

90377: "Design Patterns in C#"

Lab Manual

Wincubate ApS
20-10-2020



Table of Contents

Exercise types	4
Prerequisites.....	4
Module 2: “Abstract Factory”	5
Lab 02.1: “Tasty Factories and Products”	5
Lab 02.2: “Reflection Factory” (⭐⭐)	7
Module 03: “Builder and Fluent API”	9
Lab 03.1: “Creating Very Simple Fluent APIs” (⭐).....	9
Lab 03.2: “A Much Better Fluent API” (⭐⭐⭐).....	11
Module 04: “Decorator”	13
Lab 04.1: “Decorating Vehicles with Workshop Info” (⭐).....	13
Module 05: “Façade”	15
Lab 05.1: “Provide a Price Search Façade” (⭐).....	15
Module 06: “State”	17
Lab 06.1: “Maintainability of State Implementations”	17
Module 07: “Observer”	18
Lab 07.1: “An Alternative Observer Pattern implementation in .NET 4.0” (⭐⭐⭐).....	18
Module 08: “Command”	19
Lab 08.1: “Commands with Parameters” (⭐).....	19
Module 09: “Repository”	20
Lab 09.1: “SQL Repository” (⭐/⭐⭐).....	20
Lab 09.2: “Using Repository in Practice” (⭐⭐).....	23
Module 10: “Unit of Work”	24
Lab 10.1: “Asynchronous Versions of Repository and Unit of Work” (⭐⭐⭐).....	24
Module 11: “Proxy”	25
Lab 11.1: “Profiling with Proxy”	25
Lab 11.2: “Implementing Caching as a Proxy” (⭐⭐/⭐⭐⭐).....	26
If time permits... ..	27
Module 12: “Introduction to MVVM”	28
Lab 12.1: “Editing Customer using MVVM” (⭐⭐)	28
Model Implementation.....	28
ViewModel.....	28

View	28
Model Validation	29
Commands	29
If time permits... ..	29
Module 13: "MVVM Problems and Patterns"	30
Lab 13.1: "Simple Calculator in MVVM" (☆☆☆).....	30
Implement digits functionality	30
Implement the functionality for C.....	31
Implement the Result message box	31
Module 14: "The SOLID Principles"	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 course files accompanying the course to be extracted in some directory path, e.g.

C:\Wincubate\90377-2

with Visual Studio 2019 (or later) installed on the PC.

We will henceforth refer to the chosen installation path containing the lab files as *PathToCourseFiles* .

Module 2: “Abstract Factory”

Lab 02.1: “Tasty Factories and Products”

This exercise implements all aspects of the Abstract Factory Pattern in an example involving foreign cuisines. The overall structure of the solution will proceed in a manner similar to examples in the module presentation.

- Open the starter project in
PathToCourseFiles\Labs\02 - Abstract Factory\Lab 02.1\Starter ,
which contains a project called Cuisines.

Here you fill in all the additional code needed for implementing Abstract Factory.

Throughout this exercise a “foreign cuisine” (such as Italian or Indian) is an abstract factory interface letting the client create

1. A main course (e.g. pizza)
2. A dessert (e.g. tiramisu)

Consequently, there are two kinds of abstract products in the cuisine abstract factory: MainCourse objects and Dessert objects. These are already defined in the existing projects via the following two definitions:

```
interface IMainCourse
{
    void Consume();
}

interface IDessert
{
    void Enjoy();
}
```

Main courses should have a `void Consume()` method. The intention here is that concrete products should print to the console what is being consumed by the client.

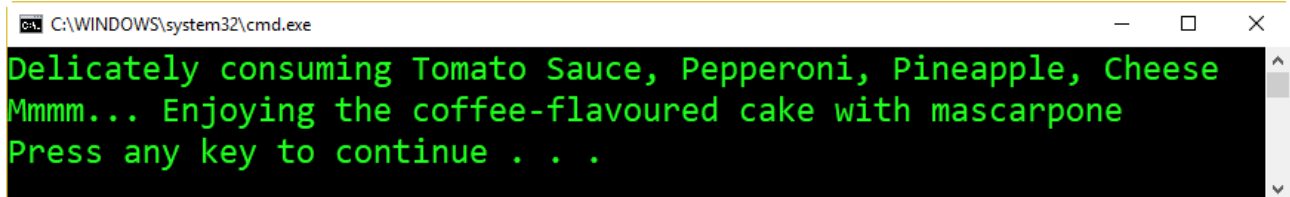
Desserts should have a `void Enjoy()` method. When invoked it should print to the console reflect what is being enjoyed by the client.

You will start by implementing an Italian cuisine using the Abstract Factory Pattern

- Implement a concrete main course product called `Pizza`
 - Its constructor should accept a sequence of topping strings.
- Implement a concrete dessert product called `Tiramisu` (without additional members)
- Create the appropriate abstract factory interface for cuisines called `IMealFactory`.
- Create a concrete factory class for the Italian cuisine, where
 - the main course being created is a pizza with “Tomato Sauce”, “Pepperoni”, “Pineapple”, and “Cheese”

- the dessert is a tiramisu,
- Test your implementation by adding the appropriate client code in Program.cs.
 - Invoke `IMainCourse.Consume()` on the created main course object.
 - Invoke `IDessert.Enjoy()` on the created dessert object.

When you run the program, the output should be the following (or equivalent):

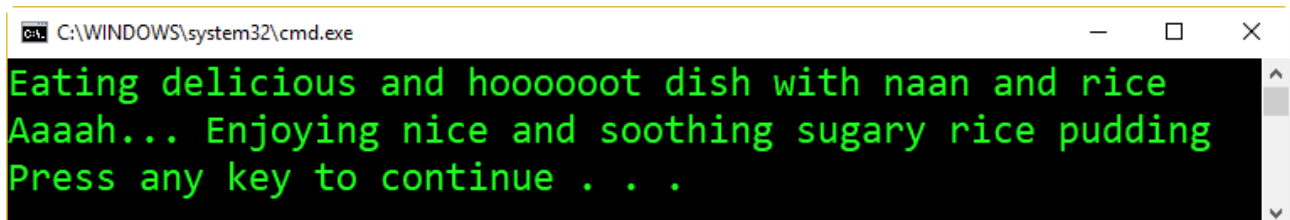


```
C:\WINDOWS\system32\cmd.exe
Delicately consuming Tomato Sauce, Pepperoni, Pineapple, Cheese
Mmmm... Enjoying the coffee-flavoured cake with mascarpone
Press any key to continue . . .
```

You have now implemented the Italian cuisine. You will then proceed to implementing the Indian cuisine as follows.

- Implement a concrete main course product called `ChickenCurry`
 - Its constructor should accept an integer indicating spicyness.
- Implement a concrete dessert product called `Kheer` (without additional members)
- Create the corresponding concrete factory class for the Indian cuisine, where
 - the main course being created is with a spicyness of 5.
- Test your implementation by changing only the Italian cuisine to the Indian cuisine in Program.cs.

When you run the program, the output should now be the following (or equivalent):



```
C:\WINDOWS\system32\cmd.exe
Eating delicious and hoooooot dish with naan and rice
Aaaah... Enjoying nice and soothing sugary rice pudding
Press any key to continue . . .
```

Lab 02.2: "Reflection Factory" (☆☆)

One often encounters variations of Abstract Factory patterns in everyday programming. Such hybrids can be quite helpful in creating "intelligent" factories.

In this exercise, we will investigate a mix between Factory Method and Abstract Factory which is sometimes employed in code to avoid compile-time dependencies to concrete classes. Below we will use the `System.Reflection` API to instantiate objects from a string description of the concrete object type.

- Open the starter project in
`PathToCourseFiles\Labs\02 - Abstract Factory\Lab 02.2\Starter` ,
which contains a project with several predefined `Pizza` classes.

The main `Pizza` class and related types are identical their definitions introduced in Lab 03.1. However, several additions have been made:

- An interface `IPizza` has been added to describe an abstract `Pizza` product
 - `Pizza` now implements this interface in order to be a concrete product.
- A number of concrete `Pizza` product classes have been added:
 - `ElDiabloPizza`
 - `HawaiiPizza`
 - `MargheritaPizza`
 - `MeatLoverPizza`

These are all ready to be instantiated by an Abstract Factory.

The following interface is already defined:

```
interface IPizzaFactory
{
    IPizza Create( string description );
}
```

Your task is now to create a concrete class implementing the Abstract Factory interface with a single Factory Method accepting a string parameter which describes the concrete `IPizza` object to create.

- Create a class `ReflectionPizzaFactory` implementing `IPizzaFactory` such that it processes the incoming description string and instantiates the corresponding concrete class, if the following statements are true
 - The "cleaned up" description string is the name of a type existing in the currently executing assembly.
 - The existing type implements the `IPizza` interface.

To be more precise,

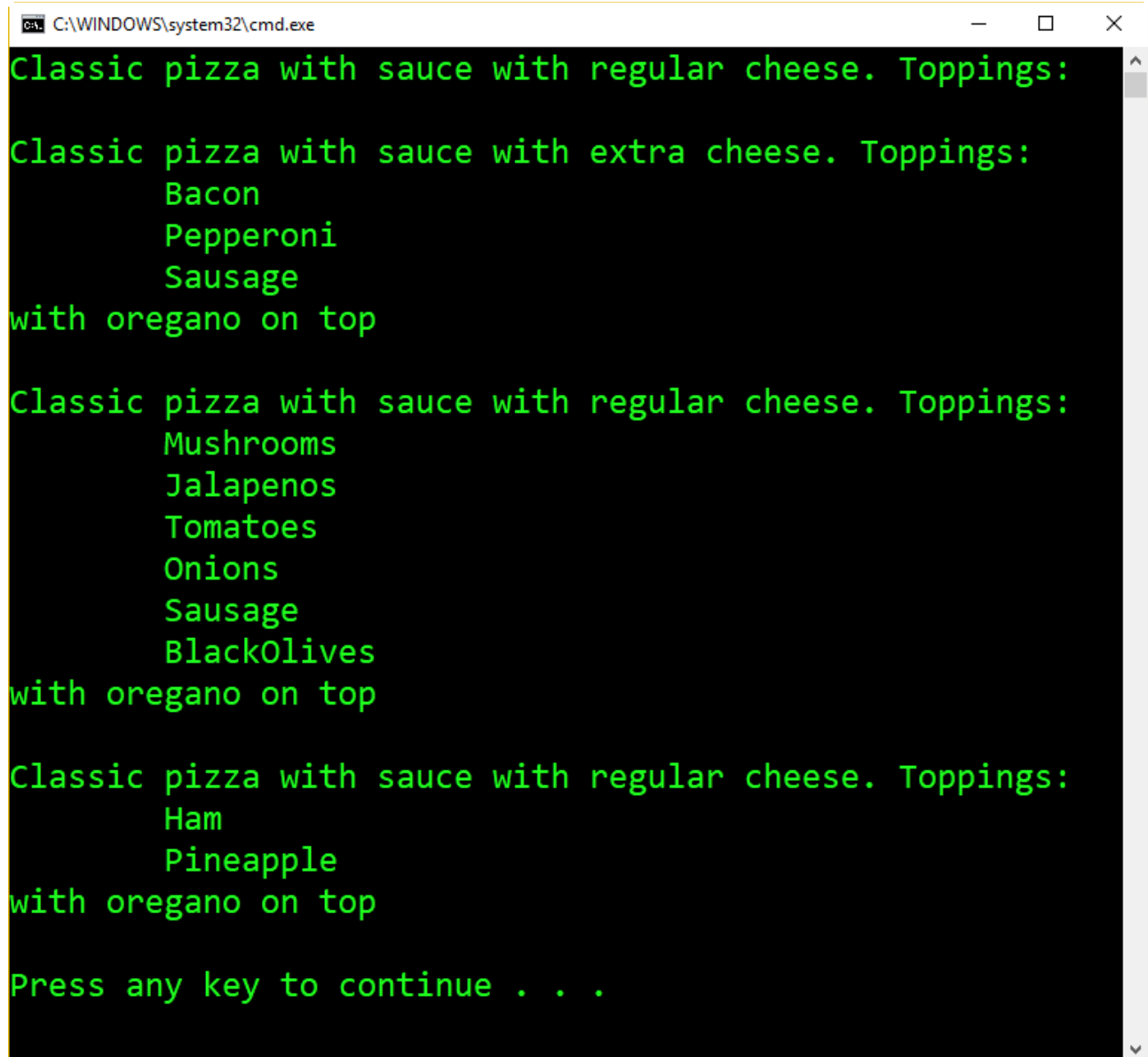
- "margherita pizza" will instantiate the type `MargheritaPizza`
- "meat lover pizza" will instantiate the type `MeatLoverPizza`
- "El Diablo pizza" will instantiate the type `ElDiabloPizza`
- "Hawaii Pizza" will instantiate the type `HawaiiPizza`

Note: Google is probably your friend when it comes to Reflection Kung-fu...!

As you complete your implementation;

- Test that your implementation works correctly by modifying the code in Program.cs accordingly.

When you run the program, the output should be the following:



```
C:\WINDOWS\system32\cmd.exe
Classic pizza with sauce with regular cheese. Toppings:
Classic pizza with sauce with extra cheese. Toppings:
    Bacon
    Pepperoni
    Sausage
with oregano on top
Classic pizza with sauce with regular cheese. Toppings:
    Mushrooms
    Jalapenos
    Tomatoes
    Onions
    Sausage
    BlackOlives
with oregano on top
Classic pizza with sauce with regular cheese. Toppings:
    Ham
    Pineapple
with oregano on top
Press any key to continue . . .
```


Module 03: “Builder and Fluent API”

Lab 03.1: “Creating Very Simple Fluent APIs” (★)

This exercise illustrates how to create a Fluent API for building pizza products using a variation of the Builder Pattern. Fluent APIs are quite popular in .NET for configuring the Builder instances in a “fluent” fashion, which is reminiscent of the flow in natural, spoken languages.

Consider the `Pizza` class defined as

```
class Pizza
{
    public CrustKind Crust { get; set; }
    public bool HasSauce { get; set; }
    public IEnumerable<ToppingKind> Toppings { get; set; }
    public CheeseKind? Cheese { get; set; }
    public bool Oregano { get; set; }
}
```

Then the well-known Hawaii pizza manually constructed in the following manner:

```
Pizza hawaii = new Pizza
{
    Crust = CrustKind.Classic,
    HasSauce = true,
    Cheese = CheeseKind.Regular,
    Toppings = new List<ToppingKind>
    {
        ToppingKind.Ham,
        ToppingKind.Pineapple
    },
    Oregano = true
};
```

could be built using an appropriate fluent API Builder as follows:

```
FluentPizzaBuilder builder = new FluentPizzaBuilder();
Pizza hawaii = builder
    .Begin()
    .WithCrust(CrustKind.Classic)
    .Sauce
    .AddCheese()
    .AddTopping(ToppingKind.Ham)
    .AddTopping(ToppingKind.Pineapple)
    .Oregano
    .Build();
```

Your task is now to create this `FluentPizzaBuilder` class.

- Open the starter project in
PathToCourseFiles\Labs\03 - Builder\Lab 03.1\Starter ,
which contains a project with the `Pizza` class and related types.
- Create the `FluentPizzaBuilder` class.
- Test that your class in the Fluent API definition correctly build a Hawaii pizza instance equivalent to the manually created instance above.

Lab 03.2: "A Much Better Fluent API" (☆☆☆)

This exercise examines how to create a better Fluent API for building pizza products.

The Fluent API solution to Lab 03.1 is simple and not too difficult to create with a little bit of practice. But it is too simplistic for professional purposes due to a number of problems:

1. Any order of invoking the fluent methods is allowed
2. Repetitions of the fluent methods are allowed
3. All methods are essentially optional (as well as repeatable)
4. It uses properties containing getters with side effects.

Your task is now to remedy all these deficiencies.

- Open the starter project in
PathToCourseFiles\Labs\03 - Builder\Lab 03.2\Starter ,
which contains a project with the solution to Lab 03.1.
- You should create a better fluent API solution, which is statically safe in the sense that the compiler will only allow fluent method sequences which are legal.

More specifically, this sequence should be allowed by the compiler:

```
Pizza hawaii = new FluentPizzaBuilder()
    .Begin()
    .WithCrust(CrustKind.Classic)
    .WithSauce()
    .AddCheese()
    .AddTopping(ToppingKind.Ham)
    .AddTopping(ToppingKind.Pineapple)
    .WithOregano()
    .Build();
```

This sequence should also be allowed by the compiler:

```
Pizza hawaii = new FluentPizzaBuilder()
    .Begin()
    .WithCrust()
    .WithoutSauce()
    .AddCheese()
    .AddTopping(ToppingKind.Ham)
    .AddTopping(ToppingKind.Pineapple)
    .Build();
```

However, this sequence should **not** be allowed by the compiler:

```
Pizza hawaii = new FluentPizzaBuilder()
    .Begin()
    .WithOregano()
    .WithCrust(CrustKind.Classic)
    .WithSauce()
    .AddCheese()
```

```
.AddTopping(ToppingKind.Ham)
.AddTopping(ToppingKind.Pineapple)
.Build();
```

Make sure that:

- Building always begins with `Begin()`
- Building always completes with `Build()`
- Proper defaults are chosen, e.g for `WithCrust()`
- Some methods are optional choices, e.g. `WithOregano()`
- Some choices are mandatory, e.g. `WithSauce()` vs. `WithoutSauce()`
- The compiler allows only correct sequences, which are in the “usual” order:
 - Crust,
 - Sauce,
 - Cheese,
 - Any sequence of toppings, and finally
 - Oregano

Module 04: “Decorator”

Lab 04.1: “Decorating Vehicles with Workshop Info” (★)

This exercise uses the Decorator Pattern for producing another, different decorator to the `IVehicle` hierarchy introduced during the presentation.

- Open the starter project in
`PathToCourseFiles\Labs\ 04 - Decorator\Lab 04.1\Starter` ,
which contains two projects:
 - An external library with the `IVehicle`-related types from the presentation.
 - A console application called `More Decorators` using the types of the library.

The console application contains a number of calls to an object of type `ServicedVehicle`, which does not exist (yet!), so the program does not currently compile.

Your task is to develop the `ServicedVehicle` class using the Decorator Pattern such that it maintains information about the service information and log for a vehicle being regularly serviced by a mechanic at a workshop.

More precisely, the `ServicedVehicle` class keeps track of

- `int` `NextServiceKilometers`
- `int` `ServiceIntervalKilometers`
- `IEnumerable<ServiceLogEntry>` `LogEntries`

and, moreover issues, a warning when the travelled number of kilometers exceed the kilometer count for the next expected service.

The Library already a `Workshop` class which facilitates the following service operation:

```
Workshop workshop = new Workshop();  
workshop.Service( servicedBmwX1 );
```

Now locate the following TODO in code

```
public class ServicedVehicle  
{  
    // TODO  
}
```

- Implement the `ServicedVehicle` class by implementing the TODO such that the existing program in `Program.cs` when run produces the output described in the screenshot below
 - Note: You should not change any existing classes in `Library` or `More Decorators`.

Expected screenshot should look like the following:

```
Microsoft Visual Studio Debug Console

Car:
    BMW X1 [Black] (Travelled: 0 km)      Sport / 5 doors
Next Service: 20000
-----
Driving...
Exceeded next service kilometers by 5176 km
Car:
    BMW X1 [Black] (Travelled: 25176 km)   Sport / 5 doors
Next Service: 20000
-----
At workshop...
Car:
    BMW X1 [Black] (Travelled: 25176 km)   Sport / 5 doors
Serviced at 25176 km by Mike Megamotor
Next Service: 40000
-----
Driving...
Car:
    BMW X1 [Black] (Travelled: 38056 km)   Sport / 5 doors
Serviced at 25176 km by Mike Megamotor
Next Service: 40000
-----
At workshop...
Car:
    BMW X1 [Black] (Travelled: 38056 km)   Sport / 5 doors
Serviced at 25176 km by Mike Megamotor
Serviced at 38056 km by Mike Megamotor
```

Module 05: “Façade”

Lab 05.1: “Provide a Price Search Façade” (★)

This exercise uses the Façade Pattern for providing a simple price lookup service for a very simple web shop.

- Open the starter project in
PathToCourseFiles\Labs\05 - Facade\Lab 05.1\Starter ,
which is essentially a simple web shop with pricing and currency conversion abilities. It contains three projects:
 - A Financial library providing currency conversion among the DKK, USD, and GBP currencies through the *CurrencyConversionService* class.
 - A WebShop library providing
 - a *ProductStorage* class providing information about products in the web shop
 - a *PriceInfoStorage* class providing pricing information (in USD) on products in the web shop given a specified *ProductId*
 - An empty *WebShopPriceSearch* console application.

Your task is to create a *PriceSearch* facade class allowing clients simple access to search for a simple substring to access pricing information in DKK for the product names matching the specified search string.

Note: You’re not allowed to change any existing classes in *Financial* or *WebShop*.

- Provide a class *PriceSearch* which exposes the following method for clients to call

```
class PriceSearch
{
    ...
    public IEnumerable<PriceSearchInfo> Lookup( string searchName );
    ...
}
```

The *PriceSearchInfo* structure is given as

```
struct PriceSearchInfo
{
    public string ProductName { get; set; }
    public decimal PriceDkk { get; set; }
}
```

- Test your implementation by commenting in the lines in the predefined *Program.cs* accordingly. The following code fragment:

```
PriceSearch ps = new PriceSearch();
IEnumerable<PriceSearchInfo> searchInfos = ps.Lookup("Switch");

foreach (PriceSearchInfo si in searchInfos)
{
    Console.WriteLine( $"{si.ProductName} costs DKK {si.PriceDkk:f2}" );
}
```

}

should produce the following output:

A screenshot of a Windows command prompt window. The title bar at the top reads "C:\WINDOWS\system32\cmd.exe". The window has standard Windows window controls (minimize, maximize, close) on the right. The command prompt area has a black background with green text. The text displayed is:
Switch costs DKK 2129,33
Switch Controller costs DKK 408,28
Press any key to continue . . .
A vertical scrollbar is visible on the right side of the command prompt area.

Module 06: “State”

Lab 06.1: “Maintainability of State Implementations”

This exercise will add a new state in the State pattern setup from the module presentation.

- Open the starter project in
PathToCourseFiles\Labs\ 06 - State\Lab 06.1\Starter ,
where the project contains the well-known types that you have seen in the presentation:
 - `TimerSetup`
 - `ITimerSetupState`
 - `TimerSetupStateBase`
 - `NormalState`
 - `SetHoursState`
 - `SetMinutesState`
 - `CompletedState`

Your task is now to test the maintainability of the State pattern by allowing the user to also include seconds in the `CompletedState` being configured by means of the `TimerSetup` class.

- Add a new state `SetSecondsState` which allows the configuration of seconds.
- Modify the existing program accordingly to incorporate the new state

When you’re done, spend a few minutes considering the following questions

- Which classes were changed?
- Are there any kind of optimizations that you feel are compelling?
 - Consider which of those are in fact good ideas

Module 07: “Observer”

Lab 07.1: “An Alternative Observer Pattern implementation in .NET 4.0” (☆☆☆)

This exercise will provide an alternative to events when implementing the Observer pattern in C#.

- Open the starter project in
PathToCourseFiles\Labs\07 - Observer\Lab 07.1\Starter ,
which contains a fully functional event-based implementation developed during the module presentation.

In .NET 4.0 two new interfaces were – more or less unnoticed by the community – added to allow a different mechanism for implementing Observer. The new interfaces are defined as follows:

```
public interface IObservable<out T>
{
    IDisposable Subscribe( IObserver<T> observer );
}

public interface IObserver<in T>
{
    void OnCompleted();
    void OnError( Exception error );
    void OnNext( value );
}
```

Your goal in this exercise is to refactor the existing solution to these new interfaces.

- Removed the event-specific aspects of the starter project and refactor the solution to use the two new interfaces for the existing *StockMarket* and *StockObserver* classes, respectively.
 - You will probably need to investigate the online documentation
 - Note: Don't worry too much about race conditions and thread-safety issues.

Module 08: “Command”

Lab 08.1: “Commands with Parameters” (★)

This exercise will illustrate a variation of the Command pattern, which extends Commands with parameters.

- Open the starter project in
PathToCourseFiles\Labs\08 - Command\Lab 08.1\Starter ,
which contains a class `LedLight` which has an interface allowing the client to set an intensity percentage between 0 and 100.

Your job is to implement the Command pattern which allows the commands to have an intensity parameter.

- Mimick the setup given in the examples of the presentation for this module and define classes – however with some variations!
 - Use the existing class `LedLight` as the Receiver.
 - `Program.cs` creates three instances of it to control through a `LoggingSwitch` defined below.
 - Create a suitable `SetCommand` class.
 - Create a `SetCommandFactory` implementing the existing `ISetCommandFactory` interface returning `SetCommand` instances.
 - Note: The factory needs to know about the three `LedLight` instances.
 - Implement a `LoggingSwitch` class which adds the command to be executed in a list before executing the command itself
 - Note: Since the commands are now parameterized, the `LoggingSwitch` cannot simply use a constant constructor-supplied command.
 - In this sense, the `LoggingSwitch` is merely a “log-and-execute” wrapper for commands generated via the command factory.

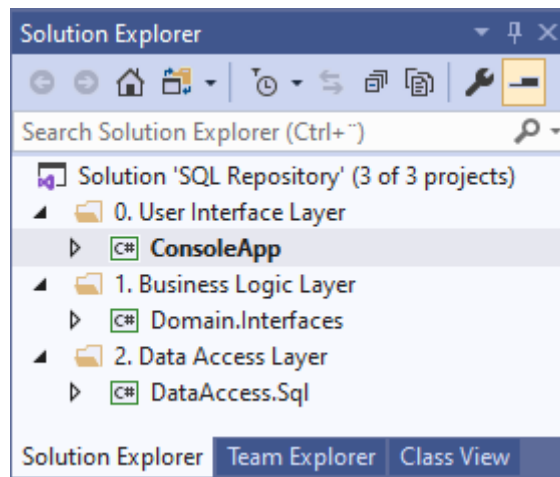
Module 09: “Repository”

Lab 09.1: “SQL Repository” (★/★★★)

This exercise will illustrate how the Repository pattern is used in practice with a SQL database.

- Open the starter project in *PathToCourseFiles\Labs\ 09 - Repository\Lab 09.1\Starter*, which constitutes a solution with three solution folders (one for each architectural layer) and three projects:
 - The project `ConsoleApp` which is an empty console application.
 - The project `Domain.Interfaces` containing the `Product` and `Category` types.
 - The project `DataAccess.Sql` containing the for `ProductContext` class, so you don’t need to perform SQL- or Entity Framework-specific code.

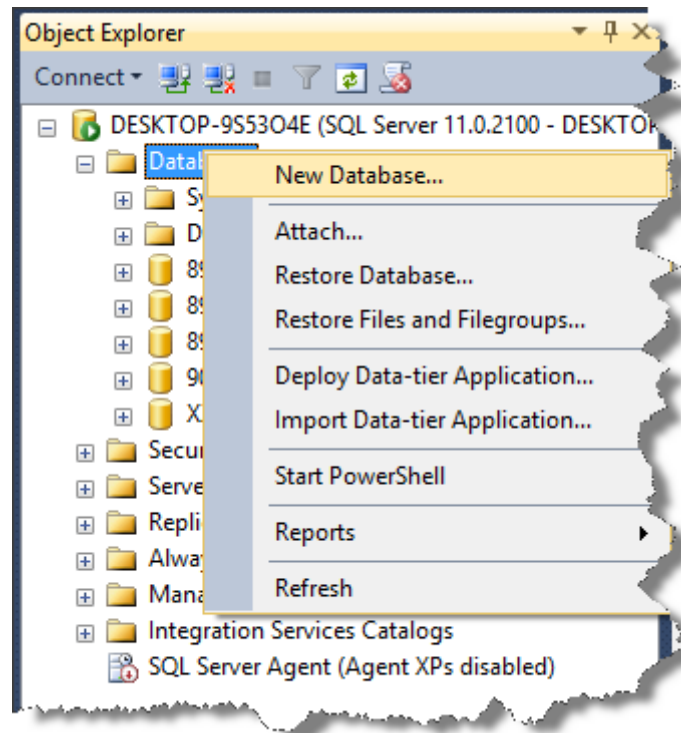
In the Solution Explorer in Visual Studio this is depicted as follows:



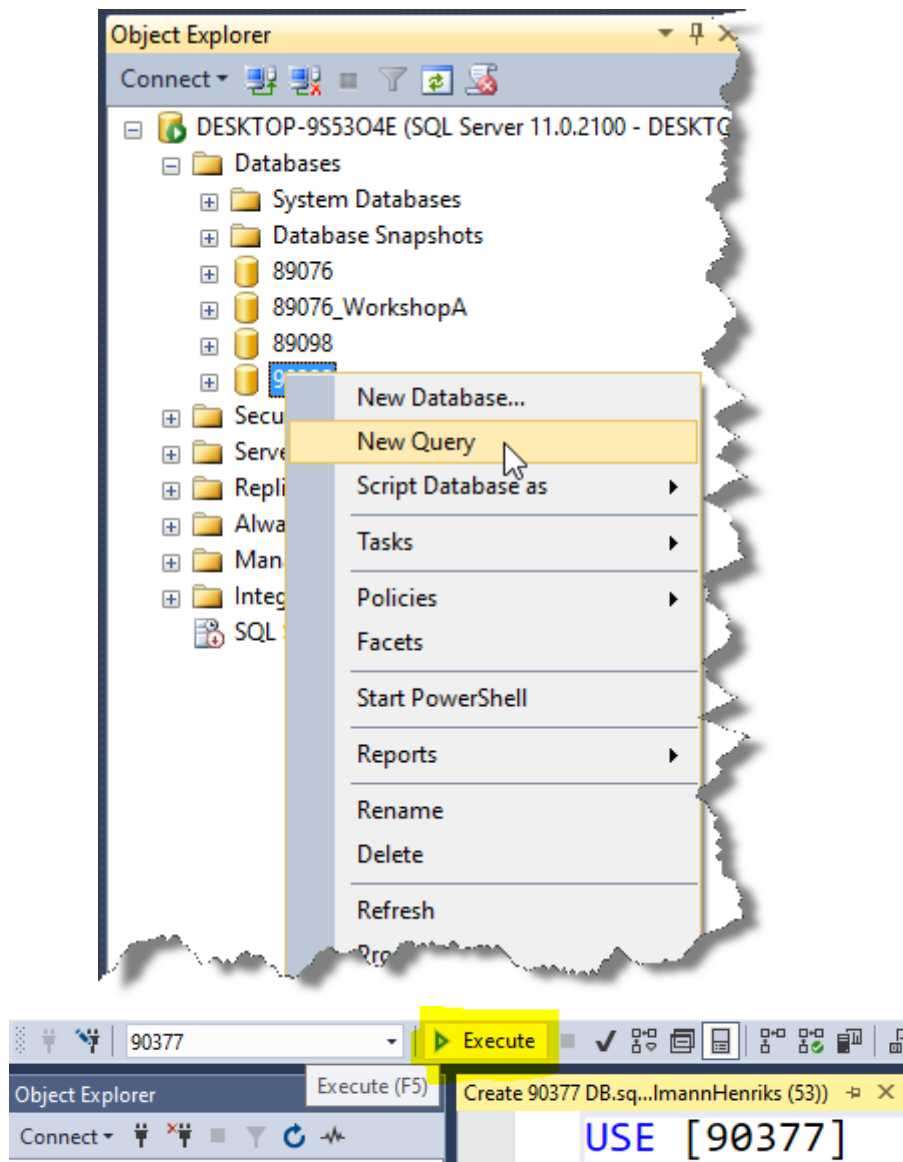
You will first need to create the database and tables used for the SQL-specific labs in some SQL database that you have access to.

Note: You are free to use either SQL Server, SQL Server Express, LocalDB, or whichever other SQL database depending upon what is available to you, but you might have to adjust the connection strings in the projects accordingly.

- Open **SQL Management Studio** and create a new database named 90377
 - Right-click the Database node and select New Database...



- Create the 90377 database using default settings.
- Generate the Movies SQL database from the scripts located in *PathToCourseFiles\Labs\Create 90377 DB.sql* by executing this script on the database just created.
 - Choose New Query and drag the script file to the query window
 - Click Execute (or press F5)



Your second task is now to implement the Repository pattern for products as illustrated in the presentation.

- Implement the Repository pattern for **Product** instances using whichever version of Repository that you see fit – Simple, non-generic, or generic!
 - Note: You probably need to slightly modify the connection string inside the **ProductContext** class of the **DataAccess.Sql** project.
 - Note: If you add new classes or files, then try to place them in the correct layer and update project references accordingly.
- Test your code by using the Repository to query all **Product** instances from the SQL database.

Lab 09.2: "Using Repository in Practice" (☆☆)

This exercise continues Lab 09.1 to modify and unit test an application based upon the Repository pattern.

If you have not completed that Lab 09.1, please complete that lab before proceeding.

- Open the starter project in
PathToCourseFiles\Labs\09 - Repository\Lab 09.2\Starter,
which constitutes a solution to Lab 09.1 with two additional (more or less empty) projects:
 - The project `Domain` contains a skeleton of the `ProductService` type.
 - The project `Domain.Test` contains an empty `ProductServiceTest` class.

You will now write a small piece of business logic by completing the implementation of the `ProductService` class, which currently contains:

```
public class ProductService
{
    // TODO: Complete the implementation of this class

    public IDictionary<Category?,int> ComputeGroupCounts()
    {
        throw new NotImplementedException();
    }
}
```

- Implement the `ComputeGroupCounts()` method such that it returns a dictionary mapping each category to the number of products in the specified category
- Run your implementation by suitably modifying `Program.cs`.

Secondly, you will of course need to unit test your implementation of the `ProductService` class.

- Consider how you would unit test the `ProductService` class, when it depends upon data which in a database?
- Implement as many unit tests as you like in the empty `ProductServiceTest` class
 - Note: A single test illustrating the point is adequate for our purposes! 😊

Module 10: “Unit of Work”

Lab 10.1: “Asynchronous Versions of Repository and Unit of Work” (☆☆☆)

This exercise investigates how to provide asynchronous versions of the Repository and Unit of Work patterns, as this would be the proper way to expose these operations in modern APIs.

- Open the starter project in
PathToCourseFiles\Labs\10 – Unit of Work\Lab 10.1\Starter,
which constitutes the outcome of the Module 10 solution providing a Unit of Work implementation for **Product** and **Comment** entities.

Your task is now the following:

- Suggest asynchronous versions of all relevant types showcased in Starter project above.
 - You will run probably run into a number of **Task**-specified annoyances and subtleties which you need to address appropriately.

Note: You will need to take several technical choices along the way, so be prepared to argue for (and/or against) specific choices that one might take in the endeavor.

Module 11: “Proxy”

Lab 11.1: “Profiling with Proxy”

This exercise uses the Proxy pattern for adding a timing aspect to an existing component whose source code we have no access to.

- Open the starter project in
PathToCourseFiles\Labs\ 11 - Proxy\Lab 11.1\Starter ,
which contains two projects:
 - An external library with the interface `IComputeOperation` and concrete class `ComputeOperation`
 - A console application invoking the library.

It seems that the library computes rather slowly when invoked as in Program.cs. We would very much like to instrument the `ComputeOperation` class with code for starting and stopping a `Stopwatch` to be able to measure the execution time of the `Compute()` method.

Unfortunately, this is an external library for which we cannot modify the source code, so we need to figure out an alternate way of proxying the existing library with `Stopwatch` measurements.

- Define an appropriate `Timing` proxy implementing `ComputeOperationDecorator`.
- Test your implementation by modifying Program.cs accordingly.

When you run your complete solution the last part of the output produced should be similar to the following:

```
86
88
90
92
94
96
98
Execution time was 00:00:05.5876512
Press any key to continue . . .
```

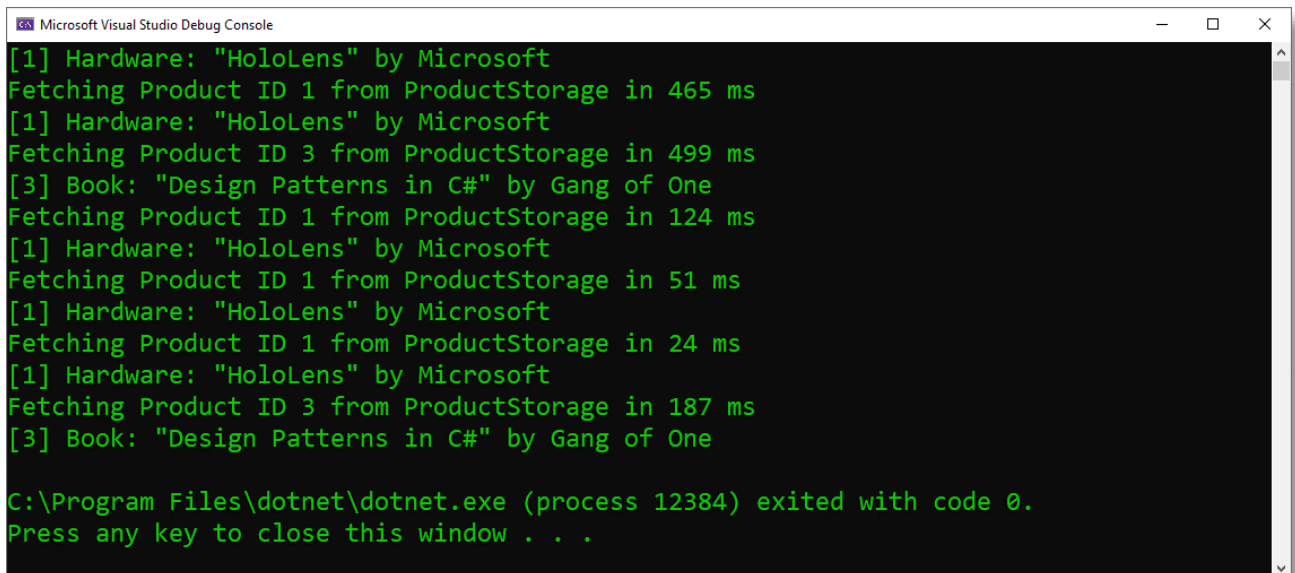
Lab 11.2: "Implementing Caching as a Proxy" (☆☆/☆☆☆☆)

This exercise extends the Web Shop example from the presentation by implementing product caching using the Proxy pattern.

- Open the starter project in
PathToCourseFiles\Labs\11 - Proxy\Lab 11.2\Starter ,
which essentially contains the project of the presentation containing the [Product](#),
[ProductStorage](#), and related classes and interfaces.

The starter project above contains some initial code in Program.cs, which retrieves the product objects with ID 1 and 3 a number of times. Unfortunately, the existing [ProductStorage](#) is rather slow at retrieving single [Product](#) instances by ID in the `GetById()` method.

A test execution of the existing implementation reveals the following depressing output:



```
Microsoft Visual Studio Debug Console
[1] Hardware: "HoloLens" by Microsoft
Fetching Product ID 1 from ProductStorage in 465 ms
[1] Hardware: "HoloLens" by Microsoft
Fetching Product ID 3 from ProductStorage in 499 ms
[3] Book: "Design Patterns in C#" by Gang of One
Fetching Product ID 1 from ProductStorage in 124 ms
[1] Hardware: "HoloLens" by Microsoft
Fetching Product ID 1 from ProductStorage in 51 ms
[1] Hardware: "HoloLens" by Microsoft
Fetching Product ID 1 from ProductStorage in 24 ms
[1] Hardware: "HoloLens" by Microsoft
Fetching Product ID 3 from ProductStorage in 187 ms
[3] Book: "Design Patterns in C#" by Gang of One

C:\Program Files\dotnet\dotnet.exe (process 12384) exited with code 0.
Press any key to close this window . . .
```

Your task is to improve upon these execution times by creating a caching Proxy called [CachingProductStorage](#) which caches the retrieved products instances such that multiple lookups of the same ID result in only a single lookup in the original [ProductStorage](#).

- Create a Proxy class [CachingProductStorage](#) such that the `GetById()` method caches duplicate lookups
 - Focus on only the `GetById()` method
 - **You can disregard caching in the `GetAll()` method for now.**
- Activate the proxy class in Program.cs and run your application.
- You should see a result along the lines of the following:

```
Microsoft Visual Studio Debug Console
Fetching Product ID 1 from ProductStorage in 259 ms
[1] Hardware: "HoloLens" by Microsoft
[1] Hardware: "HoloLens" by Microsoft
Fetching Product ID 3 from ProductStorage in 595 ms
[3] Book: "Design Patterns in C#" by Gang of One
[1] Hardware: "HoloLens" by Microsoft
[1] Hardware: "HoloLens" by Microsoft
[1] Hardware: "HoloLens" by Microsoft
[3] Book: "Design Patterns in C#" by Gang of One

C:\Program Files\dotnet\dotnet.exe (process 6048) exited with code 0.
Press any key to close this window . . .
```

Note:

- You're not allowed to change any existing classes in the WebShop class library, of course.
- Don't worry about thread-safety.
- It might be helpful to use the [MemoryCache](#) class which is referenced in .NET Core by adding the nuget package `Microsoft.Extensions.Caching.Memory` to the existing client project
 - You can assume that the underlying [ProductStorage](#) is only accessed through the caching Proxy so that cache never needs to be invalidated.
 - You don't need to implement [IDisposable](#). This will be the topic of a later module!

If time permits...

- How would the [CachingProductStorage](#) proxy work in conjunction with the [AdministratorsOnlyProxyStorage](#)?
- How would you implement caching in the `GetAll()` method?

If this is implemented correctly then adding the following lines

```
IEnumerable<Product> all = storage.GetAll();
IEnumerable<Product> all2 = storage.GetAll();
```

before all the `GetById()` invocations in `Program.cs` would produce a result similar to

```
Microsoft Visual Studio Debug Console
Fetching all products from ProductStorage in 20 ms
[1] Hardware: "HoloLens" by Microsoft
[1] Hardware: "HoloLens" by Microsoft
[3] Book: "Design Patterns in C#" by Gang of One
[1] Hardware: "HoloLens" by Microsoft
[1] Hardware: "HoloLens" by Microsoft
[1] Hardware: "HoloLens" by Microsoft
[3] Book: "Design Patterns in C#" by Gang of One

C:\Program Files\dotnet\dotnet.exe (process 11676) exited with code 0.
Press any key to close this window . . .
```

Module 12: “Introduction to MVVM”

Lab 12.1: “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
PathToCourseFiles\Labs\12 – Introduction to MVVM\Lab 12.1\Starter .
- Investigate the supplied project structure and files
 - Make a note of `Customer.cs` and the automatic properties it has.

Model Implementation

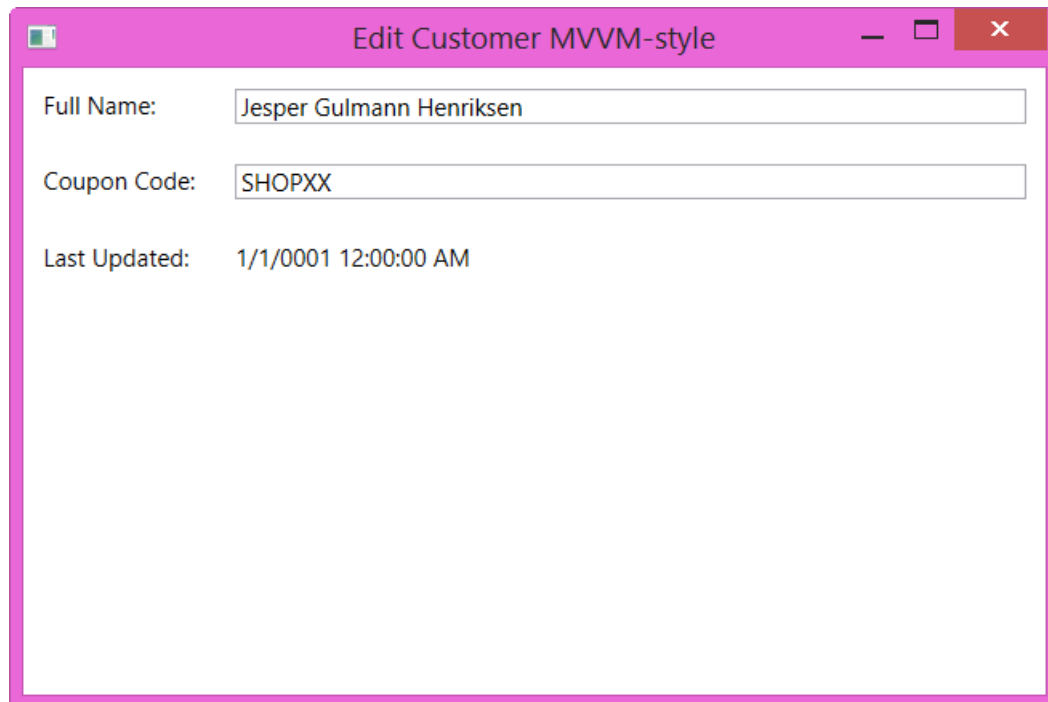
- Implement the `INotifyPropertyChanged` interface in the `Customer` class and modify all existing properties.

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
 - Full Name should consist of at least three characters
 - Coupon Code 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 `MainWindows.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 13: “MVVM Problems and Patterns”

Lab 13.1: “Simple Calculator in MVVM” (☆☆☆)

The purpose of this exercise is to re-implement the world simplest “calculator” using MVVM.

- Open the existing WPF project called “SimpleCalculator” in *PathToCourseFiles\Labs\13 – MVVM Problems and Patterns\Lab 13.1\Starter* .
- Verify that the solution contains UI and code-behind producing the following view:



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 starter project, 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 14: “The SOLID Principles”

No labs... Practice (and preach it!) for the rest of your lives! 😊