|  |
| --- |
| **EASJ Notes** |
| Object-Oriented Pro-gramming with C# |
| Application Development, Part I |

|  |
| --- |
| By Per Laursen  01-08-2018 |

[Introduction 3](#_Toc522010823)

[GUI and Object-Orientation 3](#_Toc522010824)

[Event-driven applications 3](#_Toc522010825)

[The duality of GUI components 4](#_Toc522010826)

[What is XML (and XAML) 6](#_Toc522010827)

[XAML and Visual Studio – getting started 9](#_Toc522010828)

[Simple controls 14](#_Toc522010829)

[Button 14](#_Toc522010830)

[TextBlock 14](#_Toc522010831)

[TextBox 15](#_Toc522010832)

[Image 15](#_Toc522010833)

[Slider 15](#_Toc522010834)

[Layout controls 16](#_Toc522010835)

[Grid 16](#_Toc522010836)

[StackPanel 18](#_Toc522010837)

[Control properties 18](#_Toc522010838)

[Default properties 19](#_Toc522010839)

[Complex properties 19](#_Toc522010840)

[Attached properties 19](#_Toc522010841)

[Layout properties 20](#_Toc522010842)

[Using styles 20](#_Toc522010843)

[Data Binding 21](#_Toc522010844)

[Simple binding between GUI elements 22](#_Toc522010845)

[Data binding between GUI elements and model objects 22](#_Toc522010846)

[Collection Views and Data Binding 29](#_Toc522010847)

[The ListView control – getting started 29](#_Toc522010848)

[The ListView control – displaying items 31](#_Toc522010849)

[Defining a data template 33](#_Toc522010850)

[The ListView control – binding to SelectedItem 34](#_Toc522010851)

[The GridView 36](#_Toc522010852)

[Commands 37](#_Toc522010853)

[Deleting a domain object 37](#_Toc522010854)

[The ICommand interface 39](#_Toc522010855)

[Exercises 45](#_Toc522010856)

[AD.1.0 45](#_Toc522010857)

[AD.1.1 46](#_Toc522010858)

[AD.1.2 47](#_Toc522010859)

[AD.1.3 48](#_Toc522010860)

[AD.1.4 49](#_Toc522010861)

[AD.1.5 50](#_Toc522010862)

[AD.1.6 51](#_Toc522010863)

[AD.1.7 52](#_Toc522010864)

[AD.1.8 53](#_Toc522010865)

# Introduction

So far, we have only used some very primitive facilities for interacting with the users of our small applications. For more sophisticated applications with richer user inter­action, this is clearly not enough. We will therefore now embark on the – quite large – topic of GUI (Graphical User Interface) development, which in turn will lead us to consider how to structure a larger application in general.

Modern GUI development is a rather complicated activity. There is a large number of GUI “components” or **controls** (like buttons, drop-down menus, list boxes, etc.) to choose from, and the layout of the GUI can be specified in many different ways. In this text, we only discuss the more general challenges related to GUI develop­ment, and subse­quently focus on so-called **data binding**, which relates to the problem of keeping the GUI and the underlying data model consistent at all times.

## GUI and Object-Orientation

We have so far mostly used Object-Orientation as a way of modeling aspects of a cer­tain **domain**, like a bank, a role-playing game, etc.. Object-Orientation is however al­so a useful tool in the realm of GUI development. We can imagine classes repre­sent­ing specific GUI elements – like a **Button** class, a **Window** class, etc. – and we can also imagine relations between such classes, in terms of inheritance and composi­tion. Not surprisingly, such a system of classes already exists as part of the .NET Framework class library. In fact, this part of the class library is quite large, and it is easy to get lost in the vast jungle of controls, available properties, and so on. We will only scratch the surface here.

## Event-driven applications

Applications equipped with a “real” GUI usually have a different model of execution than we have seen so far. Until now, most applications have had a very well-defined flow-of-execution, beginning execution of the logic immediately when the application is launched, and usually executing the entire logic without user interaction. A GUI-rich application can be characterised as an **event-driven** application, i.e. the applica­tion will usually wait for the user to initiate some specific action, perform that action once initiated, and then wait for the user to initiate the next action. The user may ini­ti­ate an action by e.g. clicking on a button or making a select­ion in a list box; such an **event** will in turn initiate a specific part of the logic defined in the applica­tion. As we will see later on in this chapter, this model of execu­tion makes it a bit more complica­ted to activate specific parts of the code, in response to the users actions.

# The duality of GUI components

All GUI components share a few characteristics, that influence how we in general define and use GUI components. Most importantly, the full specification of a GUI compo­nent falls into two fundamentally different categories:

* The **visual appearance** of the GUI component: How does the component mani­fest itself on the device on which the user interacts with the application?
* The **behavior** of the GUI component: What happens inside the application when the user interacts with the GUI component?

Why is this distinction important? Primarily due to the role of time. The visual appe­ar­­ance of a GUI component does not depend on the “flow of execution”, but has a more timeless nature (this is a bit simplified, but we will elaborate a bit on this short­ly). Con­versely, behavior is per definition something that starts (with respect to time) and ends. This difference is important in relation to how we should specify appear­ance and behavior, respectively.

Considering behavior first, it should not be surprising that C# itself is a very suitable language for specifying behavior – that’s what we have been doing over and over in the previous chapters. However, C# may not be the best choice for specifying appear­ance. We have characterised C# as an Object-Oriented language; it can also be cha­rac­­terised as a language with a **procedural** nature. In less academic terms, this means a language where we describe how things are done. We use sequences of state­ments to describe this, with the underlying assumption that statements are executed in a certain order with respect to time. Other languages can be characterised as being **declarative**. These languages only describe relations between certain elements, and time is not as such a factor. One example of such a language is HTML (HyperText Markup Language). HTML has traditionally been used for specifying the layout of a website, i.e. its visual appearance. The point we’re trying to make is: since the visual appearance is not as such dependent of time or the “flow of execution” of the appli­cation, it can be advantageous to use a declarative language to describe the visual appearance, rather than try to describe it in e.g. C#. Or even shorter: Use the right tools for the right job!

So, the conclusion of the above discussion is: For a GUI component, use

* C# for defining behavior
* A declarative langauge for defining the visual appearance

This conclusion leads to two new problems:

* What language should we then use for defining appearance?
* How can we define a single C# class that contains definitions of both the behavior and the appearance?

Microsoft has resolved this by introducing the language **XAML** (eXtensible Applica­tion Markup Language), which is a specialised version of **XML** (eXtensible Markup Langu­age). We will talk a bit more about XAML/XML in a moment; for now, just note that XAML is indeed a declarative language.

If XAML is used for defining appearance for a GUI component, while C# is still used for defining behavior, how is it then possible to “merge” this into a single C# class? C# does – very conveniently – allow a class definition to be split across more than one file! The keyword **partial** can be added to a class definition, which indicates to the compiler that the class definition is spread across a number of files.

With this in place, only one more piece of the puzzle remains: transformation of a XAML-based definition to a (partial) C# class. This is indeed possible to do, and is an integral part of Visual Studio. We don’t need to know how this is done, only that it is possible to do it ☺.

The diagram below should illustrate how the pieces are put together. The behavior part is written in C# “by hand”, and is stored in a file with the extension **.xaml.cs**. The appearance part is written (or imported, see later) in XAML, and is stored in a file with the extension **.xaml**. This file is then processed by Visual Studio, generating a C# file with the extension **.xaml.g.cs**. This file – which now contains a partial class defini­tion – is then combined with the hand-written C# file, which also contain a partial class definition. The end result is thus a complete C# class, containing definitions for both behavior and appearance.

MyControl

C# - generated

MyControl (behavior)

C# - written

MyControl.xaml.cs

MyControl (appearance)

C# - generated

MyControl.xaml.g.cs

MyControl (appearance)

XAML - written/imported

MyControl.xaml

This may look like a lot of trouble for creating a C# class, and it is indeed possible to write a GUI component class directly in C# without all this fuss. However, the division of definition of behavior and appearance does make it possible to define a GUI using an external tool. If the external tool is capable of exporting a GUI definition to XAML, we can simply import that definition into Visual Studio! Microsoft has made such a tool, called **Microsoft Blend**. The point is that GUI (appearance) definition is an acti­vity that doesn’t require programming skills as such, so it should be possible for a de­sign professional to work with GUI design using a tool specialised for this purpose.

# What is XML (and XAML)

We claimed above that XAML is a specialisation of XML, which in turn is a declarative language. So, what is XML?

XML (eXtensible Markup Langu­age) is a language for specifying structural relations between data. What does that mean? Suppose I have some data about a small book collection, like:

“War and Peace”, Tolstoy, 539 pages

“Huckleberry Finn”, Twain, 341 pages

As human beings, we can fairly easily understand this data. If a computer appli­cation had to process this data, it would need some extra data to make sense of the data. Suppose now we write the data as:

**<BookCollection>**

**<Book>**

**<Title>War and Peace</Title>**

**<Author>Tolstoy</Author>**

**<Pages>539</Pages>**

**</Book>**

**<Book>**

**<Title> Huckleberry Finn</Title>**

**<Author>Twain</Author>**

**<Pages>341</Pages>**

**</Book>**

**</BookCollection>**

Here we have added “data about the data”, so-called **meta-data**. This meta-data can be used by an application to e.g. search for specific types of data, or to display the data in a certain manner, depending on the type of data. Also, the meta-data defines the relation between the actual data. More specifically, we express the meta-data in terms of **tags**. A tag is a sort of keyword, that we have decided has a certain meaning. In the example, the word **Book** has a certain – hopefully obvious – meaning. We can then insert a tag into our description like this:

**<Book>**

**</Book>**

The **<Book>** tag means *“now will follow data about a Book”*, and the **</Book>** tag means *“now ends the data about a Book”*. Between these so-called **opening** and **closing** tags, we can then add data about a specific book:

**<Book>**

**<Title>Huckleberry Finn</Title>**

**<Author>Twain</Author>**

**<Pages>341</Pages>**

**</Book>**

So, it seems that there are three pieces of data about a book: title, author and (the number of) pages. All of the above (the opening tag, the details and the closing tag) defines an **element**, here of type **Book**. Note that the details themselves follow the same structure: opening tag, details, closing tag. We can thus have elements within elements, and this is precisely what enables us to express relations between ele­ments. In the XML above, the **Title** element is a “child” of the **Book** element, and **Book** elements are themselves children of the **BookCollection** element.

This is essentially what XML is; data, and meta-data specifying the structure of the data itself. You can hopefully see that such a language has a very different nature than C#, since there is no concept of “flow of execution” here. We only declare cer­tain relations between data. A file containing XML code – plus a little bit of informa­tion about the XML version being used – is formally called an **XML document**.

Even though we have just stated that XML is very different from C# (which is true), it doesn’t mean that it operates in a realm that is completely different from that of C#. If you think a bit about it, you can interpret the XML data as describing the structure between specific objects: A **BookCollection** object is just a collection-type object, that contains two **Book** objects. A **Book** object in turn has some properties **Title**, **Author** and **Pages**, that have some specific values. If we had defined similar classes in C#, we could easily create a similar structure by creating C# objects, and use compo­si­tion to relate them to each other. Some describe XML as an “object instantiation language”, which is fairly accurate. In that light, it may be easier to understand why a transfor­mation from XML to C# isn’t that complicated to perform.

We said earlier that by adding this meta-data, a computer application could make bet­ter sense of the data. This is a bit too simplified. No computer applications know as such what a “book” is… When you wish to use data on XML format, the “producer” (which could be a human being or another application) and the “consumer” (the appli­­cation receiving the XML data) of the XML data must agree on the XML “lang­u­age” used for communication. That “language” is essentially just a specification of

* Names that are considered valid tag names (i.e. data types)
* What relation elements must have to each other

Such a specification is called a **schema**. We don’t need to know much more about schemas at this point, but it is the schema that defines the type names we may use and how the types are related. For the above example, the schema may define that:

* **BookCollection**, **Book**, **Title**, **Author** and **Pages** are legal types
* A **BookCollection** element may contain any number of **Book** elements
* A **Book** element must contain exactly one **Title** element
* …and so on

In this way, it is the schema that defines a “specialisation” of XML. Our example is rela­­ted to books, so we could name this specialisation XBML (eXtensible Book Mark­up Langu­age). Likewise, Microsoft has defined a schema related to specifi­cation of gra­phi­cal elements in an application, and named it XAML (eXtensible Application Markup Langu­age). Describing the structure of a GUI fits pretty well with this way of structu­ring information. A GUI usually starts with a window, inside which is a number of pa­ges, within which there are a number of controls, and so on.

XAML makes heavy use of another XML feature called **attributes**. An attribute is just another way of representing data. In the example above, we could have chosen to use attri­butes instead:

**<Book Title="War and Peace" Author="Tolstoy" Pages="539">**

**</Book>**

The meaning is exactly the same, but expressed in terms of attributes. Attributes are written inside the **<** and **>** of the opening tag. Each attribute should be understood as a key-value pair; the value of **Title** is *“War and Peace”*. Note that all attribute values must be specified as strings, even if they are numeric (such as the page number).

It should also be mentioned that you can use a shorthand for tags, if the element only contains attributes:

**<Book Title="War and Peace" Author="Tolstoy" Pages="539" />**

The **/** symbol from the closing tag has thus been moved to the end of the opening tag. As before, the meaning is exactly the same.

When should you use attributes in favor of child elements? There is no clear-cut rule for this, and it is strictly speaking not a relevant question here, since we will have to follow the style that XAML “imposes” on us. The important point here is to under-stand what an attribute is.

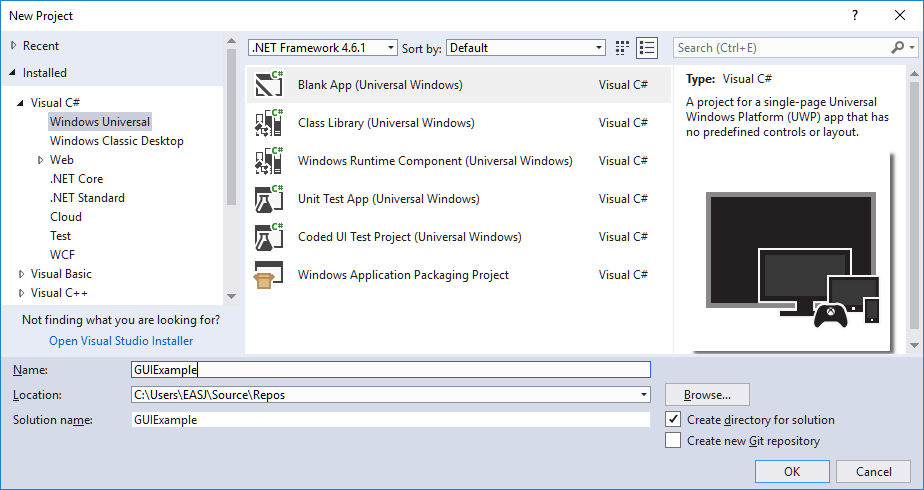
There are of course many tutorials about XML on the web, if you want to learn more. The W3Schools tutorial <http://www.w3schools.com/xml/> is a good starting point.

# XAML and Visual Studio – getting started

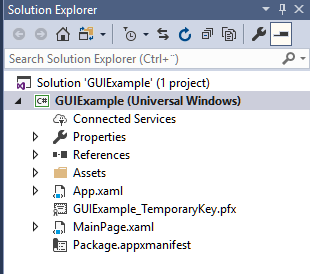
Let’s recap the story so far:

* There is a fundamental difference between the visual appearance and the beha­vior of a GUI component
* The behavior is defined by using C#
* The visual appearance is defined by using XAML (eXtensible Application Markup Language)
* You can create the (visual appearance of a) GUI directly in Visual Studio (using XAML), or by using a third-party tool

The next step is to investigate in more detail how we go about creating an application with a real GUI. In Visual Studio, we have usually created and worked with so-called “console applications”, where we use **Console.WriteLine** to print simple texts on the screen. However, we can also choose to create an application of the type **Universal Windows Platform** (UWP). In Visual Studio, we can choose this application type in the **New Project** dialog:



Here, we choose to create a “Blank App (Universal Windows)”, and have chosen the name **GUIExample**. Once Visual Studio has created the project (which may take a little while, if this is the first project of this kind you are creating), we are met with some­thing which looks radically different from the projects we have worked with so far. The **Solution Explorer** window will look something like this:



There is no **Program.cs** file, and a couple of files with the extension .xaml. The file called **MainPage.xaml** is the most interesting one right now. If you double-click this file, an view looking similar to the below will appear:



This is the **main view** of the application right now. Or more specifically; if we run the application as it is right now, this is what it will look like, if we run it on a Windows phone with a 5” screen with resolution 1920x1080. One of the fundamental ideas behind UWP is that a UWP application should be able to run on any hardware device that runs Windows 10, be it an ordinary computer, a smartphone, an Xbox, a Smart-TV or whatever. Therefore, it is possible to preview what an application will look like on a certain device. If you look closely in the upper-left corner of the previous image, you will see a list-box where **“ 5” phone (1920 x 1080) ”** has been chosen. We can choose other devices, for instance a 23” desktop with 1920x1080 resolution. We will in general work with this choice, and will not in this text dive into the details related to how to ensure that a UWP application looks reasonable on all possible devices. Assuring this involves setting up the GUI elements in a certain manner, which is not a feature we will focus on here.

Below the top part (usually called the **Design View**) is a window containing XML code; more specifically XAML code, as described earlier. This may initially look quite com­plex, but the good news is that the XAML code present from the beginning is usually not something we need to think much about, let alone change. This is actually the definitions of the “schemas” we also mentioned earlier, so this just ensures the we “speak XAML” in this file.

The only part that is directly related to the GUI is the **<Grid>** element. Visual Studio inserts this per default. To keep things simple, we will simply delete it, so we start completely from scratch. Be sure, however, that you only delete the **<Grid>** tag (from **<Grid>** to **</Grid>**, both included).

With this out of the way, we have a blank area to work with. Right now, the GUI does not contain any elements at all. How do we then add a GUI element? You can do this in a couple of ways.

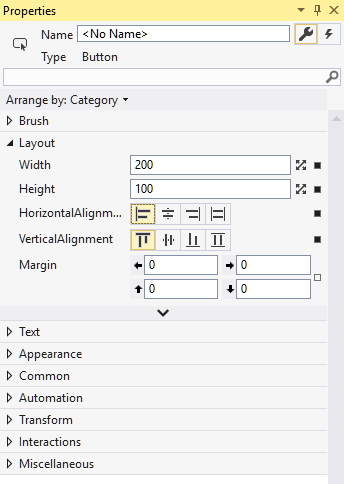
The first way is the graphical way. Just next to the **Solution Explorer** window, there should be a **Toolbox** pane to click on (alternatively, choose **View | Toolbox** in the menu). If you then expand the **Common XAML Controls** element, you will see a long list of GUI elements. One of those elements is called **Button**. If you click on the entry in the list, you can then simply drag an element onto the view. If you do so, you will probably end up with an enormous button, filling up the entire view! Don’t worry, we will fix that later on. For now, just note that some text (probably one long line of text) has been added to the XAML part of the window. The text (with a bit of formatting) will probably look like:

**<Button Content="Button" HorizontalAlignment="Left"**

**Height="1080" VerticalAlignment="Top" Width="1920"/>**

How do we read this? The tag type specifies what kind of GUI control the tag defines, in this case a **Button**. Then follows the “content” of the button, which initially simp­ly is the text **Button**. The last four elements are related to the graphical layout of the button; if you try to change 1080 to 100, and change 1920 to 200, you will see that the button changes size accordingly (remember that the values must be enclosed in **“”**, even though they are numeric!).

You have probably by now figured out that the visual view and the XAML code are just two sides of the same thing; it is simply two ways of looking at the same data. This also means that you can work with the GUI in both ways, choosing the way that you are most comfortable with. You can thus add GUI elements simply by writing proper XAML in the XAML window. There is even a third way to work with the GUI: If you select the button in the view and then press F4, a **Properties** window will open:



Here the **Layout** part has been expanded, and you can again see the values that we have already seen in the XAML code and in the Design view. So, the **Properties** view is just a third view into the same data. If you close the **Properties** view, you can always open it again by pressing F4.

All these ways to view data, the number of GUI controls and the myriad of properties you can set for a control can feel overwhelming, and there are indeed many little knobs you can turn. However, you only need to know about a few control and a few properties for controls in order to get started.

Before diving into the specifics of certain controls, a couple of things about working with these types of applications should be mentioned. First, it is as mentioned before possible to get a “preview” of how the application will look on various devices, by choosing such a device in the Design view. It is even possible to try to run the appli­cation in a device simulator, where it is not only the visual appearance that is simula­ted, but also things like memory limitations, etc.. You choose between such simula­tors by expanding the lis box attached to the well-known **Start** button. However, note that you will need to download device-specific simulators to Visual Studio, before they can be used. In or­der to keep things simple, we will in general just go with the “Local Machine” option, that just launches the application directly on your computer.

Another – somewhat more obscure – issue is related to setting access rights to the file system. If you create a UWP application – or e.g. download an exercise project – build it, and try to launch it, you may very well experience an “Access denied” error. Access to the file system is a non-trivial issue for UWP applications[[1]](#footnote-1), but the easiest way to fix theis problem is as follows:

1. Using the **File Explorer**, go to the folder where you intend to store your UWP applications.
2. Right-click the folder, and go to **Properties** in the menu.
3. Go to the **Security** tab.
4. Click on the **Edit** button.
5. Click on the **Add** button.
6. Add **SYSTEM** as a user name.
7. Give the **SYSTEM** user full access rights to the folder.

This should get rid of any problems with denied access.

## Simple controls

The term “simple controls” is not a formal one – we just use it to describe a handful of very fundamental controls, that will almost always be part of a typical GUI. You may of course encounter more specialised situations, where you will need to rely on other types of controls. If so, seek information about such controls online[[2]](#footnote-2).

## Button

We have already seen an example on a **Button** control, where the content of the but­ton was a simple text. The content of a button can be more complex – for instance an image combined with text – but the overall purpose of a button is of course to enable a user to invoke a certain action. We will later see how you associate such an action with the button, such that the action is invoked when the user clicks on the button.

## TextBlock

A **TextBlock** is just a section of text to be displayed somewhere in the GUI, without being part of another control (e.g. a **Button**). The user cannot as such interact with a **TextBlock**, but it is possible to change the text in a **TextBlock** dynamically, as we will see examples of later. A very simple **TextBlock** will in XAML look like:

**<TextBlock Text="Hello there" />**

The **Text** property is thus the holder of the content itself. In addition to this, there are of course a multitude of attributes related to the visual appearance of the text. These attributes are probably easiest to explore in the **Properties** window.

## TextBox

A **TextBox** is to some extent similar to a **TextBlock**, the major difference being that a user can usually enter text into a **TextBox** control (if the control is enabled for recei­ving user input). A simple **TextBox** in XAML could be:

**<TextBox Text="Hello there" Width="100" Height="50" />**

Just as for the **TextBlock**, the text itself can be formatted in various ways. It is quite common to use a **TextBox** together with a **TextBlock**, where the **TextBlock** can con­tain a “lead text”, describing the data you should enter into the following **TextBox**.

## Image

An **Image** control will – not surprisingly – contain an image, and it will usually be a passive element in the GUI. The most important property of an **Image** control is the **Source** property, where you specify the source for the image to display. The source is a URL; it can point to a local file, but also to a destination on the web:

**<****Image Source="http://shortly.be/content/auto\_site\_logo.png" />**

Note that an **Image** control can be used as part of the content of other controls, for instance a **Button**. It is also possible to make an **Image** control responsive, i.e. make it react to being clicked on by the user.

## Slider

The **Slider** control is maybe on the borderline of being a “simple” control, and does not occur that often in GUIs compared to the previous controls. A slider control can be used to set a value between a specified minimum and maximum value. This can be utilised to allow the user to choose a numeric value inside a certain interval, without requiring the user to enter the number into e.g. a **TextBox**, which would require that the application does some validation of the entered data. This can be avoided using a **Slider** control. A simple **Slider** control can look like.

**<Slider Maximum="800" Minimum="100" Value="400"/>**

Whenever the user slides the marker in the **Slider** control, the **Value** property will change accordingly. We will later see how that value can then be used to control the value of properties in other controls.

# Layout controls

The small examples above have not been placed in some sort of context. When a GUI becomes just a bit more complex, we need ways to organise the GUI elements into larger groups, in order to manage the visual layout. A number of **layout controls** are available for this purpose. You may recall that Visual Studio put a **Grid** element into the XAML code per default; the **Grid** control is an example of such a layout control.

## Grid

The **Grid** layout control is used if the overall layout of a window follows a regular row/ column structure, i.e. where all columns have the same number of rows, and vice versa. Placing a **Grid** tag inside the XAML code will create a 1-by-1 grid; if you wish to create a grid with several rows and columns, you will need to add a number of so-called row- and column-definitions to the **Grid**. The below XAML code defines a 3-by-2 grid:

**<Grid>**

**<Grid.RowDefinitions>**

**<RowDefinition/>**

**<RowDefinition/>**

**<RowDefinition/>**

**</Grid.RowDefinitions>**

**<Grid.ColumnDefinitions>**

**<ColumnDefinition/>**

**<ColumnDefinition/>**

**</Grid.ColumnDefinitions>**

**</Grid>**

In the Design view, you will see something like:



There are a number of ways to control how the columns and rows are sized with respect to each other; we will get back to this in the chapter on control properties. Once a suitable structure has been defined, you can add controls inside specific grid cells. You do this using the below syntax:

**<Grid>**

**<Grid.RowDefinitions>**

**<RowDefinition/>**

**<RowDefinition/>**

**<RowDefinition/>**

**</Grid.RowDefinitions>**

**<Grid.ColumnDefinitions>**

**<ColumnDefinition/>**

**<ColumnDefinition/>**

**</Grid.ColumnDefinitions>**

**<Image Grid.Row="0" Grid.Column="1"**

**Source="http://shortly.be/content/auto\_site\_logo.png"/>**

**<Slider Grid.Row="1" Grid.Column="1"**

**Maximum="800" Minimum="100" Value="400" Width="400" />**

**</Grid>**

Since the controls are defined within the **Grid** tag, they are per definition assumed to be positioned within that grid. All specifications of relative positioning of a control – like e.g. alignment – will now be done relative to the grid cell containing the control.

## StackPanel

The **StackPanel** control is in a sense more primitive than the **Grid** control, but also more flexible. Inside a **StackPanel**, you can specify a sequence (or “stack”) of controls, e.g. like this:

**<StackPanel>**

**<Image Source="http://shortly.be/content/auto\_site\_logo.png"/>**

**<Slider Maximum="800" Minimum="100" Value="400" Width="400" />**

**</StackPanel>**

This will “stack” the controls visually on top of each other. This is in itself rarely what you want. However, two additional features make the **StackPanel** really useful:

* You can orient the **StackPanel** to stack the controls either vertically (default) or horizontally
* You can put **StackPanels** inside **StackPanels**

The latter property is as such not something special for the **StackPanel**; if you wish, you can also put a **Grid** control within a **Grid** control. Still, these two features alone make it possible to create quite sophisticated layouts. Also, there is no need to specify the position of an embedded control explicitly (as was the case for the **Grid** control), since the order of the controls themselves specify the position within the **StackPanel**. Only drawback is perhaps that the nesting level of the **Stackpanel** controls tends to become quite deep.

# Control properties

We have already used some of the (many) properties that are available on controls, and we will not try to give a comprehensive description of all properties here. A good way to explore the available properties for various control types is to select a control in the Design view, and then open the **Properties** window (press F4). Also, remember that even though there might be dozens of properties to fiddle with, you will usually only need to use a few of them. In the below, we will give an overview of some impor­tant categories, and describe a few specific properties in more detail.

## Default properties

All controls have a so-called default property. The default property can be set simply by writing its value between the opening and closing tag of the control, like:

**<TextBlock>Hello all</TextBlock>**

The default property for a specific control is usually chosen such that it is a commonly used property. For a **Page** control, the default property is **Content**, which is the reason for the commonly seen error *“The property ‘Content’ is set more than once”*, if you just add a number of controls directly to a page. The error is usually fixed by wrapping the controls into a **StackPanel** tag.

## Complex properties

The term complex is not a formal one here; it is just a common denominator for properties that cannot be set by a single simple value. Consider for instance this definition of the background for a **Button** control:

**<Button.Background>**

**<LinearGradientBrush EndPoint="1,0.5" StartPoint="0,0.5">**

**<GradientStop Color="Black" Offset="0" />**

**<GradientStop Color="White" Offset="1" />**

**</LinearGradientBrush>**

**</Button.Background>**

Here we have a complex definition of the background, so we need to write out the entire hierarchy of “sub-properties” we need to set.

## Attached properties

We have already used attached properties, when we defined the positioning of a control inside a **Grid** control:

**<Image Grid.Row="0" Grid.Column="1"**

**Source="****http://shortly.be/content/auto\_site\_logo.png"/>**

**<Slider Grid.Row="1" Grid.Column="1" Maximum="800" Minimum="100"**

**Value="400" Width="400" />**

The **Grid.Row** property is an attached property for e.g. the **Image** control, since it refers back to enclosing **Grid** control.

## Layout properties

The layout of most controls can be controlled in quite detailed ways, and it can be somewhat confusing to figure out why a certain combination of layout property values result in a certain graphical layout. There are various ways to e.g. specify the size of a control; some specify the size in an absolute measure (e.g. pixels), while others specify a relative size. The commonly used properties **Height** and **Width** can be set in (at least) four ways:

|  |  |
| --- | --- |
| “Auto” | Adjusts according to the controls inside the control in question |
| “\*” | A certain part of the controls total size |
| “60” | 60 physical pixels |
| “3\*” | A certain part of the controls total size (three times larger than “\*” |

On top of this comes the fact that sizes are sometimes relative to the size of the en­clo­sing control, which may again be relative, and so on… The best advice is probably to go for a fairly simple layout initially, which can then be adjusted later.

A first attempt at organising a set of controls often results in the controls being visu­ally mashed up against each other, which is not very visually pleasing. The **Margin** property comes in handy for this problem. If you wish to create a **Button** control with a 10-pixel margin to all sides, it will look like:

**<Button Content="OK" Margin="10,10,10,10"/>**

The margins are specified in the order left-top-right-bottom. Adding just a fex pixels of space between controls will usually make a significant difference.

## Using styles

Once your GUI grows beyond a few controls, you will often be in a situation where you want several controls of the same type – say, a set of **TextBox** controls – to follow the same layout. This can of course be achieved just by explicitly setting all relevant properties for each control, but this can quickly become cumbersome to maintain, if you decide to change the layout. One way around this is to define a **Style**.

A **Style** is a set of settings for certain properties, that you wish to apply to several con­trols of the same type. A **Style** is specified as a so-called **Resource**, typically apply­ing to an entire **Page**, like:

**<Page.Resources>**

**<Style x:Key="TextBoxStyle" TargetType="TextBox">**

**<Setter Property="FontSize" Value="24"/>**

**<Setter Property="Width" Value="300"/>**

**<Setter Property="Margin" Value="5,5,5,5"/>**

**</Style>**

**</Page.Resources>**

Apart from the slightly peculiar **Setter/Value**-syntax, it is pretty straightforward. You can then apply such a style to a control of the specified type, like:

**<TextBox Style="{StaticResource TextBoxStyle}" Text="(Name)"/>**

There are of course more in-depth descriptions of layout properties – and XAML layout considerations in general – available elsewhere, for instance here[[3]](#footnote-3).

# Data Binding

As the previous chapter illustrates, it is by no means a simple task to accurately speci­fy the graphical layout of an application GUI…and we haven’t even begun to use the GUI for anything yet! The GUI is rarely an end-goal in itself; it is just a tool for helping the user to interact with the application. The ultimate goal for most applica­tions is to do some sort of data manipulation, and the GUI is the surface through which the user can manipulate the data. This in turn implies that the GUI elements must somehow be in contact with the data. So how is this achieved?

A central concept in achieving this goal is so-called **data binding**. Data binding covers the idea that data – both in simple and complex forms – can be “bound” to GUI con­trols, such that changes in the data are directly reflected in the GUI controls, and – very importantly – vice versa. If the actual data inside the application comes “out of sync” with the data presented to the user by the GUI, the results can be cata­stro­phic. Imagine a banking application, where the balance of a bank account suddenly is diffe­rent from what the user sees on the screen. Or perhaps a military control system… This problem is almost as old as computer programming itself, and has been handled in various ways. Most of these ways are variants of data binding.

## Simple binding between GUI elements

The simplest form of data binding can be between two GUI controls defined on the same page. In the previous example with an **Image** and a **Slider** control, we can add data binding in the following way (image source omitted for brevity):

**<Image x:Name="theImage" Source="…"**

**Height="{Binding ElementName=theSlider, Path=Value}"**

**Width="{Binding Height}" />**

**<Slider x:Name="theSlider" Maximum="800" Minimum="100" Value="400"**

**Width="400" Height="400"/>**

How should this be read? The two interesting entries are the **“{Binding …}”** entries. The first entry should be read as *“bind the* ***Height*** *of the* ***Image*** *control to the GUI element named* ***theSlider*** *(which is a* ***Slider****), specifically to the* ***Value*** *property”*. With this binding, the height of the image will actually change if we slide the marker in the **Slider** control back and forth. The second entry reads *“Bind the* ***Width*** *property of the* ***Image*** *to the* ***Height*** *property of the (same)* ***Image****”*. This just ensures than the width and height of the image are both changed according to the value of the slider.

Data bindings like these are fairly simple, since they only involve the GUI controls them­­­selves. Things get a bit more complex when GUI elements must be bound to data that is not part of the GUI itself, but rather part of the data “model” contained in the application; that is, the model of whatever domain the application concerns.

## Data binding between GUI elements and model objects

We mentioned in the start of this chapter, that one of the fundamental ideas in this approach to GUI development is the division of specification of appearance and be­ha­­vior. That is, we specify the visual appearance of a GUI element in XAML, and the beha­­vior (including interaction with domain model objects) in C#. However, since the XAML code is transformed into C# and compiled along with the “native” C# code, the GUI controls will effectively be nothing more than C# objects, living in the context defined by the class definition they are part of. A consequence of this fact is that it is fairly straightforward to bind a GUI control to a non-GUI property of the class in which it is defined. Still, this is not the recommended approach. We will in general aim to keep the code-behind files associated with XAML files as small as possible – preferably empty – since there are several advantages of keeping the binding specifi­cations in the XAML code. One such advantage is **portabili­ty**. XAML code can be used in other contexts than applications written in C#, but any logic placed in the code-behind files cannot be ported as easily.

The preferred approach is therefore to specifiy data bindings in the XAML code alone. Fortunately, this is also relatively easy to do. A key concept related to this is the **data context**. We said before that GUI controls are just C# objects living inside a class defi­nition. This also means that it is possible to refer to objects of other classes, e.g. do­main model classes. In XAML, this is done by specifying a **data context** for a GUI con­trol. Data contexts are “hierarchical”, in the sense that if you e.g. set a data con­text for a **Page** control, all GUI controls defined within that **Page** will also be set to use that data context (unless they themselves explicitly set a different data context). A com­mon approach is therefore to set the data context once, for the “outermost” control in a window, typically a **Page** control.

How does this look in practice? Suppose we define a simple domain class called **Car**, that for now only has one very simple property called **Brand**:

**class Car**

**{**

**public string Brand { get { return "Toyota"; } }**

**public Car() { }**

**}**

The class is just defined in the same C# project as the XAML code is defined in, and there is for now nothing special about it. In the XAML code, we add a simple **TextBox** control as well:

**<TextBox Text="(not set)"/>**

The **TextBox** control is defined inside a **StackPanel**, which again is defined inside a **Page**. We now set the data context property for the **Page** control (this must be done within the **Page** start- and end-tag):

**<Page.DataContext>**

**<local:Car/>**

**</Page.DataContext>**

The validity of this code rests on the fact that a namespace declaration for **local** was added to the XAML code at creation:

**xmlns:local="using:GUIExample"**

The **GUIExample** term is simply the name of the C# project we are working with here; in your own project, it will be substituted with the name of your own project. With this in place, we can now change the **TextBox** declaration to:

**<TextBox Text="{Binding Brand}"/>**

This should be read as “the value of the **Text** property of the **TextBox** control is now bound to the value of the **Brand** property of the **Car** object.” With the data con­text in place, this binding is not more complex w.r.t. syntax than binding to a value from another GUI control. Running the application now will indeed update the text in the **TextBox** control as desired:



There are however a couple of issues to consider. First, it was stated above that the **Text** property is bound to the **Brand** property on the **Car** object…. But which **Car** object is that? With a data context specification like this, the application will create a new **Car** object whenever the **Page** object containing the data context specification is created, e.g. when the user navigates to that page in the application. In this simple example, the behavior is always the same, since the **Brand** property always returns the same value (**Toyota**). In a more realistic setup, it will require a more elaborate appro­ach to cause this behavior to create the desired binding. We return to this in the chapter on the MVVM Architecture.

Second, we have now seen that the text in the **TextBox** control is actually updated with the text returned by the **Brand** property on the **Car** object. So, the binding going from **Car/Brand** to **TextBox/Text** (we use this informal **Class/Property** notation as a short way of saying “the **Property** property on the object of type **Class**”) seems to work. But what about the other way around? It would be natural to expect that we could type something into the text box, and expect that **Car/Brand** is updated to this new value. Achieving that does however require a few additions to the code.

First of all, the **Car/Brand** property must be changed, since it doesn’t have a **set**-part yet. The **Car** class then becomes a bit more useful:

**public class Car**

**{**

**private string \_brand;**

**public string Brand**

**{**

**get { return \_brand; }**

**set { \_brand = value; }**

**}**

**public Car()**

**{**

**\_brand = "Toyota";**

**}**

**}**

Next issue is to have a way of seeing if the value is actually changed in the **Car** object itself, if we type something into the text box. In order to do this, we create a simple **TextBlock** control, and bind it to the same property:

**<TextBox Text="{Binding Brand}"/>**

**<TextBox Text=""/>**

**<TextBlock Text="{Binding Brand}"/>**

An empty text box has also been added here; this is just for being able to leave the first **TextBox** control when having completed the typing of the new value. The GUI will as such not react to the typed value until the **TextBox** control “loses focus”, i.e. the screen cursor has been moved to a different control. Since you cannot move the cur­sor to a **TextBlock** control, an extra **TextBox** has been added to enable this. It has no other function.

If we now run the application, and type a new value into the text box, we will not see the desired behavior. This is partly due to an incomplete data binding specification. Data binding can be specified as having a certain “mode”:

|  |  |
| --- | --- |
| **One-time** | Data is only retrieved from the source once. Subsequent changes are not reflected. |
| **One-way** | Data flows from the source to the target, so changes in the source value are reflected in the target, but not vice versa. |
| **Two-way** | Data flows from the source to the target, and vice versa. Changes in both source and target values are reflected in the counterpart as well. |

We obviously want a two-way binding for the **TextBox** control. That is, however, not the default value (default is one-way). We must therefore explicitly set the data bind­ing mode for the **TextBox** control:

**<TextBox Text="{Binding Brand****, Mode=TwoWay}"/>**

All seems to be in place now, but if we run the application, we still don’t see the desi­red behavior… If you place a breakpoint in the **set**-part of the **Brand** property, you will indeed find that the **set**-part is called, and the instance field is updated as it should. What is missing?

The last part of the puzzle is to enable the source object (here the **Car** object) to sig­nal to the outside world that the value of a property has been changed. Even though the **TextBlock** con­­trol has a binding to the **Brand** property, it is not automati­cally notifed about changes in the property value! This does happen automatically when you bind to other GUI controls directly in XAML code, but not for “pure” C# classes.

The mechanism commonly used for enabling such notifications is the **INotifyProper­ty­­Changed** interface, which is part of the .NET class library. If you want a class to be able to signal changes to its properties to those interested in knowing about such changes, the class should implement this inter­face. Doing this for the **Car** class adds some extra code to the class:

**public class Car** **: INotifyPropertyChanged**

**{**

**private string \_brand;**

**public string Brand**

**{**

**get { return \_brand; }**

**set { \_brand = value; }**

**}**

**public Car()**

**{**

**\_brand = "Toyota";**

**}**

**public event PropertyChangedEventHandler PropertyChanged;**

**protected virtual void OnPropertyChanged**

**([CallerMemberName] string propertyName = null)**

**{**

**PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));**

**}**

**}**

The added code deserves some explanation: The interface **INotifyPropertyChanged** actually only consists of a so-called **event** named **PropertyChanged**, as seen in the yellow part of the highlighted code. We will discuss events in more detail later in the notes; for now, you can think of events as a mechanism for letting other parts on an application know that a certain “event” (e.g. a change in the value of a property) has occurred. Once you let a class implement **INotifyPropertyChanged**, Visual Studio will display an option to auto-generate code which implements the interface, plus a bit of extra code (the green part of the highlighted code). This extra bit of code is a method called **OnPropertyChanged**. The effect of calling this method from the **set**-part of a property, will be that all those properties that are bound to the property will be noti­fied that the value of the property has changed. This is exactly what we need.

The only code we need to explicitly add is a single line of code to the **set**-part of the **Brand** property, like this:

**public string Brand**

**{**

**get { return \_brand; }**

**set**

**{**

**\_brand = value;**

**OnPropertyChanged();**

**}**

**}**

If we now run the application, we will indeed see that as soon as we leave the **Text­Box** control (by hitting Tab or clicking in the empty text box), the text block below is updated with the new value. The **Car/Brand** property now “broadcasts” changes to its value to those interested, i.e. those elements that have set up a binding to that speci­fic property.

We can now recap all the steps needed in order to achieve a working two-way binding between a GUI control and an ordinary C# class:

1. Make sure that an appropriate **namespace** definition is part of your XAML page. It will be part of the top lines in the XAML file, and typically look like:

**xmlns:local="using:YourProjectName"**

1. Set the **data context** at an appropriate level in the XAML page, for instance for the **Page** element itself. The code will typically look like:

**<Page.DataContext>**

**<local:YourDomainClass/>**

**</Page.DataContext>**

1. Create a **data binding** for the control. The types of the control ele­ment and the source property should be compatible, i.e. a **Text** element will usually bind to a property of type **string**, and so on. A typical binding will look like:

**<TextBox Text="{Binding YourProperty}"/>**

1. Remember to set the **binding mode** as well. If a GUI element and an object property must be in sync at all times, the binding type should be two-way:

**<TextBox Text="{Binding YourProperty, Mode=TwoWay}"/>**

1. Your domain class must implement the **INotifyPropertyChanged** inteface in order to be able to notify others about changes in its property values:

**public class YourDomainClass : INotifyPropertyChanged**

**{ // Rest of class definition**

**}**

1. Finally, the **set**-part of each relevant property must call **OnPropertyChanged** when the value of the property has been updated:

**public string Brand**

**{**

**get { return \_brand; }**

**set {**

**\_brand = value;**

**OnPropertyChanged();**

**}**

**}**

# Collection Views and Data Binding

Creating bindings to simple properties like e.g. the **Text** property in a **TextBox** con­trol is thus fairly straightforward. These bindings are usually relevant in situations where the data context is suitably represented by a single domain object. If we need to ma­na­ge and interact with a **collection** of domain objects, we need to use other control types suitable for such collections. Such control types do of course exist, but require a bit more elaboration in order to specify appearance and data bindings properly.

## The ListView control – getting started

The **ListView** control is one of those controls that can handle a collection of items. In its simplest form, it can be used completely without data binding:

**<ListView>**

**<ListViewItem>Toyota</ListViewItem>**

**<ListViewItem>BMW</ListViewItem>**

**<ListViewItem>Opel</ListViewItem>**

**<ListViewItem>Volvo</ListViewItem>**

**</ListView>**

This form is of course only relevant when the items in the list are “constant”, i.e. will not change during the lifetime of the application. This is rarely the case in prac­tice, so we need to figure out how to create data bindings for a **ListView**.

Compared with the recap for single-element bindings above, the first and second steps regarding namespace and data context are still needed, in order to make the classes in the project visible to the control. The third step (data binding) will look a bit different. First, we update the **Car** class with a new property **BrandNames**:

**public List<string> BrandNames**

**{**

**get { return new List<string>() {"Toyota","BMW","Opel","Volvo"}; }**

**}**

This list is of course also “constant”, but could in principle have been populated by reading the values from a file or database; the data binding does not care about the origin of the values. The binding itself looks like this:

**<ListView ItemsSource="{Binding BrandNames}" />**

Not much more complicated than binding to a single element, except for the use of the **ListView** property **ItemsSource**. As the name indicates, we must bind this proper­ty to a collection of values, here more specifically a list of **string** values.

This is much more useful than the first example, since the population of the list can be done in any way we wish, as long as the list is ready-to-use when the **ListView** asks for it. A natural next step would be to enable the user to add new elements to the list. This will require some additions to the code. First, we add a **TextBox** control to the GUI, so the user can type in a new brand name (this is just the **TextBox** control from the previous example):

**<TextBox Text="{Binding Brand, Mode=TwoWay}"/>**

**<ListView ItemsSource="{Binding BrandNames}"/>**

Next, we make the **Car** class a bit more general, by adding an instance field to hold the list of brand names:

**private string \_brand;**

**private List<string> \_brandNames;**

**public string Brand**

**{**

**get { return \_brand; }**

**set**

**{**

**\_brand = value;**

**OnPropertyChanged();**

**}**

**}**

**public List<string> BrandNames**

**{**

**get { return \_brandNames; }**

**}**

**public Car()**

**{**

**\_brand = "Toyota";**

**\_brandNames = new List<string>() {"Toyota", "BMW", "Opel", "Volvo"};**

**}**

Is that enough? No, since we do not at any point add any new elements into the list of brand names. However, a small addition to the **set**-part of the **Brand** property can fix this (please note that this is not a very pretty nor user-friendly solution; it only serves to explore what it takes to get the data binding up-and-running…) :

**public string Brand**

**{**

**get { return \_brand; }**

**set**

**{**

**\_brand = value;**

**\_brandNames.Add(\_brand);**

**OnPropertyChanged();**

**}**

**}**

If you run this code, you will find that the situation is similar to what we saw at some stage in the previous example: it seems like the list should be updated, and running the code through the debugger reveals that the list does indeed get updated, but this is not reflected in the GUI… The underlying problem is actually also very similar: the list does not notify the outside world when it gets updated. Here the solution is less straightforward, since it does not make much sense to implement a **set**-part of the **BrandNames** property. It seems that the **Add** method on the **List** class ought to some­­how call the **OnPropertyChanged** method.

The solution is to substitute the use of the **List** class with another class from the .NET class library: the class **Observable­Collection**. This class does what we want; when an element is added or removed from the collection, a notification is broadcasted. Once the substitution is made in the code, the list in the GUI does get updated whenever we leave the text box.

The **Observable­Collection** class is quite convenient, but it does have one major draw­back. If you add or remove an element, notifications are indeed triggered. However, if you just update a value in an existing element, no notifications are triggered! If you need this sort of functionality, a more sophisticated solution must be used.

## The ListView control – displaying items

In the above example, we did not really worry about how each item in the collection (i.e. the **ObservableCollection** object) was displayed. Since each item has the type **string**, it is easy for the **ListView** control to display the items. What if we wish to dis­play a collec­tion of **Car** objects instead? First, we need to create a property in the code where such a collection can be bound to. In order to keep things simple rather than pretty, we just add an additional property to the **Car** class:

**public ObservableCollection<Car> Cars**

**{**

**get**

**{**

**return new ObservableCollection<Car>()**

**{**

**new Car() { \_brand = "Volvo" },**

**new Car() { \_brand = "BMW" },**

**new Car() { \_brand = "Opel" },**

**new Car() { \_brand = "Toyota" }**

**};**

**}**

**}**

The data binding for the **ListView** control also needs to be updated:

**<ListView ItemsSource="{Binding Cars}"/>**

Running the code with these updates does produce a **ListView** control containing four items, but they are not displayed in a particularly useful way:



What has happened is that the **ListView** has called the **ToString** method on each ob­ject, and displays the result of that call. Since we have not overrided **ToString**, this is the result. If we override **ToString** in the **Car** class, we get a better result:

**public override string ToString()**

**{**

**return "This is a " + \_brand;**

**}**



If the data you wish to display in a **ListView** control can be appropriately displayed by a simple string, you probably don’t need something much more advanced than this. It is however possible to specify more elaborate ways to display elements in a **ListView**, by defining a so-called **data template**.

## Defining a data template

The data template is a way of specifying the visual appearance of a single item in the **ListView**. Formally, the data template is set as the value of the **ItemTemplate** proper­ty of a **ListView** control, like this:

**<ListView ItemsSource="{Binding Cars}">**

**<ListView.ItemTemplate>**

**<DataTemplate>**

**(your item layout specification)**

**</DataTemplate>**

**</ListView.ItemTemplate>**

**</ListView>**

Inside the **DataTemplate** tag, you specify the layout of an item. You can use all of the available control types without restriction. If you e.g. want to display each **Car** item as a small picture followed by the brand name, it could look like this:

**<DataTemplate>**

**<StackPanel Orientation="Horizontal">**

**<Image Source="{Binding ImageSource}" Height="80" Width="80"/>**

**<TextBlock Text="{Binding Brand}"/>**

**</StackPanel>**

**</DataTemplate>**

Assuming that the **ImageSource** property has been added to the **Car** class, and that some suitable images have been made available, the **ListView** will now look like:



Since we have chosen to jam everything into the **Car** class, the data template does not need to be told explicitly that the bindings refer to properties on the **Car** class. If the class structure is more elaborate, it is possible to state explicitly what the type of the items in the **ListView** is:

**<DataTemplate** **x:DataType="local:Car">**

**<StackPanel Orientation="Horizontal">**

**<Image Source="{Binding ImageSource}" Height="80" Width="80"/>**

**<TextBlock Text="{Binding Brand}"/>**

**</StackPanel>**

**</DataTemplate>**

The **x:** syntax just implies that we are referring to a specific part of the namespaces included at the top of the XAML file.

## The ListView control – binding to SelectedItem

A final (in this text) useful feature of the **ListView** control is the ability to bind to the currently selected item in the control. The **SelectedItem** property on the control can be targeted in a data binding:

**<ListView ItemsSource="{Binding Cars}"**

**SelectedItem="{Binding SelectedCar}">**

**<ListView.ItemTemplate>**

**<DataTemplate>**

**(your item layout specification)**

**</DataTemplate>**

**</ListView.ItemTemplate>**

**</ListView>**

A **ListView** control is often used in a so-called **Master/Details view**. In this category of views, the user can select an item in the **ListView** control, and see further details of the selected item in a different part of the view. If we imagine that the **Car** class con­tains some additional properties, we could add some additional controls to the page that until now only contains the **ListView** control:

**<StackPanel Orientation="Horizontal">**

**<ListView ItemsSource="{Binding Cars}"**

**SelectedItem="{Binding SelectedCar, Mode=TwoWay}">**

**<ListView.ItemTemplate>**

**<DataTemplate x:DataType="local:Car">**

**<StackPanel Orientation="Horizontal">**

**<Image Source="{Binding ImageSource}"**

**Height="50" Width="50"/>**

**<TextBlock Text="{Binding Brand}"/>**

**</StackPanel>**

**</DataTemplate>**

**</ListView.ItemTemplate>**

**</ListView>**

**<StackPanel>**

**<StackPanel Orientation="Horizontal">**

**<TextBlock Text = "Brand"/>**

**<TextBlock Text = "{Binding SelectedCar.Brand}"/>**

**</StackPanel>**

**<StackPanel Orientation="Horizontal">**

**<TextBlock Text = "Color "/>**

**<TextBlock Text = "{Binding SelectedCar.Color}"/>**

**</StackPanel>**

**<StackPanel Orientation="Horizontal">**

**<TextBlock Text = "Seats "/>**

**<TextBlock Text = "{Binding SelectedCar.Seats}"/>**

**</StackPanel>**

**<StackPanel Orientation="Horizontal">**

**<TextBlock Text = "Price "/>**

**<TextBlock Text = "{Binding SelectedCar.Price}"/>**

**</StackPanel>**

**</StackPanel>**

**</StackPanel>**

With this setup, we get something that is at least a rough draft of a Master/Details view for a **Car** domain object:



There is obviously some work to do with regards to the visual presentation of the data, but the essential wiring with respect to data binding is largely in place now.

## The GridView

Once we know how to deal with a **ListView** control, there is actually not that much more to say about a **GridView** control. The considerations that applied to **ListView** are also valid here, so it is mostly a matter of the visual presentation. With a little bit of adjustment, a **GridView**-version of the previous example could look like:

**<GridView ItemsSource="{Binding Cars}">**

**<GridView.ItemTemplate>**

**<DataTemplate>**

**<StackPanel HorizontalAlignment="Center">**

**<Image Source="{Binding ImageSource}"**

**Height="200" Width="200"/>**

**<TextBlock FontSize="48" Text="{Binding Brand}"/>**

**</StackPanel>**

**</DataTemplate>**

**</GridView.ItemTemplate>**

**</GridView>**

The visual result is:



Again, some visual polishing would probably be in order, but the fundamental setup is almost identical to the setup we developed for the **ListView** control example.

# Commands

So far, we have mostly concentrated on how to bind GUI controls to existing data, in the form of domain model objects. We did have a single example of how to add an element to a list, but it was done in a haphazard way inside the **set**-part of a property. That is definitely not a recommendable approach for a real system. We will therefore now look closer at how to perform modifications to a domain data model, e.g. how to delete a domain object through a GUI.

## Deleting a domain object

As such, the actions we wish to perform are not particularly complicated. Suppose we now have a fuller **Car** domain class, like:

**public class Car**

**{**

**private string \_licensePlate;**

**private string \_brand;**

**private string \_model;**

**private string \_imageSource;**

**private string \_color;**

**private int \_seats;**

**private int \_price;**

**// ...and so on (Properties, etc.)**

**}**

In this class, we assume that the **LicensePlate** property (associated with the **\_license­Plate** instance field) defines a unique key for **Car** objects, i.e. no two **Car** objects will have the same value for **LicensePlate**. Also, we have created a class **CarCatalog**, which holds a collection of **Car** objects (in an **ObservableCollection**), like:

**public class CarCatalog : INotifyPropertyChanged**

**{**

**private ObservableCollection<Car> \_cars;**

**private Car \_selectedCar;**

**public ObservableCollection<Car> Cars**

**{**

**get { return \_cars; }**

**}**

**// ...and so on**

**}**

How would we go about implementing functionality for deleting a **Car** object from our model? Since we assume that the collection of car objects is indeed that entity which represents the actual cars we have in our model, deletion of a specific car will amount to de­le­ting the corresponding **Car** object from the **CarCatalog** object. We thus assume that there will always be exactly one **CarCatalog** object present, and we only need to dele­te the **Car** object from that object. Since we have defined that the **LicensePlate** field uniquely maps to a **Car** object, we can add a **Delete** method to our **CarCatalog** class:

**public void Delete(string licensePlate)**

**{**

**// Implementation of deletion functionality**

**}**

The implementation details of the deletion are as such not interesting, so we will not detail them further.

With this functionality available on the **CarCatalog** class, the remaining issues are:

* How do we select the value for **LicensePlate** to be used in a call of **Delete**?
* How is the deletion functionality activated from the GUI?

If we assume that we still have some sort of Master/Details view set up for cars, a natural way of selecting the car targeted for deletion could be to select a car in the Master view (i.e. a list view), and then have a button labeled **Delete** available for the user to click on. That is, when the **Delete** button is clicked, the **Car** object correspond­ing to the selected item in the list view should be deleted:



We have already seen that it is fairly easy to bind the selection in the list view to e.g. a **SelectedCar** property on the **CarCatalog**, so once we click on the **Delete** button, we can easily retrieve a value for **LicensePlate** to use in the call to **Delete**:

**Delete(SelectedCar.LicensePlate);**

There is a small problem lurking here; what if no **Car** object is currently selected? We’ll get back to that in a moment. Right now, the main issue is how to invoke this call of **Delete**, i.e. how do we bind the action of clicking on the **Delete** button to the invocation of the method? The prefer­red way to achieve this is to encapsulate the deletion code into a so-called **Command** object.

## The ICommand interface

A **Command** object is in this context an object which implements the **ICommand** inter­face. This interface is part of the .NET class library, and contains two methods and an event, similar to the event we saw in the code generated for **INotifyProperty­Chan­ged**. A simple class **Command** that inherits from **ICommand** could look like:

**public class Command : ICommand**

**{**

**public bool CanExecute(object parameter)**

**{**

**// Should return whether or not the command**

**// can currently be executed.**

**}**

**public void Execute(object parameter)**

**{**

**// The code to execute**

**}**

**public event EventHandler CanExecuteChanged;**

**}**

This is in itself not very useful, but illustrates the elements contained in the interface. The more important point is that we can create properties on a domain class like **Car­Catalog**, that have the type **ICommand** and return a **Command** object:

**public ICommand DeletionCommand**

**{**

**get { return new Command(); }**

**}**

We will implement a more proper deletion command class in a moment, but the point is that we can now bind the **DeletionCommand** property to a **Delete** button:

**<Button Content="Delete" Command="{Binding DeletionCommand}"/>**

A **Button** control has a **Command** property, which can be bound to a corresponding property on a non-GUI object, if the property is of type **ICommand**. The consequence of this binding is that whenever the user clicks the **Delete** button, the application will retrieve the object returned by the bound-to property, and call its **Execute** method! This is the essence of the command concept; we “encapsulate” a piece of function­ality – often some business logic – inside a command object, such that the function­ality will be activated when the **Execute** method in the command object is called. Creating a command object will thus not in itself execute the functionality, but rather make it available for execution later, when a certain “event” happens (e.g. the user clicking on a **Button** control).

The **CanExecute** method supplements this idea; it is intended to return **true** if the application is in a state where execution of the functionality is allowed, otherwise **false**. If a GUI control binds to a command object, the GUI control will be disabled if **CanExecute** returns **false**. This is quite user-friendly, since the user will never experi­ence being able to click on e.g. a **Button**, and then being told that the functionality cannot be executed anyway. With this knowledge, we can now get much closer to a proper implemention of a deletion command class:

**public class DeleteCommand : ICommand**

**{**

**private CarCatalog \_carCatalog;**

**public DeleteCommand(CarCatalog carCatalog)**

**{**

**\_carCatalog = carCatalog;**

**}**

**public bool CanExecute(object parameter)**

**{**

**return \_carCatalog.SelectedCar != null;**

**}**

**public void Execute(object parameter)**

**{**

**\_carCatalog.Delete(\_carCatalog.SelectedCar.LicensePlate);**

**}**

**public event EventHandler CanExecuteChanged;**

**}**

Several elements have now been added to the original outline of the class:

* An instance field **\_carCatalog** of type **CarCatalog**
* A constructor, taking a parameter of type **CarCatalog**
* An implementation of **Execute**
* An implementation of **CanExecute**

Since the deletion operation is performed on a **CarCatalog** object, the command object needs a reference to such an object. Once the reference has been set – this is done in the **DeleteCommand** constructor – it is fairly straightforward to implement the functionality for **Execute** and **CanExecute**, respectively:

* **Execute** simply calls the **Delete** method defined in the **CarCatalog** class, using the license plate from the cur­rently selected **Car** object as parameter.
* **CanExecute** – which decides if we can go ahead with the operation at all – only returns **true** if the **SelectedCar** property actually refers to a **Car** object (i.e. is not **null**)

It might seem a bit risky for **Execute** to try to obtain the **LicensePlate** property from the **SelectedCar** property, since that property might be set to **null**. However, that is exactly the condition that **CanExecute** tests for! **Execute** can thus only be executed if a **Car** object is actually selected.

The implementation of the **DeleteCommand** class is now finished. What remains is to create a **DeleteCommand** object in a proper place, and tie it to the view using data binding. Considering the first issue, there is not really any other place to create a **DeleteCommand** object than inside the **CarCatalog** class. But a **DeleteCommand** ob­ject also needs a reference to a **CarCatalog** object… So, can a **CarCatalog** object create a **DeleteCommand** object, and give it a reference to…itself? Yes, indeed it can! In the **CarCatalog** class, we add a new instance field:

**private DeleteCommand \_deleteCommand;**

In the **CarCatalog** constructor, we then add:

**\_deleteCommand = new DeleteCommand(this);**

The keyword **this** means “myself”, i.e. an object can provide a reference to itself in this manner. This is actually a fairly common strategy; an object creates another ob­ject, and hands it a reference to itself in the process. The question of whether this is the correct strategy for this situation is a question we will revisit later in this chapter.

All that remains is to make the **DeleteCommand** object available for a view to bind to. We defined a property **DeletionCommand** in the **CarCatalog** class earlier, and we only need to update the implementation of that property:

**public ICommand DeletionCommand**

**{**

**get { return \_deleteCommand; }**

**}**

It looks like we are done now. However, if we run the application, the **Delete** button is now always disabled… The reason for this is somewhat subtle. When the applica­tion is launched, the page (i.e. view) containing the list view is created. Since the data con­text for the page is **CarCatalog**, a new **CarCatalog** object is created, which in turn will create a new **DeleteCommand** object. Since the **DeleteCommand** object is bound to the **Delete** button, the method **CanExecute** will be called once during this initiali­sa­tion phase. At this point, the **SelectedCar** property will return **null**, since no select­ion has been made yet! This in turn causes **CanExecute** to return **false**, thereby disab­ling the **Delete** button.

This reaction is as such correct, since the **Delete** button should be disabled in this state. The problem is that changing the state – by selecting a **Car** object in the view – does not change the state of the **Delete** button. The reason is that noone has told the command object that things have now changed, in a way that might yield a different result if **CanDelete** is called again. This predicament is the reason for the **CanExecute­Changed** event property in the **ICommand** interface. When the state of the **CarCata­log** changes in a way that makes it relevant to refresh the state of the button – i.e. call **CanExecute** again – the event must be “raised”. This ought to happen exactly when the selection in the list view changes. Making it happen requires a couple of changes in the code.

First, we add a new method **RaiseCanExecuteChanged** to the **DeleteCommand** class:

**public void RaiseCanExecuteChanged()**

**{**

**CanExecuteChanged?.Invoke(this, EventArgs.Empty);**

**}**

We can then call this method when the selection in the list view changes. Since the **SelectedItem** property of the **ListView** control is bound to the **SelectedCar** property in the **CarCatalog** class, we need to update the **set**-part of that property with a call of the **RaiseCanExecuteChanged** method:

**public Car SelectedCar**

**{**

**get { return \_selectedCar; }**

**set**

**{**

**\_selectedCar = value;**

**OnPropertyChanged();**

**\_deleteCommand.RaiseCanExecuteChanged();**

**}**

**}**

A test run of the application shows that the state of the **Delete** button now indeed reflects whe­ther or not a car is selected in the list view. When a car is seleted and deleted by clicking **Delete**, the car is now deleted from the list view and the **Delete** button becomes disabled, until a new car is selected.

The addition of **RaiseCanExecuteChanged** to **DeleteCommand** is very generic, and it is therefore quite common to create a command base class that implements the **ICom­mand** interface and contains the **RaiseCanExecuteChanged** method. You could e.g. create a command base class called **CommandBase**, like this:

**public abstract class CommandBase : ICommand**

**{**

**public abstract bool CanExecute(object parameter);**

**public abstract void Execute(object parameter);**

**public event EventHandler CanExecuteChanged;**

**public void RaiseCanExecuteChanged()**

**{**

**CanExecuteChanged?.Invoke(this, EventArgs.Empty);**

**}**

**}**

A derived class will then only need to implement **Execute** and **CanExecute**, and a pro-per constructor.

The concept of command objects may seem like an overly complex way of invoking func­tion­ality, that often only amounts to a couple of lines of code. There’s not really any way around it. Properties on GUI controls can bind to properties – not methods – on other objects, so the functionality-wrapped-into-command-objects style is hard to avoid. Furthermore, the use of command objects does actually provide several advan­tages on its own. Since a command object encapsulates a piece of functionality with­out know­ledge about the context in which it is used, it is pretty easy to set up tests for a com­mand class in an artificial test environment.

Another advantage of command objects is that they can be handled like any other kind of object; we can e.g. set a command object into a queue of commands waiting for execution, if we are in a situation where commands cannot always be executed immediately. We could also imagine that facilities for e.g. executing commands asyn­chronously (meaning that we don’t wait for the execution of a command to complete before proceeding with other parts of the code) can be implemented in a general way, if all relevant functionality is wrapped into command objects. We could even imagine saving a set of waiting commands to a file, and later on read them again for execution, when the needed resources are available.

Yet another – maybe not so obvious – advantage comes from the fact that we can bind as many GUI controls to a command property as we wish. Often the GUI will allow you to invoke a functionality in several ways; we could imagine that a deletion functionality could be invoked by clicking a button, but also by choosing a menu item. The availabi­lity of such GUI elements should of course be consistent, such that if a **Delete** button is disabled, the corresponding menu item is also disabled. This feature comes for free if the relevant GUI controls all bind to the same command property.

# Exercises

|  |  |
| --- | --- |
| **Exercise** | AD.1.0 |
| **Project** | ExamAdmV10 |
| **Purpose** | Create data bindings between GUI controls |
| **Description** | The project contains a simple GUI for an exam administration system. In this version, you can just type in a name, a subject and a test score for an exam. The data is entered through two text boxes and a slider control. |
| **Steps** | 1. Open the project, and open the MainPage.xaml file. Even though the file contains quite a bit of XAML, we only need to focus on the three named controls, with the names **student­Name**, **subject** and **score** (two **TextBox** controls and a **Slider** control). Make sure you can find these three controls in the XAML code. 2. We want to bind three **TextBlock** controls to the value of the three named controls. The three **TextBlock** controls are all part of the top line of the GUI, which consists of a total of six **TextBlock** controls. For each of the three relevant **TextBlock** controls, figure out which specific named control to bind to. 3. Now create the actual bindings, using the syntax described in the notes (for a **Slider**, you bind to the **Value** property; for a **TextBox**, you bind to the **Text** property). 4. Test that your bindings work as expected |

|  |  |
| --- | --- |
| **Exercise** | AD.1.1 |
| **Project** | ExamAdmV11 |
| **Purpose** | Create data bindings between GUI controls and a domain object |
| **Description** | The project is identical to the project from ExamAdmV10, except that a class **Student** has been added. Right now, the constructor in **Student** just sets the properties to some fixed values. |
| **Steps** | 1. Open the MainPage.xaml file, and add a data context to the **Page** control, specifying **Student** as the data context. See the notes for the syntax for adding a data context. 2. Bind the three relevant **TextBlock** controls (the same as in the previous exercise) to the corresponding properties on the **Student** class. Again, see the notes if you cannot remember the syntax for this. 3. Also create bindings for the three named controls, such that each control – or more precisely; the relevant property in each control – is bound to the corresponding **Student** property. 4. Run the application, and check that the bindings work as expected. Try to change the values as well, by entering new values in the text boxes, and moving the slider. Are the changes reflected in the text line at the top? |

|  |  |
| --- | --- |
| **Exercise** | AD.1.2 |
| **Project** | ExamAdmV12 |
| **Purpose** | Create two-way data bindings between GUI controls and a domain object |
| **Description** | The project starts off where the previous exercise left off. The project does contain data bindings, but changes in the values are still not reflected in the rest of the GUI. |
| **Steps** | 1. Open and run the application. Confirm that changes in the values are not reflected in the top text line. 2. Open the **Student** class. All three properties now have a **set**-part as well. Now let **Student** inherit from the **INotifyProperty­Changed** interface, and implement the **OnPropertyChanged** method (if ReSharper if installed, Visual Studio can generate the code for you. If not, you can simply copy-paste the code from the notes). 3. Run the application again – are value changes now reflected in the text line? 4. For each property in **Student**, add a call to **OnPropertyChanged** to the **set**-part of the property, after the value has been set. 5. Run the application again – are value changes now reflected in the text line? 6. For each of the three bindings for the named controls (not the **TextBlock** controls), update the binding mode to **TwoWay**. 7. Run the application again – are value changes now reflected in the text line? 8. Why don’t we need to update the three **TextBlock** bindings to being **TwoWay**? |

|  |  |
| --- | --- |
| **Exercise** | AD.1.3 |
| **Project** | ExamAdmV13 |
| **Purpose** | Create data bindings between collection-oriented GUI controls and domain object collections |
| **Description** | We now introduce a **StudentCollection** into the application. For now, it contains a **Student** list (with one entry), and a subjects list (initia­lised with five entries). The class also has a property **SelectedStudent**, that for now just returns the single entry in the **Student** list. |
| **Steps** | 1. Open the **StudentCollection** class, and make sure you under­stand the in­stance fields and properties it contains (except the **NewSubject** property) 2. Open the MainPage.xaml file. The bindings are now a bit more complex, since the data context is now **StudentCollection**. Most properties are now bound to the corresponding property on the **SelectedStudent** property (i.e. **Student** object). Make sure you understand the new bindings. 3. Run the application (ignoring the two extra lines beneath “Score”). The application does work, since updates to **name**, **score** and **subject** are reflected in the top text line (try it!). 4. In the “New subject” line, the intention is that when a new subject is entered, it should show up in the “Subject” combo-box. Confirm that this is not the case right now (remember that you must leave the text box, before the update is triggered). 5. The “No. of subjects” field tells how many entries the **\_subjects** list in **StudentCollection** contains. Right now, the number stays at 5. Figure out how to create a binding for the text box next to the “New subject” text, such that a new entry is indeed added to **\_subjects** (Hint: Take a look at the **NewSubject** property in **StudentCollection**). 6. Once this binding works, the number should increase every time a new subject is added. Still, the new subjects do not show up in the combo-box. Figure out why this is the case, and fix it. (Hint: Are we using the correct collection class for \_**subjects**?) |

|  |  |
| --- | --- |
| **Exercise** | AD.1.4 |
| **Project** | ExamAdmV14 |
| **Purpose** | Create a data template for presenting objects in a **ListView** |
| **Description** | The application main view contains a **ListView** control, where the **Items­Source** property is bound to the **Students** property on the **Student­Collection** class. The list contains five students. However, the presen­tation of the students in the list view is not optimal. |
| **Steps** | 1. Try to improve the presentation of **Student** objects in the list view, by providing an implementation of **ToString** in the **Student** class. 2. Improve the presentation further by defining a data template for the **Student** class (Tip: you can probably use the properties in the **Student** class for this purpose). |

|  |  |
| --- | --- |
| **Exercise** | AD.1.5 |
| **Project** | ExamAdmV15 |
| **Purpose** | Create a Master/Details view |
| **Description** | The application main view again contains a **ListView** control, where the **Items­Source** property is bound to the **Students** property on the **Student­Collection** class. The list contains five students, and a reason­able data template has been provided for presentation. The **Student** class has however been extended with several additional properties. |
| **Steps** | 1. Create the Details part of a Master/Details view (the **ListView** is the Mas­ter part), such that all details of a given **Student** object are shown in the Details part. The Details part should show the details of the **Student** which is currently selected in the list view. (Tip: use the example in the notes for inspiration). 2. Use the styles **TextBlockStyle** and **TextBoxStyle** to specify the appearance of the Details part. |

|  |  |
| --- | --- |
| **Exercise** | AD.1.6 |
| **Project** | ExamAdmV16a |
| **Purpose** | Add deletion functionality to a Master/Details view |
| **Description** | The application contains a working Master/Details view for the **Student** class. We now wish to add functionality to delete a student |
| **Steps** | The steps needed to create deletion functionality are very similar to the steps described in the notes. Most changes are done in the **StudentCatalog** class. You can assume that the **Name** property can be used as a key for **Student** objects.  In the **DeleteCommand** class:   1. Implement a constructor for **DeleteCommand**, which takes a reference to a **StudentCatalog** object as parameter. 2. Add an instance field **\_catalog** of type **StudentCatalog**, and initialise it with the parameter from the constructor. 3. Implement **Execute** and **CanExecute**. The implementations will be almost identical to the implementations shown in the notes.   In the **StudentCatalog** class:   1. Change the **\_students** instance field, such that it uses **ObservableCollec­tion** instead of **List** 2. Add a **\_deletionCommand** instance field, of type **DeleteCommand** 3. Initialise the **\_deletionCommand** instance field in the constructor, by creating a new **DeleteCommand** object. 4. Add a **DeletionCommand** property, similar to the **Deletion­Command** property shown in the notes 5. Update the **set** part of the **SelectedStudent** property, such that it calls**\_deletionCommand.RaiseCanExecuteChanged**()   In the **MainPage.xaml** file:   1. Find the **Button** controlwhich implements the **Delete** button, and bind its **Command** property to **DeletionCommand** 2. Check that you can now delete students from the view! |

|  |  |
| --- | --- |
| **Exercise** | AD.1.7 |
| **Project** | ExamAdmV17 |
| **Purpose** | Consider how insertion/editing functionality can be added to the application |
| **Description** | The application contains a working Master/Details view for the **Student** class, with deletion functionality. A natural extension of the application could be to add functionality for editing existing students, and adding new students |
| **Steps** | 1. Consider what it would take in order to add insertion and editing function­ality to the view. Consider for instance:    * How can we enable editing of specific fields?    * Should all fields be editable?    * Should editable fields be editable all the time?    * How can we manage the “editability” of fields in the Details view?    * What are the detailed steps needed in order to create a new student?    * What sort of validation will be needed when creating a new student (remember we assume that student names are unique)? 2. If you are up to the challenge, feel free to start on the actual implemen­tation of the functionality 3. Finally, consider if the **Student** and **StudentCollection** classes are appro­priate classes for containing all this functionality. Can you envision a better distribution of the functionality? |

|  |  |
| --- | --- |
| **Exercise** | AD.1.8 |
| **Project** | ExamAdmV18 |
| **Purpose** | Save the day at StudentSoft A/S (see memo below) |
| **Description** | **MEMO**: Finish the Show/Hide details feature in the Student view  **From**: Maurice Fischer (StudentSoft A/S CTO)  **TO**: EASJ Intern (can’t remember the name…)  Hi,  Unfortunately, our main developer on the Exam Administration application died yesterday, due to an unfortunate incident involving a hamster, three small oranges and a large piece of brown cardboard. We would therefore like you to finish up the Show/Hide Details feature he was working on in the Students view. I think it was something about being able to toggle the visibility of parts of the Details view on and off, using a Toggle­Switch or something… Anyway, you can probably figure it out by looking in the C# project, as he said he was “almost done” with it, and he always puts…uhh, used to put comments in the code. I would like a demo of it later today, as we are shipping a new version of the application tomorrow.  Regards,  M. Fischer |
| **Steps** | Do as you’re told… |

1. https://docs.microsoft.com/en-us/windows/uwp/files/file-access-permissions [↑](#footnote-ref-1)
2. For instance at https://msdn.microsoft.com/en-us/library/windows/apps/mt185406.aspx [↑](#footnote-ref-2)
3. https://docs.microsoft.com/en-us/windows/uwp/layout/layouts-with-xaml [↑](#footnote-ref-3)