An Overview of AIPS++ Design

The State of the Project Classes at the End of April 1993
AIPS++ Implementation Memo 109

14 November 1993

R. M. Hjellming and B. E. Glendenning

1 Purpose of This Overview

1.1 Design Approach and Status

This document is a summary of the design of classes for AIPS++ as they are at the end of April 1993. Since the prime goal of AIPS++ is the development of an extensive library of classes for astronomical data processing, this summarizes the state of the most important part of the project. This document is an updating or supplement to the Software Design sectioni of the Project Bookshelf. While some of the classes mentioned or described here, particularly the infrastructure, math, and table classes, have been implemented, for most classes the header (declaration) files and associated code have not been written. Thus in many cases the prototyping may result in minor or major changes in class design.

In this document we follow the AIPS++ Project requirement that design be described in terms of the Object, Functional, and Dynamical Models of the Object Modeling Technigue of Rumbaugh et. al. (1991). The reader will note that, while there are high to middle-level Object Model diagrams for all the areas discussed, there are still deficiencies in the design description: few diagrams exist at the lower level corresponding to the essentials of C++ header files; only a few Functional Models have yet been developed; and there are no Dynamical Models. The major exceptions involve the Bottomly (1993) document on transforming telescope data sources, like magnetic tapes, into telescope data associations, and the Measure-ReferenceFrame class system.

Thus the reader must understand that much of this design has not been tested by either prototyping or the level of design required by the Object Modeling Technique.

During the period September 1992 to April 1993 there were piecemeal developments in the design, but these changes did not constitute a sufficient degree of completeness to make it easy to update the Design section of the Project Bookshelf (nee Project Book). Because of the necessity of producing a design summary for the Design Review and Prototyping Workshop in Socorro, NM, April 26 - June 4, 1993, we are writing this draft Design Summary for the Workshop. A version will be put into the AIPS++ Implementation Memo series, and the document will be updated and evolved during the Workshop, resulting in another Design Summary in June 1993. After this the Design Section of the Project Bookshelf will be updated to represent, once again, the "current" design of the Project.

1.2 The Point of View Taken for Class Development

1.2.1 Radiation Measurement

Classes designed in an object-oriented software system are strongly dependent upon the point of view taken in the design of these classes. Since January 1992 (Hjellming and Glenndening 1992), and expressed in fairly complete form in the report of the February 1992 Workshop in Green Bank (Shone and Cornwell 1992), the point of view of class design in AIPS++ has emphasized the measurements of radiation made by telescopes (particularly radio telescopes), the essential properties of these telescopes, and the reconstruction of radiation measurements of the sky in the form of intensities, spectra, images, etc.

1.2.2 Mathematical Classes

The most valuable classes are those that represent such powerful abstractions that they can be used for many purposes in a wide range of applications. It does not take much cogitation to conclude that classes accomplishing mathematical defined operations on numbers and arrays of numbers should be very important. Since September 1992 the AIPS++ Project has emphasized the development of such mathematical classes beginning with the very general Array class from which Vector, Matrix, Cube, etc. classes can be derived with their own uniquely valid operations. These can then be used to do higher level mathematical classes. Linear algebra classes are obvious derivatives from the Vector and Matrix classes, and as of the end of April most of these have been implemented for real and complex arrays. The GridTool and FFTServer classes are amongst the next level of obvious classes in which un-gridded data can be gridded into regular arrays, regular arrays can be interpolated back to un-gridded data coordinates, and regular arrays of different dimensionality can be Fourier transformed and reverse Fourier transformed.

In addition to their general usefulness, some of these math classes will appear in design diagrams as the most general case from which conceptually important classes are derived. The most obvious examples of this are in the area of array objects that constitute spectra, images, etc.

1.2.3 Table Classes

While a reasonably sophisticated and efficient data base system with a programmer-friendly data base manager is essential for AIPS++, it was decided in September 1992, at the same time the emphasis on mathematical class development was begun, that classes based upon tables of data would be sufficient for initial prototyping. As this is written at the end of April 1993 the system of classes that constitute the initial Table system have been prototyped. For the purposes of this design document, many classes use the inherited properties of the Table class, particularly in objects like tables of measurement data, series with uneven coordinate intervals, and images with uneven coordinate intervals.

1.2.4 Comprehensive List of Major Classes

In Appendix 1 there is a list of the major classes in the current AIPS++ design. The classes are listed in alphabetical order with short definitions and, where appropriate, a description of its relationship to other classes. This is a class-oriented "data dictionary", largely at the level used at the design stage of Chapter 8 of Rumbaugh et. al. (1991). In this data dictionary we use the convention followed in all design diagrams in this document whereby class names are distinguished, by convention, from class features (attributes and operations) by making the first letter of the names upper and lower case, respectively.

2 Major Subsystems

2.1 Major Subsystem Components

Figure 1 is a high level Object Model diagram for a number of the important classes in the current AIPS++ design. It corresponds to the upper level of design diagram emphasized in Chapter 8 of Rumbaugh et. al. (1991). It reflects the initial priority given to telescope data handling, including calibration and editing.

In Figure 1 the most important classes are: DataSource, MeasurementSet, TelescopeModel, TDAViewer, MeasurementModel, and MeasurementEquation. The DataSource class contains the methods for taking telescope measurement data and associated information, all of which are considered to be measurements, from a variety of sources such as telescope data tapes (TelDataMedia in general), real time data acquisition systems (RealTimeSystem), and simulation programs (TelSystemSimulator), and putting them in a TelescopeDataAssociation - which is the Associator class for an aggregation of tables containing measurements in a MeasurementSet and the tables modeling various aspects of behavior of the telescope (or telescope array) in TelescopeModel. Information about the telescope during the measurement process, calibration tables based upon one or more of a variety of models for the telescope's measurement equation (MeasurementEquation), and data quality measures (flags) associated with the measurements by either the data source or the user are included in the information maintained in TelescopeModel. The TelescopeModel::solve operation is the method whereby data with assumed models of the radiation distribution (RadiationDistModel) on the sky are used in connection with the appropriate equations derived from MeasurementEquation to determine the calibration parameters of the particular components of TelescopeModel. The TelescopeModel::interpolate operation determines, and writes calibration information into, TelescopeModel tables for measurement coordinates (like time and possibly frequency) which are different from those for which that are obtained from the TM.solve operation. Finally, the TM.apply operation is the method whereby selected measurements are extracted in calibrated as opposed to uncalibrated form, that is "inverse" measurement equations and "data" in calibration tables are used to apply calibration to the measurement data.

TelescopeDataAssociation not only maintains the necessary association information between objects in MeasurementSet (MS) and TelescopeModel (TM), it has TDA::readDataSource methods whereby DataSource for particular telescopes read data and write it into MS and TM tables. In Figure 1 each DataSource object can be optionally associated with an ObservingLog which is summary or index of measurements in that object. ObservingLog is closely related to the optionally available ObservingSchedule which, in some telescope systems, is used to control the telescope observations that produced the measurements in DataSource.

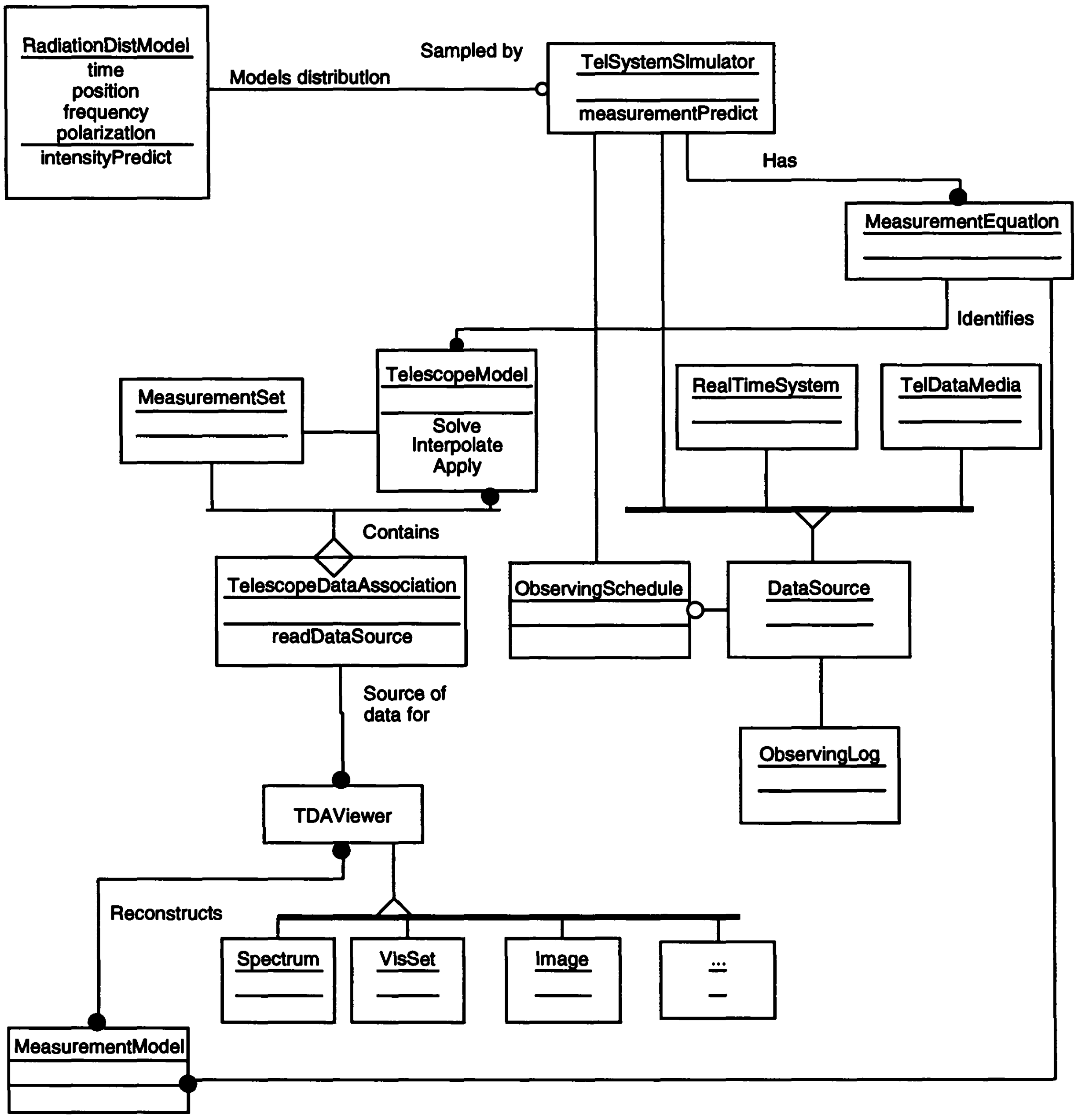


Figure 1 — Major class "subsystems"

The design of DataSource and its mapping into a TelescopeDataAssociation has been carried to the most detailed level yet in AIPS++ design, short of prototype coding. The status of this design, for both general cases and the case of VLA interferometric data, is described in a document by Bottomly (1993). Therefore we will not discuss it further in this document.

The RadiationDistModel, TelSystemSimulator, and ObservingSchedule classes are all associated with the capability to compute models of measurements. These may be used as part of the calibration or self-calibration process, or they may be used during analysis where physical source models are used to generate measurments, spectra, images, etc., for comparison with equivalents determined by measurement data.

The TDAViewer class may turn out to be just an abstract label, rather than even a base class, but it is a major conceptual development that was discussed for months early in the AIPS++ analysis and design process; however, it was clearly defined under the name ApplicationDataObject only during the mini-workshop in Charlottesville and Green Bank in November 1992. Discussion has indicated that this name should be changed, so in this document we are now calling it TDAViewer, indicating that it allows views of the data in a TelescopeDataAssociation, or an ImageDataAssociation, as seen in an application program. Selection criteria, including those based upon data editing and/or application of calibration expressed in user-oriented terms (rather than data base specifiers) are used to create views of data for processing. These views can be synchronized with the data in the TDA or IDA, that is, a change of value in the TDAViewer results in a simultaneous change in value in the DataAssociation; or in an TDAViewer can, by averaging or some other operation, produce new TFAViewer objects that can be optionally written into new tables in the TDA.

Two of the important TDAViewers related to a TDA are the Spectrum and VisSet classes, both of which can be described as major infrastructure classes for AIPS++ data processing, because of large number of other classes using data of these types will be built inheriting the properties of Spectrum and VisSet. These will be discussed in more detail later, particularly the VisSet class. The Spectrum class, for example, will have both the methods for selecting and extracting spectra from measurements in a TDA and the methods for doing obvious things with these spectra in addition to application of calibration. This class can be used for total power observations or interferometric observations in their respective TDAs. VisSet is the major class for building all visibility data sets as used in particular applications.

Figure 1, at the moment, does not show Images as parts of an ImageDataAssociation, partly because they can in principle be just added components in a TelescopeDataAssociation. The choice between these forms of organization have not been made, and it is likely that both will be useful.

The MeasurementModel class was called the ImagingModel class in early discussions, but attempts to take a unified approach towards processing both single dish and interferometric data led to using this name for the class which uses application data objects of all kinds (VisSet, Image, etc.) to produce different aspects or forms of measurements. Principle solution imaging, CLEAN deconvolution, MEM deconvolution, self-calibration, spectral component fitting, solution of calibration equations, etc., are specific cases of the general MeasurementModel. The utility of a single class for this wide range of "things" is one of the results of the conceptual analysis of imaging and calibration in AIPS++ by Cornwell (1992a and 1992b, Implementation Notes 147 and 148) . It will be discussed in more detail later in this document. If validated by prototyping, it is possibly the single most important base class in AIPS++.

2.2 An Operator Notation for the Major Processes or Classes

Up until Cornwell (1992a, 1992b) expressed imaging and calibration in a simple operator notation based upon concepts from linear algebra, all the the ideas about MeasurementSet, Images, MeasurementModel, TelescopeModel, etc., were discussed in a highly unstructured form. The operator notation he proposed has been adopted as an effective means of describing essential operations, and turns out to have a close relationship to the the major classes discussed in the previous section. Let us summarize this formulation using slightly different symbols and name descriptors that reflect the current use in design discussions.

Radiation distributions on the sky are represented by S (replacing the I used by Cornwell (1992a) to avoid confusion with the identity operator). Measurement data produced by telescopes are identified with D_{MS}, or a MeasurementSet. The MeasurementEquation for a telescope or telescope system is identified with an operator A, so the basic equation whereby Telescopes produce data is

$$A\ S = D_{\mathrm{MS}}$$

The sky distribution S may be viewed as the projection of the true sky intensity distribution S_{true} on the telescope by using a projection operator $P(S_{\mathrm{true}})$ as argued for by Redman (1993). For present purposes we will let S have a number of identities ranging from the true sky distribution (with implied dimension including all polarization) to model distributions S_{model}. The $AS = D_{\mathrm{MS}}$ equation turns into a form closely matching the major classes of the AIPS++ design if one separates A into a MeasurementModel part and a TelescopeModel part, so that

$$A_{\mathrm{TM}} A_{\mathrm{MM}}\ S = D_{\mathrm{MS}}.$$

This formulation now shows the essential role of MeasurementSet (MS), TelescopeModel(TM), and MeasurementModel(MM) in relating real or model sky distributions to the production of measurements such as visibilities, total powers, spectra, etc. The next step is to recognize that the above deal only with uncalibrated measurements. The process of determining calibration and turning measurements into calibrated form (CalMS) can be written as

$$\mathrm{inverse}(A_{\mathrm{TM}}) A_{\mathrm{TM}} A_{\mathrm{MM}}\ S = D_{\mathrm{CalMS}} = \mathrm{inverse}(A_{\mathrm{TM}}) D_{\mathrm{MS}}$$

ignoring for now such details such as whether inverse(A_{TM}) is a true linear inverse operator, or is derived from solution of non-linear equations (the general case). By definition inverse$(A_{\mathrm{TM}})A_{\mathrm{TM}} = 1$ so with calibrated data the reconstruction process for S is based upon

$$A_{\mathrm{mm}}\ S = D_{\mathrm{CalMS}}.$$

Cornwell (1992a) showed that most telescopes have measurement equations represented by simple measurement models, and that most of single dish and interferometric reconstruction of measurements (images in most cases) fits into the same approach. As we will describe in more detail later, there is a base class we call MeasurementModel that has exactly the methods his memo required for a generalized approach to measurement (image) reconstruction.

In order to provide a notational basis for more important details, and the classes that embody them, let us note that, in general, a measurement set D_{MS} with consiste of a data array as a function of a data coordinate vector $\mathbf{u}$ (embodying time, frequency, u-v-w for visibilities, etc.). Sky radiation distributions, real or modeled, have radiation coordinates $\mathbf{x}$ (embodying space-time "position", frequency). We will use these notational distinctions in our more detailed discussion of MeasurementModel classes.

These $\mathbf{u}$ and $\mathbf{x}$ coordinates will have representations (spherical polar, cartesian, plane-projected, etc.) within some reference frame (solar system barycentric, geocentric, topocentric, celestial sphere tangent plane, proper, ...). This characteristic of representations within some reference frame is shared by radiation measures like $(IQUV)$ that can be transformed to $(I_{\mathrm{RR}}, I_{\mathrm{LL}}, I_{\mathrm{RL}}, I_{\mathrm{LR}})$ or $(I_{\mathrm{XX}}, I_{\mathrm{YY}}, I_{\mathrm{XY}}, I_{\mathrm{YX}})$, or Poincare sphere representations of polarization. Thus we have introduced into AIPS++ design a Measure class that contains the essential properties of measurements of all types, including units, but which most importanly has all transformation properties embodied in associated ReferenceFrame and Representation classes. This can be described with a notation whereby $T_{\mathrm{rf,r}}(m) = m'$, where the rf,r denotes potential reference frame and/or representation changes in the same measurement, and m can be a measure of sky intensity, sky coordinates, space-time coordinates, baseline coordinates, or any quantities usefully associated with unit, reference frame, and/or representation changes.

Before continuing with more detailed discussions of higher to mid-level classes, let us discuss some general but basic classes that have been developed for AIPS++.

3 Infrastructure Classes

The classes discussed in this chapter have been implemented. However, not all implemented classes are discussed here. Classes which are mostly of use in building other classes are generally not mentioned, nor are functions or classes which make up the array or table system (these do, however, have some discussion in later chapters).

3.1 Containers

A container is a class which can contain objects of another type. The containers we have implemented do this with templates.

`Array`	The array classes can be considered to be a type of container in which objects are accessed by a (generally multidimensional) index.
`Block`	A Block is a very simple one-dimensional array container. It will most likely be used as a building block rather than as a class for application programmers.
`Map`	A Map, also known as an "associative array" or "dictionary" is used to store key/value pairs where key and value can be of different (arbitrary) types. A simple example of a Map would be using a string as a key for a catalog entry. Note that Map is an abstract base class; there are various implementations (e.g. linked list).
`Slist`	A Slist is a singly linked list. Multiple cursors (iterators) may be attached to the list.
`Stack`	A stack is a "last in first out" data structure. Useful for, e.g., evaluating expressions.

3.2 I/O

The major classes which are still required for I/O are those that the data base will need, and those for handling FITS files. Both of these are in progress. The classes which are now available include:

`AipsIO`	AipsIO is a straightforward persistence mechanism for storing binary data. It is generally available for most AIPS++ classes, and will generally work properly for the container classes with any template argument. It uses the familiar `operator<<` and `operator>>` functions.
`Viff`	This class is used to write up to 3-dimensional arrays, with optional location data, into the "Viff" format which is used by Khoros/Cantata.

3.3 Errors and Exceptions

We have implemented an exception handling mechanism that acts very much like the exception handling mechanism that has been proposed for C++. The major features it offers are:

1. An exception may be thrown from "deep" in a program and caught at a "higher" point without any intervening code being written.
2. The destructors for objects which go out of scope are called (i.e., it cleans up after itself).
3. An exception may be of essentially arbitrary type, and hence it can be used to carry any desired information from the point of error to where it is handled.
4. You can not only catch a specified type, but you can also catch any type derived from it.

The major classes are:

`Cleanup`	In the current implementation, all classes whose resources you want to be freed up during an exception should be derived from this class.
`AipsError`	This class is the root of the normal exception class hierarchy (for example, `ArrayError` is derived from it). There is a fairly rich hierarchy of error classes.
`Assert`	[Not really a class]. This is like the normal "C" assert mechanism, only it uses exceptions instead.

3.4 Miscellaneous

`Input`	This class (and its associated `Param` class are used to interpret '`keyword=input`' or '`-keyword value`' pairs. It also allows the keywords to have defaults and help strings, and to output a "pane" file for interfacing a program to Khoros.
`Plot`	This is used to print a very simple scatter-plot; generally Khoros should be used for plotting instead.
`Timer`	Used as a "stopwatch" for timing (real and cpu) programs or parts of programs.

4 Mathematical Array and Array-based Classes

The array classes discussed in this chapter have been implemented.

4.1 Fundamental Classes and Functions

The fundamental array classes are an n-Dimensional **Array** class as well as specializations (**Vector**, **Matrix**, **Cube**) which are inherited from it, as shown in Figure 2.

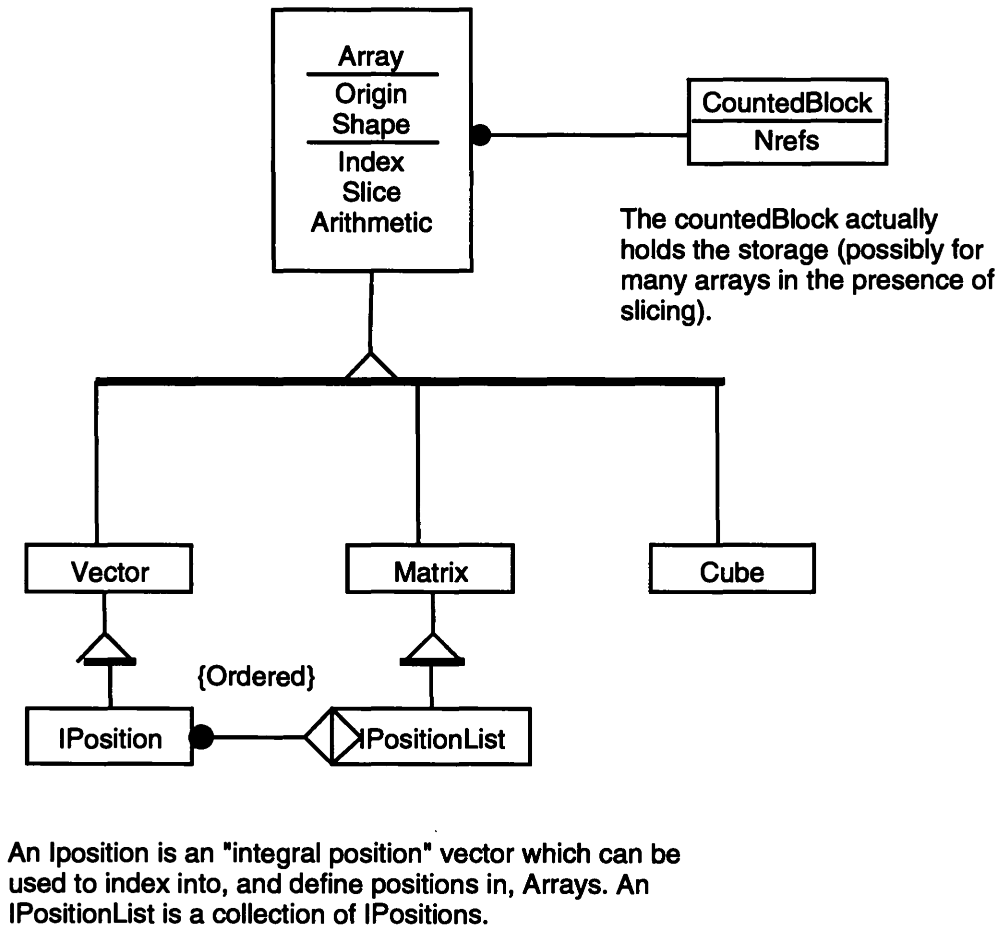

Array
Origin
Shape
Index
Slice
Arithmetic
CountedBlock
Nrefs
The countedBlock actually holds the storage (possibly for many arrays in the presence of slicing).
Vector
Matrix
Cube
{Ordered}
IPosition
IPositionList
An Iposition is an "integral position" vector which can be used to index into, and define positions in, Arrays. An IPositionList is a collection of IPositions.

Figure 2 — Object model diagram for Array-based classes.

The specializations allow more convenient (and more efficient) access to pixels and regions in arrays. Without them, one would always have to access these using `IPosition` objects, which are vectors whose elements are integers.

The array classes have a variable `origin` (defined as the value of the location of the first pixel), and they are fully templated. The arrays also have a `shape` — i.e. an `IPosition` vector which gives the length of each axis.

Because the array classes refer to a common `Block` of (reference counted) memory, it is possible for several arrays to have a common "view" of some pixels. This is mainly used to implement array sections ("slices").

All the normal arithmetic functions are defined, including transcendental functions. These operate element by element[1]. There are also miscellaneous functions, for example finding the minimum and maximum values in an array.

Functions are provided to `resize` and reshape arrays in various ways.

Various comparison (`<=` etc.) operators are available. Currently the yield a `Bool` result, e.g. `a <= b` is true if every element of "a" is less than or equal to every element of b. Functionality to return a "mask" where the comparison is true will be provided shortly.

4.2 Iteration

While the multidimensional `Array` classes are very powerful, it is in general not very convenient or efficient to deal with an array of arbitrary dimensionality. Thus the concept of an `ArrayIterator` has been introduced; the object model for the relationship of this class to other classes is shown in Figure 3.

[1] `a=b*c` is not a matrix multiply even if a,b,c are matrices

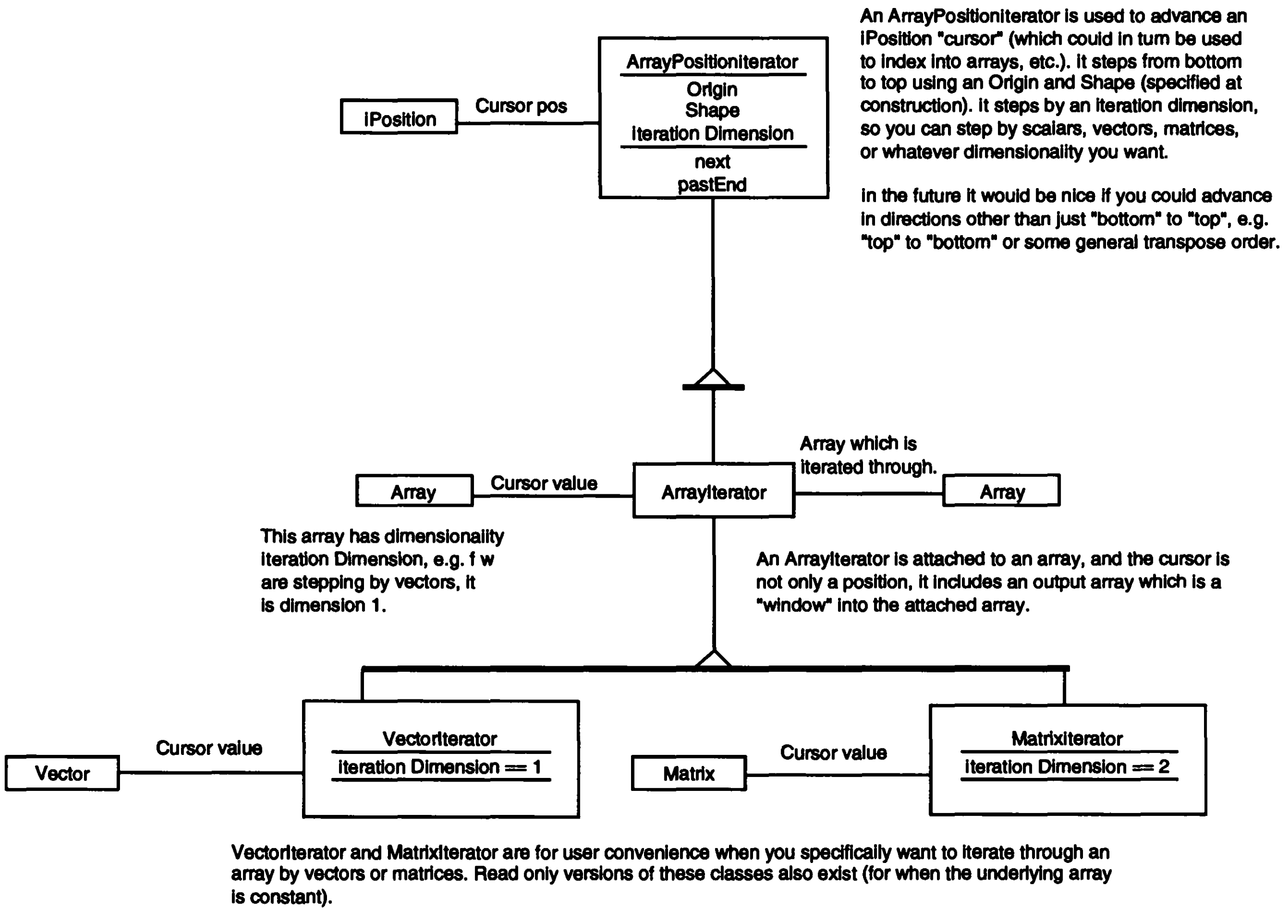

An ArrayPositionIterator is used to advance an IPosition "cursor" (which could in turn be used to index into arrays, etc.). It steps from bottom to top using an Origin and Shape (specified at construction). It steps by an iteration dimension, so you can step by scalars, vectors, matrices, or whatever dimensionality you want.
In the future it would be nice if you could advance in directions other than just "bottom" to "top", e.g. "top" to "bottom" or some general transpose order.
ArrayPositionIterator
Origin
Shape
Iteration Dimension
next
pastEnd
IPosition
Cursor pos
Array
Cursor value
ArrayIterator
Array which is iterated through.
Array
This array has dimensionality Iteration Dimension, e.g. f w are stepping by vectors, it is dimension 1.
An ArrayIterator is attached to an array, and the cursor is not only a position, it includes an output array which is a "window" into the attached array.
Vector
Cursor value
VectorIterator
Iteration Dimension == 1
Matrix
Cursor value
MatrixIterator
Iteration Dimension == 2
VectorIterator and MatrixIterator are for user convenience when you specifically want to iterate through an array by vectors or matrices. Read only versions of these classes also exist (for when the underlying array is constant).

Figure 3 — Object Model diagram for the Array Iterator classes.

An **ArrayPositionIterator** is an **IPosition** cursor which can be iterated through the array. While be default it steps through every position in the array, it can also step through the arrays by vectors, matrices, or any other dimension array (that is less than the dimensionality of the array being stepped through). While the space the cursor steps through is defined by some array (in particular, but it's **shape** and origin). This cursor can, in turn be used to index into one or more **Array** objects.

In contrast, an **ArrayIterator** has a cursor which is an actual view of a part of an array, and hence changing the cursor changes the actual values in the array. An **ArrayIterator** may be of any dimension (less than the array being iterated through!) so it is anticipated that in practice people will use either the **VectorIterator** or **MatrixIterator** specializations.

In the current implementation the iterators step through the array from beginning to end in the obvious way. In the future it will be very useful to have iterators which can step through the array in some transpose order, and also which can be told to "skip".

4.3 Other

Aside from the basic array classes, there are some higher level tools. This list should grow rapidly with time (for example, more fitting of various kinds).

`Matrix Math`
: LU-decomposition, matrix and cross products, determinants, triangularization, least squares fitting to linear equations, etc.

`FFTTool`
: Forwared and reverse Fourier transforms.

`GridTool`
: Grids irregularly sampled points onto an n-dimensional grid, and de-grids data by interpolation from a regular n-dimensional grid to irregular axis coordinates.

`Random numbers`
: A rich set of random number distributions are available.

5 Table-based Data Base System

5.1 Tables

The material in this section is largely excerpted from the Database reference manual.

The tables classes discussed in this section have a prototype implementation. At the moment the classes do not have an intelligent storage manager (cf. the next section for a preliminary discussion of that).

The traditional relational database model has two features that limit its applicability to scientific data. First, an item of data in a column of a table must be atomic; it must have no internal structure. A consequence of this restriction is that relational databases are unable to deal with arrays of data items. Second, an item of data in a column of a table must not have any direct or implied linkages to other items of data or data aggregates. This restriction makes it difficult to model complex relationships between collections of data. While these restrictions may make it easy to define a mathematically complete set of data manipulation operations, they are simply intolerable in a scientific data-handling context. Multi-dimensional arrays are frequently the most natural modes in which to discuss and think about scientific data. In addition, scientific data often requires complex calibration operations that must draw on large bodies of data about equipment and its performance in various states. The restrictions imposed by the relational model make it very difficult to deal with complex problems of this nature. We have chosen to base our database on tables, in particular they may be thought of as extended FITS binary tables.

A FITS binary table, from an abstract perspective, is a collection of rows of data fields, organized into columns. This structure is familiar to spreadsheet and relational database users. The data fields under the same column all have similar attributes. A descriptive header is associated with the table. This header applies to and describes the table as a whole. There is also a set of keywords, each of which applies to the table as a whole. A keyword is a name plus a data value. The table header is a set of data items corresponding to a fixed set of attributes; all tables have this set of attributes. Keywords, on the other hand, are optional and apply only to a particular table.There are several significant properties of such a table.

- A table consists of a header, an arbitrary number of keyword fields, and a fixed number of columns of fields organized into rows of a fixed size.
- Keyword fields have the same data types and properties as fields within columns.
- Keyword fields may be attached to individual columns or to the table as a whole.

- Fields within columns may be stored directly or indirectly.
- Fields may be single data items or multi-dimensional arrays of data items.
- The data types of the fields are an extensible set of types, but include the following: logical, bit, unsigned, short, long, character strings, float, double, single-precision complex, and double-precision complex.
- Fields can reference other tables.

As an illustration, the Figure 4 gives a possible layout of VLBA data. N.B. This is not the layout which will be used by AIPS++

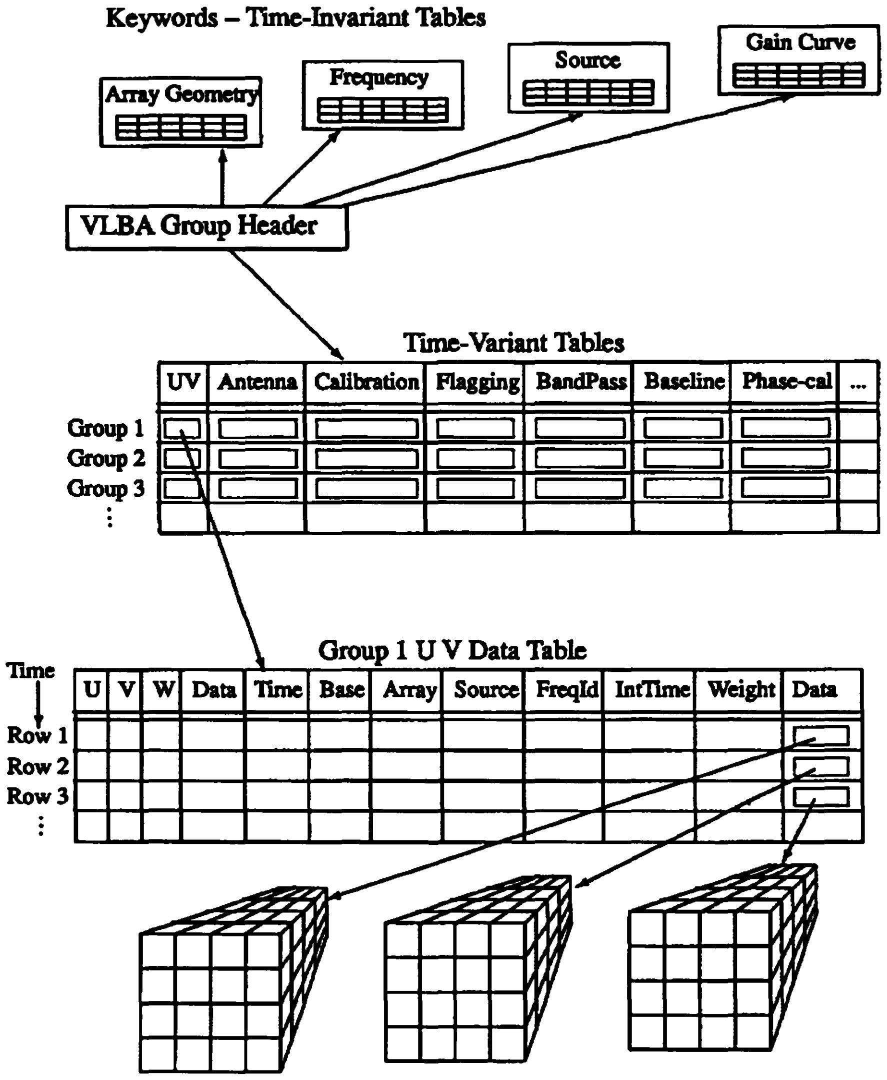


Figure 4 — VLBA Data as a Table of Tables.

Four classes form the foundation of the table system. Figure 5 shows their relations. Two classes are abstract base classes, from which classes can be derived to implement virtual tables.

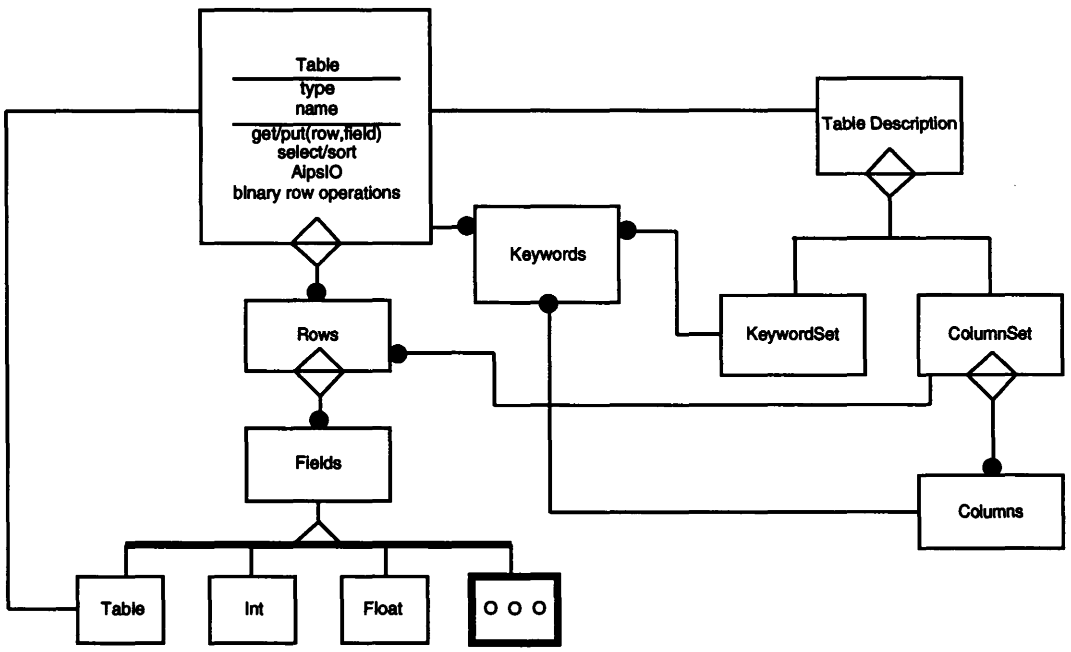
`BaseTable`
: This class represents a table. An individual keyword or column is represented by class `ColField` (see below). `BaseTable` defines the possible operations on a table, so it contains a lot of virtual functions, which in general have to be implemented in the derived classes. However, a programmer will never use the class BaseTable or its derived classes directly, since the lightweight class `Table` forms an interface to the reference counted table classes. Therefore `BaseTable` also contains a reference count and functions to manipulate it.

`ColField`
: This class represents a keyword or a column in a table. Of course this class also contains a lot of virtual functions which have to be implemented in the derived classes.

Derived from these classes are the classes `FillTable` and `FillColField` to implement the filled tables. One can also inherit from these classes, since they offer more functionality than the base classes. For instance, they offer a table and field description and a mapping of keyword and column names to `ColField` objects using the template class `OrderedMap`.

Furthermore the classes `Row` and `Field` play an important role. They glue the `BaseTable` and `ColField` classes to perform operations on a per-row basis. `Field` is quite simple, but `Row` is somewhat more complicated.

FILLED TABLE SYSTEM



The available field types will be:

Char	uChar
short	unsigned short
Int	uInt
float	double
Complex	DComplex
Bool	String

In the future the following has to be added to the system:
- join operation
- FITS reader and writer
- edit/show/plot
- query language
- analysis
- optimized storage manager

Figure 5 — The Filled Table Class system.

5.2 Storage Manager

[This area is being actively worked on, and a more fleshed-out proposal should appear during the workshop. This discussion is from Ger van Diepen and Richard Gooch.]

Currently the only (primitive) way to store AIPS++ tables is AipsIO. This requires that a table fits entirely in virtual memory, which is fine for small tables, but not for big tables like those to be used for uv-data or images. For now we have concentrated on the representation of UV-data. We have been looking into ways to store UV-data in a flexible, but yet efficient way. We have come up with a scheme which at first makes use of current software, but can be extended.

The storage manager is the layer taking care of storing the data into files and retrieving it in an efficient way. The table system forms a layer between the user and the storage manager. It is possible to have several storage managers:

1. Initially we'll provide the Karma storage manager. Karma is a software package developed in C by Richard Gooch at CSIRO. It handles storing and retrieving data (using canonical format). It is in principle based on memory-mapping of files.
2. The current AipsIO storage manager.
3. Later a FITS storage manager can be implemented to handle files in FITS format.
4. Other storage managers may be created to handle specific data formats. In particular the PLOP-hashing technique (investigated by John Karpovich at UVa) could be useful.

The user has to give information (hints) about the storage manager to use. Some more hints may also be required, which are described below.

An TDAViewer will be the object talking to the tables containing the uv-data. We propose the following:

1. The TDAViewer will split the uv-data into 2 tables:
   - The uv-data themselves with the coordinates; so this will consists of freq, baseline, time, polarisation and the complex visibility datum + weight. This will in principle be a filled table, although you can also think of a virtual table to implement an artificial observation.
   - The uvw coordinates for each time/baseline. This can be either a filled table (eg. for the VLA) or a virtual table (eg. for Westerbork).

   The reason for this splitting is twofold:
   - It makes it easier to handle cases where uv coordinates are calculated on the fly, in a more generic way.
   - It allows more freedom to store the data in any order. If uvw was stored with the data, it more or less dictates that the frequency and polarisation were stored at the bottom, otherwise the uvw had to be repeated too often (it reflects the principle of normalisation in data bases).
2. The tables mentioned in 1a and 1b will be described as columns of freq, baseline, visibilities, etc. (but they will not be stored that way). By describing the data as columns, there is no predefined ordening and the TDAViewer can take slices in any direction. The efficiency of the slice direction depends on the way the data is stored. However, with good tiling (or PLOP-hashing) all directions can be more or less equally fast.
3. Storage managers may need additional hints about the way the data should be stored and possibly about the size. Karma needs the following hints:
   - Is the column data, a coordinate in an array or a coordinate in a list? Eg. the VLA case can be stored as a list (with coordinates time, baseline) of arrays (with coordinates freq and polarisation) containing the data visibility and weight. The difference between a list and array is that a list is variable sized, while an array is fixed sized.

- The dimension sizes have to be given in advance, with optionally tiling information. Also the coordinates for the array axis have to be given.
- It is possible to specify that a list will be filled in (ascending or descending) order. This can be used to speed up to retrieval process.
- At a later stage it is possible to specify that an index should be created for list coordinates to speed up retrieval.
- At first it will be required that the table file will be filled sequentially. Later this can be made less strict, although it may require that the sizes of lists should also be known in advance.

So the VLA case could be defined as:

'`time`'	list, ascending
'`baseline`'	list, ascending
'`source-id`'	data
'`freq`'	array, 32 (+ possible tiling info)
'`pol`'	array,4
'`weight`'	data
'`vis`'	data

4. The table system and storage manager will have a few main functions.

'`Append`' to append data to the file. This is intended to fill the file.

'`Window`' to specify a window for the data to be accessed. A window for the Karma storage manager is an N-dim array of M start,end pairs. So one can define times t1-t2,t3-t4,t5-t6 for baselines b1,b2,b3-b4. The table system will allow you to specify the window in the current "select-syntax", thus as time>=t1 && time<=t2 || etc..

'`Next`' to get the next part of the data. This can be a N-dim array of data in any direction (eg. an it-slice, fit-brick, i-series), but only if the array is filled (thus all axes have same size). The coordinates will also be returned. Note that a 1-dim vector is always filled, although its length will vary per iteration if the underlying data is irregular. next will probably need an argument to specify the order (if any) in which the iteration must be done.

'`Update`' to update the data last retrieved.

The implementation of this all is not trivial (far from that). Karma needs to be changed in the following points:

1. Creating a file without first creating it first in memory.

2. Using the hints specification.
3. Support of the window and iteration mechanism.

The table system needs:

1. Full implementation of the TableVector.
2. Converting select expression to Karma window specification.
3. Support of multiple storage managers

Areas to be addressed in the future are:

1. File locking and sharing.
2. Indices (B-trees, maybe spatial indices).
3. Client-server?
4. A storage manager supporting tables where rows and columns can be added or deleted.

6 Measure, ReferenceFrame, and Representation Classes

The Measure class is a fundamental class for dealing with telescope data, images, and most data processing variables in AIPS++. It is used, in conjunction with the ReferenceFrame and Representation classes to take care of all of the following for a wide range of measures: units, transformation properties, and changes in representation. Figure 6 shows the Object Model for these and related classes.

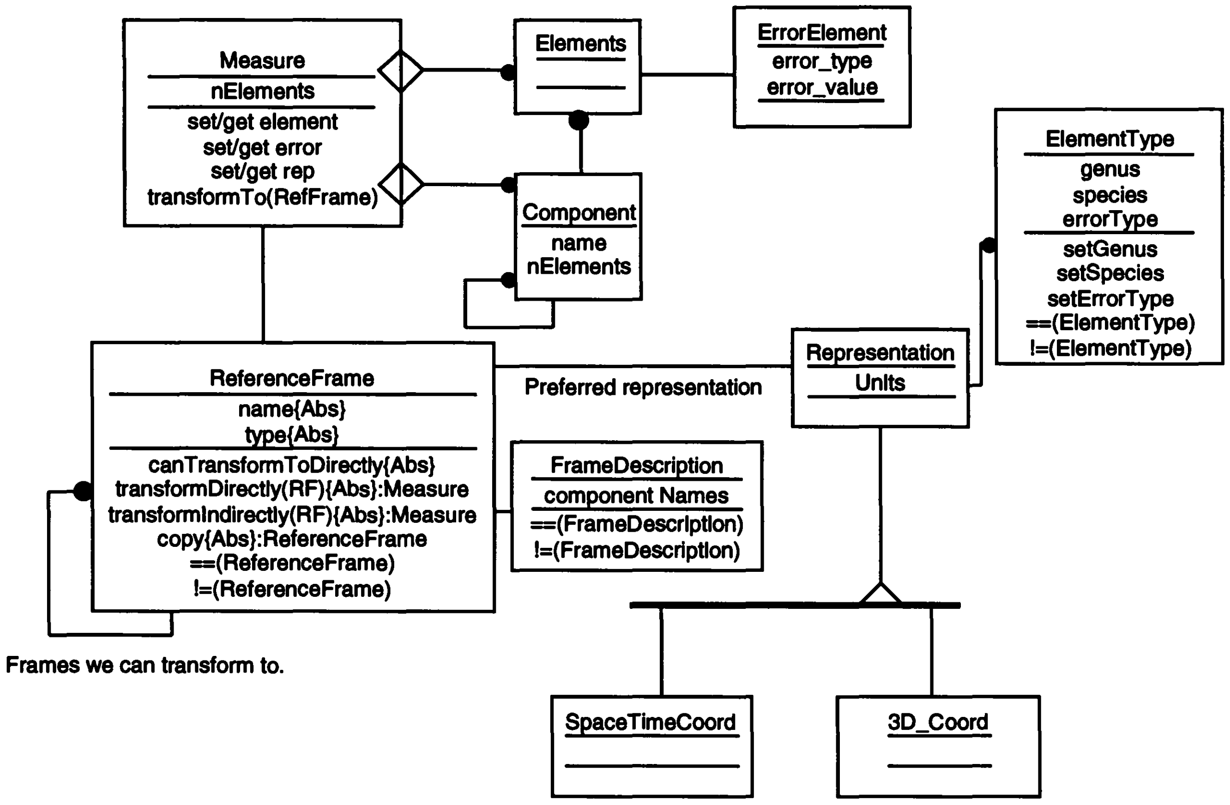


Figure 6 — Measure Class Object Model

Figure 6 illustrates that each Measure object is an aggregation of many possible Elements (vector-like) each of which is associated with an ErrorElement that describes at least an error type and value. One of the most common sorts of Measure objects are coordinates and sets of coordinates, including time, frequency, spatial coordinates, etc. Another example of a Measure object is the radiation intensities at specific spatial locations, frequencies, and time which can be described in representations like IQUV, (Irr, Ill, Irl, Ilr), (Ixx, Iyy, Ixy, Iyx), Poincare sphere parameterization, etc., and which can be changed by ReferenceFrame changes that include effects like Doppler shifts in frequency due to motions, etc.

The vector properties of the Measure object are described by a aggregation of components that can be named and counted, and which are associated with a Representation object for each ReferenceFrame object. The Representation object allows changes of units, transformation between different representations of the same Measure (e.g. spherical polar <-> cartesian representations of a 3-D vector Measure object), and each Representation object has an ElementType identifying genus (transformations between representations with the same genus are possible, those with differing genus are not possible), species (e.g. spherical polar coordinates, cartesian coordinates) that are transform possibilities for each genus, and an error type.

As shown in Figure 6 these classes are planned to allow optional error accounting and propogation for compenents of each measure. Each Measure object, with its Representation object, is associated with a ReferenceFrame object which takes care of all changes due to change of reference frame. These may range from simple Doppler shifts that alter frequencies to relativistic transformation between things like proper space-time coordinates and the equivalents for particular accelerated or non-accelerated reference frames. Changes in (u,v,w) due to geometric changes are taken care of by the Representation object, but inclusion of relativistic effects affecting antennas which lie on reference frames accelerated with respect to each other require sophisticate ReferenceFrame methods. The ReferenceFrame class contains all the methods for direct or indirect (with intermediate reference frames) transformations of Measure objects, with the associated Representation class taking care of the transformations within each reference frame.

The classes in Figure 6 should allow us to adopt simple models for potentially complicated transformation with a smooth evolution path for methods with high levels of complication. The attempt to allow error transformation in both Representation and ReferenceFrame is hoped to allow eventually complicated error propagation with methods for ensuring that transformation errors can be kept within specified bounds.

Figure 7 contains a object model showing a few of the obvious coordinate-oriented reference frame classes that will be derived from the ReferenceFrame base class. They derived classes are not complete. It also begins to show how a LocalFrame for a telescope, which may be mainly Topocentric (surface of the Earth) or OrbitingRF, may be modified to include different models of the non-spherical, non-uniform Earth (e.g. Earth tides, non-standard potential terms).

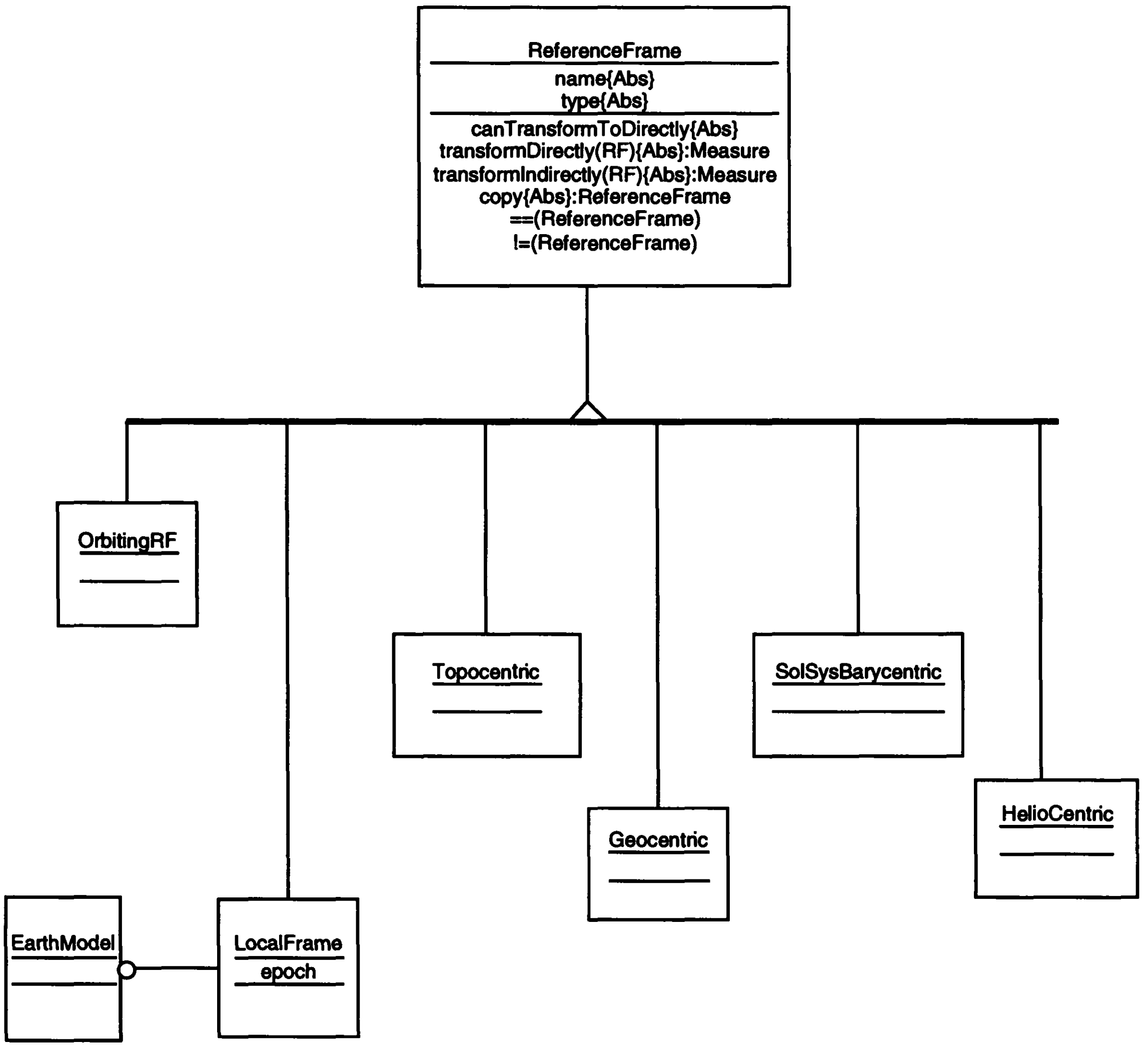


Figure 7 — ReferenceFrame Object Model

7 Telescope Data Associations

7.1 General TDA Design Considerations

The most advanced stages of AIPS++ design have, since the February 1992 workshop in Green Bank, been in the area of TelescopeModel and MeasurementSet. We now put them under the name of a Associator class called TelescopeDataAssociation. MeasurementSet is the current terminology for a set of measurement data; this name was adopted when it was decided in November 1992 that Yeg and YegSets were names that caused more communication difficulties than they clarified.

AIPS++ Implementation Note 152 (Shone 1993) contains a summary of the design conclusions from the November 1992 mini-workshop. It describes both philosophy and details for the minimalist design of attributes for TelescopeModel and MeasurementSet. It also expresses the initial description of the role of Applications Data Objects, which will be summarized later.

Figure 8 is a detailed version of the Object Model for both MeasurementSet and TelescopeModel. The principal changes from the discussions in the Software Design section of the Project Bookshelf, largely a result of the the November 1992 mini-workshop in Charlottesville and Green Bank, are:

- the diagram shows that TelescopeDataAssociation is a aggregation of MeasurementSet and TelescopeModel
- each TelescopeModel can have one or many MeasurementEquations reflecting different models of the behavior of the telescope
- the full triad of calibration-related methods, TM::solve, TM::interpolate, and TM::apply are indicated for TelescopeModel
- the main attributes of a “minimalist” approach to the essentials are listed for most classes
- the Instrument class replaces CorrelatorModel as a better general name for either single dish receivers or interferometric correlators that produce measurements for ranges of time, frequency, IF, and frequency channel
- each Receptor and Instrument class is shown as being optionally associated with classes with separable frequency bandpass and non-frequency dependent classes that may be either tables or parameterized functions; this separability is assumed for the current prototyping although in general the system must be designed for non-separable calibration in time and frequency
- each Receptor and Instrument class is also shown as optionally associated with an InterpModel class used to interpolated calibration information for the cases where the CalSolution and BandpassSolution for Receptor or Instrument are stored in table form

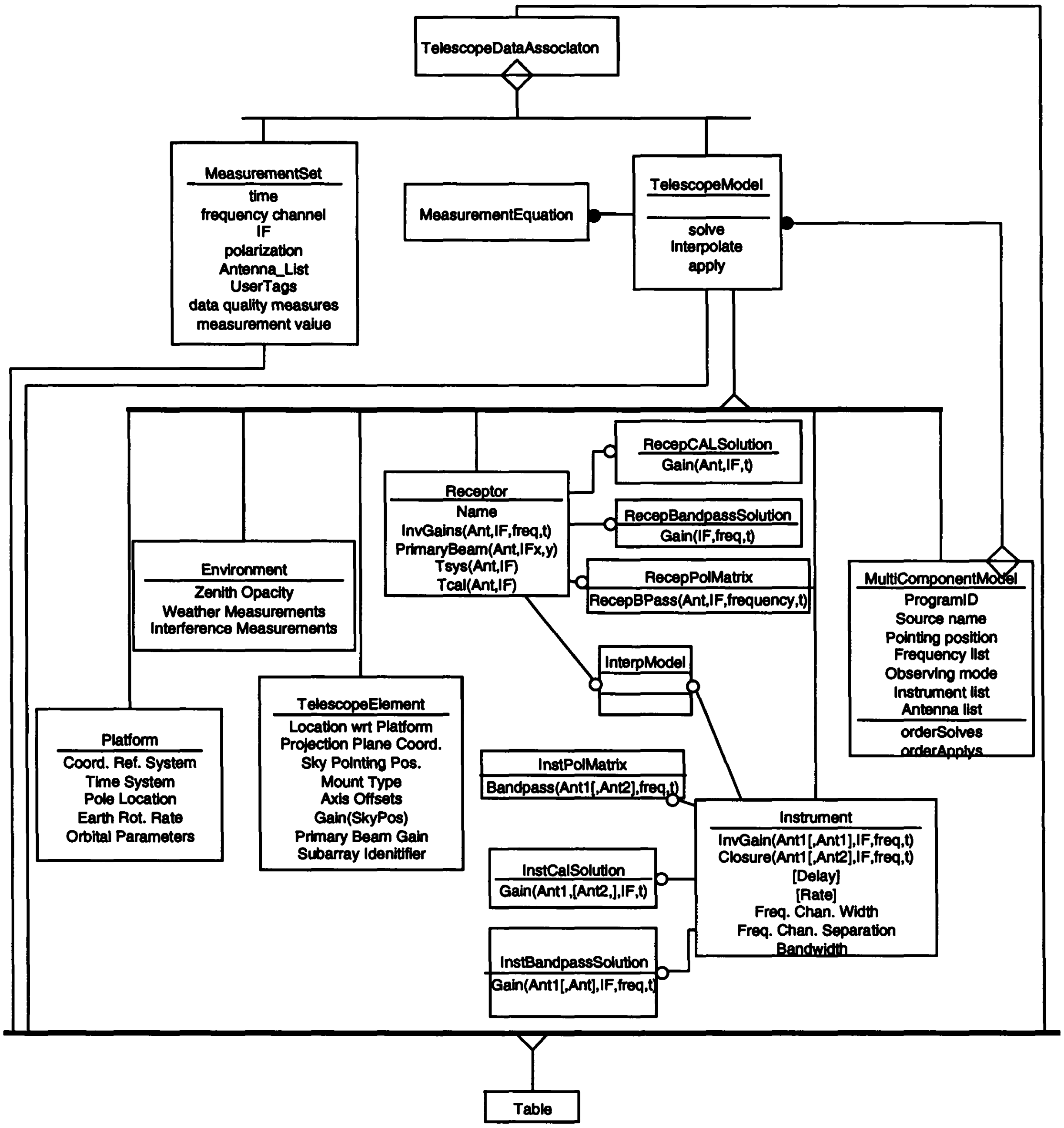


Figure 8 — Telescope Data Associations with TelescopeModel and MeasurementSet

Figure 9 shows a general FunctionalModel for the Calibration process involving MeasurementSet and TelescopeModel. In this diagram the TM:solve process is shown as a process using measurement data and measurements predicted by an assumed model of the radiation distibution involved. This covers both models for simple calibrators and models for complex sources (and self-calibration). The calibration solutions are shown as being written into Gain Solution Tables (CalSolution and BandPassSolution), although in some cases parametric calibration representations may suffice. Finally the TM.interpolate process shows inverse Gains written into Receptor and/or Instrument tables, and these are then used by an "apply" process described as "extract calibrated measurements" which can be passed on to applications data objects that use them, such as Spectrum for single dish data and VisSet for interferometric data.

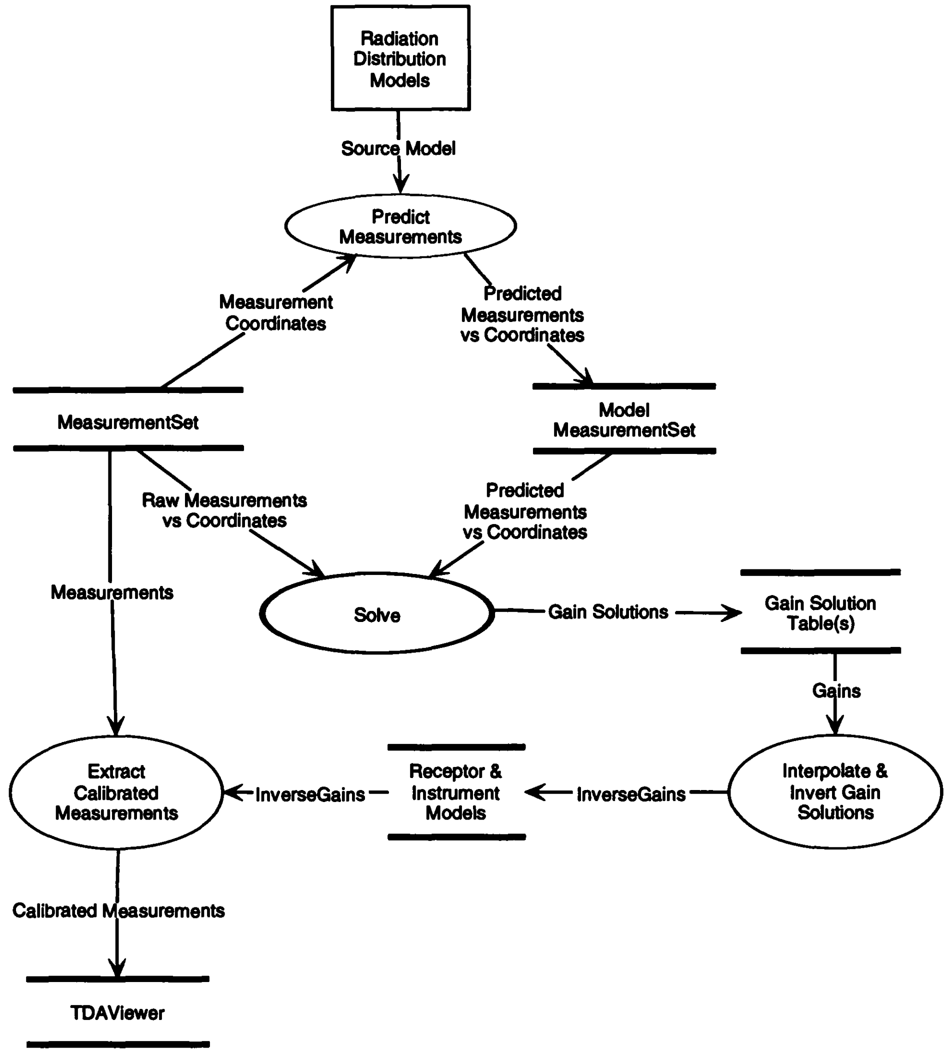

Radiation Distribution Models
Source Model
Predict Measurements
Measurement Coordinates
Predicted Measurements vs Coordinates
MeasurementSet
Model MeasurementSet
Raw Measurements vs Coordinates
Predicted Measurements vs Coordinates
Measurements
Solve
Gain Solutions
Gain Solution Table(s)
Gains
Extract Calibrated Measurements
InverseGains
Receptor & Instrument Models
InverseGains
Interpolate & Invert Gain Solutions
Calibrated Measurements
TDAViewer

Figure 9 — Functional Model for calibration in Telescope Data Associations

7.2 Single Dish vs Interferometric Telescope Data Associations

One of the principal conclusions of the November 1992 mini-workshop was that when one is careful about avoiding terminology specific to single dish or interferometric work, a surprisingly large portion of the design for data handling in these two regims is the same. The similarity and differences are described in detail in Shone (1993). An example of this is using a general Instrument class as one of the classed derived from TelescopeModel to describe both receiver backends in the single dish domain and correlators in the interferometric domain. However, because there will be a concentration of interferometric data handling during the Socorro Design Review and Workshop, we are not detailing the single dish case, although it is in there in all design discussions and diagrams where specifically interferometric aspects are not mentioned.

8 Data Area Viewers

As first documented in Shone (1993), TDAViewers, which at that time were called Applications-DataObjects, may be very important classes which are the views on data in a Data Association that are utilized in other classes and "programs". In this chapter we will summarize some of the design thinking in these areas. The TDAViewers that are designed to be the applications interfaces to data associations will be the basis for most applications programing in AIPS++.

8.1 Spectrum

The principle TDAViewer for spectral-line work, whether it be single dish or interferometry, will be the TDAViewer we are calling Spectrum. As are all TDAViewers, it is a view of data in a TelescopeData Association, and, in this case, of spectral data.

8.1.1 Construction of a Spectrum Object

Since the data from the spectra to be processed are in underlying table(s) of data, they are formed from selection criteria supplied by the program/user to the Spectrum constructor. As with all TDAViewers, the Spectrum object knows how to change its underlying data in its TDA, and in additon has methods reflecting the useful things that can be done with spectra.

The principal constructed data will appear as vectors of intensity values and vectors of frequencies that may be derived from extracted channel description information.

8.1.2 Methods of a Spectrum Object

The most important methods of the Spectrum object are those that return matching vectors of intensities and frequencies, by reference or by value depending upon whether the data are a synchrononized view of the original TDA valuse or a changed by some operation (like averaging, gridding, interpolating). The extraction process always involves selection criteria expressed in user-domain terminology (which may differ from descriptors in the TDA), and may involve a change in Measure-type if there are changes in units, reference frame, or representation involved. A primary example involves changes of the vector of frequencies through Doppler shifts implicit in changes of reference frame. In addition, there will be methods for returning a vector of velocities.

Spectra may have special "coordinates" in the TDA, such as the value of integrationPhase, that are parts of the collection of spectral data, and the extracted spectra may be formed by arithmetic on data in the TDA. An obvious example in the single dish domain is (ON - OFF) or other combinations of data taken at different phases of instrumental state (ON, OFF, REFERENCE, etc.). There can be comparable formation of "continuum baseline spectra" from specified fits to some channels and extraction of spectra based upon differencing or dividing. Thus there will methods for all the reasonable spectral manipulations.

Derived from the Spectrum object will be all the special spectral fitting, analysis, etc. classes that are incremental additions to the minimalist attributes and methods of the basic Spectrum object. Spectrum objects can be formed from model computations (represented by TelSystemSimulator in Figure 1), and derived Spectrum object can involve arithmetic, fitting, etc. between model-derived and data-derived spectra.

During the discussion in the mini-workshop in November 1992 it was jokingly suggested that the role of a Spectrum TDAViewer was such that one could write TDAViewers that emulated the behavior of many single dish data handling systems, which are largely systems for editing and arithmetic on vectors of data. It is now being seriously discussed whether the prototyping of an TDAViewer class call UniPOPS might be an interesting test of AIPS++ TDA and TDAViewer design.

8.2 VisSet

For interferometric TelescopeDataAssociations the current concept is that all applications using these data will access them through VisSet, or classes derived from VisSet, where VisSet is an application data object which has all the methods for fundamental operations on visibility data. Figure 10 shows an Object Model diagram of VisSet.

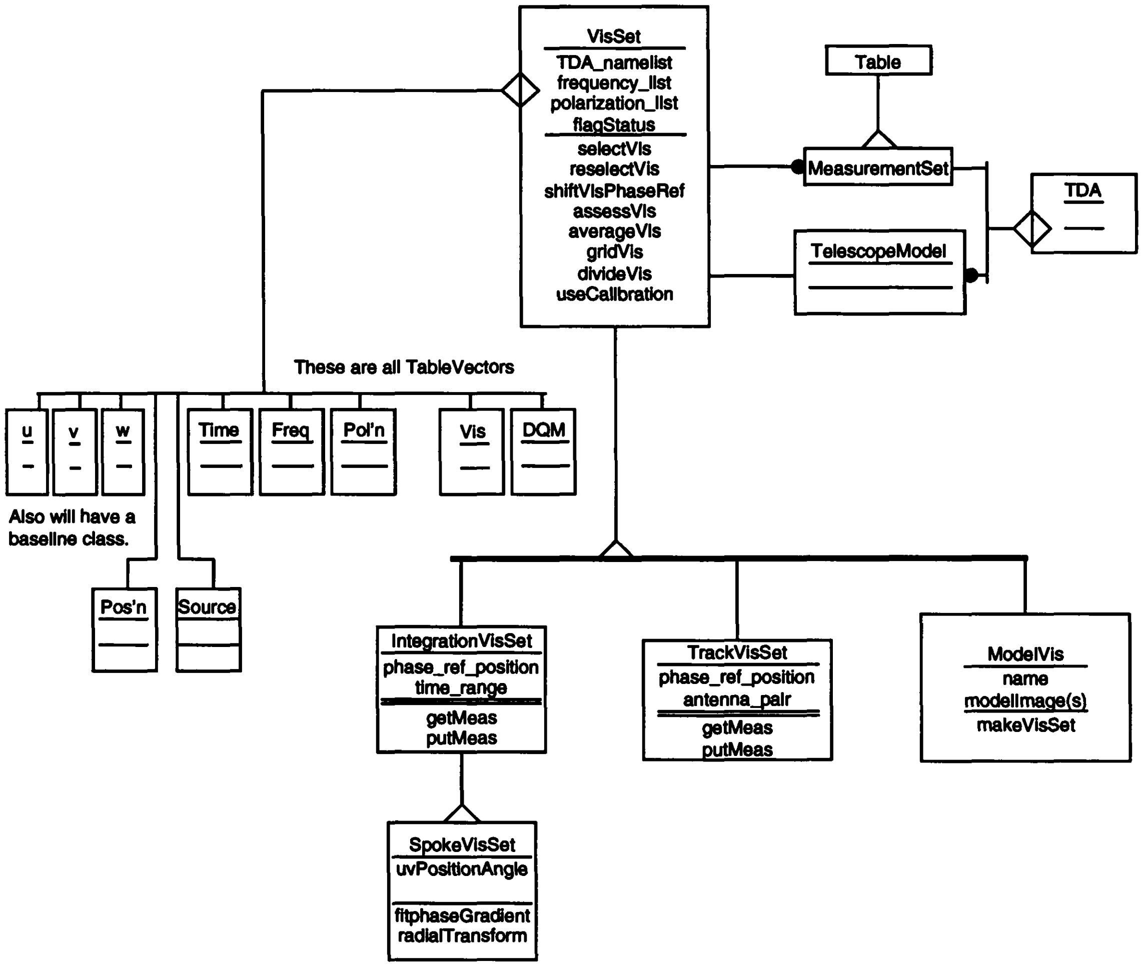


Figure 10 — VisSet Object Model

8.3 Editing Data Quality Measures and Other Data

8.3.1 Editing Application Data Objects

The initial design for editing data in a Telescope Data Association is based upon editing TDAViewers that are a user-specified view of the TDA. This may be a view of all or part, calibrated or uncalibrated, forms of the data in the TDA. Figure 11 shows a Functional Model diagram for one way to change (edit) data in a TelescopeDataAssociation using methods that operate on application data objects in general, but which we represent by VisSet/Spectrum. The principal concept is that from a VisSet/Spectrum TDAViewer formed as a view of the TDA, a subset (or copy) is formed called a WorkSet, and with each WorkSet an OriginMap is created that desribes the mapping between data values in the original TDAViewer and the WorkSet. The Workset is modified and examined by appropriate processes shown in Figure 11, and when appropriate the current state of WorkSet and the OriginMap are used to apply changes in the TDA by passing the appropriate change descriptors to change/editing methods in the VisSet/Spectrum TDAViewer.

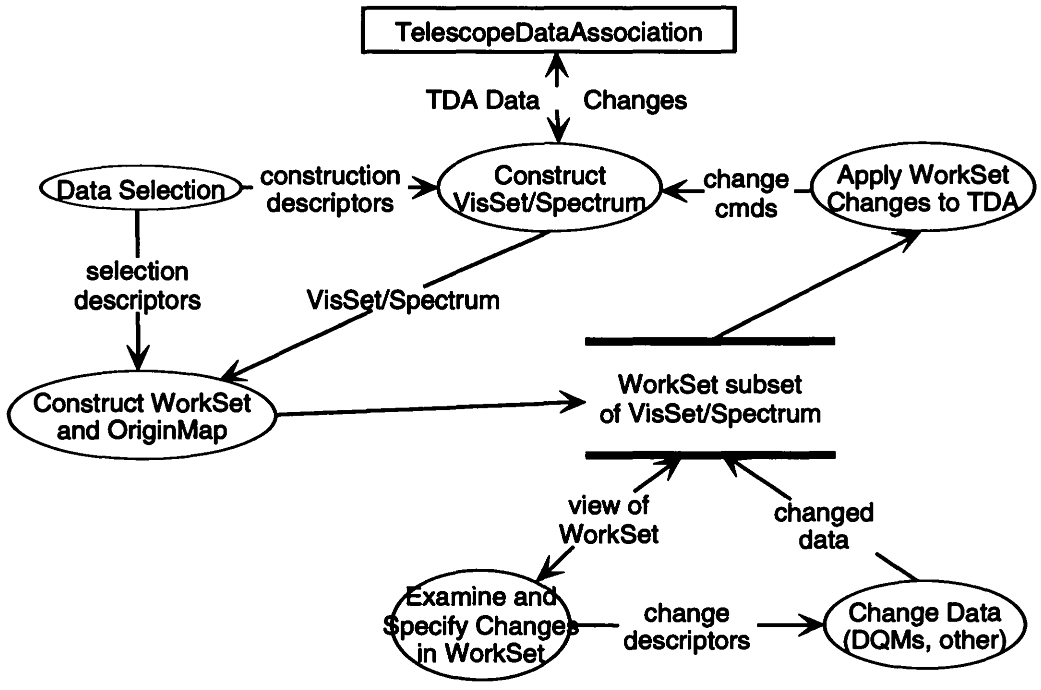

TelescopeDataAssociation
TDA Data
Changes
Data Selection
construction descriptors
Construct VisSet/Spectrum
change cmds
Apply WorkSet Changes to TDA
selection descriptors
VisSet/Spectrum
Construct WorkSet and OriginMap
WorkSet subset of VisSet/Spectrum
view of WorkSet
changed data
Examine and Specify Changes in WorkSet
change descriptors
Change Data (DQMs, other)

Figure 11 — Functional Model of Changing (Editing) TDA DATA

8.3.2 Editing Tables

It is planned to have special Application Data Objects for Editing Tables themselves independent of their relationship to other Tables. This will allow immediate and relatively low level access to data. The philosophy is to implement the sort of table editing, table arithmetic, etc. that is described in the previous section, but for tables themselves rather than associations of tables oriented to specific types of application data objects. This TableEditor will mainly need to use the Table classes and the Measure class to interpret each column in a Table in terms of its measure-type. This should provide the type of data access for both de-bugging and user manipulation that has been very successful in MIDAS.

8.3.3 Examining Data in Plots and Listings

During the prototyping phase, use of the Khoros/Cantata User Interface for development purposes implies that the level of examination of data by plots, listings, images, etc., will be involve preparing data with the Viff class. Eventually, preparing data in the form of plot input objects that can be handled by the display system will be needed; but these have not yet been designed, nor has the display or visualization system.

9 Image Classes

In previous discussions we have distinguished between stored images and images as application data objects. Most images in applications code are applications data objects, that is, they are views on stored images. Simple as some of them might be, images as classes have not yet been discussed or designed in detail. However, Figure 12 shows an Object Model diagram with several important aspects of the image class system.

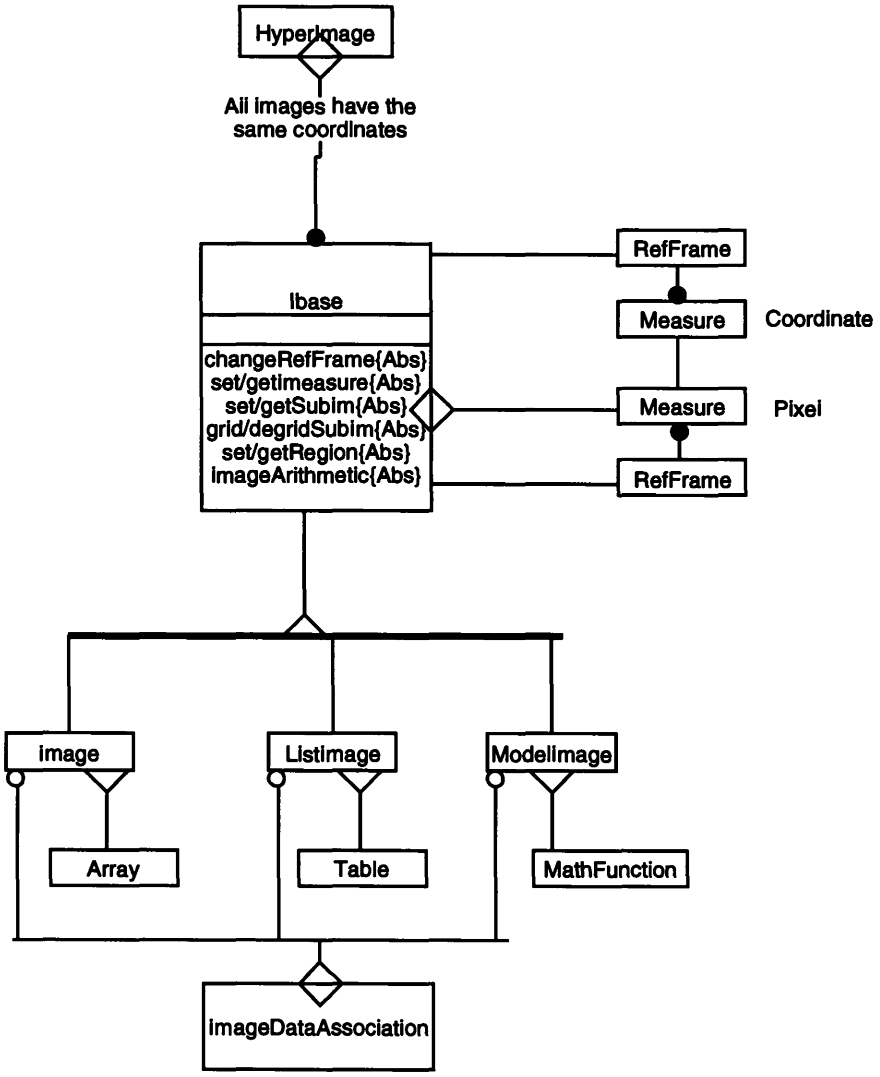


Figure 12 — Image-related classes

Because most images in astronomy are n-dimensional arrays of "values" as a function of coordinates with even intervals in each axis, we reserve the class name Image for images of this type, indicating that they inherit from the Array class. At the same level we distinguish between ListImage, which is a Table form of image that allows arbitrary coordinate intervals and non-line image coordinate transformations. We also assume there is a ModelImage class based upon mathematical functions with parameters, so it is a parameterized model of an image; this is based upon a general MathFunction class that is a base class for general parameterized functions. An early prototype of MathFunction has been used in GridTool.

In Figure 12 we show Image, ListImage, and ModelImage as components of an ImageDataAssociation. We tentatively assume that the attributes and methods needed for multiple or mosaic imaging situations are objects which are associated with ImageDataAssociation. However, for cases of multiple images with the same n-dimensional coordinates, we have the class HyperImage.

The most important class in the Image-related class system is Ibase, since it is an abstract base class for all images. Ibase, as shown in Figure 12, is associated with a Measure for each image value or pixel and a Measure for each coordinate axis. Both are associated with a ReferenceFrame (and by implication a Representation object) object. Methods of Ibase carry out operations of the following type:

- arithmetic operations between images of the same measure-type and shape
- change pixel and coordinate values based upon reference frame changes
- change measure-type pixel and coordinate values
- get and put subset images (SubImage) of an image
- get and put subset regions (irregular) in an image

Image-related classes need much more design and prototyping before we will be sure essential needs are met for a powerful set of general image-related classes that can be used for many specialized purposes.

10 MeasurementModel and Imaging

10.1 Mathematical Basis for the MeasurementModel Class

While, on the one hand, there has been extensive work on image computation based upon use of basic math classes and the higher level GridTool and FFTServer classes, a general class system for "imaging" has not yet been prototyped. Early discussions of this problem raised the question of whether "Imager" classes were valid classes. The conclusion at the time was that they were in the same sense as Coggin's "enzyme" classes, where classes themselves simple produce changes in other classes. However, as introduced earlier, the Cornwell (1992a) document indicated there was a simple conceptual system for imaging based upon an operator A that acts on the radiation distribution the sky, S, to produce measurement data D, and we believe that this leads to a natural base class we call MeasurementModel. For the current discussion we assume that we are dealing with calibrated data so the equation on which the measurement (nee imaging) models are based is

$$A_{\mathrm{MM}}\ S = D_{\mathrm{CalMS}}$$

so one is dealing with MeasurementModel transformation of the sky's radiation distribution into calibrated MeasurementSets.

Given that background, let us summarize the mathematical essence of Cornwell (1992a) and use the results to define the MeasurementModel base class. In addition to the previously defined quantities we need to define a weight vector W for each measurement set D_{CalMS}. Given that, the following matrix algebra operations are definable that constitute the essential operations of MeasurementModel:

$$D_{\mathrm{predict}} = A_{\mathrm{MM}}\ S$$

given any model distribution for S and a mathematical formution of a MeasurementModel selected from the MeasurementEquation class inherited from MathFunction. One then uses

$$W_{\mathrm{norm}} = A^{\mathrm{T}}_{\mathrm{MM}}\ W A_{\mathrm{MM}}$$

to compute normalized weights, and

$$S_{\mathrm{solve}} = A^{\mathrm{T}}_{\mathrm{MM}}\ W D_{\mathrm{CalMS}} / W_{\mathrm{norm}}$$

to produce the principal solution for S, which is sometimes called the "dirty" image. One then has

$$S_{\text{observe}} = A_{\text{MM}}^{\text{T}}\, W\, (A_{\text{MM}}\, S_{\text{model}})/W_{\text{norm}}$$

for the computation of an "observed" radiation distribution for any model of the source distribution, obviously the principle solution point spread function (PSF) or "dirty" beam if the model is that of a point source.

Many fitting or deconvolution algorithms use the services of methods for computing errors and error gradients. Errors themselves are determined from

$$\chi^2 = (A_{\text{MM}}\, S_{\text{model}} - D_{\text{CalMS}})^{\text{T}}\, W\, (A_{\text{MM}}\, S_{\text{model}} - D_{\text{CalMS}})$$

which computes the fit of any model to the data, which may be in image, visibility, etc., domain, and the gradients to the fit of model and data are computed with

$$\frac{1}{2}\frac{\partial\chi^2}{\partial S} = A_{\text{MM}}^{\text{T}}\, W(A_{\text{MM}}\, S_{\text{model}} - D_{\text{CalMS}}).$$

In Figure 13 we show an Object Model for the MeasurementModel base class with an indicated association with a MeasurementEquation, and a beginning of the obvious derived classes for principal solution imaging and image deconvolution. In Figure 13 the data vectors D are associated with coordinate vectors **u** and the source distribuition on the sky are for source coordinates **x**. The principal solution imaging class does all of its work with MM.solve and MM.observe methods of MeasurementModel. Image deconvolution classes, conceptually, and in reality for methods like MEM, do most of their work with MM.chisqResid and MM.dchisqdS methods.

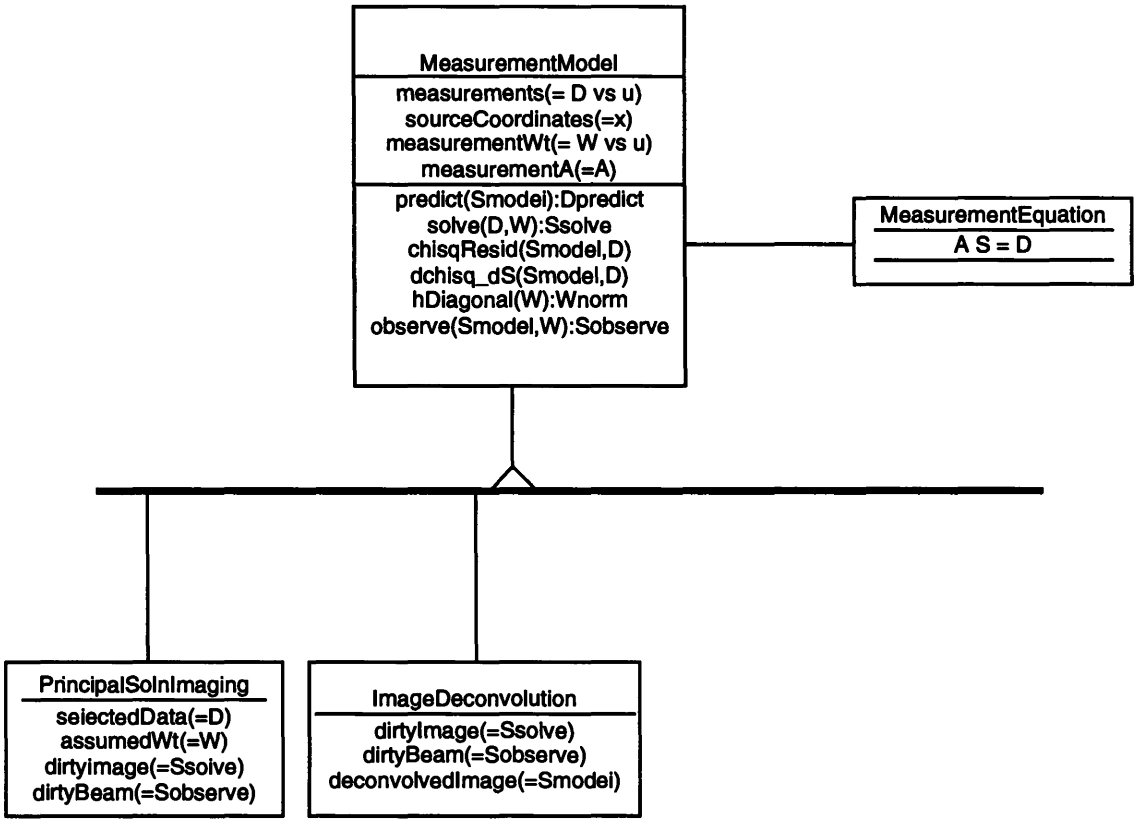


Figure 13 — Measurement Models

While the MeasurementModel class seems to be an important beginning to a class system for algorithms for the reconstrucition and improvement of many types of applications data objects, it is only a beginning to one of the most important areas of design and prototyping in AIPS++.

11 References

Bottomly, T. 1993 "AIPS++ Software Design: Telescope Data Handling, AIPS++ Implementation Memo 108. Cornwell, T.J. 1992a, "Recommendations for the AIPS++ Imaging Model", AIPS++ Implementation Note 147.

Cornwell, T.J. 1992b, "Recommendations for the AIPS++ Telescope Model", AIPS++ Implementation Note 148.

Hjellming, R.M. and Glendenning, B.E. 1992, "An Initial Design for Major AIPS++ Objects", AIPS++ Implementation Memo 102.

Shone, D.L. 1993 (ed.), "Measurement and Telescope Data Sets and Applications Objects for Single Dish Work and Interferometry", AIPS++ Implementation Note 152.

Shone, D.L. and Cornwell, T.J. 1992 (eds.), "Calibration, Imaging, and Datasystems for AIPS++ - Report of the meeting in Green Bank, West Virginia, 3rd-14th February, 1992", AIPS++ Implementation Memo 103.

12 Appendix 1 - List of Classes (Data Dictionary)

Once the major classes have been identified, on of the steps in the Rumbaugh OMT is preparation of a "data dictionary" in which terms are defined. This following is such data dictionary that matches a number of the current higher level diagrams.

`TDAViewer`	a lable or an abstract class describing a view on data in a data association with methods for extracting and changing that data.
`Array`	a class for array handling by indexing n-tuples of integer numbers with methods for a wide range of operations valid for arrays
`BandpassSolution`	a table associated with the Receptor class that contains frequency-dependent calibration tables with data derived from MeasurementSets and intended to be inserted in calibration measure columns.
`CalSolution`	a table associated with the Receptor class that contains time[-frequency]-dependent calibration tables with data derived from MeasurementSets and intended to be inserted in calibration measure columns; antenna or antenna-pair based, depending upon the table to which it is attached.
`Cube`	a class derived from Array implementing mathematical operations valid for 3-D arrays.
`DataSource`	a source of measurements for a MeasurementSet. May be telescope data media written by a telescope system, a real time data stream for a telscope system, or a TelescopeSystemSimulator
`Element`	a component of an array or measure
`ElementType`	a class encapsulating the genus, species, and ReferenceFrame identification for a measure of a particular type, e.g. coordinates, frequency, space-time vectors, IQUV, etc.
`Environment`	a class containing direct measurements of the environment of telescope systems during observations. Environment includes weather, atmospheric parameters (if measured), and interference data (if measured).
`ErrorElement`	a class with attributes and methods appropriate for describing errors and their types in the Measure class

FrameDescription
a class associated with ReferenceFrame which contains the attributes for each element of Measure

Image a class containing data and methods for measures as a function of 1-N dimensional coordinate tuples with even coordinate intervals, specific genus and species of reference frame(s), and methods for accessing and transforming these data

IntegrationVisSet
a class derived from VisSet that has organization and methods appropriate to all visibilities for a single integration time for measurements from an interferometric array.

Instrument
a class derived from TelescopeModel that describes the state of backend information for a telescope system. For a single dish it describes data and calibration information for spectrometers, etc. For a correlation interferometer it describes the state and calibration properties of the correlator output.

Matrix a class derived from Array which allows mathematical operattions valid for matrices

Measure a class identifying the value(s), errors(s), and unit(s) for a measure-type with transformation properties determined by the ReferenceFrame. Time, frequency, position, velocity, receptorID, total power, visibilities are all examples of measures with a specific genus and transformation rules for that genus.

MeasurementEquation
an abstract class which models the measurement equation for how a particular telescope system produces measurements.

MeasurementModel
an abstract class with data and methods for constructing, or reconstructing measurements, usually made by a telescope or telescope system. An ImagingModel is an Imaging-related MeasurementModel that has at various times been called an Imager.

MeasurementSet
a table of measurement data where each column is associated with a measure-type and a data-type that may be an array or another table.

MultiComponentModel
a class describing the state of a telescope system as a function of time and other coordinates which is used to store global information about the observing process and the telescope(s) and instrument(s) used in the observing process that results in measurements; it also couples the solve and apply methods of all the components it contains.

ObservingLog
a summary of measurments in a DataSource

`ObservingSchedule`	a class or table prescribing the parameters controlling the observing process for a telescope or telescope system. Optionally associated with a DataSource.
`Platform`	a class with parameters and methods for time, location, etc., used during the observations: type of RefFrame; time parameters; [orbital parameters]; etc.
`RadiationDistModel`	a table or array radiation intensities as a function of time, position, frequency, and polarization that is used to supply models of observed astronomical sources of radiation
`Receptor`	a class derived from TelescopeModel that contains antenna-IF based state information including calibration parameters like complex antenna gains. The later can include bandpass calibration, and polarization calibration parameters that may be associated with tables of calibration solutions derived from calibrator data in a MeasurementSet. Associated with an InterpModel class for interpolation from calibration solution tables to receptor calbration tables.
`Series`	a basic TDAViewer dealing with two vectors of related data inside a data association
`Spectrum`	a class derived from Series where one vector is data and the other vector contains frequencies for that data
`SpokeVisSet`	a class derived from VisSet via IntegrationVis Set with data organization and methods appropriate to a set of visibilities for a number of antenna-pairs in an East-West interferometric array.
`Table`	a set of classes for storing values in a structure of rows and columns. The type of all values in a given column is the same; valid types include arrays and tables. Tables may be directily associated with one another
`TelescopeDataAssociation`	an associator organizing an aggregation of MeasurementSet and TelescopeModel(s) derived from a DataSource of measurement data for a telescope.
`TelescopeElement`	A class with information describing a particular telescope. Its: location with associated ReferenceFrame; pointing equation/parameters; mount type; mount axis offsets; known antenna gain(az,el); known antenna beam shape; and the subarray the antenna is used in if it is part of an array.
`TelescopeModel`	a class associated with a MeasurementEquation and a MeasurementSet that describes the state of the telescope during the collection of the measures in the MeasurementSet. Telescope state information including calibration parameters are included in TelescopeModel, which itself can be related to a number of specialized telescope model classes.

`TelescopeSystemSimulator`	a simulation program, or set of programs, that emulates the operation of a telescope producing measurements.
`TrackVisSet`	a class derived from VisSet with data organization and methods pecular to a single baseline-pair or "track" in the u-v plane
`Vector`	a class derived from the Array class with methods valid for 1-D arrays
`VisSet`	a class providing the TDAViewer interface to visibilities and related data stored in an interferometric TelescopeDataAssociation

Short Contents

Table of Contents