

# 89076: "Design Patterns in C#"

---

## Workshop Description

Wincubate ApS  
16-04-2018



## Prerequisites

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

C:\Wincubate\89076

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

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

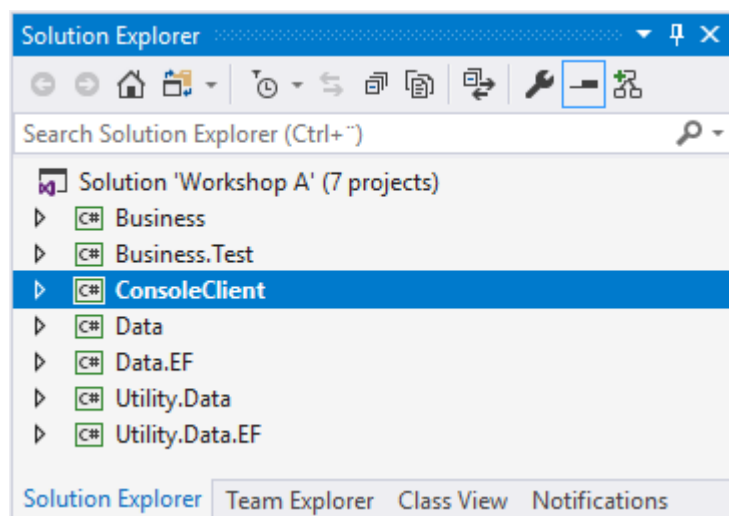
## Workshop A: “Creating a Reusable Messaging Framework”

This workshop consists of building a larger, more realistic solution for applying the 23 foundational design patterns from the Gang of Four book along with the other patterns covered on Day 4.

The workshop consists of a number of individual steps, which are specified somewhat more loosely than the module labs you have encountered so far. Your objective is to cover as many steps as you can with the time frame allocated to you by the instructor. You will probably not be able to finish all of the steps!

From Step 4 and onwards most steps of the workshop can be carried out in any order. So be sure to implement those that you find most challenging and/or interesting as you see fit.

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



You might 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. Many references between the projects have already been set up.

- **ConsoleClient**
  - Is a console application which you will use to run your application.
- **Business**
  - Contains the business logic of your application.
- **Business.Test**
  - Contains unit tests for the **Business** project.
- **Data**
  - Contains general data classes.
- **Data.EF**
  - Contains Entity Framework-specific data classes.
- **Utility.Data**
  - Contains reusable data classes.
- **Utility.Data.EF**
  - Contains reusable Entity Framework-specific data classes.

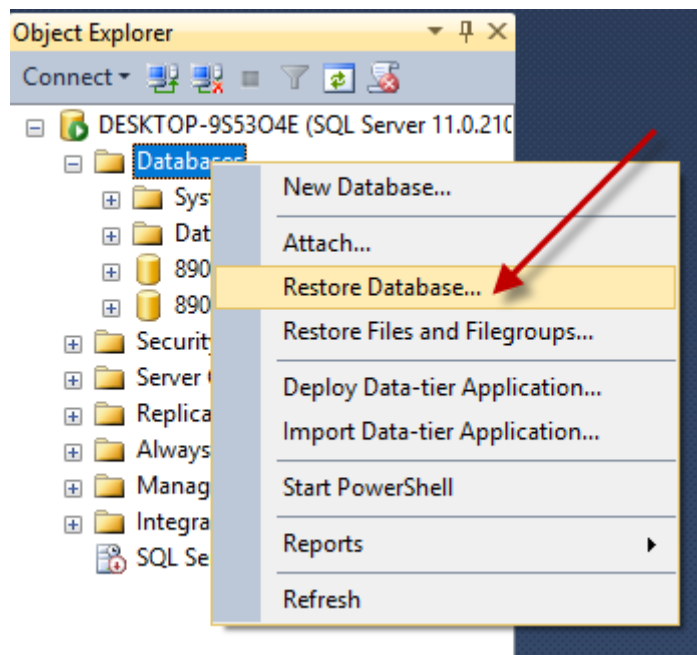
## Step 0: “Setup”

This step will get you set up for workshop development.

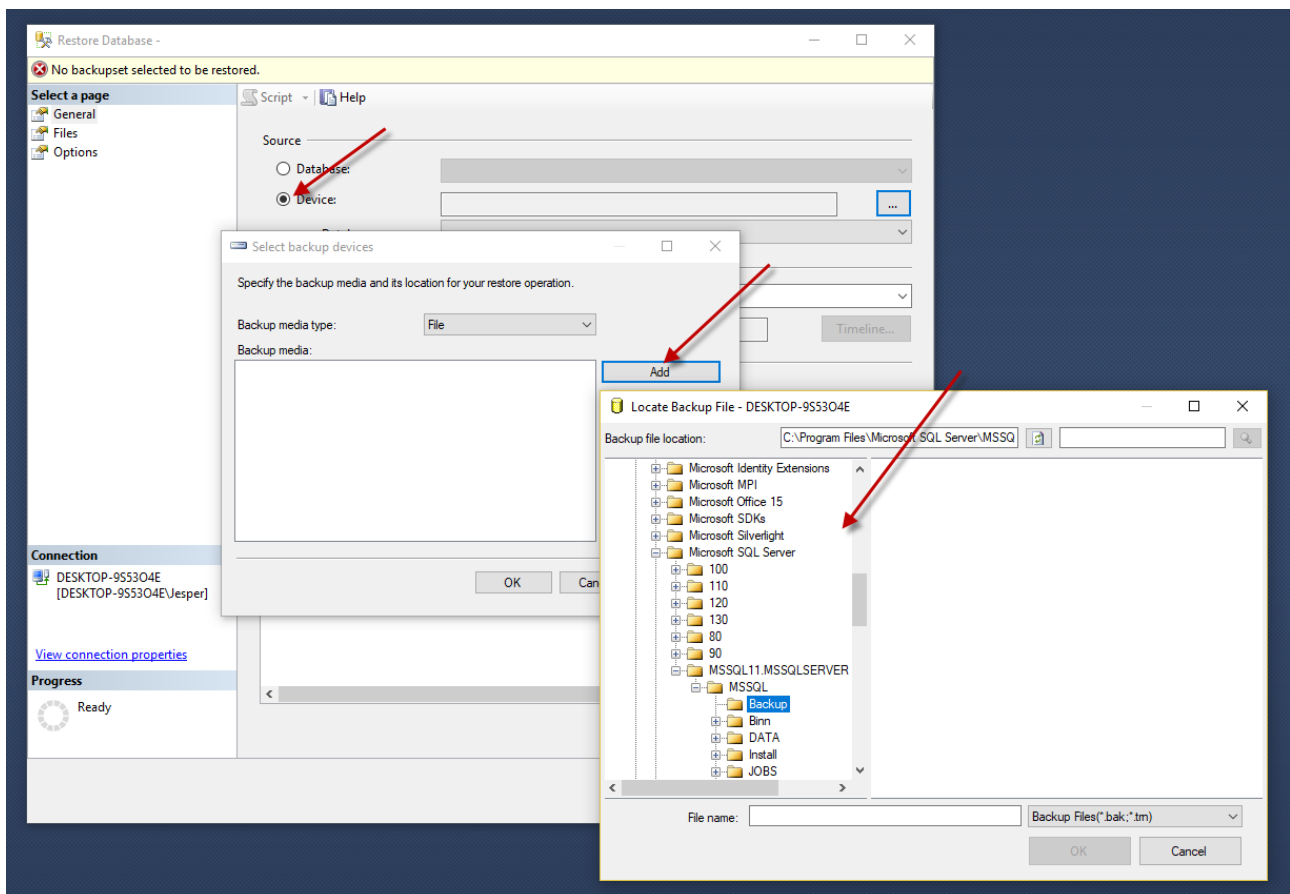
- Open the starter project in  
*PathToCourseFiles\Workshop\A\Starter* ,  
which contains the solution WorkShop A.
- Familiarize yourself with the pre-existing code and structure
  - Notice that a number of codes files have already been added.
  - Notice that some of the projects however are still empty.
- Open SQL Management Studio and restore the SQL database located in  
*PathToCourseFiles\Workshop\A\89076 \_WorkshopA.bak*

## Background

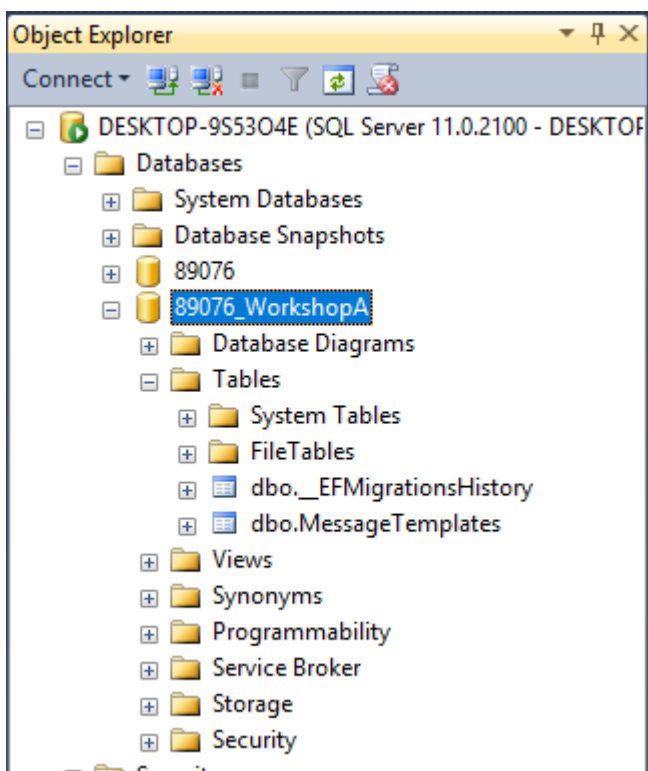
In the **Object Explorer** tool window right-click **Databases** and choose “**Restore Database...**”



Then follow the arrows and browse to the backup file to restore it:



If everything went well you should now have a SQL database in your SQL server called 89076\_WorkshopA:



## Step 1: "Create Repository Classes for Accessing MessageTemplates SQL Table"

You will need to access the predefined set of message templates located in the SQL database called "89076 WorkshopA".

- Open SQL Management Studio and inspect the contents of the MessageTemplates table.
- Use the **Repository pattern** to create
  - Repository interface
  - Repository implementation classfor accessing the **MessageTemplate** instances in the SQL database from .NET code.
- To test your repository implementation, write a simple console test program in the **ConsoleClient** project
  - Verify that all the message templates from the database are displayed correctly.

Note: You decide which version of Repository to implement – simple or generic! But aim to place the files you author in the appropriate pre-existing projects.

### Background

The following data class is already defined in the **Data** project:

```
/// <summary>
/// POCO class capturing a single message template definition
/// from the underlying database.
/// </summary>
public class MessageTemplate
{
    /// <summary>
    /// Gets or sets the Id of the <see cref="MessageTemplate"/>.
    /// </summary>
    public int Id { get; set; }

    /// <summary>
    /// Gets or sets the culture identifier of the
    /// <see cref="MessageTemplate"/>.
    /// </summary>
    public string Culture { get; set; }

    /// <summary>
    /// Gets or sets the Text template of the
    /// <see cref="MessageTemplate"/>.
    /// </summary>
    public string Text { get; set; }

    /// <summary>
    /// Provides a string representation of the
    /// <see cref="MessageTemplate"/>.
    /// </summary>
    /// <returns>String representation of the current
    /// <see cref="MessageTemplate"/>.</returns>
    public override string ToString() =>
```

```

        $"{Id}\t{Culture}\t{Text}";
    }

```

The following database context class has also been defined in the **Data.EF** project:

```

/// <summary>
/// Entity Framework context class for accessing
/// <see cref="MessageTemplate"/> instances in the underlying
/// database.
/// </summary>
public class MessageTemplatesContext : DbContext
{
    /// <summary>
    /// Provides IQueryable-access to the <see cref="MessageTemplate"/>
    /// instances in the underlying table.
    /// </summary>
    public DbSet<MessageTemplate> MessageTemplates { get; set; }

    /// <summary>
    /// Configures Entity Framework to use the SQL Express database.
    /// </summary>
    /// <param name="optionsBuilder"></param>
    protected override void OnConfiguring(
        DbContextOptionsBuilder optionsBuilder )
    {
        optionsBuilder.UseSqlServer(@"Server=.\SQLEXPRESS;Database=89076
_WorkshopA;Trusted_Connection=True;");
    }
}

```

Note: If you have placed the database tables in a location *different* from what was specified in the exercise text, you will need to modify the connection string within the `OnConfiguring()` method accordingly.

## Step 2: "Create a Text Substitution Class"

You will now create a text substitution mechanism in the **Business** project for providing actual contents for the substitution placeholders of the retrieved message templates.

- In the **Business** project, create a `TextSubstitutor` class with the following signature:  

```
public string Substitute( MessageTemplate messageTemplate,  
                        IEnumerable<object> parameters )
```
- Test your implementation validating it against the two pre-existing unit tests in the `TextSubstitutorTest` class of the **Business.Test** project to make sure it works correctly.

Note: Don't advance to the next step until these unit tests have been commented in, run, and successfully passed.

### Background

The exception-related types `MessagingException` and `MessagingExceptionReason` are already defined in the **Business** project.



### Step 3: "Create a Messenger Class Transmitting Email Messages"

You will now create a very first rough version (to be improved throughout the workshop) of a `Messenger` class which accepts a message from the client and sends the substituted (or "resolved") version of the message as an email to the specified recipient.

The message from the client to be sent will be supplied in the shape of an `IMessage` object. That type – along with the obvious implementation class `Message` – is already defined in the starter project as follows:

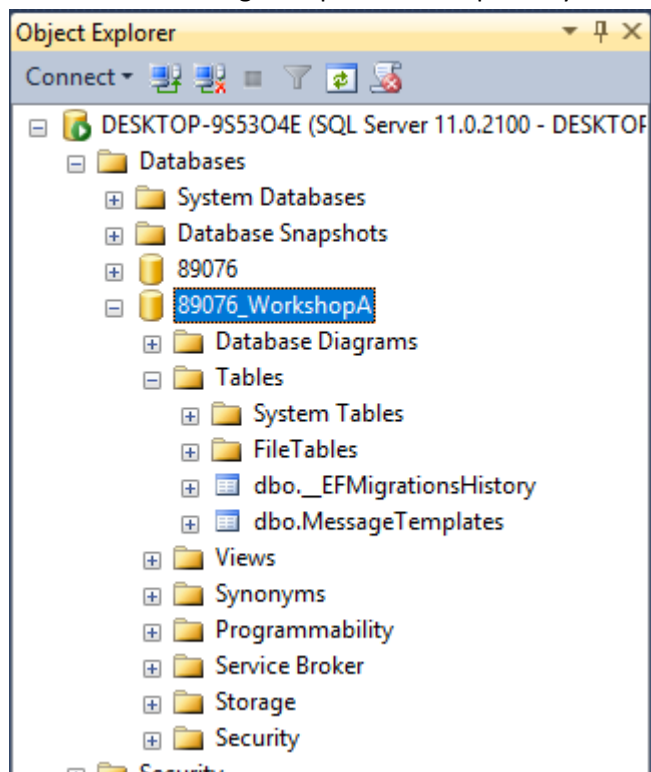
```
/// <summary>
/// General interface for messages to be resolved and sent
/// using the <see cref="Messenger"/>.
/// </summary>
public interface IMessage
{
    /// <summary>
    /// Gets the recipient of the <see cref="IMessage"/>.
    /// </summary>
    IUser Recipient { get; }

    /// <summary>
    /// Gets the Id of the <see cref="Data.MessageTemplate"/>
    /// to use.
    /// </summary>
    int MessageTemplateId { get; }

    /// <summary>
    /// Gets the parameters to be substituted into the
    /// <see cref="Data.MessageTemplate"/>.
    /// </summary>
    IEnumerable<object> Parameters { get; }
}
```

The `Messenger` implementation is an instance of the **Façade pattern** essentially wrapping these three steps to getting the message sent:

a) Retrieve the MessageTemplate from repository defined in



- Step 1: "Create Repository Classes for Accessing MessageTemplates SQL Table".
- Use the TextSubstitutor of Step 2: "Create a Text Substitution Class" to create a `SingleMessageInstance` object.
  - Physically transmit the resolved and substituted `SingleMessageInstance` message to the `IUser` by means of some transmitter – in this case via email.

The latter type `SingleMessageInstance` is also predefined:

```

/// <summary>
/// Class for instantiated instances, i.e. it has
/// already been resolved and substituted from an
/// <see cref="IMessage"/>.
/// </summary>
public class SingleMessageInstance
{
    /// <summary>
    /// Message string contents.
    /// </summary>
    public string Contents { get; }

    /// <summary>
    /// Unique application-identifier for message instance.
    /// </summary>
    public Guid InstanceId { get; }

    /// <summary>
    /// Creates a new <see cref="SingleMessageInstance"/> with
    /// the message contents specified in <paramref name="contents"/>.
    /// </summary>
    /// <param name="contents">Message string contents.</param>
    public SingleMessageInstance( string contents )
    {
        Contents = contents;
        InstanceId = Guid.NewGuid();
    }
}

```

Your job is now to perform as to construct the code of steps a) – c).

- Create a `Messenger` class with a `Send()` method with the following signature:  
`public void Send( IMessage message )`
- Implement the transmitter functionality of `Messenger` as a strategy of the **Strategy pattern** where your concrete strategy here transmits the `IMessage` by sending it as an appropriate email with some appropriate subject.
- Update Program.cs of the **ConsoleClient** project with the appropriate test code sending yourself a test email using your newly constructed `Messenger`.

## Background

The user-related types `IUser` and `User` are already defined in the **Business** project.

If you have access to a Gmail account, you can use `smtp.gmail.com` to physically transmit the emails. See [https://www.siteground.com/kb/google\\_free\\_smtp\\_server/](https://www.siteground.com/kb/google_free_smtp_server/) for more info.

Use the `System.Net.Mail.SmtpClient` class to send emails. If you use `smtp.gmail.com` then you need to set the following properties:

```
Host = host,  
Port = 587,  
EnableSsl = true,  
UseDefaultCredentials = false,  
Credentials = new NetworkCredential( "<user>@gmail.com", "<password>" )
```

If you decide to use your own Gmail account it is necessary to log into your Gmail account and create a specific password to use for external mail sending by creating one here:

## Signing in to Google

Control your password and account access, along with backup options if you get locked out of your account.


**Make sure you choose a strong password**  
A strong password contains a mix of numbers, letters, and symbols. It is hard to guess, does not resemble a real word, and is only used for this account.

### Password & sign-in method

Your password protects your account. You can also add a second layer of protection with 2-Step Verification, which sends a single-use code to your phone for you to enter when you sign in. So even if somebody manages to steal your password, it is not enough to get into your account.

**Note:** To change these settings, you will need to confirm your password.

Password	Last changed: January 30, 5:22 PM	>
2-Step Verification	On since: January 30, 5:24 PM	>
App passwords	1 password	>



You can then revoke that password at the end of this workshop.

#### Step 4: "Testing the Messenger with a Unit test"

Ideally, we would like to have the test code (not in the **ConsoleClient** project but) in a proper unit test in the **Business.Test** project. However, we want to test the **Messenger** class without sending a physical email every time we run our unit tests.

Fortunately, the transmission of the previous step was implemented using the **Strategy pattern** and not hardcoded inside the **Messenger** class. Consequently, we can define a specific in-memory strategy for use with unit testing which merely stores the messages in-memory for the unit test to validate.

Similarly, we also don't want to read message templates from the SQL database every time we run our unit tests. Once again we're quite fortunate in having chosen a good pattern – the **Repository pattern** for data access!

- Create an appropriate unit test validating that **Messenger.Send()** works correctly
  - Without connecting to the database
  - Without physically sending any emails.

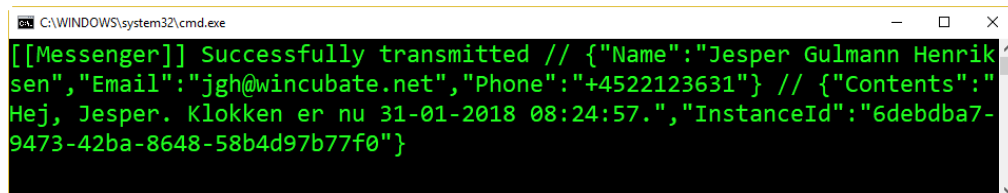
## Step 5: "Implement a Logging Mechanism for Messenger"

We will now implement diagnostic features to convey what happens inside the Messenger class.

- Use the **Abstract Factory pattern** to define a logger which logs entries to the console in an interface similar to:

```
public interface ILogger
{
    void Info( string message, params object[] additional );
    void Warn( string message, params object[] additional );
    void Error( string message, params object[] additional );
}
```

The intention is Info-logs appear as green text in the Console. Similarly, Warn and Error appear in yellow and red, respectively. A sample log for an Info-logged message with two objects could be along the lines of



- Modify `Messenger` to accept an `ILoggerFactory` at construction time and introducing logging into the `Send()` method to make an invocation perform a log of a successful send (as above).
- Note: In order for the unit tests to compile and run, you will need to use the **Null Object pattern** to device a logging mechanism suitable for unit tests.

## Step 6: "Yet Another Logger"

When running a real application as a command-line script, it is not very useful to obtain the diagnostic information in the console window as that closes after execution. With the infrastructure in place from *Step 5: "Implement a Logging Mechanism for Messenger"* it is however a simple endeavour to add another logger.

- Add a `FileLogger` class, which logs to a file in the current directory with the name passed to `Create()`.
- The log lines should in the same format as `ConsoleLogger`, but with one of the strings below prepended:
  - `"INFO -- "`
  - `"WARN -- "`
  - `"ERROR -- "`

As the formatting for both the `FileLogger` and `ConsoleLogger` are almost the same, you might refactor the existing implementation to a variation of the **Template Method pattern**.

## Step 7: "Create an SMS Transmission Strategy"

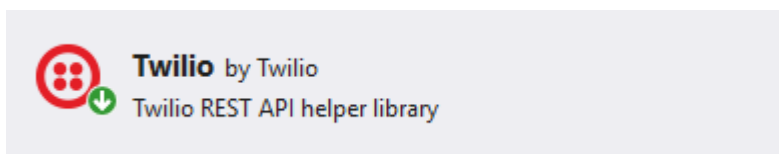
In Step 3: "Create a Messenger Class Transmitting Email Messages" we created the transmission infrastructure and instantiated a strategy for sending the messages as emails. We will now see how easy it is to extend the functionality to send SMS test messages instead.

- Create a SMS transmission strategy using the Twilio SMS API
- Test your implementation by sending yourself the messages as SMS messages.

### Background

See the Twilio SMS API documentation at <https://www.twilio.com/docs/api/messaging/send-messages>.

In order to use the Twilio API for sending SMS messages you must include their nuget package into your project:



Once it is included, you can send an SMS message using the following code

```
string _accountSid = "ACa5?64844f11c4152c5e4db4bc202c7??";  
string _authToken = "b978????5570945b775bac117f5b7059";  
  
TwilioClient.Init(_accountSid, _authToken);  
MessageResource mr = MessageResource.Create(  
    new PhoneNumber("<phone number>"),  
    from: new PhoneNumber("+4676???9439"),  
    body: "<contents of SMS>"  
);
```

Your instructor will give you the remaining digits substituting the missing ?'s in the above codes.



## Step 8: "Implement Transparent Caching"

As more and more messages are sent using the [Messenger](#), more or more lookups to database will be performed. However, the message templates in the database virtually never change so the same data is looked up very many times.

- Use the **Proxy pattern** to create a [CachingRepository](#) class to save multiple round-trips to the database.
- Note: Don't overthink this! It's not important for us here.
  - You can assume that the underlying database table does not change until application is restarted.
  - Don't care about all details such as cache expiry, immutability, thread-safety etc.

## Step 9: "Wrapping Up"

As a final step in rounding off today's work, fill the gaps we have ignored thus far.

- Carefully consider all appropriate classes to ensure that all relevant classes implemented the **Dispose pattern** (or its basic version).
  - Update your use of these classes appropriately.
- While you're at it, also consider adding loggers and logging diagnostic information where appropriate.
- Also, make sure you're wrapping all exceptions and errors internal to [Messenger](#) to [MessagingException](#) instances when leaving the [Messenger](#).