

ISAW Developer's Manual

February 6, 2002

Table of Contents

1	Basics	3
1.1	Conventions in this Manual	3
1.2	Downloading and Installing	3
1.3	Running ISAW	4
1.4	Compiling ISAW	5
1.5	Developing with CVS	5
2	Building DataSets	7
2.1	Introduction	7
2.2	Steps to Create a DataSet	7
2.2.1	Construct the Empty DataSet	7
2.2.2	Add Attributes to the DataSet	8
2.2.3	Construct a Data object	9
2.2.4	Add attributes to Data object	10
2.2.5	Add the Data object to the DataSet	10
2.3	Attributes Needed in a DataSet and Data Object	11
2.4	A Data Retriever	12
2.5	An Example	13
3	Operators	17
3.1	Introduction	17
3.1.1	A Simple Example	17
3.2	Operator Structure	19
3.3	Running the Operator	20
3.4	Search Paths for User Supplied Operators	21
3.5	Operator Mechanism, Details	22
3.5.1	Operator Categories	22
3.5.2	Operator Parameters	23
3.6	Details and Examples	23
3.6.1	EXAMPLE: LoadASCII.java	24
3.6.2	EXAMPLE: Ysquared.java	25
3.6.3	EXAMPLE: CenteredDifferences.java	25
3.6.4	EXAMPLE: IntegratedIntensityVsAngle.java	26
3.6.5	EXAMPLE: OperatorTemplate.java	27
3.7	Categories for Operators	27
3.7.1	Generic Operators(GenericOperator)	28

3.7.2	DataSet Operators (DataSetOperator)	29
3.7.3	Not Categorized Operators, Misc	32
4	Live Data	33
4.1	UDP Data Format	34
4.2	DAS Sender	34
4.3	Live Data Server	34
4.4	ISAW Client	35
5	NeXus	36
5.1	Format	36
6	ASCII File Formats	37
6.1	Table View	37
6.2	GSAS Data File Format	37
6.2.1	Standard Powder Data File	37
6.2.2	Instrument Parameter File	39
6.3	spec Standard Data File Format	40

Chapter 1

Basics

1.1 Conventions in this Manual

Since this manual refers to a variety of things, java code, things on the command line, and gui elements, the fonts will be selected as listed in Table 1.1.

Table 1.1: Font conventions for this manual

Java code	<code>fixed width font</code>
classes, methods, and variables	<code>sans-serif font</code>
command line, filenames	<code>typewriter font</code>
GUI elements	<i>italics</i>

1.2 Downloading and Installing

The normal ISAW distribution available at <ftp://zuul.pns.anl.gov/isaw/> contains the source java files for all ISAW-related programs as well as the compiled class files. In the normal installation for users, the software is run out of Java archive files (jar files), and the Java source files are not seen. In order to modify programs in the ISAW package, it is necessary to unpack the software into a directory to make the source files available for modification. For developers at IPNS, the source files can be checked out from a CVS repository in order to coordinate development among several developers. The steps necessary to unpack ISAW in order to modify it are:

1. Download the Java Development Kit (JDK) version 1.3 or later from the Sun web site, <http://java.sun.com/j2se/1.3/>.
2. Install the JDK according to the instructions for your computing platform.
3. Download the ISAW distribution (without JRE) for your platform from the IPNS ftp server at <ftp://zuul.pns.anl.gov/isaw/>.

4. Unpack the ISAW `tgz` file (for UNIX/Linux) or the self-extracting zip file (for Windows) to install ISAW.
5. Open a command/shell window in the directory where ISAW was installed.
6. Unjar the jar files using the following commands:
 - `jar -xvf sgt_v2.jar`
 - `jar -xvf jnexus.jar`
 - `jar -xvf IPNS.jar`
 - `jar -xvf Isaw.jar`

Now you have unpacked ISAW and its associated libraries down to the actual java and class files.

1.3 Running ISAW

Running ISAW is easy with everything packed up and in one directory that contains the installation. If the source is not extracted from the jar files, just use the command (on linux)

```
java -mx128000000 -cp Isaw.jar:sgt_v2.jar:IPNS.jar:jnexus.jar
    IsawGUI.Isaw
```

from within the directory containing the jar files. For windows replace the colon (:) with a semicolon (;). The `-mx128000000` allocates 128MB of initial memory to program and `-cp` sets the classpath to be the necessary jar files. The 128MB of memory is sufficient for many applications. However, the `-mx` command can be used for different amounts of memory as well to handle large files or for use on a computer with a small amount of memory. Finally the class executed is `Isaw.class` within the `IsawGUI` package. Remember that for Java, a package must be named in the file using the `package` command and the source code must be in a subdirectory of the same name. If you have unpacked the distribution to modify the source the command is

```
java -mx128000000 -cp ISAW_HOME IsawGUI.Isaw
```

where `ISAW_HOME` is the location of ISAW with the full path specified.

For either installation case the classpath can also be specified in a system file. For linux this is done in a login script by setting the `CLASSPATH` variable. The above examples assume that the java command is in the path. In both linux and windows systems the path is set in one of the login scripts using the `PATH` variable. If java will not be in the path then the fully qualified command must be specified (i.e. `/usr/java/jdk1.3.1/bin/java`).

1.4 Compiling ISAW

Before developing new classes or packages it is best to try a simpler case for compiling. The best is to go into the IsawGUI directory of the distribution and type

```
javac Isaw.java
```

then try running ISAW. If the recompilation was successful there should be no differences between ISAW before and after you recompiled it. The next task in complexity is to edit some of the source code and recompile it. This can be done with any of the source replacing, in the above example `Isaw.java` with the name of the file to be compiled. Java does not require that files which depend on a modified file be recompiled which simplifies the task.

If none of the class files are compiled then the build process is a little more complicated. Java assumes that all classes that a file depends on are either already compiled or are listed in the current compile command. This means that you can use one of two commands:

- `javac */*.java */*/*.java */*/*/*.java` (linux only)
- `jikes -depend IsawGUI/Isaw.java`

From the directory containing IsawGUI. The second command uses the `jikes` compiler from IBM with the `-depend` option. This automatically compiles files that the current file depends on, provided the source code for the class is available. Another option is to use a make utility or go into each directory and compile all source code by hand.

To create new classes, choose a package that best categorizes it and include the class in that package. If the new class does not fit into any existing package then a new one can be created simply by creating a new subdirectory in ISAW where files are placed. If you add new features which would be useful to other people, you should submit the modified software to IPNS for inclusion in the standard ISAW distribution after testing.

1.5 Developing with CVS

Concurrent Versions System (CVS) is a way of allowing multiple developers to work on the same project with less problems. Some developers work at IPNS or are given access to the CVS archive directly. This section provides a quick introduction to using CVS. For more information refer to the CVS manual at <http://www.cvshome.org/docs/manual/>. CVS is very good at allowing developers to contribute changes to different files and different sections of the same file and combining them in the source tree. It also allows steps backwards in the development if new bugs come in during the updating process. The danger of CVS is for developers not to communicate which will (in the end) really mess things up when two developers submit bug fixes or added features..

Using CVS on the CVS server is straightforward. You should check out the ISAW source development and obtain the external jar files for associated libraries (`IPNS.jar`, `jnexus.jar`, and `sgt_v2.jar`). There will also be files missing that you must obtain from the standard installation for your platform such as scripts, the splashscreen image and the nexus shared library. To checkout and unpack ISAW on linux (the CVS server runs linux):

- `mkdir ISAW`
- `cvs checkout ISAW`
- `cd ISAW`
- `jar -xvf IPNS.jar`
- `jar -xvf jnexus.jar`
- `jar -xvf sgt_v2.jar`
- compile `isaw` and the files it depends on

The non-java files mentioned above are not listed in this series of commands because they are not needed to compile java. Once the ISAW package is checked out of the archive these files can be added to the directory structure.

After doing this you should have a fully working version of ISAW that will cooperate with CVS and has all of the necessary components compiled. The java archive files that are installed using the `jar` command are not part of the development and should not be checked into the CVS system. At this point you can use a variety of cvs GUI's or cvs on the command line. Possible options are jCVS (<http://www.jcvs.org/>) and tkcvs (<http://www.twobarleycorns.net/tkcvs.html>) which support networked cvs servers. To do so at IPNS requires a kerberos account to access the cvs server. On linux do not forget to set the environment variables `CVSROOT`, `CVSEEDITOR`, and `CVS_RSH` for proper functioning. For work on the cvs server the `CVS_RSH` variable does not need to be set.

Chapter 2

Building DataSets

2.1 Introduction

A `DataSet` object, as used by ISAW, is a container object that contains zero or more `Data` objects. Each `Data` object represents a tabulated function or histogram using a collection of y-values and corresponding x-values. Both the containing `DataSet` and each `Data` object that it contains also hold several types of auxiliary information. Some of the auxiliary information is in the form of a fixed set of data fields in the objects, some is in an extensible list of "attributes" maintained by each object. Since various operations can be performed on a `DataSet` and the `DataSet` includes an extensible list of operators that can operate on the `DataSet`. Finally, the `DataSet` keeps a "log" of the operations that have been applied to the `DataSet`.

There are five major steps that are typically followed when building a `DataSet`:

1. Construct the empty `DataSet` complete with appropriate operators.
2. Add "attributes" to the `DataSet`.
3. Construct a `Data` block to add to the `DataSet`.
4. Add "attributes" to the `Data` block.
5. Add the `Data` block to the `DataSet`.

These steps would usually be done in the order listed, with the last three being repeated for each `Data` block that is added to the `DataSet`.

2.2 Steps to Create a DataSet

2.2.1 Construct the Empty DataSet

This is very easy for a time-of-flight `DataSet`. There is a `DataSetFactory` that can be used to build the empty `DataSet` and add the needed operators to the `DataSet`. For a time-of-flight `DataSet` this is used as shown below. The "title" parameter that is

passed to the constructor of the `DataSetFactory` will specify what title will be used for subsequent `DataSets` produced by the factory.

```
DataSetFactory ds_factory = new DataSetFactory( title );
DataSet ds = ds_factory.getTofDataSet( instrument_type );
```

The `instrument_type` is an integer code for the type of instrument. This is used by the `DataSetFactory` to determine which operators should be added. The values for the integer codes are defined in the file:

```
DataSetTools/instruments/InstrumentType.java
```

and currently include:

```
InstrumentType.TOF_DIFFRACTOMETER
InstrumentType.TOF_SCD
InstrumentType.TOF_SAD
InstrumentType.TOF_DG_SPECTROMETER
InstrumentType.TOF_IDG_SPECTROMETER
InstrumentType.TOF_REFLECTROMETER
```

At this time `DataSetFactory` provides a larger set of operations for the types `TOF_DIFFRACTOMETER` and `TOF_DG_SPECTROMETER`. Support, by way of special operators for the other instrument types is still being developed. In all cases, very basic operations such as add, subtract, multiply and divide by `DataSets` and scalars are included.

If you are constructing a "generic" `DataSet` with axis labels and units other than those for a time-of-flight instrument, you can use a different constructor for the `DataSetFactory` such as:

```
DataSetFactory factory = new DataSetFactory(title,
                                           "Angstroms",
                                           "d-Spacing",
                                           "Counts",
                                           "Scattering Intensity");
DataSet ds = factory.getDataSet();
```

In this case, the factory will produce `DataSets` with the given title, axis units and labels. `getDataSet` will only add the generic operators to the `DataSet`, not the operators specific to time-of-flight `DataSets`.

2.2.2 Add Attributes to the DataSet

Attributes can be added to `DataSets` and `Data` blocks at any time. However, it is probably best to add the attributes you'll need in an organized manner at the time that you are constructing the `DataSet`. Attributes are name, value pairs where the value can be things like an integer, float, array of integers, character string, etc. Attributes are classes that are derived from the abstract base class defined in

`DataSetTools/dataset/Attribute.java`. This file also contains a list of the names that we have been using for the attributes. The names are given by "constant" strings. Since each attribute is stored in its own object, it is usually necessary to create the attribute objects as they are added to the `DataSet` or `Data` block.

If a few individual attributes are being added to the `DataSet` (or `Data` block) they can be added using the `setAttribute(attribute)` method. For example, assuming that the original data file name is to be stored as an attribute of the `DataSet`, you could write:

```
ds.setAttribute(new StringAttribute(Attribute.FILE_NAME,file_name));
```

to set an attribute for the file name in `DataSet ds`. This assumes that the variable `file_name` is a string containing the file name. This will construct a new `StringAttribute` object with the name of the attribute given by the constant `Attribute.FILE_NAME="File"` and the value of the attribute given by the `file_name` string. Other attributes are treated similarly.

```
ds.setAttribute( new IntAttribute(Attribute.NUMBER_OF_PULSES, num_pulses));
```

The attribute can also be constructed separately and then set in the `DataSet` like:

```
int_attr = new IntAttribute( Attribute.NUMBER_OF_PULSES, num_pulses );  
ds.setAttribute( int_attr );
```

Finally, there are also routines to get and set the entire list of attributes at once, but the routines to get and set individual attributes are actually more efficient to use in most cases.

2.2.3 Construct a Data object

The three most crucial pieces of information held in each `Data` object are the list of y-values, an `XScale` object specifying the corresponding x-values and a unique integer ID. These three pieces of information are needed by the constructor for a `Data` object. The y-values are just an array of type `float[]` and the ID is just an integer value. However, the `XScale` is an object that either contains the x-values, or contains enough information to calculate uniformly spaced x-values.

The x-values are stored in an `XScale` object for space efficiency. That is, in many cases the x-values associated with a `Data` block are evenly spaced. In that case, they can be easily calculated as needed based on the first point, the last point and the number of points. Since we may have thousands of spectra with thousands of y-values in each, it would be a serious waste of space to store corresponding evenly spaced x-values in such cases. In this case, the x-values can be stored in a `UniformXScale` object, derived from an `XScale` object. For example, a uniform `XScale` object with 101 points evenly spaced on the interval $[0, 10]$ can be constructed as:

```
XScale x_scale = new UniformXScale( 0, 10, 101 );
```

If the x-values are not evenly spaced, a `VariableXScale` object can be used to explicitly store all of the x-values. Specifically, if an array of floats named "x-vals" contained the x-values we could create a `VariableXScale` object as follows:

```
XScale x_scale = new VariableXScale( x-vals );}
```

In either case, software using the `x_scale` can get at information such as the minimum, maximum, number of points and the actual x-values using the methods of the base class `XScale`. For a time-of-flight `Data` object, the operators assume that the times are specified in microseconds.

It also should be noted, that `Data` objects are used to store either histogram data, or tabulated function data. These two cases are distinguished based on the relationship between the number of x-values and the number of y-values. Specifically, for a tabulated function `Data` object, the number of x-values will be the same as the number of y-values. In this case, the y-values give the value of a function at the corresponding x-value. On the other hand, for a histogram, the `Data` object records the x-values at the boundaries of the histogram bins. The y-values are considered to be the y-values at the bin centers. Thus for histogram data, the number of x-values is one more than the number of y-values. The number x-values is restricted by the `Data` object constructor to be either the number of y-values, or the number of y-values plus one.

An example of building a simple `Data` block for the function $y = (x/10)^2$ on the interval $[0,49]$, with `ID = 1` is given below:

```
float y_values[] = new float[50];
XScale x_scale    = new UniformXScale( 0, 49, 50 );
for ( int i = 0; i < 50; i++ ){
    y_values[i] = (i/10.0) * (i/10.0)
}
Data data = new Data( x_scale, y_values, 1 );
```

2.2.4 Add attributes to Data object

Both `DataSet` objects and the `Data` objects that they contain implement the `IAttributeList` interface. As a result, attributes are added to `Data` objects in exactly the same way as they are added to `DataSets`. For example, if data is a `Data` object, we could add an attribute specifying that the initial energy was 120.0 as follows:

```
data.setAttribute( new FloatAttribute( Attribute.ENERGY_IN, 120.0f ) );
```

2.2.5 Add the Data object to the DataSet

Once a `Data` object has been constructed, and its attributes set, it should be added to a `DataSet`. For example, to add a `Data` object `data` to a `DataSet ds` just do:

```
ds.addData_entry( data );
```

2.3 Attributes Needed in a DataSet and Data Object

Although the above discussion describes how to construct a `Data` block and `DataSet`, more information is needed to construct a `DataSet` to hold time-of-flight data in a way that will allow useful operations to be done on the `Data`. In particular, most of the "interesting" operators for neutron scattering rely on specific attributes of the `DataSet` and `Data` objects. The attributes that are currently used by various operators include:

```
Object Attribute.DETECTOR_POS
```

```
Float Attribute.INITIAL_PATH
```

```
Float Attribute.ENERGY_IN
```

```
Int Attribute.NUMBER_OF_PULSES
```

```
Float Attribute.SOLID_ANGLE
```

```
Float Attribute.DELTA_2THETA
```

```
Float Attribute.RAW_ANGLE
```

To allow for comparing and scaling `DataSets`, some measure of the number of neutrons that hit the sample is needed. For use in scripts, this should probably be the number of pulses, at least that is what has been used for GPPD. If the number of pulses or integrated beam current is not directly available, it could possibly be approximated based on a start time and end time. At any rate, it would be useful to have the number of pulses stored as a `DataSet` attribute for any instrument.

The attributes listed above are primarily used as attributes of each `Data` object. The attributes are listed in decreasing order of importance. Interpreting the time-of-flight data almost always requires the effective (focused) detector position. The convention used in the `DataSetTools` package is that the effective detector position gives the position of the detector relative to the center of the sample. Since there are different ways of specifying this position, a class was constructed to hold the position information and provide some extra information as needed. A `Position3D` object contains a position, specified in any of the usual coordinate systems, Cartesian, cylindrical or spherical. There are methods to get and set the position in any of these coordinate systems, as well as some additional convenience routines.

The convention for the instruments at IPNS is that the coordinate system has its origin at the sample position, the x-axis points in the direction the incident beam is traveling, the y-axis is horizontal, perpendicular to the incident beam and z-axis is perpendicular to the earth's surface. This is also the convention followed by the `DataSetTools` package. Unfortunately, that coordinate system is somewhat inconvenient for describing the scattering angle (the angle between the positive x-axis and the vector from the sample to the detector). Since the operators frequently need to use the scattering angle a class `DetectorPosition` was derived from the `Position3D` class. The `DetectorPosition` class adds a method to get the scattering angle, and so it

should be used to represent the position of the detector relative to the sample. An example of code to set a detector position attribute corresponding to a detector that is at an angle of 50 degrees, 0.1 meter above the xy plane, and above a horizontal circle of radius 4.0 meters centered at the sample is shown below:

```

DetectorPosition position = new DetectorPosition();
float angle          = 50.0f * (float)(Math.PI / 180.0);
float final_path    = 4.0f;
float height        = 0.1f;
position.setCylindricalCoords( final_path, angle, height );
data.setAttribute(
    new DetPosAttribute( Attribute.DETECTOR_POS, position ) );

```

Lengths are assumed to be in meters, and angles are stored in radians. If the detector position is easier to specify in Cartesian coordinates, (x, y, z), the method `position.setCartesianCoords(x, y, z)` can be used instead.

The initial path attribute is needed for the diffractometer instruments. The initial flight path is the source to sample distance in meters. If this can be obtained, as say the float variable "length", it is easily added to the `Data` block as:

```

data.setAttribute( new FloatAttribute( Attribute.INITIAL_PATH, length ) );

```

The operators to process data from direct geometry spectrometers require the initial energy of the neutrons incident on the sample. The initial energy is assumed to be in meV. It is often necessary to calibrate this value, but at least some initial approximation will be needed by these operators. The more advanced operators for direct geometry spectrometers will require the number of pulses to stored with each `Data` block, in addition to being stored with the `DataSet` as a whole. Finally, these operators need the solid angle subtended by the detector group, measured in steradians.

The operator to produce a display of $S(Q, E)$ for spectrometers will need an approximate value for the interval of scattering angles covered by each detector. That is, each detector has non-zero dimensions. Consequently, even though the detector might be nominally at say 50 degrees, it actually covers some interval, say 49.95 to 50.05 degrees. Some approximation to the range of angles covered should be stored in a `DELTA_2THETA` attribute. This value is assumed to be stored in degrees.

The operator to produce a "TrueAngle" display of a `DataSet` requires the `DELTA_2THETA` attribute, as well as the `RAW_ANGLE`. The `RAW_ANGLE` is the actual physical scattering angle for the detector, without regard to time-focusing (time focusing may adjust the raw angle to a different effective angle). A `DETECTOR_POS` attribute is assumed to hold the effective 3D position of the detector, while a `RAW_ANGLE` attribute is assumed to hold the physical, unfocused scattering angle.

2.4 A Data Retriever

In order to easily work with different sources of data, such as IPNS runfiles, Nexus files, data acquisition hardware, etc. the system was designed to access data through

subclasses of the abstract class `DataSetTools/retriever/Retriever.java`. Currently the most robust derived class is `RunfileRetriever` that accesses IPNS runfiles. New data sources should be supported by making a new class derived from the `Retriever` class (such as `NexusRetriever`), since in that way, all data sources can be used in the same way. The `Retriever` class is quite simple:

The constructor accepts a string giving the name of the data source. For a data file, this would most likely be the file name, and the file would most likely be opened in the constructor.

The `Retriever` class then provides three methods, a method to get the number of `DataSet`s available from the source, a method to get the type of each available `DataSet` (`MONITOR_DATA_SET` or `HISTOGRAM_DATA_SET`) and a method to get a specific `DataSet` from the source. For the special case of the IPNS runfile retriever this gets used as simply as:

```
RunfileRetriever rr    = new RunfileRetriever( "gppd9898.run" );
DataSet A_monitor_ds  = rr.getDataSet( 0 );
DataSet A_histogram_ds = rr.getDataSet( 1 );
```

where we've used the simplifying assumption that the "zeroth" `DataSet` is always the monitor `DataSet` and the "first" always the first histogram `DataSet`. These simplifying assumptions make it unnecessary to find out the number of `DataSet`s and find out their types before reading.

As other types of files or data sources are supported, the `Retriever` class may need to expand slightly. However it is best to keep this class as simple as possible, since any new functionality introduced in the `Retriever` will have to be supported by ALL types of retrievers.

2.5 An Example

A simple program to demonstrate building a `DataSet` is in the file:

```
DataSetTools/trial/BuildDataSetDemo.java
```

in the latest version of `DataSetTools`. It can be compiled from within the directory containing it using:

```
javac BuildDataSetDemo.java
```

and then can be run using

```
java BuildDataSetDemo
```

Assuming that all `PATH` and `CLASSPATH` values have been set properly. The code for the demo is listed below:

```

/*
 * @(#) BuildDataSetDemo.java    1.0  2000/9/19    Dennis Mikkelson
 *
 */
import DataSetTools.dataset.*;
import DataSetTools.viewer.*;
import DataSetTools.math.*;

/**
 * This class provides a basic demo of how to construct a
 * DataSet.
 */
public class BuildDataSetDemo
{

    /**
     * This method builds a simple DataSet with a collection of 10
     * sine waves.
     *
     * @return  A sample DataSet with 10 sine waves.
     */
    public DataSet BuildDataSet()
    {
        //
        // 1. Use a "factory" to construct a DataSet with operators ---
        //
        DataSetFactory factory =
            new DataSetFactory( "Collection of Sine Waves",
                               "time",
                               "milli-seconds",
                               "signal level",
                               "volts" );
        DataSet new_ds = factory.getDataSet();

        //
        // 2. Add attributes, as needed to the DataSet -----
        //
        new_ds.setAttribute( new StringAttribute( Attribute.FILE_NAME,
                                                "BuildDataSetDemo.java" ) );
        new_ds.setAttribute(
            new IntAttribute( Attribute.NUMBER_OF_PULSES, 10000 ) );

        //
        // Now, repeatedly construct and add Data blocks to the DataSet
        //
    }
}

```

```

Data      data;          // data block that will hold info
                //                on one signal
float[]   y_values;     // array to hold the y-values for
                //                that signal
XScale    x_scale;     // "time channels" for the signal

for ( int id = 1; id < 10; id++ )          // for each id
{
    //
    // 3. Construct a Data object
    //
    x_scale = new UniformXScale( 1, 5, 50 ); // build list of
                // time channels

    y_values = new float[50];              // build list of counts
    for ( int channel = 0; channel < 50; channel++ )
        y_values[ channel ] =
            100*(float)Math.sin( id * channel / 10.0 );

    data = new Data( x_scale, y_values, id );

    //
    // 4. Add attributes as needed to the Data block
    //
                // "simple" energy in attribute
    data.setAttribute(
new FloatAttribute( Attribute.ENERGY_IN, 120.0f ) );

                // more complicated, position
                // attribute has a position
                // object as it's value
    DetectorPosition position = new DetectorPosition();
    float angle          = 50.0f * (float)(Math.PI / 180.0);
    float final_path     = 4.0f;
    float height         = 0.1f;
    position.setCylindricalCoords( final_path, angle, height );
    data.setAttribute(
        new DetPosAttribute( Attribute.DETECTOR_POS, position ) );

    //
    // 5. Add the Data object to the DataSet
    //
    new_ds.addData_entry( data );
}

```

```

    return new_ds;
}

/* ----- */
/**
 * The main program method for this object
 */
public static void main(String args[])
{
    // create the class
    BuildDataSetDemo demo_prog = new BuildDataSetDemo();

    // call the method to construct a DataSet
    DataSet test_ds = demo_prog.BuildDataSet();

    // create a viewer for the DataSet
    ViewManager view_manager = new ViewManager( test_ds,
                                                IViewManager.IMAGE );
}
}

```

Chapter 3

Operators

3.1 Introduction

The Operator concept in ISAW provides a very powerful and flexible way to implement special capabilities that are not in the basic ISAW system. A user-supplied operator can be used from the ISAW GUI and from scripts, provided it has been compiled and is in a directory where it can be found by ISAW and Java, so that it can be executed. The main ISAW application does NOT have to be recompiled to use the new operator. Also, the parameter values required by the operator can be obtained from the ISAW pop up dialog box, so that the person writing the operator doesn't have to do any of the GUI coding themselves.

The ability of the operator to interact with ISAW scripts, dialog boxes etc. comes at a small price. Specifically, an operator must include code to describe it's parameters to ISAW, to tell the scripting system what it's command name is, and to make a copy of itself. However, this code is very "stylized" and can either be adapted from the supplied operator template, or can be automatically generated by the operator builder utility.

3.1.1 A Simple Example

A sample operator illustrates the basic structure of an operator, and how operators interact with ISAW and the scripting system. The full operator code, with proper documentation comments is in the file `HelloOperator.java`. The following lists all of the necessary code with selected comments.

```
=====
HelloOperator.java
=====
package ISAW.Operators;    // The "package" for this operator. To run,
                           // this operator must be in a directory
                           // named "ISAW/Operators" and the parent
                           // directory must be on the java CLASSPATH
import DataSetTools.operator.*;
```

```

import java.util.*;

public class HelloOperator extends GenericOperator
{
    private static final String TITLE = "Hello Operator";

    public HelloOperator()                // Default constructor, used
    {                                     // by the ISAW GUI. The user
        super( TITLE );                  // will interactively specify
    }                                     // the parameters to use.

    public HelloOperator( String user_name ) // Constructor that specifies
    {                                       // parameter values. This is
        this();                             // convenient when the operator
        parameters = new Vector();          // is used from a java program.
        addParameter( new Parameter("Name", user_name) );
    }

    public String getCommand()             // Tells script system what
    {                                       // the command name is for
        return "SayHello";                 // this operator
    }

    public void setDefaultParameters()     // Load or reload default
    {                                       // parameters. This also
        parameters = new Vector();         // sets the data types.
        addParameter( new Parameter("Name", "John Doe") );
    }

    public Object getResult()              // This is called to run the
    {                                       // operator using its current
                                           // parameter values.
        String user_name = (String)(getParameter(0).getValue());
        return new String ("Hello ") + user_name;
    }

    public Object clone()                   // Utility to allow copies
    {                                       // of the operator to be
        Operator op = new HelloOperator(); // created.
        op.CopyParametersFrom( this );
        return op;
    }

    // Main program for testing.
    public static void main( String args[] ) // This allows running the
    {                                       // operator by itself to be

```

```

// sure that it is working.
System.out.println("Test of HelloOperator starting...");

// since we're not running
// in ISAW here, get the user
// name from the system.
String name = System.getProperty( "user.name" );

// make and run the operator
Operator op = new HelloOperator( name );
Object obj = op.getResult();

// display the string returned
System.out.println("Operator returned: " + obj );

System.out.println("Test of HelloOperator done.");
}
}

```

3.2 Operator Structure

The `HelloOperator` demonstrates the basic operator structure. The basic role of each of the methods of the `HelloOperator` is described in the following paragraphs. More detailed information on the role and use of the parameters is given later in the section on operator parameters.

First, there will typically be two constructors. The first constructor is the default constructor with no parameters. This constructor will be used when the operator is called from ISAW and meaningful parameter values will be provided by the user. Default values are set for the parameters by the `setDefaultParameters()` method that is called by the constructor for the `Operator` super class. The default constructor will typically just call the super class's constructor, specifying the title for this operator. The second constructor takes values for the operator parameters, so that the operator can be conveniently used by Java applications such as ISAW. This constructor is not needed to use the operator from ISAW or with the scripting system, but makes testing the operator more convenient. After calling the default constructor, this constructor creates the list of parameters in the operator, using the values specified in the arguments to the constructor.

In addition to the constructors, there are four required methods that all operators must implement. The `getCommand()` method tells the scripting system what command to use to invoke the operator from a script. The `setDefaultParameters()` method establishes a default list of parameters for the operator. The parameter names and data types from this list are used by ISAW to construct an appropriate dialog box if the operator is invoked from ISAW. The `getResult()` method is what is called to actually carry out the operation. Finally, the `clone()` method of object is overridden to allow ISAW and the scripting system to make copies of the operator if needed.

In Java, each object can have it's own main program that can be used to test the object. While it is not strictly necessary to have a main program for an operator, it is extremely helpful to be able to at least test the basic functionality of the operator before it is used in a larger system. The main program for the `HelloOperator` just prints a message indicating that the operator is being tested, then creates an instance of the `HelloOperator`, using the current user's name, calls `getResult()` and prints the string that was returned from `getResult()`.

3.3 Running the Operator

Each operator can be used in any one of three ways. First if it has a main program, the main program can be run to test the operator. It can also be used in scripts and it can be automatically incorporated into the ISAW menu system when ISAW is started. The steps needed to use the `HelloOperator` in each of these ways will be described in detail.

In order to separately test operators, each operator can have a main program that tests the basic functionality of the operator. Before it can be used from ISAW or the scripting system, the operator must be compiled and be able to execute. That means that the java interpreter must be able to find the operator's class file, so the `CLASSPATH` must be set properly. Since the `HelloOperator` was made part of the `ISAW.Operators` package, the file `HelloOperator.java` must be in a directory called `ISAW/Operators`. In order to run, the parent directory of `ISAW/Operators` must be on the java `CLASSPATH`. If these conditions are met, you can compile the operator using:

```
javac HelloOperator.java
```

from within the `Operators` directory. This creates the file `HelloOperator.class`. To run the operator, you can just use

```
java ISAW.Operators>HelloOperator
```

from anywhere. The prefix `ISAW.Operators` is needed since the `HelloOperator` is in the package `ISAW.Operators`.

To test this:

1. Create a directory in your home directory called `ISAW`.
2. Create the `Operators` subdirectory of this `ISAW` directory.
3. Copy the `HelloOperator.java` code into the new `ISAW/Operators` directory in your home directory.
4. Add your home directory to the java `CLASSPATH`.
5. Compile and run the `HelloOperator` using `javac` and `java` as described above.

If it works properly, the operator should print a message saying that it is running, then say hello to you and finish.

Once the operator has been compiled and run as a stand alone java program, it can also be used from the ISAW GUI and from scripts, provided the script interpreter can find the operator. If the `HelloOperator` is in the `ISAW/Operators` directory of the user's home directory, as described above, it will be found and "automatically" included in the ISAW menu system.

To verify that the operator can be run from ISAW, start ISAW and go to the *Macros*→*Generic* submenu. An entry *Hello Operator* should be listed in the menu. Selecting the *Hello Operator* will pop up a dialog box to allow you to enter your name. Press the *Apply* button to run the operator and display the result in the dialog box.

To verify that the operator can be used from a script, select the *Scripts* tabbed pane in ISAW and enter the following one line script in the *immediate* entry:

```
Display SayHello( "your name" )
```

then run the script. The status window should display the result of running the operator.

3.4 Search Paths for User Supplied Operators

The script interpreter will look for pre-compiled operators in several places. First, it checks for compiled operators (`.class` files) in an `ISAW/Operators` directory in the user's home directory. Additional directories to look for operators in can be specified in the properties file, `IsawProps.dat`, also located in the user's home directory. Specifically, `IsawProps.dat` can specify the location of the ISAW installation (`ISAW_HOME`), and one or more directories that contain operators that may be shared by several users (`GROUP_HOME`, `GROUP1_HOME`, `GROUP2_HOME`, etc). Relevant lines from `IsawProps.dat` might look like:

```
GROUP_HOME=/usr/share/IsawOps/  
GROUP1_HOME=/usr/local/HRCS_OPS/  
ISAW_HOME=/usr/local/IPNSjava/
```

If these lines are present in the `IsawProps.dat` file, the scripting system will look for operator class files in four directories (and sub-directories of those directories), in the order shown:

```
/home/user_name/ISAW/Operators          ( user's home directory )  
/usr/share/IsawOps/Operators  
/usr/local/HRCS_OPS/Operators  
/usr/local/IPNSjava/Operators          ( ISAW "home" directory )
```

In all cases the new operators being added to the system will be someplace in the directory tree under a directory called `Operators`. Some care will be needed to avoid package name conflicts. In the above example, there will be no name conflicts if the `CLASSPATH` includes directories:

```
/home/user_name
/usr/share
/usr/local
/usr/local/IPNSjava
```

and the operators in those directories are in the following packages, respectively:

```
ISAW.Operators
IsawOps.Operators
HRCS_OPS.Operators
Operators
```

It is probably easiest to start adding your own operators to ISAW by putting your home directory on the CLASSPATH, and developing operators in a package, ISAW, located in the directory <home directory>/ISAW/Operators. When operators are to be shared by several members of a working group, they can be placed in one of the group directories, or in the ISAW_HOME directory. If an operator is moved to such a shared directory, the package name will have to be changed and the operator will have to be recompiled.

3.5 Operator Mechanism, Details

3.5.1 Operator Categories

In order to provide some structure to the menu system, the user-supplied operators can be placed in one of several categories. The categories currently supported are:

```
GenericOperator
GenericLoad
GenericSave
GenericSpecial
```

corresponding to the menu categories under the *Macro* menu bar entry. To create an operator in a particular category, just derive the operator from that class. For example, if we wanted to put the `HelloOperator` under the *GenericSpecial* category, we would change the source code to read:

```
public class HelloOperator extends GenericSpecial
```

instead of

```
public class HelloOperator extends GenericOperator.
```

3.5.2 Operator Parameters

An operator is run when its `getResult()` method is called by the ISAW main program, a script, or some other janva application. At that time the operator will be executed using the current values of its parameters. In order for the ISAW dialog box to generate appropriate data entry lines, and for the scripting system to check the data types of the parameters, the operator must be able to provide prompts and data types for its parameters. This is done using the class `DataSetTools.operator.Parameter`.

The `Parameter` class bundles together two pieces of information, the name for the parameter and a Java object that holds the value for the parameter. The name of the parameter provides the prompt string that appears for that parameter in the dialog box. The class of the value object determines the data type of that parameter. The data types currently supported by the scripting language and dialog box include `Float`, `Integer`, `Boolean`, `String`, `DataSet`, `int []` and `StringLists`. When writing an operator, the names and default values must be provided for all of the parameters. This information must be provided in the form of a `Vector` of parameters constructed in the method `setDefaultParameters()`.

The `setDefaultParameters()` method is called from the constructor for the base class operator. If the operator is to be called from other java applications, it is also convenient to make a separate constructor that allows values for the parameters to be specified. In any case, when the `getResult()` method is called, the current values of the parameters will be used.

When an operator is called from the main Isaw GUI, the dialog box will obtain the current list of parameters from the operator, using methods implemented in the base class operator, and will populate the dialog box with appropriate software components to allow the user to enter values for the parameters. When the *Apply* button is pressed, the parameter values entered by the user are set as the current parameter values for the operator and the operator's `getResult()` method is called to execute the operator. If the operator is used from a script, the script interpreter will set the values of the parameters in the operator from the values specified in the script before calling the operator's `getResult()` method.

3.6 Details and Examples

In order to write operators that work with `DataSets`, some knowledge of the structure and attributes of `DataSets` is needed. It will probably be helpful to look over the information on `DataSets` in Chapter 2 or the ISAW user manual before proceeding. The java docs for the `DataSet`, `Attribute` and `Parameter` classes in the docs directory of the ISAW distribution directory should also be helpful.

The basic concept is quite simple and will be quickly summarized here. A `DataSet` consists of a list of Data blocks. Each Data block has a list of x -values and a list of y -values, representing a tabulated function, or a histogram. For a tabulated function, there are as many x -values as y -values, but for a histogram, there is one more x -value, since the x -values represent the bin boundaries.

The DataSet and each Data block also contain meta-data. For example, the DataSet contains a title, units and labels for the x - and y -values. The Data blocks within a DataSet must represent the same physical quantities, since the units and labels at the DataSet level are assumed to apply to all Data blocks. However, the Data blocks do not have to be aligned in any particular way, and can cover different intervals with different resolutions since each Data block has its own list of x -values. Also, the sample points do not have to be uniformly spaced. Most other meta-data is stored in extensible lists of attributes. Each Data block has its own list of attributes and the DataSet as a whole has a list of attributes.

The following example operators illustrate some of the techniques involved in operators that create and process DataSets. The examples and the ideas that they illustrate are:

- **LoadASCII** - load a simple ASCII data file into a DataSet
- **Ysquared** - access and alter the values in a DataSet
- **CenteredDifferences** - create a new DataSet from an existing DataSet
- **IntegratedIntensityVsAngle** - access detector position attributes and use of multiple parameters
- **OperatorTemplate** - DataSet, int, float, boolean and String parameters. May be modified to produce a useful operator.

3.6.1 EXAMPLE: LoadASCII.java

As a first example of a useful operator, consider the operator `LoadASCII.java`. This operator loads one histogram from an ASCII file with a very simple format. The source code, `LoadASCII.java`, and an example data file, `LoadASCII.dat`, should be in the Operators subdirectory of the latest ISAW distribution.

The `LoadASCII` operator takes only one parameter, the fully qualified file name of the ASCII data file that should be loaded. The ASCII file must contain six initial lines giving the title, units and labels for the x and y axes, and the number of histogram bins. The rest of the file has two columns of data. The first column has the x -values for the bin boundaries and the second column has the histogram values. The column of x -values must have one more entry than the column of histogram values. Except for the `getResult()` method, the structure of `LoadASCII.java` is just like the structure of `HelloOperator.java`. The `getResult()` method first gets the current value of the `file_name` parameter, then reads the data from the file into local variables and arrays. It would be easy to modify this operator to load data that just consists of x,y pairs rather than a histogram. It would only be necessary to create the array `x[]` to be of the same size as the array `y[]` and remove the line that reads the last bin boundary.

The data is read from the file using a utility class `TextFileReader` in `DataSetTools/util`. This class makes it simple to read a sequence of numeric, `char` or `String` values from a text file. It also supports putting back the last item that was

read, whether it was a single character, or a numeric or string value, so it should be convenient to use when reading more complicated ASCII data file formats. In this case, only the `read_line()` and `read_float()` methods are needed. The file is opened by the `TextFileReader` constructor. If an error is encountered while reading the file, an exception is generated and the operator will return an error message.

If the data is successfully read, the `getResult()` method constructs an empty `DataSet` using `DataSetFactory`, puts the data into a `Data` block and adds the `Data` block to the `DataSet`. The `DataSet` is constructed using a `DataSetFactory`, rather than just a `DataSet` constructor since the `DataSetFactory` adds certain basic operators to the `DataSet`. Also, a `String` indicating that the data was loaded from the specified file is added to the `DataSet`'s log.

The operator has its own main program that can be used to test it. The operator's main program just loads the data from the file `LoadASCII.dat` and then pops up a viewer to display the data. It can be tested by running the operator from within the directory containing the sample data file, `LoadASCII.dat`. Specifically, if the current directory contains `LoadASCII.dat`, and `LoadASCII.java` has been compiled, it should work to give the command: `java Operators.LoadASCII`

3.6.2 EXAMPLE: `Ysquared.java`

The `Ysquared` operator provides a very simple example of accessing and altering the y-values stored in a `DataSet`. The operator takes only one parameter, a `DataSet`. The `getResult()` method steps across each `Data` block in the `DataSet`. For each `Data` block, a reference to the array of y-values of the `Data` block is obtained. Using that reference to the array of y-values, the y-values are then altered by squaring each one.

The main test program for this operator first loads an IPNS runfile, clones it and displays it. The `DataSet` must be cloned, since the `Ysquared` operator alters the `DataSet` it is given. The main test program then constructs and applies a `Ysquared` operator to the original `DataSet` and displays the result. In order to use this main test program, it will be necessary to edit the `String` giving the `run_name` so that it points to an actual file that can be read. If the operator is used from within ISAW, the `DataSet` can be any `DataSet` that has been loaded into ISAW.

3.6.3 EXAMPLE: `CenteredDifferences.java`

The `CenteredDifferences` operator creates a new `DataSet` containing approximate numerical derivatives of a specified `DataSet`. The numerical derivatives are calculated using the centered difference approximation: $dY = (Y_{i+1} - Y_{i-1}) / (X_{i+1} - X_{i-1})$. The new `DataSet` has the same x units and label as the original `DataSet`, but has y units and label indicating that it is the derivative of the original `DataSet`.

As was the case with the `Ysquared` operator, the main test program loads an IPNS runfile and will have to be altered to point to a file on your system.

3.6.4 EXAMPLE: IntegratedIntensityVsAngle.java

The `IntegratedIntensityVsAngle` operator is somewhat more involved than the previous examples. In this example, we calculate the total number of counts over an interval $[a, b]$ and record the integrated intensity as a function of the scattering angle 2θ .

Several additional techniques are illustrated by this example. First, the operator has three parameters, a `DataSet` and two floats, a , b , giving the interval over which the histogram is integrated. Since the floats are stored internally in java `Float` objects, a little bit of effort is needed to get the values from the list of parameters. Unfortunately, the simple numeric type `float` is very different from the class `Float` in Java and so it is necessary to do some work to get at the numeric value that we want. The expression on the right hand side of the equation:

```
float a = ((Float)(getParameter(1).getValue())).floatValue();
```

gets the first parameter from the operator, then gets the `value` object for that parameter. The `value` object in this case is a `Float` object, so the object is type cast to type `Float`. Finally, we want the numeric `float` value from this `Float` object, so we use the `floatValue()` method of class `Float` to get the value that we will calculate with.

After getting the parameters, the `getResult()` method checks that the `DataSet` and interval are valid and returns an `ErrorString` if they are not valid. Next, a new `DataSet` is obtained and the log and attributes from the original `DataSet` are copied to the new `DataSet`. A new log entry indicating the new calculation is also added to the log.

Since the `X Scales` for `DataSet` must be increasing sequences, the original `DataSet` is sorted in increasing order based on the detector position attribute. The sort is carried out using a `DataSetSort` operator.

The core calculation consists of processing each histogram of the `DataSet` to obtain the total counts on the interval $[a, b]$ and to obtain the scattering angle, 2θ , for that detector. Since the user specified interval $[a, b]$ may not be aligned with the bin boundaries, the endpoints may split a histogram bin. These technicalities are taken care of by the utility `NumericalAnalysis.IntegrateHistogram()`. The detector position is assumed to be an attribute of the Data block, and the example code demonstrates how to obtain it. A `DetectorPosition` object records a position in three dimensions and has methods that return the coordinates of the position using Cartesian, cylindrical or spherical coordinates. The origin is at the sample and the x axis is assumed to be in the beam direction. The positive y axis is assumed to be pointed upward. In this example, a convenience method of the `DetectorPosition` class, `getScatteringAngle()`, is used to calculate the angle between the scattered beam and the positive x axis.

When the integrated intensity and scattering angle for each detector have been calculated and stored in arrays, one more technicality must be dealt with. It is likely that more than one detector will have the same scattering angle. The later part of the `getResult()` method checks for such duplicates and replaces values corresponding to several detectors at the same scattering angle with one value (the average) at that scattering angle.

Finally, the area and angle arrays with duplicates removed are placed in a Data block, in the new `DataSet` and the new `DataSet` is returned.

3.6.5 EXAMPLE: `OperatorTemplate.java`

The `OperatorTemplate` file includes examples of parameters of types `DataSet`, `int`, `float`, `boolean` and `String`. The `getResult()` method returns a `String` indicating that the operator was called and what the parameter values were. This "template" may provide useful starting point for building a new operator as follows:

1. Rename the `OperatorTemplate.java` file to the name of the new operator.
2. Edit the new file to change the class name, constructor names, title, command name and clone method so that they are appropriate for the new operator. Also, change the operator name in the main test program.
3. Compile the renamed template, and make sure that it still runs along and can be accessed from ISAW and Scripts, as needed.
4. Adjust the list of parameters as required in both the constructor and in the `setDefaultParameters()` method.
5. Add the code for the new operator to the `getResult()` method. Modify the main test program to do some basic tests of the operator, if possible.
6. Rewrite the documentation to match the new operator.

This process may be simplified somewhat by using the `OperatorBuilder` utility. The `OperatorBuilder` gets the required information and places it in a new java file, in the proper format. It basically creates a custom `OperatorTemplate`, and is more convenient and less error prone to use than modifying the `OperatorTemplate` by hand.

3.7 Categories for Operators

The operators can be split into the categories described in Table 3.1. Here the `Generic` operators are NOT associated with any `DataSet`. They can be invoked from the *CommandPane*. Some of these operators would NOT make sense to invoke from ISAW and would be categorized as `Batch`. Those that do make sense to invoke from ISAW could be placed in the ISAW menus. Currently there is only one such category... various forms of loading `DataSets` that could be invoked from the ISAW *File* menu.

The `DataSet` operators are kept with the `DataSet` and would appear in the same menu that they currently appear in. However, the categories introduce an additional level to the menu hierarchy.

The categories are implemented using an inheritance hierarchy so the categories could be determined via "instanceof". This is also better adapted to the hierarchy of menus, than a "flat" `getCategory()` method that only returns one category name. For compatibility, `getCategory()` would still be supported.

Two of Dongfeng's operators do not fit nicely into this scheme, `SpectrometerPlotter` and `DataSetPrint`. The `SpectrometerPlotter` operator is really just a single Data block plotter and has nothing to do with `Spectrometers`. It should be replaced by a

Table 3.1: Categories of operators and their associated java class.

CATEGORY	JAVA CLASS
Operator	Operator
• Generic	GenericOperator
◦ Load	GenericLoad
◦ Batch	GenericBatch
◦ Calculator	GenericCalculator
◦ TOF_DG_Spectrometer	GenericTOF_DG_Spectrometer
◦ TOF_SCD	GenericTOF_SCD
• DataSet Operator	DataSetOperator
◦ Edit List	DS_EditList
◦ Math	DS_Math
· Scalar	ScalarOp
· DataSet	DataSetOp
· Analyze	AnalyzeOp
◦ Attribute	DS_Attribute
◦ Conversion	DS_Conversion
· X Axis Conversion	XAxisConversionOp
· Y Axis Conversion	YAxisConversionOp
· XY Axis Conversion	XYAxisConversionOp
◦ Special	DS_Special

Viewer, like Kevin's. Similarly, the `DataSetPrint` operator should be replaced by the `TableView`. In the meantime, they could appear, uncategorized, at the end of the `DataSet` operator menu, or be put into a `Miscellaneous` category.

A detailed list of how the current and some future operators would fit into the categories is given below. Operators that are not yet implemented are marked with a question mark.

3.7.1 Generic Operators(`GenericOperator`)

Generic operators are NOT associated with any `DataSet` .

- Load (`GenericLoad`)
 - SumRunfiles
 - LoadMonitorDS
 - LoadOneHistogramDS
 - LoadOneRunfile
 - MergeRunfiles (Not yet implemented)
- Batch (`GenericBatch`)

- EchoObject
- pause
- Calculator (GenericCalculator)
- Direct Geometry Spectrometer (GenericTOF_DG_Spectrometer)
- Single Crystal Diffractometer (GenericTOF_SCD)

Table 3.2: String constants to use in menus are defined in class Operator:

public static final String	OPERATOR	= "Operator";
public static final String	GENERIC	= "Generic";
public static final String	LOAD	= "Load";
public static final String	BATCH	= "Batch";
public static final String	DATA_SET_OPERATOR	= "DataSet Operator";
public static final String	EDIT_LIST	= "Edit List";
public static final String	MATH	= "Math";
public static final String	SCALAR	= "Scalar";
public static final String	DATA_SET_OP	= "DataSet";
public static final String	ANALYZE	= "Analyze";
public static final String	ATTRIBUTE	= "Attribute";
public static final String	CONVERSION	= "Conversion";
public static final String	X_AXIS_CONVERSION	= "X Axis Conversion";
public static final String	Y_AXIS_CONVERSION	= "Y Axis Conversion";
public static final String	XY_AXIS_CONVERSION	= "XY Axes Conversion";
public static final String	SPECIAL	= "Special";

3.7.2 DataSet Operators (DataSetOperator)

Associated with a DataSet, the first three categories, Edit List, Attribute and Math, apply to ANY DataSet. The last categories, Conversion and Special will vary with instrument type and what operations have already been done.

The EditList operators have to do with changing the dataset by merging, sorting, etc. without instrument specific information.

- EditList (sort, delete, merge)(DS_EditList)
 - DataSetMerge
 - DataSetSort
 - DataSetMultiSort
 - DeleteByAttribute

- DeleteCurrentlySelected
- ExtractByAttribute
- ExtractCurrentlySelected (?)

The **Math** operators perform scalar transformations on the datasets, mathematical operations using more than one dataset, and returning results from analyzing the dataset.

- Math (DS_Math)
 - Scalar (ScalarOp)
 - * DataSetScalarAdd
 - * DataSetScalarSubtract
 - * DataSetScalarMultiply
 - * DataSetScalarDivide
 - DataSet (DataSetOp)
 - * SumByAttribute
 - * SumCurrentlySelected
 - * DataSetAdd
 - * DataSetSubtract
 - * DataSetMultiply
 - * DataSetDivide
 - * DataSetAdd_1 (?) (operations using ONE Data block of a)
 - * DataSetSubtract_1 (?) (DataSet with ALL Data blocks of a second)
 - * DataSetMultiply_1 (?) (DataSet. Not yet implemented.)
 - * DataSetDivide_1
 - Analyze (integrate, curve fit) (AnalyzeOp)
 - * DataSetCrossSection
 - * IntegrateGroup
 - * CalculateMomentOfGroup
 - * ConvertHistogramToFunction
 - * ResampleDataSet
 - * $x = T(x)$ (?) (transformations applied to x, y)
 - * $y = T(y)$ (?) (values in Data blocks)
 - * $x = T(x, y)$ (?) (Not yet implemented)
 - * $y = T(x, y)$ (?)
 - * FFT (?)
 - * FitPeak(s) (?)

The **Attribute** operators are a collection of accesor/mutator type operations.

- Attribute (DS_Attribute)
 - GetDSAttribute
 - GetDataAttribute
 - GetField
 - SetDSAttribute
 - SetDSDataAttributes
 - SetDataAttribute
 - SetField

The Conversion operators are similar to both the EditList and Math operators except they are using instrument specific methods.

- Conversion
 - X Axis Conversion (XAxisConversionOp)
 - * DiffractometerTofToD
 - * DiffractometerTofToEnergy
 - * DiffractometerTofToQ
 - * DiffractometerTofToWavelength
 - * SpectrometerTofToEnergy
 - * SpectrometerTofToEnergyLoss
 - * SpectrometerTofToQ
 - * SpectrometerTofToWavelength
 - * TofToChannel
 - Y Axis Conversion (YAxisConversionOp)
 - * TrueAngle
 - XY Axes Conversion (XYAxisConversionOp)
 - * SpectrometerTofToQE

The Special operators depend not only on the instrument but also that other operations have already been performed.

- Special (DS_Special)
 - MonitorPeakArea
 - EnergyFromMonitorDS
 - DoubleDifferentialCrossection
 - SpectrometerDetectorNormalizationFactor
 - SpectrometerFrequencyDistributionFunction
 - SpectrometerGeneralizedEnergyDistributionFunction

- SpectrometerImaginaryGeneralizedSusceptibility
- SpectrometerMacro
- SpectrometerScatteringFunction
- SpectrometerSymmetrizedScatteringFunction
- SpectrometerNormalizer (This is now obsolete)
- SpectrometerEvaluator
- TimeFocus (?) (Not yet implemented)

3.7.3 Not Categorized Operators, Misc

This is a list of operators that do not fit in anywhere else in the operator hierarchy.

- DataSetPrint (Replace with TableView ?)
- SpectrometerPlotter (Dongfeng's "operator" to plot one group, using the Australian graphics package. This really should be done with a viewer.)

Chapter 4

Live Data

A feature of ISAW that needs special documentation is the live data server. This allows someone (with a bit of setup overhead) to view data currently being measured. This is useful for determining if it is worthwhile to continue a particular measurement before too much beam time is used.

The process of viewing live data is split up between three distinct processes (which may or may not be running on the same computer):

- **DAS Sender** resides on the DAS computer and sends data to the instrument computer on a named port using MultiNet UDP. UDP is a connectionless protocol where packets are sent to a location with indifference as to the state of the recipient (whether or not it is receiving the packets).
- **Live Data Server** resides on the the instrument computer and listens for both TCP and UDP connections at different ports (6088 for TCP and 6080 for UDP by default). The server will receive whatever UDP packets are sent to it and send any TCP packets that are requested from it.
- **ISAW Client** resides on the end-user's computer. This connects to the Live Data Server and requests (using TCP) live data.

The client computer has no knowledge of the data's origin and the sender does not know if anything is receiving the data it is sending out. The details of UDP and TCP protocols are beyond the scope of this manual, information about the protocols can be found at <http://www.faqs.org/rfcs/rfc768.html> (UDP) and <http://www.faqs.org/rfcs/rfc793.html> (TCP).

A test version of the sender and the actual server come with the standard distribution of ISAW. Running the software happens in three steps.

1. The Server can be started from anywhere on the instrument computer by typing
 - `java NetComm.LiveDataServer -D full_data_path`
2. Start the Sender from the data directory using
 - `java NetComm.DASOutputTest inst_prefix run_number`

Table 4.1: The format of the packet from the UDP sender. Remember this is converted from little to big endian to be interpreted by the live data server. The total size of the packet is 16300 bytes or 4075 long data words.

HEADER		
21+l bytes - normally 25 bytes		
4 byte	magic number	sent as a pseudopassword
1 byte	instrument name length	variable but is always 4 at IPNS (l)
l byte	instrument name	can be changed using the instrument name length
4 byte	run number	
4 byte	group ID	which is the detector subgroup
4 byte	first channel index	in this packet (i_0)
4 byte	number of channels	being sent in this packet (n)
SPECTRUM		
0 to 4068 words		
4 byte	intensity[i_0]	intensity in first channel
4 byte	⋮	
4 byte	intensity[$i_0 + n$]	intensity in last channel

- From anywhere start Isaw and edit `IsawProps.dat` to change `Inst#_Name` to be descriptive of where you are running the live data server from (this name will appear in the menu) and `Inst#_Path` to be where you are running the live data from (with port number). This will allow access to the live data through the drop-down data menu.

4.1 UDP Data Format

The form of each packet being sent from the UDP Sender is seen in Table 4.1. The format of the data is read from the run file then the data itself is read from the hardware. The limitation of the packet being only 16300 bytes is to allow for the historical 16384 byte packet limit with some room for overhead.

4.2 DAS Sender

4.3 Live Data Server

As mentioned above, the Server is intended to reside on the instrument computer with two threads running: one for UDP connections and one for TCP connections. The program is really quite simple if the details are skipped. Upon calling `LiveDataServer.java` the arguments are parsed, the server name is printed to screen,

the logfile is initialized, the data directories are confirmed, then the UDP and TCP listen threads are initialized. Other than starting the threads (and what they do) the program does very little. The data directory is specified so that the `LiveDataServer` can access the empty runfile that is created when the run is started. Since data is sent as spectrum without detector positions or time information the Server obtains that information from the start run file header on the Data Server system. Currently the data is then put into a `DataSet` before being serialized and sent across the network to the client. The problem with this is that any changes in the `DataSet` Class, Attributes, or Operators associated with the data break this mechanism.

4.4 ISAW Client

Chapter 5

NeXus

The NeXus reader and writer are extremely preliminary version. Currently they work for “usual” IPNS data sets. There needs to be a lot of work done to get this to the level of reading and processing general NeXus files.

There are two files (external to ISAW) which need to be obtained before working with nexus files.

- NeXus library file
- `jnexus.jar`

The `jnexus` library provides an interface for java to read and write the NeXus files. The NeXus library file provides the actual methods that are called by `jnexus` to read and write the data.

5.1 Format

Each file can have multiple entries. An entry is data with information used to describe what was measured and how it was done. An entry consists of the elements as described in Table 5.1. While the order of this information is not very necessary (it is all obtained through library functions) the contents of the file is. It is important to note that NeXus is a specification for how information will be formatted if it is present in the file. However, there is no guarantee that the information will be present.

Table 5.1: Format of a NeXus entry.

SAMPLE
INSTRUMENT
source
equipment
detector
DATA

Chapter 6

ASCII File Formats

6.1 Table View

The *Table View* in ISAW allows for the display (and exporting) of a tab delimited table of information.

6.2 GSAS Data File Format

GSAS is a Reitveld refinement program which is used throughout the crystallography community. As a first note, GSAS reads fixed record length files. This means that it contains some headers followed by blocks of data related to the header, repeated as much as desired. All lines must be 80 characters in length.

6.2.1 Standard Powder Data File

The standard powder data file specification is found in the gsas technical manual. This will, in general, be slightly different than any powder data file that one will actually run across. For example, the version of the powder data file that ISAW writes out also contains the detector group positions (reference angle and total path length) as well as the integrated monitor counts. Applications that can read gsas files, in general, know that these extra entries exist (in certain places in the file) and will skip over them. For this reason what is described below is the ISAW extension of the powder data format.

It is easiest to start with an example file. The first eleven lines of our example are:

```
NdDyCaBa1.5Sr.5Cu2.2Ti2.8014-d,7.996gr,#2
MONITOR: 4507029.0
# BANKS      Ref Angle      Total length
#BANK 1      -144.84573     15.5
#BANK 2      -90.00005      15.5
#BANK 3      -60.000042     15.5
```

```

#BANK 4      -14.624277      15.5
#BANK 5      29.780281      15.5
BANK 000001  005600      000560 CONST 002000.0000000      05.0000000 STD
  010694  010740  010371  010488  010566  010505  010492  010372  010687  01031
  010568  010641  010335  010422  010323  010342  010517  010243  010126  01015

```

The first line is the title for then file, followed by the integrated monitor count (not used by gsas) then six comment lines (comment lines begin with '#'). Then the bank header followed by the counts in each channel. The MONITOR keyword is not recognized by gsas but is used by PDFgetN. The bank information is included for completeness by ISAW. Bank header information, in this case, is the bank number, number of channels, number of records (lines), constant time binning keyword, t_0 of the first time channel, Δt of the time channels, and STD determines that $\sigma_I = \sqrt{I}$.

The general form of a powder data file (without extensions) is Comment lines and

Title
Instrument parameter Filename
comment lines
BANK BankNum Nchan Nrec BinType Bcoef1 Bcoef2 Bcoef3 Bcoef4 Type data
comment lines
BANK BankNum Nchan Nrec BinType Bcoef1 Bcoef2 Bcoef3 Bcoef4 Type data
:

the instrument parameter file name can be omitted. Each line must be 80 characters in length. However, for the title, only the first 66 characters are retained by gsas. If the name of instrument parameter file (see below) is missing EXPEDT will ask for it. Comment lines can appear just before any bank header. The bank information (header and data) can be repeated as many times as necessary. The bank header has the following keywords:

BankNum The number assigned to the detector group. If this is zero gsas assumes it is a monitor spectrum.

Nchan The total number of intensities listed in this bank.

Nrec How many lines the data is presented in. The number of channels in a line are specified by the **Type**. For example, STD specifies that $Nrec = Nchan/10$ (rounded up).

BinType How the data is presented (what is the x-axis). Valid values of **BinType** are COND, CONST, CONQ, EDS, LOG6, LPSD, RALF, SLOG, and TIME_MAP. These are discussed in Table 6.1, values not listed are not supported by ISAW.

Bcoef1-4 Coefficients used to elaborate on the binning chosen using **BinType**.

Table 6.1: Description of the different values of BinType.

	BinType	Bcoef1	Bcoef2	Bcoef3	Bcoef4
constant t -spacing	CONST	t_0	Δt	N/A	N/A
constant d -spacing	COND	d_0	Δd	N/A	N/A
constant Q -spacing	CONQ	Q_0	ΔQ	N/A	N/A
constant $\Delta t/t$	SLOG	t_0	$\max t$	$\Delta t/t$	N/A
time map listed	TIME_MAP	TM#	N/A	N/A	N/A

Table 6.2: Description of the different values of Type. If Type is blank STD is assumed. For STD, NC is the number of counters, if it is not specified it is assumed to be 1.

Type	Nrec		
STD	Nchan/10	10[NC(2I), I(F6)]	$\sigma_I = \sqrt{I}$
ESD	Nchan/5	5[I(F8), σ_I (F8)]	

Type Specification of the errors. Valid values are STD, ESD, and ALT. If the record is left blank then STD is assumed. These are discussed in Section 6.2, ALT is not supported by ISAW.

The TIME_MAP keyword specifies that the binning was not in any of the other formats. A new listing is made which contains all of the information to reconstruct the binning. The header is of the form

TIME_MAP TM# Nvals Nrec TIME_MAP CLKWDT

TM# is the number of the time map, Nvals is number of data items, Nrec is Nvals/10 (rounded up), and CLKWDT is the number of clock pulses per microsecond. Each record contains ten data items. However, each time bin is specified using three numbers (all eight digit integers): starting channel number, t in pulses of width CLKWDT, and channel width in pulses.

6.2.2 Instrument Parameter File

```

1234567890123456789012345678901234567890123456789012345678901234567890
INS BANK 5 INS FPATH1 14.00 INS HTYPE PNTR INS NSPEC 5 INS 1 ICONS
7483.17 -2.16 -8.99 INS 1BNKPAR 1.5000 144.845 0.00 .00000 .3000 1 1
INS 1I HEAD V-rod/no shields RT CoolPower dispdex Col#3 INS 1I ITYP 2
3.0000 29.9950 33857 INS 1ICOFF1 0.297403E+01 0.154201E+09
0.130196E+03 0.598090E+04 INS 1ICOFF2 0.942353E-01 -0.282635E+04
0.253667E-01 0.000000E+00 INS 1ICOFF3 0.000000E+00 0.000000E+00
0.000000E+00 0.000000E+00 INS 1IECOF1 0.141198E+00 0.963943E+06
0.655670E+00 0.670946E+03 INS 1IECOF2 0.386717E-02 0.410999E+03

```

0.670279E-03 0.000000E+00 INS 1IECF3 0.000000E+00 0.000000E+00
0.000000E+00 0.000000E+00 INS 1IECOR1 1.000-0.645-0.553 0.259
0.347-0.258 0.414 0.000 0.000 0.000 0.000 INS 1IECOR2 0.000 0.000 1.000
0.942-0.505-0.650 0.505-0.733 0.000 0.000 INS 1IECOR3 0.000 0.000
0.000 1.000-0.653-0.798 0.655-0.825 0.000 0.000 INS 1IECOR4 0.000
0.000 0.000 1.000 0.971-0.998 0.420 0.000 0.000 0.000 INS 1IECOR5
0.000 0.000 1.000-0.973 0.616 0.000 0.000 0.000 0.000 0.000 INS
1IECOR6 1.000-0.446 0.000 0.000 0.000 0.000 0.000 1.000 0.000 0.000
INS 1IECOR7 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 INS 1IECOR8 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 INS
1PRCF1 1 7 0.01000 INS 1PRCF11 0.000000E-01 0.218000E+00 0.385300E-01
8.998e-03 INS 1PRCF12 6.0630 0.160800E+02 3.8630

6.3 spec Standard Data File Format