Reference Card for OOD Notation

Gebhard Greiter, 2011

Object oriented Design Notation (OOD) is a quite simple formal language to specify code structure: classes and interfaces, and also their *public* properties, methods, and constructors.

Private or *protected* parts of a class cannot be specified in OOD (though the code generator you use may add some of them). This will keep the design sufficiently abstract.

Design in OOD is

- easy to understand
- easy to maintain
- and precise enough to be the basis for code generation.

The first line of such design is to specify a Code Root (and thereby also the target implementation language: C# or Java).

The rest is a sequence of specifications defining Classes or Interfaces (in the sense of C# or Java) and the Name Spaces resp. Packages containing them.

The specification of a Class or Interface

- must start with a hyphen in the first column of the file followed by two spaces and again at least three hyphens
- It will end with the next empty line.

Every other text in the file is understood as comment to be ignored by the code generator.

Naming conventions may be enforced by the code generator.

Default code generator is **oodes.exe** contained in

http://greiterweb.de/spw/zu_OOD/oodes.zip

A sample specification and the code derived from it by the default code generator can be found here:

http://greiterweb.de/spw/zu_OOD/BR.ood.txt

http://greiterweb.de/spw/zu_OOD/BR.cs.txt

There are two templates:

- one for specifying Classes,
- and a similar one for specifying Interfaces.

Note: Cardinalities shown in the templates below are not to become part of your specification. They are in the template only to tell you how often the corresponding line may occur in a class or interface specification:

`0..1` means: You may have at most one such line.
`0..` means: You may have any number of such lines.

Names in OOD are always shown with a prefix C_, I_, t_, m_, p_ specifying the concept the name is referring to.

A name without such a prefix is called a ***naked*** name.

Each type is either a Class, an Interface, or a simple type. The name of a Class or Interface has to start with a capital letter, the name of a simple type is not allowed to start with a capital letter (Java and C# naming convention).

`For each type x there is also a type x[] (a value of type x[] is a sequence x[0..] of variables x[n] of type x, n any non-negative index).`

`x[1..] is the same as x[] but stressing the fact that, from the application's point of view, valid values of this array must not be empty.`

The template for specifying a Class is:

```
-   ----        C_x1
                  |
    0..1          extends     C_x2
    0..           offers      I_x3
    0..           xt          t_
    0..           xt          t_xN
    0..           xm          m_
    0..           xm          m_xN
    0..           xp          p_xN
```

such that

`x1` is the name of the class to be specified,
`x2` is the name of the unique direct superlass,
`x3` is the name of an interface to be implemented by the class,
`xp` is the name of a property (in the sense of C#) of type *xN*,
`xm` is the name of a class member that is a non-static method,
`xt` is the name of a class member that is a static method (a tool).

For methods that are to return a value of type *xN*, write *m_xN* instead of *m_* (here *xN* the naked name of the return type).

If *xm* = "`constructor`" , replace *m_* by *c_* if you want a standard implementation of the constructor.

Methods *m_* or *m_xN* may have a parameter list:

```
                (
        0..     1, parType  parName       // 1 = in
        0..     2, parType  parName       // 2 = out
        0..     3, parType  parName       // 3 = inout
                )
```

where `parType` is the naked name of a class, an interface, or a simple type (such as e.g. `int`). It may be followed by `[]` and is then an array type.

Methods *m_xN* without a parameter could be implemented as properties without a public setter (only code implementing the class itself should be able set or update their value). The default generator **oodes.exe** however is implementing them as methods.

The template for specifying an Interface is:

```
-   ----        I_x3
                  |
    0..1          extends       I_x4
    0..           xt            t_
    0..           xt            t_xN
    0..           xm            m_
    0..           xm            m_xN
    0..           xp            p_xN
```

such that

- `x3` is the name of the interface specified,
- `x4` is the name of the interface that is extended by Interface *x3*,
- `xp` is the name of a property (in the sense of C#) of type *xN*,
- `xm` is the name of a class member that is a non-static method,
- `xt` is the name of a class member that is a static method (a tool).

Note: Interfaces cannot have constructors.

Here is a sample specification showing also how to specify Name Spaces resp. Packages:

```
-  codeRoot    ./oo/cs/

       Code generator oodes.exe will transform the specification below to

              ./oo/cs/*  and (as far a required) to  ./oo/*.xsd


       If the codeRoot would end in /java/, oodes.exe could create java/*
       instead of cs/*

       But: So far only cs/* and *.XSD are supported.



-  pack,XML    BMF
   ------------------------------------------------------------------------
       |
       This is telling the code generator that we also need an XSD for the
       classes specified in pack BMF or any subpack:



-  ----        C_AbstractBMFSet
                 |
                 offers                   I_BMF.GUID


-  ----        I_GUID                     // Globally unique ID
                 |
                 constructor              c_(
                                            1, Instance_UID    instance_UID
                                            1, Generation_UID  generation_UID
                                            )
                 |
                 instance                 m_Instance_UID
                 generation               m_Generation_UID


-  pack        BMF.Programme
   ------------------------------------------------------------------------

-  use         BMF



-  ----        C_AbstractProductionElement
                 |
                 extends                  C_AbstractBMFSet
                 offers                   I_App.GetData.GetPEData
                 |
                 archiveIdentifier        p_String
                 rights                   p_Rights[]
```

You see: Sections or lines not meant as comment are to start with a minus sign in column 1 of the file followed by two spaces. Sections specifying a Class or an Interface must not contain empty lines.