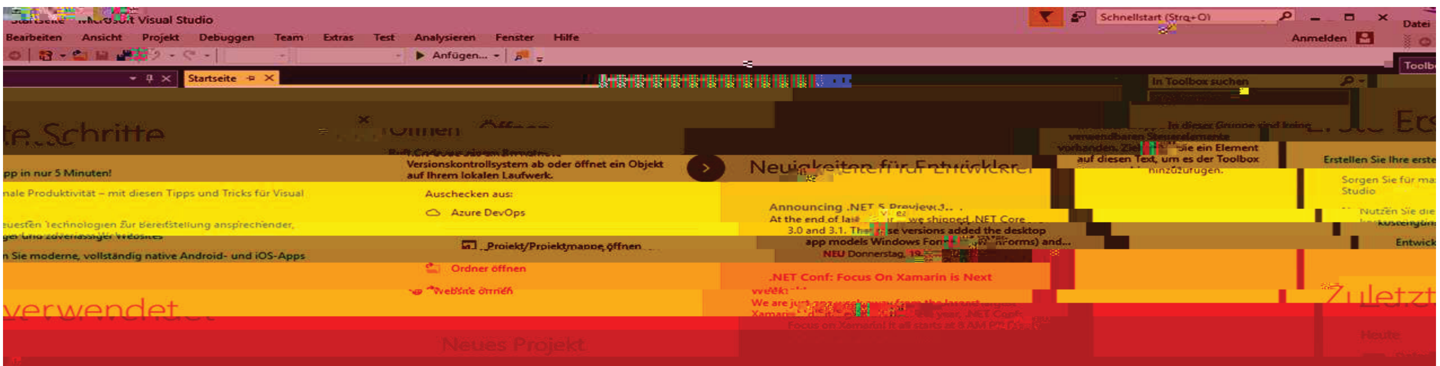
AUSGABE 03/2020

DATENBANK ENTWICKLER

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



TOP-THEMEN:

LÖSUNGEN	Rechnungsverwaltung, Teil 1: Grundlagen	SEITE 3
LÖSUNGEN	Rechnungsverwaltung, Teil 2: Rechnungspositionen	SEITE 23
REPORTING	Rechnungsbericht mit XAML	SEITE 42
SQL SERVER & CO.	Rechnungsverwaltung auf SQLite umstellen	SEITE 56

André Minhorst Verlag

datenbankentwickler.net

Rechnungsverwaltung, Teil 1: Grundlagen

Als selbständiger oder freiberuflicher Softwareentwickler braucht man am Ende vor allem eines: eine Anwendung zum Erstellen von Rechnungen. Diese wollen wir im vorliegenden Artikel programmieren – vom Entwurf des Datenmodells über die Erstellung des Entity Data Models und die Benutzeroberfläche bis zum Ausdrucken der Rechnung als PDF oder mit dem Drucker.

Funktionen der Rechnungsverwaltung

Welche Aufgaben wollen wir mit der Rechnungsverwaltung erledigen – welche Eingangsdaten haben wir, was soll am Ende herauskommen? Und welche Funktionen bilden wir dabei mit der Benutzeroberfläche ab?

Wir wollen eine kleine Anwendung bauen, die verschiedene Funktionen über das Ribbon anbietet. Dazu gehören die Verwaltung von Rechnungsempfängern und der Rechnungen selbst. Für die Rechnungsempfänger wollen wir eine Übersicht der vorhandenen Kunden in Listenform anbieten und ein Fenster zum Anlegen eines neuen Kunden. Die Seite zum Anlegen eines Kunden wollen wir auch für die Detailansicht dieses Kunden nutzen. Praktisch ist dann, wenn wir direkt die Rechnungen, die bisher für diesen Kunden erstellt wurden, in einer Liste in dieser Ansicht anzeigen. Außerdem soll das Fenster jeweils eine Schaltfläche zum Anlegen einer neuen Rechnung enthalten und eine zum Ausdrucken einer markierten Rechnung.

Für die Rechnungen wollen wir ebenfalls ein eigenes Fenster vorsehen. Dieses enthält Steuerelemente zur Eingabe der allgemeinen Daten einer Rechnung, also Rechnungsdatum, Rechnungsbetreff, Rechnungstext und Rechnungsuntertext. Außerdem soll ein Listensteuerelement alle Rechnungspositionen mit Bezeichnung, Nettobetrag, Menge und Mehrwertsteuersatz anzeigen.

Schließlich fehlt noch die Ausgabe der Rechnung – diese wollen wir über ein FlowDocument realisieren, das in einem entsprechenden Steuerelement in einem eigenen Fenster angezeigt wird und ausgedruckt oder als PDF gespeichert werden kann.

Datenmodell der Anwendung

Das Datenmodell haben wir, wenn wir schon im Artikel **Access zu EDM: Dateien erstellen** (**www.datenbankentwickler.net/219**) eine kleine Funktion zum Erstellen eines Entity Data Models programmiert haben, schnell mit Access definiert (siehe Bild 1). Wir verwenden eine Tabelle namens **tblKunden**,

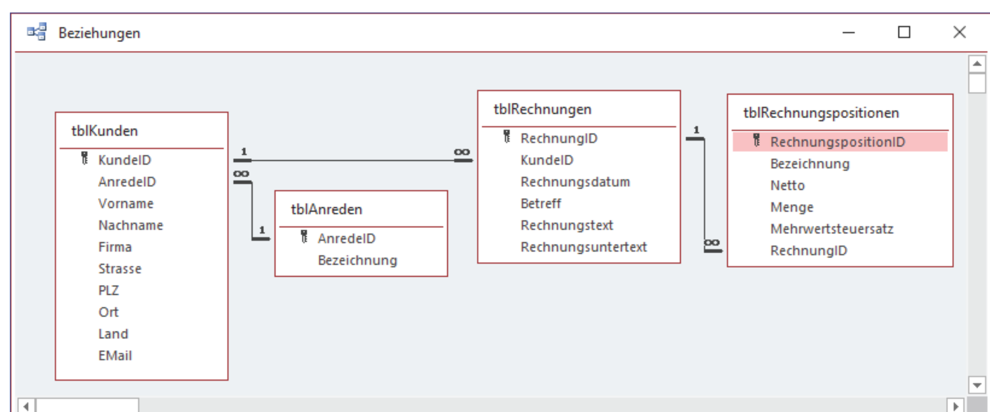


Bild 1: Datenmodell der Rechnungsverwaltung

welche die üblichen Kundendaten enthält. Die Anreden speichern wir in einer eigenen Tabelle namens **tblAnreden**, die von der Tabelle **tblKunden** aus über das Feld **AnredeID** referenziert wird. Die grundlegenden Rechnungsdaten speichern wir in der Tabelle **tblRechnungen**. Hier finden Sie das Rechnungsdatum und die übrigen Informationen, die in der Rechnung rund um die Auflistung der Rechnungspositionen abgebildet werden. Außerdem enthält die Rechnungstabelle ein Fremdschlüsselfeld namens **KundeID**, mit dem der Kunde zur jeweiligen Rechnung festgelegt wird. Schließlich folgen noch die Rechnungspositionen in der Tabelle **tblRechnungspositionen**. Die einzelnen Positionen sind über das Feld **RechnungID** mit der Tabelle **tblRechnungen** verknüpft.

Achtung: Damit das nachfolgende Erstellen des Entity Data Models funktioniert, müssen die Primärschlüsselfelder die Entität im Namen enthalten, bei **Kunde** also beispielsweise **KundeID**.

Entity Data Model erstellen

Mit dem folgenden Aufruf der Funktion **EDMDateienErstellen** legen wir im Unterverzeichnis **RechnungsverwaltungContext** der Access-Datenbank einige Verzeichnisse und Dateien an, die wir gleich im Anschluss weiterverarbeiten:

```
EDMDateienErstellen False, "RechnungsverwaltungContext", "DataModel", True, True
```

Zuvor erstellen wir noch ein neues Visual Studio-Projekt mit der Vorlage **Visual Basic|Windows Desktop|WPF-App** namens **Rechnungsverwaltung**.

Nun öffnen wir mit dem Kontextmenü-Eintrag **Hinzufügen|Neues Element...** des Projekt-Elements im Projektmappen-Explorer den Dialog **Neues Element hinzufügen**, wählen dort das Element **ADO.NET Entity Data Model** aus und geben dafür den Namen **RechnungsverwaltungContext** ein.

Im folgenden Schritt wählen wir den Eintrag **Leeres Code First-Modell** aus (siehe Bild 2).

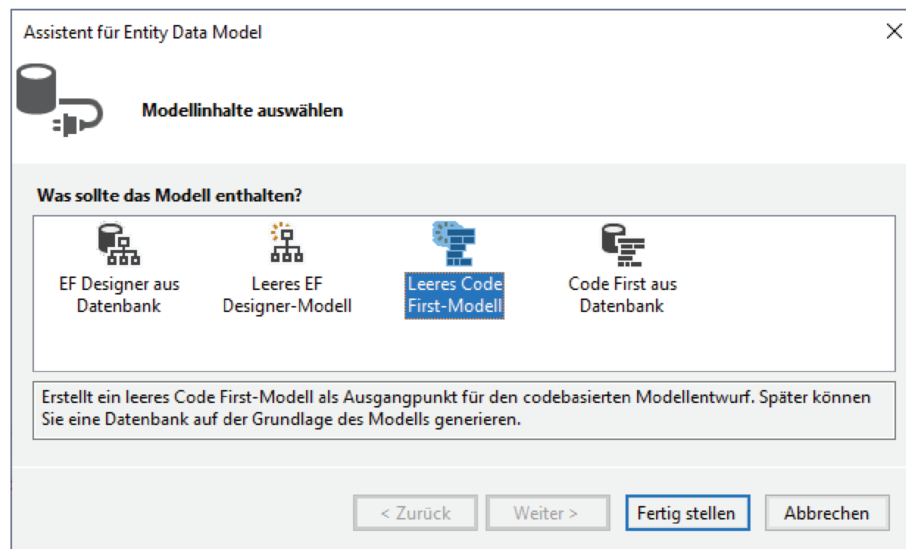


Bild 2: Typ des Modells auswählen

Danach ziehen Sie den Inhalt aus dem soeben von Access aus erstellten Ordner **RechnungsverwaltungContext** (alle enthaltenen Elemente, nicht den Ordner selbst) auf das Projekt-Element **Rechnungsverwaltung**. Dies ersetzt die Datei **RechnungsverwaltungContext.vb**, was sich Visual Studio mit der Meldung aus Bild 3 bestätigen lässt.

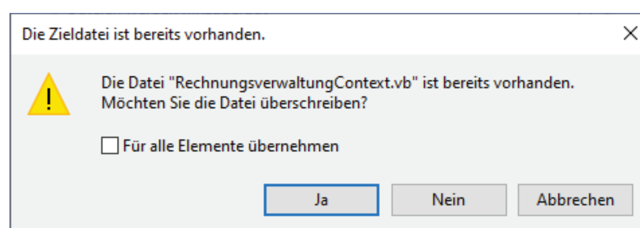


Bild 3: Überschreiben der Datei bestätigen

Mit einer weiteren Meldung bestätigen Sie außerdem, dass die extern geänderten Elemente in Visual Studio neu geladen werden sollen.

Datenbank erstellen

Wenn Sie jetzt noch in der Paket-Manager-Konsole über den Menüeintrag **Extras|Nuget-Paket-Manager|Paket-Manager-Konsole** die folgenden beiden Befehle aufrufen, wird auch noch eine SQL Server-Datenbank auf Basis des Entity Data Models erstellt, die wir im weiteren Verlauf benötigen:

```
add-migration init
update-database
```

Danach finden Sie die Datenbank im SQL Server-Objekt-Explorer vor und können gleichzeitig über das Entity Data Model auf die enthaltenen Daten zugreifen beziehungsweise neue Daten schreiben.

Ribbon der Anwendung

Damit wir die einzelnen Bereiche der Anwendung öffnen können, fügen wir dem Fenster **Main.xaml** ein Ribbon hinzu. Dazu müssen wir zunächst einen Verweis hinzufügen (Menüeintrag **Projekt|Verweis hinzufügen...**), und zwar auf die Bibliothek **System.Windows.Controls.Ribbon** (siehe Bild 4). Danach fügen wir dem Code des Fensters **Main.xaml** die folgende Ribbon-Definition hinzu:

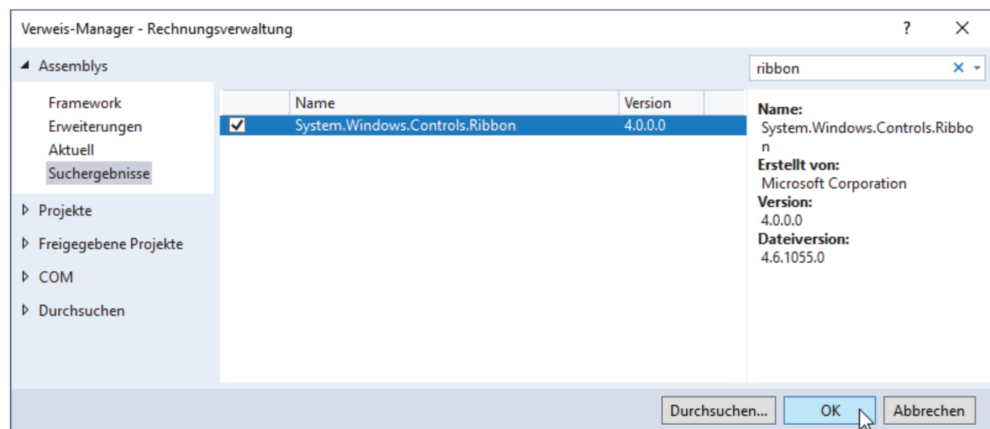


Bild 4: Hinzufügen eines Verweises auf die Bibliothek **System.Windows.Controls.Ribbon**

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"></RowDefinition>
        <RowDefinition Height="*"></RowDefinition>
    </Grid.RowDefinitions>
    <Ribbon>
        <RibbonTab Header="Rechnungsverwaltung">
            <RibbonGroup Header="Kunden">
                <RibbonButton x:Name="btnKundenuebersicht" Label="Übersicht" LargeImageSource="Images/users.png"
                    Click="BtnKundenuebersicht_Click"></RibbonButton>
                <RibbonButton x:Name="btnKundeHinzufuegen" Label="Neuer Kunde" LargeImageSource="Images/user_add.png"
                    Click="BtnKundeHinzufuegen_Click"></RibbonButton>
```

```
            </RibbonGroup>
            <RibbonGroup Header="Rechnungen">
                <RibbonButton x:Name="btnRechnungsuebersicht" Label="Übersicht" LargeImageSource="Images/invoice.png"
                    Click="BtnRechnungsuebersicht_Click"></RibbonButton>
                <RibbonButton x:Name="btnRechnungHinzufuegen" Label="Neue Rechnung"
                    LargeImageSource="Images/invoice_add.png" Click="BtnRechnungHinzufuegen_Click"></RibbonButton>
            </RibbonGroup>
        </RibbonTab>
    </Ribbon>
</Grid>
```

Damit erstellen wir ein Ribbon, das wie in Bild 5 aussieht. Die Ereignismethoden für die Schaltflächen werden automatisch angelegt, wenn Sie im XAML-Code **Click** gefolgt vom Gleichheitszeichen eingeben und dann die Tabulator-Taste betätigen.

Bild 5: Ribbon der Anwendung

Kunden anlegen

Wir beginnen mit der Seite zum Anlegen eines neuen Kunden, da wir ja noch keine Kundendaten in der Datenbank haben – und diese sollten ja vorhanden sein, damit wir Material für die Übersichtsseite der Kunden haben. Wir erstellen also explizit kein neues Fenster, sondern eine Seite (**Page**). Diese fügen wir hinzu, indem wir den Kontextmenü-Befehl des Projekt-Elements im Projektmappen-Explorer betätigen und im Dialog **Neues Element hinzufügen** den Eintrag **Seite (WPF)** auswählen. Die neue Seite soll **Kundendetails** heißen und nicht nur zum Anlegen neuer Kunden, sondern auch zur Anzeige der Details bereits angelegter Kunden dienen. Der XAML-Code sieht in gekürzter Form wie folgt aus:

```
<Page x:Class="Kundendetails" ...>
    <Page.Resources>...</Page.Resources>
    <Grid>
        ...
```

Bei den **TextBox**-Elementen wird der Inhalt des jeweiligen **Kunde**-Elements durch eine Bindung an das Feld **Kunde.ID** erreicht:

```
        <Label Content="ID:" Grid.Column="0" />
        <TextBox x:Name="txtID" Grid.Column="1" HorizontalAlignment="Left"
            Text="{Binding Kunde.ID, Mode=TwoWay}" Width="50" IsEnabled="False" BorderBrush="Transparent" />
```

Die Ausnahme ist hier die das **ComboBox**-Element **cboAnrede**. Es verwendet eine ganze Reihe von Attributen, damit es die Daten der Tabelle **Anreden** anzeigt. **ItemsSource** gibt den Namen der Eigenschaft der Code behind-Datei an, welche die Daten

für das **ComboBox**-Element liefert. **SelectedItem** legt fest, wie der ausgewählte Eintrag selektiert wird. **SelectedValuePath** gibt den Namen des Feldes an, über den das selektierte Element ermittelt wird.

Schließlich gibt das **Binding**-Element im **DataTemplate**-Element noch an, welches Feld der **Anrede**-Objekte im Kombinationsfeld angezeigt wird:

```
<Label Content="ID:" Grid.Column="0" />
<TextBox x:Name="txtID" Grid.Column="1" HorizontalAlignment="Left" Text="{Binding Kunde.ID, Mode=TwoWay}"
    Width="50" IsEnabled="False" BorderBrush="Transparent" />
<Label Content="Anrede:" Grid.Column="0" Grid.Row="1" />
<ComboBox x:Name="cboAnredeID" Grid.Row="1" Grid.Column="1" Width="100"
           ItemsSource="{Binding Anreden}"
           SelectedItem="{Binding Kunde.Anrede, ValidatesOnDataErrors=True}"
           SelectedValuePath="ID" SelectionChanged="Anrede_SelectionChanged">
    <ComboBox.ItemTemplate>
        <DataTemplate>
            <TextBlock>
                <TextBlock.Text>
                    <MultiBinding StringFormat="{}{0}">
                        <Binding Path="Bezeichnung" />
                    </MultiBinding>
                </TextBlock.Text>
            </TextBlock>
        </DataTemplate>
    </ComboBox.ItemTemplate>
</ComboBox>
<Label Content="Vorname:" Grid.Column="0" Grid.Row="2" />
<TextBox x:Name="txtVorname" Grid.Column="1" Grid.Row="2" Width="200"
    Text="{Binding Kunde.Vorname, Mode=TwoWay, ValidatesOnDataErrors=True}" />
<Label Content="Nachname:" Grid.Row="3" />
<TextBox x:Name="txtNachname" Grid.Column="1" Grid.Row="3" Width="200"
    Text="{Binding Kunde.Nachname, Mode=TwoWay, ValidatesOnDataErrors=True}" />
<Label Content="Firma:" Grid.Row="4" />
<TextBox x:Name="txtFirma" Grid.Column="1" Grid.Row="4" Width="200"
    Text="{Binding Kunde.Firma, Mode=TwoWay, ValidatesOnDataErrors=True}" />
<Label Content="Straße:" Grid.Row="5" />
<TextBox x:Name="txtStrasse" Grid.Column="1" Grid.Row="5" Width="200"
    Text="{Binding Kunde.Strasse, Mode=TwoWay, ValidatesOnDataErrors=True}" />
<Label Content="PLZ:" Grid.Row="6" />
<TextBox x:Name="txtPLZ" Grid.Column="1" Grid.Row="6" Width="50"
    Text="{Binding Kunde.PLZ, Mode=TwoWay, ValidatesOnDataErrors=True}" />
```

```
        <Label Content="Ort:" Grid.Row="7" />
        <TextBox x:Name="txtOrt" Grid.Column="1" Grid.Row="7" Width="200"
            Text="{Binding Kunde.Ort, Mode=TwoWay, ValidatesOnDataErrors=True}" />
        <Label Content="Land:" Grid.Row="8" />
        <TextBox x:Name="txtLand" Grid.Column="1" Grid.Row="8" Width="200"
            Text="{Binding Kunde.Land, Mode=TwoWay, ValidatesOnDataErrors=True}" />
        <Label Content="E-Mail:" Grid.Row="9" />
        <TextBox x:Name="txtEMail" Grid.Column="1" Grid.Row="9" Width="200"
            Text="{Binding Kunde.EMail, Mode=TwoWay, ValidatesOnDataErrors=True}" />
        <StackPanel Orientation="Horizontal" Grid.Row="10" Grid.ColumnSpan="4" >
            <Button x:Name="btnSpeichern" Click="btnSpeichern_Click" Content="Speichern">
                <Button.Style>
                    <Style TargetType="{x:Type Button}" BasedOn="{StaticResource {x:Type Button}}">
                        <Style.Triggers>
                            <DataTrigger Binding="{Binding ElementName=cboAnrede,
                                    Path=(Validation.HasError)}" Value="True">
                                <Setter Property="IsEnabled" Value="False"></Setter>
                            </DataTrigger>
                            ...
                        </Style.Triggers>
                    </Style>
                </Button.Style>
            </Button>
            <Button x:Name="btnVerwerfen"
                Click="btnVerwerfen_Click"
                Content="Verwerfen"></Button>
        </StackPanel>
    </Grid>
</Page>
```

Wir betten hier direkt auch die Funktion zum Validieren der Textfelder ein – daher finden Sie im **Binding**-Element auch jeweils die Eigenschaft **ValidationOnDataErrors** mit dem Wert **True**.

Die so erstellte Seite betten wir beim Anklicken des Ribbon-Eintrags **btnKundeAnlegen** über die folgende Methode ein:

```
Private Sub BtnKundeHinzufuegen_Click(sender As
Object, e As RoutedEventArgs)
```

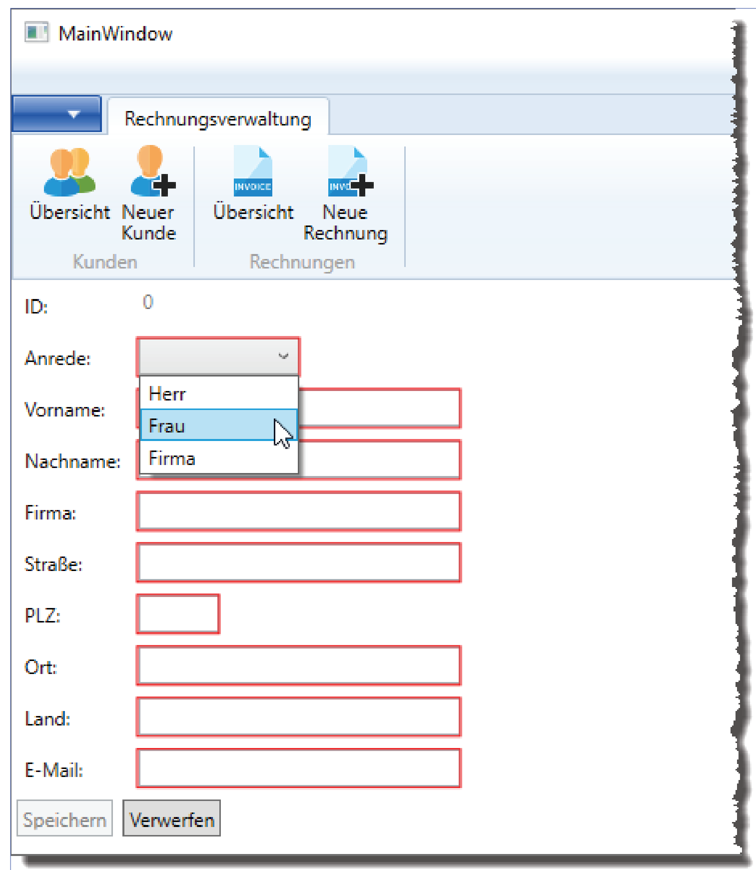


Bild 6: Die Seite **Kundendetails.xaml** im Fenster **Main.xaml**

```
    Dim pge As Kundendetails
    pge = New Kundendetails(Me.WorkZone)
    WorkZone.Content = pge
End Sub
```

Wir erstellen also ein neues Objekt auf Basis des **Page**-Elements **Kundendetails** und übergeben diesem einen Verweis auf das **Frame**-Element **WorkZone**, in dem wir es anzeigen wollen.

Diesen benötigen wir später, um die Seite wieder aus dem **Frame**-Element zu entfernen. Dann weisen wir es dem mit **WorkZone** benannten **Frame**-Element zu, das wir wie folgt im Fenster **Main.xaml** eingefügt haben:

```
<Frame x:Name="WorkZone" Grid.Row="1" NavigationUIVisibility="Hidden" Navigated="WorkZone_Navigated"></Frame>
```

Nach dem Starten des Projekts und einem Klick auf die Ribbon-Schaltfläche **Neuer Kunde** sieht das Fenster wie in Bild 6 aus. Hier wird gerade die Anrede für einen neuen Kunden ausgewählt.

Damit die Daten so wie im Screenshot auf der Seite landen, sind noch einige Zeilen in der Code behind-Datei nötig. Diese sehen wie folgt aus. Die Klasse deklariert zunächst drei Variablen. Die erste namens **_kunde** nimmt den aktuell angezeigten Kunden auf, die zweite namens **_anreden** ist eine Auflistung von Objekten des Typs **Anrede** und die dritte namens **dbContext** referenziert das Context-Objekt:

```
Class Kundendetails
    Private _kunde As Kunde
    Private _anreden As List(Of Anrede)
    Private dbContext As RechnungsverwaltungContext
```

Außerdem benötigen wir noch eine Variable, mit der wir das **Frame**-Objekt referenzieren, in dem das **Page**-Element angezeigt wird:

```
    Private _frame As Frame
```

Die private Variable **_kunde** machen wir über die öffentliche Eigenschaft **Kunde** schreib- und lesbar:

```
    Public Property Kunde As Kunde
        Get
            Return _kunde
        End Get
        Set(value As Kunde)
            _kunde = value
        End Set
    End Property
```

Gleiches erledigen wir für die Auflistung aus der privaten Variablen **_anreden**:

```
Public Property Anreden As List(Of Anrede)
    Get
        Return _anreden
    End Get
    Set(value As List(Of Anrede))
        _anreden = value
    End Set
End Property
```

Es gibt zwei verschiedene Konstruktor-Methoden, die beim Starten abhängig von den übergebenen Parametern aufgerufen werden können. Falls einer der bestehenden Kunden aus der nachfolgend beschriebenen Liste der Kunden aufgerufen wird, übergeben wir dabei den Primärschlüsselwert des jeweiligen Kunden als Parameter. Die Methode **InitializeComponent** erstellt das Fenster auf Basis der **.xaml**-Datei. Die zweite Anweisung schreibt den mit dem Parameter **objFrame** übergebenen Verweis auf das übergeordnete **Frame**-Element in die Variable **_frame**. Die nächste Anweisung initialisiert das Context-Objekt. Dann weisen wir die Code behind-Klasse als Datenquelle für die Bindung der Steuerelemente des **Page**-Elements zu. Und schließlich füllen wir die Eigenschaft **Kunde** mit dem Kunden, der den angegebenen Primärschüsselwert aufweist sowie die Liste **Anreden** mit allen Elementen des **DbSet**-Objekts **Anreden**:

```
Public Sub New(objFrame As Frame, lngKundeID As Long)
    InitializeComponent()
    _frame = objFrame
    dbContext = New RechnungsverwaltungContext
    DataContext = Me
    Anreden = dbContext.Anreden.ToList
    Kunde = dbContext.Kunden.Find(lngKundeID)
End Sub
```

Die zweite Konstruktor-Methode hat keinen zweiten Parameter. Sie wird folglich aufgerufen, wenn wir wie oben auf die Ribbon-Schaltfläche **btnKundeHinzufuegen** klicken. Sie erledigt die gleichen Aufgaben wie die zuvor beschriebene Konstruktor-Methode, holt aber keinen bestehenden Kunden, sondern legt einen neuen an:

```
Public Sub New(objFrame As Frame)
    InitializeComponent()
    _frame = objFrame
    dbContext = New RechnungsverwaltungContext
    DataContext = Me
    Anreden = dbContext.Anreden.ToList
    Kunde = New Kunde
    dbContext.Kunden.Add(Kunde)
```

Rechnungsverwaltung, Teil 2: Rechnungspositionen

Als selbständiger oder freiberuflicher Softwareentwickler braucht man am Ende vor allem eines: eine Anwendung zum Erstellen von Rechnungen. Diese wollen wir im vorliegenden Artikel programmieren – vom Entwurf des Datenmodells über die Erstellung des Entity Data Models und die Benutzeroberfläche bis zum Ausdrucken der Rechnung als PDF oder mit dem Drucker. Im zweiten Teil der Beitragsreihe fügen wir die Verwaltung von Rechnungspositionen hinzu.

Vorüberlegungen

Wenn wir unter Access Rechnungen und Rechnungspositionen verwalten wollten, war das sehr einfach möglich – mit einem Haupt- und einem Unterformular. Einzige Bedingung war, dass der Benutzer den neuen, leeren Datensatz im Hauptformular bereits bearbeitet hatte, damit dieser einen Autowert für sein Primärschlüsselfeld erhalten hat. Damit konnte man dann im Unterformular, dessen Datensätze über das Fremdschlüsselfeld mit dem entsprechenden Datensatz im Hauptformular verknüpft waren, direkt die Bestellpositionen eingeben.

Unter .NET mit WPF und dem Entity Framework ist das ein wenig aufwendiger. In diesem Beispiel wollen wir es so gestalten, dass wir ein Listenfeld zur Anzeige der vorhandenen Rechnungspositionen verwenden, die jeweils zusammen mit den Rechnungsdetails angezeigt werden. Das sieht in der Entwurfsansicht zunächst wie in Bild 1 aus. Von hier aus wollen wir ein weiteres Fenster zur Eingabe von Rechnungspositionen öffnen. Dieses enthält dann jeweils die Daten einer Rechnungsposition. Da diese an eine Rechnung gebunden sein muss, benötigen wir vor dem Speichern einer Rechnungsposition ein Rechnungsobjekt, mit dem diese Rechnung verknüpft ist oder zumindest den Primärschlüsselwert dieser Rechnung. Anderenfalls können wir die Rechnungsposition nicht speichern, da das Fremdschlüsselfeld zum Referenzieren der Rechnung nicht gefüllt wäre.

Bild 1: Erweiterung der Seite zur Anzeige der Rechnungsdetails

Bei Access reichte es aus ein Feld eines Datensatzes mit einem Wert zu füllen, um die Vergabe eines Autowerts für das Primärschlüsselfeld auszulösen. Bei dem darauf folgenden Wechsel in das Unterformular, wurde dieser Datensatz automatisch gespeichert und der Primärschlüssel stand ab sofort als Fremdschlüssel für die Anlage der untergeordneten Datensätze zur Verfügung. Unter WPF existiert dieser Automatismus nicht, weshalb wir das gewünschte Verhalten selber entwickeln müssen.

Wir haben unter anderem die folgenden Möglichkeiten:

- Hinzufügen einer Schaltfläche, mit welcher der Benutzer den Datensatz manuell speichern kann und damit die Steuerelemente zum Bearbeiten der Rechnungspositionen freigibt,

- automatisches Speichern der Rechnungsdaten, wenn der Benutzer die Schaltfläche zum Anlegen einer Rechnungsposition betätigt.

Beides ist ohnehin an die Validierung gebunden: Bevor der Rechnungsdatensatz nicht vollständig gefüllt wurde, kann dieser nicht gespeichert werden. Das bedeutet, dass wir auch die Freigabe der Schaltfläche zum Anlegen einer neuen Position an das vollständige Ausfüllen der Rechnungsdaten binden können.

Wir wollen den zweiten Weg gehen und das Erstellen der ersten Rechnungsposition erst dann freigeben, wenn der Benutzer alle Rechnungsdaten eingegeben hat – und den Rechnungsdatensatz dann automatisch speichern. Mehr dazu weiter unten – erst folgen noch ein paar Vorbereitungen.

Berechnete Felder vorbereiten

Wir wollen für die Rechnungspositionen sowohl den Bruttobetrag des Einzelpreises sowie den gesamten Bruttobetrag ausgeben. Dazu benötigen wir zwei berechnete Felder namens **Brutto** und **BruttoGesamt**. Diese beiden Felder stecken nicht in den zugrunde liegenden Tabellen, sondern sie sollen ausschließlich in der Anwendung berechnet werden.

Dazu könnten wir einfach der Klasse **Rechnungsposition** zwei Eigenschaften hinzufügen, welche die Werte auf Basis der Felder **Netto**, **Mehrwertsteuersatz** und **Menge** ermittelt. Allerdings kann es immer mal passieren, dass wir das Datenmodell ändern und die Klassen auf das neue Datenmodell anpassen wollen. Wenn wir das nicht von Hand erledigen, sondern beispielsweise auf Basis des Access-Datenmodells, wie wir es im ersten Teil der Artikelreihe verwendet haben, werden manuelle Änderungen an den Entitätsklassen jedes Mal überschrieben. Also nutzen wir die Möglichkeit, partielle Klassen anzulegen und erstellen eine neue Klasse namens **Rechnungsposition_Calc**. Diese erhält den folgenden Code:

```
Partial Public Class Rechnungsposition
    Implements INotifyPropertyChanged

    Public ReadOnly Property Brutto As System.Decimal
        Get
            Return _Netto * (1 + _Mehrwertsteuersatz / 100)
        End Get
    End Property
```

```
    Public ReadOnly Property BruttoGesamt As System.Decimal
        Get
            Return _Netto * (1 + _Mehrwertsteuersatz / 100) * Menge
        End Get
    End Property
End Class
```

Die **Property**-Eigenschaften unterscheiden sich in verschiedenen Punkten von denen, die wir in der Klasse **Rechnungsposition.vb** vorfinden:

- Sie liefern nicht die Werte von Feldern der zugrunde liegenden Tabellen zurück, sondern berechnete Felder auf Basis mehrerer Tabellenfelder.
- Sie weisen nur eine **Get**-Property, aber keine **Set**-Property auf. Das ist logisch, denn sie sollen ja nicht vom Benutzer eingestellt werden können, sondern werden nur berechnet.
- Da sie nur Werte zurückliefern, müssen wir das Schlüsselwort **ReadOnly** verwenden.

Die Eigenschaft **Brutto** ermitteln wir als das Produkt des Nettopreises mit dem Wert **1** plus dem Prozentsatz aus **Mehrwertsteuersatz** geteilt durch **100**. Für die Eigenschaft **BruttoGesamt** multiplizieren wir dies noch mit dem Wert des Feldes **Menge**.

Aktualisieren der Felder, wenn die zugrunde liegenden Felder aktualisiert werden

Die Implementierung der Schnittstelle **INotifyPropertyChanged** sorgt in den Entitätsklassen dafür, dass Steuerelemente, die an Felder dieser Klassen gebunden sind, bei Änderungen der Feldinhalte automatisch aktualisiert werden. Das ist bei den berechneten Eigenschaften nicht der Fall, da diese ja keine **Set**-Property und damit auch keine Möglichkeit für den Einbau des Aufrufs von **OnPropertyChanged** bieten.

Hier müssen wir nun doch noch in den Code der automatisch generierten Entitätsklassen eingreifen. Wir fügen den Eigenschaften, auf denen die berechneten Eigenschaften basieren, Aufrufe der Methode **OnPropertyChanged** hinzu, welche nicht nur die Eigenschaft selbst, sondern auch die berechneten Eigenschaften aktualisieren, die auf dieser Eigenschaft aufbauen:

```
Public Property Netto As System.Decimal
    Get
        Return _Netto
    End Get
    Set
        _Netto = Value
        OnPropertyChanged("Netto")
        OnPropertyChanged("Brutto")
        OnPropertyChanged("BruttoGesamt")
```

```
    End Set
End Property
```

Auf die gleiche Weise erweitern wir auch die **Property Set**-Eigenschaften **Menge** und **Mehrwertsteuersatz**. Das Ergebnis: Textfelder, welche die berechneten Felder **Brutto** und **BruttoGesamt** anzeigen, werden nun bei Änderung der Werte der Felder **Netto**, **Anzahl** und **Mehrwertsteuersatz** ebenfalls aktualisiert. Wenn Sie tatsächlich einmal das Datenmodell ändern und die Entitätsklassen neu generieren lassen, müssen Sie diese Aufrufe von **OnPropertyChanged** manuell wiederherstellen.

Steuerelemente zum Verwalten der Rechnungspositionen

Zum Verwalten der Rechnungspositionen wollen wir folgende Steuerelemente hinzufügen:

- Listenfeld zur Anzeige der vorhandenen Rechnungspositionen
- Schaltfläche zum Hinzufügen einer Rechnungsposition
- Schaltfläche zum Löschen einer Rechnungsposition
- Schaltfläche zum Bearbeiten der aktuell markierten Rechnungsposition

Außerdem soll das Listenfeld per Doppelklick auf einen Eintrag erlauben, die Rechnungsposition zu bearbeiten.

Zum Anlegen und Bearbeiten einer Rechnungsposition soll ein Popup-Fenster mit den Details der Rechnungsposition geöffnet werden. Dieses nennen wir **RechnungspositionDetail.xaml**, das im Entwurf wie in Bild 2 aussehen soll.

Bild 2: Entwurf des Fensters zur Eingabe einer Rechnungsposition

RechnungspositionDetail.xaml wollen wir nicht als Seite, sondern als Fenster anlegen. Auf diese Weise ist das Hauptfenster mit der Rechnung, für die Rechnungspositionen angelegt werden sollen, immer noch sichtbar und der Benutzer kann prüfen, welche Rechnungspositionen er bereits angelegt hat.

Damit das Fenster in der Mitte angezeigt wird und nicht irgendwo, stellen wir in **RechnungspositionenDetail.xaml** für das **Window**-Element das Attribut **WindowStartupLocation** auf **CenterScreen** ein:

```
<Window x:Class="RechnungspositionDetail"...
        Title="RechnungspositionDetail" Height="250" Width="405"
        WindowStartupLocation="CenterScreen">
```

Danach definieren wir die Zeilen und Spalten des Grids:

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

Die Steuerelemente, hier ausschließlich Textfelder, binden wir an Elemente des Objekts **Rechnungsposition** und stellen ein entsprechendes Label-Steuerelement voran:

```
        <Label>Bezeichnung:</Label>
        <TextBox x:Name="txtBezeichnung" Width="300" Grid.Column="1" Text="{Binding Rechnungsposition.Bezeichnung,
Mode=TwoWay, ValidatesOnDataErrors=True}"></TextBox>
```

Für das Feld **Netto** legen wir das Format mit **StringFormat=C** fest und sorgen so für die Anzeige des Betrags mit Euro-Zeichen:

```
        <Label Grid.Row="1">Netto:</Label>
        <TextBox x:Name="txtNetto" Width="100" Grid.Row="1" Grid.Column="1" Text="{Binding Rechnungsposition.Netto,
            StringFormat=C, Mode=TwoWay, ValidatesOnDataErrors=True}"></TextBox>
```

Abhängig von den Ländereinstellungen taucht hier dennoch das Dollar-Zeichen als Währungssymbol auf. Um dies zu ändern, fügen Sie der Anwendung für die Datei **Application.xaml.vb** die folgende Methode hinzu, die beim Starten der Anwendung aufgerufen wird. Außerdem benötigen wir dazu die folgenden beiden **Namespace**-Verweise:

```
Imports System.Globalization
Imports System.Windows.Markup

Class Application
    Protected Overrides Sub OnStartup(ByVal e As StartupEventArgs)
        FrameworkElement.LanguageProperty.OverrideMetadata(GetType(FrameworkElement),
            New FrameworkPropertyMetadata(XmlLanguage.GetLanguage(CultureInfo.CurrentCulture.IetfLanguageTag)))
        MyBase.OnStartup(e)
    End Sub
End Class
```

Das Feld **Menge** benötigt keine weitere Formatierung:

```
<Label Grid.Row="2">Menge:</Label>
<TextBox x:Name="txtMenge" Width="100" Grid.Row="2" Grid.Column="1"
    Text="{Binding Rechnungsposition.Menge, Mode=TwoWay, ValidatesOnDataErrors=True}"></TextBox>
```

Für das Feld **Mehrwertsteuersatz** definieren wir das Format mit **StringFormat={}{0} %**. Die vorgegebenen Formate multiplizieren den Wert jeweils mit 100. Sie müssten also die Prozentzahl für 1% mit 0,01 angeben. Das wäre korrekt und ließe sich ebenfalls abbilden, aber hier wählen wir die einfachere Variante – wir müssen dies nur beim Berechnen des Brutto-Preises berücksichtigen, was wir oben in den berechneten Feldern bereits erledigt haben:

```
<Label Grid.Row="3">MwSt:</Label>
<TextBox x:Name="txtMehrwertsteuersatz" Width="100" Grid.Row="3" Grid.Column="1"
    Text="{Binding Rechnungsposition.Mehrwertsteuersatz, StringFormat={}{0} %, Mode=TwoWay,
ValidatesOnDataErrors=True}"></TextBox>
```

Die beiden Felder **Brutto** und **BruttoGesamt** sind die berechneten Felder. Damit der Benutzer diese Werte nicht direkt ändern kann, deaktivieren wir die entsprechenden Textfelder durch Einstellen des Attributs **IsEnabled** auf den Wert **False**. Davon abgesehen greifen wir wie üblich auf die Felder **Brutto** und **BruttoGesamt** zu:

```
<Label Grid.Row="4">Brutto:</Label>
<TextBox x:Name="txtBrutto" Grid.Row="4" Width="100" Grid.Column="1" IsEnabled="False"
    Text="{Binding Rechnungsposition.Brutto, Mode=OneWay, StringFormat=C}"></TextBox>
<Label Grid.Row="5">Brutto gesamt:</Label>
<TextBox x:Name="txtBruttoGesamt" Grid.Row="5" Width="100" Grid.Column="1" IsEnabled="False"
    Text="{Binding Rechnungsposition.BruttoGesamt, Mode=OneWay, StringFormat=C}"></TextBox>
```

Die beiden Schaltflächen zum Speichern und Verwerfen fassen wir in ein **StackPanel**-Element ein. Die Schaltfläche **btnSpeichern** soll nur aktiviert werden, wenn die Validierungen für die einzugebenden gebundenen Felder erfolgreich sind. Daher fügen wir wieder einen DataTrigger hinzu, der die Validierung der Felder **txtBezeichnung**, **txtNetto**, **txtMenge** und **txtMehrwertsteuer** durchführt und im Falle auch nur eines negativen Ergebnisses die Schaltfläche **btnSpeichern** deaktiviert:

```
<StackPanel Orientation="Horizontal" Grid.Row="6" Grid.ColumnSpan="4">
    <Button x:Name="btnSpeichern" Click="btnSpeichern_Click" Content="Speichern">
        <Button.Style>
            <Style TargetType="{x:Type Button}" BasedOn="{StaticResource {x:Type Button}}">
                <Style.Triggers>
                    <DataTrigger Binding="{Binding ElementName=txtBezeichnung,
                            Path=(Validation.HasError)}" Value="True">
                        <Setter Property="IsEnabled" Value="False"></Setter>
                    </DataTrigger>
```

```
                        <DataTrigger Binding="{Binding ElementName=txtNetto,
                                Path=(Validation.HasError)}" Value="True">
                            <Setter Property="IsEnabled" Value="False"></Setter>
                        </DataTrigger>
                        <DataTrigger Binding="{Binding ElementName=txtMenge,
                                Path=(Validation.HasError)}" Value="True">
                            <Setter Property="IsEnabled" Value="False"></Setter>
                        </DataTrigger>
                        <DataTrigger Binding="{Binding ElementName=txtMehrwertsteuer,
                                Path=(Validation.HasError)}" Value="True">
                            <Setter Property="IsEnabled" Value="False"></Setter>
                        </DataTrigger>
                    </Style.Triggers>
                </Style>
            </Button.Style>
        </Button>
        <Button x:Name="btnVerwerfen" Click="btnVerwerfen_Click" Content="Verwerfen"></Button>
    </StackPanel>
</Grid>
</Window>
```

Code des Fensters RechnungspositionDetail.xaml

Der größte Teil der Funktionen des Fensters wird über die XAML-Definitionen abgedeckt. So bleibt noch das Füllen der anzuzeigenden oder neuen Rechnungsposition. Die dafür notwendigen Elemente werden oben in der Klasse deklariert:

```
Imports System.ComponentModel

Public Class RechnungspositionDetail
    Private dbContext As RechnungsverwaltungContext
    Private _rechnungsposition As Rechnungsposition
```

Zum Bereitstellen der Rechnungsposition nach außen verwenden wir die folgende Property:

```
    Public Property Rechnungsposition As Rechnungsposition
        Get
            Return _rechnungsposition
        End Get
        Set(value As Rechnungsposition)
            _rechnungsposition = value
        End Set
    End Property
```

Rechnungsbericht mit XAML

Wenn Sie einen Rechnungsbericht erstellen wollen, ergeben sich unter .NET zahlreiche Möglichkeiten. Leider ziehen die meisten davon Kosten nach sich in Form von Lizenzgebühren für professionelle Softwareprodukte. Wir wollen in diesem Artikel zeigen, wie Sie einfache Berichte wie etwa für eine Rechnung mit den Bordmitteln von Visual Studio erstellen können. Dazu programmieren wir ein XAML-Fenster, das genau die Größe einer DIN A4-Seite hat und fügen dieser die Steuerelemente zum Anzeigen der für eine Rechnung wichtigen Informationen hinzu. Schließlich programmieren wir auch noch eine Drucken-Funktion, um den Rechnungsbericht zu Papier zu bringen.

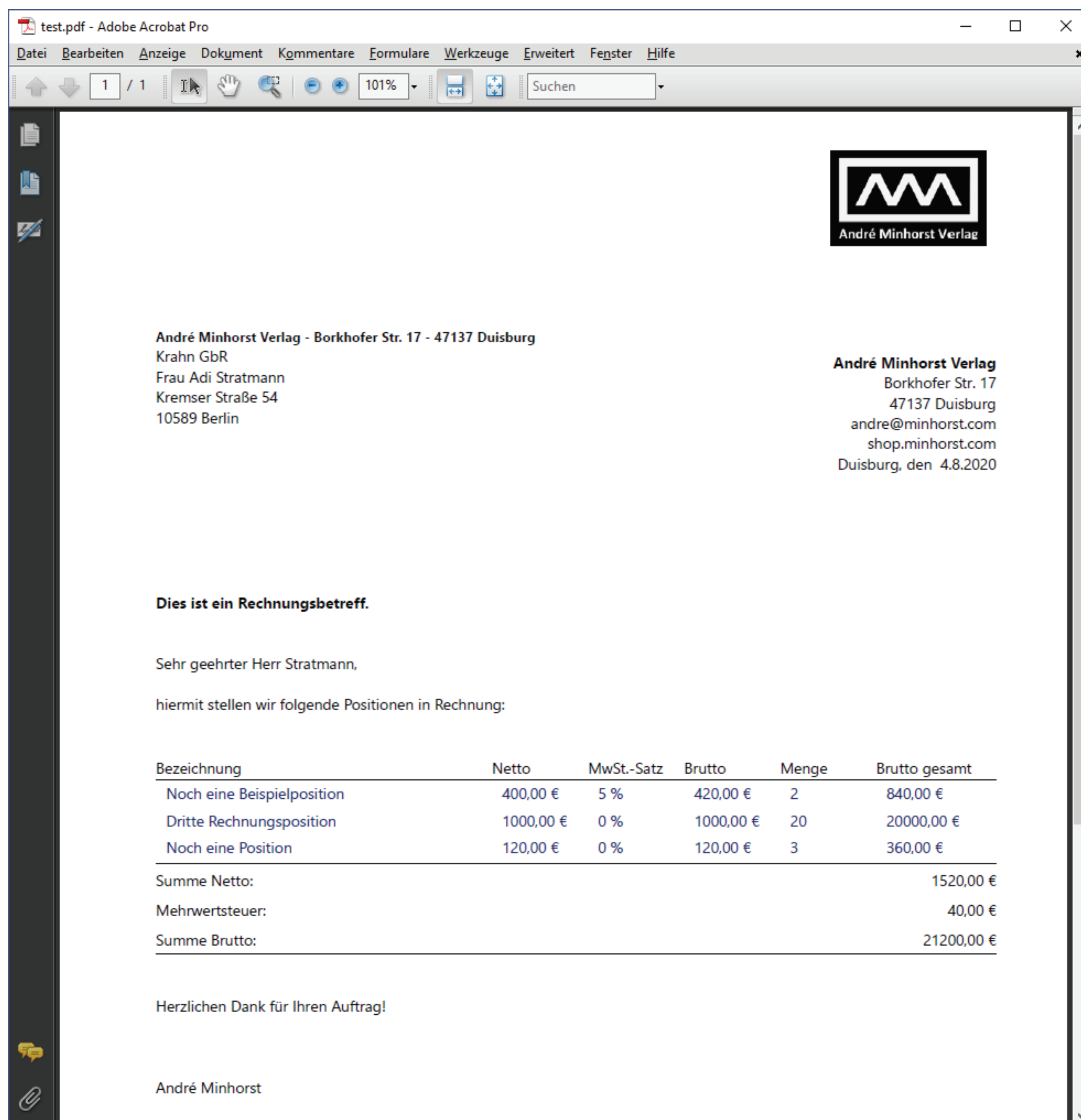
Ziel des Artikels

Wir wollen vorab definieren, wie der gewünschte Rechnungsbericht aussehen soll. Das Ergebnis sehen Sie in Bild 1. Der Bericht soll die folgenden Merkmale enthalten:

- Logo rechts oben
- Adressblock, der sowohl die Absenderadresse in einer Zeile als auch die Empfängeradresse enthält
- Informationen zum Rechnungsschreiben, in der die Absenderadresse nochmals wiederholt wird und in der sich weitere Informationen finden –zum Beispiel das Rechnungsdatum. Das kann man beliebig erweitern – um Rechnungsnummern, Auftragsnummern, Ansprechpartner, Geschäftszeichen et cetera.
- RechnungsbetreffRechnungstext, der einleitet und angibt, worauf sich die nachfolgende Aufstellung bezieht
- Aufstellung der Rechnungspositionen
- Summen der Nettopreise, der Mehrwertsteuer und der Bruttopreise
- Rechnungsuntertext, der eine Floskel sowie eine Grußformel enthält

Einige der Elemente des Berichts sind statisch und werden einmalig festgelegt. Daher ist dieses Dokument nur für die Rechnung einer Person oder eines Unternehmens gedacht. Es ist aber auch möglich, mehrere Vorlagen zu erstellen, von denen der Benutzer dann eine auswählt – darum kümmern wir uns aber nicht in diesem Artikel.

Die dynamischen Elemente sind die Daten, die wir aus der in den Artikeln **Rechnungsverwaltung, Teil 1: Grundlagen** (**www.datenbankentwickler.net/221**) und **Rechnungsverwaltung, Teil 2: Rechnungspositionen** (**www.datenbankentwickler.net/222**) vorgestellt haben. In die dort vorgestellte Anwendung binden wir die Erstellung des Rechnungsberichts auch ein. Dazu fügen wir dem Fenster **Rechnungsdetails.xaml** drei Schaltflächen wie in Bild 2 hinzu. Die Schaltflächen haben die folgenden Funktionen:


test.pdf - Adobe Acrobat Pro

André Minhorst Verlag

André Minhorst Verlag - Borkhofer Str. 17 - 47137 Duisburg
Krahn GbR
Frau Adi Stratmann
Kremser Straße 54
10589 Berlin

André Minhorst Verlag
Borkhofer Str. 17
47137 Duisburg
andre@minhorst.com
shop.minhorst.com
Duisburg, den 4.8.2020

Dies ist ein Rechnungsbetreff.

Sehr geehrter Herr Stratmann,

hiermit stellen wir folgende Positionen in Rechnung:

Bezeichnung	Netto	MwSt.-Satz	Brutto	Menge	Brutto gesamt
Noch eine Beispielposition	400,00 €	5 %	420,00 €	2	840,00 €
Dritte Rechnungsposition	1000,00 €	0 %	1000,00 €	20	20000,00 €
Noch eine Position	120,00 €	0 %	120,00 €	3	360,00 €

Summe Netto:	1520,00 €
Mehrwertsteuer:	40,00 €
Summe Brutto:	21200,00 €

Herzlichen Dank für Ihren Auftrag!

André Minhorst

Bild 1: Der geplante Rechnungsbericht

- **Rechnung XAML anzeigen**: Öffnet die Rechnung als Fenster.
- **Rechnung XAML drucken**: Druckt die Rechnung auf dem Standarddrucker aus.
- **Rechnung XAML drucken mit Dialog**: Öffnet den **Drucken**-Dialog.

Bevor wir uns die Funktion dieser Schaltflächen ansehen, bauen wir die Seite zur Anzeige der Rechnung zusammen.

XAML-Seite als Rechnung

Wir haben eigentlich vorgehabt, den Rechnungsbericht mit dem **FlowDocument**-Element zu programmieren. Das hat allerdings nicht wie gewünscht funktioniert, da dieses Element nicht die Möglichkeit geboten hat, Elemente absolut zu positionieren. Dann haben wir ein wenig mit einem herkömmlichen XAML-Fenster experimentiert und gesehen, dass dieses alle Möglichkeiten bietet, die Elemente einer Rechnung abzubilden – inklusive Datenbindung. Also haben wir diese Variante als Basis für diesen Artikel gewählt.

Mit dem XAML-Fenster können wir, wenn wir es einmal auf eine Größe von 21,7cm Breite und 29cm Höhe gebracht haben, ganz einfach die gewünschten Elemente positionieren. Um das zu erreichen, haben wir einfach die beiden Eigenschaften **Height** und **Width** auf diese Werte eingestellt:

```
<Window x:Class="Rechnungsbericht_XAML"
        ...
        Title="Rechnungsbericht_XAML" Height="29.7cm" Width="21cm">
```

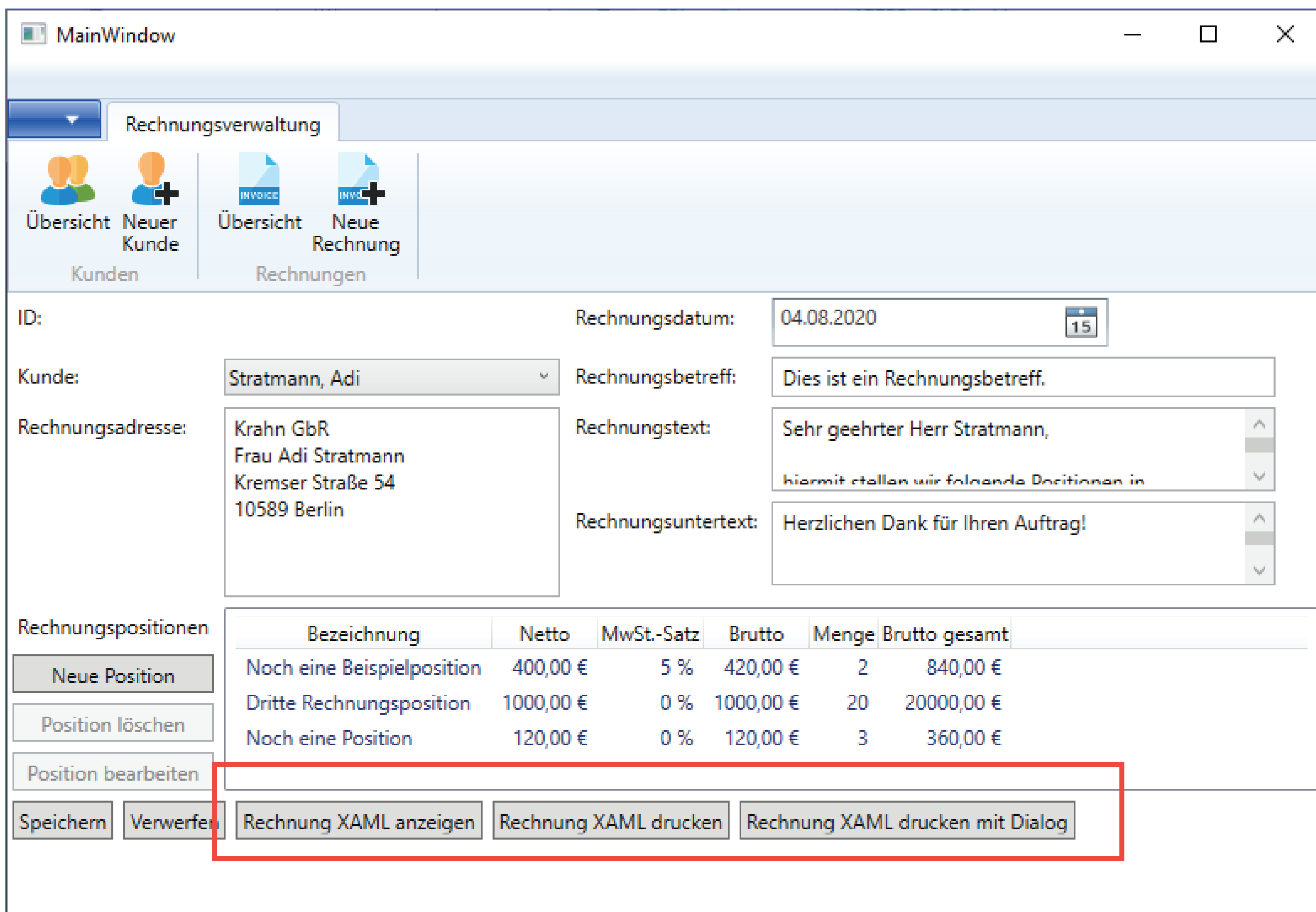


Bild 2: Schaltflächen zum Anzeigen der Rechnung, zum Schnelldrucken und zum Anzeigen des Drucken-Dialogs

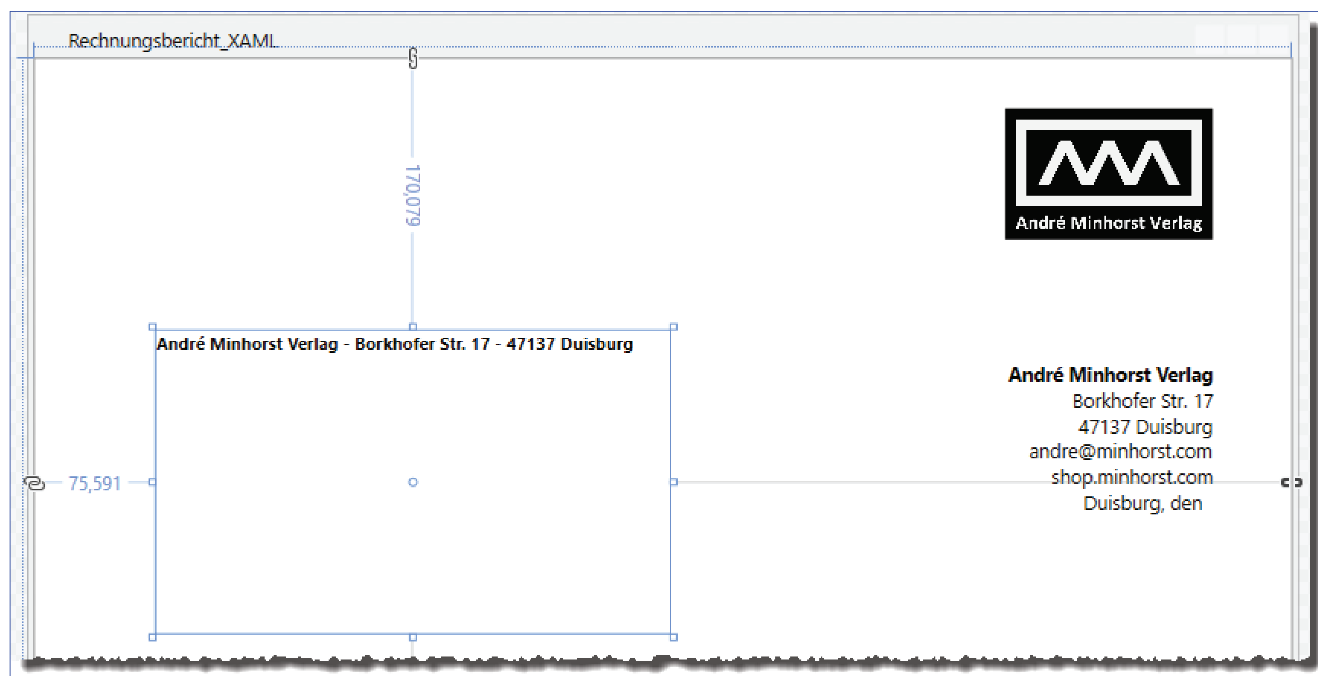


Bild 3: Hinzufügen des Adressblocks

Ein wenig Flow benötigen wir allerdings auch ohne FlowDocument in unserem XAML-Fenster: Wir verwenden ein **ListView**-Steuerelement, um die Rechnungspositionen darzustellen. Rechnungen weisen aber unterschiedlich viele Rechnungspositionen auf, sodass wir an dieser Stelle ein wenig tricksen mussten: Wir haben also die Elemente, die sich unterhalb des **ListView**-Steuerelements befinden und die gegebenenfalls mal mehr, mal weniger weit unten positioniert werden müssen, als Textblöcke in einem **StackPanel**-Element mit vertikaler Orientierung angeordnet, dessen Elemente nach unten geschoben werden, wenn ein darüber befindliches Element vergrößert wird. Doch eins nach dem anderen ...

Hinzufügen des Adressblocks

Um den Adressblock einzufügen, legen wir als erstes Element unterhalb des obligatorischen **Grid**-Elements ein **StackPanel**-Element mit vertikaler Orientierung an (siehe Bild 3).

Dessen Position legen wir über die Eigenschaften **HorizontalAlignment**, **VerticalAlignment**, **Margin**, **Width** und **Height** an. Die Ausrichtung soll links oben sein. Den Abstand vom linken und oberen Seitenrand stellen wir mit den Werten **2cm** und **4.5cm** für die Eigenschaft **Margin** ein. Durch die Eingabe der Zentimeterangaben können wir einfach die Vorgaben etwa der DIN-Norm 5008 umsetzen. Die Breite und die Höhe wollen wir nicht als letzte Werte für die Eigenschaft **Margin** eingeben, da wir diese dann berechnen müssten – wir verwenden stattdessen die Eigenschaften **Height** und **Width**.

Auf diese Weise erhalten wir schon einmal einen Rahmen für unser Adressfeld:

```
<Grid x:Name="GridRechnung">
    <StackPanel Orientation="Vertical" HorizontalAlignment="left" VerticalAlignment="top" Margin="2cm 4.5cm 0cm 0cm"
            Height="5cm" Width="8.5cm">
```

Danach folgt ein **TextBlock**-Element, mit dem wir die Absenderadresse ausgeben wollen. Diese Information ist aktuell hartkodiert, aber wir können diese natürlich auch über eine entsprechende Tabelle verfügbar machen:

```
        <TextBlock FontWeight="Bold" FontSize="8pt">André Minhorst Verlag - Borkhofer Str. 17 - 47137 Duisburg</TextBlock>
```

Unter dieser Zeile wollen wir eine optische Trennung in Form einer Linie unterbringen. Mit den Attributen **X1** und **X2** geben wir die X-Koordinaten der Linie gemessen am übergeordneten Element, hier dem **StackPanel**, an. **Stroke** und **StrokeThickness** legen die Farbe und die Dicke fest:

```
        <Line X1="0cm" X2="8.5cm" Stroke="Black" StrokeThickness="1"></Line>
```

Schließlich folgt noch ein gebundenes **TextBlock**-Element. Dieses liest per Binding die Eigenschaft **Rechnungsadresse** der Rechnung ein, die über das Code behind-Modul des Fensters bereitgestellt wird – dazu später mehr:

```
        <TextBlock Text="{Binding Rechnung.Rechnungsadresse}"></TextBlock>
    </StackPanel>
```

Informationsblock hinzufügen

Den Informationsblock gestalten wir weitgehend frei. Dabei verwenden wir einige fixe Elemente und ein dynamisches – in diesem Fall das Datum, das wir an den Ausdruck **Duisburg, den** anhängen. Die Elemente fügen wir wieder in ein **StackPanel**-Element ein, das wir entsprechend positionieren und dessen Größe wir auf 7 x 5cm anpassen. Darin fügen wir einige

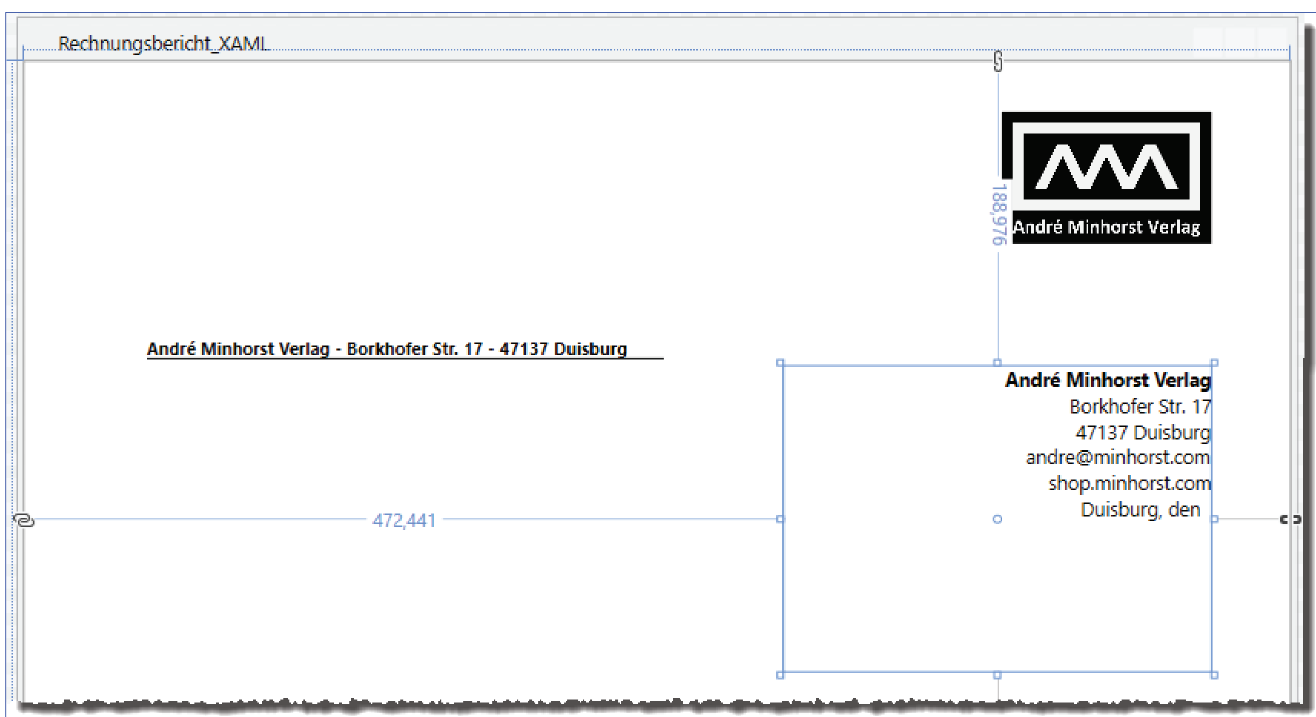


Bild 4: Hinzufügen des Informationsblocks

TextBlock-Elemente ein, von denen jedes einer Zeile des Informationsblocks entspricht. Wir könnten auch ein mehrzeiliges **TextBlock**-Element verwenden, aber dann hätten wir nicht die Möglichkeit, individuelle Formatierungen je Zeile einzufügen. In diesem Fall soll der Firmenname beispielsweise fett dargestellt werden (siehe Bild 4).

```
<StackPanel Orientation="Vertical" HorizontalAlignment="Left" VerticalAlignment="Top" Width="7cm" Height="5cm"
Margin="12.5cm,5cm,0,0">
    <TextBlock FontWeight="Bold" HorizontalAlignment="Right">André Minhorst Verlag</TextBlock>
    <TextBlock HorizontalAlignment="Right">Borkhofer Str. 17</TextBlock>
    <TextBlock HorizontalAlignment="Right">47137 Duisburg</TextBlock>
    <TextBlock HorizontalAlignment="Right">andre@minhorst.com</TextBlock>
    <TextBlock HorizontalAlignment="Right">shop.minhorst.com</TextBlock>
```

Für Teile, die wir aus statischen und dynamischen Texten kombinieren wollen, verwenden wir innerhalb des **TextBlock**-Elements das **Run**-Element. Damit geben wir erst den statischen Text aus und dann den per Binding ermittelten dynamischen Text mit dem angegebenen Format:

```
    <TextBlock HorizontalAlignment="Right">
        <Run Text="Duisburg, den "></Run>
        <Run Text="{Binding Rechnung.Rechnungsdatum, StringFormat='d.M.yyyy'}"></Run>
    </TextBlock>
</StackPanel>
```

Block mit den übrigen Elementen

Die übrigen Elemente starten von einer bestimmten Position beziehungsweise in einem bestimmten Abstand vom oberen Seitenrand aus und bauen dann abhängig von der jeweiligen Höhe aufeinander auf (siehe Bild 5).

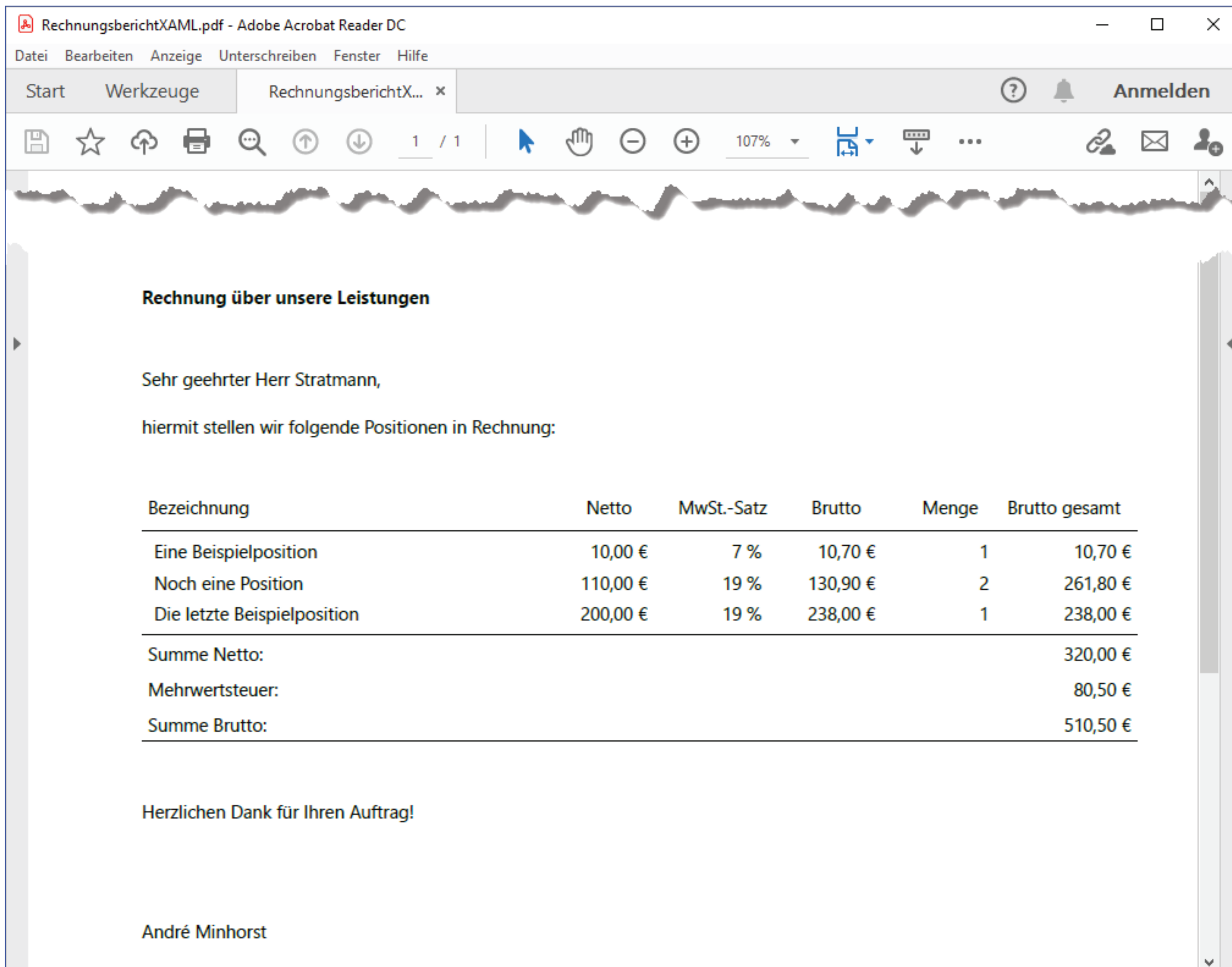
Deshalb fügen wir alle weiteren Elemente in ein weiteres **StackPanel**-Element ein, das wieder die vertikale Ausrichtung verwendet. Das erste im **StackPanel**-Element enthaltene **TextBlock**-Element bindet dabei an das Feld **Rechnung.Betreff** der Code behind-Klasse:

```
<StackPanel Orientation="Vertical" HorizontalAlignment="Left" VerticalAlignment="Top" Margin="2cm 10cm 0cm 0cm">
    <TextBlock FontWeight="Bold" Text="{Binding Rechnung.Betreff}"></TextBlock>
```

Danach folgt ein weiteres **TextBlock**-Element, für das wir über die **Padding**-Eigenschaft einen Abstand von einem Zentimeter zum darüber liegenden Element festgelegt haben. Dieses Element ist an das Element **Rechnung.Rechnungstext** der Code behind-Klasse gebunden und zeigt somit den Rechnungstext an:

```
    <TextBlock Text="{Binding Rechnung.Rechnungstext}" Padding="0 1cm 0 0"></TextBlock>
```

Im gleichen Abstand zum vorherigen Element haben wir das **ListView**-Steuerelement namens **lvwRechnungspositionen** festgelegt. Dieses ist an das Feld **Rechnung.Rechnungspositionen** der Code behind-Klasse gebunden und greift somit auf die



Rechnung über unsere Leistungen

Sehr geehrter Herr Stratmann,

hiermit stellen wir folgende Positionen in Rechnung:

Bezeichnung	Netto	MwSt.-Satz	Brutto	Menge	Brutto gesamt
Eine Beispielposition	10,00 €	7 %	10,70 €	1	10,70 €
Noch eine Position	110,00 €	19 %	130,90 €	2	261,80 €
Die letzte Beispielposition	200,00 €	19 %	238,00 €	1	238,00 €
Summe Netto:					320,00 €
Mehrwertsteuer:					80,50 €
Summe Brutto:					510,50 €

Herzlichen Dank für Ihren Auftrag!

André Minhorst

Bild 5: Der letzte Block mit den übrigen Elementen

Liste der Rechnungspositionen zu. Die Breite haben wir auf 17,5cm festgelegt und den Hintergrund auf **Transparent** eingestellt. Darüberhinaus waren einige Änderungen nötig, um das ListView beziehungsweise die darin angezeigten Daten in druckfähige Form zu überführen. Die Änderungen waren nötig, weil wir erstens keine Bildlaufleisten im **ListView**-Steuerelement haben wollten. Zweitens werden die Spaltenköpfe im **ListView**-Steuerelement in der Regel so formatiert, dass der Benutzer erkennen kann, dass er damit verschiedene Aktionen wie Anpassen der Spaltenbreiten, Verschieben der Spalten et cetera durchführen kann. Wir werden gleich durch Setzen eines geeigneten Attributs die Spaltenköpfe des **ListView**-Steuerelements ausblenden. Dafür fügen wir vor dem **ListView**-Steuerelement ein **StackPanel**-Element mit den Spaltenüberschriften hinzu:

```
<StackPanel Orientation="Horizontal" Margin="0cm 1cm 0cm 0.2cm ">
    <TextBlock Width="7cm" Margin="4,0,4,0">Bezeichnung</TextBlock >
    <TextBlock Width="2cm" TextAlignment="Center">Netto</TextBlock>
    <TextBlock Width="2cm" TextAlignment="Center">MwSt.-Satz</TextBlock>
    <TextBlock Width="2cm" TextAlignment="Center">Brutto</TextBlock>
```

Rechnungsverwaltung auf SQLite umstellen

In der Artikelreihe zum Thema Rechnungsverwaltung haben wir ein Rechnungsverwaltungsprogramm mit SQL Server-Datenbank programmiert. SQL Server ist ein sehr zuverlässiges und gutes System für viele Anwendungen. Wenn wir aber eine kleine Rechnungsverwaltung nutzen wollen, die auch bei Freiberuflern oder Selbständigen ohne SQL Server auf dem Rechner laufen soll, benötigen wir eine Alternative. Hier kommt SQLite ins Spiel: eine sehr leichte Datenbanklösung, die noch nicht einmal eine Installation erfordert. Dieser Artikel zeigt, wir wir eine SQLite-Variante von unserer Rechnungsverwaltung erstellen.

Voraussetzung

Wir gehen an dieser Stelle von der Version der Lösung aus, die wir in den drei Beiträgen **Rechnungsverwaltung, Teil 1: Grundlagen** (**www.datenbankentwickler.net/221**), **Rechnungsverwaltung, Teil 2: Rechnungspositionen** (**www.datenbankentwickler.net/222**) und **Rechnungsbericht mit XAML** (**www.datenbankentwickler.net/223**) erstellt haben.

Ziel der Umstellung

Unsere bisherige Version der Rechnungsverwaltung verwendet den SQL Server als Datenbanksoftware. Diesen muss man zunächst installieren, bevor man eine Anwendung nutzen kann, die eine SQL Server-Datenbank zum Speichern ihrer Daten einsetzt. Für viele Anwendungsfälle ist das sinnvoll – vor allem, wenn ohnehin schon eine SQL Server-Installation vorhanden ist. Falls nicht, kann man auch eine der kostenlosen Varianten des SQL Servers installieren.

Wenn man jedoch eine Anwendung programmiert, die man beispielsweise zum Download für Benutzer mit einfachen Anforderungen anbieten möchte, ist der Aufwand zum Installieren der SQL Server-Software übertrieben hoch. Der Benutzer möchte die Anwendung herunterladen, in wenigen Sekunden bis Minuten installieren und diese dann in Betrieb nehmen.

Für diese Zwecke gibt es wesentlich leichtgewichtigere SQL-Datenbanken – zum Beispiel **SQLite**. Um eine **SQLite**-Datenbank zu nutzen, brauchen Sie nur ein paar Dateien im Verzeichnis Ihrer Anwendung zu installieren – und die eigentliche Datenbank kommt ebenfalls in Form einer handlichen Datei. Deshalb möchten wir dafür sorgen, dass Sie die Rechnungsverwaltung genauso mit SQLite nutzen können wie es uns zuvor mit dem SQL Server gelungen ist.

Umstellen auf SQLite

Die Umstellung nehmen wir nun Schritt für Schritt vor. Um Probleme bezüglich der Kompatibilität zwischen den einzelnen Komponenten zu vermeiden, entfernen wir zunächst alle per NuGet-Paketmanager hinzugefügten Pakete. Dazu öffnen wir den Paket-Manager über den Kontextmenü-Eintrag **NuGet-Pakete verwalten...** des Projekt-Elements im Projektmappen-Explorer.

Hier wechseln Sie, wenn dieser nicht bereits angezeigt wird, zum Bereich **Installiert** und entfernen den einzigen Eintrag **EntityFramework**. Dazu klicken Sie diesen Eintrag an und betätigen dann die **Deinstallieren**-Schaltfläche (siehe Bild 1).

Hinzufügen der benötigten Pakete

Nun wechseln Sie zum Bereich **Durchsuchen** und geben den folgenden Suchbegriff ein:

```
System.Data.SQLite.EF6.
Migrations
```

Dieses Paket erscheint dann in der Ergebnisliste ganz oben. Markieren Sie diesen Eintrag und klicken Sie rechts auf die Schaltfläche **Installieren** (siehe Bild 2).

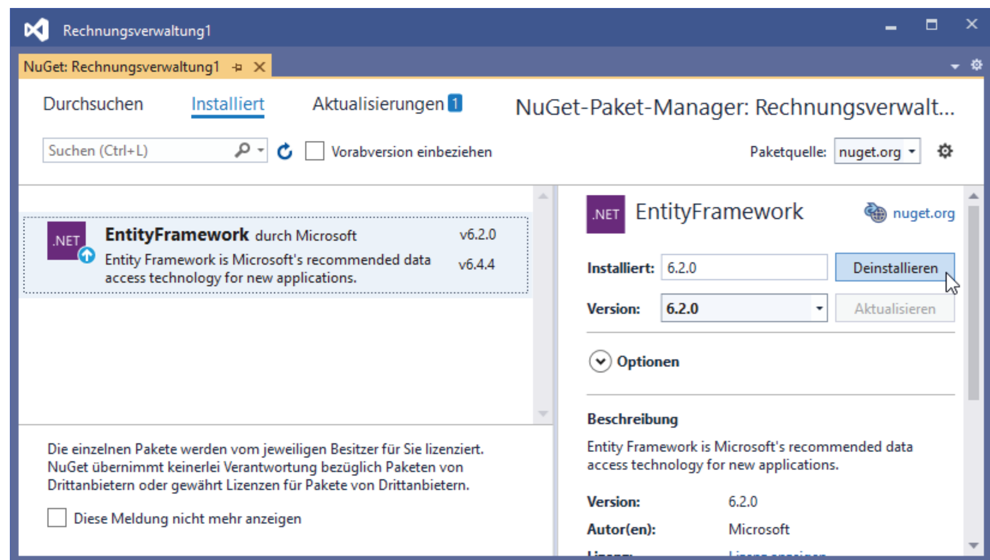


Bild 1: Entfernen des aktuellen **EntityFramework**-Pakets

Der Vorgang dauert einige Sekunden, weil wir hier das unterste Element in der Hierarchie voneinander abhängiger Pakete ausgewählt haben. Wohlwissend, dass die benötigten Elemente, die sich in der Hierarchie über diesem Paket befinden, automatisch mit installiert werden. Zwischenzeitlich akzeptieren Sie die Lizenz für das Element **Apache-2.0**. Danach werden weitere Pakete installiert.

Es kann sein, dass nun einige Fehlermeldungen im Bereich **Fehlerliste** erscheinen. Wenn Sie die Anwendung nun einmal neu erstellen (Menüeintrag **Erstellen|Projektmappe erstellen**) sollten diese jedoch verschwinden. Wechseln Sie nun unter **NuGet** zum Bereich **Installiert** und leeren Sie das Suchfeld, finden Sie die Einträge aus Bild 3 vor.

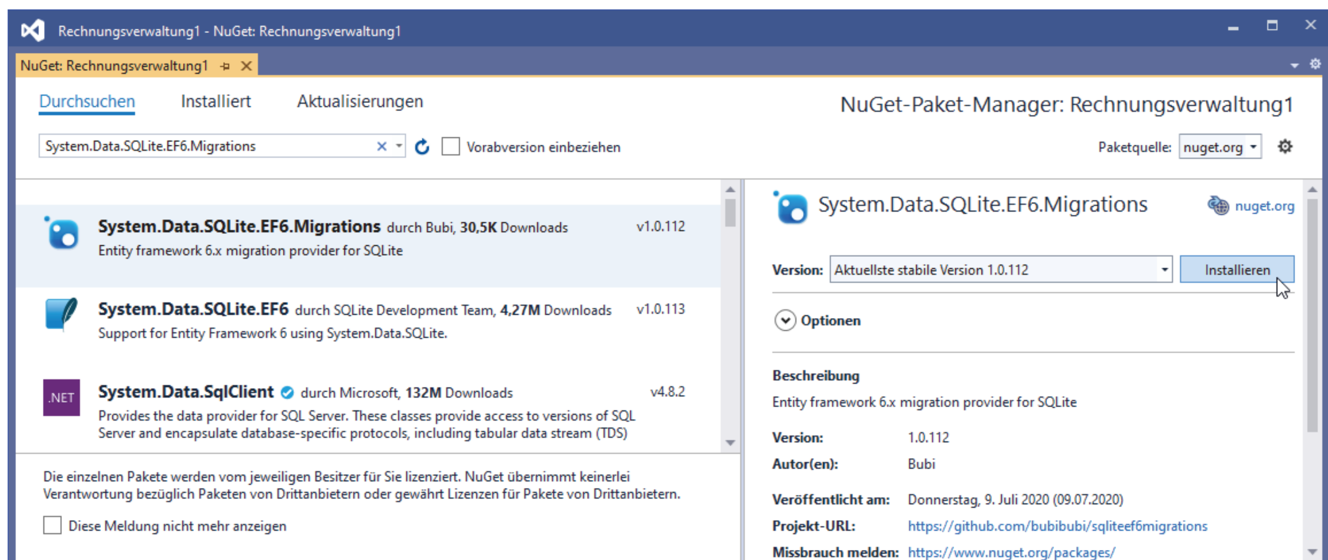


Bild 2: Hinzufügen des Pakets **System.Data.SQLite.EF6.Migrations**

NuGet hat also alle abhängigen Pakete, die für den Einsatz des Pakets **System.Data.SQLite.EF6.Migrations** nötig sind, automatisch hinzugefügt.

Aber warum haben wir überhaupt dieses Paket ausgewählt? Weil bei unseren Experimenten genau dieses Paket fehlt, um den entscheiden Baustein zum Projekt hinzuzufügen – mehr dazu weiter unten.

Änderungen in der Konfigurationsdatei

Wenn Sie sich nun die Datei **App.config** ansehen, die Anwendungseinstellungen im XML-Format bereithält, finden Sie dort zwar noch einige Elemente der unter SQL Server lauffähigen Version der Rechnungsverwaltung, aber durch die Installation der NuGet-Pakete wurden auch einige neue Elemente hinzugefügt.

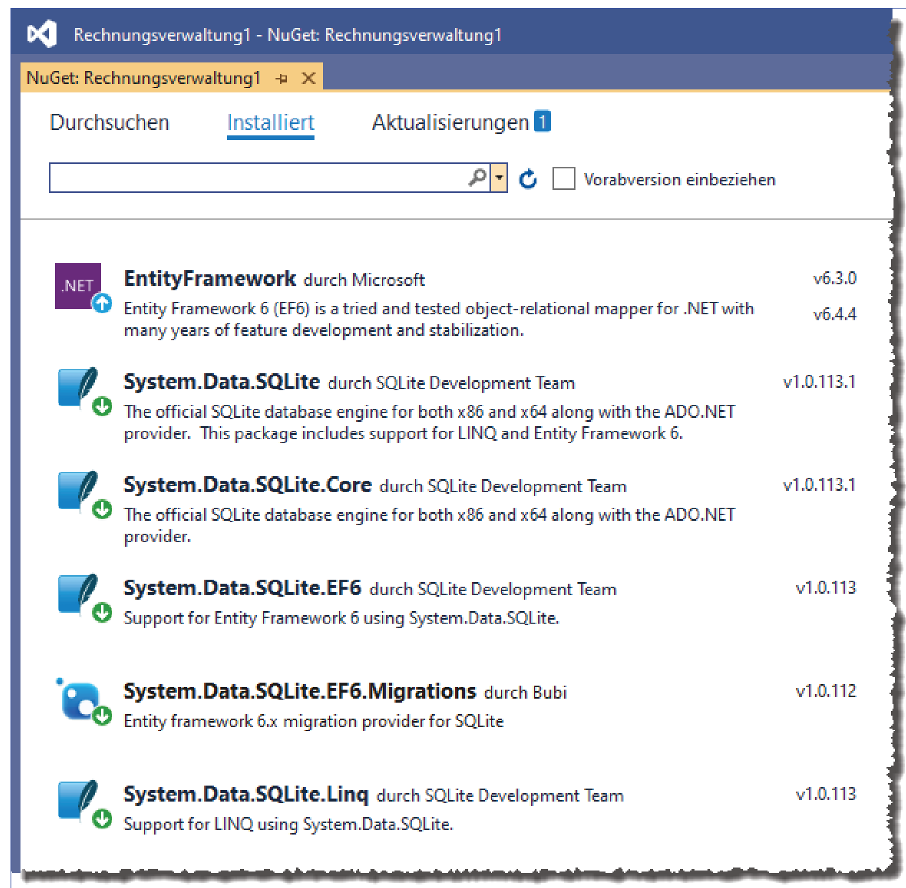


Bild 3: Automatisch mit installierte abhängige Pakete

Allerdings befindet sich dort noch die alte Verbindungzeichenfolge:

```
<connectionStrings>
  <add name="RechnungsverwaltungContext" connectionString="data source=(LocalDb)\MSSQLLocalDB;initial 
catalog=Rechnungsverwaltung1.RechnungsverwaltungContext;integrated security=True;MultipleActiveResultSets=True;App=Entity 
Framework" providerName="System.Data.SqlClient" />
</connectionStrings>
```

Diese ersetzen wir als Erstes durch die neu zu verwendende Verbindungszeichenfolge. Wichtig ist hier, dass Sie den Namen der Context-Klasse für das Attribut **Name** angeben. Für das Attribut **connectionString** brauchen Sie nur den Pfad zu der zu erstellenden Datei anzugeben – wir verwenden hier **Rechnungsverwaltung.sqlite**:

```
<connectionStrings>
  <add name="RechnungsverwaltungContext" connectionString="Data Source=.\Rechnungsverwaltung.sqlite"
      providerName="System.Data.SQLite" />
</connectionStrings>
```

Den Bereich **configurationlentityFramework** erweitern wir dann noch wie folgt um einen Eintrag für **System.Data.SQLite**:

```
<entityFramework>
  <providers>
    <provider invariantName="System.Data.SqlClient" type="System.Data.Entity.SqlServer.SqlProviderServices,
        EntityFramework.SqlServer" />
    <provider invariantName="System.Data.SQLite.EF6" type="System.Data.SQLite.EF6.SQLiteProviderServices,
        System.Data.SQLite.EF6" />
    <provider invariantName="System.Data.SQLite" type="System.Data.SQLite.EF6.SQLiteProviderServices,
        System.Data.SQLite.EF6" />
  </providers>
</entityFramework>
```

Inkompatible Datentypen anpassen

Wenn wir die Anwendung nun starten, erhalten wir die Fehlermeldung aus Bild 4. Dieser verweist darauf, dass kein Datentyp namens **datetime2** unter SQLite gefunden werden konnte. Genau diesen definieren wir aber in der Datei **Rechnung.vb** für das Feld **Rechnungsdatum**:

```
<Column(TypeName:="datetime2")>
Public Property Rechnungsdatum As System.DateTime
...
```

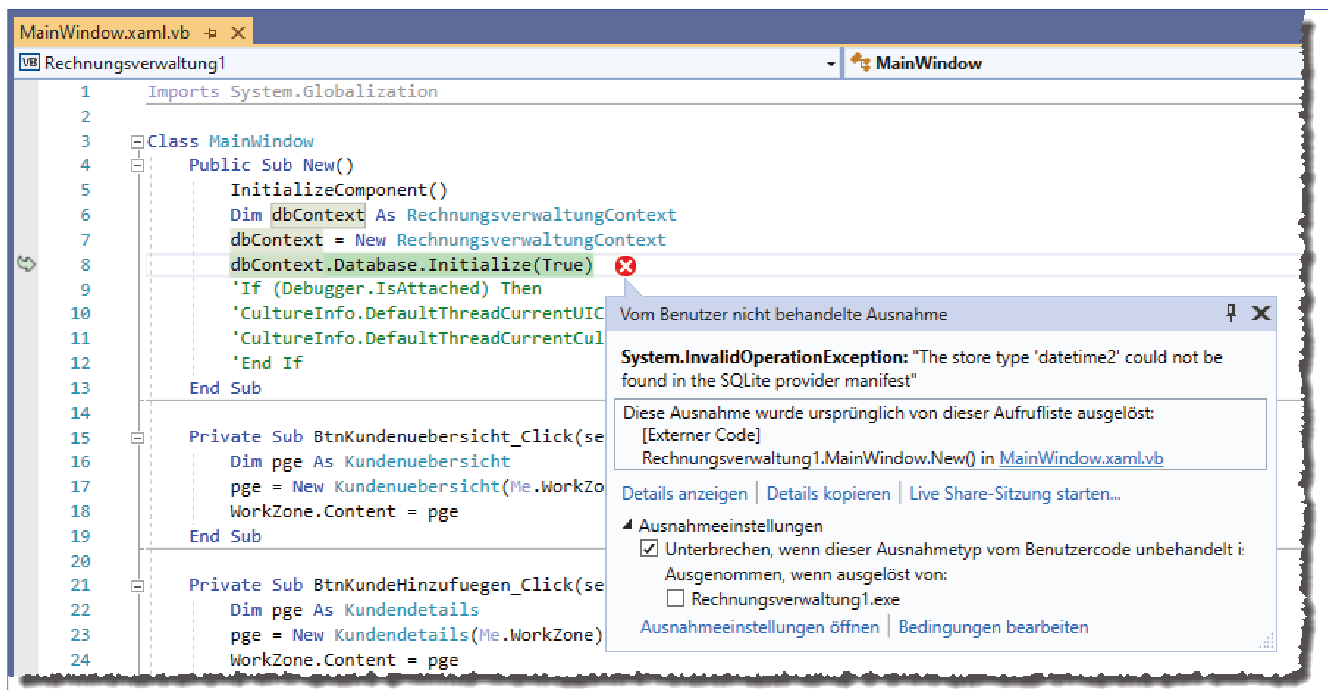


Bild 4: Fehlermeldung beim ersten Start