Aegis
A Project Change Supervisor

User Guide

Peter Miller

.

DEDICATIONS

This user guide is dedicated to my wife
Mary Therese Miller
for all her love and support
despite the computers.

And to my grandmother
Jean Florence Pelham
1905 — 1992
Always in our hearts.

This document describes aegis version 1.4.D038
and was prepared March 30, 1993.

1. Introduction

The aegis program is a CASE tool with a difference. In the spirit of the UNIX† Opertaing System, the aegis program is a small component designed to work with other programs.

Many CASE systems attempt to provide everything, from bubble charts to source control to compilers. Users are trapped with the components supplied by the CASE system, and if you don't like one of the components (it may be too limited, for instance), then that is just tough.

In contrast, UNIX provides many components of a CASE system - compilers, editors, dependency tools (such as make), source control (such as SCCS). You may substitute the tool of your choice - gcc, jove, cake, rcs (to name a few) if you don't like the ones supplied with the system.

The aegis program adds to this list with software configuration management (SCM), and true to UNIX philosophy, the aegis program does not dictate the choice of any of the other tools (although it may stretch them to their limits).

1.1. What does aegis do?

Just what is software configuration management? This question is sufficiently broad as to require a book in answer. In essence, the aegis program is a project change supervisor. It provides a framework within which a team of developers may work on many changes to a program independently, and the aegis program coordinates integrating these changes back into the master source of the program, with as little disruption as possible. Resolution of contention for source files, a major headache for any project with more than one developer, is one of the aegis program's major functions.

It should be noted that the aegis program is a developer's tool, in the same sense as make or SCCS are developer's tools. It is not a manager's tool - it does not provide progress tracking or help with work allocation.

1.2. Why use aegis?

So why should you use the aegis program? The aegis program uses a particular model of the development of software projects. This model has a master source (or baseline) of a project, consisteng of several (possibly several hundred) files, and a team of developers creating changes to be made to this baseline. When a change is complete, it is integrated with the baseline, to become the new baseline. Each change must be atomic and self-contained, no change is allowed to cause the baseline to cease to work. "Working" is defined as passing its own tests. The tests are considered part of the baseline. Aegis provides support for the developer so that an entire copy of the baseline need not be taken to change a few files, only those files which are to be changed need to be copied.

The win in using the aegis program is that there are *O(n)* interactions between developers and the baseline. Contrast this with a master source which is being edited directly by the developers - there is *O(n!)* interactions between developers - this makes adding "just one" more developer a potential disaster.

Another win is that the project baseline always works. Always having a working baseline means that a version is always available for demonstrations, or those "pre-release snapshots" we are always forced to provide.

The above advantages are all very well - for management types. Why should Joe Average Programmer use the aegis program? Recall that SCCS provides file locking, but only for one file at a time. The aegis program provides the file locking, atomically, for the set of files in the change. Recall also that SCCS locks the file the instant you start editing it. This makes popular files a project bottleneck. The aegis program allows concurrent editing, and a resolution mechanism just before the change must be integrated, meaning fewer delays for J.A.Programmer.

† UNIX is a trademark of Bell Laboratories.

1.3. How to use this manual

This manual assumes the reader is already familiar with the UNIX operating system, and with developing software using the UNIX operating system and the tools available; terms such as *RCS* and *SCCS* and *make* (1) are not explained.

There is also the assumption that the reader is familiar with the issues surrounding team development of software; coordination and multiple copy issues, for example, are not explained.

This manual is broken into a number of sections.

Chapter 2
: describes how aegis works and some of the reasoning behind the design and implementation of the aegis program. Look here for answers to "Why does it..." questions.

Chapter 3
: is a worked example of how particular users interact with the aegis program. Look here for answers to "How do I..." questions.

Chapter 4
: is a discussion of how aegis interracts with the History Tool, and provides templates and suggestions for history tools known to work with aegis.

Chapter 5
: is a discussion of how aegis interracts with the Dependency Maintenance Tool (DMT), and provides templates and suggestions for DMTs known to work with aegis.

Chapter 6
: is a discussion of how aegis interacts with the Difference Tools, and provides templates and suggestions for difference tools known to work with aegis.

Chapter 7
: describes the project attributes and how the various parameters may be used for particular projects.

Chapter 8
: is a collection of helpful hints on how to use aegis effectively, based on real-world experience. This is of most use when initially placing projects under the supervision of the aegis program.

Appendix A
: is a quick reference for placing an existing project under aegis.

Appendix B
: is a glossary of terms.

2. How Aegis Works

Before you will be able to exploit aegis fully, you will need to know just what it is that aegis does, and why aegis does these things.

The aegis program provides a change control mechanism and a repository, a subset of what CASE vendors call Software Configuration Management (SCM). In order to fit into a software engineering environment, or any place software is written, aegis needs a clearly defined place is the scheme of things.

This chapter describes the model of the software development process embodied in the aegis program, it describes some of the deliberate design decisions made for aegis, it describes some of the things aegis will and wont do for you, and it describes the situations where aegis is most and least useful.

2.1. The Model

The model of the software development process used by aegis was not plucked from thin air in some ivory tower. It evolved and grew with time in a real-live software development environment, and it has continued to be used and developed. Unfortunately, this environment was the largest experienced by the author to date, and consisted of only about fourty software engineers hammering away at a project.[1]

2.1.1. The Baseline

Most CASE systems revolve around a repository. A place where *stuff* is kept. This *stuff* is the raw data that is chewed over to produce the final product, whatever that may be. This *stuff* is the preferred form for editing or composing or whatever.

In the aegis program, the repository is known as the *baseline* and the units of *stuff* are UNIX files. The aegis program makes no distinction between text and binary files, so both are supported.

The history mechanism which must be included in any repository function is not provided by the aegis program. It is instead provided by some other per-project configurable software, such as RCS. This means that the user may select the history tool most suited to any given project. It also means that aegis is that much smaller to test and maintain. You will not be able to have binary files in your baseline if the history tool of your choice can't cope with them.

The structure of the baseline is dictated by the nature of each project, with some minor exceptions. The aegis program attempts to make as few arbitrary rules as possible. There is one mandatory file in the baseline, and one mandatory directory. The file is called *config*, and contains the per-project configuration information; the directory is called *test*, and contains all of the the tests. The contents and structure of the *test* directory are also dictated by aegis.

The baseline in aegis has one particular attribute: it always works. It is always there to show off to visiting big-wigs, it is always there to grab a copy of and ship a "pre-release snapshot" to some overly anxious customer, it is always there to let upper management "touch and feel" the progress being made towards the next release.

You may claim that "works" is comfortably fuzzy, but it is not. The baseline contains not only the source of a project, but also the tests for a project. A baseline is defined to "work" if and only if it passes all of its own tests. The aegis program has mandatory testing, to ensure that all changes to the baseline are accompanied by tests, and that those tests have been run and are known to pass. This means that no change to the baseline may result in the baseline ceasing to work[2].

The model may be summarised briefly: it consists of a *baseline* (master source), updated through the agency of an *integrator*, who is in turn fed *changes* by a team of *developers*. These terms will be explained in the following sections. See figure 1 for a picture of how files flow around the system.

The baseline is a set of files including the source files for a projects, and also all derived files (such as generated code, binary files from the compiler, etc). All files in the baseline are consistent with each other.

Thus the baseline may be considered to the the *closure* of the source files, in mathematical terms. That is, it is the source files and all implications flowing from those source files, such as object files and executables. All files in the baseline are

[1] "Unfortunately," because many real-world projects are far larger, but the author has yet to experience, and understand, the issues and procedures required. (Think of problems, yes; think of real-world solutions, no.)

[2] Well, mostly. It is possible for this restriction to be relaxed if you feel there are special circumstances for a particular change. The danger is that a change will be integrated with the baseline when that change is not actually of acceptable quality.

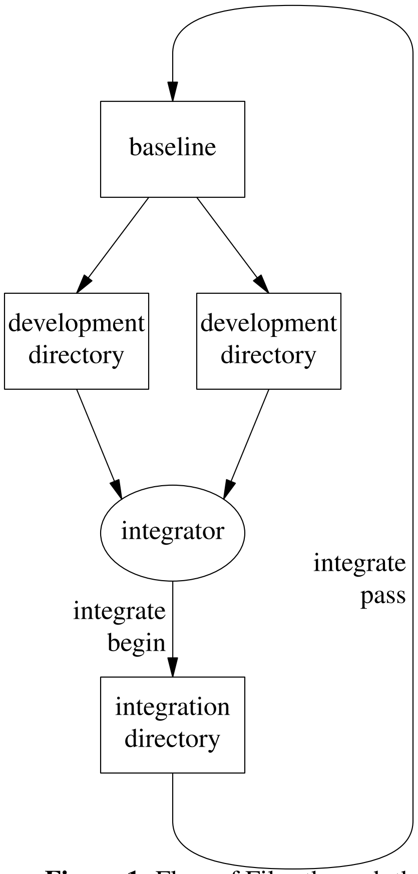

Figure 1: Flow of Files through the Model

consistent with each other; this means that development builds may expect to be able to take objects files from the baseline rather than rebuild them within the development directory.

The baseline is readable by all staff, and usually writable by none.

In many ways, the baseline may be thought of as a database, and all derived files are projections (views) of the source files. Passing its own tests may be thought of as input validation of fields. This is a powerful concept, and indeed the implementation of the aegis program performs many of the locking and synchronization tasks demanded of a database engine.

All of the files forming this database are text files. This means that they may be repaired with an ordinary text editor, should remedial action be necessary. The format is documented in section 5 of the reference manual. Should you wish to perform some query not yet available in aegis, the files are readily accessable to members of the appropriate UNIX group.

2.1.2. The Change Mechanism

Any changes to the baseline are made by atomic increments, known (unoriginally) as "changes". A change is a collection of files to be added to, modified in, or deleted from, the baseline. These files must all be so altered simultaneously for the baseline to continue to "work".[3]

For example, if the calling interface to a function were changed in one file, all calls to that function in any other file must also change for the baseline to continue to work. All of the files must be changed simultaneously, and thus must all be included in the one change. Other files which would logically be included in such an change include the reference manual entry for the function, the design document relating to that area of functionality, the relevant user documentation, tests would have to be included for the functionality, and existing tests may need to be revised.

Changes must be accompanied by tests. These tests will either establish that a bug has been fixed (in the case of a bug fix) or will establish that new functionality works (in the case of an enhancement).

2.1.3. Change States

As a change is developed using aegis, it passes through six states. Many aegis commands relate to transitions between these states, and aegis performs any validation at these times.

The six states of a change are described as follows, although the various state transitions, and their conditions, will be described later.

2.1.3.1. Awaiting Development

A change is in this state after it has been created, but before it has been assigned to a developer. This state can't be skipped: a change can't be immediately assigned to a developer by an administrator, because this disempowers the staff.

The aegis program is not a progress tracking tool, nor is it a work scheduling tool; plenty of both already exist.

[3] Whether to allow several logically independent changes to be included in the one change is a policy decision for individual projects to make, and is not dictated by the aegis program. It is a responsibility of reviewers to ensure that all new and changed functionality is tested and documented.

2.1.3.2. Being Developed

A change is in this state after it has been assigned to a developer, by the developer. This is the coal face: all development is carried out in this state. Files can be edited in no other state, this particularly means that only developers can develop, reviewers and integrators only have the power to veto a change. Staff roles will be described more fully in a later section.

To advance to the next state, the change must build successfully, it must have tests, and it must pass those tests.[4]

The new tests must also *fail* against the baseline; this is to establish that tests for bug-fixes actually reproduce the bug and then demonstrate that it is gone. New functionality added by a change will naturally fail when testsed in the old baseline, because it is not there.

When these conditions are met, the aegis program marks all of the changes files as locked, simultaneously. If any one of them is already locked, you can't leave the *being developed* state, because the file is part of a change which is somewhere between *being reviewed* and *being integrated*.

If any one of them is out-of-date with respect to the baseline, the lock is not taken, either. Locking the files at this state transition means that popular files may be modified simultaneously in many changes, but that only differences to the lastest version are ever submitted for integration. The aegis program provides a mechanism, described later, for bringing out-of-date files in changes up-to-date without losing the edits made by the developer.

2.1.3.3. Being Reviewed

A change is in this state after a developer has indicated that development is complete. The change is inspected, usually by a second party (or parties), to ensure that it matches the what it is meant to be doing, and meets other project or company standards you may have.

The style of review, and who may review, is not dictated by the aegis program. A number of alternative have been observed:

[4] It is possible for these testing requirements to be waived on either a per-project or per-change basis. How is described in a later section. The power to waive this requirement is not automatically granted to developers, as experience has shown that it is usually abused.

• You may have a single person who coordinates review pannels of, say, 4 peers, with this coordinator the only person allowed to sign-off review passes or fails.

• You may allow any of the developers to review any other developer's changes.

• You may require that only senior staff, familiar with large portions of the code, be allowed to review.

The aegis program enforces that a developer may not review their own code. This ensures that at least one other person other than the developer has scrutinized the code, and eliminates a rather obvious conflict of interest. It is possible to turn this requirement off on a per-project basis, but this is only desirable for projects with a one person team (or maybe two). The aegis program has no way of knowing that the user passing a review has actually looked at, and understood, the code.

The reviewer knows certain things about a change for it to reach this state: it has passed all of the conditions required to reach this state. The change compiles, it has tests and it passes those tests, and the changes are to the current version of the baseline. The reviwer may thus concentrate on issues of completeness, design, and standards - to name only a few. Check lists can be very helpful.

2.1.3.4. Awaiting Integration

A change is in this state after a reviewer has indicated that a change is acceptable to the reviewer(s). This is essentially a queue, as there may be many developers, but only one integration may proceed at any one time.

The issue of one integration at a time is a philosophical one: all of the changes in the queue are physically independent; because of the *Develop End* locking rules they do not have intersecting sets of files. The problem comes when one change would break another, in these cases the integrator needs to know which to bounce and which to accept. Integrating one change at a time greatly simplifies this, and enforces the "only change one thing at a time" maxim, occasionally at the expense of integrator throughput.

2.1.3.5. Being Integrated

A change is in this state when the integration of the change back into the baseline is commenced. A (logical) copy of the baseline is taken, and the change is applied to that copy. In this state, the change is compiled and tested once again.

The additional compilation has two purposes: it ensures that the successful compile performed by the developer was not a fluke of the developer's environment, and it also allows the baseline to be the closure of the sources files. That is, all of the implications flowing from the source files, such as object files and linked programs or libraries. It is not possible for aegis to know which files these are in the development directory, because aegis is decoupled from the build mechanism (this will discussed later).

To advance to the next state, the integration copy hust have been compiled, and the tests included in the change must have been run and passed.

The integrator also has the power of veto. A change may fail an integration because it fails to build or fails tests, and also just because the integrator says so. This allows the *being integrated* state to be another review state, if desired. The *being integrated* state is also the place to monitor the quality of reviews and reviewers.

Should a faulty change manage to reach this point, it is to be hoped that the integration process, and the integrator's sharp eyes, will detect it.

While most of this task is automated, this step is necessary to ensure that some strange quirk of the developer's environment was not responsible for the change reaching this stage. The change is built once more, and tested once more. If a change fails to build or test, it is returned to the developer for further work; the integrator may also choose to fail it for other reasons. If the integartor passes that change, the integrated version becomes the new baseline.

2.1.3.6. Completed

A change reaches this state when integration is complete. The (logical) copy of the baseline used during integration has replaced the previous copy of the baseline, and the file histories have been updated. Once in this state, a change may never leave it, unlike all other states.

If you wish to remove a change which is in this state from the baseline, you will have to submit another change.

2.1.4. The Software Engineers

The model of software development used by aegis has four different roles for software engineers to fill. These four roles may be overlapping sets of people, or be distinct, as appropriate for your project.

2.1.4.1. Developer

This is the coal-face. This role is where almost everything is done. This is the only role allowed to edit a source file of a project.

Most staff will be developers. There is nothing stopping a developer from also being an administrator, except for the possible conflict of interests with respect to testing exemptions.

A developer may edit many of the attibutes of a change while it is being developed. This is mostly useful to update the description of the change to say why it was done and what was actually done. A developer may not grant testing exemptions (but they may be relinquished).

2.1.4.2. Reviewer

The role of the reviewer is to check a developer's work. This review may consist of a peer examining the code, or it may be handled by a single member of staff setting up and scheduling multiperson review pannels. The aegis program does not mandate what style of review, it only requires that a reviewer pass or fail each change. If it passes, an integrator will handle it next, otherwise it is returned to the developer for further work.

In a large team, the reviewers are usually selected from the more senior members of the team, because of their depth of experience at spotting problems, but also because this is an opportunity for more senior members of staff to coach juniors on the finer points of the art.

The aegis programs makes some of the reviewer's task easier, because the reviewer knows several specific things about a change before it comes up for review: it builds, it has tests, and they have run successfully. There is also optional (per project) additional conditions imposed at the end of development, such as line length limits, or anything else which is automatically testable. The aegis program also provides a difference listing to the reviewer, so that each and every edit, to each and every file, can be pointed out to the reviewer.

There is nothing stopping a reviewer from being either an administrator or a developer. The aegis program specifically prevents a developer from reviewing his own work, to avoid conflicts of interest. (It is possible for this restriction to be waived, but that only makes sense for one person projects.)

It will occasionally be necessary to arbitrate between a developer and a reviewer. The appropriate person to do this would have line responsibility above both staff involved. Thus it is desirable that supervisors/managers not be reviewers, except in very small teams.

2.1.4.3. Integrator

The role of the integrator is to take a change which has already been reviewed and integrate it with the baseline, to form a new baseline. The integrator is thus the last line of defence for the baseline.

There is nothing preventing an administrator from being an administrator, a developer or a reviewer. The aegis program specifically prevents a developer or reviewer from integrating his own work, eliminating any conflict of interests. (It is possible for this restriction to be waived, but that only makes sense for one and two person projects.)

It will occasionally be necessary to arbitrate between an integrator and a reviewer and/or a developer. The appropriate person to do this would have line responsibility above all of the staff involved. Thus it is desirable that supervisors/mangers not be integrators, except in very small teams.

The baseline is readable by all developers, but not writable. All updates of the baseline to reflect changes produced by developers are performed through the agency of the intergator.

2.1.4.4. Administrator

The project administrator has the following duties:

• Create new changes. These may be the result of some customer bug reporting mechanism, it may be the result of new functionality being requested.

• Grant testing exemptions. By default, aegis insists that all changes be accompanied by tests. The project administrator may grant case-by-case exemptions, or a project-wide exemption.

• Add or remove staff. The four roles described in this section may be assigned to, or removed from, specific UNIX logins by the project administrator.

• Edit project attributes. There are many attributes attached to a project, only a project administrator may alter them.

• Edit change attributes. There are many attributes attached to a change, only a project administrator may alter all of them.

A project usually has only one or two administrators at any one time.

2.1.5. The Change Process

This section will examine the progression of a change through the six change states. Most of the attention will be given to the conditions which must be met in order to progress from one state to the next, as this is where the software development model employed by aegis is most often expressed.

See figure 2 for a picture of how all of the states and transitions fit together.

2.1.5.1. New Change

A project administrator creates a change. This change will consist mostly of a description at this time. The project administrator is not able (through aegis) to assign it to a specific developer.

The change is awaiting development; it is in the awaiting development state.

2.1.5.2. New Change Undo

It is possible to abandon a change if it is in the *awaiting development* state. All record of the change, including its description, will be deleted.

It is called *new change undo* to emphasize the state it must be in to delete it.

2.1.5.3. Develop Begin

A developer, for whatever reason, scans the list of changes awaiting development. Having selected a change, the developer then assignes that change to herself.

The change is now being developed; it is in the being developed state.

A number of aegis commands only work in this state, including commands to include files and tests in the change (be they new files to be added to the baseline, files in the baseline to be modified, or files to be deleted from the baseline), commands to build the change, commands to test the change, and commands to difference the change.

The process of taking sources files, the preferred form for editing of a project, and transforming them, through various manipulations and translations, into a "finished" product is known as building. In the UNIX world this usually means things like compiling and linking a program, but as fancy graphical programs become more wide-

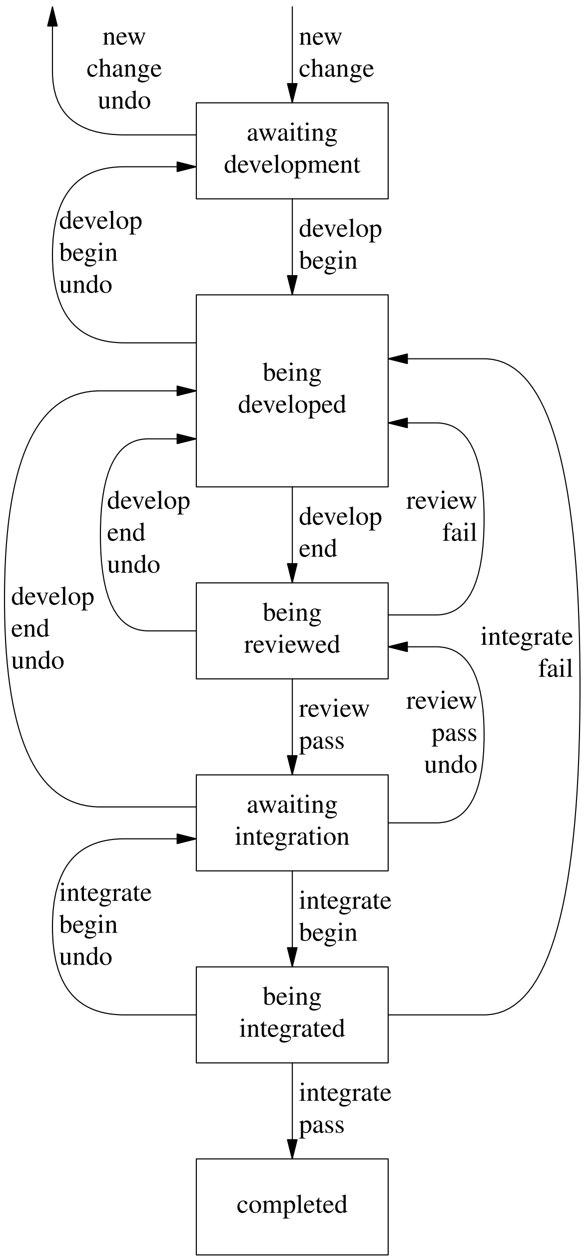

Figure 2: Change States and Transitions

spread, the source files could a binary output from a graphical Entity-Relationship-Diagram editor, which would then be run through a database schema generator.

The process of testing a change has three aspects. The most intuitive is that a test must be run to determine of the functionality works. The second requirement is that the test be run against the baseline and fail; this is to ensure that bugs are not just fixed, but reproduced as well. The third requirement is optional: all or some of the tests already in the baseline may also be run. Tests consist of UNIX shell scripts - anything that can be done in a shell script can be tested.

In preparation for review, a change is differenced. This usually consists of automatically comparing the present contents of the baseline with what the change proposes to do to the baeline, on a file-by-file basis. The results of the difference, such as UNIX *diff -c* output, is kept in a differnece file, for examination by the reviewer(s). The benefit of this procedure is that reviewers may examine these file to see every change the developer made, rather than only the obvious ones. The differencing commands are per-project configurable, and other validations, such as line length restrictions, may also be imposed at this time.

To leave this state, the change must have source files, it must have tests, it must have built sucessfully, it must have passed all its own tests, and it must have been differenced.

2.1.5.4. Develop Begin Undo

It is possible to return a change from the being developed state to the awaiting development state if it has no source files and has no tests. This is usually desired if a developer selected the wrong change by mistake.

2.1.5.5. Develop End

When the conditions for the end of development have been met (the change must have source files, it must have tests, it must have built sucessfully, it must have passed all its own tests, and it must have been differenced) the developer may cause the change to leave the being developed state and enter the being reviewed state. The aegis program will check to see that all the conditions are met at this time. There is no history kept of unsuccessful develop end attempts.

2.1.5.6. Develop End Undo

There are many times when a developer thinks that a change is completed, and goes hunting for a reviewer. Half way down the hall, she thinks of something that should have been included.

Iytt is possible for a developer to rescind a *Develop End* to allow further work on a change. No reason need be given. This request may be issued to a change in either the *being reviewed* or *awaiting integration* states.

2.1.5.7. Review Pass

This event is used to notify aegis that the change has been examined, by a method unspecified as discussed above, and has been found to be acceptable.

2.1.5.8. Review Pass Undo

The reviewer of a change may rescind a *Review Pass* while the change remains in the *awaiting integration* state. No reason must be supplied. The change will be returned to the *being reviewed* state.

2.1.5.9. Review Fail

This event is used to notify aegis that the change has been examined, by a method unspecified as discussed above, and has been found to be unacceptable.

A file containing a brief summary of the problems must be given, and will be included in the change's history.

The change will be returned to the *being developed* state for further work.

It is not the responsibility of any reviewer to fix a defective change.

2.1.5.10. Integrate Begin

This command is used to commence integration of a change into the project baseline.

Whether a physical copy of the baseline is used, or a logical copy using hard links, is controlled by the project configuration file. The change is then applied to this copy.

The integrator must then issue build and test commands as appropriate. This is not automated as some integrator tasks may be required in and around these commands.

2.1.5.11. Integrate Begin Undo

This command is used to return a change to the integration queue, with out prejudice. No reason need be given.

This is usually done when a particulary important change is in the queue, and the current integration is expected to take a long time.

2.1.5.12. Integrate Pass

This command is used to notify aegis that the change being integrated is acceptable.

The current baseline is replaced with the integration copy, and the history is updated.

2.1.5.13. Integrate Fail

This command is used to notify aegis that an integration is unacceptable, usually because it failed to build or test in some way, or sometimes because the integrator found a deficiency.

A file containing a *brief* summary of the problems must be given, and the summary will be included in the change's history.

The change will be returned to the *being developed* state for further work. The integration copy of the baseline is deleted, leaving the original baseline unchanged.

It is not the responsibility of any integrator to fix a defective change, or even diagnose what the defect may be.

2.2. Philosophy

The philosophy is simple, and that makes some of the implemmentation complex. When a change is in the *being developed* state, the aegis program is a developer's tool. Its purpose is to make it as easy for a developer to develop changes as possible. When a change leaves (or attempts to leave) the *being developed* state, the aegis program is protecting the project baseline, and does not exist to make the developer happy. The aegis program attempts to adhere to the UNIX minimalist philosophy. Least unnecessary output, least command line length, least dependence on 3rd party tools. No overlap in functionality of other tools.

2.2.1. Development

During the development of a change, the aegis program exists to help the developer. It helps him navigate around his change and the project, it copies file for him, and keeps track of the versions. It can even tell him what changes he has made.

2.2.2. Post Development

When a change has left the "being developed" state, or when it is attempting to leave that state, the aegis program ceases to attempt to help the developer and proceeds to defend the project baseline. The model used by aegis states that "the baseline always works", and aegis attempts to guarantee this.

2.2.3. Minimalism

The idea of minimalism is to help the user out. It is the intention that the aegis program can work out unstated command line options for itself, in cases where it is "safe" to do so. This means a number of defaulting mechanisms, all designed to help the user.

2.2.4. Overlap

It was very tempting while writing the aegis program to have it grow and cover source control and dependency maintenance roles. Unfortunately, this would have meant that the user would have been trapped with whatever the aegis program provided, and the aegis program is already plenty big. To add this functionality would have diverted effort, resulting in an inferior result. It would also have violated the underlying UNIX philosophy.

2.2.5. Design Goals

A number of specific ideas molded the shape of the aegis program. These include:

The UNIX philosophy of writing small tools for specific tasks with little or no overlap. Tools should bve written with the expectation of use in pipes or scripts, or other combinations.

• Stay out of the way. If it is possible to let a project do whatever it likes, write the code to let it. It is not possible to anticipate even a fraction of the applications of a software tool.

• People. The staff using aegis should be in charge of the development process. They should not feel that some machine is giving them orders.

• Users aren't psychic. Feedback must be clear, accurate and appropriate.

2.3. Security

Access to project data is controlled by the UNIX group mechanism. The group may be selected as suitable for your project, as may the umask.

All work done by developers (build, difference, etc) is all with a default group of the project's group, irrespective of the user's default group. Directores (when BSD semantics are available) are all created so that their contents default to the correct group. This ensures that reviewers and integrators are able to examine the change.

Other UNIX users not in the project's group may be excluded, or not, by the appropriate setting of the project umask. This umask is used by all aegis actions, assuring appropriate default behaviour.

A second aspect of security is to ensure that developers are unable to deliberately decieve aegis. At develop end, all files in the development directory are marked read only, aegis notes the time stamps on the files. Should the files be tampered with at any later date, aegis will notice. If a change is returned to the *being developed* state, the files are marked writable again.

2.4. Scalability

How big can a project get before aegis chokes? There are a huge number of variables in this question.

The most obvious bottleneck is the integrator. An artificial "average" example: Assume that the average integration takes an hour to build and test. A full-time integrator could be expected to get 7 or 8 of these done per day (this was the observed average on one project the author was involved in). Assume that the average time for a developer to develop a change is two weeks; a figure recommended by many text books as "the most you can afford to throw away". These two assumptions mean that for this "average" project aegis can cope with 70 to 80 developers, before integrations become a bottleneck.

The more serious bottle neck is the dependency maintenence tool. Seventy developers can churn out a staggering volume of code. It takes a very long time to wade through the file times and the rules, just to find the one or two files which changed. This can easily push the integrate build time past the one hour mark. Developers also become very vocal when build times are this long.

2.5. When (not) to use Aegis

The aegis program is not a silver bullet; it will not solve all of your problems. Aegis is suitable for some kinds of projects, useful for others, and useless for a few.

The most difficult thing about the aegis program is that it takes management buy-in. It takes effort to convince many people that the model used by aegis has benefits, and you need management backing you up when some person comes along with a way of developing software "without the extra work" imposed by the model used by aegis program.

There is extra up-front work: writing tests. The win is that the tests hang around forever, catching minor and major slips before they become embarrassing "features" in a released product. Prevention is cheaper than cure in this case, the tests save work down the track.

All of the "extra work" of writing tests is a long-term win, where old problems never again reappear. All of the "extra work" of reviewing changes means that another pair of eyes sights the code and finds potential problems before they manifest themselves in shipped product. All of the "extra work" of integration ensures that the baseline always works, and is always self-consistent. All of the "extra work" of having a baseline and separate developemnt directories allows multiple parallel development, with no inter-developer interference; and the baseline always works, it is never in an "in-between" state. In each case, not doing this "extra work" is a false economy.

The existence of these tests, though, is what determines which projects are most suited to aegis and which are not. It should be noted that suitability is a continuous scale, not black-and-white. With effort and resources, almost anything fits.

2.5.1. Projects for which Aegis is Most Suitable

Projects most suited to supervision by aegis are straight programs. What the non-systems-programmers out there call "tools" and sometimes "applications". These are programs which take a pile of input, chew on it, and emit a pile of output. The tests can then compare actual outputs with expected outputs.

As an example, you could be writing a *sed*(1) look-alike. It is a public domain clone of the UNIX sed utility. You could write tests which exercise every feature (insertion, deletion, etc.) and generate the expected output with the real UNIX sed. You write the code, and run the tests; you can immediately see if the output matches expectations.

This is a simple example. More compex examples exist, such as aegis itself. The aegis program is used to supervise its own development. Tests consist of sequences of commands and expected results are tested for.

Other types of software have been developed using aegis: compilers and interpreters, client-server model software, magnetic tape utilities, graphics software such as a ray-tracer. The range is vast, but it is not all types of software.

2.5.2. Projects for which Aegis is Useful

For many years there have been full-screen applications on text terminals. In more recent times there is increasing use of graphical interfaces.

In developing these types of programs it is still possible to use aegis, but several options need to be explored.

2.5.2.1. Limited Testing

You may choose to use aegis simply for its ability to provide controlled access to a large source. You still get the history and change mechanisms, the baseline model, the enforced review. You simply don't test all changes, because figuring out what is on the screen, and testing it against expectations, is too hard.

If the program has a command line interface, in addition to the full-screen or GUI interface, the functionality accessible from the command line may be tested using aegis.

It is possible that "limited testing" actually means "no testing", if you have no functionality accessable from the command line.

2.5.2.2. Testing Mode

Another alternative is to provide hooks into your program allowing you to substitute a file for user input, and to be able to trigger the dump of a "screen image". The simulated user input can then be fed to the program, and the screen dump (in some terminal-independent form) can be compared against expectations.

This is easier for full-screen applications, than for X11 applications. You need to judge the cost-benefit trade-off. Cost of development, cost of storage space for X11 images, cost of *not* testing.

2.5.2.3. Manual Tests

The aegis program provides a manual test facility. It was originally intended for programs which required some physical action from a user, such as "unplug ethernet cable now" or "mount tape XG356B now". It can also be used to have a user confirm that some on-screen activity has happened.

The problem with manual tests is that they simply don't happen. It is far more pleasant to say "run the automatic tests" and go for a cup of coffee, than to wait while the computer thinks of mindless things to ask you to do. This is human nature: if it can be automated, it is more likely to happen.

2.5.3. Projects for which Aegis is Least Suitable

Another class of software is things like operating system kernels and firmware; things which are "stand alone". This isolated nature makes it the most difficult to test: to test it you want to provide physical input and watch the physical output. By its very nature, it is hard to put into a shell script, and thus hard to write an aegis test for.

2.5.3.1. Operating Systems

It is not impossible, just that few of us have the resources to do it. You need to have a test system and a testing system: the test system has all of its input and outputs connected to the outputs and inputs of the testing system. That is, the testing system controls and drives the test system, and watches what happens.

For example, many operating system vendors test their products by using computers connected to each serial line to simulate "user input". The system can be rebooted this way, and using dual-ported disks allows different versions of a kernel to be tried, or other test conditions created.

For software houses which write kernels, or device drivers for kernels, or some other kernel work, this is bad news: the aegis program is probably not for you. It is possible, but there may be more cost-effective development strategies. Of course, you could always use the rest of aegis, and ignore the testing part.

2.5.3.2. Firmware

Firmware is a similar deal: you need some way to download the code to be tested into the test system, and write-protect it to simulate ROM, and have the necessary hardware to drive the inputs and watch the outputs.

As you can see, this is generally not available to run-of-the-mill software houses, but then they rarely write firmware, either. Those that do write formware usually have the download capabilities, and some kind of remote operation facility.

2.6. Further Work

The aegis program is far from finished. A number of features are known to lacking, and several interesting (but as yet unformed) ideas are lurking.

2.6.1. Hetrogeneous Environments

There is no support at the current time for tracking aspects of development in an heterogeneous environment. This is in the list of tasks, but has a low priority, first all the other stuff has to work! In time, attributes will be added to projects to indicate a set of environments, and each change will have to be built and tested in each.

The configuration file the user supplies to the dependency maintenance tool will be responsible for figuring out which environment it is in and building accordingly. It is an operating system requirement that the baseline and development directories are accessable in each environment (facilities such as RFS or NFS are assumed). Note that the baseline will expect to be self-consistent in all of the environments simultaneously.

There is no intention of ever providing the facility where a given file may have different versions depending on the environment, but all have the same name[5].

2.6.2. Hierarchy of Projects

It would be nice if there was some way to use one projects as a sort of "super change" to a "super project", so that large teams (say 1000 people) could work as lots of small teams (say 100 people). As a small team gets their chunk ready, using the facilities provided to-date by aegis, the small team's baseline is treated as a change to be made to the large team baseline.

This idea can be extended quite naturally to any depth of layering.

The desired semantics, let alone the implementation details, can not begin without more experience to show (and fix) the warts on the existing functionality.

2.6.3. Code Coverage Tool

It would be very helpful if a code coverage tool could be used to analyze tests included with changes to ensure that the tests actually exercised the lines of code changed in the change.

Another use of the code coverage tool would be to select regression tests based on the object files recompiled by a change, and those regression tests which exercise those files.

While there is freeware C code coverage tool available, based on GNU C, the interfacing and semantics still need more thought.

2.6.4. Branching

The aegis program does not provide support for branching in the history files. The semantics of the baseline, the source files and their closure, would appear to imply that a baseline is required for every leaf in the history tree.

Branching is provided in this way with the *new release* functionality. An entire new project is derived from an existing project, including another baseline. The history, however, is severed from the original project, precluding automatic merging by aegis at a later date. It also precludes haveing a single change to be applied to more than one branch.

Should better semantics become available, or a better algorithmic approach, this is certainly one area of aegis which could be improved.

2.6.5. Virtual File System

There is almost sufficient information in the aegis data base to create a virtual file system, overlaying the development directory atop the baseline[6]. This could be implemented similarly to automounters, intercepting file system operations by pretending to be an NFS server. Many commercial CASE products provide such a facility.

Such a virtual file system has a number of advantages: you don't need such a capable DMT, for starters; it only needs the dynamic include dependencies, and does not need a search path[7]. Second, many horrible and dumb compilers, notably FORTRAN and "fourth" GLs, don't have adequate include semantics; overlaying the two directories make this much easier to deal with[8]. Many graphical tools, such as bubble chart drawers, etc, when they do actually have include files, have no

[5] Some other SCM tools provide a repository with this ability.

[6] Reminiscent of Sun's TFS, but not the same.

[7] Discussed in the *Dependency Maintenance Tool* chapter.

[8] There are other ways, discussed in the *Tips and Traps* chapter.

command line specifiable search path.

The disadvantage is that this adds significant complexity to an already large program. Also, implementation is limited to NFS capable systems, or would have to be rewritten for a variety of other systems. The semantics of interactions between the daemon and other aegis commands, while clearly specifiable, are challenging to implement. Performance could also be a significant factor.

The question is "is it really necessary?" If the job can be done without it, does the effort of writing such a beast result in significant productivity gains?

3. The Change Development Cycle

As a change to a project is developed using aegis, it passes through several states. Each state is characterised by different quality requirements, different sets of applicable aegis commands, and different responsibilities for the people involved.

These people may be divided into four categories: developers, reviewers, integrators and administrators. Each has different responsibilities, duties and permissions; although one person may belong to more than one category, depending on how a project is administered.

This chapter looks at each of these categories, by way of an example project undergoing its first four changes. This example will be examined from the perspective of each category of people in the following sections.

There are six hypothetical users in the example: the developers are Pat, Jan and Sam; the reviewers are Robyn and Jan; the integrator is Isa; and the administrator is Alex[9]. There need not have been this many people involved, but it keeps things slightly cleaner for this example.

The project is called "example". It implements a very simple calculator. Many features important to a quality product are missing, checking for divide-by-zero for example. These have been omitted for brevity.

3.1. The Developer

The developer role is the coal face. This is where new software is written, and bugs are fixed. This example shows only the addition of new functionality, but usually as modifications of existing code, similar to bug-fixing activity.

3.1.1. The First Change

While the units of change, unoriginally, are called "changes", this also applies to the start of a project - a change to nothing, if you like. The developer of this first change will be Pat.

First, Pat has been told by the project administrator that the change has been created. How Alex created this change will be detailed in the "Administrator" section, later in this chapter. Pat then acquires the change and starts work.

```
pat% aedb -l -p example
Project "example"
List of Changes

Change  State           Description
------- -------         -------------
   1    awaiting_       Create initial skeleton.
        development
pat% aedb example 1
aegis: project "example": change 1: development directory "/u/pat/
        example.001"
aegis: project "example": change 1: user "pat" has begun development
pat% aecd
aegis: project "example": change 1: /u/pat/example.001
pat%
```

At this point aegis has created a development directory for the change and Pat has changed directory to the development directory[10].

Five files will be created by this change.

[9] The names are deliberately gender-neutral. Finding such a name starting with "I" is not easy!

[10] The default directory in which to place new development directories is configurable for each user.

```
pat% aenf config Howto.cook gram.y lex.l main.c
aegis: project "example": change 1: file "Howto.cook" added
aegis: project "example": change 1: file "config" added
aegis: project "example": change 1: file "gram.y" added
aegis: project "example": change 1: file "lex.l" added
aegis: project "example": change 1: file "main.c" added
pat%
```

The contents of the *config* file will not be described in this section, as it is covered in the *Project Configuration File* chapter. The contents of the *Howto.cook* file will not be described in this section, as it is covered in the *Dependency Maintenance Tool* chapter.

The file *main.c* will have been created by aegis as an empty file. Pat edits it to look like this

```
#include <stdio.h>

static void
usage()
{
        fprintf(stderr, "usage: example\n");
        exit(1);
}

void
main(argc, argv)
        int     argc;
        char    **argv;
{
        if (argc != 1)
                usage();
        yyparse();
        exit(0);
}
```

The file *gram.y* describes the grammar accepted by the calculator. This file was also created empty by aegis, and Pat edits it to look like this:

```
%token  DOUBLE
%token  NAME

%union
{
        int     lv_int;
        double  lv_double;
}

%type <lv_double> DOUBLE expr
%type <lv_int> NAME

%prec '+' '-'
%prec '*' '/'
%prec UNARY

%%
```

```
example
        : /* empty */
        | example command
command
        : expr
        | error
expr
        : DOUBLE
                { $$ = $1; }
        | '(' expr ')'
                { $$ = $2; }
        | '-' expr
                %prec UNARY
                { $$ = -$2; }
        | expr '*' expr
                { $$ = $1 * $3; }
        | expr '/' expr
                { $$ = $1 / $3; }
        | expr '+' expr
                { $$ = $1 + $3; }
        | expr '-' expr
                { $$ = $1 - $3; }
```

The file *lex.l* describes a simple lexical analyzer. It will be processed by *lex*(1) to produce C code implementing the lexical analyzer. This kind of simple lexer is usually hand crafted, but using lex allows the example to be far smaller. Pat edits the file to look like this:

```
%{
#include <math.h>
#include <gram.h>
%}
%%
[ \t\n]+        ;
[0-9]+(\.[0-9]*)?([eE][+-]?[0-9]+)?     {
                yylval.lv_double = atof(yytext);
                return DOUBLE;
        }
[a-z]   {
                yylval.lv_int = yytext[0] - 'a';
                return NAME;
        }
.       return yytext[0];
```

Note how the *gram.h* file is included using the `#include <filename>` form. This is very important for builds in later changes, and is discussed more fully in the *Tips and Traps* chapter.

The files are processed, compiled and linked together using the *aeb* command; this is known as *building* a change. This is done through aegis so that aegis can know the success or failure of the build. (Build success is a precondition for a change to leave the *being developed* state.) The build command is in the *config* file so vaguely described earlier. In this example it will use the *cook*(1) command which in turn will use the *Howto.cook* file, also alluded to earlier. This file describes the commands and dependencies for the various processing, compiling and linking.

```
pat% aeb
aegis: project "example": change 1: development build started
aegis: cook -b Howto.cook project=example change=1
        version=1.0.C001 -nl
cook: yacc -d gram.y
cook: mv y.tab.c gram.c
cook: mv y.tab.h gram.h
cook: cc -I. -I/projects/example/baseline -O -c gram.c
cook: lex lex.l
cook: mv lex.yy.c lex.c
cook: cc -I. -I/projects/example/baseline -O -c lex.c
cook: cc -I. -I/projects/example/baseline -O -c main.c
cook: cc -o example gram.o lex.o main.o -ll -ly
aegis: project "example": change 1: development build complete
pat%
```

The example program is built, and Pat could even try it out:

```
pat% example
1 + 2
3
^D
pat%
```

At this point the change is apparently finished. The command to tell aegis this is the *develop end* command:

```
pat% aede
aegis: project "example": change 1: no current 'aegis -DIFFerence'
        registration
pat%
```

It didn't work, because aegis thinks you have missed the difference step.

The difference step is used to produce files useful for reviewing changes, mostly in the form of context difference files between the project baseline and the development directory. Context differences allow reviewers to see exactly what has changed, and not have to try to track them down and inevitably miss obscure but important edits to large or complex files.

```
pat% aed
aegis: set +e; diff -c /dev/null /u/pat/example.001/Howto.cook >
        /u/pat/example.001/Howto.cook,D; test $? -eq 0 -o $? -eq 1
aegis: set +e; diff -c /dev/null /u/pat/example.001/config >
        /u/pat/example.001/config,D; test $? -eq 0 -o $? -eq 1
aegis: set +e; diff -c /dev/null /u/pat/example.001/gram.y >
        /u/pat/example.001/gram.y,D; test $? -eq 0 -o $? -eq 1
aegis: set +e; diff -c /dev/null /u/pat/example.001/lex.l >
        /u/pat/example.001/lex.l,D; test $? -eq 0 -o $? -eq 1
aegis: set +e; diff -c /dev/null /u/pat/example.001/main.c >
        /u/pat/example.001/main.c,D; test $? -eq 0 -o $? -eq 1
aegis: project "example": change 1: difference complete
pat%
```

Doing a difference for a new file may appear a little pedantic, but when a change consists of tens of files, so modifications of existing files and some new, there is a temptation for reviewers to use "more *,D" and thus completely miss the new files if it were not for this pedanticism[11].

[11] This is especially true when you use a tool such as *fcomp*(1) which gives a complete file listing with the inserts and deletes marked in the margin. This tool is also available from the author of aegis.

So that reviewers, and consciencious developers, may locate and view all of these difference files, the command

```
pat% more `find . -name "*,D" -print | sort`
...examines each file...
pat%
```

could be used, however this is a little too long winded for most users, and so the *aedmore* alias does exactly this. There is a similar *aedless* alias for those who prefer the *less* (1) command.

So now Pat is done, let's try to sign off again:

```
pat% aede
aegis: project "example": change 1: no current 'aegis -Test'
        registration
pat%
```

It didn't work, again. This time aegis is reminding Pat that every change must be accompanied by at least one test. This is so that the project team can be confident at all times that a project works[12]. Making this a precondition to leave the *being developed* state means that a reviewer can be sure that a change builds and passes its tests before it can ever be reviewed. Pat adds the truant test:

```
pat% aent
aegis: project "example": change 1: file "test/00/t0001a.sh" new
        test
pat%
```

The test file is in a weird place, eh? This is because many flavours of UNIX are slow at searching directories, and so aegis limits itself to 100 tests per directory. Whatever the name, Pat edits the test file to look like this:

```
#!/bin/sh
#
# test simple arithmetic
#
tmp=/tmp/$$
here=`pwd`
if [ $? -ne 0 ]; then exit 1; fi

fail()
{
        echo FAILED 1>&2
        cd $here
        rm -rf $tmp
        exit 1
}
```

[12] As discussed in the *Tips and Traps* chapter, aegis has the objective of ensuring that projects always work, where "works" is defined as passing all tests in the project's baseline. A change "works" if it passes all of its accompanying tests.

```
pass()
{
        cd $here
        rm -rf $tmp
        exit 0
}
trap "fail" 1 2 3 15

mkdir $tmp
if [ $? -ne 0 ]; then exit 1; fi
cd $tmp
if [ $? -ne 0 ]; then fail; fi

#
# with input like this
#
cat > test.in << 'foobar'
1
(24 - 22)
-(4 - 7)
2 * 2
10 / 2
4 + 2
10 - 3
foobar
if [ $? -ne 0 ]; then fail; fi

#
# the output should look like this
#
cat > test.ok << 'foobar'
1
2
3
4
5
6
7
foobar
if [ $? -ne 0 ]; then fail; fi

#
# run the calculator
# and see if the results match
#
$here/example < test.in > test.out
if [ $? -ne 0 ]; then fail; fi
diff test.ok test.out
if [ $? -ne 0 ]; then fail; fi
```

```
#
# this much worked
#
pass
```

There are several things to notice about this test file:

• It is a Bourne shell script. All test files are Bourne shell scripts because they are the most portable.[13] (Actually, aegis likes test files not to be executable, it passes them to the Bourne shell explicitly when running them.)

• It makes the assumption that the current directory is either the development directory or the baseline. This is valid, aegis always runs tests this way; if you run one manually, you must take care of this yourself.

• It checks the exit status of each and every command. It is essential that even unexpected and impossible failures are handled.

• A temporary directory is created for temporary files. It cannot be assumed that a test will be run from a directory which is writable; it is also easier to clean up after strange errors, since you need only throw the directory away, rather than track down individual temporary files. It mostly protects aginst rogue programs scrambling files in the current directory, too.

• Every test is self-contained. The test uses auxiliary files, but they are not separate source files (figuring where they are when some are in a change and some are in the baseline can be a nightmare). If a test want an auxiliary file, it must construct the file itself, in a temporary directory.

• Two functions have been defined, one for success and one for failure. Both forms remove the temporary directory. A test is defined as passing if it returns a 0 exit status, and failing if it returns anything else.

The most important feature to note about this test, after ignoring all of the trappings, is that it doesn't do much you wouldn't do manually! To test this program manually you would fire it up, just as the test does, you would give it some input, just as the test does, and you would compare the output against your expectations of what it will do, just as the test does.

The difference with using this test script and doing it manually is that most development contains many iterations of the "build, test, *think*, edit, build, test..." cycle. After a couple of iterations, the manual testing, the constant re-typing, becomes obviously unergonomic. Using a shell script is more efficient, doesn't forget to test things later, and is preserved for posterity (i.e. adds to the regression test suite).

This efficiency is especially evident when using commands[14] such as

```
pat% aeb && aet ; vi aegis.log
...
pat% !!
...
pat%
```

It is possible to talk to the shell extremely rarely, and then only to re-issue the same command, using a work pattern such as this.

As you have already guessed, Pat now runs the test like this:

---

[13] Portable for aegis' point of view: Bourne shell is the most widely available shell. Of course, if you are writing code to publish on USENET or for FTP, portablity of the tests will be important from the developer's point of view also.

[14] This is a *csh* specific example, unlike most others.

```
pat% aet
aegis: sh /u/pat/example.001/test/00/t0001a.sh
aegis: project "example": change 1: test "test/00/t0001a.sh"
        passed
aegis: project "example": change 1: passed 1 test
pat%
```

Finally, Pat has built the change, prepared it for review and tested it. It is now ready for sign off.

```
pat% aede
aegis: project "example": change1: no current 'aegis -Build'
        registration
pat%
```

Say what? The problem is that the use of *aent* cancelled the previous build registration. This was because aegis is decoupled from the dependency maintenance tool (*cook* in this case), and thus has no way of knowing whether or not the new file in the change would affect the success or failure of a build[15]. All that is required is to re-build, re-test, re-difference (yes, the test gets differenced, too) and sign off.

```
pat% aeb
aegis: logging to "/u/pat/example.001/aegis.log"
aegis: project "example": change 1: development build started
aegis: cook -b Howto.cook project=example change=1
        version=1.0.C001 -nl
cook: "all" is up-to-date
aegis: project "example": change 1: development build complete
pat% aet
aegis: logging to "/u/pat/example.001/aegis.log"
aegis: sh /u/pat/example.001/test/00/t0001a.sh
aegis: project "example": change 1: test "test/00/t0001a.sh"
        passed
aegis: project "example": change 1: passed 1 test
pat% aed
aegis: logging to "/u/pat/example.001/aegis.log"
aegis: set +e; diff -c /dev/null /u/pat/example.001/test/00/
        t0001a.sh > /u/pat/example.001/test/00/t0001a.sh,D; test
        $? -eq 0 -o $? -eq 1
aegis: project "example": change 1: difference complete
pat% aede
aegis: sh /usr/local/lib/aegis/de.sh example 1 pat
aegis: project "example": change 1: development completed
pat%
```

The change is now ready to be reviewed. This section is about developers, so we will have to leave this change at this point in its history. Some time in the next day or so Pat receives electronic mail that this change has passed review, and another later to say that it passed integration. Pat is now free to develop another change, possibly for a different project.

[15] Example: in addition to the executable file "example" shown here, the build may also produce an archive file of the project's source for export. The addition of one more file may push the size of this archive beyond a size limit; the build would thus fail because of the addition of a test.

3.1.2. The Second Change

The second chnage was created because someone wanted to name input and oitput files on the command line, and called the absence of this feature a bug. When Jan arrived for work, and lists the changes awaiting development, the following list appeared:

```
jan% aedb -l -p example
Project "example"
List of Changes

Change  State            Description
------  ------           ------------
   2    awaiting_        Add input and output file names to the
        development      command line.
   3    awaiting_        add variables
        development
   4    awaiting_        add powers
        development
jan%
```

The first on the list is chosen.

```
jan% aedb -c 2 -p example
aegis: project "example": change 2: development directory "/u/
        jan/example.002"
aegis: project "example": change 2: user "jan" has begun
        development
jan% aecd
aegis: project "example": change 2: /u/jan/example.002
jan%
```

The best way to get details about a change is to used the "change details" listing.

```
jan% ael cd
Project "example", Change 2
Change Details

NAME
        Project "example", Change 2.

SUMMARY
        file names on command line

DESCRIPTION
        Optional input and output files may be specified on the
        command line.

CAUSE
        This change was caused by internal_bug.

STATE
        This change is in 'being_developed' state.

FILES
        Change has no files.
```

```
HISTORY
        What            When            Who     Comment
        ------          ------          -----   ---------
        new_change      Fri Dec 11      alex
                        14:55:06 1992
        develop_begin   Mon Dec 14      jan
                        09:07:08 1992
jan%
```

Through one process or another, Jan determines that the *main.c* file is the one to be modified. This file is copied into the change:

```
jan% aecp main.c
aegis: project "example": change 2: file "main.c" copied
jan%
```

This file is now extended to look like this:

```
#include <stdio.h>

static void
usage()
{
        fprintf(stderr, "usage: example [ <infile> [ <outfile> ]]\n");
        exit(1);
}

void
main(argc, argv)
        int     argc;
        char    **argv;
{
        char    *in = 0;
        char    *out = 0;
        int     j;

        for (j = 1; j < argc; ++j)
        {
                char *arg = argv[j];
                if (arg[0] == '-')
                        usage();
                if (!in)
                        in = arg;
                else if (!out)
                        out = arg;
                else
                        usage();
        }
```

```
        if (in && !freopen(in, "r", stdin))
        {
                perror(in);
                exit(1);
        }
        if (out && !freopen(out, "w", stdout))
        {
                perror(out);
                exit(1);
        }

        yyparse();
        exit(0);
}
```

A new test is also required,

```
jan% aent
aegis: project "example": change 2: file "test/00/t0002a.sh" new
        test
jan%
```

which is edited to look like this:

```
#!/bin/sh
#
# test command line arguments
#
tmp=/tmp/$$
here=`pwd`
if [ $? -ne 0 ]; then exit 1; fi

fail()
{
        echo FAILED 1>&2
        cd $here
        rm -rf $tmp
        exit 1
}

pass()
{
        cd $here
        rm -rf $tmp
        exit 0
}
trap "fail" 1 2 3 15

mkdir $tmp
if [ $? -ne 0 ]; then exit 1; fi
cd $tmp
if [ $? -ne 0 ]; then fail; fi
```

```
#
# with input like this
#
cat > test.in << 'foobar'
1
(24 - 22)
-(4 - 7)
2 * 2
10 / 2
4 + 2
10 - 3
foobar
if [ $? -ne 0 ]; then fail; fi

#
# the output should look like this
#
cat > test.ok << 'foobar'
1
2
3
4
5
6
7
foobar
if [ $? -ne 0 ]; then fail; fi

#
# run the calculator
# and see if the results match
#
# (Use /dev/null for input in case input redirect fails;
# don't want the test to hang!)
#
$here/example test.in test.out < /dev/null
if [ $? -ne 0 ]; then fail; fi
diff test.ok test.out
if [ $? -ne 0 ]; then fail; fi
$here/example test.in < /dev/null > test.out.2
if [ $? -ne 0 ]; then fail; fi
diff test.ok test.out.2
if [ $? -ne 0 ]; then fail; fi

#
# make sure complains about rubbish
# on the command line
#
$here/example -trash < test.in > test.out
if [ $? -ne 1 ]; then fail; fi
```

```
#
# this much worked
#
pass
```

Now it is time for Jan to build and test the change. Through the magic of static documentation, this works first time, and here is how it goes:

```
jan% aeb
aegis: logging to "/u/pat/example.002/aegis.log"
aegis: project "example": change 2: development build started
aegis: cook -b /projects/example/baseline/Howto.cook
        project=example change=2 version=1.0.C002 -nl
cook: cc -I. -I/projects/example/baseline -O -c main.c
cook: cc -o example main.o /projects/example/baseline/gram.o
        /projects/example/baseline/lex.o -ll -ly
aegis: project "example": change 2: development build complete
jan% aet
aegis: logging to "/u/pat/example.002/aegis.log"
aegis: sh /u/jan/example.002/test/00/t0002a.sh
aegis: project "example": change 2: test "test/00/t0002a.sh"
        passed
aegis: project "example": change 2: passed 1 test
jan%
```

All that remains if to difference the change and sign off.

```
jan% aed
aegis: logging to "/u/pat/example.002/aegis.log"
aegis: set +e; diff -c /projects/example/main.c /u/jan/
        example.002/main.c > /u/jan/example.002/main.c,D; test $?
        -eq 0 -o $? -eq 1
aegis: project "example": change 2: difference complete
jan% aedmore
...examines the file...
jan%
```

Note how the context difference shows exactly what has changed. And now the sign-off:

```
jan% aede
aegis: project "example": change 2: no current 'aegis -Test
        -BaseLine' registration
jan%
```

No, it wan't enough. Tests must not only pass against a new change, but must fail against the project baseline. This is to establish, in the case of bug fixes, that the bug has been isolated *and* fixed. New functionality will usually fail against the baseline, because the baseline can't do it (if it could, you wouldn't be adding it!). So, Jan needs to use a variant of the *aet* command.

```
jan% aet -bl
aegis: sh /u/jan/example.002/test/00/t0002a.sh
usage: example
FAILED
aegis: project "example": change 2: test "test/00/t0002a.sh" on
        baseline failed (as it should)
aegis: project "example": change 2: passed 1 test
jan%
```

Running the regression tests is also a good idea

```
jan% aet -reg
aegis: logging to "/u/pat/example.002/aegis.log"
aegis: sh /projects/example/baseline/test/00/t0001a.sh
aegis: project "example": change 2: test "test/00/t0001a.sh"
        passed
aegis: project "example": change 2: passed 1 test
jan%
```

Now aegis will be satisfied

```
jan% aede
aegis: sh /usr/local/lib/aegis/de.sh example 2 jan
aegis: project "example": change 2: development completed
jan%
```

Like Pat in the change before, Jan will recieve email that this change passed review, and later that it passed integration.

3.1.3. The Third and Fourth Changes

This section will show two people performing two changes, one each. The twist is that they have a file in common.

First Sam looks for a change to work on and starts, like this:

```
sam% aedb -l
Project "example"
List of Changes

Change  State           Description
------- -------         -------------
   3    awaiting_       add powers
        development
   4    awaiting_       add variables
        development
sam% aedb 3
aegis: project "example": change 3: development directory "/u/
        sam/example.003"
aegis: project "example": change 3: user "sam" has begun
        development
sam% aecd
aegis: project "example": change 3: /u/sam/example.003
sam%
```

A little siniffing around reveals that only the *gram.y* grammar file needs to be altered, so it is compied into the change.

```
sam% aecp gram.y
aegis: project "example": change 3: file "gram.y" copied
sam%
```

The grammar file is changed to look like this:

```
%token DOUBLE
%token NAME
%union
{
        double  lv_double;
        int     lv_int;
};

%type <lv_double> DOUBLE expr
%type <lv_int> NAME
%left '+' '-'
%left '*' '/'
%right '^'
%right UNARY

%%
example
        : /* empty */
        | example command '0
                { yyerrflag = 0; fflush(stderr); fflush(stdout); }
        ;

command
        : expr
                { printf("%g0, $1); }
        | error
        ;

expr
        : DOUBLE
        | '(' expr ')'
                { $$ = $2; }
        | '-' expr
                %prec UNARY
                { $$ = -$2; }
        | expr '^' expr
                { $$ = pow($1, $3); }
        | expr '*' expr
                { $$ = $1 * $3; }
        | expr '/' expr
                { $$ = $1 / $3; }
        | expr '+' expr
                { $$ = $1 + $3; }
        | expr '-' expr
                { $$ = $1 - $3; }
        ;
```

The changes are very small. Sam checks to make sure using the difference command:

```
sam% aed
aegis: logging to "/u/sam/example.003/aegis.log"
aegis: set +e; diff -c /projects/example/baseline/gram.y /u/sam/
        example.003/gram.y > /u/sam/example.003/gram.y,D; test $?
        -eq 0 -o $? -eq 1
aegis: project "example": change 3: difference complete
sam% aedmore
...examines the file...
sam%
```

The difference file looks like this

```
*** /projects/example/baseline/gram.y
--- /u/sam/example.003/gram.y
***************
*** 1,5 ****
--- 1,6 ----
  %{
  #include <stdio.h>
+ #include <math.h>
  %}
  %token DOUBLE
  %token NAME

***************
*** 13,18 ****
--- 14,20 ----
  %type <lv_int> NAME
  %left '+' '-'
  %left '*' '/'
+ %right '^'
  %right UNARY
  %%
  example

***************
*** 32,37 ****
--- 34,41 ----
        | '-' expr
                %prec UNARY
                { $$ = -$2; }
+       | expr '^' expr
+               { $$ = pow($1, $3); }
        | expr '*' expr
                { $$ = $1 * $3; }
        | expr '/' expr
```

These are the differences Sam expected to see.

At this point Sam creates a test. All good software developers create the tests first, don't they?

```
sam% aent
aegis: project "example": change 3: file "test/00/t0003a.sh" new
        test
sam%
```

The test is created empty, and Sam edit it to look like this:

```
:
here=`pwd`
if test $? -ne 0 ; then exit 1; fi
tmp=/tmp/$$
mkdir $tmp
if test $? -ne 0 ; then exit 1; fi
cd $tmp
if test $? -ne 0 ; then exit 1; fi

fail()
{
        echo FAILED 1>&2
        cd $here
        chmod u+w `find $tmp -type d -print`
        rm -rf $tmp
        exit 1
}

pass()
{
        cd $here
        chmod u+w `find $tmp -type d -print`
        rm -rf $tmp
        exit 0
}
trap "fail" 1 2 3 15

cat > test.in << 'end'
5.3 ^ 0
4 ^ 0.5
27 ^ (1/3)
end
if test $? -ne 0 ; then fail; fi

cat > test.ok << 'end'
1
2
3
end
if test $? -ne 0 ; then fail; fi

$here/example test.in < /dev/null > test.out 2>&1
if test $? -ne 0 ; then fail; fi

diff test.ok test.out
if test $? -ne 0 ; then fail; fi

$here/example test.in test.out.2 < /dev/null
if test $? -ne 0 ; then fail; fi

diff test.ok test.out.2
if test $? -ne 0 ; then fail; fi
```

```
# it probably worked
pass
```

Everything is ready. Now the change cane be built and tested, just like the earlier changes.

```
sam% aeb
aegis: logging to "/u/sam/example.003/aegis.log"
aegis: project "example": change 3: development build started
aegis: cook -b /projects/example/baseline/Howto.cook
        project=example change=3 version=1.0.C003 -nl
cook: yacc -d gram.y
cook: mv y.tab.c gram.c
cook: mv y.tab.h gram.h
cook: cc -I. -I/projects/example/baseline -O -c gram.c
cook: cc -I. -I/projects/example/baseline -O -c /projects/
        example/baseline/lex.c
cook: cc -o example gram.o lex.o /projects/example/baseline/
        main.o -ll -ly -lm
aegis: project "example": change 3: development build complete
sam%
```

Notice how the yacc run produces a *gram.h* which logically invalidates the *lex.o* in the baseline, and so the *lex.c* file in the baseline is recompiled, using the *gram.h* include file from the development directory, leaving a new *lex.o* in the development directory. This is the reason for the use of

```
#include <filename>
```

directives, rather then the double quote form.

Now the change is tested.

```
sam% aet
aegis: logging to "/u/sam/example.003/aegis.log"
aegis: sh /u/sam/example.003/test/00/t0003a.sh
aegis: project "example": change 3: test "test/00/t0003a.sh"
        passed
aegis: project "example": change 3: passed 1 test
sam%
```

The change must also be tested against the baseline, and fail. Sam knows this, and does it here.

```
sam% aet -bl
aegis: logging to "/u/sam/example.003/aegis.log"
aegis: sh /u/sam/example.003/test/00/t0003a.sh
1,3c1,6
< 1
< 2
< 3
---
> syntax error
> 5.3
> syntax error
> 4
> syntax error
> 27
FAILED
aegis: project "example": change 3: test "test/00/t0003a.sh" on
        baseline failed (as it should)
aegis: project "example": change 3: passed 1 test
sam%
```

Running the regression tests is also a good idea.

```
sam% aet -reg
aegis: logging to "/u/sam/example.003/aegis.log"
aegis: sh /projects/example/baseline/test/00/t0001a.sh
aegis: project "example": change 3: test "test/00/t0001a.sh"
        passed
aegis: sh /projects/example/baseline/test/00/t0002a.sh
aegis: project "example": change 3: test "test/00/t0002a.sh"
        passed
aegis: project "example": change 3: passed 2 tests
sam%
```

A this point Sam has just enough time to get to the lunchtime aerobics class in the staff common room.

Earlier the same day, Pat arrived for work a little after Sam, and also looked for a change to work on.

```
pat% aedb -l
Project "example"
List of Changes

Change  State           Description
------- -------         -------------
   4    awaiting_       add variables
        development
pat%
```

With such a wide choice, Pat selected change 4.

```
pat% aedb 4
aegis: project "example": change 4: development directory "/u/
        pat/example.004"
aegis: project "example": change 4: user "pat" has begun
        development
pat% aecd
aegis: project "example": change 4: /u/pat/example.004
pat%
```

To get more information about the change, Pat then uases the "change details" listing:

```
pat% ael cd
Project "example", Change 4
Change Details

NAME
        Project "example", Change 4.

SUMMARY
        add variables

DESCRIPTION
        Enhance the grammar to allow variables. Only single
        letter variable names are required.

CAUSE
        This change was caused by internal_enhancement.

STATE
        This change is in 'being_developed' state.

FILES
        This change has no files.

HISTORY
        What            When            Who     Comment
        ------          ------          -----   ---------
        new_change      Mon Dec 14      alex
                        13:08:52 1992
        develop_begin   Tue Dec 15      pat
                        13:38:26 1992
pat%
```

To add the variables the grammer needs to be extended to understand them, and a new file for remembering and recalling the values of the variables needs to be added.

```
pat% aecp gram.y
aegis: project "example": change 4: file "gram.y" copied
pat% aenf var.c
aegis: project "example": change 4: file "var.c" added
pat%
```

Notice how aegis raises no objection to both Jan and Pat having a copy of the *gram.y* file. Resolving this contention is the subject of this section.

Pat now edits the grammar file.

```
pat% vi gram.y
...edit the file...
pat% aed
aegis: logging to "/u/pat/example.004/aegis.log"
aegis: set +e; diff -c /projects/example/baseline/gram.y /u/pat/
        example.004/gram.y > /u/pat/example.004/gram.y,D; test $?
        -eq 0 -o $? -eq 1
aegis: project "example": change 4: difference complete
pat%
```

The difference file looks like this

```
...
```

The new *var.c* file was created empty by aegis, and Pat edits it to look like this:

```
static double memory[26];

void
assign(name, value)
        int     name;
        double  value;
{
        memory[name] = value;
}

double
recall(name)
        int     name;
{
        return memory[name];
}
```

Little remains except to build the change.

```
pat% aeb
aegis: logging to "/u/pat/example.004/aegis.log"
aegis: cook -b /tmp/8508/example.proj/baseline/Howto.cook
        project=example change=4 version=1.0.C004 -nl
cook: yacc -d gram.y
cook: mv y.tab.c gram.c
cook: mv y.tab.h gram.h
cook: cc -I. -I/projects/example/baseline -O -c gram.c
cook: cc -I. -I/projects/example/baseline -O -c /projects/
        example/baseline/lex.c
cook: cc -I. -I/projects/example/baseline -O -c var.c
cook: cc -o example gram.o lex.o /projects/example/baseline/
        main.o var.o -ll -ly -lm
aegis: project "example": change 4: development build complete
pat%
```

A new test for the new functionality is required.

```
:
here=`pwd`
if test $? -ne 0 ; then exit 1; fi
tmp=/tmp/$$
mkdir $tmp
if test $? -ne 0 ; then exit 1; fi
cd $tmp
if test $? -ne 0 ; then exit 1; fi

fail()
{
        echo FAILED 1>&2
        cd $here
        chmod u+w `find $tmp -type d -print`
        rm -rf $tmp
        exit 1
}
pass()
{
        cd $here
        chmod u+w `find $tmp -type d -print`
        rm -rf $tmp
        exit 0
}
trap "fail" 1 2 3 15

cat > test.in << 'end'
a = 1
a + 1
c = a * 40 + 5
c / (a + 4)
end
if test $? -ne 0 ; then fail; fi

cat > test.ok << 'end'
2
9
end
if test $? -ne 0 ; then fail; fi

$here/example test.in < /dev/null > test.out 2>&1
if test $? -ne 0 ; then fail; fi

diff test.ok test.out
if test $? -ne 0 ; then fail; fi

$here/example test.in test.out.2 < /dev/null
if test $? -ne 0 ; then fail; fi

diff test.ok test.out.2
if test $? -ne 0 ; then fail; fi
```

```
# it probably worked
pass
```

The new files are then differenced:

```
pat% aed
aegis: logging to "/u/pat/example.004/aegis.log"
aegis: set +e; diff -c /projects/example/baseline/gram.y /u/pat/
        example.004/gram.y > /u/pat/example.004/gram.y,D; test $?
        -eq 0 -o $? -eq 1
aegis: set +e; diff -c /dev/null /u/pat/example.004/test/00/
        t0004a.sh > /u/pat/example.004/test/00/t0004a.sh,D; test
        $? -eq 0 -o $? -eq 1
aegis: set +e; diff -c /dev/null /u/pat/example.004/var.c > /u/
        pat/example.004/var.c,D; test $? -eq 0 -o $? -eq 1
aegis: project "example": change 4: difference complete
pat%
```

Notice how the difference for the *gram.y* file is still current, and so is not run again.

The change is tested.

```
pat% aet
aegis: logging to "/u/pat/example.004/aegis.log"
aegis: sh /u/pat/example.004/test/00/t0001a.sh
aegis: project "example": change 4: test "test/00/t0004a.sh"
        passed
aegis: project "example": change 4: passed 2 tests
pat%
```

The change is tested against the baseline.

```
pat% aet
aegis: logging to "/u/pat/example.004/aegis.log"
aegis: sh /u/pat/example.004/test/00/t0001a.sh
1,2c1,4
< 2
< 9
---
> syntax error
> syntax error
> syntax error
> syntax error
FAILED
aegis: project "example": change 4: test "test/00/t0004a.sh" on
        baseline failed (as it should)
pat%
```

And the regression tests

```
pat% aet -reg
aegis: logging to "/u/pat/example.004/aegis.log"
aegis: sh /projects/example/baseline/test/00/t0001a.sh
aegis: project "example": change 4: test "test/00/t0001a.sh"
        passed
aegis: sh /projects/example/baseline/test/00/t0002a.sh
aegis: project "example": change 4: test "test/00/t0002a.sh"
        passed
aegis: project "example": change 4: passed 2 tests
pat%
```

Note how test 3 has not been run, in any form of testing. This is because test 3 is part of another change, and is not yet integrated with the baseline.

All is finished for this change,

```
pat% aede
aegis: sh /usr/local/lib/aegis/de.sh example 4 pat
aegis: project "example": change 4: development completed
pat%
```

Anxious to get this change into the baseline, Pat now wanders down the hall in search of a reviewer, but more of that in the next section.

Some time later, Jan returns fram aerobics feeling much improved. All that is required for change 3 is to do develop end, or is it?

```
jan% aede
aegis: project "example": change 3: file "gram.y" in baseline
        has changed since last 'aegis -DIFFerence' command
jan%
```

A little sleuthing on Jan's part with the aegis list command will reveal how this came about. The way to resolve this problem is with the difference command.

```
jan% aed
aegis: logging to "/u/pat/example.003/aegis.log"
aegis: co -u'1.1' -p /projects/example/history/gram.y,v > /tmp/
        aegis.14594
/projects/example/history/gram.y,v  -->  stdout revision 1.1 (unlocked)
aegis: (diff3 -e /projects/example/baseline/gram.y /tmp/
        aegis.14594 /u/jan/example.003/gram.y | sed -e '/^w$/d'
        -e '/^q$/d';       echo '1,$p' ) | ed - /projects/example/
        baseline/gram.y > /u/jan/example.003/gram.y,D
aegis: project "example": change 3: difference complete
aegis: project "example": change 3: file "gram.y" was out of
        date, see "gram.y,D" for details
aegis: new 'aegis -Build' required
jan%
```

This was caused by the conflicty between change 4, which is now integrated, and change 3; both of which are editing the *gram.y* file. Jan examines the *gram.y,D* file, and discovers that it contains an accurate merge of the edit done by change 4 and the edits for this change. The difference file looks like this:

```
%{
#include <stdio.h>
#include <math.h>
%}
%token DOUBLE
%token NAME
%union
{
        double  lv_double;
        int     lv_int;
};

%type <lv_double> DOUBLE expr
%type <lv_int> NAME
%left '+' '-'
%left '*' '/'
%right '^'
%right UNARY

%%
example
        : /* empty */
        | example command '\n'
                { yyerrflag = 0; fflush(stderr); fflush(stdout); }
        ;

command
        : expr
                { printf("%g0, $1); }
        | NAME '=' expr
                { assign($1, $3); }
        | error
        ;
```

```
expr
        : DOUBLE
        | NAME
                { extern double recall(); $$ = recall($1); }
        | '(' expr ')'
                { $$ = $2; }
        | '-' expr
                %prec UNARY
                { $$ = -$2; }
        | expr '^' expr
                { $$ = pow($1, $3); }
        | expr '*' expr
                { $$ = $1 * $3; }
        | expr '/' expr
                { $$ = $1 / $3; }
        | expr '+' expr
                { $$ = $1 + $3; }
        | expr '-' expr
                { $$ = $1 - $3; }
        ;
```

This is because most such conflicts are actually working on logically separate portions of the file. Two diferent areas of the grammar in this case. In practice, there is rarely a real conflict, and it is usually small enough to detect fairly quickly. Notice that aegis did not automatically put the merge in place of your edited file, for just this reason.

Jan simply copies the difference file on top of the original, and rebuilds:

```
jan% mv gram.y,D gram.y
jan% aeb
aegis: logging to "/u/jan/example.003/aegis.log"
aegis: project "example": change 3: development build started
aegis: cook -b /tmp/13906/example.proj/baseline/Howto.cook
        project=example change=3 version=1.0.C003 -nl
cook: rm gram.c
cook: rm gram.h
cook: yacc -d gram.y
cook: mv y.tab.c gram.c
cook: mv y.tab.h gram.h
cook: rm gram.o
cook: cc -I. -I/projects/example/baseline -O -c gram.c
cook: rm lex.o
cook: cc -I. -I/projects/example/baseline -O -c /projects/
        example/baseline/lex.c
cook: rm example
cook: cc -o example gram.o lex.o /projects/example/baseline/
        main.o /projects/example/baseline/var.o -ll -ly -lm
aegis: project "example": change 3: development build complete
jan%
```

Notice how the list of object files linked has also adapted to the addition of another file in the baseline, without any extra work by Jan.

All that remains is to test the change again.

```
jan% aet
aegis: /bin/sh /tmp/13906/example.chan.3/test/00/t0003a.sh
aegis: project "example": change 3: test "test/00/t0003a.sh"
        passed
aegis: project "example": change 3: passed 1 test
jan%
```

And test against the baseline,

```
jan% aet -bl
aegis: /bin/sh /tmp/13906/example.chan.3/test/00/t0003a.sh
1,3c1,6
< 1
< 2
< 3
---
> syntax error
> 5.3
> syntax error
> 4
> syntax error
> 27
FAILED
aegis: project "example": change 3: test "test/00/t0003a.sh" on
        baseline failed (as it should)
aegis: project "example": change 3: passed 1 test
jan%
```

Perform the regression tests, too. This is important for a merged chnage, to make sure you didn't break the functionality you merged with.

```
jan% aet -reg
aegis: logging to "/u/jan/example.003/aegis.log"
aegis: /bin/sh /projects/example/baseline/test/00/
        t0001a.sh
aegis: project "example": change 3: test "test/00/t0001a.sh"
        passed
aegis: /bin/sh /projects/example/baseline/test/00/
        t0002a.sh
aegis: project "example": change 3: test "test/00/t0002a.sh"
        passed
aegis: /bin/sh /projects/example/baseline/test/00/
        t0004a.sh
aegis: project "example": change 3: test "test/00/t0004a.sh"
        passed
aegis: project "example": change 3: passed 3 tests
jan%
```

All done, or are we?

```
jan% aede
aegis: project "example": change 3: no current 'aegis -Diff'
        registration
jan%
```

The difference we did earlier, which revealed that we were out of date, does not show the differences since the two changes were merged, and possibly further edited.

```
jan% aed
aegis: logging to "/u/jan/example.003/aegis.log"
aegis: set +e; diff /projects/example/baseline/gram.y /u/pat/
        example.003/gram.y > /u/pat/example.003/gram.y,D; test $? -le 1
aegis: project "example": change 3: difference complete
jan%
```

This time everything will run smoothly,

```
jan% aede
aegis: project "example": change 3: development completed
jan%
```

Some time soon Jan will recieve email that this change passed review, and later that it passed integration.

Within the scope of a limited example, you have seen most of what aegis can do. To get a true feeling for the program you need to try it in a similarly simple case. You could even try doing this example manually.

3.1.4. Developer Command Summary

Only a few of the aegis commands available to developers have been used in the example. The following table (very tersely) describes the aegis commands most useful to developers.

Command	Description
aeb	Build
aeca	Change Attributes
aecd	Change Directory
aecp	Copy File
aecpu	Copy File Undo
aed	Difference
aedb	Develop Begin
aedbu	Develop Begin Undo
aede	Develop End
aedeu	Develop End Undo
ael	List Stuff
aenf	New File
aenfu	New File Undo
aent	New Test
aentu	New Test Undo
aerm	Remove File
aermu	Remove File Undo
aet	Test

You will want to read the manual entries for all of these commands. Note that all aegis commands have a −*Help* option, which will give a result very similar to the corresponding *man* (1) output. Most aegis commands also have a −*List* option, which usually lists interesting context sensitive information.

3.2. The Reviewer

The role of a reviewer is to check another user's work. You are helped in this by aegis, because changes can never reach the *being reviewed* state without several preconditions:

• The change is know to build. You know that it compiled sucessfully, so there is no need to search for syntax errors.

• The change has tests, and those tests have been run, and have passed.

This information allows you to concentrate on implementation issues, completeness issues, and local standards issues.

To help the reviewer, a set of "command D" files is available in the change development directory. Every files which is to be added to the baseline, removed from the baseline, or changed in some way, has a corresponding "comma D" file.

3.2.1. The First Change

Robyn finds out what changes are available for review by asking aegis:

```
robyn% aerpass -l -p example

Project "example"
List of Changes

Change  State           Description
------- -------         -------------
   1    being_reviewed  Place under aegis
robyn%
```

Any of the above changes could be reviewed, Robyn chooses the first.

```
robyn% aecd -p example -c 1
aegis: project "example": change 1: /u/pat/example.001
robyn% aedmore
...examines each file...
robyn%
```

The *aedmore* command walks the development directory tree to find all of the "comma D" files, and displays them using There is a corresponding *aedless* for those who prefer the command.

Once the change has bben reviewd and found acceptable, it is passed:

```
robyn% aerpass example 1
aegis: sh /usr/local/lib/aegis/rp.sh example 1 pat robyn
aegis: project "example": change 1: passed review
robyn%
```

Some time soon Isa will notice the email notification and commence integration of the change.

3.2.2. The Second Change

Most reviews have the same pattern as the first.

```
robyn% aerpass -l -p example

Project "example"
List of Changes

Change  State           Description
------- -------         -------------
   2    being_reviewed  file names on command line
robyn%
```

Always change directory to the change's development directory, otherwise you will not be able to review the files.

```
robyn% aecd -p example -c 2
aegis: project "example": change 2: /u/jan/example.002
robyn%
```

Another useful way of finding out about a change is the "list change details" command, viz:

```
robyn% ael cd -p example -c 2

Project "example", Change 2
Change Details

NAME
        Project "example", Change 2.

SUMMARY
        file names on command line

DESCRIPTION
        Optional input and output files may be specified on
        the command line.

CAUSE
        This change was caused by internal_bug.

STATE
        This change is in 'being_integrated' state.

FILES
        Type    Action  Edit    File Name
        ------- ------- ------- -----------
        source  modify  1.1     main.c
        test    create          test/00/t0002a.sh
```

```
HISTORY
        What            When            Who     Comment
        ------          ------          -----   ---------
        new_change      Fri Dec 11      alex
                        14:55:06 1992
        develop_begin   Mon Dec 14      jan
                        09:07:08 1992
        develop_end     Mon Dec 14      jan
                        11:43:23 1992
robyn%
```

Once Robyn knows what the change is meant to be doing, the files are then examined:

```
robyn% aedmore
...examines each file...
robyn%
```

Once the change is found to be acceptable, it is passed:

```
robyn% aerpass example 2
aegis: sh /usr/local/lib/aegis/rp.sh example 2 jan robyn
aegis: project "example": change 2: passed review
robyn%
```

Some time soon Isa will notice the email notification and commence integration of the change.

The reviews of the third and fourth changes will not be given here, because they are almost identical to the other changes. If you want to know how to fail a review, see the manual entry.

3.2.3. Reviewer Command Summary

Only a few of the aegis commands available to reviewers have been used in this example. The following table (very tersely) describes the aegis commands most useful to reviewers.

Command	Description
aecd	Change Directory
aerpass	Review Pass
aerpu	Review Pass Undo
aerfail	Review Fail
ael	List Stuff

You will want to read the manual entries for all of these commands. Note that all aegis commands have a *−Help* option, which will give a result very similar to the corresponding *man*(1) output. Most aegis commands also have a *−List* option, which usually lists interesting context sensitive information.

3.3. The Integrator

This section shows what the integrator must do for each of the changes shown to date. The integrator does not have the ability to alter anything in the change; if a change being integrated is defective, it is simply failed back to the developer. This documented example has no such failures, in order to keep it managebly small.

3.3.1. The First Change

The first change of a project is often the trickiest, and the integrator is the last to know. This example goes smoothly, and you may want to consider using the example project as a template.

The intergator for this example project is Isa. Isa knows there is a change ready for integration from the notification which arrived by email.

```
isa% aeib -l -p example

Project "example"
List of Changes

Change  State           Description
------- -------         -------------
   1    awaiting_       Place under aegis
        integration
isa% aeib example 1
aegis: project "example": change 1: link baseline to integration
        directory
aegis: project "example": change 1: apply change to integration
        directory
aegis: project "example": change 1: integration has begun
isa%
```

The integrator must rebuild and retest each change. This ensures that it was no quirk of the developer's environment which resulted in the success at the development stage.

```
isa% aeb
aegis: logging to "/projects/example/delta.001/aegis.log"
aegis: project "example": change 1: integration build started
aegis: cook -b Howto.cook project=example change=1
        version=1.0.D001 -nl
cook: yacc -d gram.y
cook: mv y.tab.c gram.c
cook: mv y.tab.h gram.h
cook: cc -I. -O -c gram.c
cook: lex lex.l
cook: mv lex.yy.c lex.c
cook: cc -I. -O -c lex.c
cook: cc -I. -O -c main.c
cook: cc -o example gram.o lex.o main.o -ll -ly
aegis: project "example": change 1: integration build complete
isa%
```

Notice how the above build differed from the builds that were done while in the *being developed* state; the extra baseline include is gone. This is because the integration directory will shortly be the new baseline, and must be entirely internally coinsistent and self-sufficient.

You are probably wondering why this isn't all rolled into the one aegis command. It is not because there may be some manual process to be performed after the build and before the test. This may be making a command set-uid-root (as in the case of aegis) or it may require some tinkering with the local oracle or ingres database. Instructions for the integrator may be placed in the description field of the change attributes.

The change is now re-tested:

```
isa% aet
aegis: logging to "/projects/example/delta.001/aegis.log"
aegis: sh /project/example/delta.001/test/00/t0001a.sh
aegis: project "example": change 1: test "test/00/t0001a.sh"
        passed
aegis: project "example": change 1: passed 1 test
isa%
```

The change builds and tests. Once Isa is happy with the change, perhaps after browsing the files, Isa then passes the integration, causing the history files to be updated andf the integration directory becomes the baseline.

```
isa% aeipass
aegis: logging to "/projects/example/delta.001/aegis.log"
aegis: ci -u -m/dev/null -t/dev/null /projects/example/delta.001/
        Howto.cook /projects/example/history/Howto.cook,v;
        rcs -U /projects/example/history/Howto.cook,v
/projects/example/history/Howto.cook,v  <--
        /projects/example/delta.001/Howto.cook
initial revision: 1.1
done
RCS file: /projects/example/history/Howto.cook,v
done
aegis: rlog -r /projects/example/history/Howto.cook,v | awk
        '/^head:/ {print $2}' > /tmp/aegis.15309
...lots of similar RCS output...
aegis: sh /usr/local/lib/aegis/ip.sh example 1 pat robyn isa
aegis: project "example": change 1: integrate pass
isa%
```

All of the staff involved, will receive email to say that the change has been integrated. This notification is a shell script, so USENET could be usefly used instead.

3.3.2. The Other Changes

There is no difference to integrating any of the later changes. The integration process is very simple, as it is a cut-down version of what the developer does, without all the complexity.

Your project may elect to have the integrator also monitor the quality of the reviews. An answer to "who will watch the watchers" if you like.

It is also a good idea to rotate people out of the integrator position after a few weeks in a busy project, this is a very stressful position. The position of integrator gives a unique perspective to software quality, but the person also needs to be able to say "NO!" when a cruddy change comes along.

3.3.3. Integrator Command Summary

Only a few of the aegis commands available to integrators have been used in this example. The following table (very tersely) describes the aegis commands most useful to integrators.

Command	Description
aeb	Build
aecd	Change Directory
aeib	Integrate Begin
aeibu	Integrate Begin Undo
aeifail	Integrate Fail
ael	List Stuff
aet	Test
aeupass	Integrate Pass

You will want to read the manual entries for all of these commands. Note that all aegis commands have a *−Help* option, which will give a result very similar to the corresponding *man*(1) output. Most aegis commands also have a *−List* option, which usually lists interesting context sensitive information.

3.4. The Administrator

The previous discussion of developers, reviewers and integrators has covered many aspects of the production of software using the aegis program. The administrator has responsibility for everything they don't, but there is very little left.

These responsibilities include:

• access control: The administrator adds and removes all categories of user, including administrators. This is on a per-project basis, and has nothing to do with UNIX user administration. This simply nominates which users may do what.

• change creation: The administrator adds (and sometimes removes) changes to the system. At later stages, developers may alter some attributes of the change, such as the description, to say what they fixed.

• project creation: The aegis program does not limit who may create projects, but when a project is created the user who created the project is set to be the administrator of that project.

All of these things will be examined

3.4.1. The First Change

Many things need to happen before development can begin on the first change; the projecty must be created, the staff but be given access permissions, the change must be created.

```
alex% aenpr example -dir /projects/example
aegis: project "example": project directory "/projects/example"
aegis: project "example": created
alex%
```

Once the project has been created, the project attributes are set. Alex creates a file called "fred" (it could have been called anything) and edits it to reflect the desired project attributes. It looks like this:

```
description = "Aegis Documentation Example Project";
developer_may_review = false;
developer_may_integrate = false;
reviewer_may_integrate = false;
```

Once this file has been created, the project attributes ae set as follows:

```
alex% aepa fred -p example
aegis: project "example": attributes changed
alex% rm fred
alex% ael p
List of Projects

Project Directory               Description
------- -----------             -------------
example /projects/example       Aegis Documentation Example
                                Project
alex%
```

The various staff must be added to the project. Developers are the only staff who may actually edit files.

```
alex% aend pat jan sam -p example
aegis: project "example": user "pat" is now a developer
aegis: project "example": user "jan" is now a developer
aegis: project "example": user "sam" is now a developer
alex%
```

Reviewers may veto c ahnge. There may be overlap between the various categories, as show here for Jan:

```
alex% aenr robyn jan -p example
aegis: project "example": user "robyn" is now a reviewer
aegis: project "example": user "jan" is now a reviewer
alex%
```

The next role we need to fill is an integrator.

```
alex% aeni isa -p example
aegis: project "example": user "isa" is now an integrator
alex%
```

Once the staff have been given access, Alex creates the descrfiption of the first change. The temporary file name is again "fred", and is edited to look like this:

```
brief_description = "Create initial skeleton.";
description = "A simple calculator using native \
floating point precision.  \
The four basic arithmetic operators to be provided, \
using conventional infix notation.  \
Parentheses and negation also required.";
cause = internal_enhancement;
```

Once this description file is created, the change is created as follows:

```
alex% aenc ncf -p example
aegis: project "example": change 1: created
alex% rm fred
alex%
```

At this point, Alex walks down the hall to Jan's office, to ask Jan to develop the first change. Jand has had some practice using aegis, and can be relied on to do the rest of the project configuarion speedily.

3.4.2. The Second Change

Some time later, Alex patiently sits through whining anf grumbing from an especially pedantic user. The following change description is duly entered:

```
brief_description = "file names on command line";
description = "Optional input and output files may be \
specified on the command line.";
cause = internal_bug;
```

The pedantic user wanted to be able to name files on the command line, rather than use I/O redirection. Also, having a bug in this example is useful. Alex is very systematic, and uses his trusty "fred" file again:

```
alex% aenc fred -p example
aegis: project "example": change 2: created
alex%
```

At some point a developer will notice this change and start work on it.

3.4.3. The Third Change

Other features are required for the calculator, and also for this example. The second change adds exponentiation to the calculator, and is described as follows:

```
brief_description = "add powers";
description = "Enhance the grammar to allow exponentiation.  \
No error checking required.";
cause = internal_enhancement;
```

The old-faithful fred file was used again:

```
alex% aenc fred -p example
aegis: project "example": change 3: created
alex%
```

At some point a developer will notice, and this change will be worked on.

3.4.4. The Fourth Change

A fourth change, this time adding variables to the calculator is added.

```
brief_description = "add variables";
description = "Enhance the grammar to allow variables.  \
Only single letter variable names are required.";
cause = internal_enhancement;
```

The old-faithful fred file was used again:

```
alex% aenc fred -p example
aegis: project "example": change 4: created
alex%
```

At some point a developer will notice, and this change will be worked on.

3.4.5. Administrator Command Summary

Only a few of the aegis commands available to administrators have been used in this example. The following table (very tersely) describes the aegis commands most useful to administrators.

Command	Description
aeca	set Change Attributes
ael	List Stuff
aena	New Administrator
aenc	New Change
aencu	New Change Undo
aend	New Developer
aeni	New Integrator
aenpr	New Project
aenrv	New Reviewer
aepa	set Project Attributes
aera	Remove Administrator
aerd	Remove Developer
aeri	Remove Integrator
aerpr	Remove Project
aerrv	Remove Reviewer

You will want to read the manual entries for all of these commands. Note that all aegis commands have a −*Help* option, which will give a result very similar to the corresponding *man*(1) output. Most aegis commands also have a −*List* option, which usually lists interesting context sensitive information.

3.5. What to do Next

This chapter has given a vicareous view of what using aegis feels like. As a next step in getting to know aegis, it would be a good idea if you created a project and went through this same exercise; you could use this exact example, or you could use a similar small project. This idea simply to run through many of the same steps as in the example. Typos and other natural events will ensure that you come across a number of situations not directly covered by this chapter.

If you have not already done do, a printed copy of the section 1 and 5 manual entries will be invaluable. If you don't want to use that many trees, they will be available on-line, by using the "-Help" option of the appropriate command variant. Try:

```
% aedb -help
...manual entry...
%
```

4. The History Tool

The aegis program is decoupled from the history mechanism. This allows you to use the history mechanism of your choice, SCCS or RCS, for example. You may even wish to write your own.

The intention of this is that you may use a history mechanism which suits your special needs, or the one that comes free with your flavour of UNIX operating system.

The aegis program uses the history mechanism for file *history* and so does not require many of the features of SCCS or RCS. This simplistic approach can sometimes make the interface to these utilities look a little strange.

4.1. Interfacing

The history mechanism interface is found in the project configuration file called *config*, relative to the root of the baseline. It is a source file and subject to the same controls as any other source file. The history fields of the file are described as follows

4.1.1. history_create_command

This field is used to create a new history. The command is always executed as the project owner. Substitutions available for the command string are:

${Input}	source file
${History}	history file

4.1.2. history_get_command

This field is used to get a file from history. The command may be executed by developers. Substitutions available for the command string are:

${History}	history file
${Edit}	edit number
${Output}	destination file

4.1.3. history_put_command

This field is used to add a new change to the history. The command is always executed as the project owner. Substitutions available for the command string are:

${Input}	source file
${History}	history file

4.1.4. history_query_command

This field is used to query the topmost edit of a history file. Result to be printed on the standard output. This command may be executed by developers. Substitutions available for the command string are:

${History}	history file

4.2. Using SCCS

The entries for the commands are listed below. SCCS uses a slightly different model than aegis wants, so some manuevering is required. The command strings in this section assume that the SCCS commands *admin* and *get* and *delta* are in the command search PATH, but you may like to hard-wire the paths, or set PATH at the start of each. You should also note that the strings are always handed to the Bourne shell to be executed, and are set to exit with an error immediately a sub-command fails.

4.2.1. history_create_command

This command is used to create a new project history. The command is always executed as the project owner.

The following substitutions are available:

${Input}	absolute path of the source file
${History}	absolute path of the history file

The entry in the *config* file looks like this:

```
history_create_command =
        "admin -n -i$i -y ${d $h}/s.${b $h}; \
        admin -di ${d $h}/s.${b $h}; \
        get -e -t -p -s ${d $h}/s.${b $h} > /dev/null";
```

Note that the "get -e" is necessary to put the s.file into the edit state, but the result of the get can be discarded, because the "admin -i" did not remove the file.

4.2.2. history_get_command

This command is used to get a specific edit back from history. The command may be executed by developers.

The following substitutions are available:

${History}	absolute path of the history file
${Edit}	edit number, as given by history_query_command
${Output}	absolute path of the destination file

The entry in the *config* file looks like this:

```
history_get_command =
        "get -r'$e' -s -p -k ${d $h}/s.${b $h} > $o";
```

4.2.3. history_put_command

This command is used to add a new "top-most" entry to the history file. This command is always executed as the project owner.

The following substitutions are available:

${Input}	absolute path of source file
${History}	absolute path of history file

The entry in the *config* file looks like this:

```
history_put_command =
       "cd ${d $i}; \
       delta -s -y ${d $h}/s.${b $h}; \
       get -e -t -p -s ${d $h}/s.${b $h} > $i";
```

Note that the SCCS file is left in the edit state, and that the source file is left in the baseline.

4.2.4. history_query_command

This command is used to query what the history mechanism calls the top-most edit of a history file. The result may be any arbitrary string, it need not be anything like a number, just so long as it uniquely identifies the edit for use by the *history_get_command* at a later date. The edit number is to be printed on the standard output. This command may be executed by developers.

The following substitutions are available:

${History}	absolute path of the history file

The entry in the *config* file looks like this:

```
history_query_command =
       "get -t -g ${d $h}/s.${b $h} 2>&1";
```

Note that "get" reports the edit number on stderr.

4.3. Using RCS

The entries for the commands are listed below. RCS uses a slightly different model than aegis wants, so some manuevering is required. The command strings in this section assume that the RCS commands *ci* and *co* and *rcs* and *rlog* are in the command search PATH, but you may like to hard-wire the paths, or set PATH at the start of each. You should also note that the strings are always handed to the Bourne shell to be executed, and are set to exit with an error immediately a sub-command fails.

In these commands, the RCS file is kept unlocked, since only the owner will be checking changes in. The RCS functionality for coordinating shared access is not required.

One advantage of using RCS version 5.6 or later is that binary files are supported, should you want to have binary files in the baseline.

4.3.1. history_create_command

This command is used to create a new project history. The command is always executed as the project owner.

The following substitutions are available:

${Input}	absolute path of the source file
${History}	absolute path of the history file

The entry in the *config* file looks like this:

```
history_create_command =
        "ci -u -t/dev/null $i $h,v; \
        rcs -U $h,v";
```

The "ci -u" is so that a copy remains in the baseline. The "rcs -U" is necessary to set unstrict locking.

4.3.2. history_get_command

This command is used to get a specific edit back from history. The command may be executed by developers.

The following substitutions are available:

${History}	absolute path of the history file
${Edit}	edit number, as given by history_query_command
${Output}	absolute path of the destination file

The entry in the *config* file looks like this:

```
history_get_command =
        "co -u'$e' -p $h,v > $o";
```

Note that the destination filename will *never* look anything like the history source filename, so the -p is essential.

4.3.3. history_put_command

This command is used to add a new "top-most" entry to the history file. This command is always executed as the project owner.

The following substitutions are available:

${Input}	absolute path of source file
${History}	absolute path of history file

The entry in the *config* file looks like this:

```
history_put_command =
        "ci -u -m/dev/null $i $h,v";
```

Note that the source file is left in the baseline.

It is possible that a very cautious approach could be taken, and combine history_create_command with history_put_command, viz:

```
history_create_command =
        "ci -u -m/dev/null -t/dev/null $i $h,v; \
        rcs -U $h,v";
history_put_command =
        "ci -u -m/dev/null -t/dev/null $i $h,v; \
        rcs -U $h,v";
```

4.3.4. history_query_command

This command is used to query what the history mechanism calls the top-most edit of a history file. The result may be any arbitrary string, it need not be anything like a number, just so long as it uniquely identifies the edit for use by the *history_get_command* at a later date. The edit number is to be printed on the standard output. This command may be executed by developers.

The following substitutions are available:

${History}	absolute path of the history file

The entry in the *config* file looks like this:

```
history_query_command =
        "rlog -r $h,v | awk '/^head:/ {print $$2}'";
```

4.4. Using fhist

The *fhist* program was written by David I. Bell and is admirably suited to providing a history mechanism with out the "cruft" that SCCS and RCS impose.

4.4.1. history_create_command

This command is used to create a new project history. The command is always executed as the project owner.

The following substitutions are available:

${Input}	absolute path of the source file
${History}	absolute path of the history file

The entry in the *config* file looks like this:

```
history_create_command =
        "fhist ${b $i} -create -i $i -p ${d $h} -r";
```

4.4.2. history_get_command

This command is used to get a specific edit back from history. The command may be executed by developers.

The following substitutions are available:

${History}	absolute path of the history file
${Edit}	edit number, as given by history_query_command
${Output}	absolute path of the destination file

The entry in the *config* file looks like this:

```
history_get_command =
        "fhist ${b $h} -e '$e' -o $o -p ${d $h}";
```

Note that the destination filename will *never* look anything like the history source filename, so the -p is essential.

4.4.3. history_put_command

This command is used to add a new "top-most" entry to the history file. This command is always executed as the project owner.

The following substitutions are available:

${Input}	absolute path of source file
${History}	absolute path of history file

The entry in the *config* file looks like this:

```
history_put_command =
        "fhist ${b $i} -u -i $i -p ${d $h} -r";
```

Note that the source file is left in the baseline.

4.4.4. history_query_command

This command is used to query what the history mechanism calls the top-most edit of a history file. The result may be any arbitrary string, it need not be anything like a number, just so long as it uniquely identifies the edit for use by the *history_get_command* at a later date. The edit number is to be printed on the standard output. This command may be executed by developers.

The following substitutions are available:

${History}	absolute path of the history file

The entry in the *config* file looks like this:

```
history_query_command =
       "fhist ${b $h} -l 0 -p ${d $h} -q";
```

5. The Dependency Maintenance Tool

The aegis program places heavy demands on the dependency maintenance tool, so it is important that you select an appropriate one. This chapter talks about what a dependency tool requires, and gives examples of how to use the various alternatives.

At this writing, the author has seen few sufficiently capable dependency maintenance tools.

5.1. Required Features

When selecting a Dependency Maintenance Tool it is important to keep in mind that it must be able to cope with a heirachy of parallel source directory trees.

The heart of any DMT is an *inference engine*. This inference engine accepts a *goal* of what you want it to construct and a set of *rules* for how to construct things, and attempts to construct what you asked for given the rules you specified. This is exactly a description of an expert system, and the DMT needs to be an expert system for constructing files.

This perspective on what the aegis program needs from a DMT reveals that the old-faithful *make*(1) distributed with so many flavours of UNIX simply isn't good enough, and that something like PROLOG is probably ideal.

5.1.1. Search Lists

For the union of all files in a project and all files in a change (remembering that a change only copies those files it is modifying, plus it may add or remove files) for all files you must be able to say to the dependency maintenance tool,

> "If and only if the file is up-to-date in the baseline, use the baseline copy of the file, otherwise construct the file in the development directory".

The presence of a source file in the change makes the copy in the baseline out-of-date.

Most DTMs with this capability implement it by using some sort of search path, allowing a hierachy of directories to be scanned with little or no modification to the rules.

5.1.2. Dynamic Include File Dependencies

Include file dependencies are very important, because a change may alter an include file, and all of the sources in the baseline which use that include file must be recompiled.

Consider the example given earlier: the include file describing the interface definition of a fucntion is copied into a change and edited, and so is the source file defining the function. It is essential that all source files in the baseline which include that file re recompiled, which will usually result in suitable diagnostic errors if any of the clients of the altered function have yet to be included in the change.

There are two ways of handling include file dependencies:

• They can be kept in a file, and the file can be maintained but suitable programs (maintaining it manually never works, that's just human nature).

• They can be determined by the DTM when it is scanning the rules to determine what needs updating.

5.1.2.1. Static File

Keeping include dependencies in a file has a number of advantages:

• Most existing DMTs have the ability to include another rules file, so that when performing a development build from a baseline rules file, it could include a dependencies file in the development directory.

• Reading a file is much faster than scanning all of the source files.

Keeping include dependencies in a file has a number of disadvantages:

• The file is independent of the DMT, it is either generated before the DMT is invoked, in which case it may do more work than is necessary, or it may be invoked after the DMT (or after the DMT has scanned its rules), in which case it may well be out-of-date when the DMT needs it.

For example, the use of *gcc -M* produces "dot d" files, which may be merged to construct such an includable dependency file. This happens after the DMT has read and applied the rules, but possibly before the DMT has finished executing.

• Many tools which can generate this information, such as the *gcc -M* option, are triggered by source files, and are unable to manage a case where it is an include file which is changing, to include a different set of other include files. In this case, the inaccurate dependencies file may contain refernces to the old set of nested include files, some of which may no longer exist, This causes the DMT to incorrcetly generate an error stating that the old include file is missing, when it is actually no longer required.

If a DMT can only support this kind of include file dependencies, it is not suitable for use with aegis.

5.1.2.2. Dynamic

In order for a DMT to be suitable for use with aegis, it is essential that rules for the DMT may be specified in such a way that include file dependencies are determined "on the fly" when the DMT is determining if a given rule is applicable, and before the rule is applied.

This method suffers from the problem being rather slow; but this is amenable to some caching and the losses of performance are not as bad as could be imagined.

This method has the advantage of correctness in all cases, where a static file may at times be out-of-date.

5.2. Cook

The *cook* program is the only dependency maintenance tool, known to the author, which is sufficiently capable to supply aegis' needs.[16]

Eventually this section will provide templates for constructing *Howto.cook* file (Howto.cook is to cook what Makefile is to make).

This section will discuss the *build_command* and *integrate_build_command* and *link_baseline* and *change_file_command* and *project_file_command* fields of the *config* file.

In the mean time, see *aepconf* 5 for more information.

5.2.1. Invoking Cook

The *build_command* field of the *config* file is used to invoke the relevant build command. In this case, it is set as follows

```
build_command =
       "cook -b ${s Howto.cook} project=$p change=$c version=$v -nl";
```

This command tells cook where to find the recipes. The `${s Howto.cook}` expands to a path into the baseline during development if the file is not in the change. Look in *aesub* (5) for more information about command substitutions.

The recipes which follow will all remove their targets before constructing them, which qualifies them for the next entry in the *config* file:

```
link_integration_directory = true;
```

The files must be removed first, otherwise the baseline would cease to be self-consistent.

5.2.2. The Recipe File

The file containing the recipes is called *Howto.cook* and is given to cook on the command line.

The following items are preamble to the rest of the file; they ask aegis for the source files of the project and change so that cook can determine what needs to be compiled and linked.

```
project_files =
       [collect aegis -l pf -terse -p [project] -c [change]];
change_files =
       [collect aegis -l cf -terse -p [project] -c [change]];
source_files =
       [sort [project_files] [change_files]];
```

This example continues the one from chapter 3, and thus has a single executable to be linked from all the object files

```
object_files =
       [fromto %.y %.o [match_mask %.y [source_files]
       [fromto %.l %.o [match_mask %.l [source_files]
       [fromto %.c %.o [match_mask %.c [source_files]
       ;
```

It is necessary to determine if this is a development build, and thus has the baseline for additional ingredients searches, or an integration build, which does not. The version supplied by aegis will tell us this information, because it will be *major.minor.*C*change* for development builds and *major.minor.*D*delta* for integration builds.

---

[16] The version in use when writing this section was 1.3. Earlier versions are known to work with aegis, but the recipes are horribly cumbersome.

```
if [match_mask %1C%2 [version]] then
{
        baseline = [collect aegis -cd -bl -p [project]];
        search_list = . [baseline];
}
```

The *search_list* variable in cook is the list of directories to search for dependencies; it defaults to only the current directory. The *resolve* builtin function of cook may be uased to ask cook for the name of the file actually used to resolve dependencies, so that recipe bodies may reference the appropriate file:

```
example: [object_files]
{
        [cc] -o example [resolve [object_files]] -ly -ll;
}
```

This recipe says that to cook the example program, you need the object files determined earlier, and them link them together. Object files which were up to date in the baseline are used wherever possible, but files which were out of date are constructed in the current directory and those will be linked.

5.2.3. The Recipe for C

Next we need to tell cook how to manage C sources. On the surface, this is a simple recipe:

```
%.o: %.c
{
        rm %.o;
        [cc] [cc_flags] -c %.c;
}
```

Unfortunately it has forgotten about finding the include file dependencies. The cook package includes a program called *c_incl* which is used to find them. The recipe now becomes

```
%.o: %.c: [collect c_incl -eia %.c]
{
        rm %.o;
        [cc] [cc_flags] -c %.c;
}
```

The file may not always be present to be removed (causing a fatal error), and it is irritation to execute a redundant command, so the remove is mangled to look like this:

```
%.o: %.c: [collect c_incl -eia %.c]
{
        if [exists %.o] then
                rm %.o
                        set clearstat;
        [cc] [cc_flags] -c %.c;
}
```

The "set clearstat" clause tells cook that the command will invalidate parts of its *stat* cache, and to look at the command for what to invalidate.

Another thing this recipe needs is to use the baseline for include files not in a change, and so the recipe is altered again:

```
%.o: %.c: [collect c_incl -eia [prepost "-I" "" [search_list]] %.c]
{
        if [exists %.o] then
                rm %.o
                        set clearstat;
        [cc] [cc_flags] [prepost "-I" "" [search_list]] -c %.c;
}
```

See the *Cook Reference Manual* for a description of the *prepost* builtin function, and other cook details.

Unfortunately, there is one last change that must be made to this recipe, it must use the resolve function to reference the appropriate file once cook has found it on the search list:

```
%.o: %.c: [collect c_incl -eia [prepost "-I" "" [search_list]]
        [resolve %.c]]
{
        if [exists %.o] then
                rm %.o
                        set clearstat;
        [cc] [cc_flags] [prepost "-I" "" [search_list]]
                -c [resolve %.c];
}
```

Only use this last recipe for C sources, ther others are only shown so that the derivation of the reciope is clear; while it is very similar to the original, it looks daunting at first.

5.2.4. The Recipe for Yacc

Having explained the complexities of the recipes in the above section about C, the recipe for yacc will be given without delay:

```
%.c %.h: %.y
{
        if [exists %.c] then
                rm %.c
                        set clearstat;
        if [exists %.h] then
                rm %.h
                        set clearstat;
        [yacc] [yacc_flags] -d [resolve %.y];
        mv y.tab.c %.c;
        mv y.tab.h %.h;
}
```

This recipe could be jazzed up to cope with the listing file, too, if that was deesired, but this is sufficient to work with the example.

Cook's ability to cope with transitive dependencies will pick up the generated .c file and construct the necessary .o file.

5.2.5. The Recipe for Lex

The recipe for lex is vary similar to the recipe for yacc.

```
%.c: %.l
{
        if [exists %.c] then
                rm %.c
                        set clearstat;
        [lex] [lex_flags] -d [resolve %.l];
        mv lex.yy.c %.c;
}
```

Cook's ability to cope with transitive dependencies will pick up the generated .c file and construct the necessary .o file.

5.2.6. Recipes for Documents

You can format documents, such as user guides and manual entries with aegis and cook, and the recipes are similar to the ones above.

```
%.ps: %.ms: [collect c_incl -r -eia [prepost "-I" "" [search_list]]
        [resolve %.ms]]
{
        roffpp [prepost "-I" "" [search_list]] [resolve %.ms]
                | groff -p -t -ms > [target];
}
```

This recipe says to run the document through groff, with the *pic*(1) and *tbl*(1) filters, use the *ms*(7) macro package, to produce PostScript output. The *roffpp* program comes with cook, and is like *soelim*(1) but it takes include search path options on the command line.

Manual entries may be handled in a similar way

```
%.cat: %.man: [collect c_incl -r -eia [prepost "-I" ""
        [search_list]] [resolve %.man]]
{
        roffpp [prepost "-I" "" [search_list]] [resolve %.man]
                | tbl | nroff -man > [target];
}
```

5.3. Other Packages

This section briefly discusses other DMTs which are known to the author and are known to be inadequate.

5.3.1. Traditional Make

The *make* (1) which comes with most flavors of UNIX is inadequate for both requirements: it has no dependency search path, and it has no dynamic include file dependecy facility. Many also have no ability to include files in the Makefiles, and have no ability to collect the output of a shell command.

5.3.2. GNU make

The version of GNU make evaluated in writing this section was 3.62. A later version may be more capable.

The GNU make program hase the dependency search path, in the form of its *VPATH* variable, but it has no dynamic include file dependency facility. It does have the ability to include files in the Makefiles, and the ability to collect the output of a shell command.

6. The Difference Tools

This chapter describes the difference commands in the project configuation file. Usually these commands are used by the *aegis -DIFFerence* command when differencing files, but they may be used to accomplish some other things.

6.1. Interfacing

The build commands are accessed from two fields of the project configuation file (*config*).

6.1.1. diff_command

This command is used by *aed*(1) to produce a difference listing when file in the development directory was originally copied from the current version in the baseline[17].

All of the command sibstitutions described in *aesub*(5) are available. In addition, the following substitutions are also available:

${ORiginal}
: The absolute path name of a file containing the version originally copied. Usually in the baseline.

${Input}
: The absolute path name of the edited version of the file. Usually in the development directory.

${Output}
: The absolute path name of the file in which to write the difference listing. Usually in the development directory.

An exit status of 0 means sucessful, even of the files differ (and they usually do). An exit status which is non-zero means something is wrong.

The non-zero exit status may be used to overload this command with extra tests, such as line length limits. The difference files must be produced in addition to these extra tests.

6.1.2. diff3_command

This command is used by *aed*(1) to produce a difference listing when file in the development directory is out of date compared to the current version in the baseline.

All of the command sibstitutions described in *aesub*(5) are available. In addition, the following substitutions are also available:

${ORiginal}
: The absolute path name of a file containing the version originally copied. Usually in a temporary file.

${Most_Recent}
: The absolute path name of a file containing the most recent version. Usually in the baseline.

${Input}
: The absolute path name of the edited version of the file. Usually in the development directory.

${Output}
: The absolute path name of the file in which to write the difference listing. Usually in the development directory.

An exit status of 0 means sucessful, even of the files differ (and they usually do). An exit status which is non-zero means something is wrong.

[17] Or this is logically the case.

6.2. Using diff and diff3

These two tools are available with most flavours of UNIX, but often in a very limited form. One severe limitation is the *diff3* (1) command, which often can only cope with 200 lines of differences. The best alternative is to use GNU diff, with context differences available, and a far more robust diff3.

6.2.1. diff_command

See the earlier *Interfacing* section for substitution details.

The entry in the *config* file looks like this:

```
diff_command =
        "set +e; diff -c $original $input > $output; test $? -le 1";
```

This needs a little explanation:
• This command is always executed with the shell's **-e** option enabled, causing the shell to exit on the first error. The "set +e" turns this off.
• The *diff* (1) command exits with a status of 0 if the files are identical, and a status of 1 if they differ. Any other status means something horrible happened. The "test" command is used to chnage this to the exit status aegis expects.

6.2.2. diff3_command

See the earlier *Interfacing* section for substitution details.

The entry in the *config* file looks like this:

```
diff3_command =
        "(diff3 -e $MostRecent $original $input | \
        sed -e '/^w$$/d' -e '/^q$$/d'; echo '1,$$p' ) | \
        ed - $MostRecent > $output";
```

This needs a lot of explanation.
• The *diff3* (1) command is used to produce an edit script that will incorporate into $MostRecent, all the changes between $original and $input.
• The *sed* (1) command is used to remove the "write" and "quit" commands from the generated edit script.
• The *ed* (1) command is used to apply the generated edit script to the $MostRecent file, and print the results on the standard output, which are redirected into the $output file.

6.3. Using fhist

The **fhist** program by David I. Bell also comes with two other utilities, *fcomp* and *fmerge*, which use the same minimal diference algorithm.

6.3.1. diff_command

See the earlier *Interfacing* section for substitution details.

The entry in the *config* file looks like this:

```
diff_command =
        "fcomp -w $original $input -o $output";
```

The **-w** option produces an output of the entire file, with insertions an deletions marked by "change bars" in the left margin. This is superior to context difference, as it shows the entire file as context.

For more information, see the *fcomp*(1) manual entry.

6.3.2. diff3_command

See the earlier *Interfacing* section for substitution details.

The entry in the *config* file looks like this:

```
diff3_command =
        "fmerge $original $MostRecent $input -o $output -c /dev/null";
```

The output of this command is similar to the output of the diff3_command in the last section. Conflicts are marked in the output. For more information, see the *fmerge*(1) manual entry.

7. The Project Attributes

The project attributes are manipulated using the *aepa*(1) command. This command reads a project attributes file to set the project attributes. This file can be thought of as having several sections, each of which will be covered by this chapter. You should see the *aepattr*(50 manual entry for more details.

7.1. Description and Access

The *description* field is a string which contains a description of the project. Large amounts of prose are not required; a single line is sufficient.

The *default_development_directory* field is a string which contains the pathname of where to place new development directories. The pathname must be absolute. This field is only consulted if the *uconf*(5) field of the same name is not set. Defaults to *$HOME*.

The *umask* field is an integer which is set to the file permission mode mask. See *umask*(2) for more information. This value will always be OR'ed with 022, because aegis is paranoid.

7.2. Notification Commands

The *develop_end_notify_command* field is a string which contains a command to be used to notify that a change requires reviewing. All of the substitutions described in *aesub*(5) are available. This field is optional, if it is not specified no notification will be issued. This command could also be used to notify other management systems, such as progress and defect tracking, in addition to notifying users.

The *develop_end_undo_notify_command* field is a string containing a command used to notify that a change has been withdrawn from review for further development. All of the substitutions described in *aesub*(5) are available. This field is optional, if it is not specified no notification will be issued. This command could also be used to notify other management systems, such as progress and defect tracking, in addition to notifying users.

The *review_pass_notify_command* field is a string containing the command to notify that the review has passed. All of the substitutions described in *aesub*(5) are available. This field is optional, if it is not specified no notification will be issued. This command could also be used to notify other management systems, such as progress and defect tracking, in addition to notifying users.

The *review_pass_undo_notify_command* field is a string comtaining the command to notify that a review pass has has been rescinded. All of the substitutions described in *aesub*(5) are available. This field is optional, and defaults to the *develop_end_notify_command* field if not specified. If neither is specified, no notification will be issued. This command could also be used to notify other management systems, such as progress and defect tracking, in addition to notifyin users.

The *review_fail_notify_command* field is a string containing the command to notify that the review has failed. All of the substitutions described in *aesub*(5) are available. This field is optional, if it is not specified no notification will be issued. This command could also be used to notify other management systems, such as progress and defect tracking, in addition to notifyin users.

The *integrate_pass_notify_command* field is a string comtaining the command to notify that the integration has passed. All of the substitutions described in *aesub*(5) are available. This field is optional, if it is not specified no notification will be issued. This command could also be used to notify other management systems, such as progress and defect tracking, in addition to notifying users.

The *integrate_fail_notify_command* field is a string containg the command to notify that the integration has failed. All of the substitutions described in *aesub*(5) are available. This field is optional, if it is not specified no notification will be issued. This command could also be used to notify other management systems, such as progress and defect tracking, in addition to notifying users.

7.2.1. Notification by email

The aegis command is distributed with a set of shell scripts to perform these notifications by email. They are installed into the */usr/local/lib/aegis* directory. The entries in the project attribute file look like this:

```
develop_end_notify_command =
        "sh /usr/local/lib/aegis/de.sh $project $change \
        $developer";
develop_end_undo_notify_command =
        "sh /usr/local/lib/aegis/deu.sh $project $change \
        $developer";
review_pass_notify_command =
        "sh /usr/local/lib/aegis/rp.sh $project $change \
        $developer $reviewer";
review_pass_undo_notify_command =
        "sh /usr/local/lib/aegis/rpu.sh $project $change \
        $developer $reviewer";
review_fail_notify_command =
        "sh /usr/local/lib/aegis/rf.sh $project $change \
        $developer $reviewer";
integrate_pass_notify_command =
        "sh /usr/local/lib/aegis/ip.sh $project $change \
        $developer $reviewer $integrator";
integrate_fail_notify_command =
        "sh /usr/local/lib/aegis/if.sh $project $change \
        $developer $reviewer $integrator";
```

7.2.2. Notification by USENET

The aegis command is distributed with a set of shell scripts to perform these notifications by USENET. They are installed into the */usr/local/lib/aegis* directory. The entries in the project attribute file look like this:

```
develop_end_notify_command =
        "sh /usr/local/lib/aegis/de.inews.sh $p $c alt.$p";
develop_end_undo_notify_command =
        "sh /usr/local/lib/aegis/deu.inews.sh $p $c alt.$p";
review_pass_notify_command =
        "sh /usr/local/lib/aegis/rp.inews.sh $p $c alt.$p";
review_pass_undo_notify_command =
        "sh /usr/local/lib/aegis/rpu.inews.sh $p $c alt.$p";
review_fail_notify_command =
        "sh /usr/local/lib/aegis/rf.inews.sh $p $c alt.$p";
integrate_pass_notify_command =
        "sh /usr/local/lib/aegis/ip.inews.sh $p $c alt.$p";
integrate_fail_notify_command =
        "sh /usr/local/lib/aegis/if.inews.sh $p $c alt.$p";
```

The last argument to each command is the newsgroup to post the article in, you may want to use some other group. Note that "$p" is an abbreviation for "$project" and "$c" is an abbreviation for "$change".

7.3. Exemption Controls

The *developer_may_review* field is a boolean. If this field is true, then a developer may review her own change. This is probably only a good idea for projects of less than 3 people. The idea is for as many people as possible to critically examine a change.

The *developer_may_integrate* field is a boolean. If this field is true, then a developer may integrate her own change. This is probably only a good idea for projects of less than 3 people. The idea is for as many people as possible to critically examine a change.

The *reviewer_may_integrate* field is a boolean. If this field is true, then a reviewer may integrate a change she reviewed. This is probably only a good idea for projects of less than 3 people. The idea is for as many people as possible to critically examine a change.

The *developers_may_create_changes* field is a boolean. This field is true if developers may created changes, in addition to administrators. This tends to be a very useful thing, since developers find most of the bugs.

The *default_test_exemption* field is a booean. This field contains what to do when a change is created with no test exemption specified. The default is "false", i.e. no testing exemption, tests must be provided.

This kind of blanket exemption should only be set when a project has absolutely no functionality available from the command line; examples include X11 programs. The program could possibly be improved by providing access to the functionality from the command line, thus allowing tests to be written.

7.3.1. One Person Projects

The entries in the project attributes file for a one person project look like this:

```
developer_may_review = true;
developer_may_integrate = true;
reviewer_may_integrate = true;
developers_may_create_changes = true;
```

All of the staff roles (administrator, developer, reviewer and integertor) are all set to be the same user.

7.3.2. Two Person Projects

A two person project has the opprotunity for each to review the other's work.

The entries in the project attributes file for a one person project look like this:

```
developer_may_review = false.
developer_may_integrate = true;
reviewer_may_integrate = true;
developers_may_create_changes = true;
```

All of the staff roles (developer, reviewer and integertor) are all set to allow both users.

7.3.3. Larger Projects

Once you have 3 or more staff on a project, you can assign all of the rolews to separate people. The idea is for the greatest number of eyes to see each change and detect falus before they reach the baseline.

The entries in the project attributes file for a one person project look like this:

```
developer_may_review = false.
developer_may_integrate = false;
reviewer_may_integrate = false;
developers_may_create_changes = true;
```

For smaller teams, everyone may be a developer. As the teams get larger, the more experienced staff are often the reviewers, rather than everyone.

8. Tips and Traps

This chapter contains hints for how to use the aegis program more efficiently and documents a number of pitfalls you may encounter.

This chapter is at present very "ad hoc" with no particular ordering. Fortunately, it is, as yet, rather small. The final size of this chapter is expected to be quite large.

8.1. C Include Semantics

The semantics of C include directives make the

`#include "`*filename*`"`

directive dangerous in a project developed with the aegis program.

Depending on the age of your compiler, whether it is ATT traditional C or newer ANSI C, this form of directive will search first in the current directory and then along the search path, or in the directory of the including file and then along the search path.

The first case is fairly benign, except that compilers are rapidly becoming ANSI C compliant, and an operating system upgrade could result in a nasty surprise.

The second case is bad news. If the source file is in the baseline and the include file is in the change, you don't want the source file to use the include file in the baseline.

Always use the

`#include` <*filename*>

form of the include directive, and set the include search path explicitly on the command line used by the dependency maintenance tool.

The dependency maintenance tool needs to be able to dynamically adapt to include file dependencies, because they are not static. The presence of an include file in a change means that any file which includes this include file, whether that source file is in the baseline or in the change, must have a dependency on the change's include file. Potentially, files in the baseline will need to be recompiled, and the object file stored in the change, not the baseline. Subsequent linking needs to pick up the object file in the change, not from the baseline.

8.2. Renaming Include Files

Renaming include files can be a disaster, either finding all of the clients, or making sure the new copy is used rather than the old copy still in the baseline.

If an include file is to be completely renamed, not moved to another directory but retain the same basename, then create a file in the development directory (*without* telling aegis) with the old name, and put a syntax error in it. Compiles will fail very diagnostically, and you can change the reference in the source file, probably after *aecp*(1)ing it first.

If you are moving an include file from one directory to another, but leaving the basename unchanged, create a link[18] between the new and old names, but only in the development directory. Create the link after *aenf*(1)ing the new name. This insulates you from a number of nasty Catch-22 situations in writing the dependency maintenance tool's rules file.

8.3. Writing Tests

This section describes a number of things you can do to write better tests, and some pitfalls to be avoided.

The are a number of suggestions for portability of tests; this will definitely be important if you are writing software to publish on USENET or for FTP. Portability is often require *within* an organization, also. Examples include a change in company policy from one 386 UNIX to another (e.g. company doesn't like Linux, now you must use AT&T's SVR4 offering), or the development team use *gcc* until the company finds out and forces you to use the prototype-less compiler supplied with the operating system. Things like this really have happened to the author!

8.3.1. Bourne Shell

The aegis program mandates that all tests be Bourne shell scripts. This is because this shell is available on all flavours of the UNIX operating system. The script files need not have execute permissions set, because the aegis program always invokes them as

`sh` *filename*

so tests should not expect command line arguments. The test is not passed the name of the project nor the number of the change.

This means that if you can write in in a shell script, you can test it. This includes such things

[18] A hard link uses fewer disk blocks. Symbolic links survive the subject file being deleted and recreated.

as client-server model interfaces, and multi-user synchronization testing.

Some indication that the test script is a Bourne shell script is a good idea. While many systems accept that a first line starting with a colon is a Bourne shell "magic number", a more widely understood "magic number" is

```
#! /bin/sh
```

as the first line of the script file.

8.3.2. Current Directory

Tests are always run with the currect directory set to either the development directory change under test when testing a change, or the integration directory when integrating a change, or the baseline when performing independent tests.

A test must not make assumptions about where it is being executed from, except to the extent that it is somewhere a build has been performed. A test must not assume that the current directory is writable, and must not try to write to it, as this could damage the source code of a change under development, potentially destroying weeks of work.

8.3.3. Check Exit Status

A test script should check the exit status of every single command, even those which cannot fail. Do not rely on, or use, the *set -e* shell option (it provides no ability to clean up on error).

Checking the exit status involves testing the contents of the **$?** shell variable. Do not use an *if* statement wrapped around an execution of the program under test as this will miss core dumps and other terminations caused by signals.

Checking the exit status of every command in the script ensures that strange permission settings, or disk space problems, will cause the test to fail, rather than plow on and produce spurious results.

8.3.4. Trap Interrupts

Use the *trap* statement to catch interrupts 1 2 3 and 15 an cause the test to fail. This should perform any cleanup the test requires (such as removing the temporary directory; see next item).

8.3.5. Temporary Directory

Tests should create a temporary directory in */tmp* and then *cd* into this directory.

This tends to isolate any vandalism that the program under test may indulge in, and serves as a place to write temporary files. At the end of the test, it is sufficient to *cd* out of the temporary directory and then *rm -rf* it, rather than track and remove all test files which may or may not be created.

8.3.6. PAGER

If the program under test invokes pagers on its output, a la *more*(1) et al, it should be coded to use the PAGER enviroment variable. Tests of such programs should always set PAGER to *cat* so that tests always behave the same, irrespective of invocation method (either by aegis or from the command line).

8.3.7. [test]

You should always use the *test* command, rather than the square bracket form, as many systems do not have the square bracket form, if you publish to USENET or for FTP.

8.3.8. Auxiliary Files

If a test requires extra files as input or output to a command, it must construct them itself, using *here* documents (see *sh*(1) for more information).

It is almost impossible to determine the location of an auxiliary file, if that auxiliary file is part of the project source. It could be in either the change under test or the baseline.

8.3.9. New Test Templates

It is possible to specify most of the repetitious items above in a *file template* used every time a user creates a new test. See the *aent*(1) command for more information.

Having the machine do it for you means that you are more likely to do it.

8.4. Symbolic Links

If you are on a flavour of UNIX which has symbolic links, it is often useful to create a symbolic link from the development directory to the baseline. This can make browsing the baseline very simple.

Assuming that the project and change defaults are appropriate, the following command

```
ln -s `aegis -cd -bl` bl
```

is all that is required to create a symbolic link called *bl* pointing to the baseline. Note that the *aecd* alias is inappropriate in this case.

8.5. User Setup

There are a number of things which users of aegis can do to make it more useful, or more user friendly. This section describes just a few of them.

8.5.1. .cshrc

The aliases for the various user commands used throughout this manual are obtained by appending a line of the form

```
source /usr/local/lib/aegis/cshrc
```

to the *.cshrc* file in the user's home directory.

8.5.2. The AEGIS environment variable

If users wish to use aegis for their own projects, in addition to the "system" projects, the AEGIS environment variable forms a colon separated search path of aegis "library" directories. The */usr/local/lib/aegis* directory is always implicitly added to this list.

The user should not create this library directory, but let aegis do this for itself (otherwise you will get an error message).

The *AEGIS* environment variable should be set in the *.cshrc* file in the user's home directory. Typical setting is

```
setenv AEGIS ~/lib/aegis
```

8.5.3. The .aegisrc file

The *.aegisrc* file in the user's home directory contains a number of useful fields. See *aeuconf*(5) for more information.

8.5.4. The defaulting mechanism

In order for you to specify the minimum possible information on the command line, aegis has been designed to work most of it out itself.

The default project is the project which you are working on changes for, if there is only one, otherwise it is gleaned from the *.aegisrc* file. The command line overrides any default.

The default change is the one you are working on within the (default or specified) project, if there is only one. The command line overrides any default.

8.6. The Project Owner

For the greatest protection from accidental change, it is best if the project is owned by a UNIX account which is none of the staff. This account is often named the same as the project, or sometimes there is a single umbrella account for all projects.

When an aegis project is created, the owner is the user creating the project, and the group is the user's default group. The creating user is installed as the project's first administrator.

A new project administrator should be created - an actual user account. The UNIX password should then be disabled on the project account - it will never be necessary to use it again.[19]

The user nominated as project administrator many then assign all of the other staff roles. Aegis takes care of ensuring that the baseline is owned by the project account, not any of the other staff, while development directories always belong to the developer (but the group will always be the project group, irrespective of the developer's default group).

All of the staff working on a project should be members of the project's group, to be able to browse the baseline, for reviewers to be able to review changes. This use of UNIX groups means that projects may be as secure or open as desired.

8.7. USENET Publication Standards

If you are writing software to publish on USENET, a number of the source newsgroups have publication standards. This section describes ways of generating the following files, required by many of the newsgroups' moderators:

MANIFEST	List of files in the distribution.
Makefile	How to build the distribution.
CHANGES	What happened for this distribution.
patchlevel.h	An identification of this distribution.

Each of these files may be generated from information known to aegis, with the aid of some fairly simple shell scripts.

[19] Unless bugs in aegis corrupt the database, in which case repairs can be accomplished as the project account using a text editor.

8.7.1. CHANGES

Write this section.

Look in the *aux/CHANGES.sh* file included in the aegis distribution for an example of one way to do this.

8.7.2. MANIFEST

Write this section.

Look in the *aux/MANIFEST.sh* and *aux/MANIFEST.awk* files included in the aegis distribution for an example of one way to do this.

8.7.3. Makefile

Write this section.

Look in the *aux/Makefile.sh* and *aux/Makefile.awk* files included in the aegis distribution for an example of one way to do this.

8.7.4. patchlevel.h

Write this section.

Look in the *aux/Howto.cook* file included in the aegis distribution for an example of one way to do this.

8.7.5. Building Patch Files

The *patch* program by Larry Wall is one of the enduring marvels of USENET. This section describes how to build input files for this miracle program.

Write this section.

Look in the *aux/patches.sh* file included in the aegis distribution for an example of one way to do this.

9. Appendix A: New Project Quick Reference

For those of you too impatient to read a whole great big document about how to use the aegis program, this appendix gives a quick look at how to place a project under aegis.

The style here is a bullet list. It does not try to be exhaustive. For exact details on how to use the various aegis commands, you should see the manual pages, ditto for the formats and contents of some files.

Probably the quickest start of all is to copy an already existing project. The project used in chapter 2 is complete, assuming you use the author's "cook" dependency maintenance tool. The entirety of this example may be fond, if slightly obfuscated, in the aegis source file *test/00/t0011a.sh* distributed with aegis.

9.1. Create the Project

The *aenpr* command is used to create a project. You must supply the name on the command line. The name should be ten characters or less, six characters or less if you want version numbers included.

The user who creates the project is the owner of the project, and is set as the administrator. The default group of the user who created the project is used as the project's group.

You may want to have a user account which owns the project. You must create the project as this user, and then use the *aena* and *anra* commands to add an appropriate administrator and remove the owning user as an administrator. After this, the password for the owning user may be disabled, because the aegis program will, at appropriate times, set file ownership to reflect project ownership or execute commands on behalf of the project owner *as* the project owner.

9.1.1. Add the Staff

The *aend* command is used to add developers. The *aenrv* command is used to add reviewers. The *aeni* command is used to add integrators. These commands may only be performed by a project administrator.

You will still have to do this, even if the person who created the project will be among these people, or even be all of these people.

9.1.2. Project Attributes

The *aepa* command is used to change project attributes. These attributes include the description of the project, and booleans controlling whether, for example, developers may review their own work.

The project attributes file is described in the *aepattr*(5) manual entry.

9.2. Create Change One

The *aenc* command is used to create a new change. You will need to construct a change attributes file with your favorite text editor before running this command.

The change attributes file is described in the *aecattr*(5) manual entry.

9.3. Develop Change One

This is the most grueling step. Indeed, the integration step will probably reveal things you missed, and you may return to the *being developed* state several times.

One of the people you nominated as a developer will have to use the *aedb* command to commence development of change 1. The *aecd* command can be used to change directory into the just-created development directory.

Add files to the change. The *aenf* command is used to create new files. If you don't use *aenf* then the aegis program has no way of knowing whether that file lying there in the development directory is significant to the project, or just a shopping list of the groceries you forgot to buy yesterday.

One particular new file which *must* be created by this change is the *config* file. This tells the aegis program what history mechanism you wish to use, what dependency maintenance command to use, what file difference tools to use, and much more. The *aepconf*(5) manual entry describes this file.

If you are going to use the "cook" dependency maintenance tool, another new file you will need to create in this change is the "Howto.cook" file. Some other tool will want some other rules file.

You probably have a prototype or some other "seed" you have sort-of working. Create new files for each source file and *then* copy the files from wherever they are now into the development directory.

Use the *aeb* command to build the change. It will need to build cleanly before it can advance to the next step.

Use the *aed* command to difference the change. It will need to difference cleanly before it can advance to the next step.

Use the *aent* command to add new tests to the command. It will need to have tests before it can advance to the next step.

Most existing projects don't have formal tests. These tests will form a regression test-bed, used to make sure that future changes never compromise existing functionality.

Use the *aet* command to test the change. It will need to test cleanly before it can advance to the next step.

Once the change builds, differences and tests cleanly, use the *aede* command to end development. At this point, the mode of the files will be changed to read only, preventing accidental modification of the files.

9.4. Review The Change

One of the people nominated as reviewers will have to run the *aerpass* command to say that the change passed review.

The aegis program does not mandate any particular review mechanism: you could use a single peer to do the review, you could use a panel, you could set the project so that developers may review their own work and effectively eliminating the review step. In projects with as few as two people, it is always beneficial for someone other than the developer to review changes.

Should a reviewer actually want to *see* the change, the *aecd* command may be used to change directory to the development directory of the change. The difference files all end with a "comma D" suffix, so the

```
more `find . -name "*,D"
-print | sort`
```

command may be used to search them out and see them. This is probably fairly useless for the first change, but is vital for all subsequent changes. There is a supplied alias for this command, it is *aedmore* and there is a similar *aedless* alias if you prefer the *less*(1) command.

There are some facts that a reviewer *knows* because otherwise the change would not be in the "being reviewed" state: • the change compiles cleanly, • the change passes all of its tests. Other information about the change may be obtained using the "change_details" variation of the *ael* command.

The *aerfail* command may also be used by reviewers to fail reviews and return a change to the developer for further work; the reviewer must supply a reason for the change history to record for all time. Similarly, the *aedeu* command may be used by the developer to resume development of a change at any time before it is integrated; no stated reason is required.

9.5. Integrate the Change

A person nominated as an project integrator then integrates the change. This involves making a copy of the integration directory, applying the modifications described by the change to this integration directory, then building and testing all over again.

This re-build and re-test is to ensure that no special aspect of the developers environment influenced the success up to this point, such as a unique environment variable setting. The re-build also ensures that all of the files in the baseline, remembering that this includes source files and all other intermediate files required by the build process, ensures that all of these files are consistent with each other, that the baseline is self-consistent. The definition of the baseline is that it passes its own tests, so the tests are run on the baseline.

Use the *aeib* command to begin integration.

The *aeb* command is used to build the integration copy of the change.

The *aet* command is used to test the integration copy of the change.

On later changes, the integration may also require the *aet -bl* command to test the change against the baseline. This tests ensures that the test *fails* against the baseline. This failure is to ensure that bug fixes are accompanied by tests which reproduce the bug initially, and that the change has fixed it. New functionality, naturally, will not be present in the old baseline, and so tests of new functionality will also fail against the old baseline.

Later changes may also have the regression tests run, using the *aet -reg* command. This can be a very time-consuming step for projects with a long history, and thus a large collection of tests. The *aet* command can also be used to run "representative" sets of existing tests, but a full regression test run is recommended before a major release, or, say, weekly if it will complete over the weekend. This command is also available to developers, so that they have fewer surprises from irate integrators.

The integrator may use the *aeifail* command to return a change to its developer for further work; a reason must be supplied, and should include relevant excerpts from the build log in the case of a build failure (not the *whole* log!), or a list of the tests which failed for test failures.

The *aeipass* command may be used to pass an integration. When the change passes, the file histories are updated. In the case of the first change, the history is created, and problems with the *config* file's history commands will be revealed at this point. The integration won't pass, and should be failed, so that the developer may effect repairs. There are rarely problems at this point for subsequent changes, except for disk space problems.

Once the history is successfully updated, aegis renames the integration directory as the baseline, and throws the old baseline away. The development directory is deleted at this time, too.

9.6. What to do Next

There, the first change is completed. The whole cycle may now be repeated, starting at "Create Change," for all subsequent changes, with very few differences.

It is recommended that you read the *Change Development Cycle*
chapter for a full worked example of the first four changes of an example project, including some of the twists which occur in real-world use of aegis.

Remember, too, the definition:

> **aegis** (ee.j.iz) *n.* a protection, a defence.

It is not aways the case that aegis exists to make life "easier" for the software engineers. The goal is to have a baseline which always "works", where "works" is defined as passing all of its own tests. Wherever possible, the aegis program attempts to be as helpful and as unintrusive as possible, but when the "working" definition is threatened, the aegis program intrudes as necessary. (Example: you can't do an integrate pass without the integration copy building successfully.)

All of the "extra work" of writing tests is a long-term win, where old problems never again reappear. All of the "extra work" of reviewing changes means that another pair of eyes sights the code and finds potential problems before they manifest themselves in shipped product. All of the "extra work" of integration ensures that the baseline always works, and is always self-consistent. All of the "extra work" of having a baseline and separate developemnt directories allows multiple parallel development, with no inter-developer interference; and the baseline always works, it is never in an "in-between" state. In each case, not doing this "extra work" is a false economy.

10. Appendix B: Glossary

The following is an alphabetical list of terms used in this document.

administrator
: Person responsible for administering a *project*.

awaiting_development
: The state a change is in immediately after creation.

awaiting_integration
: The state a change is in after it has passed review and before it is integrated.

baseline
: The repository; where the project master source is kept.

being developed
: The state a change is in when it is being worked on.

being integrated
: The state a change is in when it is being integrated with the baseline.

being reviewed
: The state a change is in after it is developed.

change
: A collection of files to be applied as a single atomic alteration of the baseline.

change number
: Each *change* has a unique number identifying it.

completed
: The state a change is in after it has been integrated with the baseline.

delta number
: Each time the *aeib*(1) command is used to start intgrating a *change* into the *baseline* a unique number is assigned. This number is the delta number. This allows ascending version numbers to be generated for the baseline, independent of change numbers, which are inevitably integrated in a different order to their creation.

dependency maintenance tool
: A program or programs external to aegis which may be given a set of rules for how to efficiently take a set of source files and process them to produce the final product.

DMT
: Abbreviation of Dependency Maintenance Tool.

develop_begin
: The command issued to take a change from the *awaiting development* state to the *being developed* state. The change will be assigned to the user who executed the command.

develop_begin_undo
: The command issued to take a change from the *being developed* state to the *awaiting development* state. The change must have no files associated with it.

develop_end
: The command issued to take a change from the *being developed* state to the *being reviewed* state. The change must be know to build and test sucessfully.

develop_end_undo
: The command issued to take a change from the *being reviewed* state back to the *being developed* state. The command must be executed by the original developer.

developer
: A member of staff allowed to develop changes.

development directory
: Each change is given a unique development directory in which to edit files and build and test.

history tool
: A program to save and restore previous versions of a file, usually by storing edits between the versions for efficiency.

integerate_pass
: The command used to take a change from the *being integrated* state to the *completed* state. The change must be known to build and test successfully.

integrate_begin
: The command used to take a change from the *awaiting integration* state to the *being integrated* state.

integrate_begin_undo
: The command used to take a change from the *being integrated* state to the *awaiting integration* state.

integrate_fail
: The command used to take a change from the *being integrated* state back to the *being developed* state.

integration
: The process of merging the *baseline* with

the *development directory* to form a new baseline. This includes building and testing the merged directory, before replacing the original *baseline* with the new merged version.

integration directory
: The directory used during *integration* to merge the existing *baseline* with a chage's *development directory* .

integrator
: A staff member who performs *integration* s.

new_change
: The command uased to create new changes.

new_change_undo
: The command used to destroy changes.

review_fail
: The command used to take a change from the *being reviewed* state back to the *bein developed* state.

review_pass
: The command used to take a change from the *being reviewed* state to the *awaiting integration* state.

reviewer
: A person who may review *changes* and either pass or fail them (*review_pass* or *review_fail* respectively).

state
: Each *change* is in one of six states: *awaiting development* , *being developed* , *being reviewed* , *awaiting integration* , *being integrated* or *completed* .

state transition
: The event resulting in a *change* changing from one state to another.

11. Appendix C: Product Comparisons

This appendix compares the aegis program with other pieces of software. Only a few work in the same area as the aegis program, or a commonly compared to the aegis program.

11.1. CVS

A comparison with CVS (Concurrent Version System) is frequently requested.

A fast comparison can be made by saying that *CVS is over RCS* and that *aegis is over RCS and make.* This is a very simplistic view, but it illustrates the fact that aegis has a wider scope than make. While CVS covers a subset of the functionality covered by aegis, it natuarally does that subset better.

The model used by aegis has a baseline *and* a repository. The baseline is all of the most up to date source file **plus** all the implications flowing from those source files, such as object files and executable programs or libraries. This means there is always an "official" executable present, representing the progress made by the project at that very instant. These source files are available for linking or inclusion, the object files available for linking, at no extra effort for the developers.

The reporitory is only for history, using the aegis model. It is used to store everything that has happened, so that developers may receraete earlier version when fixing bugs, and aegis may use it for some other purposes.

CVS provides the repository functionality well. It provides branching nicely, and it has symbolic file sets. While is does not provide a baseline, it does not prevent a project implementing it in some other fashion.

The existence of the baseline for aegis is what limits branching. The semantics of branching in the repository would appear to imply that a (logically) separate baseline is required for each leaf in the history tree. This is why aegis only provides branching without the possibility of merging, by deriving a whole new project from an existing one; thus creating a whole new baseline as well. Should a better set of semantics be made aparrent, this is certainly an area for improvement.

Another aspect of Software Configuration Management (SCM) is consistency checking. For example, if two developers are working on two (logically) separate changes, which both need to alter the same file, one of them will inevitably return it to the baseline first, and the other will then be working on an out of date copy. Both aegis is CVS provide a conflict resolution mechanism to handle this; the same algorithm is used by both, although implemented a little differently in each case.

There is a second aspect to consistency checking required by SCM; it is important that a change does not "break" the project, and it is thus important that all the source files in the project agree with each other. For example, if a developer alters an include file as part of a change, and the portion of this include file describing a function prototype is changed, it is essential that all of the clients of that function be recompiled, to ensure that they are still compatible with that prototype, and may need to be included with the change to update their use of the function. Similarly, the definition of the function needs to be recompiled, to ensure that it, too, agrees with the prototype, and have the interface and 9implementation changed to agree with it. This type of consistency checking is provided by aegis, but not bu CVS; although CVS does not prevent a project implementing this in a different way.

Another aspect of SCM provided by aegis is validation. The aegis program insists that changes be accompanied by tests. This allows greater confidence that a change actually works, and the accumulation of such tests provides a regression test suite to verify that later changes do not "break" the functionality provided by this one. The aegis program also insists that each change be reviewed; by whom is configurable. While testing is not a silver bullet, and neither is peer review, both are vital components producing quality software. CVS does not provide validation functionality; althoug it does not prevent a project from implementing this in some other way.

The CVS program aims to provide the repository function for SCM, and it does this well. The aegis program aims to provide a baseline, to provide a reporitory, to provide baseline consistency, and to provide baseline change validation. There are situations where one is more suited, and the other less, and vice versa. The tools selected for a project, from Gandt charts to assemblers, must be chosen for their "fit" with each individuat project.

Another point to note is that CVS requires RCS; whereas aegis may be configured to use almost any history tool. If SCCS is bundled with your operating system, or your company has its own history tool, aegis can be configured to use that, instead.

Table of Contents