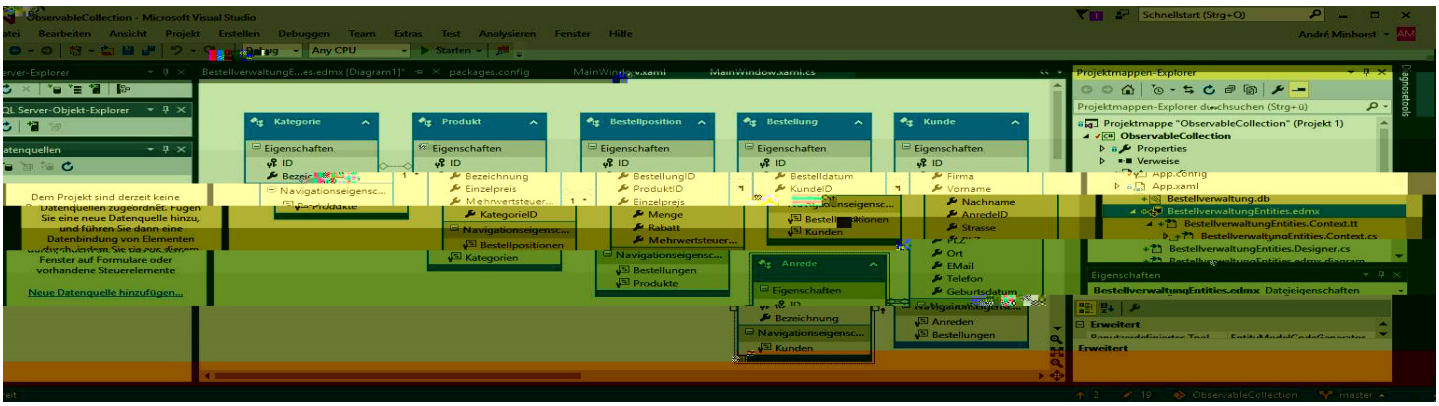
AUSGABE 01/2022

DATENBANK ENTWICKLER

MAGAZIN FÜR DIE DATENBANKENTWICKLUNG MIT VISUAL STUDIO FÜR DESKTOP, WEB UND CO.



TOP-THEMEN:

André Minhorst Verlag

Bitmaps programmieren mit VB.NET

Wer von Access/VBA kommt, kennt das Problem: Alles, was mit der Erstellung und Bearbeitung von Grafiken zu tun hat, ist sehr kompliziert und erfordert in der Regel die Verwendung von API-Funktionen. Unter VB.NET ist dies sehr viel einfacher. Daher bietet es sich nicht nur an, die Grafikfähigkeiten des Namespaces System.Drawing für .NET-Anwendungen zu nutzen. Sie können die gewünschten Funktionen beispielsweise auch in DLLs integrieren, die Sie dann von Access aus einbinden und aufrufen können. Im vorliegenden Artikel wollen wir uns jedoch zunächst einmal die grundlegenden Möglichkeiten zum Erstellen und Bearbeiten von Bilddateien unter .NET ansehen.

Vorbereitung: LINQPad

LINQPad ist unsere Spielwiese, wenn es um das Ausprobieren von VB.NET-Code geht. Hier brauchen wir zum Testen einiger Zeilen Code nicht extra ein VB-Projekt anzulegen und dieses jedes Mal neu zu starten, um die Funktionsweise zu prüfen. Wir geben die gewünschten Zeilen einfach in eine Methode ein und starten diese mit **F5** – ganz wie es beispielsweise im VBA-Editor möglich ist. Wie Sie LINQPad nutzen, lesen Sie im Artikel **LINQPad: LINQ, C# und VB einfach ausprobieren** (**www.datenbankentwickler.net/100**).

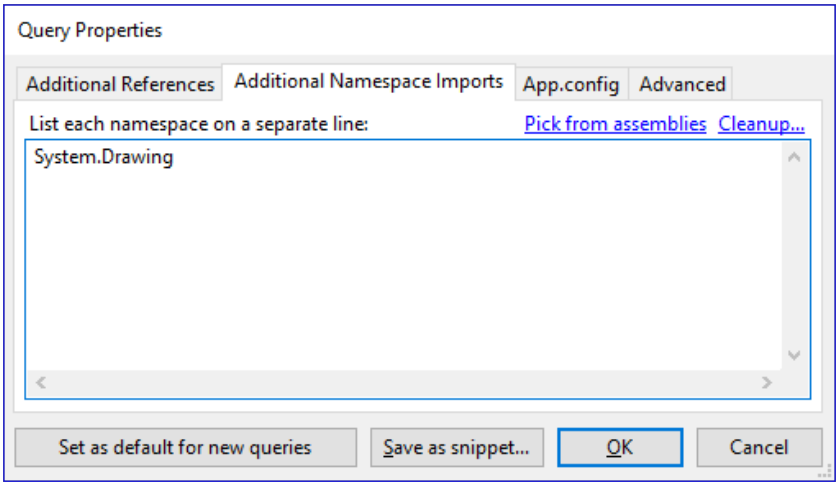


Bild 1: Importieren des Namespaces **System.Drawing**.

Für unser aktuelles Beispiel benötigen wir lediglich eine neue LINQPad-Query mit dem Wert **VB Program** im Auswahlfeld **Language**. Damit wir die Elemente des Namespaces **System.Drawing** verwenden können, fügen wir diesen zu der Query hinzu. Dazu klicken Sie mit der rechten Maustaste auf die Registerseite der Query und wählen den Eintrag **Query Properties...** aus. Hier wechseln Sie zur Registerseite **Additional Namespace Imports** und geben den Namespace **System.Drawing** einfach ein – ohne **Imports**-Anweisung, wie es direkt in Modulen nötig wäre (siehe Bild 1). Anschließend schließen Sie den Dialog mit der **OK**-Schaltfläche wieder.

Pfad zum Speichern der erstellten Bilddateien definieren

Die erzeugten Bilder wollen wir in einem Verzeichnis speichern, wo wir diese schnell wiederfinden. Den Namen dieses Verzeichnisses hinterlegen wir in einer Konstanten, die wir oben im Code wie folgt definieren:

```
Const strPath As String = "C:\...\BitmapTest\"
```

Eine neue Bilddatei erstellen

Wenn Sie einfach nur eine neue Bilddatei erstellen und unter einem bestimmten Namen speichern wollen, gelingt das sehr einfach. Dazu initialisieren Sie ein neues Objekt auf Basis der Klasse **Bitmap** und speichern diese einfach mit der **Save**-Methode. Diese erwartet den Namen der zu erstellenden Datei als Parameter. In einem einfachen Fall übergeben Sie die Höhe und die Breite der zu erstellenden Datei als Parameter beim Initialisieren des Objekts:

```
Dim bmp As New Bitmap(100, 100)
bmp.Save(strPath & "test.bmp")
```

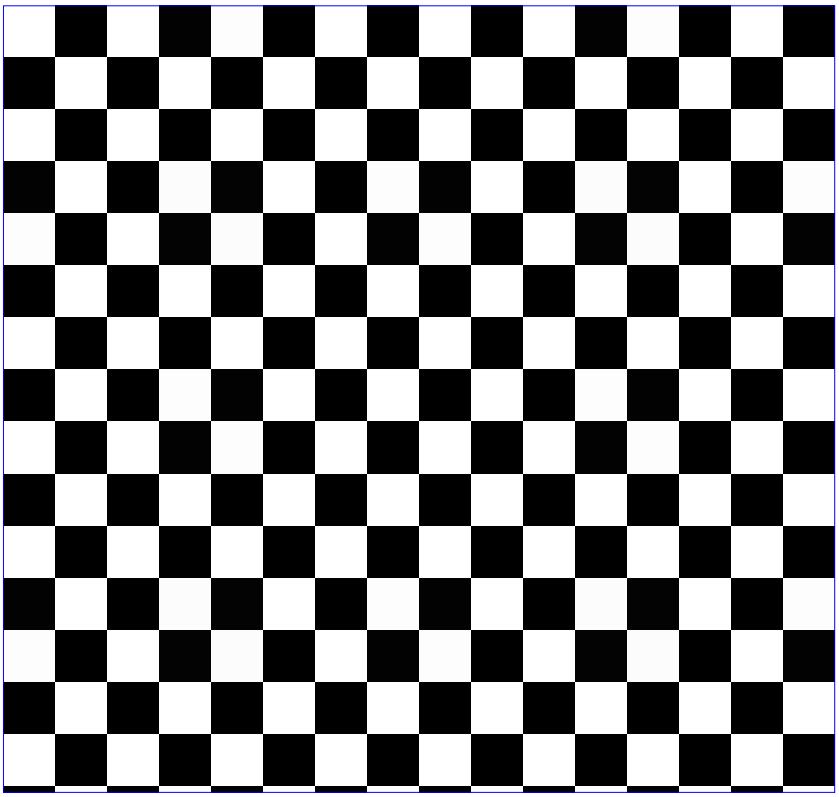


Bild 2: Eine frisch erstellte, komplett transparente Datei

Die so erzeugte Bitmap-Datei finden Sie im angegebenen Verzeichnis vor. Wenn Sie mit einem Bildprogramm öffnen, das in der Lage ist, transparente Bereiche hervorzuheben, sehen Sie, dass die erstellte Datei komplett transparent ist (siehe Bild 2).

Datei speichern und im Standardprogramm öffnen

Bevor wir uns weitere Techniken in Zusammenhang mit dem Bitmap-Objekt anschauen, wollen wir die Testprozedur noch ein wenig praktischer gestalten. Dazu fügen wir als Erstes eine neue Variable namens **strFilename** ein. Diese nimmt auch gleich den Namen der zu erstellenden Datei auf, hier **Test.bmp**. Der Grund ist, dass wir den Dateinamen gleich noch öfter benötigen. Wenn wir diesen dann ändern wollen, brauchen wir diesen nur an einer Stelle anzupassen.

Nach dem Erstellen und Speichern der Datei unter den angegebenen Dateinamen wollen wir diese noch in der dafür vorgesehenen Anwendung öffnen. Das erledigt ein Aufruf der **Start**-Methode der **Process**-Klasse:

```
Sub Main
    Dim bmp As New Bitmap(100, 100)
    Dim strFilename As String
    strFilename = "Test.bmp"
    bmp.Save(strPath & strFilename)
    Process.Start(strPath & strFilename)
End Sub
```

Hinweis: Wenn Sie die Standarddatei ändern wollen, die in Zusammenhang mit einer bestimmten Dateiendung aufgerufen wird, klicken Sie mit der rechten Maustaste auf den Dateinamen im Windows Explorer. Wählen Sie aus dem nun erscheinenden Kontextmenü den Eintrag **Öffnen mit|Andere App auswählen** aus. Selektieren Sie dort die gewünschte Anwendung und setzen Sie einen Haken für die Option **Immer diese App zum Öffnen von .bmp-Dateien verwenden** (siehe Bild 3).

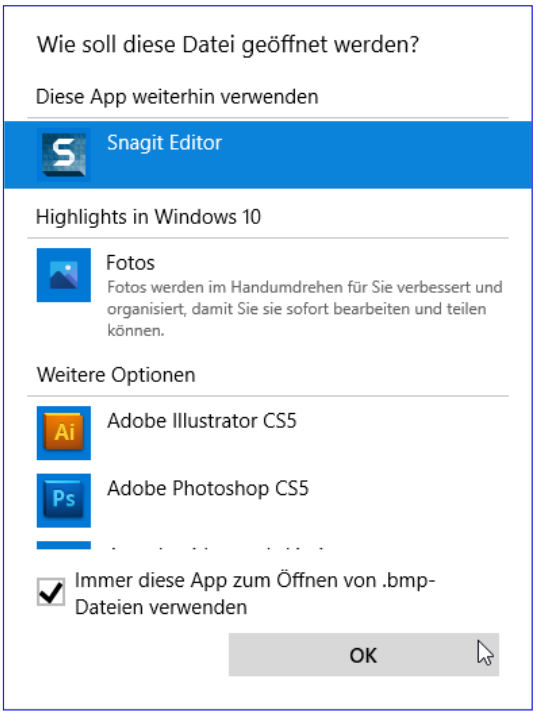


Bild 3: Festlegen der Standardapp für eine Dateiendung

Bilddatei in verschiedenen Formaten speichern

Die **Save**-Methode bietet eine Überladung an, mit der Sie mit dem ersten Parameter den Namen der zu erstellenden Datei übergeben und mit dem zweiten das Bildformat. Hierfür können Sie die folgenden Werte angeben, die Sie über die Auflistung **Imaging.ImageFormat** finden:

- **Bmp**: Speichert das Bild im **Bitmap**-Format.
- **Emf**: Speichert das Bild im **EMF**-Format.
- **Gif**: Speichert das Bild im **GIF**-Format.
- **Jpeg**: Speichert das Bild im **JPEG**-Format.
- **Png**: Speichert das Bild im **PNG**-Format.
- **Tiff**: Speichert das Bild im **TIFF**-Format.
- **Wmf**: Speichert das Bild im **WMF**-Format.

Wenn wir also beispielsweise ein Bild im **PNG**-Format speichern wollen, nutzen wir den folgenden Aufruf der **Save**-Methode:

```
bmp.Save(strPfad & "test.png",Imaging.ImageFormat.Png)
```

Was ist ein Bitmap?

Ein Bitmap ist eine rechteckige Matrix aus wiederum viereckigen Elementen, die Pixel genannt werden. Für jedes Pixel wird ein Farbwert festgelegt. Eine Datei, die ein Bitmap enthält, besteht aus einigen Bytes mit grundlegenden Informationen über das Bitmap, die zum Beispiel die Anzahl der Pixel und die Anzahl der möglichen Farben je Pixel angeben, also die Farbtiefe.

Für die möglichen zur Verfügung stehenden Farben gibt es verschiedene Sets, vom einfachsten, das nur die beiden Farben Schwarz und Weiß enthält bis zu solchen, die 16.777.216 verschiedene Farben je Pixel darstellen können – und gegebenenfalls zusätzlich noch weitere 256 Abstufungen für einen sogenannten Alphakanal zum Definieren von Masken oder Transparenz.

Dementsprechend variiert die Speichergröße einer Datei mit einem Bitmap je nach der angegebenen Farbtiefe. Diese können wir übrigens über eine spezielle Eigenschaft ermitteln.

Wie aber erklären sich beispielsweise die 16.777.216 Farben für jedes Pixel? Diese resultieren aus dem Anteil der Farben Rot, Grün und Blau, die jeweils mit einem Wert von 0 bis 255 angegeben werden.

Es gibt auch noch Bitmaps mit Graustufen – beispielsweise mit 8 Bit. Das bedeutet, dass **0** einem schwarzen Pixel entspricht, **255** einem weißen und die Werte dazwischen geben Grautöne in den verschiedenen Helligkeiten wieder.

Farbtiefe und Farbraum eines Bitmap-Objekts ermitteln

Wenn wir wie oben ein einfaches **Bitmap**-Objekt mit einer beliebigen Auflösung erstellen, können wir mit der Eigenschaft **PixelFormat** eine Information über den Farbraum und die Farbtiefe ausgeben lassen:

```
Dim bmp As New Bitmap(100, 100)
Debug.Print (bmp.PixelFormat.ToString)
```

Die hier verwendete Enumeration **PixelFormat** finden wir im Namespace **System.Drawing.Imaging**, den wir wie **System.Drawing** zur Liste der Namespaces in der LINQPad-Abfrage hinzufügen.

Das Ergebnis lautet:

```
Format32bppArgb
```

Das bedeutet, dass wir ein **Bitmap**-Objekt mit 32-Bit-RGB erstellt haben, also ein 24-Bit-RGB-Bitmap mit 8-Bit-Alphakanal.

Farbtiefe und Farbraum eines Bitmap-Objekts einstellen

Sie können Farbtiefe und Farbraum auch direkt beim Erstellen eines **Bitmap**-Objekts definieren. Dazu nutzen Sie eine Überladung, bei der Sie als dritten Parameter einen Eintrag der Enumeration **PixelFormat** übergeben. Wenn Sie eine Alternative zum 32-Bit-Bitmap benötigen, finden Sie in der Enumeration ausreichend Möglichkeiten.

Da Sie meist auf Bilder mit 32-Bit stoßen werden, gehen wir an dieser Stelle nicht näher auf alternative Versionen ein.

Transparente Bilddatei erstellen

Gegebenenfalls stoßen Sie auf **Bitmap**-Objekte, beispielsweise aus Bilddateien, die keinen Alphakanal enthalten, aber Sie möchten diesen einen solchen hinzufügen. Wir demonstrieren das an einem mit dem Typ **Format24bppRgb** erstellten **Bitmap**-Objekt. Nachdem wir sicherheitshalber einmal den Wert der Eigenschaft **PixelFormat** ausgegeben haben, rufen wir die Methode **MakeTransparent** auf und übergeben dieser den Wert der Farbe, die transparent gemacht werden soll. Anschließend geben wir erneut den Typ aus der Eigenschaft **PixelFormat** aus:

```
Dim bmp As New Bitmap(16, 16, PixelFormat.Format24bppRgb)
Debug.Print(bmp.PixelFormat.ToString)
bmp.MakeTransparent(Color.Black)
Debug.Print(bmp.PixelFormat.ToString)
```

Das Ergebnis lautet:

```
Format24bppRgb
Format32bppArgb
```

Wir fügen mit der **MakeTransparent**-Methode also den Alphakanal zum **Bitmap**-Objekt hinzu.

Einem Pixel eine Farbe zuweisen

Haben Sie ein **Bitmap**-Objekt erstellt, möchten Sie den Pixeln des Objekts gegebenenfalls Farben zuweisen. Das erledigen Sie mit der Methode **SetPixel**. Diese erwartet zumindest die folgenden drei Parameter:

- **x**: x-Koordinate des Pixels
- **y**: y-Koordinate des Pixels
- **Color**: Farbe, die diesem Pixel zugefügt werden soll.

Die Werte für **x** und **y** sind **0**-basiert, bei einem **Bitmap**-Objekt mit 16x16 Pixeln erwartet die Methode **SetPixel** für die ersten beiden Parameter also Werte von **0** bis **15**.

Der dritte Parameter erwartet eine Struktur, welche die zu setzende Farbe repräsentiert. Hier können Sie durch Angabe der Klasse **Color** einige Einträge per IntelliSense auswählen, welche den gängigen Farbwerten entsprechen, beispielsweise **Color.White** oder **Color.Black**. Diese liefern **Color**-Objekte mit den Einstellungen für den **Rot**-, **Grün**- und **Blau**-Anteil sowie gegebenenfalls den Alphakanal und weiteren Eigenschaften.

Sie können die Farbe aber auch direkt aus den Werten für **R**, **G**, **B** und **A** zusammenstellen. Dazu nutzen Sie die Methode **FromArgb** der Klasse **Color** als dritten Parameter der **SetPixel**-Methode.

Bitmap mit schwarzen und weißen Linien

Als Nächstes ein konkretes Beispiel für das Erstellen einer Bilddatei auf Basis eines **Bitmap**-Objekts, das wir mit abwechselnden horizontalen schwarzen und weißen Linien füllen wollen. Hier einmal die Prozedur im Überblick:

```
Sub Main
    Dim strFilename As String = "Test.bmp"
    Dim bmp As New Bitmap(16, 16)
    For y = 0 To bmp.Height - 1
        For x = 0 To bmp.Width - 1
            If y Mod 2 = 0 Then
                bmp.SetPixel(x, y, color.Black)
            Else
                bmp.SetPixel(x, y, color.White)
            End If
        Next x
    Next y
    bmp.Save(strPath & strFilename)
    Process.Start(strPath & strFilename)
End Sub
```

Wir erstellen also ein neues **Bitmap**-Objekt mit 16x16 Pixeln. Dann durchlaufen wir die Zeilen in einer **For... Next**-Schleife über die Variable **y** und die Spalten in einer weiteren Schleife mit der Variablen **x**. Dabei nutzen wir die Werte von **0** bis zur Breite beziehungsweise Höhe minus **1**, da der Index für die Höhe und die Breite **0**-basiert ist.

Innerhalb der inneren Schleife prüfen wir, ob der Wert von **y**, also der Wert für die Zeile, durch **2** geteilt den Rest **0** ergibt (**y mod 2 = 0**). Diese Bedingung ist für jede zweite Zeile wahr. In diesem Fall stellen wir für das Pixel mit den Koordinaten aus x und y die schwarze Farbe ein, in den anderen Fällen die weiße Farbe.

Das Ergebnis sieht wie in Bild 4 aus.

Kariertes Muster erzeugen

Wenn Sie die Prozedur aus dem vorherigen Beispiel wie folgt verändern, erhalten Sie in der Ausgabe ein kariertes Muster aus schwarzen und weißen Pixeln. Hier wechseln wir den Wert der **Boolean**-Variablen **bolBlack** mit jeder Zeile und jeder Spalte im **Bitmap**-Objekt. Hat **bolBlack** den Wert **True**, setzen wir ein schwarzes Pixel, anderenfalls ein weißes:

```
Dim bolBlack As Boolean
```

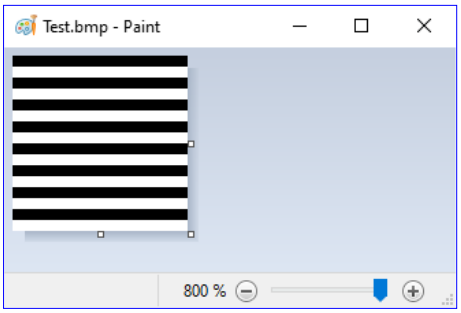


Bild 4: Bild mit vertikalen schwarzen und weißen Linien

Zeichnen in Bitmaps mit der Graphics-Klasse

Im Artikel »Bitmaps programmieren mit VB.NET« haben wir gezeigt, wie Sie mit VB.NET Bitmaps erzeugen und die einzelnen Pixel mit Farben füllen beziehungsweise die enthaltenen Farben auslesen. Im vorliegenden Artikel gehen wir einen Schritt weiter: Wir nutzen eine Klasse namens Graphics, um in einem Bitmap zu zeichnen. Dabei zeigen wir die Techniken zum Erstellen der grundlegenden Formen und liefern Know-how, wie Sie Füllungen und Schraffuren hinzufügen. Schließlich speichern wir die erzeugten Bitmaps im Dateisystem und zeigen diese in der jeweiligen Standardapp für die entsprechende Dateiendung an.

Vorbereitung: LINQPad

Genau wie im oben genannten Artikel **Bitmaps programmieren mit VB.NET** (**www.datenbankentwickler.net/297**) verwenden wir auch in diesem Artikel LINQPad als Spielwiese zum Ausprobieren der Beispiele. Hier brauchen wir zum Testen einiger Zeilen Code nicht extra ein VB-Projekt anzulegen und dieses jedes Mal neu zu starten, um die Funktionsweise zu prüfen. Wir geben die gewünschten Zeilen einfach in eine Methode ein und starten diese mit **F5** – ganz wie es beispielsweise im VBA-Editor möglich ist. Wie Sie LINQPad nutzen, lesen Sie im Artikel **LINQPad: LINQ, C# und VB einfach ausprobieren** (**www.datenbankentwickler.net/100**).

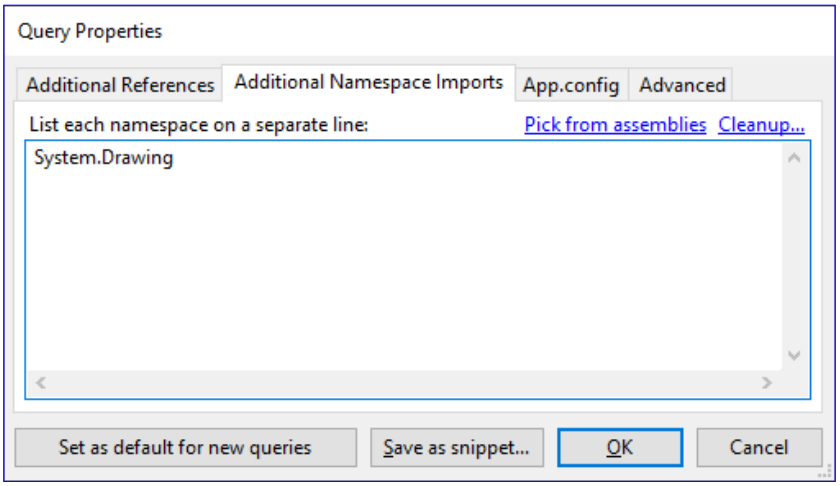


Bild 1: Hinzufügen eines Verweises auf den Namespace **System.Drawing**

Für unser aktuelles Beispiel benötigen wir lediglich eine neue LINQPad-Query mit dem Wert **VB Program** im Auswahlfeld **Language**. Damit wir die Elemente des Namespaces **System.Drawing** verwenden können, fügen wir diesen zu der Query hinzu. Dazu klicken Sie mit der rechten Maustaste auf den Registerreiter der Query und wählen den Eintrag **Query Properties...** aus. Hier wechseln Sie zur Registerseite **Additional Namespace Imports** und geben den Namespace **System.Drawing** einfach ein – ohne **Imports**-Anweisung, wie es direkt in Modulen nötig wäre (siehe Bild 1). Anschließend schließen Sie den Dialog mit der **OK**-Schaltfläche wieder.

Auch in diesem Artikel wollen wir die erzeugten Bilder in einem Verzeichnis speichern, wo wir diese schnell wiederfinden. Den Namen dieses Verzeichnisses hinterlegen wir in einer Konstanten, die wir oben im Code wie folgt definieren:

```
Const strPath As String = "C:\...\BitmapTest\"
```

Zeichnen mit Graphics

Während Sie in Objekten der **Bitmap**-Klasse mit der Methode **SetPixel** nur die Farbe für einzelne Pixel festlegen können, bietet die **Graphics**-Klasse einige weitere Möglichkeiten. Die **Graphics**-Klasse stellt Methoden wie **DrawLine** zum Zeichnen von Linien oder **DrawString** zum Einfügen von Texten zur Verfügung.

Dabei referenziert das Objekt auf Basis der **Graphics**-Klasse zunächst das Element, dem beispielsweise Linien oder Texte hinzugefügt werden sollen. Um die im oben genannten Artikel vorgestellte **Bitmap**-Klasse direkt einzubinden, nutzen wir die dort erzeugten Bitmaps quasi als Zeichenblatt. Das **Graphics**-Objekt füllen wir dabei mit der **FromImage**-Methode mit dem **Bitmap**-Element.

```
Dim bmp As New Bitmap(480, 320)
Dim objGraphics As Graphics
objGraphics = Graphics.FromImage(bmp)
```

Danach können wir die nachfolgenden Methoden nutzen, um dem **Bitmap**-Objekt über die **Graphics**-Klasse Elemente hinzuzufügen:

- **DrawArc**: Zeichnet einen Bogen.
- **DrawBezier**: Zeichnet eine Bézier-Kurve.
- **DrawBeziers**: Zeichnet mehrere Bézier-Kurven.
- **DrawClosedCurve**: Zeichnet eine geschlossene Cardinal-Splinekurve.
- **DrawEllipse**: Zeichnet eine Ellipse.
- **DrawIcon**: Zeichnet das durch das angegebene Icon dargestellte Bild.
- **DrawIconUnstretched**: Zeichnet das durch das angegebene Icon dargestellte Bild, ohne das Bild zu skalieren.
- **DrawImage**: Zeichnet das angegebene Bild in seiner ursprünglichen physischen Größe.
- **DrawImageUnscaled**: Zeichnet das angegebene Bild in seiner ursprünglichen physischen Größe.
- **DrawImageUnscaledAndClipped**: Zeichnet das angegebene Bild ohne Skalierung und beschneidet es ggf. auf die Größe des angegebenen Rechtecks.
- **DrawLine**: Zeichnet eine Linie.
- **DrawLines**: Zeichnet eine Reihe von Linien.

- **DrawPath**: Zeichnet einen grafischen Pfad:
- **DrawPie**: Zeichnet eine durch eine Ellipse definierte Kreisform.
- **DrawPolygon**: Zeichnet ein Vieleck.
- **DrawRectangle**: Zeichnet ein Rechteck.
- **DrawString**: Zeichnet die angegebene Textzeichenfolge.

Diese Methoden sehen wir uns in den folgenden Abschnitten im Detail an und stellen dabei weitere benötigte Klassen vor wie beispielsweise solche zur Angabe des zu verwendenden Stiftes.

Eine einfache Linie hinzufügen

Den Start machen wir mit einer einfachen Methode, die eine Linie zu einem Bitmap von 480 x 320 Pixeln hinzufügen soll. Im ersten Beispiel nutzen wir die Überladung der **DrawLine**-Methode, welche mit dem ersten Parameter ein **Pen**-Objekt entgegennimmt und mit den übrigen vieren die Koordinaten des Startpunkts und des Endpunkts der zu zeichnenden Linie. Da die Koordinaten **0**-basiert sind, soll unsere diagonale Linie vom Punkt 0,0 zum Punkt 479, 319 verlaufen:

```
Sub EinfacheLinie
    Dim strFilename As String = "EinfacheLinie.bmp"
    Dim bmp As New Bitmap(480, 320)
    Dim objGraphics As Graphics
    Dim objPen As New Pen(color.Black)
    objGraphics = Graphics.FromImage(bmp)
    objGraphics.DrawLine(objPen, 0,0,479,319)
    bmp.Save(strPath & strFilename)
    Process.Start(strPath & strFilename)
End Sub
```

Hier definieren wir außerdem ein **Pen**-Element, bei dessen Initialisierung wir eine Überladung verwenden, welche nur die Farbe des zu verwendenden Stiftes entgegennimmt – hier **Color.Black**. Das mit **objPen** referenzierte **Pen**-Element übergeben wir als ersten Parameter der **DrawLine**-Methode. Anschließend speichern wir das **Bitmap**-Objekt und zeigen es in der entsprechenden Anwendung an – hier in **Paint**. Das Ergebnis sehen sie in Bild 2.

Einfache Linie mit Point-Elementen als Parameter

Gegebenenfalls liegen Ihnen der Start- und der Endpunkt der Linie als Koordinaten bereits vor und Sie möchten diese in Form von **Point**-Objekten erfassen. Dazu erstellen Sie wie folgt zwei neue **Point**-Objekte, und zwar vor dem Aufruf der **DrawLine**-Methode:

```
Dim objStartPoint As New Point(0,0)
Dim objEndpoint As New
Point(479,319)
```

In der **DrawLine**-Methode geben Sie die Punkte aus **objStartpoint** und **objEndpoint** dann als zweiten und dritten Parameter an:

```
objGraphics.DrawLine(objPen, objSt-
artPoint, objEndpoint)
```

Den Stift mit der Pen-Klasse anpassen

Die oben gezeichnete Linie ist eine schwarze, durchgezogene Linie mit einer Breite von einem Pixel. Die Farbe Schwarz haben wir beim Erstellen des **Pen**-Elements als Parameter angegeben, die übrigen Eigenschaften werden standardmäßig eingestellt:

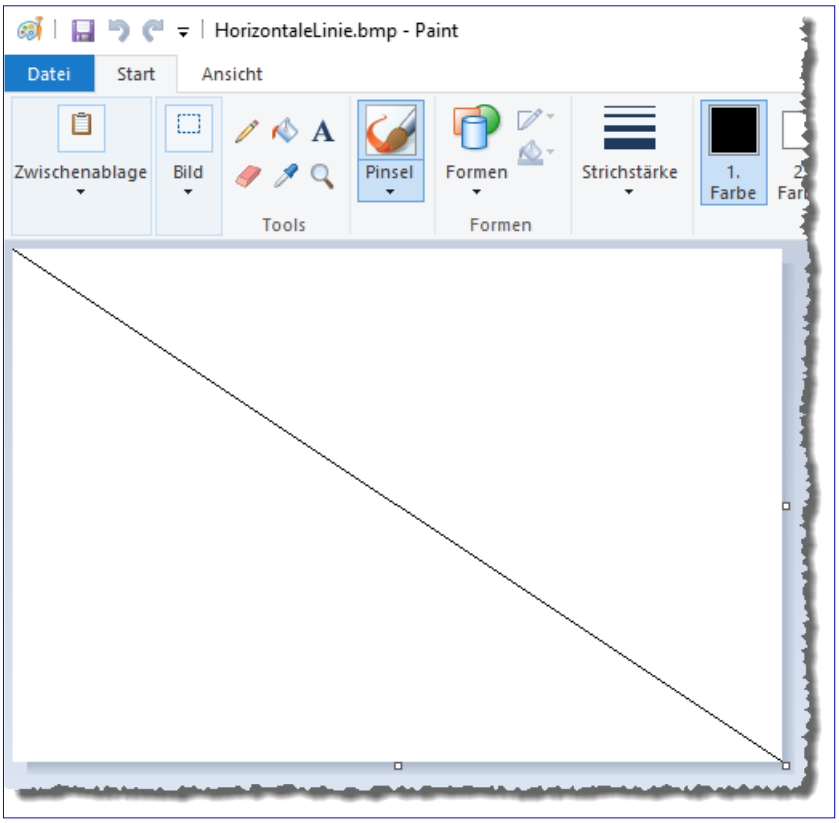


Bild 2: Eine mit der **DrawLine**-Methode erstellte Linie.

```
Dim objPen As New Pen(Color.Black)
```

Die Breite können Sie mit dem zweiten Parameter einer weiteren Überladung des Konstruktors der **Pen**-Klasse angeben, hier zum Beispiel für eine Linie mit einer Breite von drei Pixeln:

```
Dim objPen As New Pen(Color.Black, 3)
```

Später schauen wir uns weitere Möglichkeiten der **Pen**-Klasse an.

Einen Linienzug zeichnen

Die Methode **DrawLines** erlaubt das Zeichnen mehrerer zusammenhängender Linien gleichzeitig, wobei Sie die Koordinaten der Start- und Endpunkte als **Point**-Elemente jeweils mit einem Array übergeben.

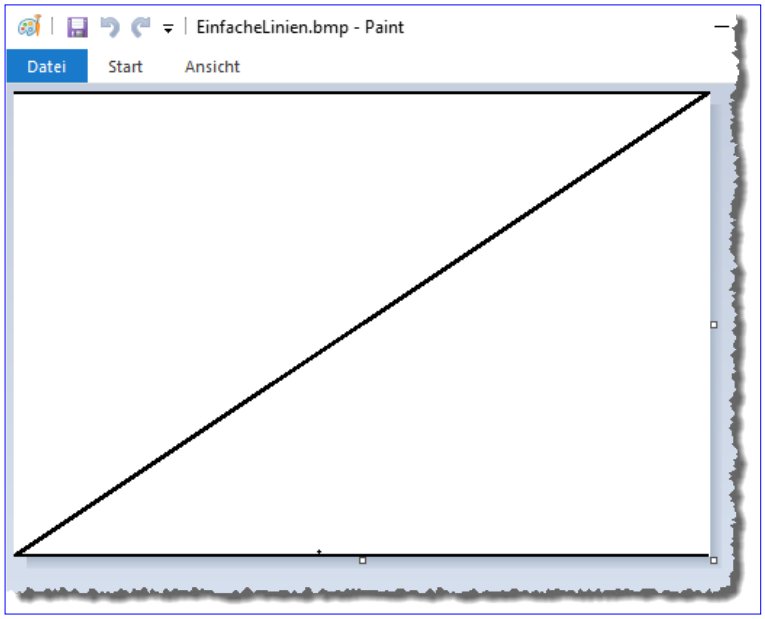


Bild 3: Ein Linienzug in Z-Form

Im folgenden Beispiel erstellen wir wieder ein Bitmap mit 480 x 320 Pixeln und referenzieren dieses mit dem **Graphics**-Element. Dann erstellen wir das Array der Punkte, wobei die Punkte jeweils als neue **Point**-Elemente angeben. Danach rufen wir die **DrawLines**-Methode auf und übergeben dieser als zweiten Parameter das Array aus **objPoints** (siehe Methode **EinfacherLinienzug** in der Beispieldatei):

```
Dim strFilename As String = "EinfacheLinien.bmp"
Dim bmp As New Bitmap(480, 320)
Dim objGraphics As Graphics
Dim objPoints As Point() = {New Point(0,0), New Point(479,0), New Point(0,319), New Point(479,319)}
Dim objPen As New Pen(color.Black, 3)
objGraphics = Graphics.FromImage(bmp)
objGraphics.DrawLines(objPen, objPoints)
bmp.Save(strPath & strFilename)
Process.Start(strPath & strFilename)
```

Das Ergebnis sehen wir in Bild 3. Der resultierende Linienzug liefert ein Z.

Rechtecke zeichnen mit DrawRectangle

Als Nächstes schauen wir uns an, wie wir Rechtecke zeichnen können. Ein Rechteck erfordert prinzipiell genau wie eine Linie nur die Angabe zweier Koordinaten – die des oberen linken und des unteren rechten Punktes. Zum Erstellen eines Rechtecks verwenden wir die Methode **DrawRectangle**.

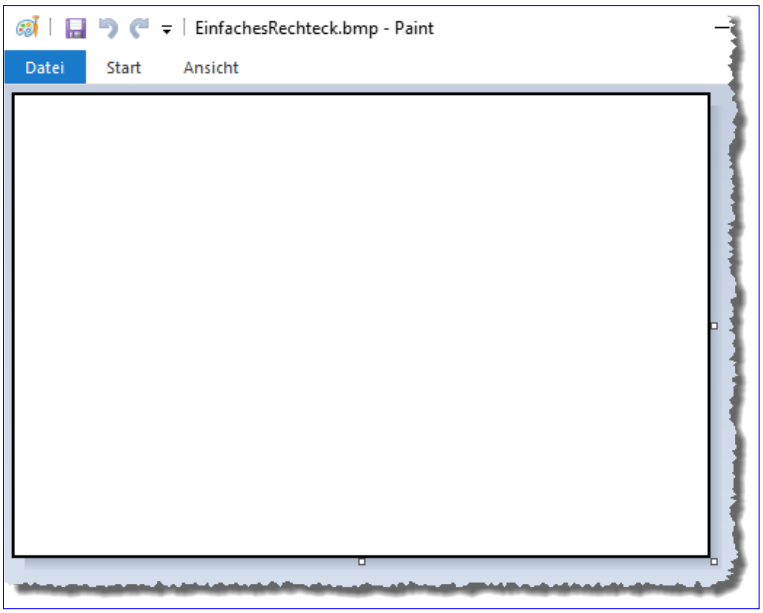


Bild 4: Ein einfaches Rechteck

Diese erwartet in einer Überladung statt der oben angegebenen Koordinaten jedoch nur eine absolute Koordinate für den ersten Punkt mit x und y sowie mit weiteren Parametern die Angabe der gewünschten Breite und Höhe. Das sieht für ein Rechteck um das komplette Bitmap wie folgt aus (siehe Bild 4):

```
Dim strFilename As String = "EinfachesRechteck.bmp"
Dim bmp As New Bitmap(480, 320)
Dim objGraphics As Graphics
Dim objPen As New Pen(color.Black, 3)
objGraphics = Graphics.FromImage(bmp)
objGraphics.DrawRectangle(objPen, 0, 0, 479, 319)
```

Rechteck auf Basis einer Rect-Struktur erstellen

Sie können allerdings auch eine **Rect**-Struktur erstellen, welche prinzipiell die gleichen Informationen enthält, und diese als zweiten Parameter an die **DrawRectangle**-Methode übergeben. Wir haben das Rechteck nun mit einem Abstand von 10 Pixeln zu den Seitenrändern erstellt.

Das Ergebnis sieht wie in Bild 5 aus, die vollständige Methode finden Sie unter dem Namen **EinfachesRechteck_Rect** in der Beispieldatei:

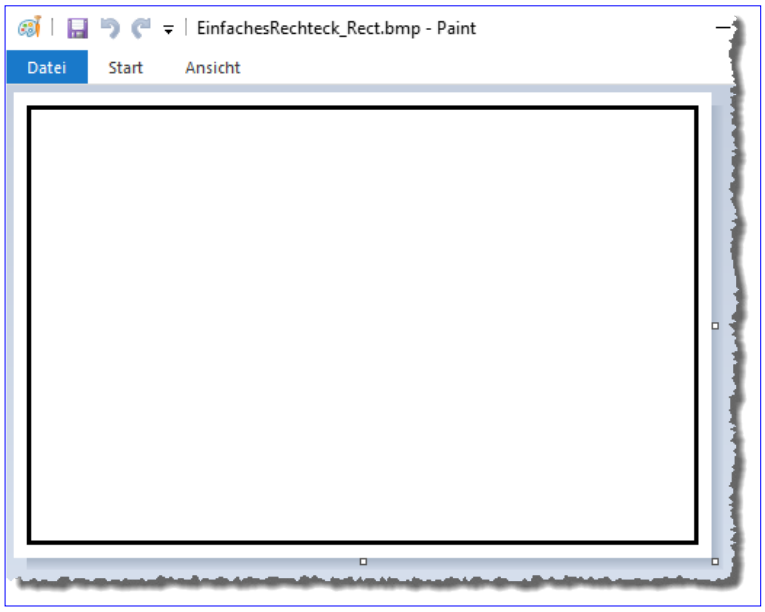


Bild 5: Ein Rechteck mit 10 Pixeln Abstand zum Rand des Bitmaps

```
Dim strFilename As String = "Einfaches-
Rechteck_Rect.bmp"
Dim bmp As New Bitmap(480, 320)
Dim objGraphics As Graphics
Dim objRectangle As New rectangle(10, 10, 459, 299)
Dim objPen As New Pen(color.Black, 3)
objGraphics = Graphics.FromImage(bmp)
objGraphics.DrawRectangle(objPen, objRectangle)
```

Füllen eines Rechtecks mit einer Farbe

Gegebenenfalls möchten Sie das Rechteck auch mit einer Farbe füllen wie in Bild 6.

Bild 6: Gefülltes Rechteck

Wider Erwarten gibt es keine Überladung für die **DrawRectangle**-Methode, welche die Angabe einer Füllfarbe zulässt. Stattdessen gibt es eine weitere Methode namens **FillRectangle**, der wir ebenfalls eine **Rect**-Struktur mit den Abmessungen des Rechtecks übergeben können. Außerdem erwartet diese die Füllfarbe in Form eines **Brush**-Elements. Die **FillRectangle**-Methode füllt allerdings kein zuvor noch zu erstellendes Rechteck, sondern sie erstellt ein neues Rechteck, das direkt mit der angegebenen Farbe gefüllt

wird. Wenn Sie also beispielsweise ein Rechteck mit einem schwarzen Rahmen erstellen wollen, das mit roter Farbe gefüllt ist, benötigen Sie sowohl den Aufruf einer **DrawRectangle**- als auch einer **FillRectangle**-Methode. In diesem Fall definieren wir das Rechteck mit dem schwarzen Rahmen wie zuvor. Damit das rote, ausgefüllte Rechteck genau innerhalb des schwarzen Rechtecks angezeigt wird und den Rahmen nicht überdeckt, beginnt es bei den Koordinaten **x=12** und **y=12** (die Linie des schwarzen Rechtecks startet bei **(10|10)**, aber da wir als Breite 3 Pixel angegeben haben, wird auch noch das Pixel **(11|11)** vom Rand des Rechtecks eingenommen – und natürlich noch die anderen Pixel, die innerhalb der eigentlichen Linie liegen). Die Breite und die Höhe stellen wir ebenfalls entsprechend ein. So erstellen wir also zwei **Rectangle**-Objekte namens **objRectangle** und **objRectangleFill** und nutzen diese dann jeweils als zweiten Parameter der Methoden **DrawRectangle** und **FillRectangle**:

```
Dim strFilename As String = "Rechteck_Gefuellt.bmp"
Dim bmp As New Bitmap(480, 320)
Dim objGraphics As Graphics
Dim objRectangle As New Rectangle(10, 10, 459, 299)
Dim objRectangleFill As New Rectangle(12, 12, 456, 296)
Dim objPen As New Pen(color.Black, 3)
Dim objBrush As New SolidBrush(color.Red)
objGraphics = Graphics.FromImage(bmp)
objGraphics.DrawRectangle(objPen, objRectangle)
objGraphics.FillRectangle(objBrush, objRectangleFill)
```

Zu beachten ist hier noch, dass **FillRectangle** kein **Pen**-, sondern ein **Brush**-Element als ersten Parameter erwartet. **Brush** ist dabei nur eine Schnittstelle, die durch verschiedene Klassen implementiert wird – beispielsweise durch **SolidBrush**, welche wir hier verwenden. Anschließend schauen wir uns verschiedene **Brush**-Ableitungen an.

Füllen mit der Brush-Schnittstelle

Brush ist wie bereits beschrieben nur eine Schnittstelle, Sie können also nicht einfach ein Objekt des Typs **Brush** erstellen wie hier:

```
Dim objBrush As Brush
```

Stattdessen nutzen Sie eine der **Brush**-Ableitungen, von denen die einfachste die **SolidBrush**-Klasse ist. Es gibt beispielsweise noch die folgenden **Brush**-Ableitungen:

- **SolidBrush**: Definiert eine einfache Füllfarbe.
- **LinearGradientBrush**: Definiert zwei Farben und zwei Punkte. Zwischen den beiden angegebenen Punkten gehen die Farben ineinander über.
- **HatchBrush**: Liefert schraffierte Füllmuster.

Texte in Bitmaps einfügen

In zwei weiteren Artikeln haben wir beschreiben, wie man mit dem Bitmap- und dem Graphics-Objekt arbeitet, um Bilddateien zu erstellen und diese mit einzelnen Pixeln oder auch mit Linien, Formen und Kurven füllt. Im vorliegenden Artikel gehen wir noch einen Schritt weiter und schauen uns an, wie sich Text in Bitmaps einfügen lassen und welche Möglichkeiten sich für ihre Platzierung und Ausrichtung ergeben.

Vorbereitende Informationen

Die beiden weiteren Artikel heißen **Bitmaps programmieren mit VB.NET** (**www.datenbankentwickler.net/297**) und **Zeichnen in Bitmaps mit der Graphics-Klasse** (**www.datenbankentwickler.net/298**). Hier finden Sie die grundlegenden Informationen und Techniken, auf die wir auch in diesem Artikel zugreifen.

Genau wie in den oben genannten Artikeln verwenden wir auch in diesem Artikel **LINQPad** als Spielwiese zum Ausprobieren der Beispiele – dort finden Sie auch Informationen zur Vorbereitung von LINQPad für das Reproduzieren unserer Beispiele. Wenn Sie LINQPad bereits installiert haben, können Sie einfach die Datei **Graphics-Programmierung_Texte.linq** aus dem Download zu diesem Artikel per Doppelklick öffnen und finden alles Notwendige.

Texte einfügen

Anlass für diesen und die vorherigen Artikel war eine Anfrage eines Kunden, der eigentlich mit Access arbeitet und dort in einem Bericht Label für Behälter ausdrucken sollte, wobei der Text sich in einer Raute befindet. Aus Platzgründen wollte er diese Raute aber um 45° drehen, sodass er ein Quadrat erhält, in dem jedoch der Text um 45° gekippt enthalten ist. Das wiederum kann man mit Access-Berichten nicht abbilden. Die Idee war dann, die komplette Raute mit Text direkt um 45° gekippt in einer Bilddatei abzubilden, die man dann über die Datensatzquelle und ein Bild-Steuerelement in den Bericht einbinden kann.

Welche Möglichkeiten bietet nun also .NET zum Einfügen von Texten in Bilddateien? Schauen wir uns zunächst ein einfaches Beispiel an:

```
Sub TextInBitmap_Simple
    Dim strFilename As String = "TextInBitmap.png"
    Dim bmp As New Bitmap(480, 480)
    Dim objGraphics As Graphics
    objGraphics = Graphics.FromImage(bmp)
    objGraphics.DrawString("Beispieltext", New font("Tahoma", 24), Brushes.Black, 0, 0)
    bmp.Save(strPath & strFilename)
    Process.Start(strPath & strFilename)
End Sub
```

In diesem Beispiel definieren wir als Erstes den Dateinamen für die zu erstellende Datei. Dann erstellen wir ein neues **Bitmap**-Element mit einer Größe von 480x480 Pixeln. Dieses nutzen wir dann als Basis für ein neues **Graphics**-Element, welches das **Bitmap**-Element mit der **FromImage**-Methode referenziert. Anschließend nutzen wir bereits die DrawString-Methode, um einen Text zu dem **Graphics**-Objekt hinzuzufügen. Dabei übergeben wir mit dem ersten Parameter den anzuzeigenden Text, mit dem zweiten ein Objekt, das die Schriftart definiert sowie mit dem dritten den zu verwendenden »Pinsel«. Danach folgt bereits das Speichern des erstellten Bitmap-Objekts als **.png**-Datei. Hier nutzen wir noch die Konstante mit dem Verzeichnis, das wir zum Speichern nutzen und die wir im Modulkopf wie folgt definiert haben:

Bild 1: Ausgabe eines ersten Beispiels

```
Const strPath As String = "C:\...\GraphicsTest\"
```

Die letzte Anweisung öffnet die standardmäßig für diesen Dateityp vorgesehene App und zeigt das erstellte Bild darin an (siehe Bild 1).

Die DrawString-Methode

Die **DrawString**-Methode ist das Werkzeug der Klasse **Graphics**, wenn es um die Ausgabe von Texten geht. Sie bietet insgesamt sechs Überladungen an, von denen wir uns die wichtigsten Parameter ansehen:

- **String**: Zeichenfolge, die gezeichnet werden soll
- **Font**: Font-Objekt, das Informationen über die zu verwendende Schriftart, Schriftgröße und weitere Eigenschaften enthält
- **Brush**: Erwartet ein Objekt, das auf einer Implementierung der Brush-Klasse basiert. Im einfachsten Fall nutzt man einfach beispielsweise **Brushes.Black** für einen schwarzen »Pinsel«.
- **x**: x-Koordinate, an welcher der Text platziert werden soll
- **y**: y-Koordinate, an welcher der Text platziert werden soll
- **Format**: Gibt die Formatierung für den zu zeichnenden Text an wie Zeilenabstand, Ausrichtung und so weiter.

Schriftart festlegen

Um die gewünschte Schriftart festzulegen, kann man einfach ein neues Font-Objekt unter Angabe der wichtigsten Parameter erstellen. Für eine fette Schrift des Typs **Courier New** in der Größe 24 sieht das Erstellen eines solchen Objekts wie folgt aus:

```
Dim objFont As New Font("Courier New", 24, FontStyle.Bold)
```

Die Objektvariable **objFont** können wir dann einfach als zweiten Parameter der **DrawString**-Methode nutzen:

```
objGraphics.DrawString("Beispieltext", objFont, Brushes.Black, 0, 0)
```

Brush festlegen

Das **Brush**-Objekt gibt an, in welcher Farbe und mit welchen weiteren »Pinsel«-spezifischen Eigenschaften der Text gezeichnet werden soll. Wenn einfach eine Farbe angegeben wird, mit welcher der Text gezeichnet werden soll, bietet sich eines der Elemente der **Brushes**-Auflistung an. IntelliSense bietet hier alle verfügbaren Farben an, die für den alltäglichen Gebrauch ausreichen sollten (siehe Bild 2).

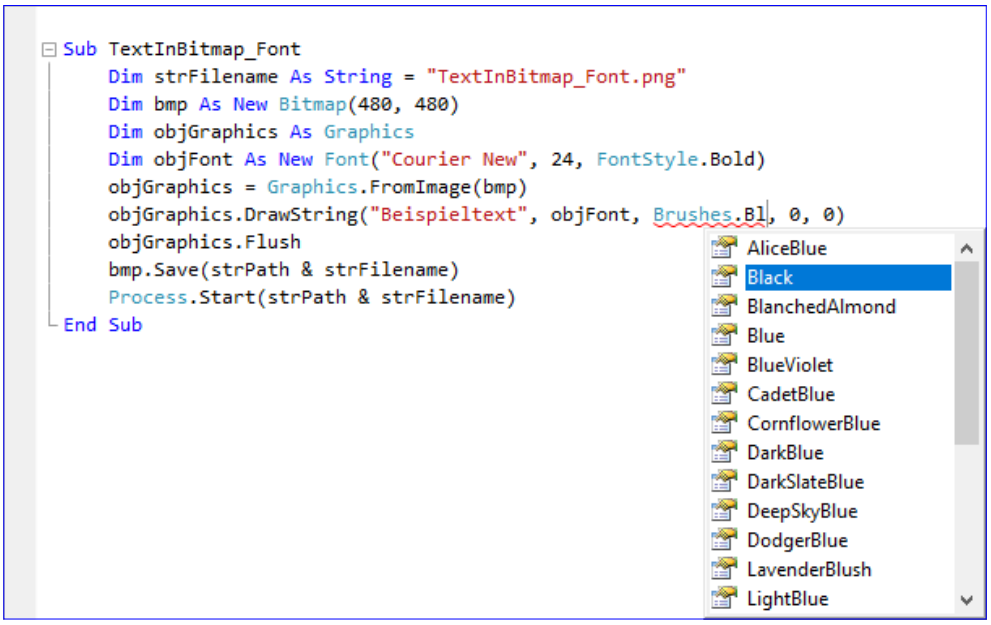


Bild 2: Auswahl eines »Pinsels« über die **Brushes**-Auflistung

Sie können auch eine eigene Objektvariable für das **Brush**-Element deklarieren. Dabei ist wichtig, dass Sie nicht versuchen, die **Brush**-Klasse selbst zu nutzen, da diese nur eine Schnittstellenklasse ist. Die Implementierungen der **Brush**-Klasse wie **HatchBrush**, **LinearGradientBrush** oder **SolidBrush** aber liefern

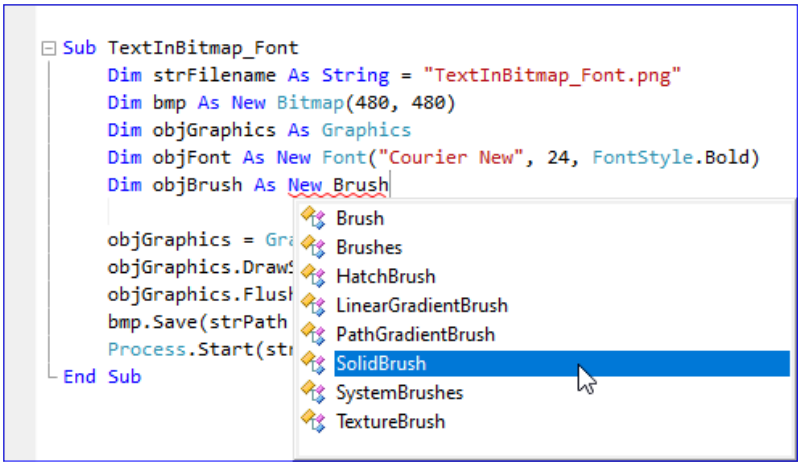


Bild 3: Implementierungen der **Brush**-Klasse

IntelliSense, wenn Sie hinter dem **New**-Schlüsselwort **Brush** eingeben (siehe Bild 3).

Beim Initialisieren einer der **Brush**-Implementierungen mit **New** müssen Sie außerdem die Farbe für den Pinsel angeben. Das können Sie beispielsweise über die Auflistung **Color** erledigen, hier wieder für einen schwarzen Pinsel der Klasse **SolidBrush**:

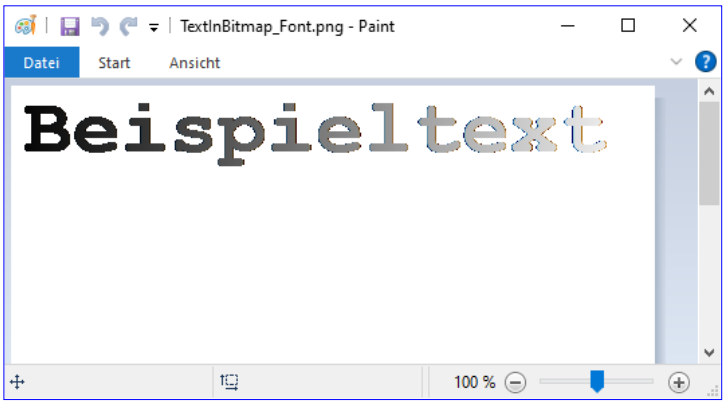


Bild 4: Schrift mit Farbverlauf

```
Dim objBrush As New SolidBrush(Color.Black)
```

Dieses **SolidBrush**-Objekt weisen Sie dann per Parameter wie folgt der **DrawString**-Methode zu:

```
objGraphics.DrawString("Beispieltext", objFont, objBrush, 0, 0)
```

Alternativ können Sie auch andere Pinsel nutzen, beispielsweise um die Schrift mit einem Farbverlauf anzuzeigen. Wenn die Schrift links schwarz sein soll und dann nach rechts einen Verlauf in Richtung Weiß nehmen soll, nutzen Sie einen Pinsel, der auf Basis der Klasse **LinearGradientBrush** erstellt wird. Der erste und zweite Parameter geben den Start- und Endpunkt des Verlaufs an und der dritte und vierte die Farben für den Verlauf:

```
Dim objBrush As New LinearGradientBrush(New Point(0,0), New Point(480,0), Color.Black, Color.White)
```

Wenn wir die Schriftgröße so anpassen, dass der Text **Beispieltext** sich über den gesamten Farbverlauf mit einer Breite von 480 Pixeln erstreckt, sieht das Ergebnis wie in Bild 4 aus (siehe Methode **TextInBitmap_Font** in der Beispieldatei).

Platzierung des Textes

Die Parameter **x** und **y** geben den linken, oberen Punkt des für den Text benötigten Bereichs an. Hier sind keine weiteren Erläuterungen nötig. Wir müssen uns aber über die Alternativen unterhalten, die in weiteren Überladungen der **DrawString**-Methode zu finden sind. Die erste wirkt sich bezüglich der Positionierung des Textes nicht anders aus als bei der Angabe der Koordinaten von x und y: Sie erwartet statt dieser beiden Parameter eine **Point**-Struktur, welche wiederum die x- und die y-Koordinate enthält.

Eine andere Wirkung, gerade bezogen auf die weiter unten besprochene Ausrichtung des Texts, hat jedoch die Variante mit der Angabe eines **Rect**-Elements. Dieses erwartet die Angabe der x- und y-Koordinate der linken, oberen Ecke sowie die Breite und die Höhe des Bereichs, in dem die Schrift platziert werden soll. Ein Beispiel unter Verwendung eines **Rect**-Elements sieht wie folgt aus:

```
Dim objRect As New Rectangle(20, 20, 440, 440)
...
```

EPC-QR-Code für Rechnungen

Was für eine Erlösung! Nach jahrelanger Quälerei beim Eintippen ewig langer IBANs und Rechnungsnummern in die Banking-App im Smartphone gibt es mittlerweile nicht nur ein Format für einen QR-Code, der alle notwendigen Daten bereitstellt, sondern auch Banken, deren Banking-App das Einlesen von Rechnungsdaten wie den Zahlungsempfänger, IBAN, BIC, Verwendungszweck und Zahlungsbetrag aus einem QR-Code erlauben. Noch schöner: Das Format ist überschaubar und mithilfe einfacher Erweiterungen zum Erstellen von QR-Codes bauen wir uns unsere eigenen EPC-QR-Codes, die wir unseren Rechnungen hinzufügen können. Auf dass unsere Kunden uns feiern, weil sie viel Zeit und Nerven beim Bezahlen unserer Rechnungen sparen!

Der EPC-QR-Code oder GiroCode

Der EPC-QR-Code, der auch als GiroCode bekannt ist, wurde vom European Payments Council standardisiert. Der QR-Code enthält alle notwendigen Informationen für eine SEPA-Überweisung, also beispielsweise für Überweisungen im deutschen Zahlungsraum. Der QR-Code erfasst neben einigen technischen Daten die folgenden Informationen.

Bild 1: Beispiel für einen EPC-QR-Code

Es gibt zwei Versionen, deren Anforderungen sich unwesentlich unterscheiden – hier ist die von uns verwendete mit BIC als Pflichtangabe:

- BIC der Empfängerbank: Verpflichtend nur in Version 1.
- Name des Zahlungsempfängers (maximal 70 Zeichen)
- IBAN des Empfängerkontos
- Zahlungsbetrag im Format **EUR#.##**
- Zweck nach dem DTA-Verfahren (optional)
- Referenz als 35-Zeichen-Code gemäß ISO 11649 RF Creditor Reference (optional)
- Verwendungszweck: maximal 140 Zeichen langer Text (optional)

Ein Beispiel für einen EPC-QR-Code mit den relevanten Informationen sieht wie in Bild 1 aus.

Der in diesem EPC-QR-Code kodierte Text lautet wie folgt:

```
BCD
001
1
SCT
BFSWDE33BER
Spende fuer Wikipedia
DE33100205000001194700
EUR10.00


Wikimedia Foerdergesellschaft
```

In diesem Artikel

In diesem Artikel erstellen wir eine Lösung, mit der wir die Daten, die im EPC-QR-Code abgebildet werden sollen, eingeben und das Ergebnis direkt als Bilddatei im Fenster der Anwendung anzeigen können.

In einem weiteren Artikel namens **EPC-QR-Code per DLL** (**www.datenbankentwickler.net/301**) schauen wir uns an, wie wir eine handliche DLL programmieren können, die wir unter anderem per VBA nutzen können – beispielsweise, um Access-Rechnungsberichte mit diesem EPC-QR-Code zu versehen.

Win-Win

Dadurch, dass der Rechnungsempfänger keine Daten mehr manuell von der Rechnung ablesen und in das Onlineformular einer Überweisung eingeben muss, entstehen zwei wesentliche Vorteile: der Empfänger spart sehr viel Zeit und Empfänger und Rechnungsersteller profitieren davon, dass keine Fehler mehr beim Eingeben der Überweisungsdaten entstehen.

Anlegen des Projekts

Wir erstellen das Projekt als WPF-App mit Visual Basic als Sprache und speichern es unter dem Namen **QRUeberweisung**.

NuGet-Paket zum Erstellen von QR-Codes hinzufügen

Mit einem Klick auf den Menübefehl **Projekt|NuGet-Pakete verwalten...** öffnen wir den **NuGet-Paket-Manager**. Hier suchen wir im Bereich **Durchsuchen** nach Einträgen, die dem Suchbegriff **qr code** entsprechen und finden zum Beispiel den Eintrag **QrCode.Net**, den wir für dieses Projekt nutzen wollen.

Ein Klick auf die Schaltfläche **Installieren** fügt das NuGet-Paket zum Projekt hinzu (siehe Bild 2).

Fenster zur Eingabe der benötigten Daten programmieren

Die Daten wie BIC, IBAN, Empfänger, Verwendungszweck und Betrag wollen wir über ein Fenster eingeben können. Neben den entsprechenden Textfeldern soll das Fenster noch eine Schaltfläche enthalten, mit der wir den QR-Code erstellen.

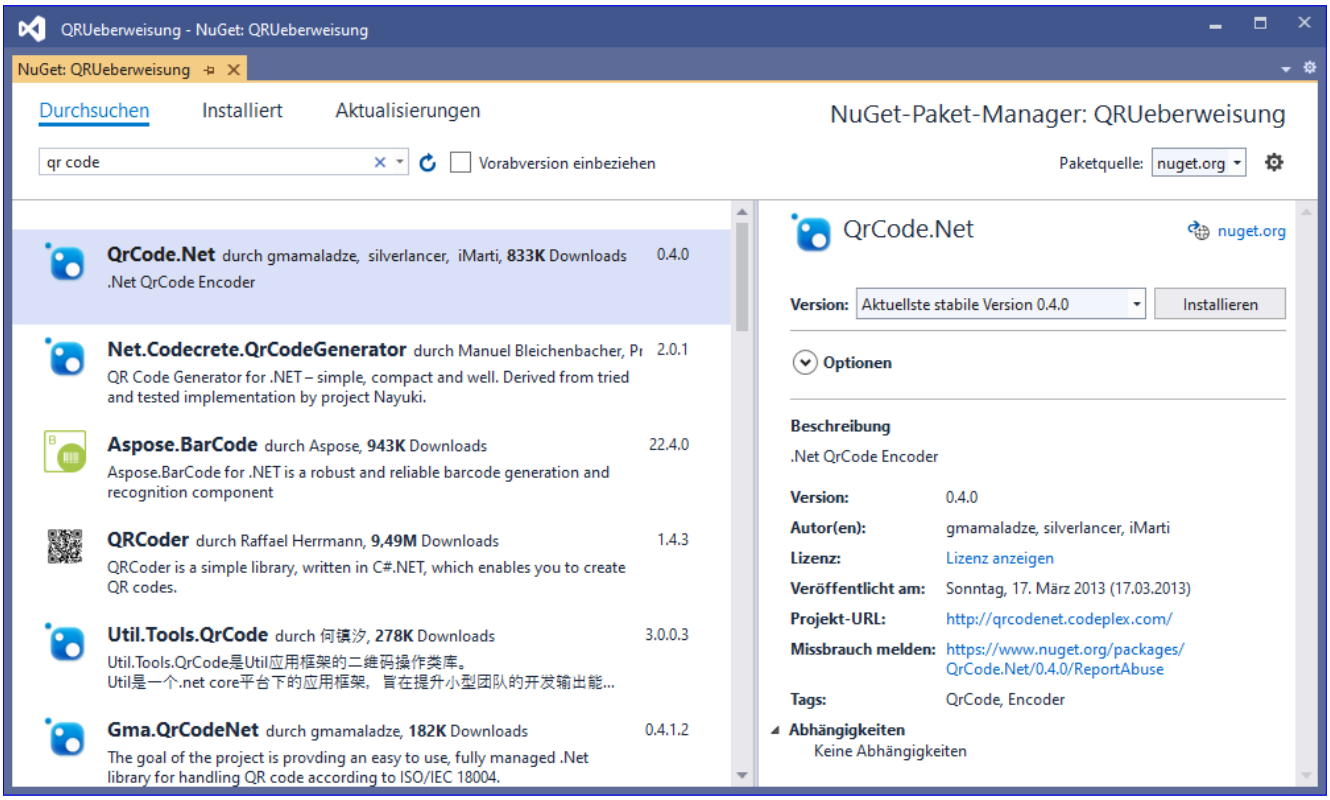


Bild 2: NuGet-Paket zum Erstellen von QR-Codes hinzufügen

Außerdem möchten wir zu Testzwecken schnell ein paar Testdaten in die Textfelder füllen. Das Beispielformular soll schließlich wie in Bild 3 aussehen.

Schließlich wollen wir noch eine Schaltfläche bereitstellen, mit welcher der Benutzer eine Datei mit dem erzeugten EPC-QR-Code auf der Festplatte speichern können soll.

Bild 3: Beispielfenster zum Eingeben und Erstellen von EPC-QR-Codes

Der XAML-Code für das Fenster sieht wie folgt aus, wobei wir hier für eine hübschere Optik ein Icon namens **qr_code.ico** hinzugefügt haben, das im Projekt in einem Ordner namens **images** gespeichert ist:

```
<Window x:Class="MainWindow" ... Title="Beispiel EPC-QR-Code" Height="350" Width="600" WindowStartupLocation="CenterScreen" Icon="Images/qr_code.ico">
```

Damit wir einige Einstellungen nicht für jedes Textfeld und jeden Button definieren müssen, haben wir diese im Bereich **Window.Resources** definiert:

```
<Window.Resources>
    <Style TargetType="TextBox">
        <Setter Property="Margin" Value="5"></Setter>
        <Setter Property="Padding" Value="5"></Setter>
        <Setter Property="HorizontalAlignment" Value="Left"></Setter>
    </Style>
    <Style TargetType="Button">
        <Setter Property="Margin" Value="5"></Setter>
        <Setter Property="Padding" Value="5"></Setter>
        <Setter Property="HorizontalAlignment" Value="Left"></Setter>
    </Style>
</Window.Resources>
```

Die Zeilen und Spalten des Grids haben wir wie folgt definiert, um die Steuerelemente strukturiert anlegen zu können:

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"></RowDefinition>
        <RowDefinition Height="Auto"></RowDefinition>
        <RowDefinition Height="Auto"></RowDefinition>
        <RowDefinition Height="Auto"></RowDefinition>
        <RowDefinition Height="Auto"></RowDefinition>
        <RowDefinition Height="Auto"></RowDefinition>
        <RowDefinition Height="*"></RowDefinition>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto"></ColumnDefinition>
        <ColumnDefinition Width="Auto"></ColumnDefinition>
        <ColumnDefinition Width="Auto"></ColumnDefinition>
        <ColumnDefinition Width="*"></ColumnDefinition>
    </Grid.ColumnDefinitions>
```

Steuerelemente des Fensters

Danach folgen bereits die Definitionen für die Steuerelemente.

Das erste ist das Eingabefeld für den Bank Identifier Code (BIC) und wird samt Bezeichnungsfeld wie folgt definiert, wobei wir die maximale Anzahl Zeichen auf 11 Zeichen begrenzt haben:

```
<Label>BIC:</Label>
<TextBox x:Name="txtBIC" Grid.Column="1" Width="100" MaxLength="11"></TextBox>
```

Das Textfeld **txtZahlungsempfaenger** für den Zahlungsempfänger darf bis zu 70 Zeichen lang sein:

```
<Label Grid.Row="1">Zahlungsempfänger:</Label>
<TextBox x:Name="txtZahlungsempfaenger" Grid.Row="1" Grid.Column="1" Width="250" MaxLength="70"></
TextBox>
```

Die **International Bank Account Number** (**IBAN**) darf maximal 34 Zeichen enthalten und wird in dem wie folgt definierten Textfeld **txtIBAN** eingegeben:

```
<Label Grid.Row="2">IBAN:</Label>
<TextBox x:Name="txtIBAN" Grid.Row="2" Grid.Column="1" Width="250" MaxLength="34"></TextBox>
```

Für die Eingabe des Betrags haben wir eine Validierung eingebaut, die lediglich die Eingabe von Zahlen und maximal einem Komma erlaubt. Die Validierung erfolgt in der Methode **PreviewTextInput** und die Definition des Textfeldes sieht wie folgt aus, die Methode für das **PreviewTextInput**-Attribut schauen wir uns weiter unten an:

```
<Label Grid.Row="3">Betrag:</Label>
<TextBox x:Name="txtBetrag" Grid.Row="3" Grid.Column="1" Width="100"
    PreviewTextInput="txtBetrag_PreviewTextInput"></TextBox>
```

Auch der Verwendungszweck hat eine begrenzte Zeichenzahl, nämlich 140. Hier können Sie allerdings Zeilenumbrüche einfügen, in diesem Fall per Eingabetaste. Damit die Eingabetaste als Zeilenumbruch erkannt wird, stellen wir für das Textfeld **txtVerwendungszweck** das Attribut **AccepptReturns** auf **True** ein. Auch sollen Zeilen, die über die Textfeldbreite hinausragen, umgebrochen werden, was wir durch den Wert **Wrap** für das Attribut **Wrapping** erreichen:

```
<Label Grid.Row="4">Verwendungszweck:</Label>
<TextBox x:Name="txtVerwendungszweck" Grid.Row="4" Grid.Column="1" Height="100" Width="250" Text-
Wrapping="Wrap" AcceptsReturn="True" MaxLength="140"></TextBox>
```

Schaltflächen des Fensters

Damit kommen wir zu den beiden unteren Schaltflächen, die wir in einem **StackPanel**-Element organisieren, das sich in der fünften Zeile befindet und zwei Spalten umfasst. Die erste Schaltfläche heißt **btnErstellen**, die zweite **btnBeispieldaten**. Auch ihre Ereignismethoden stellen wir weiter unten vor:

```
        <StackPanel Orientation="Horizontal" Grid.Row="5" Grid.Column="0" Grid.ColumnSpan="2">
            <Button x:Name="btnErstellen" Click="btnErstellen_Click">QR Code erstellen</Button>
            <Button x:Name="btnBeispieldaten" Click="btnBeispieldaten_Click">Beispieldaten einfügen</Button>
        </StackPanel>
```

Image-Steuerelement und Speichern-Schaltfläche

Schließlich folgen noch Steuerelemente, die einen Bereich einnehmen, der sich in der dritten Spalte über die oberen fünf Zeilen erstreckt, wobei diese wiederum in einem **StackPanel**-Element zusammengefasst werden. Das Image-Steuerelement zur Anzeige des EPC-QR-Codes heißt **imgQRCode** und die Schaltfläche zum Speichern nennen wir **btnSpeichern**:

```
        <StackPanel Orientation="Vertical" Grid.Column="3" Grid.Row="0" Grid.RowSpan="5">
            <Label>QR-Code:</Label>
            <Border BorderBrush="#FFABADB3"  BorderThickness="1" VerticalAlignment="Top"
                    HorizontalAlignment="Left" Margin="5">
                <Image x:Name="imgQRCode" Stretch="None" Margin="5" Width="180" Height="180" ></Image>
            </Border>
            <Button x:Name="btnSpeichern" Click="btnSpeichern_Click">Speichern unter ...</Button>
        </StackPanel>
    </Grid>
</Window>
```

Der Entwurf sieht schließlich wie in Bild 4 aus.

Validierung des Betrags

Damit der Benutzer nur Zahlen und maximal ein Komma in das Feld **txtBetrag** eingeben kann, haben wir eine Ereignismethode erstellt, die durch das Ereignis **PreviewTextInput** ausgelöst wird, also bevor der eingegebene Text im Textfeld erscheint.

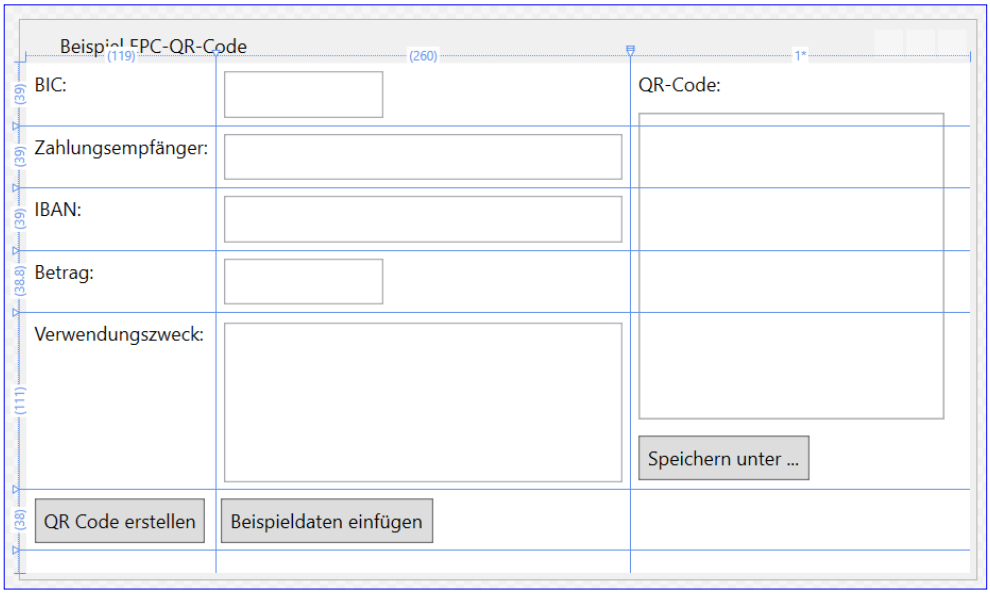


Bild 4: Entwurf des Fensters

Die Methode referenziert als Erstes den mit dem Parameter **sender** gelie-

EPC-QR-Code per DLL

Das Bezahlen von Rechnungen macht heutzutage noch weniger Spaß als es ohnehin schon tut – neben der unausweichlichen Verminderung des Kontostandes ist es auch keine Freude, ellenlange IBAN-Folgen und kryptische Verwendungszwecke einzugeben. Wie Ihre Kunden demnächst wenigstens letzteres schwungvoll erledigen können, haben wir im Artikel EPC-QR-Code für Rechnungen (www.datenbankentwickler.net/300) gezeigt. Damit gelingt das Erzeugen eines QR-Codes, der alle notwendigen Informationen enthält, die von den gängigen Smartphone-Apps der Banken problemlos per Kamera eingelesen und in die Felder des Überweisungsformulars füllt. Im genannten Artikel können Sie den QR-Code für eine Rechnung vorbereiten, aber wenn Sie mehrere Rechnungen auf einen Rutsch stellen wollen, dann wollen Sie die Daten beispielsweise aus einer Datenbank direkt per Code verarbeiten und den erstellten QR-Code direkt in eine Rechnung einbetten. Die Voraussetzungen erledigen wir im vorliegenden Artikel, indem wir eine DLL mit allen notwendigen Funktionen zum Erstellen des notwendigen QR-Codes programmieren.

Aufgaben der DLL

Was soll diese DLL erledigen? Grob gefasst soll sie die Daten entgegennehmen, die für eine Überweisung benötigt werden, und auf irgendeine Art und Weise den dann zu erstellenden EPC-QR-Code zurückliefern.

Auf welche Weise könnte das geschehen? Dazu gibt es zum Beispiel die folgenden Möglichkeiten:

- Man übergibt beim Initialisieren der DLL einen Pfad zu einer Datei, in welcher der Code dann gespeichert wird.
- Oder die DLL stellt ein direkt weiterverarbeitbares Objekt mit dem entsprechenden Objekttyp zurück – für die Weiterverarbeitung unter Access beispielsweise als **StdPicture**. Hier wäre zu klären, ob .NET dies kann.

Übergabe der benötigten Daten

Welche Werte wollen wir der DLL zum Verarbeiten im EPC-QR-Code übergeben?

- BIC
- IBAN
- Verwendungszweck
- Empfängername
- Betrag

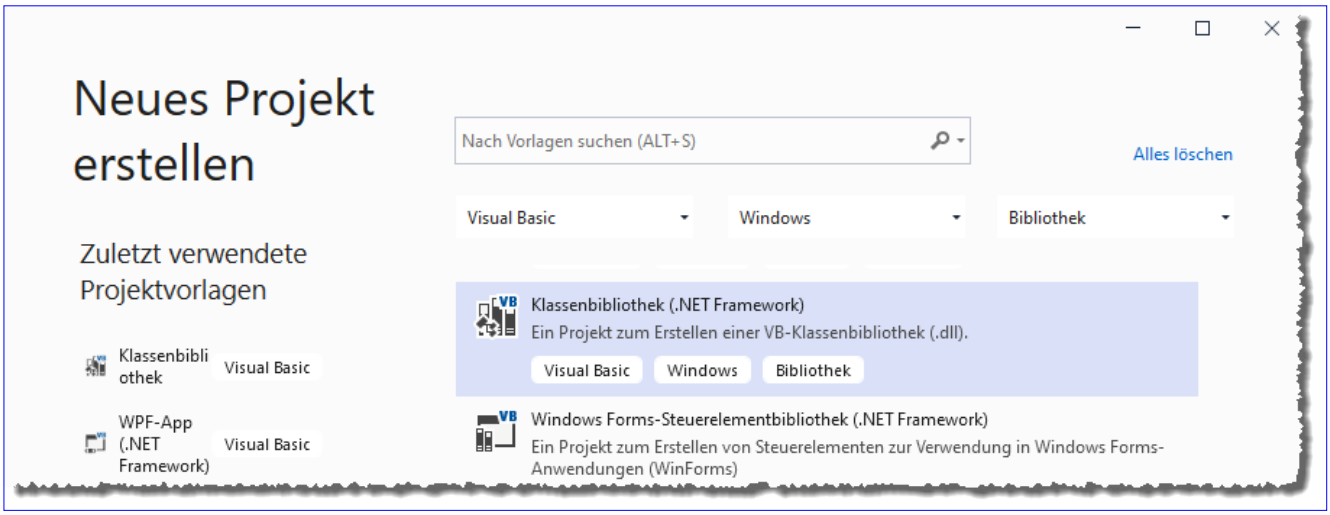


Bild 1: Auswahl des Projekttyps

Art der Übergabe der benötigten Informationen

Wie wollen wir die Informationen an die DLL übergeben? Für diese Informationen können wir entweder Eigenschaften für die in der DLL enthaltenen Klasse, denen wir die Werte zuweisen können, oder wir statten die Funktion, welche den QR-Code erstellt und dann in der einen oder anderen Form zurückliefert, mit den entsprechenden Parametern aus. Wir implementieren einfach beide Varianten, damit der Benutzer die DLL nach seinen eigenen Wünschen einsetzen kann.

Projekt erstellen

Als Erstes erstellen wir ein neues Projekt mit der Vorlage **Klassenbibliothek (.NET Framework)** für die Sprache **Visual Basic** (siehe Bild 1).

Anschließend geben wir als Namen **amvFotoueberweisung** ein und wählen den gewünschten Pfad aus (siehe Bild 2). Hier wählen wir auch das gewünschte Zielframework aus, in diesem Fall **.NET 4.7.2**.

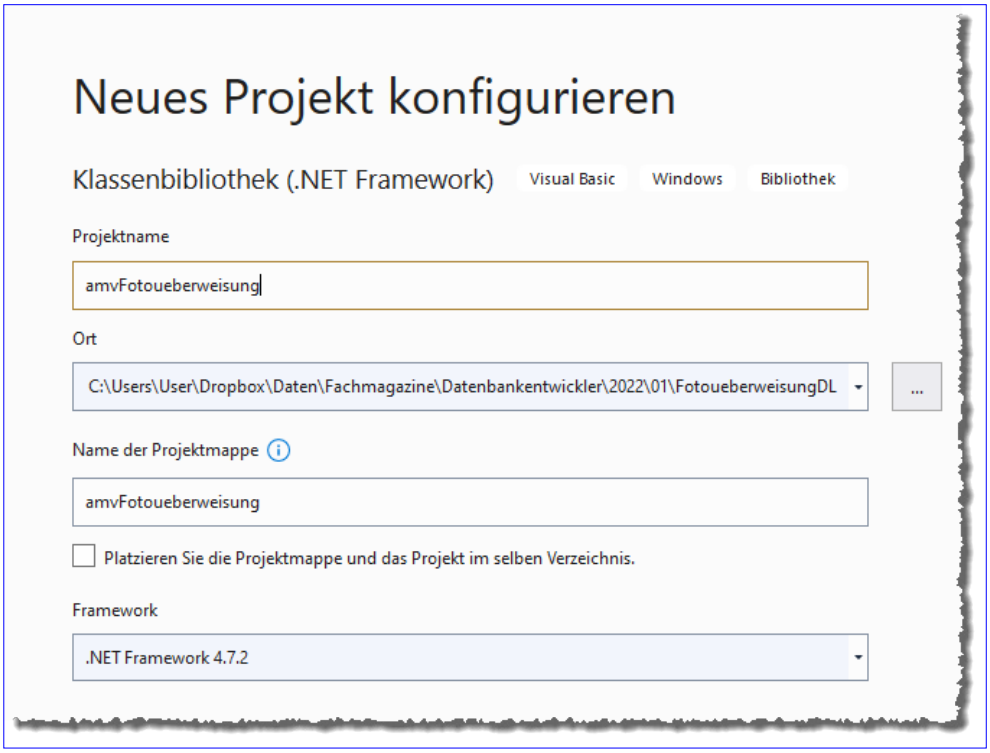


Bild 2: Konfigurieren des neuen Projekts

Klassennamen anpassen

Danach stellen wir den Klassennamen von **Class1.vb** auf **EPCQRCode.vb** um. Die anschließende Nachfrage, ob auch alle Verweise auf die Klasse geändert werden sollen, akzeptieren wir mit einem Klick auf **Ja**.

Öffentliche Eigenschaften hinzufügen

Anschließend erweitern wir die noch leere Klasse um mehrere Elemente, als Erstes um den Import des Namespaces **System.Runtime.InteropServices**:

```
Imports System.Runtime.InteropServices
```

Danach könnten wir bereits die Klasse **EPCQRCode** hinzufügen und für diese die fünf öffentlichen Eigenschaften definieren, die für die Übergabe der für das Erstellen des QR-Codes notwendigen Daten verwendet werden. Wenn wir diese Klasse später im VBA-Editor einer Access-Anwendung referenzieren, zeigt diese Klasse jedoch nicht nur die von uns hinzugefügten Eigenschaften an, sondern auch die Standardeigenschaften einer Klasse wie **GetHashCode**, **GetType** und **ToString**. Diese verwirren in diesem Kontext jedoch mehr als dass sie helfen, also sorgen wir dafür, dass unsere Klasse nur die von uns definierten Member anzeigt.

Dies erreichen wir, indem wir eine öffentliche Schnittstelle namens **IEPCQRCode** definieren, welche nur die gewünschten Eigenschaften vorgibt. Für diese legen wir überdies ein Attribut fest, und zwar **InterfaceType** mit dem Wert **ComInterfaceType.InterfaceIsDual** (dafür benötigen wir den oben genannten **Namespace**-Verweis):

```
<InterfaceType(ComInterfaceType.InterfaceIsDual)>
Public Interface IEPCQRCode
    Property Verwendungszweck As String
    Property BIC As String
    Property IBAN As String
    Property Betrag As <Runtime.InteropServices.MarshalAs( _
        Runtime.InteropServices.UnmanagedType.Currency)> Decimal
    Property Empfaenger As String
End Interface
```

Für den Betrag ist ein kleiner Trick notwendig: Der Datentyp **Decimal** von .NET kann von VBA nicht verarbeitet werden, daher wandeln wir diesen in **Currency** um.

Nun implementieren wir diese Schnittstelle in Form der Klasse **EPCQRCode**, für die wir ebenfalls ein Attribut festlegen, nämlich **ClassInterface** mit dem Wert **ClassInterfaceType.None**. Die Klasse enthält Implementierungen aller in der Schnittstelle festgelegten Eigenschaften, was durch das jeweilige Anhängsel wie **Implements IEPCQRCode.Verwendungszweck** gekennzeichnet wird:

```
<ClassInterface(ClassInterfaceType.None)>
Public Class EPCQRCode
```

```
    Implements IEPCQRCode
    Public Property Verwendungszweck As String Implements IEPCQRCode.Verwendungszweck
    Public Property BIC As String Implements IEPCQRCode.BIC
    Public Property IBAN As String Implements IEPCQRCode.IBAN
    Public Property Betrag As Decimal Implements IEPCQRCode.Betrag
    Public Property Empfaenger As String Implements IEPCQRCode.Empfaenger
End Class
```

Um diese zuerst einmal zu testen, bevor wir die Eigenschaften mit Leben füllen und die Methoden und Funktionen hinzufügen, nehmen wir nun einige weitere Einstellungen am Projekt vor:

- Registrieren für COM-Interop
- Assembly COM-sichtbar machen

Registrieren für COM-Interop

Die Überschrift gibt bereits den Namen der Eigenschaft wieder, die wir aktivieren müssen. Dazu klicken Sie mit der rechten Maustaste auf den Projekteintrag im Projektmappen-Explorer und wählen dort den Eintrag **Eigenschaften** aus.

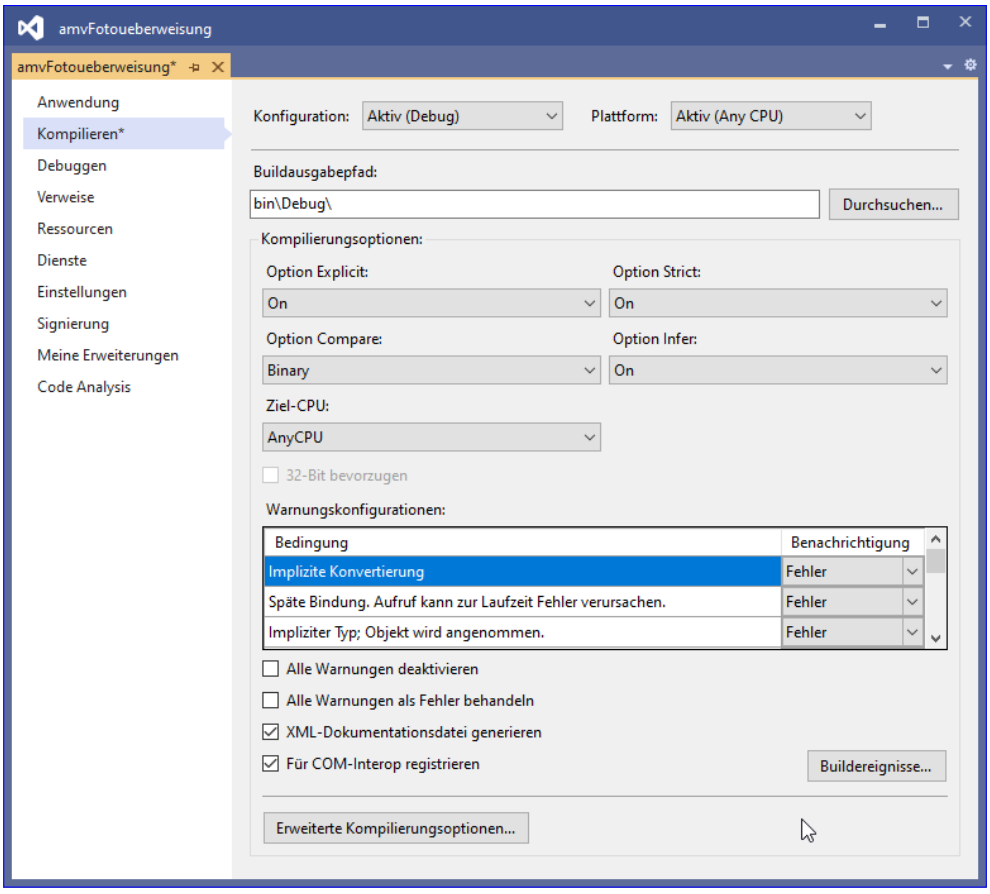


Bild 3: Aktivieren der Eigenschaft **Für COM-Interop registrieren**

Im nun erscheinenden Eigenschaften-Bereich wechseln wir links zum Registerreiter **Kompilieren**. Dort finden wir unten die Eigenschaft **Für COM-Interop registrieren**, die wir nun aktivieren (siehe Bild 3).

Assembly COM-sichtbar machen

Für die zweite wichtige Einstellung wechseln wir links zum Registerreiter **Anwendung**. Hier finden wir die Schaltfläche **As-**

semblyinformationen..., die wir anklicken (siehe Bild 4).

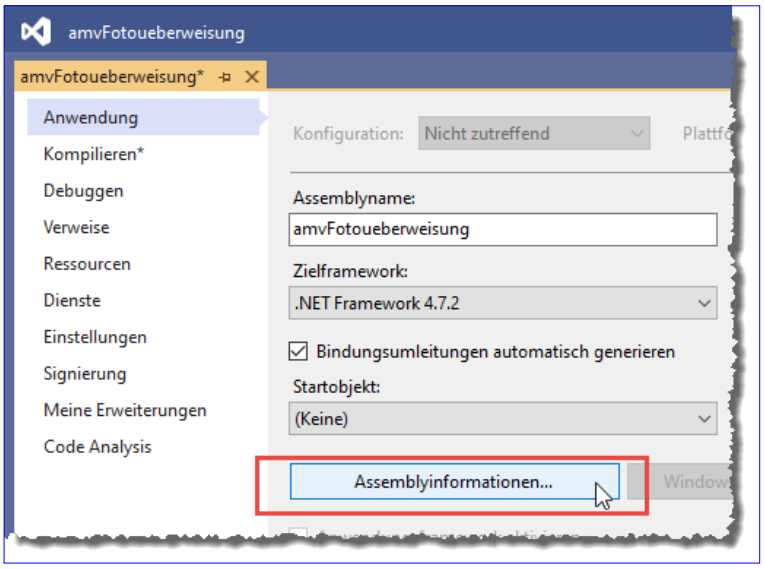


Bild 4: Öffnen der Assemblyinformationen

Im folgenden Dialog **Assemblyinformationen** finden wir einige interessante Einstellungen. Am wichtigsten für die gewünschte Funktion ist hier jedoch die Aktivierung der Option **Assembly COM-sichtbar machen** beziehungsweise die Prüfung, ob diese bereits aktiviert ist (siehe Bild 5).

DLL erstellen

Nun wollen wir die DLL erstmals erstellen. Vor diesem Schritt könnte es möglich sein, dass Sie Visual Studio erneut öffnen müssen – und zwar mit Administratorrechten. Dazu klicken Sie mit der rechtem Maustaste auf den Visual Studio-Eintrag, klicken dort erneut mit der rechten Maustaste zum Beispiel auf **Visual Studio 2019**, wenn das Ihre Version ist, und wählen nun den Eintrag **Als Administrator ausführen** aus dem Kontextmenü aus.

Nun rufen Sie den Menübefehl **Erstellen|Projektmappe erstellen** auf und prüfen die Dateien, die Sie im Windows Explorer im Projektordner unter **bin\Debug** finden. Hier sollte auf jeden Fall eine Datei mit der Dateiendung **.tlb** enthalten sein (siehe Bild 6). Dies ist ein Hinweis darauf, dass beim Erstellen auch die Einträge in der Registry vorgenommen wurden, die für das Erkennen der DLL im VBA-Editor von Office-Anwendungen notwendig ist.

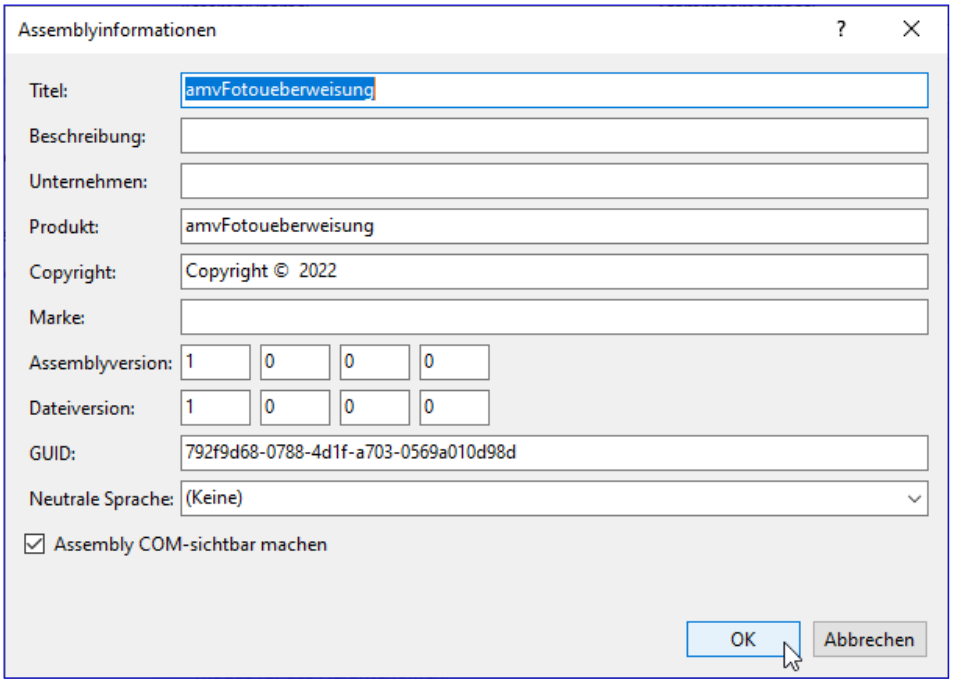


Bild 5: Aktivieren der Option **Assembly COM-sichtbar machen**

DLL unter Access testen

Nun öffnen wir eine beliebige Access-Datenbank und wechseln mit **Alt + F11** zum VBA-Editor. Hier öffnen wir mit dem Menüeintrag **Extras|Verweise** den **Verweise**-Dialog. Suchen

Setup für COM-DLLs mit Inno Setup

Eine COM-DLL mit Visual Studio zu entwickeln ist je nach der Aufgabenstellung schnell erledigt. Ein Klick auf Erstellen und die DLL kann auch schon in die jeweilige Anwendung eingebunden werden. Etwas komplizierter wird es, wenn diese DLL auf einem anderen Rechner installiert werden soll. Dazu reicht es zwar auch aus, die DLL und andere benötigte Dateien auf diesen Rechner zu kopieren und diese mit der App »Regasm.exe« zu registrieren, die auf jedem Rechner vorhanden ist. Allerdings möchte man das dem Kunden nicht unbedingt zumuten. Viel einfacher gelingt dies mit einem Setup, das erstaunlich schnell erstellt ist. Wie Sie ein Setup für die COM-DLL aus dem Artikel EPC-QR-Code per DLL (www.datenbankentwickler.net/301) erstellen, lesen Sie im vorliegenden Artikel.

Wenn Sie eine Lösung auf dem Rechner des Kunden beziehungsweise Benutzers installieren wollen, ist die gute alte **Setup.exe**-Datei immer noch eines der einfachsten Mittel. Wenn man weiß, wie es geht, kann man damit verschiedene Voraussetzungen prüfen, benötigte Verzeichnisse anlegen, Dateien an den gewünschten Ort kopieren und Einträge in die Registry vornehmen, soweit notwendig. Das sind auch gleich die Aufgaben, die wir in diesem Artikel besprechen werden.

Wir gehen an dieser Stelle nicht ausführlich auf Inno Setup ein, sondern zeigen nur, wie Sie das Setup damit erstellen können. Wir nutzen dabei die kostenlose Version von Inno Setup, die Sie unter folgendem Link finden:

```
https://jrsoftware.org/
```

Nachdem Sie Inno Setup installiert haben, können Sie es gleich starten. Der Inno Setup Compiler präsentiert sich dann mit einem Fenster wie in Bild 1. Hier haben wir allerdings bereits den Code für unser Setup eingefügt.

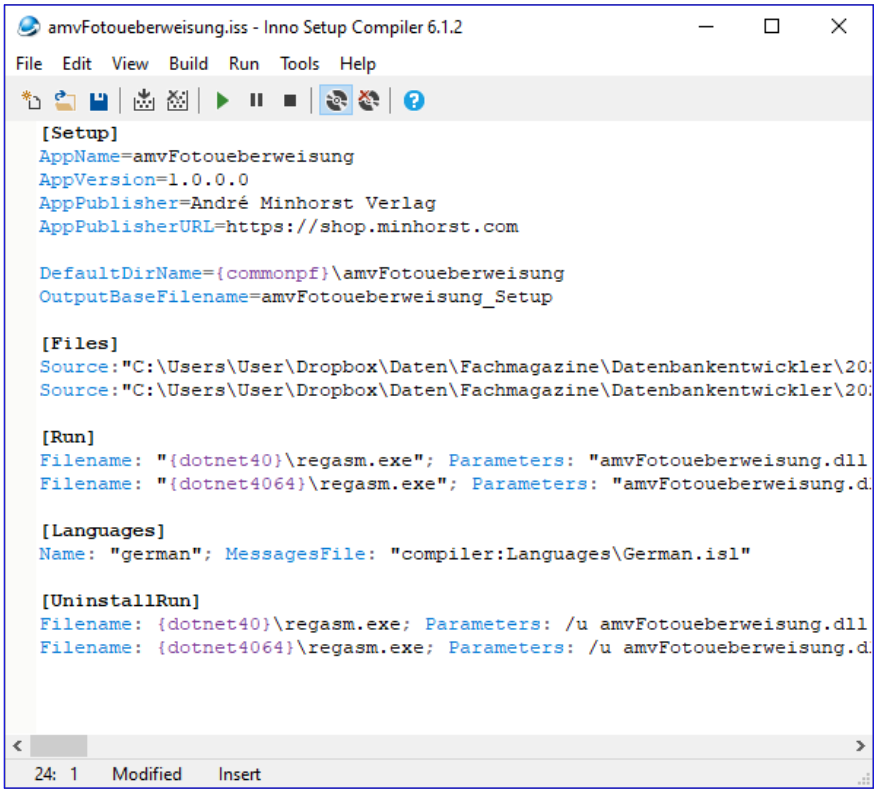


Bild 1: Der Inno Setup Compiler

Code des Setups

Diesen Code schauen wir uns als Nächstes an. Das Setup besteht aus mehreren Bereichen, in diesem Fall

Setup, **Files**, **Run**, **Languages** und **UninstallRun**. Der erste Teil namens **[Setup]** enthält die wichtigsten Informationen, in diesem Fall Name und Version der Anwendung, Name und URL des Herstellers sowie das standardmäßig zu verwendende Verzeichnis für die Installation und den Namen der zu erstellenden Setup-Datei ohne Dateiendung:

```
[Setup]
AppName=amvFotoueberweisung
AppVersion=1.0.0.0
AppPublisher=André Minhorst Verlag
AppPublisherURL=https://shop.minhorst.com

DefaultDirName={commonpf}\amvFotoueberweisung
OutputBaseFilename=amvFotoueberweisung_Setup
```

Danach folgt der Teil **[Files]** mit den Dateien, die dem Setup hinzugefügt werden sollen. Hier geben wir sowohl den Pfad an, unter dem wir die Dateien finden sowie das Zielverzeichnis. Dieses wird hier mit dem Platzhalter **{app}** angegeben, was dem **Programme**-Verzeichnis unter Windows entspricht:

```
[Files]
Source:"C:\...\amvFotoueberweisung.dll"; DestDir: "{app}"
Source:"C:\...\Gma.QrCodeNet.Encoding.dll"; DestDir: "{app}"
```

Der **[Run]**-Teil enthält Aufrufe von Tools, die bei der Installation ausgeführt werden sollen. In diesem Fall rufen wir das Programm **regasm.exe** auf, welches das Erstellen einer **.tlb**-Datei und das Registrieren der DLL durchführt. In Zeiten von 32-Bit und 64-Bit-Windows fragen wir noch ab, welche Variante vorliegt und rufen die entsprechende Version der App **regasm.exe** auf:

```
[Run]
Filename: "{dotnet40}\regasm.exe"; Parameters: "amvFotoueberweisung.dll /codebase /tlb"; WorkingDir:
"{app}"; Flags: runhidden; StatusMsg: "32-Bit-Version wird installiert"; Check: not IsWin64
Filename: "{dotnet4064}\regasm.exe"; Parameters: "amvFotoueberweisung.dll /codebase /tlb"; WorkingDir:
"{app}"; Flags: runhidden; StatusMsg: "64-Bit-Version wird installiert"; Check: IsWin64
```

Unter **[Languages]** stellen wir die Sprache für das Tool ein, hier Deutsch:

```
[Languages]
Name: "german"; MessagesFile: "compiler:Languages\German.isl"
```

Schließlich folgt noch ein Abschnitt namens **[UninstallRun]**, in dem wir hinterlegen, was beim Deinstallieren der DLL geschehen soll. Das Deinstallieren führen Sie übrigens über den Bereich **Programme hinzufügen oder entfernen** der Systemsteuerung aus: