Software engineering practices in AIPS++

Cornwell and Kemball, eds.

August 16, 2000

Contents

1 Summary

This note contains a brief overview of the software engineering practices used in the AIPS++project.

2 Introduction

The software engineering practices and policies are sub-divided by category below.

3 Planning

The release schedule for AIPS++in the current operational phase is two releases per year, nominally scheduled for the months of April and October. This schedule establishes a natural development cycle lasting approximately six months. The planning activities during each development cycle proceed by the following stages, starting two weeks before the date the master distribution for the preceding release is finalized:

1. Each person formally responsible for a module or area of development in the system is asked to prepare a preliminary planning document describing the module priorities, current status and proposed development items for the coming development cycle, but without making any target assignments. A list of deferred or postponed development items from previous cycles is maintained within the project documentation system for reference, but the planning is zero-based, i.e. planning for each cycle is considered afresh with new items competing for scheduling with deferred or carry-over development items on the basis of their merit alone at the time of each planning cycle. This prevents planning inertia, and allows the project to be responsive to evolving user needs and external advice; however, it does retain continuity.

2. The collected planning proposals are circulated internally for general comment, and to allow other development items to be motivated from within the consortium.

3. After this period of comment, the development plan is finalized in discussions between the project management and the site managers, with final arbitration by the project management, whose job it is in this context to ensure that overall project goals and user requirements are met and external advice is heeded. Target descriptions and assignments will then be established from the agreed development items.

4. The plan becomes final when each site manager, after consultation with the site Director as required, accepts the agreed targets for their site. Each target constitutes a work package, and includes a full description of the

task, a time estimate, any other targets on which it depends, an assigned developer and completion criteria.

5. All deferred or postponed items are added to the documented list of items carried forward to the next development cycle, to be reconsidered at the start of the next planning cycle. A basic priority (short-, medium- or long-term) will be assigned to all deferred items, but are not binding due to the zero-based planning.

6. The targets for the current cycle are entered in the target tracking system, which records weekly progress reports from all developers with assigned targets. These weekly e-mail reports are an important communication mechanism within the project.

7. Progress on the plan is monitored and reported by the project management. Mid-course planning revisions are discouraged, unless a major planning deficiency is uncovered in implementation. This is to ensure that the project maintains a focus on delivering the agreed targets at the end of the development cycle. Items can be submitted at any time for consideration in the next planning cycle, and are added to the recorded list of possible future development items.

4 Releases

1. **Code masters:** The project operates with two code distribution masters, a development master used for active development, and a release master, established at the time of the last release.

2. **Patches:** Check-in to the release master is permitted only to the designated release manager, and is confined to patches which meet the following criteria:

   - Fix serious defects in capabilities included in the release.
   - Are adequately isolated to allow patch generation.

   Patches submitted by developers need to compile against the release master, as opposed to the development master. Each consortium site is expected accordingly to maintain a release build, or to use the project center release build to verify the correctness of any patch submitted. A mechanism is provided to end-users to update their binary releases with the accepted patches.

3. **OS support:** The operating systems supported for each release are established by project management in consultation with the consortium. These are chosen on the basis of their prevalence, at a given revision level, within the user community targeted by AIPS++. The project may lag the highest revision levels available for a given operating system by this criterion.

Binary releases are not necessarily prepared for all architectures used for development in the consortium. This depends on available resources.

4. **Developer support:** Developer releases, which can be updated by the standard consortium code distribution system, will be made periodically, subject to available resources. The primary, but not sole, consumers of AIPS++, however, are the scientific end-users of the package.

5. **Stable releases:** Stable releases are designated periodically during each development cycle. These are builds which meet acceptance standards, as certified by the Quality Assurance Group (QAG).

5 Code development

The following practices apply to the code development cycle.

5.1 Requirements

Scientific requirements for any given area of development are established in consultation with external users competent in the relevant area, in broad consultation within the consortium. The requirements are expected to be specific and well-defined, and to include their perceived scientific priority. Requirements may be sub-divided by instrument, as appropriate.

5.2 Design

The following level of design control and review is adopted:

1. **Initial design:** A design is expected to be drafted for each component under development for any given target. This should constitute the **.h** files, explanatory notes and may be accompanied by a thin prototype where appropriate.

2. **Design review:** A review of the initial component design will be conducted by the developer responsible for the module or area of development for consistency with the overall module design. This may include a higher level review for consistency with the project design or integration standards, as noted below.

3. **Documentation:** All designs need to be documented in the project documentation on acceptance.

4. **Integration:** All module or component designs need to be consistent with higher-level integration policies, and the overall design established for the project. This is arbitrated at higher levels of coordination responsibility assigned within the project, including project management.

5.3 Implementation

During implementation, the following policies are applicable:

1. **Coding practices:** Coding conventions, standards and practices are set by the technical leader, on advice from the QAG.

2. **Project compiler:** The project has a policy of a single project compiler in order to ensure maximum build stability so that applications can be delivered on time to end-users. All developers are expected to use the project compiler for development, and all code checked-in needs to compile with this designated compiler. The project compiler is set periodically subject to the following criteria:

   - An improvement over the current project compiler in reliability, language support or optimization, or a combination thereof.
   - Operation on all consortium development architectures, or those planned for the release.
   - Inter-operability with current development tools used in the project.

3. **Secondary compilers:** Secondary compilers may be designated by the technical leader if resources are available for their support. Code only needs to be checked against the project compiler, however, as noted above. A designated group will be responsible for ensuring a clean build on each secondary system, and are expected to resolve the overwhelming majority of any compiler or syntax support issues on these systems. Other developers may however be consulted for assistance or advice if changes need to be made to their code, particularly if these changes are subtle. In the event of syntax support conflicts, the C++ standard takes precedence unless not supported by the project compiler in the specific instance. Compiler defects are expected to be flagged with #ifdef statements for ease of later removal. Compiler defects need to be recorded for reference by other developers, and submitted to the vendor for correction.

4. **Code development tools:** The project tests code development tools, such as memory leak detectors, periodically. If they work with the AIPS++system, and are useful, they are added to the list of recommended tools for use in debugging or testing. Consortium sites are not required to purchase these tools however, although passing tests with some of these tools may be part of the code review standards recommended by the QAG.

5. **Code re-use:** Developers are expected to re-use AIPS++library code wherever possible, without duplicating core library capabilities.

5.4 Code check-in and check-out

All code is under revision control on each master. Check-in and check-out proceed by the following guidelines:

Check-out: Any file which is being actively modified by a developer, as opposed to testing or evaluation, needs to be checked-out with locking enabled. Files should be locked for the minimum period of time consistent with completing the work required.

Check-in: Files checked back into the system need to be tested for compiler and run-time errors, and should form part of a self-consistent group of files, which will not break the build. This includes making changes in all other parts of the system which depend on the code which has been modified, for all except contributed code. In rare circumstances, check-in's which require multiple days and may break the system will be scheduled after consultation with the project. Code which breaks the daily build inconveniences other developers and delays applications delivery.

RCS logs: Descriptive RCS entries are required for each file on check-in. The general guidelines for RCS entries are: i) Descriptive, complete and readable. Useful to another developer; ii) Reference defect numbers which are being fixed, but provide a one-line defect title with the defect ID; and iii) Please avoid: i) "Fix typo"; ii) "Fix error"; iii) "Fix bug", or; iv) "Null Ctrl-D".

Changelog entries: The changelog aims to document changes to the code distribution at a level higher than file-based RCS entries, as currently added by ai. The changelog is intended to summarize a sequence of related RCS changes in an overarching description, accessible to both scientific end-users and other AIPS++ developers. The overall goals of the changelog in this context are: i) flag changes in AIPS++ which are of interest to end-users of AIPS++; or ii) highlight changes which are of relevance to other AIPS++ developers using the affected code. Changes which meet any of the following criteria require an entry: i) important changes in design; ii) changes in the user interface; iii) addition of new features or adoption of new algorithms; iv) correction of major errors found in the code; v) major implementation changes, including significant code re-organization. These principles are taken to cover new code and documentation added to the system, code removed from the system, modification of existing code, and changes required to correct defects. Changes which do not requires changelog entries include: i) Modest changes in implementation, not visible or relevant to other developers or end-users; ii) Editing of documentation; iii) Defect correction with limited impact outside of the affected code or module; iv) Minor design or user interface changes.

5.5 Testing

The AIPS++project requires unit test programs for all code, which are exercised automatically after specified builds, and the results recorded. These are used to track and ensure build correctness. Integration testing is regarded as equally important, and usually takes the form of Glish scripts, to ensure the inter-operability of higher-level modules. The following forms of testing are encouraged, in order:

- Testing against truth, including known outcomes or simulated data.
- Regression testing against previous results.
- Completeness testing.

For a specific testing
Check list:

- 1. Is test case(s) provided?
- 2. Does the provided test case cover all functionalities of code.
- 3. If no test case is provided, develop a test case.
- 4. Make sure the test case has correct inputs, which include data files, database records, configuration files, etc., exclude keyboard and mouse input, which are specified in the detailed test procedures.
- 5. Execute the test and create test reports and log files.
- 6. Bug reports.
- 7. Backup the test case.
- 8. Give an assessment.
- 9. List Features not be tested, give the reason, for example, no suitable input document.
- 10. Constraints.
- 11. Performance observation.

Test case development
A good test case has the following attributes.

- 1. It has a high probability of finding a bug.
- 2. It is repeatable.
- 3. It has clearly defined expected results and pass/fail criteria.
- 4. It should not be redundant.

Test case Template

- Test name
- Documant location

- Test configuration
- Test interdependencies
- Test objectives
- Test procedure
- Test result
- Notes

5.6 Documentation

Code documentation policies are established and publicized by the QAG. The utility **cxx2html** is used at present to extract C++ documentation tags and convert them to html. The code documentation policies include style guides to be used in document preparation.

5.7 Code acceptance

Code is initially developed in the **trial** area, and is migrated to **aips** on acceptance. The code review process is coordinated by the QAG. Code review is an important process, and time will be specifically scheduled for this purpose in the planning process.

6 Defect handling

All defects are tracked in a defect tracking system, currently ClearDDTS (Rational). Sub-queues are established for the different geographical regions (Europe, North America, and Australia at present). If not resolved by a regional representative, defects are forwarded to the main defect queue. Defects are graded by severity, subject to the following criteria:

Severity 1: A catastrophic and unrecoverable error.

Severity 2: Severely broken, and no work-around.

Severity 3: A defect that needs to be fixed, but there is a work-around.

Severity 4: A defect that has a small impact, and is easy to work around.

Severity 5: A minor error or enhancement request.

In addition to defect reporting, enhancement requests can also be made through the defect tracking system.

Defect handling etiquette is as follows:

- On receiving an assigned defect, developers are expected to respond with a message to the user acknowledging and opening the defect. The cause of the defect does not necessarily need to be determined at this stage.

- Any changes in severity need to be explained briefly to the user.

- The severity of the defect is its classification. Severity 1 and 2 defects require a rapid response.

- Each developer is expected to devote up to 20% of development time to defect correction, but no more. Defect time should be assigned as evenly as possible throughout the development cycle. If a large number of defects accumulate in a specific area, additional time may be scheduled in the development cycle as a separate target.

- Defects are not a mechanism for revising planning in mid-course or for assigning the time of fellow developers in the consortium. Defects are aimed at code already in the system, at some degree of completeness and advertised for use.

- All previously postponed defects will be re-opened at the start of each development cycle.

- All resolved defects need to be verified by the submitter or by a developer assigned in each geographic region to verify defects submitted from end-users. Verification should not be denied for frivolous reasons. In the event of dead-lock, project management will make a final decision on closing a particular defect.

- Defects submitted by developers within the project are held to a higher standard than those submitted by external users. They are expected to be accompanied by rudimentary debugging or differential testing.

7 Quality assurance group

The QAG consists of a code-cop, a chief tester, and an additional member. The primary goal of the QAG is to ensure code quality through the following mechanisms:

1. **Testing:** The chief tester is responsible for routine testing of the system, as described above, as well as interaction with external user test groups which may be formed, and the expansion of test programs or scripts within the package. Routine testing is expected to primarily take the form of automated testing, but with interactive testing as required.

2. **Code reviews and coding standards:** The code review process is coordinated by the QAG, who are also called upon to make recommendations to the technical leader regarding coding and documentation standards.

Recommended technical changes, if accepted, are scheduled through the standard planning procedure.

3. **Release certification:** The QAG is responsible for establishing standards for accepting both binary and stable releases, and for certifying that releases meet these standards.

8 External reviews

Two external review committees are envisaged for the project:

User committee: A user committee formed from the pool of active users, who have extensive practical experience in using the package will meet periodically with the primary purpose of commenting on the scientific priorities and capabilities from the user perspective. Members will rotate on the committee, with membership suggestions made by the Executive Committee and site managers.

Technical advisory committee: A smaller technical advisory committee will meet separately to comment on technical questions alone, consisting of members who have experience in developing astronomical software systems or have technical expertise in specific, relevant areas. Members can be nominated by the Executive Committee and site managers, and membership will rotate on the committee.

9 Technical coordination

Technical coordination and decisions are made by the designated technical leader, in consultation with a technical group and the project management. The technical group is charged with solving technical problems facing the project, summarizing technical questions for the project as a whole, and taking decisions regarding their solution. Significant changes of project-wide implication are expected to be handled through the change proposal mechanism described below, however, to allow general comment.

10 Change proposals

The role of change proposals is to formally motivate changes to project design or implementation, which have a significant impact within the project. Change proposals should be prepared in consultation with the project management. They allow a period of open discussion of the proposed change, and its formal adoption by the project. Acceptance of a change proposal does not imply immediate implementation however, and forms part of the planning stack considered at the start of each planning cycle, as discussed above.

11 Decision log

Project management will maintain a decision log recording project decisions, which will be available for reference within the consortium. This includes decisions taken after e-mail discussion, other telephone or direct meetings within the project, and decisions taken by project management or the technical group.

12 Non-consortium code development

Membership in the consortium is subject to acceptance by the Executive Committee. Code from non-consortium sites is checked-in to a contributed area, and will not be actively maintained by the project. Decisions about distribution of this code will be reviewed by project management at the time of each release.

13 Use of third-party libraries

The goal of the policy on third-party libraries is to ensure that the tradeoffs between the implementation advantages and the maintenance costs are considered carefully.

The policy is as follows:

- All consortium sites are expected to obtain and maintain current copies of approved external libraries.
- For core and consortium code, no external libraries beyond those currently approved may be used.
- If a new external library is to be used, a change proposal must be written and approved to extend the list of approved external libraries. The change proposal must justify why native AIPS++ facilities are not used or extended as needed.