1 Introduction

This document describes how one of the example programs (Scala.py) works, to help people trying to develop program wrappers. This assumes a decent working knowledge of Python, and the availablility of a Python 2.4 interpreter (with the win32api module on windows.)

2 Example Program: CCP4

2.1 Introduction

This program makes use of the CCP4-ified Driver interface - this provides automated handling and so on of the "standard" CCP4 keywords - HKLIN and friends.

2.2 The Code

```
#!/usr/bin/env python
# Scala.py
# Maintained by G.Winter
# 31st May 2006
#
# An illustration wrapper for the program scala.
#

import os
import sys

if not os.environ.has_key('XIA2CORE_ROOT'):
    raise RuntimeError, 'XIA2CORE_ROOT not defined'

sys.path.append(os.path.join(os.environ['XIA2CORE_ROOT'],
                             'Python'))

from Driver.DriverFactory import DriverFactory
from Decorators.DecoratorFactory import DecoratorFactory

def Scala(DriverType = None):
    '''Create a Scala instance based on the requirements proposed in the
    DriverType argument.'''

    DriverInstance = DriverFactory.Driver(DriverType)
    CCP4DriverInstance = DecoratorFactory.Decorate(DriverInstance, 'ccp4')

    class ScalaWrapper(CCP4DriverInstance.__class__):
        '''A wrapper for Scala, using the CCP4-ified Driver.'''

        def __init__(self):
            # generic things
            CCP4DriverInstance.__class__.__init__(self)
            self.setExecutable('scala')

            self._scalepack = None

        def setScalepack(self, scalepack):
            self._scalepack = scalepack

        def scale(self):
            self.setTask('Scale reflections from %s to %s' % \
                         (os.path.split(self.getHklin())[-1],
                          os.path.split(self.getHklout())[-1]))

            self.checkHklin()
            self.checkHklout()

            if self._scalepack:
                self.addCommand_line('SCALEPACK')
                self.addCommand_line(self._scalepack)

            self.start()
```

```
            if self._scalepack:
                self.input('output polish unmerged')

            self.input('resolution 1.65')
            self.input(
                'scales rotation spacing 5 secondary 6 bfactor on tails')
            self.input('cycles 20')
            self.input('anomalous on')
            self.input('sdcorrection full 1.0 15.0 0.02')
            self.input('sdcorrection partial 1.0 15.0 0.00')

            self.close_wait()

            return self.get_ccp4_status()

    return ScalaWrapper()

if __name__ == '__main__':

    # this must be run after the Sortmtz.py example

    hklin = '12287_1_E1_sorted.mtz'
    hklout = '12287_1_E1_scaled.mtz'

    s = Scala()

    s.setHklin(hklin)
    s.setHklout(hklout)

    print s.describe()

    status = s.scale()
    s.write_log_file('scala.log')

    print '%s' % (status)
```

2.3 Discussion

This example is poor in the sense that all of the parameters are hard-coded. However, the overall principles are sound. The first few lines represent a header. This brings the DriverFactory and DecoratorFactory into scope. The idea here is that you get a Driver implementation from the DriverFactory, possibly qualified by the DriverType (e.g. "script" or "simple".) This Driver instance is then "decorated" to make it a CCP4Driver, by adding HKLIN, HKLOUT etc handling methods. This is performed by the DecoratorFactory.Decorate() method. At this stage you then have a class to inherit from in your wrapper implementation.

All of this is implemented in a factory method to allow the dynamic inheritance to work - in effect making the class hierarchy decidable at runtime.

Inside the class all is sensible: the constructor calls the generated classes constructor first, then configures itself (this example has only one option.) At this stage it is appropriate to configure the executable name.

Inside the scale() method a description is provided as to what is happening. This could be printed out once the job has finished along with the return status as an indicator of progress. The values of the required files is then checked (this should probably be done first!) and finally the input written to the child program.

The close_wait() method closes the standard input and then waits for the output to finish. Finally the get_ccp4_status looks for the record towards

the end of the file written with the termination status.