ESA PSS-05-07 Issue 1 Revision 1
March 1995

Guide to the software operations and maintenance phase

Prepared by:
ESA Board for Software
Standardisation and Control
(BSSC)

Approved by:

The Inspector General, ESA

european space agency / agence spatiale européenne
8-10, rue Mario-Nikis, 75738 PARIS CEDEX, France

DOCUMENT STATUS SHEET

DOCUMENT STATUS SHEET			
1. DOCUMENT TITLE: **ESA PSS-05-07 Guide to the Software Operations and Maintenance Phase**			
2. ISSUE	3. REVISION	4. DATE	5. REASON FOR CHANGE
1	0	1994	First issue
1	1	1995	Minor updates for publication

Issue 1 Revision 1 approved, May 1995
Board for Software Standardisation and Control
M. Jones and U. Mortensen, co-chairmen

Issue 1 approved, 15th June 1995
Telematics Supervisory Board

Issue 1 approved by:
The Inspector General, ESA

Published by ESA Publications Division,
ESTEC, Noordwijk, The Netherlands.
Printed in the Netherlands.
ESA Price code: E1
ISSN 0379-4059

TABLE OF CONTENTS

PREFACE

This document is one of a series of guides to software engineering produced by the Board for Software Standardisation and Control (BSSC), of the European Space Agency. The guides contain advisory material for software developers conforming to ESA's Software Engineering Standards, ESA PSS-05-0. They have been compiled from discussions with software engineers, research of the software engineering literature, and experience gained from the application of the Software Engineering Standards in projects.

Levels one and two of the document tree at the time of writing are shown in Figure 1. This guide, identified by the shaded box, provides guidance about implementing the mandatory requirements for the software Operations and Maintenance Phase described in the top level document ESA PSS-05-0.

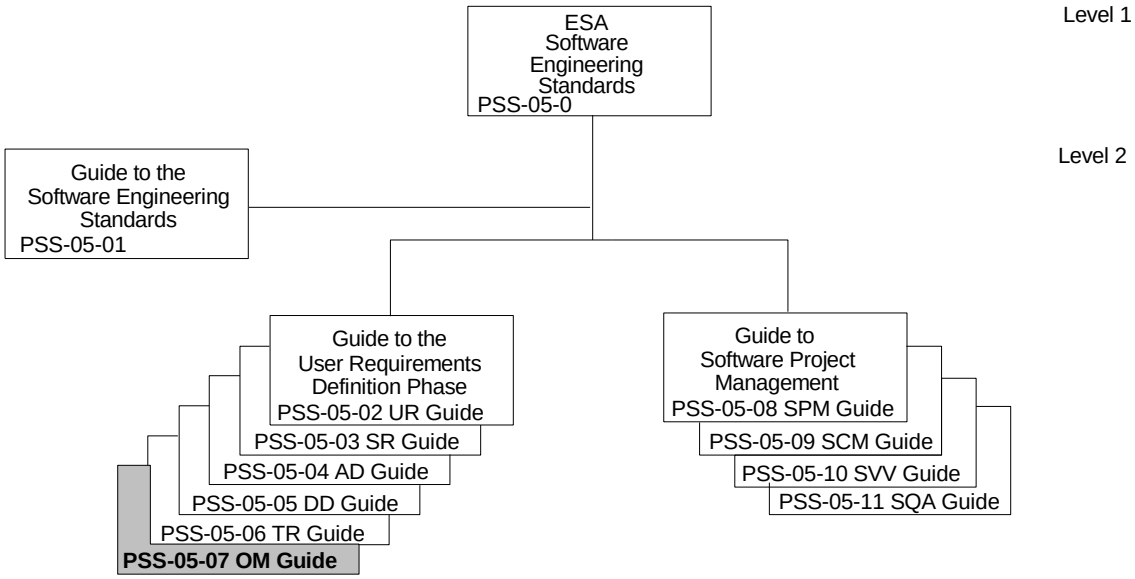

Figure 1: ESA PSS-05-0 document tree

The Guide to the Software Engineering Standards, ESA PSS-05-01, contains further information about the document tree. The interested reader should consult this guide for current information about the ESA PSS-05-0 standards and guides.

The following past and present BSSC members have contributed to the production of this guide: Carlo Mazza (chairman), Gianfranco Alvisi, Michael Jones, Bryan Melton, Daniel de Pablo and Adriaan Scheffer.

The BSSC wishes to thank Jon Fairclough for his assistance in the development of the Standards and Guides, and to all those software engineers in ESA and Industry who have made contributions.

Requests for clarifications, change proposals or any other comment concerning this guide should be addressed to:

BSSC/ESOC Secretariat
Attention of Mr M Jones
ESOC
Robert Bosch Strasse 5
D-64293 Darmstadt
Germany

BSSC/ESTEC Secretariat
Attention of Mr U Mortensen
ESTEC
Postbus 299
NL-2200 AG Noordwijk
The Netherlands

CHAPTER 1
INTRODUCTION

1.1 PURPOSE

ESA PSS-05-0 describes the software engineering standards to be applied for all deliverable software implemented for the European Space Agency (ESA), either in house or by industry [Ref 1].

ESA PSS-05-0 defines the fourth phase of the software development life cycle as the 'Transfer Phase' (TR phase). The outputs of this phase are the provisionally accepted software system, the statement of provisional acceptance, and the Software Transfer Document (STD). The software enters practical use in the next and last phase of the software life cycle, the 'Operations and Maintenance' (OM) phase.

The OM Phase is the 'operational' phase of the life cycle in which users operate the software and utilise the end products and services it provides. The developers provide maintenance and user support until the software is finally accepted, after which a maintenance organisation becomes responsible for it.

This document describes how to maintain the software, support operations, and how to produce the 'Project History Document' (PHD). This document should be read by everyone involved with software maintenance or software operations support. The software project manager of the development organisation should read the chapter on the Project History Document.

1.2 OVERVIEW

Chapter 2 discusses the OM phase. Chapters 3 and 4 discuss methods and tools for software maintenance. Chapter 5 describes how to write the PHD. Chapter 6 discusses life cycle management activities.

All the mandatory practices in ESA PSS-05-0 relevant to the software operations and maintenance phase are repeated in this document. The identifier of the practice is added in parentheses to mark a repetition. This document contains no new mandatory practices.

This page is intentionally left blank

CHAPTER 2
THE OPERATIONS AND MAINTENANCE PHASE

2.1 INTRODUCTION

Operations provide a product or a service to end users. This guide discusses operations from the point of view of their interactions with software maintenance activities and the support activities needed to use the software efficiently and effectively.

Software maintenance is 'the process of modifying a software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment' [Ref 7]. Maintenance is always necessary to keep software usable and useful. Often there are very tight constraints on changing software and optimum solutions can be difficult to find. Such constraints make maintenance a challenge; contrast the development phase, when designers have considerably more freedom in the type of solution they can adopt.

Software maintenance activities can be classified as:

- corrective;
- perfective;
- adaptive.

Corrective maintenance removes software faults. Corrective maintenance should be the overriding priority of the software maintenance team.

Perfective maintenance improves the system without changing its functionality. The objective of perfective maintenance should be to prevent failures and optimise the software. This might be done, for example, by modifying the components that have the highest failure rate, or components whose performance can be cost-effectively improved [Ref 11].

Adaptive maintenance modifies the software to keep it up to date with its environment. Users, hardware platforms and other systems all make up the environment of a software system. Adaptive maintenance may be needed because of changes in the user requirements, changes in the target platform, or changes in external interfaces.

Minor adaptive changes (e.g. addition of a new command parameter) may be handled by the normal maintenance process. Major adaptive changes (e.g. addition of costly new user requirements, or porting the software to a new platform) should be carried out as a separate development project (See Chapter 5).

The operations and maintenance phase starts when the initiator provisionally accepts the software. The phase ends when the software is taken out of use. The phase is divided into two periods by the final acceptance milestone (OM03). All the acceptance tests must have been successfully completed before final acceptance (OM02).

There must be a maintenance organisation for every software product in operational use (OM04). The developers are responsible for software maintenance and user support until final acceptance. Responsibility for these activities passes to a maintenance team upon final acceptance. The software project manager leads the developers. Similarly, the 'software maintenance manager' leads the maintenance team. Software project managers and software maintenance managers are collectively called 'software managers'.

The mandatory inputs to the operations and maintenance phase are the provisionally accepted software system, the statement of provisional acceptance and the Software Transfer Document (STD). Some of the plans from the development phases may also be input.

The developer writes the Project History Document (PHD) during the warranty period. This gives an overview of the whole project and a summary account of the problems and performance of the software during the warranty period. This document should be input to the Software Review Board (SRB) that recommends about final acceptance. The PHD is delivered to the initiator at final acceptance (OM10).

Before final acceptance, the activities of the development team are controlled by the SPMP/TR (OM01). The development team will normally continue to use the SCMP, SVVP and SQAP from the earlier phases. After final acceptance, the maintenance team works according to its own plans. The new plans may reuse plans that the development team made, if appropriate. The maintenance team may have a different configuration management system, for example, but continue to employ the test tools and test software used by the developers.

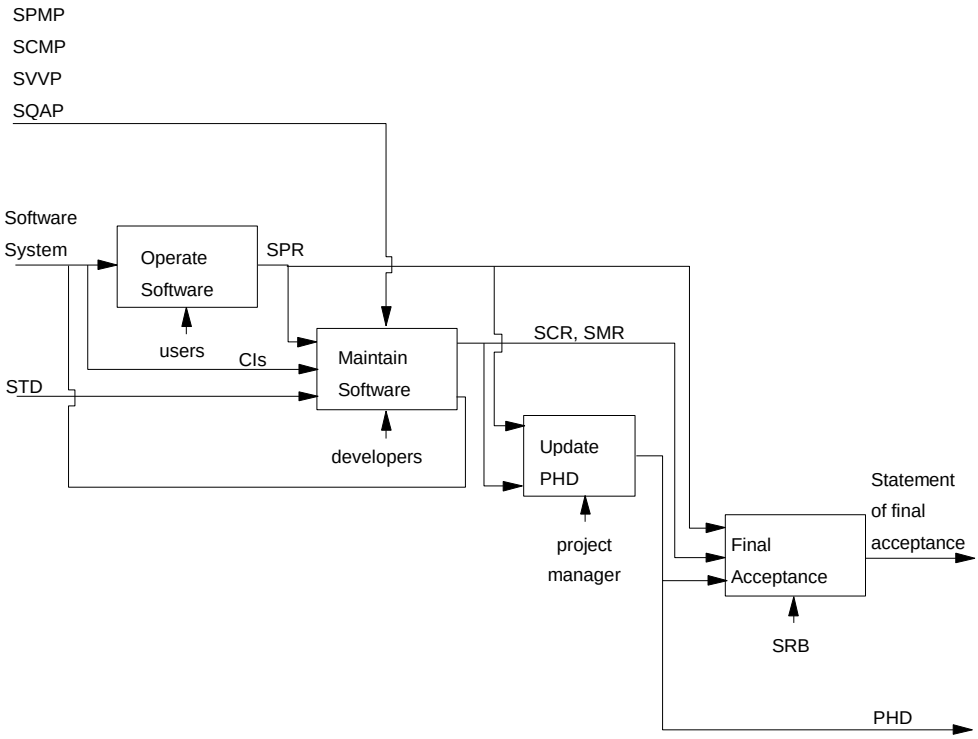

Figure 2.1A: OM phase activities before final acceptance

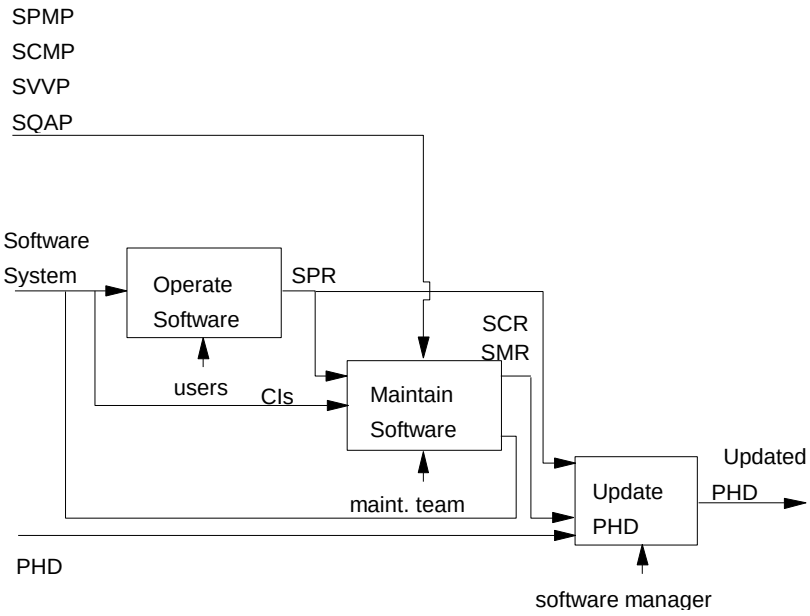

Figure 2.1B: OM phase activities after final acceptance

Figures 2.1A and 2.1B show the activities, inputs and outputs of the phase, and the information flows between them. The arrows into the bottom of each box indicate the group responsible for each activity. The following sections discuss in more detail the activities of operate software, maintain software, update Project History Document and final acceptance.

2.2 OPERATE SOFTWARE

The way software is operated varies from system to system and therefore cannot be discussed in this guide. However there are two activities that occur during most software operations:

- user support;
- problem reporting.

These activities are discussed in the following sections.

2.2.1 User support

There are two types of user:

- end user;
- operator.

An 'end user' utilises the products or services of a system. An 'operator' controls and monitors the hardware and software of a system. A user may be an end user, an operator, or both.

User support activities include:

- training users to operate the software and understand the products and services;
- providing direct assistance during operations;
- set-up;
- data management.

Users should receive training. The amount of training depends upon the experience of the users and the complexity or novelty of the software. The training may range from allowing the users time to read the SUM and familiarise themselves with the software, to a course provided by experts, possibly from the development or maintenance teams.

Software User Manuals and training are often insufficient to enable users to operate the software in all situations and deal with problems. Users may require:

- direct assistance from experts in the development or maintenance teams;
- help desks.

Full-time direct assistance from experts is normally required to support critical activities. Part-time direct assistance from experts is often sufficient for software that is not critical and has few users.

When the number of users becomes too large for the experts to be able to combine user support activities with their software maintenance activities, a help desk is normally established to:

- provide users with advice, news and other information about the software;
- receive problem reports and decide how they should be handled.

While help desk staff do not need to have expert knowledge of the design and code, they should have detailed knowledge of how to operate it and how to solve simple problems.

The set-up of a software system may be a user support activity when the software set-up is complex, shared by multiple users, or requires experts to change it to minimise the risk of error.

Data management is a common user support activity, and may include:

- configuring data files for users;
- managing disk storage resources;
- backup and archiving.

2.2.2 Problem reporting

Users should document problems in Software Problem Reports (SPRs). These should be genuine problems that the user believes lie in the software, not problems arising from unfamiliarity with it.

Each SPR should report one and only one problem and contain:

- software configuration item title or name;
- software configuration item version or release number;
- priority of the problem with respect to other problems;
- a description of the problem;
- operating environment;
- recommended solution (if possible).

The priority of a problem has two dimensions:

- criticality (critical/non-critical);
- urgency (urgent/routine).

The person reporting the problem should decide whether it is critical or non-critical. A problem is critical if the software or an essential feature of the software is unavailable. The person should also decide whether the solution is required as soon as possible (urgent) or when the Software Review Board decides is best (routine).

Users should attempt to describe the problem and the operating environment as accurately as possible to help the software engineers to reproduce the problem. Printouts and log files may be attached to the Software Problem Report to assist problem diagnosis.

Problems may occur because the software does not have specific capabilities or comply with some constraints. Users should document such omissions in SPRs, not in a modification to the User Requirements Document. The URD may be modified at a later stage when and if the Software Review Board (SRB) approves the new user requirement.

Figure 2.2 shows the life cycle of a software problem report. An SPR is prepared and submitted to the maintenance team for diagnosis. The maintenance team prepares a Software Change Request (SCR) if they decide that a software change is required to solve the problem. The SPR and related SCR are then considered at a Software Review Board meeting.

The Software Review Board decides upon one of four possible outcomes for the SPR:

- reject the SPR;
- update the software based upon the related SCR;

- action someone to carry out further diagnosis;
- close the SPR because the update has been completed.

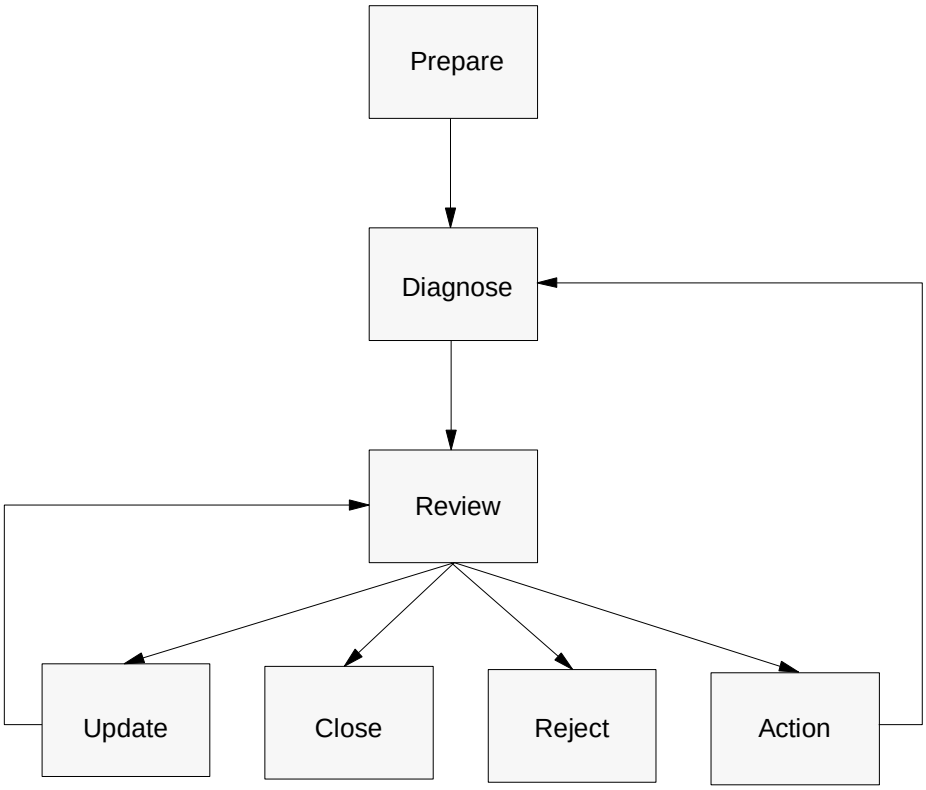

Figure 2.2: The life cycle of an SPR

2.3 MAINTAIN SOFTWARE

Software maintenance should be a controlled process that ensures that the software continues to meet the needs of the end user. This process consists of the following activities:

- change software;
- release software;
- install release
- validate release.

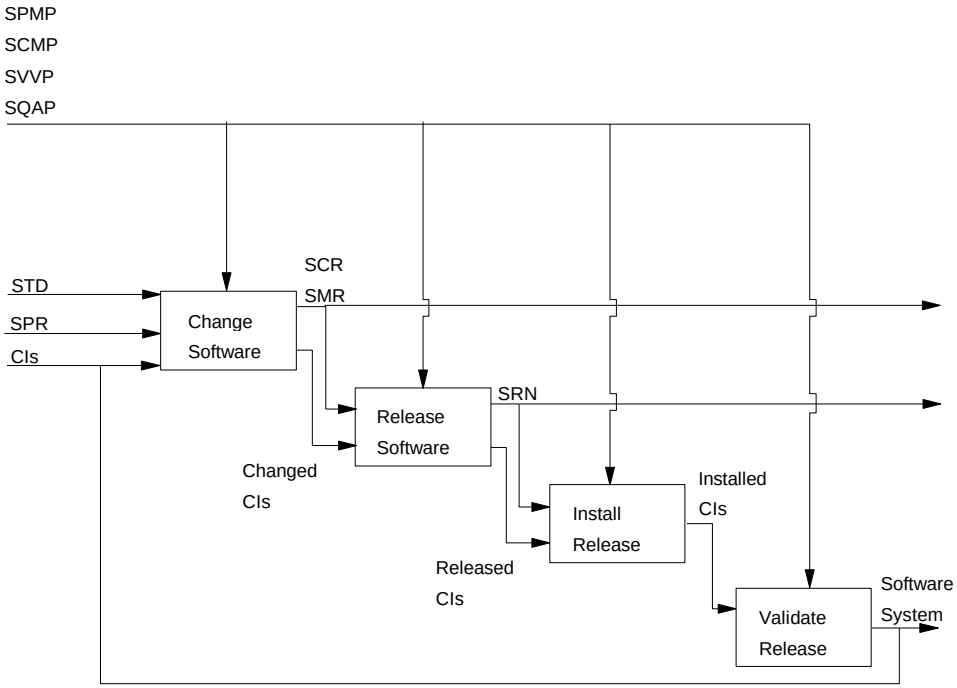

Figure 2.3: The software maintenance process

Figure 2.3 shows the inputs and outputs of each activity, which are discussed in more detail in the following sections.

2.3.1 Change Software

Software change should be governed by a change control process defined in the Software Configuration Management Plan. All change control processes should be based upon the code change control process described in ESA PSS-05-09, Guide to Software Configuration Management [Ref 4], shown in Figure 2.3.1. Configuration management tools should be used to control software change. These are described in the Guide to Software Configuration Management.

Software change is a four stage process:

- diagnose problems;
- review change requests;
- modify software;
- verify software.

The following sections describe each stage.

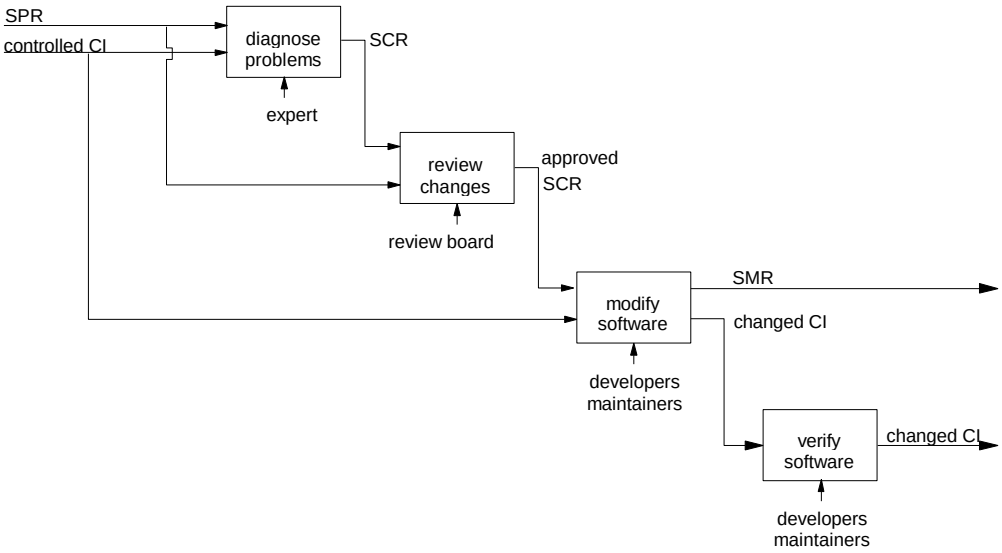

Figure 2.3.1: The basic change control process

2.3.1.1 Diagnose Problems

Software Problem Reports (SPRs) arising from software operations are collected and assigned to an expert software engineer from the maintenance team, who examines software configuration items and diagnoses the cause of the problem. The software engineer may recommend software changes in a Software Change Request (SCR).

The steps in problem diagnosis are:

- examine SPR;
- reproduce the problem, if possible;
- examine code and/or documentation;
- identify the fault;
- identify the cause;
- write a Software Change Request, if necessary.

Examination of code and documentation may require backtracking from code and Software User Manual through the Detailed Design Document, the Architectural Design Document, the Software Requirements Document and ultimately the User Requirements Document.

Software engineers often diagnose a problem by building a variant configuration item that:

- is able to work with debugging tools;
- contains diagnostic code.

The software engineers run the variant and attempt to reproduce the problem and understand why it happened. When they have made a diagnosis they may insert prototype code to solve the problem. They then test that the problem does not recur. If it does not, the person responsible for modifying the software may use the prototype code as a starting point. The maintenance team should be wary of the first solution that works.

When the cause of a problem has been found, the software engineer should request a change to the software to prevent the problem recurring. Every Software Change Request (SCR) should contain the following information:

- software configuration item title or name;
- software configuration item version or release number;
- changes required;
- priority of the request;
- responsible staff;
- estimated start date, end date and manpower effort.

The software engineer provides the first three pieces of information. The software manager should define the last three.

The specification of the changes should be detailed enough to allow management to decide upon their necessity and practicality. The SCR does not need to contain the detailed design of the change. Any changes identified by the person who diagnosed the problem, such as marked-up listings of source code and pages of documentation, should be attached to the SCR.

Like software problems, the priority of a software change request has two dimensions:

- criticality (critical/non-critical);
- urgency (urgent/routine).

The software manager should decide whether the change request is critical or non-critical. A change is critical if it has a major impact on either the software behaviour or the maintenance budget. The criticality of a software change request is evaluated differently from the criticality of a software problem. For example an SPR may be classified as critical because an essential feature of the software is not available. The corresponding SCR may be non-critical because a single line of code is diagnosed as causing the problem, and this can be easily modified.

The software manager should decide whether the change is urgent or routine. A change is urgent if it has to be implemented and released to the user as soon as possible. Routine changes are released in convenient groups according to the release schedule in the software project management plan. The software manager normally gives the same urgency rating as the user, although he or she has the right to award a different rating.

The SCR is normally presented as a form, but it may be a document. A change request document should contain sections on:

- user requirements document changes;
- software requirements document changes;
- architectural design document changes;
- detailed design document changes;
- software project management plan update;
- software verification and validation plan for the change;
- software quality assurance plan for the change.

Each section should include or reference the corresponding development documents as appropriate. For example a change to the code to remedy a detailed design fault might:

- put 'no change' in the URD, SRD and ADD change sections;
- contain a new version of the DDD component description;

- contain a listing of the code with a prototype modification suggested by the expert who made the change request;
- contain a unit test design, unit test case and unit test procedure for verifying the code change;
- identify the integration, system and acceptance test cases that should be rerun;
- contain a work package description for the change;
- identify the software release that will contain the change.

Investigation of a software problem may reveal the need to implement a temporary 'workaround' solution to a problem pending the implementation of a complete solution. The workaround and the complete solution should have a separate SCR.

2.3.1.2 Review Changes

The Software Review Board (SRB) must authorise all changes to the software (OM08). The SRB should consist of people who have sufficient authority to resolve any problems with the software. The software manager and software quality assurance engineer should always be members. Users are normally members unless the software is part of a larger system. In this case the system manager is a member, and represents the interests of the whole system and the users.

The SRB should delegate responsibility for filtering SPRs and SCRs to the software manager. The order of filtering is:

- criticality (critical/non-critical);
- urgency (urgent/routine).

The decision tree shown in Figure 2.3.1.2 illustrates the four possible combinations of urgency and criticality, and defines what action should be taken.

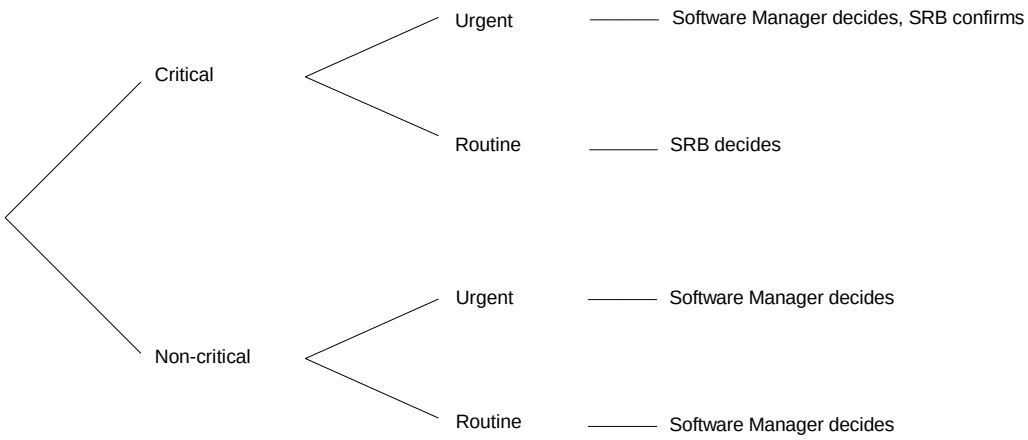

Figure 2.3.1.2: SPR and SCR filtering

The three possible actions are:

1. **critical and urgent:** the software manager decides upon the implementation of the change and passes the SCR and associated SPRs to the SRB for confirmation;
2. **critical and routine:** the software manager passes the SCR and associated SPRs to the SRB for review and decision;
3. **non-critical:** the software manager decides upon the implementation of the change and passes the SCR and associated SPRs to the SRB for information.

Software Review Board meetings should use the technical review process described in ESA PSS-05-10 Guide to Software Verification and Validation [Ref 5], with the following modifications:

- the objective of an SRB review is to decide what software changes will be implemented;
- the inputs to the SRB review are the SCRs, SPRs and attachments such as part of a document, a source code listing, a program traceback or a log file;
- the preparation activity is to examine the SPRs and SCRs, not RIDs;
- SRB review meetings are concerned with the SPRs and SCRs, not RIDs, and follow the typical agenda described below.

A typical SRB review meeting agenda consists of:

1. introduction;
2. review actions from the previous meeting;
3. review of the SPR and SCR criticality and urgency classification;
4. review of the critical SPRs;
5. review of the critical SCRs;
6. decision on the SPRs and SCRs;
7. conclusion.

Actions from the previous meeting may have included software change requests. The SRB should close SPRs that are associated with satisfied change requests. Every closed SPR must have one or more associated Software Modification Reports (SMRs) that describe what has been done to solve the problem.

The criticality and urgency of SPRs and SCRs should be reviewed. Members may request that they be reclassified.

Critical SPRs that have not been closed are then discussed. The discussion should confine itself to describing the seriousness and extent of the problem. The Software Review Board decisions should be one of 'update', 'action or 'reject' (See Figure 2.2). The status should become 'update' when the SRB is satisfied that the problem exists and requires a change to the software. The status should become 'action' when there is no satisfactory diagnosis. The status should become 'reject' if the SRB decides that the problem does not exist or that no update or action is necessary.

Critical SCRs are then discussed. The discussion should confine itself to reviewing the effects of the requested changes and the risks involved. Detailed design issues should be avoided. The SRB may discuss the recommendations of the software manager as to:

- who should be the responsible staff;
- how much effort should be allocated;
- when the changes should be implemented and released.

Any part of the SCR may be modified by the SRB. After discussion, the SRB should decide upon whether to approve the change request.

2.3.1.3 Modify Software

When the software change request has been approved, the maintenance team implement the change. The remaining tasks are to:

- modify the documents and code;
- review the modified documents and code;
- test the modified code.

The outputs of these tasks are a Software Modification Report (SMR) and modified configuration items. The SMR defines the:

- names of configuration items that have been modified;
- version or release numbers of the modified configuration items;
- changes that have been implemented;
- actual start date, end date and manpower effort.

Attachments to the SMR should include unit, integration and system test results, as appropriate.

Software systems can be ruined by poor maintenance, and people responsible for it should:

- evaluate the effects of every change;
- verify all software modifications thoroughly;
- keep documentation up to date.

2.3.1.3.1 Evaluating the effects of a change

Software engineers should evaluate the effect of a modification on:

a. performance;
b. resource consumption.
c. cohesion;
d. coupling;
e. complexity;
f. consistency;
g. portability;
h. reliability;
i. maintainability;

j. safety;

k. security.

The effect of changes may be evaluated with the aid of reverse engineering tools and librarian tools. Reverse engineering tools can identify the modules affected by a change at program design level (e.g. identify each module that uses a particular global variable). Librarian tools with cross reference facilities can track dependencies at the source file level (e.g. identify every file that includes a specific file).

There is often more than one way of changing the software to solve a problem, and software engineers should examine the options, compare their effects on the software, and select the best solution. The following sections provide guidance on the evaluation of the software in terms of the attributes listed above.

a. Performance

Performance requirements specify the capacity and speed of the operations the software has to perform. Design specifications may specify how fast a component has to execute.

The effects of a change on software performance should be predicted and later measured in tests. Performance analysis tools may assist measurement. Prototyping may be useful.

b. Resource Consumption

Resource requirements specify the maximum amount of computer resources the software can use. Design specifications may specify the maximum resources available.

The effects of a change on resource consumption should be predicted when it is designed and later measured in tests. Performance analysis tools and system monitoring tools should be used to measure resource consumption. Again, prototyping may be useful.

c. Cohesion

Cohesion measures the degree to which the activities within a component relate to one another. The cohesion of a software component should not be reduced by a change. Cohesion effects should be evaluated when changes are designed.

ESA PSS-05-04 Guide to the Software Architectural Design Phase identifies seven types of cohesion ranging from functional (good) to coincidental (bad) [Ref 2]. This scale can be used to evaluate the effect of a change on the cohesion of a software component.

d. Coupling

Coupling measures the interdependence of two or more components. Changes that increase coupling should be avoided, as they reduce information hiding. Coupling effects should be evaluated when changes are designed.

ESA PSS-05-04 Guide to the Software Architectural Design Phase identifies five types of coupling ranging from 'data coupling' (good) to 'content coupling' (bad) [Ref 2]. This scale can be used to evaluate the effect of a change on the coupling of a software component.

e. Complexity

During the operations and maintenance phase the complexity of software naturally tends to grow because its control structures have to be extended to meet new requirements [Ref 12]. Software engineers should contain this natural growth of complexity because more complex software requires more testing, and more testing requires more effort. Eventually changes become infeasible because they require more effort than is available to implement them. Reduced complexity means greater reliability and maintainability.

Software complexity can be measured by several metrics, the best known metric being cyclomatic complexity [Ref 8]. Procedures for measuring cyclomatic complexity and other important metrics are contained in ESA PSS-05-04 Guide to the Software Architectural Design Phase and ESA PSS-05-10, Guide to Software Verification and Validation [Ref 2, 5]. McCabe has proposed an 'essential complexity metric' for measuring the distortion of the control structure of a module caused by a software change [Ref 13].

f. Consistency

Coding standards define the style in which a programming language should be used. They may enforce or ban the use of language features and define rules for layout and presentation. Changes to software should conform to the coding standards and should be seamless

Polymorphism means that a function will perform the same operation on a variety of data types. Programmers extending the range of data types can easily create inconsistent variations of the same function. This can cause a polymorphic function to behave in unexpected ways. Programmers modifying a polymorphic function should fully understand what the variations of the function do.

g. Portability

Portability is measured by the ease of moving software from one environment to another. Portability can be achieved by adhering to language and coding standards. A common way to make software portable is to encapsulate platform-specific code in 'interface modules'. Only the interface modules need to be modified when the software is ported.

Software engineers should evaluate the effect of a modification on the portability of the software when it is designed. Changes that reduce portability should be avoided.

h. Reliability

Reliability is most commonly measured by the Mean Time Between Failures (MTBF). Changes that reduce reliability should be avoided.

Changes that introduce defects into the software make it less reliable. The software verification process aims to detect and remove defects (see Section 2.3.1.). Walkthroughs and inspections should be used to verify that defects are not introduced when the change is designed. Tests should be used to verify that no defects have been introduced by the change after it has been implemented.

The effect of a modification on software reliability can be estimated indirectly by measuring its effect on the complexity of the software. This effect can be measured when the change is designed.

Reliability can be reduced by reusing components that have not been developed to the same standards as the rest of the software. Software engineers should find out how reliable a software component is before reusing it.

i. Maintainability

Maintainability measures the ease with which software can be maintained. The most common maintainability metric is 'Mean Time To Repair'. Changes that make software less maintainable should be avoided.

Examples of changes that make software less maintainable are those that:

- violate coding standards;
- reduce cohesion;
- increase coupling;
- increase essential complexity [Ref 13].

The costs and benefits of changes that make software more maintainable should be evaluated before such changes are made. All modified code must be tested, and the cost of retesting the new code may outweigh the reduced effort required to make future changes.

j. Safety

Changes to the software should not endanger people or property during operations or following a failure. The effect on the safety of the software should first be evaluated when the change is designed and later during the verification process. The behaviour of the software after a failure should be analysed from the safety point of view.

k. Security

Changes to the software should not expose the system to threats to its confidentiality, integrity and availability. The effect of a change on the security of the software should first be evaluated when the change is designed and later during the verification process.

2.3.1.3.2 Keeping documentation up to date

Consistency between code and documentation must be maintained (OM06). This is achieved by:

- thorough analysis of the impact of every change before it is made, to ensure that no inconsistencies are introduced;
- concurrent update of code and documentation;
- verification of the changes by peer review or independent review.

Tools for deriving detailed design information from source code are invaluable for keeping documentation up to date. The detailed design component specification is placed in the module header. When making a change, programmers:

- modify the detailed design specification in the header;
- modify the module code;
- compile the module;
- review the module modifications;
- unit test the module;
- run a tool to derive the corresponding detailed design document section.

Programmers should use the traceability matrices to check for consistency between the code, DDD, ADD, SRD and URD. Traceability matrices are important navigational aids and should be kept up to date. Tools that support traceability are very useful in the maintenance phase.

2.3.1.4 Verify software modifications

Software modifications should be verified by:

a. review of the detailed design and code;

b. testing.

a. Detailed design and code reviews

The detailed design and code of all changes should be reviewed before they are tested. These reviews should be distinguished from earlier reviews done by the software manager or Software Review Board (SRB) that examine and approve the change request. The detailed design and code are normally not available when the change request is approved.

The detailed design and code of all changes should be examined by someone other than the software engineer who implemented them. The walkthrough or inspection process should be used [Ref 5].

b. Tests

Modified software must be retested before release (SCM39). This is normally done by:

- unit, integration and system testing each change;

- running regression tests at unit, integration and system level to verify that there are no side-effects.

Changes that do not alter the control flow should be tested by rerunning the white box tests that execute the part of the software that changed. Changes that do alter the control flow should be tested by designing white box tests to execute every new branch in the control flow that has been created. Black box tests should be designed to verify any new functionality.

Ideally system tests should be run after each change. This may be a very costly exercise for a large system, and an alternative less costly approach often adopted for large systems is to accumulate changes and then run the system tests (including regression tests) just before release. The disadvantage of this approach is that it may be difficult to identify which change, if any, is the cause of a problem in the system tests.

2.3.2 Release Software

Changed configuration items are made available to users through the software release process. This consists of:

- defining the release;
- documenting the release;
- auditing the release;
- delivering the release.

2.3.2.1 Define release

Software managers should define the content and timing of a software release according to the needs of users. This means that:

- solutions to urgent problems are released as soon as possible only to the people experiencing the problem, or who are likely to experience the problem;
- other changes are released when the users are ready to accommodate them.

Software managers, in consultation with the Software Review Board, should allocate changes to one of three types of release:

- major release;
- minor release;

- emergency release (also called a þatch).

Table 2.3.2.1 shows how they differ according to whether:

- adaptive changes have been made;
- perfective changes have been made;
- corrective changes have been made;
- all or selected software configuration items are included in the release;
- all or selected users will receive the release.

	Adaptive Changes	**Perfective Changes**	**Corrective Changes**	**CIs**	**Users**
Major Release	Yes	Yes	Yes	All	All
Minor Release	Small	Yes	Yes	All	All
Emergency Release	No	No	Yes	Selected	Selected

Table 2.3.2.1: Major, minor and emergency releases

The purpose of a major release of a software system is to provide new capabilities. These require adaptive changes. Major releases also correct outstanding faults and perfect the existing software. Operations may have to be interrupted for some time after a major release has been installed because training is required. Major releases should therefore not be made too frequently because of the disruption they can cause. A typical time interval between major releases is one year. Projects using evolutionary and incremental delivery life cycle approaches would normally make a major release in each transfer phase.

The purpose of a minor release is to provide corrections to a group of problems. Some low-risk perfective maintenance changes may be included. A minor release of a software system may also provide small extensions to existing capabilities. Such changes can be easily assimilated by users without training.

The frequency of minor releases depends upon:

- the rate at which software problems are reported;
- the urgency of solving the software problems.

The purpose of an emergency release is to get a modification to the users who need it as fast as possible. Only the configuration items directly affected by the fault are released. Changes are nearly always corrective.

2.3.2.2 Document release

Every software release must be accompanied by a Software Release Note (SRN) (SCM14). Software Release Notes should describe the:

- software item title/name;
- software item version/release number;
- changes in the release;
- list of configuration items included in the release;
- installation instructions.

Forms are used for simple releases and documents for complex releases.

2.3.2.2.1 Release number

The SRN should define the version number of the release. The structure of the version number should reflect the number of different types of releases used. A common structure used is two or three integers separated by a full stop:

major release number.minor release number[.emergency release number]

The square brackets indicate that the emergency release number is optional because it is only included when it is not zero. When any release number is incremented the succeeding numbers are set to zero.

2.3.2.2.2 Changes in the release

This section of the SRN should list all the changes in the release. For each change:

- give a paragraph summarising the effects that the users will see resulting from the change;
- enumerate the related SPRs and SCRs (SCM36).

2.3.2.2.3 List of configuration items included in the release

This section of the SRN should list the configuration identifiers of all the configuration items in the release.

2.3.2.2.4 Installation instructions

This section of the SRN should describe how to install the release. This is normally done by referencing the installation instructions in the Software User Manual[1], or providing updated instructions, or both.

2.3.2.3 Audit release

Functional and physical audits shall be performed before the release of the software (SVV03). The purpose of the audits is to verify that all the necessary software configuration items are present, consistent and correct.

A functional audit verifies that the development of a configuration item has been completed satisfactorily, that the item has achieved the performance and functional characteristics specified in the software requirements and design documents [Ref 7]. This is normally done by checking the test reports for the software release. A functional audit also verifies that the operational and support documents are complete and satisfactory.

A physical audit verifies that an as-built configuration conforms to its documentation. This is done by checking that:

- all the configuration items listed in the SRN are actually present;
- documentation and code in a software release are consistent (SCM37).

---

[1] In ESA PSS-05-0 Installation instructions are placed in the STD. The SUM is normally a more suitable place to put the installation instructions.

2.3.2.4 Deliver release

The software can be delivered when the audits have been done. The maintenance team is responsible for copying the software onto the release media, packaging the media with the software documentation and delivering the package.

The maintenance team must archive every release (SCM38). This is best done by keeping a copy of the delivered package. Although users may wish to retain old releases for reference, the responsibility for retaining a copy of every release lies with the maintenance team.

2.3.3 Install Release

Upon delivery, the contents of the release are checked against the configuration item list in the Software Release Note (SRN) and then the software is installed. The installation procedures are also described or identified in the SRN.

Installation should be supported by tools that

- automate the installation process as much as possible;
- issue simple and clear instructions;
- reuse configuration information from the existing system and not require users to reenter it;
- make minimal changes to the system configuration (e.g. modifying the CONFIG.SYS and AUTOEXEC.BAT files of a PC);
- always get permission before making any changes to the system configuration;

The installation process should be reversible, so that rollback of the system to the state before the installation process was started is always possible. One way to do this is to make a backup copy of the existing system before starting the installation.

The installation software should remove obsolete configuration items from the directories where the new version of the system is installed.

2.3.4 Validate release

After installation, users should run some or all of the acceptance tests to validate the software. The acceptance test specification should have

been updated to include tests of any new user requirements that have been implemented.

2.4 UPDATE PROJECT HISTORY DOCUMENT

The purpose of the Project History Document (PHD) is to provide a summary critical account of the project. The PHD should:

- describe the objectives of the project;
- summarise how the project was managed;
- state the cost of the project and compare it with predictions;
- discuss how the standards were applied;
- describe the performance of the system in OM phase;
- describe any lessons learned.

The benefits of the PHD are:

- the maintenance team is informed about what the development team did, so they can avoid repeating mistakes or trying out inappropriate solutions;
- the maintenance team are told how well the system performs, as they may have to make good any shortfall;
- managers of future projects will know how much a similar project is likely to cost, and problems and pitfalls they are likely to experience.

The software manager should write the PHD. Information should be collected throughout the project. Work on the PHD should start early in the project, with updates at every major milestone. Detailed guidance on writing the PHD is provided in chapter 4.

2.5 FINAL ACCEPTANCE

A review of the software should be held at the end of the warranty period in order to decide whether the software is ready for final acceptance. All the acceptance tests must have been completed before the software can be finally accepted (OM02).

The review team should consist of the Software Review Board (SRB) members. The responsibilities and constitution of the Software Review

Board are defined in ESA PSS-05-09 'Guide to Software Configuration Management' Section 2.2.1.3 [Ref 4].

The software review should be a formal technical review. The procedures for a technical review are described in Section 2.3.1 of ESA PSS-05-10, 'Guide to Software Verification and Validation' [Ref 5].

The final acceptance review meeting may coincide with an ordinary SRB meeting. The decision upon final acceptance is added to the agenda described in Section 2.3.1.2.

Inputs to the review are the:

- Project History Document (PHD);
- results of acceptance tests held over to the OM phase;
- Software Problem Reports made during the OM phase;
- Software Change Requests made during the OM phase;
- Software Modification Reports made during the OM phase.

The Software Review Board reviews these inputs and recommends, to the initiator, whether the software can be finally accepted.

The SRB should evaluate the degree of compliance of the software with the user requirements by considering the number and nature of:

- acceptance test cases failed;
- acceptance tests cases not attempted or completed;
- critical software problems reported;
- critical software problems not solved.

Solution of a problem means that a change request has been approved, modification has been made, and all tests repeated and passed.

The number of test cases, critical software problems and non-critical software problems should be evaluated for the whole system and for each subsystem. The SRB might, for example, decide that some subsystems are acceptable and others are not.

The SRB should study the trends in the number of critical software problems reported and solved. Although there are likely to be 'spikes' in the trend charts associated with software releases, there should be a downward

trend in the number of critical problems during the OM phase. An upward trend should be cause for concern.

Sufficient data should have accumulated to evaluate the Mean Time Between Failures (MTBF) and Mean Time To Repair (MTTR). The MTBF may be estimated by dividing the total number of critical software problems raised during the phase by the total time spent operating the software during the phase. The MTTR may be estimated by averaging the difference between the start and end dates of the modifications completed.

If the SRB decides that the degree of compliance of the software with the user requirements is acceptable, it should recommend to the initiator that the software be finally accepted.

The statement of final acceptance is produced by the initiator, on behalf of the users, and sent to the developer (OM09). The finally accepted software system consists of one or more sets of documentation, source, object and executable code corresponding to the current versions and releases of the product.

CHAPTER 3
TOOLS FOR SOFTWARE MAINTENANCE

3.1 INTRODUCTION

Tools used during software development will continue to be used during the operations and maintenance phase. New versions of the tools may become available, or a tool may become unsupported and require replacement. Tools may be acquired to support new activities, or to support activities that previously took place without them.

The reader should refer to the appropriate guides for the tools that support:

- user requirements definition;
- software requirements definition;
- architectural design;
- detailed design and production;
- transfer;
- software project management;
- software configuration management;
- software verification and validation;
- software quality assurance.

This chapter is concerned with the tools that are normally used for the first time in the life cycle during the operations and maintenance phase. These tools are:

- navigation tools;
- code improvement tools;
- reverse engineering tools.

3.2 NAVIGATION TOOLS

Navigation tools enable software engineers to find quickly and easily the parts of the software that they are interested in. Typical capabilities are:

- identification of where variables are used;

- identification of the modules that use a module;
- display of the call tree;
- display of data structures.

Knowledge of where variables and modules are used is critical to understanding the effect of a change. Display of the call tree and data structures supports understanding of the control and data flow.

In addition, navigation tools for object-oriented software need to be able to:

- distinguish which meaning of an overloaded symbol is implied;
- support the location of inherited attributes and functions.

The maintenance of object-oriented programs is an active research area [Ref 14, 15]. In particular, the behaviour of a polymorphic function can only be known at runtime when argument types are known. This makes it difficult to understand what the code will do by means of static analysis and inspection. Dynamic analysis of the running program may be the only way to understand what it is doing.

3.3 CODE IMPROVEMENT TOOLS

Code improvement tools may:

- reformat source code;
- restructure source code.

Code reformatters, also known as 'pretty printers', read source code and generate output with improved layout and presentation. They can be very useful for converting old code to the style of a new coding standard.

Code restructuring tools read source code and make it more structured, reducing the control flow constructs as far as possible to only sequence, selection and iteration.

3.4 REVERSE ENGINEERING TOOLS

Reverse engineering tools process code to produce another type of software item. They may for example:

- generate source code from object code;
- recover designs from source code.

Decompilers translate object code back to source code. Some debuggers allow software engineers to view the source code alongside the object code. Decompilation capabilities are sometimes useful for diagnosing compiler faults, e.g. erroneous optimisations.

Tools that recover designs from source code examine module dependencies and represent them in terms of a design method such as Yourdon. Tools are available that can generate structure charts from C code for example. These reverse engineering tools may be very useful when documentation of code is non-existent or out of date.

This page is intentionally left blank.

CHAPTER 4
THE PROJECT HISTORY DOCUMENT

4.1 INTRODUCTION

The Project History Document (PHD) summarises the main events and the outcome of the project. The software manager should collect appropriate information, summarise it, and insert it in the PHD phase-by-phase as the project proceeds. Much of the information will already exist in earlier plans and reports. When final acceptance is near, the software manager should update the document taking into account what has happened since the start of the operations and maintenance phase.

4.2 STYLE

The Project History Document should be plain, concise, clear and consistent.

4.3 EVOLUTION

After delivery, the section of the Project History Document on the performance of the software in the OM phase should be updated by the maintenance team at regular intervals (e.g. annually).

4.4 RESPONSIBILITY

The software project manager is responsible for the production of the Project History Document. The software maintenance manager is responsible for the production of subsequent issues.

4.5 MEDIUM

The Project History Document is normally a paper document.

4.6 CONTENT

ESA PSS-05-0 recommends the following table of contents for the Project History Document.

1 Description of the project

2 Management of the project
- 2.1 Contractual approach
- 2.2 Project organisation
- 2.3 Methods and tools[2]
- 2.4 Planning

3 Software Production
- 3.1 Product size[3]
- 3.2 Documentation
- 3.3 Effort[4]
- 3.4 Computer resources
- 3.5 Productivity[5]

4 Quality Assurance Review

5 Financial Review

6 Conclusions

7 Performance of the system in OM phase

4.6.1 PHD/1 DESCRIPTION OF THE PROJECT

This section should:

- describe the objectives of the project;
- identify the initiator, developer and users;
- identify the primary deliverables;
- state the size of the software and the development effort;
- describe the life cycle approach;
- state the actual dates of all major milestones.

Information that appears in later parts of the document may be summarised in this section.

---

[2] In ESA PSS-05-0 this section is called "Methods used".

[3] In ESA PSS-05-0 this section is called "Estimated vs. actual amount of code produced"

[4] In ESA PSS-05-0 this section is called "Estimated vs. actual effort"

[5] In ESA PSS-05-0 this section is called "Analysis of productivity factors"

Critical decisions, for example changes in objectives, should be clearly identified and explained.

4.6.2 PHD/2 MANAGEMENT OF THE PROJECT

4.6.2.1 PHD/2.1 Contractual approach

This section should reference the contract made (if any) between initiator's organisation and the development organisation.

This section should state the type of contract (e.g. fixed price, time and materials).

4.6.2.2 PHD/2.2 Project organisation

This section should describe the:

- internal organisation of the project;
- external interfaces of the project.

The description of the organisation should define for each role:

- major responsibilities;
- number of staff.

If the organisation and interfaces changed from phase to phase, this section describes the organisation and interfaces in each phase. If the number of staff varied within a phase, the staffing profile should be described.

4.6.2.3 PHD/2.3 Methods and tools

This section should identify the methods used in the project, phase by phase. The methods should be referenced. This section should not describe the rules and procedures of the methods, but critically discuss them from the point of view of the project. Aspects to consider are:

- training requirements.
- applicability.

Any tools used to support the methods should be identified and the quality of the tools discussed, in particular:

- degree of support for the method;
- training requirements;

- reliability;
- ease of integration with other tools;
- whether the benefits of the tools outweighed the costs.

4.6.2.4 PHD/2.4 Planning

This section should summarise the project plan by producing for each phase the:

- initial work breakdown structure (but not the work package descriptions);
- list of work packages added or deleted;
- Gantt chart showing the predicted start and end dates of each activity;
- Gantt chart showing the actual start and end dates of each activity;
- Milestone trend charts showing the movement (if any) of major milestones during the phase.

4.6.3 PHD/3 SOFTWARE PRODUCTION

4.6.3.1 PHD/3.1 Product size

This section should state the number of user requirements and software requirements.

This section should state the number of subsystems, tasks or programs in the architectural design.

This section should state the amount of code, both for the whole system and for each subsystem:

- predicted at the end of the AD phase;
- produced by the end of the TR phase;
- produced by final acceptance.

The actual amount of code produced should be specified in terms of the number of:

- lines of code;
- modules.

This section should make clear the rules used to define a line of code. Comment lines are not usually counted as lines of code. Historically,

continuation lines have been counted as separate lines of code, but it may be more meaningful to ignore them and count each complete statement as one line of code. Non-executable statements (e.g. data declarations) are also normally counted as lines of code.

4.6.3.2 PHD/3.2 Documentation

This section should identify each document produced and state for each document the number of pages and words.

These values should be summed to define the total amount of documentation produced.

4.6.3.3 PHD/3.3 Effort

This section should state the estimated and actual effort required for each work package. The unit should be man-hours, man-days or man-months. Significant differences between the estimated and actual effort should be explained.

Values should be summed to give the total amount of effort required for all activities in:

- each of the SR, AD, DD and TR phases;
- the whole development (i.e. sum of the SR, AD, DD and TR phases);
- OM phase.

4.6.3.4 PHD/3.4 Computer resources

This section should state the estimated and actual hardware, operating software and ancillary software required to develop and operate the software. Significant differences between actual and estimated resources should be explained.

4.6.3.5 PHD/3.5 Productivity

This section should state the actual productivity in terms of:

- total number of lines of code produced divided by the total number of man-days in the SR, AD, DD and TR phases;
- total number of lines of code in each subsystem divided by the total number of man-days expended on that subsystem in the DD phase.

The first value gives the global 'productivity' value. The second set of values gives the productivity of each subsystem.

Productivity estimates used should also be stated. Significant differences between the estimated and actual productivity values should be explained.

4.6.4 PHD/4 QUALITY ASSURANCE REVIEW

This section should review the actions taken to achieve quality, particularly reliability, availability, maintainability and safety.

This section should summarise and discuss the effort expended upon activities of:

- software verification and validation;
- software quality assurance.

This section should analyse the quality of all the deliverables by presenting and discussing the number of:

- RIDs per document;
- SPRs and SCRs per month during the DD, TR and OM phases;
- SPRs and SCRs per subsystem per month during the DD, TR and OM phases.

Measurements of Mean Time Between Failures (MTBF) and Mean Time To Repair (MTTR) should be made in the OM phase. SPR, SCR, SMR and operations log data may be useful for the calculation. Measurements should be made at regular intervals (e.g. monthly) and trends monitored.

Average availability may be calculated from the formula MTBF/(MTBF+MTTR). In addition, the durations of all periods when the software was unavailable should be plotted in a histogram to give a picture of the number and frequency of periods of unavailability.

All 'safety' incidents where the software caused a hazard to people or property should be reported. Actions taken to prevent future incidents should be described.

4.6.5 PHD/5 FINANCIAL REVIEW

This section is optional. If included, this section should state the estimate and actual cost of the project. Costs should be divided into labour costs and non-labour costs.

4.6.6 PHD/6 CONCLUSIONS

This section should summarise the lessons learned from the project.

4.6.7 PHD/7 PERFORMANCE OF THE SYSTEM IN THE OM PHASE

This section should summarise in both quantitative and qualitative terms whether the software performance fell below, achieved or exceeded the user requirements, the software requirements and the expectations of the designers.

Performance requirements in the software requirements document may be stated in terms of:

- worst case;
- nominal;
- best case value.

These values, if specified, should be used as benchmarks of performance.

The results of any acceptance tests required for final acceptance should be summarised here.

This page is intentionally left blank.

CHAPTER 5
LIFE CYCLE MANAGEMENT ACTIVITIES

5.1 INTRODUCTION

Software maintenance is a major activity, and needs to be well managed to be effective. ESA PSS-05-0 identifies four software management functions:

- software project management;
- software configuration management;
- software verification and validation;
- software quality assurance.

This chapter discusses the planning of these activities in the operations and maintenance phase.

The plans formulated by the developer at the end of detailed design and production phase should be applied, with updates as necessary, by the development organisation throughout the transfer phase and operations and maintenance phase until final acceptance.

ESA PSS-05-0 does not mandate the production of any plans after final acceptance. However the maintenance organisation will need to have plans to be effective, and is strongly recommended to:

- produce its own SPMP and SQAP;
- reuse and improve the SCMP and SVVP of the development organisation.

5.2 SOFTWARE PROJECT MANAGEMENT

Until final acceptance, OM phase activities that involve the developer must be carried out according to the plans defined in the SPMP/TR (OM01). After final acceptance, the software maintenance team takes over responsibility for the software. It should produce its own SPMP. Guidelines for producing an SPMP are contained in ESA PSS-05-08, Guide to Software Project Management [Ref 3].

In the maintenance phase, both plans should define the:

- organisation of the staff responsible for software maintenance;
- work packages;
- resources;
- activity schedule.

5.2.1 Organisation

A maintenance organisation must be designated for every software product in operational use (OM04). A software manager should be appointed for every maintenance organisation. The software manager should define the roles and responsibilities of the maintenance staff in the software project management plan. Individuals should be identified with overall responsibility for:

- each subsystem;
- software configuration management;
- software verification and validation;
- software quality assurance.

Major adaptations of the software for new requirements or environmental changes should be handled by means of the evolutionary life cycle, in which a development project runs in parallel with operations and maintenance. Sometimes an evolutionary approach is decided upon at the start of the project. More commonly the need for such an approach only becomes apparent near the time the software is first released. Whichever way the evolutionary idea arises, software managers should organise their staff accordingly, for example by having separate teams for maintenance and development. Ideally, software engineers should not work on both teams at the same time, although the teams may share the same software manager, software librarian and software quality assurance engineer.

5.2.2 Work packages

Work packages should be defined when software change requests are approved. One work package should produce only one software modification report, but may cover several change requests and problem reports.

5.2.3 Resources

The operations and maintenance phase of software normally consumes more resources than all the other phases added together. Studies have shown that large organisations spend about 50% to 70% of their available effort maintaining existing software [Ref 9, 10]. The high relative cost of maintenance is due to the influence of several factors such as the:

- duration of the operations and maintenance phase being much longer than all the other phases added together;
- occurrence of new requirements that could not have been foreseen when the software was first specified;
- presence of a large number of faults in most delivered software.

Resources must be assigned to the maintenance of product until it is retired (OM07). Software managers must estimate the resources required. The estimates have two components:

- predicted non-labour costs;
- predicted labour costs (i.e. effort).

The cost of the computer equipment and consumables are the major non-labour costs. The cost of travel to user sites and the cost of consumables may have to be considered.

Labour costs can be assessed from:

- the level of effort required from the development organisation to support the software in the transfer phase;
- the number of user requirements outstanding;
- past maintenance projects carried out by the organisation;
- reliability and maintainability data;
- size of the system;
- number of subsystems;
- type of system;
- availability requirements.

Software managers should critically analyse this information. Lehman's laws and a code size method are outlined below to assist the analysis.

5.2.3.1 Lehman's Laws

Lehman and Belady have examined the growth and evolution of several large software systems and formulated five laws to summarise their data [Ref 12]. Three of the laws are directly relevant to estimating software maintenance effort. Simply stated they say:

- the characteristics of a program are fixed from the time when it is first designed and coded (the 'law of large program evolution');
- the rate at which a program develops is approximately constant and (largely) independent of the resources devoted to its development (the 'law of organisational stability');
- the incremental change in each release of a system is approximately constant (the 'law of conservation of familiarity').

The inherent characteristics of a program, introduced when it is first designed and coded, will have a fundamental effect on the amount of maintenance a program needs. Poorly designed software will incur higher maintenance costs. Removing the inherent design defects amounts to rewriting the program from scratch.

The law of diminishing returns applies at a very early stage in software maintenance. No matter how much effort is applied to maintaining software, the need to implement and verify modifications one at a time limits how fast changes can be made. If the mean time to repair the software does not change when the number of maintenance staff is increased, the apparent inefficiency is not the fault of the maintenance staff, but is due to the sequential nature of the work.

Software maintenance projects settle into a cycle of change. Modifications are requested, implemented and released to users at regular intervals. The release rate is constrained by the amount of change users can cope with.

In summary, software managers must be careful not to waste effort by:

- maintaining software that is unmaintainable;
- staffing the maintenance team above a saturation threshold;
- releasing changes too fast for the users to cope with.

5.2.3.2 Code size maintenance cost estimating method

The effort required to produce L lines of code when the productivity is P is L divided by P. For example, the addition or modification of 500 lines of code in a 20000 line of code system that took 1000 man-days to develop would require 500/(20000/1000) = 25 man days. This example assumes that the productivity observed in maintenance is the same as that in development. However productivity in maintenance often falls below that in development because of the need to assess the cause of a problem and perform regression tests.

5.2.4 Activity schedule

The activity schedule should show the dates of the next releases, and the work packages that must be completed for each release. This should be presented in the form of a Gantt chart.

5.3 SOFTWARE CONFIGURATION MANAGEMENT

A good configuration management system is essential for effective software maintenance, not only for the software but also for the tools and ancillary software items. Software configuration management is discussed in ESA PSS-05-09 Guide to Software Configuration Management [Ref 4].

The maintenance team should define and describe its software configuration management system in a Software Configuration Management Plan (SCMP). The team may reuse the plan and system made by the development team, or produce its own. The SCMP must define the procedures for software modification (OM05). The procedures should be based upon the change process discussed in Section 2.3.1 The SCMP should define, step-by-step, how to modify the software, omitting only the details specific to individual changes.

5.4 SOFTWARE VERIFICATION AND VALIDATION

Software verification and validation consumes a significant proportion of effort during the operations and maintenance phase because of the need to check that changes not only work correctly, but also that they have no adverse side effects. Software verification and validation are discussed in ESA PSS-05-10 Guide to Software Verification and Validation [Ref 5].

The maintenance team should reuse the Software Verification and Validation Plan (SVVP) produced during the development phases. It should be updated and extended as necessary.

The Software Review Board (SRB) review procedure should have been defined in the SVVP/DD. This procedure may need to be updated or even added to the SVVP either in the TR or OM phase. The SRB should use the technical review procedure described in ESA PSS-05-10 Guide to Software Verification and Validation, modified as indicated in Section 2.3.1.2.

Walkthroughs and inspections of the design and code are not only useful for training new staff, they are essential for maintaining and improving quality. Even if software inspection procedures were not used during development, consideration should be given to introducing them in the maintenance phase, especially for subsystems that have serious quality problems. All new procedures should be defined in updates to the SVVP.

Every change to the software should be examined from the point of view of:

- what should be done to verify it;
- whether the SVVP defines the verification procedure.

The SVVP will have to be extended to include new test designs, test cases and test procedures for:

- tests of new or modified software components;
- regression tests of the software.

Regression tests may be:

- random selections of existing tests;
- targeted selections of existing tests;
- new tests designed to expose adverse side effects.

Targeted tests and new tests depend upon knowledge of the actual change made. All such regression tests imply a new test design. The first two reuse test cases and test procedures.

Some of the acceptance tests may require a long period of time to complete, and are therefore not required for provisional acceptance. The reports of these tests should be compiled in the operations and maintenance phase. All the tests must be completed before final acceptance (OM02).

5.5 SOFTWARE QUALITY ASSURANCE

Software quality assurance activities must be continued during the operations and maintenance phase. ESA PSS-05-0 specifies ten mandatory practices for OM phase. ESA PSS-05-11 Guide to Software Quality Assurance contains guidelines on how to check that these practices are carried out [Ref 6]. In particular, staff responsible for SQA should:

- be members of the Software Review Board;
- perform functional and physical audits before each release of the software;
- monitor quality, reliability, availability, maintainability and safety.

The SQAP/TR should define the SQA activities that the development team should carry out until final acceptance of the software. The maintenance organisation that takes over after final acceptance should examine that plan, consider the current status of the software and then produce its own plan.

This page is intentionally left blank.

APPENDIX A
GLOSSARY

A.1 LIST OF TERMS

Terms used in this document are consistent with ESA PSS-05-0 [Ref 1] and ANSI/IEEE Std 610.12 [Ref 2]. Additional terms not defined in these standards are listed below.

building

The process of compiling and linking a software system.

development environment

The environment of computer hardware, operating software and external software in which the software system is developed.

end user

A person who utilises the products or services of a system.

installation

The process of copying a software system into the target environment and configuring the target environment to make the software system usable.

operator

A person who controls and monitors the hardware and software of a system.

operational environment

The target environment, external software and users.

target environment

The environment of computer hardware, operating software and external software in which the software system is used.

user

A person who utilises the products or services of a system, or a person who controls and monitors the hardware and software of a system (i.e. an end user, an operator, or both).

A.2 LIST OF ACRONYMS

AD	Architectural Design
ANSI	American National Standards Institute
AT	Acceptance Test
BSSC	Board for Software Standardisation and Control
CASE	Computer Aided Software Engineering
CI	Configuration Item
DD	Detailed Design and production
OM	Operations and Maintenance
PHD	Project History Document
SCM	Software Configuration Management
SCMP	Software Configuration Management Plan
SCR	Software Change Request
SMR	Software Modification Report
SPM	Software Project Management
SPMP	Software Project Management Plan
SPR	Software Problem Report
SQA	Software Quality Assurance
SQAP	Software Quality Assurance Plan
SR	Software Requirements definition
SRD	Software Requirements Document
SRN	Software Release Note
STD	Software Transfer Document
SVV	Software Verification and Validation
SVVP	Software Verification and Validation Plan
TR	Transfer
UR	User Requirements definition
URD	User Requirements Document

APPENDIX B
REFERENCES

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

2. Guide to the Software Architectural Design Phase, ESA PSS-05-04, January 1992.

3. Guide to Software Project Management, ESA PSS-05-08, Issue 1 Draft, July 1994

4. Guide to Software Configuration Management, ESA PSS-05-09, Issue 1, November 1992

5. Guide to Software Verification and Validation, ESA PSS-05-10, Issue 1, February 1994

6. Guide to Software Quality Assurance, ESA PSS-05-11, Issue 1, July 1993.

7. IEEE Standard Glossary of Software Engineering Terminology, ANSI/IEEE Std 610.12-1990

8. Standard Dictionary of Measures to Produce Reliable Software, ANSI/IEEE Std 982.1-1988

9. Software Engineering, I. Sommerville, Addison Wesley, Fourth Edition, 1992

10. Software Maintenance Management, B.P. Lientz and E.B. Swanson, Addison Wesley, 1980

11. Software Evolution, R.J. Arthur, Wiley, 1988

12. Program Evolution. Processes of Software Change, M.M. Lehman and L. Belady, Academic Press, 1985.

13. Structured Testing: A Software Testing Methodology Using the Cyclomatic Complexity Metric, T.J. McCabe, National Bureau of Standards Special Publications 500-99, 1982.

14. Maintenance support for object-oriented programs, N.Wilde and R. Huitt, IEEE Transactions on Software Engineering, Vol 18 Number 12, IEEE Computer Society, December 1992

15. Support for maintaining object-oriented programs, M. Lejter, S. Meyers and S.P. Reiss, IEEE Transactions on Software Engineering, Vol 18 Number 12, IEEE Computer Society, December 1992

APPENDIX C
MANDATORY PRACTICES

This appendix is repeated from ESA PSS-05-0 appendix D.7

OM01 Until final acceptance, OM phase activities that involve the developer shall be carried out according to the plans defined in the SPMP/TR.

OM02 All the acceptance tests shall have been successfully completed before the software is finally accepted.

OM03 Even when no contractor is involved, there shall be a final acceptance milestone to arrange the formal hand-over from software development to maintenance.

OM04 A maintenance organisation shall be designated for every software product in operational use.

OM05 Procedures for software modification shall be defined.

OM06 Consistency between code and documentation shall be maintained.

OM07 Resources shall be assigned to a product's maintenance until it is retired.

OM08 The SRB ... shall authorise all modifications to the software.

OM09 The statement of final acceptance shall be produced by the initiator, on behalf of the users, and sent to the developer.

OM10 The PHD shall be delivered to the initiator after final acceptance.

This page is intentionally left blank

APPENDIX D
INDEX