EasyCSVNet 1.0.1

A free and easy CSV importing and exporting library with XML definition support written in VB.Net

HowTo quick guide

Version: **1.0.1**	Last revision: **2017-09-11**
Author: **Àlex de Buen Lapena** (alex.debuen@gmail.com)	

1. Introduction

The CSV ("Comma Separated Value") file format is a well-known simple plain text open format often used to exchange data between disparate applications. Its formal definition is given by RFC 4180 and can be found here:

https://www.ietf.org/rfc/rfc4180.txt

A lot of widespread applications (like MS Excel ©, among many others) deal with this kind of files, making CSV a very handy format to manipulate for a great number of programmers' everyday work needings.

EasyCSVNet is a simple and lightweight VB.Net solution to deal with these kind of files in a somewhat naïve way. It is not centered in formal details concerning the RFC specification mentioned above, but in providing a short, usable and hands-on implementation of an 'OCM' (*Object-to-CSV Mapping*) library to get CSV input files parsed into DTOs (*Data Transfer Objects*) which can be comfortably manipulated from business layer in your applications. So the main effort in EasyCSVNet development was not being strictly CSV-compliant, but offering mechanisms to support and deal with data files implementing the spec and also usual little 'deviations' from it (inclusion of comment lines, multiple non-comma field separators, …) which are frequently seen in practice, and let the programmer free to spend his/her effort in more important and usually critical parts of his/her project.

CSV importing process takes place in this way: the mapping schema is declared in a separate XML definition file, which in short specifies rules for binding every desired field in a CSV line to a DTO attribute, so every CSV line is mapped to a DTO object instance, and consequently the whole CSV input file to a `List` of them. One of the interesting points here is that reflection carries away all CSV field-to-attribute binding, type inference and casting work underneath at this stage, although customization is also possible by means of inheritance and hook overriding over default importer classes (to be explained in detail below). Syntactic inspection against an XSD schema (also included in project) adds value to the import process, as it raises detailed error messages pointing out wrong lines/fields in input CSV data, which can be quickly detected therefore. Really useful to detect flaws and missing/malformed data when typically working with very big CSV files!

CSV exporting is also supported in a more minimalistic mode, not based in XML this time, as this reverse way mapping (from DTO to CSV) appears as simpler than import process.

No benchmarking info or performance metrics are included, as the aim of this piece of software is simply offering a minimally invasive and more declarative and 'aseptic' way to deal with quotidian CSV-or-the-like files as an alternative to more conventional approaches based in ad-hoc line-by-line parsing, which tend to hide real business logic or bury it under disturbing boilerplate code. Developers who decide to give EasyCSVNet a try within their projects surely will be able to build and run suitable unit tests in very little time and with little effort, which will give them real metrics to decide if they go ahead with this tool or not.

So, in sum, as a developer, I've written the (first version of the) tiny library I would have liked to have the first time I needed to parse a considerable amount of different CSV files as part of a project, and I can say it has proven to be useful for my needings from that day on in several times. I will be glad if it proves to be useful for other developers too, and saves them time and effort!

2. About this solution's code and sample tests included

EasyCSVNet is a standalone VB.Net solution, written with MS Visual Studio 2017 Community © IDE. It comes in the form of a single solution splitted in two projects:

- A *Class library* ('**EasyCSVNet**' properly) containing all library code and ready to be exported to other projects; you can take away the whole code or just the compiled DLL **`bin/Release/EasyCSVNet.dll`**)
- A *Console Application* ('**TestConsoleApp**') runnable project which references to the library, and serves to illustrate how to use it in a couple of sample tests

Both projects are configured by default to be run on .NET Framework 4.6.1+ target over 'Any CPU' architecture (although they have been developed in a x86 32bit architecture environment under MS Windows 7 SP1 OS and .NET Framework 4.6.1 - the only scenario in which they have been run actually). EasyCSVNet library has no dependencies with 3rd party packages, nor even extra references imported other than these commonly included by default in a new .Net 4.6.1+ class library project by MS Visual Studio.

So the recommended way to use EasyCSVNet for the first time is to download/checkout the whole solution code (`EasyCSVNet` directory), import the solution file (`EasyCSVNet.sln`) inside from MS Visual Studio 2017 Community © and press *Start* button. You should see a shell window

appearing, showing the output results of the two sample tests included, and waiting for a key to be pressed by the user to end the execution. The whole code will be also visible and browsable (for example, in the *Solution Explorer* side window typically).

You can also copy **`EasyCSVNet/bin/Release/EasyCSVNet.dll`** binary in a newly created `TestConsoleApp/lib` directory, add a reference to it from `TestConsoleApp` project, and drop the entire `EasyCSVNet` project as long as you are not interested anymore in EasyCSVNet's guts. Even better, if that's the case, you can download and install EasyCSVNet from **NuGet** repository in the usual way:

- From Package Manager:

```
PM> Install-Package easycsvnet -Version 1.0.1
```

- or from .Net CLI:

```
> dotnet add package easycsvnet --version 1.0.1
```

In both cases, you just need to import `EasyCSVNet.easycsvnet.*` required classes.

Here is the relevant path structure of the solution briefly explained, all directories included in a root `EasyCSVNet` parent:

- ***EasyCSVNet*** project:
  - **`bin`**: Contains compiled binaries, which will be automatically (re)generated at each run by the IDE. In particular it contains **`bin/Release/EasyCSVNet.dll`** binary, to be exported if you want
  - **`conf/csvOCM`**: It includes the XSD schema for OCM (Object-to-CSV Mapping) in an XML file (`schema/csvOCM_SimpleCSVFileImporter_schema.xsd`).
  - **`doc`**: There is contained the documentation you are reading just now and some other stuff related
  - **`src`**: It includes all `*.vb` code files of the project, classified in namespaces:
    - **`easycsvnet.csv`**: Generic classes involved in all CSV manipulation jobs
    - **`easycsvnet.csv.import`**: Generic classes and interfaces used in CSV importing
    - **`easycsvnet.csv.export`**: Generic classes and interfaces used in CSV exporting
    - **`easycsvnet.util`**: Helper utility classes, somewhat heterogeneous, used transversally in part by the main ones
- ***TestConsoleApp*** project
  - **`bin`**: Contains compiled binaries, which will be automatically (re)generated at each run by the IDE
  - **`conf/csvOCM`**: It includes XML definition files for the two sample tests (`csvOCM_TestColourDTO_def.xml` and `csvOCM_TestPatientDTO_def.xml`) which implement the XSD schema mentioned above, as every other XML definition file shall do
  - **`src`**: It includes all `*.vb` code files of the project, classified in namespaces:
    - **`testconsoleapp.test`**: Runnable sample tests (and related classes consumed in **`testconsoleapp.test.test001`** and **`testconsoleapp.test.test002`**)
  - File **`MainModule.vb`** is the only code `*.vb` file placed outside the **`src`** directory, in the project root path. It contains the `main()` runnable starting method (which calls unit test suite method `UnitTestSuite.runAllTests()` in turn).

Any exception thrown by the application will be catched inside `UnitTestSuite.runAllTests()` method and sent to `UnitTestSuite.handleExceptionCustomCallback()` method, as we can see in the code inside the first one:

```
        Try
                [...]'Run unit tests 001 and 002
        Catch myException As Exception
              _exitCode = ExitCode.EXIT_ERROR
              handleExceptionCustomCallback(myException)
        Finally
              Console.WriteLine(vbCrLf & "Press a key to exit...")
              Console.ReadKey(True)
        End Try
```

By default, the exception will be turned into a detailed error message and shown by stdout console for simplicity. More sophisticated log tracing strategies will be up to each developer, just need to deal with thrown exception in the desired way.

Now we are talking about log traces and error messages, let's see just a pair of examples of which kind of alerts we typically get when we try to parse a syntactically wrong CSV input file:

```
Invalid CSV file format @line #2 :: illegal field number doesn't match DTD XML schema
                        [line number: 2, field position: 3, attribute name: '', attribute type: 'System.String'
                         , field value: '']
```

This exception message we get in case CSV input file has a line with wrong number of fields: in particular field number 3 (that is, the 4th one, as the index is 0-based) in text line #2 in CSV input file is not matching XML mapping definition, which specifies a lower number of fields per CSV line to be expected.

```
Input string was not in a correct format.
                        [line number: 2, field position: 1, attribute name: '_price'
                         , attribute type: 'System.Nullable`1[System.Single]', field value: 'NaN']
```

And this one we get in case CSV input file has a line with wrong field values: in particular field number 1 (that is, the 2nd one, as the index is 0-based) in text line #2 in CSV input file has incorrect format and cannot be properly parsed and mapped to DTO's attribute '*_price*', as '*NaN*' string value cannot be casted into `Single?` expected type. In this particular case, if we wanted this kind of non-numerical strings to be correctly processed, we should register a custom type mapper linked to this type, as we will explain in chapter 4.c below.

Finally, before we get into more detailed explanations, some general words about tests included in *TestConsoleApp* project. `UnitTestSuite.runAllTests()` method (called from `MainModule.main()`) will run the two sample unit tests provided:

- `UnitTestSuite.runTest001()` will run first test and show results by stdout console, which involves:
  - importing the `ext/test/test_001/colours.csv` input CSV file
  - parsing it by means of rules defined in XML definition file `conf/csvOCM/csvOCM_TestColourDTO_def.xml`
  - building up a list of `TestColourDTO` business objects
  - exporting this list to the `ext/test/test_001/colours_output.csv` output CSV file (inside `bin` directory)
- `UnitTestSuite.runTest002()` will run second test and show results by stdout console, which involves:
  - importing the `ext/test/test_002/patients.csv` input CSV file
  - parsing it by means of rules defined in XML definition file `conf/csvOCM/csvOCM_TestPatientDTO_def.xml`
  - building up a list of `TestPatientDTO` business objects
  - exporting this list to the `ext/test/test_002/patients_output.csv` output CSV file (inside `bin` directory)

Details about each one of these steps will be explained in chapters 4 and 5 below.

3. Class diagram and design notes on the solution

Here we present a simplified UML class diagram of the whole EasyCSVNet solution. Hopefully it will help to figure out the project «big picture» in a single glimpse. We have intentionally omitted most of private methods, attributes and inner classes, for example, and skipped also interface inherited methods representation in descendant classes.

Dotted lines not indicating interface realization stand for simple 'usage' relation between classes. Gray boxes indicate native VB.Net classes (such as the ones included in `System` package). Green packages and classes belong to *TestConsoleApp* project.

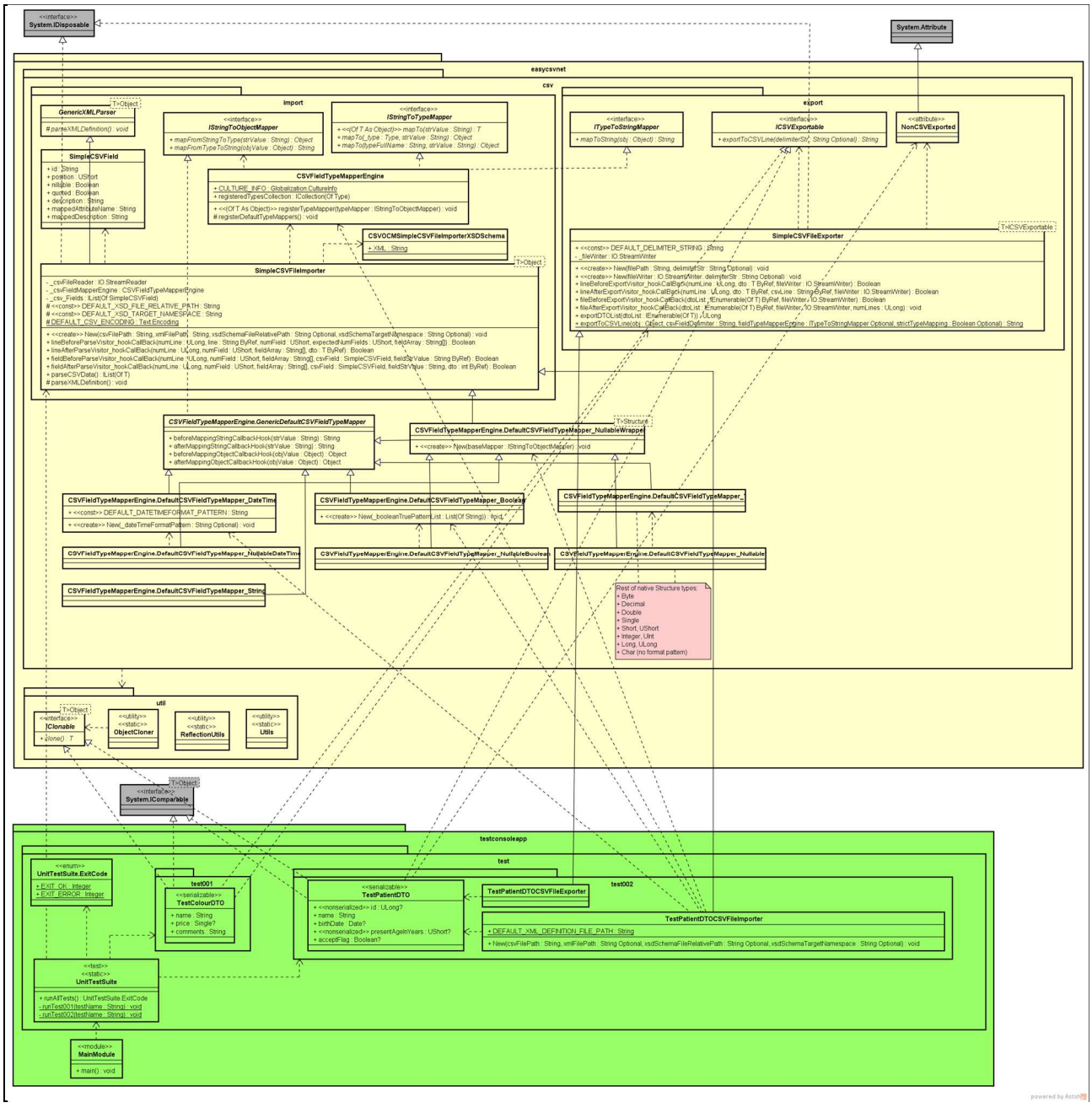

Figure 1 – Simplified UML class diagram of 'EasyCSVNet' solution

4. Importing a CSV file

Here we present the ordered sequence of steps to be followed in order to get a CSV input file parsed into a list of high-level DTO objects ready to be used in business layer logic. Obviously, we assume you reference `EasyCSVNet` library from the test application.

4.1. Creating a DTO object for the business layer

First of all, we define the target DTO object to which a single CSV input line should be mapped into.

In its simplest and shortest implementation, it should be a plain object with a private attribute and a public property attached, each one corresponding to every CSV field to be persisted in the object. Also, a default constructor (`New()` method without parameters) is needed (as a requirement for class instantiation by reflection).

Let's see the header section of the implementation of the DTO object corresponding to the first sample test, `testconsoleapp.test.test001.TestColourDTO`:

```
[...]
            <Serializable()>
            Public Class TestColourDTO
                Implements IComparable(Of TestColourDTO),
                           IClonable(Of TestColourDTO),
                           ICSVExportable

#Region "Private attributes"

                Private _name As String = ""
                Private _price As Single? = Nothing
                <SimpleCSVFileExporter.NonCSVExported()> Private _comments As String = Nothing

#End Region 'Private attributes

#Region "Public methods"

#Region "Constructors"

                Sub New()'Default empy-args constructor definition needed by CSV importer
                    Me.New(Nothing)
                End Sub

                Sub New(ByVal _name As String, Optional ByVal _price As Single? = Nothing
                        , Optional _comments As String = Nothing)
                    With Me
                        .name = _name
                        .price = _price
                        ._comments = _comments
                    End With
                End Sub

#End Region 'Constructors

#Region "Properties (getters & setters)"

                Public Property name() As String
                    Get
                        Return Me._name
                    End Get
                    Set(ByVal value As String)
                        Me._name = Utils.coalesce(value)
                    End Set
                End Property

                Public Property price() As Single?
                    Get
                        Return Me._price
                    End Get
                    Set(ByVal value As Single?)
                        Me._price = value
                    End Set
                End Property

                Public Property comments() As String
                    Get
                        Return Me._comments
                    End Get
                    Set(ByVal value As String)
                        Me._comments = Utils.coalesce(value)
                    End Set
                End Property

#End Region 'Properties (getters & setters)
[...]
```

As we can see, basic mandatory requirements over DTO objects to be CSV importable are minimally invasive: just private methods with getter and setter properties, and a default constructor method with no arguments declared.

Nevertheless, in the examples we can also see some extra elements, which are optional, but tend to be useful in every business object, so they have been implemented in addition to mandatory ones:

- `TestColourDTO` is annotated as a serializable class with the attribute `<Serializable()>`
- `TestColourDTO` implements interface `IComparable`, so it will be equipped with method `CompareTo()`
- `TestColourDTO` implements interface `easycsvnet.util.IClonable`, so it will be equipped with method `clone()`
- `TestColourDTO` implements interface `easycsvnet.csv.export.ICSVExportable`, so it will be equipped with method `exportToCSVLine()`, which will be used in CSV exportation. Moreover, attribute `TestColourDTO._comments` is marked with the attribute `<SimpleCSVFileExporter.NonCSVExported()>`, which means that it will be omitted by the default CSV export methods. We will see this point explained in chapter 5 below.
- Although not seen in the code fragment included here, `TestColourDTO` also overrides methods `Object.toString()` and `Object.Equals()` with custom behaviour.

Let's also note that using `Nullable` types for attributes when possible can be a useful practice, in accordance to CSV fields whose value can be missing in the input CSV file (in connection to `nillable` attribute in the XML definition file, as we will see soon).

Implementation of the DTO object corresponding to the second sample test, `testconsoleapp.test.test002.TestPatientDTO`, is totally analogous at this level to `TestColourDTO`.

So we are done with DTO objects! Although we don't need anything more for our unit test purposes (constrained only to CSV import and export), obviously we can treat DTOs pretty much like any other objects, and add as many attributes, properties and methods as needed besides the ones directly involved with CSV mapping described here.

4.2. Creating an XML definition file

Second phase: we need an XML file describing how mapping between a CSV line and the DTO object we have just created will take place, field by field (that is, attribute by attribute in the DTO object instance related to the CSV line).

XSD file `conf/csvOCM/schema/csvOCM_SimpleCSVFileImporter_schema.xsd` in `EasyCSVNet` library contains the formal description against which our XML definition file will be syntactically checked (although XSD schema content will actually be provided by `CSVOCMSimpleCSVFileImporterXSDSchema` class). Let's see a simple example in the XML definition file used in the first sample test, `conf/csvOCM/csvOCM_TestColourDTO_def.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE catalog
    PUBLIC "-//OASIS//DTD Entity Resolution XML Catalog V1.0//EN"
           "http://www.oasis-open.org/committees/entity/release/1.0/catalog.dtd">
<!--
         Object to CSV mapping (OCM) XML definition file
         @author alex.debuen@gmail.com
         @version 1.0.0.0 (build 20170824_150000)
-->
<catalog xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xmlns:xsd="http://www.w3.org/2001/XMLSchema"
         xmlns:csv="urn:csv">
         <csv:csv xsi:schemaLocation="urn:csv schema/csvOCM_SimpleCSVFileImporter_schema.xsd">
                  <metadata>
                           <author>alex.debuen@gmail.com</author>
                           <version>1.0.0.0 (build 20160824_150000)</version>
                           <revision>2017-08-24+01:00</revision>
                           <mappedobjectfqdn>testconsoleapp.test.test001.TestCoulourDTO</mappedobjectfqdn>
                           <description>XML schema for CSV file import (TestColourDTO)</description>
                  </metadata>
                  <data>
                           <settings>
                                    <titles>true</titles>
                                    <delimiter>|</delimiter>
                                    <stricttypecast>true</stricttypecast>
                                    <encoding>System.Text.Encoding.Default</encoding>
                           </settings>
                           <fields>
                                    <field id="NAME" position="0" nillable="false" quoted="false"
                                     description="Colour name"/>
                                    <field id="PRICE" position="1" nillable="false" quoted="false"
                                     description="Colour price"/>
                                    <field id="COMMENTS" position="2" nillable="true" quoted="false"
                                     description="Colour comments"/>
                           </fields>
                           <mapping>
                                    <map field="NAME" attribute="_name"
                                   description="Mapping NAME to TestColourDTO._name"/>
                                    <map field="PRICE" attribute="_price"
                                   description="Mapping PRICE to TestColourDTO._price"/>
                                    <map field="COMMENTS" attribute="_comments"
                                   description="Mapping COMMENTS to TestColourDTO._comments"/>
                           </mapping>
                  </data>
         </csv:csv>
</catalog>
```

Besides all the XML formal skeleton and a `<metadata>` purely descriptive section, all really interesting things happen – you guessed it! – inside the `<data>` section:

- A first `<settings>` subsection sets up the global parameters which will be applied to the whole file during the import process:
  - `titles` = *true* will skip first line, as it will be considered to not contain data, just a header or comment (typically describing field names)
  - Every `<delimiter>` tag will specify a character to be interpreted as field separator when appearing inside each line. There can be more than one. In pure CSV format, only comma (',') is allowed, but here we are not so restrictive.
  - `stricttypecast` = *true* always recommended, as it will force custom string-to-object casting in field to attribute binding during import process. This casting task will be carried out by an instance of class `easycsvnet.csv.import.CSVFieldTypeMapperEngine` (to be explained later), otherwise performed by native converts, which are not so much customizable and secure.
  - `<encoding>` tag (optional) refers to the whole CSV file character encoding, and must point out to an existing `System.Text.Encoding` native class.
- A second `<fields>` subsection specifies the fields to be read from CSV lines. Every `<field>` tag includes the following attributes:
  - `id` provides a unique label to identify every field
  - `position` (starting from 0) describes the field position inside the CSV line (from left to right)
  - `nillable` = *true* will admit empty values for that field, otherwise an exception will be thrown in runtime when a line with empty value in that position is reached by the parser
  - `quoted` = *true* will drop quotation chars (simple ' and double " quotation marks) from the field value (interpreted as a string, before casting)
  - `description` (optional) includes a textual free message as a comment to annotate the field tag, if needed
- A third `<mapping>` subsection specifies the rules to assign a target DTO attribute to every CSV field described in the previous `<fields>` subsection. Every rule is formed by a `<map>` tag with the following attributes:
  - `field` serves as a foreign key to the attribute `<field>.id` from the previous `<fields>` subsection, it indicates the CSV field targeted by the rule
  - `attribute` specifies the name of the private attribute of the target DTO object which will store the field value, once it has been casted to the type of the attribute (this type is discovered automatically by reflection, and casting is also performed by the parser in accordance to it)
  - `description` (optional) includes again a textual free message as a comment to annotate the mapping rule this time, if needed

We can see another example in the XML definition file `conf/csvOCM/csvOCM_TestPatientDTO_def.xml` used in the second sample test. It is practically analogous to the first example; let's only mention that in this second example, `<delimiter>` is set up to the standard comma value ',' and that `<titles>` is set to *false* (not because there is no header line before data, but because there are *more than one* indeed, in the form of comment lines, and therefore they must be skipped another way: overriding hook method `lineBeforeParseVisitor_hookCallBack()` in a custom inherited CSV importer object, as we will see in the next chapter).

4.3. Instantiating or extending a CSV importer object

Once we have created an XML definition file which will govern the Object-to-CSV Mapping (OCM) process (translation from CSV to DTO object list), we need an importer object which will take this file and the input CSV file (where the CSV data to be imported actually are) as arguments, and parses the second using the rules described in the first (once they have been syntactically revised according to the XSD underlying schema also automatically). Moreover, this importer object will need to take into account also the DTO target type, so as to end up the parsing process with a `List` of instances of this DTO type, one for each CSV line parsed.

Good news: there is a `easycsvnet.csv.import.SimpleCSVFileImporter` class ready and waiting for you to use it. Consider it as the 'default' CSV importer class.

Now: «To extend or not to extend, that is the question». I mean: should we create a new importer object which inherits from the default one and extends it to fit the custom needings of the particular DTO we are targeting, or can we use the default `SimpleCSVFileImporter` with little or even no changes at all? Well, sometimes both alternatives will be allowable, but luckily our dilemma will be easier to solve than Hamlet's in most cases.

Let's start with the easiest scenario: no need to extend. This is the first sample test case:

```
'Instantiate CSV importer
Dim csvFileImporter As SimpleCSVFileImporter(Of TestColourDTO) =
    New SimpleCSVFileImporter(Of TestColourDTO)(csvInputFilePath, definitionXMLFilePath)
```

Just a single line. As you can see, there is no need in this case to make further modifications over the importer's default behaviour. Let's only highlight the fact that `SimpleCSVFileImporter` depends explicitly on the target DTO type (`TestColourDTO` here) through a generic covariant type to be passed to the constructor.

Let's now see a bit more sophisticated case, illustrated by the second sample test:

```
'Instantiate CSV importer
Dim csvFileImporter As TestPatientDTOCSVFileImporter = New TestPatientDTOCSVFileImporter(csvInputFilePath)
```

Apparently just a single line too, but in this case we are instantiating a custom CSV importer class, `TestPatientDTOCSVFileImporter`, which inherits from default `SimpleCSVFileImporter` and encapsulates all custom behaviour:

```
Public Class TestPatientDTOCSVFileImporter
    Inherits SimpleCSVFileImporter(Of TestPatientDTO)

    Public Shared DEFAULT_XML_DEFINITION_FILE_PATH As String =
                                                    "conf/csvOCM/csvOCM_TestPatientDTO_def.xml"

    Public Sub New(ByVal csvFilePath As String, Optional ByVal xmlFilePath As String = Nothing
                   , Optional ByVal xsdSchemaFileRelativePath As String = DEFAULT_XSD_FILE_RELATIVE_PATH
                   , Optional ByVal xsdSchemaTargetNamespace As String = DEFAULT_XSD_TARGET_NAMESPACE)

        ' Provide a default value for XML definition file path
        MyBase.New(csvFilePath, Utils.coalesce(xmlFilePath, DEFAULT_XML_DEFINITION_FILE_PATH)
                   , xsdSchemaFileRelativePath, xsdSchemaTargetNamespace)

        ' Override some default CSV-field-to-type mapping actions by setting a customized
          CSVFieldTypeMapperEngine instance:
        ' Boolean and Boolean? types -> Should map 'Yes' literal to True
        Dim csvFieldTypeMapperEngine As New CSVFieldTypeMapperEngine()
        Dim csvFieldTypeMapper_Boolean As CSVFieldTypeMapperEngine.GenericDefaultCSVFieldTypeMapper
            = New CSVFieldTypeMapperEngine.DefaultCSVFieldTypeMapper_Boolean(New List(Of String)({"Yes"}))
        Dim csvFieldTypeMapper_NullableBoolean As CSVFieldTypeMapperEngine.GenericDefaultCSVFieldTypeMapper
            = New CSVFieldTypeMapperEngine.DefaultCSVFieldTypeMapper_NullableWrapper
                  (Of Boolean)(csvFieldTypeMapper_Boolean)
        ' DateTime and DateTime? types -> Should take 'yyyy-MM-dd' format pattern for dates
        Dim csvFieldTypeMapper_DateTime As CSVFieldTypeMapperEngine.GenericDefaultCSVFieldTypeMapper
            = New CSVFieldTypeMapperEngine.DefaultCSVFieldTypeMapper_DateTime("yyyy-MM-dd")
        Dim csvFieldTypeMapper_NullableDateTime As CSVFieldTypeMapperEngine.GenericDefaultCSVFieldTypeMapper
            = New CSVFieldTypeMapperEngine.DefaultCSVFieldTypeMapper_NullableWrapper
                 (Of DateTime)(csvFieldTypeMapper_DateTime)
        With Me.csvFieldTypeMapperEngine
            .registerTypeMapper(Of Boolean)(csvFieldTypeMapper_Boolean)
            .registerTypeMapper(Of Boolean?)(csvFieldTypeMapper_NullableBoolean)
            .registerTypeMapper(Of DateTime)(csvFieldTypeMapper_DateTime)
            .registerTypeMapper(Of DateTime?)(csvFieldTypeMapper_NullableDateTime)
        End With

    End Sub

    ' Custom restriction in default CSV line acceptance behavior
    Public Overrides Function lineBeforeParseVisitor_hookCallBack(ByVal numLine As ULong
                              , ByRef line As String, ByVal numField As UShort
                              , ByVal expectedNumFields As UShort, ByRef fieldArray As String()) As Boolean
        Const COMMENT_LINE_START_CHARACTER As Char = "#"c
        'Skip comment lines starting with '#'
        Return Not line.StartsWith(COMMENT_LINE_START_CHARACTER)
    End Function

    ' Custom restriction in default CSV field parsing
    Public Overrides Function fieldAfterParseVisitor_hookCallBack(ByVal numLine As ULong
                              , ByVal numFields As UShort, ByVal fieldArray As String()
                              , ByVal csvField As SimpleCSVField, ByVal fieldStrValue As String
                              , ByRef dto As TestPatientDTO) As Boolean
        Select Case csvField.mappedAttributeName
            Case "_birthDate"
                'Calculate present age just after having parsed field 'birthDate'
                dto.calculatePresentAgeInYears()
        End Select
        Return True 'No lines skipped
    End Function

    ' Custom restriction in default CSV line acceptance behavior
    Public Overrides Function lineAfterParseVisitor_hookCallBack(ByVal numLine As ULong
                              , ByVal numField As UShort, ByVal fieldArray As String()
                              , ByRef dto As TestPatientDTO) As Boolean
        With obj
            ' Skip lines with TestPatientDTO.acceptFlag <> True
            Return (Not IsNothing(.acceptFlag)) AndAlso .acceptFlag.HasValue AndAlso .acceptFlag.Value
        End With
    End Function

End Class
```

As we can see, there are several customizations needings which motivate the inheritance approach:

- We provide a default value for the XML definition file path, taken from the `app.config`, and we use it in an overloaded constructor which simplifies the instantiation with respect to the base class making the related parameter also optional
- We override some default CSV-field-to-type mapping actions by setting a customized `CSVFieldTypeMapperEngine` instance by means of calling `registerTypeMapper()` method for each modified base type:
    - In `Boolean` and `Boolean?` types, literal '*Yes*' must be mapped to `True`. This we achieve by instantiating `CSVFieldTypeMapperEngine.DefaultCSVFieldTypeMapper_Boolean` with a proper constructor yet ready for

us which luckily covers our needings (in case no suitable constructor were present, we shall need to implement the type mapper objects by ourselves, much like it is done inside `CSVFieldTypeMapperEngine`: extending from `CSVFieldTypeMapperEngine` or directly implementing `IStringToObjectMapper` interface). Note also that for `Nullable` types we can reuse base type mappers yet implemented with the help of `CSVFieldTypeMapperEngine.DefaultCSVFieldTypeMapper_NullableWrapper`, as we do here

    - In `DateTime` and `DateTime?` types, we must take '*yyyy-MM-dd*' as the format pattern for imported dates. This we achieve in an analogous manner as the former `Boolean` and `Boolean?` types, as we find again a suitable constructor ready to accept the desired format pattern as argument for the default `Datetime` mapper class `CSVFieldTypeMapperEngine.DefaultCSVFieldTypeMapper_Datetime`

- Now, we also need to interfere in default parsing process at different stages so as to customize the treatment of several specific CSV lines and fields by the parser. This we achieve by *hooking*, that is, overriding predefined callback functions (*hooks*) which in sum get executed by the parser at a specific moment in the parsing process, allowing child class to consult or even manipulate in some cases data involved in that step, and accepting or discarding the inclusion of CSV line being processed in the import job by returning `True` or `False` correspondingly as a `Boolean` callback result:
    - We skip comment lines starting with '#' by overriding `lineBeforeParseVisitor_hookCallBack()`, that is, by discarding unwanted lines just before they get parsed into fields
    - We calculate present age just after field `birthDate` has been parsed and casted by overriding `fieldAfterParseVisitor_hookCallBack()`
    - We skip lines with `TestPatientDTO.acceptFlag` <> `True` just after the whole CSV line has been parsed into a `TestPatientDTO` DTO by overriding `lineAfterParseVisitor_hookCallBack()`
    - Although we do not make us of it here in none of our two example tests, there is also a `fieldBeforeParseVisitor_hookCallBack()` method available which allows us to act on field values just after then have been splitted from CSV line as mere string chunks, and right before they become casted to their final DTO attribute type

  Please refer to UML class diagram in chapter 3 below (Figure 1) to get a complete description of these hook callback signatures (look at `SimpleCSVFileImporter` class description).

4.4. Parsing the CSV input file into a DTO list

Just execute `parseCSVData()` method on the CSV importer object, et voilà! You get the parsed DTO object list (technically and precisely, a polymorphic `IList` type) ready to be used in your business logic. See both example tests' code if you don't believe me, don't take my word for it ☺.

5. Exporting a CSV file

Here we present the ordered sequence of steps to be followed in order to get a list of `ICSVExportable` objects (this is actually the only restriction imposed to them to be exportable, as explained in chapter 5.a below) exported to a CSV file. As we said in the introduction, exporter classes are lighter than importer ones, as no XML support is needed to be provided to orchestrate object-to-csv line serialization, which is understood to be a simpler process than the reverse way importing covered in previous chapter 4. Obviously, we assume you reference `EasyCSVNet` library from the test application.

5.1. Implementing `ICSVExportable` interface

In order for an object to be processed by the default `SimpleCSVFileExporter` class or its descendants (to be explained below) and exported to a CSV line inside an output file, it must simply realize `easycsvnet.csv.export.ICSVExportable` interface, that requires implementing only a single method: `exportToCSVLine()`, which accepts the field delimiter character or string as an optional argument, and returns directly the string containing the CSV line to be exported to the final file. This serialization of the `ICSVExportable` object to a CSV line can be performed manually or with the help of the `SimpleCSVFileExporter.exportToCSVLine()` static method, as we will see in the next chapter.

5.2. Instantiating or extending a CSV exporter object

`SimpleCSVFileExporter` is the default CSV exporting class provided. In sum, it's just a simple class equipped with three groups of methods:

- An `exportDTOList()` method, which takes a list (properly said, an `IEnumerable` type) of objects which implement `ICSVExportable` interface described above, and iterates over the whole list exporting the serialized CSV line obtained by `exportToCSVLine()` call over each object element to the final output CSV file
- A bunch of *hook* callback methods which can be overridden to interfere in several stages of the exportation process, quite in the same maner as we did for importing process (explained at the end of chapter 4.c above):
    - `fileBeforeExportVisitor_hookCallBack()`: Called at the beginning of exportation, prior to first data line creation in output CSV file
    - `lineBeforeExportVisitor_hookCallBack()`: Called before each CSV data line exportation
    - `lineAfterExportVisitor_hookCallBack()`: Called after each CSV data line exportation (but not written yet in CSV output file)
    - `fileAfterExportVisitor_hookCallBack()`: Called at the end of all CSV lines exportation, but prior to file stream closing (so we are yet in time to write ending content to the output file, for example)
- A static `exportToCSVLine()` method with the only purpose to serve as a generic helper object-to-csv line serializer which automatically discovers base type of the object passed and applies the suitable object-to-string mapper from the default `CSVFieldTypeMapperEngine` implementation (we could also provide a customized version of this object as a '`fieldTypeMapperEngine`' argument to the method)

`SimpleCSVFileExporter` depends explicitly on the specific object to be exported through a covariant generic type.

Now again raises the same question we encountered in chapter 4.c above: should we extend generic `SimpleCSVFileExporter` class to fit our specific needings for every exportable object considered, or can we simply reuse the class as it comes? And again, decision is up to us in most cases; but my recommendation is not to extend if you can avoid doing it and, in case you can't and do need to extend, use *hook* callbacks. It should suffice for most purposes. For instance, in the first example test, we didn't need to extend `SimpleCSVFileExporter`, we use it as it comes:

```
        Dim csvFileExporter As SimpleCSVFileExporter(Of TestColourDTO) =
                            New SimpleCSVFileExporter(Of TestColourDTO)(csvOutputFilePath)
```

Let's note that in this case, we also use pre-built generic `SimpleCSVFileExporter.exportToCSVLine()` serializer method in `TestColourDTO` implementation (the same DTO imported object is used to be exported for simplicity):

```
#Region "ICSVExportable interface implementation"

                Public Function exportToCSVLine(Optional ByVal delimiterStr As String =
                                SimpleCSVFileExporter(Of TestColourDTO).DEFAULT_DELIMITER_STRING) As String
                       Implements ICSVExportable.exportToCSVLine
                    Return SimpleCSVFileExporter(Of TestColourDTO).exportToCSVLine(Me, delimiterStr)
                End Function

#End Region 'ICSVExportable interface implementation
```

On the other hand, in the second example test, we extend `SimpleCSVFileExporter`, as we need to introduce a simple custom variation of the default exporting behaviour at the very beginning of the file creation: we want to insert a first non-data comment line. So we implement child class `TestPatientDTOCSVFileExporter` and override `fileBeforeExportVisitor_hookCallBack()` method:

```
            Public Class TestPatientDTOCSVFileExporter
                Inherits SimpleCSVFileExporter(Of TestPatientDTO)

#Region "Public methods"

#Region "Constructors"

                Public Sub New(ByVal filePath As String
                               , Optional ByVal delimiterStr As String = DEFAULT_DELIMITER_STRING)
                    MyBase.New(filePath, delimiterStr)
                End Sub

                Public Sub New(ByVal fileWriter As System.IO.StreamWriter
                               , Optional ByVal delimiterStr As String = DEFAULT_DELIMITER_STRING)
                    MyBase.New(fileWriter, delimiterStr)
                End Sub

#End Region 'Constructors

#Region "Overriden methods from SimpleCSVFileExporter"

                ' Custom restriction in default CSV file creation, at the beginning of the file
                Public Overrides Function fileBeforeExportVisitor_hookCallBack(
                                           ByRef dtoList As IEnumerable(Of TestPatientDTO)
                                           , fileWriter As IO.StreamWriter) As Boolean
                    ' Insert a first non-data comment line
                    fileWriter.WriteLine("# Test 2 - Example : CSV file with a list of patients exported")
                    Return True
                End Function

#End Region 'Overriden methods from SimpleCSVFileExporter

#End Region 'Public methods

            End Class
```

By the way, let's note that in this case, instead of using the pre-built generic serializer method `SimpleCSVFileExporter.exportToCSVLine()` as in the previous example inside `TestPatientDTO.exportToCSVLine()` method, we chose to implement the serialization process manually on our own (it's very simple indeed). We can see it inside `TestPatientDTO` class implementation:

```
#Region "ICSVExportable interface implementation"

                Public Function exportToCSVLine(Optional ByVal delimiterStr As String =
                                SimpleCSVFileExporter(Of TestPatientDTO).DEFAULT_DELIMITER_STRING) As String
                       Implements ICSVExportable.exportToCSVLine
                    Const EXPORT_DATE_FORMAT_PATTERN As String = "dd/MM/yyyy"
                    With Me
                        Return String.Join(delimiterStr, {Utils.toStr(.id), Utils.toStr(.name),
                                   Utils.toStr(.birthDate, EXPORT_DATE_FORMAT_PATTERN), Utils.toStr(.acceptFlag)})
                    End With
                End Function

#End Region 'ICSVExportable interface implementation
```

We could have used again the generic serializer method given by `SimpleCSVFileExporter.exportToCSVLine()`, but passing it a custom type mapper object with a new custom `IStringToObjectMapper` instance registered for the types `DateTime` and `DateTime?` to export dates in the desired format, for example… Nevertheless, in this case maybe a manual approach like the given above is simpler and shorter; a choice always to be considered.

5.3. CSV Output

All heavy work concerning CSV export is already implemented at this point. Just call `exportDTOList()` method over the exporter object, and output CSV file will be saved. You will get in return the final number of CSV lines exported to the file, by the way.

6. Credits and license

EasyCSVNet 1.0.1 was created by **Àlex de Buen Lapena** (alex.debuen@gmail.com), and was released freely to community on **September 11th 2017** via uploading to **GitHub** repository '**EasyCSVNet**' (https://github.com/ADeBuen/EasyCSVNet , branch **1.0.1**) under **GNU General public license 3.0**. Feel free to check out code and use it in your projects if you like it. You will find the whole VB.Net solution source (both *EasyCSVNet* library and *TestConsoleApp*) explained in this document there.

EasyCSVNet 1.0.1 library was also uploaded in compiled package form (**`easycsvnet.1.0.1.nupkg`**) to **NuGet**, the free .Net repository. You can find it there at https://www.nuget.org/packages/easycsvnet, and include it in your projects in the usual way via NuGet PM or .Net CLI as explained in chapter 2 above. NET Framework version 4.6.1+ is required.

Please refer all kind of suggestions and code improvements to my e-mail.

Thanks for reading this brief HowTo guide, and hope you enjoy EasyCSVNet!