BS"D

dotNet 5786 mini Project - Stage 5

Table of Contents

Stage Goals 3

Instructions for Execution and Submission of the Stage 4

Outline - Stage 5 4

Specific Instructions - Stage 5 4

Chapter 1 - Adding the Observer Design Pattern to the BL Layer 4

1a. The Observer Design Pattern and its Integration into Our Project 5

1b. Adding the Ready-Made Class ObserverManager to the BL Layer 5

1c. Adding the Ready-Made Interface IObservable to the BL Layer 6

1d. Updating Every Logical Service Entity to be Observable 7

1e. Updating Every Static Helper Class (Managers) to Report Changes Through the Observer Mechanism 8

1f. Updating the Logical Management Entity IAdmin + AdminImplementation to be Observable 10

Chapter 2 - General Requirements for the Presentation Layer (PL) 11

Chapter 3 - Adding a WPF Project Type PL Project 14

Chapter 4 - Creating Windows for the PL Project 15

4a. Details of the Windows that Will Constitute the Project in This Stage 15

4b. Creating a Subdirectory for Windows of that Logical Entity 16

4c. Creating a Window for Displaying the List of Items 16

4d. Creating a Window for Displaying a Single Item 17

Chapter 5 - List Display Window - Design 17

5a. Structure of the List Display Window 17

5b. Designing the External Grid in the List Display Window 18

5c. Creating the Internal Top Grid in the List Display Window 19

5d. Creating Controls Within the Internal Top Grid 20

5e. Adding a Control for Displaying the List, in the Middle Part of the External Grid 21

5f. Adding a Button Control to Open a Window for Adding an Item to the List, in the Bottom Part of the External Grid 22

Chapter 6 - Main Management Window - Design and Implementation 23
6a. Main Management Window - Control Design 23
6b. Main Management Window - Creating a BL Layer Access Object 24
6c. Main Management Window - Defining the DataContext 24
6d. Main Management Window - Linking the System Clock Display to a Dependency Property in the Code-Behind 24
6e. Main Management Window - Creating Buttons for Advancing the System Clock 25
6f. Main Management Window - Configuration Variables Area 26
6g. Main Management Window - Defining Observers for the System Clock and Configuration Variables 27
6j. Main Management Window - Creating a Button to Open the List Display Window 29
Chapter 7 - List Display Window - Displaying List Details 30
7a. List Display Window - Creating a BL Layer Access Object 30
7b. List Display Window - Defining the DataContext for the Window and Defining a Dependency Property for the List 31
Chapter 8 - List Display Window - Filtering the List 32
8a. List Display Window - Populating the ItemsSource of the ComboBox with Values 33
8b. List Display Window - Filtering the Item List During the ComboBox SelectionChanged Event 34
8c. List Display Window - Defining Observers for the Item List in the Database 36
8d. List Display Window - Refining the List Display 37
Chapter 9 - Single Item Display Window 37
9a. Single Item Display Window - Control Design 38
9b. Single Item Display Window - Linking the Window to the Object 39
9c. Single Item Display Window - IsReadOnly Attribute and Visibility Attribute in the 2 Window Modes 41
9d. Single Item Display Window - Clicking the Add/Update Button to Implement Add/Update Operations 42
9e. Single Item Display Window - Defining Observers for the Single Item 43
Chapter 10 - Opening the Single Item Display Window from the List Display Window .. 44
10a. List Display Window - Linking the Selected Item in the List to a Property in the Code-Behind 44

10b. Opening the Single Item Display Window in Update Mode from the List Display Window 45
10c. Opening the Single Item Display Window in Add Mode from the List Display Window 45
10d. Refreshing the List After Adding/Updating an Item 45
Chapter 11 - List Display Window - Deleting a Single Item from the List 46
Chapter 12 - Customizing the Window Design 46
Chapter 13 - Recap 46
Chapter 14 - Creating a Tag and Submitting Stage 5 on Moodle 47
Appendices - Stage 5 48
Appendix 1 - Window Examples 48
Appendix 2 - Example of Defining and Using a Converter 50

Stage Goals

Creating a basic **Presentation Layer (PL)** (Graphical User Interface) for one Logical Service Entity.

In Stages 1-4, we built the **Data Access Layer** (DAL) and the **Business Logic Layer** (BL) of the project and wrote main programs to test each layer.

In this stage:

We will add the **Observer pattern** mechanism to the BL layer:

- To each Logical Service Entity, we will add the option to be "Observable," meaning to notify the Presentation Layer (PL) (the Observer) about changes and updates in configuration variables or lists in the database.

We will add a basic Presentation Layer for one Logical Service Entity and use its Service Interface.

- We will create a few windows (screens):
  - A main management window.
  - A window for displaying a list of items from that entity.
  - A window for viewing/adding/updating that entity.

    - We will operate these windows and the displayed/managed data by calling the Business Layer (BL).
    - We will add the ability for the windows to "observe" changes in the database.

Instructions for Execution and Submission of the Stage

- **Note!** The instructions throughout the document are accompanied by important explanations for understanding each step. This understanding will greatly help you during the oral defense of the project and the exam.
- Both partners must submit a link on Moodle according to the submission guidelines.
- **Note!** If there is an imbalance in the percentage of activity (push) on Git between the two partners, the grade will be different between the two partners. By recording the activity (push) in Git, we can verify that both partners contribute to the exercise.
- It is mandatory to read the general project description document before starting work on this stage.
- Instructions considered optional (elective) are marked in orange.
- It is mandatory to perform this stage in the same Git repository, cloud repository, and the same Solution as in Stage 0 (the preliminary stage).
- It is mandatory to continue adhering to the code writing style (indentations, naming conventions, etc.).
- It is mandatory to document all types, methods, fields, and properties using formatted documentation (///).

Outline - Stage 5

- Adding the **Observer pattern** mechanism to the BL layer.
- General requirements for the Presentation Layer (PL).
- Creating a new project and 3 new windows within it.
- Designing the windows - adding controls.
- Implementing the code-behind for the windows.

Specific Instructions - Stage 5

Chapter 1 - Adding the Observer Design Pattern to the BL Layer

Just before we start developing the Presentation Layer (PL), we will add some improvements to the BL layer so that it implements its part of the Observer Design

Pattern. This chapter mainly contains technical instructions, and you will receive ready-made classes from us, but you must thoroughly understand what is being done.

1a. The Observer Design Pattern and its Integration into Our Project

The Observer Design Pattern is a programming solution for a situation where an object (called the **Subject**; the "Observable") manages a list of objects (called **Observers**; the "Observers") that listen for changes in it, and automatically updates them when a change occurs in its state.

This pattern allows for **loose coupling** between the objects, so that the Observers can react to changes without the Subject being directly dependent on their implementation.

In our project:

- The Logical Service Entities (interfaces) in the BL layer will be the **Subject** - the "Observable" object.
- The changes that may occur in the BL layer are: adding/updating/deleting an object from the list in the database, as well as updating configuration variables or the system clock.
- The windows in the PL layer will be the **Observers** - the "Observers."
- The windows will want to receive a notification about changes occurring in the BL in order to automatically update the window's display.

We will implement the Observer Design Pattern using:

- In the BL layer - defining **events** in each Logical Service Entity.
- In the PL layer - each window will contain a method through which it wishes to know about changes that have occurred, and it will register this method to the appropriate event in the BL.

1b. Adding the Ready-Made Class ObserverManager to the BL Layer

For the correct management of the "Observable" and the "Observers" who will register to receive changes, we have prepared a helper class for you called `ObserverManager.cs`.

The class allows managing the observers for the Logical Service Entities in the BL layer and offers the following infrastructure:

- `event delegate` - for methods that want to observe entire lists in the database. They want to know if the list has changed (an object was added/deleted/updated) to update the entire list display on the list display window.

- `hash table for event delegate` - for methods that want to observe a specific object that has been updated to update the display of the single object on the single item display window.
- Methods that allow adding an observer method to each of the above delegates.
- Methods that allow removing an observer method from each of the above delegates.
- A method that will invoke all the methods registered to observe changes in the entire list.
- A method that will invoke all the methods registered to observe changes in a specific object.

The file can be found at the following link: ObserverManager.cs

1. Using File Explorer, copy the physical file `ObserverManager.cs` under the subdirectory named `BL\Helpers` and then add it to the project under `BL\Helpers`.

1c. Adding the Ready-Made Interface IObservable to the BL Layer

We will need to extend the Logical Service Entity interfaces (the Observables) to allow the display windows in the PL (the Observers) to register for changes in the logical layer.

For this purpose, we have prepared an interface called `IObservable` that allows:

- Adding a method that will observe updates in an entire list:

```
void AddObserver(Action listObserver);
```

- Adding a method that will observe updates in a single object:

```
void AddObserver(int id, Action observer);
```

- Removing a method that observed updates in an entire list:

```
void RemoveObserver(Action listObserver);
```

- Removing a method that observed updates in a single object:

```
void RemoveObserver(int id, Action observer);
```

The file can be found at the following link: IObservable.cs

1. Using File Explorer, copy the physical file `IObservable.cs` under the subdirectory named `BL\BlApi` and then add it to the project under `BL\BlApi`.

BS"D

1d. Updating Every Logical Service Entity to be Observable

1. For each of the Logical Service Entities in BL (interfaces), except for the Logical Management interface `IAdmin`:
2. Add a static internal field of type `ObserverManager` named `Observers` to its helper class (the Manager class) and initialize it.

For example:

```
namespace Helpers;
using DalApi;
using System;
internal static class StudentManager
{

    internal static ObserverManager Observers = new(); //stage 5
    //...
}
```

Extend the service interface with the IObservable interface.
For example:

```
namespace BlApi;
public interface IStudent : IObservable //stage 5 extending interface
{
    void Create(BO.Student boStudent);
    //...
}
```

In the implementation class of that logical interface, implement the 4 new methods added to the interface so that they call the ObserverManager of the corresponding Manager class to add/remove the received method to the observation mechanism.

For example:

```
using BlApi;
using Helpers;
namespace BlImplementation;
internal class StudentImplementation : IStudent
{
    // ...
#region Stage 5
    public void AddObserver(Action listObserver) =>
        StudentManager.Observers.AddListObserver(listObserver); //stage 5
    public void AddObserver(int id, Action observer) =>
        StudentManager.Observers.AddObserver(id, observer); //stage 5
    public void RemoveObserver(Action listObserver) =>
```

```
        StudentManager.Observers.RemoveListObserver(listObserver); //stage 5

    public void RemoveObserver(int id, Action observer) =>
        StudentManager.Observers.RemoveObserver(id, observer); //stage 5

#endregion Stage 5
}
```

1e. Updating Every Static Helper Class (Managers) to Report Changes Through the Observer Mechanism

Now that we have made every Logical Service Entity Observable, we will add change reports in it wherever needed, using the mechanism we added.

1. For each of the static helper classes in BL, all the Managers (except for the `AdminManager`, `ObserverManager` classes):
2. Inside the implementation of all helper methods, in every place where a change is made to the objects of this Logical Entity, add a call to the appropriate method that will trigger all the methods observing this change.

Examples from the `StudentManager` class:

- From the `Create`/`Delete` method, after a Student is created/deleted, we want to report an update of the entire list following the addition/deletion of that student. In practice, the report will go to the window that displays the list, which is registered to receive updates on the entire list.
- From the `Update` method, after a Student is updated, we want to report an update of the entire list following this, as well as a single update of that student. In practice, the report will go to the window that displays the list, which is registered to receive updates on the entire list. And also to the window that is currently displaying that single student whose details were updated from another place simultaneously.
- From the periodic student update method, every time we identify that a student has been updated, we want to report this, and consequently, report an update of the entire list. In practice, the report will go to the window that displays the list, which is registered to receive updates on the entire list. And also to the window that is currently displaying that single student whose details were updated from another place simultaneously.

```
using DalApi;
namespace Helpers;
internal static class StudentManager //stage 4
{
```

BS"D

```
    internal static ObserverManager Observers = new(); //stage 5
    private static readonly IDal s_dal = Factory.Get; //stage 4
    internal static void CreateStudent(BO.Student boStudent) //stage 4
    {
        // make all the logical tests here and throw exception if needed
        //...
        if (boStudent.Address is not null && boStudent.Address != "")
        {
            Tools.Location? location = Tools.GetLocationOfAddressSync(boStudent.Address);
            if (location is not null) {

                boStudent.Latitude = location.Latitude;
                boStudent.Longitude = location.Longitude;
            }
            else
                throw new BO.BLBadAddress($"The address of the student is invalid
({boStudent.Address})");
        }

        DO.Student doStudent = studentBoToDo(boStudent, update: false);
        try {
            s_dal.Student.Create(doStudent);
        } catch (DO.DalAlreadyExistsException ex) {
            throw new BO.BlAlreadyExistsException($"Student with ID={boStudent.Id} already
exists", ex);
        }
        Observers.NotifyListUpdated(); //stage 5
    }

    internal static void DeleteStudent(int id) //stage 4
    {
        try {
            s_dal.Student.Delete(id);
        }

        catch (DO.DalDoesNotExistException ex) {
            throw new BO.BlDoesNotExistException($"Student with ID={id} does Not exist",
ex);
        }

        Observers.NotifyItemUpdated(id); //stage 5
        Observers.NotifyListUpdated(); //stage 5
    }

    internal static void LinkStudentToCourse(int studentId, int courseId) //stage 4
    {
        LinkManager.LinkStudentToCourse(studentId, courseId);
        Observers.NotifyItemUpdated(studentId); //stage 5
    }

    internal static void PeriodicStudentsUpdates(DateTime oldClock, DateTime newClock)
//stage 4
```

```
    {
        //...
        bool studentUpdated = false; //stage 5
        List<DO.Student> doStudentList = s_dal.Student.ReadAll().ToList();
        foreach (var doStudent in doStudentList)
        {
            if (AdminManager.Now.Year - doStudent.RegistrationDate?.Year >=
s_dal.Config.MaxRange)
            {
                //...
                studentUpdated = true;
                s_dal.Student.Update(doStudent with { IsActive = false });
                Observers.NotifyItemUpdated(doStudent.Id); //stage 5
                //...
            }
        }

        bool yearChanged = oldClock.Year != newClock.Year; //stage 5
        if (yearChanged || studentUpdated)
                Observers.NotifyListUpdated(); //stage 5
    }
}
```

1f. Updating the Logical Management Entity IAdmin + AdminImplementation to be Observable

The `AdminManager` class we provided in Stage 4 manages, among other things, the observation of the system clock and configuration variables, as well as their update.

The class contains an `event` named `ConfigUpdatedObservers` for anyone who wants to observe the update of configuration variables and also an `event` named `ClockUpdatedObservers` for observing the system clock.

The class also contains a method for updating configuration variables named `SetConfig` which you updated in Stage 4, and now you can explain and see that at its end it contains 2 lines that report to the observers about updates to the configuration entity.internal static void SetConfig(BO.Config configuration) //stage 4

```
{
    bool configChanged = false; // stage 5
    if (s_dal.Config.MaxRange != configuration.MaxRange) //stage 4
    {
        s_dal.Config.MaxRange = configuration.MaxRange;
        configChanged = true;
    }

    //TO_DO: //stage 4
```

```
    //add a condition+assignment for each configuration property
    //...
    //Calling all the observers of configuration update

    if (configChanged) // stage 5
        ConfigUpdatedObservers?.Invoke(); // stage 5
}
```

1. Add four methods to the Logical Management interface `IAdmin` that will allow adding/removing observer methods for the system clock and configuration variables.

```
#region Stage 5
void AddConfigObserver(Action configObserver);
void RemoveConfigObserver(Action configObserver);
void AddClockObserver(Action clockObserver);
void RemoveClockObserver(Action clockObserver);
#endregion Stage 5
```

2. In the implementation class `AdminImplementation`, implement the 4 new methods added to the `IAdmin` interface, which will call `AdminManager` to add/remove the received method to the observation mechanism for the clock and configuration variables.

```
#region Stage 5
public void AddClockObserver(Action clockObserver) =>
    AdminManager.ClockUpdatedObservers += clockObserver;

public void RemoveClockObserver(Action clockObserver) =>
    AdminManager.ClockUpdatedObservers -= clockObserver;

public void AddConfigObserver(Action configObserver) =>
    AdminManager.ConfigUpdatedObservers += configObserver;

public void RemoveConfigObserver(Action configObserver) =>
    AdminManager.ConfigUpdatedObservers -= configObserver;
#endregion Stage 5
```

Chapter 2 - General Requirements for the Presentation Layer (PL)

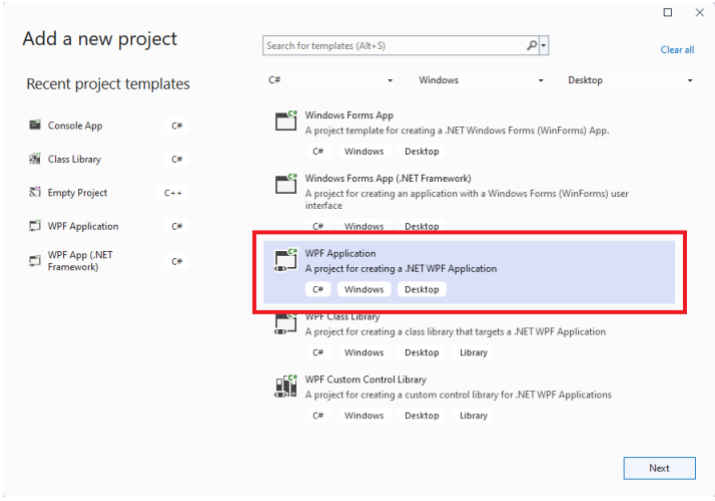
1. In the Presentation Layer, it is mandatory to catch all exceptions and display an orderly message (MessageBox) to the user about the problem.
2. The message about the problem must be clear and understandable to the system user (a regular person and not just a programmer) - meaning the goal is not to display the exception message and variable names, etc., but to display an

informative message at the system user level with a meaningful title, understandable content, and an appropriate icon.

3. In the entity addition windows - after successful completion of the addition - the addition window should close automatically, and the list window must be updated automatically to include the added entity.
4. In the entity update windows - after successful completion of the update - the update window should close automatically, and the list window must be updated automatically to include the updated entity.
5. Automatic update of the display on the list window following an addition/update will be performed using the **Observer Design Pattern** in the logical layer, and the registration for observers will be done from the PL layer.
6. There will be no direct sharing of information between the windows - information transfer will be done by passing arguments for the parameters of the window constructor.
7. In order to allow maintaining the MVVM (Model-View-ViewModel) principle, which separates logic from design - it is forbidden to give names to controls through XAML to use them in the code-behind. Instead, the **Data Binding** mechanism, which allows the separation between logic (code-behind) and design (XAML), should be used.
8. The code-behind should be minimized to the necessary minimum for handling user input and other WPF events:
    - It is mandatory to use **Data Binding** fully:
        - For data displayed on the screen, such as: the content of a single entity or a list of entities, it is mandatory to define corresponding properties in the code-behind of each window. The corresponding properties in the code-behind are intended to hold the displayed and/or updated data. The properties in the code-behind must be defined so that updating these properties through the code-behind will cause them to be automatically updated in the display. The properties will be defined in the code-behind using one of two options:
            - **Option 1:** An event of the `INotifyPropertyChanged` interface will be defined in the class, and the event will be activated accordingly in the `set` of these properties.
            - **Option 2:** The properties will be defined as **Dependency Properties**.
        - The binding of the controls' attributes to the data will be performed only through the XAML in each of the windows to synchronize the display with the data (and not through the code-behind).
        - It is forbidden to change the `DataContext` or `ItemSource` through the code-behind - because these are also control attributes, but they must be bound from XAML to the appropriate data.
        - Thanks to the Binding done through XAML, changing the data in the code-behind will automatically affect the attributes of the controls in the display.

    - In cases where it is necessary to bind between the attributes of two controls - it is allowed to name the control with the target attribute.
    - If you want to add a description to a control (inside XAML) - so that it is clear what the control is for - it is advisable to use comments in the format: `<!-- ... comment ... -->`
    - A convenient way to add a comment: write the comment text where you want it, stand on the comment line, and then click the comment button, in the top menu.
- It is disallowed to access the properties of the `sender` that comes as a parameter in event handling methods (even after casting to the appropriate type) - but always rely on DataBinding.
- In this layer, no project logic will be executed, except for basic input validation (format, legal characters, etc.).
- It is mandatory to use `IConverter` in combination with Data Binding for conversions or to dynamically block input in controls or dynamically hide controls.
- In every operation performed from the GUI, a maximum of one request should be sent to the BL, and remember to catch the possible exceptions resulting from this operation.
- In this stage, you can already use (but it is not yet mandatory, you can postpone it to Stage 6) a variety of advanced WPF components beyond the basic controls, for example:
    - Layouts of various kinds (Grid, StackPanel, etc.)
    - Using Resources
    - Using Style
    - Data Templates, in general, and element templates in a multiple data control, in particular.

Chapter 3 - Adding a WPF Project Type PL Project



1. Add a new project named **PL** of type **WPF Application** to the Solution.
2. After the new project opens under the solution, double-click the project name in the Solution Explorer.
3. The file for managing the project properties will open for editing. Add the following lines (highlighted in yellow in the original) to the file at the end of the `PropertyGroup` element:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>WinExe</OutputType>
    <TargetFramework>net8.0-windows</TargetFramework>
    <Nullable>enable</Nullable>
    <UseWPF>true</UseWPF>
    <BaseOutputPath>$(SolutionDir)\bin\</BaseOutputPath>
    <OutputPath>$(SolutionDir)\bin\</OutputPath>
    <AppendTargetFrameworkToOutputPath>false</AppendTargetFrameworkToOutputPath>
    <AppendRuntimeIdentifierToOutputPath>false</AppendRuntimeIdentifierToOutputPath>
    <UseCommonOutputDirectory>true</UseCommonOutputDirectory>
  </PropertyGroup>
</Project>
```

4. Save the file.
5. Add references from the PL project towards the BL project.
6. Verify in the project properties file that its text has been updated after adding the reference:

```
<ItemGroup>
  <ProjectReference Include="..\BL\BL.csproj"/>
```

```
</ItemGroup>
```

7. Mark the PL project as the startup project of the Solution, the project that will run when you click the green PLAY button:
8. Right-click on the PL project.
9. Then click on **Set as Startup Project**.

Chapter 4 - Creating Windows for the PL Project

In this stage, we will create basic display windows for one selected Logical Service Entity, through which we will present the relevant Logical Data Entities for it. The general project description document will specify the Logical Entity you will work on in this stage, as well as the required windows for it in Stage 5.

4a. Details of the Windows that Will Constitute the Project in This Stage

In Stage 5, we will not create the system login window, but will allow direct entry to the main management window. In Stage 6, you will add the system login window, which will lead to the main management window.

The 3 windows we will create in this stage:

1. **Temporary Main Management Window** - through which it will be possible to reach the other windows. To all the windows we will create in the future, and specifically to window number 2.
2. **Item List Display Window** - a window for displaying a list of all items of the Logical Data Entity type. The items displayed in the list will contain limited information about another comprehensive Logical Data Entity. Clicking on an item in the list will display the more extended information in window number 3.
3. **Single Item View/Add/Update Window** - a window for managing a single item of the Logical Data Entity type.

For example:

- Assume that the comprehensive Logical Data Entity `BO.Course` is the one selected in the general project description document.
- Then, in window number 2, a list of items of the Logical Data Entity type `BO.CourseInList` will appear. This is an entity containing limited information about a course.
- Double-clicking on an item in the list opens window number 3, which displays full and detailed information about that item of the `BO.Course` entity type. And in this window, it is possible to view the full information and/or update it.

- In addition, if you want to add a new course, the same window number 3 will open, but with empty fields. And after they are filled with values, that course will be added to the list in window number 2.

You can see that with the creation of the project, window number 1, the main management window, named `MainWindow`, has already been automatically created for us. We will use it and update it later in the document.

Now, we will create 2 more windows, window number 2 - for displaying the list of items, and window number 3 - for viewing/updating/adding a single item to the list.

In this stage, we will offer you a design proposal and guide you step by step. Afterwards, each pair will develop and refine the design as they wish.

It is recommended, before continuing, to see an example of the 3 windows at the end of this document: Appendix 1 - Window Examples

4b. Creating a Subdirectory for Windows of that Logical Entity

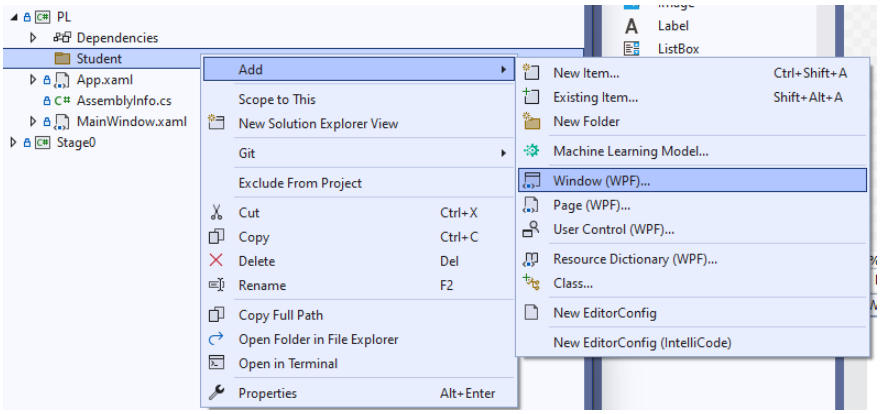
We want to create a subdirectory for all the windows related to that Logical Service Entity (this is how you will also do it in the future, when you add windows for additional Logical Service Entities). In the context menu of the PL project (right-click on the PL project):

1. Click on **Add → New Folder** and name the directory with the name of the Logical Service Entity.
2. For example: `Course`

4c. Creating a Window for Displaying the List of Items

To add a new window for displaying the list of items of the Logical Service Entity type:

1. In the context menu of the subdirectory (right-click on it), click on: **Add → Window(WPF)**



2. Name the window appropriately.
3. For example: `CourseListWindow`
4. Let's look, for example, at the XAML code created for the list display window:

```
<Window x:Class="PL.Course.CourseListWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  xmlns:local="clr-namespace:PL.Course"
  mc:Ignorable="d"
  Title="CourseListWindow" Height="450" Width="800">
  <Grid>
  </Grid>
</Window>
```

5. You can see that the window was created with a `Grid` layout component inside it, which is currently empty.

4d. Creating a Window for Displaying a Single Item

To add a new window for displaying a single item of the selected comprehensive Logical Data Entity type:

1. Perform the same steps 1-4 from the previous section to create one more new window, with an appropriate name, for displaying a single item.
2. For example: `CourseWindow`

Chapter 5 - List Display Window - Design

In this chapter, we will only deal with the design of controls in the list display window, without implementing their functionality.

5a. Structure of the List Display Window

As mentioned, it is forbidden to name controls, so an important recommendation: add a comment in XAML above significant controls so that the XAML code is clear and we know the purpose of the control. After we finish the design, in the following chapters we will explain how to implement their functionality using full use of the **Data Binding** mechanism.

A design proposal for the list display window - it will contain a `Grid` control (the external Grid), which is composed of 3 rows:

- **Top row**, a `Grid` control (the internal Grid), which will be composed of 2 columns:

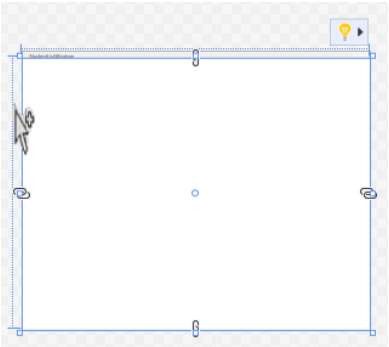
    - Left column, a `Label` control - which will contain the name of the category by which we want to filter the list of items displayed on the screen.
    - Right column, a `ComboBox` control - which will contain the options for filtering the list, and through which a value can be selected to filter the list.
- **Middle row**, a list of type `ListView` or `DataGrid` - which will display the list of items in an advanced format - clicking on an item in the list will allow viewing/updating an item in the list.
- **Bottom row**, a `Button` control - clicking on it will open a new window through which we can add an item to the list.

5b. Designing the External Grid in the List Display Window

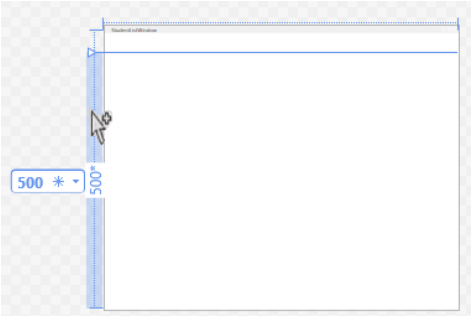
1. Through the XAML, change the height of the window inside the `<Window>` tag to `Height="640"`.
2. This will be the external Grid, and it will later contain an internal Grid. It is recommended to add a comment above it to mark that it is the external one.
3. Divide the Grid into 3 rows with a ratio of `40*:500*:Auto`. You can do this in one of the following 2 ways:
    - **option 1:** Write the changes manually to the XAML inside the `<Grid>` tag:

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="40*"/>
        <RowDefinition Height="500*"/>
        <RowDefinition Height="Auto"/>
    </Grid.RowDefinitions>
</Grid>
```

    - **option 2:** On the graphical screen, move the mouse to the borders of the grid, the mouse cursor will change to an arrow with a small *plus* sign .



If you left-click, this line will remain and will constitute a division of the grid.

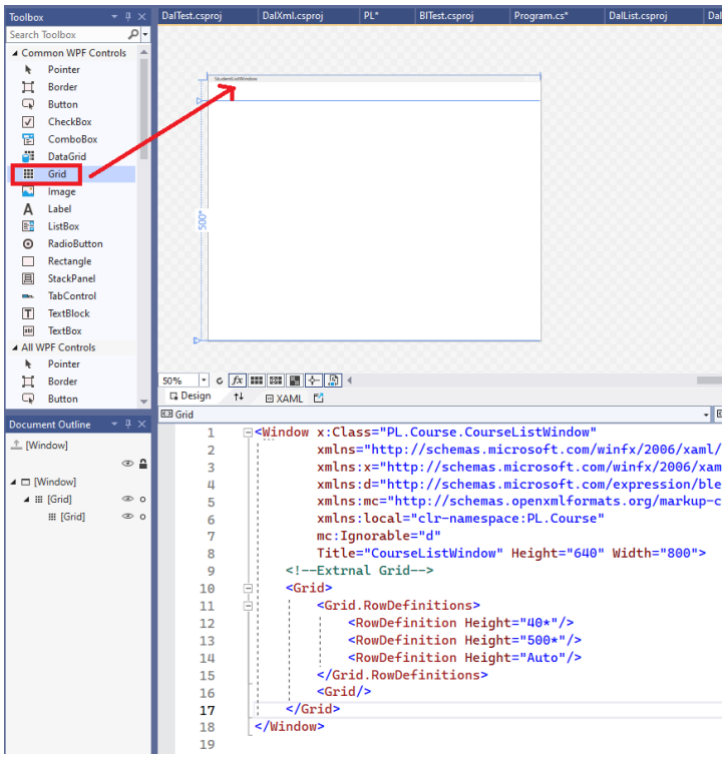


4. Review the code obtained in the XAML pane: You received 3 rows of `RowDefinition`. What does the value given to the `Height` attribute symbolize?
5. **Numbers only** - the values will remain constant when resizing the window.
6. **Numbers with an asterisk** – aspect ratio will be preserved when resizing the window.
7. **Asterisk only** - aspect will be preserved when resizing the window.
8. The meaning of the **Auto** value in the attributes of the row height or column width of a Grid - the value will be determined automatically according to the sizes of the controls that the row or column will contain within its cells (if there are no controls yet, the row or column will not be displayed, but it still exists).

5c. Creating the Internal Top Grid in the List Display Window

1. Create an internal Grid that will appear inside the top row of the main external Grid, as follows: Drag the `Grid` graphic component from the Toolbox pane with the mouse. Drag it to the top row of the main external Grid. This Grid will be used to contain the upper part of the list display window.

BS"D



2. As a result of the drag, a new empty XML tag of type `Grid` was created, which looks like this: `</Grid>`
3. Replace this entire tag with the following definition:

```
<!--Internal, Upper Grid-->
<Grid HorizontalAlignment="Stretch" Height="auto" Grid.Row="0"
VerticalAlignment="Stretch" Width="auto">
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*"/>
        <ColumnDefinition Width="*"/>
    </Grid.ColumnDefinitions>
</Grid>
```

Explanation: The new internal grid is located in row 0 of the external grid in which it is contained (counting starts at 0), it is stretched width-wise and height-wise and receives its height and width automatically. It is divided into 2 columns of equal width (*).

5d. Creating Controls Within the Internal Top Grid

We want to allow filtering in the list display. The filtering will be performed according to the property defined in the general description document. For this purpose, we will use a `ComboBox` control accompanied by a `Label` control next to it.

- `ComboBox` is a graphic component that will allow the user to select from a drop-down list.
- For example: a list of semesters by which we want to filter the list of courses displayed on the screen.
- And `Label` is a graphic component that serves as a text label, usually accompanying an additional control to describe it.
- For example: a label that says: "Select Semester:"

1. Drag a `Label` control from the Toolbox pane with the mouse into the first column (the left one, column number 0) of the internal top Grid. Alternatively, you can add a `Label` control by editing the XAML code. As a result of the drag/edit, a new XML tag of type `<Label>` will be created inside the internal `<Grid>` tag.
2. Drag a `ComboBox` control from the Toolbox pane with the mouse into the first column (the left one, column number 0) of the internal top Grid. Alternatively, you can add a `ComboBox` control by editing the XAML code. As a result of the drag/edit, a new XML tag of type `<ComboBox>` will be created inside the internal `<Grid>` tag.
3. Through the XAML, verify that the `Label` control is under the internal top Grid and that its column attribute is 0 (meaning it is in the left column of the internal top Grid).

```
Grid.Column="0"
```

4. Through the XAML, verify that the `ComboBox` control is under the internal top Grid and that its column attribute is 1 (meaning it is in the right column of the internal top Grid).

```
Grid.Column="1"
```

5. For the `ComboBox` control:
6. Change these attributes as follows:

```
HorizontalAlignment="Stretch" VerticalAlignment="Stretch"
```

7. Remove the other attributes that appear in the XAML for the `ComboBox` control.

5e. Adding a Control for Displaying the List, in the Middle Part of the External Grid

We want to add, in the middle part of the external Grid, a control that will contain the list of items of the Logical Helper Entity type. And it will display it graphically, as a formatted table with columns and rows in an advanced template. For this purpose, there are various controls for displaying a tabular list in an advanced format, such as: `ListView`,

`DataGrid`, etc. These controls are based on the simpler control for displaying a list (`ListBox`), but allow adding more sophisticated design to the items.

1. Drag the advanced list control you chose to work with from the Toolbox pane with the mouse. Drag it into the external Grid, below the internal top Grid.
2. Through the XAML, note that you must define the row in which the control will appear: `Grid.Row="1"`. This row refers to the row inside the external Grid. That is, the list control will be placed in the second row of the external Grid (the internal top Grid control is already placed in the first row (number 0)).
3. If you chose to work with `ListView`, then through the XAML, remove the attribute that was automatically added to `ListView` (if it was added):

```
d:ItemsSource="{d:SampleData ItemCount=5}"
```

4. Whether you chose to work with a `DataGrid` or `ListView` control - you must include within it a definition of a **Template** for some of the columns (each row represents an entity object and each column represents an entity property). For example, a Boolean property should be displayed with a `CheckBox` control, a different text color for the content of a specific column that you wish to emphasize, and so on.
5. You may prefer to deal with the Template later after you have already connected the control to the logical list and handled the option of list filtering. Here: 8d. List Display Window - Refining the List Display.

5f. Adding a Button Control to Open a Window for Adding an Item to the List, in the Bottom Part of the External Grid

We want to add, in the bottom part of the external Grid, a `Button` control such that clicking on it will open a new window through which we can add an item to the list.

1. The button must be created in the third row `Grid.Row="2"` of the external Grid. Since the third row of the external Grid was defined with an `Auto` height, it is difficult to drag a button into it through the Toolbox, so add the following line directly in the XAML of the list display window, below the list control:

```
<Button Content="Add" Grid.Row="2" HorizontalAlignment="Center" Height="50"
Width="100"/>
```

BS"D

Chapter 6 - Main Management Window - Design and Implementation

As mentioned, with the creation of the project, window number 1, the main window named `MainWindow`, was automatically created for us. We will use this window as the main management window and temporarily design it. In the next stage of the project, we will refine it.

Note: In this stage (Stage 5), the main management window will serve as a temporary entry point to the system for a user in the "manager" role. But in the next stage (Stage 6), you will add a system login window that includes an ID and enters a user in the "manager" role into the main management window and a user in the "volunteer" role into the main volunteer window.

6a. Main Management Window - Control Design

1. Define rows and columns as you wish inside the Grid to position all the required display components on this window. Or even a few internal Grids inside the main Grid. At your discretion.
2. As mentioned, it is forbidden to name controls, so an important recommendation: add a comment in XAML above significant controls so that the XAML code is clear and we know the purpose of the control.
3. Position the following components inside the Grid you defined:
   - **"System Clock" area:**
     - `Label` - which will display the current system clock value.
     - 5 buttons (`Buttons`) for advancing the system clock: advance by a minute, advance by an hour, advance by a day, and advance by a year.
   - **"Configuration Variables" area:**
     - `TextBox` - a separate text box for each configuration variable, which will display its value, and also allow editing/changing it.
     - Only those that are determined in the display, such as: maximum number of years for a student to study, and not those that are determined internally, such as: running number.
     - `Button` - one button that clicking on it will update the configuration object (which contains the values of all configuration variables) in the database.
     - `Button` - a button that clicking on it will lead to opening the list display window.
     - `Button` - a button that clicking on it will lead to initializing (`Initialize`) the database (creating initial data).
     - `Button` - a button that clicking on it will lead to resetting (`ResetDB`) the database (clearing the database).

6b. Main Management Window - Creating a BL Layer Access Object

1. From the main management window, we want to access the "Management" Logical Service Entities. For this purpose, we will need access to the BL. Therefore, in the code-behind of the window (`class MainWindow`), add a static private field named `s_bl` of type `IBL` that will be initialized with an object of the `BL` class that implements the `IBL` interface. Exactly as in `BlTest`.

```
static readonly BlApi.IBl s_bl = BlApi.Factory.Get();
```

6c. Main Management Window - Defining the DataContext

As mentioned, we want to avoid naming controls in all windows (especially in the main management window). If so, how do we access the attributes of the controls through the code-behind?

Through the XAML, we will use the **Data Binding** mechanism to link the attributes of the controls to **Dependency Properties** that we will define in the code-behind of the window class itself.

1. Through the XAML file of the window, define the DataContext of the entire window to be linked to its own properties. Add an attribute named `DataContext` with a relative value that links it to itself, to the `Window` element.

```
<Window x:Class="PL.Course.CourseListWindow"
    ...
        DataContext="{Binding RelativeSource={RelativeSource Mode=Self}}">
```

2. From now on, inside the XAML of the management window, we will use the **Data Binding** mechanism to link control attributes to a corresponding Dependency Property that we will define in the code-behind. We will demonstrate this immediately.

6d. Main Management Window - Linking the System Clock Display to a Dependency Property in the Code-Behind

1. In the code-behind of the main management window, define a **Dependency Property**, with a name you choose, of type `DateTime` that will represent the value of the date displayed on the screen. It is recommended to use the snippet named `propdp`.

For example:

BS"D

```
public DateTime CurrentTime
{
    get { return (DateTime)GetValue(CurrentTimeProperty); }
    set { SetValue(CurrentTimeProperty, value); }
}

public static readonly DependencyProperty CurrentTimeProperty =
    DependencyProperty.Register("CurrentTime", typeof(DateTime), typeof(MainWindow));
```

. .

2. Through the XAML, access the `Label` control you defined for displaying the current value of the system clock. Set its `Content` attribute using the Binding mechanism, and link it to the `DateTime` Dependency Property that you just defined in the code-behind.

   For example:
   `<Label Content="{Binding CurrentTime}" FontWeight="Bold" FontSize="20"/>`

3. Now, changes in the code-behind to the system clock property (`CurrentTime`) will affect the display of the date in the Label on the screen.

6e. Main Management Window - Creating Buttons for Advancing the System Clock

1. Through the XAML, access the 5 `Button` controls you defined for advancing the system clock: advance by a minute, advance by an hour, advance by a day, and advance by a year.
2. For each button:
3. Edit its `Content` attribute according to its purpose.
4. In the code-behind, add a method that will register to the button's click event (`Click`) for the purpose of displaying the list display window.
5. Verify that the method is properly registered for the click event (you can also see this through the XAML of each `Button` element).
6. Give the event method a meaningful name (changing the name of an event requires changing the name also at the event registration point. Tip: the simple way to do this is to change the name of the method using **CTRL+R+R**, and then the method name is automatically updated in all places where it is used).
7. Implement the event method so that a call is made to the clock advancement method of the management interface with the appropriate time unit for advancement.

For example in XAML:

```
<Button Content="Add One Minute" Click="btnAddOneMinute_Click"/>
```

And in the code-behind:

```
private void btnAddOneMinute_Click(object sender, RoutedEventArgs e)
{
    s_bl.Admin.ForwardClock(BO.TimeUnit.MINUTE);
}
```

6f. Main Management Window - Configuration Variables Area

1. In the code-behind of the main management window, define a **Dependency Property**, named `Configuration`, that will represent the logical configuration object `BO.Config`.

For each configuration variable that is determined in the display and for which you prepared a `TextBox`, perform the following actions:

1. Link the display of the configuration variable's value (`TextBox`) to a Dependency Property in the code-behind:
2. Through the XAML, access the `TextBox` control you defined for displaying the content of that configuration variable. Set its `Text` attribute using the Binding mechanism, and link it to the Dependency Property of the type you just defined in the code-behind. For example:

```
<TextBox Text="{Binding Configuration.MaxRange, UpdateSourceTrigger=PropertyChanged}"
Height="NaN" Width="50" HorizontalAlignment="Left" Canvas.Left="107"
VerticalAlignment="Top"/>
```

3. Now, changes in the code-behind to the value of the configuration variable will affect its display on the screen, and vice versa.

- Implement the button click operation (`Button`) so that it updates the value of the configuration object in the DAL:
    - In the code-behind, add a method that will register to the button's click event (`Click`) for the purpose of updating the configuration object.
    - Verify that the method is properly registered for the click event and give the event method a meaningful name.
    - Implement the event method so that a call is made to the configuration object update method:

```
s_bl.Admin.SetConfig(Configuration);
```

BS"D

6g. Main Management Window - Defining Observers for the System Clock and Configuration Variables

1. Define 2 private methods for the `MainWindow` class that will "observe" changes made to the clock value and the rest of the configuration variables in the database.
2. `clockObserver` - "Clock Observation Method" - which will be invoked by the `BL.ObserverManager` class every time the system clock value is updated.
3. In the body of the method, reassign the current value of the system clock (by calling the `Admin` interface of the BL) into the `DateTime` Dependency Property in the code-behind.

```
CurrentTime = s_bl.Admin.GetClock();
```

4. `configObserver` - "Configuration Variables Observation Method" - which will be invoked by the `BL.ObserverManager` class every time the value of one of the configuration variables is updated.
5. In the body of the method, reassign the value of the configuration object (by calling the `Admin` interface of the BL) into the corresponding Dependency Property.

```
Configuration = s_bl.Admin.GetConfig();
```

6h. Main Management Window - Window Loaded Event

1. In the code-behind of the main management window, add a method that will register to the event of loading the entire window `Loaded`.

2. Verify that the method is properly registered for the loading event and give the method a meaningful name.

3. In the code-behind, implement the event method so that the following operations are performed when the window loads: (Note to perform all of them through the `Admin` interface of the BL)

4. Assignment of the current value of the system clock (by calling the `` `Admin` `` interface of the BL) into the `DateTime` Dependency Property in the code-behind.

```
CurrentTime = s_bl.Admin.GetClock();
```

5. Assignment of the current value of the configuration object (by calling the `Admin` interface of the BL) into the corresponding Dependency Property in the code-behind.

```
Configuration = s_bl.Admin.GetConfig();
```

6. Through the interface method `Admin.AddClockObserver`, add the "**Clock Observation Method**" that you defined in the previous section as an observer.

```
s_bl.Admin.AddClockObserver(clockObserver);
```

7. Through the interface method `Admin.AddConfigObserver`, add the "**Configuration Variables Observation Method**" that you defined in the previous section as an observer.

```
s_bl.Admin.AddConfigObserver(configObserver);
```

6i. Main Management Window - Window Closing Event

1. In the code-behind of the main management window, add a method that will register to the event of closing the entire window (**Window_Closed**).

2. Verify that the method is properly registered for the closing event and give the method a meaningful name (you can also see this through the XAML in the `Window` element).

3. In the code-behind, implement the event method so that the following operations are performed when the window closes: (Note to perform all of them by calling the `Admin` interface of the BL)

4. Through the interface method `Admin.RemoveClockObserver`, remove the "**Clock Observation Method**" that you defined in the previous section so that it no longer observes.

```
s_bl.Admin.RemoveClockObserver(clockObserver);
```

8. Through the interface method `Admin.RemoveConfigObserver`, remove the "**Configuration Observation Method**" that you defined in the previous section so that it no longer observes.

```
s_bl.Admin.RemoveConfigObserver(configObserver);
```

6. Run the project and check the results.

6j. Main Management Window - Creating a Button to Open the List Display Window

1. Through the XAML, access the `Button` control you defined for opening the list display window. Edit its `Content` attribute according to its purpose, for example:

```
Content="Handle Courses"
```

2. As mentioned, it is forbidden to name controls, so an important recommendation: add a comment in XAML above the control so that the XAML code is clear and we know the purpose of the control.
3. In the code-behind, add a method that will register to the button's click event (`Click`) for the purpose of displaying the list display window.
4. Verify that the method is properly registered for the click event and give the event method a meaningful name.
5. Implement the event method so that a new window of the list display window type is opened. After opening the list display window, it must be possible to continue accessing the main window (using the `Show` method and not the `ShowDialog` method).

For example:

```
private void btnCourses_Click(object sender, RoutedEventArgs e) {
   new CourseListWindow().Show();
}
```

6. Run the project and check the results.

6k. Main Management Window - Creating Buttons for Initializing and Resetting the Database

In the main management window, you created one button that clicking on it will lead to database initialization and a second button that clicking on it will lead to database reset.

For each of the buttons you created:

1. Through the XAML, access the appropriate `Button` control for it.

2. In the code-behind, add a method that will register to the button's click event (`Click`) for initialization or reset, respectively.

3. Verify that the method is properly registered for the click event and give the event method a meaningful name.

4. The implementation of the click event method will include:

5. A message to the user `MessageBox` to confirm that they indeed want to perform the initialization/reset, and only if they answer "Yes" should the method execution continue. Otherwise, nothing should be done.

6. Closing all open windows, except for the main window, which will remain open.

7. Calling the appropriate method in the BL's management interface:

```
s_bl.Admin.InitializeDB();

s_bl.Admin.ResetDB();
```

8. During the time it takes to initialize/reset the database, the mouse icon should change to an hourglass.
9. Run the project and check the results.

Chapter 7 - List Display Window - Displaying List Details

Now we will return to handling the functionality of the list display window that we already designed in Chapter 5 - List Display Window - Design.

The list display window - must display a list of all items of the Logical Data Entity type. The items displayed in the list will contain limited information about another comprehensive Logical Data Entity. Clicking on an item in the list will display the more extended information in the single item display window.

The list will be populated by running a query in the BL. That is, we want to assign the list to the `ItemsSource` attribute of the `ListView` control or the `DataGrid` control (depending on which advanced list display control you chose in 5e. Adding a Control for Displaying the List, in the Middle Part of the External Grid).

7a. List Display Window - Creating a BL Layer Access Object

1. In the code-behind of the window, we will need access to the BL in order to request the list of entities from it. Therefore, in the code-behind, add a static

private field named `s_bl` of type `IBL` that will be initialized with an object of the `BL` class that implements the `IBL` interface. Exactly as in `BlTest`.

```
static readonly BlApi.IBl s_bl = BlApi.Factory.Get();
```

7b. List Display Window - Defining the DataContext for the Window and Defining a Dependency Property for the List

As mentioned, we want to avoid naming controls (especially the advanced list display controls `ListView`/`DataGrid`), so we will not be able to access the `ItemsSource` attribute of the display control through the code-behind. In addition, we want that every time the collection changes, the list is also updated in the display. Therefore:

1. Through the XAML file of the window, define the DataContext of the entire window to be linked to its own properties. Add an attribute named `DataContext` with a relative value that links it to itself, to the `Window` element.

```
<Window x:Class="PL.Course.CourseListWindow"
...
DataContext="{Binding RelativeSource={RelativeSource Mode=Self}}">
```

2. In the code-behind of the window, define a **Dependency Property**, with a name you choose, of type `IEnumerable` of the constrained logical entity. So that we can load the logical list of items into it and link it to the `ItemsSource` of the `ListView`/`DataGrid`. It is recommended to use the snippet named `propdp`.

For example:

```
public IEnumerable<BO.CourseInList> CourseList {
    get { return (IEnumerable<BO.CourseInList>)GetValue(CourseListProperty); }
    set { SetValue(CourseListProperty, value); }
}

public static readonly DependencyProperty CourseListProperty =
    DependencyProperty.Register("CourseList", typeof(IEnumerable<BO.CourseInList>),
typeof(CourseListWindow), new PropertyMetadata(null));
```

3. Through the XAML file of the window, add an `ItemsSource` attribute to the `ListView`/`DataGrid` element and link it through the Data Binding mechanism to the Dependency Property as you just defined in the code-behind.

```
<ListView Margin="5" Grid.Row="1" ItemsSource="{Binding Path=CourseList}">
```
or:
```
<DataGrid Margin="5" Grid.Row="1" ItemsSource="{Binding Path=CourseList}">
```

Since the `Path` attribute is the default for the Binding mechanism, this line is equivalent to the following line:<ListView Margin="5" Grid.Row="1" ItemsSource="{Binding CourseList}">

4. If you run the program, you will discover that the list is still empty in the display, because we have not yet run the query in the code-behind that loads the list into the `CourseList` Dependency Property. In the meantime, continue to the next chapter.

Chapter 8 - List Display Window - Filtering the List

We will continue to handle the functionality of the list display window, and now we want to use the values of the list items that appear in the `ComboBox` for the purpose of filtering the list displayed on the screen.

First, we will populate the `ComboBox` with possible values according to the required filtering criterion, which is defined in the general project description document. And then, we will respond to the event of selecting the value in the `ComboBox` and display in the item list on the screen only the items belonging to the selected category in the `ComboBox`.

The values with which we want to populate the `ComboBox` are defined as some `enum` in the BL.

For example, if this is the `enum` defined in BO:

```
public enum SemesterNames
{
    WinterA,
    SpringB,
    Year,
    Summer,
    Elul,
    None
}
```

Then we want to display the list of possible semesters in the `ComboBox`, and every time a different value is selected in the `ComboBox`, we will filter the list of courses displayed according to the selected semester.

BS"D

8a. List Display Window - Populating the ItemsSource of the ComboBox with Values

Since we want to avoid naming controls (especially the `ComboBox`), we will not be able to access the `ItemsSource` attribute of the `ComboBox` through the code-behind.

Therefore, through the XAML, we will use the `Source` attribute of the **Data Binding** mechanism to link the `ItemsSource` attribute of the `ComboBox` control to a static resource that will be linked to a local logical collection of the `Enums` values. We will define the static resource in XAML and the logical collection as a new type in the code-behind.

1. In the PL project, add a new code file named `Enums.cs`.
2. Define a new type inside it of type collection that returns the entire list of items in the `Enum` we defined (define a new class of type `IEnumerable` that returns `IEnumerator`).

For example, for an enum of type `BO.SemesterNames`, we will define the logical collection in PL inside the `Enum.cs` file as follows:namespace PL;

```
internal class SemestersCollection : IEnumerable {
    static readonly IEnumerable<BO.SemesterNames> s_enums =
        (Enum.GetValues(typeof(BO.SemesterNames)) as IEnumerable<BO.SemesterNames>)!;
    public IEnumerator GetEnumerator() => s_enums.GetEnumerator();
}
```

3. In the project's resource file `App.xaml` (it is also possible in the XAML file of the window itself), define a static resource with an `x:Key` key with a name you choose, and link it to the local logical collection object that you defined in the previous section.

For example:

```
<Application x:Class="PL.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="clr-namespace:PL"
    StartupUri="MainWindow.xaml">
  <Application.Resources>
    <local:SemestersCollection x:Key="SemestersCollectionKey" />
  </Application.Resources>
</Application>
```

4. Through the XAML file of the window, add an attribute named `ItemsSource` to the `ComboBox` control that defines its data source. Link it through the `Source` attribute

of the Data Binding mechanism to the static resource with the key you defined for it.

For example:

```
<ComboBox ItemsSource="{Binding Source={StaticResource SemestersCollectionKey}}"
Grid.Column="1" HorizontalAlignment="Stretch" VerticalAlignment="Stretch"/>
```

5. Run the project and check the results. Verify that the `ComboBox` is populated with the ENUM values.

8b. List Display Window - Filtering the Item List During the ComboBox SelectionChanged Event

To filter the list displayed on the screen according to the selected value in the `ComboBox`, we will link the selected value in the display to a regular property in the window's code-behind. And we will react to the change event by re-executing the query according to the new filtering criterion.

Since we want to avoid naming controls (especially the `ComboBox`), we will not be able to access the `SelectedValue` attribute of the `ComboBox` through the code-behind. In addition, we want that every time the selection in the `ComboBox` changes, we will re-execute the query according to the new filtering criterion.

Therefore, through the XAML, we will use the **Data Binding** mechanism to link the `SelectedValue` attribute of the `ComboBox` control to a regular property that we will define in the window's code-behind. The value of this property will be updated every time the selected value in the `ComboBox` changes.

1. In the previous sections, you defined the DataContext of the entire window through the XAML file of the window to be linked to its own properties. You added a `DataContext` attribute with a relative value (`RelativeSource`) that links it to itself (`Self`) to the `Window` element.
2. In the code-behind of the window, define a **regular property** (meaning not a Dependency Property), with a name you choose, of the type of the enum defined in BO, by which to filter. We will link this property to the selected item in the `ComboBox`. This property will only be updated from the display towards the code-behind and not vice versa, and therefore it can be a regular property and not a Dependency Property.
3. Set an initial value for the property, according to the value by which you want the list to be filtered when the window opens.

4. In most cases, we want the list to be displayed in full and without filtering when the window opens, so it is advisable to add an additional value to the enum that represents a "general category" such as `None` or `All`.

For example:

```
public BO.SemesterNames Semester { get; set; } = BO.SemesterNames.None;
```

5. Through the XAML file of the window, add an attribute named `SelectedValue` to the `ComboBox` control that represents the selected value in it. Link it through the `Path` attribute of the Data Binding mechanism to the enum type property you defined in the previous section. With `Mode=TwoWay`.

Note: Because we defined the linked property as a regular property in the code-behind and not as a Dependency Property, it will not report changes to the display. The direction of the change will be from the display, using the Binding mechanism which will update the property in the code every time the selection in the display changes. The only time the property will receive a value from the code towards the display will be during the window initialization, and therefore we still defined `Mode=TwoWay`. If we change the value of the property through the code, it will not affect the display. For it to affect the display, we would need to define the property as a Dependency Property, and `Mode=TwoWay` is not enough.

For example:

```
<ComboBox Grid.Column="1" SelectedValue="{Binding Path=Semester, Mode=TwoWay}"
ItemsSource="{Binding Source={StaticResource SemestersCollectionKey}}"/>
```

And again, you can also omit the `Path`:

```
<ComboBox Grid.Column="1" SelectedValue="{Binding Semester, Mode=TwoWay}"
ItemsSource="{Binding Source={StaticResource SemestersCollectionKey}}"/>
```

6. After linking this property to the selected value in the `ComboBox`, we want to react to the value change event and cause the list of items to be updated in the display. Use the default event of clicking on the `ComboBox`, the event is called `SelectionChanged`.
7. Give the event method a meaningful name.
8. In the implementation of the event method:
9. The value of the filtering criterion is automatically updated into the enum property you defined in the code-behind, according to the selection in the `ComboBox` in the display. And there is no need to add anything further.
10. According to the value of the filtering criterion that will be automatically updated, you must use a call to the BL layer method that returns a filtered list and update the

`IEnumerable` Dependency Property that you defined in the code-behind of the window. This update will automatically cause the list in the display to be updated.

For example:

```
CourseList = (Semester == BO.SemesterNames.None) ?
    s_bl?.Course.ReadAll()! : s_bl?.Course.ReadAll(null, BO.CourseFieldFilter.SemesterName,
Semester)!;
```

8c. List Display Window - Defining Observers for the Item List in the Database

1. In the code-behind of the list display window, define a "List Observation Method" - a private method that will "observe" changes in the list of items in the database.
2. The method will not be invoked in PL but will be invoked by the BL implementation class of that entity every time the list is updated in the database (adding/updating/deleting an item in the list). As we implemented in Chapter 1 of this stage.
3. In the implementation of the method, run the same query from the previous section that returns the filtered list again. (Code that repeats itself? Maybe it is advisable to wrap it in a private helper method?)
4. In the code-behind of the list display window, register for the list display window loading event (`Loaded`), and in the implementation of the event method, add the observation method from the previous section as an observer in the BL.
5. In the code-behind of the list display window, register for the list display window closing event (`Closed`), and in the implementation of the event method, remove the observation method from the previous section from being an observer in the BL.

For example:

```
private void queryCourseList()
    => CourseList = (Semester == BO.SemesterNames.None) ?
        s_bl?.Course.ReadAll()! : s_bl?.Course.ReadAll(null,
BO.CourseFieldFilter.SemesterName, Semester)!;

private void courseListObserver()
    => queryCourseList();

private void Window_Loaded(object sender, RoutedEventArgs e)
    => s_bl.Course.AddObserver(courseListObserver);

private void Window_Closed(object sender, EventArgs e)
    => s_bl.Course.RemoveObserver(courseListObserver);
```

6. Run the program and open the list display window. Change the selection in the `ComboBox` and verify that the list is updated according to the filter.

7. If you still used `ListView`/`DataGrid` in a simple way, the resulting display inside the `ListView`/`DataGrid` is actually the `ToString` of the entity. In the next chapter, we will talk about refining the list display.

8d. List Display Window - Refining the List Display

Whether you chose to work with a `DataGrid` or `ListView` control - you must include within it a definition of a **Template** for some of the columns (each row represents an entity object and each column represents an entity property).

For example, a Boolean property should be displayed with a `CheckBox` control, a different text color for the content of a specific column that you wish to emphasize, and so on.

In such a case, each column must be linked separately to the corresponding property in the Logical Data Entity displayed in the list through the **Data Binding** mechanism.

For example, if you chose the `DataGrid` control for displaying the list:

```
<DataGrid ItemsSource="{Binding Path=StudentList}" IsReadOnly ="True"
   AutoGenerateColumns="False"
   EnableRowVirtualization="True" RowDetailsVisibilityMode="VisibleWhenSelected"
   MouseDoubleClick="dgStudentList_MouseDoubleClick" SelectedItem="{Binding SelectedStudent}">
   <DataGrid.Columns>
        <DataGridTextColumn Binding="{Binding Id}" Header="Id" Width="Auto"/>
        <DataGridTextColumn Binding="{Binding Name}" Header="Name" Width="Auto"/>
         ...
        <DataGridTemplateColumn Header="Is Active" Width="Auto">
            <DataGridTemplateColumn.CellTemplate>
               <DataTemplate>
                    <CheckBox IsEnabled="False" IsChecked="{Binding Path=IsActive}" />
               </DataTemplate>
            </DataGridTemplateColumn.CellTemplate>
        </DataGridTemplateColumn>
        ...
   </DataGrid.Columns>
</DataGrid>
```

Chapter 9 - Single Item Display Window

Now, we will create a window that represents a single item with all its properties, of the selected comprehensive Logical Data Entity type. For example: **BO.Course**.

For each property of the Logical Entity, a pair of controls will be displayed on the window - description and value. The values of the controls will be linked in XAML, through the Binding mechanism, to the fields of an object of the Logical Data Entity type that will be initialized in the code-behind.

BS"D

The window will open in 2 modes:

- **Add mode** - all the fields of the object will be without a value or with a default value. The user will fill in the values through the display and then click the `Add` button, which will add the new object to the database using the BL layer.

- **Update mode** - all the fields of the object will be filled with the values of an existing object. The user will update the values he wishes to update through the display and then click the `Update` button, which will update the existing object in the database using the BL layer.

Reminder: Do not name display controls, but work with full Data Binding.

9a. Single Item Display Window - Control Design

The window is supposed to represent a single object of the selected comprehensive Logical Data Entity type.

1. For each property of the Logical Entity, drag a pair of controls from the ToolBox into the Grid onto the window:

2. Right - the property description `Label`.

3. Left - the property value (choose a control of type `TextBox`, `ComboBox`, `CheckBox`, etc. according to the value type).

4. Each such control has an attribute that represents the value - we will link these attributes through the Binding mechanism to the fields of an object of the Logical Entity type that will be initialized in the code-behind - we will elaborate on how to do this shortly.

5. You must ensure that every column of controls is aligned to the same point and of the same size. Think about how to do this.

6. You can set these attributes for each control, but it is recommended to use advanced XAML commands such as `Style`, `Resources` to set the designs globally, alternatively.

7. You must take into account the "Don't Repeat Yourself" (DRY) principle in order to minimize as much as possible the code smell of "Needless Repetition".

8. You may prefer to do this in Stage 6 and not now.

9. In addition, using an internal Grid may help, but it is not mandatory.

10. Note to match the control type to the property type, for example:

11. A property of type string will match a `TextBox` control.

12. A property of type `Enum` will match a `ComboBox` control.

13. A property of type Boolean will match a `CheckBox` control.

14. A property of type `DateTime` will match a `DatePicker` control.

15. Create a `Button` control at the bottom of the window, inside the `Grid`. This single button will be used both for the add operation and for the update operation. After the user fills in/updates all the values of the controls/object, they will click the button to add a new entity or update an existing entity.

16. You must ensure that the text that appears on the add/update button is determined from the window's constructor according to the mode in which the window is opened: add/update.

17. Through the XAML, you will need to set the value of the `Content` attribute of the button using the Binding mechanism and link it to a Dependency Property (e.g., named `ButtonText`) of type string that you will define in the code-behind of the window.

```
<Button Content="{Binding ButtonText}" Click="btnAddUpdate_Click"/>
```

- From the window's constructor (even before the call to `InitializeComponent`), set the text of the Dependency Property according to the mode in which the window is opened: add/update.

```
ButtonText = id == 0 ? "Add" : "Update";
```

9b. Single Item Display Window - Linking the Window to the Object

The window represents a single item of the selected comprehensive Logical Data Entity type (e.g.: `BO.Course`). We want that when the window opens, all the controls in XAML representing the properties of the entity are linked to an existing object of that entity type:

- If the window is in add mode, we want them to be linked to a new object we will create with empty/default values.
- If the window is in update mode, we want them to be linked to an existing object we will retrieve from the BL, with full values of that item for update.

1. In the code-behind, create a private field that will allow access to the BL, as you did in the previous window. So that you can request the desired object from it.
2. Through the XAML file of the window, define the `DataContext` of the entire window to be linked to its own properties. Add an attribute named `DataContext` with a relative value that links it to itself, to the `Window` element.

```
<Window x:Class="PL.Course.CourseListWindow"
        ...
        DataContext="{Binding RelativeSource={RelativeSource Mode=Self}}">
```

3. In the code-behind of the window, define a **Dependency Property**, with a name you choose, of the Logical Data Entity type. So that we can load the appropriate object from the BL into it. It is recommended to use the snippet named `propdp`.

```
public BO.Course? CurrentCourse
{
        get { return (BO.Course?)GetValue(CurrentCourseProperty); }
        set { SetValue(CurrentCourseProperty, value); }
}

public static readonly DependencyProperty CurrentCourseProperty =
        DependencyProperty.Register("CurrentCourse", typeof(BO.Course), typeof(CourseWindow),
new PropertyMetadata(null));
```

4. In the code-behind of the window, add an `int` parameter with a default value of 0 to the constructor, which represents the Id of the entity whose properties the window will display. In the next chapter, we will implement the code that creates the window from the list display window, and through this parameter, we will transfer information to the current window whether it is in update mode for an existing entity or creation mode for a new entity.
5. In the code-behind of the window, add commands to the body of the constructor, after `InitializeComponent`, that will perform an assignment of an object of the Logical Entity type into the Dependency Property, with the name you chose. According to the `Id` parameter received in the window's constructor:
6. If an `Id` with a default value arrives - then a new object should be created (`new`), with empty values or default values, for every property of the object.
7. Otherwise - the appropriate BL method should be called, according to the `Id` to get the existing object from the Dal (remember to catch exceptions).

For example:

```
CurrentStudent = (id != 0) ? s_bl.Student.Read(id)! : new BO.Student() { Id = 0, CurrentYear = BO.Year.None, RegistrationDate = s_bl.Admin.GetClock() };
```

8. We want to link the appropriate attributes of every single control on the window to its corresponding property in the object in the code-behind, all without naming the controls. In addition, we want that every time the value of the control changes in the display, the property linked to it in the object also changes. Through the XAML file of the window, for every control that represents a value, link the relevant attribute through the **Binding** mechanism to the corresponding field of the Dependency Object that you defined in the code-behind.

For example, for a string type field represented on the window with a `TextBox`:

```
<TextBox Grid.Row="2" Grid.Column="1" Text="{Binding CurrentCourse.CourseName, Mode=TwoWay,
        NotifyOnValidationError=true,  ValidatesOnExceptions=true}"
        HorizontalAlignment="Left" Height="NaN" Margin="3"  VerticalAlignment="Center" Width="120"/>
```

And for example, for an Enum type field represented on the window with a `ComboBox`:

```
<ComboBox Grid.Row="3" Grid.Column="1" HorizontalAlignment="Left" Height="NaN" Margin="3"
        ItemsSource="{Binding Source={StaticResource YearsCollectionKey}}"
        SelectedValue="{Binding CurrentCourse.InYear, Mode=TwoWay, NotifyOnValidationError=true,
        ValidatesOnExceptions=true}" VerticalAlignment="Center" Width="120"/>
```

Note that in this window as well, you must populate the list of items in the `ComboBox` using the Binding mechanism, as you learned to do in the previous window and without naming the controls.

9c. Single Item Display Window - IsReadOnly Attribute and Visibility Attribute in the 2 Window Modes

Note: This section requires the use of a **Converter**, a slightly advanced topic. Those for whom this is difficult can postpone it to Stage 6.

An explanation of the definition and use of a Converter can be seen in this document in: Appendix 2 - Example of Defining and Using a Converter.

Since the window is intended to handle 2 modes that do not exist simultaneously, but sometimes the window will open in add mode and sometimes in update mode - the differences between the 2 modes must be handled in terms of control display. The `Id` property of the object that we defined in the window's Dependency Property is what determines the mode; if it has a default value, we are in add mode, otherwise - update mode.

1. If there are entity properties that should not be updated in "update" mode, but we still want them to appear on the screen, then we must ensure that when the window opens, their corresponding controls are "visible" but "disabled" `IsReadOnly="True"`. For example, the `Id` property:
2. If the entity's `Id` property is a "running number," then the control linked to the object's `Id` property should be "visible" but "disabled" in both modes, because:
3. In update mode - the object arrives with an existing Id and it is forbidden to change it.
4. In add mode - the Id arrives with a default value, and receives a value automatically only after the actual addition of the new object to the BL. And does not receive a value from the user.
5. Since it is forbidden to name controls, through the XAML, the `IsReadOnly` attribute must be set, once when the window opens, using the Binding mechanism, and linked to the object's `Id` property. And since they are not of the same type, an appropriate **Converter** that converts values accordingly must also be defined.
6. The same applies to properties that should not be visible at all in a certain mode (add or update).

For example:

```
<TextBox Grid.Row="0" Grid.Column="1"  HorizontalAlignment="Left" Height="NaN" Margin="3"
 Text="{Binding CurrentStudent.Id, Mode=TwoWay, NotifyOnValidationError=true, ValidatesOnExceptions=true}"
 IsReadOnly="{Binding ButtonText, Converter={StaticResource ConvertUpdateToTrueKey}}"
 VerticalAlignment="Center" Width="120"/>

<Label Grid.Row="5" Grid.Column="0" Content="Registration Date:"
    Visibility="{Binding ButtonText, Converter={StaticResource ConvertUpdateToVisibleKey}}"
         HorizontalAlignment="Left" Margin="3"  VerticalAlignment="Center"/>

<DatePicker IsEnabled="false" Grid.Row="5" Grid.Column="1"
    Visibility="{Binding ButtonText, Converter={StaticResource ConvertUpdateToVisibleKey}}"
         HorizontalAlignment="Left" Height="NaN" Margin="3" SelectedDate ="{Binding
CurrentStudent.RegistrationDate, Mode=TwoWay, NotifyOnValidationError=true, ValidatesOnExceptions=true}"
VerticalAlignment="Center" Width="120"/>
```

9d. Single Item Display Window - Clicking the Add/Update Button to Implement Add/Update Operations

After the user fills in/updates all the values of the controls/object, they will click the button to add a new entity or update an existing entity.

1. Add the default click event (`Click`) to the button and implement the event method as follows:
2. It is possible to distinguish between the modes by the value of the Dependency Property that represents the text on the add/update button.

```
if (ButtonText == "Add") ...
```

3. Depending on the window mode (add/update), call the appropriate method in the BL that receives the entity object (Dependency Property) as a parameter.

s_bl.Student.Create(CurrentStudent!);
or
s_bl.Student.Update(CurrentStudent!);

4. After performing the add/update operation, an appropriate success message must be sent to the user and the single item display window must be closed.
5. If a BL exception was thrown, it must be caught and an appropriate `MessageBox` must be sent as a result.
6. Run the program and verify that after every operation, the list in the list display window was updated automatically.

9e. Single Item Display Window - Defining Observers for the Single Item

Of course, every window you add to the project from now on must be defined as an observer of the content it displays. In the same way that we taught you to define observers in the previous windows. This is clear, but we will mention it anyway.

1. An observation method must be defined that will refill the item.

```
int id = CurrentStudent!.Id
CurrentStudent = null;
CurrentStudent = s_bl.Student.Read(id);
```

2. You must register for the window loading event and add the observation method as an observer for a single item.

```
if (CurrentStudent!.Id != 0)
    s_bl.Student.AddObserver(CurrentStudent!.Id, studentObserver);
```

3. You must register for the window closing event and remove the observation method as an observer for a single item.
4. You can verify that you defined the observation correctly if you open the single item window twice simultaneously on the same item. And as soon as you update the content of the item and click update, the content of the second window will be automatically updated (and of course also the content of the list display window).

BS"D

Chapter 10 - Opening the Single Item Display Window from the List Display Window

Now, we will return to the list display window to open the single item display window through it:

- Double-clicking on an item in the list will open the single item display window in update mode for the selected item.
- Clicking the `Add` button at the bottom of the list will open the single item display window in add mode.

10a. List Display Window - Linking the Selected Item in the List to a Property in the Code-Behind

We want to define a property in the code-behind of the window of the constrained Logical Data Entity type represented as a single item in the list, so that we can automatically link it to the selected item from the list.

1. In the code-behind of the window, we will define a **regular property** (meaning not a Dependency Property), with a name you choose, of the Logical Data Entity type. We will link this property to the selected item in the `ListView`/`DataGrid`. This property will only be updated from the display towards the code-behind and not vice versa, and therefore it can be a regular property and not a Dependency Property.

For example:

```
public BO.CourseInList? SelectedCourse { get; set; }
```

2. Through the XAML file of the window, add an attribute named `SelectedItem` to the `ListView`/`DataGrid` control that represents the selected value in it. Link it through the Data Binding mechanism to the property you defined in the previous section.

For example:

```
<ListView Grid.Row="1" ItemsSource="{Binding CourseList}" SelectedItem="{Binding
SelectedCourse}" MouseDoubleClick="lsvCoursesList_MouseDoubleClick"/>
```

3. This is how the link is created automatically.

BS"D

10b. Opening the Single Item Display Window in Update Mode from the List Display Window

In order to open the single item display window in update mode, after double-clicking on an item in the list:

1. Add a double-click event (`MouseDoubleClick`) on an item in the list to the `ListView`/`DataGrid` control.
2. Implement the event method so that a new window of the single item display window type is created in update mode (meaning the Id of the selected item in the list must be sent to the constructor of the created window).
3. Verify that the window opens in `Show` mode, meaning that as soon as it opens, it is possible to return to the previous window until the current window is closed.

For example:

```
private void lsvCoursesList_MouseDoubleClick(object sender, MouseButtonEventArgs e)
{
        if (SelectedCourse != null)
        new CourseWindow(SelectedCourse.Id).Show();
}
```

10c. Opening the Single Item Display Window in Add Mode from the List Display Window

In order to open the single item display window in add mode, after clicking the `Add` button:

1. At the beginning of the document, we added an `Add` button at the bottom of the list display window. Add the default click event (`Click`) to the button.
2. Implement the event method so that a new window of the single item display window type is created in add mode (meaning the Id of any item does not need to be sent to the constructor of the created window).
3. Verify that the window opens in `Show` mode, meaning that as soon as it opens, it is possible to return to the previous window until the current window is closed.

10d. Refreshing the List After Adding/Updating an Item

1. Run the program, open the list display window, add/update items in the list through the single item display window.
2. Verify that after every addition/update/deletion of an item in the list - the list is updated automatically thanks to the observer mechanism you implemented.

BS"D

Chapter 11 - List Display Window - Deleting a Single Item from the List

1. In the list display window, using the **Template** of `ListView`/`DataGrid`, add a separate button for each row in the list. So that clicking the button will perform the deletion of the item to which the button belongs in the list:
2. Before deleting, verify using a `MessageBox` that the user is sure they want to delete the selected item.
3. Call the `Delete` method in the BL, which will try to perform the deletion of the item.
4. If the deletion was successful, the item will automatically disappear from the list thanks to the observer mechanism we implemented.
5. If the deletion failed (the BL layer checks if it is allowed to delete, etc., as mentioned), the exception from the BL must be caught and an appropriate `MessageBox` must be sent instead.
6. Another option for implementing single item deletion - as stated in the general document: implement the deletion from the "Single Item Display Window" - your choice. The main thing is that there is an option to delete a single item. Note that the delete button is "visible" only when the window opens in add/update mode. Of course, in each of the ways, the rules of correct deletion must be maintained.

Chapter 12 - Customizing the Window Design

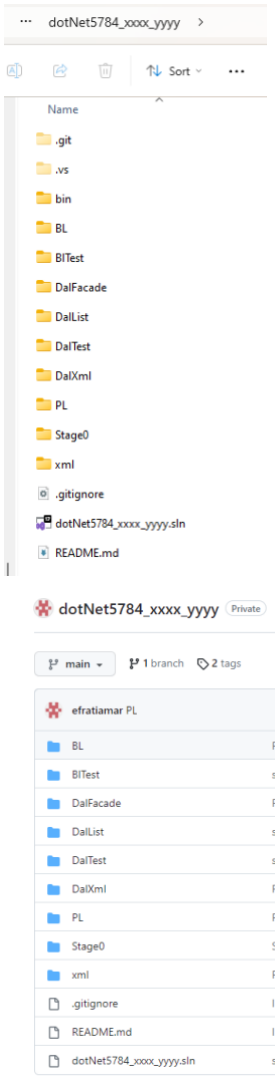
Here you have a free hand to design the windows as your imagination allows, you can choose colors, fonts, titles, and more.

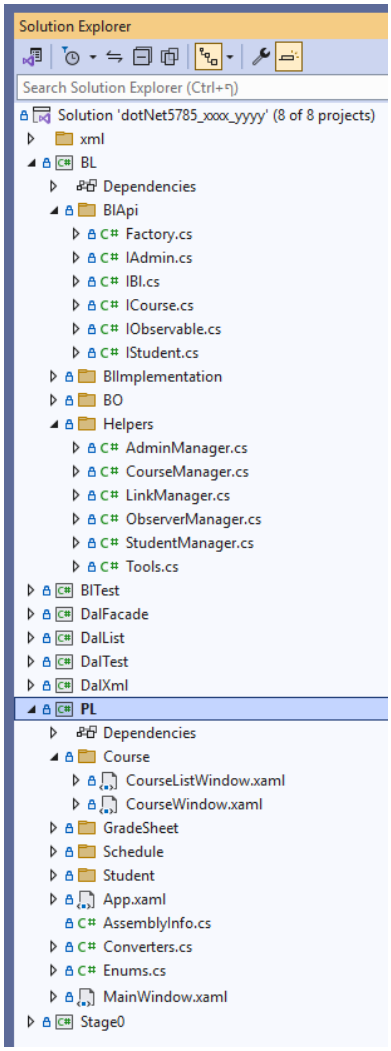
You can also use advanced WPF topics as they appear at the beginning of this document. But you can wait with them until Stage 6.

Chapter 13 - Recap

After completing all chapters of Stage 5, the repository on Github, the Solution in Visual Studio, and the folders in File Explorer should look like this:

BS"D





Chapter 14 - Creating a Tag and Submitting Stage 5 on Moodle

According to the instructions explained in Stage 0:

- Create a tag named: `Stage5 Final commit`
- Submit a link to the tag on Moodle for each of the partners.

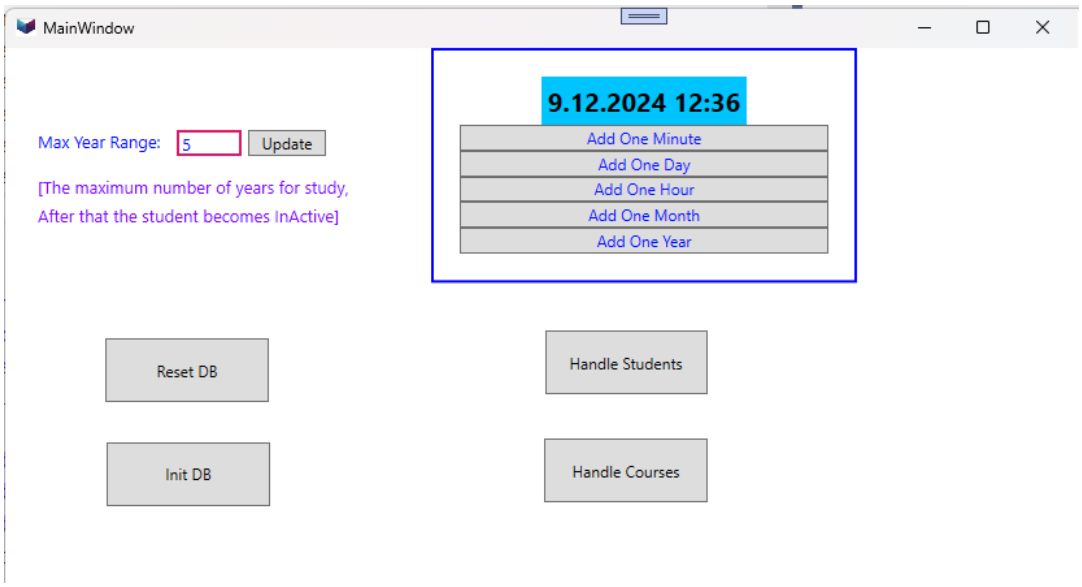
End of Stage 5 - Good luck!

BS"D

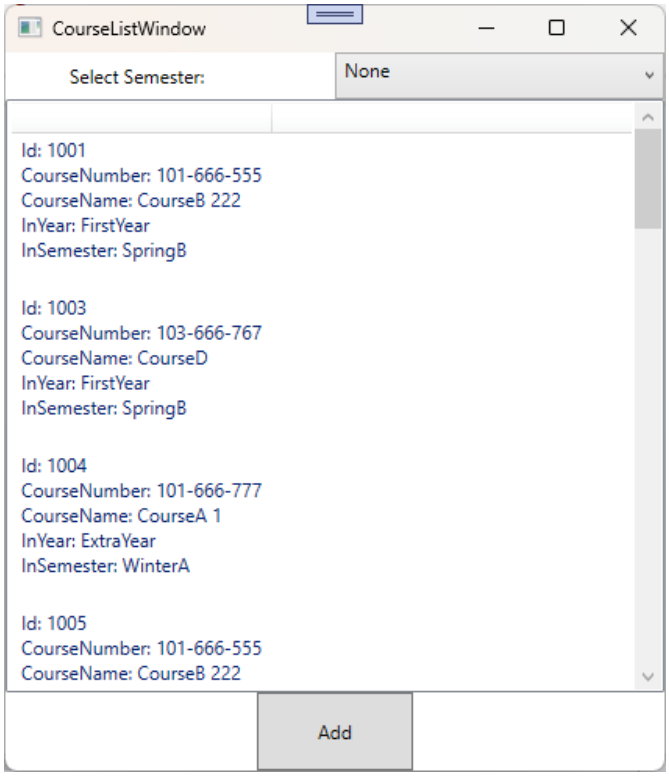
Appendices - Stage 5

Appendix 1 - Window Examples

Description
Example of Main Management Window - Screen Number 1

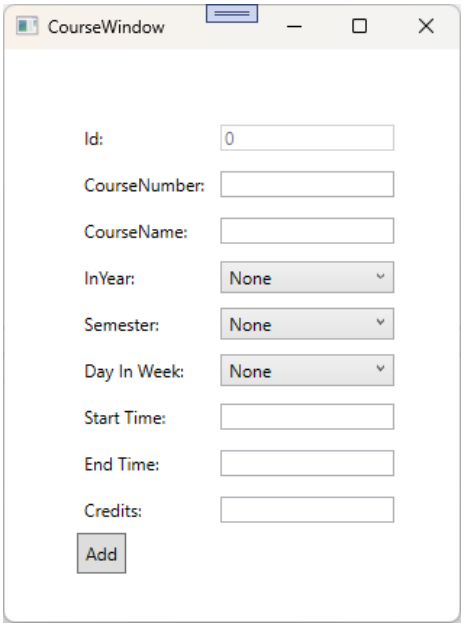


Example of List Display Window - Screen Number 2: The list display is not advanced - and the requirement in this stage is to refine it and display it in a more sophisticated way using a Template

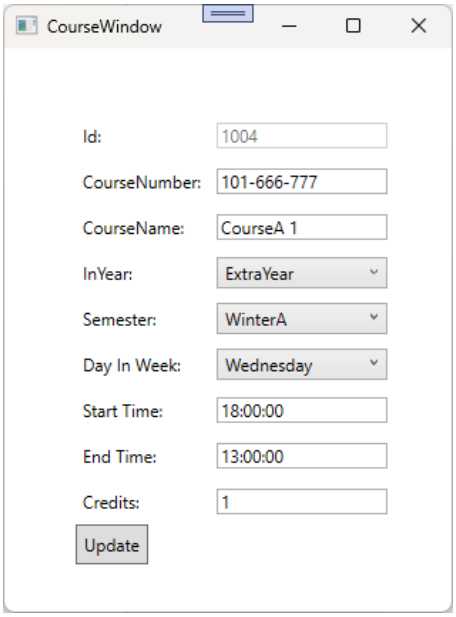


Example of Single Item Display Window in Add Mode - Window Number 3

Empty form



After adding information



Appendix 2 - Example of Defining and Using a Converter

Assume we want to link between 2 properties of a graphic component and a logical component in XAML through the Binding mechanism that are not of the same type. Since the 2 properties are not of the same type, an appropriate **Converter** that converts from one type to the other must also be defined in the XAML in the Binding area.

For example, assume we want to display an `Enum` type value, which represents a course year of study, inside a `TextBlock` control.

The `Text` property of the control is linked using the Binding mechanism as usual.

We want a link between the `Enum` value and the `Background` color of that control - and they are not of the same type!

For this purpose, we will define a Converter:

1. In the PL project, create a new file named `Converters.cs`, which will be dedicated to all the Converters you will need in the PL project.
2. Define a new class inside the new file that inherits from `IValueConverter`.

For example

```
class ConvertYearToColor : IValueConverter
{
        public object Convert(object value, Type targetType, object parameter, CultureInfo culture)
        {
        BO.Year year = (BO.Year)value;
        switch (year)
        {
        case BO.Year.FirstYear:
                return Brushes.Yellow;
        case BO.Year.SecondYear:
                return Brushes.Orange;
        case BO.Year.ThirdYear:
                return Brushes.Green;
        case BO.Year.ExtraYear:
                return Brushes.PaleVioletRed;
        case BO.Year.None:
                return Brushes.White;
        default:
                return Brushes.White;
        }
        }

        public object ConvertBack(object value, Type targetType, object parameter, CultureInfo culture)
        {
        throw new NotImplementedException();
        }
}
```

3. In the project's resource file **`App.xaml`**, add a definition of a static resource with an **x:Key** key with a name you choose, and link it to the Converter you defined in the previous section.

For example:

```
<Application.Resources>
      <local:ConvertYearToColor x:Key="ConvertYearToColorKey" />
</Application.Resources>
```

4. We will specify the name of the Converter using the Binding mechanism through the XAML:

For example:

```
<DataGridTemplateColumn Header="InYear" Width="Auto">
       <DataGridTemplateColumn.CellTemplate>
       <DataTemplate>
              <TextBlock Text="{Binding InYear,Mode=TwoWay}" FontSize="16"
Background="{Binding Path=InYear, Converter={StaticResource ConvertYearToColorKey}}" />
       </DataTemplate>
       </DataGridTemplateColumn.CellTemplate>
</DataGridTemplateColumn>
```