```
R"""The Open-ORB encapsulation meta model
```

Author : Anders Andersen
Created On : Mon Jul 11 03:34:11 1998
Last Modified By: Anders Andersen
Last Modified On: Tue Mar 16 14:54:31 1999
Status : Unknown, Use with caution!

```
"""


# Need to do some type checking
import types

# Copy objects
import copy

# IRef, IObj
from lbind import IRef, IObj, IMethod


class EncapsException(Exception):
    R"""Encapsulation exception
```

All new exceptions or error-types introduced by the encapsulation module is handled by this exception class.

```
    """
    pass


class _UnboundMethod:
    R"""Unbounded methods
```

Use an instance of this class for new methods and methods with pre- and post-methods in an object.

```
    """

    # No pre- and post-methods initially
    premethods = []
    postmethods = []

    def __init__(self, object = None, method = None, key = None):
        R"""Initialize a bounded method
```

Save the object, the method key and the method. These values will be used to build a message for the method calls.

```
        """
        self.object = object
        self.method = method
        self.key = key

    def __call__(self, *args, **kw):
        R"""Call the bounded method
```

Call the method `self.method` with the possible arguments `args` or `kw`. Also call the pre- and post-methods if they exists.

```
        """

        # Build a message to pre- and post-methods
        msg = {'object': self.object, 'key': self.key, 'method': self.method,
               'args': args, 'kw': kw, 'result': None}

        # Call each pre-method
        for pre in self.premethods:
            pre(self.object, msg)

        # Call the actual method
        msg["result"] = self.apply(msg)

        # Call each post method
        for post in self.postmethods:
            post(self.object, msg)

        # Return result (possibly changed by the post-methods)
        return msg["result"]

    def apply(self, msg):
        R"""Do the actual method call

        Call the method with the right arguments (including self).

        """
        return apply(msg['method'], msg['args'], msg['kw'])


class _BoundMethod(_UnboundMethod):
    R"""Bounded methods

    Use an instance of this class for new methods and methods with pre- and post-methods in an object.

    """

    def apply(self, msg):
        R"""Do the actual method call

        Call the method with the right arguments (including self).

        """
        return apply(msg['method'], (msg['object'],) + msg['args'], msg['kw'])


class IgnoreAttr:
    R"""Attributes ignored when inspecting

    Attributes to ignore when inspecting an object or an interface. The class has 2 members: inObject is
    the list of attributes ignored when inspecting an object and inClass is list of attributes ignored when
    inspecting a class.

    """

    # Attributes ignored (hidden) while inspecting objects
    inObject = ['__doc__', '__module__']

    # Attributes ignored (hidden) while inspecting classes
    inClass = inObject


def _collectClassAttr(c, test, dict):
    R"""Collect selected attributes in a class

    Collects attributes in class c satisfying test. Collected attributes are inserted in the dictionary dict.

    """
    for (key, attr) in c.__dict__.items():
        if test(attr) and attr != _hiddenMethod:
            if not dict.has_key(key) and key not in IgnoreAttr.inClass:
                dict[key] = attr


def _collectAllClassAttr(c, test, dict):
    R"""Collect all selected attributes in a class
```

Collect all attributes in class `c` (also inherited) satisfying `test`. Collected attributes are inserted in the dictionary `dict`.

```
    """
    _collectClassAttr(c, test, dict)
    for base in c.__bases__:
        _collectAllClassAttr(base, test, dict)


def _hiddenMethod(*args, **kw):
    R"""A hidden method
```

A function inserted to hide (remove) a method.

```
    """
    raise AttributeError, 'Hidden method'


def _isAttrSubClass(o, name, c):
    R"""Is attribute of given class?
```

Tests if attribute `name` in object (or class) `o` exists and is of class `c`. The result is either true or false.

```
    """
    try:
        return (issubclass(o.__dict__[name].__class__, c))
    except:
        return 0


class _Proxy:
    R"""A proxy for an object
```

This class is used to make a proxy for an object with a metaobject. All access of the object is redirected to the metaobject through an instance of this class and the `_BoundMethod` wrapper for methods.

```
    """

    def __repr__(self):
        R"""Redirect repr to the metaobject
```

Returns a string representing the actual object.

```
        """
        return self.__meta__.repr()      # Corrected by Fabio Moreira Costa

    def __getattr__(self, key):
        R"""Redirect getattr to the metaobject
```

Get the value of an attribute through the metaobject. All attributes except the methods are accessed through this method.

```
        """
        return self.__meta__.getattr(key)

    def __setattr__(self, key, value):
        R"""Redirect setattr to the metaobject

        Set the value of an attribute through the metaobject. All attributes are given a new value through
        this method.

        """
        self.__meta__.setattr(key, value)

    def __delattr__(self, key):
        R"""Redirect delattr to the metaobject

        Deletes the attribute through the metaobject.

        """
        self.__meta__.delattr(key)


class Encaps:
    R"""The encapsulation meta object

    Hmmm

    """

    getAttrMethods = {}
    setAttrMethods = {}

    def __init__(self, o):
        R"""Initialize the metaobject

        Initialize the encapsulation meta object. Builds an environment for the object o.

        """

        # Test if the object allready has a metaobject
        if isinstance(o, _Proxy) and o.__dict__.has_key('__meta__'):
            raise EncapsException, 'Encaps: meta object exists'

        # Create a new object
        self.object = o
        inspected = self.inspectObject()
        self._ns = apply(inspected['class'], ())
        self._ns.__dict__ = inspected['vars']

        # Save name space and method space
        self._org = o
        if isinstance(o, IRef):
            self.object = self._ns.__local__["object"]
            self._ms = self._ns.__local__["iobj"]
        else:
            self.object = self._ms = self._ns

        # Make a proxy for the object
        o.__dict__ = {}
        o.__meta__ = self
        o.__class__ = _Proxy

    def inspect(self):
        R"""Inspect the object

        Returns the dictionary representing the inspected object or interface.

        """
        if isinstance(self._ns, IRef):
            return self.inspectInterface()
        else:
            return self.inspectObject()

    def inspectObject(self):
        R"""Inspect an object
```

Inspects the object of this meta object. The result is a dictionary with 4 members: `'class'` is the class of the object, `'vars'` are the attributes in the object (not including the class attributes), `'allattr'` is all the attributes of the object (not including methods, but including the class attributes) and `'exported'` is all methods for the object (including the inherited methods).

```
        """

        # Initialise (nothing found yet)
        methods = {}; attr = {}; ivars = {}

        # Collect methods from class
        _collectAllClassAttr(
            self.object.__class__, lambda a: type(a) is types.FunctionType,
            methods)

        # Collect attributes from class
        _collectAllClassAttr(
            self.object.__class__, lambda a: type(a) is not types.FunctionType,
            attr)

        # Collect attributes and methods in this object
        for (key, var) in vars(self.object).items():
            if not key in IgnoreAttr.inObject:
                if (hasattr(var, '__class__') and
                    issubclass(var.__class__, _UnboundMethod)):
                    methods[key] = var
                    ivars[key] = methods[key]
                elif var == _hiddenMethod:
                    del methods[key]
                else:
                    attr[key] = var
                    ivars[key] = attr[key]

        # Return the result as a dictionary
        return {
            'class': self.object.__class__, 'vars': ivars,
            'exported': methods, 'allattr': attr}

    def inspectInterface(self):
        exported = {}
        for m in self._ns.__expID__:
            exported[m] = self._ms.__dict__[m]
        return {
            'object': self.object,
            'exported': exported, 'imported': self._ns.__impID__}

    def repr(self):
        R"""Return a string representing the object
```

Returns a string representing the object.

```
        """
        return 'self.object'

    def getattr(self, key):
        R"""Get the value of an attribute

        Gets the attribute key from the object.

        """
        if self.getAttrMethods.has_key(key):
            for method in self.getAttrMethods[key]:
                apply(method, (self._ns, key))
        return getattr(self._ns, key)

    def setattr(self, key, value):
        R"""Set the value of an attribute

        Sets the attribute key in the object to the given value.

        """
        if self.setAttrMethods.has_key(key):
            for method in self.setAttrMethods[key]:
                apply(method, (self._ns, key, value))
        setattr(self._ns, key, value)

    def delattr(self, key):
        R"""Delete an attribute

        Deletes the attribute key in the object.

        """
        delattr(self._ns, key)

    def addGetAttr(self, key, function):
        if isinstance(self._ns, IRef):
            raise EncapsException, "addGetAttr: not available on interfaces"
        if not hasattr(self._ns, key):
            raise EncapsException, "addGetAttr: attribute doesn't exists"
        if self.getAttrMethods.has_key(key):
            self.getAttrMethods[key].append(function)
        else:
            self.getAttrMethods[key] = [function]

    def delGetAttr(self, key, function=None):
        if isinstance(self._ns, IRef):
            raise EncapsException, "delGetAttr: not available on interfaces"
        if self.getAttrMethods.has_key(key):
            if function:
                self.getAttrMethods[key].remove(function)
            else:
                self.getAttrMethods[key] = []
        else:
            raise EncapsException, "delGetAttr: unknown key"

    def addSetAttr(self, key, function):
        if isinstance(self._ns, IRef):
            raise EncapsException, "addSetAttr: not available on interfaces"
        if self.setAttrMethods.has_key(key):
            self.setAttrMethods[key].append(function)
        else:
            self.setAttrMethods[key] = [function]

    def delSetAttr(self, key, function=None):
        if isinstance(self._ns, IRef):
            raise EncapsException, "delSetAttr: not available on interfaces"
        if self.setAttrMethods.has_key(key):
            if function:
                self.setAttrMethods[key].remove(function)
            else:
                self.setAttrMethods[key] = []
        else:
            raise EncapsException, "delGetAttr: unknown key"

    def addMethod(self, name, function, override=0):
        R"""Add a method to an object
```

Adds a method with the key `name` and the implementation `function` to the object of this meta object. The first argument of the function should be the object it self (the `self` argument). If `override` is false (default) an exception will occur if a method with the key `name` exists.

```
        """

        # We have to check if the method allready exists (if not override)
        if not override:
            inspected = self.inspect()
            if inspected['exported'].has_key(name):
                raise EncapsException, \
                      'addMethodObject: method %s exists' % (name,)

        # Make an object which behaves as a method and add it to the object
        self._ms.__dict__[name] = _BoundMethod(
            object=self.object, method=function, key=name)
        if isinstance(self._ns, IRef):
            if not name in self._ns.__expID__:
                self._ns.__expID__.append(name)

    def delMethod(self, name, completely=0):
        R"""Delete a method from an object
```

Deletes a method with the key `name` from the object of this mete object. A `EncapsException` exception is raised if the method does not exists. If the completely argument is true (not default) the method will be hidden completely (also inherited methods with this key).

```
        """

        # Remove method from object if it exists
        if not completely:
            if self._ms.__dict__.has_key(name):
                del self._ms.__dict__[name]
            else:
                raise EncapsException, \
                      'delMethodObject: %s does not exists' % (name,)

        # Hide method completely
        else:
            inspected = self.inspect()
            if inspected['exported'].has_key(name):
                self._ms.__dict__[name] = _hiddenMethod
            else:
                raise EncapsException, \
                      'delMethodObject: %s does not exists' % (name,)

        # Update export info too
        if isinstance(self._ns, IRef):
            if name in self._ns.__expID__:
                expID = []
                for item in self._ns.__expID__:
```

```
                if item != name:
                    expID.append(item)
            self._ns.__expID__ = expID

    def _addPPMethod(self, name):
        R"""Prepare for pre- and post-methods
```

Create a _BoundMethod replacement for the function `name` if it is not allready there.

```
        """
        if _isAttrSubClass(self._ms, name, _UnboundMethod):
            return

        method = None
        if isinstance(self._ns, IRef):
            if self._ms.__dict__.has_key(name):
                if isinstance(self._ms.__dict__[name], IMethod):
                    cls = _UnboundMethod
                    method = self._ms.__dict__[name]
        else:
            methods = {}
            _collectAllClassAttr(
                self.object.__class__, lambda a: type(a) is types.FunctionType,
                methods)
            if methods.has_key(name):
                cls = _BoundMethod
                method = methods[name]
        if method:
            self._ms.__dict__[name] = apply(cls, (self.object, method, name))
        else:
            raise EncapsException, \
                  '_addPPMethod: %s does not exists' % (name,)

    def addPreMethod(self, name, function):
        R"""Add a pre-method to a method
```

Adds a pre-method with the implementation `function` for the method with the key `name` in the object of the meta object. The new method will be inserted first in the list of pre-methods.

```
        """

        # Preprocess
        self._addPPMethod(name)

        # OK, this should now be a _BoundMethod
        if _isAttrSubClass(self._ms, name, _UnboundMethod):
            self._ms.__dict__[name].premethods = (
                [function] + self._ms.__dict__[name].premethods)

    def delPreMethods(self, name, function=None):
        R"""Delete pre-methods from a method
```

Deletes the pre-methods of the method with the key `name` in the object of the meta object. If `function` is given, only the given function is deleted. Otherwise all pre-methods are deleted.

```
        """
        if _isAttrSubClass(self._ms, name, _UnboundMethod):

            # Delete pre-methods (one if function given)
            if function:
                self._ms.__dict__[name].premethods.remove(function)
            else:
```

```
            self._ms.__dict__[name].premethods = []

        # It has to be a _BoundMethod object
        else:
            raise EncapsException, 'delPreMethods: wrong attribute type'

    def addPostMethod(self, name, function):
        R"""Add a post-method to a method

        Adds a post-method with the implementation function for the method with the key name in the
        object of the meta object. The new method will be appended last to the list of post-methods.

        """

        # Preprocess
        self._addPPMethod(name)

        # OK, this should now be a _BoundMethod
        if _isAttrSubClass(self._ms, name, _UnboundMethod):
            self._ms.__dict__[name].postmethods.append(function)

    def delPostMethods(self, name, function=None):
        R"""Delete post-methods from a method

        Deletes the post-methods of the method with the key name in the object of the meta object. If
        function is given, only the given function is deleted. Otherwise all the post-methods are deleted.

        """

        if _isAttrSubClass(self._ms, name, _UnboundMethod):

            # Delete post-methods (one if function given)
            if function:
                self._ms.__dict__[name].postmethods.remove(function)
            else:
                self._ms.__dict__[name].postmethods = []

        # It has to be a _BoundMethod object
        else:
            raise EncapsException, 'delPostMethods: wrong attribute type'

    def changeClass(self, newclass):
        R"""Change the class

        Change the class of the object to newclass.

        """
        if isinstance(self._ns, IRef):
            raise EncapsException, 'changeClass: not allowed on interfaces'
        else:
            self.object.__class__ = newclass

    def changeObject(self, newobject):
        if isinstance(self._ns, IRef):
            self.object = newobject
            self._ns.__local__ = {}
            self._ns.__local__["object"] = newobject
            self._ns.__testExpInterface__(self._ns.__expID__)
        else:
            raise EncapsException, 'changeObject: only on interfaces'

    def restore(self):
        self._org.__class__ = self._ns.__class__
        self._org.__dict__ = self._ns.__dict__


def encapsulation(o):
    R"""Get the meta object of an object or interfaces

    Returns the meta object of the object or interfaces o. A new metaobject is created on the fly.

    """
    return Encaps(o)


def restore(o):
    R"""Restore an object

    Removes the metaobject from the object o.

    """
    if o.__dict__.has_key('__meta__'):
        o.__meta__.restore()
    else:
        raise EncapsException, "restore: nothing to restore"
```