Creating an Object-Oriented Software System — The AIPS++ Experience

B.E. Glendenning

National Radio Astronomy Observatory Socorro, NM 87801

Abstract.

AIPS++ is a large astronomical data analysis system which is being developed using object-oriented techniques. It is being written in the C++ language. As the first such major effort in the astronomical community, and the largest to date, the AIPS++ project has discovered many issues — technical, cultural, and managerial — that other organizations may encounter in their own object-oriented efforts. This paper describes the good and bad experiences of the AIPS++ project in dealing with those issues.

1. AIPS++

1.1. Scope of AIPS++

AIPS++ (Astronomical Information Processing System) is intended to be the next-generation large software system for the radio astronomy community. It is to be the successor of the very successful AIPS (Astronomical Image Processing System) and UniPOPS (Unix People-Oriented Parsing System) packages. A summary of the AIPS project is available in Bridle & Greisen (1994), and information on UniPOPS is available from its home page[1].

AIPS++ is presently the largest strictly object-oriented post-processing software effort in the astronomical community. It started development in earnest at the beginning of 1992. This paper summarizes the general experience of the AIPS++ project with object-oriented development, it does not describe in detail the actual software designs or implementations. For the latter one can read papers by Glendenning (1994) and van Diepen (1994) who describe some important class subsystems in AIPS++, and by Schiebel (1996) and Shannon (1996) who describe user interface related parts of AIPS++, and by Crutcher et al. (1996) and Garwood (1996) who describe end-user applications. The creation of AIPS++, and its early development, is described by Croes (1993). Of course the latest information on AIPS++ should always be available from its home page[2].

AIPS++ is being developed by a consortium of seven observatories:

- Australia Telescope National Facility (ATNF), Australia.

[1] http://info.cv.nrao.edu/html/unipops/unipops-home.html

[2] http://info.cv.nrao.edu/aips++/docs/html/aips++.html

- Berkeley-Illinois-Maryland Association (BIMA), U.S.A.
- Herzberg Institute for Astrophysics (HIA), Canada.
- National Radio Astronomy Observatory (NRAO), U.S.A.
- Netherlands Foundation for Research in Astronomy (NFRA), the Netherlands.
- Nuffield Radio Astronomy Laboratory (NRAL) U.K.
- Tata Institute of Fundamental Research (TIFR), India.

The project has about 15 full-time equivalent (FTE) people working on it at the time of this writing (late 1995), about one half of which are provided by the NRAO. The ATNF, BIMA, and the NFRA presently provide the bulk of the additional manpower. The HIA, NRAL and TIFR are not at present actively developing AIPS++, although they intend to do so in the future. Most of the worlds important radio interferometers, and many important single dish radio telescopes, are operated by the institutions in the AIPS++ consortium.

Fundamentally, the purpose of AIPS++ is to calibrate, image, and analyze data from radio interferometers and single dish telescopes (interferometers and single dishes have traditionally had separate software packages). AIPS++ must do this for existing consortium instruments, instruments which are presently being constructed (notably NRAO's Green Bank Telescope (GBT)), and instruments of the future which are still in their planning stages (the Millimeter Array (MMA) and Square Kilometer Array Interferometer (SKAI)). AIPS++ must not only eventually replace the functionality of its predecessor packages, it must also handle hard "new" problems, for example, non-isoplanatic imaging and fitting models to three-dimensional spectral line cubes.

Besides the fundamental science capabilities of the package, it must also have the following features:

Programmable
1. It must be possible for the domain expert to introduce new calibration or imaging methods into the system without greatly perturbing it. Moreover, the system should be attractive for the expert to program in, so these new algorithms are made available to the user community as quickly as possible.
2. The astronomer should readily be able to perform arbitrary calculations on his data.

Free To encourage "tinkering" with algorithms and data analysis methods, and sharing of the results, the source code must be freely modifiable and redistributable. Most AIPS++ source code is available under the various GNU copyright licenses.

Modern This means a number of things, but the largest single element is having a GUI interface.

Near-real-time capable It must be possible to use AIPS++ while observing at a telescope. While this is presently most important for single dish astronomers, it is becoming increasingly more important at interferometers as well.

Portable A large software investment like AIPS++ must be long lived. To ensure this, it must be able to cope with changes in its underlying operating environment, for example from UNIX based systems, which presently dominate scientific data processing, to "PC" based operating systems, which may do so in the future.

Meeting these requirements with modest resources is an ambitious undertaking, and was the major impetus behind the decision to implement AIPS++ using object-oriented techniques.

1.2. Status of AIPS++

AIPS++ is a controversial project. Much of the controversy stems from the twinned observations that AIPS++ is being implemented with new techniques, and that the AIPS++ project is running considerably behind the schedule it originally promised. These observations engender a deep skepticism.

The AIPS++ project started in earnest in January 1992. Croes (1993) stated that the AIPS++ would begin constructing major applications by mid-1993. Based on this statement the AIPS++ project is two to three years behind schedule.

A review of the AIPS++ project by a panel of independent experts (in astronomical computing and computer science) was held in December of 1994. The panel made a number of important observations and recommendations in their report (Offen *et al.* 1994). I would summarize the most important of them as follows:

1. The overall management of the project was poor. The project management was significantly reorganized and simplified in the spring of 1995.

2. The ties of the project to the astronomical community were inadequate. This is slowly improving, but my impression is that this will (rightly) only significantly improve when parts of AIPS++ are widely available in the community.

3. The software that the project had written to that point is state-of-the-art and forward looking.

4. The project is important, and the project participants are capable of implementing it.

> "We believe that the success of the AIPS++ project is critical for the mid- and long-term future of radio-astronomy research and that technical expertise is available in sufficient quality and quantity to assure this success."

Since this paper is aimed at describing our experiences with object-oriented technology, I will concentrate on the technical issues we encountered in writing the software. I will, however, touch on some management issues that are directly related to object-orientation.

2. Object-Oriented Technology

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

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

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

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

In C++, polymorphism is achieved through inheritance (derived classes may substitute for base classes) and templates (generic types).[3]

The above short summary cannot hope to do justice to the subject. For more details on Object-Oriented design see Rumbaugh et al. (1991). For an introduction to C++ by the creator of the language, see Stroustrup (1991). For an excellent description of the C++ idioms necessary to build a complex, real-world system, see Barton & Nackman (1994).

An important point to make about object-oriented technology is that it is now in the computing mainstream. A number of years ago one had to endure expositions of object-oriented technology that bordered on the mystical.[4]

3. Development Issues

3.1. Revolution vs. Evolution

The AIPS++ project is a completely new implementation rather than an augmentation or partial reimplementation of an existing package. It is a revolution, not an evolution.

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

[4] This author's favorite example of such is a Journal of Object Oriented Programming editorial (October 1993) entitled "Object Frameworks: The Golden Path to Object Nirvana." The actual content of the editorial is quite sensible.

There are a number of advantages to an evolutionary approach. It is safer: one is making changes to an already working system. If some of the changes do not work out, one can revert the software to a previous version. It has a smaller impact on users — they do not have to learn an entirely new system. It may be cheaper — if large parts of the system are still satisfactory, they do not have to be recreated. On the other hand, if the whole system needs to be replaced in the end, it might be cheaper to do it all at once rather than integrating a large number of major changes into the system over time. This is essentially also the reason why a revolution might be required: it is very hard to make fundamental changes incrementally, since fundamental design decisions have far-reaching implications in a software system.

3.2. Personnel

There are two personnel issues that the AIPS++ project has faced.

The first is that no consortium member had on staff experts in object-oriented technology and/or C++, so we had to develop this experience within the project. It takes at least six months to develop this expertise, and each independent site needs an expert. However once these "gurus" are available, new programmers can become productive much more rapidly (almost immediately if they are implementing a class that has already been designed, or using classes which already exist). It would clearly have been advantageous for AIPS++ to have had object-oriented technology experts available from the inception of the project.

The other personnel issue that the AIPS++ project faced was that it had fewer "astronomer programmers" actively working on it than has been typical in astronomical data processing packages. For example, while 100% of the technical (programming) AIPS staff have a Ph.D. in Astronomy, only about one half of the AIPS++ staff do, and the fraction was only about 20% when the project started. Another similar observation is that this author is the only member of the AIPS++ project who has been a member of the AIPS project.

While this is not inherently a problem — diversity of backgrounds should (arguably) be a benefit — not having a strong complement of astronomical programmers, particularly experts in calibration and imaging, directly attached to the project limited progress in the areas that are most fundamentally important. This problem is exacerbated by the understandable reluctance of astronomers to become significantly involved with software which is being constructed and has no immediate benefit. While the problem has now been alleviated, it does point out that if development teams are split between current and new packages, that the personnel split should be chosen carefully: putting all the "new guys" on the "new package" is not the optimal technical result. Neither is it the optimum sociological or political result.

3.3. Design Methodology

A decision that a software project needs to make is whether it will use a formal software development methodology and any CASE tools that might be available to support it. For object-oriented software development the two most likely candidates are the Object-Modeling Technique (OMT) described by Rumbaugh *et al.* (1991), and the Booch Method described in Booch (1991). A fusion of these two methods appears to be likely (the two principals now work for the same company).

The AIPS++ project does not use such a methodology. It experimented with using OMT early in its development. It was not adopted at that time for two principle reasons. First, the software culture at our institutions did not include the use of such methodologies, individual programmers were reluctant to adopt it, and management was insufficiently certain of its efficacy to insist on its adoption. Secondly, at that time there were no good CASE tools that used our methodology of choice (OMT) on our computer platform of choice (Sun).

In practice, designs are communicated inside the project through a combination of informal English-language documents, OMT diagrams, and illustrative interface (".h") files.

Clearly very large software projects must use a formal design methodology; it seems equally clear that it is not necessary for small groups (one or two persons). It is unclear to me and most project members whether adopting one more forcefully would have been worthwhile for AIPS++. My suspicion is that, if the CASE tools have improved sufficiently, it would be of benefit for similarly sized projects to adopt such a methodology.

3.4. Distributed Development

AIPS++ developers are spread across several sites in three continents — four when all sites are contributing. Thus the issue of distributed development is one which the AIPS++ project had to face.

A critical issue is the distribution and maintenance of a common set of code and documentation. This is implemented via an elaborate set of homebrew utilities which automate synchronization of the slave sources with the master (typically weekly, but it can be as often as the slave site desires) along with utilities which allow programmers to check in and out (with locks) sources from the master repository.

Distributed collaborative design of complex sub-systems does not work well[5]. Our experience is that such designs must be arrived at during face to face meetings over a period of many weeks. On the other hand, distributed implementation of agreed upon designs can work quite well so long as the required infrastructure classes already exist. If the required infrastructure classes are being developed concurrently, the results are often poor.

3.5. Implementation Language Choice

There are two opinions about the importance of implementation language. The first is that it is relatively unimportant: the object-oriented design can be implemented in any language (which is an implementation detail). The other view holds that just as human language importantly shapes the ideas which can be formulated, the same is true of computer language. For systems like AIPS++ that have a library API as a major product, I believe the latter view is correct. Design constructs which do not map fairly directly into programming language constructs may be difficult to absorb by "third party" applications programmers.

[5]One wag suggests that the *First Law of Distributed Design* is: Don't do it.

Table 1. Tradeoffs among possible implementation languages.

Language	Object Support	Rapid Prototyping	Long Lived, Portable	Efficient	Complex
F77	Emulate[a]	N	Y	Y[c]	N
C	Emulate[a]	N	Y	Y	N
F90	Object-based[b]	N	Y	Y[c]	N
C++	OO	N	Y	Y	Y
Smalltalk	OO	Y	N	N	N
Eiffel	OO	N	N	Y	N
Java	Y	Y	N	N	N

[a]Objects may be emulated in procedural languages using techniques such as opaque structures (C) and indexing into "private" COMMON blocks (F77).

[b]Object-based languages provide support for encapsulation, but not inheritance or polymorphism.

[c]Fortran based languages are usually the fastest of all for numerics, largely because function arguments are assumed to be alias-free.

Some likely implementation languages are listed in Table 1. The listed languages are meant to be illustrative of the possible choices, not exhaustive. Languages which are listed as not long lived or widely portable might become so in the future of course. Java in particular may have a bright future.

The AIPS++ project chose C++ because:

1. We wanted a language which directly supported object-orientation; and
2. We wanted a language which was very likely to be widely available on any computer platform now and for a timescale of 10+ years.

However other projects might have different requirements. For example a project which is being developed for internal use only might choose a language like Eiffel which is less widely available but (arguably) a better language than C++.

3.6. C++ Issues

Language and Compiler Issues The C++ language is still undergoing international standardization. It is expected that it will become a Draft International Standard (DIS) in late 1996, and to become a ballotted International Standard (IS) in late 1997. Even though the formal standard is approximately two years from completion, as a pragmatic matter the language is rapidly stabilizing now[6].

The AIPS++ project decided to utilize a fairly large subset of what we expected the ultimate C++ language to be. In particular, we heavily use *templates*, and *exceptions* via a portable library and macro emulation. While the use of these features reduces the likelihood of a large-scale rewrite of our foundation classes, it greatly limits the number of compilers we are able to use. Most notably our library does not yet compile with the GNU C++ compiler, *g++*.

[6]For example, the Extensions subcommittee disbanded in early 1995

Another problem has been performance of the compilers we have been able to use. While newer compilers are much faster than the CFront based compilers we originally (and still) used, the compile and link times are still slower than C or Fortran programmers are used to. This is largely caused by the use of templates. The compilers themselves often have bugs that can be awkward to work around, and C++ often does not work well with tools such as debuggers and profilers.

Application Performance A concern about whether object-oriented programming and C++ necessarily result in slow applications is often voiced. So far our experience indicates that execution time can be optimized adequately — comparable to FORTRAN.

The main thing that causes C++ (and probably object-oriented languages in general) programs to run dramatically more slowly than their procedural equivalents is the manipulation of very many small objects, especially when they are frequently created and destroyed. In practice this does not seem to cause too much difficulty because:

1. Astronomy tends to have a fairly small number of fairly large objects (*e.g.*, Image), so the amortized overhead is relatively low; and

2. C++ has many features which allow these overheads to be optimized (*e.g.*, optimizing heap allocation on a per-class basis).

Probably the most important performance issue has nothing to do with object-orientation: it is the pointer aliasing problem which is inherited from C[7]. There are a number of observations to be made about this problem:

1. There is no reason why Fortran should not be used for hotspots. Use Fortran where it is strong (computing on regular arrays), and C++ where it is strong (complicated "bookkeeping").

2. An array class with alias-free properties will probably become part of the standard C++ library. This array class ("valarray") is not very full-featured, but it will be possible to layer higher-level classes on top of it.

Language Complexity C++ is a complex language. This complexity largely results from two decisions made during the creation and evolution of C++. First, C++ is largely a superset of C, and inherits many of C's quirks (*e.g.*, promotion of arrays to pointers, complex declaration syntax). Second, C++ is a multi-level (systems programming or high-level programming) multi-paradigm (procedural, object-based, or object-oriented) programming language. The accommodations required to allow all the different types of programming available in C++ necessarily increases the complexity of the language. Of course, it also increases the expressive power of the language.

[7] In brief, C does not have a first-class array type. Array arguments degenerate into pointers when passed to a function. Compilers cannot usually track the pointers beyond function boundaries, hence they must make the pessimistic assumption that the pointers might not be pointing at uniquely referenced storage which means that several optimizations might be lost (usually memory must be queried and set more often, rather than being left in a register).

There is no question that mastering all or or most of C++ is a considerable undertaking. However, it is also true that most programmers should not need to master all of C++. Writing programs with existing classes requires very much less knowledge than that needed to create classes. Similarly, creating "high-level" classes is very much easier than creating the foundation classes they are built upon. Of course we hope that much of the time end-users and programmers will be able to use the AIPS++ scripting language, Glish (Schiebel 1996), to program their *ad hoc* calculations and algorithmic explorations.

4. Conclusions

Proponents of object-oriented technology claim a number of benefits. After nearly four years of development, my opinion about those benefits are as follows:

Reliability I strongly believe this to be true. Encapsulation really does work in keeping bugs isolated and relatively easy to find and fix.

Reusability A great hope for object-oriented technology was that it would promote an industry of "software IC's:" well constructed classes which could be purchased, rather than reinvented. AIPS++ is unable to use commercial libraries by policy — our software must be freely redistributable. The freely available libraries were generally made archaic by rapid evolution in the language, but, more particularly, by the lack of standard library classes which greatly impedes the ability to profitably use third-party libraries[8]. While we were able to reuse some classes from the GNU libg++ library (*e.g.*, Complex and String) we were not able to save a significant amount of work. The library which is being developed as part of standardized C++ should be a boon in this regard.

On the other hand, internal reuse of the class libraries we have developed has worked very well.

Decoupled development As described in section 3.4. this can work very well once the design is finalized and the foundation classes are stable.

Productivity Since the AIPS++ project is behind schedule it would be perverse to claim we have seen any productivity gains. It is plausible that we might see them in the future since we have already climbed the learning curve and the technological problems are rapidly diminishing.

Maintainability No strong statement can be made since we are not in the maintenance phase of our project. However the factors that result in greater reliability should also improve the maintainability of the software.

The bottom line for me is that object-oriented technology allows one to write reliable software which is much more flexible (*i.e.*, complicated) than is possible with procedural programming. Since the problems we try to solve in

[8]For example, if every library has its own String class, it is tedious to call one library with the results from another.

software tend to become more difficult with time, object-oriented technology will become more common in the astronomical programming community, as it has in the wider programming community.

Because of its pioneering status, the AIPS++ project has faced many difficulties, most of which are rapidly disappearing as the industry progresses. Nevertheless I believe that our software has greatly benefited by using object-oriented technology and C++, and that this benefit will compound with time.

Acknowledgments. The National Radio Astronomy Observatory is a facility of the National Science Foundation operated under cooperative agreement by Associated Universities, Inc.

References

Barton, John J. & Nackman, Lee R. 1994, Scientific and Engineering C++ (Addison Wesley)

Booch, Grady 1991, Object Oriented Design With Applications (Benjamin Cummings)

Bridle, A.H. & Greisen, E.W. 1994, The NRAO AIPS Project — A Summary (AIPS Memo 87, NRAO)

Croes, G.A. 1993, in ASP Conf. Ser., Vol. 52, Astronomical Data Analysis Software and Systems II, ed. R. J. Hanisch, R. J. V. Brissenden & Jeannette Barnes (San Francisco: ASP), 156

Crutcher, Richard M., Baker, Pauline M., Baxter, George, Pixton, John & Ravlin, Harold 1996, this volume

Garwood, R.W. 1996, this volume

Glendenning, B.E. 1994, in ASP Conf. Ser., Vol. 61, Astronomical Data Analysis Software and Systems III, ed. Dennis R. Crabtree, R. J. Hanisch & Jeannette Barnes (San Francisco: ASP), 413

Offen, Ray, Brouw, Wim, Coggins, James, Cornwell, Tim, Gannon, Dennis & Hanisch, Bob 1994, AIPS++ Review: Report of the Review Panel (AIPS++ Memo 112)

Rumbaugh, James, Blaha, Michael, Premerlani, William, Eddy, Frederick & Lorensen, William 1991, Object-Oriented Modeling and Design (Prentice Hall)

Schiebel, D.R. 1996, this volume

Shannon, Paul 1996. this volume

Stroustrup, Bjarne 1991, The C++ Programming Language, Second Edition (Addison Wesley)

van Diepen, G. 1994, in ASP Conf. Ser., Vol. 61, Astronomical Data Analysis Software and Systems III, ed. Dennis R. Crabtree, R. J. Hanisch & Jeannette Barnes (San Francisco: ASP), 417