

University of West Bohemia in Pilsen
Faculty of Applied Sciences
Department of Computer Science and Engineering

MVE - 2

Commenting of MVE2 library

Author: **Petr Dvorak**
Date: December 15 2005

1. Introduction

The MVE-2 provides a system how to create self-describing library of modules and data structures. This document describes several rules, possibilities of attributes and comment writing to fit *MMDoc* tool.

2. Basic terms

2.1. Attributes

The language C# allows adding of further information associated with classes, methods, return values etc. into the compiled file. So-called *attributes* are the information inserted into *metadata* by compiler. However the term attributes generally denotes the member variables of the classes. Both of terms will be distinguished in this text.

The attribute is added by command:

```
[Name_Of_Attribute(Parameter1, Parameter2, Optional_parameter = value)]  
Element_Of_Source_Code //definition of a class, method, member variable...  
...
```

E.g.:

```
[ModuleInfo("Miroslav F.", "Module is useful for cycles", IconName = „Delay.ico“)]  
public class DelayModule : Module  
{  
    ...  
}
```

where `ModuleInfo` is the name of the attribute, `IconName` is an optional parameter and `DelayModule` is an element of source code, to which the attribute belongs.

3. Attributes of library

They can be found and changed in the file named `AssemblyInfo.cs`. This file is automatically created by Visual Studio. It's useful to define at least:

- `AssemblyTitleAttribute(string)` – name of the library
- `AssemblyDescriptionAttribute(string)` – description of library
- `AssemblyCompanyAttribute(string)` – a company, which own a copyright
- `AssemblyCopyrightAttribute(string)` – copyrights

For example, see following lines taken from *MVECore* assembly:

```
[assembly: AssemblyTitle("Modular Visualization Environment 2 - Core")]  
[assembly: AssemblyDescription("Assembly contains pivotal classes of MVE2.")]  
[assembly: AssemblyCompany("University of West Bohemia in Pilsen, Czech Republic")]  
[assembly: AssemblyCopyright("Milan Frank 2003 - 2004")]
```

4. Attributes of modules

4.1 About writing attributes

All the additional information is saved in a source code by two ways:

- attributes
- comments of source code

The attributes are optional, but it's highly recommended to use them. They are used in runtime (tooltips, list of modules, detail information about module etc.) as well as in generated documentation (using by `MMDoc` tool). Therefore they should be brief and apposite. Character constants can be used to a simple string formatting (e.g. in the description of function of a module or port, which can be longer). Most frequent is `\n` – new line, but naturally also a rest of the character constants can be used.

It can be added two types of attributes to the modules:

- `ModuleInfoAttribute`
- `PortInfoAttribute`

.NET standard attributes can be added to the properties of modules:

- `DescriptionAttribute`
- `BrowsableAttribute`

These are the alternative way to document the function of module instead of `InvokeSetup` method. (see section *Module comments*). They describes all items that can be set in setup dialog of module which contains .NET component `PropertyGrid`.

Note: A word attribute can be omitted. A compiler adds it automatically.

4.2 *ModuleInfoAttribute*

`ModuleInfoAttribute` provides information about a module. Attribute can be inserted only once before a definition of a class.

It has 2 mandatory parameters (strings):

- *author* of a module
- a brief *description* of a function of a module

3 optional parameters (strings):

`IconName`

- A name of an icon in the compiled library, which represents the module (adding icon – see section 4.2.1.).
- If no icon is entered, the default icon from core is used.

Assembled

- The date of the version of the module (it's entered in format RRRR-MM-DD ... internally saved as *.NET* type `DateTime`).
- If no date is entered, the date of the compilation is used.

Category

- It indicates a group, to which the module belongs.
- E.g.: a loader module (`XmlLoader`), a system module (`DelayModule`), renderer module (`Renderer`).
- The module is classed into the group in the list of modules according to this attribute (*not implemented yet – now modules are shown by namespaces*).
- The default value is unspecified.

E.g.:

```
[ModuleInfo("Miroslav F.", "Module is used for cycles", IconName =
"Delay.ico", Assembled = "2004-06-20", Category = "System")]
```

4.2.1. Adding custom icon

Several steps must be completed to use custom icon. First it must be added to the project in *Solution Explorer*. Next the property of this icon - *Build properties* - must be changed to the value *Embedded Resource*. It can be found after clicking right mouse button on the icon and choosing *Properties*. It ensures that the icon will be compiled with the library. At the end set `IconName` to the name of the icon, when you define `ModuleInfo` attribute.

4.3 PortInfoAttribute

`PortInfoAttribute` provides information about one particular port of a module. It is allowed to assign several of these attributes to the module according to a number of module ports.

It has 2 obliged parameters:

- *name* of the port – unique in a scope of the one module, must be equal to name used during adding port in source code of module
- function of the port – a brief *description* of the port function

1 optional parameter:

`IconName`

- A name of an icon in a compiled library, which represents the port (adding see above to section 4.2.1.).
- If no icon is entered, the default icon from core is used.
- It's recommended to set user icon in very special cases only.

E.g.:

```
[PortInfo("Input", "Default value")]
[PortInfo("Output", "Delayed data")]
[PortInfo("Initial", "Initial input", IconName="init.ico")]
```

5. Attributes of data structures

5.1 About writing attributes

Same rules govern as in modules (see clause 4.1).

5.2 DataObjectAttribute

`DataObjectAttribute` provides the information about *MVE2* data structure. Only one attribute can be assigned to the class or interface.

It has 1 mandatory parameter:

- *description* – a brief description of the data structure

E.g.:

```
[DataObject("Representation of the vector in 2D")]
```

6. Comments of modules and data structures

6.1 About writing comments

Comments should provide a detailed and exhaustive description of parts of a module and data structure for their users. They are the standard documentation comments (`///).`

It's allowed to use also HTML tags. Most frequently used will be:

```
<p>Paragraph</p>
<br /> ... new line
<b>a bold text</b>
<i>an italic</i>
<a href="adresa">Link</a>

```

Attention: The commentaries must comply *XML* syntax, but also *HTML*. Therefore there are space and backslash at the end of the tags `
` `<a />` ``. In addition characters „<“ and „>“ mustn't be used in other meaning than in tag definition. *Visual Studio* omits such “*not well-formatted*” comments in generated *XML* file. They have to be replaced with `<`; and `>`; strings.

Only some of the comments are important for generating of documentation by `mmdoc.exe` program. They are:

- `<summary>` assigned to classes
- `<summary>` assigned to public and protected methods, constructors, events, properties and member variables of data structures
- `<param>` assigned to public and protected methods and constructors

- `<returns>` assigned to public and protected methods

6.2. *Module comments*

- Module class should be commented. Comments should explain:
 - a purpose of the module
 - potential use
 - limitation given by computation
 - limitation given by knowledge of application domain
 - what modules can current module cooperate with
 - etc.
- commentary associated with method `InvokeSetup` (builds module setup dialog)
 - it should describe the module setup dialog for module users
 - it is recommended to append a screenshot via tag ``
- commentaries associated with property with `Description` attribute eventually with `Browsable` attribute set to `True` value

6.3. *Data structure comments*

- Data structure class should be commented. Comments should explain:
 - what data does it represent
 - description
 - some pictures are useful
 - limitation of data types
 - limitation given by knowledge of application domain
 - what definitions or theorems does it answer (e.g. in the geometric modeling Euler's theorem)
 - what data structures does current data structure relation with
 - etc.
- commentaries of particular public method, property and member variables
 - they're mainly provided for advanced users – module's programmers
 - users search here functions of data structures, which use in their modules

7. Examples

7.1. *AssemblyInfo.cs*

```
using System.Reflection;
using System.Runtime.CompilerServices;

// General Information about an assembly
[assembly: AssemblyTitle("Modular Virtual Environment 2 - Example")]
[assembly: AssemblyDescription("Example of MVE2 library")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("University of West Bohemia in Pilsen, Czech Republic")]
[assembly: AssemblyProduct("Examples")]
[assembly: AssemblyCopyright("Milan Frank 2004")]
[assembly: AssemblyTrademark("ZCU-MVE2.NET")]
```

```
[assembly: AssemblyCulture("")]

// Version information for an assembly consists of the following four
[assembly: AssemblyVersion("1.0.*")]

// In order to sign your assembly you must specify a key to use.
[assembly: AssemblyDelaySign(false)]
[assembly: AssemblyKeyFile("")]
[assembly: AssemblyKeyName("")]
```

Information used by mmdoc and showed in the resulting documentation is emphasized by bold font.

Overview of library in the generated documentation:



MVE2 Module Documentation

Examples

Title: *Modular Virtual Environment 2 - Example*
 Description: *Example of MVE2 library.*
 Company: *University of West Bohemia in Pilsen, Czech Republic*
 Copyright: © 2004 - 2005 *University of West Bohemia, Czech Republic*
 Version: *1.0.1936.19787*
 Runtime version: *1.1.4322*

Modules

Examples.AllEventSumator	Sample module. It sums two input numbers. Reaction to all events is implemented.
Examples.Convolution	Module performs a convolution of a image.
Examples.GenerateGraph	Module creates 100 samples of connected function in <0, 10> interval.
Examples.PromennePorty	Sample module. Example of varying ports.
Examples.NumberSource	Sample module. Source of one random scalar number.
Examples.Sinus	Module calculates sinus of given number.
Examples.Sumator	Sample module. It sums two input numbers.

Data structures

Examples.ScalarNumber	One scalar number (double)
---------------------------------------	----------------------------

7.2. Example of module

```
using System;
using Zcu.Mve.Core;

namespace Examples
{
    /// <summary>
    /// Source of one random scalar number.
    /// It's double-precision floating point number in the interval &lt;0.0, 1.0>.
    /// Possibly you can define own output number.

```

```

/// </summary>

[ModuleInfo("Milan Frank", "Sample module. Source of one random scalar number.",
  IconName = "numbersource.ico", Assembled = "2004-06-13", Category = "Example")]

[PortInfo("Output", "Random or user defined scalar number.")]

public class NumberSource : Zcu.Mve.Core.Module
{
    /// <summary>
    /// Create ports.
    /// </summary>
    public NumberSource()
    {
        AddOutPort("Output", typeof (Examples.ScalarNumber));
    }

    ...
    //class body
    ...

    /// <summary>
    /// You can choose from two modes of function of module:
    /// <ol>
    /// <li><p>Generating of random number:</p>
    /// <p></p></li>
    /// <li><p>Generating of defined number (2.0 in this case):</p>
    /// <p></p></li>
    /// </ol>
    /// </summary>
    /// <returns>User control.</returns>
    public override ModuleSetup InvokeSetup()
    {
        //some code
    }
} // NumberSource
} // namespace

```

Important parts are emphasized by bold font.


There is the detailed description of the module between tag `<summary>` at the beginning. Notice the technique of writing down of the interval `<0.0, 1.0>`.

The attribute `ModuleInfoAttribute` is below commentary. 3 optional parameters (`IconName`, `Assembled` and `Category`) are defined in it. Further there is the attribute `PortInfoAttribute`. Notice that name of the port (first parameter) is same as name of port entered during port creation in the constructor.

Next there is the definition of class of module. It is inherited from `Zcu.Mve.Core.Module`. Then there is a source code, which defines function of module.

Finally there is the method `InvokeSetup`. The comment between tags `<summary>` describes possibilities of configuration of the module through the settings dialog. *HTML* and inserting of the pictures from actual directory are used here. Notice the closing of the tag `` by space and backslash.

Overview of module in the generated documentation:



MVE2 Module Documentation

Examples.NumberSource

[Index](#) | [Overview](#) | [Settings](#) |

Sample module. Source of one random scalar number. A double-precision floating point number greater than or equal to 0.0, and less than 1.0. Possibly you can define own output number.

Ports:


- *Output* - Random or user defined scalar number.

Details:


Author: *Milan Frank*
Type of module: *Example*
Version: *13.6.2004*

Created by MMDoc 20.4.2005 12:00:08

Module settings (InvokeSetup comment) in the generated documentation:



MVE2 Module Documentation

[Examples](#).NumberSource 

[Index](#) | [Overview](#) | [Settings](#) |

You can choose from two modes of function of module:

1. Generating of random number:



2. Generating of defined number (2.0 in this case):



Created by MMDoc 20.4.2005 12:00:08

7.3. Example of data structure

```
using System;
using Zcu.Mve.Core;

namespace Examples
{
    /// <summary>
    /// It represents one scalar number (accuracy as double type in .NET).
    /// Can be shared by MVE2 modules.
    /// </summary>
}
```

```

[DataObject("One scalar number (double)")]

public class ScalarNumber : DataObject
{
    /// <summary>
    /// Value carried by this data object.
    /// </summary>
    private double val;

    /// <summary>
    /// Constructor with default settings.
    /// </summary>
    public ScalarNumber()
    {
        val = 0.0f;
    }

    ...
    //class body
    ...

    /// <summary>
    /// Gets/Sets scalar vaule of this object.
    /// </summary>
    public double Val
    {
        get
        {
            return val;
        }
        set
        {
            val = value;
        }
    }

    /// <summary>
    /// Writes data object into XML file.
    /// </summary>
    /// <param name="xmlTextWriter">Xml stream.</param>
    public override void WriteData(System.Xml.XmlTextWriter xmlTextWriter)
    {
        xmlTextWriter.WriteString(" " + this.Val);
    } // WriteData()

    /// <summary>
    /// User defined deep copy of this data object.
    /// </summary>
    /// <returns></returns>
    public override IDataObject DeepCopy()
    {
        ScalarNumber ret = new ScalarNumber();
        ret.Val = this.Val;

        return ret;
    }

    /// <summary>
    /// Report about data structure. It's used for example by ConsolePrinter.
    /// </summary>
    /// <returns></returns>
    public override string ToString()
    {
        return this.val.ToString(Globals.nfi);
    }

} // ScalarNumber
} // namespace

```

Important parts are emphasized by bold font.

There is a detailed description of *MVE2* data structure at the beginning. Further there is a little bit shorter attribute `DataObjectAttribute`.

Bellow is the definition of class. In this case it is inherited from data type `DataObject`, which implements necessary interface `IDataObject`.

Finally there are definitions of member variables, constructor, methods (`DeepCopy()`, `ToString()`) and properties (`Val`).

Overview of data structure in the generated documentation:



MVE2 Module Documentation

Examples.ScalarNumber

[Index](#) | [Overview](#) | [Members](#)

Sample data class. It represents one scalar number (accuracy as double type in .NET). Can be shared by modules.

Inheritance:

```
System.Object
  Zcu.Mve.Core.DataObject
    Examples.ScalarNumber
```

Implements:

```
Zcu.Mve.Core.IDataObject
```

Created by MMDoc 20.4.2005 12:00:08

Members of data structure in the generated documentation:



MVE2 Module Documentation

Examples.ScalarNumber

[Index](#) | [Overview](#) | [Members](#)

Constructors

```
ScalarNumber ( )
```

Constructor wit default setting.

Public methods

```
virtual System.Void WriteData ( System.Xml.XmlTextWriter  
xmlTextWriter ) ;
```

Writes data object into XML file.

- `xmlTextWriter`
Datovy tok pro zapis dat do Xml souboru.

```
virtual System.String ToString ( ) ;
```

Report of the content of data structure. It's used by ConsolePrinter for example.

```
virtual System.Boolean CheckConsistence ( ) ;
```

User defined check of consistence of this data object. Warnings can be written to Console as well as reasons of inconsistency.

Returns:

True if it's consistent, false otherwise.

```
virtual System.Void ReadData ( System.Xml.XmlTextReader  
xmlTextReader ) ;
```

Reads data from XML file into data object.

- `xmlTextReader`
Datovy tok pro cteni dat z Xml souboru.

```
virtual Zcu.Mve.Core.IDataObject DeepCopy ( ) ;
```

User defined deep copy of this data object.

Public properties

```
System.Double Val { get;set; }
```

Gets/Sets scalar value of this object.