Submitted for publication in Baltic Astronomy, March 1999
(= GAIA SAG-LL-027, 1999 Jan 18)

AN OBJECT-ORIENTED FRAMEWORK FOR GAIA DATA PROCESSING

W. O'Mullane[1], L. Lindegren[2]

[1] *Space Science Department, ESTEC, NL–2200 Noordwijk AG, The Netherlands*
[2] *Lund Observatory, Box 43, SE–22100 Lund, Sweden*

Abstract. A short prototyping exercise has been completed on the topic of data processing for a global space astrometry mission like GAIA. A prototype was constructed using the Hipparcos Intermediate Astrometry data, the Java programming language, and Objectivity (an object-oriented database system). Some features of the prototype are described in this paper and some design principles are suggested for large scale data processing in an object-oriented manner. Particular attention is paid to distribution of the processing and insulating algorithms from distribution and data storage mechanisms.

1 INTRODUCTION

The GAIA mission, studied as a candidate for a cornerstone project within the European Space Agency's science programme, features two astrometric and one spectrophotometric telescope within a single satellite structure. Launched to the Sun-Earth L2 libration point and continuously observing the sky during some five years, the satellite will generate about 20 terabytes of data pertaining to about 10^9 objects down to 20th magnitude. The main objective is to survey the stellar content of the Galaxy by means of astrometric, photometric and radial-velocity measurements. The astrometric accuracy aimed at will give absolute trigonometric parallaxes and annual proper motions to within a few microarcsec for bright stars ($<$ 12th magnitude), 10 microarcsec at 15th magnitude, and about 1 milliarcsec at 20th magnitude. In contrast with Hipparcos, no pre-defined observing list (input catalogue) will be used: objects are detected in real-time, as they enter the fields of view, by means of dedicated sky mapper CCDs.

The processing of the GAIA raw data into consistent sets of astrophysical data is an extremely challenging task. It is not just the volume of data that is formidable, but even more so the intricate relationships between different pieces of information gathered with the various instruments throughout the mission, and the complexities of the objects themselves at the required resolution and sensitivity. A highly automated, yet sophisticated data processing system will be required to take care of the bulk reductions. At the same time a great deal of flexibility and interaction is needed to cope with special objects or astrophysical investigations, many of which cannot be foreseen at the software design stage. On the other hand, the delicate calibration of instruments and celestial pointings, necessary to interpret the data in terms of absolute astrometric and photometric quantities, must be protected from unintentional modification.

It is envisaged that an object oriented (OO) database might provide a suitable environment for the GAIA data processing. Data access and processing could be organised in layers or shells for proper system integrity. The outermost shell would provide a general interrogation facility, much like many other astronomical data bases from the user's perspective. Deeper shells would give access to raw or intermediate data, calibration data, and procedures acting on and sometimes changing these data. At the core there would be the processing of perhaps 100 million well-behaved (i.e., apparently single and undisturbed) stars and extragalactic objects leading to a self-consistent data set for the astrometric reference frame, attitude determination and instrument calibrations (geometric and photometric).

Using the Hipparcos Intermediate Astrometric data (ESA 1997) a simplified model of the GAIA data processing was constructed. This lead to some interesting findings and below a framework is suggested for the building of an OO data processing software for GAIA.

1.1 The Global Iterative Solution

A general scheme for the astrometric core processing for GAIA, the Global Iterative Solution, was outlined in Chapter 23 of Volume 3 of the Hipparcos and Tycho Catalogues (ESA 1997). The same scheme can be used for the photometric processing — indeed there are good reasons to do it in parallel with the astrometry. The Global Iterative Solution consists of a cyclic sequence of three processes applied to the four data sets:

0. the CCD data, i.e. the telemetry signals from all the instrument detectors;
1. the calibration data, containing the geometric and photometric characteristics of all instruments and detectors;
2. the sky data, containing the geometric and photometric characteristics of the objects observed, in particular the astrometric parameters for single stars;
3. the attitude data, describing the celestial pointings of the instrument axes as functions of time.

Starting with provisional versions of the sky and attitude data, the processes can be schematically represented as $1 \leftarrow 0 + 2 + 3$ (calibration), $2 \leftarrow 0 + 1 + 3$ (astrometry and photometry) and $3 \leftarrow 0 + 1 + 2$ (attitude determination). This sequence is iterated until convergence.

A major practical complication is the way in which the CCD data need to be accessed in these processes. For the astrometry and photometry of a given object one needs access to all the short stretches of data in which the object was observed. These are spread out in a quasi-random manner along the whole mission time line and among the various detectors. The attitude determination, on the other hand, is done by smoothing the instrument pointings as function of time, for which one needs access to all the measurements in a given time interval, irrespective of object. For the calibration, finally, one typically needs to consider all the data collected on a given CCD chip. The calibration, sky and attitude data similarly need to be accessed in different ways. It is possible that the complex issues of efficient data storage, indexing and accessing are best met by a ready-made database system.

1.2 Some Object Oriented Terminology

In this paper we will make reference to terms with specific meaning within a database or object-oriented programming context. These are:

class	: encapsulates common behaviour of a group of objects
inheritance	: class may inherit attributes and methods from other classes
attributes	: data members of a class
methods	: functions which may be performed on instances of a class
object	: an instance of a class
interface	: similar to a class but has only methods (see Sect. 4.1)
abstract class	: a class from which no instances may be created.

The terms 'instance' and 'object' may be used interchangeably.

2 THE PROTOTYPE

Concepts for the GAIA data processing may be tested by means of a prototype system based on the Hipparcos Intermediate Astrometry data published (on CD-ROM) as part of the Hipparcos Catalogue (ESA 1997). Aspects of the GAIA data processing that can be tested by means of the prototype include:

- the definition of a number of subject areas with one or several classes in each area, and with some (realistic) degree of complex relations between them;
- the management of all this within a database;
- flexibility in modifying or adding to the structure, e.g. by introducing new sub-classes;
- overall speed and convergence of the global iteration scheme.

The notion is to make a 'global iterative solution' of Hipparcos intermediate astrometry, in analogy with the GAIA core processing outlined above, but on a scale roughly 10 000 times smaller than the actual GAIA data processing.

Table 1: Correspondence between GAIA data and Hipparcos Intermediate Astrometry data, and between the main processes for GAIA and the prototype. The numbers indicate the approximate data volumes in number of reals.

GAIA	Hipparcos Intermediate Astrometry
Data:	
telemetry data (10^{13})	(see text)
raw objects (10^{12})	abscissae (3×10^7) [one consortium]
attitude data (10^8)	great-circle harmonics (4×10^4)
calibration data (10^7)	chromaticity data (1000)
star catalogue (2×10^{10})	star catalogue (2×10^6)
Processes:	
astrometry	astrometry
attitude determination	determination of harmonics
instrument calibration	determination of chromaticity

2.1 Analogues

The correspondence between GAIA data and the Hipparcos intermediate astrometry is outlined in Table 1. The numbers indicate the approximate data volumes in number of reals.

The determination of the astrometric parameters (called *astrometry* in Table 1) is in principle identical for GAIA and the Hipparcos analogue: the one-dimensional coordinates of a star at the times of observation are combined in a least-squares solution of the position, parallax and proper motion. Only stars that are not manifestly non-single are considered here.

The *determination of harmonics* analyses the positional residuals along a reference great circle (RGC) in terms of systematic periodic errors (harmonics), again by means of least-squares solutions. Only statistically significant harmonics are kept. This process is somewhat analogous to the attitude determination for GAIA, in that all the positional residuals in a short time interval are combined to a smooth function of time.

The *determination of chromaticity* analyses the positional residuals in terms of star colour and time. This is done very simply by accumulating mean values in a number of cells. This process is similar to the CCD calibration process in GAIA, where mean residuals are calculated for each CCD pixel column, possibly also as function of time. In the prototype the determination of harmonics and chromaticity have been combined into one process.

There is no analogue to the GAIA raw data (telemetry) stream, as the abscissae are already sorted according to HIP number and thus correspond to a GAIA 'raw object'. It is not clear if one would actually sort the GAIA data in this way, or keep them in the original (chronological) order, or even maintain both

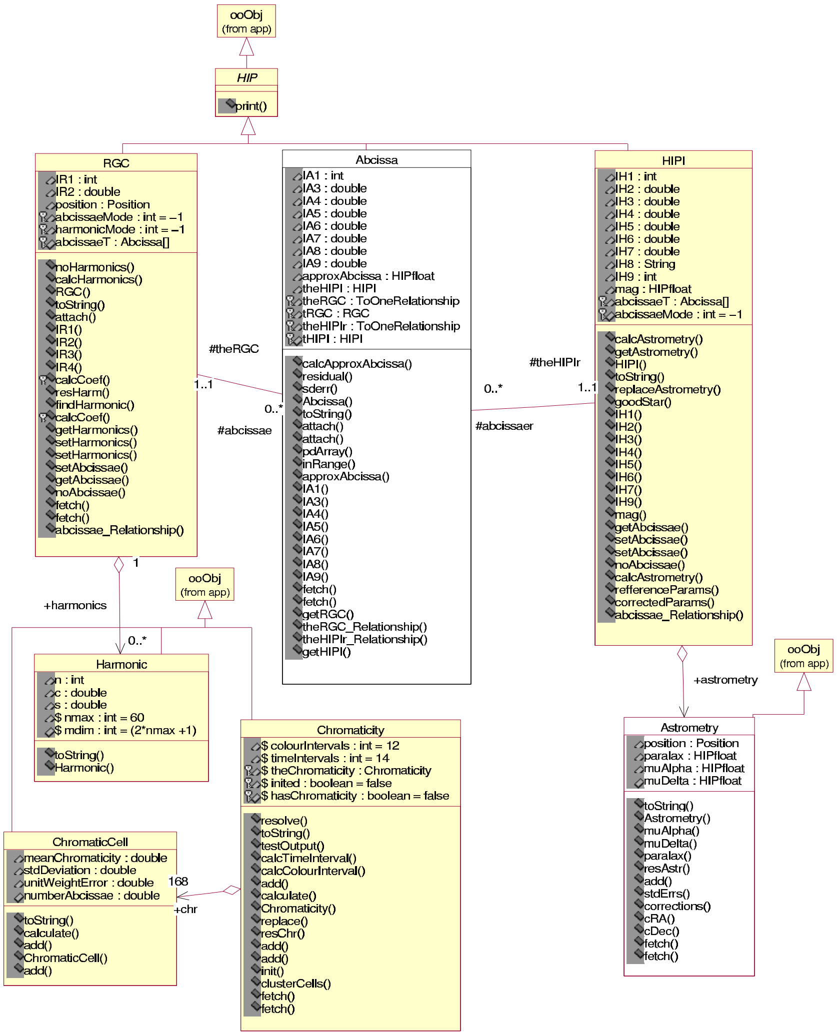

Figure 1: Data model

orderings in parallel. In the prototype the data is ordered according to position — time-ordered access is done via the RGC which has database pointers to all abscissae it needs to access.

2.2 Development

An initial data model was produced using Rational Rose and preserved throughout the exercise. From the Rose Model both documentation and Java code were derived. Algorithms were originally provided in Fortran and (manually) translated to appropriate Java routines within the OO model. This was achieved very quickly and indeed the model did not change very much thereafter. The latest incarnation is shown in Fig. 1.

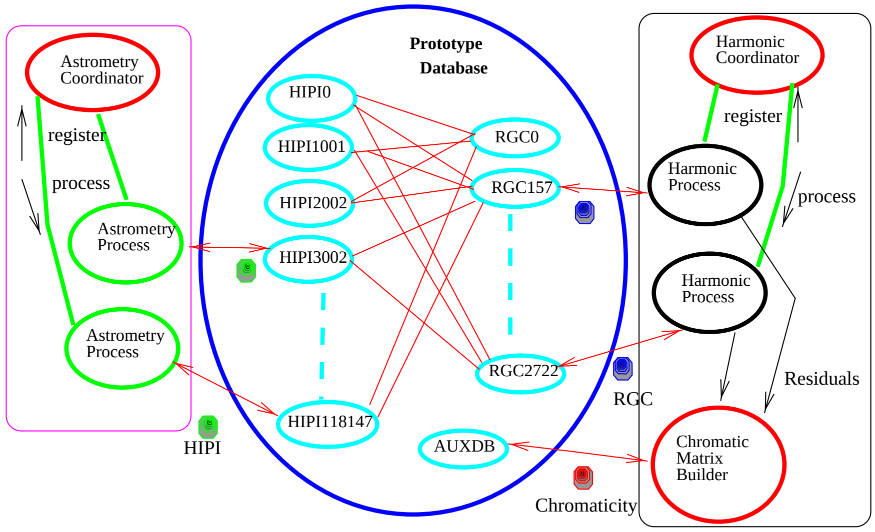

Figure 2: Prototype processing

The changes introduced in the initial model were for Objectivity (an object-oriented database management system) and included the special Relationship attributes and the ooObj class. Significant attention was given to the problem of organising the database properly: simply creating objects and putting them in the database is inefficient and causes major problems even for the small amount of data being dealt with here (hundreds of megabytes). In the prototype the data has been split into 119 databases for the intermediate astrometry data (HIPI and Abscissa classes in Fig. 1), and 23 databases for the reference great circle data (RGC class in Fig. 1). The partitioning in this case was arbitrary but in a final system something like the Hierarchical Triangular Mesh (Szalay & Brunner 1998) would be useful as a partitioning criterion. Initially it was attempted to decouple the implementation from Objectivity. However it became clear that the more one distances oneself, the less efficient the code is likely to be. It seems that embracing the database system and using its features efficiently is a better approach.

2.3 The Processing

A processing framework was set up to allow distributed processing of the database (Fig. 2). This is described in more detail in Sect. 5.1. Basically a processor could be started on a number of machines and would then perform a unit of work given to it by its coordinator. In this case the unit of parallel processing was a database. Java's RMI (Remote Method Invocation) framework was used for the communication between processes, but all data access and modification was done directly via the database. Remote access to the database is directly supported by Objectivity in a seamless manner.

Although the data is distributed across many databases the Abscissa objects are still referenced by the appropriate RGC object and vice versa (as depicted in the data model Fig. 1); the links are maintained by Objectivity and are easily traversable. One advantage of a system like Objectivity is that any of these individual databases can be made to reside on a different machine without any change to the software written using the database. Hence reorganisation when new processors and disks are available is facilitated — this will be essential for GAIA where highly distributed processing is foreseen.

The performance increase which should have resulted by using many processors was impeded by input/output on the server and by poor network performance, and so was only around a factor of two. Distribution of the databases and a high-speed network would improve this. One iteration takes approximately 17 hours at present and mean absolute update decreases with each iteration. The processing converged to an acceptable level after three iterations at which point the mean update was significantly less than 0.1 milliarcsec for all five astrometric parameters.

3 DISTINCTION OF PROCESSING AND DATA

In traditional data processing systems the data and processing are kept quite separate. This may seem to contradict the OO approach where it is usual to attach processing close to the class definitions which the processes act on. However, embedding processing in the class definition can lead to an unwieldy class library which is burdened by both data access and processing. Any update to the database schema as well as any update to the processing software would both require a new release of the class library. Preserving the distinction between data and processing is probably essential in any data processing system.

For example in the prototype system there is a method calcHarmonics which calculates the harmonics for a given reference great circle. This is attached to the RGC class as this is the class which is at the correct level for this processing — it has access to all abscissae required for the calculation. In a production system this could become quite complex. Another problem exists in the manner of storage of the data. If an OO approach is being followed an obvious approach is to use an OO database. This however requires some modification of the classes in the data model, hence breaking one of the nicer claims of the OO approach which is the uniformity of the model across all phases of development. Moreover, a change of storage mechanism will require further changes to the model. This, coupled with the above problem of the processing software, would lead to any update becoming a non-trivial task.

Below an OO approach is outlined which, it is believed, can cope with these problems and allow large data modelling activities to continue all the way through the project life cycle. The basic notion here is to keep the schema of the data and its access methods at a relatively abstract level, and use these abstract class definitions as the units of data processing. Meanwhile a particular implementation can be tied to a database vendor and thus use all possible features of that system.

4 DATA MODEL AND ITS REALISATION

All software design approaches and standards insist on the necessity of a logical as opposed to a physical model of the system. This is enshrined in ESA's software engineering standards PSS-05-0 (ESA 1991) as part of the software requirements phase. Coad & Yourdon (1991) refer to it as the 'Problem Domain Model' which is worked on during the analysis phase. As systems grow in complexity this logical approach becomes ever more important in order to insulate one from underlying libraries and databases which may be used to implement the system. It is however usually left up to the reader how to best bring the logical model into physical reality and frequently, at the physical design stage, radical changes are made to the logical model. Below an approach is outlined for defining an OO data model which allows for many different implementations while presenting a consistent interface to clients. The goal is similar to that of 'The Bridge' framework as described in (Gamma et al. 1994), but the approach outlined here requires less effort and incurs no overhead at runtime.

4.1 The Interface

The Unified Modelling Language (UML) (Eriksson & Penker 1998) introduces the notion of an *interface*. This is like a class but it defines only a set of abstract operation signatures. Other classes may opt to implement this interface, which means they must provide the concrete implementation of the operation. In Java, Interface exists as a programming construct while in C++ this could be seen as an abstract class with only virtual operations defined. Operations of other classes can be done purely in terms of these Interfaces and thus remain completely independent of the actual implementation of the operations.

In a sense this seems like a duplication of effort as in one place the interface is defined and in another the concrete implementation, which looks very similar. The important thing to note however is that several implementations for an interface may exist, but other users of the interface need not know which implementation they are dealing with. A brief example is given in Fig. 3. This example shows two implementations of the interface, although many more could be available. UML notation is used here: solid lines with triangles mean Inheritance while the dashed line means there is an Implements relationship. It should be noted that the interface definitions refer to other interfaces: e.g. getRGC() of Abscissa returns an RGC rather than some particular implementation of it; likewise getAbcissae() of the RGC interface returns an array of abscissae. Other software written in the system should then deal with the interfaces only, and not directly with any implementation class.

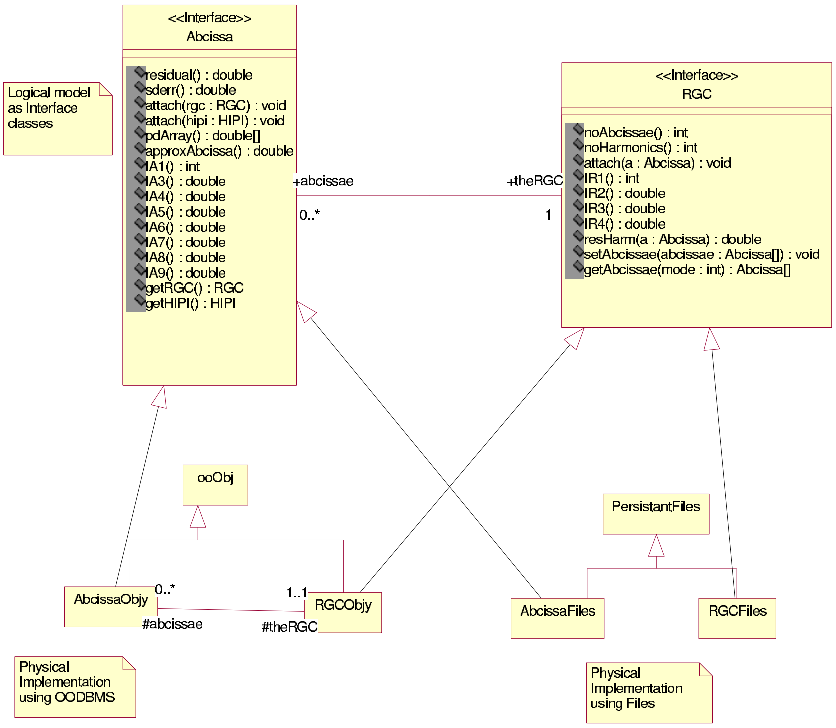

Figure 3: Interfaces and implementations

4.2 Initialisation

It is clear that the above scenario will function only after one has started — e.g. if a piece of code has reference to an RGC interface, then it will always get other interfaces: but how may a particular 'instance' be obtained? For this one or more 'factory' (Gamma et al. 1994) classes are needed. Let us say we have a particular interface Store, which must support creation, storage and retrieval of objects by using only their interfaces. The Store implementation must return the correct implementations for each interface. Effectively the Store encapsulates the database implementation, even if the database were just to be some files. When choosing the names for the methods it seems sensible to use names similar to those of the Database Class as defined by the Object Data Management Group (Catell et al. 1997).

However we still need some concrete way of getting this interface. There must be a class in the system, called InterfaceProvider here, which has a static method which returns the Store interface. This is the only real class defined in the model and it must decide which implementation to return when getStore is called. This should be indicated through the use of an environment variable or resource (Fig. 4).

The use of interfaces would allow one to define a complete data model and then publish a set of C++ abstract classes and a set of Java interfaces which

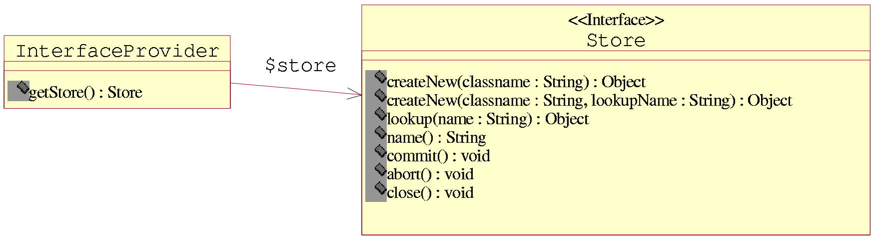

Figure 4: The InterfaceProvider and Store interface

any group could use in their development work with the confidence that they will have compatibility with later standard implementations. It would also allow an implementation to be tightly coupled to the database management system while still forcing it to provide the expected interface. In this way the implementation will comply with the logical model and the usual implementation changes to the logical model will not have to be made.

5 PROCESSING MODEL

In the present prototype the processing model is a global iterative approach — the algorithm is written to work on specific objects and must be applied to all objects. The traditional concept of a pipeline was not introduced to the prototype. However some level of coordination between the processing steps should be introduced, and this we see being met by an event handling system.

5.1 The Distribution Framework

In the prototype, Java's Remote Method Invocation (RMI) has been used as a basis to set up a simple framework for distribution of processing. There are three participants in the framework:

- One Coordinator runs on a server.
- Many Processors may run on one or more machines. A Processor registers with a particular Coordinator when it starts up and is told which data to process by the Coordinator. In the prototype this is the name of a database.
- Each Processor is started with a particular Process which it uses to process the data. The process actually connected to the database retrieves the data, processes it, and commits the result to the database. Note that in Fig. 5 this is the ProcessImp as the class Process already exists in Java.

In the prototype there is a clear distinction between the process which does something to data and the coordination between processes. Systems such as RMI or CORBA (Common Object Request Broker Architecture) are excellent

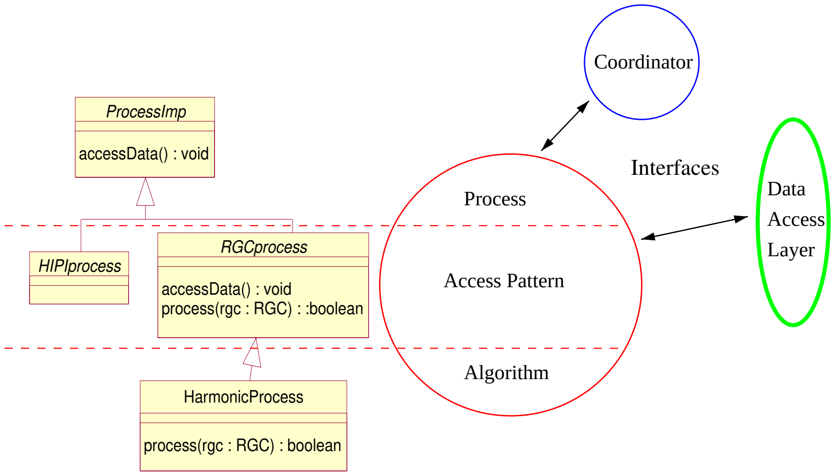

Figure 5: A three-tier process

for coordination of processes. However the transfer and control of data is best done directly against the database, which has excellent transaction management.

The process itself is broken down into three parts: the coordination part, the access pattern to the data, and the algorithm to be applied to the data. Abstract classes which have the knowledge for accessing a particular chunk of data are created, and concrete subclasses of these must be made which implement the method process. This allows a set of algorithms to share an access and distribution mechanism about which they know nothing — the algorithm is data driven, and acts upon data presented to it rather than requesting specific data. Improvements in the access or distribution lead to improvements for all algorithms using the access pattern. This may be represented as a three tier class hierarchy as depicted in Fig. 5. The first two tiers are abstract: only the bottom tier contains instantiatable classes which are capable of processing data. The big circle depicts the fact that through inheritance any bottom tier process contains the two tiers above; a given process is composed of the three essential parts while each part is quite independent and may be used by several subordinates.

In this framework no OO rules are broken. The algorithm is still tied to that class of objects because it is in a process method, the argument of which is a specific class (or classes). This link could (and probably should) be enforced further in the design by having a dynamic list of available process classes for a given class — a set of dynamic methods basically. For ease of use one may still provide methods such as calcHarmonics on a given class, but the implementation of this should probably be to call the most recent version of some HarmonicCalculator process.

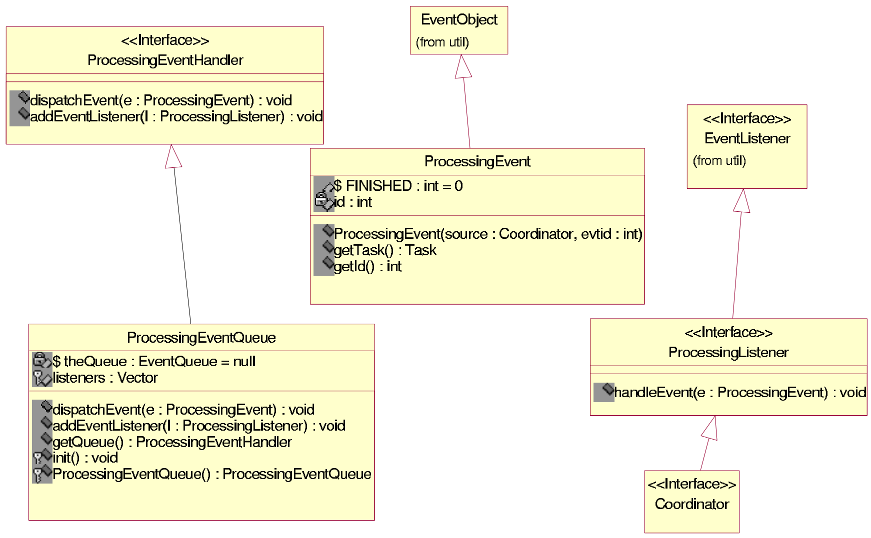

Figure 6: Event handler

5.2 System Event Handler

Here the notion would be that processes attach to an event handling system and notify it of particular events. Likewise, Listeners may attach to the event handler and receive notification of events. This is depicted in Fig. 6. This assumes a broadcast model for events. All ProcessingEvents are dispatched through the ProcessingEventHandler. The ProcessingEventQueue is a particular implementation of this. Classes implementing the ProcessingListener interface can be informed of events if they register with the ProcessingEventHandler. From the distribution model, for example, the Coordinator would need to be a ProcessingListener: it should also issue events when it starts, finishes, etc.

6 CONCLUSION

The global iterative approach seems feasible and appropriate for the GAIA data processing. Furthermore this is inherently distributable and an OO approach can help in allowing rapid development of a distributed processing framework. It is felt that the processing needs of GAIA would best be met by a 'farm' of relatively inexpensive machines connected by a high-speed network rather than by a large mainframe. Interestingly, several institutes are building cheap multi-processor arrays to do this type of processing (see for example Beowulf; http://cesdis.gsfc.nasa.gov/linux/beowulf/beowulf.html).

A simple framework was presented which could provide a solid basis for insulating processing algorithms from the complexity of distribution and data access, thus allowing different people to work on that part of the problem which best suits their background.

On the data handling side, use of interfaces to define the model will give good independence from data storage. Careful thought must however still go into the implementation. The object database helps by providing a transaction management and ease of distribution of data — a must for distributed processing.

REFERENCES

Catell R.G.G. et al. 1997, Object Database Standard: ODMG2.0, Morgan Kaufman

Coad P., Yourdon E. 1991, Object-Oriented Analysis, Prentice Hall

Eriksson H., Penker M. 1998, UML Toolkit, Wiley

ESA 1997, The Hipparcos and Tycho Catalogues, ESA SP–1200

ESA 1991, Software Engineering Standards, ESA PSS-05-0 Issue 2

Gamma E. et al. 1994, Design Patterns, Addison Wesley

Szalay A.S, Brunner R.J. 1998, Astronomical Archives of the Future: A Virtual Observatory. Accepted to a special issue of the Elsevier journal Future Generation Computer Systems, Dec. 1998 (astro-ph/9812335)