xdata

Release 0.7.3

Erwan Adam

September 14, 2009

DEN/DM2S/SFME/LGLS
CEA Saclay, 91191 Gif/Yvette
Email: erwan.adam@cea.fr

Abstract

The python language (www.python.org) is an object oriented high level language used in a lot of projects. Many useful tools provide a *python wrapping* to interact with the tool via python (pyqt, vtk wrapping python, omniorb python, ...).

One particuliar feature of python is its lack of type checking. It means that you can write a function in python and call it with an integer, a float, a string or a more evoluate structure. The work of type checking is on developper side.

In this context, the python `xdata` module provide ways to declare types in python classes. The type declared for an attribute can be very restrictive (for example, an integer with a min and a max value) or not restrictive at all (for instance, the attribute can be an integer or a list or an instance of another class).

Once a set of classes is written using the `xdata` grammar described in details in this documentation, a simple gui (graphical user interface) written in pyqt is provided. It allows to interact graphically with the set of classes : instance creations, tooltips declarations (for documentation), popup declarations on each object, ...

More-over, when a class if fully typed using the `xdata` grammar and when a version of salome (www.salome-platform.org) is installed on the system at compilation stage, a salome component is automatically generated and fully usable without more work from the developper. The main `xdata` jobs is to write the idl (interface description language) files necessary for salome and to branch the gui dialog boxes from the pyqt `xdata` gui tool to the salome desktop.

There are two templates provided with `xdata` distribution (they are located in share/xdata/templates after installation stage). The easiest way to begin with `xdata` is to copy one of this templates locally and modify the source files to fit your developpements.

If you have already written classes, `xdata` module support multiple inheritence (as python do) and in that case, the only job you have to do is to write wrapping classes to declare what are the types of the attributes and methods.

If you have already written dialog boxes, `xdata` module provide a way to replace the default `xdata` dialog boxes by your own ones. It is sufficiant in that case to declare a `createDialog` class method in your class. This method of class will be called at each dialog box creation.

Any questions, requests, bugs should be sent to erwan.adam@cea.fr.

CONTENTS

CHAPTER

ONE

XTypes

exception XValueError()
The exception XValueError is raised when the check of a value is false.

class XType()
This class is abstract, don't use it directly. The XType instances are callable with a value to be tested. If the value is ok, returns it (converted in the good type if necessary). Raise XValueError if check is false.

1.1 XNone

class XNone()
The XNone check if a value can be understood as None. The following code shows the possibilities of XNone

```
# --
# Copyright (C) CEA, EDF
# Author : Erwan ADAM (CEA)
# --

import unittest

from xdata import *

class XNoneTestCase(unittest.TestCase):
    def test(self):
        x = XNone()
        self.failUnlessEqual(x(None), None)
        self.failUnlessEqual(x("None"), None)
        self.failUnlessRaises(XValueError, x, "titi")
        return
    pass

if __name__ == '__main__':
    unittest.main()
    pass
```

1.2 XInt

class XInt([*min*], [*max*], [*into*], [*not_into*])
The XInt check if a value can be understood as an integer, eventually with conditions min, max, into and not_into. The parameters min and max must be integer, the parameters into and not_into must be a list of integer. The following file shows the possibilities of XInt

```
# --
# Copyright (C) CEA, EDF
# Author : Erwan ADAM (CEA)
# --

import unittest

from xdata import *

class XIntTestCase(unittest.TestCase):
    def test(self):
        x = XInt()
        self.failUnlessEqual(x(123), 123)
        self.failUnlessEqual(x("123"), 123)
        self.failUnlessEqual(x(2*3), 6)
        self.failUnlessEqual(x("2*3"), 6)
        self.failUnlessRaises(XValueError, x, "toto")
        self.failUnlessRaises(XValueError, x, "3.2")
        self.failUnlessRaises(XValueError, x, 3.2)
        return
    def test_into_min_max(self):
        # into
        x = XInt(into=[1, "2", 3, "10", 20])
        self.failUnlessRaises(XValueError, x, 15)
        self.failUnlessEqual(x("10"), 10)
        self.failUnlessRaises(XValueError, XInt, into="toto")
        self.failUnlessRaises(XValueError, XInt, into=["toto", "titi"])
        # min, max
        self.failUnlessRaises(XValueError, XInt, min=2, max=4.5)
        x = XInt(into=[1, 2, 3, 5, 10, 20], min=5, max=15)
        self.failUnlessRaises(XValueError, x, 1)
        self.failUnlessEqual(x(5), 5)
        self.failUnlessEqual(x("10"), 10)
        self.failUnlessEqual(x(10), 10)
        return
    def test_not_into(self):
        x = XInt(not_into=[0, 1, 5])
        self.failUnlessRaises(XValueError, x, 0)
        self.failUnlessRaises(XValueError, x, 1)
        self.failUnlessRaises(XValueError, x, 5)
        self.failUnlessEqual(x(2), 2)
        self.failUnlessEqual(x("2"), 2)
        return
    def test_boolean(self):
        x = XInt()
        self.failUnless(x(True) is 1)
        self.failUnless(x(False) is 0)
        self.failUnless(x("True") is 1)
        self.failUnless(x("False") is 0)
        return
    pass

if __name__ == '__main__':
    unittest.main()
    pass
```

1.3 XFloat

class `XFloat` ([*min*], [*max*], [*open_min*], [*open_max*])

Use it to check if a value can be understood as an floating point number, eventually with conditions `min`, `max`, `open_min` and `open_max`. The parameters `min`, `max`, `open_min` and `open_max` must be floating point numbers. The conditions `open_min` and `open_max` do not accept the limit values ...

```
# --
# Copyright (C) CEA, EDF
# Author : Erwan ADAM (CEA)
# --

import unittest

from xdata import *

class XFloatTestCase(unittest.TestCase):
    def test(self):
        x = XFloat()
        self.failUnlessRaises(XValueError, x, x)
        self.failUnlessRaises(XValueError, x, "toto")
        self.failUnlessEqual(x(3.2), 3.2)
        self.failUnlessEqual(x("3.2"), 3.2)
        self.failUnlessEqual(x(123), 123)
        self.failUnlessEqual(x("123"), 123)
        self.failUnlessEqual(x("3.2e3"), 3200)
        #
        self.failUnlessEqual(x("3.*2"), 6.0)
        self.failUnlessEqual(x("3./2"), 1.5)
        return
    def test_min_max(self):
        x = XFloat(min=10.0, max=15.3)
        self.failUnlessRaises(XValueError, x, 5)
        self.failUnlessRaises(XValueError, x, 20)
        self.failUnlessEqual(x("10"), 10)
        self.failUnlessEqual(x(10.0), 10)
        return
    def test_openmin_openmax(self):
        x = XFloat(open_min=0.0, open_max=10)
        self.failUnlessRaises(XValueError, x, 10)
        self.failUnlessRaises(XValueError, x, 0)
        self.failUnlessEqual(x("9.0"), 9)
        return
    def test_xtypes(self):
        self.failUnlessRaises(XValueError, XFloat, min="toto")
        self.failUnlessRaises(XValueError, XFloat, max="toto")
        self.failUnlessRaises(XValueError, XFloat, open_min="toto")
        self.failUnlessRaises(XValueError, XFloat, open_max="toto")
        return
    pass

if __name__ == '__main__':
    unittest.main()
    pass
```

1.4 XString

class **`XString`**([*len*], [*len_min*], [*len_max*], [*into*], [*not_into*])

Use it to check if a value can be understood as a string ... The parameters `into` and `not_into` are list of string, The parameters `len`, `len_min` and `len_max` must be integer.

```
# --
# Copyright (C) CEA, EDF
# Author : Erwan ADAM (CEA)
# --

import unittest

from xdata import *

class XStringTestCase(unittest.TestCase):
    def test(self):
        x = XString()
        self.failUnlessRaises(XValueError, x, 3.2)
        self.failUnlessEqual(x("test"), "test")
        return
    def test_into(self):
        self.failUnlessRaises(XValueError, XString, into=[1.1])
        x = XString(into=["toto", "titi"])
        self.failUnlessRaises(XValueError, x, "test")
        self.failUnlessEqual(x("toto"), "toto")
        return
    def test_len(self):
        self.failUnlessRaises(XValueError, XString, len="toto")
        self.failUnlessRaises(XValueError, XString, len=-1)
        x = XString(len=3)
        self.failUnlessRaises(XValueError, x, "a")
        self.failUnlessRaises(XValueError, x, "aaaaa")
        self.failUnlessEqual(x("aaa"), "aaa")
        return
    def test_len_min_max(self):
        x = XString(len_min=2, len_max=3)
        self.failUnlessRaises(XValueError, x, "a")
        self.failUnlessRaises(XValueError, x, "aaaa")
        self.failUnlessEqual(x("aa"), "aa")
        self.failUnlessEqual(x("aaa"), "aaa")
        return
    def test_into_nocase(self):
        x = XString(into=["toto", "titi"])
        self.failUnlessRaises(XValueError, x, "test")
        self.failUnlessEqual(x("ToTo"), "toto")
        self.failUnlessEqual(x("TITI"), "titi")
        x = XString(into=["Castem", "PorFlow"])
        self.failUnlessRaises(XValueError, x, "test")
        self.failUnlessEqual(x("castem"), "Castem")
        self.failUnlessEqual(x("porflow"), "PorFlow")
        return
    pass

if __name__ == '__main__':
    unittest.main()
    pass
```

1.4.1 XBoolean

class `XBoolean`()

Use it to define a value as `true` or `false`. When called with ``` ``true'' ``` (whatever the case), return `true`, When called with ``` ``false'' ``` (whatever the case), return `false`. In every other case, return `not not value`.

1.4.2 XFileName

class `XFileName`([*suffix_into*])

Abstract class. Don't use it as is. The parameter `suffix_into` must be a string or a list of string.

class `XInputFileName`()

Inherit `XFileName`. Add a input browse popup on gui part.

class `XOutputFileName`()

Inherit `XFileName`. Add a output browse popup on gui part.

1.4.3 XDirName

class XDirName ([*suffix_into*])
Abstract class. Don't use it as is. The parameter `suffix_into` must be a string or a list of string.

class XInputDirName ()
Inherit `XDirName`. Add a input browse popup on gui part.

class XOutputDirName ()
Inherit `XDirName`. Add a output browse popup on gui part.

1.5 XList

class XList ([*sequence*], [*len*], [*len_min*], [*len_max*])
Use it to check if a value can be understood as a list ... The parameter `sequence` must be a `XType` instance or a list of `XType` instances, the parameters `len`, `len_min` and `len_max` must be integer.

```
# --
# Copyright (C) CEA, EDF
# Author : Erwan ADAM (CEA)
# --

import unittest

from xdata import *

class XListTestCase(unittest.TestCase):
    def test(self):
        x = XList()
        self.failUnlessEqual(x(3.2), [3.2])
        self.failUnlessEqual(x([]), [])
        return
    def test_len_len_min_len_max(self):
        self.failUnlessRaises(XValueError, XList, len=3.2)
        x = XList(len=2)
        self.failUnlessRaises(XValueError, x, ["toto"])
        self.failUnlessEqual(x([1, 2]), [1, 2])
        x = XList(len_min=3, len_max=5)
        self.failUnlessRaises(XValueError, x, [1,2])
        self.failUnlessRaises(XValueError, x, [1,2,3,4,5,6])
        self.failUnlessEqual(x([1,2,3]), [1,2,3])
        return
    def test_sequence(self):
        #
        self.failUnlessRaises(XValueError, XList, sequence="toto")
        #
        x = XList(sequence=XInt(min=0))
        self.failUnlessRaises(XValueError, x, ["toto"])
        self.failUnlessRaises(XValueError, x, [-1])
        self.failUnlessEqual(x(["1"]), [1])
        #
        x = XList(sequence=XTuple(sequence=XInt(), len=2))
        self.failUnlessEqual(x([(1, 1), (2, 2)]), [(1, 1), (2, 2)])
        self.failUnlessRaises(XValueError, x, [(3, 1, 0)])
        #
        x = XList(sequence=(XString(), XInt()))
        self.failUnlessRaises(XValueError, x, ["a"])
        self.failUnlessEqual(x(["a", "1"]), ["a", 1])
        self.failUnlessRaises(XValueError, x, ["a", 1, "b"])
        self.failUnlessEqual(x(["a", "1", "b", "2"]), ["a", 1, "b", 2])
        self.failUnlessRaises(XValueError, x, ["a", 3.2, "b", 12])
        self.failUnlessRaises(XValueError, x, ["a", "toto", "b", 12])
        #
        return
    def test_sequence_tolist(self):
        x = XList(sequence=XInt())
        self.failUnlessEqual(x(1), [1])
        return
    def test_string_with_comma_tolist(self):
        x = XList(sequence=XInt())
        self.failUnlessEqual(x('1, 2, 3, 4'), [1, 2, 3, 4])
        return
    def test_len_multiple(self):
        x = XList(sequence=(XInt(), XFloat()), len_multiple=1)
        x = XList(len_multiple=1, sequence=(XInt(), XFloat()))
        self.failUnlessEqual(x([1]), [1])
        self.failUnlessEqual(x([1, 2.3]), [1, 2.3])
        self.failUnlessEqual(x([1, 2.3, 2]), [1, 2.3, 2])
        return
    pass

class A(XObject):
    __init__xattributes__ = [
        XAttribute("l", xtype=XList(), default_value=None),
        ]
    pass

class ATestCase(unittest.TestCase):
    def test(self):
        a = A(None)
        self.failUnlessEqual(a.l, None)
        return
    pass

if __name__ == '__main__':
    unittest.main()
    pass
```

1.6 XTuple

class `XTuple` ([*sequence*], [*len*], [*len_min*], [*len_max*])

Use it to check if a value can be understood as a tuple ... The parameter `sequence` must be a `XType` instance or a list of `XType` instances, the parameters `len`, `len_min` and `len_max` must be integer.

```
# --
# Copyright (C) CEA, EDF
# Author : Erwan ADAM (CEA)
# --

import unittest

from xdata import *

class XTupleTestCase(unittest.TestCase):
    def test(self):
        x = XTuple()
        self.failUnlessEqual(x(3.2), (3.2, ))
        self.failUnlessEqual(x(("a", 1)), ("a", 1))
        self.failUnlessEqual(x(("a",)), ("a",))
        return
    def test_sequence(self):
        self.failUnlessRaises(XValueError, XTuple, sequence="toto")
        self.failUnlessRaises(XValueError, XTuple, sequence=[])
        #
        x = XTuple(sequence=XInt(min=0))
        self.failUnlessRaises(XValueError, x, ("toto", ) )
        self.failUnlessRaises(XValueError, x, (-1,) )
        self.failUnlessEqual(x( (1, ) ), (1, ))
        self.failUnlessEqual(x( ("1", ) ), (1, ))
        #
        x = XTuple(sequence=[XString(), XInt()])
        self.failUnlessRaises(XValueError, x, ("toto", ) )
        self.failUnlessRaises(XValueError, x, ("toto", "toto") )
        self.failUnlessRaises(XValueError, x, (1, 2, 3) )
        self.failUnlessEqual(x( ("a", 1) ), ("a", 1))
        self.failUnlessEqual(x( ("a", "1") ), ("a", 1))
        return
    def test_sequence_totuple(self):
        x = XTuple(sequence=XInt())
        self.failUnlessEqual(x(1), (1, ))
        return
    def test_list_to_tuple(self):
        x = XTuple()
        self.failUnlessEqual(x([1, 2, 3,]), (1, 2, 3,))
        return
    def test_xtuple_str(self):
        x = XTuple(sequence=(XString(), XFloat(), ))
        val = x("('He', 0.4)")
        self.failUnlessEqual(val, ('He', 0.4))
        val = x("He, 0.4")
        self.failUnlessEqual(val, ('He', 0.4))
        val = x("[He, 0.4]")
        self.failUnlessEqual(val, ('He', 0.4))
        return
    pass

if __name__ == '__main__':
    unittest.main()
    pass
```

1.7 XDict

class XDict ([*keys*], [*values*])

Use it to check if a value can be understood as a dict ... The parameters `keys` and `values` must be a `XType` instance.

```
# --
# Copyright (C) CEA, EDF
# Author : Erwan ADAM (CEA)
# --

import unittest

from xdata import *

class XDictTestCase(unittest.TestCase):
    def test(self):
        x = XDict()
        self.failUnlessRaises(XValueError, x, 0)
        self.failUnlessEqual(x({}), {})
        return
    def test_keys(self):
        self.failUnlessRaises(XValueError, XDict, keys="toto")
        x = XDict(keys=XString())
        self.failUnlessRaises(XValueError, x, {1: None})
        self.failUnlessEqual(x({"a":None, "b":None}), {"a":None, "b":None})
        x = XDict(keys=XInt())
        self.failUnlessRaises(XValueError, x, {"toto": None})
        self.failUnlessEqual(x({1: None, 2:None}), {1:None, 2:None})
        self.failUnlessEqual(x({"1": None, "2":None}), {1:None, 2:None})
        return
    def test_keys_values(self):
        self.failUnlessRaises(XValueError, XDict, values="toto")
        x = XDict(keys=XString(), values=XInt())
        self.failUnlessRaises(XValueError, x, {"a":None, "b":None})
        self.failUnlessEqual(x({"a": 1, "b": 2}), {"a": 1, "b": 2})
        self.failUnlessEqual(x({"a": "1", "b": "2"}), {"a": 1, "b": 2})
        return
    pass

if __name__ == '__main__':
    unittest.main()
    pass
```

1.8 XInstance and XSalomeReference

class `XReferenceAbstract`()
Abstract class.

1.8.1 XInstance

class `XInstance`(*classes*)
Use it to check if a value is an instance of one of classes ... The items in classes can be a class or a string "module.class"

```
# --
# Copyright (C) CEA, EDF
# Author : Erwan ADAM (CEA)
# --

import unittest

from xdata import *

class A:
    pass

class B(object):
    pass

class C(object):
    pass

class XInstanceTestCase(unittest.TestCase):
    def test(self):
        self.failUnlessRaises(XAttributeError, XInstance)
        return
    def test_init(self):
        self.failUnlessRaises(XValueError, XInstance, classes=3.2)
        self.failUnlessRaises(XValueError, XInstance, classes=[3.2])
        self.failUnlessRaises(XValueError, XInstance, 3.2)
        self.failUnlessRaises(XValueError, XInstance, "XXXXXX")
        x = XInstance(classes=[A, B])
        self.failUnlessEqual(x.classes, [A, B])
        x = XInstance(A, B)
        self.failUnlessEqual(x.classes, [A, B])
        x = XInstance("xdata.XString")
```

```
            return
        def test_classes(self):
            x = XInstance(A)
            self.failUnlessRaises(XValueError, x, 3.2)
            x = XInstance(A)
            a = A()
            self.failUnlessEqual(x(a), a)
            x = XInstance(B)
            self.failUnlessRaises(XValueError, x, A())
            b = B()
            self.failUnlessEqual(x(b), b)
            x = XInstance(classes=[A, B])
            a = A()
            self.failUnlessEqual(x(a), a)
            b = B()
            self.failUnlessEqual(x(b), b)
            c = C()
            self.failUnlessRaises(XValueError, x, c)
            return
        def test_string_classes(self):
            x = XInstance("xdata.XString")
            self.failUnlessRaises(XValueError, x, "toto")
            v = XString()
            self.failUnlessEqual(x(v), v)
            return
        pass

    if __name__ == '__main__':
        unittest.main()
        pass
```

1.8.2 XSalomeReference

class `XSalomeReference` (*component, idl, module, interface*)

Use it to check if a value is a salome corba reference. Attribute `component` is a string giving the salome component name (if `component=COMP`, it means that the variable `COMP_ROOT_DIR` must be defined), Attribute `idl` is a string giving the idl file name which shoulb be in `$COMP_ROOT_DIR/idl/salome`. Attribute `module` is the corba module and attribute `module` is the corba interface.

1.9 XMulTypes

class `XMulTypes` (**args*)

Use this if a value can be of one of the xtypes defined in *args

```
# --
# Copyright (C) CEA, EDF
# Author : Erwan ADAM (CEA)
# --

import unittest

from xdata import *

class XMulTypesTestCase(unittest.TestCase):
    def test(self):
        self.failUnlessRaises(XValueError, XMulTypes, "toto")
        #
        x = XMulTypes(XInt(min=3), XFloat(max=-12))
        self.failUnlessRaises(XValueError, x, 0)
        self.failUnlessEqual(x("14"), 14)
        self.failUnlessEqual(x("-44.56"), -44.56)
        #
        x = XMulTypes(XInt(), XList(sequence=XInt()))
        self.failUnlessRaises(XValueError, x, "toto")
        self.failUnlessRaises(XValueError, x, ["toto"])
        self.failUnlessEqual(x("1"), 1)
        self.failUnlessEqual(x([1, 2, 3]), [1, 2, 3])
        #
        x = XMulTypes(XInt(), XList(sequence=[XString(),XInt()]))
        self.failUnlessEqual(x(1), 1)
        self.failUnlessEqual(x(["a", "1", "b", "2"]), ["a", 1, "b", 2])
        self.failUnlessRaises(XValueError, x, "toto")
        self.failUnlessRaises(XValueError, x, ["a", "1", "b"])
        self.failUnlessRaises(XValueError, x, ["a", "1", "b", "toto"])
        return
    pass

if __name__ == '__main__':
    unittest.main()
    pass
```

CHAPTER

TWO

XAttribute

class **`XAttribute`**(*name*, [*xtype — default_value*], [*mode*])

The `XAttribute` is used to define an attribute to a class inherited from `XObject` (see section about XObject for more precisions), there is no interest to use this class directly ... The `name` is mandatory and must be a non-empty string. The `xtype` is used to test the values that the attribute should accept. The `default_value` is used to set the attribute if the user does not. Note that `default_value` and `xtype` are optionals but you must give informations to check the type of your attribute. It means that at least one of `default_value` and `xtype` or, of course, both must be given. The `mode` precises if the attribute is readonly (`'r'`) or readwrite (`'rw'`). This keyword is optional but its default value differs if the `XAttribute` instance is used in a constructor or to describe an object attributes.

Caution: The particuliar case where default_value is given to be `None` (a very common case indeed) is special. In that case, we assume that the value `None` should be accepted even if the `xtype` given does not accept it. To ensure that, the `xtype` is modified (for instance, if the `xtype` was `XFloat()`, the real xtype will be `XMulTypes(XNone(), XFloat())`. It is the only case where the developper `xtype` is modified ...

The following code is the unittest file used in the `xdata` developpement. It shows the use of `XAttribute`

```
# --
# Copyright (C) CEA, EDF
# Author : Erwan ADAM (CEA)
# --

import unittest

from xdata import *

class XAttributeTestCase(unittest.TestCase):
    def test_argslen(self):
        self.failUnlessRaises(XAttributeError, XAttribute, "toto", None)
        x= XAttribute("toto", default_value=None)
        return
    def test_name(self):
        self.failUnlessRaises(XValueError, XAttribute, 2.3, default_value=None)
        self.failUnlessRaises(XValueError, XAttribute, "", default_value=None)
        name = "x"
        x = XAttribute(name, default_value=None)
        self.failUnlessEqual(name, x.name)
        self.failUnlessEqual(name, x.getName())
        return
    def test_noargs(self):
        x = XAttribute("aaa", xtype=XString())
        self.failUnlessEqual(x.hasDefaultValue(), 0)
        self.failUnlessRaises(XValueError, x.getDefaultValue)
        self.failUnlessEqual(x.xtype("toto"), "toto")
        return
    def test_default_value_without_xtype(self):
        default_value = 123
        x = XAttribute("aaa", default_value=default_value)
        self.failUnlessEqual(x.hasDefaultValue(), 1)
        self.failUnlessEqual(x.getDefaultValue(), default_value)
        self.failUnlessEqual(x.xtype("256"), 256)
        self.failUnlessEqual(x.xtype(default_value), default_value)
        self.failUnlessRaises(XValueError, x.xtype, 1.2)
        self.failUnlessRaises(XValueError, x.xtype, "toto")
        #
        x = XAttribute("a", default_value=None)
        self.failUnlessEqual(x.hasDefaultValue(), 1)
        self.failUnlessEqual(x.getDefaultValue(), None)
        #
        return
    def test_xtype(self):
        self.failUnlessRaises(XValueError, XAttribute, "aa", xtype="toto")
        name = "toto"
        x = XAttribute(name, xtype=XFloat(min=0.5))
        self.failUnlessRaises(XValueError, x.xtype, "toto")
        self.failUnlessRaises(XValueError, x.xtype, -2.3)
        return
    def test_default_value_with_types(self):
        # particuliar case when default_value is None
        x = XAttribute("x", default_value=None, xtype=XString())
        self.failUnlessEqual(x.xtype("toto"), "toto")
        self.failUnlessEqual(x.xtype("None"), None)
        # imcompatibility between "toto" and XFloat()
        self.failUnlessRaises(XValueError, XAttribute, "x", default_value="toto", xtype=
        return
    pass

if __name__ == '__main__':
    unittest.main()
    pass
```

CHAPTER

THREE

XMethod

class `XMethod`(*name*, [*in_xattributes*], [*out_xtype*])

The `XMethod` is used to precise the arguments and the result type of a method defined in a class inherited from `XObject` (see section about XObject for more precisions), there is no interest to use this class directly ... The `name` is mandatory and must be a non-empty string. The optional `in_xattributes` member is a list of `XAttribute` instances. The optional `out_xtype` member is a `XType` instance which precise what is the result type.

CHAPTER

FOUR

XObject and XNamedObject

class XObject()

The XObject class should not be used directly but with inheritence. Indeed, if a class inherit the XObject, a python Meta-class is used to transform a list of keywords (declared as class attributes) in something understood by python itself. Note that, this job is done at the class creation and not at the first instance. The documented keywords are `__init__xattributes__`, `__init__argslen__`, `__object__xattributes__` and `__object__xmethods__`.

class XNamedObject()

The XNamedObject class inherit from XObject, and thus the comportment is similar. The difference is that an attribute name is defined for the classes which inherit XNamedObject. More-over, the name is initialized at a value which “makes sense” in the user python script. Indeed, the name is found by the code calling the class. Technically, it is not done by parsing the code source but by disassembling the code itself (see python module dis). Using XNamedObject avoid to write code like `a = A(name='a')`. For instance:

```
# --
# Copyright (C) CEA, EDF
# Author : Erwan ADAM (CEA)
# --

import unittest

from xdata import *

class A(XNamedObject):
    pass

class ATestCase(unittest.TestCase):
    def test(self):
        a = A()
        self.failUnlessEqual(a.getName(), "a")
        self.failUnlessEqual(a.name, "a")
        a.name = "aaa"
        self.failUnlessEqual(a.getName(), "aaa")
        self.failUnlessEqual(a.name, "aaa")
        return
    pass

if __name__ == '__main__':
    unittest.main()
    pass
```

If the name cannot be found, an exception is not raised at the object creation but if one try to access the name:

```
# --
# Copyright (C) CEA, EDF
# Author : Erwan ADAM (CEA)
# --
```

```
import unittest

from xdata import *

class A(XNamedObject):
    pass

def f(x):
    return x

class ATestCase(unittest.TestCase):
    def test(self):
        x = f(A())
        # Obsolete : getName() does not raise but return None
        # self.failUnlessRaises(XAttributeError, x.getName)
        # self.failUnlessRaises(XAttributeError, getattr, x, "name")
        return
    pass

if __name__ == '__main__':
    unittest.main()
    pass
```

4.1 Constructor attributes (keyword `__init__xattributes__`)

The keyword `__init__xattributes__` must contains a list of `XAttribute`'s. This keyword allows to define the constructor of the class. You can access each attribute directly by its name or by get and set methods. For instance, if an xattribute has name `internal_radius`, you can access it by `internal_radius`, `getInternalRadius` and `setInternalRadius`. For instance:

```
# --
# Copyright (C) CEA, EDF
# Author : Erwan ADAM (CEA)
# --

import unittest

from xdata import *

class A(XObject):
    __init__xattributes__ = [
        XAttribute("x", xtype=XInt(min=0)),
        ]
    pass

class ATestCase(unittest.TestCase):
    def test__init__(self):
        self.failUnlessRaises(XAttributeError, A)
        self.failUnlessRaises(XAttributeError, A, 1, 2)
        self.failUnlessRaises(XAttributeError, A, y=2)
        a = A(1)
        a = A(x=3)
        return
    def test_accessors(self):
        a = A(0)
        self.failUnlessEqual(a.x, 0)
        self.failUnlessEqual(a.getX(), 0)
        a.x = 1
        self.failUnlessEqual(a.x, 1)
        a.setX(2)
        self.failUnlessEqual(a.getX(), 2)
        a.x = "2+5+3"
```

```
            self.failUnlessEqual(a.x, 10)
            return
        def test_value(self):
            self.failUnlessRaises(XValueError, A, -1)
            self.failUnlessRaises(XValueError, A, "toto")
            a = A(0)
            self.failUnlessRaises(XValueError, a.setX, -1)
            self.failUnlessRaises(XValueError, a.__setattr__, "x", -1)
            self.failUnlessRaises(XValueError, setattr, a, "x", -1)
            return
        pass

if __name__ == '__main__':
    unittest.main()
    pass
```

Of course, the value can be optional:

```
# --
# Copyright (C) CEA, EDF
# Author : Erwan ADAM (CEA)
# --

import unittest

from xdata import *

class A(XObject):
    __init__xattributes__ = [
        XAttribute("x", default_value=0, xtype=XInt(min=0)),
        ]
    pass

class ATestCase(unittest.TestCase):
    def test__init__(self):
        a = A()
        self.failUnlessEqual(a.x, 0)
        a = A(1)
        self.failUnlessEqual(a.x, 1)
        a = A(x=2)
        self.failUnlessEqual(a.x, 2)
        return
    pass

if __name__ == '__main__':
    unittest.main()
    pass
```

4.1.1 Implicit or Explicit attributes (keyword __init__argslen__)

The keyword `__init__argslen__` must be an integer. It can be use, for instance, to force users to give all the attributes explicitely. See the following code:

```
# --
# Copyright (C) CEA, EDF
# Author : Erwan ADAM (CEA)
# --

import unittest

from xdata import *

class A(XObject):
    # 'x', 'y', 'z' can be given implicitely
    __init__xattributes__ = [
        XAttribute("x", xtype=XInt(min=0)),
        XAttribute("y", xtype=XInt(min=0)),
        XAttribute("z", xtype=XInt(min=0)),
        ]
    pass

class ATestCase(unittest.TestCase):
    def test(self):
        a = A(1, 2, 3)
        a = A(1, 2, z=3)
        a = A(z=3, y=2, x=1)
        return
    pass

class B(XObject):
    __init__argslen__ = 0 # Everything must be given explicitely
    __init__xattributes__ = [
        XAttribute("x", xtype=XInt(min=0)),
        XAttribute("y", xtype=XInt(min=0)),
        XAttribute("z", xtype=XInt(min=0)),
        ]
    pass

class BTestCase(unittest.TestCase):
    def test(self):
        b = B(x=1, y=2, z=3)
        b = B(z=3, y=2, x=1)
        self.failUnlessRaises(XAttributeError, B, 1, 2, 3)
        self.failUnlessRaises(XAttributeError, B, 1, y=2, z=3)
        return
    pass

class C(XObject):
    __init__argslen__ = 1 # Only 'x' can be given implicitely
    __init__xattributes__ = [
        XAttribute("x", xtype=XInt(min=0)),
        XAttribute("y", xtype=XInt(min=0)),
        XAttribute("z", xtype=XInt(min=0)),
        ]
    pass

class CTestCase(unittest.TestCase):
    def test(self):
        c = C(1, z=3, y=2)
        c = C(z=3, y=2, x=1)
        self.failUnlessRaises(XAttributeError, C, 1, 2, 3)
        self.failUnlessRaises(XAttributeError, C, 1, 2, z=3)
        return
    pass

if __name__ == '__main__':
    unittest.main()
    pass
```

4.1.2 What about the user constructor ?

You can write your own `__init__` method. In that case, the meta `__init__` is run to check all the values passed to the constructor and to set all the attributes. Then, the user `__init__` is run:

```
# --
# Copyright (C) CEA, EDF
# Author : Erwan ADAM (CEA)
# --

import unittest

from xdata import *

class A(XObject):
    __init__xattributes__ = [
        XAttribute("x", xtype=XInt(min=0)),
        ]
    def __init__(self, *args, **kwargs):
        self.test = 1
        return
    pass

class ATestCase(unittest.TestCase):
    def test(self):
        a = A(1)
        self.failUnlessEqual(a.test, 1)
        return
    pass

if __name__ == '__main__':
    unittest.main()
    pass
```

4.2 What about the user accessors ?

On the same way than for the constructor, you can write your own `set` for all the attributes defined in your class. It allows to do job that is not done by xdata, to set other attributes or to do whatever you want ! In that case, the meta `set` is run to check the value and to set the attribute, then the user `set` is run:

```
# --
# Copyright (C) CEA, EDF
# Author : Erwan ADAM (CEA)
# --

import unittest

from xdata import *

class A(XObject):
    __init__xattributes__ = [
        XAttribute("x", xtype=XInt(min=0)),
        ]
    def setX(self, value):
        self.test = 1
        return
    pass

class ATestCase(unittest.TestCase):
    def test(self):
        a = A(1)
        self.failUnlessEqual(a.test, 1)
        return
    pass

if __name__ == '__main__':
    unittest.main()
    pass
```

4.3 Other attributes (keyword __object__xattributes__)

The keyword `__object__xattributes__` is used to define attributes which do not appear in the constructor. Note that the default `mode` for attributes in `__object__xattributes__` is readonly (`'r'`). If you want this attribute to be readwrite (`'rw'`), you must specify it like that:

```
# --
# Copyright (C) CEA, EDF
# Author : Erwan ADAM (CEA)
# --

import unittest

from xdata import *

class A(XObject):
    __object__xattributes__ = [
        XAttribute("x", xtype=XInt(min=0), mode='rw'),
        ]
    pass

class ATestCase(unittest.TestCase):
    def test(self):
        a = A()
        a.x = 1
        self.failUnlessEqual(a.x, 1)
        a.setX(2)
        self.failUnlessEqual(a.getX(), 2)
        return
    pass

if __name__ == '__main__':
    unittest.main()
    pass
```

Of course, on the same way than attributes defined in `__init__xattributes__`, you can define your own accessors for the `__object__xattributes__` attributes:

```
# --
# Copyright (C) CEA, EDF
# Author : Erwan ADAM (CEA)
# --

import unittest

from xdata import *

class A(XObject):
    __object__xattributes__ = [
        XAttribute("x", xtype=XInt(min=0), mode='rw'),
        ]
    def setX(self, value):
        self.test = 1
        return
    pass

class ATestCase(unittest.TestCase):
    def test(self):
        a = A()
        a.x = 1
        self.failUnlessEqual(a.test, 1)
        return
    pass
```

```
if __name__ == '__main__':
    unittest.main()
    pass
```

4.4 Read-only attributes

See the following code to understand what I mean by read-only attributes. In the `Circle` class, the `diameter` is setted in `setRadius` and we don't want a user can change its value directly. It is what is checked in `def test_readonly(self)`.

```
# --
# Copyright (C) CEA, EDF
# Author : Erwan ADAM (CEA)
# --

import unittest

from xdata import *

class Circle(XNamedObject):
    __init__xattributes__ = [
        XAttribute("radius", xtype=XFloat(open_min=0.0)),
        ]
    __object__xattributes__ = [
        XAttribute("diameter", xtype=XFloat(open_min=0.0)),
        XAttribute("test_init", xtype=XFloat(open_min=0.0)),
        XAttribute("test_method", xtype=XFloat(open_min=0.0)),
        ]
    def __init__(self, *args, **kwargs):
        self.test_init = 1.0
        return
    def setRadius(self, value):
        self.diameter = 2*self.radius
        return
    def method(self):
        self.test_method = 1.0
        return
    pass

class CircleTestCase(unittest.TestCase):
    def test(self):
        c = Circle(1.0)
        self.failUnlessEqual(c.radius, 1.0)
        self.failUnlessEqual(c.diameter, 2.0)
        c.setRadius(2.0)
        self.failUnlessEqual(c.getRadius(), 2.0)
        self.failUnlessEqual(c.getDiameter(), 4.0)
        return
    def test_readonly(self):
        c = Circle(1.0)
        self.failUnlessRaises(AttributeError, c.setDiameter, 2.0)
        self.failUnlessRaises(AttributeError, c.__setattr__, "diameter", 2.0)
        self.failUnlessRaises(AttributeError, setattr, c, "diameter", 2.0)
        #
        c.method()
        return
    pass

if __name__ == '__main__':
    unittest.main()
    pass
```

4.5 Dependencies between attributes

There is, for the moment, nothing in `xdata` to check dependencies between attributes ... but there is the possibility, as explained in section about user accessors, for the developper to write his own accessors. For instance, in the following example, we check that the `external_radius` is greater than the `internal_radius` (Note the use of an instance of `XFloat` to do the job :))

```
# --
# Copyright (C) CEA, EDF
# Author : Erwan ADAM (CEA)
# --

import unittest

from xdata import *

class GuideTube2D(XNamedObject):
    __init__xattributes__ = [
        XAttribute("external_radius", xtype=XFloat(open_min=0.0)),
        XAttribute("internal_radius", xtype=XFloat(min=0.0), default_value=0.0),
        ]
    def setExternalRadius(self, value):
        er = self.external_radius
        try:
            ir = self.internal_radius
        except AttributeError:
            return
        XFloat(open_min=ir).testValue(er)
        return
    def setInternalRadius(self, value):
        er = self.external_radius
        ir = self.internal_radius
        XFloat(open_max=er).testValue(ir)
        return
    pass

class GuideTube2DTestCase(unittest.TestCase):
    def test(self):
        g = GuideTube2D(3.0, 2.0)
        self.failUnlessRaises(XValueError, g.setInternalRadius, 4.0)
        self.failUnlessRaises(XValueError, g.setExternalRadius, 1.0)
        self.failUnlessEqual(g.external_radius, 3.0)
        self.failUnlessEqual(g.internal_radius, 2.0)
        return
    pass

if __name__ == '__main__':
    unittest.main()
    pass
```

4.6 Object methods (keyword __object__xmethods__)

The keyword `__object__xmethods__` is used to declare methods, you can specify it like that:

```
# --
# Copyright (C) CEA, EDF
# Author : Erwan ADAM (CEA)
# --

import unittest
```

```
from xdata import *

class A(XObject):
    __object__xmethods__ = [
        XMethod("run",
                in_xattributes = XAttribute("value", xtype=XString(into=["Castem", "PorFlow"
                out_xtype = XString(),
                )
        ]
    def run(self, value):
        return value
    pass

class TestCase(unittest.TestCase):
    def test(self):
        a = A()
        self.failUnlessEqual(a.run('CASTEM'), "Castem")
        return
    pass

if __name__ == '__main__':
    unittest.main()
    pass
```

4.7 XDATA_VERBOSE

When creating a class inherited from `XObject`, a lot of things are done parsing the `__init__xattributes__`, `__object__xattributes__`, ... members. If you want to check what is done to your class (for instance, for debugging) you can set the XDATA_VERBOSE environment variable to `1`. To reverse, unset it or set it to `0`.

CHAPTER

FIVE

Other utilities

5.1 Version

You can check the version of your `xdata` module with `import xdata ; print xdata.version`.

5.2 getXClasses

The function `getXClasses` gives a tuple of user-defined classes which inherit for `XObject`. For instance

```
# --
# Copyright (C) CEA, EDF
# Author : Erwan ADAM (CEA)
# --

import unittest

from xdata import *

class getXClassesTestCase(unittest.TestCase):
    def test(self):
        #
        class A(XObject):
            pass
        #
        self.failUnlessEqual(getXClasses(), (A, ))
        #
        class B(XObject):
            pass
        #
        self.failUnlessEqual(getXClasses(), (A, B, ))
        return
    pass

if __name__ == '__main__':
    unittest.main()
    pass
```

5.3 getXObjects

The function `getXObjects` gives a tuple of user-defined objects. For instance

```
# --
# Copyright (C) CEA, EDF
# Author : Erwan ADAM (CEA)
# --
```

```
import unittest

from xcontext import setInterface

setInterface('tui')

from xdata import *

class A(XObject):
    pass

class getXClassesTestCase(unittest.TestCase):
    def test(self):
        a1 = A()
        self.failUnlessEqual(getXObjects(), (a1, ))
        a2 = A()
        self.failUnlessEqual(getXObjects(), (a1, a2, ))
        return
    pass

if __name__ == '__main__':
    unittest.main()
    pass
```

CHAPTER

SIX

Customization of xdata ...

Of course, different developpers have different needs and I'm sure that `xdata` does not cover all the needs of everybody ... but I think you can customize easily `xdata` if all the above is not sufficiant for you.

The point is that `xdata` itself is developped using `xdata` !! So it should be very easy to do specific job. In the following, I show different example of evolutions.

6.1 Adding a condition to XInt

In the following example, we write a new class which inherit from `XInt` to ensure that a value is a multiple of something:

```
# --
# Copyright (C) CEA, EDF
# Author : Erwan ADAM (CEA)
# --

import unittest

from xdata import *

class XIntWithMultiple(XInt):
    __init__xattributes__ = XInt.__init__xattributes__ + [
        XAttribute("multiple", xtype=XInt(not_into=[0]), default_value=None),
        ]
    def __call__(self, value):
        value = XInt.__call__(self, value)
        multiple = self.multiple
        if multiple is not None:
            residual = value%multiple
            if residual != 0:
                msg = "%s is not a multiple of %s"%(value, multiple)
                raise XValueError(msg)
            pass
        return value
    pass

class XIntWithMultipleTestCase(unittest.TestCase):
    def test(self):
        x = XIntWithMultiple(multiple=5)
        self.failUnlessEqual(x(10), 10)
        self.failUnlessEqual(x(0), 0)
        self.failUnlessEqual(x("15"), 15)
        self.failUnlessRaises(XValueError, x, 1)
        return
    def test_multiple(self):
        self.failUnlessRaises(XValueError, XIntWithMultiple, multiple="toto")
        self.failUnlessRaises(XValueError, XIntWithMultiple, multiple=0)
        return
    def testwithmin(self):
        x = XIntWithMultiple(multiple=5, min=1)
        self.failUnlessEqual(x(10), 10)
        self.failUnlessEqual(x("15"), 15)
        self.failUnlessRaises(XValueError, x, 0)
        self.failUnlessRaises(XValueError, x, 1)
        return
    pass

if __name__ == '__main__':
    unittest.main()
    pass
```

6.2 Adding keywords to XAttribute

In the following example, we write a new class which inherit from `XAttribute`. Some attributes have been added to add comments to attributes:

```
# --
# Copyright (C) CEA, EDF
# Author : Erwan ADAM (CEA)
# --

import unittest

from xdata import *

class XAttributeWithComments(XAttribute):
    __init__xattributes__ = XAttribute.__init__xattributes__ + [
        XAttribute("en_comment", xtype=XString(), default_value=""),
        XAttribute("fr_comment", xtype=XString(), default_value=""),
        XAttribute("de_comment", xtype=XString(), default_value=""),
        ]
    pass

class XAttributeWithCommentsTestCase(unittest.TestCase):
    def test(self):
        x = XAttributeWithComments("toto",
                                   en_comment = "hello",
                                   fr_comment = "bonjour",
                                   de_comment = "guten Tag",
                                   )
        self.failUnlessEqual(x.en_comment, "hello")
        self.failUnlessEqual(x.fr_comment, "bonjour")
        self.failUnlessEqual(x.de_comment, "guten Tag")
        return
    pass

if __name__ == '__main__':
    unittest.main()
    pass
```

6.3 Using a customized XAttribute class

Of course, as `XAttributeWithComments` inherit from `XAttribute`, it can be used in `__init__xattributes__` to describe other classes. See the following:

```
# --
# Copyright (C) CEA, EDF
# Author : Erwan ADAM (CEA)
# --

import unittest

from xdata import *

class XAttributeWithComments(XAttribute):
    __init__xattributes__ = XAttribute.__init__xattributes__ + [
        XAttribute("en_comment", xtype=XString(), default_value=""),
        XAttribute("fr_comment", xtype=XString(), default_value=""),
        XAttribute("de_comment", xtype=XString(), default_value=""),
        ]
    pass

class Point(XNamedObject):
    __init__xattributes__ = [
        XAttributeWithComments("x", xtype=XFloat(), en_comment="x coordinate"),
        XAttributeWithComments("y", xtype=XFloat(), en_comment="y coordinate"),
        XAttributeWithComments("z", xtype=XFloat(), en_comment="z coordinate"),
        ]
    pass

class PointTestCase(unittest.TestCase):
    def test(self):
        #
        x = Point(0, 0, 0)
        #
        xattrs = x.__class__.__init__xattributes__
        xattr1 = xattrs[0]
        self.failUnlessEqual(xattr1.en_comment, "x coordinate")
        self.failUnlessEqual(xattr1.fr_comment, "")
        return
    pass

if __name__ == '__main__':
    unittest.main()
    pass
```

CHAPTER

SEVEN

Meta-information used for Salome and for Graphical User Interfaces

The `xdata` module provides a way to

- Integrate easily a new component in Salome
- Interact with the classes via a gui (Graphical User Interface)

To do the job, the best way is to copy one of the templates given with `xdata` distribution and located in `XDATA_INSTALLATION_PATH/share/xdata/templates`. Then, you have to modify the sources included to fit your developpements.

The goal is to describe a whole set of classes, the menubar menus, the popup menus associated to objects, the action related ... all what is necessary to a graphical interface.

7.1 The boostrap files (_xdata.py files)

The first thing to do is to give a name to the whole set of your classes. This name will be the name of your component in salome in addition of pre-existing salome components (like GEOM, SMESH, VISU, ...). In python gui provided with xdata, it will be the same. In fact, in python gui, you usually have only one component so its name is less important because if you have only one component, it is activated by default, but you can imagine to have more that one, and in that case, the behaviour will be exactly the same as in salome.

For instance, if you want to create a component named `AAA`, the only thing to do to accomplish that is to create a `AAA_xdata.py` file in your sources. For obscure reasons (in salome), the name must be uppercase ...

Now, you have to indicate what are the classes, actions and so on contained in your component. To do that, you have to imagine your component as a graphical one and in your `_xdata` file, you just describe the menus you want to appear in the menu bar of the application.

For instance, the `TECHOBJ_xdata.py` boostrap file present in the `TECHOBJ` template is the following :

```
# --
# Copyright (C) CEA, EDF
# Author : Erwan ADAM (CEA)
# --

__xdata__items__ = [
    "tefile",
    "geometry",
    "salomeref",
    "material",
    "technologyobject",
    "menuxtypes",
    "hydraulicmodule",
    ]
    __comment__ = "Techobj template from xdata (not the real techobj !)"
```

The `__xdata__items__` keyword indicates the menus that you want to be present in the main menubar. A little subtility is that you can insert your own menus into pre-existing menus (like `File` or `Help` for instance), to do that you have to add a `'file'` or `'help'` line into the `__xdata__items__`. The second solution is to add `'myfile'` in `__xdata__items__` and declare `__xdata__name__ = 'File'` into `'myfile.py'`. The `__xdata__name__` keyword will be explained later.

Now let see how the process run, in short how the `__xdata__items__` list is treaten.

7.2 __xdata__items__ keyword

The `__xdata__items__` contains a list of string which allow to define a part of your application.

How the `__xdata__items__` is parsed ?

For each item in `__xdata__items__`, there are two possibilities:

- The item correspond to an existing python module. More precisly, `__import__(item)` does not raise ... In that case, a new menu is inserted in GUI and the current process is launched with this new module.
- This item does not correspond to an existing python module ... There are two possibilities:
  - The item is the empty string (or contains only white space). In that case, a separator is inserted in GUI mode.
  - Otherwise, an item whom name is the string is inserted into the current menu. What happen when the item is activated is described in section 8.1.

7.3 __xdata__name__ keyword

When inserting a menu in the menu bar, the item is named with the python module name capitalized unless (__xdata__name__) is defined. In python, it is written:

```
try:
    xdata_name = module.__xdata__name__
except AttributeError:
    xdata_name = module.__name__.capitalize()
    pass
```

CHAPTER

EIGHT

Graphical User Interface special features

8.1 Dialog creation

When activating an item from the menus and sub-menus in menubar, there are two possibilities :

- The string corresponding to the item refer a subclass of `XObject` declared in the current python module (the one where `__xdata__items__` is). In that case, the `createDialog` classmethod is called on the class. The signature of this class method is explained in section 8.1.1. Thus, if the class does not define the `createDialog` method, the default implementation defined in `XObject` is called ...
- The item is not declared in the current module (in that case `target = item`) or is defined but does not correspond to a `XObject` subclass (in that case `target = getattr(mod, item)`). In those two cases, the function `mod.createDialog(parent, target)` is launched where `parent` is the main window. Note that if `mod.createDialog` does not exist, a warning is launched.

8.1.1 createDialog class method

The signature of the `createDialog` class method is the following :

```
def createDialog(cls, parent, instance_name, xobject, editor, xattribute):
```

thus if you re-implement this class method, you have to follow this signature. The attributes are:

- `cls` : the class,
- `parent` : the qt parent object,
- `instance_name` : two solutions for this attribute. If the dialog is called for an object creation (in short, from menubar), the `instance_name` is a proposition for the instance name based on the class name and the already presents objects. If the dialog is called for the edition of an existing object (via a popup, Edit), `instance_name` is `None`
- `object` : `None` for an object creation, the object for an object edition.
- `editor` : `None` unless the dialog is called from the edition on an attribute in a edition dialog of another object. For instance, an object from the study is edited, this object has an attribute `a` which is an instance of class `A`, then if you are editing `a` for the edition dialog, the `editor` attribute will be the edition dialog.
- `xattribute` : `None` unless the dialog edition is called on an attribute of an object in the study. It is called the attribute direct edition. For instance, if you need to modify an integer attribute in an object, you just select the attribute, and with a popup, Edit, you can modify only this attribute.

The default implementation of `createDialog` coded in `XObject` class launch a dialog box with a line by `__init__xattributes__` for an object creation and a line by `__init__xattributes__` plus a line for each read-write `__object__xattributes__` for an object edition. The edition of an attribute in this dialog box is described in section 8.2.

8.2 Attribute edition : the createEditor execution

When clicking an attribute in a creation/edition dialog box, the `createEditor` class method is executed. The signature is the following :

```
def createEditor(cls, xattribute, parent, xobject, value, text)
```

where `cls` is the current class, `xattribute` is the xattribute corresponding to the item edited, `parent` is the parent qt widget. The `xobject` is quite special. If it is an object edition, this attribute contains the object being edited, if it is an object creation, this attribute contains a dictionnary whom keys are the attributes names already present and values are the value founded in the dialog edition box. It allows to deals with the dependencies between attributes at gui level. `value` attribute contains the current value or `None` if not already setted and `text` attribute contains the current text or `None` if not already setted.

The default implementation of `createEditor` class method coded in `XObject` class do nearly nothing ... it's only called the `createEditor` method on the `xattribute` object with the following signature:

```
def createEditor(self, cls, parent, xobject, value, text)
```

where `self` is the `XAttribute` instance and other attributes are the same as described previously.

The default implementation of `createEditor` method is coded in `XAttribute` class. It does nearly nothing ... it's only called the `createEditor` method on the `xtype` object contained in the `xattribute` object with the following signature:

```
def createEditor(self, xattribute, cls, parent, xobject, value, text)
```

where `self` is now an instance of `XType` class. The default implementation is coded in `XType` class and create the default behaviour *i.e.* a `QLineEdit`.

All this mechanism provide a way to intercept easily the way that the editors are created. A usual configuration is to define his own `XType` subclass and re-implement the `createEditor` method for this class. For instance :

```
from xdata import *

class MyXInstance(XInstance):

    def createEditor(self, *args, **kwargs):
        ## do before job
        editor = super(MyXInstance, self).createEditor(*args, **kwargs)
        ## do after job
        ## return the editor !
        return editor

    pass
```

8.3 Popups from study

When an object is created, the result is inserted in the study widget. When selecting an item (or more than one) in the study and clicking with the mouse right button, the popup definition is activated. There are two possibilities:

- All the selected items are instances of a subclass of `XObject`. In that case, a common base class of all instances is found. This class exists since `XObject` is a common base class but usually, another subclass is found. Note that when selecting only one object, the common base class is the class of the object. The common base class is called the container.

- If one of the selected items is not an instance of `XObject`, the container is the `_xdata` python module (the one coded in `AAA_xdata.py` file if the current module is `AAA`).

Once the container is founded, the popup definition calls `customPopup` on the container. If the container is a python module, `customPopup` is a function and the signature is:

```
customPopup(parent, popup, nodes)
```

if the container is a `XObject` subclass, `customPopup` is a class method and the signature is:

```
customPopup(cls, parent, popup, nodes).
```

In both cases, parent is the qt widget, popup is the `QPopupMenu` object and nodes is a list containing the selected objects.

Using `customPopup` is a good way to customize popups but you have to know qt a little to activate the correct qt signals when items are selected. If you don't want to know about qt, you can use the second way to intercept the popups. Indeed, the default implementation of `customPopup` calls the `getPopupItems` facility and connect the qt signals. The signature of `getPopupItems` is

```
getPopupItems(parent, nodes)
```

if the container is a python module. if the container is a `XObject` subclass, `getPopupItems` is a class method and the signature is:

```
getPopupItems(cls, parent, nodes).
```

`getPopupItems` must return a list of string to be inserted in the current popup menu. When an item from `getPopupItems` is activated, the `container.popupActivated` facility is activated. If the container is a python module, `popupActivated` must be a function with signature

```
popupActivated(parent, target, nodes),
```

if the container is a `XObject` subclass, `popupActivated` is a class method and the signature is:

```
popupActivated(cls, parent, target, nodes),
```

where `parent` and `nodes` are the parent qt widget and the selected object, `target` is the string representing the popup item which has been activated.

The default implementation of `XObject.getPopupItems` return a list of string based on the `__object__xmethods__` list defined in the class.

8.4 Popups from editor

When an attribute is edited via the `createEditor` action described in section 8.2, the editor have popup properties (like `Cancel`, `New`, `Edit`, ...). You can customize this popup using the `customEditorPopup` facility. The default implementation is a class method coded in `XObject` and the signature is

```
def customEditorPopup(cls, xattribute, parent, popup, xobject).
```

The default implementation call the `customEditorPopup` method on the xattribute object. The default implementation is coded in XAttribute

```
def customEditorPopup(self, cls, parent, popup, xobject):
```

where self is now the xattribute object. Once again, the default implementation call the customEditorPopup method on the xtype object contained in the xattribute

```
def customEditorPopup(self, xattribute, cls, parent, popup, xobject)
```

where self is now the xtype object. The default implementation do nothing. So, you have to re-implement it if you want to customize the behaviour of the popup editor.

8.5 Tooltips facility

If you want to introduce tooltips in your application, you can use the getToolTip facility. The usual way is to define a tooltip for a class and for its attributes. To do that, just define the getToolTip class method in your class with the following signature :

```
getToolTip(cls, target)
```

where target is the class itself or one of the attributes defined in __init__xattributes__ or __object__xattributes__.

CHAPTER

NINE

Integration in a Python Graphical User Interface

The xdata module provides a way to interact with the classes written with the xdata grammar via a python gui (Graphical User Interface). It require the python wrapping of qt and vtk to work (indeed, the vtk module is not required, a empty widget will be displayed if not found ...).

The sources of the template are included in the directory XDATA_INSTALLATION_PATH/share/xdata/templates. The example below is used in compiling and installing the TECHOBJ template. To do that, you should copy the TECHOBJ_ROOT arborescence from XDATA_INSTALLATION_PATH/share/xdata/templates in a local directory to not ¡¡pollute¿¿ your xdata installation. Then, follow the instructions in the README file of TECHOBJ_ROOT.

This file is reproduced here :

```
# --
# Copyright (C) CEA, EDF
# Author : Erwan ADAM (CEA)
# --

Introduction
============

You are in the template TECHOBJ which use the xdata python
module for developpement. We assume as this level that xdata
is correctly installed on your system.

Building and Installing
=======================

cd TECHOBJ_ROOT
cd ..
mkdir TECHOBJ_BUILD
cd TECHOBJ_BUILD
../TECHOBJ_ROOT/build_configure
../TECHOBJ_ROOT/configure --prefix=INSTALLATION_PATH
make install
make check (optional)

After all, please define a TECHOBJ_ROOT_DIR variable
to refer your INSTALLATION_PATH :

setenv TECHOBJ_ROOT_DIR INSTALLATION_PATH (csh)
export TECHOBJ_ROOT_DIR=INSTALLATION_PATH (sh)

Troubleshooting
===============

Please, send a mail to erwan.adam@cea.fr ...
```

9.1 Launcher

Once TECHOBJ is installed and the TECHOBJ_ROOT_DIR variable defined, you can launch the `xdatagui` python graphical user (which is in the `bin` directory of your xdata installation) via `xdatagui --modules=TECHOBJ`.

The `xdatagui` launcher has the following optionnal command line options:

- `--with-splitter=yes/no`. Set if the main window is splitted (the default) or not.
- `--with-menu=AAA,BBB,...`. Set the menus which must appear in the menubar.
- `--without-menu=AAA,BBB,...`. Set the menus which must not appear in the menubar.

More-over, the qt line options (like `-style`, `-geometry`) are taking into account.

CHAPTER

TEN

Integration in the Salome environnement

In this chapter, we assume that you have installed correctly the KERNEL module of salome and thus you have defined the variable `KERNEL_ROOT_DIR`. We assume that you have `omniORB` and its python wrapper correctly installed (it should be the case if you have installed KERNEL).

If those conditions are satisfied and if you have installed the TECHOBJ template as explained in “Integration in a Python Graphical User Interface” chapter, you should have a new Salome module available without any more manipulation ... Moreover, this new module can be involved in batch mode or by the Salome IAPP (Interface Applicative)

10.1 Pure corba batch mode

This module can be called in pure corba style. See the file:

```
# --
# Copyright (C) CEA, EDF
# Author : Erwan ADAM (CEA)
# --

import unittest

from xsalome import XSalomeSession

class XDataTestTestCase(unittest.TestCase):
    def test(self):
        #
        salome = XSalomeSession(modules=["TECHOBJ"], logger=1)
        engine = salome.lcc.FindOrLoadComponent("FactoryServerPy", "TECHOBJ")
        self.failUnless(engine)
        from TECHOBJ_CORBA import TECHOBJ_Component
        engine = engine._narrow(TECHOBJ_Component)
        self.failUnless(engine)
        #
        radius = 1.0
        #
        circle = engine.newCircle(
            radius,
            )
        self.failUnless(circle)
        self.failUnlessEqual(circle.getR(), radius)
        #
        bounds = circle.getBounds()
        self.failUnlessEqual(bounds[0], -radius)
        self.failUnlessEqual(bounds[1], +radius)
        self.failUnlessEqual(bounds[2], -radius)
        self.failUnlessEqual(bounds[3], +radius)
        #
```

```
            mat = engine.newMaterial(
                engine.newXString("mmm"),
                )
            self.failUnlessEqual(mat.getName().value(), "mmm")
            #
            to = engine.newTechnologyObject(
                mat,
                circle,
                )
            #
            return
        pass

if __name__ == '__main__':
    unittest.main()
    pass
```

10.2 Wrapping corba batch mode

The previous way to call the module is quite complicated and the usual way in Salome to hide the technical lines is to write a client file which does the corba job (see for instance `geompy.py` or `smeshpy.py` for the GEOM and SMESH modules).

Indeed, in `xdata` module, the client classes are the python classes themselves !!! It's the same code with some flags at some points to ensure the compatibility pure python / corba. The advantage is that the same code can be called in the two modes without any modification. For instance, the test:

```
# --
# Copyright (C) CEA, EDF
# Author : Erwan ADAM (CEA)
# --

from circle import Circle

c = Circle(r=1.0)

assert c.r == 1.0

bounds = c.bounds

assert bounds == [-1.0, 1.0, -1.0, 1.0]

from material import Material

m = Material()

assert m.name == "m"

from technologyobject import TechnologyObject

to = TechnologyObject(material=m,
                      shape=c,
                      )

c = to.shape

assert c.r == 1.0
assert c.bounds == [-1.0, 1.0, -1.0, 1.0]
```

is called in pure python mode with

```
# --
# Copyright (C) CEA, EDF
# Author : Erwan ADAM (CEA)
# --

import unittest

class XDataTestTestCase(unittest.TestCase):
    def test(self):
        import TECHOBJusecase
        return
    pass

if __name__ == '__main__':
    unittest.main()
    pass
```

and in corba mode with

```
# --
# Copyright (C) CEA, EDF
# Author : Erwan ADAM (CEA)
# --

import unittest

class XDataTestTestCase(unittest.TestCase):
    def test(self):
        from xsalome import XSalomeSession
        salome = XSalomeSession(modules=["TECHOBJ"], logger=1)
        #
        import TECHOBJusecase
        # --
        # Testing the hidden corba objects
        to = TECHOBJusecase.to
        self.failIfEqual(to.__corba__component__, None)
        to = to.__corba__object__
        shape = to.getShape()
        bounds = shape.getBounds()
        self.failUnlessEqual(bounds, [-1.0, 1.0, -1.0, 1.0])
##        # --
##        # Testing if xtype accept corba objects
##        # since it is needed from salome gui ...
##        from technologyobject import TechnologyObject
##        xattrs = TechnologyObject.getAllInitXAttributes()
##        shape_xattr = None
##        for x in xattrs:
##            if x.name == "shape":
##                shape_xattr = x
##                break
##            pass
##        xtype = x.xtype
##        xtype(shape)
        #
        return
    pass

if __name__ == '__main__':
    unittest.main()
    pass
```

10.3 Interaction via the Salome Interface Applicative

If TECHOBJ is correctly installed and you have defined the `TECHOBJ_ROOT_DIR` variable to point your installation path, you can run the Salome GUI via a command like:

```
$KERNEL_ROOT_DIR/bin/salome/runSalome --modules=TECHOBJ --logger=1
```

This should launch a salome window with the TECHOBJ module inside. Just click the module to activate it.

10.4 Interaction via the Salome Supervision

Important: Redhat 9 users should define a good LD_ASSUME_KERNEL variable to use old linux threads implementation:

```
setenv LD_ASSUME_KERNEL 2.4.18-4
```

To launch Salome with the supervision to be available, you can use a line like:

```
$KERNEL_ROOT_DIR/bin/salome/runSalome --modules=TECHOBJ,SUPERV --logger=1
```

A window containing TECHOBJ and SUPERV should appears. Please, refer to the supervision documentation to see how it works !

CHAPTER

ELEVEN

Interaction with other Salome Modules

11.1 MED salome module

The xdata module provides a way to interact with med objects in pure python or in the salome environment with the same lines of code. The template MEDFIELDCREATOR shows an exemple of that.

The sources of the template are included in the directory XDATA_INSTALLATION_PATH/share/xdata/templates. The example below is used in compiling and installing the MEDFIELDCREATOR template. To do that, you should copy the MEDFIELDCREATOR_SRC arborescence from XDATA_INSTALLATION_PATH/share/xdata/templates in a local directory to not ¡¡pollute¿¿ your xdata installation. Then, follow the instructions in the README file of MEDFIELDCREATOR_SRC

This file is reproduced here :

```
# --
# Copyright (C) CEA, EDF
# Author : Erwan ADAM (CEA)
# --

Introduction
============

You are in the template MEDFIELDCREATOR which use the xdata python
module for developpement. We assume as this level that xdata
is correctly installed on your system.

Building and Installing
=======================

cd MEDFIELDCREATOR_SRC
cd ..
mkdir MEDFIELDCREATOR_BUILD
cd MEDFIELDCREATOR_BUILD
../MEDFIELDCREATOR_SRC/build_configure
../MEDFIELDCREATOR_SRC/configure --prefix=INSTALLATION_PATH
make install
make check (optional)

After all, please define a MEDFIELDCREATOR_ROOT_DIR variable
to refer your INSTALLATION_PATH :

setenv MEDFIELDCREATOR_ROOT_DIR INSTALLATION_PATH (csh)
export MEDFIELDCREATOR_ROOT_DIR=INSTALLATION_PATH (sh)

Troubleshooting
===============

Please, send a mail to erwan.adam@cea.fr ...
```

In this template, we have defined only one class which is used to receive a med mesh at initialisation, and to product a field on this mesh. The code source is reproduced here:

```
# --
# Copyright (C) CEA, EDF
# Author : Erwan ADAM (CEA)
# --

from xdata import *

from libMEDClient import *

class MedFieldCreator(XNamedObject):
    __init__xattributes__ = [
        XAttribute("mesh", xtype=XInstance(MESH)),
        ]
    __object__xmethods__ = [
        XMethod("run"),
        ]
    __object__xattributes__ = [
        XAttribute("field", xtype=XInstance("libMEDClient.FIELD_")),
        ]

    def run(self):
        mesh = self.mesh
        support = SUPPORT(mesh, "my_support", MED_CELL)
        field = FIELDDOUBLE(support, 1)
        # Keep a ref (avoid garbage of local support)
        field.__ref__support__ = support
        field.setName("my_field")
        #
        from time import sleep
        print "very complex computation ... patience"
        sleep(3)
        #
        nb_vals = support.getNumberOfElements(MED_ALL_ELEMENTS)
        vals = [ float(i)/(nb_vals-1) for i in range(nb_vals) ]
        field.setValue(vals)
        self.field = field
        return

    pass
```

11.1.1 Pure python mode

Users can interact with med in pure python as in the following use case

```
# --
# Copyright (C) CEA, EDF
# Author : Erwan ADAM (CEA)
# --

import unittest

class TestCase(unittest.TestCase):
    def test(self):
        from os import getenv
        medfile  = getenv("MED_ROOT_DIR")
        medfile += "/share/salome/resources/med/pointe.med"
        #
        from libMEDClient import MED, MED_DRIVER
        #
        med = MED(MED_DRIVER, medfile)
        mesh = med.getMesh(med.getMeshName(0))
        mesh.read()
        #
        import medfieldcreator
        mfc = medfieldcreator.MedFieldCreator(mesh)
        mfc.run()
        field = mfc.getField()
        print field
        print dir(field)
        support = field.getSupport()
        print support
        print dir(support)
        mesh = support.getMesh()
        print mesh
        print dir(mesh)
        return
    pass

if __name__ == '__main__':
    unittest.main()
    pass
```

11.1.2 Salome batch mode

Users can interact with med in salome batch mode as in the following use case

```
# --
# Copyright (C) CEA, EDF
# Author : Erwan ADAM (CEA)
# --

import unittest

class TestCase(unittest.TestCase):
    def test(self):
        #
        from xsalome import XSalomeSession
        salome = XSalomeSession(modules=["MED","MEDFIELDCREATOR"],
                                logger=1,
                                study=1,
                                )
        #
        # Sorry, but I don't know how to use the MED component
        # in batch mode without opening a study !!!
        #
        study_name = "test"
        study = salome.study_manager.NewStudy(study_name)
        #
        med_gen = salome.lcc.FindOrLoadComponent("FactoryServer", "MED")
        self.failUnless(med_gen)
        from SALOME_MED import MED_Gen
        med_gen = med_gen._narrow(MED_Gen)
        self.failUnless(med_gen)
        #
        from os import getenv
        medfile  = getenv("MED_ROOT_DIR")
        medfile += "/share/salome/resources/med/pointe.med"
        #
        print medfile
        med = med_gen.readStructFile(medfile, study_name)
        mesh_names = med.getMeshNames()
        self.failUnlessEqual(mesh_names, ['maa1'])
        mesh = med.getMeshByName(mesh_names[0])
        #
        import medfieldcreator
        mfc = medfieldcreator.MedFieldCreator(mesh)
        mfc.run()
        field = mfc.getField()
        print field
        support = field.getSupport()
        print support
        mesh = support.getMesh()
        print mesh
        #
        xattr = medfieldcreator.MedFieldCreator.__init__xattributes__[0]
        xtype = xattr.xtype
        xtype(mesh)
        return
    pass

if __name__ == '__main__':
    unittest.main()
    pass
```

11.1.3 Interaction via the Salome Interface Applicative

If MEDFIELDCREATOR is correctly installed and you have defined the `MEDFIELDCREATOR_ROOT_DIR` variable to point your installation path, you can run the Salome GUI via a command like (Note that we use the visu module too):

```
$KERNEL_ROOT_DIR/bin/salome/runSalome --modules=MED,MEDFIELDCREATOR,VISU
--logger=1
```

A window containing MED, MEDFIELDCREATOR and VISU should appears.