Poseidon House
Castle Park
Cambridge CB3 0RD
United Kingdom

TELEPHONE: **Cambridge (01223) 515010**
INTERNATIONAL: **+44 1223 515010**
FAX: **+44 1223 359779**
E-MAIL: **apm@ansa.co.uk**

APM

Requirements for Software Management in APM

Ian Macmillan, Youcef Laribi, Nicola Howarth, Ben Crawford.

Abstract

APM is moving into a more commercial role, with a greater emphasis on production of software. The approach to software management currently employed in house is perceived to have some weaknesses, which now merit attention in the light of these changes in emphasis.

This document therefore examines the approach to software management currently used, considers requirements for an ideal approach, and then prioritieses the issues identified, as a basis for future work to improve the effectiveness of software management.

APM.1511.00.01 **Draft** 12th September 1995

Project Management (confidential to ANSA consortium for 2 years)

Distribution:
Supersedes:
Superseded by:

Requirements for Software Management in APM

Requirements for Software Management in APM

Ian Macmillan, Youcef Laribi, Nicola Howarth, Ben Crawford.

APM.1511.00.01

12th September 1995

The material in this Report has been developed as part of the ANSA Architecture for Open Distributed Systems. ANSA is a collaborative initiative, managed by Architecture Projects Management Limited on behalf of the companies sponsoring the ANSA Workprogramme.

The ANSA initiative is open to all companies and organisations. Further information on the ANSA Workprogramme, the material in this report, and on other reports can be obtained from the address below.

The authors acknowledge the help and assistance of their colleagues, in sponsoring companies and the ANSA team in Cambridge in the preparation of this report.

Architecture Projects Management Limited

Poseidon House
Castle Park
CAMBRIDGE
CB3 0RD
United Kingdom

TELEPHONE UK	(01223) 515010
INTERNATIONAL	+44 1223 515010
FAX	+44 1223 359779
E-MAIL	apm@ansa.co.uk

Contents

1 Introduction

Since software is one of APM's most important products, APM aims to have an effective approach to developing, maintaining and providing software.

This document discusses:

1. The current approach in APM, with particular regard to its weaknesses.
2. Requirements for an improved approach, in terms of
   - General Issues
   - Software Development
   - Software Maintenance
   - Software Provision
3. Prioritisation of these requirements.

Weaknesses in the current approach are one source of new requirements. The requirements discussed here for a 'wish list'; it is not anticipated that all of them will be met in the foreseeable future.

Prioritisation will examine which requirements can provide the most benefit in immediate future (3-6 months). The output of this prioritisation will be a statement about which areas will be tackled, and what aims should be achieved, in what timescales.

2 (Weaknesses in) The current approach

This chapter describes the approach to software management adopted by ANSAware, since this is the only code that APM currently ships. The chapter is structured along the same lines as §3 *Requirements*.

2.1 General

2.1.1 QA process

We have neither a formal nor informal QA process.

2.2 Software Development

2.2.1 Design standards

There are none.

2.2.2 Coding standards

There are none. ANSAware code covers the spectrum from reasonably well structured to downright poor. In general, comments are conspicuous by their absence. There appears to be no policy on the use of industry standards.

2.2.2.1 Software documentation

Documentation for ANSAware is limited to that which we ship with the distribution, i.e. manual pages describing the main interfaces and the manual which covers things such as installation. There is no other detailed design documentation either in the code or as separate documents.

In addition there are virtually no comments in the code, so that it is difficult to determine what the contorted bits are doing, and there is no tutorial level documentation. The best on offer are walk-throughs of the echo and simple bank examples in the application manuals.

2.2.3 Software reuse

Anyone who has had recent cause to peruse the ANSAware source will realise that we have no mechanism in place to facilitate software reuse. There is considerable duplication of code and various overlapping mechanisms.

2.2.4 Software integration

There is no structured approach to software integration.

2.2.5 Use of external software

2.2.6 Source control management

Source control management is performed using simple RCS. Files are modified by checking out a copy from the master tree, making the changes and checking the file back in. Frequently, several people will be making simultaneous modifications which may be unrelated.

Major changes are supported by scripts layered on top of RCS which provide the concept of a package. This is a self-contained part of the source tree which may be copied as a single unit. There is no automated support for merging packages back into the master tree.

2.2.7 Build process

ANSAware source is divided into generic (platform independent) and platform specific code. However, the split is not handled in a consistent manner. Platform specific code is variously:

- located in subdirectories, named according to the platform to which they apply, e.g. CORE_unix for unix specific code;
- located in platform specific files in otherwise generic directories and named with a suitable prefix or suffix, e.g. Makefile.msdos;
- located within generic code files and controlled by conditional compilation.

The situation is further confused in that some of the platform specific code is compiled in situ whilst other files are copied into generic directories by the installation process. Even files that are intentionally entirely platform specific often erroneously contain conditionally included code for other platforms.

The build process itself is performed by running scripts which perform site specific configuration before invoking the 'make' utility. The scripts are platform specific being written for the target platform's native command interpreter.

Apart from the scripts, the build process employs several tools: ansaimake, ansadepend and make. The first two are supplied in source form with the ANSAware distribution, whilst the latter must be available on the target platform. Only the unix variant of make is supported, largely because the make files contain a considerable amount of bourne shell script.

There is a further problem with ansaimake, in that it uses the C preprocessor (CPP) to perform the various macro expansions required to convert ANSAware imakefiles to makefiles suitable for make. Neither the syntax nor semantics of CPP are really appropriate to this task leading to considerable difficulty with things like token pasting and the use of quotes. Furthermore, the existing imakefiles make use of various CPP constructs with undefined semantics, most of which work on unix but not on other target platforms.

2.2.8 Testing

There are no automated tests for ANSAware and only a few examples, the output of which must be verified by inspection. The examples test only a small part of ANSAware's functionality.

2.2.9 Development environment

There is no standard development environment. The environment, including many of the tools used to build the software, depend on the developers's particular machine.

2.3 Software Maintenance

2.3.1 Bug fixing and tracking

2.3.2 Fault reporting and tracking

The current system is mainly manual, tedious, and always out of date. Fault reports/queries are sent in by email to <ansaware@ansa.co.uk>, and filed automatically, as well as being sent to the ansaware group. Nicola maintains a list of all reports (by emacs), by mailnote number, date, topic, and whether or not fixed. In addition, two files are used to hold more details, one of bugs waiting to be fixed, and the other of bugs which have been fixed. The idea is that when we produce a new release, all (or most) of the unfixed bugs should be fixed in it.The fact that these files are so out of date indicates what a chore this system is to use. Should really be done on a simple database.

2.3.3 Change request management

There is no specific procedure for requesting changes other than mailing ansaware@ansa.co.uk.

2.3.4 Regression testing

There is no formal procedure for running regression tests and, in the absence of suitable test cases, no easy way to implement one. Partial regression testing can be informally carried out by running the examples that are shipped with ANSAware. However, there is no record of what constitutes a successful result and the test coverage is very limited.

2.4 Software Provision

2.4.1 Supply of software

ANSAware is normally supplied as generic (multi-platform) source code, written to a suitable tape for the target platform using the unix ‘tar’ utility. Various platform specific versions of the product can be built using the scripts supplied on the tape.

It is also possible to produce an MS DOS specific distribution on DOS format floppy disks. However, this is seldom done in practice as most ANSAware sites that use PCs also have access to a unix machine.

We do not currently ship binary only versions of ANSAware.

2.4.2 Installation of software

This is not a particularly straightforward process. First the contents of the distribution tape must be restored to the target machine using ‘tar’. Then the distribution must be ‘installed’ by running a script that ‘creates’ a master tree. Various shadow trees, corresponding to particular platforms, may then be produced from the master tree. Finally, the source is compiled for the target platform in its respective shadow tree. There is also an option to ‘install’, or copy the resultant binaries to an alternative location.

The scripts are complex and almost totally devoid of comment or explanation. It’s very difficult to track down the script which actually performs any of the

individual jobs, since they are heavily nested. So it's difficult to know why/ where it's going wrong. The user has to run two scripts - why not one? Also final "installation" is optional, but you can't run the trader without it, so not much point.

For MS DOS platforms, the situation is further complicated for the typical generic distribution in that the shadow tree produced on unix must be FTP'ed to the target before building. Since the scripts used on unix are all written in bourne shell and awk, different scripts are supplied for MS DOS. Furthermore, these scripts do not present the same interface to the user.

The scripts supplied for all platforms are crude and make little attempt to validate the responses requested from the user performing the build. The result of an erroneous response is often not evident until the build completes with errors, or the resultant executables will not run properly. Either way, there may be a considerable amount of time wasted.

2.4.3 Configuration of installations

This is part of the build and install procedure, the scripts requesting details such as the trader's internet address. Re-configuring the software after it has been built is non-trivial although the behaviour may be modified on a user by user basis by setting the respective users environment variables.

2.5 Conclusion

The current situation regarding the management of ANSAware is not encouraging. Whilst recognising that this may not be entirely inappropriate given the nature of the software (prototype demonstrator) and the use to which it is put, this still leaves a lot to be done if APM desires ISO 9000 accreditation.

The situation may be broadly summarised as follows:

- No formal process or quality standards for
  - Design
  - Code
  - Building software
  - Installation and configuration
  - Tests
  - Documentation
- Poor tool support for source code management and version control
- Inconsistent development environment and associated tools
- Minimal support for fault reporting and tracking.

Note: Summary of problems.

3 Requirements

Software production is a complex process that needs to be rationalised, managed and controlled in order to ensure quality products within the time and budget assigned. Once a process has been defined, it can be supported with improved software engineering technology; but such technology only helps once the organizational framework is in place.

Consequently, requirements must address a process for software production, and an environment in which to conduct the process efficiently.

The software production process encompasses several (potentially interleaving) activities:

1. Writing new source code.
2. Correcting, adapting, enhancing or reusing existing (internal or external) source code.
3. Testing and quality control of software.
4. Maintaining source code by keeping track of its evolution and being able to trace back the recent modifications on it and restore old versions of it.
5. Documenting source code, by describing both the functionality and design implemented by the software and commenting the algorithms used and the different components of the source tree.
6. Shipment and installation procedures.
7. User support, keeping track of bug reports, change requests and user's feedback.

This chapter discusses requirements for an approach to software management in APM. These requirements take the weaknesses identified in §2 into account.

They also take into account the following aspects of APM's business which are important or atypical, and which may cause APM's requirements to differ from the norm:

1. Several distinct but possibly related projects going on at once.
2. Software produced for multiple platforms (Unixes and Windows-NT).
3. Shipping source code in addition to binaries.
4. Robust prototypes, rather than industrial strength software.
5. Products should place minimal constraints on customer's environment.
6. ISO 9000 accreditation not held but would be beneficial.

Detailed requirements are discussed in the remainder of the section using the following categorisation:

1. General
1. Software development

2. Software maintenance
3. Software provision (distribution, installation, configuration)

3.1 General

3.1.1 QA process

Quality assurance involves two fundamental issues:

1. Defining a manageable process, adherence to which can be ensured (and demonstrated) by monitoring activities and recording metrics.
2. Defining a process which will produce the kind of products which the organisation and its customers desire.

ISO 9000 / BS5750 accreditation is a useful indication that an organisation has some infrastructure to manage the first of these issues, and as such is of commercial value.

3.1.1.1 Defining a manageable process

In order to be manageable, a process should:

- Be clearly documented in a well-publicised and accessible place.
- Be broken down into tasks which are carried out in a specified order and which generate deliverables of a defined nature.
- Include descriptions of the required deliverables, so that it is clear when a deliverable has been completed to the required standard.
- Describe a review process for each type of deliverable as a mechanism for verifying that it has reached the required standard.
- Describe a sign-off mechanism for each review process which prevents deliverables from being recognised as acceptable if they have not been produced according to the process and reviewed correctly.
- Specify what information about the process should be recorded to enable the process to be monitored and audited.
- Specify how this recorded information may be used to demonstrate the process has been adhered to.
- Specify how this recorded information may be used to suggest enhancements to the definition of the process.

3.1.1.2 Enforcing conformance

It is of limited benefit to define a process if it is not universally followed, Techniques to encourage and/or enforce conformance are therefor required. There are three practical strategies:

1. To enforce conformance by requiring that a task cannot be accepted as complete without sign-off from some authority. For example, by ensuring that code can only be checked in by an authority whose job it is to ensure that the quality procedure has been followed before argeeing to check in.
2. To encourage conformance by making it easier to adhere to the process than not to. For example, by wrapping tools which make life easier inside tools which make life harder e.g. checking metrics and performing regression testing as part of code check-in.

3. To encourage conformance by rewarding (verifiable) evidence of quality in the form of metrics.

These approaches often work best when combined.

3.1.1.3 Other general issues

A QA process represents a significant overhead, and will require significant resource both to define the process and to bring about the cultural changes necessary to see it effected. Once the process is 'up and running', it can be expected to require 5-20% more resource than the same tasks would, were they conducted in a non-quality-controlled fashion.

Given the degree of automation possible in the software engineering industry, a concise process with tool support should impose an overhead at the bottom of this range.

It is not possible to define an ideal, complete process at first pass, so iteration should be expected and a commitment to continuous improvement is required. Such improvement should be supported by the metrics specified in the process definition itself.

3.1.1.4 Software engineering processes

We might expect a software engineering process to include the stages of the lifecycle, with documentation and/or software of the required quality to be delivered at the end of each stage. Where the process is iterative, a deliverable is still required on each iteration though it may be more lightweight; for example, we might require an issued design document only after the implementation stage is complete, in which case a draft (and reviewed as such) design document would still be required at the end of the design stage.

The stage which each project is at should be recorded, and the project only permitted to progress to the next stage once the required deliverables have been reviewed and found to be acceptable; the end-of-stage requirements are called quality gates. Provision must be made in planning for review activities and for the output of reviews to be taken into account. Work may continue on a subsequent stage before the gate has been passed at the project managements risk.

3.1.2 Different Software qualities

We may elect to distinguish between different kinds of software, such as:

- Tools Software (e.g INSTALL).
- Prototype Software (e.g DIMMA software).
- Product Software (e.g Object Lab Code?).

We may then apply different quality constraints to each. For example, we do not give up anything in the code conventions and guidelines, whatever the quality of software targeted. However, in a prototype case, we might be less stringent on documentation, intensive testing, etc.

3.2 Software Development

Because of its complexity, the software development process needs to be supported by an adequate development environment, which automates some of the activities and helps in accomplishing others. Modern software development processes and environments typically involve the following:

1. Design standards (modularity, methods)
2. Coding standards
3. Software reuse
4. Software integration
5. Use of external software
6. Source control management
7. Build tools.
8. Testing framework
9. A controlled development environment

3.2.1 Design standards

3.2.1.1 Modularity Issues

This is related to the implementation strategy, and can rely on software modelling issues such as object-oriented techniques. The issues that can be mentioned here are:

- Separate clearly different components into different modules.
- Think in an object-oriented style (needs some training). Some references are good to read on this issue.

The developed code should be modular, with well-defined and commented interfaces between modules (minimizing the use of global variables).

3.2.1.2 Methods and notations

There are a wide range of design methods and associated notations. Use of such methods may provide the following benefits:

- They enable software engineers to understand each others work more easily
- They provide a checklist of areas to be covered and helps to ensure that important issues are not overlooked
- They provide a process to be followed during design activity, which facilitates planning and management
- They enable tool support such as code generation

In practice, existing design methods are incomplete and poorly defined, and are usually specialised in-house (through guidelines oh “how the method should be applied”). In addition, some methods are better at certain kinds of problem than others. Methods change and move in and out of fashion, and tool support varies in quality. Complete code generation, for example, is rare and found only in niche products.

There are a range of options for use of standard methods, from prescribing one after evaluation to ignoring them completely and leaving engineers to work as

they see fit. In practice, an agreed notation is recommended, but the benefits of other aspects of methods are debatable.

3.2.2 Coding standards

3.2.2.1 Programming Language

The main language currently used in APM is C/C++ although this is potentially subject to change. The situation is further complicated by there being different variants of some languages. For example, there are some minor differences between different C/C++ versions (e.g K&R, ANSI C++). In situations like this it is prudent to adopt a particular flavour for new code: the current 'standard' or the one in greatest (potential) use, for example. Obviously backwards compatibility must be maintained with existing code.

In the case of C/C++, it is recommended that the ANSI standard variant is adopted. This guarantees code stability and wide deployment. Even if some features of the ANSI C++ language are not yet supported by all available compilers, e.g. The GNU C++ compiler doesn't support ANSI C++ exceptions, it is generally possible and desirable to restrict code to use only the subset supported by most compilers.

It is also good practise, to compile code with the highest warning level and with different C/C++ compilers, to ensure compiler-independence (we cannot guarantee that the customer will use the same compiler as ours).Using different compilers, also has the advantage of detecting the greatest number of errors at compile-time (some potential bugs are detected by some compilers and not by others).

The above issues all pertain to development standards and the requirement is to document the decisions and procedures. The coding standards document as proposed in §3.2.2 would seem to be an appropriate place.

3.2.2.2 Coding conventions

To enable other developers to easily read, understand, modify and maintain existing code, we need to establish coding conventions that all APM code will respect. This will also facilitate cross-code reviewing.

Coding conventions may encompass issues like:

- Constructs of the language to use or to avoid (e.g "goto"s).
- Naming conventions of files, variables, macros, constants, types, functions, classes, etc.
- Memory allocation (e.g Always test the validity of a pointer returned by new).
- Portability issues.
- Indentation and spacing conventions.
- Code splitting issues (e.g What to put in header files and in source files).
- Source documentation (e.g Copyright, Module Identification).

Coding conventions can be split up into rules (mandatory) and guidelines (suggestions). It is suggested that these be structured along the lines of a language independent base set with language specific supplements.

It would also be beneficial to support these conventions by associated checking tools, (like we already do to check our html pages conformance), even if it's clear that not all conventions can be checked automatically.

3.2.2.3 Documenting source

Documentation describing a software component may take several forms. For example, the description of the high level design often takes the form of an independent document, prepared using a tool such as Framemaker. Alternatively, such a design may be expressed as a model using a suitable CASE tool. Further discussion of this type of documentation is outside the scope of this paper.

It is argued that much low level documentation belongs with the code, for example, the description of a procedure's functionality. This aids code readability and helps increase the probability that the description will match the code.

Code documenting is often seen by programmers as an unwelcome task and is often postponed to the end of the development phase (all too frequently it becomes postponed indefinitely). Consequently, it is advisable to mandate a certain minimum level of comments, which should be part of the coding standards.

What should be documented is a difficult question to answer. It is obvious that documenting each source line is not the objective. However, guidelines like documenting the purpose of each function, its interface, algorithms and portions of code semantically related inside a function is considered desirable.

Whilst the benefits are clear, existing software development environments offer little support for this activity. For instance, there is usually no way of semantically associating a comment with a portion of code, or to be automatically notified on a code change, so that the associated comments can be updated accordingly.

3.2.2.4 Portability and Standards

We have already mentioned adopting the ANSI C/C++ as the standard coding language. However, a software component is never self-contained. For example, a component will often use the OS APIs to access OS services and GUI API's to access the Windowing system services. To maximise portability and interoperability, the developed code should be minimally dependent on proprietary interfaces and services, and must use standard interfaces for the services it makes use of, where available. For example, on Unix systems, it is advisable to use OS services reflected in the POSIX API. If it is not possible to do so, the code that uses non-standard interfaces should be documented and optionally compiled (when the proprietary interface is known to be existent), so as to facilitate portability. For instance, the GNU autoconf utility allows for testing on globally set flags to determine whether the target system contains a particular feature or not. In those cases, the programmer must write two versions of the code: one with the assumption of the presence of the feature and one without.

3.2.3 Software reuse

APM mainly develops prototypes to demonstrate the value of its research into innovative ideas in distributed computing. Up to now, there has been no code reuse strategy within the company. New prototypes are always developed from

scratch, which hinders productivity. Hence the need for developing a strategy for code reuse.

There are two sides to the issue of reuse:

1. How to build reusable software
2. How to identify candidates for reuse, and how to decide whether reuse is appropriate or not.

The first issue is essentially about building software which is of good quality and is widely applicable. As such, it is a design issue which is discussed briefly in §3.2.1.

The second issue is more relevant to source management. Some way of controlling and describing software components must be found, so that existing components can be matched against new requirements. This could be a manual, paper based system or could take the form of a software repository with associated browsing tools.

3.2.3.1 Management of software repository

We are in favour of an APM-wide source code library (or repository), where code deemed to be useful to many projects, or which might be re-used in the future is put in the library and documented (e.g String, Trader, URL, Interface Reference,...).

Adding, replacing or removing items from the library must follow a documented procedure, to ensure that any change to the library doesn't break the existing software. We need also to document what is in the library, and offer facilities to search for items inside it. A librarian is the person who looks after these activities and ensures the library integrity. He or she is also responsible for setting up access rights for who can read/change/delete items from the library.

It is very important to automate the business of searching, as re-use often doesn't happen because:

- Timescales are often tight, and it is difficult to justify effort spent either making software more re-usable, or searching for existing software to re-use, because neither of these activities necessarily benefit the immediate project.
- Developers are often reluctant to modify anothers code: preferring instead to solve the problem themselves, because this is more interesting, and gives them a more complete understanding of the problem.

In practice, re-use initiatives only seem to work with some kind of incentive. This may be aimed at the individual developers (e.g. some sort of bonus), or the projects they work on (e.g. extra resources assigned to projects). Whatever incentive is used, it should be applied to both the re-users, and the producer of the re-used software.

Maintenance of re-used software may become an issue. There should always be an owner assigned to each component, who is responsible for its maintenance. Ideally, ownership of reused software is adopted by software librarians in the long term, as they have no project bias. In practice there are not likely to be enough librarians, so the most likely owner is the original developer if available.

3.2.4 Software integration

When different developers, or groups of developers, are producing components of a solution, these must eventually be integrated.This may involve only the new components, or it may require that new components be combined with existing code. In either case, this may require changes to some of the components. The situation may be further complicated by simultaneous working on a particular component.

Since there may be a conflict of interest between the parties performing the development, it is suggested that the responsibility for integration be assigned to an independent party, e.g. a software librarian. This has the additional benefit of providing greater consistency since all developed code would pass through the hands of a single authority.

A related point is that developers producing robust prototypes are seldom knowledgable about or even particularly interested in platform specific issues. This argues for the build and installation process being in the hands of an integration authority.

3.2.5 Use of external software

It is all very well to lay down guidelines for software written exclusively in-house, but this does not address the problems of:

1. Use of external software internally
2. Development of products which include external software

3.2.5.1 Internal use of external software

External software must be built and configured, then version managed. It should be built and configured in a consistent way for all platforms as part of ensuring a consistent environment. This can be facilitated by:

1. Only holding a single copy of source for each version of each product.
2. Only holding a single copy of binary for each version of each product for each platform.
3. Publicising which components are held under this control.
4. Automation to enable groups of components to be rebuilt together, e.g. all components for one platform, all platform-specific variants of one component.
5. Use of files to describe any parameters which are not described by the components own configuration files.

The main aim is to prevent the proliferation of different copies of the same thing, and to remove the need to know anything about the component in order to rebuild it. In other words, external software needs to be incorporated in the regular source control management system used for other components.

3.2.5.2 Inclusion of external software in products

Software produced externally may well not be quality conformant. The fundamental question is whether to:

- modify external components to adhere to internal standards
- modify our standards to be consistent with external components when we use them

- some position in between, such as wrapping external software to make it appear conformant and planning to replace it gradually over time or make it conformant only with respect to certain key aspects.

Where an external source component is large and is modified only slightly, it is usually preferable to keep the modifications in line with the style of the component. For small components with larger changes, it may be worth bringing all the source in line with internal guidelines, but it may be difficult to justify the time. As a guideline to begin with, a component of up to 1000 lines is probably worth making conformant even if unmodified, whereas a component of 10,000 lines isn't worth it unless it is anticipated that, after modification, at least 50% will be new or modified for reasons other than quality.

The following points should be considered:

1. Modification of the component may violate some licensing concerns or prejudice existing support for the component.
2. The external component may be a widely known one, in which case modification will frustrate users who know the component in its original form.
3. The component may be constructed in such a way that it interferes with the operation of internal tools. This implies a deficiency in the tools, which may only be overcome by modification of the component: build tools are a likely candidate for such problems.
4. There is a stronger incentive for modifying components which play a more important or central role in our own products, but less value to be gained where the component is peripheral or incidental.
5. If our product will include a recognised external component which can clearly be distinguished from the software produced internally, there may be less need to guarantee its quality; this is only significant where the external component can be replaced with some alternative by the customer (e.g. if we ship a free dbms which the customer is expected to replace with their favourite product, such as oracle or sybase).
6. In situations where a distinction is made between parts of a product which are conformant and parts which are not, it is mandatory that all errors which are caused by non-conformant parts should be notified to the user as such. It is also preferable to include some commentary on the non-conformant parts to describe the nature of the non-conformance (e.g. source code complexity, results of testing.
7. If there is a compelling reason for altering an external component, it may be possible to persuade the producer of the component either to alter it, or to incorporate alterations made by APM into a subsequent version.

Irrespective of whether external software is modified or not, if it is to be included in a product then it should be kept under strict version control in the same fashion as internal components.

3.2.6 Source control management

A Source Control Management (SCM) system deals with all the functions related to managing software during its production phase and afterwards. It includes procedures for archiving, identifying, keeping track of code written and used.

3.2.6.1 Definitions

We define some terms that we will use in the rest of the section.

1. Basic Component

We define a “basic component” as the smallest piece of software that is managed by software development tools as a single unit (e.g source control, build system). This can be a file, a set of files or an entire directory hierarchy.

2. Composite Component

A “composite component” is defined in terms of other components (basic or composite). A composite component can be defined statically by enumerating the components that comprise it. It can also be defined by a property or a predicate, which is evaluated when the component is to be manipulated (e.g the set of files which have been modified by “yl” after 13 March 1995 and before 15 April 1995).

3. Variant component

A variant component is a composite component whose constituent components are semantically equivalent but are adapted to different environments or tuned for different purposes (e.g different implementations for different platforms, recovery blocks for fault-tolerance).

3.2.6.2 Requirements

The requirements that we see as valuable in a source control management tool are:

1. Ability to support the three flavours of components defined above.
1. Management of different versions and variants of a component.
2. Sophisticated version naming (by date, author, project, etc.).
3. Simultaneous use of a component (automatic merge tools).
4. Ability to trace back the history of a component (audit trails).
5. Minimum disk space (e.g use of deltas).
6. Independent of source structure.

3.2.7 Build process

3.2.7.1 Requirements

The build process is the set of procedures required to obtain an immediate usable (executable) form of a software component. In the context of APM, we are interested in a build process that has the following properties:

1. Cross-platform (at least on major flavours of Unix and Windows-NT).
2. Independent from the SCM system, so that we can ship the source and its build procedures without its source control system (allows the customer to use a different SCM system).
3. Can be delivered with source code (reasonable size, unencumbered from licensing issues, etc.).

Since APM’s products are basically shipped in source form, we are constrained to ship the build procedures with the shipped software which puts constraints on the build process to be used. Of the current options, the current favourite is to use “make” as the basic build utility (with some associated scripts), because of its wide-deployment and its availability on major platforms.

Current build procedures make extensive use of shell scripts and are hence inherently linked to Unix. This is unsatisfactory and makes it difficult to port to non-Unix platforms such as Windows NT.

3.2.8 Testing

Testing usually involves first executing the software and modifying it until it appears on informal inspection to perform its central functions correctly, and then developing a test harness to verify the behaviour of the software more thoroughly and formally.

3.2.8.1 Debugging

Usually, the debugging activity is the longest in the software development process. The debugging phase ensures that before the software is handed out for intensive testing, it is free of the most common bugs, and runs at least for the most frequently exercised cases. Coding conventions and wide error - coverage compilers help in avoiding and detecting some common bugs (e.g if (x = y)) and hence shorten the length of this phase.

The debugging activity is often slow and may be difficult. Hence, it needs to be supported by powerful tools to ease the developer's activity. The alternative of inserting debugging statements into the code to trace the software activity, pollutes it and makes it difficult to read. Moreover, some of this debugging code remains after the debugging phase, and adds to the complexity and maintenance cost of the software. It is therefore preferable for the majority of one-off debugging, to use source-level debuggers (e.g. gdb), preferably with a graphical and friendly interface. These may be used to set breakpoints, examine the state of program's variables and the execution stack at different stages of the program's execution.

In addition to examining the program state, it is also desirable for the debugger to permit the explicit execution of procedures that are not part of the software being debugged (i.e. contained in a debug library), to check for the occurrence of conditions that are of interest to the developer.

The use of assertions when developing software may help the early detection of errors. Assertions are boolean statements that must evaluate to TRUE if the program is behaving correctly (like postconditions in Hoare's logic). If not, an error is raised and typically the program terminates giving full detail about the error that it encounters.

3.2.8.2 Informal testing

Informal testing may be conducted using a debugging tool such as gdb or sparcworks, or by adding appropriate debug statements to the code. It is thought that good debugging tools, especially those with GUIs, represent the most efficient approach. The usability of such tools is very important; if a sufficiently usable and portable tool can be identified then it is the preferred route.

Cross-platform development means that developers may need to learn several different debugging tools. In these cases so it may be more expedient to use debug statements. Also, in the absence of more formal approaches, debug statements may be useful in achieving some measure of regression testing capability.

3.2.8.3 Formal testing

Formal testing involves developing test scripts to specify a set of test cases, which exercise the code to the required level of thoroughness and automatically check results against correct values supplied by the tester.

Usually, each test script specifies the operation to be invoked, the input parameters and the expected results. A test harness can then be used to execute the tests specified by the scripts, and produce a report which identifies any errors; i.e. cases where the actual postconditions do not match the expected postconditions. Common postconditions are values for returned parameters, the state of files and database tables, the existence of objects and processes.

It is very important that this report should also include coverage metrics (e.g. condition and statement coverage) indicating the extent to which the software has been exercised by the tests: further test cases can be specified until the level of coverage exceeds the prescribed minimum.

A variety of test harness products of this kind are available, such as Cantata, McCabe and DejaGnu. Some such tools provide a range of different useful metrics in addition to coverage metrics. Some important characteristics to look for are:

- The presence of the test harness does not affect the behaviour of the component being tested (some products affect the behaviour of certain system calls).
- The test harness code, whether library or auto-generated, is well-behaved in terms of potential incompatibilities such as symbol names.
- Support for specification of expected postconditions for variables is sympathetic to use of user-defined types.

In practice there are some situations where a non-specific test harness of the kind described above would be inappropriate. For example, one approach to testing of a communications stack might be to write a specialised script which instantiated two instances of the stack and exchanged messages with itself via both instances repeatedly, checking messages contents algorithmically and varying non-functional characteristics such as the load; in such situations, coverage metrics may not be the most important measurements of testing thoroughness.

The testing process should be able to accommodate specialised cases such as this. There are many different approaches to testing (e.g. mutation testing, random testing), and no one approach is universally suitable.

3.2.8.4 Management of the test process

When components are submitted to version control and flagged as ‘unit tested’, it may be desirable to have test cases automatically executed and checked before check-in is allowed. In order to support this, it would be necessary for specialised test harnesses to produce compatible output.

Test scripts and associated reports should always be kept under version control in the same way as standard software components.

Testing should also be part of the build process to ensure that what it is built is usable: testing of this kind need only ensure that common installation and build errors, such as absence or misconfiguration, have not occurred.

3.2.9 Development environment

3.2.9.1 Requirements for a development environment

The major choice here is whether to adopt an integrated development environment, with facilities for editing, building and debugging; or whether to use a platforms native tools, integrated by some form of scripting language.

Some issues to consider are:

- Portability. Are the tool(s) available for different platforms.
- Open vs closed. Can an external build process be used for example (the issue here is that we would not wish to ship the entire development environment with the source).
- Flexibility. For example, is it possible to substitute alternative tools within an integrated environment.
- Consistency of use. Having to learn many different tools is time consuming and inefficient.
- Integration. Can models or documentation of existing code be easily accessed to facilitate code re-use.Can this be supported by an automated repository.

3.2.9.2 Browsing Existing Code

A code browser is a valuable tool for accelerating the understanding of existing code. Basically, the tool analyses source components and compiles the relevant information into a database which allows to answer questions like:

1. Where this function is defined?
2. What are the functions that call this function?
3. Where this global variable is defined?

This facilitates the understanding of externally acquired code (e.g. Mosaic code), and helps the maintenance of existing code (for instance, adding a parameter to a function is less prone to error because the tool can locate all the locations where this function is used).

3.2.9.3 Consistent development environment

Often, several developers and testers may share common components. In such cases, it is important that behaviour encountered by one person is reproducible by the others. In general, this is unlikely to be achieved unless all developers share a standard (cross-platform) user environment, and all machines used for development are set up in a consistent fashion.

Common stumbling blocks include:

1. Different paths.
2. The presence of different software tools or different versions of the same tool, such as compilers and linkers.
3. The presence of personalised configuration files for tools.
4. Different values for environment variables.

In addition, the use of a standard environment reduces the overall administration overhead, facilitates automation and avoids the perpetual problems of file permissions.

A standard user account should be defined and agreed by the development community, and assigned a name (e.g. 'developer') and password. All configuration files which comprise this environment should be kept under strict version control.

Each machine should have access to standardised areas of file systems set aside for holding development tools and the like in a controlled way, and the standard environment should not use resources that are not controlled in this way.

One potential concern with such standardisation is that it may cause software to be tested less thoroughly, because the software is not tried under a range of different environments internally; and yet diversity in end-user environments is inevitable, unless the software is for in-house demos only.

However, this concern is better addressed by performing tests of behaviour in different environments explicitly as part of formal testing, not randomly through coincidental variation in-house environments.

Note also that the use of a standard developer account does not preclude use of individual accounts, provided that the two are suitably dissociated.

3.2.10 Conclusions

The production of software is the main activity of an APM software developer, and an important one for the APM business, because we ship products mainly in a source form to parties who have not been involved in their development, and who will potentially build on our products. This issue will become more critical with the Object Lab perspective.

An APM software developer should be made aware of the importance of producing good quality code. Following conventions and procedures when developing new code, or dealing with existing code, shouldn't be seen by developers as a burden, but as a clear benefit in the overall scheme of things. It is true that the learning curve may take some time before developers master the conventions and the procedures, and see the benefits of such a process. But once done, less time will be spent on doing some repetitive tedious tasks, and it will become easier to focus on the more important tasks.

The programmer's awareness of the importance to follow the conventions and the procedures, is the only guarantee to ensure good quality; tools and procedures can only help into achieving that goal. Therefore, his/her involvement in setting up these conventions, and providing feedback on them is vital to the success of the process in the future.

3.3 Software Maintenance

Maintenance is generally held to consume at least 80% of the total resources expended on supported software products, so it is obviously important to promote production of more maintainable software, not just to maintain it effectively after it is released.

By the time software reaches the maintenance phase of its life, many of the decisions which most strongly affect maintainability have already been taken; in fact arguably, the quality of the design is the most important single factor.

An approach to software maintenance can be discussed in terms of the following issues:

1. Bug tracking
2. Fault reporting and tracking
3. Change request management
4. Regression testing.

3.3.1 Fault reporting and tracking

Erroneous behaviour of the software may be observed either in-house, or externally. In both cases, it is very important that the fault reporting process should:

1. Be as simple and time-efficient as possible, as this saves in-house resources and encourages both internal and external users of the system to take the necessary trouble to record fault incidences.
2. Promote the supply of a consistent set of information about faults observed, in a consistent form. This makes manual handling of fault reports more efficient, and simplifies automation.

One way of achieving this might be to provide the user with a utility which:

- Allows the customer to fill in a standardised form.
- Holds a simple, concise description of what each field on the form means.
- Sends the report to the support team for the right product and version, for example via email.
- Collects information about the environment automatically, if requested.

Ideally, the same utility should also be used in-house. Failing this, it would be desirable for internal and external utilities to produce similar fault reports. It may be worth tying this utility to the monitoring and error handling of the products themselves.

The suggested structure for fault reports is:

- Name and version of faulty product
- Name, organisation and contact information of report originator
- Estimated seriousness
- Time of occurrence
- Description of aberrant behaviour observed
- Frequency (some faults occur at specified time intervals)
- Repeated (has the fault been recreated?)
- Hardware and software platform used:

  - Machine
  - Operating system name and version
  - Network type & software
  - Names and versions of other software directly supporting product (e.g. if the product is being used with a particular DBMS)
  - Names and versions of other software being run at the time
- Description of configuration of product:
  - Installation options used
  - Configuration file contents
  - Relevant environment variables
- For products where source is distributed:
  - Compiler and linker name and version

Perhaps only some of this information should be mandatory; there is a balance between demanding enough information to be useful and demanding so much that people are not inclined to fill in the form. This problem can be alleviated greatly if some of the information gathering can be automated.

Fault reports received should be logged and assigned identifiers for future reference. Ideally these identifiers should be passed back to the originator of the report, preferably automatically.

Once logged, fault reports should be investigated; the first problem is to prioritise them. This is difficult to do, since they have often been recorded externally, but there may be some benefit in asking for an 'estimated seriousness' with the fault report.

Investigation of a fault report is likely to include the following steps:

1. Decide whether the behaviour described in the report is contrary to the specified behaviour of the system.
2. Attempt to reproduce the fault from the information described in the report.
3. Analyse the fault to find the bug in the software.
4. Search through outstanding fault reports to identify other manifestations of the same bug.
5. Examine what components and products are affected by the bug, estimate the seriousness of it and the resources required to fix it.
6. Fill in a bug report form. The bug is assigned an identifier which should be added to the fault reports which the bug relates to, and these fault reports should be marked and archived.

3.3.2 Bug fixing and tracking

The suggested structure for a bug report is:

- Name and version of faulty component
- Name, organisation and contact information for developer
- Estimated seriousness
- Assigned priority, and id of budget holder.
- Time form filled in.

- Fault reports addressed by bug
- Description of bug
- List of affected source components (e.g. files, functions)
- Estimate of effort required
- Description of fix

Usually, someone with authority for budget must prioritise the bugs discovered, to say which should be fixed and in what order. This is often a significant overhead, so it may be sensible to devolve authority for certain kinds of bugs. If some authority or approval is required before the software is modified (for example, from the 'owner(s) of the component(s) affected), then this should be sought before any software is modified.

Bug fixing involves the following steps:

1. Submit bug reports to budget holder for prioritisation.
2. For each bug in turn according to prioritisation, follow the 'software change process' described below.
3. When the fix has been completed, it should be checked, signed off and archived.

The software change process:

1. Notify owners of affected components.
2. Retrieve the necessary source code components from configuration control.
3. Modify the software and test it. Comments including the developer id and the reason for the change (change request id or bug id) should be included in the source. This is a sensible time for the change to be checked by another developer.
4. Measure and check any metrics required and perform regression testing.
5. Put the modified components into configuration control.
6. Update any obsolete documentation for the component, and components which depend upon it, to come into line with the changes made.
7. Notify the owners of all relevant components and products that the change has been made, so that they can make decisions about incorporating the modifications.

3.3.3 Change request management

Change requests are requests for alterations to the specification of the system.

They are derived from change suggestions made by users and developers: these suggestions are not modelled as carefully as are fault reports, but should include contact information for the originator, and the name and version of each relevant product. They should also be assigned identifiers to support requirements traceability.

Again, it is desirable to make it as easy as possible for customers to send change requests, so it may be beneficial to include a relevant utility with products as suggested for fault reporting.

Change suggestions should be logged. They should be analysed when a new release of the software is under consideration, and one or more change

requests generated. Each change request should be a sensible, cohesive modification to the current system specification.

Change requests should be structured in a similar way to full specifications of systems, to facilitate comparison of change requests with existing specifications and with each other. Each change request document should also reference the change suggestions which it incorporates, for traceability. Change suggestions which are completely catered for by defined change requests should be marked and archived.

Change requests are then processed:

1. Change requests are prioritised and assigned to component releases.
2. The revised specification for a new release is reached by applying the change requests to the existing specification
3. The design documentation is updated.
4. Finally, the software is modified using the software change process described above.
5. When change requests are completed, they should be checked and signed off, and any appropriate metrics (e.g. time taken) should be recorded.

3.3.4 Regression testing

Regression testing involves re-applying existing test suites to ensure that existing software has not been unintentionally altered, and maintaining the test suite so that regression testing will be possible for subsequent changes.

It includes these stages:

1. Add new test cases to test suite; this involves retrieving existing test cases from configuration control for the changed components and merging them with new or modified ones.
2. Re-run all test cases for all modified components.
3. If old test cases fail, then some existing functionality has been unintentionally modified; the source code should be revisited.
4. Manually peruse the test suite for test cases that are obviously obsolete, and remove them.
5. Put the modified test suite back under configuration control.

There are all sorts of different formal and semi-formal ways of maintaining test suites and weeding out redundant cases. None of these seem to be worth the effort in practice though. Since there is little benefit to be gained, only a brief search for obsolete cases should be made.

Note that whenever it is decided to include a modified component into a release, the whole release should be regression tested. Presumably with modular components, there is little likelihood that such regression tests will fail!

3.4 Product Provision and Support

This section covers software distribution, installation and configuration. It also briefly covers the automation of product marketing and ordering.

A software provision approach can be discussed in terms of the following issues:

1. Supply of software (how do we get it to the customer)
2. Installation of software
3. Configuration of installations (altering an existing installation, including upgrade)
4. Electronic marketing and sales

3.4.1 Supply of software

3.4.1.1 Distribution media

Software may be distributed on a variety of media. It is likely that several different forms will have to be supported, depending on the type of media being used by customers. Currently, we ship on various types of tapes and on floppy disk.

In the near future, distribution is more likely to become electronic, using standard communications software such as FTP.

Certainly for the foreseeable future, APM is unlikely to ship software in sufficient volume to make distribution cost effective on media such as CD.

3.4.1.2 Platform specific support

APM currently ships its ANSAware software as source that may be configured and built on a variety of supported platforms. This implies that the configuration and building tools must be distributed with the source.

Source distribution is appropriate for proof-of-concept software such as ANSAware since it is typically used as a basis for further development. However, not everyone will wish to build a product in order to use it and this attitude is likely to be become more prevalent with the Object Labs perspective. This suggests that APM will also have to consider offering the option of platform specific binary distributions.

3.4.2 Installation of software

ANSAware is difficult to install and requires a certain minimum level of expertise. Object Lab software must be engineered so that it is easier to install and should ideally utilise the 'normal' installation utility provided by the target platform. Being familiar to the users, this should reduce the amount of specialist knowledge required to install the product.

Installation should involve minimal user intervention, at least for the default case. The software should run more or less, "out of the box", the installation offering appropriate defaults to any questions.

3.4.3 Configuration of installations

Configuration includes such things as:

- Building for a specific platform from the installed source

- Re-configuring installation options (e.g. location of trader)
- Upgrading to a new version of the software

A distribution containing source requires that the product be built (compiled and linked) for the target platform. Where the distribution is intended for a single platform, this may be automated as part of the installation procedure, provided the target is compatible with that on which the software is installed. In any case, some sort of build procedure and associated tools must be supplied with the distribution although these should not preclude the use of an alternative user-supplied mechanism. Parameters required by the build, should be obtained through some form of interactive front end.

Reconfiguration should be similarly easy (e.g. specify different characteristics from the default ones using at least a file, preferably a front end).

Upgrade to a new version should involve minimal disruption and offer the choice of overwriting the old installation or creating a new one.

These requirements imply that the configuration information must be 'soft' and separate from the binaries, i.e. it should not be necessary to re-build the software simply to alter the configuration parameters. The configuration should therefore be held in a file, preferably presented to the user through some sort of front end.

In addition, it is important to distinguish between site specific configuration parameters and user specific ones. Separate configuration files, held in user specific directories, may be required to address the latter.

3.4.4 Electronic marketing and sales

It is the intention that Object Lab present an electronic face to the world. It should be possible for customers to browse our wares, order software and report problems using an electronic front end to the network. Delivery of product using something like FTP should also be offered. The most obvious candidate technology for this is the World Wide Web.

4 Priorities

It is clear from the previous discussion that considerable work must be done to put in place a software management structure that could be considered for ISO 9000 approval. It is also evident that this must be performed in an incremental manner to spread the effort and minimise disruption. To this end, the items that must be addressed are identified and prioritised in this chapter.

It is anticipated that each item identified will require further work in determining a detailed solution. For example, in the area of tools, this might involve obtaining information and demonstrations of the capabilities of potential candidates, prior to deciding what to adopt.

The requirements fall into two main categories:

- Process
- Tool support

The following sections consider these in more detail.

4.1 Process

The most obvious omission in how we currently manage software development and the biggest impediment to obtaining ISO 9000 accreditation, is the lack of a documented process. Furthermore, it is impossible to define a QA process without first having a development process to monitor and control. Consequently, documenting the existing procedures, defining additional procedures and producing the associated standards must be considered the highest priority tasks.

The following documents are required (in descending order of priority):

- The software development process (may be more appropriate to construct this as a framework referencing other more detailed documents)
- Coding standards and guidelines
- QA process

4.2 Tool support

Tools are required both to support the process identified above and to aid the actual job of developing software.

4.2.1 Software development tools

The major decision that must be made relating to software development tools is whether to go for an integrated development environment or whether to use a collection of individual tools. In either case, the required functionality is as follows:

- Source code management
- Software library or repository (may be part of the above)
- Build procedure (compile and link)
- Source code debugger for the languages selected
- Installation and configuration utility
- Code browser
- Automated test harness
- QA metrics analyser (e.g. test coverage)

4.2.2 Process related tools

Simple tools could be produced in-house to support the software development process:

- Standard source template construction for new code
- Coding standards conformance checker
- Product browser and ordering WWW client
- Bug reporting WWW client

Note: Bens notes.

We could prioritise more effectively if we knew more about the business direction of APM.

Separate process for software production and marketing.

It might be easier to prioritise if we drew up a table of esimated effort to complete vs expected benefit and timescale for benefit to show. Very rough but might help. One such table per business direction.

prioritisation given above = development process, development standards/ guidelines, development env, QA... for each area there is process, standards, tools, development is most important area. Guidelines are usually easier than tools so some first. Next level of detail is which kinds of tool to address first - SCM seems likely along with tools to check conformance. On t'other hand theres may fave grumble that a decent debugger would improve productivity a lot very quickly...

References

[LINDEN 93]

van der Linden R. J., *An Overview of ANSA*; **AR.000.00**, APM Ltd., Cambridge U.K., May 1993.