# MVVM

## Enterprise & Mobile .Net

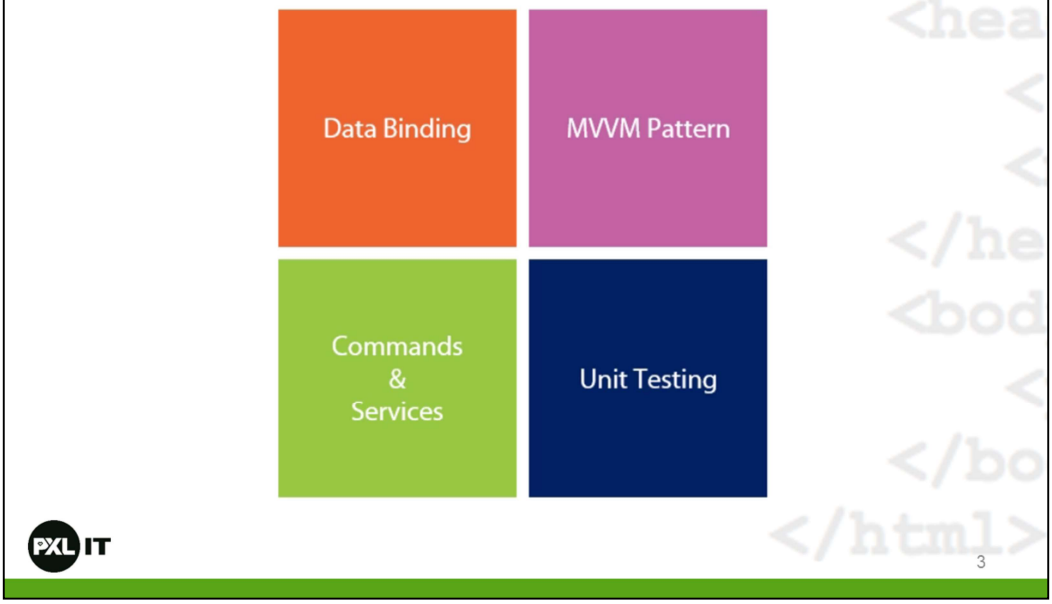
DE HOGESCHOOL
MET HET NETWERK

Hogeschool PXL – Elfde-Liniestraat 24 – B-3500 Hasselt
www.pxl.be - www.pxl.be/facebook

# MVVM

- Model – View – ViewModel pattern
- Used for XAML based applications
- Makes applications more testable and easier to maintain

# We'll learn about




PXL IT

3

# Before you can learn about MVVM

You have knowledge of **XAML**

You are familiar with **WPF** on a basic level

You want to learn how to create **maintainable** and **testable** apps

PXL IT

4

# Learning and refactoring to MVVM

Outline

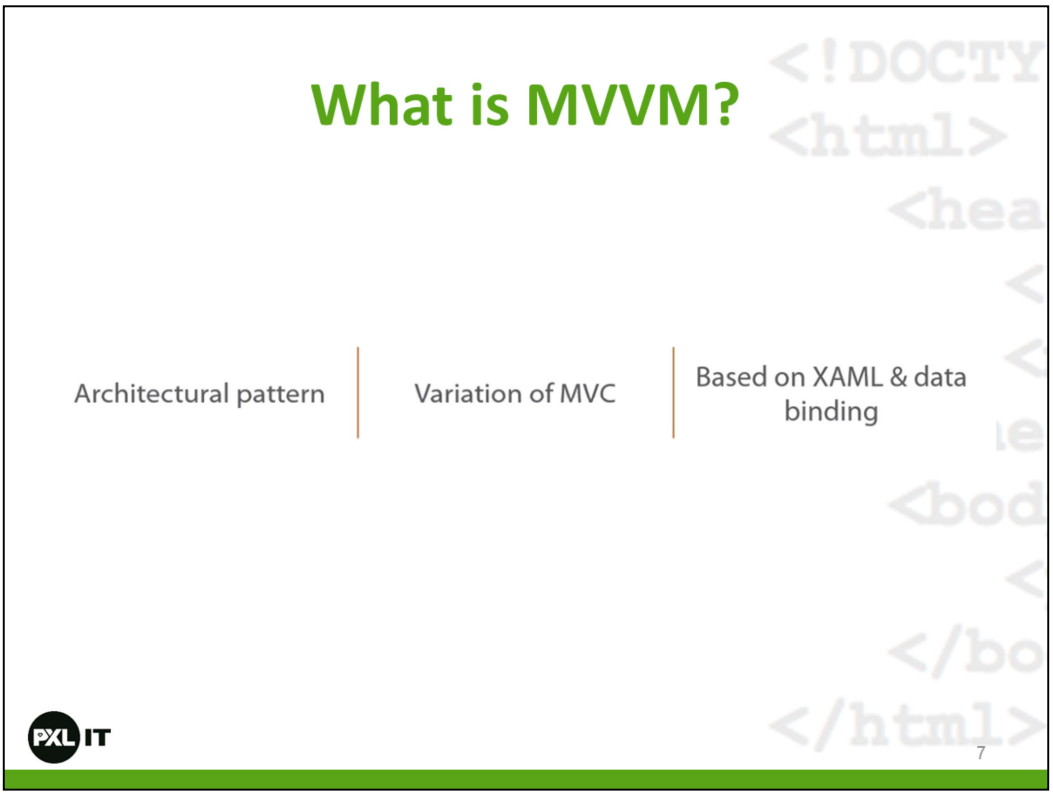
- The MVVM Pattern
- Building Blocks of MVVM
- Linking the View and the View Model

PXL IT

5

# Target of this module

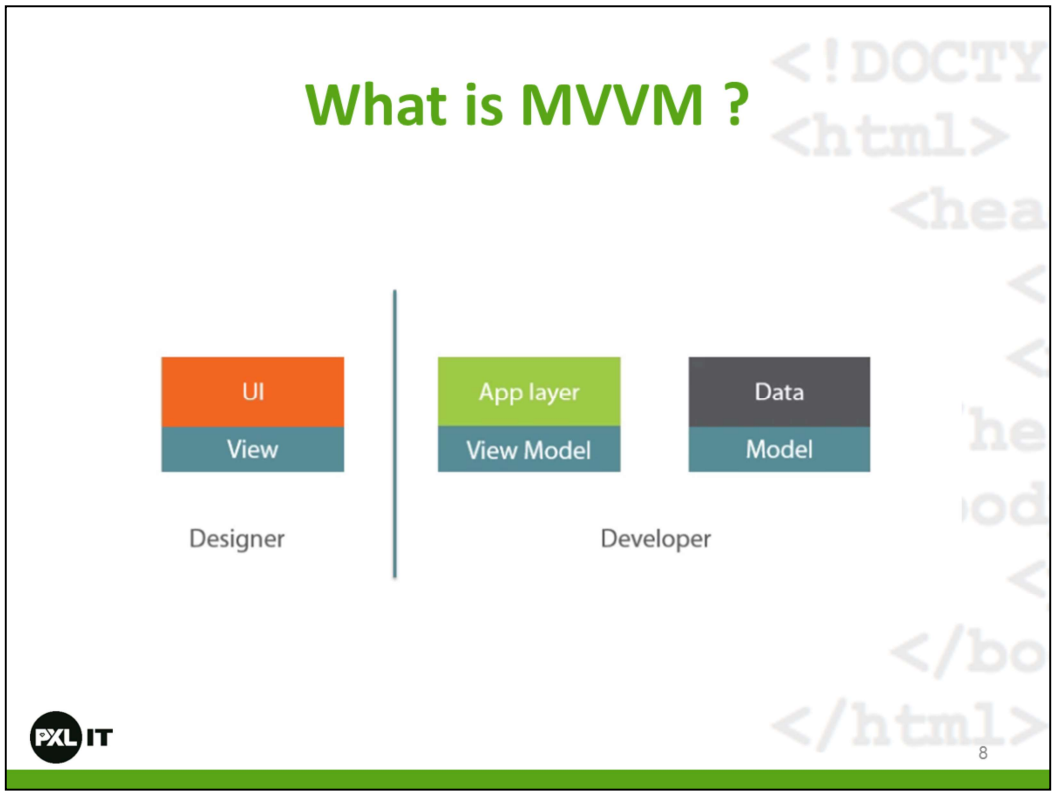
- Understand the MVVM pattern and its parts
- Understand why it is useful




MVVM aka the Model-View-ViewModel pattern is in itself nothing more than a software or architectural pattern. It's in fact a variation of the MVC pattern and thus can be seen as a variation of the presentation model, which has been around for many years already. Just like the presentation model, as well as MVC, MVVM has as its most important goal, a better separation of concerns. MVVM is relying heavily on what XAML has to offer. Without databinding, MVVM wouldn't be able to work as it does. MVVM is therefore a pattern which is very often used in XAML based applications and works simularly in WPF, Silverlight, UWP and Xamarin.

Remember that we said that databinding is the enabler for MVVM. We'll see soon how it is the case here.

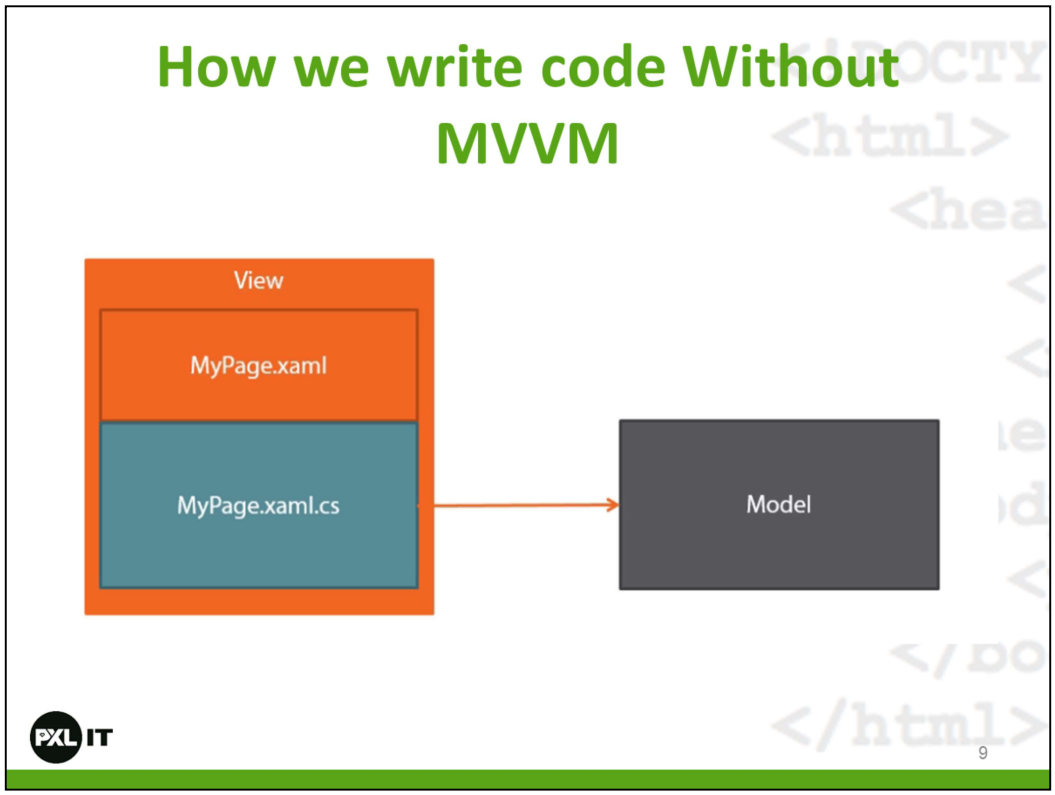
Reference: http://martinfowler.com/eaaDev/PresentationModel.html




As mentioned, one of, if not the most important goal of MVVM is creating a clean separation between the different aspects of our code. Typically we can identify a number of layers in any XAML application.

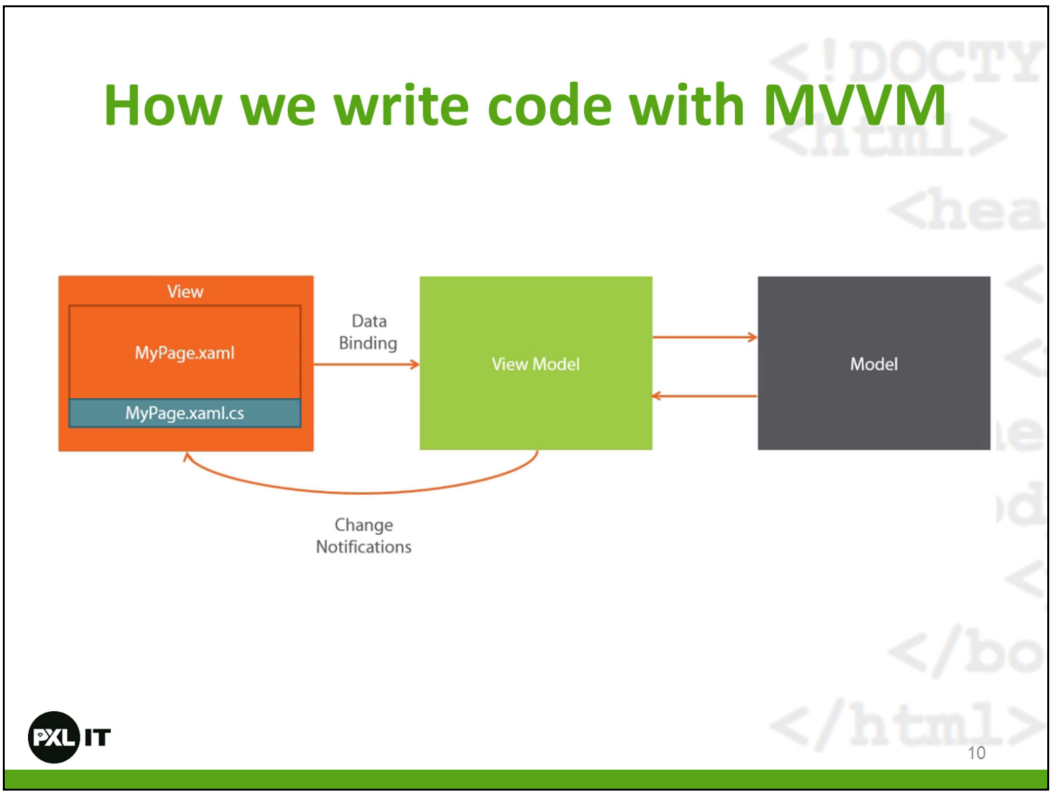
On the left we have the **view** code, which determines how the UI will look. The view is typically the area where the designer does his work. On the right, we have the functionallity of our application, which will live in the **viewmodel**. Here we are writing C# code and it's thus part of what the developer is working on. The Viewmodel will interact with the **model** or the data. Again this is part of what the developer will be working on as well.

In an ideal world, we would have the ability to give all the UI work to the designer and all the coding work to the developer. We don't live in an ideal world and therefore this doesn't always work. However, using MVVM will give us a better separation of the different layers and even if you are the only person working on the application, you'll still find a lot of benefit when using MVVM.




If we aren't using the MVVM pattern, how are we structuring our code? On the left, we have our view, typically this view code consists out of some XAML code, which can live in e.g. MyPage.xaml. This is purely the UI code which contains no functionality. The functionality can be found in the code-behind e.g. in the MyPage.xaml.cs. In this code-behind we have often quite a lot of code, including event handlers, which will react to interactions of the user with the view. Very often too, this code interacts directly with the model. Perhaps you want to get some data from a service, so directly from the code-behind, we will be interacting with this service. And when data is retrieved you want to update the UI again. The problem that raises with this code-behind approach, is that **everything is very tightly coupled**. On one line, we are connecting with the service, and on the next one, we are updating the UI. It's very hard, if not, impossible to test any code in isolation by writing a unit test.

The main culprit is that we are directly using our UI controls from code. There has to be a better way.



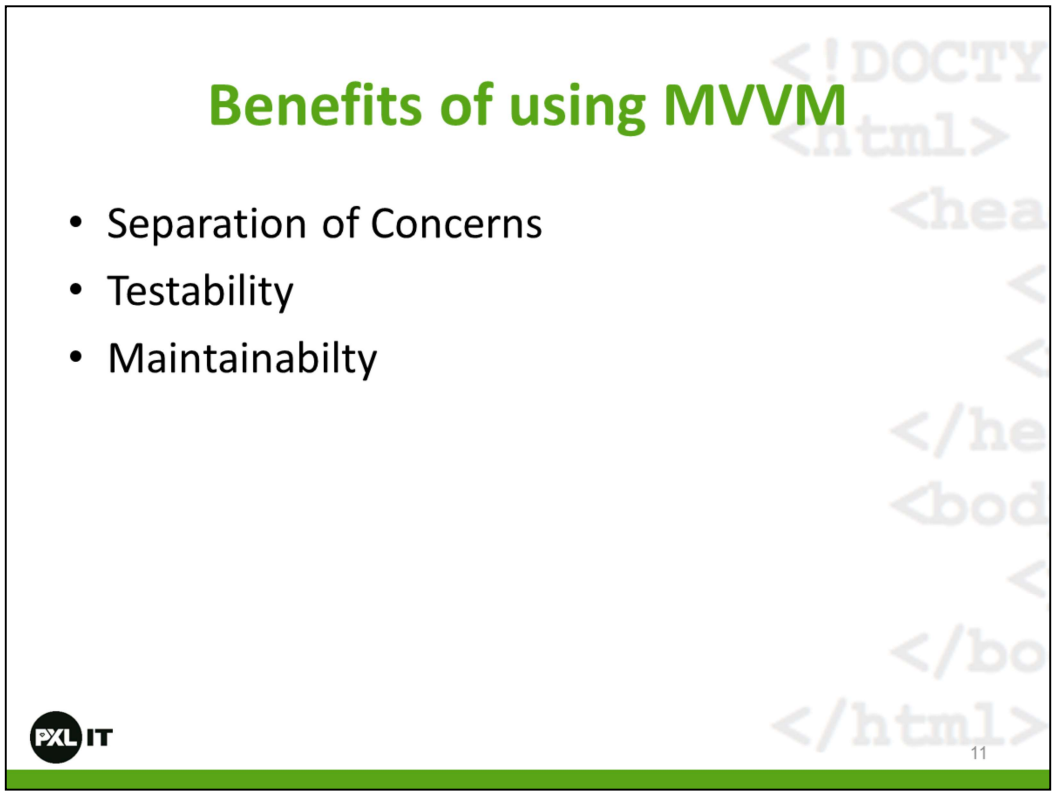

If we move into using MVVM, things will be quite different. Of course we will still have a View block and inside the view we still have XAML code which makes up the visual part of the code. Some things may be different here, but the XAML will still be roughly the same. We'll still have code-behind as well.

Notice that the code behind is now a lot smaller compared to what we had in the previous slide. This is intentional. There will be very little code in the code-behind. In fact, there shouldn't be any code within there, which directly interacts with the model. So any code, that we may want to test using e.g. unit tests, shouldn't be in the code-behind. However code which does view-specific tasks such as setting the focus, will still live in the code-behind.

The bulk of the code will now be living in the viewmodel. The Viewmodel can be seen as an abstraction of the View. The View will have a ViewModel associated with it. Typically, this is a one-to-one relationship. Through databinding, the view will be binding on properties, exposed on the viewmodel. And because of notifications being sent by the viewmodel, changes in the value of Viewmodel properties, will result in the view being updated. This is realised by the INotifyPropertyChanged Interface, which we have already discussed in the previous module.

Notice that there's no longer a tight link between the view and the model. It will

be the viewmodel which does the communication with the model. The Viewmodel will for example ask the model to load data from a webservice. And when this data has been retrieved, it can notify the viewmodel about this. The viewmodel in turn can do things with the data and then raise a notification so that the view can pick up on this. All communication happens through databinding. This means that we have **loose coupling between the building blocks**, and that is of course a good thing. Do remember that the viewmodel is exposing properties. Properties can be seen as the state of the ViewModel. The Viewmodel will also expose operations in the form of commands (which will be discussed further).

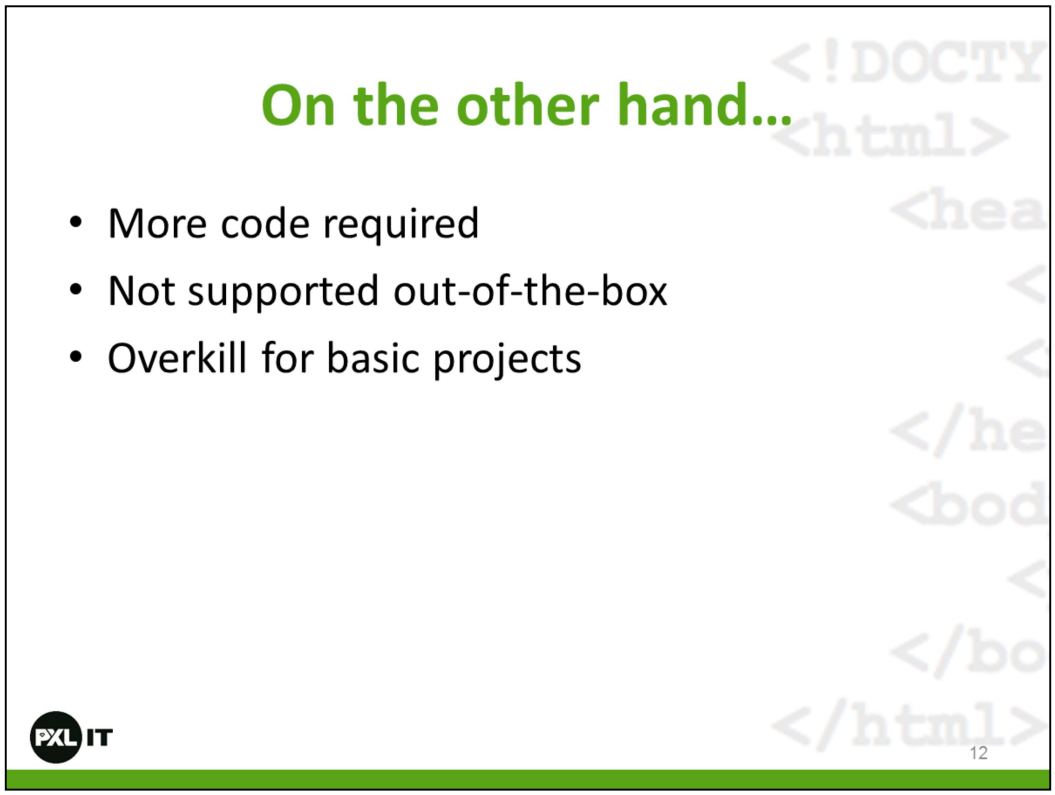



So what are the main benefits of using MVVM?

It should be clear by now that writing code using the MVVM pattern results in better **separation of concerns**. The view will be just about the view: the XAML code and perhaps some UI logic, such as setting the focus. The Viewmodel can be seen as the controller of the whole setup. It contains the logic of the application. There is no hard link anymore between the view and the viewmodel.

What this leaves us with,  is that our code will be much more testable. It will be possi.zble to write unit tests on the code in the Viewmodel. There are typically no UI control references in your Viewmodel which is what you need in order to be able to write those unit tests easily.

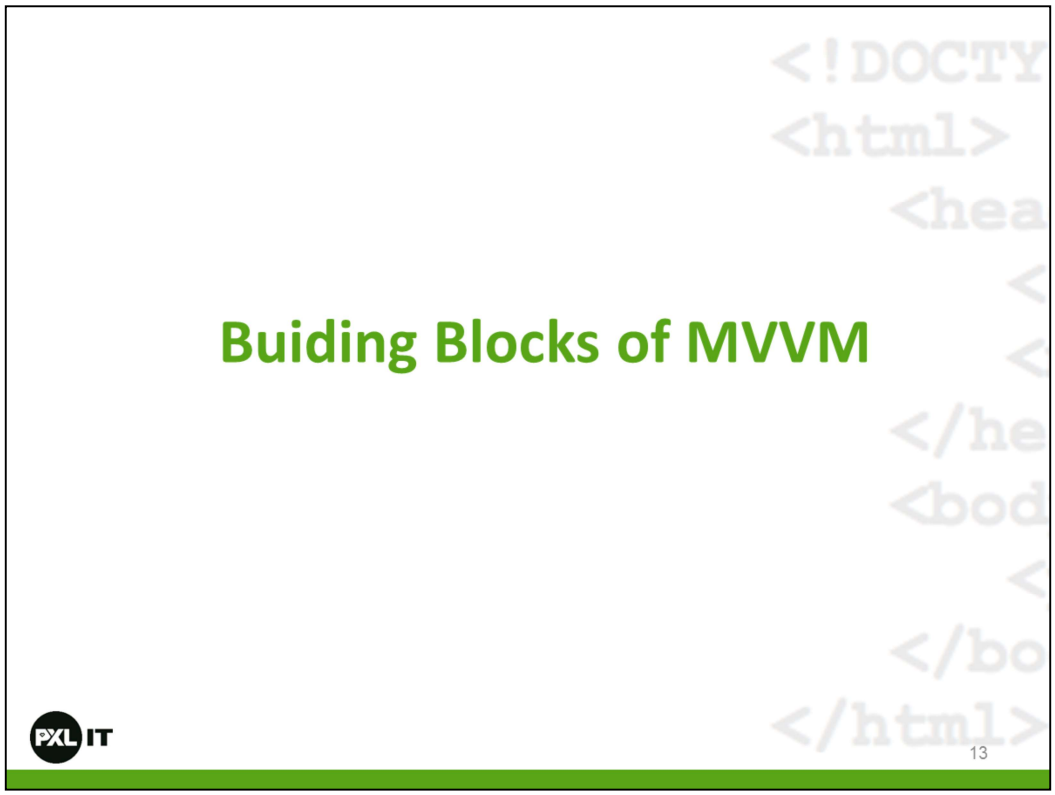
The code that we write will also be easier to maintain. Since there is no tight coupling between the layers anymore, we can make changes for example to the UI without requiring any changes to be made on the viewmodel. Also when the Viewmodel needs changing, it is not required to change the view.



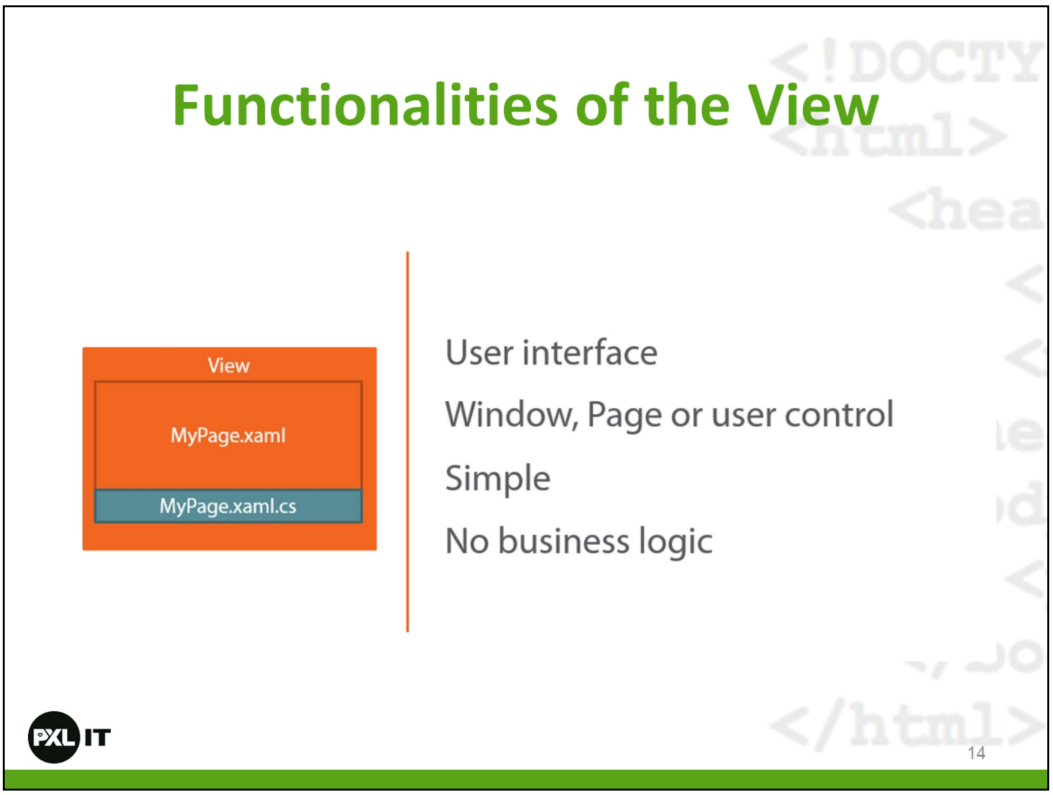

Of course there are some disadvantages to writing your code based on the MVVM pattern as well.

In general, you will be writing more code when using the MVVM pattern. One of the main reasons for this of course, MVVM isn't supported out-of-the-box. By default, when you install VS, there are no templates available that use MVVM. There are several frameworks however, which can get you on your way when using MVVM.

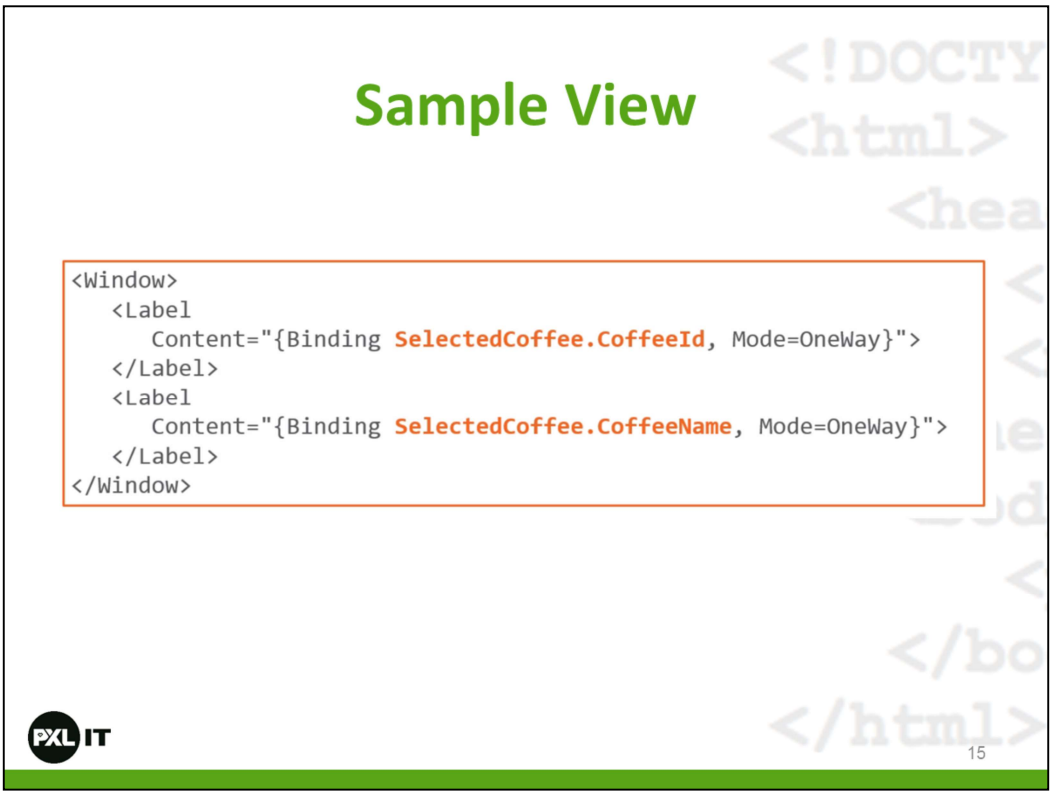
The two facst above can direct you into thinking that using MVVM isn't feasible for all types of projects. Mainly for small projects, using MVVM, can be a bit of a overhead. However, it may be usefull even for smaller projects. You never know how large an initially small project might become in the future. And a refactoring will be much more work at that time.



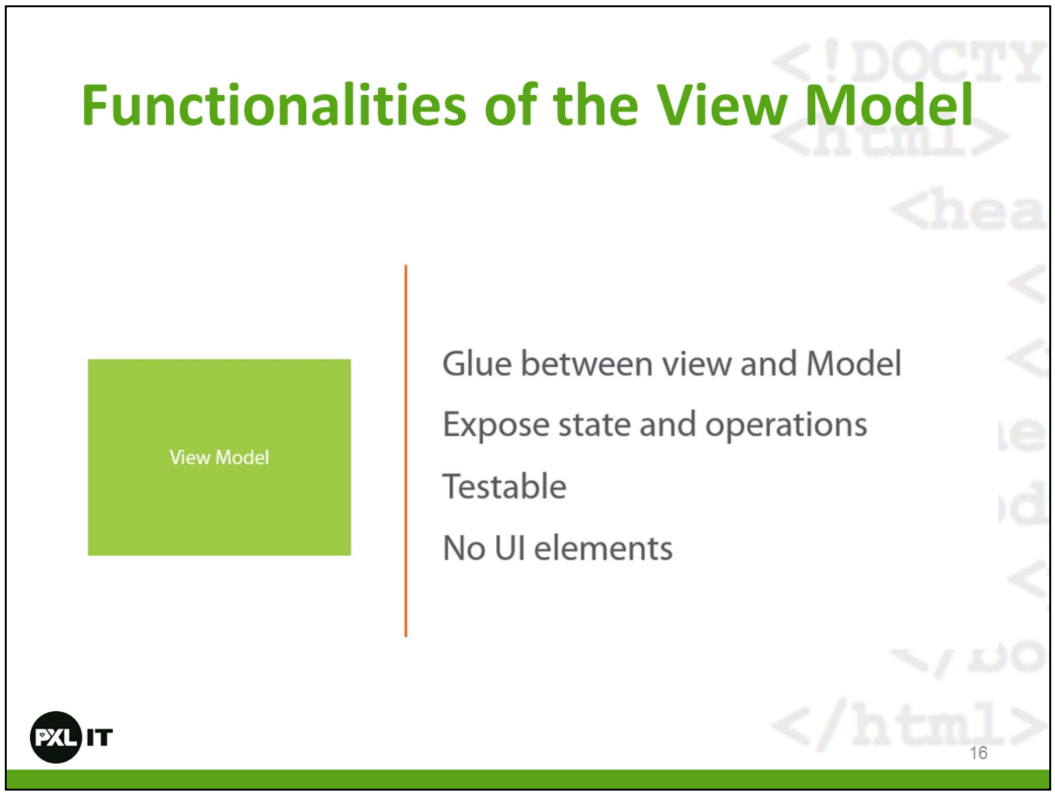

Now that we know what the MVVM pattern is, and why we should or shouldn't be using it, lets have a look at the different building blocks in a bit more detail, together with some sample code for each.




As mentioned, the view is the user interface and it contains mainly XAML code. Typically this will be a window, or a page or user controls. This is depending a bit on the XAML technology that you are using. The view is pretty simple. Definitely when we look at the code behind. The xaml.cs file is pretty empty. It shouldn't contain any business logic. The only code we are having here is the code which does things with the UI: setting focus, ordering columns, changing colors, but typical things that we can execute from the code behind file. Although in some cases, they might involve working with the data, there shouldn't be any business logic. So code which goes to the model.



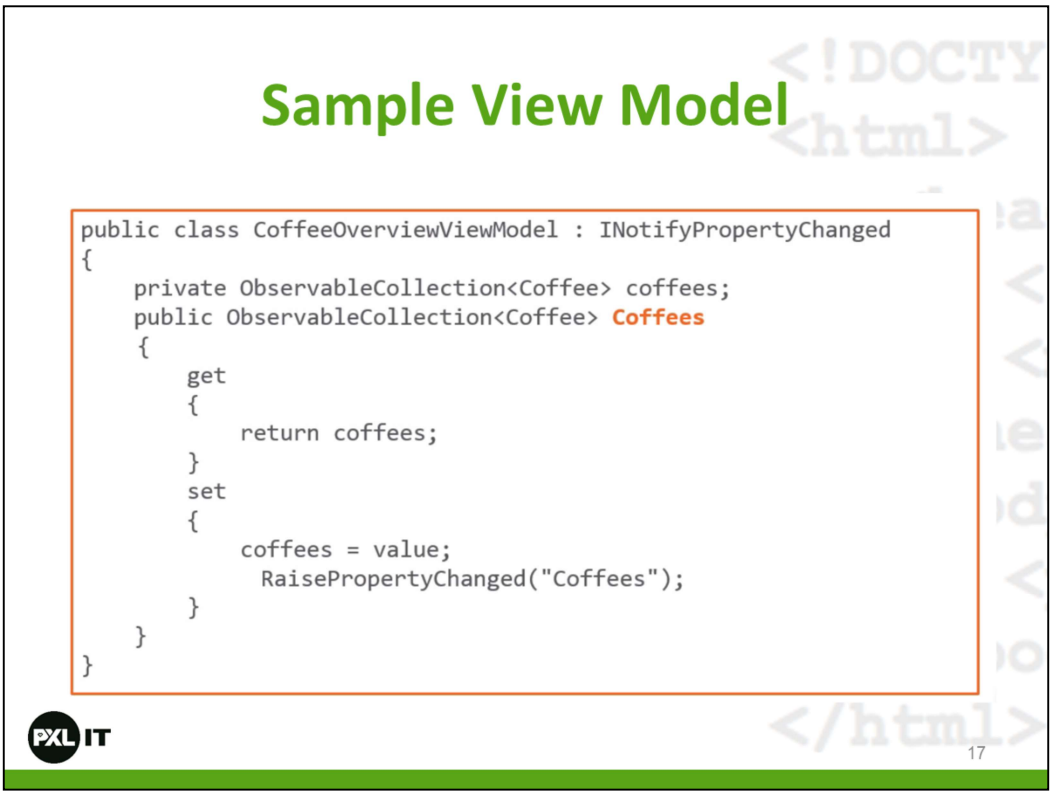

On this slide we see a sample view and in the sample view, we have a window which contains a couple of labels. For each of the labels, the content property is being databound to a property of the SelectedCoffee property. A SelectedCoffee is a **property on our ViewModel**. So what we are doing here is databinding to properties on objects we expose on the ViewModel. So we are already leveraging the databinding features of XAML.



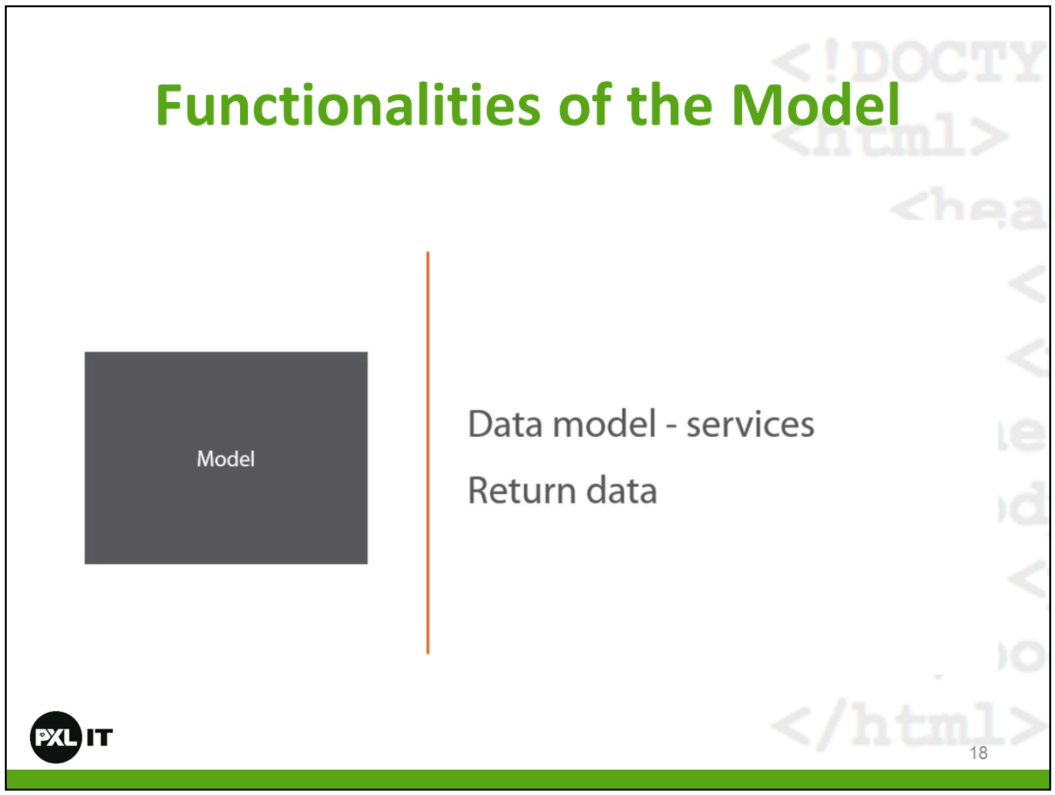

The ViewModel is our next Building Block and it's probably in the entire MVVM context, the most important one.

As you've already learned, the viewmodel sits between the View and the Model. It's the glue between these two layers. The ViewModel exposes state, another form of properties, for the view to bind on. Also the view exposes operations. In a non-MVVM scenario, you would have event handlers to handle the interactions of the user. In an MVVM scenario, these will channel through the ViewModel and will be handled using commands we can create on the ViewModel.

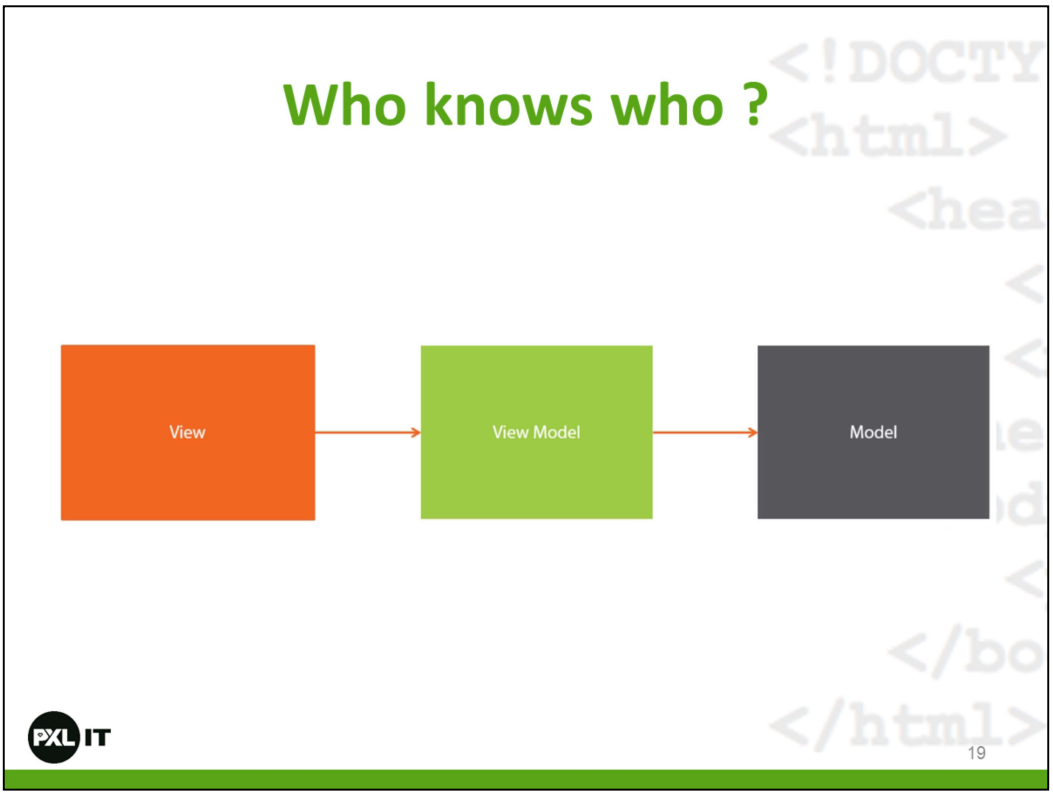
We'll talk about commands further on. Since we are creating the ViewModel as an abstraction for the view, the resulting ViewModel code is testable. It's possible to write unit tests since we can isolate functionality and we'll have the ability to mock dependencies. I've mentioned this already, but this last one is the crucial one. In order to keep things testable, there shouldn't be any UI elements or references in the ViewModel code. Remember that one of the main goals of using MVVM is writing testable code and having UI references in code which needs to be unit tested is difficult. So keep an eye out for this one.




On this slide here, we see a sample ViewModel. A ViewModel is nothing more than a class which implements the INotifyPropertyChanged interface. A ViewModel exposes state and operations. In this example, we only see some state being exposed and that is the Coffees property.



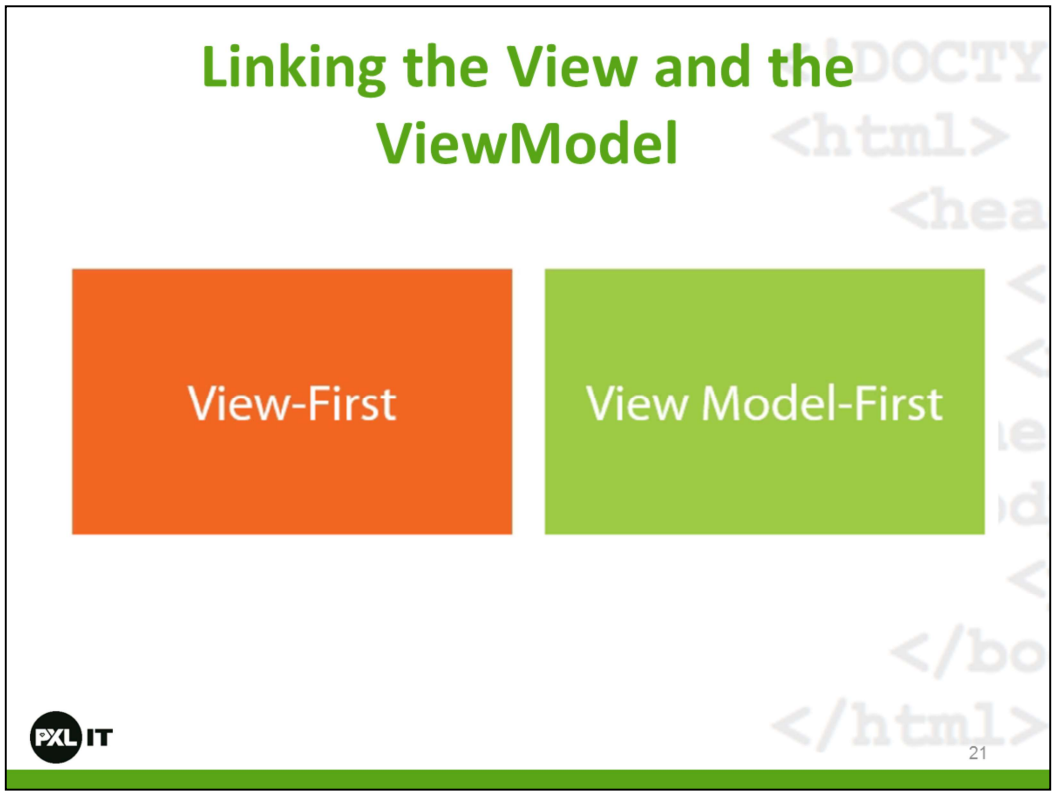

The final Building Block is the model. The model is often exposed to the viewmodel as a data service which interacts with a repository. The repository in term perhaps connects with a database or downloads data from a webservice. For this, our model is using the object model and in this, returning model instances to the ViewModel. In fact, the model in an MVVM scenario isn't very different from the model in a non-MVVM scenario.




One final thing before we go to the demo, a little bit of Who knows Who in MVVM.  I've already mentioned quite a few times, MVVM will help us with separation of concerns and that the resulting code is much more loosely coupled. However, the different building blocks in MVVM at some point need to know each other. The view as we have seen needs to bind on the ViewModel, so the View needs a reference to the ViewModel, that's for sure. Once it has this reference, it will use data binding to bind on that ViewModel. Using databinding doesn't create tight coupling. The ViewModel in term doesn't need to know the View. It exposes properties for the View to bind on, but it should therefore not have a reference to the view. However the viewModel needs to have a reference to the model. This way it can ask the model to load in data and return this data when it is ready loading. In the opposite way, the model doesn't need to know the ViewModel. To keep things loosely coupled, we can for example raise an event when the data is ready to be consumed from the ViewModel. This way we avoid creating a reference from the ViewModel in the Model.

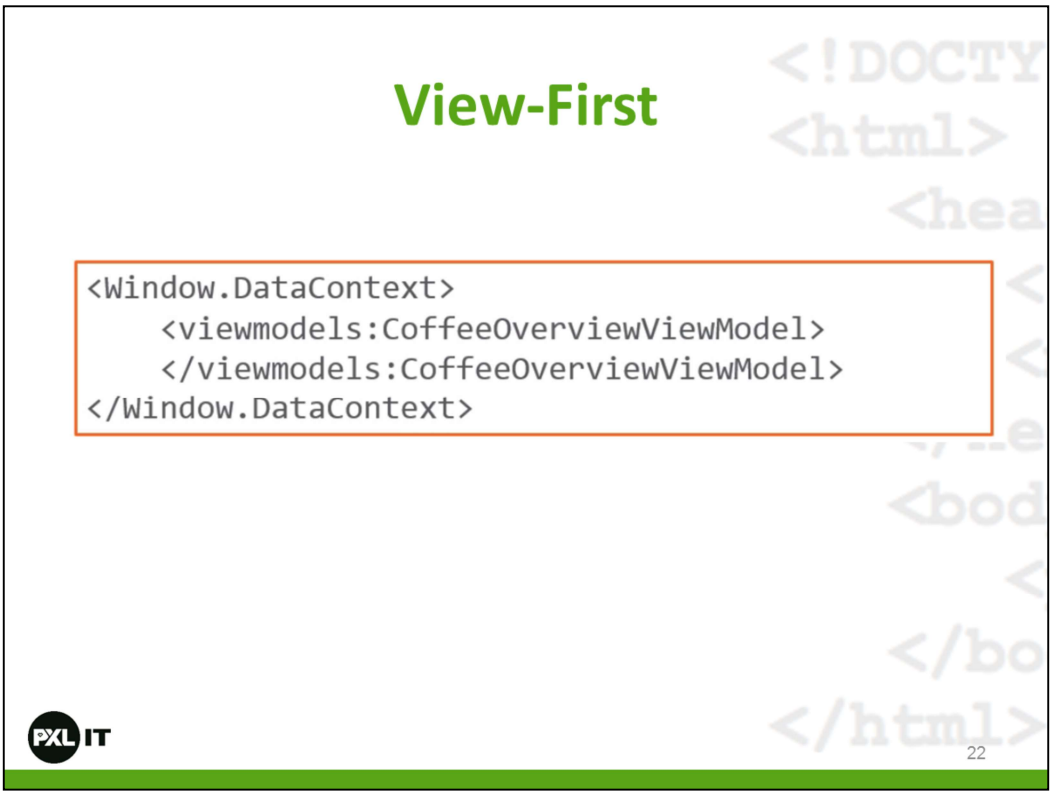
# Demo: Refactoring to MVVM


PXL IT

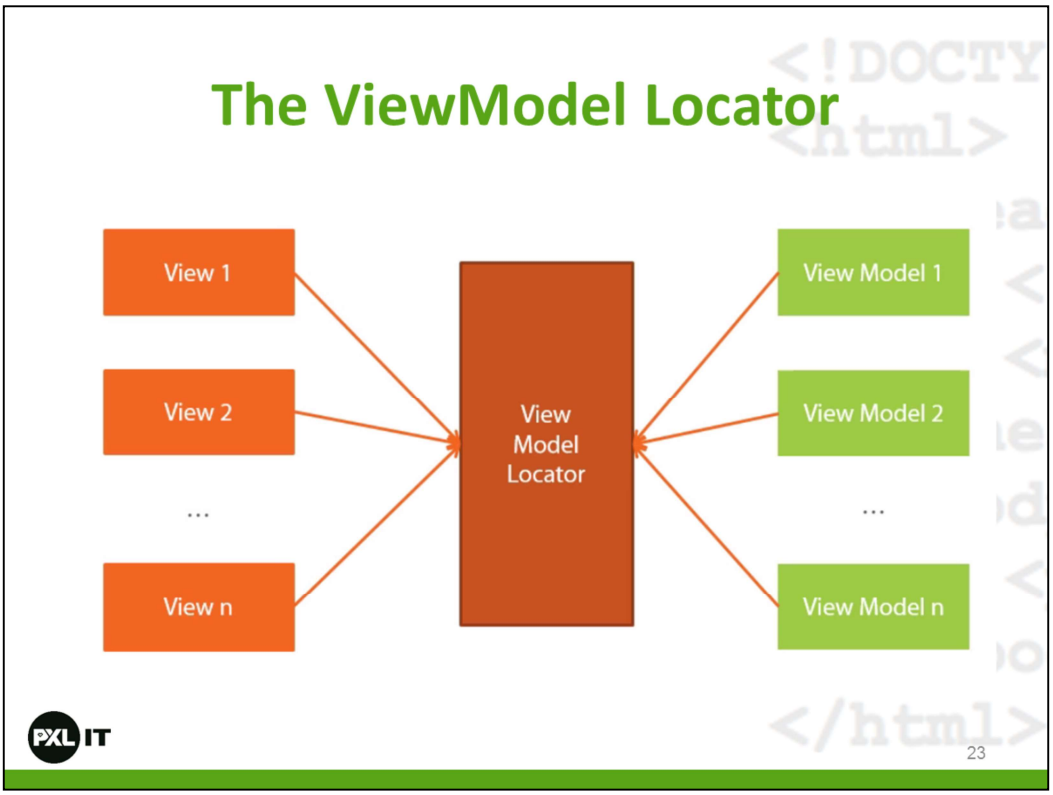



We have already seen how we can write our View and our ViewModels and how our ViewModels can get data from the model. But we don't have a fully working application yet, while quite a few things ar still missing. We're gonna need to fix some of these issues. To finish this module, we'll start with a missing link. How can we make sure that the view knows or finds the ViewModel?

We are now aware that we need to make sure that the view needs to know what ViewModel it will need to bind upon. In fact there are two common ways of creating this link. The possible options are: the View-First approach and the ViewModel-First

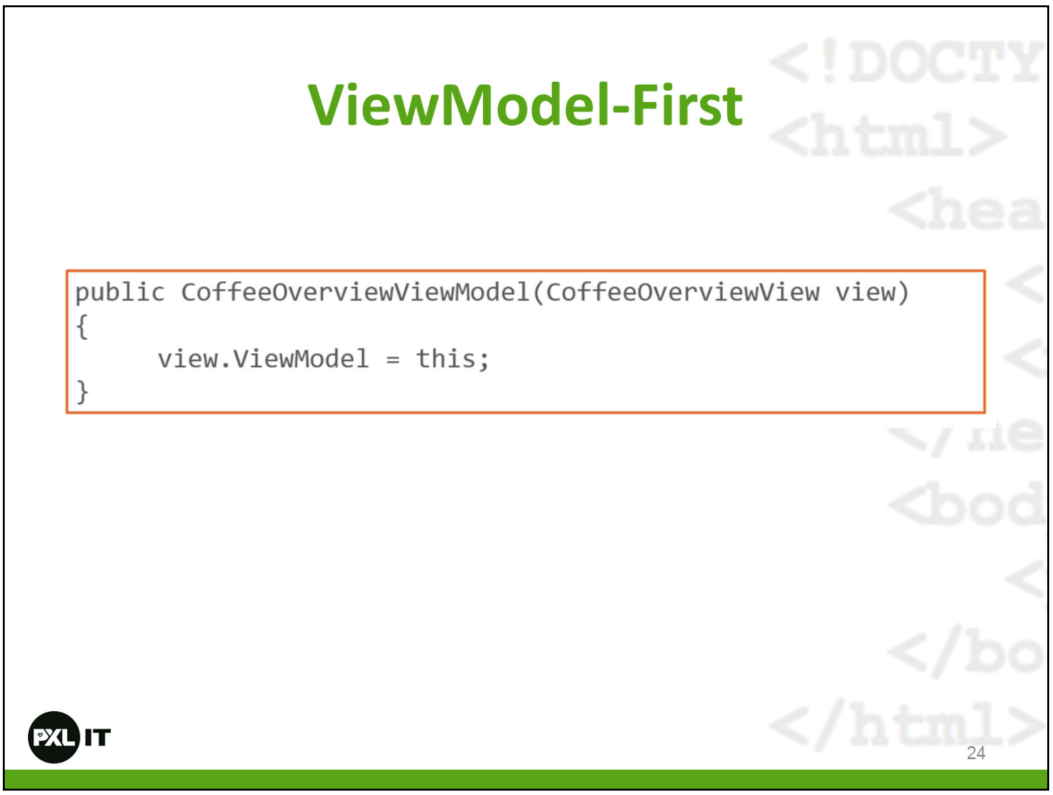



Let's first take a look at the View-First approach. In this case, we are instantiating the view first. The view will itself be responsible for the creation of the ViewModel. This ViewModel instance is then set as the datacontext of the View. In the sample code we see here, we are doing exactly that. The View is going to instantiate a CoffeeOverviewViewModel and sets this as the datacontext  of the view, so the window.




For most scenario's this approach will work. In fact, this approach works if for the bulk of your application you have an easy mapping of one view to one viewmodel. If in your application however, you have dynamically created views, such as e.g. the "New Mail" window in Outlook, this approach won't work. In this case, you should be looking at a ViewModel locator which is in fact the locator pattern.

So what is this thing? The locator is a class which knows about all the viewmodels in the application. All views know about the locator and can ask it to return a ViewModel instance or even create one on the fly.

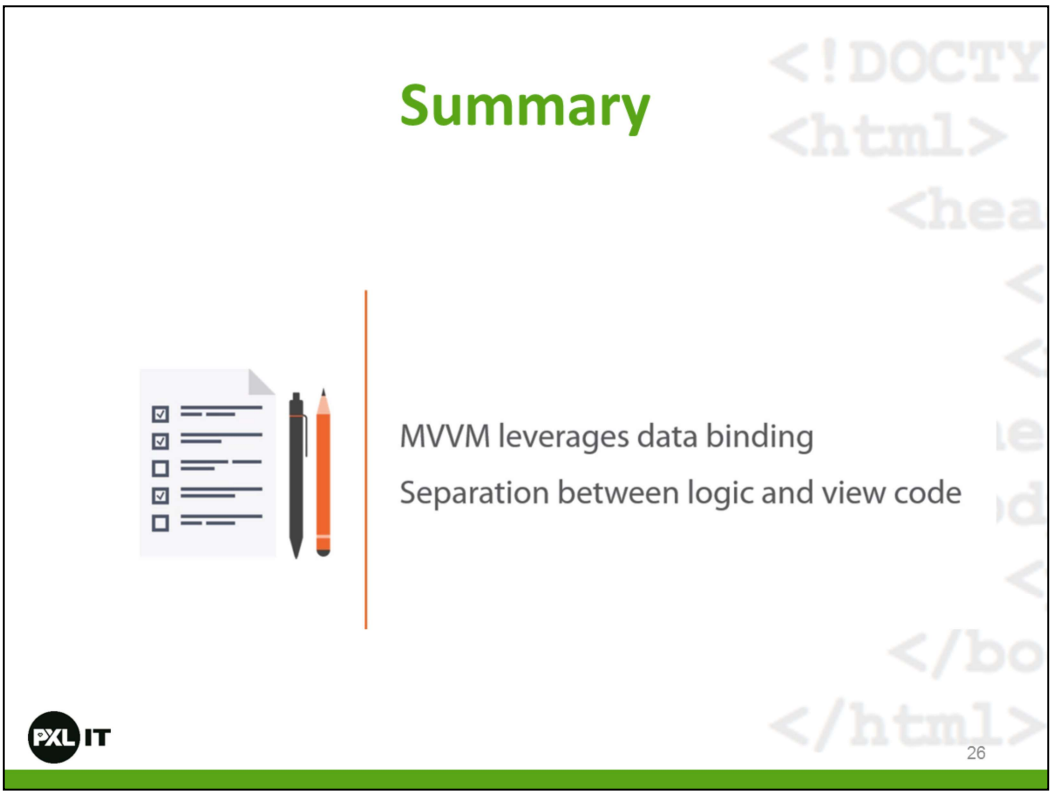



An alternative way of linking the View and the ViewModel is using the ViewModel-First approach. In this case, the ViewModel is created first and gets in an instance of the View it works with. We can then set the, lets call it View Property from the View to the ViewModel instance at hand. This approach is not that commonly used, and we will not discuss it any further.

# Demo: Adding a View Model Locator


PXL IT

25




So far, we have a partly working application again. But there's still a lot of work to be done. Making sure that everything works will be the focus of the next module. Before we go there, a quick summary of what we have seen in this module. We have focused on the MVVM pattern. And we have seen that it works together with data binding to display data on the views which is exposed by the ViewModels. Because of this, we achieve a loosely coupled architecture and we succeed in having a clean separation between the logic in the ViewModel and the View code in the View.

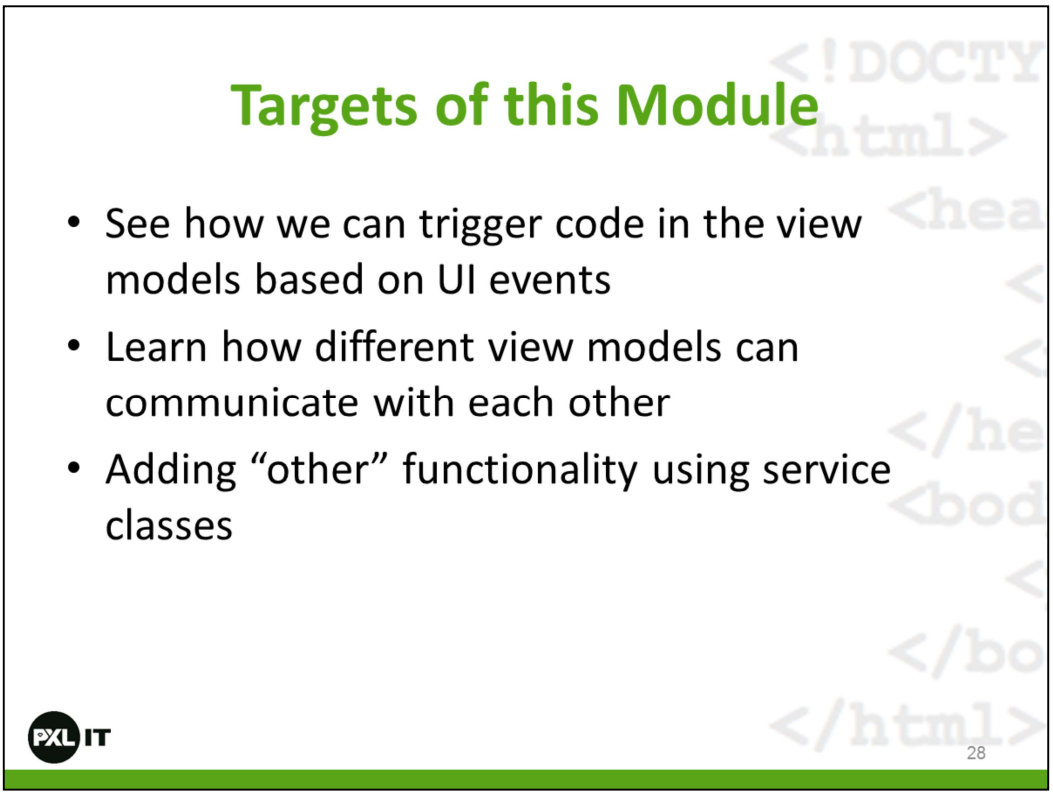
# Responding to Commands and Using Services

Outline

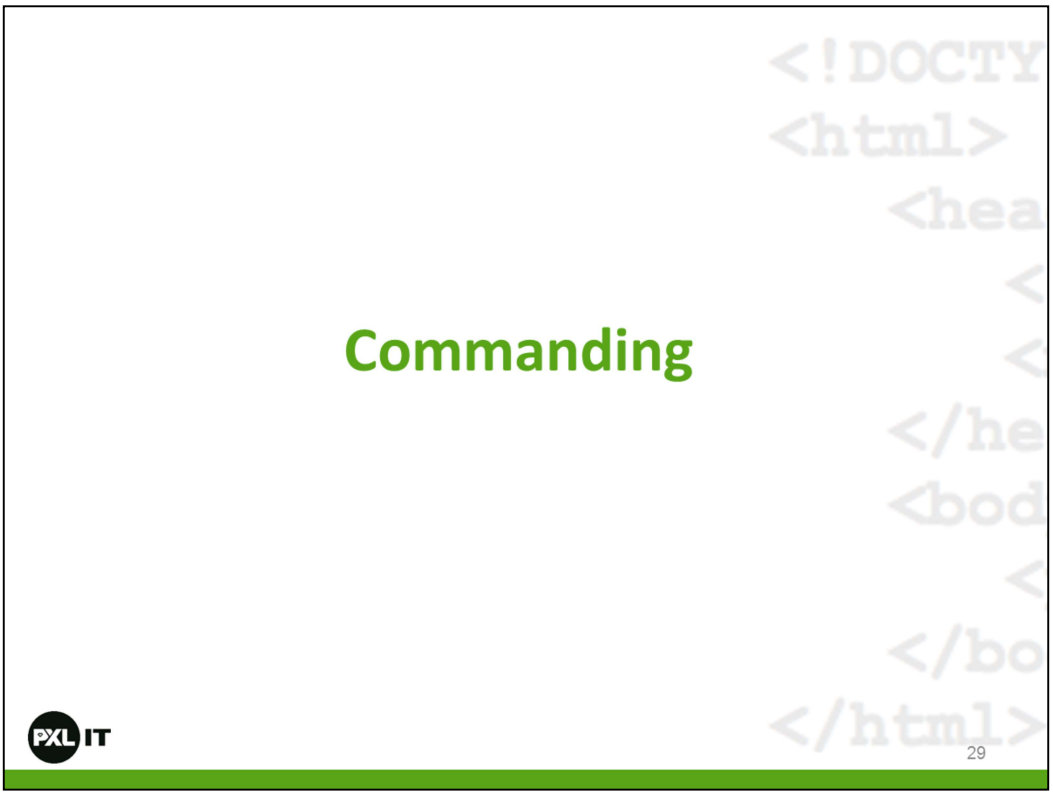
- Commanding
- Using Behaviours
- View Model Communication
- Services

PXL IT

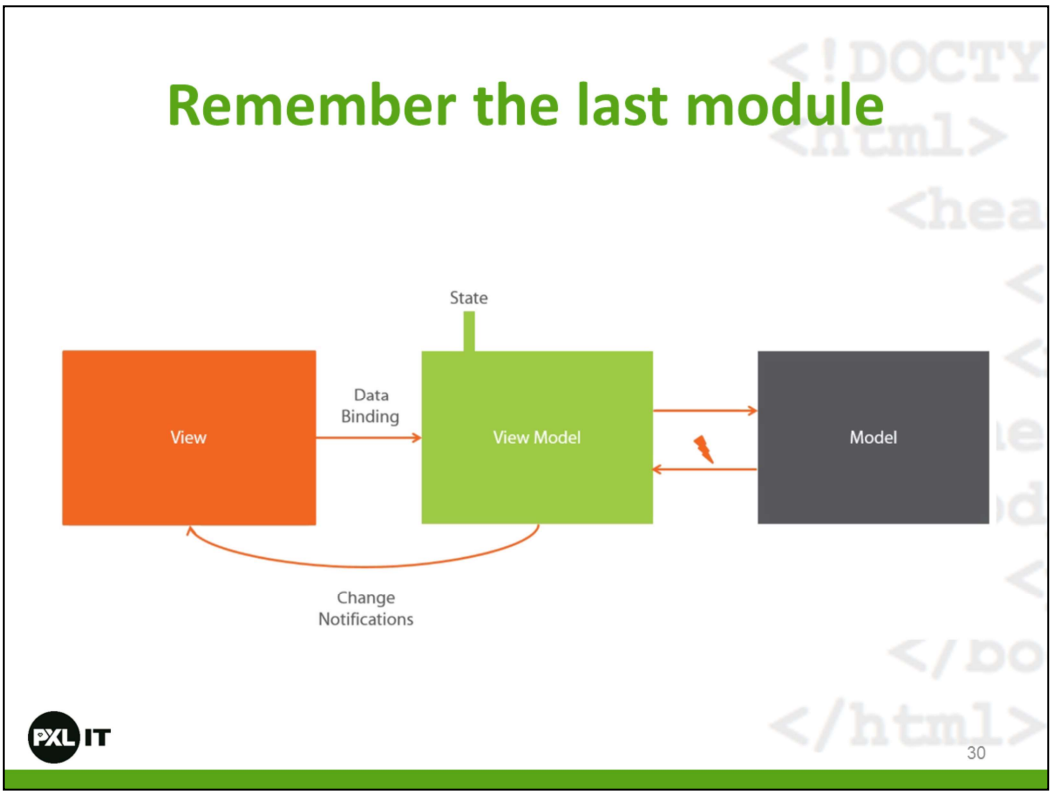
27




The targets of this module are mainly that you learn to understand how you can fill the voids in what we have learned so far in our MVVM base application. How can we trigger code, how can we invoke a viewmodel from another viewmodel and how can we cover all other missing pieces such as dialogs or navigation. I think it’s safe to say at the end of this lesson, we’ll have a solid understanding of all aspects of MVVM. Except for one thing: Testing.



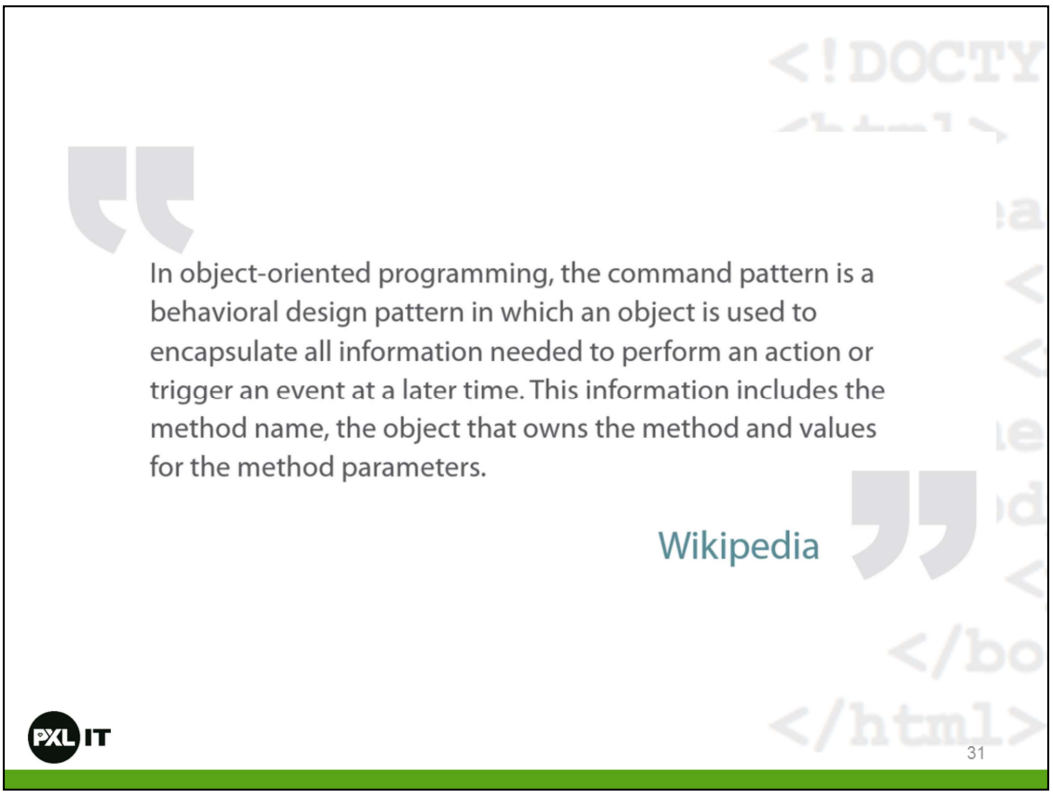

So we'll kick of this module by looking at Commanding. Commands are very common in MVVM, and play a crucial role in allowing users to interact with your application. Which is something that we haven't covered yet so far in our MVVM story.




You probably will remember this image from earlier in this course.

This image shows us the conceptual schema of an MVVM architecture where the view will bind on the ViewModel. The ViewModel is exposing properties for the view to bind on. This way we can display data in our views. Secondly, if the viewmodel data changes for some reason through the notification model, the view can be notified about this and thus update itself. This was done by implementing an interface already covered namely the INotifyPropertyChanged interface. With what I just summarized, we are limited to apps which will just show data.

Of course we'll also need to allow the users to interact with the application somehow. Clicking a button is traditionally something we would handle in code-behind. We write code in an event handler  which will be handling whatever we want to do when the user has actually clicked that particular button. Of course the same goes for many other interactions such as repositioning the thumb of a progress bar or selecting an option in a combobox. To cover this our ViewModels will not only expose state for properties. Instead they will also expose operations in the form of commands. And again, through databinding we will be able to link these commands from our view so will still have loose coupling.



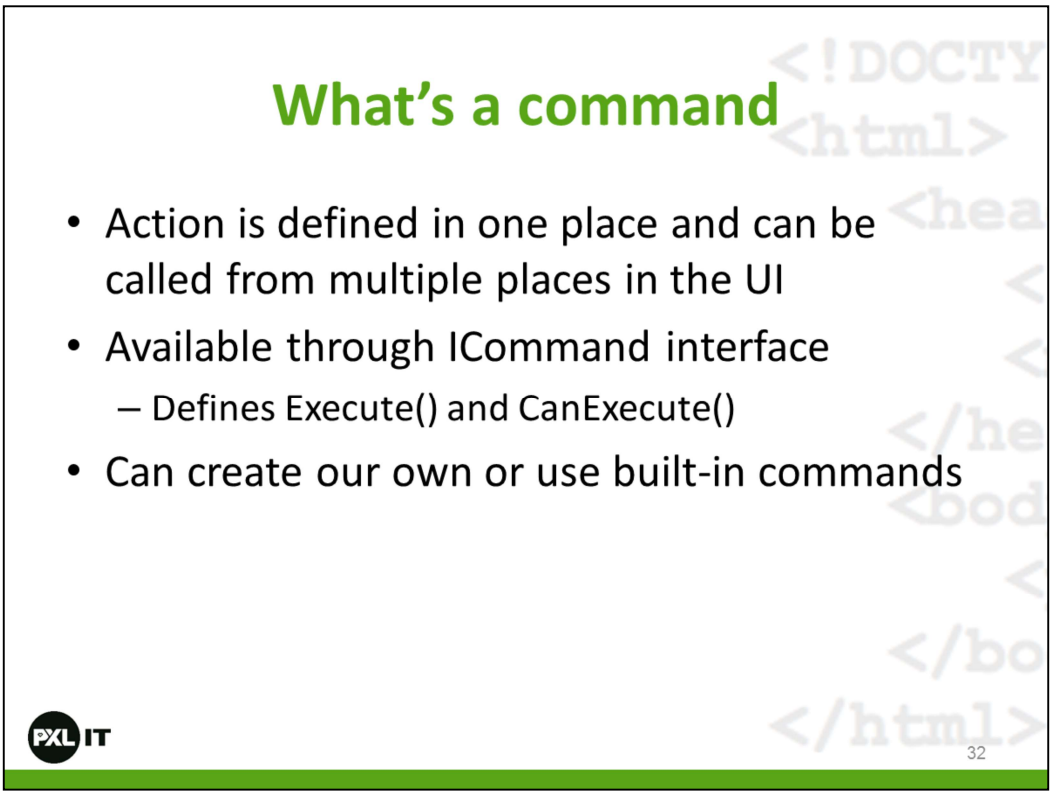

Before we look further at how commands can help us when we use MVVM, lets start by looking what exactly is a command. Commands are basically implementations of the command pattern. If you're not familiar with the software pattern, a simple search on the internet brings us to this wikipedia page, where the pattern is explained. There we can find the following definition.

A quick translation lets say basically says that a command is an object that wraps all information to be able to react to a trigger at some later point

References:

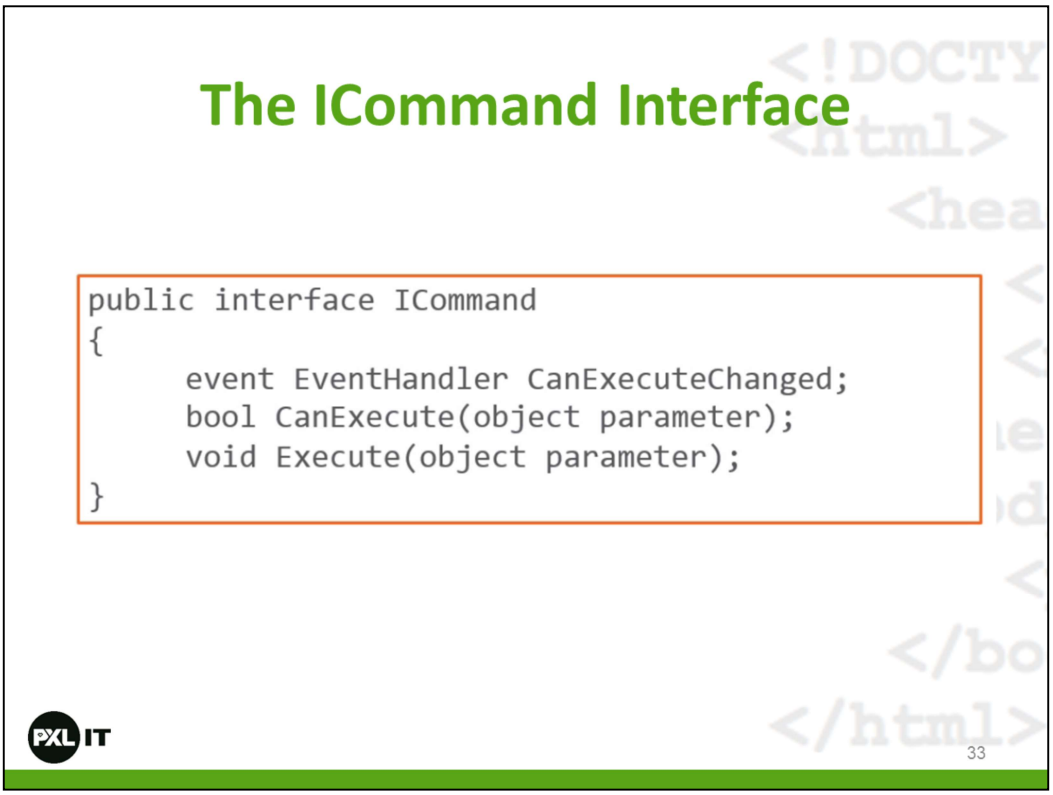
https://en.wikipedia.org/wiki/Command_pattern

http://www.dofactory.com/net/command-design-pattern

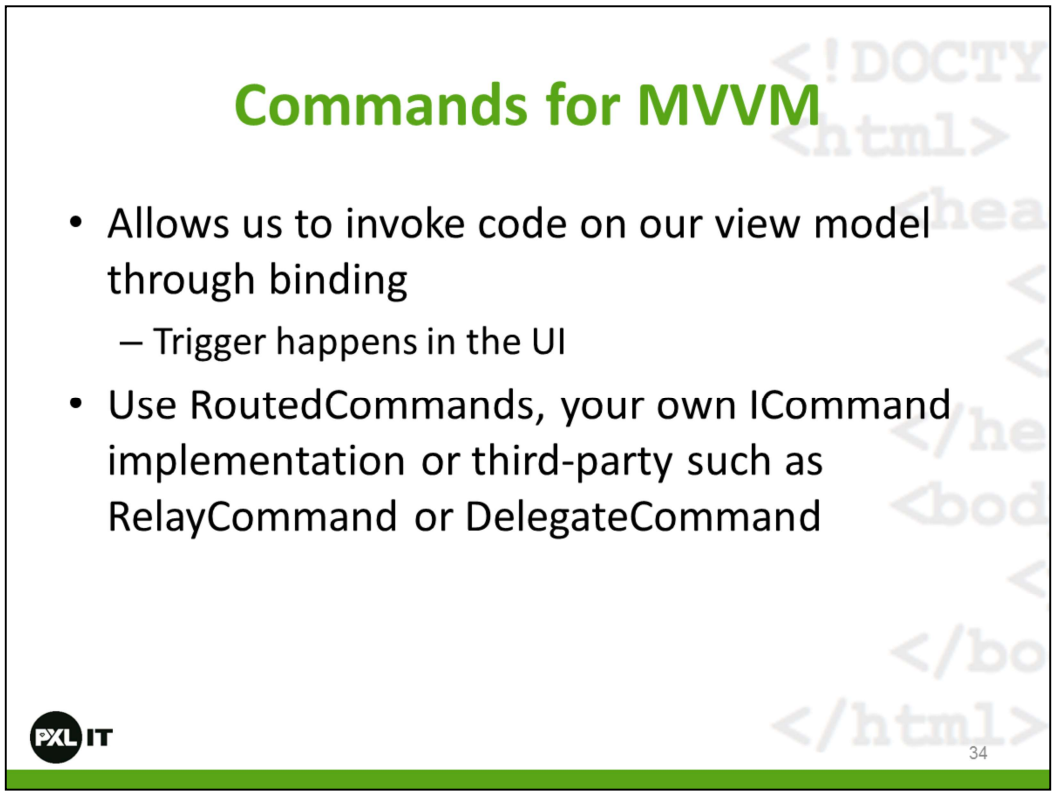



Commands have been introduces with WPF from the very beginning. Using commands, Microsoft tried to solve a common problem which had existed for a very long time with winforms. The problem was that very often the same functionality had to be executed from different places in the UI. For example, we may have a button in a toolbar as well as a menu item in a menu both doing the same thing. Typically, this was solved by writing two event handlers and have both of these point to a common function.

Now using commands, we could write the functionality in one location and refer to that one location from different entry points in the UI.
In WPF commands have been introduced with the ICommand interface. This interface which we'll see in just a second, is pretty simple. It defines an Execute and a CanExecute. You can build your own commands or use one of the built-in commands. WPF comes with about 100 built-in commands.




On this slide here, we see the ICommand Interface. As mentioned, it's a pretty simple interface which defines just one method: Execute, a boolean CanExecute and an event CanExecuteChanged. The Execute method will contain the code we want to execute when the command is actually being invoked. CanExecute will return a boolean which indicates if a command can be called at this point. CanExecuteChanged, the event, can be raised to make sure that WPF will re-evaluate the value of CanExecute.

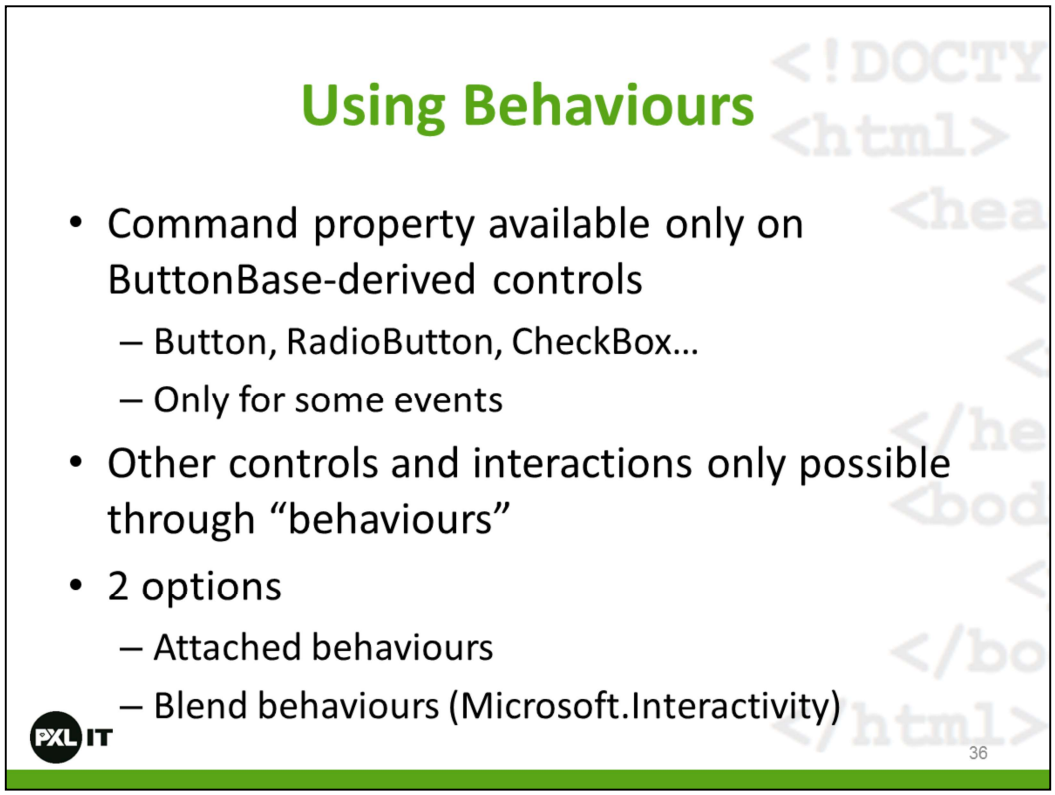



Now this may be a bit confusing, where is the actual link with our MVVM approach? While using MVVM we'll also be creating ICommand implementations. Instances of this implementation, so commands, will be exposed on our ViewModels. Basically every interaction we want to enable or expose from the viewmodel, needs to be exposed as a command. The view can then bind using the command property from the control and so when the user triggers the control, the bound command will be invoked. Using this approach, we are continuing to have a loose coupling between our views and our viewmodels. The only link is again data binding. WPF comes, as I said, with a couple of built-in command implementations. The RoutedCommand is probably the most commonly used type of Command. For MVVM scenarios, we have the option to create our own or use a third-party implementation such as RelayCommand, which comes with MVVMLight, or DelegateCommand, which comes with PRISM.

# DEMO: using Commands


PXL IT

35




In the demo that we have just seen, we have learned how we can create commands, and invoke code in our viewmodels using these commands. However, we have quickly hit a wall, the Commands are only available on a limited number of controls. There must be a solution to this problem. In this module, we’ll take a look at a solution to fix this. Let’s take a look at Behaviours.

As you have seen, the command property was only available on classes which inherit from ButtonBase. So that would be a button, a radiobutton, a checkbox,… and a few others. And even for the button, only the click event can be used in combination with the command. So there’s no support for example for DoubleClick. And also for other controls this would mean that we can’t invoke commands when the user is selecting an object in a drop-down, or clicking on a slider, that would be a huge limitation. But luckily for us it is possible to go around this. And ths solution lies in a Behaviour.

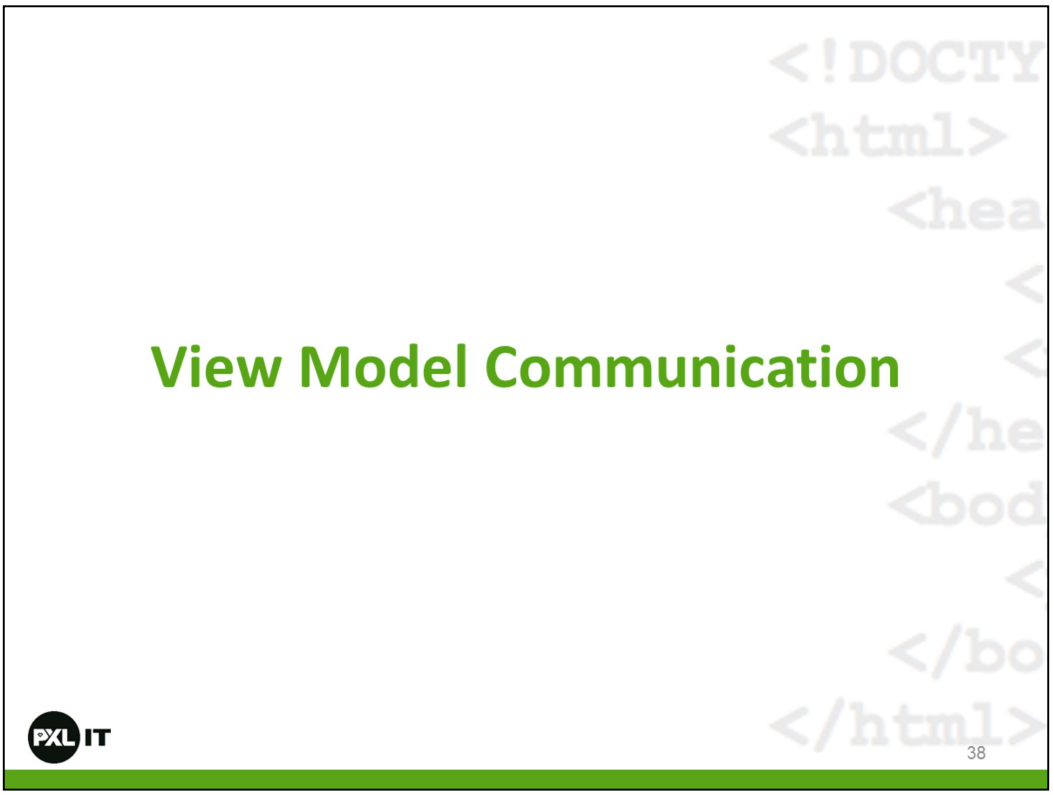
Now first, what is a behaviour? A behaviour lets us to wrap functionality in a reusable component which can then be attachted on to a control. This way we are extending the behaviour of a control, without changing the code of the original control. We’re just adding to its funcationality. The behaviours can be compared with extension methods in C#, but of course, this time in XAML. When talking about behaviours, there are two ways of working with them. You

can use an attached behaviour, or you can use a Blend behaviour.

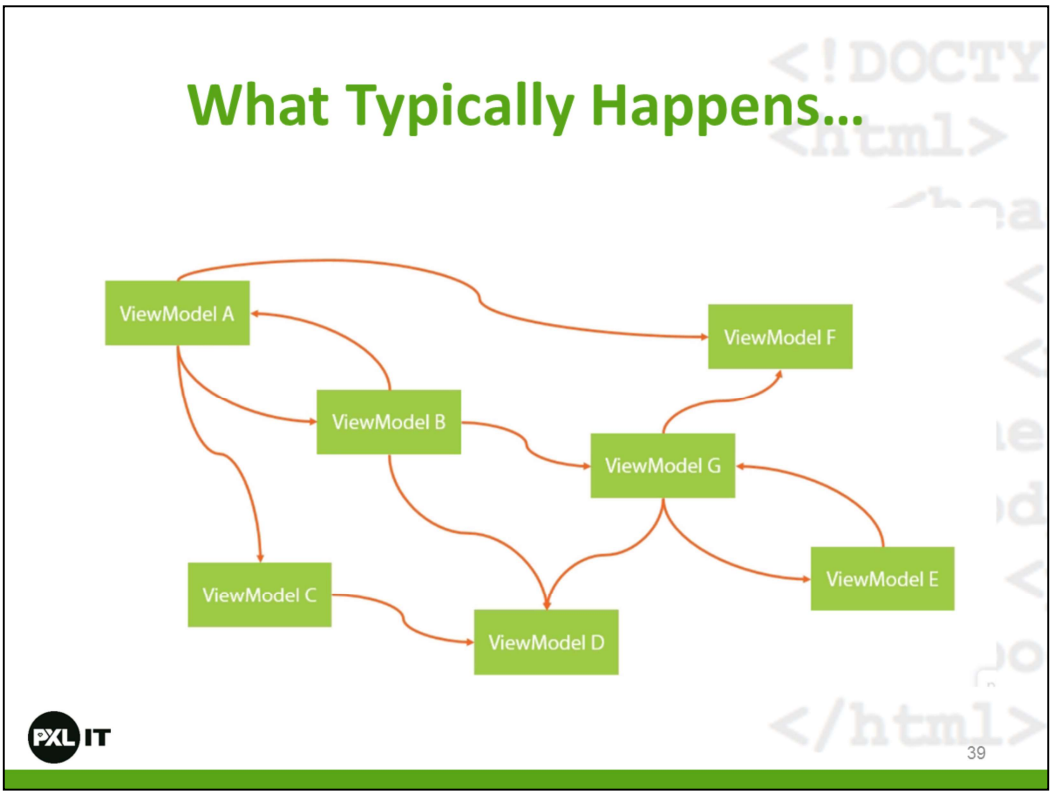
I won’t go into detail to explain the difference between the two, we will be using Blend behaviours. They are supported in the design interface in Blend as well as having a clean code structure that saves you from being vulnurable to memory leaks. Blend behaviours are supported in your projects, by adding a reference to Microsoft.Interactivity which comes with the Blend SDK. By default, this reference is automatically there for you, nowadays. Also notice that this approach is applicable again in other than WPF, XAML based technologies.

# DEMO: More Commands

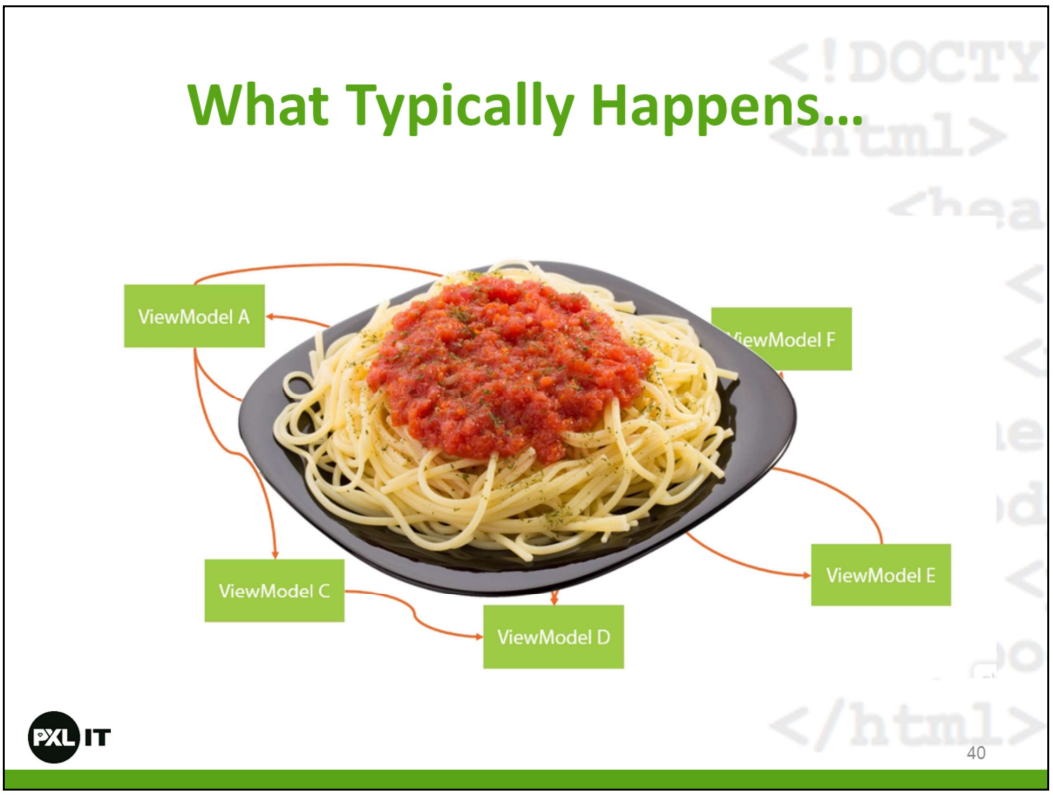

PXL IT



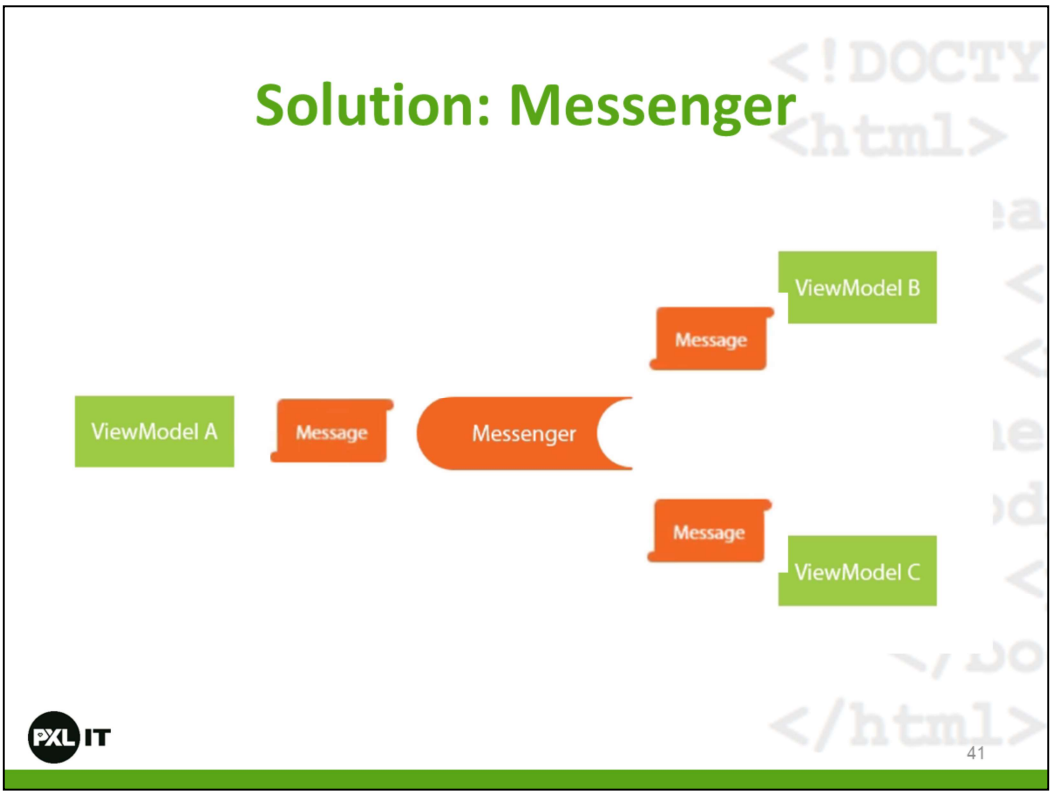

Another thing that we need to get working in our application is the abbility to pass data from one viewmodel to another. In other words, how can we have our viewmodels communicate with each other.




If you think back of module 2, where we looked at the advantages of MVVM, one of the most important ones was the ability to have better separation of concerns. In the end you want to have viewmodels which are easy to test in isolation, in other words, we don't want to have hard links and we also want to be able to mock our dependencies. In about any application, we will end up with quite a lot of viewmodels. Remember that in general, we have one viewmodel per view. So it is easy to get an idea on the average amount of viewmodels, you'll typically have in your own applications. These viewmodels need to communicate with each other. A Viewmodel A will need to pass data to viewmodel B, and Viewmodel B in turn needs  data from viewmodel G, and Viewmodel G also needs to pass data back to Viewmodel A and so on and so on… I think you get the picture.



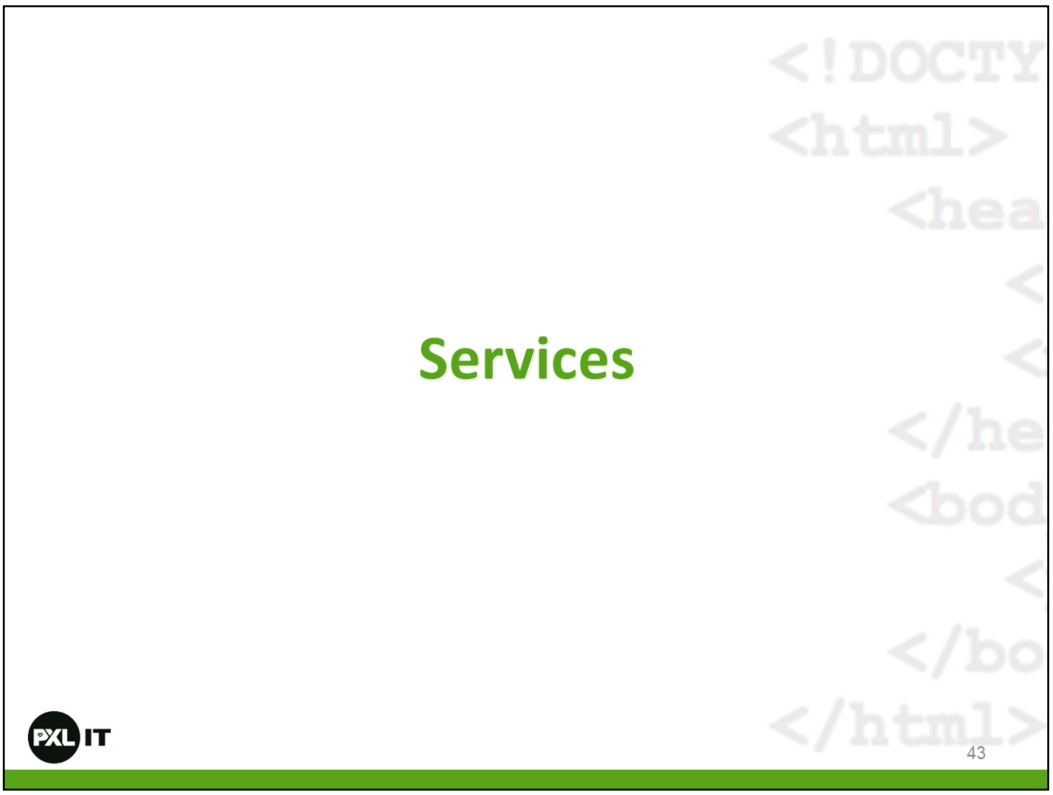

. The problem we are creating, is that all of this communication paths, by default, are creatig hard references. If Viewmodel A has a reference to Viewmodel B, and Viewmodel B will have a reference to Viewmodel A, so you'll end up with a spaghetti of links between viewmodels. And also you like spaghetti, it's not something you like in your code. Whoever wants to test Viewmodel A will run into problems, because we have a tight dependency on Viewmodel B, which can be hard to mock up. It's not what we want, it's not why we were creating applications with MVVM in the first place.




The solution for this problem is using a mediator or a messenger. Basically, the messenger is the middle man in the communication between different Viewmodels. Everyone knows the middle man, and they all communicate through that middle man. Take a look at the items on this slide. Again we have a couple of Viewmodels. Imagine that Viewmodel A needs to say something to Viewmodel B and Viewmodel C. As mentioned in the previous slide, by default, we would create a reference to Viewmodel B and Viewmodel C inside your Viewmodel A. It is, like I said, not what we want. Instead we are adding a messenger. And we'll have Viewmodel A register with this messenger. It will say to this messenger that it may at some point in the future, send messages of a certain type. Next, Viewmodels B and C are interested in what Viewmodel A has to say. So they will say to the messenger, Whenever you receive a message of that particular type, let us know.  So with this, we have succeeded in allowing the different viewmodels to talk with each other in a loosely coupled way. There are no hard references between the different viewmodels.

# Demo: introducing a messenger


PXL IT

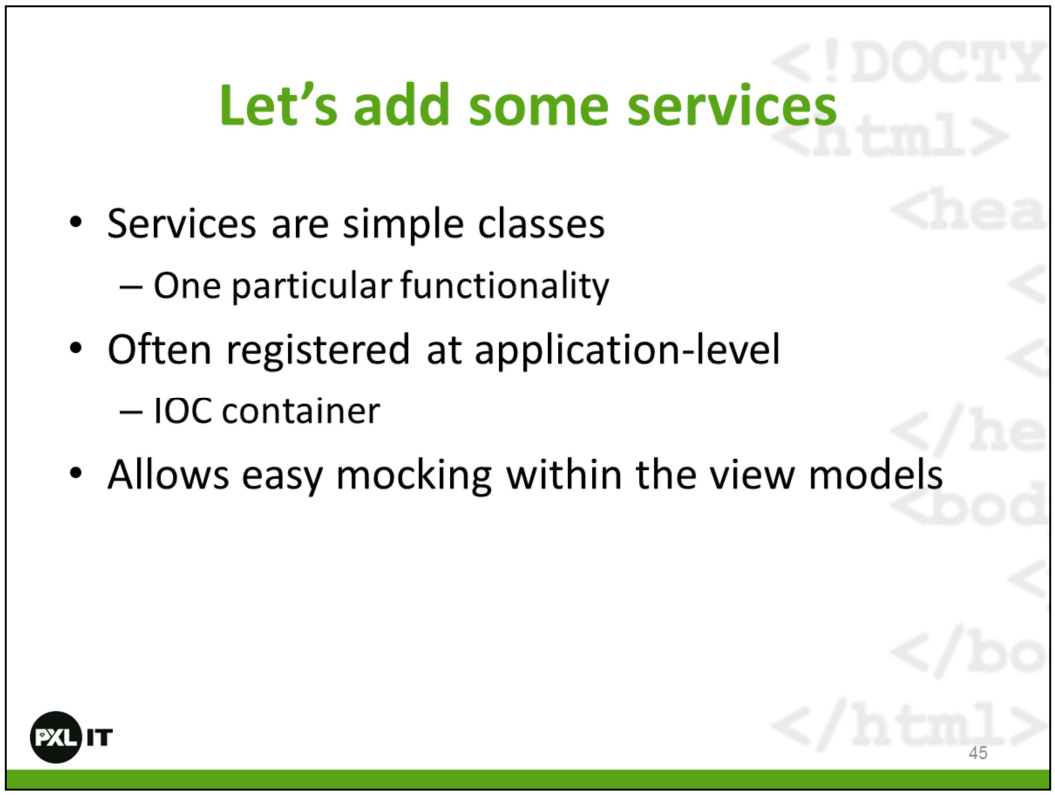



We have one more topic to cover in this module, and in this last part, I'll be talking about services in an MVVM setup. Services is a word which has been overused in software development so many times it's getting devious. So let me explain what I mean with services in the context of MVVM


When we think of where we are at this point in our application, it is pretty obvious that we are still missing quite a few things. We haven't covered anyting yet how to navigate from one screen to another. How can we open a detail window or navigate to a detail window In the same area, how can we display a dialog in our application that shows an error message.

We talked a lot about the model, but how do we work with that model. We need some way of accessing the model from the Viewmodel. And there are quite e few other places in our application where we will need to do things we can't simply put in the viewmodel and definitely can't put in the view.

But still, these loose ends somehow need to be covered in order to have a fully working application which was the intention of this course in the first place.




A solution to all these problems is 'Services'. Now what are services in this context?

By saying services, I mean simple classes which typically have one specific task. For example to dislay dialogs in our application, we'll introduce a class which does exactly that: showing dialogs, that's its responsability. We'll need at some point, need to display a dialog. This need will rise in the Viewmodel. Perhaps to say to the user that a save operation in the back-end was successful. It is not the job of the Viewmodel to actually show this dialog. Remember that we need to adhere to the single responsability as much as possible. So it is not the task of the Viewmodel to know how it should be displaying a dialog. The only thing that the Viewmodel does is saying that we need to display a dialog. For this very reason, we'll have our dialog service class. So what will happen is that the Viewmodel will have a reference to this dialog service class and ask it to do that actual display of this dialog. All the types of services, such as the navigation service, data service,…. work in the same way and have the same purpose: being used by the Viewmodel to perform certain tasks, so that the Viewmodel itself, doesn't need to know how to actually do it.

Very often, you'll see that for our services classes, a single instance is kept at the application level. In many applications in architectures you'll see an IOC

container being used for this. IOC stands for **inversion of control**. In the next module, we'll be using a very simple IOC implementation. In the end, the service classes we created this way, are helping with creating an easy to test code-base. We can plug our service classes into our viewmodels, for example through the constructor. If we then unit-test the viewmodels, we can plugin our mock version of the service, resulting in code which is easier to test. And creating that type of code, was one of the main goals of using MVVM.

# Demo: Adding Services


PXL IT

# Summary

Commands give us the ability to trigger code in the View Model based on UI interactions

Messenger creates loose coupling between View Models

Services fill the gaps we still had

PXL IT

47