Binding to a database

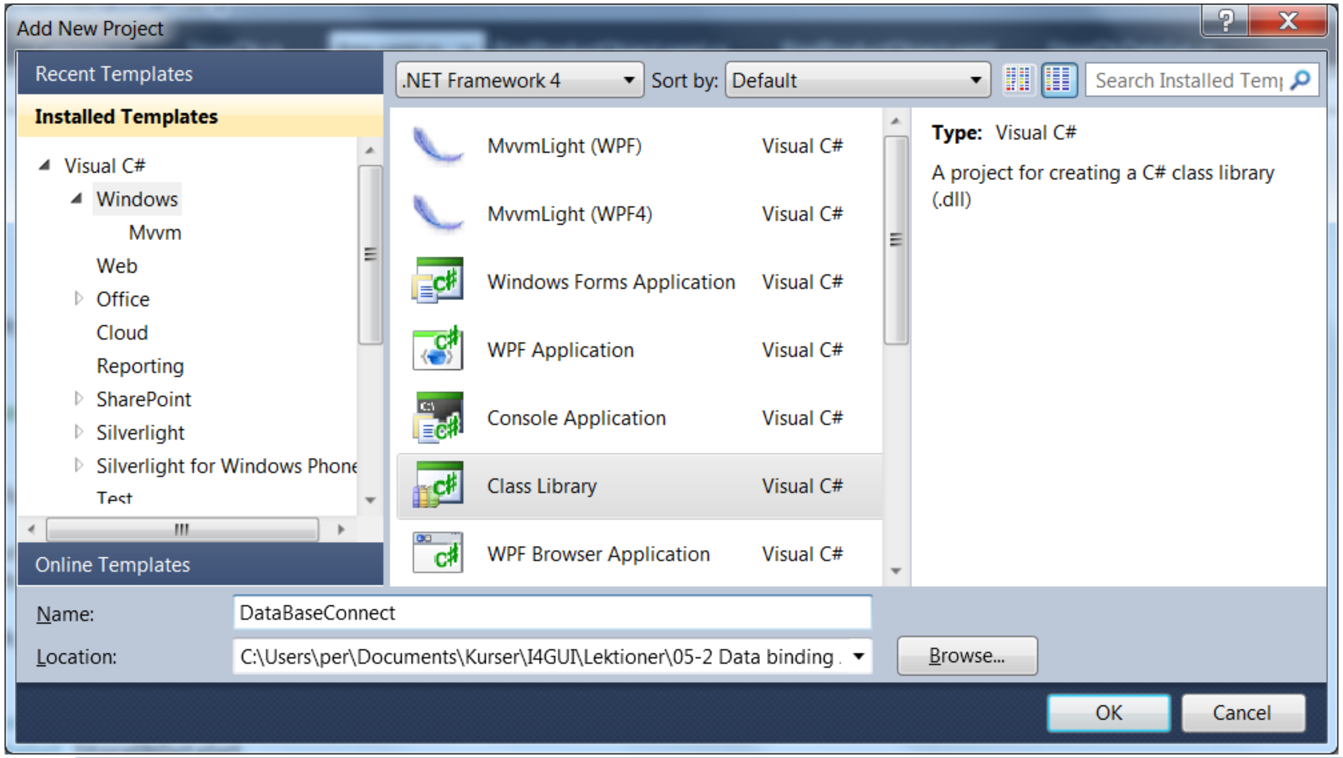
AARHUS UNIVERSITY
SCHOOL OF ENGINEERING

Agenda

- Binding to a Database with custom objects

Build a Data Access Component!

- In professional applications, database code is not embedded in the code-behind class for a window!
  - But encapsulated in a dedicated class
  - Or these data access classes can be pulled out of your application altogether and compiled in a separate DLL component (Class Library)



MvvmLight (WPF)	Visual C#
MvvmLight (WPF4)	Visual C#
Windows Forms Application	Visual C#
WPF Application	Visual C#
Console Application	Visual C#
Class Library	Visual C#
WPF Browser Application	Visual C#

Type: Visual C#

A project for creating a C# class library (.dll)

Name: DataBaseConnect

Location: C:\Users\per\Documents\Kurser\I4GUI\Lektioner\05-2 Data binding

AARHUS UNIVERSITY
SCHOOL OF ENGINEERING

A Few Basic Guidelines

- Open and close connections quickly
  - Open the database connection in every method call, and close it before the method ends (use a using block)
- Implement error handling to make sure that connections are closed even if an exception occurs
  - Close the connection in a `finally` clause
- Follow stateless design practices.
  - Accept all the information needed for a method in its parameters, and return all the retrieved data through the return value
- Store the connection string in one place
  - typically in the configuration file for your application

Data Access Implementation

- The data access class is exceedingly simple
  - it provides just a single method that allows the caller to retrieve one product record.

```
public class StoreDB
{
  // Get the connection string
  private string connectionString =
                    Properties.Settings.Default.StoreDatabase;

  public Product GetProduct(int ID)
  {
    ...
  }
}
```

AARHUS UNIVERSITY
SCHOOL OF ENGINEERING

Access DB by use of SqlDataReader

```
public class StoreDb {
  string connectionStr = StoreDatabase.Properties.Settings.Default.Store;

  public Product GetProduct(int ID) {
    SqlConnection con = new SqlConnection(connectionStr);
    SqlCommand cmd = new SqlCommand(
                             "SELECT * FROM Products WHERE ProductID="+ID, con);
    Product product = null;
    try {
      con.Open();
      SqlDataReader reader = cmd.ExecuteReader(CommandBehavior.SingleRow);
      if (reader.Read()) { // Create a Product object that wraps the record.
        product = new Product((string)reader["ModelNumber"],
           (string)reader["ModelName"], (decimal)reader["UnitCost"],
           (string)reader["Description"], (string)reader["ProductImage"]);
      }
      finally {
        con.Close();
      }
    return product;
  }
```

How To Reach the StoreDB class?

- You have several options for making the StoreDB class available to the windows in your application:
  - The window could create an instance of StoreDB whenever it needs to access the database
  - You could change the methods in the StoreDB class to be static
  - **You could create a single instance of StoreDB and make it available through a static property in another class - e.g. the App class (kind of "factory" pattern)**

```
public partial class App : System.Windows.Application
{
  private static StoreDB storeDB = new StoreDB();
  public static StoreDB StoreDB
  {
    get { return storeDB; }
  }
}
```

Build a Data Object (or DTO)

- The data object is the information package that you plan to display in your user interface by use of data binding
- Any class works, provided it consists of public properties

```
public class Product
{
  private string modelNumber;
  public string ModelNumber
  {
    get { return modelNumber; }
    set { modelNumber = value; }
  }

  private string modelName;
  public string ModelName
  {
    get { return modelName; }
    . . .
```

Design the Window and Use Data Binding

```
<Grid Name="gridProductDetails">
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="Auto"></ColumnDefinition>
    . . .
  <TextBlock Margin="7">Model Number:</TextBlock>
  <TextBox Margin="5" Grid.Column="1"
           Text="{Binding Path=ModelNumber}"></TextBox>
  <TextBlock Margin="7" Grid.Row="1">Model Name:</TextBlock>
  <TextBox Margin="5" Grid.Row="1" Grid.Column="1"
           Text="{Binding Path=ModelName}"></TextBox>
  <TextBlock Margin="7" Grid.Row="2">Unit Cost:</TextBlock>
  <TextBox Margin="5" Grid.Row="2" Grid.Column="1“
           Text="{Binding Path=UnitCost}"></TextBox>
  <TextBlock Margin="7,7,7,0"
             Grid.Row="3">Description:</TextBlock>
  <TextBox Margin="7" Grid.Row="4" Grid.Column="0“
           Grid.ColumnSpan="2“
           TextWrapping="Wrap"
           Text="{Binding Path=Description}"></TextBox>
</Grid>
```

Use DataContext to Supply the Source

- When the user clicks the button at runtime, you use the StoreDB class to get the appropriate product data
- And you use the DataContext property on the Grid (or Window) to supply all your data binding expressions with a source object
  - And all your binding expressions will use it to fill themselves with data

```
private void cmdGetProduct_Click(object sender,
                                 RoutedEventArgs e)
{
  int ID;
  if (Int32.TryParse(txtID.Text, out ID))
  {
    try
    {
       gridProductDetails.DataContext =
                               App.StoreDB.GetProduct(ID);
     }
. . .
```

Null Values

- The results of binding a null value are predictable:
  - the target element shows nothing at all
- You can change how WPF handles null values by setting the TargetNullValue property in your binding expression:

```
Text="{Binding Path=Description, TargetNullValue=[No Description
Provided]}"
```

Updating the Database

- Add an UpdateProduct() method to the StoreDB class that
  - Grab the current Product object from the data context and use it to commit the update

```
Product product = (Product)gridProductDetails.DataContext;
try
{
  App.StoreDB.UpdateProduct(product);
}
catch
{
  MessageBox.Show("Error contacting database.");
}
```

Binding to XML

- The `XmlDataProvider` allows simple XAML-based declaration of XML resources for use in a WPF application.

```
<Window.Resources>
  <XmlDataProvider
      x:Key="cve"
      Source="X:\Path\to\allitems.xml"
      XPath="/cve/item"
      IsAsynchronous="False"
      IsInitialLoadEnabled="True"
      debug:PresentationTraceSources.TraceLevel="High"
   />
</Window.Resources>
```

- XPath is a standard for defining selections within XML
  - `/cve/item` says to select all the item elements underneath the root cve element.

CVE Viewer
Demo 30

When to use XPath

- When the binding source is XML data instead of a CLR object, the XPath property is used instead of the Path property to indicate the path to the value on the binding source to use.
- By setting the XPath property, you are creating an XPath query to select a node or a collection of nodes

```
<TextBlock Text="{Binding XPath=@name}" />
```

Binding to ADO.NET database objects

- WPF lets you bind data directly from a database to your UI
  - But think carefully before you do
  - There *are* situations in which this is an appropriate thing to do
    - But binding directly to objects in the DAL layer bypasses the BLL-layer!

```
public DataTable Bookmarks
{
  get { return library.Tables["Bookmarks"]; }
}

public event PropertyChangedEventHandler PropertyChanged;

private void NotifyPropertyChanged(String propertyName)
{
  if (PropertyChanged != null)
      PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
}
```

AARHUS UNIVERSITY
SCHOOL OF ENGINEERING