



89098: "WPF with C# for Terma"

Lab Manual

Wincubate ApS
03-02-2021



Table of Contents

Exercise types	4
Prerequisites.....	4
Module 1: "A Quick Tour of WPF Fundamentals"	5
Lab 1.1: "Layout of WPF UI"	5
Lab 1.2: "Aligning the WPF UI"	6
Lab 1.3: "Transforming the WPF UI" (⭐⭐)	7
Lab 1.4: "Factoring into resources"	8
Lab 1.5: "Factoring into default style"	9
Lab 1.6: "Triggers for TextBoxes" (⭐)	10
Lab 1.7: "Triggers and animations"	11
Module 2: "Data Binding Properties [Foundation]"	13
Lab 2.1: "Scaling images by data binding"	13
Lab 2.2: "Notifying data binding of property changes" (⭐)	14
Module 3: "Data Binding Collections [Foundation]"	15
Lab 3.1: "Master-Detail data binding" (⭐)	15
Data Templates.....	15
Lab 3.2: "More data templates" (⭐⭐)	17
Module 4: "Events and Commands [Introduction]"	18
Lab 4.1: "Bubbling events" (⭐)	18
Lab 4.2: "Commands" (⭐)	19
Module 5: "Control Templates and User-defined Controls"	21
Lab 5.1: "Ellipse control template for Button"	21
Lab 5.2: "Creating a User Control" (⭐⭐)	23
If time permits... ..	23
Module 6: "Threads and Asynchrony in WPF [Foundation]"	24
Lab 6.1: "Tasks" (⭐⭐)	24
Module 7: "MVVM Design Pattern [Foundation]"	25
Lab 7.1: "Editing Customer using MVVM" (⭐⭐)	25
Model Implementation.....	25
ViewModel.....	25
View	25

Model Validation	26
Commands.....	26
If time permits... ..	26
Module 8: “WPF Testing and Debugging”	27
Module 9: “Data Binding [Deeper Dive]”	28
Lab 9.1: “Conversion of values during data binding” (⭐⭐).....	28
Lab 9.2: “Sorting data-bound elements” (⭐)	29
Module 10: “MVVM Design Patterns [Deeper Dive]”	30
Lab 10.1: “SimpleCalculator in MVVM” (⭐⭐⭐)	30
Implement digits functionality	30
Implement the functionality for C.....	30
Implement the Result message box	31
Module 11: “Threads and Asynchrony in WPF [Deeper Dive]”	32
Lab 11.1: “Making Threaded Updates Thread-safe in WPF” (⭐⭐).....	32

Exercise types

The exercises in the present lab manual differs in type and difficulty. Most exercises can be solved by applying the techniques from the presentations in the slides in a more or less direct manner. Such exercises are not categorized further.

However, the remaining exercises differs slightly in the sense that they are not necessarily easily solvable. These are categorized as follows:



Labs marked with a single star denote that the corresponding exercises are a bit more loosely specified.



Labs marked with two stars denote that the corresponding exercises contain only a few hints (or none at all!) or might be a bit more difficult or nonessential. They might even require additional searches for information elsewhere than in the slide presentations.



Labs marked with three stars denote that the corresponding exercises are not expected in any way to be solved. These are difficult, tricky, or mind-bending exercises for the interested participants – mostly for fun! 😊

Prerequisites

The present labs require the lab files accompanying the course to be extracted in

C:\Wincubate\89098

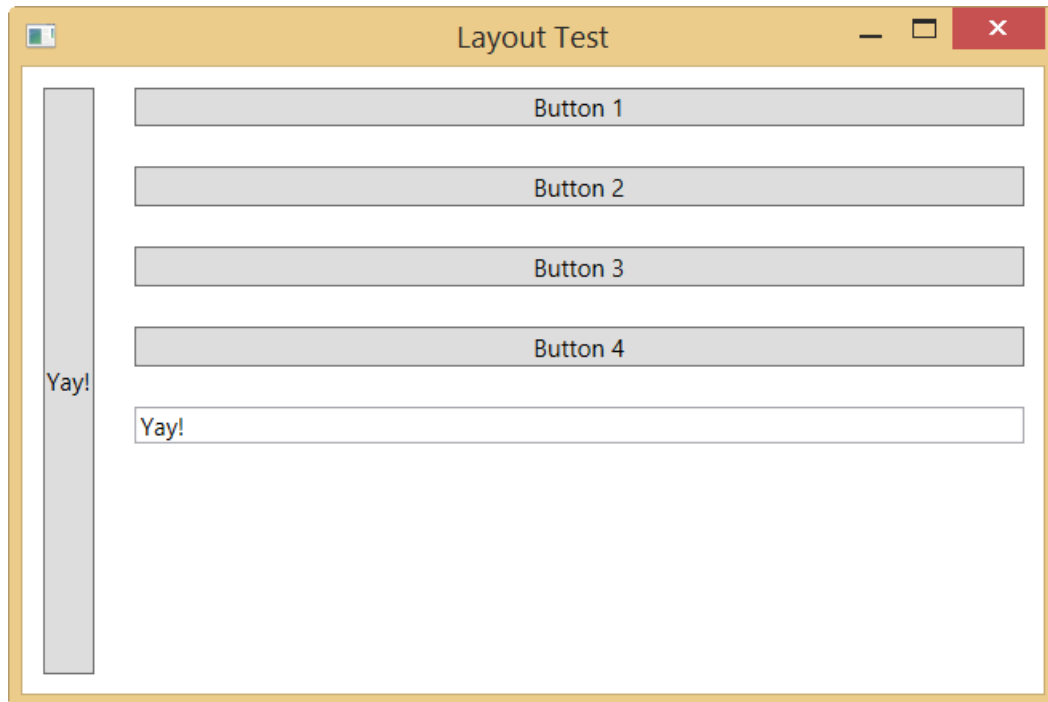
with Visual Studio 2017 (or later) installed on the PC.

Module 1: “A Quick Tour of WPF Fundamentals”

Lab 1.1: “Layout of WPF UI”

This exercise considers various layout controls in WPF as well as simple event handling.

Consider the following screenshot of a WPF application:

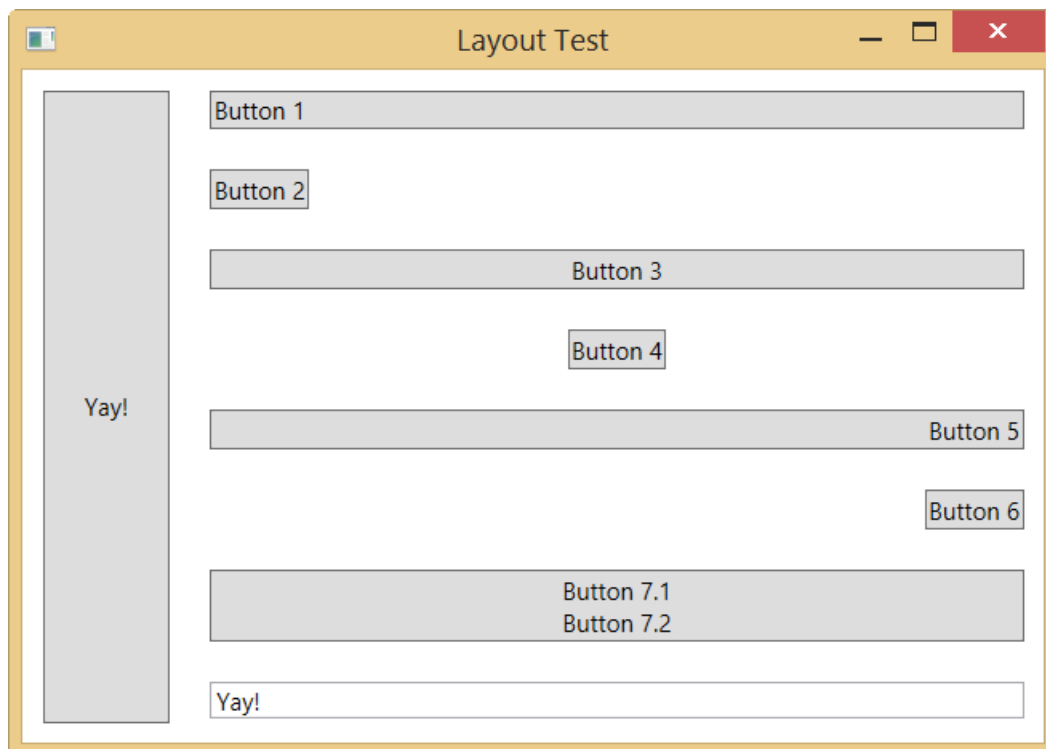


- Use Visual Studio to create a new project of type “WPF Application” in `C:\Wincubate\89098\Module 1\Lab 1.1\Starter` .
- Write some appropriate XAML inside `MainWindow.xaml` producing the user interface depicted above (or something very similar 😊).
- Create a Click-handler for Button 1 such that clicking that button displays a `MessageBox` with the message
`I did it!! ;-)`
- Create a Click-handler for Button 2 such that clicking that button makes left-most button text constitute the text contained in the single `TextBox`.

Lab 1.2: "Aligning the WPF UI"

This exercise continues Lab 1.1 by adding alignment properties to the UI elements.

- Open the starter project in
C:\Wincubate\89098\Module 1\Lab 1.2\Starter ,
which contains the solution to Lab 1.1.
- Modify the markup such that the user interface looks like the screenshot below:

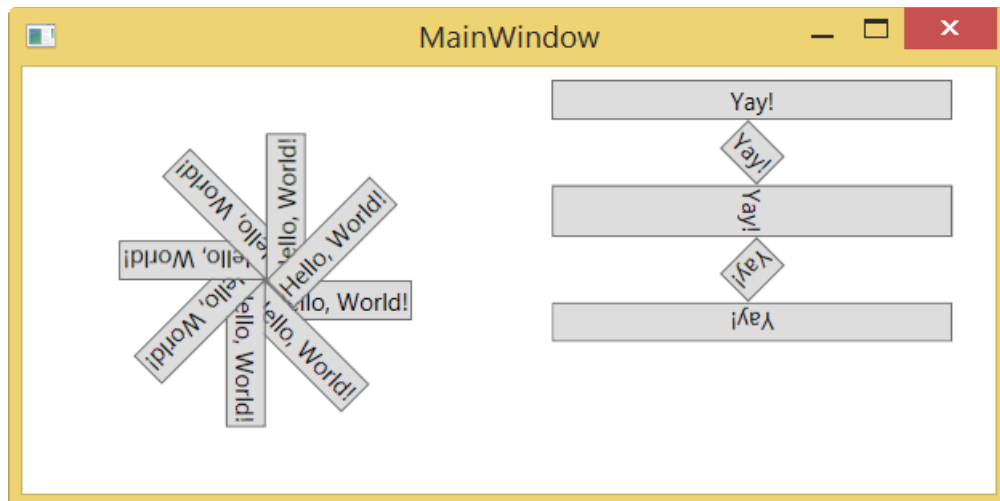


- Note: Make sure to pay attention to the size of the Yay-button. 😊

Lab 1.3: "Transforming the WPF UI" (☆☆)

This exercise considers transforms in WPF as well as composite layouts.

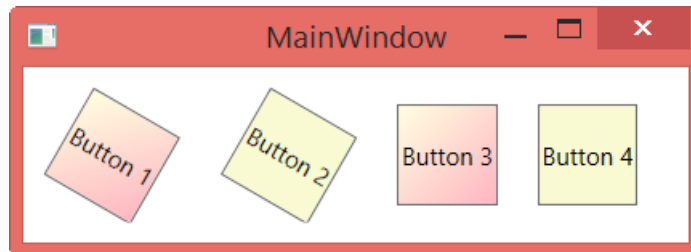
- Can you produce a user interface similar to the screenshot below?



Lab 1.4: "Factoring into resources"

This exercise illustrates how to create a resource and reusing them in markup.

- Open the starter project in
C:\Wincubate\89098\Module 1\Lab 1.4\Starter ,
which contains a project with a window with the following appearance:



The markup contains several repeated resources, which are copied throughout the code. Refactor the code as follows:

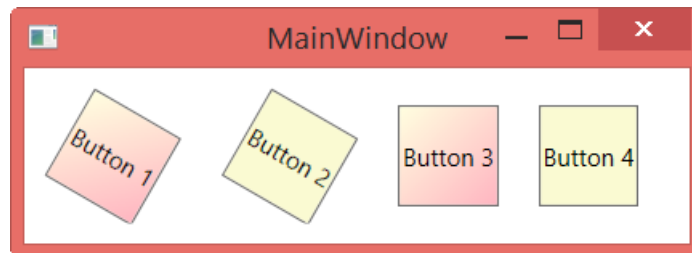
- Create a static resource of type SolidColorBrush and apply it to Buttons 2 and 4.
- Create a static resource of type LinearGradientBrush and apply it to Buttons 1 and 3.
- Create a static resource of type RotateTransform and apply it to Buttons 1 and 2.
- Check that the appearance of the window remains identical to the screenshot above.
- Finally, illustrate the flexibility of the new approach by changing the property values of the various resources to some other values and check how the appearance of the IU changes accordingly.

Lab 1.5: "Factoring into default style"

Note: This lab continues from Lab 1.4. If you have not completed Lab 1.4 yet, you should do so first.

This exercise illustrates how to create a default style for a control type.

- Open the starter project in
C:\Wincubate\89098\Module 1\Lab 1.5\Starter ,
which contains the solution project for Lab 1.4 with a window with the following appearance:



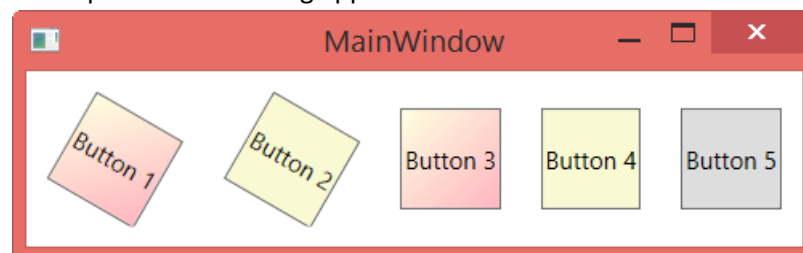
Even though the markup was nicely refactored in Lab 1.4, you could still make the markup a bit more modular and maintainable by defining a proper default style for the buttons.

Refactor the markup in the starter project as follows:

- Define a default style for Button which sets the Background to the solid color brush defined earlier.
- Simplify the markup for the button correspondingly.
- Check that the appearance of the window remains identical to the screenshot above.
- Illustrate the flexibility of the new approach by changing the property values of the various resources to some other values and check how the appearance of the IU changes accordingly.

We will now examine how to "remove" a style otherwise applied to an element.

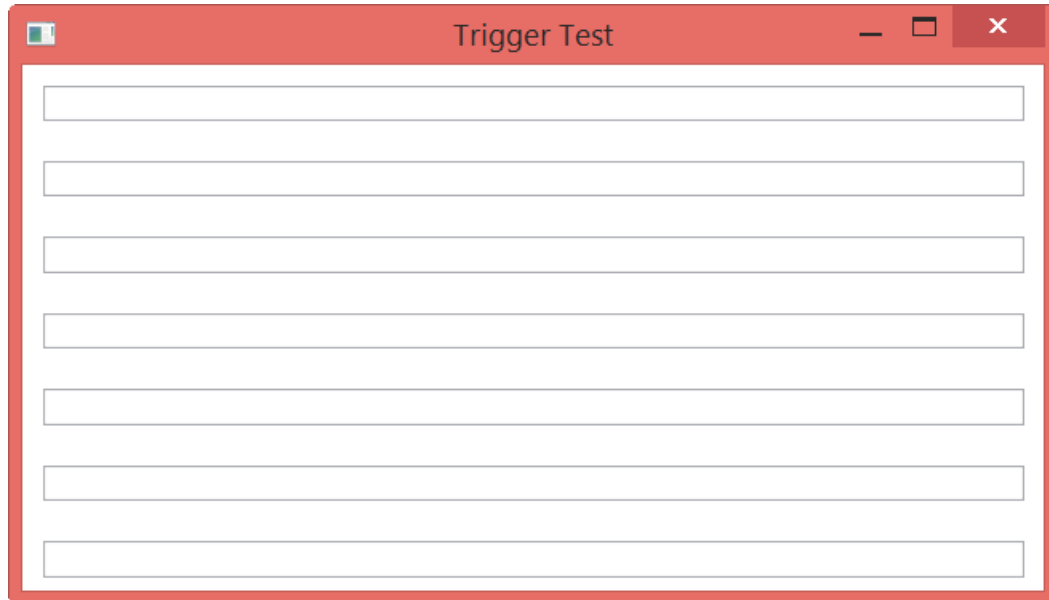
- Extend the markup with yet another button (label it "Button 5"), which does not have the newly defined style applied to it, but instead uses WPF's default Button appearance.
- Ensure that the markup has the following appearance:



Lab 1.6: "Triggers for TextBoxes" (★)

This exercise illustrates how to create triggers for properties.

- Open the starter project in
C:\Wincubate\89098\Module 1\Lab 1.6\Starter ,
which contains a project and a window with the following appearance:



- Create a trigger such that the background of each TextBox turns LightBlue whenever the mouse is located in the given TextBox.
- "Activate" the trigger in a suitable manner such that you are reusing the same trigger for all the TextBoxes.

Lab 1.7: "Triggers and animations"

This exercise investigates animations and event triggers.

- Open the starter project in
C:\Wincubate\89098\Module 1\Lab 1.7\Starter ,
which contains a project and a user interface with a single button.
- Examine the existing markup for the user interface.
- Create appropriate markup to create an animation such that the button rotates 360 degrees in a 5-second interval when clicked.
 - Hint: Create an EventTrigger inside the Button.Triggers collections with actions consisting of a Storyboard with a DoubleAnimation of the RotateTransform's Angle property.

Run your program and test your animation by clicking the button.

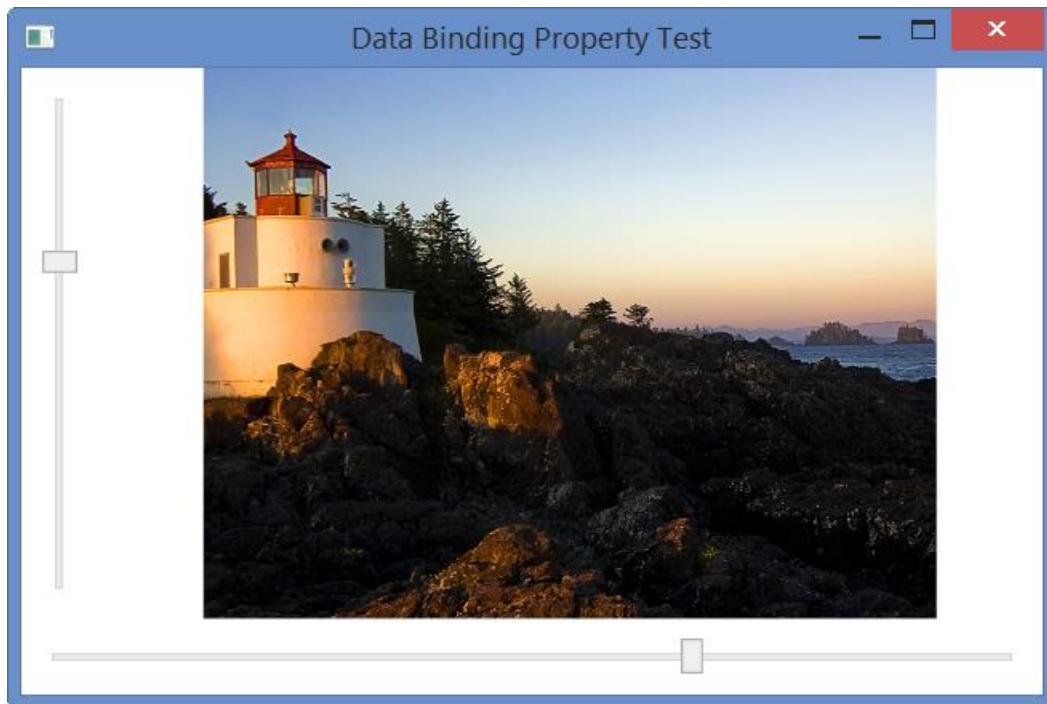
- Extend the markup appropriately such that the button also – repeatedly – changes its background back and forth between White and CornFlowerBlue every 3 seconds indefinitely.

Module 2: “Data Binding Properties [Foundation]”

Lab 2.1: “Scaling images by data binding”

This exercise investigates data binding of single dependency properties to control property values.

- Open the starter project in
C:\Wincubate\89098\Module 2\Lab 2.1\Starter ,
which contains a project and a window with the following appearance:

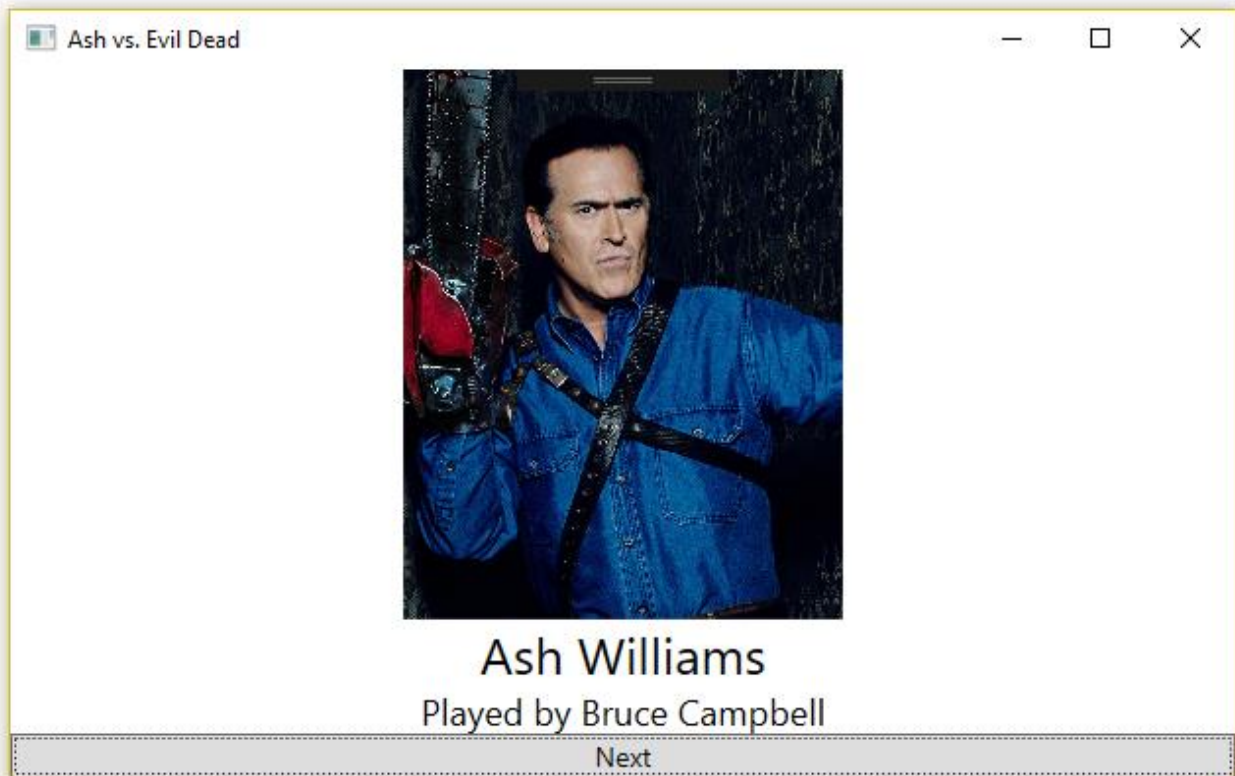


- Create appropriate data bindings such that
 - the horizontal slider controls the ScaleX property of the image render transform
 - the vertical slider controls the ScaleY property of the image render transform
- Test that your solution works
 - Would it be possible by databind the “opposite direction” such that each transform property is data-bound to a slider?

Lab 2.2: "Notifying data binding of property changes" (★)

This exercise investigates how model classes should notify data binding whenever a relevant property of the model object changes.

- Open the starter project in
C:\Wincubate\89098\Module 2\Lab 2.2\Starter ,
which contains a project and a window with the following appearance:



Unfortunately, the currently displayed cast member does not update automatically when the "Next" button is clicked.

- Examine the existing source code and figure out what the problem is.
- Correct this problem by **only** changing the `CastMember` class.
 - Note: You should **not** modify any other part of the code!
- Test that your solution works.

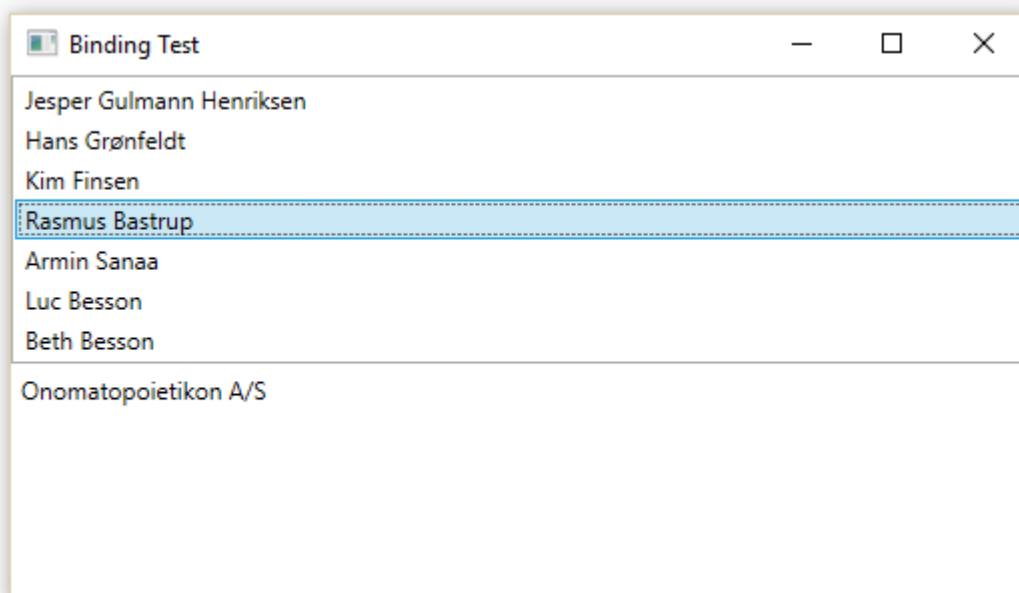
Module 3: “Data Binding Collections [Foundation]”

Lab 3.1: “Master-Detail data binding” (★)

This exercise investigate data binding to collections of data objects.

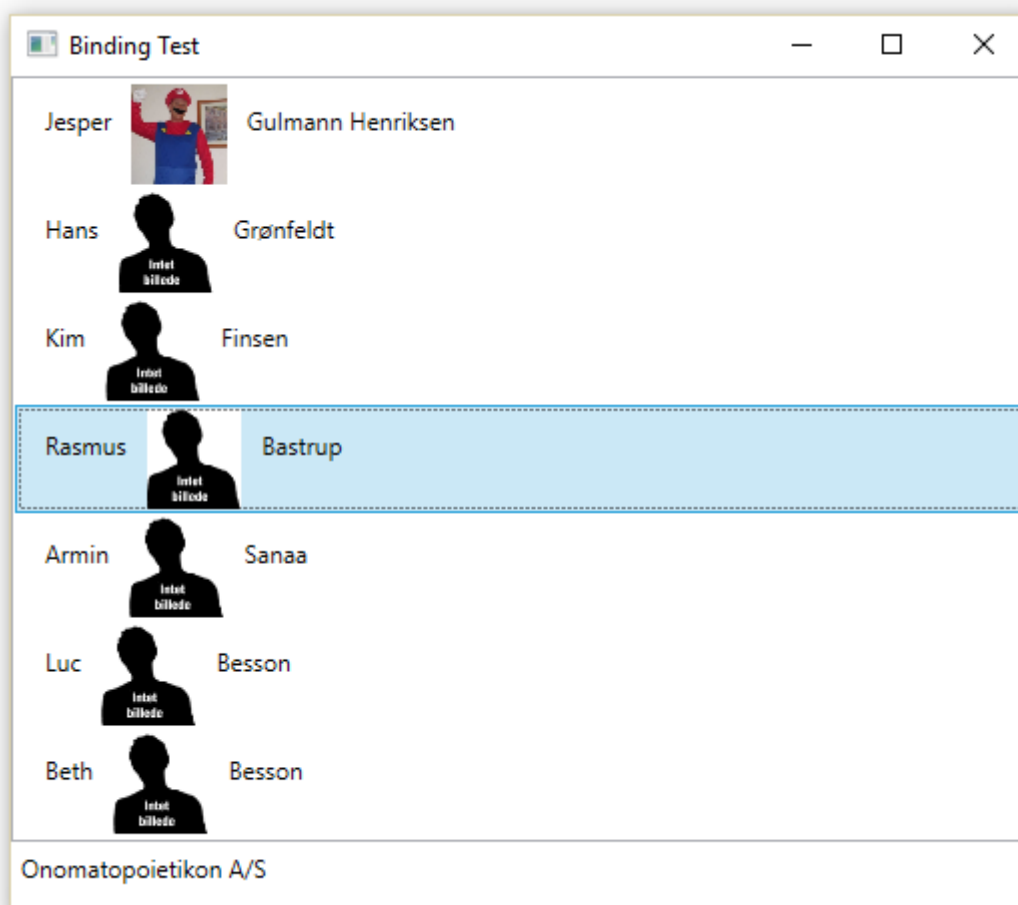
- Open the starter project in
C:\Wincubate\89098\Module 3\Lab 3.1\Starter .
- Modify the XAML markup such that an instance of Participants from the Data-project is created.
- Perform the following steps
 - Databind the DataListBox to Participants
 - Databind the Label to Company
 - Note: Make sure that Company is automatically updated whenever a new item is selected in the ListBox

Your user interface should resemble the screenshot below:



Data Templates

- Create an appropriate data template for Participant og modify the XAML markup such that the user interface will be similar to the following screenshot:



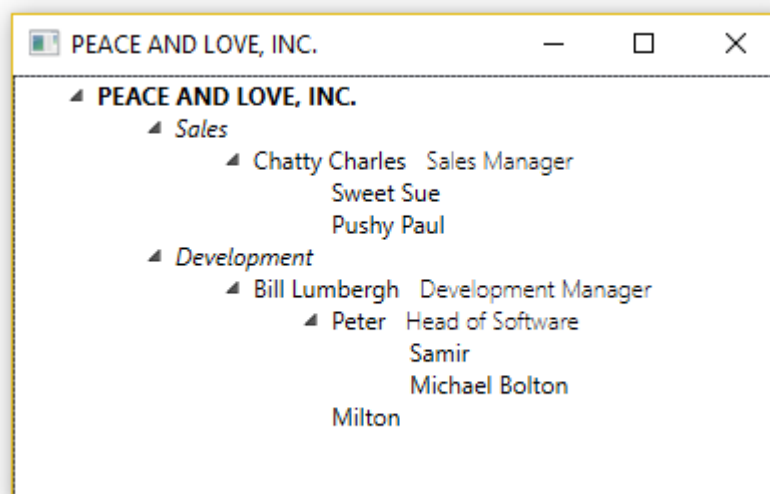
Lab 3.2: "More data templates" (☆☆)

This exercise experiments with "non-recursive" data templates for an organizational hierarchy.

- Open the starter project in
C:\Wincubate\89098\Module 3\Lab 3.2\Starter ,
which in `Organization.cs` contain an example object hierarchy.

It describes an **Organization** consisting of a number of **Departments**, which in turn are managed by a number of **Managers** each responsible for a number of **Employees**.

Unfortunately, some the markup in the `MainWindow.xaml` needs some data template love in order to make it look as follows:



- Create appropriate XAML-fragments such that the application will have an appearance similar to the above.
 - Hint: You only need to add the proper data templates.

Module 4: “Events and Commands [Introduction]”

Lab 4.1: “Bubbling events” (★)

This exercise investigates event handling.

- Create a new WPF project called “SimpleCalculator” in
C:\Wincubate\89098\Module 4\Lab 4.1\Starter
which contains a project and a window with the following appearance:

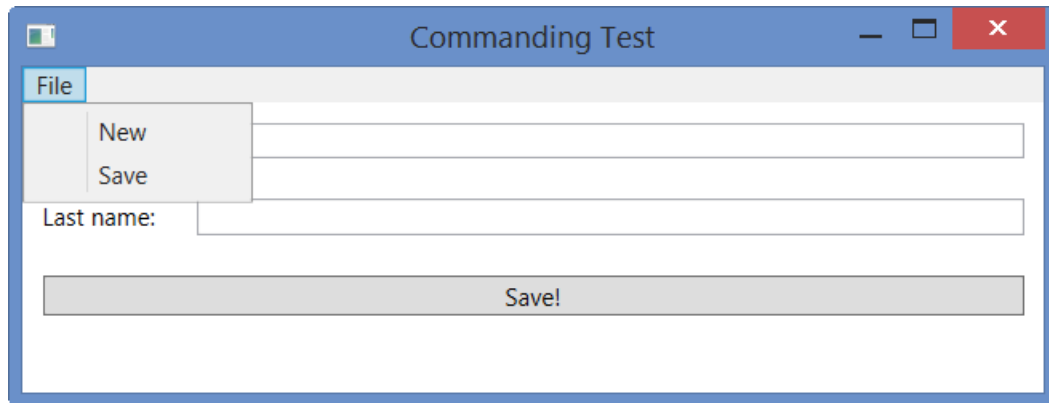


- Extend the existing project with functionality with the following features:
 - The display consists of a TextBox which is to be updated whenever a digit-button is clicked.
 - The ‘C’-button should clear the display.
 - The ‘Result’-button should show the result in a MessageBox and then clear the display.
- Create proper event handling to support the specified functionality of the simple calculator
- Note: You are not allowed to create event handlers on the digit-buttons directly! ☺

Lab 4.2: "Commands" (★)

This exercise deals with handling operations via suitable commands.

- Open the starter project in
C:\Wincubate\89098\Module 4\Lab 4.2\Starter ,
which contains a project and a window with the following appearance:



- The code-behind contains two methods
 - DoNew()
 - DoSave()
- You should now wire up commands such that the UI satisfies the following constraints
 - The "New" menu item invokes the DoNew() method.
 - This operation is always enabled
 - The "Save" menu item invokes the DoSave() method.
 - This operation is enabled only when nonempty first name and nonempty last name are provided in the TextBoxes.
 - The "Save" button also invokes the DoSave() method
 - This operation is enabled only when nonempty first name and nonempty last name are provided in the TextBoxes.
 - Note: Make sure that both the corresponding buttons and menu items are enabled/disabled exactly when the underlying operations are enabled/disabled!
- Hint:
 - Use `ApplicationCommands.Save` and `ApplicationCommands.New` and create command bindings for them

Commanding Test

File

New

Save

Last name:

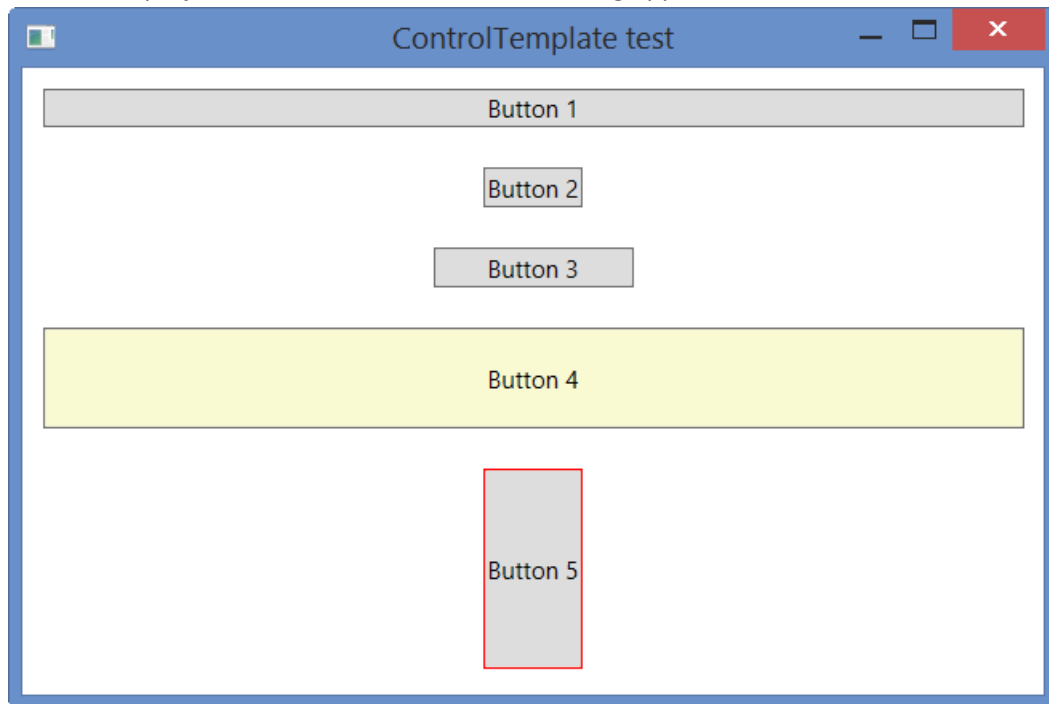
Save!

Module 5: “Control Templates and User-defined Controls”

Lab 5.1: “Ellipse control template for Button”

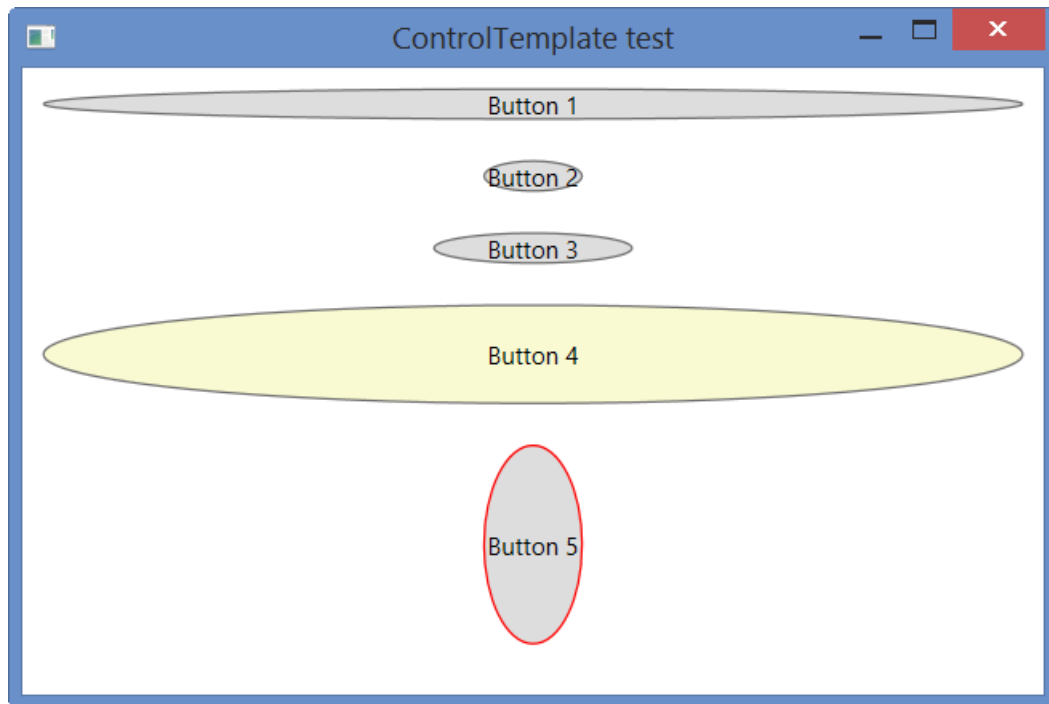
This exercise provides a new control template for Button.

- Open the starter project in
C:\Wincubate\89098\Module 5\Lab 5.1\Starter ,
which contains a project and a window with the following appearance:



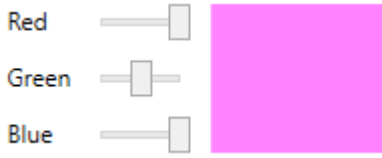
- Create an ellipse-shaped control template for Button, respecting both the general sizing such as Width and Height as well as the following properties
 - Background
 - BorderBrush
- Hint: Construct a control template consisting of a Grid with an Ellipse.

When completed, your user interface should look similar to:



Lab 5.2: "Creating a User Control" (☆☆)

This exercise investigates user controls and dependency properties. You will construct a new WPF user control and equip it with appropriate code and data.

- Create a new solution in
C:\Wincubate\89098\Module 5\Lab 5.2\Starter .
and add a WPF User Control Library project called Toolkit.
- Within the Toolkit project, you must create a UserControl called ColorSelector with the following appearance:

- The three sliders range from 0 to 255 and corresponds to the Red, Green, and Blue components, respectively, of a color which is to become the output selected by the ColorSelector user control.
- Equip ColorSelector with a dependency property called Color of type Color.
 - Use the propdp snippet in Visual Studio.
- The Rectangle on the right-hand side of the user control should be updated live whenever the underlying Color property value changes
 - Either by moving a slider,
 - Or via data-binding, animations or some other "external" source.
- Hint:
 - Register the Color property with appropriate property metadata containing a PropertyChangedCallback being called whenever Color changes.
- Add a WPF Application project called Test to your solution
 - Add a reference to Toolkit
 - Instantiate the ColorSelector user control
 - Test that it works as expected! 😊

If time permits...

- Equip the ColorSelector with a bubbling ColorChanged routed event

Module 6: “Threads and Asynchrony in WPF [Foundation]”

Lab 6.1: “Tasks” (☆☆)

This exercise deals with tasks, asynchronous methods and the await keyword. We will construct a WPF application starting three tasks, which each fetches the string contents of a newspaper front page, and subsequently computes the total length of all the incurring strings.

- Open the WPF application project located in
C:\Wincubate\89098\Module 6\Lab 6.1\Starter .
- First implement the following method
`Task<string> CreateFetchTask(string url)`
which creates an instance of WebClient and downloads the string from the specified URL using `WebClient.DownloadStringTaskAsync()`.
- Now implement
`async Task<int> ComputeLengthSum()`
by completing the TODO-part (and add async modifier!), such that the method constitutes a task which – when all sub-tasks have completed – outputs the sum of the lengths of all the strings downloaded by tasks t1, t2 and t3.
- Finally, implement
`void OnComputeClick(object sender, RoutedEventArgs e)`
Such that it calls `ComputeLengthSum()` and awaits that the computation of the result is done, and subsequently updates the user interface control.

Module 7: “MVVM Design Pattern [Foundation]”

Lab 7.1: “Editing Customer using MVVM” (☆☆)

This exercise supplies an introductory example showing how typical MVVM applications are structured and what techniques are required to implement basic functionality using the MVVM pattern. You will define a `Customer` class and provide user interface with simple validation to edit the customer as well as implement foundational command functionality.

We will develop every without using an MVVM framework to give you an understanding of exactly what amount of code is needed to provide the necessary functionality.

- Open the starter project in
C:\Wincubate\89098\Module 7\Lab 7.1\Starter .
- Investigate the supplied project structure and files
 - Make a note of `Customer.cs` and the automatic properties it has.

Model Implementation

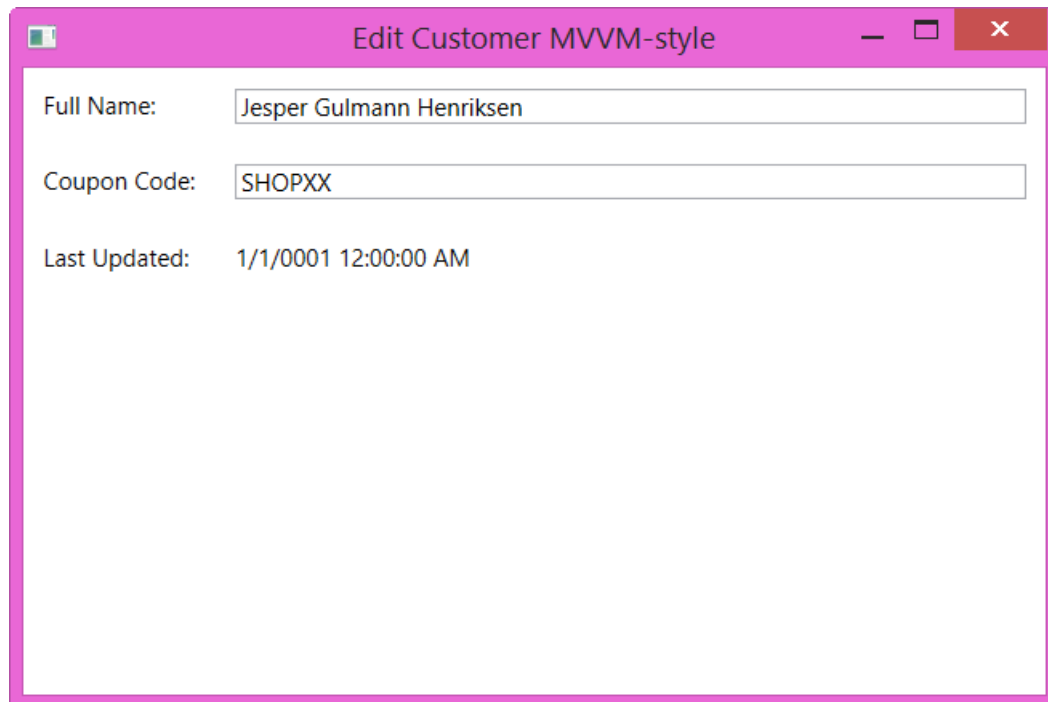
- Implement the `INotifyPropertyChanged` interface in the `Customer` class and modify all existing properties.

ViewModel

- Add the following to the `MainViewModel` class
 - A property of type `Customer` called `ModelCustomer`
 - A constructor initializing `ModelCustomer` to contain
 - Your full name
 - Some appropriate coupon code.
- Implement the `INotifyPropertyChanged` interface in the `MainViewModel` class.

View

- Add appropriate XAML (including data context and data bindings) to the view to make it somewhat similar to the following screenshot:



- Run the program and test that the textboxes reflect the default customer info

Model Validation

- Add validation to the Customer class by implementing the [IDataErrorInfo](#) interface accordingly
 - FullName should consist of at least three characters
 - CouponCode should consist of exactly six characters.
- Modify the bindings such that the [DataErrorValidationRule](#) is enabled
 - Hint: Set ValidatesOnDataErrors to true.
- Run the program and test that validation works as expected.

Commands

- Add a SaveCustomerCommand to the implementation by
 - Creating a [SaveCustomerCommand](#) class implementing [ICommand](#)
 - Create a constructor accepting an Action to be executed
 - Make sure that this Action is executed when the command is executed
 - Make the command always enabled.
 - Create an instance of [SaveCustomerCommand](#) in the [MainViewModel](#) class
 - When the command is executed it should just update the LastUpdated property.
- Add a Save button to MainWindow.xaml executing the [SaveCustomerCommand](#).
- Run the program and test your implementation.

If time permits...

- How easy / difficult do you think it is to only enable the Save button exactly when the coupon code validation succeeds?

Module 8: “WPF Testing and Debugging”

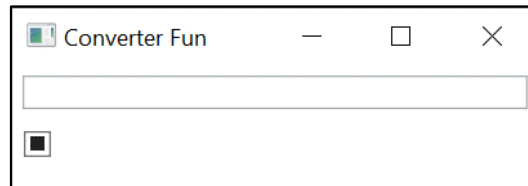
No labs

Module 9: “Data Binding [Deeper Dive]”

Lab 9.1: “Conversion of values during data binding” (☆☆)

This exercise illustrates the use of converters for data binding single properties of controls.

- Open the starter project in
C:\Wincubate\89098\Module 9\Lab 9.1\Starter ,
which contains a project and a window with an appearance like the following:



- Create a two-way value converter `YesNoToBooleanConverter`, which converts strings to `Nullable Boolean` values such that
 - “Yes” corresponds to `true`
 - “No” corresponds to `false`
 - Any other value corresponds to `null`
- Apply the converter appropriately such that changes to the text in the textbox are reflected in the checkbox (and vice versa)
- Test that your solution works by entering various texts in the textbox and observing the resulting updates.

If time permits:

- Make the converter multi-lingual, such that it works for all three (neutral) cultures
 - en
 - “Yes”
 - “No”
 - de
 - “Ja”
 - “Nein”
 - da
 - “Ja”
 - “Nej”
- How do you specify which culture the converter should use in the program?
- Test your solution for each of the three specified cultures.

Lab 9.2: "Sorting data-bound elements" (★)

This exercise illustrates sorting collections of data objects.

- Open the starter project in
C:\Wincubate\89098\Module 9\Lab 9.2\Starter ,
which contains the solution to Lab 3.1.
- Create appropriate XAML-fragments such that the participants are displayed as before, but now sorted by their last name.
 - Note: Make sure you do this without writing any code-behind..!

Module 10: “MVVM Design Patterns [Deeper Dive]”

Lab 10.1: “SimpleCalculator in MVVM” (☆☆☆)

The purpose of this exercise is to re-implement the SimpleCalculator of Lab 4.1 – but this time using MVVM!

- Open the existing WPF project called “SimpleCalculator” in
C:\Wincubate\89098\Module 10\Lab 10.1\Starter
- Verify that the solution contains the UI and code-behind of Lab 4.1:



The display consists of a [TextBox](#) which is to be updated whenever a digit-button is clicked. The 'C'-button should clear the display.

The 'Result'-button should show the result in a [MessageBox](#) and then clear the display.

- Implement the same functionality as in the solution to Lab 4.1, but using the Model-View-ViewModel pattern!
 - You either implement/copy all necessary [ViewModelBase](#) parts and helpers from anywhere you like – or perhaps use an MVVM framework of your choice. 😊

Implement digits functionality

- First implement the functionality of pressing the digits and updating the display
 - Create the necessary ViewModel, Commands etc. and make sure to wire up all bits and pieces correctly.
 - If you feel “pure at heart”, consider introducing an appropriate model object.
- Implement parameterized commands for the digits

Implement the functionality for C

- Implement the command for C

Implement the Result message box

There are a number of different ways to implement the functionality of Result, which displays a message box containing the current display contents. Pick whichever one you feel is right for you!

- Implement the functionality of the Result button

Note: This is not always easy! The complexity of your solution depends upon whether you're implementing your application from scratch or using some MVVM framework.

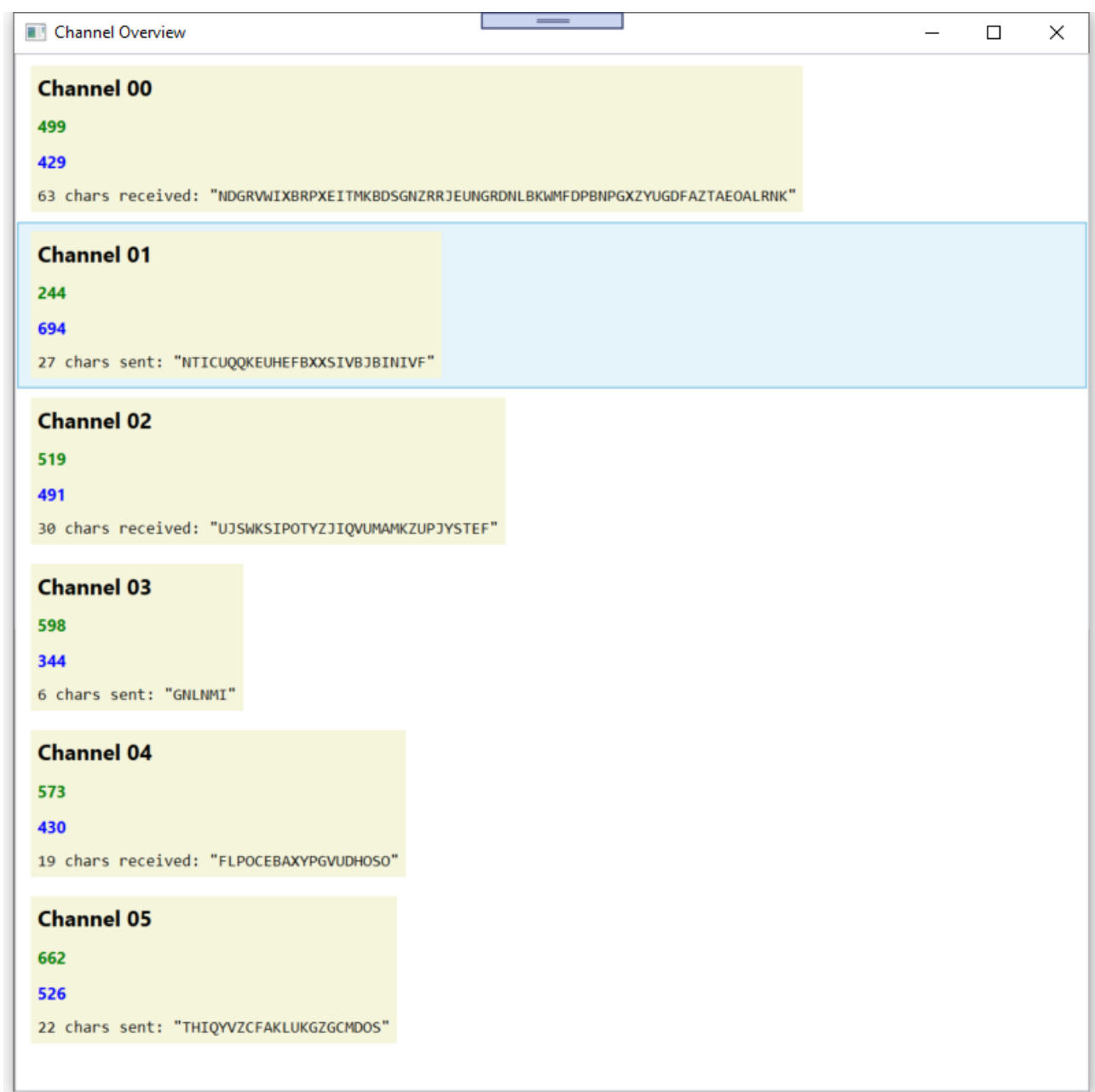
Module 11: “Threads and Asynchrony in WPF [Deeper Dive]”

Lab 11.1: “Making Threaded Updates Thread-safe in WPF” (☆☆)

This exercise investigates how to make a WPF UI both thread-safe and “update-safe” providing detailed insight into how both updates to collections as well as to single elements are handled.

- Open the existing WPF application located in
C:\Wincubate\89098\Module 11\Lab 11.1\Starter ,
and get an overview of the existing classes, views, and view models.

The solution contains an application skeleton, where you need to process the updates of `ChannelMessage` instances from a dedicated worker thread to the UI such that your UI will resembles the following:



The application skeleton provided for you above for each incoming channel display a status consisting of:

- **Channel Description**
- **Number of chars received**
- **Number of chars sent**
- Latest status

The models, views, and view models are already provided to you and are wired up appropriately. The only missing piece is the `MainViewModel.OnNewChannelMessage()` which needs to be completed:

```
// This will be called on a non-Dispatcher thread
internal void OnNewChannelMessage(ChannelMessage channelMessage)
{
    // TODO: Handle updates from worker thread correctly...
}
```

- You should complete the method in a manner such that
 - When the first `ChannelMessage` for a specified Index is encountered, a new, the view is updated with a new `ChannelViewModel` appropriately initialized with the data in the `channelMessage`.
 - When a `ChannelMessage` for a `ChannelViewModel` already existing in the UI is encountered, the existing `ChannelViewModel` is updated using `ChannelViewModel.UpdateWith()`
- Note: This needs to be done in a manner which is
 - **update-safe**, i.e the UI updates properly without exceptions when
 - new elements are added to the UI (the underlying collection)
 - properties of existing elements in the UI are updated (a single element)
 - **thread-safe**, i.e. if other sources (perhaps button presses or timers) also update the UI, they updates in `MainViewModel.OnNewChannelMessage()` are not “disturbed” in any way.