The AIPS++ Programming Manual
AIPS++ standards and libraries

B.E. Glendenning W. Young(Eds.)

October 15, 2006

Contents

AIPS++ programming manual.

The AIPS++ consortium may be reached by email at aips2-request@nrao.edu. The postal address is: AIPS++ Consortium, c/o NRAO, 520 Edgemont Rd., Charlottesville, Va. 22903-2475 USA.

Part I

Programming in AIPS++

Chapter 1

Introduction

This document is intended for use by people who are either implementing code for `aips++` or who are using `aips++` libraries for their own work but are not actively extending the system.

While this document is meant for people who are actively writing code, it must be emphasized that work in the `aips++` system should be proceeded by analysis and design phases unless the functionality is very straightforward. The methodology that the `aips++` project has chosen is the *Object Modeling Technique* (OMT) [1] or *Rumbaugh* methodology.

The Programmers Reference Manual is part of the `aips++` *Documentation System* — other titles which may be of interested include:

- System System installation and management.
- Getting Started in AIPS++ A User's guide to getting started in AIPS++.
- AIPS++ How to .* Useful tidbits from the AIPS++ group.

This manual assumes that you are reasonably fluent in C++. If that is not the case, we suggest that the following would make a good bookshelf of C++ books:

- *The C++ Programming Language (Second Edition)*; Stroustrup (Addison Wesley:1991). This is a good introductory book (the first edition, however, was terrible).
- *The C++ Primer???? (Second Edition)*; Lippman (Addison Wesley?:1991??). This is another good introductory book.

[1] Object-Oriented Modeling and Design; Rumbaugh, Blaha, Premerlani, Eddy, and Lorensen (Prentice-Hall:1991).

- *The Annotated C++ Reference Manual*; Ellis and Stroustrup (Addison Wesley:1990). This is the book the (unfinished) C++ standardization effort is based on. Until the language is standardized this is the "bible."
- *Advanced C++: Programming Styles and Idioms*; Coplien (Addison Wesley: 1992). A book that shows you how to do things when the more elementary techniques fail.
- *Effective C++*; Meyer (Addison Wesley:1992). A very useful compendium of things to know about programming in C++. You should probably be able to understand everything in this book before writing any `aips++` classes.
- *C++ Programming Guidelines*; Plum and Saks (Plum Hall:1991). This book is dull reading by it has a lot of useful information.

Please let us (Brian Glendenning, bglenden@nrao.edu, or Wes Young, wyoung@nrao.edu) know of any enhancements or changes that you feel should be made to this document. If you are an aips2-worker you can of course make any additions or changes yourself (please, however, check with us first).

A word about *Standards* and *guidelines*. There are two competing views about these:

1. That they are invaluable. A consistent "look and feel" allows you to much more quickly understand what the code is trying to do. Moreover, some standards will outlaw questionable practices and squash bugs before they can be written.
2. They are not very important. The semantics of the code take all the time it takes to understand it. We're professional enough to understand what we're doing and rigid rules won't always work. Besides, I just gotta be me.

In `aips++` we adopt the first view-point. It might only alleviate only the most superficial problems, but better something than nothing. The current list is not that large or onerous in any event. (In fact, for standards and guidelines to be followed, there must be few enough of them that it is reasonable to expect people to know them).

Chapter 2

Introduction to Programming for AIPS++

This chapter [1] provides a gentle introduction to programming with `aips++` classes.

2.1 A First Program

We will start with a small sample program which prints the values generated by a simple function. The object, of course, is merely to explain the conventions and procedures used in an `aips++` program (The source code for this program is found in $AIPSROOT/code/trial/implement/test/tExample1.cc).

```
#include <aips/aips.h>
#include <aips/Exceptions.h>
#include <aips/Mathematics/Math.h>
#include <iostream.h>

main(int argc, char **argv)
{
 const Int     n(8);
 const Float   interval(10.0);
 Float   x, y;

 try {
```

[1] Originally written by R.T. Duquet and updated by W.K. Young.

```
13        for (Int i=0; i<n; i++) {
14           x = i;
15           y = i * sin(i / interval);
16           cout << x << "\t" << y << endl;
17        }
18     }
19
20     catch (AipsError x) {
21        cerr << "aipserror: error " << x.getMesg() << endl;
22        return 1;
23     }
24     return 0;
25    }
```

The following discussion refers to the line numbers which have been prepended to the code above:

1- Every `aips++` program must contain this line. It declares variable types Int, Float, etc. plus other goodies used for debugging.

2- This class handles exceptions for `aips++` programs.

3- This calls the routine that defines the sine function. The various C++ vendors do not always provide the expected routines in their math libraries. This invocation allows the `aips++` routines to call whichever versions are appropriate.

4- This is the standard IO for this C++ package. It is not directly related to `aips++`.

6- The declaration of the main program must provide for argument input. The use of argc and argv causes the compiler to issue a warning in this instance because they are not used. (They will be used in a later example.) Note that these data types are NOT written with a leading capital letter since the compiler wants native-mode variables in this instance.

8-10- Note the use of first letter capitals in the type declarations. These are defined in *aips.h*. Note too the specification of values on these lines rather than in the code body. This is to make it easier to make changes to the code. The use of the qualifier *const* means that these parameters cannot be accidentally altered in the code.

12- The *try* block (code between { and }) define the extend of the code that is to be monitored by the handlers (defined by the *catch* clauses below).

13- The use of inline declaration of variables (*Int i*) in *for* loops is strongly recommended.

20- The *catch* statement intercepts various types of errors. In this case we are asking the program to report on general errors related to `aips++`. Note that the argument x is NOT related in any way to the variable x declared on line 10.

21- In this case we are telling the program to print out the fact that an error has occured on the terminal.

To compile this program one should use the *gmake* command which will report on any errors and which will create an executable module with the name supplied. The *gmake* macro has built-in knowledge of the location of each of the included *<aips/xxxxx>* files. In order to use *gmake* you must be logged in to one of a limited number of areas. In particular, you should be in *~/aips++/code/aips/test*

Assuming the source code was named *sample1* the command would simply be

gmake NODEP=1 sample1 where the string *NODEP=1* merely tells the compiler to take a short-cut which is quite allright for modules that are not intended to be part of the `aips++` library. Note that we did not have to specify the suffix appended to the source module name. However that suffix must be *cc*.

The program is started simply by typing its name: *sample1*. The output is as follows (note the default format):

```
0       0
1       0.0998334
2       0.397339
3       0.886561
4       1.55767
5       2.39713
6       3.38785
7       4.50952
```

2.2 Second Version

We now continue by slightly expanding this program.

First, we will supply input arguments to the program in order to set various parameters and to determine the type of output desired.

Next, we will replace the scalars x and y by a class of objects known to `aips++` as a *Vector*. In order to do this we must include *<aips/Vector.h>*. As we will see, that the same class can be used to contain different types of objects. This is known in C++ as a *template*.

Elements of a *Vector* may be addressed as they are in FORTRAN, that is by the name of the vector followed by a subscript in parentheses. `aips++` however allows some operations to be applied to an entire vector. We will illustrate this.

Finally, we will provide the possibility of creating a file in `aips++` format i.e. in a format that can be read by other `aips++` programs. (The source code for this program is found in $AIPSROOT/code/trial/implement/test/tExample2.cc).

```
#include <aips/aips.h>
#include <aips/IO/AipsIO.h>
#include <aips/IO/AipsIOError.h>
#include <aips/Inputs.h>
#include <aips/Mathematics.h>
#include <aips/Arrays/Vector.h>
#include <iostream.h>

main(int argc, char **argv)
{
 Input inputs(1);
 inputs.Version("$@w{Revision}$");
 inputs.Create("vectorsize","15","Number of vector elements?","Integer");
 inputs.Create("usefile","True","Create an AIPS++ File?","Bool");
 inputs.Create("file","/tmp/sample2.test","Output file name?","Outfile");
 try {
    inputs.ReadArguments(argc, argv);

    const Int hold = inputs.GetInt("vectorsize");
    const Bool usefile = inputs.GetBool("usefile");
    const char *filename = inputs.GetString("file");

    const Float   interval(10.0);
```

```
24       const uInt    version(1);
25       Vector<float> x(hold), y(hold);
26
27       indgen(x);
28       y = x * sin(x / interval);
29       if (usefile) {
30          AipsIO outfile(filename, AipsIO::Out);
31          outfile << x << y;
32       }else{
33          for (Int i=0; i<x.nelements(); i++) {
34             cout << x(i) << "\t" << y(i) << endl;
35          }
36       }
37    }
38
39
40    catch (AipsIOError x) {
41       cerr << "aipsioerror: error " << x.getMesg() << endl;
42       return 1;
43    }
44    catch (AipsError x) {
45       cerr << "aipserror: error " << x.getMesg() << endl;
46       return 1;
47    }
48
49    return 0;
50   }
```

The following points should be observed concerning this version (as before, the numbers refer to those prepended to each line of the listing):

2- The *AipsIO* class refers to real IO as opposed to the acquisition of command line arguments which is the role of the *Input* class.

7- The *Vector* class is used to handle `aips++` style of vectors which differs, of course, from the simple arrays predefined in the C++ language.

13-15- The *Create* member function of *Input* builds from the command line three parameters referenced as *vectorsize*, *usefile* and *outfile*. All parameters are optional. The value stated as the second argument in each case is the default value suplied for use when the program is started without that specific parameter. The third argument is used for prompting (when requested by the program user - see below). The fourth argument is the type of the parameter's value. Two additional arguments are optional and are omitted in this sample program.

17- This line supplies the value of each parameter. The default values will be used unless specific values have been supplied. A *keyword* that doesn't match those anticipated in the program will be discarded as an error but the program will continue.

19-21- Each of the parameters read from the input line or supplied by default is assigned to a *const* variable to be used (but not modified) in the program. Note the use of *char* datatype, rather that a *String*, for the variable *filename*.

25- The `aips++` *Vector* data type is a *template* which means that vectors may consist of different data types such as *Int* or *Float* or *Complex*. The space required by the vectors is assigned at run time so the arrays can be dimensioned on the basis of an input parameter. Note the use of the round, rather than the square, brackets for the dimensions of the Vector.

27- The function *indgen* is supplied as part of the *ArrayMath* class. In this simple case it fills each element of the array specified as an argument with the integers *0,1,2,3,...,nelements()-1* (In other, more elaborate cases, it can fill the array with other sequences.)

28- This Vector statement is the equivalent of a loop operating element by element.

30- The definition of the AipsIO item called *outfile* within the *if* block prevents the file from being opened unless it is needed. The type of file (output) is specified in the second argument. Options are: *In, Out, Update, Append, New, NewNoReplace, Scratch* and *Delete*. Note that we did not need to OPEN the file nor do we need to CLOSE it unless it needs to be rewritten (or reread).

31- In this line the << operator is overloaded so that the output is an entire vector.

33-34- Here we deliberately use the subscript notation for elements of a *Vector*. One of the side benefits of using the `aips++` *Vector* data type is that the class contains a function, called *nelements()* which returns the current size of the vector. Its use here is obviously more flexible and, at the same time, more robust than the use of a constant.

40-43- A new class of error is caught by these statements.

As before, to compile this program one should use the *gmake* command. Once compiled, the program can be started by simply typing *sample2*. The result is a file in the */tmp* directory. This file is `aips++` compliant which means that the information is preceeded by some `aips++` identification. As noted at the beginning of this section, the file is in binary format which means, of course, that it is not suitable for display on a terminal.

As it runs, the program DOES display one line on the terminal; that line identifies the version of the program being run. (The reason this line did not appear in the output produced by the *sample1* program is because we never read the *inputs*.)

The same program can be started with a request to send output to the screen i.e. to override the default value of the *usefile* and *vectorsize* parameters. When we start it by typing @* *sample2 usefile=false vectorsize=5* we get the following

```
sample2: Version $@w{Revision}$
0       0
1       0.0998334
2       0.397339
3       0.886561
4       1.55767
```

The version number has NOT been inserted for the string @*wRevision* because we have not submitted the program to the `aips++` system library.

It my very well happen that you wish to run a program but you do not remember the names of the arguments nor their expected values. The solution is to run the program by stating its name followed by the string *help=prompt*. For example, if we type *sample2 help=prompt* we get the following:

```
sample2: Version $@w{Revision}$

Number of vector elements? [15]: vectorsize=

Create an {\tt aips++} File? [True]: usefile=
```

```
Output file name? [/tmp/sample2.test]: outfile=
```

Notice the appearance of the text that we inserted as one of the *inputs.Create* arguments. That text is followed by the default value (also specified as a *inputs.Create* argument). Finally we have the name of the argument. We can either supply a new value or merely hit the *RETURN* key to accept the default value shown in the square parentheses.

2.3 An advanced example

The following program is meant to illustrate some further features of the AIPS++ system. In particular, we will show a program that was used to test the development of two functions which read and write an ASCII file. The file is expected to contain a 2-dimensional array (of any rectangular dimensions) of numbers and place the values it contains (regardless of type) into a matrix.

The work will be described in three parts - which correspond to the three directories that should be used for this type of work. They are:

1. The directory in which the main routine is composed and compiled. This is the one that has been used in previous examples. It is: **~/aips/code/aips/test**.

2. A directory in which the **.h** file is defined. This is: **~/aips/code/aips/include**.

3. A directory in which the functions are composed and compiled. This one also will contain a private library which will be searched automatically by the **gmake** process. This one is: **~/aips++/code/aips/implement**.

2.3.1 The main program

We start with the main program which will simply call a couple of routines to check the coding (The source code for this program is found in $AIPS-ROOT/code/trial/implement/test/tExample3.cc).

```
#include <aips/aips.h>
#include <aips/AipsIO.h>
#include <aips/ArrayIO.h>
#include <aips/Input.h>
#include <aips/Math.h>
#include <aips/Matrix.h>
#include <aips/AsciiFileIO.h>

main(int argc, char **argv)
{
 Input inputs(1);
 inputs.Version("$w{Revision}$");
 inputs.Usage("Testing Ascii IO routines");
 inputs.Create("infile1","Input1","Float Input file name?","Infile");
 inputs.Create("infile2","Input2","Int Input file name?","Infile");
```

```
inputs.Create("infile3","Input3","Double Input file name?","Infile");
inputs.Create("outfile","Output","Output file name?","Outfile");
inputs.ReadArguments(argc, argv);

const char *filein1 = inputs.GetString("infile1");
const char *filein2 = inputs.GetString("infile2");
const char *filein3 = inputs.GetString("infile3");
const char *fileout = inputs.GetString("outfile");

Int rows, cols;
try {
   Matrix <Float> mat1;
   Matrix <Int> mat2;
   Matrix <Double> mat3;

   ReadAsciiFile(filein1, mat1);
   mat1.shape(rows, cols);
   cout << "The first Matrix is " << rows << " by " << cols << endl;

   ReadAsciiFile(filein2, mat2);
   mat2.shape(rows, cols);
   cout << "The second Matrix is " << rows << " by " << cols << endl;

   ReadAsciiFile(filein3, mat3);
   mat3.shape(rows, cols);
   cout << "The third Matrix is " << rows << " by " << cols << endl;

   WriteAsciiFile(fileout, mat1);

   cout.precision(12);
   cout << mat1 << mat2 << mat3;
  }

return 0;

 catch (AipsError x) {
 cerr << "aipserror: error " << x.getMesg() << endl;
       return 1;
      }

```

```
58  }
```

The following discussion refers to the line numbers which have been prepended to the code above:

7- This line includes the new routines in the code for **AsciiFileIO**.

15-17- The program will read 3 different arrays of differing characteristics.

28-30- The Matrices are declared to be of three differing types but the sizes of the arrays is not stated. It will be set dynamically by the subroutines.

32- The **ReadAsciiFile** function is called to read a matrix of floating point values. The type is specified in the declaration of **mat1** which is used as an argument. The size of the array is still unknown. Note that the data in the array may be in integer form but it will be stored as an array of float values.

33- The member function **shape** is called to determine the dimensions of the matrix that was just read.

36- The same new function, **ReadAsciiFile**, is called to read an array of integers. The data may be specified with decimal points and fractional parts but they will be placed in the matrix as integers.

37- The member function **shape** is used to get the dimensions of the matrix **mat2**. Note that these dimensions are quite independent of the dimesions found for the matrix **mat1**.

40-41- The same comments hold for a matrix of doubles.

44- The routine to write the array, **WriteAsciiFile**, need not pass the dimensions of the array since they can be determined in the same way as they were in lines 33, 37 and 41.

46- The property of cout, **precision**, is used to set the number of decimal places that will appear in the output (where they are not all zero). Other properties of the **format state** may also be set. For example, one could set the number of decimal places displayed to a constant (regardless of whether or not they were all zeroes) by using the statement **cout.setf(ios::showpoint)**.

2.3.2 The include file

This file containd the **.h** module. The working part of this file is really only two lines long but the required "boiler-plate" takes up the first 25 lines (approximately). This text must be included with every file that is intended for the AIPS++ system (i.e. that could be distributed outside of NRAO).

This include file may be found in $AIPSROOT/code/trial/implememnt/test/FileIO.h.

```
//# FileIO.h: this defines ileIO functions, which ...
//# Copyright (C) 1996,1999,2001
//# Associated Universities, Inc. Washington DC, USA.
//#
//# This library is free software; you can redistribute it and/or modify it
//# under the terms of the GNU Library General Public License as published by
//# the Free Software Foundation; either version 2 of the License, or (at your
//# option) any later version.
//#
//# This library is distributed in the hope that it will be useful, but WITHOUT
//# ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
//# FITNESS FOR A PARTICULAR PURPOSE.  See the GNU Library General Public
//# License for more details.
//#
//# You should have received a copy of the GNU Library General Public License
//# along with this library; if not, write to the Free Software Foundation,
//# Inc., 675 Massachusetts Ave, Cambridge, MA 02139, USA.
//#
//# Correspondence concerning AIPS++ should be addressed as follows:
//#        Internet email: aips2-request@nrao.edu.
//#        Postal address: AIPS++ Project Office
//#                        National Radio Astronomy Observatory
//#                        520 Edgemont Road
//#                        Charlottesville, VA 22903-2475 USA
//#
//#
//# $Id: IntroPrg.tex,v 19.0 2003/07/16 04:18:22 aips2adm Exp $

#if !defined (AIPS_FILEIO_H)
#define AIPS_FILEIO_H

#if defined(_AIX)
#pragma implementation ("AsciiFileIO.cc")
#endif

#include <aips/aips.h>
#include <aips/Arrays/ArrayIO.h>
```

```
//# Forward Declarations

// <summary>
Input/output using ASCII format
// </summary>

// <use visibility=local>   or   <use visibility=export>

// <reviewed reviewer="" date="yyyy/mm/dd" tests="" demos="">
// </reviewed>

// <prerequisite>
//   <li> SomeClass
//   <li> SomeOtherClass
//   <li> some concept
// </prerequisite>
//
// <etymology>
// </etymology>
//
// <synopsis>
// </synopsis>
//
// <example>
// </example>
//
// <motivation>
// </motivation>
//
// <templating arg=T>
//    <li>
//    <li>
// </templating>
//
// <thrown>
//    <li>
//    <li>
// </thrown>
//
// <todo asof="yyyy/mm/dd">
```

```
 79  //   <li> add this feature
 80  //   <li> fix this bug
 81  //   <li> start discussion of this possible extension
 82  // </todo>
 83
 84 template <class T>
 85    void
 86    readAsciiFile(const char *filein, Matrix<T> &mat){
 87       readAsciiMatrix(mat, filein);
 88    }
 89 template <class T>
 90    void
 91    readAsciiFile(const char *filein, Vector<T> &vec){
 92       readAsciiVector(vec, filein);
 93    }
 94
 95 //
 96
 97 template <class T>
 98    void
 99    writeAsciiFile(const char *fileout, const Matrix<T> &mat){
100       writeAsciiMatrix(mat, fileout);
101    }
102 template <class T>
103    void
104    writeAsciiFile(const char *fileout, const Vector<T> &vec){
105       writeAsciiVector(vec, fileout);
106    }
107  #endif
```

1-27- Boilerplate which is to be included in each module.

29-30- The code begins with a label that identifies this file. The macros **if ! defined** and **define** prevent the code from being loaded more than once in a given program. (Note the **endif** macro at the end.)

34- Every header file MUST include this line which, in turn, must NOT be included with the code for the function.

35- The Matrix class is included because this header file describes functions that take a Matrix as an argument.

39-82- These lines are the documentation template for each class. Cxx2html will read these lines and produce formated HTML for your class/functions. A more complete description maybe found in the Programmer's Reference Manual If you ever want to check you're code in then you must fill them in.

84-86- These lines (which, you will note, are a single statement) declare the function **ReadsAsciiFile** to be a template i.e. to be defined once for multiple types of argument. The symbol uses here to represent the argument type is the letter **T** (declared in the pointed brackets with the keyword **class**). The function itself is like any other function declaration except that the Matrix argument is defined for the multiple types represented by the letter **T**.

The implications of this are important. Since the class is only specified later on (when the routine is called) the compiler cannot create ahead of time a module that will be linked with the calling code. So even though we will place these routines in a library, it really is a source library and every time a module is used it must be compiled at that time!

87-89- The lines declare the second function **WriteAsciiFile**.

91- This matches the test for label declaration on line 31.

2.4 The functions

The following code defines the two functions. (The source code for this program is found in $AIPSROOT/code/trial/implement/test/tExample1.cc).

```
#include <aips/IO/AipsIO.h>
#include <aips/Arrays/ArrayIO.h>
#include <aips/Arrays/ArrayMath.h>
#include <aips/Containers/Block.h>
#include <aips/Inputs/Input.h>
#include <aips/Mathematics/Math.h>
#include <aips/Arrays/Matrix.h>
#include <aips/Arrays/Vector.h>
#include <iostream.h>
#include <fstream.h>             // needed for file IO
#include <strstream.h>           // needed for internal IO


 template <class T>
 void
 ReadAsciiFile(const char *filein, Matrix<T> &mat)
   {
    const Int      bufSize = 1024, numberSize = 50;
          char     buf[bufSize], buf2[numberSize];
          uInt     blockSize = 100;
          Block<T> temp(blockSize);

    ifstream iFile;
    iFile.open(filein, ios::in);
    if (! iFile) {
       cerr << "Cannot open " << filein << "for reading\n"; exit (-1);}

    Int rows = 0, cols = 0, saveCols = 0;
    uInt havePoint = 0;

    while (iFile.getline(buf, bufSize)) {
       Int blankLine = 1;
       for (Int j1=0;j1<bufSize;j1++) {
          if (buf[j1] == '\0') break;
          if (buf[j1] != ' ') {blankLine = 0; break;}
```

```
         }

       if (! blankLine) {
          if (havePoint > (blockSize - saveCols - 10)) {
             blockSize *= 2;
             temp.resize(blockSize);
            }

          rows += 1; cols = 0;
          Int ch = 0, startedNew = 0;
          for (Int i2=0; i2<bufSize; i2++) {
             if (buf[i2] == '\0' || buf[i2] == ' ') {
                if (ch > 0) {
                   buf2[ch] = ' ';
                   istrstream(buf2,sizeof(buf2)) >>  temp[havePoint];
                   havePoint += 1;
                   ch = 0;
                  }
                startedNew = 0;
                if (buf[i2] == '\0') break;
               }

             if (buf[i2] != ' ' && startedNew == 0) {
                cols += 1;
                startedNew = 1;
               }

             if (startedNew) buf2[ch++] = buf[i2];
            }

          if (rows == 1) saveCols = cols;
          else if (cols != saveCols) {
             cout << "Array is not regular.  Number of elements was "
                  << saveCols << " at row 1"
                  << " but is " << cols << " at row " << rows << endl;
             exit (1);
            }
         }
      }
    iFile.close();
```

```
 76
 77      mat.resize(rows, cols);
 78      Int k3 = 0;
 79      for (Int i3=0;i3<rows;i3++)
 80         for (Int j3=0;j3<cols;j3++)
 81            mat(i3,j3) = temp[k3++];
 82
 83    }
 84
 85
 86   template <class T>
 87   void
 88   WriteAsciiFile(const char *fileout, Matrix<T> &mat)
 89     {
 90      Int rows, cols;
 91
 92      mat.shape(rows,cols);
 93      ofstream oFile;
 94      oFile.precision(12);
 95      oFile.open(fileout, ios::out);
 96      for (Int i1=0;i1<rows;i1++) {
 97         for (Int j1=0;j1<cols;j1++) {
 98            oFile << mat(i1,j1) << "  ";
 99           }
100          oFile << endl;
101         }
102      }
```

10- The inclusion of <**fstream.h**> code is used to handle the reading from and writing to an external file.

11- The <**strstream.h**> is used to handle in-memory I/O i.e. to read a string that is already in memory (presumably to read it with a new format).

14-16- This is the function declaration. The first argument is prefixed with the keyword **const** which tells the compiler that this argument is NOT to be modified by the function code. Note that the type of data to be stored in the Matrix is STILL undefined i.e. it is declared to be of whatever type corresponds to the Matrix used in the calling sequence.

18- The dimension of the input string (each line of data) is stated as 1024. A bit overgenerous perhaps, but what the heck!

19- Two arrays, **buf** and **buf2** of type **char** are used as described below.

20- The initial size of the Block that will first contain the values to be read is stated here. Unlike previous examples it is NOT a **const** because we might well need to increase its value during the operation of the routine.

21- A Block is used used to contain the values initially. This is used, rather than a Matrix, because a Block can change size during the operation of a routine without losing its current content. (And since we have no idea how large a data set we need to read we must be able to expand.) A Matrix does NOT have that ability. Note that the dimension of a Block is stated within a FORTAN-like pair of parentheses rather than square brackets!

23-26- The ASCII file from which we want to read the data is opened. If the file cannot be opened for any reason the program will quit with an error message.

28- The dimensions of the Matrix will be **rows** and **cols** whose values will be determined by the input. Since we will only deal with rectangular arrays, the variable **saveCols** will be used to check that each row has the same number of columns.

29- The variable **havePoint** will keep track of how full the **temp** Block is becoming. If the Block is becoming too full, it is expanded (doubled) by invoking the function **resize** (in lines 39-41).

31- Reads an input line of up to **bufSize** characters. If there are no more lines go to the end of the **while** statement at line 74.

32-36- Check for a completly blank input line. If one is found, the variable **blankLine** will be left at the value 1 (TRUE) and it will be simply skipped by the test on line 38.

39- The test for the Block becoming too small makes provision for 10 more than the next row. If the line holds more values than that, then there are other problems to worry about!

44- Each line read (which may contain as many as 1024 characters) is presumed to contain a complete row.

45- The variable **ch** counts the characters in each new variable being read from the input string. The variable **startedNew** will be FALSE while we are bypassing an indefinite number of blanks before each value.

46- Starts the loop to scan an input line.

47-56- Check to see if we are at the end of a line or at a blank. In either case see if we have already picked up the characters for a new number. If so, add a terminating blank to the new number then call the internal IO routine to read it into a number in the **temp** array. Keep track of the number of points read in this row. In any case set the flag to bypass any more blanks on this line.

58-61- If we have encountered a character and we were skipping blanks set the **startedNew** flag to TRUE.

63- If we are reading a new number, place the latest character on a string.

75- Close the input file.

77- Now we know the size of the array so go ahead and make the Matrix be the appropriate size.

78-81- Transfer the data from the Block to the Matrix. NOTE the round parentheses surrounding the Matrix subscripts but the square brackets around the Block subscript! AAARGH!

86-88- This is declaring the second function, **WriteAsciiFile**.

92- Since we need to handle ANY shape, we need to recover the value of the **rows** and **cols** variables.

93- Declare an ASCII ouput file.

94- Set the precision with which we want to record output values. (The default is 6 digits.)

95- Open the output file.

96-98- Copy the Matrix to the output, placing one space between each value.

100- Terminate each output line with an EOL (end-of-line aka newline).

2.5 Sample Application

The previously described program was used to read from (and write to) an ASCII file. The input consisted of the following three matrices:

```
1.2345678901234  2.3456  3.4567 4.5678 5.6789 6.123456767  7.89789789
   1.1  2.2     3.     4      0      6.666        7.777
  0     1       2      3      4      5      6



1 2
3 4
5 6



123.45678911234567   456.78   890.12  234.56789123456789
1.23     456.78   8.90    2.34
12.345   678.90   123.56  78.9012
567.89   567.99   345.65  12.3435
```

The output produced by the program was:

```
The first Matrix is 3 by 7
The second Matrix is 3 by 2
The third Matrix is 4 by 4


Axis Lengths: [3, 7]  (NB: Matrix in Row/Column order)
[1.23456789012, 2.3456, 3.4567, 4.5678, 5.6789, 6.123456767, 7.89789789
 1.1, 2.2, 3, 4, 0, 6.666, 7.777
 0, 1, 2, 3, 4, 5, 6]

Axis Lengths: [3, 2]  (NB: Matrix in Row/Column order)
[1, 2
 3, 4
 5, 6]

Axis Lengths: [4, 4]  (NB: Matrix in Row/Column order)
[123.456787109, 456.779998779, 890.119995117, 234.567886353
 1.23000001907, 456.779998779, 8.89999961853, 2.33999991417
 12.345000267, 678.900024414, 123.559997559, 78.9011993408
```

```
 567.890014648, 567.989990234, 345.649993896, 12.3435001373]
```

The file written by the program looked like this (the first line is shown on two lines because of page-width limitations):

```
1.23456788063  2.34559988976  3.45670008659  4.56780004501  5.67889976501
  6.12345695496  7.89789772034
1.10000002384  2.20000004768  3  4  0  6.66599988937  7.77699995041
0  1  2  3  4  5  6
```

2.6 The Vector modules

The previous sections of this example demonstrated the use of **templates**. The next two, briefer, sections will demonstrate **polymorphism**, namely the ability to define a function with two or more sets of different argument types. In particular, we will add to the definition of **ReadAsciiFile** a module to read **Vectors** instead of **Matrices**. The following is the program which calls the new functions (The source code for this program is found in $AIPSROOT/code/trial/implement/test/tExample4.cc).

```
#include <aips/aips.h>
#include <aips/IO/AipsIO.h>
#include <aips/Arrays/ArrayIO.h>
#include <aips/Inputs/Input.h>
#include <aips/Mathematics/Math.h>
#include <aips/Arrays/Matrix.h>

main(int argc, char **argv)
{
 Input inputs(1);
 inputs.Version("$@w{Revision}$");
 inputs.Create("infile1","Input1","Float Input file name?","Infile");
 inputs.Create("infile2","Input2","Vector Input file name?","Infile");
 inputs.Create("outfile","Output","Vector Output file name?","Outfile");
 inputs.ReadArguments(argc, argv);

 const char *filein1 = inputs.GetString("infile1");
 const char *filein2 = inputs.GetString("infile2");
 const char *fileout = inputs.GetString("outfile");

 Int rows, cols;
 try {
    Matrix <Float>  mat1;
    Vector <Float>  vect1;

    readAsciiFile(filein1, mat1);    mat1.shape(rows, cols);
    cout << "The Matrix is " << rows << " by " << cols << endl;
    readAsciiFile(filein2, vect1);   vect1.shape(rows);
    cout << "The Vector is " << rows << " long" << endl;

```

```
31       writeAsciiFile(fileout, vect1);
32       cout << mat1 << vect1 << endl;
33    }
34
35    catch (AipsIOError x) {
36       cerr << "aipsioerror: error " << x.getMesg() << endl;
37       return 1;
38    }
39    catch (AipsError x) {
40       cerr << "aipserror: error " << x.getMesg() << endl;
41       return 1;
42    }
43    return 0;
44  }
```

6- We inadvertently omitted the reference to **Vectors** but got away with it! The reason is that the definition of **Vectors** is included in the definition of **Matrix**. It is NOT good practice to do this since modules may change.

13-14- We ask for the Vector (rather than Matrix) name. (A minor point.)

24- We will only demonstrate **Float** data type for **Vectors** but they will be defined for the full variety of data types.

28- We call for **Vector** data. Note that the function **shape** works for 1-dimensional arrays as well as for the 2-dimensional ones.

2.7 The expanded include modules

In this module we placed the boiler-plate that was discussed in the earlier example but we do not show it merely to save space.

```
30
31  #if !defined (AIPS_ASCIIFILEIO_H)
32  #define AIPS_ASCIIFILEIO_H
33
34  #include <aips/aips.h>
35  #include <aips/Matrix.h>
36  #include <aips/Vector.h>
37
38  template <class T>
39     void
40     ReadAsciiFile(const char *filein, Matrix<T> &mat);
41  template <class T>
42     void
43     WriteAsciiFile(const char *fileout, Matrix<T> &mat);
44
45  template <class T>
46     void
47     ReadAsciiFile(const char *fileout, Vector<T> &vect);
48  template <class T>
49     void
50     WriteAsciiFile(const char *fileout, Vector<T> &vect);
51
52  #endif
```

45-50- These declarations are similar to the ones for **Matrices** except that they take **Vectors** as their arguments. Note that we did NOT have to change the function name - hence polymorphism!

Chapter 3

Coding Fundamentals

3.1 Compilers

AIPS++ has adopted the gnu C++ compiler, g++, as the one true compiler for compiling C++ code. All code must compile and link with g++, vendor compilers, such as the SUN's native compiler, are useful for debugging, catching errors missed by g++, and more highly optimized code. Ralph Marson has written an AIPS++ Note 196 which describes how to handle templates with g++.

3.2 Include Files

All files should should ensure that `<aips/aips.h>` is included in their source file. It currently makes sure the following are properly defined:

1. `#pragmas` that are required for the TI preprocessor (presently used to implement templates (**??**) and run time type identification (**??**).
2. It will make some portability adjustments and ensure that fundamental types (**??**) are available.
3. Ensure that the namespace management (**??**) is turned on.
4. And any other miscellaneous things that are required to ensure that programmers can see a common environment.

Some other include files that are of general note are:

- aips/aips_exit.h If you are having name conflicts with another library you can include this file and refer to `aips++` classes by their full name.

- aips/Constants.h If you have a common mathematical constant it should be available in here. If it isn't, please think about adding it. [1] This file should also have astronomical and physical constants at some point. Don't hide what might be generally useful constants in your code — make them generally available!

- aips/Error.h If you intend to catch or throw any `aips++` exceptions (**??**) you should include this file.

- aips/Math.h You should include this file instead of `#include <math.h>` since unfortunately math.h doesn't always have a uniform set of functions. Also a few additional functions are in Math.h, in particular `min` and `max` functions.

The strategy for including files is very important. Every file which is `#include`'d in your .h file will normally [2] be opened and read in turn as your header file is scanned, *even if you have protected your header file* (**??**). This can greatly increase your compilation time. Instead of `#include` in your header files, you should instead forward declare all the classes you need, and then do the `# include` in your source (".C") file. The only times you might want to violate this rule of thumb is when the classes you are defining in your header file will always or almost always be used with another set of classes, then for convenience you might want to package them together this way. You might also be conservative and always include `aips/aips.h`.

So, instead of:

```
#include <aips/Foo.h>

class Bar
{
   Foo doit();
};
```

We would have:

```

class Foo; // Forward

class Bar
```

[1] Please consult with Mark Calabretta, mcalabre@atnf.csiro.au, first.

[2] Some systems which support *precompiled headers* can avoid this under some circumstances.

```
{
   Foo doit();
};
```

3.3 Fundamental Types

[This part should be moved into standards/guidelines.]

Rather than the fundamental C types, the following should be used in their stead:

- Int A signed integer guaranteed to be at least 32 bits long.
- uInt An unisgned integer guaranteed to be at least 32 bits long.
- Char A signed character.
- uChar An unsigned character.
- Float
- Double Float and Double are guaranteed to be different sizes. Normally they will correspond to "float" and "double", however occasionally they might be set to, e.g., "double" and "long double" if more precision is required.
- Bool An enumeration for Boolean results. True and False have been defined for it. Note that C++ will have a native Boolean type in the future.

Note that there are some circumstances, typically involving I/O, where the native types are required; however normally one of the above typedef's should be used.

3.4 Debugging Checks

[This section should change. I want to use the assertions etc. as in the Usenix 1992 C++ proceedings. If anybody wants to jump in and do this please let me know!]

There are certain kinds of checks that are too expensive to be made when a program is in routine production, but are useful to turn on while debugging or writing code. At the moment these are implemented by using the symbols `AIPS_DEBUG` and `aips_debug`.

If the preprocessor macro `AIPS_DEBUG` is defined during compilation, then aips_debug is a global Bool variable whose value is inititialized to True. If AIPS_DEBUG is not set, then aips_debug is just defined to be `(0)` so that optimizers should be able to remove this during dead-code optimizations. Thus code like:

```
if (aips_debug) {
  if (some_condition)
     throw(SomeError);  // or whatever
 }
```

should be cheap. If the debugging might happen very commonly in very tight loops you might instead want to enclose your checks in

```
#ifdef AIPS_DEBUG
     if (some_condition)
        throw(SomeError);  // or whatever
#endif
```

just to be absolutely sure there is no runtime cost.

It is useful to be able to check the *invariants* of a class. To do this a useful convention is to define `virtual Bool ok() const` in classes. This function should check that the class state is consistent. This would be particularly useful if used in conjunction with `aips_debug` to consistently check the invariants of a class in every member function.

Note: The `assert.h` macros should not be used other than in test programs because:

1. They will unconditionally exit whereas all real code should have the possibility of error recovery.
2. They have been observed not to work under some circumstances (because ANSI token catenation has not made it into all preprocessors in a consistent way).

3.5 Templates

Templates are an extremely important of C++. Ralph Marson has written an AIPS++ Note 196 which describes how to handle templates with g++.

Chapter 4

Programming Tips

4.1 Identifiers

Avoid identifiers which begin with one or two underscores, "_" or "__".

4.2 Classes

A virtual destructor will be defined for a base class when *ALL* of the following hold: [1]

- The destructor in a derived class is different from the base class destructor
- Derived class objects may be deleted via a pointer or reference to base class objects.

However, the designer may have reasons for making the destructor virtual when these do not hold.

Use copy constructors sparingly to avoid unecessary overhead.

Watch out for calls by reference and calls by value.

4.3 Type Conversions

Explicit type conversions ("*cast*"s) are to be avoided except in the following cases:

[1] Plum, Saks; C++ Programming Guidelines; Plum Hall, 1991. (p. 208)

- Conversion of a pointer to a base class to a pointer of a derived class within a type-safe container class
- Conversion of an external data representation (bit-stream, packed message, etc.) into an object. (Where possible use, RTTI and/or previously defined I/O methods).

"*cast*"s for reasons other than those mentioned should be clearly commented.

Avoid dependence on *implicit type conversions* – those that are performed on function arguments by the compiler.

4.4 Portability

4.4.1 Data Representation/Byte Ordering

Character constants should only contain one character because of differences in portability; a string literal may be used instead.

Avoid hard-coding offsets into structures. The alignment of structure members is implementation dependent. The pre-defined "`offsetof`" macro should be used to determine structure member offset. **??**

4.4.2 Execution Order

Avoid dependence on initialization order of global static objects.

Avoid dependence on order of operand evaluation.

Avoid Dependence on initialization order in constructors. **??**

4.5 Examples – Warnings

```
#include <stddef.h>

   ....

   struct MyStruct {
      int x;
      int y;
   };
```

```
   ...

   int theYOffset = offsetof(MyStruct, y);

   ...
```

```
class A {
public:
   A() : y(9), x(7+y) {};   // NO !!! y may not be initialized before x
private:
   int x;
   int y;
}; // A
```

Chapter 5

Exceptions

5.1 Introduction

The main intent of this paper is to provide a user level introduction to the `C++` exception mechanism with a basic introduction to the way it is used in `aips++`. This should allow developers to use the mechanism.

`C++` provides a standard method to handle exceptional conditions. When such conditions arise, control is transferred to a code block which can handle the exception, if one exists. If no handler exists which is capable of handling the exception, the program is aborted. This package handles cleaning up the call stack which may have been created between when the handler was *installed* and when the exception was thrown. In addition, any objects which were created on the stack during this period will be deleted.

This provides a mechanism for transferring control from the point where an exceptional condition occurs to the code to handle the exception without memory leaks. Some of the advantages of this approach are:

- The code to display the error message can be moved out to the appropriate level. Otherwise, the code to display an error message might have to be deep within an application in a place which could otherwise be independent of the graphical user interface. Without exceptions error messages must be carefully propagated out via return values until the appropriate level for their display is reached.

- The exception mechanism allows for an incremental graceful exit. The same exception can be thrown a number of times so that each layer of the program can have a chance to respond to the exception as appropriate. This can be important for maintaining the integrity of portions of the program, e.g. the user interface or database portions.

- Using exceptions a program can continue after an error even though the error may have been fatal at the point where it occurs. In a more traditional approach, this would require careful checks of return codes.

So while everything which can be accomplished with exceptions can be accomplished with returned error codes or other traditional approaches, the use of exceptions is cleaner.

5.2 Interface

This section describes the exception mechanism from the perspective of a "user" of the system. It discusses the `C++` interface, the mechanisms for automatic deletion of dynamically created objects, and the steps necessary to create new exception types whose objects can be *thrown*.

5.2.1 C++ Interface

The `C++` exceptions are based on the idea of `try` and `catch` blocks. The `try` block contains the code which may throw an exception, and the `catch` block, which must follow immediately after the `try` block, contains the handlers for the various exceptions which may be thrown in the `try` block. So in the simple case, one might have:

```
try {
  if (how_many == 1) throw(MajorError(1));
  if (how_many == 2) throw(MinorError(2));
  if (how_many == 3) throw(TypedError(3));
} catch (TypedError xx) {
  cout << "Caught TypedError" << endl;
} catch (MinorError xx) {
  cout << "Caught MinorError" << endl;
} catch (MajorError xx) {
  cout << "Caught MajorError" << endl;
}
```

In this example, depending on the variable `how_many` a different exception will be thrown, via `throw()`. The exception will then be caught by the first `catch` block that matches the exception. Thus, if `MinorError` was derived from `TypedError`, then the `TypedError` block would be executed before the `MinorError` block for a thrown `MinorError`.

It is also possible to have incremental recovery. A *caught* exception can be *rethrown* with `throw`. The following example illustrates this usage:

```
try {
  if (how_many == 1) throw(MajorError(1));
  if (how_many == 2) throw(MinorError(2));
  if (how_many == 3) throw(TypedError(3));
} catch (MinorError xx) {
  cout << "Caught MinorError" << endl;
} catch (MajorError xx) {
  cout << "Caught MajorError" << endl;
  throw;
} catch (TypedError xx) {
  cout << "Caught TypedError" << endl;
}
```

In this example, the `MajorError` catch block handles a `MajorError` when it occurs and then *rethrows* the exception so that it could be handled by other catch blocks. Thus, if `MajorError` is derived from `TypedError`, then the `TypedError` catch block would have the opportunity to handle its portion of the error after the `MajorError` catch block was finished.

These `try/catch` blocks can be nested to the level necessary as long as the introductory `try` is balanced with a closing brace. The following example contains a nested try block:

```
try {
  if (how_many == 0) throw(ExcpError(1));
  try {
    if (how_many == 1) throw(ExcpError(1));
    if (how_many == 2) throw(ExcpError(2));
    if (how_many == 3) throw(ExcpError(3));
  } catch (ExcpError xx) {
    throw(ExcpError(9));
  }
} catch (ExcpError xx) {
  cout << "Caught ExcpError" << endl;
}
```

5.2.2 Deleting Objects

Another important portion of the exception handling mechanism is that it deletes the objects created between the point where the `try/catch` blocks are entered and the point where the exception is thrown. These object are automatically cleaned up when an exception occurs. The destructors are called for automatically created objects, but not for dynamically created objects which are not deleted by other objects.

The programmer should use the templated `PtrHolder` class to hold pointers to dynamically created objects. Because the PrtHolder object is automatically created, its destructor is called in case of an exception. That destructor deletes the pointer it holds, thus deletes the dynamically created object.

```
try {
  // True below means it's an array, False (the default) would mean
  // a singleton object.
  PtrHolder<Int> iholder(new Int[10000], True);
  some_function_that_throws_exception();   // pointer is deleted
} catch (...) {
  cout << "Int array is nicely deleted" << endl;
}
```

Note that `...` is standard `C++` syntax and means any exception.

5.2.3 Creating Exception Types

One of the advantages of the `C++` exception mechanism is that one can design a class hierarchy of exceptions. This results in an a great amount of flexibility. The reason for this is that one can *catch* whole groups of exceptions with one catch block, and then handle them as appropriate. One can also catch an exception with a specific handler, and the *rethrow* the exception so that it can be caught with a more general handler.

The root of the exception hierarchy is the class `AipsError`. This class must be the base class for all new exception hierarchies. All of the exceptions which are part of the `aips++` kernel, `IndexError`, `AllocError`, etc., are derived from the `AipsError`. Thus, if one wished, all exceptions resulting from the `aips++` kernel could be caught by one catch block which catches `AipsError`s. Generally, it is a good practice to derive all exceptions for a given library from a library specific exception class, e.g. `TableError`, which is in turn derived from `AipsError`.

`AipsError` offers the const member function `getMesg()` which returns the message stored with the exception. It can be used as:

```
try {
```

```
  ...
} catch (AipsError& x) {
  cerr << "Unexpected exception: " << x.getMesg() << endl;
}
```

Chapter 6

Standards and Guidelines

This chapter was written by P. Shannon as AIPS++ Note 167

6.1 Introduction

These standards and guidelines have been chosen to promote a consistent style, clarity and organization in AIPS++ code. *Standards* are coding practices which everyone must follow; *guidelines* are recommended practices, and are identified below by (∗). Use of standards and guidelines will in general reduce the number of initial defects by about 25%, and make maintenance of code easier.

Effective C++ and *More Effective C++* by Scott Meyers is a good guide to C++ programming. Most of his guidelines apply to AIPS++.

6.2 Code Organization

6.2.1 Classes

Every class will have a header file,[1] an implementation file,[2] a test program [3] and (optionally) a demo program. Very closely related classes[4] may

[1] See *code/install/codedevl/template-class-h*

[2] See *code/install/codedevl/template-class-cc.* Neither totally inlined classes, nor abstract base classes may need an implementation file.

[3] See *code/install/codedevl/template-main-cc*

[4] A good example is a read-only class, and a read-write class derived from it.

sometimes be combined into single header and implementation files.[5]

All files shall follow the latest template file as a guide for the standard layout.

It is sometimes advisable to separate the implementation into different files. Examples are to separate templated and non-templated methods, and when a sub-set of the methods is often used. Such implementation files have in general names like *name2.cc* etc. See *code/aips/implement/Arrays/Array2.cc* for an example.

The test program(s) shall exercise every member function, every exception, and cover at least 90% of the lines of code in the implementation file.[6]

6.2.2 Include files

Whenever possible the header files should use forward declarations[7] in stead of include files. Include files should be reserved for the implementation files.

6.2.3 Data Types

Only the standard AIPS++ data types [8] shall be used, coded in the AIPS++ way.

6.2.4 Modules

Closely related classes should be organized into a module – which requires an appropriately named, separate directory.[9] Modules are assigned by the AIPS++ system manager. The module will have a "module header file" – whose primary purpose is to simplify things for the application programmer by providing sufficient help information for the general use of the classes in the module. Whenever possible, application programmers should not be required to learn the intricacies of a complicated module, nor to understand specific implementation classes within the module. In these cases they should only have to include a single header file into their application

[5]If you do this, make sure that the documentation sections – *synopsis, etymology, motivation*, etc. – are completed for *every* class contained in the file.

[6]Special language tools can measure coverage. See AIPS++ Note 170, "The AIPS++ Code Review Process" for suggestions on writing test programs.

[7]Explicitly or implicitly like in *iosfwd* or *Complexfwd*

[8]Char, uChar, Int, uInt, Float, Double, lDouble, Complex, DComplex, String. In special circumstances the use of Long and uLong is allowed.

[9]See *code/install/codedevl/template-module-h.* The Tables module, *code/aips/implement/Tables* is a good example to study.

program. In cases where the role of the module is more a conglomerate of related, but independent, classes, individual header files should be used.[10] ||

The module header file [11] will live in the module's parent directory: *code/aips/implement/Tables.h*, for example, is the module header file for, and describes the classes in *code/aips/implement/Tables/*.

6.2.5 Global Functions

It is sometimes appropriate to write global functions to act on class objects, rather than creating class member functions. As a general rule, you should do this if there is no class object state preserved from one function call to the next, and all of the information about the class object/s can be obtained from the public interface. However, for reasons of namespace management || it is often advised if there exists a strong binding between the class and the global function, to make the global function a static member of the class; or use the namespace keyword. There are two places where global functions may be declared and defined: as part of the *.cc* and *.h* files of the class they are most closely associated with, or in their own, separate *.cc* and *.h* files. Use these criteria to decide:

1. If there is a large conceptual distance between the functions and the class, then it is probably best to use separate files.

2. If there are a large number of related global functions, use separate files.

3. If the functions are templated, using separate files (at least separate implementation files) will reduce template instantiation dependencies. ||

6.2.6 Templates

||

Template instantiations are done explicitly within AIPS++. Make sure templates needed by your code are defined in the package *templates* file in the *_Repository directory*. Templates needed only in your test or demo programs should exist in your *test/templates* file. See the *System Manual* for details about template management.[12]

---

[10] As a rule, it is advisable for application programmers to include in their code the include files for the individual classes being used.

[11] See *code/install/codedevl/template-module-h*

[12] reident, unused, duplicates programs

6.2.7 File Size (∗)

Files should rarely – if ever – exceed 2000 lines: a 1000 line maximum will usually apply. Well-designed classes often have short header and, certainly, short implementation files, usually less than 500 lines including documentation.

6.2.8 Function Length (∗)

Functions should rarely exceed 100 lines in length; shorter, well-focused functions should dominate, and will usually be less than 50 lines.

6.3 Documentation and Naming

6.3.1 Documentation in Header Files

Header files shall contain clear and complete documentation for classes and modules. Templates (or "boilerplate") for these files are *code/install/codedevl/template-class-h* and *template-module-h*. That directory also has templates for other standard kinds of files.

6.3.2 Documentation in Implementation Files

Background, usage, and overall design is presented in class header files, and implementation files do not need to repeat them. But non-obvious sections of code do need to be accompanied by explanation, including references to manuals or texts as appropriate. "Obvious" is to be judged from the perspective of a competent programmer, generally knowledgeable – but not necessarily expert – in the program domain. The guiding principle is: do not force those who read your code to spend unnecessary time deciphering it. It will *always* be a net savings of time if you document what you have done, and why you did it.

6.3.3 Names for Classes, Functions and Variables

All class, function and variable names will reflect a balance between clarity and convenience, with clarity having priorty. Idiomatic abbreviations, acronyms and contractions are discouraged. Names for classes, structures, and enumerated types shall begin with an uppercase letter. Identifers which consist of more than one word (a recommended practice) shall use uppercase at the beginning of each new word ("*GoodClassName*"). Local variables, class data members, class member functions, and global functions names shall begin with lowercase letters ("*goodVarName, goodFunctionName*"). Class data members shall have names prepended by *its*, or postpended with _p ("*itsCount, blockPointer_p*").

Note: Glish function names follow a different policy: they shall use lower case letters exclusively.

6.3.4 Names for Files

The only restrictions here are commonsense, and the Unix library archive utility "*ar*". It is strongly suggested that the file name equals the name of the one or dominant class in the file. The above suggests that file names should be long enough to convey meaning to the reader, but not so long as to be a burden to type. This translates to: file names should be from about 8 to about 30 characters long.

"*ar*" requires that all object file names be unique in the first 14 characters.

Underscores are not allowed in file names.

Any file (a shell script, or a C++ program, for example) which can be invoked at the command line as an executable program, must be named with lower case chararacters only. (This policy follows the glish function naming policy of the previous section; both are designed to present the end user with only lower case commands to type, but note that underscores are prohibited in file names.)

Use the standard extensions (*.c, .cc, .h, .f, .g*).

6.3.5 Format for Dates

Programmers (and code reviewers) must enter dates when documenting code. These dates indicate, for example, when code has been reviewed, and when a list of "to do" tasks was last updated.[13] AIPS++ uses a single mandatory format for expressing dates. It has the virtue of being unambiguous, it is comprehensible to readers around the world, and it is also used by the project's version control system *RCS*. This format is *yyyy/mm/dd.* An example is *1995/02/27.*

[13] See, for instance, the documentation tags *reviewed* and *todo* in *template-class-h.*

6.4 Coding

6.4.1 Forward Declarations

#include only those header files your class absolutely requires. Use forward declarations when that will suffice.[14]

6.4.2 Protected Data Members (∗)

Avoid protected data members, using protected member functions for access to private data instead.

6.4.3 Access to Private Data

Do not return pointers or references to private data, except to accomodate Fortran or C libraries, or if efficiency absolutely requires it.[15]

6.4.4 Label All Virtual Functions Explicitly

Virtual functions in a derived class shall be explicitly labelled with the *virtual* keyword in the class declaration, even though this is not required by the language.

6.4.5 Document Loose Ends

Indicate unfinished or questionable code with the documentation extractor's tag <*todo*>.

6.4.6 Standard Class Member Functions

Provide definitions for all of the standard member functions which the compiler would (otherwise) generate automatically: default constructor[16], destructor[17], copy constructor, and the assignment operator. For any of these which you really wish to disallow, declare them private, and create minimal implementations (if any at all). For example:

---

[14] This may sometimes involve the use of special forwarding include files like iosfwd.

[15] Or declare the references and pointers const – but be aware of threading problems in that case

[16] Note that this is only created when no user constructor at all present

[17] always supply a virtual destructor in a base class

```
...
private:
   SomeClass () {;}   // disable the default constructor
...
```

6.4.7 The Order of Function Arguments

Functions shall be declared so that their arguments (parameters) appear in this order: output parameters first; input/output parameters next; input parameters last.[18] Functions which return only one value should usually use the function's return-value to return that value, rather than add an additional output parameter to the parameter list. If a function modifies its single argument *in place*, then that argument is best understood as an input/output argument.[19]

6.4.8 Formal Arguments and Default Parameters

Specify formal arguments and default parameters in the function declaration; use the same names in the function definition.[20] ||

6.4.9 One argument constructors ||

Constructors with one argument should be made explicit, or have an explanation why automatic conversion is wanted (as an example: *Unit(String)* is an obvious candidate for automatic conversion, to be able to write e.g. "km/s" rather than Unit("km/s")).

6.4.10 Return Types

Specify explicit return types for all functions. Return an appropriate integer value from *main*.

6.4.11 const Functions ||

All functions that do not change the state of the class should be declared *const*. If only the "hidden" state (like a cache) changes, the function should

[18]This order may seem unnatural, but it is a direct consequence of the C++ rule that default parameters appear *last* in the parameter list.

[19]But also consider *strcpy* from the standard *C* library, which both modifies an argument, and returns that same value as the function return value.

[20]In the case of a "placeholder" parameter that is not yet used in the definition, omit the name *in the definition* to stop warnings.

be declared *const*, but with a *mutable* cache variable. *mutable* should be used carefully.

6.4.12 Compound Statements

Multi-line compound statements should have braces. E.g.

```
if (condition) {
   statement1;
} else {
   statement2;
};
```

6.4.13 Template arguments

Minimize the number of template arguments by utilising the *typename* construct for user classes, and the *traits* methodology for fundamental types.[21]

6.4.14 Casting

Only C++ style casting is allowed. The use of *reinterpret_cast* is not allowed. Allowed are: *static_cast*, *const_cast* (but be aware of what you are doing) and *dynamic_cast*.

6.4.15 Macros

(*) Macros (especially *#if* type, should be limited to the *include guard*. Other cases should be checked with the QAG, who often can point out other solutions.

6.4.16 Exceptions

Exceptions should be reserved for truly exceptional circumstances, and not used to replace status flags or condition tests. Exceptions should be documented with the <thrown> tag (and maybe with the thrown declaration specifier).

6.4.17 Compiler Warnings

All code should compile without producing warnings from the compiler. Most compilers do not produce many spurious warnings any more.

[21] A good example can be found in the MeasConvert class

Chapter 7

Code Review Process

This chapter was written by P. Shannon as AIPS++ Note 170

7.1 Introduction

The AIPS++ project adopted a formal procedure for quality checking code in the late summer of 1994. The procedure requires that all new core[1] source code be submitted to the "code cop" (*email: aips2-cop@nrao.edu*) before it can become sanctioned AIPS++ code. Once received, the cop assigns a reviewer, who judges the code against the **AIPS++ Standards and Guidelines** (note 167). A dialog between the programmer and the reviewer may follow, eventually leading to approval of the code, and its formal inclusion into AIPS++.

In this note, I discuss the review procedure

- in general
- from the perspective of the programmer submitting new code
- from the perspective of the reviewer
- with special attention to test programs

7.2 General

There are eight documents, in addition to this one, which are essential to the code preparation and review process. Both programmer and reviewer

[1] *aips, dish, synthesis* and *vlbi* packages

should study them carefully, and base their work upon them.

- *code/install/codedevl/template-class-h:* how to construct a class header file.
- *code/aips/implement/Lattices/Slicer.h:* good example of a class header.
- *code/install/codedevl/template-module-h:* how to construct a module headers file.
- *code/aips/implement/Tables.h:* good example of a module header.
- *code/install/codedevl/template-global-functions-h:* how to construct a header file for global functions.
- *code/aips/implement/Arrays/ArrayIO.h:* good example of a global function header file.
- *code/doc/notes/167.latex:* Standards and Guidelines Checklist.
- *http://www.cv.nrao.edu/aips++/docs/html/cxx2html.html* An online guide to the documentation extractor, with good descriptions of the markup tags to be used in creating header files.

There are three main goals for the code review, all of which are equally important:

- Guarantee that the fundamental design of the class, as exposed in the public interface, is sensible and reasonably complete. We are more concerned about overall design than implementation details, because we assume that the programmer is competent; that the implementation may be too complicated to analyze in a reasonable amount of time; and that test programs and early usage will indicate errors and gross inefficiencies. The public interface, though, is everyone's concern, it is a good indicator of overall design, and it is painful to change once a class has been checked in.
- Make sure that the class header file is a sufficient guide to the class for client programmers. This is a very high standard, and demands a great deal of work from the programmer and the reviewer. "Sufficient" means that a programmer, previously unacquainted with the class, can read the header file and quickly reach a point where he or she can write code that uses the class. This standard of explanation is often met by

well-written programming books, and is rarely met by programmers in their own internal documentation. It is, nonetheless, the standard for all AIPS++ source code.

- Use the test program(s) to make sure that memory leaks are discovered and removed. See that code coverage is about 90% complete[2].

A great deal can be learned from reading through a class, and running test and demo programs. But no review is complete until other programmers use the new code. For this reason, we have a package called **trial**.[3] This package is automatically distributed by the AIPS++ code distribution system. If you ask the code cop to install your code in the trial directory, it will be available for everyone to examine and use. (All of these possible users will understand that *trial* code is subject to change with little or no notice.)

7.3 For the Programmer

- When you are within a week of submitting code, send the cop email briefly explaining the purpose of your class, and when it will arrive. This lead time will allow the cop to begin the search for a suitable reviewer.

- If your class is called, for example *"NewClass"*, then you need to submit the following files:

  1. *NewClass.h*: the class header
  2. *NewClass.cc*: the class implementation
  3. *tNewClass.cc*[4]: a test program – see section devoted to this topic below
  4. *dNewClass.cc* (optional): a short, heavily commented and well-focused program demonstrating how to use *NewClass*.

[2] Tools like "Purify" and "TestCenter" should be used if they are available.

[3] *(your-workspace)/code/trial* – at the same logical level as *aips, dish, vlbi, synthesis* and all of the site-specific packages

[4] If there is more than one test, exec, output or demo program, name subsequent files, for example *tNewClass1.cc, tNewClass2.cc* – the first program is implicitly number zero.

- Your code (header, source, test, and demo programs) should be combined into a standard unix "shar" file and emailed to *aips2-cop@nrao.edu*[5]. This shar file should preserve the directory structure needed to build the code once it is unshar'd.
- The cop will pass the shar file onto the reviewer, and tell you who the reviewer is.
- All subsequent email exchange between you and the reviewer should be CC'd to the cop.
- If you have test coverage and memory leak statistics (*strongly* recommended for all programmers who have access to the right tools) please send them to the cop in a separate email message.

7.4 For the Reviewer

Your job is a difficult one and requires a lot of time and effort. It may also require imagination and diplomacy. Imagination is needed because you must cultivate the perspective of the non-expert client programmer, intentionally forgetting things that you know, and scrutinizing the code (the class header in particular) to make sure that it explains everything the client programmer will need to know. You must also be able to take on the domain expert's point of view, and carefully examine the new class for the quality of its overall design. You may need considerable diplomacy to persuade a recalcitrant author that they need to make changes in their code and documentation.

Your responsibililites include these specific actions:

- Let the cop and the author know how long it will take to make the first review of the code, and how well you are sticking to that schedule as the time goes by.
- All email between you and the author which is related to the new class should be CC'd to the code cop.
- Be prepared for the cop to suggest changes beyond those you require.
- Base most of your evaluation on the header file for each class. A quick reading of the implementation file will allow you to judge the adequacy of its documentation, and detect any troubling features.

[5]If the class source code amounts to many thousands of lines of text, explain this to the cop – it may be possible for the reviewer to examine the code "in place", on your machine, or checked in to the *trial* directory.

7.5 Test Programs

The programmer must provide one or more programs that thorougly test the newly submitted code. These programs should be very simple to run, and should exercise all aspects of the new code.[6]

To indicate a successful test, it will often suffice for the test program to simply print"OK" to standard output, and return *0* as the value of the main program. In the test program fails, it should print a descriptive error message, and return some non-zero value to the environment. Some classes and global functions (especially those which employ numerical methods) can only be adequately tested by comparing a list of expected results against actual results, making allowance for tolerable roundoff differences[7].

The programmer may use two strategies here:

1. Create tMyClass.out, which contains the *expected* results, and tMyClass.exec, a shell script which runs tMyClass, redirects the actual results to a file, and compares them to the contents of tMyClass.out.

2. Alternatively, the tMyClass program *itself* creates the new results file, and runs the comparison, using a *system* call; when the comparison of results is complete, tMyClass prints "OK" and returns *0* to the environment if it is successful, just as the simpler test programs do.

[6] A handy way to construct a test program for a new class is to copy the class header file into a new file – which will become the test program – and edit the member function declarations into member function calls. This simple approach guarantees that all of the member functions are called. The next step would be to intentionally call all of the member functions in such a way that all the exceptions are thrown: if you trigger and catch all of the exceptions, then you have gone a long way towards exercising all of the code. "Domain" coverage should be attempted too: exercise the member functions by calling them with a range of inputs, paying particular attention to the boundaries of the domain.

[7] We hope that, in time, the AIPS++ project will find or develop a "diff" tool that is smart enough to compare numbers, and distinguish between allowable differences (due to word-length round-off) and unacceptable numerical error.

Chapter 8

Documentation

This chapter was witten by D. Schiebel as AIPS++ note 156

8.1 Introduction

The objective of this system is to provide a way for programmers to document source code in a way which allows documentation to be extracted and converted into a reference document. To accomplish this, it is necessary to provide a way for the programmer to specify information which supplements the regular comments. This "extra" information is know as markup. The markup provides a means of structuring the comments, and allows them to be formatted in ways more sophisticated than ASCII text.

This system allows both the source code and the documentation for the source to exist in the same file. This close proximity will increase the chances that the reference document will remain up to date. In addition, the goal is to have a parser which understands most of the C++ language. This allows much of the reference document to be generated automatically from the source code.

The first revision of this system used a language similar to the `roff` dialects. This language was abandoned in favor of SGML, Standard Generalized Markup Language. There are a few reasons for this:

- Since SGML is an ISO/ANSI standard many tools are likely to be available in the future to support it.
- Generalized markup is a better model for information reuse and flexibility. It allows the information it structures to be used by a variety

of applications.

- SGML provides a good basis for future expansion.
- Many successful applications already use SGML, and the work they and the various standardization groups have done can be leveraged to our benefit.

8.1.1 Generalized Markup

The motivation for generalized markup is a departure from that of traditional markup languages. With traditional markup languages, the goal is to add additional information to a document to allow it to be formatted for presentation. The markup is formatting information. With generalized markup, however, the markup is used convey the logical structure of the information. How this structure is formatted should not be the concern of the writer. The provider of the information should only be concerned with providing the important information with the appropriate logical structure. The actual formatting will be performed by systems which will use this information. The logical structure of the information is much more valuable than the way it should be formatted. If the information is structured correctly, a variety of post-processors should be able to use the information which is based on a given tag-set.

So for example a piece of procedural markup might look like:

```
.TH DF 1V "16 September 1989"
.SH NAME
df \- report free disk space on file systems
.SH SYNOPSIS
.B df
```

This is a section taken from the `df` Unix man page. Here, the *.TH* command sets up the header – the reference page is `DF`, the section is *1V*, and the last field is the date of the most recent change. The *.SH* command sets up a section heading with the given label, and the *.B* command displays text in bold. This is relatively typical of procedural markup. While all of the information necessary to present a nicely formatted man page is there, it is all superficial. The sections do not convey the content. The section command is just a section command, and does not convey the fact that the section is the synopsis. In a generalized markup language, SGML in this case, this section of the man page might be represented as:

```
<man ref=DF sect='1V' date="16 September 1989">
<name> df
<summary> report free disk space on file systems </summary>
<synopsis> <com>df</com>
```

Here the document is labeled as a man page and the <*man*> attributes [1] are the information that previously generated the header. The name is no longer the text of a section heading; it is labeled as a <*name*>. The synopsis is labeled as a synopsis, <*synopsis*>, instead of a section with the heading "SYNOPSIS". The "df" reference in the synopsis is a command, <*com*>, instead of a bold section of arbitrary text.

Generalized markup allows the important information to be processed unambiguously by many different systems. With generalized markup, a <*synopsis*> may be treated as a section for printed text, but for a hypertext system the synopsis may only be displayed if someone presses the synopsis button (however that is done). The information represents a synopsis instead of formatting details. The formatting decisions are left to the discretion of the system which will use the information.

Generalized markup is one tool which will allow information to be reused for a variety of purposes. Once freed from a particular formatting language, many applications can utilize the same information source for many purposes. The same logically structured information can be used to generate a printed manual, a hypertext system, a database, or practically anything else that can be done with information. All that is necessary is that the proper set of tags are defined and processors exist to manipulate information structured with the tags.

8.1.2 Standard Generalized Markup Language

SGML is in some sense a meta-language. It allows one to specify one of a family of generalized markup languages. SGML provides the language to specify the elements of the markup language and the rules for composition of these elements. This capability enables mechanical analysis of documents which utilize an SGML specified markup language. In addition, since SGML is an ANSI/ISO standard compatibility of SGML systems is guaranteed. In fact, SGML has been used for representing brail, music, hypermedia, information for tutoring systems, published books. SGML will be one of the tools which will help to make free interchange of information a reality.

---

[1] Single or double quotes can be used to distinguish literal values in SGML. If an attribute value is a single word, quotes are not required.

Document Type Definition

One of the most important sections of an SGML document is the section which defines the tag-set and composition rules which will be used in the document. This “grammar” is typically located in another file and is included at the top of the document in much the same way that LaTeXmacro packages would be included in LaTeXdocuments. This “grammar” section is called the *Document Type Definition*, **DTD**.

Elements The **DTD** specifies the elements, the tags which delimit elements, and the rules for composition of elements. An element is the smallest unit of concern for SGML. It represents a single concept or logical unit. So for example, a paragraph might be one element in the **DTD**, and one paragraph element might be delimited:

```
<p> This is my paragraph. </p>
```

This forms one instance of the paragraph element. The starting, *<p>*, and ending, *</p>*, tags delimit the element. The “p” in the tags is called a generic identifier, and it serves to distinguish paragraph elements from other sorts of elements.

Generic identifiers, **GI**s, can have attributes. These attributes describe important characteristics of the **GI**. The attributes are specified within the starting tag of the element. For example:

```
<category lib=aips sect="math">
```

Here the attributes are *lib* and *sect*. The value for *lib* is “aips” and the value for *sect* is “math”. Either single quotes, “ ’ ” or double quotes, “ ” ”, may be used to delimit literal attribute values. If the attribute value is one word, no quotes are necessary. The attribute name is typically limited to eight characters.

Entities In addition to tags, SGML also allow for the definition of entities. Entities provide a means for simple keyword expansion. This is useful not only for a shorthand notation, but also as a way of parameterizing a document to make future changes easy. Entity references begin with an ampersand and end with a semicolon. So for example, a reference entity for the less than symbol, “<”, might be *<*. All entities are typically defined in the **DTD**.

Entities are particularly useful to prevent text from being interpreted as markup. The characters which are of particular concern are “<” and “&”

because the introduce element references and entities references respectively. However, this should typically not be a problem because misinterpretation is only possible when these opening characters are followed directly by a non-space character. So "<*test*" could be interpreted as a tag but "<@: *test*" could not. Other characters which could cause problems are ">" and ";" because these are the characters which end element references and entity references respectively.

Body

Once the **DTD** has been specified, the user can then structure the information in the body of the SGML document. Once the tags are specified they can be used like the commands in other markup languages. If the elements are composed incorrectly, a parser will point out the problems. This is the level at which most SGML users will operate. This is the level at which developers entering comments will operate. For the developer entering comments, there will not be a "body" as such. The comments will each be a portion of the "body" of the SGML document. These pieces will be ordered and assembled into the body of the document by an extractor.

8.2 Document Generation

The conflicts and complications between the language in which the source code is expressed and the language in which the comments are expressed must somehow be resolved. The information contained in these two languages must be used to generate a reference document. These issues are discussed in this section along with the tags which are used to structure the comments.

8.2.1 Mixing Languages

The introduction of a structuring language for comments implies a mixing at some level of the language being documented and the language in which the documentation is expressed. This mixing can happen in one of three ways:

1. Provide a system which can integrate the languages at a level higher than the file level. In this way, the system would provide mechanism to edit code where the code would be brought up in the programmers favorite editor, but the documentation would be brought up in a SGML

editor. This gives the best of both worlds. A text editor is used for code, but a more powerful WYSIWYG editor can be used to construct the documentation.

2. Provide a system where the code is described as part of the documentation system. In this system, the code is extracted from the documentation for compilation purposes. This is the literate programming environment envisioned by Knuth.

3. Provide a system where the structure of the documentation is described inside of comments in the source code, and then the whole thing is massaged into a useful shape by a processor which turns the code/comment combination into a nice document.

Of these three alternatives I think the first is the cleanest, and it could be the most programmer friendly. However, it also involves the most work, especially if it is to be truly programmer friendly. It involves much more work than the latter two options. In addition, it would require a great deal of trial and error to arrive at a system which was flexible enough to be used for serious development. The second choice seems nice in theory, but I fear that this "literary" nature of development is quite foreign to many developers. The extra processing of turning documentation into code could also add too much extra time in the edit–compile–debug loop for it to be useful in practice. Thus, the third alternative was chosen. It has the advantage that the source is always kept in a state which can be compiled.

Reference documents will be assembled out of the documentation in the comments and the information extracted directly from the source code. This mixing of source code objects and comment objects in the same place requires a processor to mesh the two languages and rearrange the derived SGML objects into a coherent reference document.

8.2.2 Language and Comment Elements

The code units in a source file will be called "language elements". This includes for example a class definition, a function definition, a definition of one or more variables, a class declaration, a function declaration, etc. Basically, a language element is any complete statement in the grammar for the language. These are the pieces of the language which can have comment elements associated with them. Likewise, the comment elements are the comments which must *precede* the language element with which they are associated. So, for example, the following code fragment shows a function definition along with its associated comment.

```
// <div>
// <p> This function computes the area of a triangle and returns the
// result.
// </div>
float triangle::area() \{
  return (0.5 * base() * height());
\};
```

In this example, the code element is the definition of the function *triangle::area()*. The comment element preceding the definition is associated with the function. This pair will be converted into one piece of the documentation generated by the documentation extractor.

While comment elements can be provided for any C++ language elements or preprocessor elements, all comments will not be extracted. Initially, comments will only be extracted for functions and classes. Later elements like "*#define*"s, structs, enums, etc. will be added.

8.2.3 SGML Tags for Documentation

This section discusses the tags which were developed for describing the logical structure of elements of comments. These are described as an SGML specification in appendix A.

DTD Elements for Source Code Documentation

Title – ***title*** Many documentation elements contain an optional <*title*> as their first component. This allows for the specification of a label for that section of the overall document. So for example, one might have:

```
<div> <title> A simple section </title>
<p>The contents of the section.
</div>
```

The elements which can have titles are <*div*>, <*warn*>, <*note*>, <*verbatim*>, <*literal*>, <*code*>, <*enum*>, and <*list*>. These are explained below.

Text Separators – ***div, p*** There are two basic levels of division where "plain" text is concerned. These divisions are at the paragraph and section level.

To divide paragraphs the <*p*> tags should be used. These tags mark off a section of text indicating that it makes up a paragraph. The paragraph contains parsable character data, "#PCDATA". This means that the data in the paragraph can be fully processed by SGML for the expansion of entities and elements. Both starting and ending paragraph tags are optional in the cases where no ambiguity exists.

As a short-cut, paragraph beginnings and ends can be implied by blank lines. So for example, the following comment:

```
//
// This function computes the volume of a cylinder
//
// Its OK
//
float Cylinder::volume() {
  return C::pi * radius() * radius() * height();
}
```

would be converted into:

```
<div>
<p>
This function computes the volume of a cylinder
</p><p>
Its OK
</p>
</div>
```

The *<div>*s are added automatically by the documentation extractor (discussed below). and the paragraph elements, *<p>*, are implicit. Note that the leading and trailing blank lines are significant. They are the short-hand notation to start the initial paragraph and end the last paragraph.

The *<div>* tag is the generic tag for sections within a document. These sections are designed to be referentially transparent and readily relocatable. Typically comment elements will be divisions. Both the starting and ending tags are required for divisions. To obtain subsections, *<div>s* can be nested. For example,

```
<div>
  <div> <title> Nested Division </title>
     <p> This is the first simple paragraph.
     <p> This is the second simple paragraph.
   </div>
</div>
```

In this case, the closing paragraph tags can be left off because a paragraph cannot contain another paragraph. Typically the closing paragraph tag can be left off.

Warnings and Notations – ***warn*, *note*** These sections of text are for the display of information which should be important to the user. In the case of <*note*>, this should be used to bring a portion of text to the attention of the user. The <*warn*> text should tell the user information which can result in damage (of some sort) if the information is not known. For example,

```
<warn> ... deletion of this object will result in a memory leak under
       these circumstances.
<note> The new operator is private to prevent dynamic creation of
       objects of this class.
```

Literal Sections – ***verbatim*, *literal*, *code*** Literal sections allow one to create sections which undergo very little modification in the process of generating output. There is one subtle distinction between these sections.

In the case of <*verbatim*> and <*code*>, very little processing is done. The characters between the starting and ending tags for these elements are simply treated as a character string; no elements are processed and no entities are expanded. The one difference between these two is that <*code*> should be used to display sections of source code, otherwise <*verbatim*> should be used. These are to be used when **no** expansion of entities is required.

In the case of <*literal*>, minimal processing is also done on the characters in a <*literal*> section. In this case, however, the entities are expanded correctly.

Lists – ***enum*, *list*** All of the lists that are used have a common tag for the elements of the list, <*item*>. The contents of this tag can only be parsable character data and the list items do not nest. So for the most part, the closing <*item*> tag can typically be omitted.

There are two "generic" list choices. The first type of generic list is a numbered list, <*enum*>. The second generic list type is a list which distinguishes the elements of the list with bullets, <*list*>. Both of these lists can have a <*title*> before any of the elements of the list. So these might look like:

```
<list>
  <item> First Element
  <item> Second Element
</list>
<enum> <title> An empty list </title>
  <item> first element
</enum>
```

Hyperlinks *To be added later (probably based on* HTML *or HyTime) ...*

Descriptors There are several elements which are devoted to describing details about a given class or function. These descriptors list things like the exceptions thrown, the I/O devices accessed, etc.

Class Descriptors

The *<descriptor>* element list several important aspects about a given class. A given *<descriptor>* might look like:

```
<descriptor>
  <execution> <sequential>
  <bounded>
  <memory> <counted>
  <iterator>
  <cached>
</descriptor>
```

So, this description would describe a class that is purely sequential, has a maximum object size, is reference counted, has an iterator class, and has a builtin caching mechanism. If *<bounded>* were not specified, the assumption is that the object size is unbounded.

This *<descriptor>* list corresponds to the object descriptors required in the AIPS++ coding standards. The options are as follows:

- *<execution>* can contain *<sequential>*, *<guarded>*, *<concurrent>*, or *<multiple>*.
- If *<bounded>* is specified then the object size growth is bounded, otherwise it is unbounded.
- *<memory>* can contain *<counted>*, *<gc>*, or *<unmanaged>*

- If <*iterator*> is specified, then the object has an iterator, otherwise it does not.
- If <*persistent*> is specified, then the object is persistent, otherwise it is not.
- If <*cached*> is specified then the object has cached management, otherwise it does not.

Using this list of attributes a great deal of information about the class can be expressed succinctly.

Device I/O

Often, it is useful to have the ability to track functions which depend on particular files or I/O devices. This will often target which functions have direct access to the operating system. So the following would label a function which performed I/O on the files `/etc/hosts` and `/etc/motd` using OS specific routines, e.g. *"<stdio.h>"*, *"<iostream.h>"*:

```
<iodev> <level><os>
   <item> /etc/hosts
   <item> /etc/motd
</iodev>
```

It is also useful to be able to target functions which access files using a particular abstraction, e.g. **AipsIO**, because although these files do not depend directly on the operating system function calls they are tied to particular files. So for example a function that performs operations on the file `/usr/local/var/aipsppdb` using **AipsIO** would be labeled:

```
<iodev> <level> AipsIO
   <item> /usr/local/var/aipsppdb
</iodev>
```

In this way, the dependencies between the AIPS++ and the environment in which it operates can be tracked.

Exceptional Conditions

It is important to know the exceptions which can be thrown by a given class or function. This information gives the user of a function the list of exceptions which he may be required to catch. This information can be presented as follows:

```
<thrown>
  <item> AllocError
  <item> ArrayNDimError
</thrown>
```

These thrown exceptions should be specified at the member functions which throw the exceptions. However, the <*thrown*> descriptor could also be used in the comment element for a class. At some time in the future, the extractor will hopefully be able to generate call trees, and thus pick up a complete list of all of the exceptions thrown by a given function automatically. However, this ability depends on having a fully functional C++ parser available.

8.2.4 Extractor Commands

There are extractor commands which provide the user with control over how and if the comments are extracted. All of these commands have the form `//*` where the "//" is the beginning of a C++ comment, and the "*" is one or more consecutive non-space command characters.

Non-processable Comment

Often one will want to specify a comment which only belongs in the source file, and should not be extracted. This can be accomplished as follows:

```
//\#   This comment will not be extracted
//\#Neither will this one
```

Group Command

It is useful to provide a comment and specify that this comment applies to a group of language elements. This prevents the comment inconsistencies which can result from duplication of comments. This can be achieved as follows:

```
// Generally use of this should be shunned, except to use a FORTRAN
// routine or something similar. Because you can't know the state of
// the underlying implementation.
//+grp
T *getStorage(Bool \&deleteIt);
const T *getStorage(Bool \&deleteIt) const;
void putStorage(T *storage, Bool deleteAndCopy);
```

```
// <warn> An added problem with freeStorage is that it ...
void freeStorage(const T *storage, Bool deleteIt) const;
//-grp
```

The *"//+grp"* is the start group command. This command tells the extractor that the comment it just encountered should be used as the "base" comment for languages elements until a end group command, *//-grp*, is encountered. Any comments that are introduced within the group will be added to the end of the base comment for the appropriate language element(s). Group commands can be nested as long as the start and end group commands are balanced.

Literal Command

All comment elements, the comment preceding a language element, are typically a division, <*div*>. However, inside source files this results in a great deal of clutter in comments which would otherwise be quite readable. As a result, the extractor will attempt to correctly add the <*div*> tags. The user, however, can prevent the extractor for adding these by using the literal extractor command. The extractor will *not* try to spruce up documentation between the start literal command, *"//+lit"*, and the end literal command, *"//-lit"*. The start literal command should occur at the beginning of the comment block, and the end literal command should occur at the end. The literal command does not extend over multiple comment elements. The following example demonstrates the use of these commands.

```
//+lit
// <category lib=aips sect=io>
// <div> <title> Problems with this implementation </title>
//   The body of the section.
// </div>
//-lit
```

These commands prevent the extractor from adding any *user level* SGML commands. Typically all that the extractor would add is <*div*>s around a comment element. This allows the user to specify:

```
//
// This computes the area of the circle.
//
float circle::area() {
  return(C::pi * radius() * radius());}
```

instead of:

```
// <div>
// This computes the area of the circle.
// </div>
float circle::area() {
  return(C::pi * radius() * radius());}
```

8.3 Document Type Definition

This appendix has the document type definition for the comment markup language. It is written in SGML syntax. The *DTD* defines "elements" and "entities". The elements are the lowest level of granularity with which SGML deals. The entities function much like macro; entity references are replaced by the "body" of the entity.

There are two basic kinds of entities, general and parameter. The primary reason a distinction is drawn between these two types of entities is so that the general entity name space is unique from the parameter entity name space. Parameter entities are typically used by the designer of the *DTD*. As a result, parameter entities are typically used within other entity or element definitions. This leaves the person writing a particular document free to use any names when defining general entities.

Entity definitions begin with "*<!entity*". So the definition of a general entity might look like:

```
<!entity dquot sdata '\{\tt "\}' >
```

This defines a general entity which can be used to insert "double quotes" into a document. This is done by placing a reference to the entity in the document, i.e. "*&dquot;*". Here the "&" introduces the entity reference and the ";" ends the entity reference. This double quote entity is defined in terms of the formatting language into which the SGML document will be translated, in this case LaTeX. So these, entities must be changed for each SGML processor. For a plain ASCII system, the entity might look like:

```
<!entity dquot sdata '"' >
```

Parameter entities are defined and used in much the same way as general entities. Except that parameter entities are typically used by the *DTD* designer. The following is an example of a parameter entity definition:

```
<!entity \% bridgecontent "(p$|$warn$|$note$|$code$|$enum$|$list)" >
```

This defines a entity *bridgecontent*. The % indicates that this is a parameter entity instead of a general entity. This parameter entity is later used to define the elements that a *<bridge>* element can contain:

```
<!element bridge - o ((\%bridgecontent)*) >
```

The *%bridgecontent* entity reference is replace by the body of the entity, *(p|warn|note |code|enum|list)*. Element definitions are introduced with the" *<!element*", as shown above. The element definition has four fields, the name e.g. "bridge", a pair of fields which indicate if the opening tag and closing tag are optional or required — an "o" indicates that it is optional and an "-" indicates that it is required —, and a field which indicates the contents of the element. The syntax of the contents field is similar to lex:

- "*" – indicates zero or more occurrences
- "+" – indicates one or more occurrences
- "?" – indicates zero or one occurrence
- "," – indicates a sequence, e.g. (foo,bar) is *<foo>* followed by *<bar>*
- "|" – indicates an "or", e.g. (foo|bar) means one of *<foo>* or *<bar>* must occur
- "&" – indicates an "and", e.g. (foo&bar) means *<foo>* and *<bar>* must occur, but in any order
- "()" – parentheses provide grouping

So, the *<bridge>* example above declares a *<bridge>* element which can in turn contain zero or more *<p>*, *<warn>*, *<note>*, *<code>*, *<enum>*, or *<list>* elements. The opening tag is required, but the closing tag is optional. Elements are the most basic information units handled by SGML. They serve a purpose very similar to productions in a grammar.

In addition to elements and entities, there are several "special" characters which are useful to know:

- *&#TAB;* – Horizontal tab
- *&#RE;* – Record end
- *&#RS;* – Record start
- *&#RS;B* – Record start followed by one or more spaces and/or tabs

- *&#RS;&#RE;* – Empty record
- *&#RS;B&#RE;* – Record containing only one or more spaces and/or tabs
- *B&#RE;* – One or more trailing spaces and/or tabs
- *BB* – Two or more spaces and/or tabs.

These "invisible" characters are most useful when trying to define keyboard shortcuts which prevent document preparers from having to enter all of the necessary tags. For instance, inserting paragraph tags can be made easer as follows:

```
<!entity psplit '</p><p>' >
<!shortref pmap "\&\#RS;B\&\#RE;" psplit
                "\&\#RS;\&\#RE;" psplit >
```

This allows one to insert paragraph breaks by entering a blank line, i.e. a blank line or an empty line.

8.3.1 Parameter Entities

Sections

- <!entity % divcontent @w"(p|warn|note|verbatim|literal| code|enum|list|descriptor|iodev|thrown)">
- <!entity % bridgecontent "(p|warn|note|code|enum|list)" >

Paragraphs

- <!entity % pcontent "#PCDATA" >
- <!entity ptag '<p>' >
- <!entity psplit '</p><p>' >
- <!shortref pmap "&#RS;B&#RE;" psplit "&#RS;&#RE;" psplit >
- <!usemap pmap p>

8.3.2 Elements

Sections

- <!element div - - (title?, divbody) >
- <!element divbody o o ((%divcontent)* | divset) >
- <!element divset o o ((div,bridge?)*, div) >
- <!element bridge - o ((%bridgecontent)*) >

Paragraphs

- <!element p o o (%pcontent) * >

Warnings and Notations

- <!element warn - o (title?, #PCDATA) >
- <!element note - o (title?, #PCDATA) >

Literal Sections

- <!element verbatim - - (title?, (#CDATA)) >
- <!element literal - - (title?, (#RCDATA)) >
- <!element code - - (title?, (#CDATA)) >

Lists

- <!element enum - - (title?, (item*)) >
- <!element list - - (title?, (item*)) >
- <!element item - o ((p*)|div) >

Titles

- <!element title o o (#PCDATA) >

Class Categories

- <!element category - o (#EMPTY) >
- <!attlist category lib CDATA ”misc”>
- <!attlist category sect CDATA ”general”>

Descriptors

- <!element descriptor - - ((execution|bounded|memory|iterator|persistent)*) >
- <!element execution - o (sequential|guarded|concurrent|multiple) >
- <!element bounded - o EMPTY >
- <!element memory - o (counted|gc|unmanaged) >
- <!element iterator - o EMPTY >
- <!element persistent - o EMPTY >
- <!element cached - o EMPTY >
- <!element concurrent - o EMPTY >
- <!element memory - o EMPTY >
- <!element manage - o EMPTY >
- <!element iter - o EMPTY >
- <!element persist - o EMPTY >
- <!element counted - o EMPTY >
- <!element gc - o EMPTY >
- <!element unmanaged - o EMPTY >

Device I/O

- <!element iodev - - (level?,item*) >
- <!element level - o (os|(#PCDATA)) >
- <!element os - o EMPTY >

Exceptions

- <!element thrown - - (item*) >

8.3.3 General Entities (for LaTeX)

Required Because of Tags

- <!entity lt sdata ”{\lt}” >
- <!entity amp sdata ”\&” >

Typically Unnecessary

- <!entity dquot sdata ’{\tt ”}’ >
- <!entity num sdata ”\#” >
- <!entity gt sdata ”{\gt}” >
- <!entity percnt sdata ”\
- <!entity lpar sdata ”(” >
- <!entity rpar sdata ”)” >
- <!entity ast sdata ”\mch{\ast}” >
- <!entity plus sdata ”+” >
- <!entity comma sdata ”,” >
- <!entity hyphen sdata ”-” >
- <!entity colon sdata ”:” >
- <!entity semi sdata ”;” >
- <!entity equals sdata ”=” >
- <!entity commat sdata ”@” >
- <!entity lsqb sdata ”[” >
- <!entity rsqb sdata ”]” >
- <!entity circ sdata ”\verb+ˆ+” >

- <!entity lowbar sdata "_" >
- <!entity lcub sdata "\{" >
- <!entity rcub sdata "\}" >
- <!entity verbar sdata "{\verbar}" >
- <!entity tilde sdata "\verb+ +" >
- <!entity mdash sdata "{-{}-{}-{}}" >
- <!entity ndash sdata "{-{}-{}}" >
- <!entity hellip sdata "{\ldots}" >

8.4 Example Header File

```
\#if !defined(AIPS\_COUNTEDPTR\_H)
\#define AIPS\_COUNTEDPTR\_H
//\# Copyright (C) 1992,1996
//\# Associated Universities, Inc. Washington DC, USA.
//\#
//\# This program is free software; you can redistribute it and/or modify
//\# it under the terms of the GNU General Public License as published by
//\# the Free Software Foundation; either version 2 of the License, or
//\# (at your option) any later version.
//\#
//\# This program is distributed in the hope that it will be useful,
//\# but WITHOUT ANY WARRANTY; without even the implied warranty of
//\# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
//\# GNU General Public License for more details.
//\#
//\# You should have received a copy of the GNU General Public License
//\# along with this program; if not, write to the Free Software
//\# Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
//\#
//\# The AIPS++ consortium may be reached by email at aips2-request@nrao.edu.
//\# The postal address is: AIPS++ Consortium, c/o NRAO, 520 Edgemont Rd.,
//\# Charlottesville, Va. 22903-2475 USA.

template<class t> class CountedPtr;
```

```
template<class t> class CountedConstPtr;

//
// This is a dummy class which is not part of "aips/CountedPtr.h". I've
// included it for the purpose of demonstration. The things to note in
// the comments in general
// are:
//  <list>
//    <item> The opening blank comment line is significant. It is the
//           marker for the first paragraph.
//    <item> Blank lines between paragraphs are significant. They
//           separate paragraphs.
//    <item> The blank comment line at the end of the comment is
//           significant. It closes the last paragraph.
//  </list>
//
// In addition, although typically the divisions, "div", are not
// specified for a comment they are generated before and after the comment.
//
template<class t> class  generic\_useless\_template \{
private:
  t *val;
public:
  //
  // Generic ctor comment.
  //
  generic\_useless\_template(t *nv) : val(nv) \{\}
  //
  // This is one of the indirection operators. It is here to demonstrate
  // the use of the "+grp" extractor command.
  //+grp
  t \&operator*() \{ return *val;\}
  t *operator->() \{return val;\}
  //-grp

  //
  // Just another function.
  //
  void set(t *n) \{
    if (val) delete val;
```

```
    val = n;
  \}
  //
  // Generic dtor comment.
  //
  ~generic\_useless\_template() \{ delete val;\}
\};


//
// <category lib=aips sect=memory>
//
// This class stores the reference count and the pointer to the
// "real data".
//
// It is currently a template and is used such that
// "t" is the "true" type of the stored pointer. This means, however,
// that when it is used a template instantiation must be done for
// each type which "t" assumes. This makes debugging easier, but
// in the future all of these pointers could be declared with void
// type to avoid template instantiations.
//
template<class t> class PtrRep \{
friend class CountedPtr<t>;
friend class CountedConstPtr<t>;
private:
  t *val;
  unsigned int count;
protected:
  //
  // This constructor sets up the reference count to "1" and
  // initializes the pointer to the "real" data.
  //
  PtrRep(t *v) : val(v), count(1) \{\}
  //
  // <warn>
  // This destructor, if called, deletes the "true" data.
  //
  ~PtrRep() \{delete val;\}
\};
```

```
//
// <category lib=aips sect=memory>
//
// This class maintains a count of pointers which point to particular
// data, and it deletes the data only when no other counted pointers
// are pointing at it.
//
// This class is used as a pointer to constant data. As such, it
// only has the subset of the "CountedPtr" functions which are relevant
// for constant data.
//
template<class t> class CountedConstPtr \{
protected:
  PtrRep<t> *ref;
public:
  //
  // After the counted pointer is initialized the value should no
  // longer be manipulated by the raw pointer of type "t*".
  //
  CountedConstPtr(t *val) : ref(new PtrRep<t>(val)) \{\}
  //
  // After the counted pointer is initialized the value should no
  // longer be manipulated by the raw pointer of type "t*".
  //
  CountedConstPtr(const CountedConstPtr<t> \&val) : ref(val.ref) \{
    if (ref)
      (*ref).count++;
  \}
  //
  // The destructor deletes the data only when there are no other
  // counted pointers pointing at it.
  //
  ~CountedConstPtr() \{
    if (ref) \{
      (*ref).count--;
      if ((*ref).count == 0)
delete ref;
    \}
  \}
```

```
  //
  // The CountedConstPtr indirection operator simply returns a
  // reference to the value being protected.
  //
  // <note> The address of the reference returned should not be
  // stored for later use.
  //
  const t \&operator*() const \{
    return(*(*ref).val);
  \}
  //
  // This dereferencing operator behaves as expected; it returns the
  // pointer to the value being protected, and then its dereferencing
  // operator will be invoked as appropriate.
  //
  const t *operator->() const \{
    return ((*ref).val);
  \}
\};

//
// <category lib=aips sect=memory>
//
// This class maintains a count of pointers which point to particular
// data, and it deletes the data only when no other counted pointers
// are pointing at it.
//
// This class is used as a pointer to non-constant data, and it has
// all of the possible manipulation functions.
//
// <note> It is possible that use the <code>replace</code> function
// can change the data to which a <code>CountedConstPtr</code> points.
//
template<class t> class CountedPtr : public CountedConstPtr<t> \{
public:
  //
  // Simple function which leaves the work to the parent constructor.
  //
  CountedPtr(t *val) : CountedConstPtr<t>(val) \{\}
  //
```

```
  // Simple function which leaves the work to the parent constructor.
  //
  CountedPtr(const CountedPtr<t> \&val) : CountedConstPtr<t>(val) \{\}
  //
  // This assignment operator allows CountedPtrs to be freely assigned to
  // each other.
  //
  CountedPtr<t> \&operator=(CountedPtr<t> \&val) \{
    if (ref) \{
      (*ref).count--;
      if ((*ref).count == 0)
delete ref;
    \}
    ref = val.ref;
    if (ref)
      (*ref).count++;
    return *this;
  \}
  //
  // This function changes the value for this CountedPtr and all of
  // the other CountedPtrs which pointed to this same value.
  //
  // <note> This can change the value to which a CountedConstPtr points.
  //
  void replace(t *v) \{
    if (ref) \{
      if ((*ref).val)
delete (*ref).val;
      (*ref).val = v;
    \}
  \}
  //
  // This assignment operator allows the object to which the current
  // CountedPtr points to be changes.
  //
  CountedPtr<t> \&operator=(t *v) \{
    if (ref) \{
      (*ref).count--;
      if ((*ref).count == 0)
delete ref;
```

```
    \}
    ref = new PtrRep<t>(v);
    return *this;
  \}
  //
  // The CountedConstPtr indirection operator simply returns a
  // reference to the value being protected.
  //
  // <note> The address of the reference returned should not be
  // stored for later use.
  //
  t \&operator*() \{
    return(*(*ref).val);
  \}
  //
  // This dereferencing operator behaves as expected; it returns the
  // pointer to the value being protected, and then its dereferencing
  // operator will be invoked as appropriate.
  //
  t *operator->() \{
    return ((*ref).val);
  \}
\};

\#endif
```

Chapter 9

The Design of AIPS++

This chapter was written by B.E. Glendenning as AIPS++ Implementation Memo 111

9.1 Introduction

AIPS++ (Astronomical Information Processing System) is a project that is being developed by seven radio astronomical observatories to provide a new generation software package for the reduction and analysis of astronomical data. General information on the goals of the project, and reasons for its creation are described in [?].

AIPS++ is following in the footstep of its very successful (and still active) predecessor, AIPS (Astronomical Image Processing System, also known as Classic AIPS). AIPS is described in [?].

A persistent, and correct, criticism of the AIPS++ project is that it has inadequately expressed its design in written form. This document, along with companion documents which are also under preparation, is intended to address this criticism. To be useful to the project in a continuing fashion, these documents need to be maintained — they must not merely be a snapshot. It thus behooves both AIPS++ management and readers of this document to be vigilant in noting when this document, or its evolutionary successors, needs updating.

The goal of this document, then, is to summarize the design of the major subsystems of AIPS++ software. Where more detailed companion documents exist, the discussion in this document might be brief. In particular, the discussion of interferometric and single dish calibration and imaging is very abbreviated here.

The presented designs generally omit the many necessary complications of real implementations. Also not included are descriptions of important utility classes which are used in the system such as strings and associative arrays (also known as "maps" or "dictionaries"). Also not generally described are mid-level classes used to implement the design, such as classes for operations like gridding. Generally, the class reference documentation should be consulted for this information.

This document is presently one person's view of the work of many people. This document should evolve into a more collaborative document with time. Until then, perhaps the uniform view of a single individual will help make up for lapses in perspective.

The opinions and ideas of readers of this document are actively solicited, and should be sent either directly to the author or to an appropriate email reflector.

This document is informally written in English text, rather than using a more formal methodology. The diagrams are composed with the notation described in [**?**]. While it would be useful to read that book, it might suffice to note that classes are in boxes, and that inheritance relationships have triangles between the base and derived classes with the point of the triangle pointing to the base class. Other connections denote "uses" or "has" relationships. Very important member functions or instance variables might be shown in the class box, however the figures are mostly intended to convey inter-class relationships.

The reader who wants to trace the evolution of ideas in presented in this paper should consult [**?**].

9.2 Guiding Principles

9.2.1 Goals

The AIPS++ project has a mission statement ([**?**]). Somewhat simplified, the portions of it which have design ramifications are:

- AIPS++ shall provide to users a useful software system for calibrating, editing, image formation, and image analysis of (primarily) radio astronomical data. It will allow both interactive manipulation and programming as well as batch use.
- AIPS++ will provide support to programmers through useful application class libraries.

- AIPS++ will be easily programmable, maintainable, and extensible.
- AIPS++ will primarily be written in C++, but will offer programming interfaces to other scientific programming languages (*i.e.*, FORTRAN and C).
- AIPS++ shall provide for multiple user interfaces, including GUI's.
- AIPS++ shall be portable.
- AIPS++ shall be usable in a telescope operations environment.

It is probably worthwhile quoting verbatim the final point of the mission statement:

"15. In summary, AIPS++ shall be programmed to be flexible and versatile to use, to modify, and to enhance. Researchers other than the main support staff shall be invited and encouraged to add applications and functionality."

Some other issues related to the implementation of these goals by the AIPS++ project are described in [**?**].

9.2.2 Object-oriented (OO) programming

A fundamental decision was made to use object oriented concepts in designing and implementing AIPS++. Largely because of the requirement for portability, C++ was chosen as the implementation language. While C++ implementations presently tend to be immature and painful to use, on balance it has been a suitable choice.

Object oriented design and implementation offers some major benefits. Probably the primary advantage is that of *encapsulation*. Encapsulation imposes the discipline that data can only be modified through a well defined and consistent *interface*. The unit of encapsulation in C++ is the *Class*, which is (more or less) the same thing as a type. With encapsulation, if only the implementation of functionality needs to be changed, but not its interface, then no source code changes outside the class need be considered (*i.e.*, changes in implementation are localized).

In a sense, adding classes to an OO language can be considered to be tuning the language to the problem domain (especially in a language like C++ which has syntactic sugar such as operator overloading).

Another advantage of OO is *inheritance*: the ability to create a new *derived* class by adding on to an existing class.

Much more important than inheritance is *polymorphism*, which allows classes with a sufficiently similar interface to substitute for one another. This allows, for example, a new kind of clean deconvolution to be introduced without having to change any client code that needs such deconvolution. That is, not only does a class interface hide its own implementation, it can also hide the details of exactly what class is being used, allowing the class in question to be substituted without causing changes in the clients.

In C++, polymorphism is achieved through inheritance (derived classes may substitute for base classes) and templates.[1]

The above short summary cannot hope to do justice to the subject. For more details on OO design and programming in general see [**?**, **?**, **?**]. For an introduction to C++ by the creator of the language, see [**?**]. For an excellent description of the C++ idioms necessary to build a complex, real-world system, see [**?**].

Reusable software libraries

The gems of any software project are its application-domain algorithms and unique support structures. It is vital that these be reusable, not constantly reinvented.

The primary consequence of this is that application-domain expertise should be available in libraries, not "locked up" in an executable. Functionality which is only available through an executable is only able to serve those needs the application programmer was able to foresee.

Applications should thus be fairly simple. In some sense they are merely a convenient packaging of some aspects of a library. At the very least, functionality should at least be put into application specific classes and functions which can later be migrated to a library.

9.2.3 Multi-level programming

Users and programmers will want to approach programming in AIPS++ in a variety of ways:

- Users who want to perform calculations on their data in a fashion that is not supported by a canned application, or by a simple processing sequence of canned applications.

[1] Although it is often not thought of as polymorphism, template polymorphism is in many respects more flexible than inheritance polymorphism, one merely requires that an interface have certain features. The disadvantages are code bloat and some loss of semantics enforcement.

This type of computing may be performed via a command line interpreter (CLI). For users with unusual ideas or needs, it lets them perform ad-hoc computations or try out new algorithms without having to enlist a "pro." While much can be done merely with a language that allows whole-array computations, to be most useful the application-domain functionality in libraries must also be available to the CLI programmer. In fact, the CLI can be thought of as the most flexible interface to a library.

- AIPS++ application developers.

  If no significant classes or functions need to be written, this programmer should consider programming at the CLI level instead.

  When AIPS++ is more mature, this programmer will typically *use* AIPS++ classes, but will not necessarily write any, or at least the classes written will be fairly simple. This programmer might well write significant functions (algorithms) however.

- Class developer.

  At the next level down is the class developer. This programmer might need to do something fairly straightforward, like introduce a new CLEAN variant (straightforward once the algorithm is thought of!), or more involved like implementing a set of classes to encapsulate a calibration scheme.

  Below this level lie the programmers who provide the support classes (*e.g.*, arrays and tables) that the whole system is built upon. These programmers have a difficult job since all other programmers leverage off of their efforts.

- FORTRAN or C application writer.

  See the next section (9.2.4).

- Library users

  In the future there may be people who want to use the services in our libraries, but are otherwise uninterested in AIPS++; for example the FITS classes. While such uses aren't in our core constituency, this reuse will allow them to concentrate on their application domain; this also advances the cause of Science.

- Foreign packages

> Non-AIPS++ packages may wish to use parts of AIPS++, or in turn be used from AIPS++. While many levels of operability might be possible, the most promising would appear to be:
>
> 1. To interoperate at the source code level through common abstract base classes. (The derived classes might be very package specific).
> 2. To interoperate at the executable level by following common conventions for the interchange of IPC messages.
>
> To be meaningful[2] a negotiation between different software packages would be required. While it seems to this author that such discussions might be worthwhile, the arguments are political, not technical.

Parallel to the above programming levels ("how much stuff is done by each line of code") are levels that correspond to how much of an underlying model (*e.g.*, a calibration and imaging model) the programmer has to accept.

Since there are certain to be disagreements about such models, programmers who are too impatient to learn them, and new instruments or disciplines that do not fit them; it is important that such programmers be accommodated.

To accommodate such programmers, the actual computations they might need must be available at a mid-level. For example, rather than having CLEAN embedded solely in a high-level "Measurement model," the actual computational kernel should be available in mid-level classes that work on arrays and/or images. Irrespective of other concerns, such layering is good software practice.

The X-Windows system has a similar problem and solution. The core functionality is encapsulated in policy-free libraries, and the higher level "models" are encapsulated in additional libraries (Motif) and entities (*e.g.*, window managers).

9.2.4 Access to data

A decision which has caused some controversy is that all[3] AIPS++ data should reside in a common data structure, the AIPS++ Table (described in section 9.4). This decision allows several important capabilities:

- It allows FORTRAN and C programmers to work inside of AIPS++.

[2] Every package is happy to interoperate with packages which adopt *its* conventions.

[3] Or almost all; there are bound to be some things which reside in flat text files, *e.g.*, CLI initialization files

Given the "impedance mismatch" between C++ and C or (particularly) FORTRAN, it is unrealistic to assume that programmers in those latter languages will have full access to the functionality developed in C++. However, if the FORTRAN programmer at least has access to all the data, he can perform any desired computations.

- Back door access for users.

  While computations should normally take place through well defined classes, occasionally an arbitrary calculation needs to be performed or arbitrary values need to be inserted. If all data ultimately resides in tables, such operations can always be undertaken. An example of this would be modifying a calibration value through a GUI table editor.

- Common tools.

  Creating class-specific tools to sort, display, plot, print, ..., can be quite expensive. With all data in tables, general table tools can be used in the cases where tools tuned to the specific application or classes have not been developed. An analogy can be drawn to Unix which has many tools to manipulate a common data structure (text streams). These tools can often be applied to data which represent higher-level abstractions.

It is certainly true that the user can get into trouble by arbitrarily modifying tables (since software might not understand what to do with those tables if sufficiently changed). However, the alternative is worse; after all the data does *belong* to the user, and sometimes scientific exploration will require operations that no software system can anticipate.

9.2.5 Other goals

While the following run the danger of being non-controversial statements of motherhood and apple pie, perhaps they will be of some use written down:

Flexible It should be possible to use the classes to build unanticipated applications. Ideally this should be possible both from compiled languages and an interactive command line interpreter. Flexibility implies generality at some level, *e.g.*, tools should be N-dimensional when possible.

Efficient Our production codes must be about as efficient as the ones they are replacing. It is acceptable to recode critical sections with less flexible functions if necessary to achieve this. This most critically constrains the data system.

Portable There are several types of portability that are important. The software must be portable to various computers and architectures. This implies that the use of host features that can't be emulated on other systems should be avoided for the core system. Until there is a C++ standard, portability between compilers needs to be addressed.[4] Data needs to be portable at run time on different host architectures, which we solve by storing that data in a canonical format. Data also needs to be portable to other software systems, including those in the future, which means storing offline data in an archival format, *i.e.*, FITS.

Implementable We have an ambitious project with relatively modest resources; at times this will imply that we must implement a subset of the functionality we would actually like.

9.3 Astronomical Calibration and Imaging

This section describes the calibration and imaging "model" that the AIPS++ project has adopted. This section is only intended to be a relatively brief introduction to the general approach, describing the top level classes. If more detail is required than is available in this overview, see [**?**] for radio interferometry, and [**?**] for single dish.

9.3.1 The Green Bank approach

Radio telescopes are very inhomogeneous in their characteristics, and hence in the data that they produce. While the differences between a radio interferometer[5] and a single dish are the most striking (the fundamental data from an interferometer is from the Fourier (pupil) plane, from a single dish, the image plane), every instrument has different characteristics that are reflected in the underlying data "form" and values. For example, some instruments observe linear polarizations, others circular, some interferometers observe all baselines in lock-step, others can observe at baseline-dependent rates. More important than the "form" of the data, the operations which

[4] It would be ideal if the GNU C++ compiler, g++, was sufficiently reliable. Unfortunately that is not yet the case.

[5] More precisely, an earth rotation aperture synthesis array.

can be performed on it vary widely: calibration of a connected-element interferometer at low frequency is very different than calibration of a VLBI experiment at high frequency — the connected element interferometer can probably count on phase stability, but it may also "see" that the sky is not flat and need to may use more sophisticated imaging algorithms.

The situation for single dish telescopes is no simpler. Perhaps because a detector for single dish is only needed on one antenna, not on many, instrumentation innovation on these telescopes has been high; for example, multi-beam receivers are becoming common. Moreover, single-dish observers usually observe more interactively than interferometrists have in the past, interactively reducing their data and steering their ongoing observations.

Next generation interferometry instruments such as the Millimeter Array (MMA) will combine many of the above characteristics.

The challenge for a software system that wants to be able to deal with more than one of the above telescopes, let alone all of them, is daunting. It must be capable of exploiting the unique features any particular telescope might have, while providing common tools that operate as widely as possible.

The approach described here was invented at a workshop in Green Bank, West Virginia, in early 1992. It was originally described in [**?**].

The approach taken is to (abstractly) model the processes involved in calibrating and imaging data. It should be pointed out that this approach has been somewhat controversial; for example see [**?**]. An objection to the model as outlined in the original paper is that it was insufficiently data-centric and overly complicated. The present version of the model has evolved to at least partially meet these objections.

Data from a telescope is collected into a *MeasurementSet*. The telescope sampling process is described by a *MeasurementModel*, and the effects (primarily instrumental and atmospheric) which have corrupted the data are described by *TelescopeComponent* objects within a *TelescopeModel*. The goal, of course, is normally[6] to produce the best representation of the sky as possible (usually in an *Image*, or possibly in a *SourceModel*) by correcting to the extent possible for the corruption and sampling processes.

Cornwell [**?**, **?**] developed an elegant mathematical formulation (informally called the "A Matrix Formalism") of the Green Bank approach to calibration and imaging. While this approach has not yet been developed further by the project, the approach is very interesting. In some sense, the

[6]However, one can imagine that for, say, geodesy, the values in a telescope component that describe the displacement of a telescope from a normative position might be the desired result.

(large) matrices in Cornwell's formulation have been replaced here by tables, and matrix operators have been replaced by operations on tables. (A table is of course one way to represent a very large, sparse matrix.)

MeasurementSet and views of data

The *MeasurementSet* is the lowest level interface to data. The data might be either uncalibrated or calibrated (or partially calibrated). The MeasurementSet is intimately involved with AIPS++ Table objects, which are described in section 9.4. This close relationship is not accidental; Tables were of course designed explicitly to handle the processing of astronomical data.

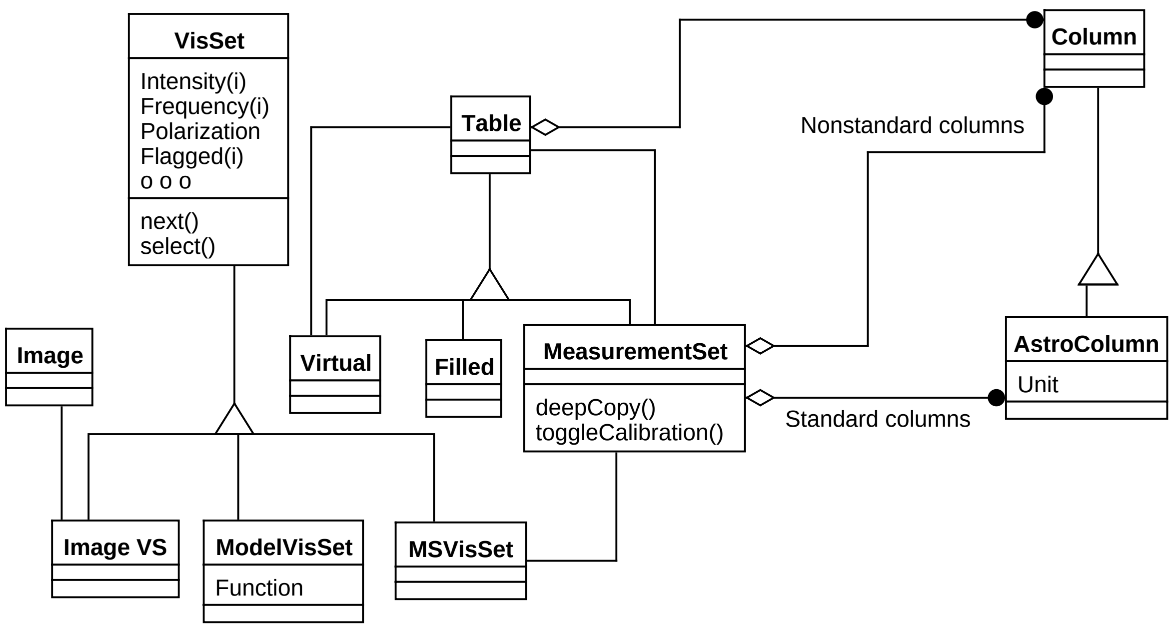
It is intended that the MeasurementSet will hold not only the usual astronomical data, but possibly also data that is normally considered telescope monitor and control data (of course this isn't required, and the user might filter it out). This is to allow the user to make judgments about his data using all available information.

A MeasurementSet *IsA* table with certain conventions. In particular, there are a set of standard columns (*e.g.,* effective integration time) which might be in a MeasurementSet. While a particular MeasurementSet need not have any particular columns, if it does have one of the standard columns that column must have its usual meaning. There are also a set of conventions that the standard columns obey, for example to associate units with the values in the column. In addition to the standard columns, the MeasurementSet might have other columns to represent some very telescope-dependent data, or even a column that a user has attached to hold some additional information.

It is important to note that the MeasurementSet is a *single* table, not a collection of tables. Other software systems have tended to separate things which vary at different rates (per observation, per source, per integration *etc.*) into different tables to prevent data bloat (caused by repeating constant values for several or many rows). Besides the navigational difficulties in finding the value that corresponds to the current row in another table, a more fundamental problem is that values can vary at different rates for different instruments (and observing modes), and hence finding a particular value might require a moderately complicated runtime lookup.

Instead, we have chosen to use the so-called "big table" view of data. This approach has all data appearing in a single table. This makes the navigation problem trivial — all rows contain all values — however it places a burden on the table system to provide a mechanism for avoiding unaccept-

Module: imcal Sheet: 1: MS Panel: (0,0) Time: Sun Nov 27 17:57:39 1994



A SpectrumSet class like VisSet is not illustrated to save space.

Figure 9.1: MeasurementSet and other data view classes.

able data bloat. See the section on table storage management (section 9.4.2) for details on how this is accomplished.

One often wants to query a MeasurementSet to discover what, for example, all the possible pointing centers are. This operation is less convenient with a "big-table" view when the underlying values truly vary infrequently. This may require optimizations or new functions in the Table system, or a caching convention in the MeasurementSet (for example, by using per=column keywords).

A MeasurementSet row is an important level of granularity. In general, a row of a table will contain:

1. A vector of data values (spectrum or time series).
2. Values to uniquely identify the data (time, spectral window, polarization, feed, *etc*). That is, these values uniquely identify a particular row.
3. Values related to the observation (integration time, object, sky position *etc.*
4. Data quality measures (flags, and noise estimates).
5. Tabs supplied by the user and telescope on line system ("calibrator").
6. Miscellaneous pieces of telescope-specific information.

For a more detailed description of columns in a MeasurementSet, see [**?**].

The original concept was that each row would contain a truly atomic piece of information. This is conceptually attractive — one does not have to worry about some members of, say, a visibility spectrum being flagged but not others. However, since this would place extreme demands on the table data system, we have decided to make the atomic unit the vector of data. Note, however, that we have, for example, separated separate polarizations onto separate rows (logically; the underlying storage manager might actually store visibility data as a brick when the number of channels, baselines, and polarizations are the same for all times).

When data from several different observations are to be combined, the data would appear in a single MeasurementSet, with a column to make unambiguous which instrument a particular row comes from. The columns in the joint MeasurementSet would be those which are common to all the MeasurementSets.[7]

[7]While such concatenations will be physical in the first instance, a Table class which

The MeasurementSet is a view of data. The data might consist of raw values read from disk. However, it might also consist of data which is being calibrated "on-the-fly," or indeed the data is being entirely simulated.[8] This is accomplished by both deriving MeasurementSet from Table for its interface, and by having it reference a table (which might itself be a virtual table) for its values. The mechanisms to enable this are further described in sections 9.3.1 and 9.4.1 below.

While the MeasurementSet is complete, it can be tedious to use, or at least overkill, for some applications. Thus we define abstract classes which define the interface we want for our application, and attach them to the data source — normally a MeasurementSet (*MeasurementSetVisSet*) — but possibly also a function (*ModelVisSet* or image (*ImageVisSet*). These classes are known as data views in that they don't contain any data on their own, they refer to data which is elsewhere. While there might be a wide variety of such data views, the two which have been identified to date are the *SpectrumSet* and the *VisSet*; the former consisting of image-plane spectra, and the latter of visibility spectra.[9] These data views would be constructed with various selection criteria, for example, to only return Stokes I.

While it is possible of course to construct many different views of data, the SpectrumSet and VisSet are chosen to be rich enough to implement an imaging algorithm after the data is calibrated. So, for example, the VisSet would contain at least UVW, Time, Antenna1, Antenna2, Flags and Weights, and Complex Visibilities. Also, the data view might have some convenience functions build in, for example, to shift the visibilities before returning them to the user.

Image, Source Model, and PSF

The usual end result of as calibration and imaging process is an estimate of a sky brightness distribution in the form of a *SourceModel* and/or *Image* which the user then analyzes to obtain his science.

An Image is a regularly sampled set of intensities with coordinate information. It is described in more detail in section 9.5 below. The SourceModel is a more general description of the sky brightness distribution than an Im-

---

presents a single table interface to multiple underlying tables will be available at some point.

[8] Or partially simulated; a pure east-west array might have a zero column of W's which are returned from a (trivial) virtual column.

[9] It seems likely that these classes will at least have a common base class, and possibly that VisSet will be derived from SpectrumSet.

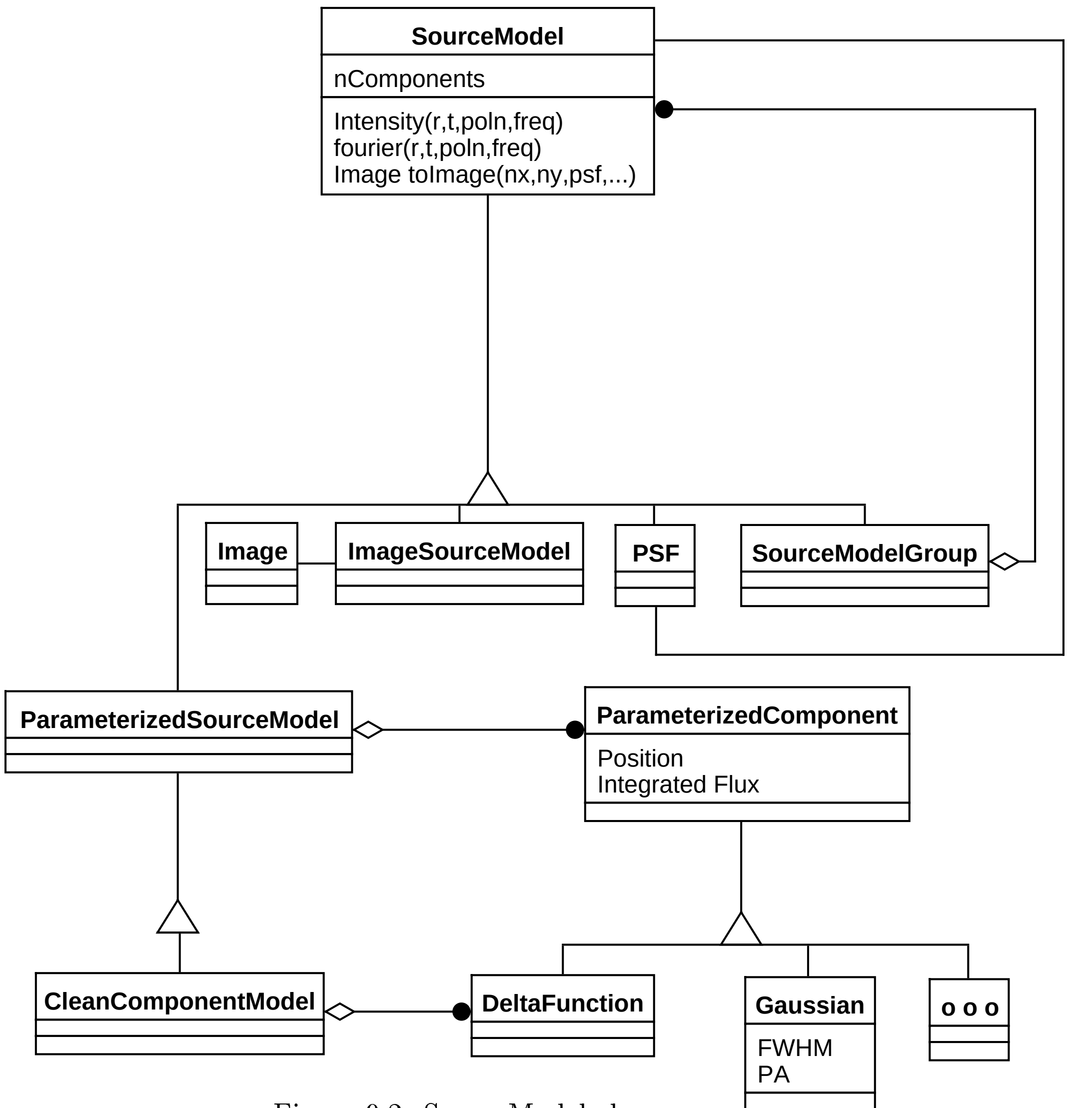


Figure 9.2: SourceModel classes

age. Fundamentally, it returns an intensity as a function of position, time, polarization, and frequency. It is used to represent astronomical sources before observation, *i.e.*, it is not convolved with a point spread function (PSF).

There are several different types of SourceModels. A *Parameterized-SourceModel* is implemented as a series of parameterized components. A Clean component list is a specialized version of this. An ImageSourceModel is one in which the components consist of the pixels from an Image. A SourceModelGroup is implemented with other SourceModels (in linear combination). So, for example, this could be used to combine an ImageSource-Model and a ParameterizedSourceModel.

Besides getting the intensity of the source model (in either the image or Fourier plane), a SourceModel may be gridded into an image (after specifying things like image size and point spread function).

The point spread function of an instrument is its response to a point source — a delta function is the ideal point spread function. The point spread function might in principle be a function of time and position. Often, however, the PSF can be simply parameterized, or it can be represented with a single image. The same machinery that is used to implement the SourceModel can also be used for the PSF. For example, a ImageSource-Model could be used to hold a synthesized beam, a simple Parameterized-SourceModel could hold a single antenna beam, and a more complicated ParameterizedSouceModel (or a new derived class) could hold an HST PSF.

The SourceModel and PSF are astronomically distinct ideas. However it is not clear that the software difference between them (how the "intensity" is interpreted) Is large enough to warrant a new inheritance hierarchy. It is shown in figure 9.2 as a derived class. This relationship may be changed.

An Image will normally be associated with a PSF. The produces of the PSF needs clarification; it may be the *joint* responsibility of a Measurement-Model ("dirty beam") and TelescopeModel ("primary beam").

Telescope model and components

As previously described, a MeasurementSet might present data which is either uncalibrated (*i.e.*, raw) or which has had some calibration applied to it. The *TelescopeModel* and *TelescopeComponent* classes implement this process.[10]

[10] Incidentally, earlier versions of this scheme had calibration being applied in the data view (*e.g.*, VisSet). I believe that the requirement that we allow for partially calibrated data requires that calibration results in a MeasurementSet, not a VisSet. In other words,

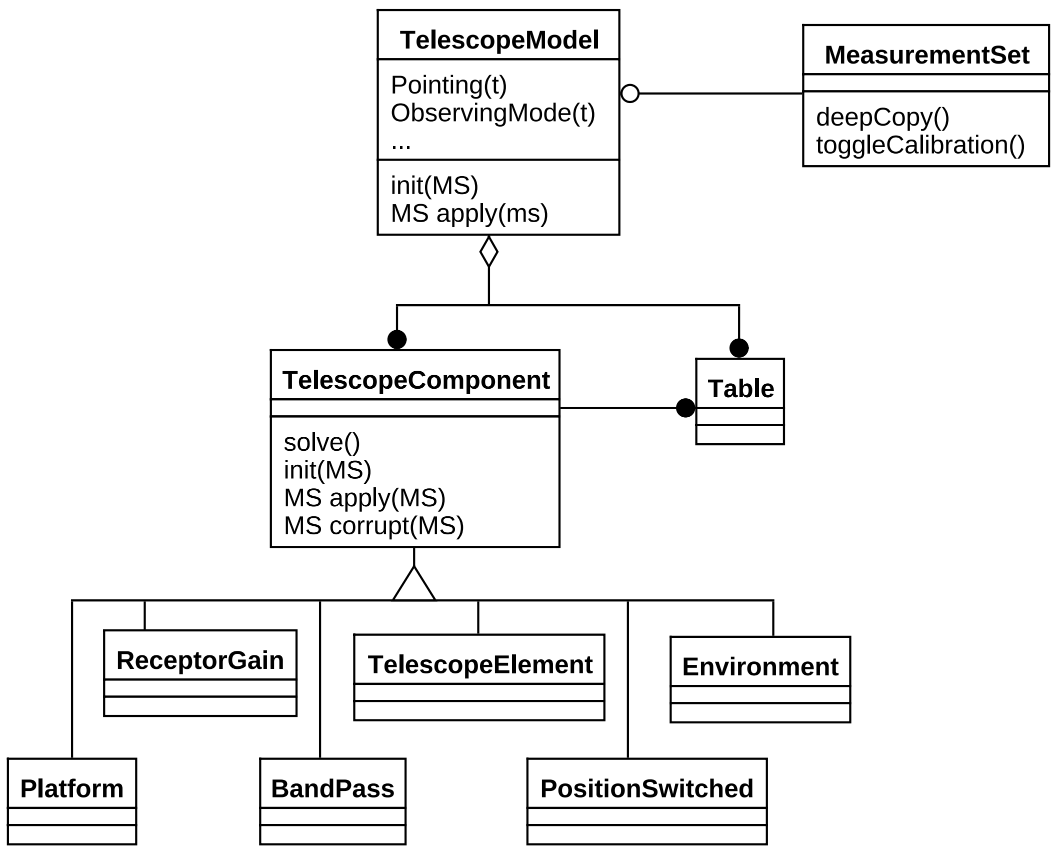


Figure 9.3: TelescopeModel related classes

The TelescopeModel, as its name implies, is a software model of the real observing system; in particular how the observing system it corrupts the data. The goal is, of course, to correct the data as much as possible (or, occasionally, to simulate the effect of observations on perfect (simulated!) data).

Calibration is actually performed by a *TelescopeComponent*. The TelescopeComponent embodies a particular type of calibration; for example, a *ReceptorGain* would be used for an antenna-based gain solution, and a *BandPass* component would be used for squaring up the passband of the instrument. The TelescopeModel then consists of an ordered set of TelescopeComponent objects which apply their calibration in turn. To add a new type of calibration, one merely needs to create a new TelescopeComponent and slot it into the calibration chain. If the calibrations cannot be calculated independently, one merely creates a single TelescopeComponent in the place of those that are coupled; for instance, the Australia Telescope Compact Array (ATCA) performs the receptor gain, polarization, and bandpass calibration in a coupled fashion — here one would produce an instrument specific TelescopeComponent which would replace the more generic versions.

So, the fundamental member function is *apply* (or its inverse, *corrupt*). For a TelescopeComponent, apply applies the particular calibration it knows about to a MeasurementSet. The TelescopeModel pushes a MeasurementSet through its active TelescopeComponents.

It needs to be emphasized that while apply logically creates a new MeasurementSet, normally the produced MeasurementSet will not make a physical copy of the data with calibration applied: instead, it will perform an "on-the-fly" ("on-demand") calibration, referring to values in their original tables (although it is possible to physically apply the calibration and make a deep copy if desired). The mechanisms for doing this are described in the section on Table Data Management (9.4.2) below. Essentially, the Telescope merely needs to produce a new column which hides the old one, which in turn might be hiding an older one, *etc.*, until we finally arrive at the raw data.

TelescopeComponent objects will in general have state associated with them; for example, a ReceptorGain gain component has a gain table: a table which contains columns of complex multiplicative gains which vary

---

we need to produce calibration chains of indeterminate length —$f(g(h(\ldots(data))))$ — if we want to be able to change and replace functions in the calibration chain, it is required that the domain and range of those functions be the same, *i.e.*, the MeasurementSet.

with time, a gain per antenna. Additionally, it will have state for things like interpolation policies.

This state needs to be initialized and updated. Generally the initial (default) values can be obtained from a MeasurementSet, so the TelescopeComponent has a method to initialize itself from a MeasurementSet, or the TelescopeModel can initialize all its contained components in turn. (Of course, the components state might also be set by the user in an ad-hoc way.)

There is, unfortunately, no way to set the "solve" (update) parameters in a generic way — the operations in the particular telescope components are too diverse. Put another way, the user can use data without knowing precisely how it has been calibrated, but the calibration solution cannot actually be performed without the user specifying some calibration-specific information (in general; many particular TelescopeComponents, for example a single dish PositionSwitched telescope component, might be able to behave quite sensibly in a default setting).

When the solution information has been set ("associate this calibrator with those sources, use only long baselines") *solve* itself may be called repeatedly (for example, in a self-calibration loop) to the extent that the parameters to be used during the solution do not need to be changed.

The TelescopeModel has two roles. The first, as described above, is to organize, and marshal data through, an ordered list of TelescopeComponent objects to perform a calibration. This function is reasonably well understood. Its other responsibility is to report on the state of the telescope in general ("where was I pointing at some time, where are my antennas"). In my opinion, the division of responsibilities between the TelescopeModel and the MeasurementSet are not clear in this respect and are in need of clarification.[11]

MeasurementModel

The *MeasurementModel* class encapsulates how a perfect telescope samples the sky producing a MeasurementSet, and the inverse process. The particular MeasurementModel will have assumptions and simplifications built into it; for example when imaging a small portion of the sky it might be considered to be flat.

The primary methods of a MeasurementModel are:

[11] However this is not a large point; we merely have to decide what values we want the TelescopeModel to have and export them — normally it would fill its internal tables from the MeasurementSet.

invert Given a MeasurementSet, return an Image and a PSF (dirty beam). The name of the method is perhaps unfortunate for image-plane to image-plane MeasurementModels (like in single dish).

predict Given a SourceModel and a MeasurementSet paraform of where we want to sample, return a MeasurementSet that would be observed by a perfectly calibrated telescope.

MeasurementModels might be quite complicated; for examples see [**?**, **?**].

The "predict" method is used in iterative calibration and imaging processes, like self-calibration. It uses the MeasurementSet to determine the locations at which to perform the prediction. The values in the original MeasurementSet might not be needed at all (the values from the SourceModel are used).

As presently formulated, image-plane corrections (*e.g.*, primary beam) for interferometers would take place in the MeasurementModel, not in the TelescopeModel.

9.3.2 Status

Fairly complete MeasurementSets have been written for single dish and the ATCA; only a very simple one for general-purpose interferometers has been produced. A combined general MeasurementSet for both interferometers and single dish is expected to become available at about the time of this writing (late 1994).

For single dish, a reasonably complete position switched telescope component has been implemented, along with an on-the-fly imaging application. The latter has not been integrated into a MeasurementModel. The next few months should see single dish work start to rapidly fill out.

The effort in interferometry has been largely dormant in 1994. A more active effort to implement core interferometric functionality will occur in early 1995.

9.4 Table Data System

The underlying raw data for both radio interferometry and single dish is often both quite complex and quite voluminous. Moreover the access patterns users and programmers desire is often quite demanding.

AIPS++ has chosen tabular interface as the fundamental interface to data. Tables in general have been a very successful data type in many astronomical data processing systems (AIPS, IRAF/STSDAS, Midas). Tables are also widely used in FITS for non-image data. The AIPS++ Table interface, as described in the next section, is similar in spirit, although considerably different in detail, than these others.

We have separated out from the Table interface exactly how the bytes are staged from disk (or, indeed, from elsewhere). Indeed, a given table can have different parts which are handled separately. The details about this separation are described in the section on *Data Management* (section 9.4.2). This separation of the data interface from the details of its implementations allows us to use different (possibly data dependent) I/O strategies, which the user of data need not be aware of (the creator of the data needs to set up the strategies that seem to be appropriate).

9.4.1 Table interface

An AIPS++ table consists of a header, and a main data table.

The main data table consists of a number of rows and columns. A value is stored at the intersection of each row and column. All values in a column must be of the same type.

The header consists of a set of keywords. A keyword is a named value ("keyword=value" pair). Keywords can either be associated with an entire table (*e.g.*, general information about the observation) or with a particular column (*e.g.*, units for the values in the column).

A value is normally one of the following types (see *Virtual columns* in section 9.4.2 for a generalization):

A scalar All the usual types are available: integer (short and long), floating point (single and double precision), complex (single and double), string, binary string, and boolean.

An array An array of the above scalar types may be a table value. The array may be of any dimensionality and shape.

A table A table itself may be stored in a column or a keyword. In this way, the table data structure is hierarchical and can directly support groupings (*e.g.* many-to-many relationships, or attaching a calibration table to a dataset).

Note in particular that any value which may be stored in a column, may also be stored in a keyword. Thus one can, for example, store a rotation matrix in a single keyword rather than having to encode it in multiple keywords.

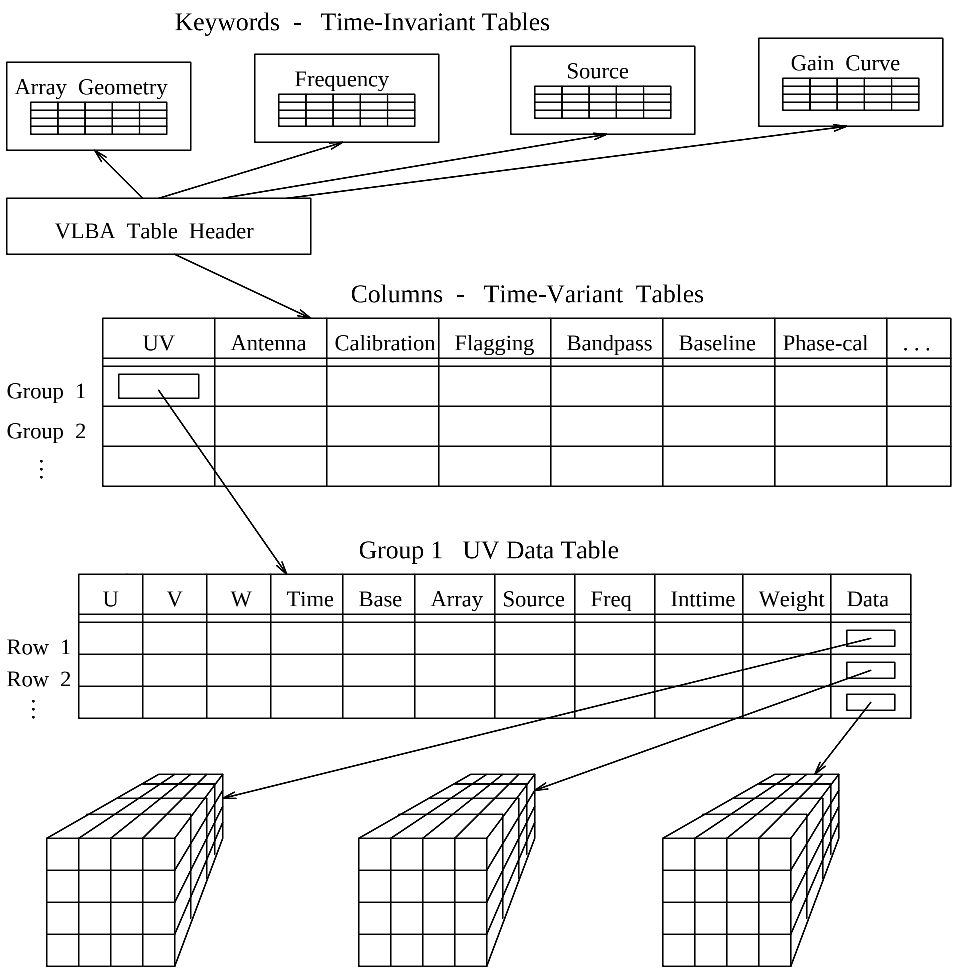


Figure 9.4: An illustration of table features, including tables and multidimensional arrays as values.

An array or table may either be stored directly or indirectly. A direct array or table is embedded directly in its containing table; when in a column, a direct array or table must have an identical structure on each different row[12] An indirect array is stored externally to the enclosing table, and its shape (and hence dimensionality) may vary from row to row.[13] Similarly, an indirect table may vary in structure from row to row; moreover, an indirect table may be referred (indirectly) from multiple tables. Figure 9.4 illustrates a possible decomposition of VLBA data into AIPS++ Tables.

Figure 9.5 gives an overview of the main table classes. There are classes used to:

Access the data Where the data might be in columns (*ScalarColum, ArrayColumn, SubTableColumn*) or in Keywords (*TableKeywordSet*). Alternatively, a column might be viewed as a vector; this is described in section 9.4.1, below.

Describe the table "layout" The description of the entire Table is described by the *TableDesc*ription, the description of a particular column by the *ColumnDesc*ription.

Iterate through the table This is described in section 9.4.1, below.

Manage the data For example, perform I/O to bring the requested data into the user's address space. Data management is described in section 9.4.2 below.

The structure of a table is described by a table descriptor. A table description can be used to create new tables (*i.e.*, with no rows). Thus a table description can be used both as a template for creating new tables, and for describing the structure of existing tables. Note that in using a table description as a template, it only describes the minimum of what a table must have, additional columns and keywords may be added.[14]

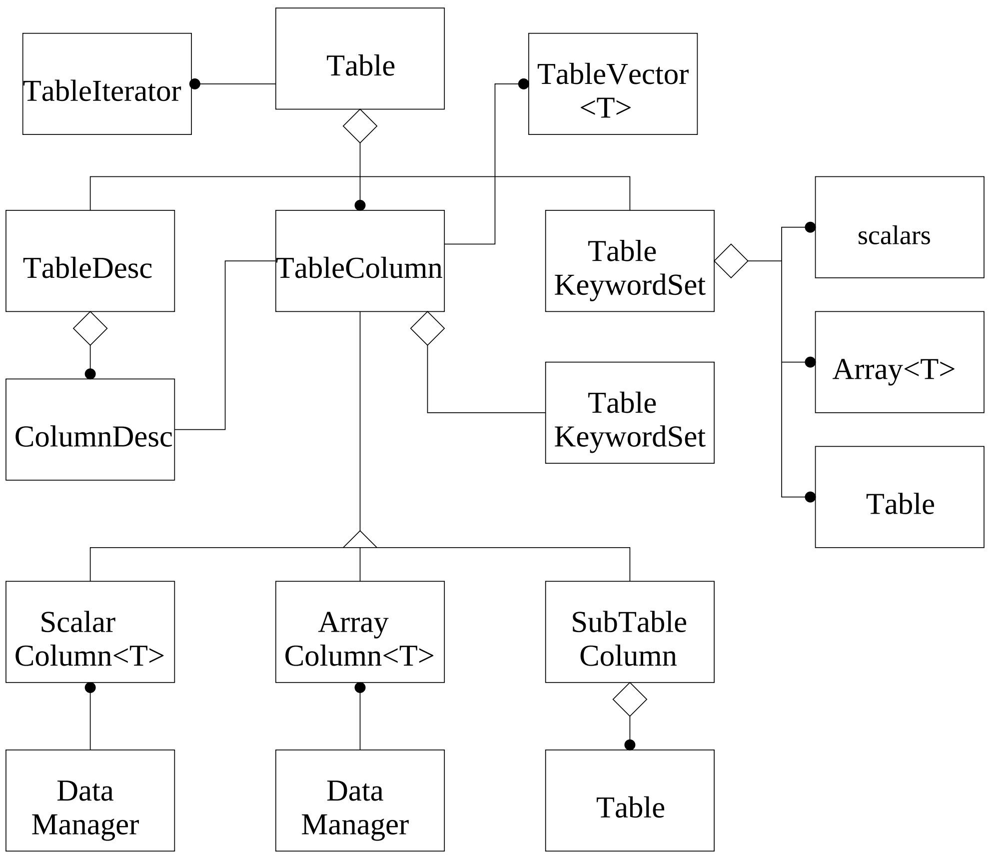
[12] For these purposes, all indirect arrays and tables are identical; think of them as pointers. This rule ensures that all rows have the same size.

[13] Actually the dimensionality may be fixed in the table description if desired.

[14] A description can also be incompletely specified, for example:

- A column named "Data" containing complex arrays must exist;
- A column named "Data" containing 1-D complex arrays must exist;
- A column named "Data" containing 1-D complex arrays of length 512 must exist.

In practice, the second (or possibly first) alternative would be likely be the right level of detail. One wouldn't want to forbid the number of channels of the detector from



Conceptual overview of the main table system classes

Table	interface to the keywords and columns of any type of table
TableKeywordSet	keywords attached to a table or a column
TableDesc	table description
ColumnDesc	column description
TableColumn	interface to common attributes of columns
ScalarColumn<T>	interface to a column containing scalars
ArrayColumn<T>	interface to a column containing arrays
SubTableColumn	interface to a column containing tables

There are also classes ROTableColumn, ROScalarColumn<T>, ROArrayColumn<T> and ROSubTableColumn to be used for readonly columns.

Figure 9.5: Overview of the main table classes

"Virtual" tables

It needs to be emphasized that the Table is an *interface* to data. The actual data may exist on disk in some files. However, it might also exist in some other underlying table objects, or it might be computed on-demand via some computation when the user requests it.

For example, the data might be stored on disk as 16 bit integers, and "decompressed" into floating point for the user. Or, a column might perform an on-the-fly calibration for the user.

Tables where the data are available from files in the normal way are referred to as "filled" tables. Tables and columns in which some of the data are computed (or come from some other source) are known as "virtual" tables or "virtual" columns. This usage of "virtual" is probably unfortunate, though descriptive, given the common C++ meaning of that term. The mechanisms by which virtual columns are created are described in section 9.4.2 below.

Selection and Iteration

A particularly important type of virtual table is one in which all the data is actually in another table. This is known as a reference table. Essentially the reference table has an association with another table, as well as an ordered list of row numbers which map the other table's rows into row numbers of the reference (*i.e.,* virtual) table.

Reference tables are most commonly formed as the result of:

- A selection

  "All rows where column 'Flux' is `>=` 0." Both a C++ set of classes and a grammar exist for performing selections. Both logical operations and arithmetic are supported.

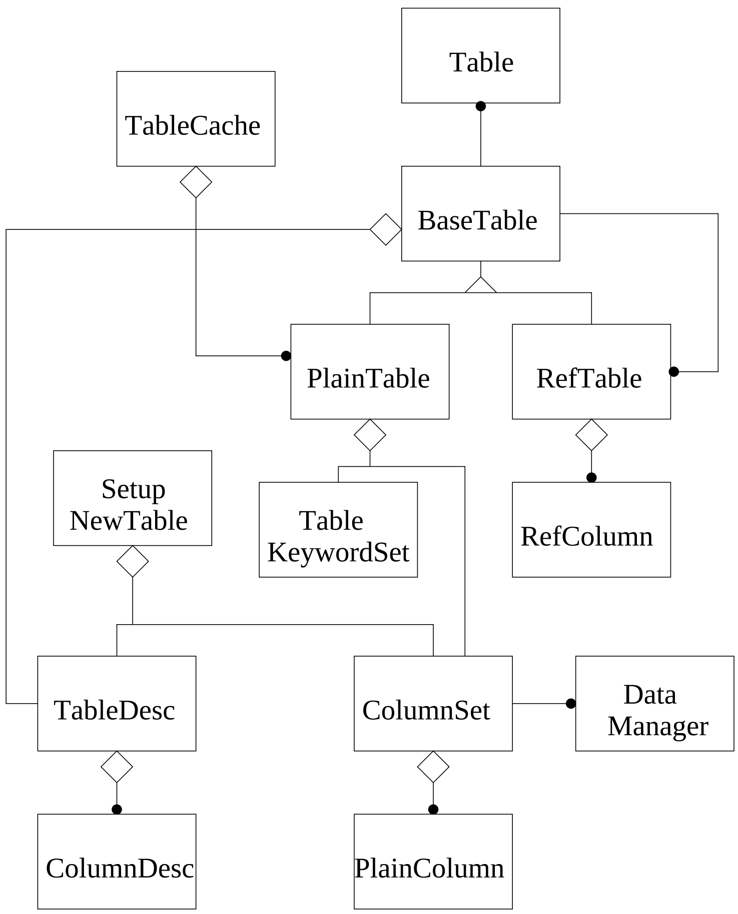
- A sort

  The table can be sorted using multiple columns as primary, secondary1, secondary2, *etc* keys (in ascending or descending order).

- Manual specification via an array of row numbers and/or column names.

A reference table is thus a new view of an existing table. If the reference table is modified, the underlying data in the original table is changed. While

changing, on the other hand the user of the table shouldn't have to engage in a lot of run-time detection of the table structure.



The class hierarchy of the Table classes

Table	envelope class to generic base table class
TableCache	cache of already opened tables
SetupNewTable	class to create a new table
RefTable	table resulting from a select or sort
PlainTable	a table itself with filled and/or virtual columns
TableKeywordSet	keywords attached to a table or a column
ColumnSet	the set of all columns in a plain table
TableDesc	table description
ColumnDesc	column description
PlainColumn	abstract base class for plain columns

Figure 9.6: More detailed Table class diagram

this is normally what is wanted, a reference table may be deepened by making a physical copy if desired.

Another important type of virtual table is the iterator table. One often wants to iterate through a table with a "cursor" which is a smaller table than the original (fewer rows and columns). Once the iterator is formed, the columns viewed remain constant. However the rows which are seen change as the cursor is moved through the underlying table. Commonly, a table iterator is used to read through data grouping rows in some specified order, for example, all rows with a given time or baseline. Note that the rows which are contiguous in the iterator need not be contiguous in the underlying table.[15]

Table vectors

Often one might want to perform calculations using entire columns. One approach would be to merely read the column into a one-dimensional array and then calculate normally using the available functions which calculate on arrays.

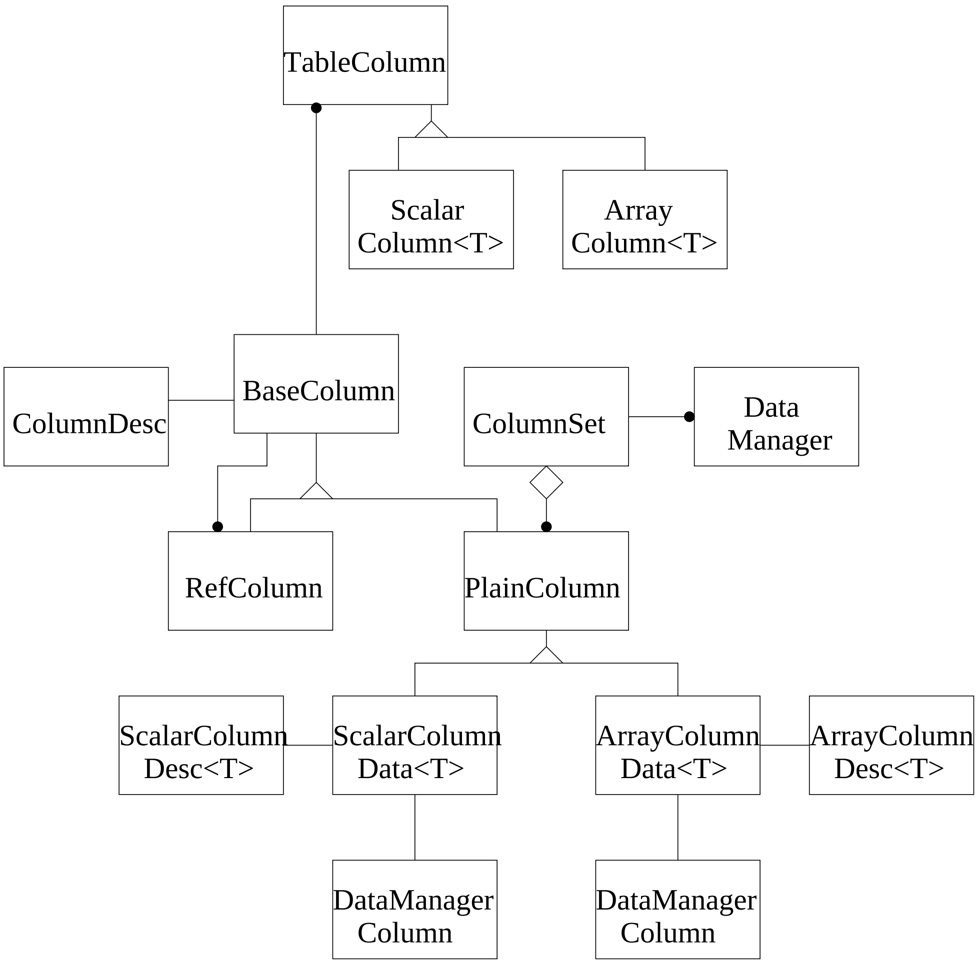
However this is somewhat unsatisfying for the following reasons:

- While an entire column can be read into an array in one go, it can still be mildly tedious to get out a bunch of columns, compute with them, and put them back.

- Copying the data in and out of the columns might be expensive enough that it would instead be necessary to loop over the rows explicitly, losing the clarity of whole-array arithmetic.

- Tables might have very many rows (certainly many millions). Thus the temporary arrays used for the calculation might cause out-of-memory problems.

The solution we have chosen to solve this problem is to introduce the TableVector class. It is logically an entire column which can be manipulated as an array (*e.g.*, arithmetic, logical operations, *etc*). However, it is not (necessarily) entirely memory resident. The addition of two table vectors would result in a buffer sliding through the table. However, this I/O would be entirely hidden from the user.



The class hierarchy of the Table Column classes

TableColumn	user class for any table column
ScalarColumn	user class for column containing scalars
ArrayColumn	user class for column containing arrays
BaseColumn	abstract base class for any table column
PlainColumn	abstract base class for any column in a plain table
RefColumn	column in a RefTable (referencing another column)
ScalarColumnData	internal class for column containing scalars
ArrayColumnData	internal class for column containing arrays

Figure 9.7: How columns are attached to Data Managers.

9.4.2 Data Management

Data is mapped to and from a table interface via data managers. A data manager fundamentally maps "get" and "put" requests to the implementation data structures (or functions, for virtual columns). Multiple columns are bound to a data manager, and a table may have one or more data managers attached to it. This is an important part of the design: it allows a single table to have multiple types of underlying I/O (presumably tuned for data dependencies) or virtual columns attached. The classes which are involved in attaching columns to data managers are shown in figure 9.7.

While the Data management layer is below the level at which table users are required to be knowledgable, it is a level which developers who (particularly) need to add additional types of virtual columns need to be aware.

The creator of a table may also need to be aware of the different types of storage and data managers so he can choose the ones which optimize the access that he foresees.

Storage management

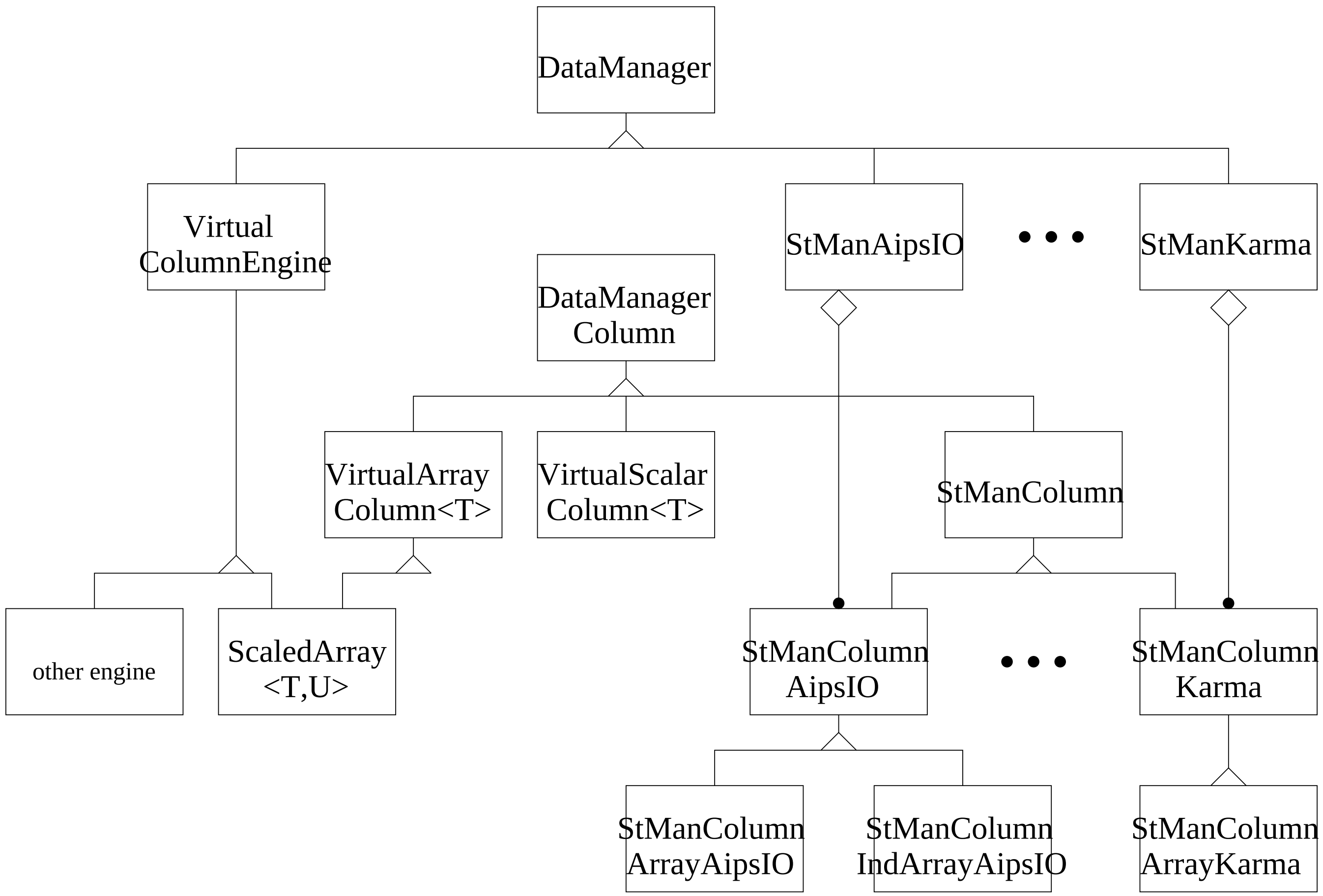
Data managers which physically store and retrieve values from a storage device are known as storage managers. Besides staging data to and from disk[16], they are responsible for canonicalizing it (in particular, to IEEE Big Endian) so that computers with different word formats can access the data.

There will be several different types of storage managers in AIPS++, each with different properties. The ones which are either presently implemented or which are being implemented are:

AipsIO AipsIO is a simple I/O system which is used to store object values (describe in the class reference manual). AipsIO has no partial buffering: an object is either in memory or on disk. As used as a storage manager, all active columns are stored entirely in memory. Thus AipsIO is most useful for columns which are small enough to be memory resident. An exception to this is columns of indirect arrays, which are only read on demand. Moreover, if only a section of the indirect array is desired, AipsIO will only read in that section. Thus, AipsIO is appropriate for small columns and columns of indirect arrays.

[15] Presently iterator tables are actually implemented with reference tables.

[16] In principle storage managers could be written for tape drives or other devices.



The class hierarchy of the DataManager classes

Figure 9.8: Relationships among DataManager classes

Karma Karma[17] is a storage manager which is optimized for dealing with data which can be organized as hypercubes, for example interferometric visibilities with a constant number of stokes, channels, and baselines per timestamp. It is optimized for taking slices in any direction (for example, all visibilities for a range of channels for a given baseline). It achieves these optimizations through use of "tiling," which is a technique in which the data is broken into chunks which may be readily assembled in any direction.

Miriad-like When data in some columns varies slowly, a risk is run that the total size of the table will become bloated as the slowly varying column becomes replicated over many rows. Another risk, is that data which logically belongs together will be split apart for this implementation reason. In AIPS++, we have defined a storage manager based on the ideas in the Miriad software system ([**?**]) to alleviate this problem. With the Miriad storage manager, values are only written when they change. That is, the underlying implementation is list-like. (However, indices are layered on top to make random access reasonably efficient). (The need for this is also described in section 9.3.1.)

It should be clear that all of the above have different performance and access requirements.[18] This lets the table creator choose tradeoffs that he feels are appropriate. No software to automatically migrate from one storage manager to another exists yet (short of a physical copy of the table).

Virtual columns

Within this framework, virtual columns may be readily constructed. The only thing that is required is the creation of a so-called VirtualColumnEngine, which is merely a protocol for storing and returning values given a column and a row number.

The first virtual column engine which has been implemented is one in which values of one type are scaled to values of another type via a simple $new = old \times scale + offset$ calculation. For relatively low signal to noise

[17]Karma is a programmers toolkit for scientific computing with communications support, event management, data structure manipulation, graphics display and user interface support. Information on Karma may be found at `http://www.atnf.csiro.au/karma`. Karma was developed by R. Gooch.

[18]For example, it is expensive to change a value in the middle of a Miriad-like dataset, however it is quite cheap to extend it. On the other hand, a Karma data set cannot be extended at all once it is created (short of a copy).

data, it can make sense to "compress" floating point data down to short integers (for example). However this compression is an optimization that the consumer of the data does not need to be aware of; he just computes normally on his floating point data.

Virtual columns have one capability that filled columns do not: they may contain any type, not just the scalars,arrays of scalars, and tables which may be stored directly.

9.4.3 Status

The design of the AIPS++ Table Data System was initially formulated by A. Farris of the Space Telescope Science Institute, and implemented by G. van Diepen of the NFRA (and also kindly supplied the figures in this section).

The classes and functionality described in this section are entirely implemented with the following exceptions:

- Adding a column to an existing table
- Tables in columns (tables in keywords are supported).
- Integrating the TableVector class with the Lattice class (section 9.5.1).

The table system saw a complete overhaul in 1994 in response to suggestions from clients of the previous version of the table classes.

Besides finishing the above items, future work will involve such things as I/O optimizations, and improving the ability of end users to directly manipulate tables.

9.5 Array and Image Plane Computations

Arrays have traditionally played an important role in scientific computation. While it is certainly true that some of the reliance on arrays was due to the paucity of other data structures in FORTRAN, it is also true that computation on arrays reflects the common occurrence of regularly sampled multi-dimensioned data in science. As a pragmatic matter, high performance computers often have optimizations (in the hardware and compilers) for array operations.

Besides arrays, there are other structures which have array-like characteristics. The most important for AIPS++ is the Image: typically a regularly

Module: lattice Sheet: 1: lattice Panel: (0,0) Time: Thu Dec 1 17:03:34 1994

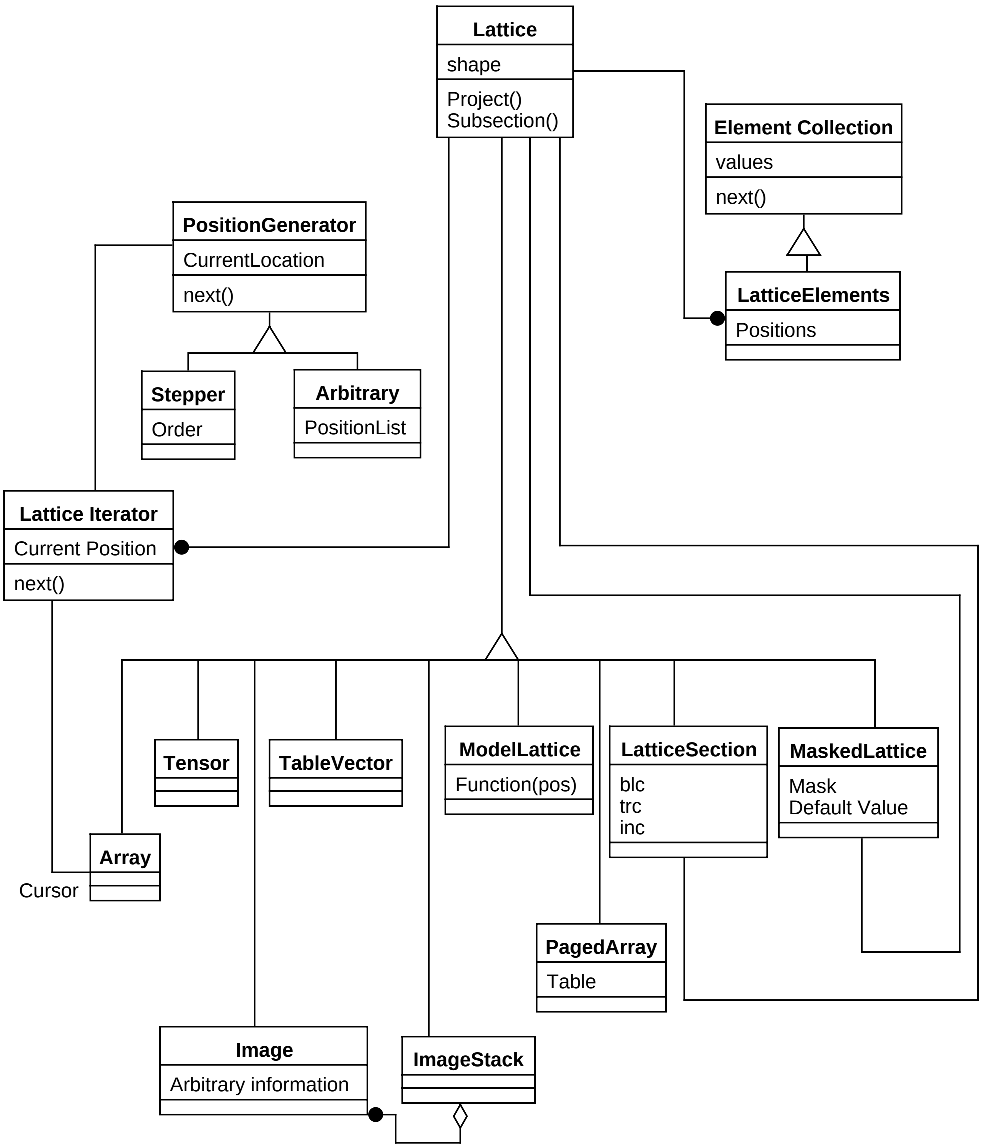


Figure 9.9: Lattice classes

sampled sky intensity in some polarization along axes related to position in the sky, and frequency.

This chapter presents a coherent set of classes and functions for computing upon arrays, images, and regular structures in general.

9.5.1 Array and Lattice classes

A *lattice* is an abstract base class which encapsulates a regular multidimensional data structure with a constant number of positions along each axis. The length of the axes, expressed as an ordered tuple of integers, is known as the *shape* of the lattice. Elements contained by the lattice are of homogenous type and may be placed into the lattice, or retrieved from it, at positions which correspond to intersections of the positions along each axis. A lattice contains a total number of elements equal to the product of its shape. Note however that it is not required that all the elements be distinct, for instance in a symmetric data structure changing one element may also change it at its mirrored locations.

The defining characteristic of a lattice is its shape. The fundamental operations on a lattice on a lattice involve setting and retrieving single elements, or collections of elements. The location of an element in a lattice is defined by a tuple of integers, giving the position of the element along each axis. This tuple is a class known as an *IPosition* (Integral Position). (Logically an IPosition is a 1D array of integers.)

An *Array* is a type of lattice in which all elements are unique and fit into memory. Moreover, the position of the elements in memory may be calculated solely from their position in the array; thus a sparse matrix implemented as run-length encoded lists of elements would not be an array in this sense.

It will often be convenient to have 1-, 2-, and 3-dimensional specializations versions of array classes to relieve the tedium of indexing arrays of arbitrary dimensionality, and to provide operations that only make sense for an array of a given dimension. Generally speaking, arrays opertions should be fast — both for whole-array operations and individual element indexing.

Arrays play another special role in the lattice hierarchy. Often computations will be required on a regular set of elements from a lattice ("the m'th plane"; "every i'th row, j'th column, ..."). These operations would be too slow if every element access required a function call; instead, the region is accessed through an array. That is, arrays give efficient access to elements from a regular section of the lattice.

Note in the figure (9.9) that a lattice is often associated with an other lattice, for example *MaskedLattice*. That generally means that the derived class is a *view* of another lattice. Moreover, since Lattice is a base class, implicit in this diagram is that the MaskeLattice might in fact be *e.g.*, an Array.

Note also that all these classes are templated (also known as "parameterized" or "generic" classes in some languages). This means that the lattice classes are general containers, they may (in principle) hold elements of any type (however, all elements of a particular lattice are of the same type).

Element collections/iteration

The simplest thing a user or client of a lattice might want to do is operate on some collection of elements from a lattice, where there is not (necessarily) any intrinsic regularity in the locations of the values. That is, there is only a "next()" function. An example for where this type of interface would suffice is calculating statistics, either on a whole lattice, or on some irregular set of points from it[19] (for example, defined by a user selecting points from a GUI display of an image). In general, calculations which proceed "element by element" can use an element collection. In practice, this element collection is more like an iterator than a container class.

An element collection has no positions: it's merely a source of values (a linearized traversal). In fact, an element collection does not need to be attached to a container, it could equally well be attached to an input source.

Unlike an ElementCollection, the *LatticeElements* class has position from the lattice for each element. So, for example, if one is iterating through all the pixels of an image that meet some brightness threshold, one would know not only their brightness but also their location in the underlying lattice.

Lattice Sections

A *LatticeSection* is a view of a regular portion of a lattice. The portion of the underlying lattice which is being viewed may be described by a start, a length, and a stride along each axis. Note that a LatticeSection *isA* Lattice (it has a shape), and it also references an underlying lattice (that actually holds the values). The underlying lattice can of course be any derived lattice type, such as an Image.

[19] Actually element iterators like these are of wider utility than for only traversing lattice regions; they are of wide utility for containers in general. In C++, the best example of this is the "Standard Template Library" (STL) which is being adopted by the ISO/ANSI C++ standardization committee. The AIPS++ element iterators should be STL compatible to enable their use with third party libraries and functions.

A LatticeSection will normally be produced by one of the following:

Projection to lower dimensionality. This results in a LatticeSection with a diminished dimensionality (*e.g.*, row() and column() from a 2D structure, plane() from a 3D one).

Subsection which results in a LatticeSection with the same dimensionality.

Masked Lattices

Another useful type of Lattice is a *MaskedLattice*, in which positions are marked as either "valid" or "invalid" through the use of a mask.[20] When elements which are marked as invalid are accessed, either directly or through an iterator, a default value is returned instead of the actual underlying value. In practice the most common default values are apt to be zero, or a not-a-number (NaN) value.

Masks can come from a variety of sources:

- A "bad data" marker, either directly ascertained from the telescope data, or deduced by some application.
- From the user, *e.g.*, through drawing regions with a GUI image display device.
- Through application of functions which operate on masked arrays, for example:

```
image(image > 5.0) = 5.0;    // Clip
```

Another approach which is similar to this is to use "magic values" to mark the data as good or bad. That is, a special ("magic") value is written into the lattice, and then various support functions look for that value.

We may want to implement magic-value blanking as well as masking. While less flexible, it is more efficient in space, and it irretrievably marks data as bad. A mask can be lifted off the data. While this is often desirable, it does lead to the possibility that invalid data might leak into computations. Of course, the data behind the mask could be set to a NaN value when this is a serious concern.

[20] Actually, the mask is an implementation detail. The same result could be had by storing a list of valid (or invalid) positions. However, the use of a mask results in a more predictable cost. Our implementation of a MaskedLattice contains two lattices: one for the data and the other for the mask.

Lattice Iteration

Very commonly, the most convenient method of computing on a lattice is to process it "chunk by chunk." Common examples of this are processing a spectral line cube plane by plane — each plane is an image at a particular frequency – or processing it spectrum by spectrum.

In AIPS++, we iterate through a lattice in this way by using a *LatticeIterator*. The lattice iterator has an array *cursor* (window) attached to it, which is scrolled through the image. The locations in the lattice that the cursor is filled from is defined by a *PositionGenerator*, which has at least two derived classes:

Stepper A stepper is used to implement the above "usual" traversals. Basically, the cursor is moved by a set amount each time in directions specified by the user (*e.g.*, transpose orders). Note that the stepper does not need to move by a width related to the width of the cursor. For example, if one wanted to boxcar average an image along the frequency axis, one could move the cursor by one in "Z", even though the cursor width in "Z" is the number of channels one wants to average over.

Arbitrary One might want to, for example, examine small patches around a series of points in an image. An arbitrary position generator has within it a set of points that specifies the locus of positions the cursor will move through.

So, the cursor *shape* defines the chunks that the user is interested in, and the *PositionGenerator* describes the movement of the cursor.

One problem that occurs is an iteration which would extend the cursor partially beyond the edge of the lattice. There are (at least) three solutions that are appropriate in different circumstance, and hence the user must be given the choice of selecting the desired policy:

Move the cursor to the next position where it is entirely contained within the lattice.

Resize the cursor so that it only extends exactly to the end of the lattice.

Fill the portion of the cursor that extends beyond the edge of the lattice with some default value (often 0.0).

The cursor itself is an array. This provides both efficient access to the individual pixels, and provides access to all of the operations which are defined for arrays.

Note that this iteration abstraction *is* the I/O mechanism for lattices which do not fit entirely within memory (*i.e.*, large images). The cursor is the "I/O" buffer, and the cursor shape and position define the disk head seeks required to fill it.

ModelLattice

From time to time one will want to view a (multidimensional) function as a Lattice. For example, one might want to use image analysis and display tools on a simulation. While it would be possible to sample the function and use an array (for example), it might be more convenient (or efficient in space) to just use a ModelLattice — a Lattice view of a function.

Array and Tensor classes

As described earlier in this section, *Array* classes are lattices in which all the elements are distinct and stored in memory. In a sense, an Array can be considered a multidimensional view of a chunk of memory which is inherently linear. In practice it, is convenient to have specialized Array1D, Array2D, and Array3D classes to allow, for example, more convenient (and efficient!) indexing.[21]

Logically the array classes need to be separated from the *Tensor* classes, which have different storage layouts (*e.g.*, symmetries) than the Array classes, and different mathematical properties. Of course, some of these classes will use the corresponding array class for storage of the values.

In general, the Array classes are appropriate for applications like image processing, and the Tensor classes for linear algebra and related applications.

PagedArray

A *PagedArray* is a lattice which consists of unique values as with an *Array*. However, there are too many values for the lattice to be kept entirely in (virtual) memory, so a mechanism for staging values to and from disk (*i.e.*, I/O) is required.

As with all Lattices, the user merely decides how he wants to iterate through the data. The actual I/O is performed by the table data system (section refsec:TableDataSystem). In particular, the actual data is stored

[21]In C++, we should probably also make a distinction between arithmetic arrays and general arrays. We might also want array classes that, for example, wrap around Karma library arrays.

as an indirect array in a column. Of course, that column will have an appropriate storage manager attached to it (Karma or AipsIO).

9.5.2 Image related classes

An *Image* in AIPS++ consists of a Lattice, which holds the pixel values, and an *ImageCoordinateSystem*. An ImageCoordinateSystem is an engine which turns pixel positions into real-world coordinates (and the reverse). While the Lattice Image is associated with will normally be a PagedArray, it might be another Lattice type. For example, a MaskedLattice which is in turn associated with a PagedArray corresponds to a Classic AIPS image with blanks.

In the past it has often been convenient for miscellaneous ("arbitrary") information related to the image to be attached to it. This facility is also offered in the Image class, although we need to be careful that this isn't used as a substitute for designing and implementing appropriate classes. It uses the traditional keyword/value protocol.

One feature which has not presently been added to the image class if error estimates (or, alternatively, weights). It seems to this author that this is still a research topic — simple independent, normally distributed, errors with the usual error propagation rules are apt to be very misleading. If, nevertheless, error estimates are required, I would suggest merely associating an error value with each pixel (behind the interface it might be subsampled to save on storage).

An image will normally have a processing log associated with it.

Coordinates and Units

An image pixel's origin is at its center, not one of its corners. While there is no compelling reason for this convention, as discussed in [**?**] this convention has the nice feature that it is invariant to rotations.

A *Measure* is a base class that represents a physical measurement, for example *Direction* is used to describe a position on the sky. Every measure has a *MeasureRepresentation* which has the units of the physical measurement, and allows operations operations rotations. Every Measure also has an associated*MeasureReferenceFramd*, for example topocentric *vs.* barycentric sky coordinates.

An *ImageCoordinateSystem* is used to tie pixel coordinates to their underlying Measure(s). More than one measure may be in an Image (*Direction* and *DopplerVelocity* most commonly). More than one image axis might be

Module: measure Sheet: 1: 1 Panel: (0,0) Time: Thu Dec 1 17:22:20 1994

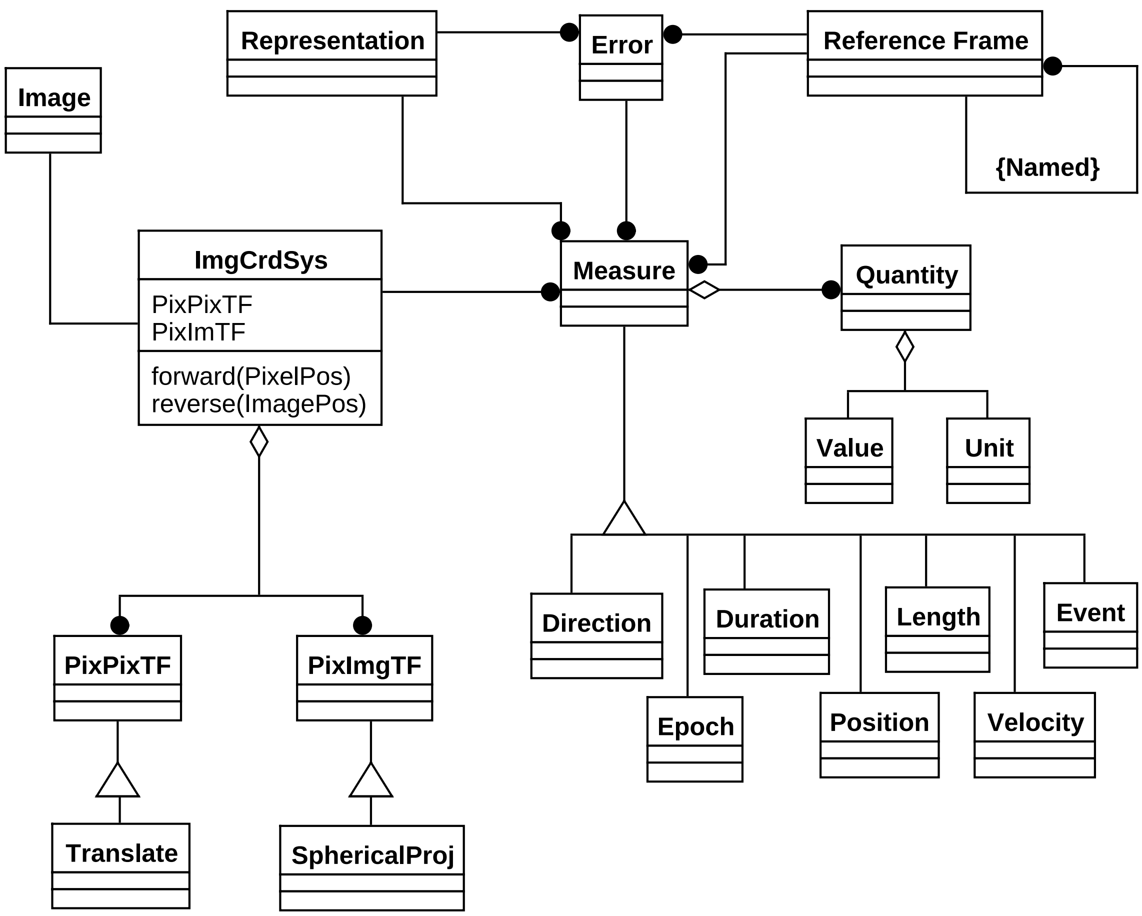


Figure 9.10: Coordinate related classes

required for a given Measure (a Direction requires two, for example). The image axes for a given Measure may not be contiguous in the image (it might have been arbitrarily transposed). An *AxisMapping* is used to map axes to their appropriate Measures.

A pixel coordinate is turned into an image coordinate through a chain of *PixelToPixelTransformations*, followed by a single *PixelToImage* coordinate transformation. (Pixel coordinates may be non-integral, *i.e.*, positions within a pixel may be specified). PixelToPixelTransformations will normally involve linear transformations (translation, scale, rotate *etc.*), however they can in general be also be nonlinear functions.

The final PixelToImage coordinate transformation does any final mapping (usually a spherical projection for a Direction), as well as associating the correct output axes with the correct Measures via the AxisMapping.

Note that if one wants (say) coordinates in an image reported in galactic coordinates rather than right ascension and declination, one merely changes the representation in the Measure. Similarly, if a change from a barycentric DopplerVelocity to one against the LSR (Local Standard of Rest) results from changing the ReferenceFrame.

The data values in the Image in principle also have a ReferenceFrame (so that one ccould apply a red-shift correction to the image through a change in ReferenceFrame). Initially the image may only attach units to the data values.

The *Unit* classes are used to represent physical units. Units may be composed ($velocity = length/time$) multiplicatively, and unit arithmetic is supported (you can add furlongs to kilometers and get the result in parsecs). The Unit is normally part of the MeasureRepresentation, although it may also be used on its own.

ImageStack

It can be convenient to deal with a collection of similar images as a unit. The *ImageStack* class was invented for this purpose.

The rule[22] which can be applied to determine whether or not an axis is a valid for an image, is that it must make physical sense to interpolate along an axis of that type. So, for example, a frequency axis is a valid axis, but a Stokes axis, or any other axis that is essentially an enumeration, is not.

An ImageStack is a collection of one or more images of the same size and with the same coordinate system. The ImageStack might contain Stokes IQUV

[22] First formulated by Mark Calabretta.

images or it might contain the image at different stages of processing. Because the sizes and coordinate systems must be the same, features will "line up." The fundamental operations on an ImageStack involve setting and retrieving images ("give me the 'Q' image"), and getting the values of the images at some position ("What are the IQUV values at some interesting position on the sky").

Note that ImageStack is inherited from Lattice, so that many image-like operations which only require pixel coordinates may, in fact, be carried out on ImageStack objects.

9.5.3 Functions on lattices

Besides the structures which hold and provide the access to data, powerful functions which act upon Lattices are also required for users to be able to concisely express the calculations they wish to be performed.

Fortunately, there have been a wide variety of successful languages[23] which implement such functions, so not much invention is required here. (As always, implementation can be tricky.)

Besides application-domain specific functions (gridding, FFT'ing, and the like), a powerful library of more generic functions is useful. It should include at least:

- Whole-lattice arithmetic.
- Logical expressions.
- General evaluation functions (evaluate a function at all points in a lattice).
- General reduction operators, inner products, outer products, and the like.
- Statistics, special functions, *etc.*
- Various plotting and display functions.
- Data fakers/simulators.

[23] APL, IDL, Mathematica, Fortran 90, S+ *etc.*

Image analysis functions

While an Image IsA Lattice, and hence all the Lattice functions may be called upon an Image, it will often behoove the user to use functions which are more Image-aware. Such functions might need to take into account:

- Coordinate system coincidence. Two images might have have the same shape but very different (non-coincident) coordinate systems.
- Compatible units. For example, when adding two images, they should have the same type of unit; if you are multiplying them, you need to adjust the output unit type.

9.5.4 Status

A set of array classes have been available inside of AIPS++ for approximately two years. There is an N-dimensional *Array* class, along with 1-, 2-, and 3-dimensional specializations named Vector, Matrix, and Cube. As discussed above, these names should be changed to Array1D, Array2D, Array3D.

Lattice and Image classes have been available for a few months. The array classes have not been integrated with the Lattice yet, *i.e.*, the array classes are not yet derived from lattice. However, there is a Lattice class which uses an array which allows much the same effect.

We anticipate writing a new set of array classes in the first half of 1995 to complete the merger between the Lattice and Array classes (and contains various small fixes that two years of use have turned up). The old classes will need to coexist in the system for some long time for backwards compatibility.

The LatticeIterator and PositionGenerator (with a stepper subclass) have been implemented.

The Array classes presently have whole array arithmetic and logical expressions. For example:

```
a = b + c*sqrt(d);    // arithmetic
a(a > 5.0) = 5.0;     // logicals
```

is supported. The corresponding operations for Lattices in general is not yet supported. In other words, a fairly mature set of array classes exist, but integration with the ideas outlined in this section are required.

A somewhat controversial point has been whether or not lattices and derived classes should support variable (*i.e.*, non-zero) origins. While they have been provided to date, on balance it seems that their potential for bugs outweighs their usefulness, and variable origins are deprecated and will eventually be removed.

Another controversial point has been whether or not operator* should perform element-by-element arithmetic, or perform a linear algebra multiplication (although a decision still would need to be made about inner versus outer products). This confusion was caused by the lack of separation between the linear algebra classes and the array classes. Array classes should continue to use element by element arithmetic, and the linear algebra classes should choose an appropriate multiplication to be performed by operator*.

Only trivial coordinates are available. Units classes have been submitted but are not yet in use.

Efficiency Considerations

While whole-array operations have been efficient, operations on arrays where a lot of array-indexing is done in loops is fairly inefficient. Presently the actual indexing of, for example, a Matrix is accomplished by code like the following:

```
T &operator()(Int i, Int j) { return data[xyoffset + i*xinc + y*yinc]; }
```

where xyoffset is related to non-zero origins, xinc is related to the stride in x, and yinc is related to the stride in y and the length of the x axis.

A sufficiently able compiler should be able to optimize (*e.g.*, constant propagation) indexing when it is in tight loops. Many (not all) compilers in fact do not efficiently optimize such loops. Apparently they are unable to determine that the indexing constants are not aliased by the pointer "data." Compilers seem to do much better in general when indexing is performed with an operation like:

```
T &operator()(Int i, Int j) { return dope[i][j]; }
```

So, if we want to make indexing more efficient we need to change the array classes in the following ways:

1. Use "arrays of pointers" rather than multiplication (at least have this as a compile time option).

2. Break out "view" classes so that multiplication by an increment is done as infrequently as possible.

3. Remove the variable origin.

While our experience to date is that the effect of inefficient indexing is well confined in the code (*e.g.,* inserting about 10 lines of FORTRAN made the Single Dish On-The-Fly imaging application run quickly enough) these changes are probably worth making.

A goal of AIPS++ is to allow the user to process image cubes without having to transpose them. For this to be realistic goal, the layout of the image data needs to be carefully defined. If it is done naively, processing a $256 \times 256 \times 128$ cube "spectrum by spectrum" would result in 128 head seeks per spectrum, which would result in grossly unacceptable I/O behavior. To solve this problem, we rely on the Karma storage manager which can tile multidimensional lattice data.[24]

A problem under some circumstances are the creation of temporary objects. A delayed evaluation strategy might be attractive; it would also allow expressions to be analyzed so that, for example, $a = b + c * d$ would only result in one loop, giving the optimizer more opportunities for optimization.

9.6 Processing Environment

The discussion so far in this paper has mostly revolved around classes which are used to construct applications. Also important is the method by which applications interoperate with one another and communicate with the user.

Philosophically, the level of the AIPS++ system needs to be set. At one extreme, it is a virtual operating system which totally hides its users from the underlying OS, so, for example, users can process data on a Unix machine or VMS machine and not notice any difference. At the other extreme, one has independent executables which merely share a data format. AIPS++ is in the middle. It doesn't pretend to be an operating system (we don't have the expertise to write a very good one in any event), yet to allow for convenient processing, close communication between applications must be possible.

9.6.1 Architecture

The goal of the AIPS++ runtime environment is to create, direct, and destroy entities which carry out radio astronomical calculations under both user and programmatic control. The runtime environment is not directly

[24]Actually, it is possible to get reasonable performance without tiling if you assume that you can hold an entire XZ slice of the cube (in this example) in memory at once.

concerned with the nature of those calculations, however, it must be rich enough to support the required interactions.

It is assumed that the environment is typical of those on modern workstations (multi-tasking; the process is the normal level of granularity of computing; networked, hierarchical filesystems). We do not assume that the underlying operating system provides any object-oriented facilities.

The approach presented here is, while perhaps more flexible than a traditional package, still fairly conservative in the sense that it is readily implementable, and doesn't present a pure object-oriented approach at the system level. While I believe this is appropriate for us if it becomes inadequate other approaches are possible.

The AIPS++ runtime environment consists of asynchronously running *processes* running under the control of a *control hub* (which is itself a special process). All inter-process communication is conceptually through the control hub, although from time to time point-to-point communications may be obtained for efficiency.

A process will normally be playing one of several roles:

Task A task is a "program" which is started up with some parameters, some or all of which might be modified as the task executes.

Server A server is a long lived process which performs some computation(s) on demand. While the line between a server and a task is somewhat arbitrary, a task normally carries out some computation once for a user, whereas a server carries out similar types of computations repetitively on the request of users or other tasks.

Proxy object server A proxy object server is a server process which carries out requested operations (*e.g.*, invokes member functions) for objects which it contains. This is a simple implementation of distributed objects; eventually we expect native operating system support for this functionality.

Figure 9.11 shows an example of what the environment might look like in practice.

The distinction between user processes and system processes is not a sharp one, however the user is likely to have directly activated the processes on his side of the line, whereas the system processes probably started at the behest of some other entity.

In the example, the user is directly manipulating an object in an object server through CLI manipulations. Those objects might be tables or images,

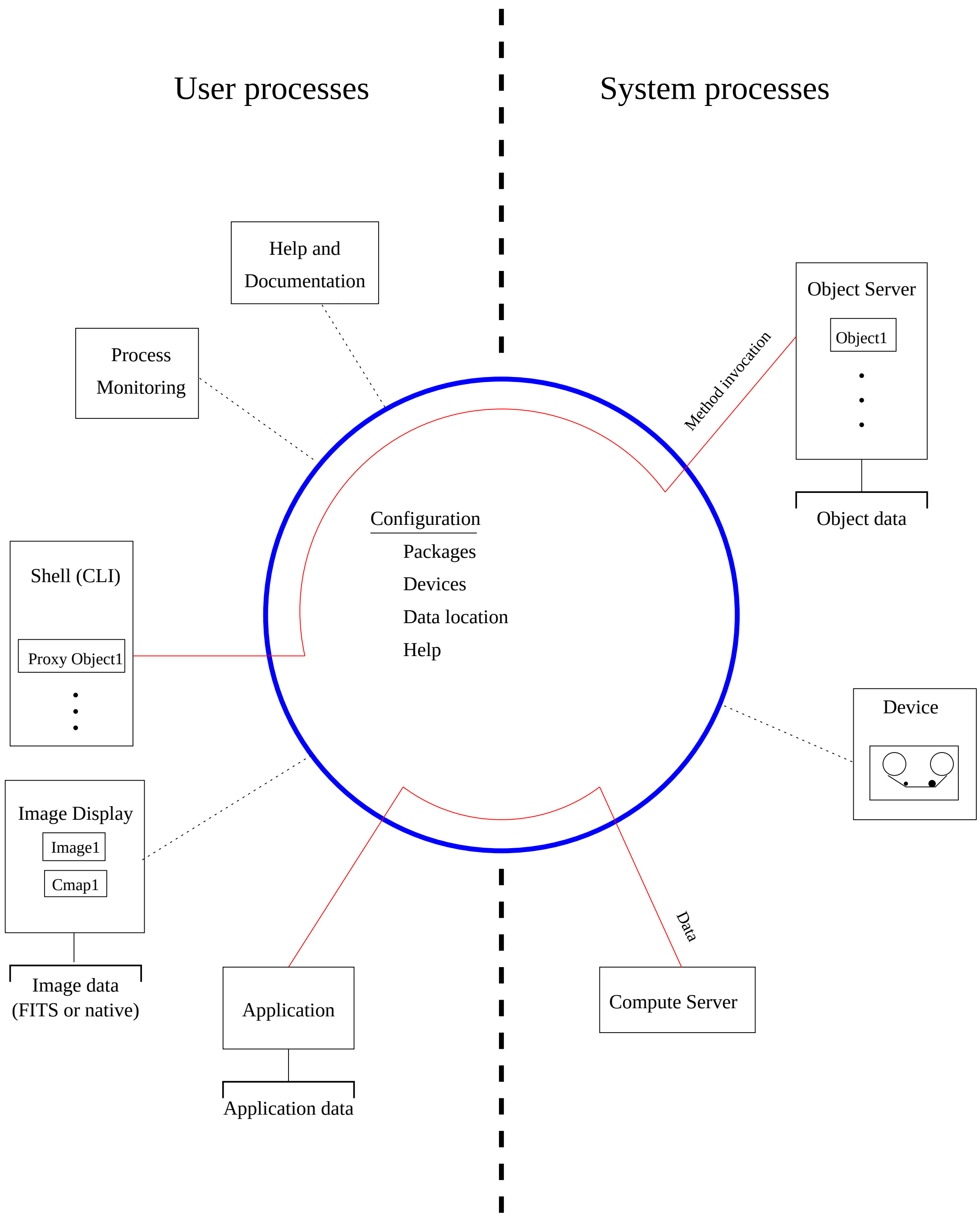


Figure 9.11: Example runtime environment.

or something less generic. Object servers will often need access to data (*i.e.*, Tables), particularly if they are persistent.

The user has previously started some application which is presently communicating with some compute server (*e.g.*, FFT). C++ applications would not normally use a compute server, applications written as a CLI script often will. Compute servers will often be stateless (pure functions), so they may not need a database.

A particularly interesting process is the image display process. It is both capable of displaying and manipulating images in a standalone fashion (at least FITS and AIPS++ native images), as well as accepting connections from other processes. For example, it might display an image sent to it, or give some other process a subregion of the image (for example, to compute statistics upon it).

Also attached to the control hub is a device interface process; these will be particularly important for tape drives. Also shown attached is a "Help" process[25] and a process monitor (log messages, percent completed displays, and the like).

Message-passing processes

It needs to be stressed that the above roles are purely a convention. At the lowest levels we merely have inter-communicating processes. Layers above the system interpret whether a process is an application task, server, or proxy object server.

With this framework, a simple underlying mechanism will suffice; in particular a message passing client-server architecture should be sufficient. The role a process fills is then established by what messages the process responds to and emits at runtime. Note that this means a given binary executable file might be capable of producing processes which act as both servers and tasks (which role for a given process would normally be deduced from command line arguments).

Every process has a name, a typelist, and an address ("ID"). The name is an appellation for the process. It will typically be the same for all processes from a given executable, although it may in principle be picked (or even changed) at run time. The "typelist" is a set of "types" that this process considers itself, and it will respond to broadcasts that correspond to any type in its list. The address is used to direct communications; it is very implementation specific. It is unique for every process.

[25] Almost certainly a thin layer on top of a WWW browser like Mosaic.

The messages themselves consist of a *message type*, and a set of *keyword=value pairs*. The message-type is used to show whether a given message contains task parameters, distributed object method-invocation parameters, a log message, *etc.* The values will consist of at least all the usual scalar types, arrays of scalar types, a "null" (unset) type, as well as sets of keyword=value pairs. The latter makes it a hierarchical structure.

It will take some experimentation to discover when to send data by value (*i.e.*, stuffing the actual values into messages) or by reference (sending a "file name"). Generally, when the receiver might only want an unknown subset of a large dataset or when efficiency is paramount, sending by reference is probably appropriate. In any event, point-to-point communications should probably be enabled before sending extremely large messages.

This design is at least partially chosen to be readily implementable in Glish. It is not, however, a direct correspondence; in particular the notion of broadcasts and a typelist will need to be layered on top of Glish (described below).

Control hub

If the underlying processes merely are asynchronous entities which are capable of exchanging messages with one another, conventions need to be built which define the responsibilities and message protocols that different classes of processes understand.

The architecture is considerably simplified if an overall controlling and organizing process (per-user, per-machine) exists. We call this special process the *control hub*.

While the existence of a control hub is a "single point of failure", the simplifications in design and implementation it affords make it a worthwhile one. Many computational tasks and other tools (particularly GUI display tasks) can operate quite successfully without communicating with other processes. These processes should have a standalone mode where they can run independently.

Aside from the role of the hub as described above, a philosophical point can be made. Experts will write the hub and control its evolution. Applications will often be written by naive programmers. If there is a choice of where to put complexity — in the process or in the hub — it should go in the hub.

It is probable that various kinds of functionality not presently envisioned will be required in the future. Having a control hub which is itself readily programmable will greatly ease the burden of providing additional or different functionality in the future.

The control hub maintains a list of active processes, including their name, typelist, and address. Processes communicate by exchanging messages. Messages may be sent at any time, although whether or not it "interrupts" the recipient process depends on whether that program wishes that sort of interaction. A message may either be sent to a particular ID, or it may be broadcast to a typelist. Messages which aren't received immediately are queued in the hub.[26]

For efficiency, a process may request a direct point-to-point connection to a particular address. Messages to other addresses and broadcasts are not possible until the point-to-point connection is broken. (This might be unnecessarily restrictive.) The sender will probably also block in this circumstance.

The hub keeps a list of all active processes and related information in an active process list. For each process, the hub records at least:

name The name of the process. If the process decides to change its name while running, it must send a message to the hub to register that fact.

typelist The list of types the process considers itself is likewise maintained.

address The address the process is cached to enable the physical communications.

status A process is either *running*, *waiting* for a message, *stopped*, or *connected* directly with another process for communications. Messages for *running* processes are queued in the hub until the process is *waiting*. A *stopped* process must be restarted by the hub.

priority The process may be running at some altered priority (for example, as a "background task").

monitoring information The process may send *status* messages to the hub giving information such as "percent done." This status information might be passed on to monitoring process (*e.g.*, showing an hourglass).

A group of processes may be indicated by either of:

1. An list of processes (*i.e.*, explicit list of addresses).

[26] Perhaps "sender blocks until reception would be a better policy?"

2. Regular expressions which may be applied to the name and/or type-lists.

The active process list can be requested by a process via a message to the hub. The requestor may define a group of processes of interest or it may request the entire list.

Other messages related to control of processes that the hub will respond to include:

Activate The activation message contains the name of the executable, along with any special arguments that the caller knows the executable requires. (The hub will probably have to supply some additional standard arguments.) The hub will respond with the new record in the active process list if the activation succeeds, otherwise it will respond with an error message.

Stop The hub may be told to stop a given process (or group of processes). It responds with a message that indicates success or failure, and the active process list is updated appropriately.

Continue A stopped (paused) process(es) may be told to continue. The process list is updated, and the initiator is notified of success or failure.

Shutdown A process may request that another process (or group of process) shut down "gracefully." The process list is only updated when the process being shutdown confirms that it is (just about) finished. No return message is sent to the initiator; the shutdown process can take a long time depending on what needs to be tidied up.

Kill A process can request that the hub unceremoniously kill (via host operating system commands) a process or group of processes. The active process list is updated, and the initiator is notified of success or failure. (The hub can't normally kill a process owned by another user.)

Priority The hub may be told to set a process, or group of processes, to run at a different priority.

Packages and configuration

To the hub, a package consists of a set of executables (which can be command line scripts as well as compiled binaries) and on-line help files. When a package is added, the hub interrogates each executable (probably by running the executable with special flags) and notes at least the name, typelist, and a human-readable string describing the executable (and probably keywords, categories, *etc.*) If the executable corresponds to a task, its inputs are recorded. If the executable corresponds to a class repository, its meta-information is recorded.

This information is cached by the hub so it doesn't need to regenerate it each time (which might consist of running many executables).

The hub will respond to certain messages to display the list of packages, available binaries, etc. This can be used by humans to browse the online set of available functionality. The hub will also respond to messages telling it to update its package cache (*e.g.*, an executable is added or changed; timestamps can be used to determine which executables and directories need to be examined).

Proxy objects

Until such time as we have language support for making objects "distributed", creating a distributed object will require some binding process. This binding process must not be so painful that it is not carried out for important classes.

There are various reasons why we need distributed objects. Perhaps the most compelling reason is to allow users at the command line to take advantage of functionality in the library. Without access to this functionality, the CLI user would only have limited access to functions compiled into the library. Whether a given piece of library functionality should be bound as a simple computation server or as a distributed object, of course, depends on the complexity of the operations. An FFT can likely be a simple compute server; a Table object should appear in an object server.

A related reason is to allow for introduction of new derived classes at run time without forcing recompilations. Take the case of tables: a lot of effort has been spent to allow users to derive new types of tables (so-called *virtual* tables). Suppose a user creates a new type of table and then wishes to display that table with the *table browser*. If this is to occur without the table browser being at least relinked (and probably slightly modified), a facility like that offered by "distributed objects" is necessary.

Another use is to allow computation to be farmed out onto separate machine(s). For example, cleaning a spectral line cube in parallel, one plane per machine.

To reiterate, there are two main uses for a distributed object:

1. To allow use of objects which are not compiled into a given executable (very-late binding).

2. To run standard objects in a different address-space (possibly on a different machine) for performance or other reasons.

The fact that an object is distributed is an implementation detail to the user of that object, at least after it has been constructed.

To create a distributed object one must:

1. Create a (client-side) stub class through derivation from a base class. Its member functions marshal arguments into message records (and retrieve return codes). Much or most of this communication will be encapsulated into classes.

2. Create server executable which implement the distributed object. In C++ one would normally use an existing class, or write a new class, but note that it would be possible to implement the server in another compiled language (like C or FORTRAN) or even through the use of CLI scripts. All the server needs to do is respond to the messages from the client stub that correspond to the object member functions. Typically a server executable might be responsible for several different types, and a server process may contain many different objects. Note that a given object server might have many different objects inside of it.

3. Register the class and its meta-information (*i.e.*, parent classes) with the control hub. At the minimum, we need to be able to map a "type" into a server executable. We would also like to be able to list the available member functions, return types, *etc.*

9.6.2 Persistent objects

This section discusses persistence in general, as well as catalogs. It has to be realized that the general persistence problem does not have to be solved from the beginning. It will suffice for some time to be merely able to store high-level astronomical objects (image, visibility dataset, *etc.*) as well as tables and groupings (associations) thereof.

Persistence is the ability to reform an object into a running process from data on some storage medium (*i.e.*, normally disk). A somewhat subtle distinction can be made between persistent values[27], where a new object contains the values (*i.e.*, state) of some previous object, and persistent objects[28], where the same object is considered to survive multiple invocations in different processes. Ideally a persistence mechanism is easily extensible to handle new classes. It may well be that practicality requires us to stage the implementation of persistence in two stages; the first of which only handles a few predetermined classes.

A catalog is a structure for organizing collections of persistent objects. It also normally contains additional information so that a user may browse the catalog to select objects to be used in further data processing.

An association is a grouping of related objects. This is somewhat different than merely making persistent an object that has "pointers" to other objects. An association will often be made or broken by users through the CLI. One particular object of an association is normally considered "primary." Hierarchical associations should be supported.

Distributed objects and persistence

As mentioned previously, distributed objects might play a useful role in the solution of a particularly difficult persistence problem. In particular, one wants to use a persistent instance of a class that was not available when the program was compiled and linked.

While the general problem is very difficult, it is considerably simplified if one only needs to be able to use such an object polymorphically through a base class that *was* available when the executable was linked. A (fairly) simple strategy that would suffice is to create an inherited class that talks to some object in a server. In this implementation, if a class wasn't available when a particular executable was created, you can still access it via IPC to some object server (at some loss of efficiency) through the hub.

Another possibility would be to use dynamic linking, but that is less portable.

---

[27] Writing the values contained in an object out in some way that its values can be reconstituted into a similar object. This is essentially "copy" semantics: some object is overwritten with values that came out of some previous object. (AipsIO behaves like this.) `persist >> object` is a reasonable way to think of this.

[28] A PARTICULAR object whose lifetime can exceed that of the process in which it was created. `persist >> object` is a bad way to think of this, you are not changing some (possibly empty) existing object, you are "activating" an already existing object.

Persistence open issues

How do you "navigate" between persistent objects which might be on different machines or have different permissions?

How do we handle versioning?

Concurrency. One thinks of persistent objects living in one process at a time. People are used to, say, having one "writer" of a data-set (*i.e.*, object) but possibly many readers. Is this consistent?

Answers to these questions are likely to be strongly influenced by implementability considerations, *i.e.,* simple.

9.6.3 User Interface

CLI

The command line interpreter has to serve both the modest user who only wants to run pre-written tasks, as well as the user who wants to do powerful interactive manipulations of his data and write new applications in a very high level language.

To be a useful GUI, the language must both have a reasonable syntax, and must allow access to high-level operations. For the latter, having a language with "whole-array" expressions (as in languages like FORTRAN 90 or IDL) is a very good start. To have a yet more powerful CLI requires that the types of domain operations encapsulated within the library be available through the CLI. Thus a mechanism for readily binding new functionality from a library to the CLI is important to allow CLI programmers to be "first-class" programmers, *i.e.*, to give them the same sorts of tools that are available to the programmer of the compiled language. The CLI should in fact be the most flexible possible interface to the libraries of the system.

It is important to note that the CLI is distinct from the control hub. In principle the CLI is readily replaceable (alternatively, a choice of CLI's might be available). Additional CLI's merely have to be able to send and receive standard messages to and from the control hub.

GUI tools

There are a variety of GUI tools that one can identify as being useful to. These include:

- 1-, 2-, and 3-D graphics. (Integrated; or at least must be able to integrate 1- and 2-D.)

- Table browser.
- Online documentation/help and gripe system.
- Parameter editor.
- Data/catalog browser.

There are several different ways that GUI tools or components from GUI tools might be useful:

1. Standalone, integrate applications.
2. To make up new GUI tools from parts of an application, *e.g.*, a raster display might be used both by an image viewer and by a visibility data flagger.
3. Callable from a CLI, *e.g.*, display an array or image that has just been modified by the CLI. (Of course this could be useful from any program, not just the CLI although that might be the most common application.)
4. Directly used as a remote object.

To allow these sorts of operations, the standalone applications should be capable of intercommunicating in the AIPS++ system using the usual communications mechanisms. Additionally, the important GUI classes should be separable from the application in which they are embedded to allow the user to build new tools using those components (an interactive data editor from an image display component and table browser).

9.6.4 Standard servers

There is a variety of additional centralized functionality that the control hub provides access to. This list will undoubtedly increase with time.

Help Asynchronous help messages are sent to the hub, where they may be handled in a variety of ways; normally a help display program (Mosaic under X) will be directed to the appropriate page. Help messages contain the originator, a priority (informational, important, urgent, *etc.*) and a reference into the help database.

Log messages Logging messages will similarly be directed at the hub which in turn may do a variety of things with them (display them, put them in a file, filter on priority, *etc.*).

Monitoring Some tasks may emit various messages that mark their progress (percent done) or can be used diagnostically (current residual image from a Clean deconvolution).

Devices The hub will have a repository of device information, and will probably also control servers which interface to various devices, in particular, tape drives.

9.6.5 Status

Glish has been adopted for the control hub, and sample computation servers and proxy object servers have been implemented. Glish event conventions are still informal. Presently Glish can only communicate with processes which it has started. Partly because of this, the clean separation between the hub and the CLI has not yet been made (this problem may be solved by a non-AIPS++ Glish developer, otherwise we shall have to do it ourselves). Proxy objects have only been written in Glish, not C++, to date.

Glish has been augmented for use as a CLI by adding $N > 1$ dimensional array and complex data types, and by adding command line editing.

Persistent values are only implemented for a few important types. There is no catalog classes yet, we are using the unix file system.

A GUI table browser is available. It does not yet show table keywords, nor does it allow the table values to be modified. Two independent GUI graphics tools are being developed; at least one of them is capable of being "programmed" via Glish events.

Some classes to isolate OS dependencies have been developed. More are required.

9.7 Glossary

Many of the following definitions are taken from [**?**, **?**]. Unfortunately, this glossary is not yet complete. Please forward suggestions for new entries to this author.

"A Matrix" formalism Consider the matrix equation $D = AS$, where D is a vector of measured data, S is a vector representing the sky brightness distribution, and the "A" matrix represents a linear measurement process such as observing total power with a single dish or sampling the Fourier transform of the sky brightness distribution with an interferometer. Such an approach has been taken in [**?**, **?**] in analyzing the *MeasurementModels* of several different types of observations.

Abstract base class An abstract base class is a *class* which has no implementation, only an *interface*. Thus it is used *polymorphically* through a derived class.

AIPS Astronomical Image Processing System; the (still active) predecessor to AIPS++. Also known as Classic AIPS. Described in [**?**]. Written in FORTRAN.

AIPS++ Astronomical Information Processing System: described in this paper. Written in C++.

Application A program directly invoked by end users to perform desired operations.

Application developer A person who writes application programs. This person typically uses C++ *libraries* or writes *CLI* scripts.

apply() some *Telescope Components* contain calibration parameters which must be solved for and then applied to the data. apply() creates a new MeasurementSet from an old one. Usually it will perform an *on-the-fly calibration* to avoid data duplication. apply() is also available through the *TelescopeModel*. It merely calls apply on all its contained TelescopeComponents in turn.

Array An array is a *Lattice* for which all the elements are uniquen and fit into memory. Also, an *aperture synthesis* radio telescope is known as an array, but that usage is avoided in this paper.

Association Often in object oriented systems, one type of object will need to be associated with another object. For example, the *MeasurementSet* needs to know which particular TelescopeModel is describing its state of calibration. See also *Catalog*.

Base classes A *class* from which other classes are derived through *inheritance*.

Baseline For *single dish* the contribution to a spectrum from "uninteresting" sources: the instrument itself, the atmosphere, or even from an unwanted astronomical source.

Big table view of data A view of telescope data in which all the data appears in a single *Table*, rather than being separated into tables which vary at different rates. To prevent unacceptable data bloat, the table requires a sufficiently sophisticated *storage manager*, in particular the *Miriad storage manager*.

Catalog the catalog system deals with keeping track of high level objects for the user such as *MeasurementSets*, *MeasurementModels*, *TelescopeModels*, *etc.* that the user has created to store results in. There will be strong interaction between the catalog system and the user interface. An *ObjectID* is the fundamental identifier of a catalogued, or any *persistent*, object.

Class In C++, a Class is a user-defined type.

Class developer A programmer who implements *Classes.*

Class interface A class may only be accessed through a well defined set of functions. This ensures that the class is manipulated only in predetermined (safe) ways. This situation is different from manipulating a traditional structure or record, which has to rely on conventions for ensuring that it remains valid. See also *encapsulation.*

CLEAN a deconvolution algorithm devised by Jan Högbom for use in radio interferometry [J. A. Högbom, Aperture synthesis with a non-regular distribution of interferometer baselines, *Astron. Astrophys. Suppl. Ser.,* **15** (1974) 417–426]. Denote (the discrete representations of) the dirty map by g and the dirty beam by b. The algorithm iteratively constructs discrete approximants f_n to a solution f of the equation $b*f = g$, starting with an initial approximant $f_0 \equiv 0$. At the nth iteration, one searches for the peak in the residual map $g-b*f_{n-1}$. A δ-function component, centered at the location of the largest residual, and of amplitude μ (the *loop gain*) times the largest residual, is added to f_{n-1} to yield f_n. The search over the residual map is restricted to a region A termed the *clean window.* The iteration terminates with an approximate solution f_N either when N equals some iteration limit N_{max}, or when the peak residual (in absolute value) or the r.m.s. residual decreases to some given level.

To diminish any spurious high spatial frequency features in the solution, f_N is convolved with a narrow elliptical Gaussian function h, termed the *clean beam.* Generally h is chosen by fitting to the central lobe of the dirty beam. Also, one generally adds the final residual map $g - b * f_N$ to the approximate solution $f_N * h$, in order to produce a final result, termed the *clean map,* with a realistic-appearing level of noise.

[This is a description of the Högbom CLEAN algorithm. In practice, a variant of it (Clark, Cotton-Schwab, Steer-Dewdney-Ito) is chosen.]

CLEAN component in the Högbom CLEAN algorithm, a δ-function component which is added to the $(n-1)$st iterate in order to obtain the nth iterate. Its location is the location of the peak residual after the $(n-1)$st iteration, and its amplitude is a fraction μ (the *loop gain*) of the largest residual.

CLI *Command line interpreter.*

Command line interpreter An end user program with which the user can perform interactive calculations (for example, whole-array arithmetic) and start up other *applications*.

Control hub A central process which is a repository for centralized configuration information. It is also responsible for starting and stopping *processes*, and mediating communications among them.

DataManagers map values to Tables. It is the *base class* for various *StorageManagers* which perform actual I/O of data values from a storage device. A DataManager is also used to implement *virtual tables and columns.*

Deconvolution the numerical inversion of a convolution equation, either continuous or discrete, in one or several variables; i.e., the numerical solution (for f) of an equation of the form $f * g = h + noise$, given g and given the right-hand side of the equation. Except in trivial cases, deconvolution is an ill-posed problem: In the absence of constraints or extra side-conditions, and in the case of noiseless data—assuming that some solution exists—there usually will exist many solutions. In the case of noisy data, there usually will exist no exact solution, but a multitude of approximate solutions. In the latter case, if one is not careful in the choice of a numerical method, the computed approximate solution is likely not to have a continuous dependence on the given data. The so-called *regularization method* (*q.v.*) (of which the *maximum entropy method* is a special case) is an effective tool for the deconvolution problem.

Discrete two-dimensional deconvolution is an everyday problem in radio interferometry, owing to the fact that—under certain simplifying assumptions—the so-called *dirty map* is the convolution of the *dirty beam* with the true celestial radio image. In addition to the maximum entropy method, the *CLEAN algorithm* is commonly applied to this problem.

Derived classes A class which is created from a *base class*. A derived class may have more functionality than its base class, particularly if it is an *abstract base class*. It can also modify the meaning of a *member function*, especially useful if the class is to be used *polymorphically*.

Encapsulation Programming errors can be minimized if data can be changed only in a well regulated way. Encapsulation is the process of enforcing these well regulated changes, by only allowing access to the data through a class *interface*. This is also known as data hiding.

FFT a fast algorithm for the computation of the *discrete Fourier transform* (DFT) $y_0, \ldots, y_{n-1}$ of a sequence of n complex numbers $x_0, \ldots, x_{n-1}$,

$$y_k = \sum_{j=0}^{n-1} x_j e^{2\pi ijk/n} \, ,$$

typically requiring only $O(n \log n)$ arithmetic operations — or a multidimensional generalization thereof. By contrast, straightforward, or naïve evaluation of the DFT requires $O(n^2)$ operations.

Filled tables are *Table* objects which stage data values to and from a secondary storage device. See also *Reference tables*, *Virtual tables*, and *Storage managers*.

FITS (Flexible Image Transport System) a magnetic tape data format well-tailored for the transport of image data among observatories. See [D. C. Wells, E. W. Greisen, and R. H. Harten, FITS: A flexible image transport system, *Astron. Astrophys. Suppl. Ser.*, **44** (1981) 363–370].

Glish Glish is a *command line interpreter* whish is loosely based on the S programming language, a language commonly used for statistics.. It offers a convenient syntax for manipulating whole arrays (augmented by the AIPS++ group to be multidimensional), and it also has a very flexible, and hierarchical, record data structure. Besides its data manipulation language, it also implements a "software bus" which allows functionality in *processes* to be readily integrated. The master version of Glish is available from *ftp://ee.lbl.gov/glish*, and the current AIPS++ version can be found at *ftp://aips2.nrao.edu/pub/RELEASED/glish*. (At any particular time, the AIPS++ version may have features which have not yet been merged into the mainline Glish distribution).

Graphical User Interface is a term used for the graphical elements, such as menus and buttons, an *application*, or set of applications, presents to the user.

"Green Bank" model are a set of classes, originally described in [**?**], that describe a software model for calibrating and imaging astronomical data. It features a *MeasurementSet* class, which presents the telescope data, possibly with *on-the-fly calibration* being applied. This calibration is supplied by a *TelescopeModel* which marshals the particular calibrations available in various *TelescopeComponents*. A *MeasurementModel* describes how a particular instrument turns a MeasurementSet into a *SourceModel* or *Image*.

GUI See *Graphical User Interface*.

HasA is a term used when a *class* has another class embedded within it (aggregation), or sometimes when it is associated with it (a reference or "pointer" relationship). Another relationship is *IsA*.

Image a class derived from *Lattice* which contains values of homogenous type arranged in a finite-volume, linear, rectangular, or hyper-rectangular structure. Besides the data values, the image contains an *image coordinate system* to associate its pixel positions with astronomical *Measures*, usually related to position on the sky and doppler velocity. "Arbitrary" information can also be attached through the use of *keywords*, and the Image will normally have a processing *log* attached to it.

Image coordinate system is an engine attached to an *Image* for turning positions within the lattice to *Measure*s, usually related to position on the sky and doppler velocity. An image, for example a spectral line cube, might have more than one measure associated with its coordinate system.

ImageStack A collection of N-dimensional *Images* images which share the same *image coordinate system*, viewed as a Lattice.

Inheritance describes the creation of a new *derived class* from a *base class*.

Interface is the portion of the *class* which may be used externally, it enforces *encapsulation*.

Interferometer

invert() a method of the MeasurementModel is the opposite of predict(). Consider the *A Matrix formalism*: $D = AS$. Invert is the operation which, when applied to D, yields an estimate of S:

$$(A^T A)^{-1} A^T D = S$$

IPosition is an ordered tuple of integers, one per axis, that specifies a particular value in a *Lattice*.

IsA is a term used for a class which is a *derived class* of another. An Array IsA Lattice.

IPC Inter-process communications.

Karma Karma is a programmers toolkit for scientific computing with communications support, event management, data structure manipulation, graphics display and user interface support. Information on Karma may be found at `http://www.atnf.csiro.au/karma`. Karma was developed by R. Gooch. AIPS++ is using Karma for a *StorageManager* and for graphics.

Keyword A named value ("keyword=value"). In AIPS++ the value can normally be a scalar type, *Array* of scalar, or *Table*. The particular types that may be attached to a keyword are class specific.

Lattice is an *abstract base class* which presents an *interface* to a finite-volume, linear, rectangular, or hyper-rectangular structure. The Lattice is fundamentally described by its *shape*.

Log A sequence of informational messages with a priority and a type (both of which may be filtered on). The log is typically attached to both the data being modified and to the user interface (for a history of all operations). Ideally programs may be rerun directly from information contained in the log messagse.

Magic value blanking An invalid value which is marked by placing a special value (typically an IEEE NaN) into it. Unlike a *Mask*, magic value blanking is irreversible.

Mask A collection of Boolean values which are used to mark whether a value is valid or invalid.

Measure a class identifying the value(s), errors(s), and unit(s) for a measurement with transformation properties determined by the ReferenceFrame and Representation. Epoch, duration, direction, and position are examples of different types of Measures.

Measurement model an abstraction of the measurement process with a perfect telescope (ie, with no calibration errors). The MeasurementModel has methods *predict()*, *invert()*. Specific MeasurementModels will include the Interferometer MeasurementModel, the SingleDish MeasurementModel, the MosaicingMeasurementModel, and the NonCoplanarBaselineMeasurementModel. A *MeasurementSet* will have an associated MeasurementModel. When a MS is passed to a MM, the MM will convert the MS to the data view which it requires.

Measurement set All of the data produced by a Telescope which the user cares to keep, including both astronomical data, instrumental data, and monitor data, is the body of the MeasurementSet. A measurement set is *derived* from a *Table*, and presents a *"big table" view* of data. The MS has little intelligence, but it has an associated *Telescope Model* which is able to calibrate data *on the fly* in the MeasurementSet, and an associated *MeasurementModel* which is able to convert the data into a *Source model* or *Image* or able to convert a model brightness distribution into a model MeasurementSet.

Member function a function that belongs to a *class* and is able to manipulate class data. Only member functions are allowed to directly manipulate class data[29]. This is the mechanism by which *encapsulation* is enforced. The set of public member functions of a class make up its *interface*.

Message The atomic unit of information exchanged between two *processes* via *IPC*.

Method see *member function*.

Miriad Multichannel Image Reconstruction, Image Analysis and Display. A software package for the reduction of radio interferometric data. The ideas from its data system are being used in the *Miriad storage manager*.

[29] If the class is designed properly.

Miriad storage manager A *Storage manager* which only stores values when they change. This storage manager is thus appropriate for *Table* columns which vary slowly, which is important for the *"Big Table* view of data. It is based on ideas in the *Miriad* software system.

ObjectID A globally unique identifier for objects. It is presently a concatenation of the host address, process ID, time, and a per-process unique sequence number. This heuristic suffices for systems that do not "roll over" their process ID numbers in a second or less. It is important for *persistence.*

"On-the-fly" calibration or on-demand calibration occurs when computations are interposed between the user and the raw data. This is accomplished through the use of *virtual Tables.*

"On-the-fly" mapping a form of single dish observing and subsequent imaging in which the antenna is slewed continuously across the sky and data are recorded on a timescale which is short compared to the time the antenna moves an entire beamwidth. The pointing position, possibly including pointing error estimates, is recorded with the data. Proper imaging requires use of the pointing data.

Operator overloading The C++ term for user-defined functions which use the same characters as built-in operators. For example, with operator overloading one can type `a = b + c;` where a, b, and c are *Class* objects, such as an *Array.*

PagedArray is a *Lattice* for arrays which are both *persistent* and possibly larger than *memory.* The PagedArray uses the *Table* for its I/O. The PagedArray is used by the *Image.*

Persistence Persistence is the ability to reform an object into a running process from data on some storage medium (*i.e.*, normally disk).

Pixel coordinates are values related to the position of the pixel in the Lattice, *i.e.*, these are coordinates in the frame set by the axes of the Lattice without any astronomical meaning.

Polymorphism allows classes with a sufficiently similar interface to substitute for one another. This allows, for example, a new kind of clean deconvolution to be introduced without having to change any client code that needs such deconvolution.

Point spread function the response of a system or an instrument to an impulsive, or point source, input.

predict() is used by the *MeasurementModel* to create a new *MeasurementSet* using a *SourceModel* and another MeasurementSet as a paraform of where to sample the data.

Primary beam For a *single dish*, the *point spread function* of the image. For an interferometer, the average of the primary beams of all the antennas in the array.

Proxy object A proxy object is a stub that is used to manipulate the real object which is in somewhere else, probably in an object *server*. The proxy object forwards its *member function* invocations to the server, and retrieves the results and presents them to the user of the proxy object. This is a particularly valuable technique for allowing *CLI* users to manipulate objects created in another language, *i.e.*, C++.

PSF See *Point spread function*.

Section A regular portion of a *Lattice* defined by a start position, and end position, and a constant increment for each axis. Often a *view* of another Lattice.

Server A *process* that performs calculations at the behest of another process.

Shape An ordered tuple of integers that defines the length of each axis in a *Lattice*.

Single dish A radio telescope that consists of a single antenna. Traditionally a single dish telescope has only had a single receiver, but multi-beam receivers are becoming more common.

solve() some Telescope Components contain calibration parameters which must be solved for and then applied to the data. These TCs will have a "solve()" method, an "apply()" method, and a Table to store the parameters in.

Source Model is an estimated sky intensity as a function of position, time, polarization, and frequency. Unlike an *Image*, it is not necessarily regularly sampled; it will often be parameterized.

Spectrum set A *view of data* for image plane spectra. Single dish equivalent of *VisSet*.

Stokes parameters the four coordinates relative to a particular basis for the representation of the polarization state of an electromagnetic wave propagating through space.

StorageManager is a *DataManager* which physically store and retrieves values from a storage device. Storage managers with several different properties are available; a *Karma* storage manager, a *Miriad storage manager*, and a storage manager based on the AipsIO class.

Table A table consists of a header, and a main data table. The main data table consists of a number of rows and columns. A value is stored at the intersection of each row and column. All values in a column must be of the same type. The header consists of sets of *keywords*: one set for the whole table, and (potentially) one set per column.

Telescope model class which describes the state of the telescope during the collection of data in a MeasurementSet. It consists of a number of TelescopeComponents, each containing some telescope state information and calibration parameters (Platform, Receptor, Environment, etc). Its basically a collection of TelescopeCompoenents. The MeasurementSet will have an associated TelescopeModel.

Telescope component A *TelescopeModel* consists of one or more TelescopeComponents. Each TelescopeComponent serves to model some feature of the function of the telescope. For example, TelGeometry deals with the geometrical aspects of the Telescope in relation to the earth and sky and knows where the antennas are located and how to calculate (u,v,w), elevation, parallactic angle, etc. ReceptorGains deals with antenna based complex gains and knows how to solve for and apply them to the data. PrimaryBeam stores the model of the primary beam.

User can refer to either end users (*i.e.*, astronomers), or to users of a *class* (*i.e.* programmer).

View of data occurs when an object has no data of its own, it references data in some underlying object. This means that modifying the data through the view will also modify it in it's underlying object. A C++ example of this is `matrix.diagonal() = 1;`. The result of the diagonal() member function is to return a Vector which is a view of a portion of the underlying matrix (its diagonal). This view object is set to unity, which changes the underlying Matrix.

Virtual tables and columns Tables and columns in which some of the data are computed (or come from some other source) are known as "virtual" tables or "virtual" columns.

VisSet A *view of data* for interferometric data that presents a sufficient interface for imaging applications. For example, it would have time, anntennas, pointing centers, weights and the actual visibilities. A VisSet is normally attached to a *MeasurementSet*.

9.8 Glish Programming Idioms

Glish is a procedural language with whole-array[30] manipulations loosely based on the "S" language. Glish also has powerful facilities for asynchronously controlling and communicating with other processes (either other Glish scripts or compiled programs). This document assumes a reasonable familiarity with Glish; an excellent user manual comes with the Glish distribution.[31]

The AIPS++ group is attempting to use Glish as an end-user command line interpreter (CLI). Since Glish was originally designed for a very different purpose (instrument control) some work and experimentation will be required for Glish to be truly suitable. Much of the work will be "environmental": command line editing[32], on-line help, and the like. Some of this work can, however, can be done by establishing programming *conventions* within Glish so the end user sees a consistent and convenient way of interacting with Glish values and functions.

9.8.1 Object Oriented Glish

The desire to have object-oriented programming features in Glish originally stemmed from the desire to allow the user to have access to overloaded functions. That is, some of the entities that the user would be manipulating would logically be more complex than Glish values, for example AIPS++

[30] Glish originally only had 1-D arrays; it has recently been augmented to handle multidimensional arrays by Darrell Schiebel of the NRAO AIPS++ group.

[31] Available via anonymous ftp from *ee.lbl.gov* in the *glish* subdirectory. The principal author of Glish is Vern Paxson (*vern@ee.lbl.gov*).

[32] Also recently added by the AIPS++ group.

Tables, images, *etc.* There are many generic functions the user might want to call on these "objects" (*display, help, history,* and the like).

Neither of the obvious alternatives:

1. Encode the type name in the function: *display_table, display_column, display_row, display_image, etc.*

2. Since these types will undoubtedly be represented inside Glish as Glish records, we could case on a type field inside the various functions.

were very attractive. The first solution requires the user to remember a lot of logically identical functions, the second solution is fragile since it requires that every function be looked at when a new "type" is added.

Instead, polymorphic methods seemed to solve the problem readily. Additionally, the OO convention diminishes the "impedance mismatch" between the C++ libraries and the CLI user, allowing the proxy objects to more closely follow the C++ versions.

Obviously an idea solution would be for Glish to provide built-in support, , i.e. evolve towards *S+* from S. We shall investigate the costs and benefits of doing so.

9.8.2 Asynchronous Glish

Many user data reduction and analysis operations take sufficiently long that a user would not necessarily want to wait for them to finish. AIPS (Classic) for instance has "verbs" which are relatively low-cost computations which always run synchronously, and "tasks" which are inherently asynchronous, although the user may specify that "tasks" are to operate synchronously.

Glish has excellent facilities for asynchronous programming. The full set of facilities — *whenever* statements, *subsequences*, *etc.,* — is far richer than the typical user will to want to deal with.

We can instead establish a convention where the user specifies during the function invocation whether or not the execution takes place asynchronously. In the latter case, the user also needs to know where to find the result, and to know when the computation has finished.

```
x := fft(y)*z;        # fft defaults to synchronous; just use it as any
                      # other function.

fft(y, async=T);      # Run fft asynchronously. Eventually the
... other work ...    # user gets notified that it has finished.
x := results.fft*z;   # Get the result and compute with it
```

While in the above example "results.fft" is an array, it could clearly just as easily be a Glish record for computations which produce multiple output values.

A flaw in the previous example is that since there is only one place to look for the result, only one asynchronous "fft" may take place at a time, since there's only one place to look for it. This may not only be inconvenient, it would disallow the user to take advantage of coarse parallelism by spreading computations out over multiple cpu's of a single machine, or multiple hosts on a network.

For example suppose the user is on a four processor machine and wants to fft a cube a plane at a time, spread over all four processors.

```
nplanes := cube::shape[3];
for (i in 1:nplanes) {
    request[i] := fft(cube[,,i], async=T, client_number=i%4+1);
}
wait_for(requests);
for (i in 1:nplanes) {
    cube[,,i] := result[request[i]];
}
```

A slightly more complicated example would have the client adaptively fill up the fastest CPU's in case some are faster than others (*e.g.*, sharing with another user). Note that this example assumes that the underlying client (*i.e.*, compiled binary) is single threaded; that is, we need one client per CPU. If this wasn't the case, we could just leave out the client number.

A similar example would allow the computation to be spread over a network of machines by specifying *host = hostname*.

Here we have made the user remember a request number, where before they only had to remember "result.fft." The increase in generality appears to me to be worth the price.

9.8.3 Status

Object-oriented Glish works and is useful, although more language support would make it more convenient for "class" implementers.

The asynchronous idiom presented here has not been implemented, although it should be straightforward

9.9 References

Part II

AIPS++ Packages Reference

Aips++ is divided into a series of packages. They are currently

- aips
- dish
- nrao
- synthesis
- trial
- vlbi

Each package consists of a series of modules which are further broken into classes.

Index