1. INTRODUCTION

In response to the wishes of the 'users' of the SPIRE instrument and the inevitable accretion of developers I have written this short note on how to code up a simple task using Java and its usage within a Jython interactive environment. It is especially relevant to those who are not over-familiar with either languages but have some programming experience. I have provided a complete example to start the coding process off. One thing will become clear from this example; it doesn't do anything. The meat (the execute() method) is devoid of mathematical manipulation. The intention is to teach the developer the syntax for inputting a product, unpacking the data contained within, processing the data and re-packing the product ready for further processing. I've included some commented out code which you can play with but be warned the code I've left in is pretty meaningless.

I have pitched this document at a non-software engineer so hopefully it will be jargon-light.

1.1 Task

A Task is a class within the HCSS which provides a uniform infrastructure for coding data processing steps. The developer defines only the input, output and execution method. The HCSS provides all the other functionality required for the task to perform it role within a jython session or Java program and other functions which need not concern the scientifically-minded developer!
The task I will code up in this example is a flatfield. This will involve taking a table of data and a flatfield file which is identical nin format to the input data. The output will be a modified (i.e. flatfielded) data table.

1.2 Product

A Product is analogous to a FITS file, but with more flexibility. It can contain single arrays, table, multiple tables, images and mixtures of these. The developer can attached meta data (header in a FITS format-speak) to any one of the types and to the product as a whole. There will probably be several form of product; a detector timeline product, a calibration product, a post-processing product etc. These definition are currently embryonic so I will stick to a simple Product. This product has one table (the TableDataset class from the herschel.ia.dataset package) contained within and few lines of metadata (e.g. names,dates,author, version number etc).

Please read the detailed description of both of the classes provided by Steve Guest and JJ Mathieu respectively to fully understand their role in IA.

2. THE CODING

There are two things to understand when coding up a task:
* the input and output parameters (which will be objects) will have fixed names so the usage of the task will be predictable to users.
* The input and output objects will always be Products

!!Jargon crime!! - 'objects'. This can be thought of as the fits file existing as one thing; a set of tables and meta data (i.e.header) within the computer not separate as a bunch text fields and arrays of numbers.

The easiest way of demonstrating the coding is to provide a complete example for you to code up on your own machine and then modify. I have included extensive comments.

```
// First parts are for the version controlling (cvs tagging)

/*
 * $Id$
 * Copyright (c) Imperial College London
 *
 */

// The import statements for the relevant packages in the hcss structure
import herschel.ia.dataset.*;
import herschel.ia.numeric.*;
import herschel.ia.task.*;
import herschel.ia.task.api.*;

public class Flatfield extends Task {    // Inherits all the functionality of
the Task class


    public Flatfield(String name) throws SignatureException {  // I'm making
the constructor always have a name.
      super(name);    // Uses the superclass constructor.  In this case it
is Task
      System.out.println("Initialising the Flatfield Task");
      // Prints out to the command line.  Proper treatment would send this
to a logging object and where the ultimate destination of the information is
decided.

      // This where we set the one of the input parameters by defining its
name and type of input.  'flatfield' is the name which users of this task
will reference this input parameter.
      TaskParameter p = new TaskParameter("flatfield",Product.class);

//I've made these parameters mandatory for obvious reasons.  If one required
some optional inputs it would be the p.setMandatory(false);
      p.setMandatory(true);
      addTaskParameter(p);
```

```
// This where we set the other of the input parameters by defining its name
and type of input.  'data' is the name which users of this task will
reference this input parameter.

        p = new TaskParameter("data",Product.class);
      p.setMandatory(true);

        addTaskParameter(p);

        // This where we set the output parameter  'result' is the name
which users of this task will reference this output parameter.
      p = new TaskParameter("result", Product.class);
        p.setMode(TaskParameter.OUTPUT);


          // We attach our output parameter

      addTaskParameter(p);

    }

    //Now we define the execute method.  Note there are no input arguments.

    public void execute() {


      //All we are doing here is getting the two input products using the
getValue methods inherited from 'Task'

Product tableProduct = (Product) getValue("data");

Product flatProduct = (Product) getValue("flatfield");


//We now have two options either to get the default tabledataset using
getDefault(), this is only safe when you are sure there is only one present
in the product. The best way is to name the table you require from the
product. This requires prior knowledge of the incoming data products
contents.
//I've commented out the two getDefault() calls so the direct 'get' works.

//TableDataset table = (TableDataset) tableProduct.getDefault();
//or
TableDataset table = (TableDataset) tableProduct.get("DATA_TABLE");


//TableDataset flat = (TableDataset) flatProduct.getDefault();
//or
TableDataset flat = (TableDataset) flatProduct.get("FLAT_TABLE");


// What follows is all commented out because it is dependent on the detail
of the individual datasets but I've left it there as a guide to navigating
your way round and extracting the data from the tables. It vaguely follows
this hierarchy:
//Product then TableDataset then Column then Real1D (or Float1D, Boolean2D
etc) then double[] (or int[], boolean)

//The following line puts together a few methods to get the Real1D.
// Real1D data = (Real1D) table.getColumn(table.nameOf(0)).getData();
// The following line gets the basic array of doubles.
//      double[] x = data.getArray();
//    Real1D flatData = (Real1D) flat.getColumn(table.nameOf(0)).getData();
//      double[] y = flatData.getArray();
//    double[] flatted = new double[x.length];
//    for (int i=0;i<x.length;i++)  flatted[i] =  x[i]/y[i];
//    Real1D resData = new Real1D(flatted);


//    System.out.println("Putting flatted information into table dataset");

      TableDataset result = table;

      //    result.addColumn("result", new Column(resData));
      table = result;
      tableProduct.set("DATA_TABLE",table);

      //Now we set the output contents using setValue(); and the name
'result' which we set as a TaskParameter.

      setValue("result",tableProduct);

    }


}
```

2.1 Compiling the Java code.

There are two methods for compiling the java code either using the 'javac' command which places the resultant class file in the directory you are in or using the 'jake' command which places the class files in a separate 'out' tree. For the minute it is probably best to use javac and then make sure the the class you create is in the 'import' statements of the Jython script.

Now we trigger our JIDE/Jconsole/QLAConsole environment to run the task in the Jython environment. JIDE comes with the HCSS build and as long as you classpaths are set up for the HCSS in general all the following code should work. You will ultimately have to 'build' the SPIRE part of the development tree (stored in develop/main/herschel/spire/yacketyschmack) rather than just sticking your class just anywhere.
Type 'jide' at the command prompt on the terminal and you will see the three windows (the text is huge for demos this can be changed). Cut and paste the scripts in or open the script in the top window and use the 'play' button to execute them line by line. If you leave gaps in your code things will get confused so be careful.

```
The first parts are equivalent to the import commands of the java code;
#
# IMPORTANT CHANGE THE NEXT LINE TO CONTAIN YOUR TASK
```

```
from  whereeveryouputtheFlatfieldclassfile import Flatfield
#
from herschel.ia.io.fits import FitsArchive
from herschel.ia.task.demo.data import RawData
from herschel.ia.task.api import *
from herschel.ia.task import *
from herschel.ia.dataset import *
from  herschel.ia.numeric import *
# hash is the comments symbol
#
#Remember Jython doesn't like gaps in the code so put hashes there.
#
#  This section makes some dummy products to put into the flatfield routine.
p=Product()
t=TableDataset()
p.set("DATA_TABLE",t)  # Put 't' in product 'p' as an empty data table.
pf = Product()
tf = TableDataset()
pf.set("FLAT_TABLE",tf)  # Put 'tf' in as an empty faltfield calibration
table.
# Now we create a Flatfield object with name Banana flat and a description .
flat = Flatfield("Banana flat", description="Flatfielding a raw image")
#

#
#
#At last we are at the point where we use the task.
res=flat(data=p,flatfield=pf)
#
#
#Now we can look at whats inside the product (of course its nothing apart
from the name DATA_TABLE)
print res
{description="", meta=[creator, creationDate, instrument, modelName,
startDate, endDate], datasets=[DATA_TABLE], history=None}
#
#
```

And that's it.

When the SPIRE product has been better defined a lot of the extracting of the data from the inside of the products will be taking care of by the methods of the product itself.   Once you have defined the input and output products you can code up a mini pipeline to see how Jython string together Java classes.