WPF-Tutorial 1: CdManager

Teil 1: WPF-Basics

Wir erstellen eine einfache CD-Verwaltung. Die Daten werden zur Vereinfachung lediglich von einem File eingelesen und in einer Collection der Repository Klasse gespeichert.

Das Assembly Model ist bereits in der Vorlage vorhanden.

Im Model Assembly sind die Klassen Cd, Track und Repository implementiert. Im Repository wird die Liste der Cds gespeichert. Jede Cd hat einen Artist, Title und eine Liste von Tracks.

```
namespace CdManager.Model
{
    public class Cd
    {
        public string Artist { get; set; }
        public string AlbumTitle { get; set; }
        public List<Track> Tracks { get; set; }

        public Cd()
        {
            Tracks = new List<Track>();
        }
    }
}
```

Ein Track (z.B. ein Song) hat folgende Felder zur Speicherung der Detaildaten:

```
public class Track
{
    public string Title {get;set;}
    public int Year {get;set;}
    public string SongWriter { get; set; }
    public string LeadVocals {get;set;}
}
```

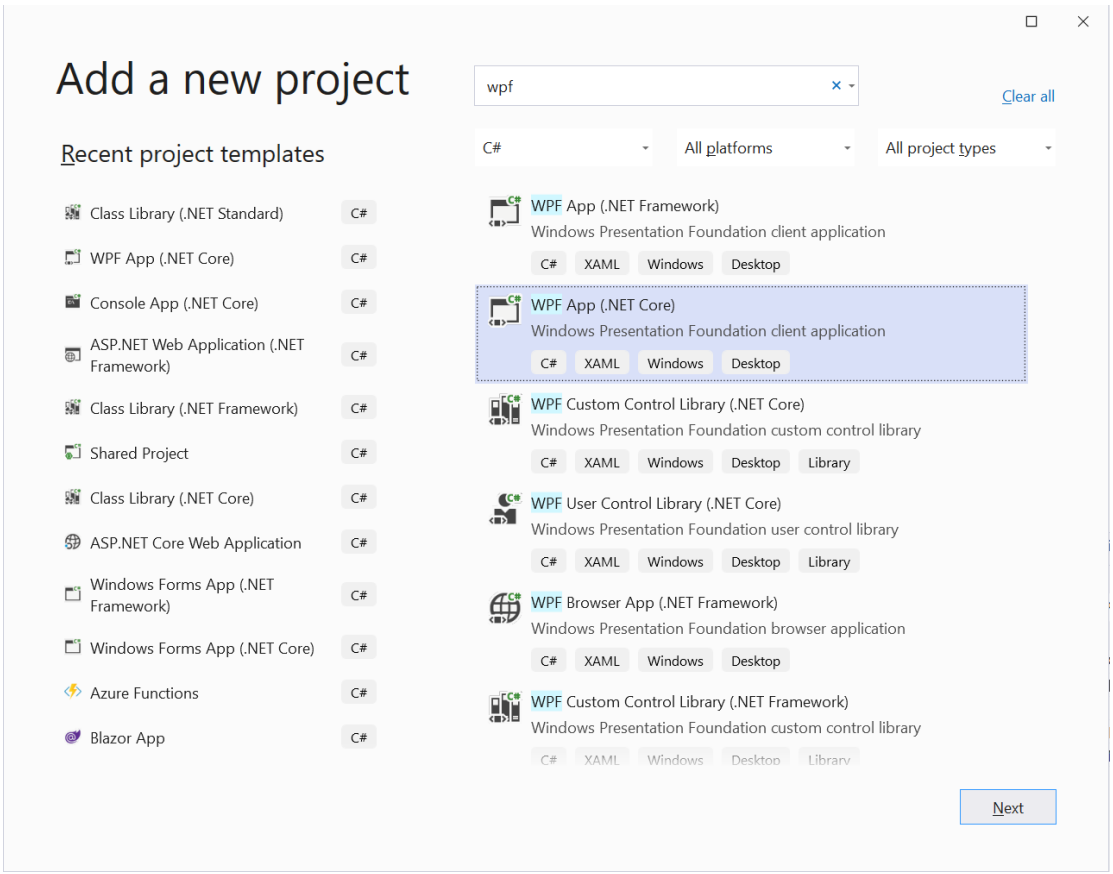
Die Klasse Repository ist als Singleton ausgeführt. In ihr werden im Konstruktor die Beispieldaten aus einem csv-File geladen und diese in einer Collection gespeichert.

Für die Interaktion mit unserem WPF-Projekt ist bereits eine Methode zum Hinzufügen einer Cd (AddCd) sowie eine Methode zum Zurückliefern aller Cds als neue Liste implementiert (GetAllCds):

```
public class Repository
{
    private const string fileName = "AlbumCds.csv";

    private static Repository instance;

    List<Cd> cds;

    private Repository()
    {
        cds = new List<Cd>();
        LoadCdsFromCsv();
    }

    public static Repository GetInstance()
    {
        if (instance == null)
            instance = new Repository();

        return instance;
    }

    private void LoadCdsFromCsv()
    {
        string[][] cdCsv = fileName.ReadStringMatrixFromCsv(true);
        cds = cdCsv.GroupBy(line => new { Artist = line[0], Album = line[3] }).Select(x =>
            new Cd
            {
                AlbumTitle = x.Key.Album,
                Artist = x.Key.Artist,
                Tracks = cdCsv.Where(tr => tr[3] == x.Key.Album).Select(cdTrack =>
                    new Track
                    {
                        Title = cdTrack[1],
                        Year = Convert.ToInt32(cdTrack[2]),
                        SongWriter = cdTrack[4],
                        LeadVocals = cdTrack[5]
                    }).ToList()
            }).ToList();
    }

    public void AddCd(Cd cd)
    {
        cds.Add(cd);
    }

    public List<Cd> GetAllCds()
    {
        return cds.OrderBy(x=>x.AlbumTitle).ToList(); //Erstellt neue Liste!
    }
}
```

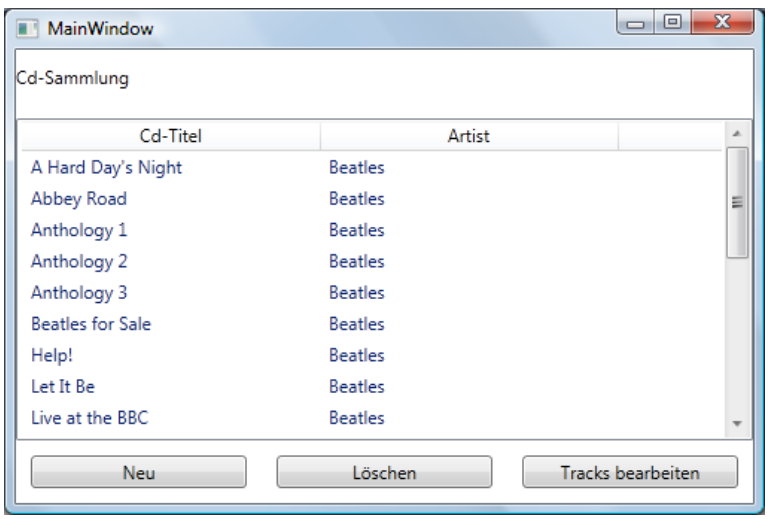
Wpf-Anwendung erstellen

Zunächst fügen wir der Solution ein neues WPF-Projekt hinzu (Name: CdManager.Wpf)



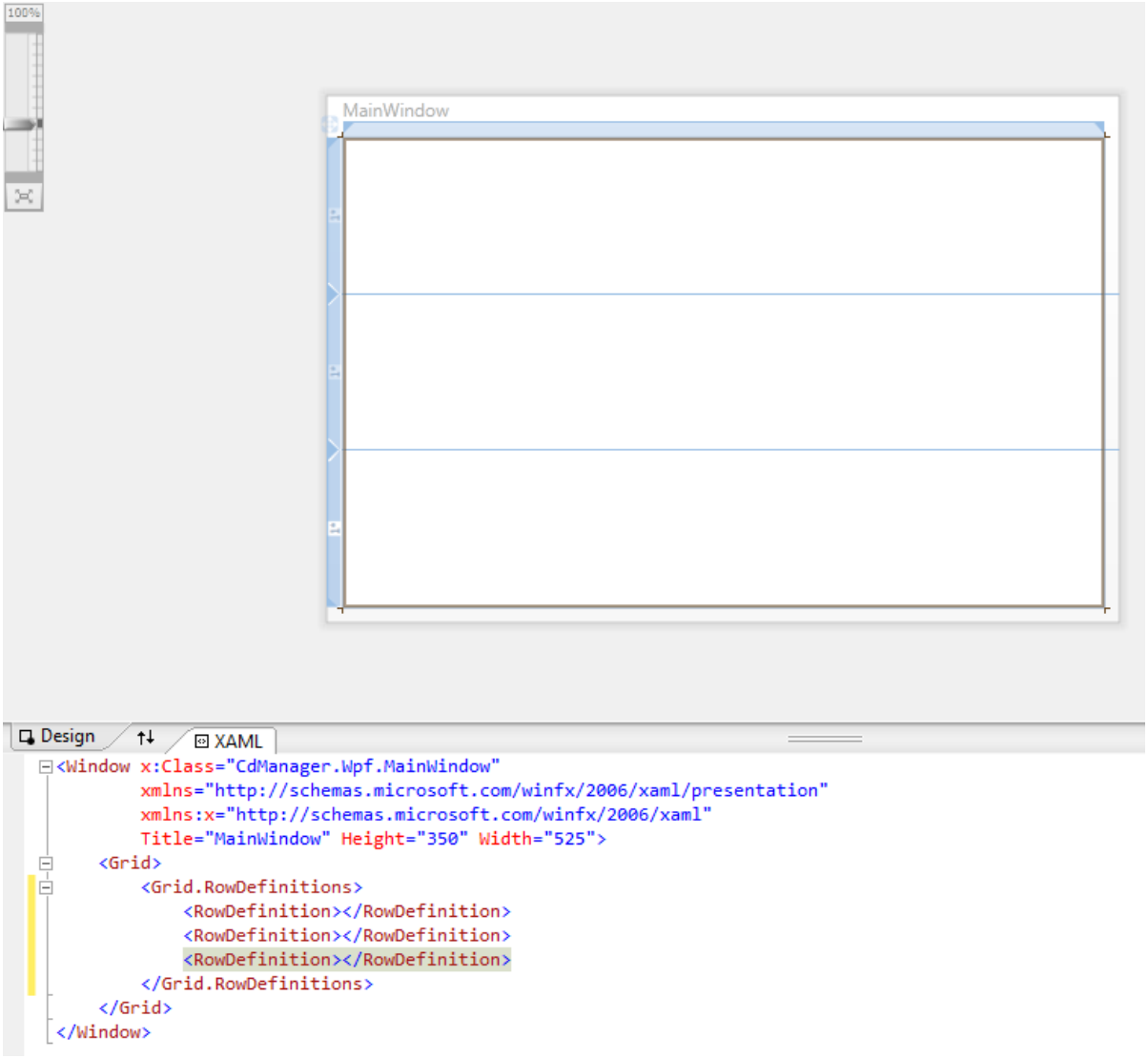
Nicht vergessen das WPF Projekt als Startup-Projekt auszuwählen!

Das Formular soll so aufgebaut werden, dass im oberen Bereich die Überschrift platziert werden kann, darunter wird eine Liste mit allen Cds angezeigt und wiederum darunter werden die Auswahl-Buttons (Neu, Löschen, Tracks bearbeiten) nebeneinander platziert:



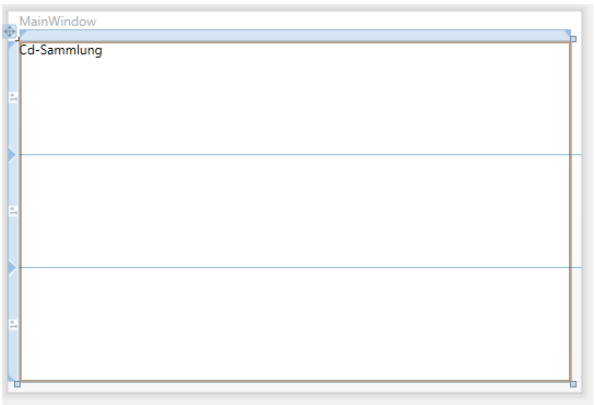
Das ist unser Ziel (noch ohne Layout / Styles)

Wir wechseln nun in den XAML-Editor des MainWindow (durch Doppelklick auf MainWindow.xaml). Zunächst teilen wir das Formular in ein Grid mit 3 Zeilen ein.



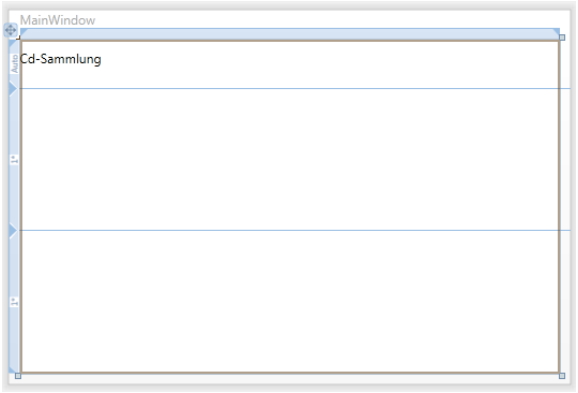
In die oberste Gridzeile platzieren wir zunächst eine TextBlock Komponente mit dem Text „Cd-Sammlung“ → dies wird unsere Überschrift:

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition></RowDefinition>
        <RowDefinition></RowDefinition>
        <RowDefinition></RowDefinition>
    </Grid.RowDefinitions>
    <TextBlock Grid.Row="0">Cd-Sammlung</TextBlock>
</Grid>
```



Wie zu erkennen ist, ist die oberste Zeile nun viel zu groß für die Überschrift (standardmäßig werden die Größenverhältnisse im Grid gleich aufgeteilt). Um dies zu ändern, einfach die Größe auf „Auto“ setzen → Gridzeile wird so groß wie erforderlich, bzw. so klein wie möglich. Damit sie jedoch nicht zu klein wird, setzen wir zusätzlich bei unserer Textbox noch einen Rand (Margin) oben und unten (später über Styles):

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"></RowDefinition>
        <RowDefinition></RowDefinition>
        <RowDefinition></RowDefinition>
    </Grid.RowDefinitions>
    <TextBlock Grid.Row="0" Margin="0,10,0,20">
        Cd-Sammlung
    </TextBlock>
</Grid>
```



In der Mitte platzieren wir zunächst eine ListBox-Komponente um die einzelnen Cds darzustellen. Damit wir aus der CodeBehind-Datei das DataBinding setzen können, vergeben wir der Komponente einen Namen. Pro Cd zeigen wir zunächst nur den AlbumTitle in der Liste an (DataTemplate in ItemTemplate legt das Aussehen je Datensatz fest!):

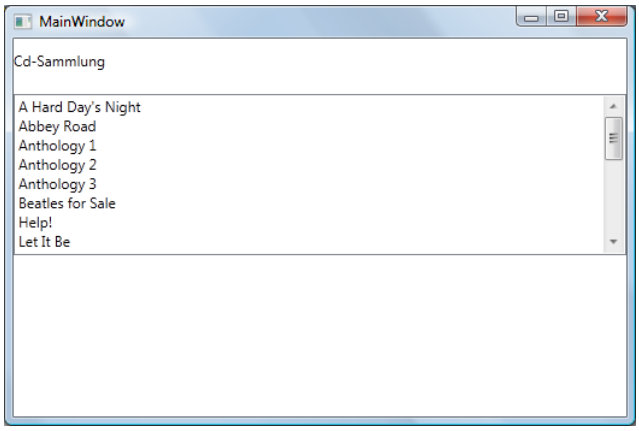
```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"></RowDefinition>
        <RowDefinition></RowDefinition>
        <RowDefinition></RowDefinition>
    </Grid.RowDefinitions>
    <TextBlock Grid.Row="0" Margin="0,10,0,20">
        Cd-Sammlung
    </TextBlock>
    <ListBox Grid.Row="1" Name="lbxCds">
        <ListBox.ItemTemplate>
            <DataTemplate>
                <TextBlock Text="{Binding Path=AlbumTitle}"/>
            </DataTemplate>
        </ListBox.ItemTemplate>
    </ListBox>
</Grid>
```

In der CodeBehind-Datei (MainWindow.xaml.cs) laden wir nun die Daten aus dem Repository und setzen eine Datenbindung mit der ListBox. (ItemSource bei List-Komponenten).
Dazu muss zunächst CdManager.Model als Referenz ins Wpf Projekt eingebunden werden und ev. ein „using CdManger.Model“ ergänzt werden.

Hinweis: Das Laden der Daten aus Repositorien, bzw. Abonnieren von Events, Setzen von Datenbindungen etc., sollte nicht direkt im Konstruktor des Formulars erfolgen, sondern erst wenn das Loaded-Event eintritt. Erst dann ist garantiert alles geladen/initialisiert. Aus diesem Grund wird im Konstruktor lediglich das Loaded-Event abonniert und erst im Loaded-Handler der Rest initialisiert:

```
public partial class MainWindow : Window
{
    List<Cd> _cds;

    public MainWindow()
    {
        InitializeComponent();
        Loaded += new RoutedEventHandler(MainWindow_Loaded);
    }

    void MainWindow_Loaded(object sender, RoutedEventArgs e)
    {
        Repository rep = Repository.GetInstance();
        _cds = rep.GetAllCds();
        lbxCds.ItemsSource = _cds;
    }
}
```

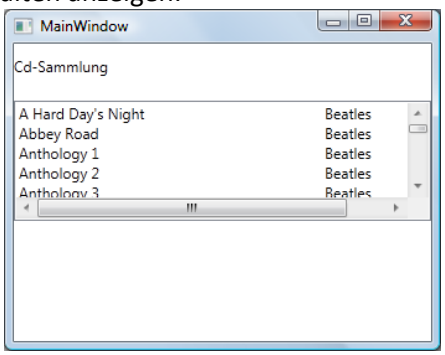
Nun kann unsere WPF-Anwendung gestartet werden und wir sehen bereits alle eingelesenen Cds!



ListBox-Anpassungen / ListView:

Möchte man mehrere Felder je Cd mit der ListBox anzeigen, so muss im DataTemplate eine Anpassung vorgenommen werden.
Beispiel: Die Felder nebeneinander in unterschiedlichen Spalten anzeigen:

```
<ListBox Grid.Row="1" Name="lbxCds">
    <ListBox.ItemTemplate>
        <DataTemplate>
            <Grid Width="{Binding ElementName=wdMain, Path=Width}">
                <Grid.ColumnDefinitions>
                    <ColumnDefinition Width="70*"/>
                    <ColumnDefinition Width="30*"/>
                </Grid.ColumnDefinitions>
                <TextBlock Grid.Column="0" Text="{Binding Path=AlbumTitle}"/>
                <TextBlock Grid.Column="1" Text="{Binding Path=Artist}"/>
            </Grid>
        </DataTemplate>
    </ListBox.ItemTemplate>
</ListBox>
```



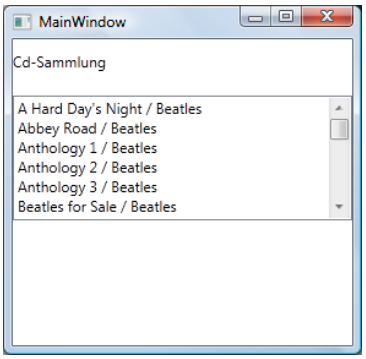
Beachte dabei: Die Gridgröße für jede Datensatzzeile wird im obigen Beispiel über die Bindung der Elementeigenschaft „Width“ des MainWindows immer genauso groß gesetzt, wie die Fenstergröße des MainWindow. Daher wird bei Vergrößerung des Fensters auch die Spaltenaufteilung dynamisch angepasst.

Bei der Definition des Fensters wurde dazu das Name Attribut gesetzt:

```
<Window x:Class="CdManager.Wpf.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525"
        Name="wdMain">
```

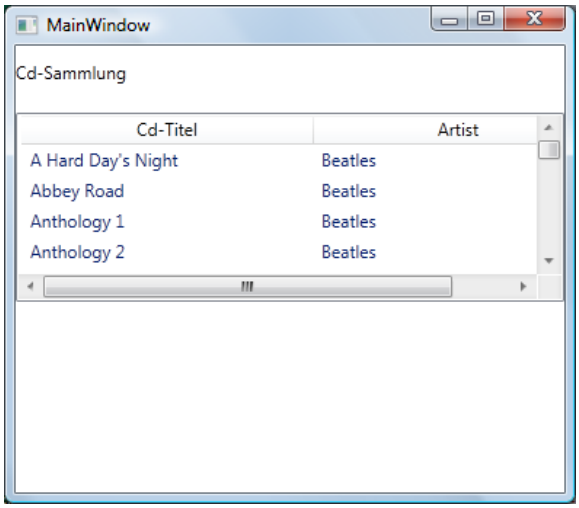
Alternativ könnten die Felder einfach mit einem / getrennt werden:

```
<ListBox Grid.Row="1" Name="lbxCds">
    <ListBox.ItemTemplate>
        <DataTemplate>
            <StackPanel Orientation="Horizontal">
                <TextBlock Text="{Binding Path=AlbumTitle}"/>
                <TextBlock Text=" / "/>
                <TextBlock Text="{Binding Path=Artist}"/>
            </StackPanel>
        </DataTemplate>
    </ListBox.ItemTemplate>
</ListBox>
```



Natürlich kann auch die ListView Komponente verwendet werden, damit können einfach Spalten und Spaltenüberschriften definiert werden:

```
<ListView Grid.Row="1" Name="lbxCds">
    <ListView.View>
        <GridView>
            <GridView.Columns>
                <GridViewColumn Header="Cd-Titel" Width="200" DisplayMemberBinding="{Binding Path=AlbumTitle}"/>
                <GridViewColumn Header="Artist" Width="200" DisplayMemberBinding="{Binding Path=Artist}"/>
            </GridView.Columns>
        </GridView>
    </ListView.View>
</ListView>
```



Buttons Neu/Löschen/Tracks bearbeiten:

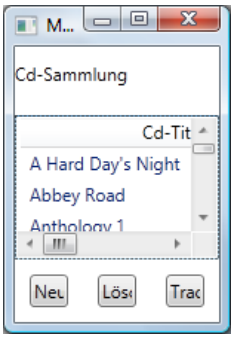
Nun werden in der dritten Gridzeile des MainForm die drei Buttons nebeneinander platziert, dabei wird jedem Button ein Name zugewiesen, um diesen später mit dem Click-Event verknüpfen zu können.
Alle Buttons sollen gleich breit sein und einen kleinen Abstand zur Umgebung aufweisen. Die Gridzeile mit den Buttons soll nur so hoch wie erforderlich sein → Bei Vergrößerung des Formulars soll dadurch nur die Höhe der mittleren Zeile (die Liste) mitwachsen (breiter werden natürlich alle Zeilen)!

```
<UniformGrid Grid.Row="2" Rows="1" Columns="3">
    <Button Margin="10,10,10,10">Neu</Button>
    <Button Margin="10,10,10,10">Löschen</Button>
    <Button Margin="10,10,10,10">Tracks bearbeiten</Button>
</UniformGrid>
```

Außerdem setzen wir dazu noch das Attribut „Height“ der dritten Gridzeile auf „Auto“:

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"></RowDefinition>
        <RowDefinition></RowDefinition>
        <RowDefinition Height="Auto"></RowDefinition>
    </Grid.RowDefinitions>
    <TextBlock Grid.Row="0" Margin="0,10,0,20">
        Cd-Sammlung
    </TextBlock>
```

Kleinen Schönheitsfehler beheben:

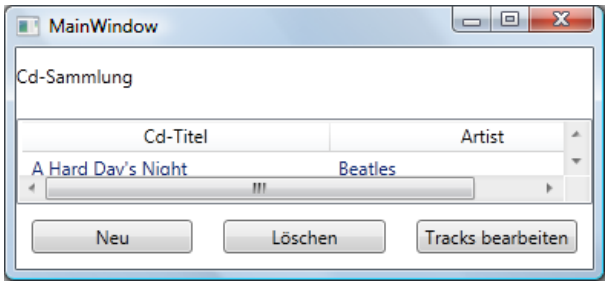


Wenn wir unser Fenster nun sehr verkleinern, so kann man ab einem gewissen Punkt, die Texte der Buttons nicht mehr lesen, da dafür zu wenig Platz bleibt:

Dies kann verhindert werden, indem man die minimale Größe des Formulars angibt (`MinHeight, MinWidth`).

Tipp: Welche Größe hierfür passend ist, lässt sich ganz gut mithilfe des Designers ermitteln: Fenster im Designmodus auf die kleinste sinnvolle Größe ziehen. Während des Verkleinerns wird die Größe neben dem Fenster angezeigt…

```
<Window x:Class="CdManager.Wpf.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="340" Width="511" MinHeight="180" MinWidth="392"
        Name="wdMain">
```



Kleiner lässt sich das Fenster nicht mehr ziehen!

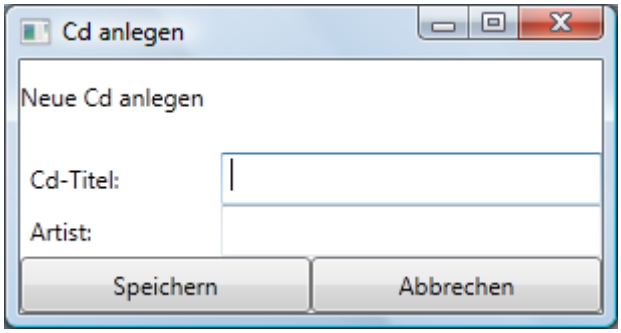
Neue Cd erstellen

Wenn der „Neu“ Button gedrückt wird, soll ein neues WPF-Formular angezeigt werden. Dazu müssen wir zunächst das neue WPF-Formular erstellen: Z.B. durch rechte Maustaste auf das WPF-Projekt und Auswahl von Add / Window (Bezeichnung AddCdWindow).

Nun abonnieren wir in der CodeBehind-Datei des MainWindows das Click Event des Neu-Buttons und öffnen im Handler das neue Formular:

```
void MainWindow_Loaded(object sender, RoutedEventArgs e)
{
    Repository rep = Repository.GetInstance();
    cds = rep.GetAllCds();
    lbxCds.ItemsSource = cds;

    btNew.Click += new RoutedEventHandler(btNew_Click);
}

void btNew_Click(object sender, RoutedEventArgs e)
{
    AddCdWindow addCdWindow = new AddCdWindow();
    addCdWindow.ShowDialog();
}
```

Das neue Formular soll **zunächst** folgendermaßen aussehen:



Dazu verwenden wir wieder ein Grid mit drei Zeilen. In der obersten Zeile platzieren wir die Überschrift. In der Mitte befinden sich jeweils ein Label und eine TextBox nebeneinander zur Dateneingabe der beiden Felder Cd-Titel und Artist (TextBox soll in der Breite mit dem Fenster mitwachsen, das Label bleibt immer gleich groß).

In der untersten Zeile sollen sich zwei Buttons zum Speichern und Abbrechen den übrigen Platz aufteilen:

Design XAML

```
<Window x:Class="CdManager.Wpf.AddCdWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Cd anlegen" Height="162" Width="306">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
            <RowDefinition/>
        </Grid.RowDefinitions>
        <TextBlock Grid.Row="0" Margin="0,10,0,20">Neue Cd anlegen</TextBlock>
        <Grid Name="grdCdFields" Grid.Row="1">
            <Grid.RowDefinitions>
                <RowDefinition/>
                <RowDefinition/>
            </Grid.RowDefinitions>
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="100"/>
                <ColumnDefinition/>
            </Grid.ColumnDefinitions>
            <Label Grid.Column="0" Grid.Row="0">Cd-Titel:</Label>
            <TextBox Name="tbTitle" Grid.Column="1" Grid.Row="0"  Text="{Binding Path=AlbumTitle}" ></TextBox>
            <Label Grid.Column="0" Grid.Row="1">Artist:</Label>
            <TextBox Name="tbArtist" Grid.Column="1" Grid.Row="1" Text="{Binding Path=Artist}" ></TextBox>
        </Grid>
        <UniformGrid Rows="1" Grid.Row="2">
            <Button Name="btSave">Speichern</Button>
            <Button Name="btCancel">Abbrechen</Button>
        </UniformGrid>
    </Grid>
</Window>
```

Sämtliche Komponenten, auf die von der CodeBehind Datei aus zugegriffen werden soll, müssen einen Namen erhalten (Name Attribut setzen).

Für die beiden TextBox Komponenten wurde bereits jeweils ein Binding Path angegeben. Dadurch können diese dann aus der CodeBehind Datei direkt mit einer Data Source verknüpft werden. (Datasource muss Felder mit derselben Bezeichnung beinhaltet).

Wir abonnieren nun wieder die beiden Button-Click Events. Beim Drücken des „Abbrechen“ Buttons wird einfach ein Close() ausgeführt:

```
public partial class AddCdWindow : Window
{
    public AddCdWindow()
    {
        InitializeComponent();
        Loaded += new RoutedEventHandler(AddCdWindow_Loaded);
    }

    void AddCdWindow_Loaded(object sender, RoutedEventArgs e)
    {
        btSave.Click += new RoutedEventHandler(btSave_Click);
        btCancel.Click += new RoutedEventHandler(btCancel_Click);
    }

    void btCancel_Click(object sender, RoutedEventArgs e)
    {
        Close();
    }

    void btSave_Click(object sender, RoutedEventArgs e)
    {
        //Neue Cd in Repository hinzufügen
    }
}
```

Wenn „Speichern“ gedrückt wird, so tragen wir die neue Cd ins Repository ein und beenden die Eingabemaske:

```
void btSave_Click(object sender, RoutedEventArgs e)
{
    //Neue Cd in Repository hinzufügen - ohne Fehlerprüfung
    Cd newCd = new Cd();
    newCd.AlbumTitle=tbTitle.Text;
    newCd.Artist=tbArtist.Text;
    Repository.GetInstance().AddCd(newCd);
    Close();
}
```

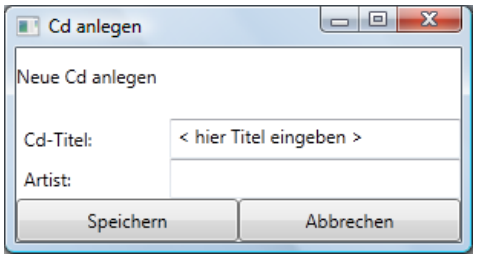
Eine weitere Möglichkeit der Umsetzung dieses Anlegen-Formulars wäre es, direkt im Loaded-Handler ein neues Cd Objekt zu erstellen und dieses als DataContext an das Formular zu binden. (Es genügt auch, dieses an das Grid mit den Eingabefeldern zu binden.)

Zur Demonstrationszwecken wird als Titel beim neuen Cd Objekts der Text „< hier Titel eingeben >“ eingetragen. Dieser Text wird dann durch die Datenbindung beim Aufruf des Formulars im Titel Feld angezeigt!

Da bei den Textbox-Komponenten im XAML bereits der Binding-Path gesetzt wurde, ist dort nichts mehr abzuändern.

Beim btSave_Click kann dann direkt das newCd-Objekt weitergegeben werden. Die Werte des Objekts sind bereits aktuell, da sie an die Eingabefelder gebunden sind. (Achtung! Möchte man, dass sich auch die Eingabefelder am Formular bei Änderung des Objektes automatisch ohne „Refresh“ aktualisieren, so muss Cd das Interface INotifyPropertyChanged implementieren! → Später mehr dazu):

```
public partial class AddCdWindow : Window
{
    Cd newCd;

    public AddCdWindow()
    {
        InitializeComponent();
        Loaded += new RoutedEventHandler(AddCdWindow_Loaded);
    }

    void AddCdWindow_Loaded(object sender, RoutedEventArgs e)
    {
        btSave.Click += new RoutedEventHandler(btSave_Click);
        btCancel.Click += new RoutedEventHandler(btCancel_Click);

        newCd = new Cd();
        newCd.AlbumTitle = "< hier Titel eingeben >";
        grdCdFields.DataContext = newCd;
    }

    void btCancel_Click(object sender, RoutedEventArgs e)
    {
        Close();
    }

    void btSave_Click(object sender, RoutedEventArgs e)
    {
        //Neue Cd in Repository hinzufügen - ohne Fehlerprüfung
        Repository.GetInstance().AddCd(newCd);
        Close();
    }
}
```



Ein Testlauf wird nun zeigen, dass wir den Datensatz anlegen können, dieser wird jedoch nach Beendigung des Anlegen-Dialogs noch nicht in der Liste angezeigt, da die Daten noch nicht aktualisiert werden.

Dies lässt sich einfach lösen, indem die Daten **im MainWindow** nach Rückkehr vom AddCdWindow neu geladen werden:

```
void btNew_Click(object sender, RoutedEventArgs e)
{
    AddCdWindow addCdWindow = new AddCdWindow();
    addCdWindow.ShowDialog();
    //Nachdem der "Neuanlegen Dialog" geschlossen wurde
    //Liste der Cds aktualisieren:
    cds = Repository.GetInstance().GetAllCds();
    //Bei normaler Collection muss zusätzlich ItemSource neu gesetzt werden
    //um Aktualisierung auszulösen:
    lbxCds.ItemsSource = cds;
}
```

Da cds eine List<Cd> ist, wird die ListBox nicht automatisch über die Änderung der Daten informiert → ItemsSource muss neu gesetzt werden!
Alternativ könnte man eine ObservableCollection<Cd> verwenden. Diese verständigt automatisch die verbundenen WPF-Komponenten. Da wir jedoch ohnehin die gesamten Daten nach jeder Änderung neu vom Repository laden und eine List zurückgegeben wird, ist dies hier nicht erforderlich.

Das Löschen, Anzeigen und Bearbeiten der Tracks mittels eigener Knöpfe sollte nun eigenständig umgesetzt werden! Repository muss natürlich erweitert werden.

Hinweise:

- MessageBox.Show(..) stellt Funktionalität zum Anzeigen von Hinweismeldungen bzw. auch zum Abfragen von Bestätigungen zur Verfügung. Sehen Sie sich die verschiedenen Überladungen an.
- Die Eigenschaft „SelectedItem“ liefert in einer ListView/ListBox das aktuell ausgewählte Element.

Fortsetzung im Tutorial Teil 2