90322: ”SOLID Programming in C#”

Workshop Description

Wincubate ApS

11-02-2022

V1.0

Table of Contents

Prerequisites

The present labs require the course files accompanying the course to be extracted in some directory path, e.g.

```
C:\Wincubate\90322
```

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

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

Workshop A: "Creating a SOLID Domain Layer Structure"

A.1: "Initial Setup and Inspection of Project"

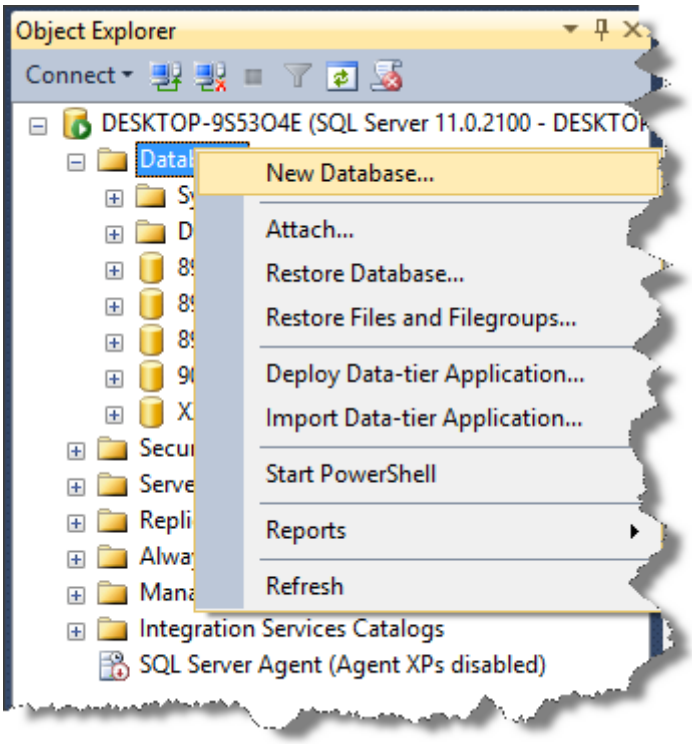
This step will get you set up for workshop development.

- Open the starter project in
  PathToCourseFiles`\Workshop\A.1\Starter` ,
  which contains the Visual Studio solution `Cinema`.
- Familiarize yourself with the pre-existing code and structure
  - Notice that a number of code files have already been added.

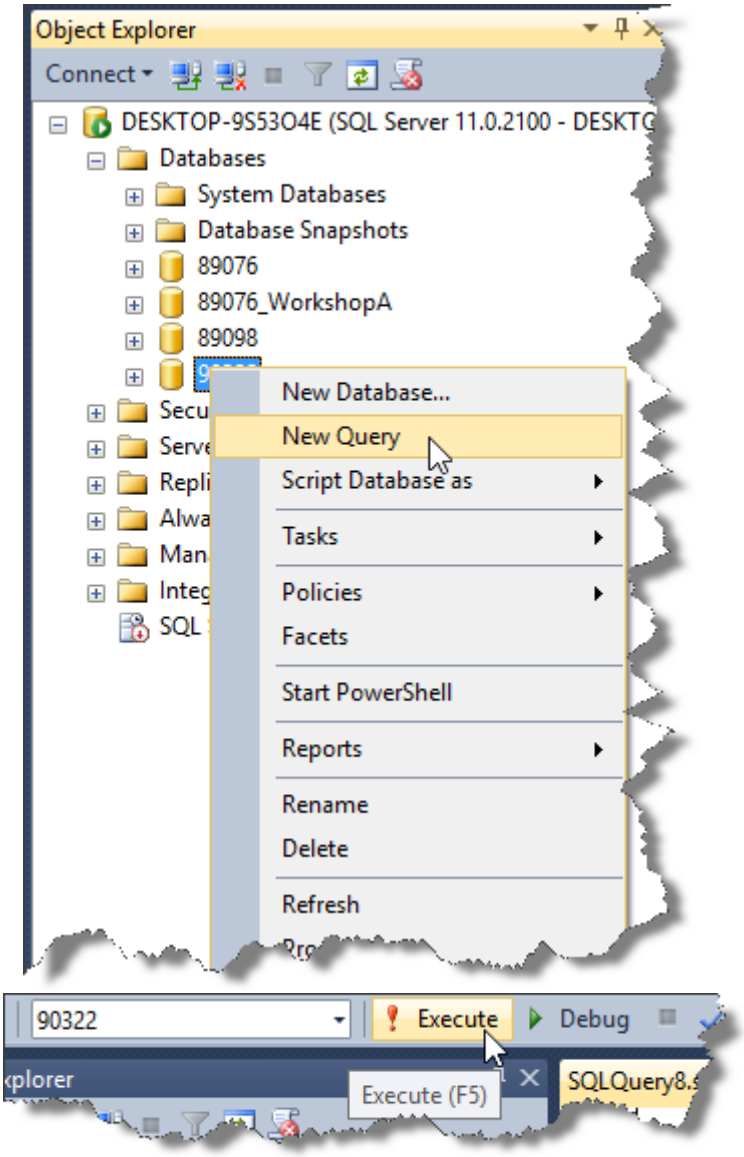
You will now create the database and tables used throughout this workshop in a SQL Server Express.

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

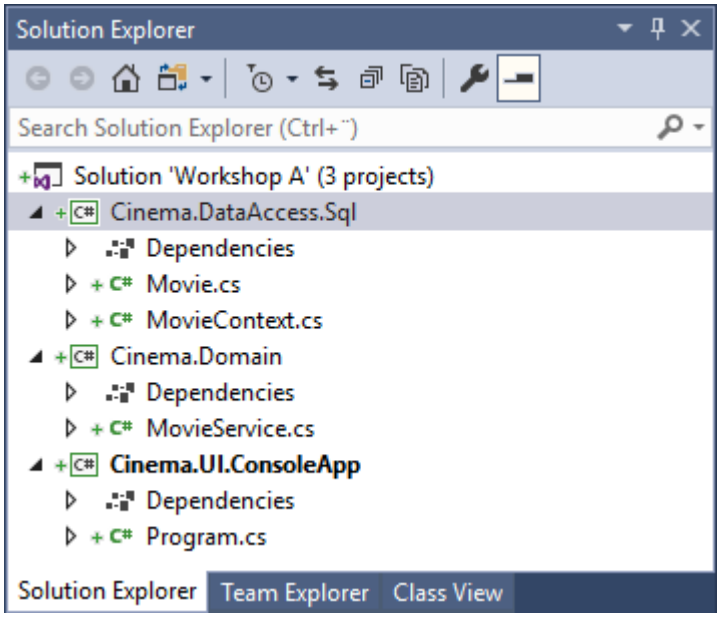
- Open SQL Management Studio and create a new database named `90322`
  - Right-click the `Database` node and select `New Database`…



  - Create the 90322 database using default settings.
- Generate the `Movies` SQL database from the scripts located in
  PathToCourseFiles`\Workshop\01 - Create Movies DB.sql`
  by executing this script on the database just created.
  - Choose `New Query` and drag the script file to the query window
  - Click `Execute` (or press F5)



You are provided with an existing Starter solution consisting of a number of distinct projects. An overview is provided here:



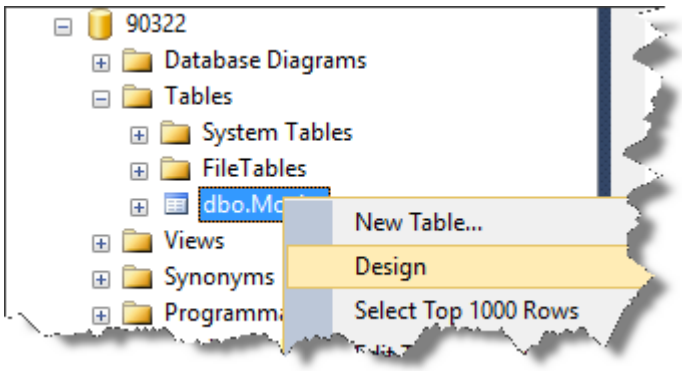
You change existing projects and add more projects to the above solution as you progress through the workshop. Class libraries are .NET Standard whereas the executable projects are .NET Core. References between the projects have already been set up.

- **Cinema.UI.ConsoleApp**
  - Is a console application which you will use to run your application.
- **Cinema.Domain**
  - Contains domain layer (or business logic) of your application.
- **Cinema.DataAccess.Sql**
  - Contains a SQL-based Data Access Layer unit tests for the **Cinema.Domain** project.

The solution currently does not compile! To make it compile and run, you will need to add the contents of the `Movie` class with properties reflecting the schema of the `Movies` table in the 90322 database, as it is currently empty:

```
public class Movie
{
    // TODO: Add properties
}
```

- Inspect the schema of the `Movies` tables by right-clicking the the table in SQL Management Studio and choosing `Design` :



- Add properties to the Movie class mapping to the appropriate .NET types, e.g.
    - `Id` should be of type `Guid`
    - `Year` should of type `short`
    - etc…

  Remember to make to nullable SQL types also nullable C# types, e.g.
    - `ImdbRating` should be `double?`
    - `TicketPrice` should of type `decimal?`
    - `LastUpdated` should of type `DateTime?`

When everything compiles;

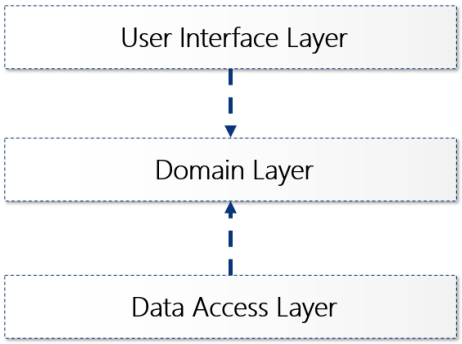
- Ensure that the **Cinema.UI.ConsoleApp** is set as startup project.
- Run the program and checks that the application displays a view of the movies in the database which are currently showing.

A.2: "Data Access Layer with Repository"

Following the discussion in Module 02, you will now fix most of the problems with the current design. You will also implement the **Repository** pattern for improved separation, maintainability, and testability of the application.

Improve the application by completing the following steps:

- Fix the references between the layers such that it follows the guidelines presented in the module presentation:



  - Move classes and references for e.g. `Movie` and `MovieContext` to the right layers.
  - Note: If you want, you're allowed to split one or more layers into two projects, though this is in no way mandatory.
- Implement the **Repository** pattern for data access.
  - Feel free to implement whichever version of repository you see fit; Either of Simple, Queryable, Generic, … will do!
- Run your new, beautiful program and check that it still outputs the correct set of movies.

If Time Permits…

- Since there is no particular reason to have all the unused members of `Movie` propagate to the higher layers of the system, modify the service to return a more focused domain type, e.g. `MovieShowing` containing only the relevant properties of `Movie`.
  - What would this type look like?
  - And what is the correct location for this type?
- Consider if it would be possible to introduce a Presentation Logic Layer with view models and supporting classes?
- Would such a step somehow simplify or accelerate creating other UIs in e.g. WPF or ASP.NET (or both) presenting the same information?

A.3: "Test the Domain Layer and Change the Data Access Layer"

In this part of the workshop we will illustrate just how testable and adaptable the new application design is. In fact, we can supply positive answers to all three questions from the module discussion.

Unit Testing the Movie Service

Firstly, we address the testability of our design.

- Create a unit test of the `MovieService` class ensuring that it returns exactly those movies where the `IsShowing` property is true
    - **Important**: To create a unit project, you can add an **MSTest Test Project (.NET Core)** project to the solution in Visual Studio
        - Depending upon your Visual Studio configuration, you might have to update the **MSTest.*** nuget packages after project is added
    - Note: During the test, you should not read any files, config, or access the SQL database.

If Time Permits…

Secondly, we will illustrate the loosely-coupled nature of the design by swapping out the current SQL Data Access Layer for an XML-based implementation.

- Change the Data Access Layer to load the movies from an XML file with the following contents:

```
<?xml version="1.0"?>
<Movies xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Movie>
    <Id>180eb091-c529-4d57-8a9d-ce51d362a495</Id>
    <Name>XML as an exciting data source... Not!</Name>
    <Year>2019</Year>
    <Plot>The movie is set in a galaxy far, far away where XML is
considered exciting</Plot>
    <ImdbRating>4.2</ImdbRating>
    <TicketPrice>79</TicketPrice>
    <IsShowing>true</IsShowing>
    <LastUpdated>05/06/2019 11:19:54</LastUpdated>
  </Movie>
  <Movie>
    <Id>eb7031f8-ff4c-49a8-9920-5490c61c7c1c</Id>
    <Name>Singleton is an evil motherf****r</Name>
    <Year>2019</Year>
    <Plot>This is the story about a good guy from 1994 which turned evil
over time</Plot>
    <ImdbRating>9.9</ImdbRating>
    <TicketPrice>109</TicketPrice>
    <IsShowing>true</IsShowing>
    <LastUpdated>05/06/2019 11:19:54</LastUpdated>
  </Movie>
</Movies>
```

Note: The file is also available in

PathToCourseFiles\Workshop\02 – Movies.xml

- If you have not done so already, now might be a good time to introduce Solution Folders in Visual Studio for organizing the projects into the layers to which the belong.

If Time Still Permits…

If you did not finish the *"If Time Permits"* in Workshop A.2, then you might have time to revisit it here:

- Since there is no particular reason to have all the unused members of `Movie` propagate to the higher layers of the system, modify the service to return a more focused domain type, e.g. `MovieShowing` containing only the relevant properties of `Movie`.
    - What would this type look like?
    - And what is the correct location for this type?
- Consider if it would be possible to introduce a Presentation Logic Layer with view models and supporting classes?
- Would such a step somehow simplify or accelerate creating other UIs in e.g. WPF or ASP.NET (or both) presenting the same information?

A.4: "A Test of Time in the Domain Layer"

With the movie service working a nice and simple way, we will now add a common feature to the Domain Layer: Time…!

Friday is the top night of the week, so let's offer a great deal for customers.

- Implement a "Happy Hour on All Movies" campaign applying a discount of 50% off any movie for the entire duration of Fridays (and Fridays only!)
- Create unit tests asserting that your movie service provides the correct Happy Hour discount on Fridays – and no discounts any other day of the week.

Sounds simple, right? 😊

A.5: "Create a Null User Context"

Your boss wants to introduce a Members' Club, where users can register and log in. When properly authenticated Club Members will receive an (additional) 10% discount on all pricing – even on top of Happy Hour!

Your boss wants you to complete the movie service today and put it in production tomorrow. You think he's crazy because you have not at all discussed how users should log in or authenticate. And you don't know if the service is going to run in an WPF app or as a web site. The authentication schemes for each are totally different!

He claims that you can easily implement and test(!) the movie service today and worry about the actual authentication scheme next month. He instructs you to use the following enum for user levels:

```
public enum UserRole
{
    Customer,
    ClubMember,
}
```

and the following interface for the users:

```
public interface IUserContext
{
    bool IsInRole(UserRole role);
}
```

Your objectives are now clear:

- Complete implementation of the service providing Club Members with a 10% discount.
- Test that your completed service satisfies both the new (and the existing) ticket price specifications.

Note: Feel free to use any test framework you find appropriate; MS Test, NUnit, or similar.

(Extra:) A.6: "Adding Dependency Injection"

In this workshop we will add dependency injection containers to the mix. You can use any DI container that you see find interesting, but we have assumed that you will be using Unity developed by Microsoft.

Firstly;

- Add the Unity DI Container to your solution to Workshop A
  - Add it to the Console App UI version first.
  - Use the **Register-Resolve-Release** pattern.
  - Note: If you want to test your injection with your XML provider, you might need to use the
    - `IUnityContainer.RegisterFactory()` or
    - `IUnityContainer.RegisterInstance()` methods,

    if you supplied the file name as a construction parameter.

Secondly;

- Add the Unity DI Container to all automatic tests in your solution to Workshop A
  - How is this slightly different from Unity injection in your Composition Root of your app?
  - Does it make sense to do injections everywhere in the tests?

You're free to slightly change the design if it makes everything more beautiful, of course.