1 Introduction

The structure of the X2 Driver is a little more sophisticated that the previous implementation. Here the objective is to define a standard Driver interface, which provides all of the necessary fuctions, and a set of Driver implementations which satisfy this interface. Instances of these can be obtained from a DriverFactory which decides at call time what is needed.

On top of this standard Driver implementation, an idea is to provide customised Driver instances by using dynamic inheritance and something a little like the Decorator pattern from the GoF book. For instance, adding hklin methods and the like.

2 Structure

The class structure looks like...

```
     DefaultDriver (defines interface, basic API)

      /         \

ScriptDriver   SimpleDriver ..... (implementations of Driver interface
                                   accessed through a DriverFactory)
      \         /

      CCP4Decorator (which adds the CCP4 stuff - other decorators
                     are possible though. accessed through the
                     DecoratorFactory)
```

3 Access

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

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

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

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

        def __init__(self):
            CCP4DriverInstance.__class__.__init__(self)
            etc...
```

That is:

- Bring factories into scope.
- Define a new factory to make MyClass objects.
- Create a driver and dress it in CCP4 clothes.
- Extend for your particular application.

This example talks CCP4 - other decorators for other suites are equally valid.

4 Implementation for Clusters

Running programs on clusters will require support for a fairly large number of batch queuing systems. These could be individually implemented as separate Driver implementations, which are then all accessed through the DriverFactory. A more elegant solution, however, would be to subclass the Driver interface to provide general "cluster friendly" functionality (i.e. that which is required across all platforms) and then subclass *this* for different batch systems. This is best done through a ClusterDriverFactory, which could be delegated by the DriverFactory to produce Driver instances in cluster environments.

4.1 Supported Clusters

There are a number of clustering packages out there, but in the first instance I am looking to support Sun Grid Engine and Condor. These share much of the same facility, mostly differing by how the job is submitted.

Slight challenge with this - if the program being run on windows is a script, the batch file will escape once you have run the job - the next command in the batch file won't be called. This is a complication. However, on windows we canot get the status out anyway, which suggests that this is not really worth worrying about!

Classes implementing the ClusterDriver interface should overload the sibmit() method, which is used to add the job to a queue. Should also define cleanup() to delete the queue submission specific files, for example the script.sh.o1234 file for SGE.