1 Version Info

Document version: `$Id: X2C.tex,v 1.17 2006/09/01 10:46:32 gwin Exp $`

2 Introduction

X2C - the XIA 2 core. This document is being written as a plan of how to design the new core, to ensure maximum functionality and minimum development cost.

2.1 The Core

The core is defined to be those components of XIA which are absolutely necessary to getting automated processes working. In the context of the old XIA design, this would correspond to the Driver and Output classes, though others are likely to appear.

3 Requirements

During development of the old XIA Driver core, it became apparent that there were a number of environmental problems with the design. In particular

- Changing the system to work in different environments (e.g. cluster vs. workstation) was tricky.
- Porting the code, as it stood, to alien platforms such as Microsoft Windows was too hard to consider.
- Customising classes to work more optimally for certain families of programs (for instance the CCP4 suite) was also tricky.

Clearly it is possible and worthwhile to factor these design requitements into a new implementation of the core.

There are a small number of additional requirements:

- Ability to set a default working directory for new Driver instances - this would be a method of the DriverFactory then.
- A module for trapping "standard errors" e.g. bus error, segv etc. A "CCP4" version of this could also be implemented which will trap standard CCP4 formatted errors.
- Easy running of jobs on a cluster e.g. via QRSH (already planned). Use of this could also be configured through the DriverFactory.

- Caching Driver.input() records - this has already been described though under "methods of running". In particular want to be able to get a dump of the commands issued when a program failed.

These should be integrated into the rest of the plans.

4 Implementation Plans

4.1 Platforms & Languages

The following platforms are planned for support in the first iteration of the redesign:

- GNU/Linux
- Microsoft Windows
- Macintosh OS X

and developed in the following languages (in order of priority):

- Python (2.4+)
- Java (1.4.2+)
- C++ (ANSI)

to provide maximum coverage and applicability of the software. Once the design is sorted, and the test cases are available, porting to different languages should not be difficult.

4.2 Test Cases

During development of the original XIA, I found a number of *features* of programs which are run which need to be trapped. To this end I have written programs which illustrate these features before starting on development of the new core, so that we can be sure that they will be trapped *by design*. In particular, there are the following programs (available identically on Windows and Linux) which are self explanitory:

- `ExampleProgram` - a "normal" program.
- `ExampleProgramCommandLine` - a program which reads a token from the command line.
- `ExampleProgramLooseLoop` - a program which gets stuck in a loop where output is written to the screen each iteration.

- `ExampleProgramRaiseException` - a program which raises an exception as soon as it is run, similar to a missing dynamic library.
- `ExampleProgramStandardInput` - a program which reads tokens from the standard input.
- `ExampleProgramTightLoop` - a program which gets stuck in a tight loop and will need to be killed.

Between them, these represent many of the things which should be trapped by the Driver system (e.g., typical problems not those which are specific to the "class" of programs being run).

4.3 Compiled Test Cases

- `EPAbrt` - a program where the "abort" signal is raised.
- `EPKill` - a program where the "kill" signal is raised.
- `EPBus` - a program where a bus error happens.
- `EPSegv` - a program where a segmentation fault happens.
- `EPLib` - a program with a missing library.

These are hopefully reasonably portable, though will require compiling (makefile, anyone?) and give the following output:

```
./EPAbrt
Aborted

./EPBus
Bus error

./EPKill
Killed

./EPLib
./EPLib: error while loading shared libraries:
EPLib2.so: cannot open shared object file: No such file or directory

./EPSegv
Segmentation fault
```

So these should help fix common problems.

5 Configuration

Any program will require some configuration. A set of libraries for automation will require more than most, in particular the desired mode of operation and the location where the separate components or modules can be found. Often the enviroment is used for passing around this kind of information, and in this case this will be a perfectly portable and simple way of working.

So far only one environment variable has been defined

```
XIA2CORE_ROOT=/path/to/XIACore
```

which is thus to allow for versioning of the separate modules. The name "`XIA2CORE_ROOT`" indicates "XIA 2 core root directory". This will set up enough that the setup script within will set the environment CLASSPATH, LD_LIBRARY_PATH and PYTHONPATH appropriately.

This is, in my humble opinion, the least intrusive way of working. Setup scripts in this directory are already available for `(t)csh`, `(ba)sh` and Windows `cmd.exe`. Other shells could be supported, but will probably boil down to one of these three.

6 Customization

In some cases there will be families of programs (e.g. CCP4) where a set of standard options exist. In these cases the solution previously was to develop a new class inherited from a concrete implementation of the Driver. Since there will no longer be a concrete implementation of Driver, another solution is needed. Probably the best thing to do in this case is to develop a new Decorator class which will inherit from the Driver *interface* and provide extra functionality - in Python terms equivalent to:

```
def __init__(self):
    DriverFactory(''cluster'').__init__(self)
    DecoratorFactory(''ccp4'').__init__(self)
```

where everything comes from Factory classes to provide maximum flexibility. This factory could also be configured from external defaults for instance so that all Driver instances were configured to work on a cluster or via script running.

FIXME: Need to establish if it is possible to have multiply-implemented decorators, for instance decorating a driver as a CCP4 Driver and then also a FrameProcessorDriver. I suspect that this is not do-able. This could therefore be a problem with the initial design... The problem here is not with the initial design, but the implementation of the decorator pattern in Python.

More FIXME: It may prove to be worth changing the pattern used here from Decorator to Proxy, and delegating the request to the "parent" or contained object. This would have the advantage of working with the created INSTANCE of a Driver, and so may avoid some of the more nasty smelling spells needed to make the decorator patterns work. Perhaps this is a discussion for the CCP4 automation/developer list?

More FIXME: Implementation as a proxy would be relatively straightforward - could define the python stuff to, if an attribute is missing, look at the "real" object for that attribute and return such. I am not sure how well this would work though. Sounds easy enough... and would probably be implementable in the current framework *without* changing any of the actual application code. This could use self.__setattr__ and self.__getattr__ methods to implement the required functionality. For example

```
class C proxies D:
  def __getattr__(self, attr):
    if hasattr(self.D, attr):
      return getattr(self.D, attr)
    raise AttributeError, whatever
```

This would get around the problem reasonably tidily and would certainly work for multiply decorated classes. Since this is all accessed through the factory, is it reasonable to presume that this can be a code-neutral (at the application layer) change? Though this will probably alter the way in which the application wrappers are written, so it isn't strictly core-neutral then... Test this out.

Yes, this will involve changes to how the factories work - is this a good thing? How often will this be a problem where inheritance won't help? Why is this needed in the first place?

FIXME: Also found a buildin function in Python 2.3+ called issubclass() - this could be useful.

7 Python Implementation Notes

7.1 Introduction

The initial development focus will be on developing a portable Python implementation of the X2C. Thus, it is worth hunting around to see what features exist to provide this portable functionality.

7.2 Driving Programs

New to Python 2.4 is a module called "subprocess" - sounds appropriate. It includes a class called Popen, which claims to be portable to Windows, Mac and UNIX, and appears to have functionality to poll and get return values

- useful. Under UNIX the return values will even indicate what SIGNAL killed the process if appropriate, returning -SIGNAL.

Need to test this out somewhat.

The api for the subprocess module is:

```
class Popen(args, bufsize=0, executable=None,
            stdin=None, stdout=None, stderr=None,
            preexec_fn=None, close_fds=False, shell=False,
            cwd=None, env=None, universal_newlines=False,
            startupinfo=None, creationflags=0)
```

This gives a Popen instance with the following methods and attributes: Instances of the Popen class have the following methods:

- poll() Check if child process has terminated. Returns returncode attribute.
- wait() Wait for child process to terminate. Returns returncode attribute.
- communicate(input=None) Interact with process: Send data to stdin. Read data from stdout and stderr, until end-of-file is reached. Wait for process to terminate. The optional input argument should be a string to be sent to the child process, or None, if no data should be sent to the child.

  communicate() returns a tuple (stdout, stderr).

  Note: The data read is buffered in memory, so do not use this method if the data size is large or unlimited.
- stdin If the stdin argument is PIPE, this attribute is a file object that provides input to the child process. Otherwise, it is None.
- stdout If the stdout argument is PIPE, this attribute is a file object that provides output from the child process. Otherwise, it is None.
- stderr If the stderr argument is PIPE, this attribute is file object that provides error output from the child process. Otherwise, it is None.
- pid The process ID of the child process.
- returncode The child return code. A None value indicates that the process hasn't terminated yet. A negative value -N indicates that the child was terminated by signal N (Unix only).

PIPE above refers to the symbol in subprocess e.g. subprocess.PIPE. Likewise subprocess.STDOUT.

7.3 KILL

The os.kill() method only exists on UNIX not windows, so I can't just rely on this. Hence I will need to implement a switch of some kind, which uses os.kill() on unix and

```
kill = subprocess.Popen("taskkill /PID %i" % w.pid, shell=True)
```

otherwise. Not sure if this is available on all windows versions though - I suspect that it doesn't exist on XP HOME. There are other alternatives from the ASPN Python Cookbook:

```
# Create a process that won't end on its own
import subprocess
process = subprocess.Popen(['python.exe', '-c', 'while 1: pass'])


# Kill the process using pywin32
import win32api
win32api.TerminateProcess(int(process._handle), -1)


# Kill the process using ctypes
import ctypes
ctypes.windll.kernel32.TerminateProcess(int(process._handle), -1)


# Kill the proces using pywin32 and pid
import win32api
PROCESS_TERMINATE = 1
handle = win32api.OpenProcess(PROCESS_TERMINATE,
                              False, process.pid)
win32api.TerminateProcess(handle, -1)
win32api.CloseHandle(handle)


# Kill the proces using ctypes and pid
import ctypes
PROCESS_TERMINATE = 1
handle = ctypes.windll.kernel32.OpenProcess(PROCESS_TERMINATE,
                                            False, process.pid)
ctypes.windll.kernel32.TerminateProcess(handle, -1)
ctypes.windll.kernel32.CloseHandle(handle)
```

7.4 Unit Tests

There are a number of example programs (listed above.) These need to have associated unit tests written using the Python unittest module to ensure that the resulting Driver implementations work as described.

8 The Python 2.4 Implementation

8.1 Introduction

This section describes the python implementation after the event - that is, it is now written. One comment - to provide the Decorator like functionality in the absence of interfaces in Python, this relies heavily on Factories to produce the objects. This is of particular importance in the CCP4Decorator.

8.2 API

8.2.1 Driver Pydoc

```
class DefaultDriver:
  A class to run other programs, specifically from the CCP4 suite
  but also others, to achieve crystallographic processes. This will also
  provide functionality for controlling the job, limited only by the
  needs of portability across Windows, Macintosh OS X and Linux.

  Methods defined here:

  __del__(self)

  __init__(self)
      Initialise the Driver instance.

  addCommand_line(self, command_line_token)
      Add a token to the command line.

  addInput_file(self, input_file)

  addOutput_file(self, output_file)

  check(self)
      Check that the running process is ok - this is an optional
      interface which may not be defined for some implementations of
      Driver. Returns True if children are all ok, False otherwise.

  check_for_errors(self)
      Work through the standard output of the program and see if
      any standard error conditions (listed in DriverHelper) can be
      found. This will raise an appropriate exception if an error
      is found.

  clearCommand_line(self)
      Clear the command line.

  close(self)
      Close the standard input channel.

  close_wait(self)
      Close the standard input channel and wait for the standard
      output to stop. Note that the results can still be obtained through
      self.get_all_output()...

  describe(self)
      Give a short description of what this job will do...

  finished(self)
      Check if the program has finished.

  getExecutable(self)
      Get the name of the executable.
```

```
  getTask(self)
      Return a helpful record about what the task is doing.

  get_all_output(self)
      Return all of the output of the job.

  input(self, record, newline=True)
      Pass record into child program via _input & copying mechanism.

  kill(self)
      Kill the child process.

  output(self)
      Pull a record from the child program via _output.

  setCommand_line(self, command_line)
      Set the command line which wants to be run.

  setExecutable(self, executable)
      Set the name of the executable. Note that since 1/SEP/06 this will
      raise an exception if the executable isn't found in the path.

  setTask(self, task)
      Set a helpful record about what the task is doing.
```

Use of this interface can be seen in XIA2CORE_ROOT/Python/Examples/SHELX.

8.2.2 CCP4Decorator Pydoc

This example was instantiated inheriting from the ScriptDriver - SimpleDriver and InteractiveDriver are very similar. This class is what you may use if you want to run CCP4 programs with the minimum of fuss.

```
class CCP4Decorator(ScriptDriver.ScriptDriver):
  A decorator class for a Driver object which will add some nice CCP4
  sugar on top.

  Method resolution order:
      CCP4Decorator
      ScriptDriver.ScriptDriver
      DefaultDriver.DefaultDriver

  Methods defined here:

  __init__(self)

  checkHklin(self)

  checkHklout(self)

  checkMapin(self)

  checkMapout(self)

  checkXyzin(self)

  checkXyzout(self)

  describe(self)
      An overloading of the Driver describe() method.

  getHklin(self)

  getHklout(self)

  getMapin(self)

  getMapout(self)

  getXyzin(self)

  getXyzout(self)

  get_ccp4_status(self)
      Check through the standard output and get the program
      status. Note well - this will only work if called once the
      program is finished.
```

```
setHklin(self, hklin)

setHklout(self, hklout)

setMapin(self, mapin)

setMapout(self, mapout)

setXyzin(self, xyzin)

setXyzout(self, xyzout)

start(self)
    Add all the hklin etc to the command line then call the
    base classes start() method.

----------------------------------------------------------------------
Methods inherited from ScriptDriver.ScriptDriver:

check(self)
    NULL overloading of the default check method.

close(self)
    This is where most of the work will be done - in here is
    where the script itself gets written and run, and the output
    file channel opened when the process has finished...

kill(self)
    This is meaningless...

setName(self, name)
    Set the name to something sensible.

----------------------------------------------------------------------
Methods inherited from DefaultDriver.DefaultDriver:

__del__(self)

addCommand_line(self, command_line_token)
    Add a token to the command line.

addInput_file(self, input_file)

addOutput_file(self, output_file)

check_for_errors(self)
    Work through the standard output of the program and see if
    any standard error conditions (listed in DriverHelper) can be
    found. This will raise an appropriate exception if an error
    is found.

clearCommand_line(self)
    Clear the command line.
close_wait(self)
    Close the standard input channel and wait for the standard
    output to stop. Note that the results can still be obtained through
    self.get_all_output()...

finished(self)
    Check if the program has finished.

getExecutable(self)
    Get the name of the executable.

getTask(self)
    Return a helpful record about what the task is doing.

get_all_output(self)
    Return all of the output of the job.

input(self, record, newline=True)
    Pass record into child program via _input & copying mechanism.

output(self)
    Pull a record from the child program via _output.

setCommand_line(self, command_line)
    Set the command line which wants to be run.

setExecutable(self, executable)
    Set the name of the executable.

setTask(self, task)
```

```
      Set a helpful record about what the task is doing.

  setWorking_directory(self, working_directory)
      Set the working directory for this process.

  getWorking_directory(self)
      Get the working directory for this process.

  status(self)
      Check the status of the child process - implemented by _status
      in other Driver implementations.

  write_log_file(self, filename)

------------ FOR INFORMATION -------------

The above was generated by:

if __name__ == '__main__':
    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',
                                 'Driver'))

    from DriverFactory import DriverFactory

    # for this example inherit through the ScriptDriver
    d = DriverFactory.Driver('script')

    # this is usually accessed through DecoratorFactory(d, 'ccp4')
    d = CCP4DecoratorFactory(d)

    # generate documentation
    from pydoc import help
    print help(d.__class__)
```

Use of this interface can be seen in XIA2CORE_ROOT/Python/Examples/CCP4.

Since this was written an additional method - parse_ccp4_loggraph() has been added which will return the loggraphs from the program as a dictionary:

```
results = s.parse_ccp4_loggraph()
results[table name] = {'columns':column labels
                       'data':list of [list of column entries]}
                              as strings (e.g. not typed)
```

8.2.3 CCP4Decorator Spells

This relies heavily on the idea of a factory. Firstly a DecoratorFactory is used to instantiate the Decorator. This uses the following code to implement this pattern:

```
def CCP4DecoratorFactory(DriverInstance):
    '''Create a CCP4 decorated Driver instance - based on the Driver
    instance which is passed in. This is an implementation of
    dynamic inheritance. Note well - this produces a new object and
    leaves the original unchanged.'''

    DriverInstanceClass = DriverInstance.__class__

    # verify that the input object satisfies the Driver interface -
    # in this case that at some point it inherited from there.
    # note well - instances of DefaultDriver must not be used
    # directly.

    DriverInstanceBaseClasses = []

    for b in DriverInstanceClass.__bases__:
        DriverInstanceBaseClasses.append(b.__name__)

    if not 'DefaultDriver' in DriverInstanceBaseClasses:
        raise RuntimeError, 'object %s is not a Driver implementation' % \
              str(DriverInstance)
```

```
    # verify that the object matches the Driver specification

    class CCP4Decorator(DriverInstanceClass):
        '''A decorator class for a Driver object which will add some nice CCP4
        sugar on top.'''

        # to get hold of the original start() methods and so on - and
        # also to the parent class constructor
        _original_class = DriverInstanceClass

        def __init__(self):
            # note well - this is evil I am calling another classes constructor
            # in here. Further this doesn't know what it is constructing!

            self._original_class.__init__(self)
```

Unfortunately this also cascades down to the use of this class in applications. The client must instantiate the class as follows:

```
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')
```

8.3 Exceptions

There are a few places in the Driver code and associated bits where exceptions are raised. These should probably be implemented as DriverExceptions, DecoratorExceptions etc., to provide stronger typing when exceptions occur.

To-do: This actually needs implementing as a set of exception classes, probably something like DriverException, DecoratorException, subclassed to CCP4DecoratorException or something.