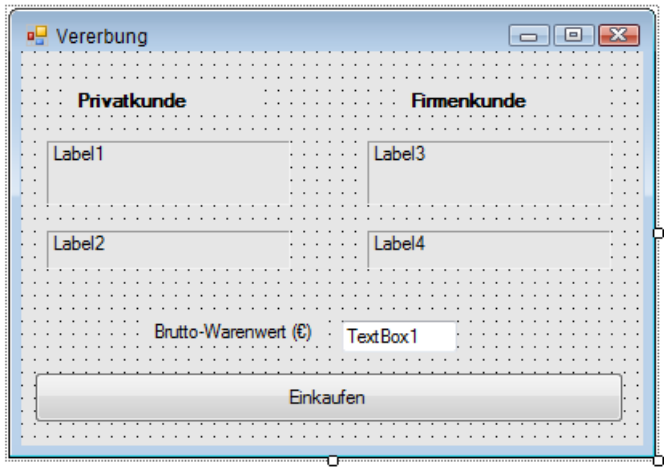
Und zum Schluss noch eine ganz normale Methode:

```
    Public Function getMWst() As Double
        Return _mwst
    End Function
End Class
```

Die Implementierung unserer drei Klassen ist geschafft!

Testoberfläche

Um die Funktionsfähigkeit der drei Klassen zu testen, gestalten Sie folgende Benutzerschnittstelle:



3.9.4 Objekte implementieren

Es genügt, wenn wir mit nur zwei Objekten (ein Privat- und ein Firmenkunde) arbeiten:

```
Public Class Form1
    Private kunde1 As CPrivatKunde
    Private kunde2 As CFirmenKunde
```

Im Konstruktor des Formulars werden die beiden Objekte erzeugt. Fügen Sie zunächst den Rahmencode des Konstruktors hinzu, indem Sie in den beiden Comboboxen am oberen Rand des Codefenster Folgendes einstellen: *Klassenname = Form1*, *Methodenname = New.*

Die Ja-/Nein-Eigenschaft *StammKunde* muss allerdings extra zugewiesen werden, da es dazu keinen passenden Konstruktor gibt.

```
Public Sub New()

        ' Dieser Aufruf ist für den Windows Form-Designer erforderlich.
        InitializeComponent()
        ' Fügen Sie Initialisierungen nach dem InitializeComponent()-Aufruf hinzu.

        kunde1 = New CPrivatKunde("Herr", "Krause", "Leipzig")
        kunde1.Stammkunde = False

        kunde2 = New CFirmenKunde("Frau", "Müller", "Master Soft GmbH")
        kunde2.Stammkunde = True
        TextBox1.Text = "100"
    End Sub
```

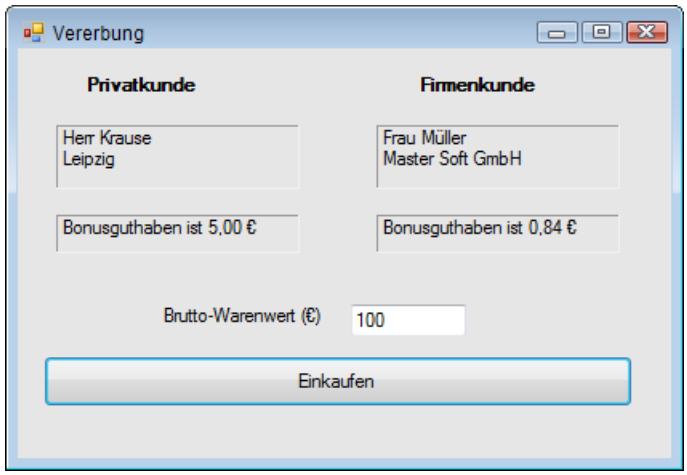
Bei Klick auf den "Einkaufen"-Button werden für jedes Objekt diverse Eigenschaften abgefragt und Methoden aufgerufen:

```
    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        Dim brutto As Decimal = Convert.ToDecimal(TextBox1.Text)
```

```
        Label1.Text = kunde1.getAdresse()
        kunde1.addGuthaben(brutto)
        Label2.Text = "Bonusguthaben ist " + kunde1.Guthaben.ToString("C")
        Label3.Text = kunde2.getAdresse()
        kunde2.addGuthaben(brutto)
        Label4.Text = "Bonusguthaben ist " + kunde2.Guthaben.ToString("C")
    End Sub
End Class
```

Praxistest

Überzeugen Sie sich nun davon, dass die drei Klassen wie gewünscht zusammenarbeiten und dass Vererbung tatsächlich funktioniert.



Die Werte in der Laufzeitabbildung sind wie folgt zu interpretieren:

- Dem Privatkunden Krause wurde ein Guthaben von 5 € (5% aus 100 €) zugebilligt (Stammkundschaft spielt bei Privatkunden keine Rolle, da die Methode *addGuthaben()* komplett überschrieben ist).
- Frau Müller ist eine Firmenkundin und erhält – nur weil sie Stammkundin ist – ein mickriges Guthaben von 0,84 € (1% auf den Nettowert).
- Durch wiederholtes Klicken auf "Einkaufen" kumulieren die Bonusguthaben.

3.9.5 Ausblenden von Mitgliedern durch Vererbung

Durch in eine abgeleitete Klasse oder Struktur eingeführte Mitglieder (Konstanten, Felder, Eigenschaften, Methoden, Ereignisse oder Typen) werden alle gleichnamigen Basisklassenelemente verdeckt bzw. ausgeblendet.

BEISPIEL 3.37: Zur Klasse *CKunde* fügen wir eine Methode *test* hinzu

VB

```
Public Class CKunde                    ' Basisklasse
    ...
    Public Sub test()
        MessageBox.Show("Hallo Kunde!")
    End Sub
End Class
```

Eine Methode gleichen Namens fügen wir auch zur Klasse *CPrivatkunde* hinzu:

```
Public Class CPrivatkunde              ' abgeleitete Klasse
    Inherits CKunde

    Public Sub test()
        MessageBox.Show("Hallo Privatkunde!")
    End Sub

End Class
```

Im Quellcode-Editor erscheint der Name *test()* grün unterschlängelt. Der entsprechende Warnhinweis lautet: *Sub "test" führt Shadowing für einen überladbaren Member durch, der in der Basisklasse CKunde deklariert ist. Wenn Sie die Basismethode überladen möchten, muss die Methode als "Overloads" deklariert werden.*

Sie lassen diese Warnung unbeachtet. Der Test erfolgt in *Form1*:

```
Public Class Form1
 Private kunde1 As New CPrivatkunde()

 Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
     kunde1.test()
 End Sub
End Class
```

Ergebnis

Wie Sie sehen, wurde die Methode *test()* der Klasse *CKunde* durch die Methode *test()* der Klasse *CPrivatKunde* ausgeblendet:

Um Missverständnissen vorzubeugen (und um obigen Warnhinweis im Quellcode zu vermeiden), sollte man die gleichnamige Methode in der abgeleiteten Klasse mit dem Schlüsselwort *Overloads* markieren.

BEISPIEL 3.38: Die folgende Änderung macht das Vorgängerbeispiel transparenter (Ergebnis bleibt dasselbe)

VB
```
Public Class CPrivatkunde            ' abgeleitete Klasse
  Inherits CKunde
  ...
  Public Overloads Sub test()
      MessageBox.Show("Hallo Privatkunde!")
  End Sub

End Class
```

3.9.6 Allgemeine Hinweise und Regeln zur Vererbung

Nachdem wir nun am praktischen Beispiel die Programmierung von Vererbungsbeziehungen kennen gelernt haben, werden wir auch die folgenden Regeln und Hinweise verstehen:

- Alle öffentlichen Eigenschaften und Methoden der Basisklasse sind auch über die abgeleiteten Subklassen verfügbar.
- Methoden der Basisklasse, die von den abgeleiteten Subklassen überschrieben werden dürfen (so genannte *virtuelle* Methoden), müssen mit dem Schlüsselwort *Overridable* deklariert werden.
- Fehlt das Schlüsselwort *Overridable* bei der Methodendeklaration, so bedeutet das, dass dies die einzige Implementierung der Methode ist.
- Methoden der Subklassen, welche die gleichnamige Methode der Basisklasse überschreiben, müssen mit dem Schlüsselwort *Overrides* deklariert werden.
- Wenn Sie das *Overrides*-Schlüsselwort in der Subklasse vergessen, wird angenommen, dass es sich um eine "Schattenfunktion" der originalen Funktion handelt. Eine solche Funktion hat denselben Namen wie das Original, überschreibt dieses aber nicht.
- Private Felder der Basisklasse, auf die die Subklassen zugreifen dürfen, müssen mit *Protected* deklariert werden.
- Die Basisklasse wird der Subklasse durch das der Klassendeklaration nachgestellte Schlüsselwort *Inherits* bekannt gemacht:

SYNTAX:
```
Public Class SubKlasse
Inherits Basisklasse
     ' ... Implementierungscode
End Class
```

- Eine Subklasse kann immer nur von einer einzigen Basisklasse abgeleitet werden (keine multiple Vererbung möglich).
- Mit dem *MyBase*-Objekt kann von den Subklassen auf die Basisklasse zugegriffen werden, mit dem *Me*-Objekt auf die eigene Klasse.

- Wenn die Basisklasse einen eigenen Konstruktor verwendet, so müssen in den Subklassen ebenfalls eigene Konstruktoren definiert werden (Konstruktoren können nicht vererbt werden!).
- Der Konstruktor einer Subklasse muss den Konstruktor seiner Basisklasse aufrufen (*MyBase*-Schlüsselwort).
- Falls aber die Basisklasse über keinen eigenen Konstruktor verfügt, wird der Standardkonstruktor automatisch aufgerufen, wenn ein Objekt aus einer Subklasse erzeugt wird.

3.9.7 Polymorphe Methoden

Untrennbar mit der Vererbung verbunden ist die so genannte Polymorphie (Vielgestaltigkeit). Polymorphes Verhalten bedeutet, dass erst zur Laufzeit einer Anwendung entschieden wird, welche der möglichen Methodenimplementierungen aufgerufen wird, da dies zum Zeitpunkt des Kompilierens noch unbekannt ist.

Im obigen Beispiel hatten wir von den Vorzügen der Polymorphie allerdings noch keinen Gebrauch gemacht, denn Privat- und Firmenkunde wurden in einzelnen Objektvariablen gespeichert und bereits per Programmcode fest mit ihren Methoden *getAdresse()* und *addGuthaben()* verbunden.

Um Polymorphie sichtbar zu machen, müssen wir das bei der Implementierung der Objekte zielgerichtet ausnutzen. Wie wir gleich sehen werden, treten die Vorzüge von Polymorphie besonders augenscheinlich zutage, wenn Objekte unterschiedlicher Klassenzugehörigkeit nacheinander in Arrays oder Auflistungen abgespeichert werden.

BEISPIEL 3.39: Polymorphe Methoden

VB

Wir nehmen die drei Klassen des Vorgängerbeispiels (*CKunde*, *CPrivatKunde*, *CFirmenKunde*) als Grundlage. An deren Implementierungen brauchen wir keinerlei Veränderungen vorzunehmen, denn polymorphes Verhalten ergibt sich als logische Konsequenz aus der Vererbung von Klassen. Änderungen müssen wir lediglich beim Abspeichern der Objektvariablen vornehmen, die diesmal in einem Array mit drei Feldern vom Typ der Basisklasse *CKunde* abgelegt werden sollen:

```
Public Class Form1
    Private Kunden(2) As CKunde          ' Referenz auf Basisklasse CKunde!
```

Im Konstruktorcode des Formulars erzeugen wir einen Privat- und zwei Firmenkunden:

```
    Public Sub New()
        InitializeComponent()
        Dim kunde1 As New CPrivatKunde("Herr", "Krause", "Leipzig")
        kunde1.Stammkunde = False
        Dim kunde2 As New CFirmenKunde("Frau", "Müller", "Master Soft GmbH")
        kunde2.Stammkunde = True
        Dim Kunde3 As New CFirmenKunde("Herr", "Maus", "Manfreds Internet AG")
        Kunde3.Stammkunde = False
```

Unser eingangs deklariertes Array nimmt nun Privat- und Firmenkunden in wahlloser Reihenfolge auf:

BEISPIEL 3.39: Polymorphe Methoden

```
        Kunden(0) = kunde1
        Kunden(1) = kunde2
        Kunden(2) = Kunde3
        TextBox1.Text = "100"
    End Sub
```

Das Array wird zunächst in einer *For-Next*-Schleife durchlaufen. Dabei werden die polymorphen Methoden (das sind die mit *Overridable* bzw. *Overrides* deklarierten) für alle Objekte aufgerufen. Es werden also sowohl die Adressen ausgegeben als auch für jeden Kunden das Guthaben berechnet, nachdem er für 100 € Waren gekauft hat.

```
    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        Dim brutto As Decimal = Convert.ToDecimal(TextBox1.Text)
        Label1.Text = String.Empty
        For i As Integer = 0 To Kunden.Length - 1
            Kunden(i).addGuthaben(brutto)
            Label1.Text = Label1.Text & vbCrLf & Kunden(i).getAdresse & vbCrLf &
                          Kunden(i).Guthaben.ToString("C") & vbCrLf
        Next
    End Sub
```

Obwohl im Array die Objekte bunt durcheinander gewürfelt sind, "weiß" das Programm zur Laufzeit genau, welche Implementierung der Methoden *getAdresse* und *addGuthaben* für den Privat- und für den Firmenkunden die richtige ist: Genau darin liegt der springende Punkt zum Verständnis der Polymorphie!

BEISPIEL 3.40: Die alternative Implementierung obigen Codes mittels *For Each*-Schleife bringt das Problem der Polymorphie noch deutlicher auf den Punkt

VB

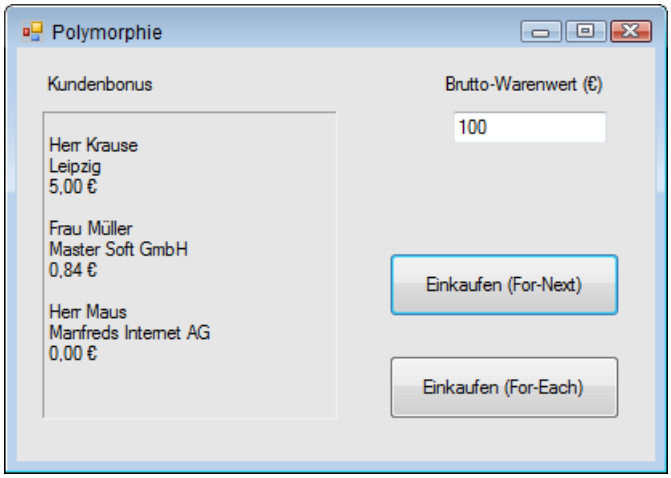
```
    Private Sub Button2_Click(sender As Object, e As EventArgs) Handles Button2.Click
        Dim brutto As Decimal = Convert.ToDecimal(TextBox1.Text)
        Label1.Text = String.Empty
```

Die Schleifenvariable *ku* ist eine Referenz auf die Basisklasse *CKunde*:

```
        For Each ku As CKunde In Kunden
            ku.addGuthaben(brutto)
            Label1.Text = Label1.Text & vbCrLf & ku.getAdresse & vbCrLf &
                                        ku.Guthaben.ToString("C") & vbCrLf
        Next
    End Sub
End Class
```

Praxistest

Das Ergebnis anhand der abgebildeten Testoberfläche beweist, dass Vererbung und Polymorphie tatsächlich untrennbar miteinander verknüpft sind. Egal ob Privat- oder Firmenkunde – es werden immer die passenden Methodenimplementierungen aufgerufen:



HINWEIS: Das tiefere Verständnis der Polymorphie ist mit Sicherheit der schwierigste Part der OOP, deshalb wurde unser Beispiel bewusst einfach gehalten, damit Sie zunächst zu einem Grundverständnis gelangen, welches Sie später weiter ausbauen können.

3.10 Besondere Klassen und Features

Zum Schluss wollen wir noch auf einige wichtige Klassen und Features eingehen, die in der OOP eine besondere Rolle spielen.

3.10.1 Abstrakte Klassen

Klassen, die lediglich ihr "Erbmaterial" an andere Klassen weitergeben und von denen selbst keine Instanzen gebildet werden, bezeichnet man als *abstrakt*. Typische Beispiele für abstrakte Klassen wären *Fahrzeug*, *Tier* oder *Nahrung*[1]. Um zu verhindern, dass von abstrakten Klassen Instanzen gebildet werden, können diese mit dem Schlüsselwort *MustInherit* gekennzeichnet werden.

BEISPIEL 3.41: In unserem Vorgängerbeispiel werden von der Klasse *CKunde* keine Instanzen gebildet, sie kann deshalb als abstrakt deklariert werden.

VB
```
Public MustInherit Class CKunde
...
End Class
```

Während die Referenzierung nach wie vor möglich ist

```
Dim Kunde As CKunde
```

[1] Können Sie sich vielleicht vorstellen, wie eine Instanz der Klasse *Fahrzeug* konkret aussehen soll?

BEISPIEL 3.41: In unserem Vorgängerbeispiel werden von der Klasse *CKunde* keine Instanzen gebildet, sie kann deshalb als abstrakt deklariert werden.

schlägt der Versuch einer Instanziierung fehl:

```
Kunde = New CKunde("Herr", "Krause")              ' Fehler
```

HINWEIS: Abstrakte Klassen ähneln einem weiteren wichtigen Softwarekonstrukt der OOP, der Schnittstelle (siehe Abschnitt 3.10.1).

3.10.2 Abstrakte Methoden

In Verbindung mit polymorphem Verhalten finden sich innerhalb abstrakter Klassen oft auch *abstrakte Methoden*, diese enthalten grundsätzlich keinen Code, da sie in den abgeleiteten Klassen komplett mit *Overrides* überschrieben werden. Zur Kennzeichnung abstrakter Methoden verwenden Sie das Schlüsselwort *MustOverride*.

HINWEIS: Die Deklaration einer abstrakten Methode erfolgt in einer Zeile, also ohne Rumpf.

BEISPIEL 3.42: Die Funktion *getAdresse* einer abstrakten *CKunde*-Klasse wird in den Subklassen komplett überschrieben und kann deshalb anstatt mit *Overridable* mit *MustOverride* deklariert werden.

VB
```
Public MustInherit Class CKunde
...
  Public MustOverride Function getAdresse() As String       ' abstrakte virtuelle Methode
...
End Class

Public Class CPrivatKunde
    Inherits CKunde
...
    Public Overrides Function getAdresse() As String  ' überschreibt Methode der Basisklasse
        Return _Anrede & "  " & _Name & "  " & _Wohnort
    End Function
...
End Class
```

3.10.3 Versiegelte Klassen

Wenn Sie unbedingt verhindern möchten, dass andere Programmierer von einer von Ihnen entwickelten Komponente weitere Subklassen ableiten, so müssen Sie Ihre Klasse mit Hilfe des Modifikators *NotInheritable* schützen.

BEISPIEL 3.43: Die Klasse *CPrivatKunde* wird versiegelt und darf deshalb keine Nachkommen haben.

VB
```
Public NotInheritable Class CPrivatKunde
    Inherits CKunde
    ...
End Class
```

Beim Versuch, davon eine Subklasse abzuleiten, schlägt Ihnen der Compiler erbarmungslos auf die Pfoten:

```
Public Class CStudent           ' Fehler!!!
    Inherits CPrivatKunde
    ...
End Class
```

HINWEIS: Eine versiegelte Klasse kann niemals eine Basisklasse sein. Vererbungsmodifikatoren wie *MustInherit* und *Overridable* führen in einer versiegelten Klasse zum Compilerfehler, da sie keinen Sinn ergeben!

Übrigens: Ein bekanntes Beispiel für eine versiegelte Klasse ist der *String*-Datentyp, was jedweden Begehrlichkeiten einen Riegel vorschiebt.

3.10.4 Partielle Klassen

Das Konzept partieller Klassen ermöglicht es, den Quellcode einer Klasse auf mehrere einzelne Dateien aufzusplitten. In Visual Studio wird zum Beispiel auf diese Weise der vom Designer automatisch erzeugte Layout-Code (z.B. *Form1.Designer.vb*) vom Code des Entwicklers (*Form1.vb*) getrennt, was zu einer gesteigerten Übersichtlichkeit beiträgt, wovon man sich nach Öffnen eines neuen Windows Forms-Projekts selbst überzeugen kann).

Die Programmierung ist denkbar einfach, denn alle Teile der Klasse sind lediglich mit dem Modifizierer *Partial* zu kennzeichnen.

BEISPIEL 3.44: Eine einfache Klasse *CKunde*

VB
```
Public Class CKunde
        Private _name As String
        Protected _guthaben As Decimal = 0

        Public Property NachName() As String
            Get
                Return _name
            End Get
            Set(value As String)
                _name = value
            End Set
        End Property
```

BEISPIEL 3.44: Eine einfache Klasse *CKunde*

```
        Public Property Guthaben() As Decimal
            Get
                Return _guthaben
            End Get
            Set(value As Decimal)
                _guthaben = value
            End Set
        End Property
        Public Sub addGuthaben(betrag As Decimal)
            _guthaben += betrag
        End Sub
    End Class
```

Obige Klasse könnte (als eine von mehreren Möglichkeiten) wie folgt in drei partielle Klassen aufgesplittet werden:

```
Partial Public Class CKunde
        Private _name As String
        Protected _guthaben As Decimal = 0
End Class

Public Class CKunde
        Public Property NachName() As String
            Get
                Return _name
            End Get
            Set(value As String)
                _name = value
            End Set
        End Property

        Public Property Guthaben() As Decimal
            Get
                Return _guthaben
            End Get
            Set(value As Decimal)
                _guthaben = value
            End Set
        End Property
End Class

Partial Public Class CKunde
        Public Sub addGuthaben(betrag As Decimal)
            _guthaben += betrag
        End Sub
End Class
```

Wie Sie sehen, kann (muss aber nicht) bei einer der partiellen Definitionen auf *Partial* verzichtet werden, diese Klasse ist dann gewissermaßen die Ausgangsklasse, die durch den Code der anderen partiellen Klassen erweitert wird.

HINWEIS: Auch *Structure*- oder *Interface*-Definitionen können mittels *Partial*-Modifizierer gesplittet werden!

3.10.5 Die Basisklasse System.Object

Jedes Objekt in .NET ist von der Basisklasse *System.Object* abgeleitet. Diese Klasse ist Teil des Microsoft .NET Frameworks und beinhaltet die Basiseigenschaften und -methoden, wie sie für ein .NET-Objekt erforderlich sind.

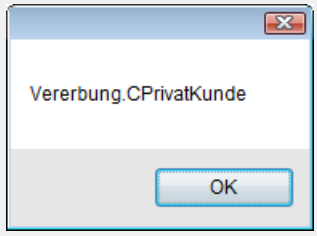
Alle öffentlichen Eigenschaften und Methoden von *System.Object* stehen automatisch auch in jedem Objekt zur Verfügung, welches Sie erzeugt haben. Beispielsweise ist in *System.Object* bereits ein Standardkonstruktor enthalten. Wenn Sie in Ihrem Objekt keinen eigenen Konstruktor definiert haben, wird es mit diesem Konstruktor erzeugt.

Viele der öffentlichen Eigenschaften und Methoden von *System.Object* haben eine Standardimplementation. Das heißt, Sie brauchen selbst keinerlei Code zu schreiben, um sie zu verwenden.

BEISPIEL 3.45: Die *ToString*-Methode liefert den Namen der Anwendungskomponente (die Windows-Anwendung heißt hier Vererbung) und die Klassenzugehörigkeit von *kunde1*.

VB
```
MessageBox.Show(kunde1.ToString)
```

Ergebnis



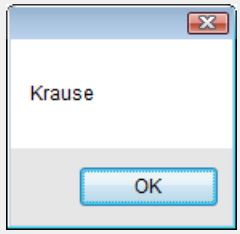
Sie können das standardmäßige Verhalten von *ToString* mittels *Overrides*-Schlüsselwort verändern. Dies erlaubt Ihnen eine individuelle Implementierung einiger Eigenschaften bzw. Methoden von *System.Object*.

BEISPIEL 3.46: Die gleiche *ToString*-Methode des Vorgängerbeispiels liefert nun den Namen des Kunden, wenn Sie die folgende Methode zum Klassenkörper von *CKunde* hinzufügen.

VB
```
Public Overrides Function ToString() As String
   Return _name
End Function
```

BEISPIEL 3.46: Die gleiche *ToString*-Methode des Vorgängerbeispiels liefert nun den Namen des Kunden, wenn Sie die folgende Methode zum Klassenkörper von *CKunde* hinzufügen.

Ergebnis



3.10.6 Property-Accessors

Die *Get*- und *Set*- Accessoren von Eigenschaften hatten in den ersten .NET-Versionen von Visual Basic die gleiche Sichtbarkeit wie die Eigenschaft zu der sie gehören. Seit .NET 2.0 ist es möglich, den Zugriff auf einen dieser Accessoren zu beschränken. Meist ist dies für den *Set*-Accessor sinnvoll, während der *Get*-Accessor in der Regel öffentlich bleiben soll.

BEISPIEL 3.47: Eine Eigenschaft mit *Get*- und *Set*-Accessoren. Der *Get*-Accessor besitzt die gleiche Sichtbarkeit wie die *KontoNummer*-Eigenschaft, während der *Set*-Accessor nur einen *Friend*-Zugriff erlaubt.

VB

```
Private _knr As String
Public Property KontoNummer() As String
    Get
        Return _knr
    End Get
    Friend Set(value As String)
        _knr = value
    End Set
End Property
```

3.10.7 Nullbedingter Operator

Dieser mit VB 2015 eingeführte Operator ist eine bequeme Abkürzung für all jene Fälle, in denen auf *Nothing* zu prüfen ist:

BEISPIEL 3.48: Nullbedingter Operator

VB

```
Dim frm = firma.Addresse?.Land
```

ist eine Abkürzung für

```
Dim _tmp = firma.Addresse
Dim frm = If(_tmp IsNot Nothing, _tmp.Addresse.Land, Nothing)
```

BEISPIEL 3.49: Es wird nur dann abgezweigt, wenn der Bewerber zugewiesen und älter als 60 Jahre ist

VB
```
...
If bewerber?.Age > 60 Then ...
...
```

BEISPIEL 3.50: Ein Standardnamen wird zugeteilt wenn der Kunde nicht zugewiesen ist

VB
```
...
Dim name = If(kunde?.Name, "Müller")
...
```

BEISPIEL 3.51: Nur wenn ein *kunde*-Objekt vorhanden ist wird *ToString*() ausgeführt

VB
```
...
Dim s = kunde?.ToString()
...
```

3.11 Schnittstellen (Interfaces)

Das .NET-Framework (die CLR) unterstützt keine Mehrfachvererbung, d.h., eine Unterklasse kann immer nur von einer einzigen Oberklasse erben. Dies ist wohl mehr ein Segen als ein Fluch, denn allzu leicht würde sonst der Programmierer im Gestrüpp mehrfacher Vererbungsbeziehungen über mehrere Hierarchie-Ebenen hinweg die Übersicht verlieren, instabiler Code und Chaos wären die Folge.

Einen Ausweg bietet die Verwendung von Schnittstellen, diese bieten fast alle Möglichkeiten der Mehrfachvererbung, vermeiden aber deren Nachteile.

HINWEIS: Schnittstellen dienen dazu, um gemeinsame Merkmale ansonsten unabhängiger Klassen beschreiben zu können.

Eine Schnittstelle können Sie sich zunächst wie eine abstrakte Klasse (siehe 3.10.1) vorstellen, in welcher nur abstrakte Methoden definiert werden[1].

3.11.1 Definition einer Schnittstelle

Eine Schnittstelle können Sie zu Ihrem Projekt genauso hinzufügen wie eine neue Klasse. Anstatt des Schlüsselworts *Class* verwenden Sie aber *Interface*.

HINWEIS: Laut Konvention sollte der Namen einer Schnittstelle immer mit "I" beginnen.

[1] Dieser Vergleich hinkt natürlich wegen der auch bei abstrakten Klassen nicht möglichen Mehrfachvererbung.

BEISPIEL 3.52: Eine Schnittstelle *IPerson*, die zwei Eigenschaften und eine Methode definiert [2]

VB
```
Public Interface IPerson
  Property Nachname As String
  Property Vorname As String
  Function getName() As String
End Interface
```

Vielleicht vermissen Sie im obigen Beispiel die Zugriffsmodifizierer (***Public*** *Property* ...), diese aber haben in einer Schnittstellendefinition generell nichts zu suchen.

HINWEIS: Die Festlegung der Zugriffsmodifizierer für die Mitglieder der Schnittstelle ist allein Angelegenheit der Klasse, die die Schnittstelle implementiert!

3.11.2 Implementieren einer Schnittstelle

Die implementierende Klasse benutzt anstatt des Schlüsselworts *Inherits* das Schlüsselwort *Implements*.

HINWEIS: Die implementierende Klasse geht die Verpflichtung ein, ausnahmslos **alle** Mitglieder der Schnittstelle zu implementieren!

BEISPIEL 3.53: Die Klasse *CKunde* implementiert die Schnittstelle *IPerson*

VB
```
Class CKunde
   Implements IPerson
        ...
```

Die von *IPerson* geerbten abstrakten Klassenmitglieder müssen implementiert werden:

```
        Public Property Nachname As String Implements IPerson.Nachname
        Public Property Vorname As String Implements IPerson.Vorname

        Public Function getName() As String Implements IPerson.getName
           Return _Vorname & " " & _Nachname
        End Function
```

Es folgen die normalen Klassenmitglieder:

```
        ...
End Class
```

Den kompletten Code finden Sie im Praxisbeispiel

► 3.12.4 Schnittstellenvererbung verstehen

[2] Das Interface verwendet selbstimplementierende Eigenschaften.

Dort wird auch gezeigt, wie man eine mit abstrakten Methoden ausgestattete abstrakte Klasse ganz leicht in eine Schnittstelle überführen kann.

3.11.3 Abfragen, ob eine Schnittstelle vorhanden ist

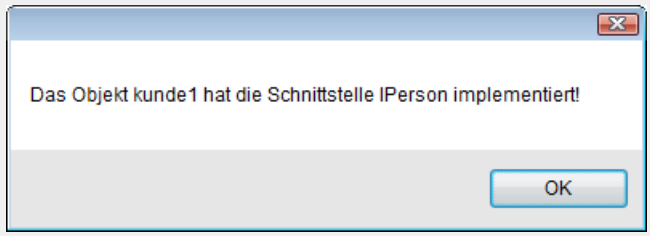
Manchmal möchte man vor der eigentlichen Arbeit mit einem Objekt wissen, ob dieses eine bestimmte Schnittstelle implementiert hat. Eine Lösung bietet eine Abfrage mit dem *TypeOf*-Operator.

BEISPIEL 3.54: Wir ergänzen die Bedienoberfläche des Vorgängerbeispiels um eine zweite Schaltfläche.

VB

```
Private Sub Button2_Click(sender As Object, e As EventArgs) Handles Button2.Click
    If TypeOf kunde1 Is IPerson Then
        MessageBox.Show("Das Objekt kunde1 hat die Schnittstelle IPerson implementiert!")
    End If
End Sub
```

Ergebnis



3.11.4 Mehrere Schnittstellen implementieren

Eine Klasse kann nicht nur eine, sondern auch mehrere Schnittstellen gleichzeitig implementieren, was quasi Mehrfachvererbung bedeutet, wie sie mit der klassischen Implementierungsvererbung unmöglich ist.

BEISPIEL 3.55: Eine Klasse implementiert zwei Schnittstellen

VB

```
Public Class CPrivatkunde
    Implements IPerson, IKunde
    ...
End Class
```

3.11.5 Schnittstellenprogrammierung ist ein weites Feld

... und bis jetzt haben wir nur an der Oberfläche gekratzt. Wichtige Prinzipien hier nochmals in Kürze:

- Anstatt von einer abstrakten Klasse zu erben, werden die abstrakten Methoden über eine Schnittstelle veröffentlicht. Damit erlangt man gewissermaßen die Funktionalität der Mehrfachvererbung und umgeht deren Nachteile.

- Eine Schnittstelle ist wie ein Vertrag: Sobald eine Klasse eine Schnittstelle implementiert, muss sie auch ausnahmslos alle (!) Mitglieder der Schnittstelle implementieren und veröffentlichen.
- Der Name der implementierten Methode sowie deren Signatur muss mit deren Definition in der Schnittstelle exakt übereinstimmen.
- Mehrere Schnittstellen können zu einer neuen Schnittstelle zusammengefasst werden und selbst wieder Schnittstellen implementieren.

HINWEIS: Mehr zur Schnittstellenprogrammierung finden Sie beispielsweise im Kapitel 5 (*IEnumerable*-Interface).

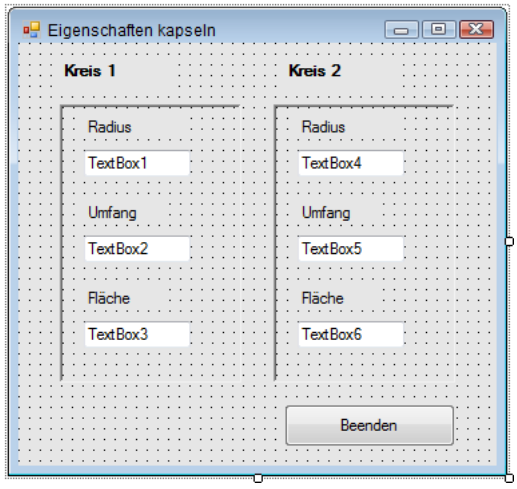
3.12 Praxisbeispiele

3.12.1 Eigenschaften sinnvoll kapseln

Das Deklarieren von Eigenschaften als öffentliche Variablen der Klasse heißt immer, das Brett an der dünnsten Stelle zu bohren. Der fortgeschrittene Programmierer verwendet stattdessen sogenannte Property-Methoden, die einen kontrollierten Zugriff erlauben. Außerdem ermöglichen die Property-Methoden auch die Implementierung von *berechneten Eigenschaften*, die aus den (privaten) Zustandsvariablen ermittelt werden. Im vorliegenden Beispiel handelt es sich um eine Klasse *CKreis* mit den Eigenschaften *Radius*, *Umfang* und *Fläche*. Diese Klasse speichert intern eine einzige Zustandsvariable *radZ*, aus welcher direkt beim Zugriff alle Eigenschaften berechnet werden.

Oberfläche

Um einen weiteren Vorteil der OOP zu demonstrieren, d.h. ohne viel Mehraufwand beliebig viele Instanzen aus einer Klasse bilden, wollen wir mit zwei Objekten (*Kreis1* und *Kreis2*) arbeiten.



Quellcode CKreis

Unsere Klasse wird außerhalb von *Form1* definiert, wir werden sie sogar in ein separates Klassenmodul auslagern. Wählen Sie das Menü *Projekt|Klasse hinzufügen...* und geben Sie dem Klassenmodul den Namen *CKreis.vb*.

```
Public Class CKreis
```

Die private Zustandsvariable speichert den Wert des Radius:

```
    Private radZ As Double
```

Die Eigenschaft *Radius* (Rahmencode wird automatisch erzeugt!):

```
    Public Property Radius() As String
        Get
            Return radZ.ToString("#,#0.00")
        End Get
        Set(value As String)
            If value <> String.Empty Then
                radZ = CDbl(value)
            Else
                radZ = 0
            End If
        End Set
    End Property
```

Die Eigenschaft *Umfang*:

```
    Public Property Umfang() As String
        Get
            Return (2 * Math.PI * radZ).ToString("#,#0.00")
        End Get
        Set(value As String)
            If value <> String.Empty Then
                radZ = CDbl(value) / 2 / Math.PI
            Else
                radZ = 0
            End If
        End Set
    End Property
```

Die Eigenschaft *Fläche*:

```
    Public Property Fläche() As String
        Get
            Return (Math.PI * Math.Pow(radZ, 2)).ToString("#,#0.00")
        End Get
        Set(value As String)
            If value <> String.Empty Then
                radZ = Math.Sqrt(CDbl(value) / Math.PI)
            Else
                radZ = 0
```

```
            End If
        End Set
    End Property
End Class
```

Wechseln Sie nun in den Klassencode von *Form1*.

Quellcode Form1

```
Public Class Form1
```

Ein Objekt wird erzeugt:

```
    Private Kreis1 As New CKreis()
```

Die folgenden Event-Handler sind einfach und übersichtlich, da die Objekte die inneren Funktionalitäten wegkapseln. Den Radius ändern:

```
    Private Sub TextBox1_KeyUp(sender As Object, e As KeyEventArgs) Handles TextBox1.KeyUp
        With Kreis1
            .Radius = TextBox1.Text
            TextBox2.Text = .Umfang
            TextBox3.Text = .Fläche
        End With
    End Sub
```

Den Umfang ändern:

```
    Private Sub TextBox2_KeyUp(sender As Object, e As KeyEventArgs) Handles TextBox2.KeyUp
        With Kreis1
            .Umfang = TextBox2.Text
            TextBox1.Text = .Radius
            TextBox3.Text = .Fläche
        End With
    End Sub
```

Die Fläche ändern:

```
    Private Sub TextBox3_KeyUp(sender As Object, e As KeyEventArgs) Handles TextBox3.KeyUp
        With Kreis1
            .Fläche = TextBox3.Text
            TextBox1.Text = .Radius
            TextBox2.Text = .Umfang
        End With
    End Sub
```

Der Code für *Kreis2* ist analog aufgebaut und braucht deshalb hier nicht wiederholt zu werden (siehe Beispieldaten).

```
End Class
```

Test

Sobald Sie eine beliebige Eigenschaft ändern, werden die anderen zwei sofort aktualisiert! Wegen der in der Klasse eingebauten Eingabeprüfung verursacht ein leerer Eingabewert keinen Fehler. Aus Gründen der Übersichtlichkeit wurde aber auf das Abfangen weiterer Eingaben, die sich nicht in einen numerischen Wert konvertieren lassen, verzichtet.

HINWEIS: Geben Sie als Dezimaltrennzeichen immer das Komma (,) ein, als Tausender-Separator dürfen Sie den Punkt (.) verwenden.

Objektinitialisierer

Man kann ein Objekt auch dann erzeugen und initialisieren, wenn es dazu keinen Konstruktor gibt. Sie könnten also im Code von *Form1* die Instanziierung der Klasse durch direktes Zuweisen ihrer Eigenschaften wie folgt vornehmen:

```
Private Kreis1 As New CKreis With {.Radius = "1.0"}
```

HINWEIS: Mehr zu Objektinitialisierern erfahren Sie im Abschnitt 3.8.2!

3.12.2 Eine statische Klasse anwenden

Als "statisch" wollen wir hier solche Klassen bezeichnen, die lediglich statische (*Shared*) Mitglieder haben. Solche Klassen eignen sich beispielsweise ideal für Formelsammlungen (siehe *Math*-Klasse), da keine Objekte erzeugt werden müssen, denn es kann gleich "losgerechnet" werden. Das vorliegende Beispiel demonstriert dies anhand einer statischen Klasse *CKugel* zur Berechnung des Kugelvolumens bei gegebenem Durchmesser (und umgekehrt).

$V = 4/3 * Pi * r^3$

Nimmt man anstatt des Radius den Durchmesser *d* der Kugel, so ergibt sich daraus nach einigen Umstellungen die folgende Berechnungsformel für das Volumen *V*:

$V = d^3 * Pi/6$

Oberfläche

Lediglich ein *Form*ular mit zwei *TextBox*en zur Eingabe von Kugeldurchmesser und Kugelvolumen ist erforderlich (siehe Laufzeitansicht).

Quellcode CKugel

Statische Funktionen werden mit dem Schlüsselwort *Shared* gekennzeichnet.

```
Public Class CKugel
    Public Shared Function Durchmesser_Volumen(durchmesser As String) As Double
        Dim dur As Double = System.Double.Parse(durchmesser)
        Dim vol As Double = Math.Pow(dur, 3) * Math.PI / 6.0
```

```
        Return vol
    End Function

    Public Shared Function Volumen_Durchmesser(volumen As String) As Double
        Dim vol As Double = System.Double.Parse(volumen)
        Dim dur As Double = Math.Pow(6 / Math.PI * vol, 1 / 3.0)
        Return dur
    End Function
End Class
```

Quellcode Form1

Die Verwendung der Klasse im Formularcode:

```
Public Class Form1
```

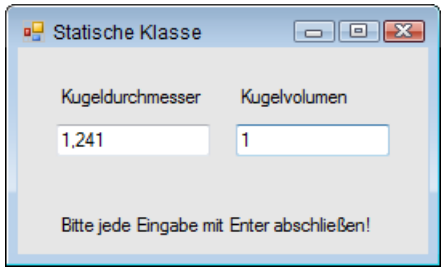
Die Berechnung startet nach Betätigen der *Enter*-Taste:

```
 Private Sub TextBox1_KeyUp(sender As Object, e As KeyEventArgs) Handles TextBox1.KeyUp
        If e.KeyCode = Keys.Enter And TextBox1.Text <> String.Empty Then
            TextBox2.Text = CKugel.Durchmesser_Volumen(TextBox1.Text).ToString("#,##0.000")
        End If
    End Sub

    Private Sub TextBox2_KeyUp(sender As Object, e As KeyEventArgs) Handles TextBox2.KeyUp
        If e.KeyCode = Keys.Enter And TextBox2.Text <> String.Empty Then
            TextBox1.Text = CKugel.Volumen_Durchmesser(TextBox2.Text).ToString("#,##0.000")
        End If
    End Sub
End Class
```

Test

Es ist egal, ob Sie den Radius oder das Volumen eingeben. Nach Betätigen der *Enter*-Taste wird der Inhalt des jeweils anderen Textfelds sofort aktualisiert.

Die Maßeinheit spielt bei der Programmierung keine Rolle, da sie für beide Eingabefelder identisch ist. Um beispielsweise einen Wasserbehälter mit 1 Kubikzentimeter Inhalt zu realisieren, ist eine Kugel mit dem Durchmesser von 1,241 Zentimetern erforderlich, für 1 Kubikmeter (1000 Liter) wären es 1,241 Meter:



3.12.3 Vom fetten zum dünnen Client

Lassen Sie sich durch den martialischen Titel nicht irritieren, wir wollen damit lediglich Ihr Interesse für eine Methodik wecken, die Ihnen hilft, den Horizont herkömmlicher Programmiertechniken zu überschreiten und damit einen leichteren Zugang zur OOP ermöglicht. Dabei gehen wir von folgender Erfahrung aus:

Typisch für den OOP-Ignoranten ist, dass er getreu der Devise "Hauptsache es funktioniert" mit Ausdauer und Beharrlichkeit immer wieder so genannte "fette" Clients (Fat Clients) programmiert. In einem solchen *Fat Client* ist in der Regel die gesamte Intelligenz (Geschäfts- bzw. Serverlogik) der Anwendung konzentriert, d.h., eine Aufteilung in Klassen bzw. Schichten hat nie stattgefunden.

Ein qualifizierter objektorientierter Entwurf zeichnet sich aber dadurch aus, dass der Client möglichst "dumm" bzw. "dünn" ist. Ein *Thin Client* verwaltet ausschließlich das User-Interface, die Aufgaben beschränken sich auf die Entgegennahme der Benutzereingaben und deren Weiterleitung an die Geschäftslogik bzw. umgekehrt auf die Ausgabe und Anzeige der von der Geschäftslogik ermittelten Ergebnisse.

Der Server hingegen umfasst die Geschäftslogik und kapselt damit die gesamte Intelligenz der Anwendung.

Die Vorteile einer solchen mehrschichtigen "Thin Client"-Strategie sind:

- gesteigerte Übersichtlichkeit und leichte Wiederverwendbarkeit der Software,
- Realisierung als verteilte Anwendung im Netzwerk ist möglich,
- Wartbarkeit und Erweiterbarkeit der Geschäftslogik sind möglich, ohne dass die Clients geändert werden müssten.

In unserem zweiteiligen Beispiel geht es um einen einfachen "Taschenrechner", den wir in zwei Versionen realisieren wollen.

In unserer ersten Windows Forms-Anwendung haben wir es mit einem Musterbeispiel für einen "fetten" Client zu tun. Im zweiten Teil verwandeln wir das Programm in eine mehrschichtige Anwendung mit einem "dünnen" Client. Neugierig geworden?

Oberfläche

Siehe Laufzeitansicht.

Quellcode (Fat Client)

```
Public Class Form1
```

Über die Bedeutung der folgenden drei globalen Zustandsvariablen brauchen wir wohl keine weiteren Worte zu verlieren:

```
    Private op As Char                  ' aktueller Operator   (+, - , *, /)
    Private reg1 As String = Nothing    ' erstes Register   (Operand)
    Private reg2 As String = Nothing    ' zweites Register  (Operand)
```

Wir wollen zur Steuerung des Programmablaufs eine spezielle Variable *state* verwenden, die den aktuellen Zustand speichert:

```
Private state As Byte = 1           ' aktuelles Register (1 oder 2)
```

Typisch für die nun folgenden Ereignisbehandlungen ist, dass die durchgeführten Aktionen vom Wert der Zustandsvariablen *state* abhängig sind.

Zur Eingabe einer Ziffer (0...9) benutzt der gesamte Ziffernblock eine gemeinsame Ereignisbehandlung:

```
Private Sub ButtonZ_Click(sender As Object, e As EventArgs)  Handles Button1.Click,
     Button2.Click, Button3.Click, Button4.Click, Button5.Click, Button6.Click,
        Button7.Click, Button8.Click, Button9.Click, Button10.Click, Button11.Click
    Dim cmd As Button = CType(sender, Button)
    Select Case state
        Case 1                ' zum ersten Operanden hinzufügen:
            reg1 &= cmd.Text.Chars(0)
            Label1.Text = reg1
        Case 2                ' zum zweiten Operanden hinzufügen:
            reg2 &= cmd.Text.Chars(0)
            Label1.Text = reg1 & " " & op & " " & reg2
    End Select
End Sub
```

Für die Eingabe der Operation (+, -, *, /) wird ähnlich verfahren:

```
Private Sub ButtonOp_Click(sender As Object, e As EventArgs) _
              Handles ButtonAdd.Click, ButtonSub.Click, ButtonMult.Click, ButtonDiv.Click
    Dim cmd As Button = CType(sender, Button)
    Select Case state
        Case 1
            op = cmd.Text.Chars(0)      ' neuer Operand ...
            state = 2                   ' ... und Zustandswechsel
        Case 2
            ergebnis()        ' erst Zwischenergebnis mit altem Operand ermitteln ...
            op = cmd.Text.Chars(0)               ' ... dann neuer Operand
     End Select
     Label1.Text = reg1.ToString & " " & op
     reg2 = Nothing                              ' Reg2 löschen
 End Sub
```

Die folgende Hilfsprozedur führt die Rechenoperation aus und speichert deren Ergebnis in *reg1*:

```
 Private Sub ergebnis()
    Dim r1 As Double = Convert.ToDouble(reg1)
    Dim r2 As Double = Convert.ToDouble(reg2)
    Select Case op
        Case "+"c
            reg1 = (r1 + r2).ToString
        Case "-"c
            reg1 = (r1 - r2).ToString
```

```
            Case "*"c
                reg1 = (r1 * r2).ToString
            Case "/"c
                reg1 = (r1 / r2).ToString
        End Select
        reg2 = Nothing  ' löscht zweites Register
    End Sub
```

Die Ergebnistaste (=):

```
    Private Sub ButtonResult_Click(sender As Object, e As EventArgs) Handles ButtonResult.Click
        If state = 2 Then
            ergebnis()
            Label1.Text &= " = " & reg1
            state = 1
        Else
            Label1.Text = reg1
            reg2 = Nothing   ' löscht zweites Register
        End If
    End Sub
```

Letztes eingegebenes Zeichen löschen (CE):

```
    Private Sub ButtonCE_Click(sender As Object, e As EventArgs) Handles ButtonCE.Click
        Select Case state
            Case 1
                If Not (reg1 = Nothing) Then
                    reg1 = reg1.Remove(reg1.Length - 1, 1)
                    Label1.Text = reg1
                End If
            Case 2
                If Not (reg2 = Nothing) Then
                    reg2 = reg2.Remove(reg2.Length - 1, 1)
                    Label1.Text = reg2
                End If
        End Select
    End Sub
```

Alle Register sowie Anzeige löschen und Anfangszustand herstellen:

```
    Private Sub ButtonCLR_Click(sender As Object, e As EventArgs) Handles ButtonCLR.Click
        reg1 = Nothing : reg2 = Nothing : Label1.Text = String.Empty
        state = 1
    End Sub
```

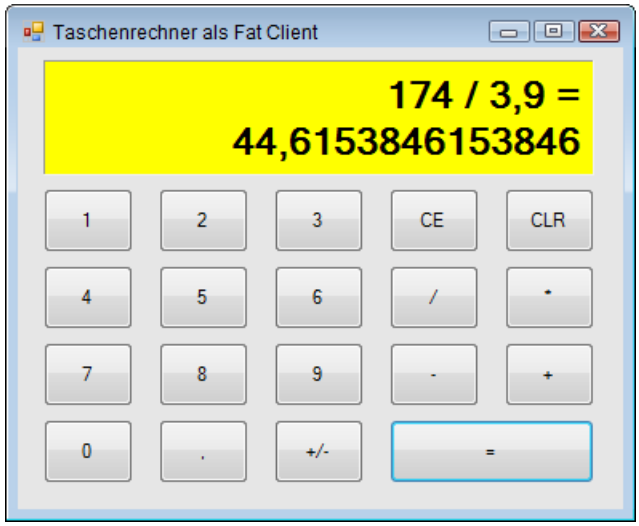
Schließlich noch der Vorzeichenwechsel (+/-):

```
    Private Sub ButtonVZ_Click(sender As Object, e As EventArgs) Handles ButtonVZ.Click
        Dim r As Double
        Select Case state
            Case 1
                r = -Convert.ToDouble(reg1)
```

```
                reg1 = r.ToString
                Label1.Text = reg1
            Case 2
                r = -Convert.ToDouble(reg2)
                reg2 = r.ToString
                Label1.Text = reg1 & " " & op & " " & reg2
        End Select
    End Sub
End Class
```

Test

Der Vorzug gegenüber üblichen Rechnern (oder auch dem im Windows-Zubehör) sticht sofort ins Auge: Man kann den Rechenvorgang mitverfolgen, weil der komplette Ausdruck angezeigt wird.



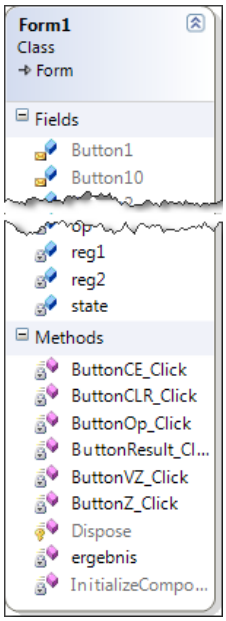
HINWEIS: Wie bei jedem anderen einfachen Taschenrechner auch, bleibt hier die Rangfolge der Operatoren (Punktrechnung geht vor Strichrechnung) unberücksichtigt. Bei der Eingabe von mehreren Operationen hintereinander, z.B. 3 + 4 * 12, ist deshalb zu beachten, dass erst die höherwertige Operation auszuführen ist (4 * 12).

Bemerkungen zum fetten Client

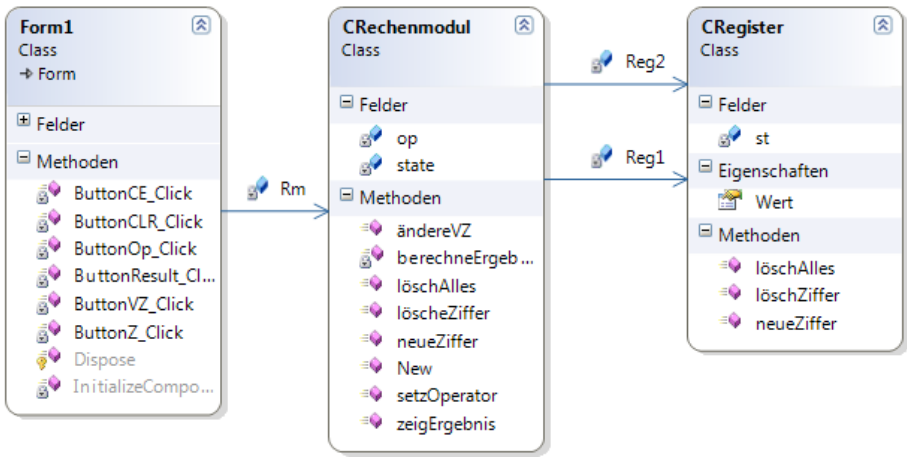
- Den Programmablauf könnte man in Gestalt eines Zustandsüberführungsdiagramms (*State Chart*) noch anschaulicher darstellen (siehe Kapitel 25).
- Leider ist die gesamte Intelligenz der Anwendung in der Benutzerschnittstelle *Form1* enthalten, also ein typischer "fetter" Client. Transparenz, Wiederverwendbarkeit und Wartbarkeit des Codes sind demzufolge katastrophal! Wie man das Programm auf ein höheres objektorientiertes Niveau heben kann, soll die folgende alternative Realisierung unseres Taschenrechners zeigen.

Abmagerungskur für den fetten Client

Dass es sich bei unserem alten Programm tatsächlich um einen Fat Client handelt, zeigt das zugehörige Klassendiagramm. Klicken Sie dazu im Projektmappen-Explorer mit der rechten Maustaste auf *Form1.vb* und wählen Sie im Kontextmenü *Klassendiagramm anzeigen*. Es vergeht eine kleine Weile und dann bietet sich Ihnen der in der Abbildung nur auszugsweise gezeigt Anblick mit einer schier endlosen Auflistung der in der Klasse implementierten Felder und Methoden.



Schluss mit diesem Chaos! Durch Auslagern der Intelligenz in die Klassen *CRechenmodul* und *CRegister* erhalten wir nach einigem Hin und Her als Ergebnis der "Abmagerungskur" schließlich das abgebildete neue Klassendiagramm:



Die Klasse *Form1* (Thin Client) ist gegenüber dem Vorgänger deutlich abgemagert und beschränkt sich nur noch auf ihre eigentliche Aufgabe, nämlich die Verwaltung der Benutzerschnittstelle. Die Klassen *CRechenmodul* und *CRegister* stellen hingegen die zweischichtige Geschäftslogik (Server) der Anwendung dar, kapseln also die Intelligenz der Anwendung.

HINWEIS: Damit Sie im Klassendiagramm die durch Pfeilverbindungen gekennzeichneten Zuordnungen (Assoziationen) zwischen den Klassen sehen, müssen Sie mit der rechten Maustaste auf das Feld *Rm* in der Klasse *Form1* klicken und im Kontextmenü den Eintrag *Als Zuordnung anzeigen* wählen. Analog verfahren Sie mit den Feldern *Reg1* und *Reg2* in *CRechenmodul*.

Eine Erklärung des Klassendiagramms mit anderen Worten: Unser Thin Client benutzt eine Instanz der Klasse *CRechenmodul*. In dieser wiederum sind zwei Instanzen der Klasse *CRegister* enthalten. Hier geht es also noch nicht um Vererbung, Polymorphie etc., sondern nur um das zweckmäßige "Wegkapseln" von Funktionalität, wie das in unserem Fall etwa auch der physikalischen Realität entspricht, denn auch ein "richtiger" Taschenrechner enthält ein Rechenmodul, in diesem wiederum sind ein oder mehrere Register enthalten.

Allerdings stellt diese Thin Client-Lösung nur eine von mehreren Möglichkeiten dar und ist das Ergebnis einer Analyse des Ausgangscodes nach den Kriterien der Wiederverwendbarkeit ("Code Reuse").

Quellcode für CRegister

```
Public Class CRegister
```

Die globale Variable *st* speichert den Registerinhalt als Zeichenkette:

```
    Private st As String = Nothing
```

Zugriff auf den numerischen Wert von *st:*

```
    Public Property Wert() As Double
        Get
            Try
                Wert = Convert.ToDouble(st)
            Catch
                Wert = 0
            End Try
        End Get
        Set(value As Double)
            st = value.ToString
        End Set
    End Property
```

Hinzufügen einer einzelnen Ziffer und Rückgabe des Anzeigestrings:

```
    Public Function neueZiffer(z As Char) As String
        If (Char.IsDigit(z) Or z = ",") Then
            st &= z
```

```
            Return (st)
        Else
            Return (String.Empty)
        End If
    End Function
```

Letzte Ziffer löschen und Anzeigestring zurückgeben:

```
    Public Function löschZiffer() As String
        If st.Length > 0 Then
            st = st.Remove(st.Length - 1, 1)
        End If
        Return (st)
    End Function
```

Gesamtes Register löschen:

```
    Public Sub löschAlles()
        st = String.Empty
    End Sub

End Class
```

Quellcode für CRechenmodul

```
Public Class CRechenmodul

    Private state As Byte = 1          ' Startmodus (Zustandsvariable)
    Private op As Char                 ' aktueller Operator
    Private Reg1, Reg2 As CRegister    ' zwei Rechenregister
```

Im Konstruktor werden zwei Register-Objekte erzeugt:

```
    Public Sub New()
        Reg1 = New CRegister()
        Reg2 = New CRegister()
    End Sub
```

Zifferneingabe in aktuelles Register:

```
    Public Function neueZiffer(z As Char) As String
        If state = 1 Then                       ' zum ersten Register hinzufügen:
            Return (Reg1.neueZiffer(z))
        Else                                    ' zum zweiten Register hinzufügen:
            Return (Reg1.Wert.ToString & " " & op & " " & Reg2.neueZiffer(z))
        End If
    End Function
```

Letzte Ziffer des aktuellen Registers löschen und resultierenden Registerinhalt zurückgeben:

```
    Public Function löscheZiffer() As String
        If state = 1 Then
            Return (Reg1.löschZiffer())
```

```
        Else
            Return (Reg2.löschZiffer())
        End If
    End Function
```

Vorzeichen des aktuellen Registers umkehren:

```
    Public Function ändereVZ() As String
        If state = 1 Then
            Reg1.Wert = -Reg1.Wert
            Return (Reg1.Wert.ToString)
        Else
            Reg2.Wert = -Reg2.Wert
            Return (Reg1.Wert.ToString & " " & op & " " & Reg2.Wert.ToString)
        End If
    End Function
```

Der Operator wird übernommen. Rückgabewert ist der String des ersten Operanden mit abschließendem Operatorenzeichen:

```
    Public Function setzOperator(o As Char) As String
        If state = 1 Then
            state = 2
        Else
            berechneErgebnis()               ' Zwischenergebnis (mit altem Operator) ermitteln
        End If
        op = o                                          ' neuen Operator übernehmen
        Reg2.löschAlles()                               ' zweites Register löschen
        Return (Reg1.Wert.ToString & " " & op)
    End Function
```

Die abschließende Rechenoperation ausführen und das Ergebnis liefern:

```
    Public Function zeigErgebnis() As String
        Dim s As String = ""
        If state = 1 Then      ' im Startmodus wird noch nichts berechnet, ...
            Reg2.löschAlles()    ' ... lediglich zweites Register gelöscht
        Else
            s = " = " & berechneErgebnis()
            state = 1
        End If
        Return (s)
    End Function
```

Eine Hilfsmethode zum Ausführen der Rechenoperation nebst Abspeichern des Ergebnisses im ersten Register (überschreibt erstes Register mit Ergebnis der Operation):

```
    Private Function berechneErgebnis() As String
        Select Case op
            Case "+"c
                Reg1.Wert = Reg1.Wert + Reg2.Wert
            Case "-"c
```

```
                Reg1.Wert = Reg1.Wert - Reg2.Wert
            Case "*"c
                Reg1.Wert = Reg1.Wert * Reg2.Wert
            Case "/"c
                Reg1.Wert = Reg1.Wert / Reg2.Wert
        End Select
        Reg2.löschAlles()       ' zweites Register löschen
        Return Reg1.Wert.ToString
    End Function
```

Alle Register löschen und Startzustand wieder herstellen:

```
    Public Sub löschAlles()
        Reg1.löschAlles()
        Reg2.löschAlles()
        state = 1
    End Sub
End Class
```

Quellcode für Form1

Die Oberfläche unseres Thin Client entspricht 100%ig der seines "fetten" Vorgängers. Die Programmierung ist allerdings – dank des *CRechenmodul*-Objekts – deutlich einfacher und transparenter geworden:

```
Public Class Form1
```

Einzige globale Variable ist eine Instanz der Klasse *CRechenmodul*:

```
    Private Rm As New CRechenmodul()
```

Eine Ziffer eingeben (0..9):

```
     Private Sub ButtonZ_Click(sender As Object, e As EventArgs) Handles Button1.Click,
           Button9.Click, Button8.Click, Button7.Click, Button6.Click, Button5.Click,
              Button4.Click, Button3.Click, Button2.Click, ButtonKomma.Click, Button0.Click
        Dim cmd As Button = CType(sender, Button)
        Label1.Text = Rm.neueZiffer(cmd.Text.Chars(0))
    End Sub
```

Die Operation eingeben (+, -, *, /):

```
     Private Sub ButtonOp_Click(sender As Object, e As EventArgs) _
           Handles ButtonAdd.Click, ButtonSub.Click, ButtonMult.Click, ButtonDiv.Click
        Dim cmd As Button = CType(sender, Button)
        Label1.Text = Rm.setzOperator(cmd.Text.Chars(0))
    End Sub
```

Ergebnis anzeigen (=):

```
    Private Sub ButtonResult_Click(sender As Object, e As EventArgs) Handles ButtonResult.Click
        Label1.Text &= Rm.zeigErgebnis
    End Sub
```

Letztes eingegebenes Zeichen löschen (CE):

```
    Private Sub ButtonCE_Click(sender As Object, e As EventArgs) Handles ButtonCE.Click
        Label1.Text = Rm.lösche Ziffer()
    End Sub
```

Alle Register sowie Anzeige löschen und Anfangszustand wieder herstellen:

```
    Private Sub ButtonCLR_Click(sender As Object, e As EventArgs) Handles ButtonCLR.Click
        Rm.löschAlles()
        Label1.Text = String.Empty
    End Sub
```

Vorzeichenwechsel (+/-):

```
    Private Sub ButtonVZ_Click(sender As Object, e As EventArgs) Handles ButtonVZ.Click
        Label1.Text = Rm.ändereVZ()
    End Sub

End Class
```

Test

Sie werden keinerlei Unterschied in Aussehen und Funktion unseres "dünnen" Taschenrechners zu seinem "fetten" Vorgänger feststellen, was uns in der Auffassung bestätigt, dass den Hauptnutzen aus der OOP nicht der Anwender, sondern der Programmierer hat!

Bemerkungen

- Unter Einsatz einer Formelparser-Klasse wären auch Klammerrechnungen möglich, das dürfte weniger aufwändig sein als das Hinzufügen weiterer Register.
- Einen wesentlich leistungsfähigeren (wissenschaftlichen) Taschenrechner, der allerdings nach einem völlig anderen Prinzip funktioniert (Code DOM), finden Sie im Praxisbeispiel 3.12.8.

3.12.4 Schnittstellenvererbung verstehen

Ein mächtiges Feature der OOP ist das Konzept der Schnittstellenvererbung (siehe Abschnitt 3.11). Die Schnittstellenvererbung erschließt sich dem Einsteiger am leichtesten, wenn er sich vorher das Konzept abstrakter Klassen und Methoden (siehe Abschnitt 3.10.1) verinnerlicht hat.

In unserem Demobeispiel wollen wir Geldbeträge auf das Konto eines Kunden einzahlen bzw. von dort abheben. Die erste Lösung soll mittels einer abstrakten Klasse erfolgen[1]. Die zweite Lösung benutzt eine Schnittstelle (Interface).

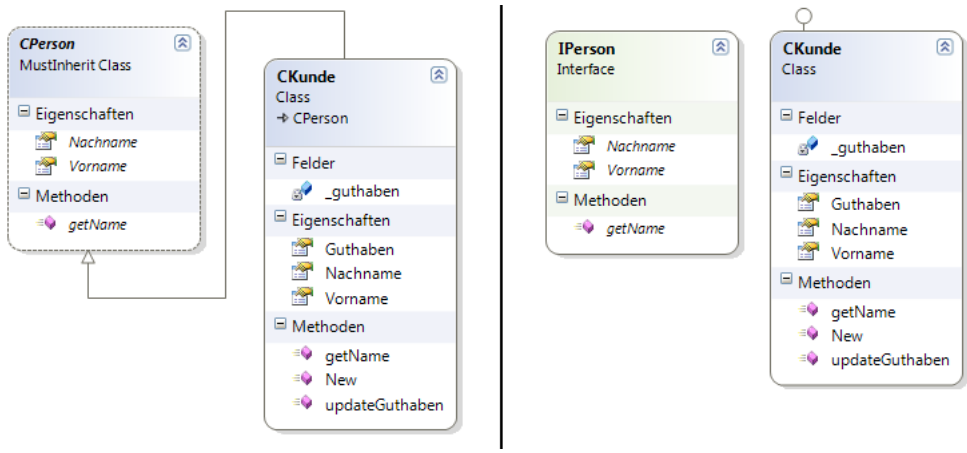
Klassendiagramme

Die erste Variante zeigt das links abgebildete Klassendiagramm. Hier erbt die Klasse *CKunde* von der abstrakten Klasse *CPerson*. Letztere verfügt ausschließlich über abstrakte Klassenmitglieder.

[1] Dies ist allerdings die extremste Form der Implementierungsvererbung, denn es wird de facto keinerlei Code vererbt.

Diese enthalten nur die Eigenschafts- bzw. Methodendeklaration, also keinerlei Code. Die Implementierung muss komplett in der erbenden Klasse *CKunde* erfolgen.

Das rechte Klassendiagramm zeigt die zweite Lösung, bei welcher die abstrakte Klasse *CPerson* von der Schnittstelle *IPerson* ersetzt wird. Weitere Unterschiede sind auf den ersten Blick nicht zu erkennen, dazu müssen wir uns den Quellcode näher anschauen.



Oberfläche Form1

Öffnen Sie eine neue Windows Forms-Anwendung und gestalten Sie das Startformular wie in der Laufzeitansicht gezeigt.

Lassen Sie uns mit der ersten Variante beginnen!

Quellcode CPerson

Fügen Sie dem Projekt eine neue Klasse *CPerson* hinzu:

```
Public MustInherit Class CPerson
    Public MustOverride Property Nachname As String
    Public MustOverride Property Vorname As String
    Public MustOverride Function getName() As String
End Class
```

Wie Sie sehen, ist die Klasse abstrakt und enthält die abstrakten Eigenschaften *Nachname* und *Vorname*) sowie die abstrakte Methode *getName*.

Quellcode CKunde

Fügen Sie dem Projekt eine Klasse *CKunde* hinzu:

```
Public Class CKunde
    Inherits CPerson

    Private _guthaben As Decimal
```

Die drei von *CPerson* geerbten abstrakten Klassenmitglieder müssen überschrieben werden (wir verwenden selbst implementierende Eigenschaften, die die entsprechenden Zustandsvariablen *_Vorname* und *_Nachname* automatisch im Hintergrund anlegen):

```
    Public Overrides Property Vorname As String
    Public Overrides Property Nachname As String

    Public Overrides Function getName() As String
        Return _Vorname & " " & _Nachname
    End Function
```

Es folgen die normalen Klassenmitglieder:

```
    Public Sub New(vor As String, nach As String)       ' ein Konstruktor
        _Vorname = vor
        _Nachname = nach
    End Sub

    Public ReadOnly Property Guthaben As Decimal
        Get
            Return _guthaben
        End Get
    End Property

    Public Sub updateGuthaben(betrag As Decimal)
        _guthaben += betrag
    End Sub
End Class
```

HINWEIS: Es müssen **alle** geerbten abstrakten Klassenmitglieder überschrieben werden, ansonsten erfolgt eine Fehlermeldung des Compilers.

Quellcode Form1

```
Public Class Form1
```

Zu Beginn wird ein Kunde erzeugt, initialisiert und angezeigt:

```
    Private kunde1 As CKunde
    Private Sub Form1_Load(sender As Object, e As EventArgs) Handles MyBase.Load
        kunde1 = New CKunde("Max", "Müller")
        TextBox1.Text = kunde1.Vorname
        TextBox2.Text = kunde1.Nachname
        TextBox3.Text = "10,50"
    End Sub
```

Bei jedem Klick auf die Schaltfläche werden Vor- und Nachname des Kunden neu zugewiesen. Der eingegebene Betrag wird dem Guthaben hinzugefügt bzw. (bei negativem VZ) abgezogen:

```
    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        kunde1.Vorname = TextBox1.Text
```

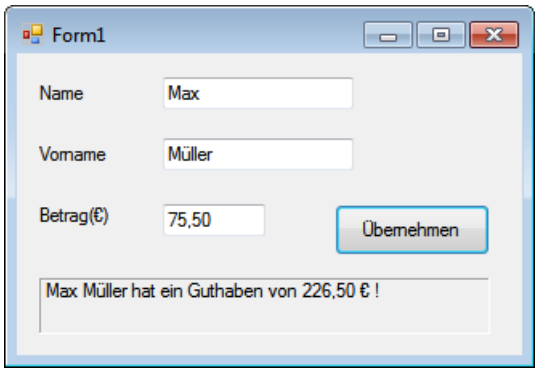
```
        kunde1.Nachname = TextBox2.Text
        Dim betrag As Decimal = Convert.ToDecimal(TextBox3.Text)
        kunde1.updateGuthaben(betrag)
```

Die abgeschlossene Buchung wird mit einem Meldungstext quittiert:

```
        Label1.Text = kunde1.getName() & " hat ein Guthaben von " &
                                          kunde1.Guthaben.ToString("C") & " !"
    End Sub
End Class
```

Test

Nehmen Sie einige Ein- oder Auszahlungen vor.



Nach erfolgreichem Test dieser ersten Variante wollen wir die zweite Variante in Angriff nehmen und die Klasse *CPerson* durch ein Interface *IPerson* ersetzen.

Quellcode IPerson

Benennen Sie einfach im Projektmappenexplorer die Klasse *CPerson* in *IPerson* um und vereinfachen Sie den Code wie folgt:

```
Public Interface IPerson
    Property Nachname As String
    Property Vorname As String
    Function getName() As String
End Interface
```

Quellcode CKunde

Auch hier sind nur minimale Änderungen erforderlich: An die Stelle von *Inherits* tritt *Implements*. Bei den von *IPerson* geerbten Schnittstellenmitgliedern fehlen die *Overrides*-Modifizierer. Stattdessen werden durch abschließende Kennzeichnung der Methodendeklarationen mit *Implements* und Benennung der Schnittstellenmitglieder Mehrdeutigkeiten vermieden.

```
Public Class CKunde
    Implements IPerson

    Private _guthaben As Decimal
    Public Property Vorname As String Implements IPerson.Vorname
    Public Property Nachname As String Implements IPerson.Nachname

    Public Function getName() As String Implements IPerson.getName
        Return _Vorname & " " & _Nachname
    End Function
...
```

Der restliche Code bleibt unverändert.

Das war es auch schon, denn der Quellcode von *Form1*, in welchem die Klasse CKunde instanziiert und verwendet wird, gleicht bis ins letzte Detail dem seines Vorgängers.

Auch beim Testen des Codes werden Sie keinerlei Veränderungen zum Vorgängerprojekt feststellen.

Vergleichen Sie beide Varianten, so stellen Sie fest, dass die Realisierung mittels Schnittstelle die Transparenz und Übersichtlichkeit des Codes deutlich steigert.

3.12.5 Aggregation und Vererbung gegenüberstellen

Jeder Programmierer hat den Ehrgeiz, mit möglichst wenig Schreibarbeit auszukommen und möglichst viel von seinem Code wieder verwenden zu können. Voraussetzung dafür sind optimale Klassendiagramme, für die es unter dem Aspekt der Wiederverwendbarkeit von Code zwei wesentliche Beziehungen gibt:

- Aggregation/Komposition
- Vererbung

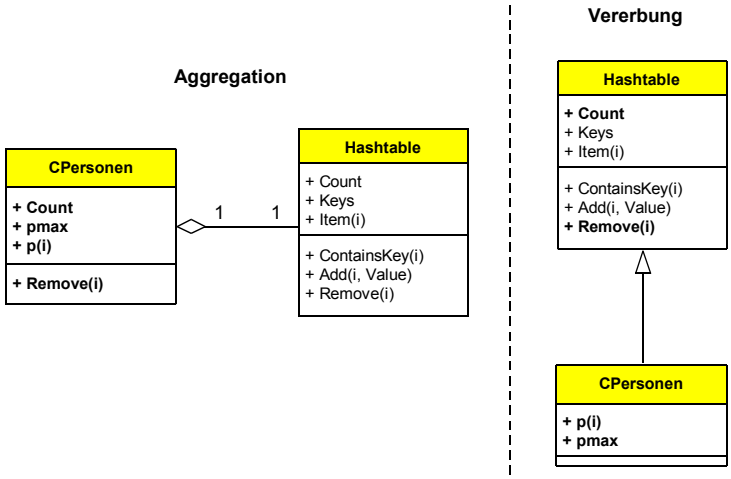
Damit sind wir bereits bei zwei fundamentalen Begriffen der OOP angelangt. Ob wir von *Aggregation* oder *Komposition* sprechen ist in diesem Zusammenhang unerheblich, denn die *Komposition* stellt lediglich die stärkere Form der *Aggregation* dar.

Unter Vererbung ist hier genau genommen die *Implementierungsvererbung* gemeint, denn die unter .NET ebenfalls mögliche *Interfacevererbung* erspart keinerlei Schreibarbeit.

HINWEIS: Sowohl Aggregation/Komposition als auch Vererbung verlangen eine spezifische Herangehensweise bei der Implementierung, die clientseitige Nutzung der Klasse ist aber identisch.

In unserem Beispiel soll dies an Hand einer kleinen Personalverwaltung demonstriert werden, wobei alle Personen in einer Auflistung gespeichert sind, welche die Funktionalität der recht leistungsfähigen *Hashtable*-Klasse nutzt.

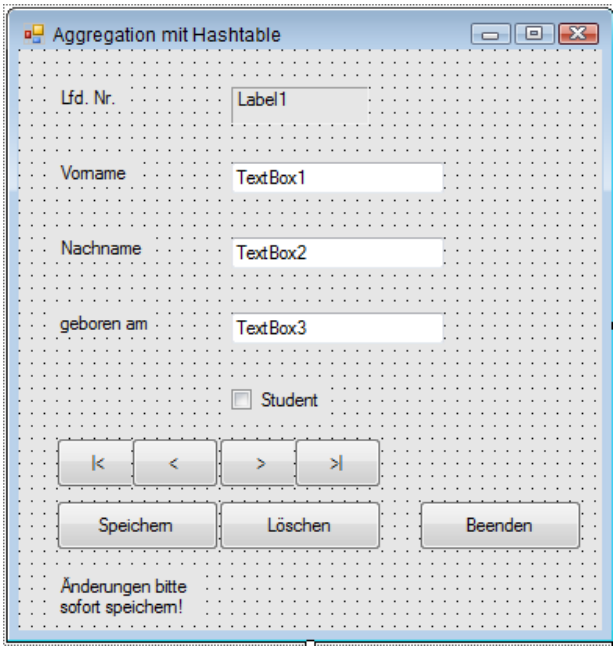
In der folgenden Abbildung sind die Klassendiagramme beider Varianten gegenübergestellt.



Die Klasse *CPersonen* soll über die Standardeigenschaft *p(i)* einen indizierten Lese- und Schreibzugriff auf die Elemente der Auflistung bereitstellen, um damit die *Item*- und die *Add*-Methode von *Hashtable* zu kapseln. Weitere Eigenschaften sind *Count* (Gesamtzahl der abgespeicherten Personen) sowie *pmax* (höchster Index bzw. Schlüsselwert innerhalb der Auflistung). Die von *Hashtable* direkt geerbte *Remove*-Methode ermöglicht das Löschen einer bestimmten Person.

Bedienoberfläche

Das abgebildete Hauptformular bedarf wohl keiner besonderen Erläuterung:



Variante 1: Klassen CPerson und CPersonen mit Aggregation

```
Public Class Form1
```

Der Übersichtlichkeit halber haben wir hier den Code beider Klassen zum *Form1*-Klassenmodul hinzugefügt, Sie könnten ihn aber auch ohne weiteres in ein oder zwei separate Klassenmodule auslagern.

```
    Public Class CPerson
        Public Vorname, Nachname As String
        Public Geburt As Date
        Public student As Boolean
    End Class

    Public Class CPersonen
```

Innerhalb der Klasse *CPersonen* wird die Klasse *Hashtable* instanziiert – das ist Aggregation pur!

```
        Private ht As New Hashtable()
```

Mit einem kleinen Trick, wir definieren eine Eigenschaft *p* als Standardeigenschaft, implementieren wir einen *Quasi-Indexer*[1] für den indizierten Zugriff auf die Personenliste.

HINWEIS: Mittels dieser Standardeigenschaft kann eine *CPersonen*-Collection auf die gleiche Weise wie ein Array indiziert werden!

```
        Default Public Property p(i As Integer) As CPerson
            Get
                If ht.ContainsKey(i) Then
                    Return CType(ht.Item(i), CPerson)
                Else
                    Return Nothing
                End If
            End Get
            Set(ByVal value As CPerson)
                If ht.ContainsKey(i) Then
                    ht.Item(i) = value        ' überschreiben, falls Schlüssel vorhanden
                Else
                    ht.Add(i, value)          ' anhängen, falls Schlüssel noch nicht besetzt
                End If
            End Set
        End Property
```

Das Löschen eines Elements der Auflistung:

```
        Public Sub Remove(i As Integer)
            ht.Remove(i)
        End Sub
    End Class
```

[1] Im Unterschied zu C# erlaubt Visual Basic nicht die direkte Implementierung eines Indexers.

Nachdem die Klassen *CPerson* und *CPersonen* implementiert sind, geht es mit dem eigentlichen Code von *Form1* weiter.

Objekte instanziieren:

```
Private pListe As New CPersonen()     ' die Personenliste
Private person As CPerson             ' die aktuelle Person
```

Zustandsvariablen zum Steuern der Anzeige:

```
Private pos As Integer = 1            ' die aktuelle Position
Private pmax As Integer               ' die max. Anzahl von Personen
```

Die Startaktivitäten:

```
Protected Overrides Sub OnLoad(e As EventArgs)
    Label1.Text = pos.ToString()
    anzeigeLöschen()
    MyBase.OnLoad(e)
End Sub
```

Die Anzeige der aktuellen Person:

```
Private Sub anzeigen()
    Label1.Text = pos.ToString()
    person = pListe(pos)              ' Zugriff wie über Indexer!
    Try
        With person
            TextBox1.Text = .Vorname
            TextBox2.Text = .Nachname
            TextBox3.Text = .Geburt.ToString("dd.MM.yyyy")
            CheckBox1.Checked = .student
        End With
    Catch
        anzeigeLöschen()
    End Try
End Sub
```

Die Hilfsroutine zum Löschen der Anzeige:

```
 Private Sub anzeigeLöschen()
    Label1.Text = String.Empty
    TextBox1.Text = String.Empty : TextBox2.Text = String.Empty
    TextBox3.Text = "00:00:00"
    CheckBox1.Checked = False
End Sub
```

Vorwärts blättern:

```
Private Sub Button3_Click(sender As Object, e As EventArgs) Handles Button3.Click
    pos += 1
    anzeigen()
End Sub
```

Rückwärts blättern:

```
    Private Sub Button2_Click(sender As Object, e As EventArgs) Handles Button2.Click
        If pos > 1 Then
            pos -= 1
            anzeigen()
        End If
    End Sub
```

Zum Anfang:

```
    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        pos = 1
        anzeigen()
    End Sub
```

Zum Ende:

```
    Private Sub Button4_Click(sender As Object, e As EventArgs) Handles Button4.Click
        pos = pmax
        anzeigen()
    End Sub
```

Speichern der aktuellen Person:

```
    Private Sub Button5_Click(sender As Object, e As EventArgs) Handles Button5.Click
        person = New CPerson()            ' nur Instanzen können hinzugefügt werden!!!
        With person
            .Vorname = TextBox1.Text
            .Nachname = TextBox2.Text
            .Geburt = Convert.ToDateTime(TextBox3.Text)
            .student = CheckBox1.Checked
        End With
        pListe(pos) = person              ' Zugriff wie über Indexer
        If pos > pmax Then pmax = pos     ' max. Schlüsselwert
       anzeigen()
    End Sub
```

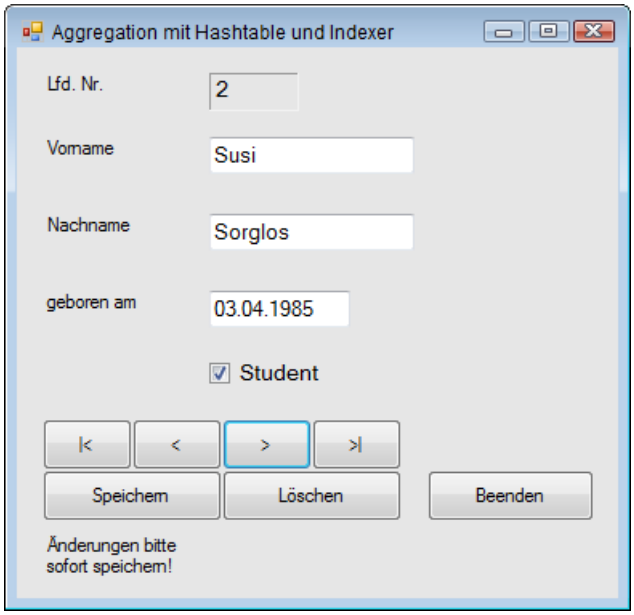
Löschen der aktuellen Person:

```
    Private Sub Button6_Click(sender As Object, e As EventArgs) Handles Button6.Click
        pListe.Remove(pos)
        anzeigeLöschen()
    End Sub
    ...
End Class
```

Test

Es können beliebig viele Personen eingegeben werden. Der zugeteilte Schlüsselwert entspricht der "Lfd.Nr"-Anzeige. Wenn unter dem Schlüssel bereits eine Person existiert, wird diese überschrieben, anderenfalls neu angelegt.

HINWEIS: Das Abspeichern passiert nicht automatisch beim Weiterblättern, sondern Sie müssen **vor** dem Weiterblättern die *Speichern*-Schaltfläche klicken (ansonsten sind die Änderungen futsch)!



Variante 2: Klasse CPersonen mit Vererbung

Alternativ zur Aggregation können wir die Klasse *CPersonen* auch so implementieren, dass die Eigenschaften/Methoden direkt von der Klasse *Hashtable* "geerbt" werden (siehe Abbildung zu Beginn).

```
Public Class CPersonen
    Inherits Hashtable              ' Vererbung

    Default Public Property p(i As Integer) As CPerson
        Get
            If ContainsKey(i) Then
                Return CType(Item(i), CPerson)
            Else
                Return Nothing
            End If
        End Get
        Set(ByVal value As CPerson)
            If ContainsKey(i) Then
                Item(i) = value             ' überschreiben, falls Schlüssel vorhanden
            Else
                Add(i, value)               ' anhängen, falls Schlüssel noch nicht besetzt
            End If
```

```
            End Set
        End Property
    End Class
```

Vergleichen Sie diesen Code mit der ersten Variante, so stellen Sie fest, dass eine Instanziierung von *Hashtable* nicht mehr erforderlich ist. Stattdessen können die benötigten Eigenschaften und Methoden der Basisklasse direkt aufgerufen werden.

Da dank Implementierungsvererbung alle öffentlichen Eigenschaften/Methoden von *Hashtable* jetzt auch in der Schnittstelle von *CPersonen* verfügbar sind, entfällt auch die Implementierungen der *Remove*-Methode.

Bis auf die vereinfachte Klasse *CPersonen* sind, gegenüber der ersten Variante (Aggregation) keine weiteren Unterschiede festzustellen, der Client "sieht" die gleiche Schnittstelle.

Test

Das Ergebnis ist erwartungsgemäß identisch mit Variante 1.

HINWEIS: Damit der mühsam eingegebene Personalbestand das Ausschalten des Rechners überlebt, ist das Abspeichern in eine Datei erforderlich.

3.12.6 Eine Klasse zur Matrizenrechnung entwickeln

Eine Matrix ist nichts weiter wie der mathematische Begriff für ein Array. In diesem Beispiel soll am Beispiel einer Klasse *CMatrix* die grundlegende Vorgehensweise bei der Entwicklung einer Klasse erläutert werden, die schon etwas anspruchsvoller ist als z.B. eine triviale *CPerson*-Klasse.

Die Schwerpunktthemen sind:

- überladener Konstruktor
- überladene Methoden
- Eigenschaftsmethoden
- Standardeigenschaft als Indexer
- Unterschied zwischen statischen (Shared-) Methoden und Instanzen-Methoden

Die Klasse *CMatrix* soll Funktionalität zur Verfügung stellen, die Sie zur Ausführung von Matrixoperationen benötigen (Addition, Multiplikation...).

Obwohl wir hier nur die Addition implementieren werden, kann die Klasse von Ihnen nach dem gezeigten Muster selbständig um weitere Matrizenoperationen erweitert werden, wie z.B. Multiplikation oder Inversion.

HINWEIS: Wer sich nicht für Mathematik interessiert, kann das Beispiel trotzdem sehr gut verwenden, da der Schwerpunkt auf den verwendeten Programmiertechniken im Zusammenhang mit dem Array-Zugriff liegt!

Quellcode der Klasse CMatrix

Wir beginnen diesmal nicht mit dem Startformular (*Form1*), sondern erweitern zunächst über den Menüpunkt *Projekt|Klasse hinzufügen...* unser Projekt um eine neue Klasse mit dem Namen *CMatrix*.

Die Klasse *CMatrix* verwaltet ein zweidimensionales Array aus *Double*-Zahlen. Die Zustandsvariablen *_rows* und *_cols* speichern die Anzahl der Zeilen und Spalten.

```
Public Class CMatrix
    Private _rows, _cols As Integer
    Private _array(,) As Double
```

Ein neues Array wird über den Konstruktor instanziiert, der in zwei Versionen vorliegt. Falls Sie später *New()* ohne Argument aufrufen, wird eine Matrix mit einem einzigen Element generiert, ansonsten mit den gewünschten Dimensionen.

```
    Sub New()
        MyBase.New()
        _rows = 1
        _cols = 1
        ReDim _array(_rows, _cols)
    End Sub

    Sub New(R As Integer, C As Integer)      ' überladener Konstruktor
        MyBase.New()
        _rows = R
        _cols = C
        ReDim _array(_rows, _cols)
    End Sub
```

Der Zugriff auf die (privaten) Zustandsvariablen *_rows* und *_cols* wird über die Eigenschaften *Rows* und *Cols* gekapselt.

```
    Public Property Rows() As Integer          ' Eigenschaft zum Zugriff auf Zeilenanzahl
        Get
            Return _rows
        End Get
        Set(Value As Integer)
            _rows = Value
        End Set
    End Property

    Public Property Cols() As Integer         ' Eigenschaft zum Zugriff auf Spaltenanzahl
        Get
            Return _cols
        End Get
```

```
        Set(Value As Integer)
            _cols = Value
        End Set
    End Property
```

Der Zugriff auf ein bestimmtes Matrix-Element wird elegant über die Standardeigenschaft realisiert, die hier quasi wie ein Indexer funktioniert:

```
    Default Public Property Cell(row As Integer, col As Integer) As Double
        Get
            Return _array(row, col)
        End Get
        Set(Value As Double)
            _array(row, col) = Value
        End Set
    End Property
```

Die *Add*-Methode akzeptiert entweder ein oder zwei *CMatrix*-Objekte als Parameter, falls Sie nur ein *CMatrix*-Objekt übergeben, wird die aktuelle Instanz der Matrix als zweiter Operand verwendet.

Die erste Überladung der *Add*-Methode ist statisch, sie wird also nicht über einem *CMatrix*-Objekt, sondern direkt über der *CMatrix*-Klasse ausgeführt! Die Methode nimmt beide Operanden (*CMatrix*-Objekte) entgegen und liefert ein *CMatrix*-Objekt zurück.

```
    Public Overloads Shared Function Add(A As CMatrix, B As CMatrix) As CMatrix
        If Not (A.Rows = B.Rows And A.Cols = B.Cols) Then
            Add = New CMatrix()
            Exit Function
        End If
        Dim newMatrix As New CMatrix(A.Rows, A.Cols)
        For row As Integer = 0 To A.Rows - 1
            For col As Integer = 0 To A.Cols - 1
                newMatrix(row, col) = A(row, col) + B(row, col)
            Next
        Next
        Return newMatrix
    End Function
```

Obige Methode wird mit einer leeren "Verlegenheitsmatrix" verlassen, wenn beide Operanden nicht die gleichen Dimensionen aufweisen sollten.

Bei der zweiten Überladung handelt es sich um eine normale Instanzen-Methode, sie nimmt als Parameter nur ein einziges *CMatrix*-Objekt entgegen. Der zweite Operand ist naturgemäß die aktuelle *CMatrix*-Instanz, die diese Methode aufruft.

```
    Public Overloads Function Add(A As CMatrix) As CMatrix
        If Not (A.Rows = MyClass.Rows And A.Cols = MyClass.Cols) Then
            Return New CMatrix()
            Exit Function
        End If
        Dim newMatrix As New CMatrix(MyClass.Rows, MyClass.Cols)
```

```
        For row As Integer = 0 To MyClass.Rows - 1
            For col As Integer = 0 To MyClass.Cols - 1
                newMatrix(row, col) = A(row, col) + MyClass.Cell(row, col)
            Next
        Next
        Return newMatrix
    End Function

End Class
```

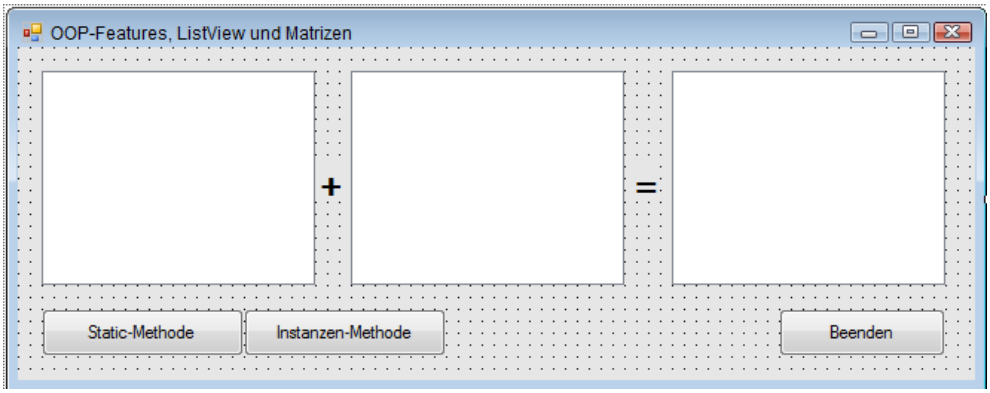
Der Unterschied zwischen statischen- und Instanzen-Methode dürfte Ihnen so richtig erst beim Sichten des Codes von *Form1* klar werden, wo beide Überladungen aufgerufen werden.

Hier ein Vorgriff auf den Code von *Form1*:

```
    Dim A, B, C As CMatrix
    ...
    C = CMatrix.Add(A, B)               ' Aufruf der  statischen Methode
    C = A.Add(B)                        ' Aufruf der Instanzen-Methode
```

Oberfläche

Wir benötigen drei *ListView*-Komponenten und drei Schaltflächen. Setzen Sie folgende zwei Eigenschaften für jede *ListView*: *View* = *Details* und *GridLines* = *True*.



Quellcode von Form1

```
Public Class Form1
```

Wir verwenden für beide Schaltflächen einen gemeinsamen Eventhandler:

```
    Private Sub Button_Click(sender As Object, e As EventArgs) _
                                                Handles Button1.Click, Button2.Click
```

Unser Beispiel benutzt Matrizen mit 9 Zeilen und 6 Spalten:

```
        Const rows As Integer = 9    ' Anzahl Zeilen
        Const cols As Integer = 6    ' Anzahl Spalten
```

Zufallszahlengenerator instanziieren:

```
Dim rnd As New System.Random()
```

Die Matrix *A* instanziieren, mit Zufallszahlen füllen und anzeigen (man beachte den bequemen Zugriff über den Indexer!):

```
 Dim A As New CMatrix(rows - 1, cols - 1)
For i As Integer = 0 To A.Rows - 1
    For j As Integer = 0 To A.Cols - 1
        A(i, j) = rnd.Next(100)           ' Zugriff auf Matrixelement über Indexer!
    Next
Next
showListView(A, ListView1)                ' Anzeige in linker ListView
```

Gleiches geschieht mit Matrix *B*:

```
Dim B As New CMatrix(rows - 1, cols - 1)
For i As Integer = 0 To B.Rows - 1
    For j As Integer = 0 To B.Cols - 1
        B(i, j) = rnd.Next(100)
    Next
Next
showListView(B, ListView2)                ' Anzeige in mittlerer ListView
```

Die resultierende Matrix *C* berechnen wir – in Abhängigkeit vom geklickten *Button* – mit der ersten oder mit der zweiten Überladung der *Add*-Methode.

HINWEIS: Beide Überladungen der *Add*-Methode leisten absolut das Gleiche, nur die Aufruf-Syntax ist unterschiedlich!

```
Dim C As CMatrix
If CType(sender, Button) Is Button1 Then
    C = CMatrix.Add(A, B)                 ' Aufruf Shared-Methode
Else
    C = A.Add(B)                          ' Aufruf Instanzen-Methode
End If
showListView(C, ListView3)                ' Anzeige in rechter ListView
End Sub
```

Der Anzeigeroutine *showListView* werden ein *CMatrix*-Objekt und eine *ListView*-Komponente übergeben:

```
Private Sub showListView(M As CMatrix, lv As ListView)
    With lv
        .Clear()
```

Alle Spalten erzeugen und beschriften:

```
        .Columns.Add("", 20, HorizontalAlignment.Right)     ' linke (leere) Randspalte
        For j As Integer = 0 To M.Cols - 1
```

Spaltennummerierung und Formatierung in Kopfzeile:

```
                .Columns.Add(j.ToString, 30, HorizontalAlignment.Right)
            Next
```

Alle Zeilen erzeugen, beschriften und Zellen füllen:

```
            For i As Integer = 0 To M.Rows - 1
```

Pro Zeile ein *ListViewItem*, Zeilennummerierung in linke Randspalte eintragen:

```
                Dim item As New ListViewItem(i.ToString)
                For j As Integer = 0 To M.Cols - 1
```

Alle Zellen füllen (pro Zelle ein *SubItem*):

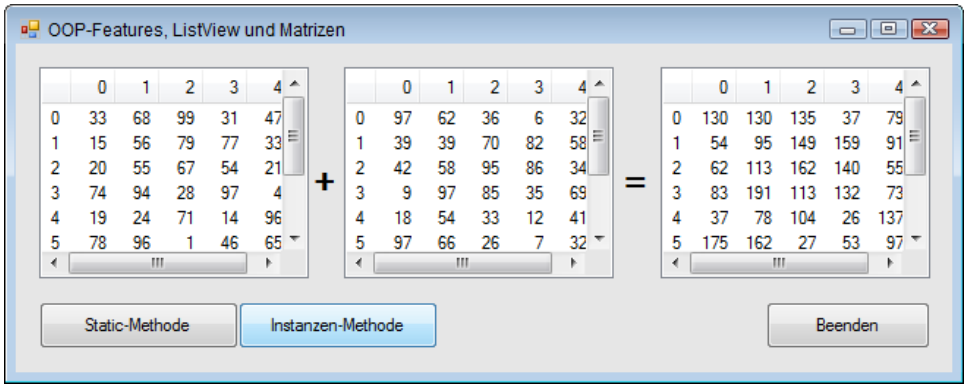
```
                    item.SubItems.Add(M(i, j).ToString)
                Next
```

Zeile zur *ListView* hinzufügen:

```
               .Items.Add(item)
            Next
        End With
    End Sub
End Class
```

Test

Nach Programmstart werden die beiden ersten Matrizen mit Zufallszahlen zwischen 0 und 100 gefüllt. Ob Sie dann *Button1* oder *Button2* klicken ist völlig egal, in beiden Fällen wird die resultierende Summenmatrix mit dem richtigen Ergebnis gefüllt:



Bemerkung

Das Resultat einer Matrix-Operation ist immer eine neue Matrix, wenn allerdings beide Matrizen inkompatibel sind, wird eine leere Matrix zurückgegeben. Alternativ könnte in einem solchen Fall auch eine Ausnahme innerhalb der Methoden erzeugt werden (*Throw New System.ArgumentException()*).

3.12.7 Rechner für komplexe Zahlen

Auch mit dieser Anwendung wollen wir nicht nur die Lösung eines mathematischen Problems zeigen, sondern (was viel wichtiger ist) grundlegendes Handwerkszeug des .NET-Programmierers demonstrieren:

- Sinnvolle Auslagerung von Quellcode in Klassen, um das Verständnis der OOP zu vertiefen,
- Prinzip der Operatorenüberladung in Visual Basic,
- Strukturierung des Codes der Benutzerschnittstelle nach dem EVA-Prinzip (Eingabe – Verarbeitung – Ausgabe).

Doch ehe wir mit der Praxis beginnen, scheint ein kurzer Abstieg in die Untiefen der Mathematik unumgänglich.

Was sind komplexe Zahlen?

Eine besondere Bedeutung haben komplexe Zahlen beispielsweise in der Schwingungslehre und in der Wechselstromtechnik, einem bedeutenden Teilgebiet der Elektrotechnik.

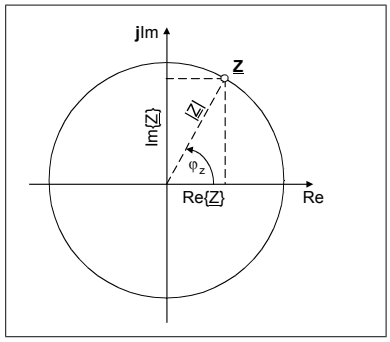
Zur Darstellung einer komplexen Zahl **Z** bieten sich zwei Möglichkeiten an:

- Kartesische Koordinaten (Real-/Imaginärteil)
- Polarkoordinaten (Betrags-/Winkeldarstellung)

Die folgende Tabelle zeigt eine Zusammenstellung der Umrechnungsformeln:

Kartesische Koordinaten	Polarkoordinaten
$\underline{Z} = \mathrm{Re}\{\underline{Z}\} + j\,\mathrm{Im}\{\underline{Z}\}$	$\underline{Z} = \lvert\underline{Z}\rvert\, e^{j\varphi_z}$
Realteil: $\mathrm{Re}\{\underline{Z}\} = \lvert\underline{Z}\rvert \cos\varphi_z$	Betrag: $\lvert\underline{Z}\rvert = \sqrt{(\mathrm{Re}\{Z\})^2 + (\mathrm{Im}\{Z\})^2}$
Imaginärteil: $\mathrm{Im}\{\underline{Z}\} = \lvert\underline{Z}\rvert \sin\varphi_z$	Phasenwinkel: $\varphi_Z = \arctan\frac{\mathrm{Im}\{Z\}}{\mathrm{Re}\{Z\}}$

Am besten lassen sich diese Zusammenhänge am Einheitskreis erläutern, wobei **Z** als Punkt in der komplexen Ebene erscheint:



Die kartesische Form eignet sich besonders gut für die Ausführung von Addition und Subtraktion:

Mit

$$\mathbf{Z_1 = a_1 + jb_1} \quad \text{und} \quad \mathbf{Z_2 = a_2 + jb_2}$$

ergibt sich

$$\mathbf{Z_1 + Z_2 = a_1 + a_2 + j(b_1 + b_2)} \quad \text{bzw.} \quad \mathbf{Z_1 - Z_2 = a_1 - a_2 + j(b_1 - b_2)}$$

Andererseits bevorzugt man für Multiplikation und Division die Zeigerform:

Mit

$$\mathbf{Z_1 = c_1 \cdot e^{j\varphi_1}} \quad \text{und} \quad \mathbf{Z_2 = c_2 \cdot e^{j\varphi_2}}$$

erhalten wir

$$\mathbf{Z_1 \cdot Z_2 = c_1 \cdot c_2 \cdot e^{j(\varphi_1 + \varphi_2)}} \quad \text{bzw.} \quad \mathbf{Z_1/Z_2 = c_1/c_2 \cdot e^{j(\varphi_1 - \varphi_2)}}$$

Für die Angabe des Phasenwinkels hat man die Wahl zwischen Radiant (Bogenmaß) und Grad. Für die gegenseitige Umrechnung gilt die Beziehung

$$\varphi(Rad) = \frac{\pi}{180}\varphi(Grad)$$

HINWEIS: Die Maßeinheit "Grad" wird aufgrund ihrer Anschaulichkeit vom Praktiker für die Ein- und Ausgabe bevorzugt, während "Radiant" für interne Berechnungen günstiger ist.

Programmierung der Klasse CComplexN

Öffnen Sie ein neues Projekt vom Typ Windows Forms-Anwendung. Das Startformular *Form1* lassen Sie zunächst unbeachtet liegen, denn der routinierte .NET-Programmierer kapselt seinen Code in Klassen anstatt ihn einfach zum Formularcode hinzuzufügen.

Die zweckmäßige Aufteilung einer praktischen Problemstellung in verschiedene Klassen und die Definition der Abhängigkeiten ist sicherlich der schwierigste Part der OOP und erfordert einige Übung und Routine, bis das dazu erforderliche abstrakte Denken schließlich zur Selbstverständlichkeit wird[1].

Die hier vorgeschlagene Lösung benutzt die Klasse *CComplexN*, welche eine komplexe Zahl repräsentiert. Diese Klasse speichert in den Zustandsvariablen *Re* und *Im* (die in unserem Fall gleichzeitig Eigenschaften sind) den Wert der komplexen Zahl in Kartesischen Koordinaten. Die beiden anderen Eigenschaften (*Len* und *Ang*) repräsentieren dieselbe Zahl in Polar-Koordinaten. Allerdings werden *Len* und *Ang* nicht direkt in den Objekten gespeichert, sondern in so genannten *Eigenschaftenmethoden* (*property procedures*) jeweils aus *Re* und *Im* berechnet.

Über das Menü *Projekt|Klasse hinzufügen...* erstellen Sie den Rahmencode der Klasse.

[1] Die UML (Unified Modelling Language) stellt dazu spezielle Werkzeuge bereit.

```
Public Class CComplexN
```

Die beiden öffentlichen Zustandsvariablen *Re* und *Im* bilden das "Gedächtnis" der Klasse und können quasi wie Eigenschaften benutzt werden[1]:

```
    Public Re, Im As Double                  ' Real- und Imaginärteil
```

Ein Konstruktor (den Rahmencode können Sie sich von der IDE erzeugen lassen) setzt die Zustandsvariablen auf ihre Anfangswerte:

```
    Public Sub New(r As Double, i As Double)
        Re = r : Im = i
    End Sub
```

Die "intelligente" Eigenschaftsmethode *Ang* berechnet den Phasenwinkel aus den Zustandsvariablen *Re* und *Im*:

```
    Public Property Ang() As Double
        Get
                Dim g As Double = 0
                If Re <> 0 Then
                g = 180 / Math.PI * Math.Atan(Im / Re)
                If Re < 0 Then g += 180
                Else
                If Im <> 0 Then
                      If Im > 0 Then g = 90
                Else
                     g = -90
                 End If
               End If
               Return g
            End Get
            Set(value As Double)
                Dim b, l As Double
                b = value * Math.PI / 180
                l = Math.Sqrt(Re * Re + Im * Im)
                Re = l * Math.Cos(b)        ' neuer Realteil
                Im = l * Math.Sin(b)        ' neuer Imaginärteil
          End Set
        End Property
```

Die Eigenschaft *Len* ermittelt den Betrag (die Länge des Zeigers) aus *Re* und *Im*:

```
        Public Property Len() As Double
          Get
             Return Math.Sqrt(Re * Re + Im * Im)
          End Get
          Set(value As Double)
             Dim b As Double = Math.Atan(Im / Re)
```

[1] Die Verwendung öffentlicher Zustandsvariabler als Eigenschaften ist zwar nicht der sauberste, in unserem Fall aber der effektivste Weg.

```
            Re = value * Math.Cos(b)
            Im = value * Math.Sin(b)
         End Set
      End Property
```

Besonders interessant sind die folgenden drei (statischen) Methoden, welche die Operatorenüberladungen für Addition, Multiplikation und Division neu definieren.

Der "+"-Operator erhält eine neue Bedeutung, er addiert jetzt zwei komplexe Zahlen:

```
      Public Shared Operator +(a As CComplexN, b As CComplexN) As CComplexN
         Dim z As New CComplexN(0, 0)
         z.Re = a.Re + b.Re
         z.Im = a.Im + b.Im
         Return z
      End Operator
```

Der "*"-Operator multipliziert zwei komplexe Zahlen:

```
      Public Shared Operator *(a As CComplexN, b As CComplexN) As CComplexN
         Dim z As New CComplexN(0, 0)
         z.Re = a.Re * b.Re - a.Im * b.Im
         z.Im = a.Re * b.Im + a.Im * b.Re
         Return z
      End Operator
```

Der "/"-Operator dividiert zwei komplexe Zahlen:

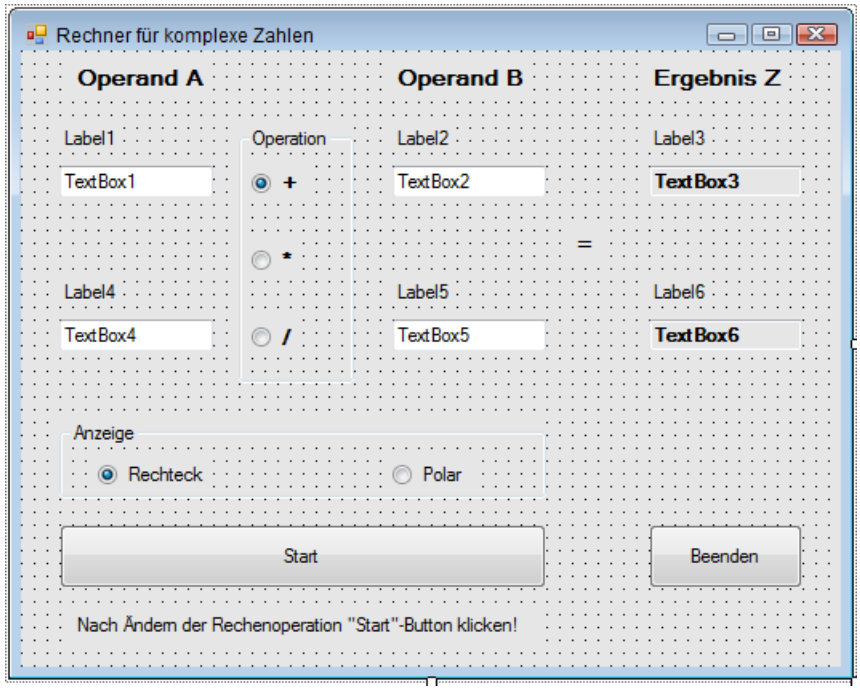
```
       Public Shared Operator /(a As CComplexN, b As CComplexN) As CComplexN
          Dim z As New CComplexN(0, 0)
          z.Re = (a.Re * b.Re + a.Im * b.Im) / (b.Re * b.Re + b.Im * b.Im)
          z.Im = (a.Im * b.Re - a.Re * b.Im) / (b.Re * b.Re + b.Im * b.Im)
          Return z
      End Operator
End Class
```

HINWEIS: Vielleicht sticht Ihnen bereits jetzt ein gravierender Unterschied zur klassischen "Geradeausprogrammierung" ins Auge: Spezielle Methoden zur Umrechnung zwischen Kartesischen- und Polarkoordinaten sind Fehlanzeige, da ein Objekt vom Typ *CComplexN* beide Darstellungen bereits als Eigenschaften kapselt!

Bedienoberfläche für Testprogramm

Um uns von der Funktionsfähigkeit der entwickelten Klassen zu überzeugen, brauchen wir ein kleines Testprogramm, das die Ein- und Ausgabe von komplexen Zahlen und die Auswahl der Rechenoperation sowie der Koordinatendarstellung ermöglicht.

Wir verwenden dazu das bereits vorhandene Startformular *Form1*, das wir entsprechend der folgenden Abbildung gestalten.



HINWEIS: Es kann nicht schaden, wenn Sie *ReadOnly* für *TextBox3* und *TextBox6* auf *True* und *TabStop* auf *False* setzen, da Sie diese beiden rechten Felder nur zur Ergebnisanzeige brauchen.

Quellcode für Testprogramm

Das an legendäre DOS-Zeiten erinnernde EVA-Prinzip (Eingabe, Verarbeitung, Anzeige) hat auch unter .NET nichts von seiner grundlegenden Bedeutung eingebüßt.

Der clientseitige Quellcode entspricht vom prinzipiellen Ablauf her der klassischen Geradeausprogrammierung, ist allerdings deutlich übersichtlicher und problemnäher, denn wir arbeiten mit drei Objektvariablen, die bereits komplexe Zahlen sind und nicht mit einer Vielzahl skalarer Variablen!

```
Public Class Form1
```

Die benötigten Objektvariablen:

```
    Private A As New CComplexN(1, 1)          ' Operand A
    Private B As New CComplexN(1, 1)          ' Operand B
    Private Z As New CComplexN(0, 0)          ' Ergebnis Z
```

Unter Berücksichtigung der eingestellten Anzeigeart (Rechteck- oder Polarkoordinaten) liest die folgende Methode die Werte aus der Eingabemaske in die Objekte:

```
Private Sub Eingabe()
    If RadioButton4.Checked Then   ' Rechteck-Koordinaten
        A.Re = Convert.ToDouble(TextBox1.Text)
        B.Re = Convert.ToDouble(TextBox2.Text)
        A.Im = Convert.ToDouble(TextBox4.Text)
        B.Im = Convert.ToDouble(TextBox5.Text)
    Else                          ' Polar-Koordinaten
        A.Len = Convert.ToDouble(TextBox1.Text)
        B.Len = Convert.ToDouble(TextBox2.Text)
        A.Ang = Convert.ToDouble(TextBox4.Text)
        B.Ang = Convert.ToDouble(TextBox5.Text)
    End If
End Sub
```

Die Verarbeitungsroutine führt die gewünschte Rechenoperation mit den bekannten Symbolen für Addition, Multiplikation und Division aus. Dazu werden die in der Klasse *CComplexN* definierten Operatorenüberladungen benutzt:

```
Private Sub Verarbeitung()
         If RadioButton1.Checked Then Z = A + B      ' Addition
         If RadioButton2.Checked Then Z = A * B      ' Multiplikation
         If RadioButton3.Checked Then Z = A / B      ' Division
End Sub
```

Als Pendant zur *Eingabe*-Methode sorgt die Methode *Ausgabe* für die Anzeige von *A*, *B* und *Z*, wozu auch die Anpassung der Beschriftung der Eingabefelder gehört:

```
Private Sub Anzeige()
    If RadioButton4.Checked Then          ' Anzeige in Rechteck-Koordinaten
        Label1.Text = "Realteil A"
        Label2.Text = "Realteil B"
        Label3.Text = "Realteil Z"
        Label4.Text = "Imaginärteil A"
        Label5.Text = "Imaginärteil B"
        Label6.Text = "Imaginärteil Z"
        TextBox1.Text = A.Re.ToString()     ' Anzeige Realteil A
        TextBox4.Text = A.Im.ToString()     ' Anzeige Imaginärteil A
        TextBox2.Text = B.Re.ToString()     ' Anzeige Realteil B
        TextBox5.Text = B.Im.ToString()     ' Anzeige Imaginärteil B
        TextBox3.Text = Z.Re.ToString()     ' Anzeige Realteil Z
        TextBox6.Text = Z.Im.ToString()     ' Anzeige Imaginärteil Z
    Else                                  ' Anzeige in Polarkoordinaten
        Label1.Text = "Betrag A"
        Label2.Text = "Betrag B"
        Label3.Text = "Betrag Z"
        Label4.Text = "Winkel A"
        Label5.Text = "Winkel B"
        Label6.Text = "Winkel Z"
```

```
            TextBox1.Text = A.Len.ToString()    ' Anzeige Betrag A
            TextBox4.Text = A.Ang.ToString()    ' Anzeige Winkel A
            TextBox2.Text = B.Len.ToString()    ' Anzeige Betrag B
            TextBox5.Text = B.Ang.ToString()    ' Anzeige Winkel B
            TextBox3.Text = Z.Len.ToString()    ' Anzeige Betrag Z
            TextBox6.Text = Z.Ang.ToString()    ' Anzeige Winkel Z
        End If
    End Sub
```

Die *Start*-Schaltfläche:

```
    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        Eingabe()
        Verarbeitung()
        Anzeige()
    End Sub
```

Nach Umschaltung zwischen Rechteck- und Polarkoordinaten muss die Anzeige aktualisiert werden:

```
    Private Sub RadioButton4_CheckedChanged(sender As Object,
                        e As EventArgs) Handles RadioButton4.CheckedChanged
        Anzeige()
    End Sub
...
```

Programmtest

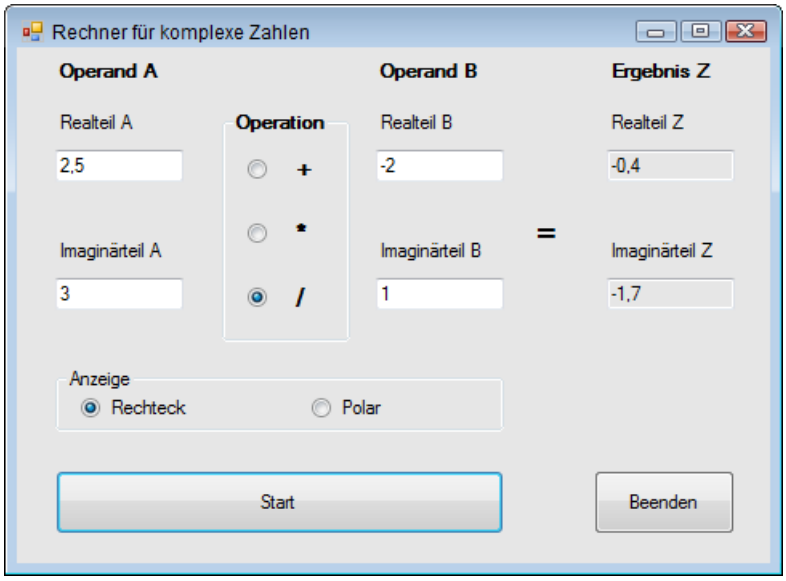
Wenn zum Beispiel die Aufgabe

(2.5 + 3j) / (-2 + j)

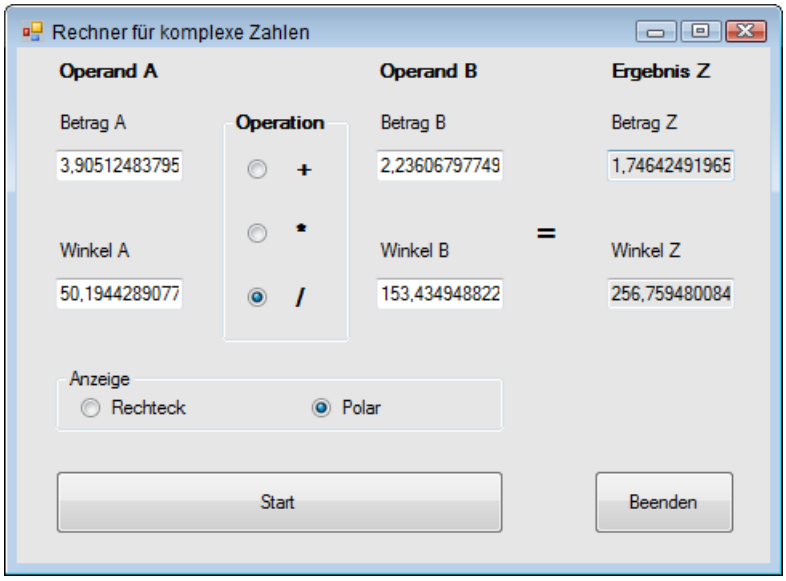
gelöst werden soll, so stellen Sie zunächst die Anzeige auf "Rechteck" ein. Geben Sie dann links oben den Realteil (2,5) und darunter den Imaginärteil (3) des ersten Operanden ein. Analog dazu geben Sie rechts oben den Realteil (-2) und darunter den Imaginärteil (1) des zweiten Operanden ein. Abschließend klicken Sie auf die gewünschte Operation (/).

Nach Betätigen der *Start*-Taste erscheint als Ergebnis die komplexe Zahl -0.4 -1.7j (siehe folgende Abbildung).

Die äquivalenten Polarkoordinaten liefern für das gleiche Beispiel einen Zeiger mit der Länge von ca. 1.746 und einem Winkel von ca. 256.76 Grad.



HINWEIS: Wenn Sie die Anzeige zwischen Rechteck- und Polarkoordinaten umgeschaltet haben, müssen Sie anschließend die *Start*-Schaltfläche klicken!



Bemerkungen

- Die Vorteile eines gut strukturierten, objektorientierten Programms liegen bekanntermaßen in der leichteren Lesbarkeit des Quellcodes ("sprechender" Code) und in der besseren Wartbarkeit und Erweiterungsfähigkeit.
- Beim Arbeiten mit Visual Studio informiert Sie die Intellisense stets aktuell über die vorhandenen Klassenmitglieder und deren Signaturen.
- Bereits mit .NET 4.0 wurde die Klasse *System.Numerics.Complex* eingeführt. Alle Versuche der Autoren, diese Klasse als Ersatz für *CComplexN* zu verwenden und den "Rechner für komplexe Zahlen" damit zu realisieren, scheiterten am unbefriedigenden und praxisfremden Programmiermodell der *Complex*-Klasse. So sind die Eigenschaften *Real* und *Imaginary* schreibgeschützt und nur über den Konstruktor zuweisbar, die Polarkoordinaten hingegen können nur über eine statische Methode gesetzt werden. Der Code wird dadurch unnötig aufgebläht und verliert an Transparenz. Wir haben deshalb auf die Anwendung dieser Klasse verzichtet und bevorzugen weiterhin unsere "Eigenproduktion" *CComplexN*.

HINWEIS: Wer mit der systemeigenen Klasse *Complex* dennoch experimentieren möchte, muss in der Regel vorher einen Verweis auf die *System.Numerics.dll* hinzufügen.

3.12.8 Formel-Rechner mit dem CodeDOM

Jeder, der in einer Mathematik-Ausbildung steht oder im Bereich wissenschaftlich-technischer Anwendungen arbeitet, hat sicher schon vor der Aufgabe gestanden, Berechnungen von Formelausdrücken durchzuführen, sei es um eine Werteliste zu erstellen oder um eine Diagramm auszudrucken.

Wer jetzt befürchtet, dafür erst einen aufwändigen Formelparser entwickeln zu müssen, den können wir beruhigen, denn unter .NET erlaubt das *Code Document Object Model* aus dem Namensraum *System.CodeDOM* eine verblüffend einfache Realisierungsmöglichkeit: Sie können den Quellcode einer .NET-Programmiersprache zur Laufzeit "zusammenbasteln", kompilieren und ausführen! Aus der so erzeugten Assembly kann mittels Reflexion die "zusammengebastelte" Funktion aufgerufen und das Ergebnis ausgewertet werden!

HINWEIS: Ein Rechner nach diesem Prinzip stellt bezüglich seiner Leistungsfähigkeit die bekannten Windows-Taschenrechner weit in den Schatten!

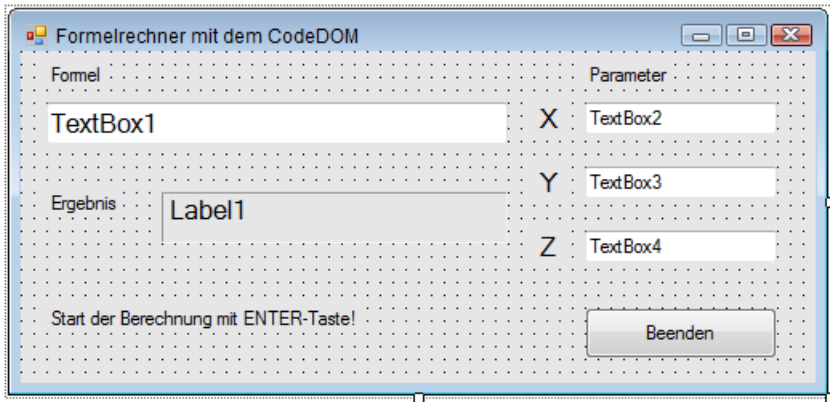
In welcher Sprache Sie die zu berechnende Formel zusammenbauen ist egal, Voraussetzung ist lediglich das Vorhandensein eines zum Compiler passenden *CodeDomProvider*s. Im vorliegenden Fall haben wir uns, wen wundert's, für VB entschieden. Wir hätten aber auch den C#-Provider nehmen können, für den Endanwender ist aber die VB-Syntax einfacher zu verstehen, scheitern doch viele bereits an der peniblen Groß-/Kleinschreibung von C#.

HINWEIS: Ganz abgesehen von seinem Nutzen als universeller Formelrechner bietet dieses Beispiel eine eindrucksvolle Demonstration des Prinzips und der Leistungsfähigkeit des Reflection-Mechanismus von .NET.

Das Grundprinzip des Formelrechners soll zunächst an einer auf das Wesentliche beschränkten Variante demonstriert werden.

Entwurf Bedienoberfläche

Öffnen Sie eine neue Windows Forms-Anwendung und erstellen Sie ein Formular mit folgendem Aussehen:



Bei der Gestaltung der Benutzerschnittstelle (*Form1*) haben Sie viel Spielraum, sodass obige Abbildung lediglich als Vorschlag zu verstehen ist. Wir können sogar auf eine Ergebnis-Schaltfläche ("=") verzichten, wie sie bei "normalen" Taschenrechnern üblich ist. Stattdessen werden wir die Berechnung durch einfaches Betätigen der Enter-Taste starten.

Außerdem gönnen wir uns noch drei weitere *TextBox*en, um auch Parameter in die Formel einbauen zu können (es empfiehlt sich, dazu die *KeyPreview*-Eigenschaft des Formulars auf *True* zu setzen).

Die Klasse CCalculator

Fügen Sie zum Projekt eine neue Klasse mit dem Namen *CCalculator* hinzu. Die Klasse stellt einzig und allein die statische Methode *Calc()* bereit, welcher der zu berechnende Ausdruck als String zu übergeben ist. Der Rückgabewert (*Double*) entspricht dem Ergebnis der Berechnung.

Die *Calc()*-Methode erzeugt den Quellcode für ein gültiges VB-Modul mit einer Klasse, die eine ganz einfache Funktion (ebenfalls mit dem Namen *Calc*) zur Berechnung dieses Ausdrucks kapselt. Der Code wird kompiliert und ausgeführt. Um den Code dem VB-Compiler zu übergeben, kommt das CodeDOM (*Code Document Object Model*) zum Einsatz, mit dem sich aus einer Anwendung heraus Programmcode erzeugen lässt. Nach dem Kompilieren wird mittels Reflection auf die erzeugte Assembly zugegriffen und der Ausdruck berechnet.

```
Imports System.CodeDom.Compiler
Imports System.Reflection

Public Class CCalculator
```

Zwischenspeichern der Assembly und der Verweise:

```
    Private Shared ass As Assembly
    Private Shared aClass As Type
    Private Shared aMethode As MethodInfo
    Private Shared obj As Object
```

Der Berechnungsfunktion wird der Formelausdruck als String übergeben:

```
    Public Shared Function Calc(expr As String) As Double
        If expr.Length = 0 Then Return 0.0
```

Im Formelausdruck werden die Dezimalkommas durch Dezimalpunkte ersetzt:

```
        expr = expr.Replace(",", ".")
```

Compilerparameter definieren:

```
        Dim opt As New CompilerParameters(Nothing, String.Empty, False)
        opt.GenerateExecutable = False
        opt.GenerateInMemory = True
```

Den zu kompilierenden VB-Quellcode müssen wir natürlich noch zeilenweise zusammenbauen, mittendrin findet sich unser zu berechnender Ausdruck. Durch die Anweisung *Imports System.Math* können wir mathematische Funktionen wie *Sin* ... auch ohne vorangestellten Namespace schreiben:

```
        Dim src As String = "Imports System.Math" & vbCrLf &
                            "Public Class Calculate" & vbCrLf &
                                "Public Function Calc() As Double" & vbCrLf &
                                    "Return " & expr & vbCrLf &
                                "End Function" & vbCrLf &
                            "End Class" & vbCrLf
```

Nun kann unser VB-Quellcode kompiliert werden:

```
        Dim res As CompilerResults = New VBCodeProvider().CompileAssemblyFromSource(opt, src)
```

Auf eine Fehlerauswertung sollte nicht verzichtet werden:

```
        If res.Errors.Count > 0 Then
            Dim errors As String = String.Empty
            For Each cerr As CompilerError In res.Errors
                errors = errors & cerr.ToString() & vbCrLf
            Next
            ass = Nothing
            expr = String.Empty
            Throw New ApplicationException(errors)
        End If
```

Die vom Compiler erzeugte Assembly kann nun ermittelt und mit dem *Reflection*-Mechanismus ausgewertet werden:

```
        ass = res.CompiledAssembly
```

Die interne Klasse aus der Assembly "herausziehen":

```
        aClass = ass.GetType("Calculate")
```

Jetzt kommen wir auch an die interne *Calc*-Methode heran:

```
        aMethod = aClass.GetMethod("Calc")
```

Eine Instanz der internen Klasse erzeugen, die interne *Calc*-Methode aufrufen und das Ergebnis zurück liefern:

```
        obj = Activator.CreateInstance(aClass)
        Return Convert.ToDouble(aMethod.Invoke(obj, Nothing))
    End Function

End Class
```

Quellcode Form1

```
Public Class Form1
```

Die zentrale Anlaufstelle nach Änderung der Eingabewerte ist der Aufruf dieser Methode:

```
    Private Sub Berechnung()
```

Den Formelausdruck zuweisen:

```
        Dim str As String = TextBox1.Text.ToUpper()
```

Die Parameter X, Y, Z direkt in den Formelausdruck einbauen:

```
        str = str.Replace("X", TextBox2.Text).Replace("Y", TextBox3.Text). _
                                                          Replace("Z", TextBox4.Text)
```

Start der Berechnung (eine Instanziierung der Klasse *CCalculator* kann entfallen, da lediglich ein statischer Methodenaufruf erfolgt). Aufgrund der vielen möglichen Compilerfehler bei Syntaxverstößen wird der entscheidende Methodenaufruf in einer Fehlerbehandlung gekapselt:

```
        Try
            Dim res As Double = CCalculator.Calc(str)
            str = res.ToString()
```

Um das Dezimaltrennzeichen einheitlich als Punkt darzustellen, wandeln wir im Ergebnisstring das Komma einfach in einen Punkt um:

```
            Label1.Text = str.Replace(",", ".")
        Catch ex As Exception
            Label1.Text = String.Empty
            MessageBox.Show(ex.Message)
        End Try
    End Sub
```

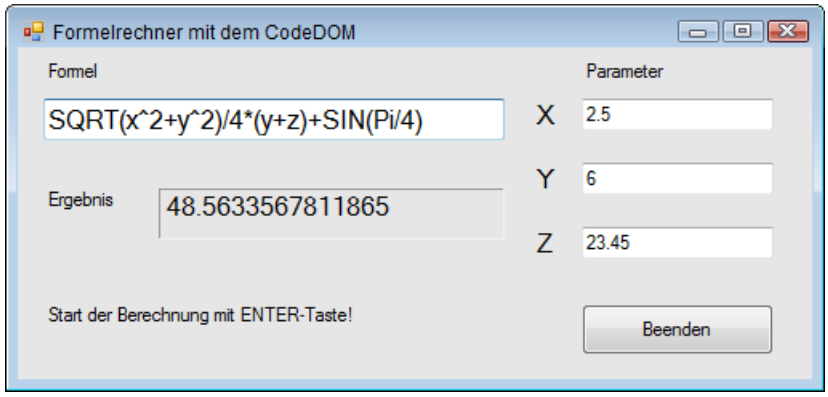
Alle vier *TextBox*en verwenden den folgenden gemeinsamen *KeyPress*-Eventhandler, er sorgt dafür, dass die Berechnung mittels Enter-Taste gestartet wird:

```
    Private Sub TextBox_KeyPress(sender As Object, e As KeyPressEventArgs) _
            Handles TextBox1.KeyPress, TextBox4.KeyPress, TextBox3.KeyPress, TextBox2.KeyPress
        If e.KeyChar = ChrW(Keys.Enter) Then
            Berechnung()
            e.Handled = True
        End If
    End Sub
    ...
End Class
```

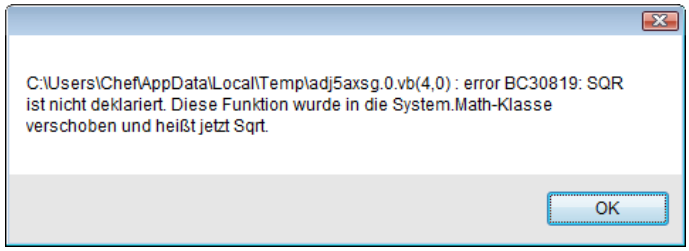
Test

Geben Sie einen beliebig komplizierten bzw. verschachtelten arithmetischen Ausdruck ein (mit oder ohne Parameter *x*, *y*, *z*). Grundlage ist die VB-Syntax, d.h. auch, die Groß-/Kleinschreibung ist ohne Bedeutung.

Starten Sie die Berechnung mit der *Enter*-Taste!



Bei syntaktischen Verstößen erfolgen in der Regel recht ausführliche Fehlermeldungen. Das Beispiel in der folgenden Abbildung zeigt die Meldung, wenn versehentlich der VB-Quadratwurzel-Operator *SQRT* mit *SQR* verwechselt wurde:



Bemerkungen

- Die Klasse *CCalculator* ist ausbaufähig, denn sie kann nicht nur einen einzigen Ausdruck, sondern auch einen kompletten Algorithmus berechnen, in welchem weitere Funktionen aufgerufen werden können. In diesem Fall empfiehlt sich ein mehrzeiliges Textfeld, in das der VB-Code einzugeben ist. Vorher ist auf das Vorhandensein der *Return*-Anweisung zu prüfen, sodass diese nur im Bedarfsfall (wie in unserem Beispiel) per Programmcode hinzugefügt werden muss.
- Wollen Sie für den zu berechnenden Ausdruck nicht die VB-, sondern die C#-Syntax verwenden, muss natürlich ein anderer Quellcode "zusammengebaut" werden und Sie müssen den entsprechende C#-Codeprovider instanziieren. Bei der Eingabe der Berechnungsformel wäre dann penibel auf die Groß-/Kleinschreibung zu achten.

3.12.9 Einen Funktionsverlauf grafisch darstellen

In der Basisversion des Formelrechners wurde gezeigt, wie man mittels CodeDOM beliebig komplizierte mathematische Formeln auswerten kann. Will man allerdings eine grafische Auswertung (Funktionsdiagramm) vornehmen, so sind in der Regel einige hundert aufeinanderfolgende Aufrufe der Formel bei schrittweise sich ändernden Parametern erforderlich. Damit stößt das Verfahren an seine Grenzen, denn das wiederholte Kompilieren und Auswerten würde auch bei einem schnellen Rechner unzumutbar viel Zeit verbrauchen.

Da es nicht sinnvoll ist, bei geänderten Parametern aber gleichem Formelausdruck die Assembly immer wieder erneut zu kompilieren, speichern wir die entsprechenden Verweise auf die einmal erzeugte Assembly intern (im Arbeitsspeicher) ab und greifen bei Bedarf darauf zu.

Wir wollen die Vorgehensweise anhand einer bescheidenen Grafik für die bekannte Spaltfunktion *sin(x)/x* erläutern.

Oberfläche

Das nackte Startformular (*Form1*), ausgestattet mit einer einzigen *TextBox*, genügt! Weisen Sie der *TextBox* die *Text*-Eigenschaft "*Sin(x) / x*" zu (Leerzeichen und Groß-/Kleinschreibung spielen dabei keine Rolle).

Die Klasse CCalculatorX

Die Änderungen gegenüber der Klasse *CCalculator* sind durch Fettdruck hervorgehoben:

```
Imports System.CodeDom.Compiler
Imports System.Reflection

Public Class CCalculatorX
    Private Shared ass As Assembly
    Private Shared aClass As Type
    Private Shared aMethod As MethodInfo
    Private Shared obj As Object
```

```
    Private Shared expression As String

    Public Shared Function Calc(expr As String, x As Double) As Double
        If expr.Length = 0 Then Return 0.0
        expr = expr.Replace(",", ".")
```

Ein erneutes Kompilieren ist nur dann erforderlich, wenn sich der Formelausdruck *expr* geändert hat:

```
        If expr <> expression Then
            expression = expr
            Dim opt As New CompilerParameters(Nothing, String.Empty, False)
            opt.GenerateExecutable = False
            opt.GenerateInMemory = True

            Dim src As String = "Imports System.Math" & vbCrLf &
            "Public Class Calculate" & vbCrLf &
                "Public Function Calc(X As Double) As Double" & vbCrLf &
                    "Return " & expr & vbCrLf &
                "End Function" & vbCrLf &
            "End Class" & vbCrLf

            Dim res As CompilerResults = New VBCodeProvider().CompileAssemblyFromSource(
                                                                    opt, src)
            If res.Errors.Count > 0 Then
                Dim errors As String = String.Empty
                For Each cerr As CompilerError In res.Errors
                    errors = errors & cerr.ToString() & vbCrLf
                Next
                ass = Nothing
                expr = String.Empty
                Throw New ApplicationException(errors)
            End If
            ass = res.CompiledAssembly
            aClass = ass.GetType("Calculate")
            aMethod = aClass.GetMethod("Calc")
            obj = Activator.CreateInstance(aClass)
        End If
```

Falls sich *expr* nicht geändert hat, genügt die Übergabe des neuen Wertes für den Parameter *x*:

```
        Return Convert.ToDouble(aMethod.Invoke(obj, New Object() {x}))
    End Function

End Class
```

Quellcode Form1

```
Public Class Form1
```

Wir überschreiben die *OnPaint*-Methode des Formulars, dadurch wird auch nach vorübergehendem Verdecken des Fensters das Diagramm automatisch neu erstellt:

```
    Protected Overrides Sub OnPaint(e As System.Windows.Forms.PaintEventArgs)
```

Um den Quellcode überschaubar zu halten, haben wir die Einstellungen des Koordinatensystems für das Diagramm der Funktion *y = sin(x) / x* optimiert:

```
        Dim x As Single = -40
        Dim y As Single = 0
        Dim xold As Single = x
        Dim yold As Single = y
        Dim scalex As Single = Me.Width / (2 * x)
        Dim scaley As Single = -(Me.Height - 20) / 2
        Dim incr As Single = 80.0F / Me.Width
        Dim p As New Pen(Color.Black, 2)
```

Graphics-Objekts, auf welches das Diagramm gezeichnet werden kann:

```
        Dim g As Graphics = e.Graphics
```

Koordinatensystem verschieben:

```
        g.TranslateTransform(Me.Width / 2, Me.Height / 2)
```

Koordinatenachsen zeichnen:

```
        g.DrawLine(Pens.Red, -Me.Width, 0, Me.Width, 0)
        g.DrawLine(Pens.Red, 0, -Me.Height, 0, Me.Height)
```

Anzeige optimieren:

```
        g.SmoothingMode = System.Drawing.Drawing2D.SmoothingMode.AntiAlias
```

Die einzelnen Funktionswerte berechnen und zeichnen:

```
        For i As Integer = 0 To Me.Width - 1
            If x = 0 Then x += 0.000001F
            y = Convert.ToSingle(CCalculatorX.Calc(TextBox1.Text, x))
            x += incr
            g.DrawLine(p, x * scalex, y * scaley, xold * scalex, yold * scaley)
            xold = x
            yold = y
        Next
```

Die folgende Anweisung nicht vergessen, sonst wird *Paint*-Event nicht ausgelöst:

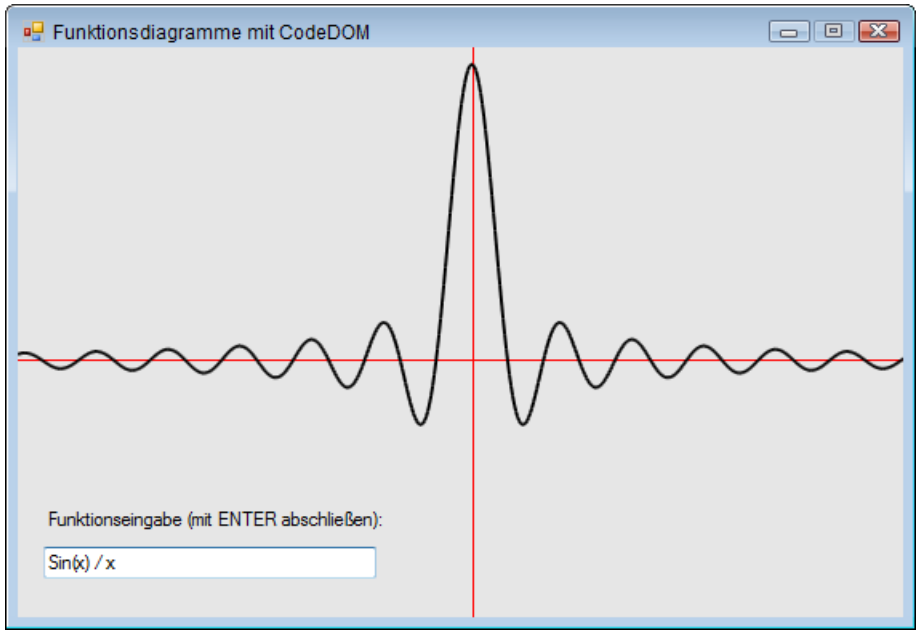
```
        MyBase.OnPaint(e)
    End Sub
```

Der folgende Eventhandler sorgt für die Umwandlung eines Dezimalkommas in einen Dezimalpunkt und für das Neuzeichnen des Formulars nach Betätigen der Enter-Taste:

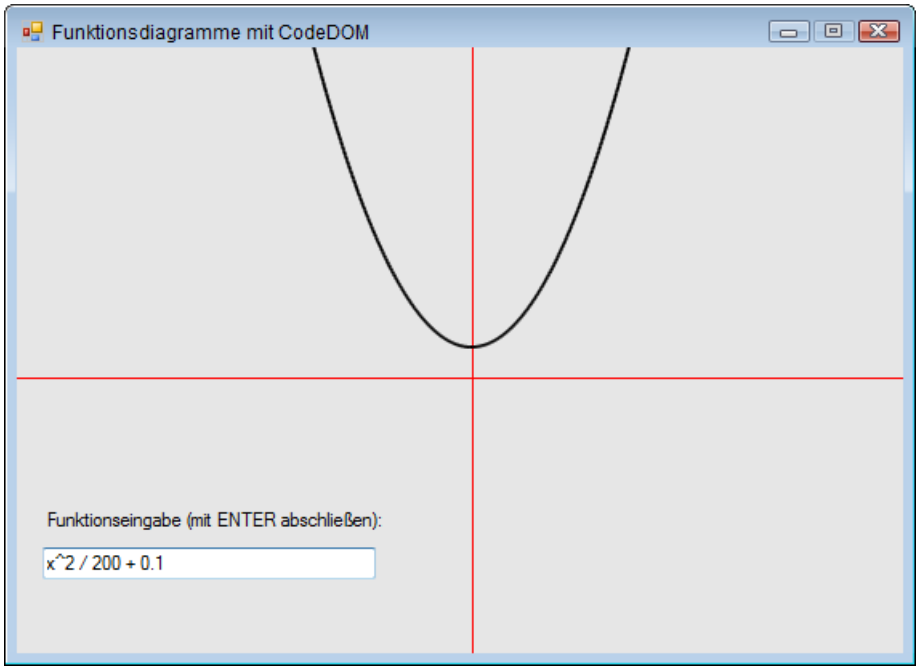
```
    Private Sub TextBox1_KeyPress(sender As Object, e As KeyPressEventArgs) _
                                                        Handles TextBox1.KeyPress
        If e.KeyChar = "," Then
            e.KeyChar = "."
        Else
            If e.KeyChar = ChrW(Keys.Enter) Then
                Me.Refresh()
                e.Handled = True
            End If
        End If
    End Sub
End Class
```

Test

Gleich nach Programmstart erscheint das Diagramm der Funktion *sin(x)/x*. Die Grafik wird auch nach vorübergehendem Verdecken des Fensters in Windeseile wieder aufgebaut, da nur einmal kompiliert werden muss.



Ändern Sie den Formelausdruck und schließen Sie mit der *Enter*-Taste ab. Unter der Voraussetzung, dass das starre Koordinatensystem eine geeignete Darstellung zulässt, lassen sich interessante Experimente durchführen:



Bemerkungen

- Das Programm lässt sich durch Hinzufügen weiterer Bedienelemente für Inkremente, Maßstabsfaktoren, Skalenteilung etc. so verfeinern, dass eine optimale Darstellung nahezu beliebiger Funktionen ermöglicht wird.
- Die spezialisierte Klasse *CCalculatorX* hat gegenüber der Klasse *CCalculator* den Vorteil, dass sie wiederholte Berechnungen mit einem geänderten Parameter *x* gestattet, ohne dass dazu die Assembly erneut erzeugt werden müsste. Das bedeutet einen erheblichen Performancegewinn.
- Der Klasse *CCalculator* bleibt hingegen der Vorteil der universellen Verwendbarkeit für beliebige Ausdrücke mit beliebig vielen Parametern, sie eignet sich deshalb besonders für einmalig auszuführende komplizierte Berechnungen.

3.12.10 Sortieren mit IComparable/IComparer

Haben Sie eine *ArrayList* oder eine generische *List(Of T)* von Typen, wie Strings oder Integers, die bereits *IComparer* unterstützen, so können Sie dieses Array oder die Liste ohne irgendeine explizite Referenz auf *IComparer* sortieren. In diesem Fall werden die Elemente des Arrays automatisch in die standardmäßige Implementierung von *IComparer* gecastet, eine *Sort*-Methode ist also bereits "eingebaut".

Haben Sie aber nutzerdefinierte Objekte, so müssen Sie selbst entweder eines oder beide der Interfaces *IComparable* oder *IComparer* implementieren.

Oberfläche

Öffnen Sie eine neue WPF-Anwendung und platzieren Sie eine *ListBox* (*Name=lb*) und zwei Schaltflächen im Formular:

```
<Grid Margin="5">
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*"/>
        <ColumnDefinition Width="*"/>
    </Grid.ColumnDefinitions>
    <ListBox Name="lb"  Grid.Column="0" ItemsSource="{Binding}"/>
    <StackPanel Grid.Column="1">
        <Button Content="IComparable" Click="Button_Click_1" />
        <Button Content="IComparer"  Click="Button_Click_2" />
    </StackPanel>
</Grid>
```

Klasse CStudent

Fügen Sie zum Projekt eine neue Klasse *CStudent* hinzu, die das *IComparable*-Interface implementiert:

```
Public Class CStudent
 Implements IComparable(Of CStudent)

 Public Property Nummer As Integer
 Public Property Vorname As String
 Public Property Nachname As String

 Public Sub New(numm As Integer, vor As String, nach As String)
     Nummer = numm
     Vorname = vor
     Nachname = nach
 End Sub
```

Die alphabetische Sortierung nach dem Nachnamen wird als Standardvergleich festgelegt:

```
    Public Function CompareTo(stud As CStudent) As Integer _
                                Implements IComparable(Of CStudent).CompareTo
        Return Me.Nachname.CompareTo(stud.Nachname)
    End Function
```

Um die Datenbindung zu vereinfachen überschreiben wir die *ToString*-Methode:

```
    Public Overrides Function ToString() As String
        Return Nummer.ToString() & " ,  " & Nachname & " , " & Vorname
    End Function
End Class
```

MainWindow.xaml.vb

Im Hauptfenster implementieren wir zunächst eine Methode, die eine Studentenliste erzeugt und initialisiert:

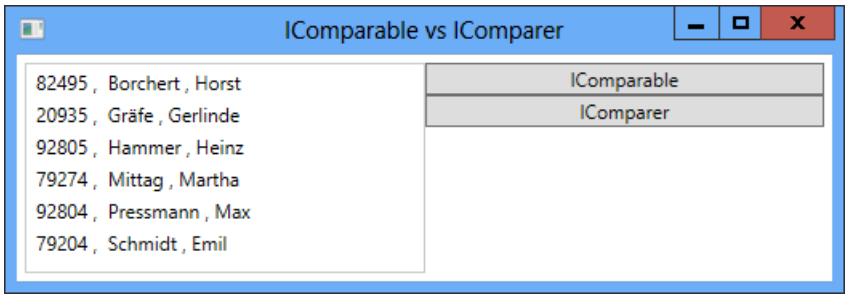
```
Class MainWindow

    Private Function getStudenten() As List(Of CStudent)
        Dim studenten As New List(Of CStudent)()
        studenten.Add(New CStudent(82495, "Horst", "Borchert"))
        studenten.Add(New CStudent(20935, "Gerlinde", "Gräfe"))
        studenten.Add(New CStudent(79274, "Martha", "Mittag"))
        studenten.Add(New CStudent(79204, "Emil", "Schmidt"))
        studenten.Add(New CStudent(92804, "Max", "Pressmann"))
        studenten.Add(New CStudent(92805, "Heinz", "Hammer"))
        Return studenten
    End Function
```

Im *Click*-Event des ersten *Button*s wird die Liste (entsprechend der Implementierung von *CompareTo* in *CStudent*) sortiert und angezeigt:

```
     Private Sub Button_Click_1(sender As Object, e As RoutedEventArgs)
        Dim studenten = getStudenten()
        studenten.Sort()
        lb.DataContext = studenten
     End Sub
...
```

Test von IComparable

Bei Klick auf die Schaltfläche *IComparable* erscheint die Liste alphabetisch nach den Nachnamen sortiert:



Bemerkungen zu IComparable

- Wenn eine Klasse das *IComparable* Interface implementiert, müssen wir auch die Methode *CompareTo(T)* implementieren. In dieser Methode können wir unseren Sortieralgorithmus definieren (Standardvergleich). In unserem Beispiel haben wir die Liste in alphabetischer Reihenfolge sortiert.

- Wir verwenden *IComparable(Of T)*, wenn die Klasse über einen Standardvergleich verfügen soll. Das Sortierkriterium muss also bereits bekannt sein, bevor wir mit der Implementierung der Klasse beginnen. In unserem Beispiel mussten wir deshalb vorher entscheiden, dass wir nach dem Nachnamen und nicht nach der Studentennummer sortieren wollen. Es gibt aber Situationen, wo wir nicht nur den Standardvergleich, sondern mehrere Sortierkriterien benötigen.
- Um letztgenanntes Problem zu lösen, stellt .NET ein spezielles Interface *IComparer(Of T)* bereit, welches über eine Methode *Compare()* verfügt, die zwei Objektparameter X, Y entgegennimmt und ein Integer zurückgibt.

Klasse CSortNummer

Fügen Sie zum Projekt eine Klasse *CSortNummer* hinzu, welche das *IComparer*-Interface implementiert:

```
Public Class CSortNummer
    Implements IComparer(Of CStudent)
```

Eine Zustandsvariable bestimmt die Sortierung in aufsteigender (*True*) bzw. absteigender (*False*) Folge:

```
    Private _auf As Boolean
```

Wir übergeben im Konstruktor einen Parameter, der die gewünschte Sortierfolge spezifiziert.

```
    Public Sub New(auf As Boolean)
      _auf = auf
    End Sub
```

Die Implementierung des Sortierkriteriums erfolgt in der *Compare*-Methode:

```
  Public Function Compare(a As CStudent, b As CStudent) As Integer _
                                          Implements IComparer(Of CStudent).Compare
    If _auf Then         ' aufsteigend
        If (a.Nummer > b.Nummer) Then
            Return 1
        ElseIf a.Nummer < b.Nummer Then
            Return -1
        Else
            Return 0
        End If
    Else                 ' absteigend
        If a.Nummer < b.Nummer Then
            Return 1
        ElseIf a.Nummer > b.Nummer Then
            Return -1
        Else
            Return 0
        End If
```

```
        End If
    End Function
End Class
```

Ergänzung MainWindow.xaml.vb

Wir ergänzen den Codebehind, um die zweite Sortiervariante zu testen:

```
    ...
    Private Sub Button_Click_2(sender As Object, e As RoutedEventArgs)
        Dim studenten = getStudenten()
```

Wir wollen die Sortierfolge umkehren:

```
        Dim son As New CSortNummer(False)
```

Die Liste wird nun nach fallenden Nummern sortiert:

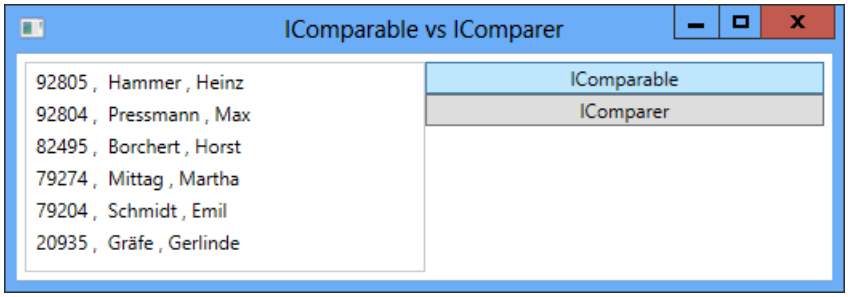
```
        studenten.Sort(son)
```

Noch eine Bemerkung zu obiger Zeile: Die IntelliSense wird Ihnen zwei Überladungen der *Sort*-Methode anbieten, eine davon ist die bereits implementierte Standardsortierung (*IComparable*), die andere die in *CSortNummer* definierte zusätzliche Sortierung, die wir hier ausgewählt haben.

```
        lb.DataContext = studenten
    End Sub
    ...
```

Test von IComparer

Die Liste ist jetzt nach fallenden Nummern sortiert:



Bemerkungen zu IComparer

- Wir verwenden *IComparer* dann, wenn wir ein anderes Sortierkriterium als den von der Klasse bereitgestellten Standardvergleich benötigen (oder mehrere Vergleichskriterien).
- Pro zusätzliches Sortierkriterium ist eine Klasse erforderlich, die das *IComparer*-Interface (*Compare*-Methode) implementiert.
- *IComparer* funktioniert auch dann, wenn keine Standardsortierung existiert, also wenn *IComparable* nicht implementiert ist.

3.12.11 Objektbäume in generischen Listen abspeichern

Die generische *List*-Klasse hat gegenüber der altbackenen *ArrayList* gravierende Vorteile, so entfallen vor allem die umständlichen Typkonvertierungen. Nicht nur einfache Objekte, sondern auch komplette Objektbäume können mittels *BinaryFormatter* bequem serialisiert bzw. deserialisiert werden, um die Daten dauerhaft zu sichern.

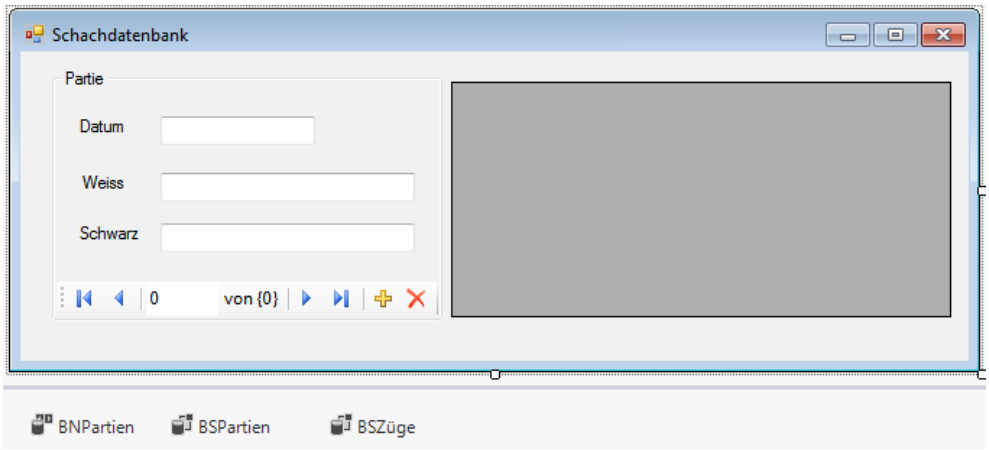
Wir demonstrieren dies am Beispiel einer ausbaufähigen Schachdatenbank, in der Sie beliebig viele Schachpartien abspeichern können[1].

HINWEIS: Das vorliegende Beispiel verwendet keine Datenbank, sondern eine einfache Binärdatei als Speichermedium!

Oberfläche

Starten Sie Visual Studio und öffnen Sie eine neue Windows Forms-Anwendung. Auf dem Startformular *Form1* findet links eine *GroupBox* ihren Platz. In diese setzen Sie drei *TextBox*-Steuerelemente und einen *BindingNavigator*, der am unteren Rand angedockt wird (*Dock* = *Bottom*) und den Sie in *BNPartien* umbenennen.

Rechts platzieren Sie eine *DataGridView*-Komponente. Fügen Sie noch zwei *BindingSource*-Komponenten hinzu, die Sie in *BSPartien* und *BSZüge* umbenennen. Verbinden Sie nun die *BindingSource*-Eigenschaft von *BNPartien* mit *BSPartien*.



Über das Menü *Projekt/Klasse hinzufügen...* erweitern Sie das Projekt um die Klassen *CZug*, *CPartie* und *CSchachDB*, die beide am Objektbaum beteiligt sind. Alle werden mit dem *Serializable*-Attribut ausgestattet, da wir den Objektbaum nicht nur im Arbeitsspeicher ablegen, sondern auch auf der Festplatte sichern wollen.

[1] Um den Sinn dieses Beispiels zu verstehen, muss man kein Schachspieler sein.

Die Klasse CZug

Diese Klasse stellt die Blattebene unseres Objektbaums dar und repräsentiert einen Doppelzug nebst Kommentar (einfachheitshalber sind alle Eigenschaften selbst implementierend):

```
<Serializable> Public Class CZug
    Public Property Weiss As String       ' z.B. "e4" oder Le7, Se6, 0-0, Lb4+, c:d5, S:a4, ...
    Public Property Schwarz As String
    Public Property Kommentar As String   ' z.B. "schwerer Fehler!"
End Class
```

Die Klasse CPartie

```
<Serializable> Public Class CPartie
```

Eine generische Liste speichert alle Züge einer Partie:

```
    Private _züge As List(Of CZug)
```

Im Konstruktor wird die (zunächst leere) Zugliste erstellt:

```
    Sub New()
        _züge = New List(Of CZug)
    End Sub
```

Der Zugriff auf die Zugliste:

```
    Public Property Züge As IList(Of CZug)
        Get
            Return _züge
        End Get
        Set(value As IList(Of CZug))
            _züge = value
        End Set
    End Property
```

Einige selbst implementierende Eigenschaften:

```
   Public Property Datum As DateTime             ' Datum der Partie
   Public Property Weiss As String               ' Name des Weiss-Spielers
   Public Property Schwarz As String             ' Name des Schwarz-Spielers
End Class
```

Die Klasse CSchachDB

Dies ist die Wurzelklasse unseres Objektbaums.

```
<Serializable> Public Class CSchachDB
```

Die folgende (generische) Liste speichert alle Partien der Datenbank:

```
   Private _partien As IList(Of CPartie)
```

Im Konstruktor wird die (zunächst leere) Partienliste erzeugt:

```
Sub New()
     _partien = New List(Of CPartie)
End Sub
```

Der Zugriff auf die Liste der Partien:

```
Public Property Partien() As IList(Of CPartie)    ' Liste der Partien
    Get
        Return _partien
    End Get
    Set(value As IList(Of CPartie))
        _partien = value
    End Set
End Property

End Class
```

Die Klasse CPersistenz

Damit nach dem Ausschalten des Rechners nicht alle Daten auf Nimmerwiedersehen verschwunden sind, stellt diese Klasse die Methoden *saveObject* und *loadObject* bereit, die sich um die Datenpersistenz kümmern.

```
Imports System.IO
Imports System.Runtime.Serialization.Formatters.Binary

Public Class CPersistenz
```

Aufgabe dieser Methode ist es, den kompletten Objektbaum zu serialisieren und als Datei abzuspeichern:

```
    Public Shared Sub saveObject(o As Object, pfad As String)
        Dim fs As New FileStream(pfad, FileMode.Create, FileAccess.Write, FileShare.None)
        Dim bf As New BinaryFormatter()
        bf.Serialize(fs, o)
        fs.Close()
    End Sub
```

Die folgende Methode lädt die Datei zurück in den Arbeitsspeicher und rekonstruiert den Objektbaum:

```
   Public Shared Function loadObject(pfad As String) As Object
        Dim fs As New FileStream(pfad, FileMode.Open, FileAccess.Read, FileShare.Read)
        Dim bf As New BinaryFormatter()
        Dim o As Object = bf.Deserialize(fs)
        fs.Close()
        Return o
    End Function
End Class
```

Die Klasse Form1

```
Public Class Form1
```

Der komplette Objektbaum:

```
    Private _schachDB As New CSchachDB()
```

Der Standort der Datenbankdatei:

```
    Private Const PFAD As String = "SchachDB.dat"
```

Der Programmstart:

```
    Private Sub Form1_Load(sender As Object, e As EventArgs) Handles MyBase.Load
```

Zunächst wird versucht, die Datenbankdatei zu laden, im Fehlerfall entsteht eine neue leere Datei:

```
        Try
            _schachDB = CType(CPersistenz.loadObject(PFAD), CSchachDB)
        Catch ex As Exception
            MessageBox.Show(ex.Message)
        End Try
```

Verbinden der *Partien*-Liste mit der entsprechenden *BindingSource*-Komponente:

```
        BSPartien.DataSource = _schachDB.Partien
```

Anbinden der Steuerelemente:

```
        TextBox1.DataBindings.Add("Text", BSPartien, "Datum", True)
        TextBox2.DataBindings.Add("Text", BSPartien, "Weiss", True)
        TextBox3.DataBindings.Add("Text", BSPartien, "Schwarz", True)
```

Anbinden des Datengitters an die Liste der Züge:

```
        DataGridView1.DataSource = BSZüge
    End Sub
```

Erzeugen Sie einen Eventhandler für das *CurrentChanged*-Ereignis der *BindingSource* der Partienliste:

```
    Private Sub BSPartien_CurrentChanged(sender As Object, e As EventArgs) _
                                        Handles BSPartien.CurrentChanged
```

Beim Navigieren zu einer anderen Partie wird die *BindingSource* der Zügeliste umgeklemmt:

```
        Dim partie As CPartie = CType(BSPartien.Current, CPartie)
        BSZüge.DataSource = partie.Züge
    End Sub
```

Beim Schließen des Formulars versuchen wir, den Objektbaum in der Datei abzuspeichern:

```
    Private Sub Form1_FormClosing(sender As Object, e As FormClosingEventArgs) _
                                                Handles Me.FormClosing
        Try
            CPersistenz.saveObject(_schachDB, PFAD)
        Catch ex As Exception
```

```
            MessageBox.Show(ex.Message)
        End Try
    End Sub
```

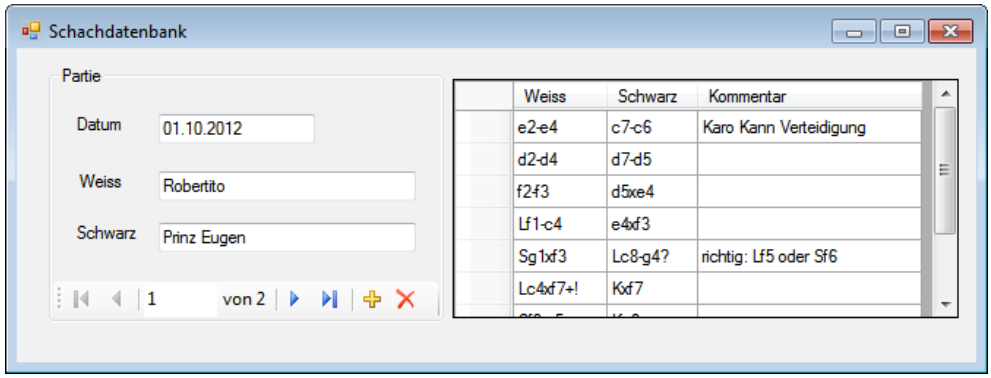
Damit Sie nicht leichtfertig eine komplette Schachpartie mitsamt allen mühsam eingegebenen Zügen löschen, soll nach Klick auf die *Delete*-Schaltfläche des *BindingNavigator*s zunächst eine Sicherheitswarnung erscheinen:

```
    Private Sub BindingNavigatorDeleteItem2_Click(sender As Object, e As EventArgs) _
                                                Handles BindingNavigatorDeleteItem2.Click
        If MsgBox("Soll die Partie wirklich gelöscht werden?", MsgBoxStyle.YesNo,
                                              "Sicherheitsabfrage") = MsgBoxResult.Yes Then
            BSPartien.RemoveCurrent()
        End If
    End Sub

End Class
```

HINWEIS: Obige Methode funktioniert nur dann, wenn Sie die *DeleteItem*-Eigenschaft des *BindingNavigator*s auf *None* (keine) gesetzt haben.

Test

Lassen Sie sich nicht davon irritieren, dass nach Programmstart zunächst ein Meldungsfenster erscheint, welches Sie auf die noch nicht vorhandene Datei hinweist.



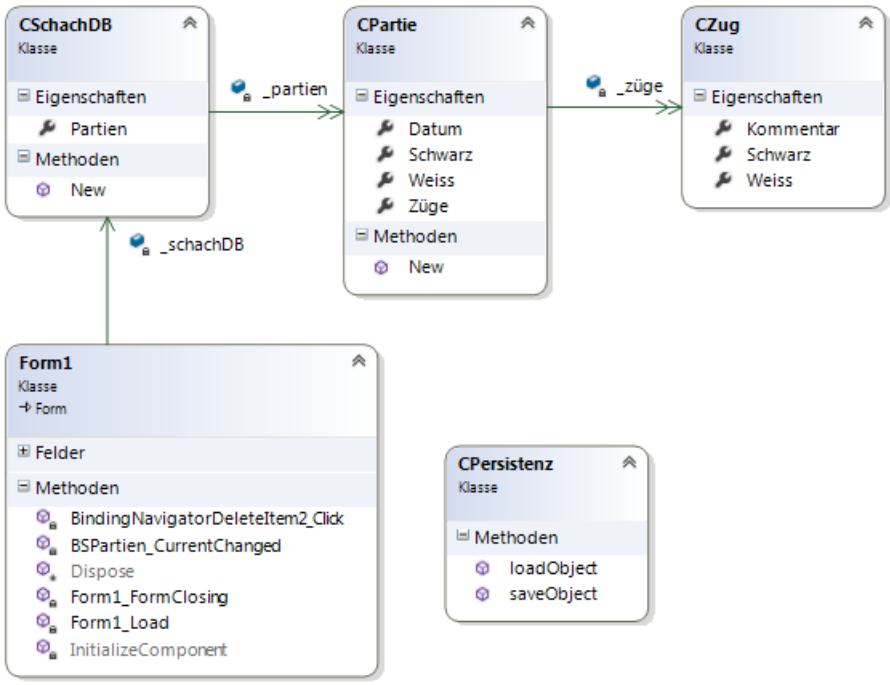
Klicken Sie zunächst auf die "+"-Schaltfläche im *BindingNavigator*, um eine neue Schachpartie mit dem Datum[1] und den Namen der Spieler anzulegen. Danach können Sie auf bekannte Weise im Datengitter nacheinander die einzelnen Züge eintippen.

[1] Haben Sie ein ungültiges Datum eingegeben, so lässt sich das Formular erst dann wieder schließen, wenn Sie diesen Fehler korrigiert haben (generische Listen sind typsicher!).

Nach Schließen des Formulars werden alle Partien automatisch in der im Anwendungsverzeichnis abgelegten Datei *Schachpartien.dat* gespeichert und stehen nach erneutem Programmstart wieder zur Verfügung.

Bemerkungen

- Für den Schachinteressenten ergeben sich mannigfaltige Erweiterungsmöglichkeiten, wie zum Beispiel die Suche nach Partien mit einem bestimmten Gegner oder nach gleichen Eröffnungszügen. Ziemlich aufwändig, aber auch sehr praktisch ist eine grafische Brettdarstellung, mittels welcher man die Züge per Drag&Drop eingeben kann.
- Einen guten Überblick über die Programmstruktur und die Beziehungen zwischen den Klassen liefert das **Klassendiagramm**, welches man sich im Projektmappen-Explorer generieren lassen kann (Kontextmenü *Klassendiagramm anzeigen*). Die Auflistungszuordnungen (Assoziationen) der Felder *_partien* und *_züge* sind an den doppelten Pfeilspitzen erkennbar. Man erhält sie, wenn man zunächst auf das entsprechende Feld in der Klasse klickt und dann im Kontextmenü *Als Auflistungszuordnung anzeigen* wählt. Die folgende Abbildung zeigt nur eine von vielen möglichen Versionen des Klassendiagramms.



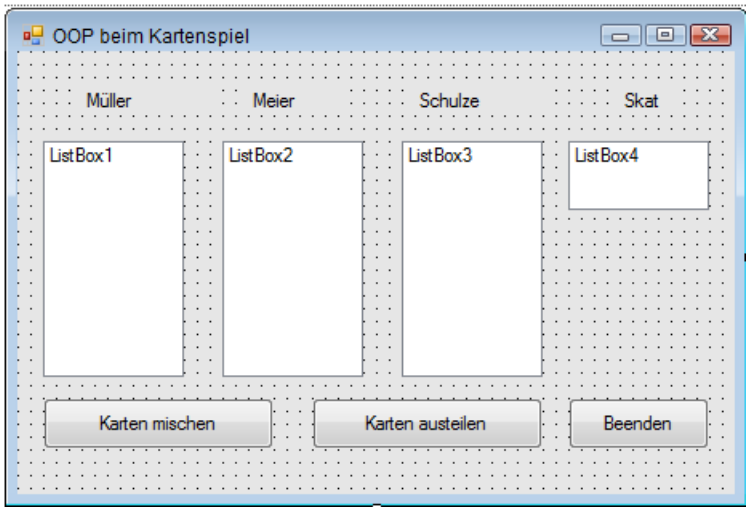
3.12.12 OOP beim Kartenspiel erlernen

Das vorliegende Beispiel soll Ihnen helfen, Ihre OOP-Grundlagen "spielend" weiter auszubauen. Es wird gezeigt, wie die in einem Array abgespeicherten *Karten*-Objekte durch einen Konstruktor erzeugt werden und selbst wiederum nach außen als Eigenschaften einer Klasse *Spiel* in Erscheinung treten.

HINWEIS: Zum Verständnis dieses Praxisbeispiels ist die Kenntnis der Skatregeln keinesfalls Voraussetzung. Es reicht aus zu wissen, dass das Spiel aus 32 Karten besteht und jeder der drei Spieler zu Beginn 10 Karten erhält und die restlichen zwei davon im so genannten "Skat" verbleiben.

Oberfläche

Die folgende Abbildung bedarf wohl keines weiteren Kommentars.



Quellcode

Es dient der Übersicht, wenn man für jede Klasse ein eigenes Klassenmodul verwendet. Diesmal aber wollen wir zeigen, dass man auch Klassen ineinander verschachteln kann. In diesem Sinne implementieren wir innerhalb des Klassencodes von *Form1* die Klassen *CKarte* und *CSpiel*:

```
Public Class Form1
    Public Class CKarte
        Public Farbe As String
        Public Wert As String
```

Der Konstruktor erzeugt ein bestimmtes Kartenobjekt, welches durch Farbe und Wert charakterisiert ist:

```
        Public Sub New(f As String, w As String)
```

```
            Farbe = f
            Wert = w
        End Sub
    End Class                       ' von CKarte
```

Die Klasse *CSpiel* kapselt 32 Instanzen der Klasse *CKarte*:

```
    Public Class CSpiel
```

Die Eigenschaft *Karten* (das ist ein Array mit 32 Karten!):

```
        Public Karten(32) As CKarte
```

Eine Hilfsmethode soll das Generieren der Karten vereinfachen:

```
        Private Sub createKarten(farbe As String, a As Integer)
            Karten(a) = New CKarte(farbe, "Sieben")
            Karten(a + 1) = New CKarte(farbe, "Acht")
            Karten(a + 2) = New CKarte(farbe, "Neun")
            Karten(a + 3) = New CKarte(farbe, "Zehn")
            Karten(a + 4) = New CKarte(farbe, "Bube")
            Karten(a + 5) = New CKarte(farbe, "Dame")
            Karten(a + 6) = New CKarte(farbe, "König")
            Karten(a + 7) = New CKarte(farbe, "As")
        End Sub
```

Der Konstruktor erzeugt und füllt das Spiel der Reihe nach mit allen 32 Karten:

```
        Public Sub New()
            Call createKarten("Eichel", 1)
            Call createKarten("Grün", 9)
            Call createKarten("Rot", 17)
            Call createKarten("Schell", 25)
        End Sub
```

Die Methode zum Mischen der Karten:

```
        Public Sub mischen()
            Dim z, i As Integer
```

Der Zwischenspeicher für den Kartentausch:

```
            Dim tmp As CKarte
            Dim rnd As New Random()
```

Alle Karten nacheinander durchlaufen:

```
            For i = 1 To Karten.Length - 1
```

Den Index einer zufälligen anderen Karte bestimmen:

```
                z = rnd.Next(1, Karten.Length)
```

Die zufällige Karte mit der aktueller Karte vertauschen:

```
                tmp = Karten(z)
                Karten(z) = Karten(i)
```

```
            Karten(i) = tmp
        Next
    End Sub
End Class                ' von CSpiel
```

HINWEIS: Aus Gründen der Einfachheit bleibt der Index 0 des *Karten*-Arrays ungenutzt.

Nun kommen wir zum eigentlichen Klassencode von *Form1*:

Ein neues Kartenspiel wird erzeugt:

```
Dim spiel As New CSpiel()
```

Eine Hilfsroutine zum Löschen der Anzeige:

```
Private Sub loeschen()
    ListBox1.Items.Clear(): ListBox2.Items.Clear()
    ListBox3.Items.Clear(): ListBox4.Items.Clear()
End Sub
```

Alle Karten mischen:

```
Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
    loeschen()
    spiel.mischen()
End Sub
```

Alle Karten austeilen:

```
Private Sub Button2_Click(sender As Object, e As EventArgs) Handles Button2.Click
   loeschen()
```

Die Karten für Müller:

```
    For i As Integer = 1 To 10
        ListBox1.Items.Add(spiel.Karten(i).Farbe & "  " & spiel.Karten(i).Wert)
    Next
```

Die Karten für Meier:

```
    For i As Integer = 11 To 20
        ListBox2.Items.Add(spiel.Karten(i).Farbe & "  " & spiel.Karten(i).Wert)
    Next
```

Die Karten für Schulze:

```
    For i As Integer = 21 To 30
        ListBox3.Items.Add(spiel.Karten(i).Farbe & "  " & spiel.Karten(i).Wert)
    Next
```

Die restlichen zwei Karten wandern in den Skat:

```
    For i As Integer = 31 To 32
        ListBox4.Items.Add(spiel.Karten(i).Farbe & "  " & spiel.Karten(i).Wert)
    Next
```

```
    End Sub
    ...
End Class                       ' Form1
```

Test

Wenn Sie, vor lauter Ungeduld, unmittelbar nach Programmstart auf die Schaltfläche *Karten austeilen* klicken, werden Sie von allen drei Spielern laute Protestrufe ernten, da die Karten offensichtlich noch nicht gemischt wurden:

Erst nach ein- oder mehrmaligem Klick auf die Schaltfläche *Karten mischen* hat die Gerechtigkeit ihren Einzug gehalten und der Zufall bestimmt, welche Karten die Spieler Müller, Meier und Schulze erhalten:

Index

B

C

D

E

F

G

H

I

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z