

OOP features in MDF5

This document describes the implementation of OOP features into our development system.

How does our implementation fit the OOP concepts?

Abstraction

The *abstraction* could be achieved by two different ways in our system. The first one is to use *abstract classes* - classes which are marked abstract because they does not implement all of their methods, just defines them. The second way is to use *interfaces*, which does not implement any code at all. They only define what attributes and what methods will be public - available to use from the “outside world” (code which is not part of the class).

```
<!-- abstract class, operations are implemented later -->
<class id="Shape" abstract="1">
  <attribute id="position"> ... </attribute>
  <method id="Draw" type=" ... " />
  <method id="getArea" type=" ... " />
  <method id="getCircumference" type=" ... " />
</class>

<!-- an interface, the classes/object must implement the methods -->
<interface id="Geometry">
  <method id="getArea" type=" ... " />
  <method id="getCircumference" type=" ... " />
</interface>
```

Encapsulation

Encapsulation as grouping data and code is possible in both classes and interfaces because not only the classes can define attributes - the interfaces can define them too (in contrast to few current programming languages, which can not define attributes in interfaces).

```
<!-- data and code together -->
<class id="Complex">
  <attribute id="re"> ... </attribute>
  <attribute id="im"> ... </attribute>
  <method id="abs" type=" ... " />
  <method id="phi" type=" ... " />
</class>
```

Polymorphism

In the meaning of different behavior - the *polymorphism* can be achieved by the actual use of interfaces, so the real functionality is hidden behind that interface. Another solution to provide a possibility to change the behavior in runtime is to use *events* and handlers, which can be set up and changed when the program is running.

```
<!-- i/o interface -->
<interface id="Stream">
  <method id="read" type=" ... " />
  <method id="write" type=" ... " />
</interface>
```

Inheritance

A simple *inheritance* is that in both classes and interfaces we can inherit a definition from another class or interface (actually, this is rather *extension* than *inheritance* because a class/interface is an extension of another one). When there's a need for inheritance from multiple sources, then one can use interfaces to inherit functionality definitions. Real inheritance of the code is not possible (at least not yet).

```
<!-- inheritance -->
<class id="Button" extends="Component">
  :
</class>
```

Building blocks

We have these basic building blocks for OOP:

- definitions
- references

Both definitions and references are only of two kinds - dealing with:

- classes
- interfaces

Interfaces

The interfaces are composed of:

- constants
- attributes
- method definitions
- event definitions
- implementation hints

Classes

The classes can contain all of the above (that's what the interfaces contain) and in addition to that, they could and often does contain also:

- implementation
 - of methods (defined in some class)
 - of interface itself (in meaning - without any method implementation)
 - of interface methods
 - of special methods (such as constructor, destructor, get, set and clone)
- can use the visibility of attributes and methods (interfaces have to contain only public attributes and definitions of public methods)

Content of classes and interfaces

These parts have an identifier, which is usually the *id* attribute, but the interface and method implementation is identified in different way. These components use the *interface* and *method* attributes, which could be understand also in a way that they do not define the component - they are refer to something what is already defined elsewhere.

Definitions

Constants have a type and a constant value. Constants should be atomic values because structured ones are not well supported in all languages (instead of a structured constant, make an initialized static attribute).

Attributes have a *visibility* (private, protected, public), *access mode* (readonly, readwrite) and a *type* definition. Optionally they could have an initial value and/or they could be marked as *static* which means that there is only one instance of the attribute for the whole class, and if is public then it could be accessed without any object (instance of that class)

Method definitions share some features with attributes: *visibility*, *static*, *id*. They have a type, but it has to be realized as a type reference (mostly to force users to make the code reusable).

Implementations

The implementation refers to the interface or method which is to be implemented, and when it is a method implementation, it actually contains the code of that method. Also note, that the method implementation could be found inside interface implementation, so it refers to the method of that interface, but when located outside, in the class, then it refers to the method defined in the class, or its base classes.

Example class

```
<!-- example class -->
<class id="Disk" extends="Shape">
```

Constant definition

```
<!-- needed in area calculation -->
<constant id="PI">
  <float />
  <const:float>3.1415926</const:float>
</constant>
```

Attribute definitions

```
<!-- size specification of the disk -->
<attribute id="inner"><float /></attribute>
<attribute id="outer"><float /></attribute>
```

Method definitions

```
<!-- private methods for calculating areas -->
<method id="innerArea" type="function/float" visibility="private" />
<method id="outerArea" type="function/float" visibility="private" />
```

Interface implementation

```
<!-- for calculation of area -->
<implements interface="Geometry">
```

Implementation of a method from interface

```
<!-- are of the disk, without the hole -->
<implementation method="getArea">
  :
  : // return this.outerArea() - this.innerArea();
  :
</implementation>

</implements>
```

Implementation of a method from the class

```
<!-- calculate the area of the hole -->
<implementation method="innerArea">
  :
  : // return self::PI * sqr( this.inner );
  :
</implementation>

<!-- calculate the area of the whole disk -->
<implementation method="outerArea">
  :
  : // return self::PI * sqr( this.outer );
  :
</implementation>
```

Implementation of a special method - the constructor

```
<magic>
  <constructor type="Disk/constructor">
    :
    : // this.inner = inner;
    : // this.outer = outer;
    :
  </constructor>
</magic>

</class>
```