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

Guide to the software detailed design and production phase

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

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-05-05 Guide to the Software Detailed Design and Production Phase**			
2. ISSUE	3. REVISION	4. DATE	5. REASON FOR CHANGE
1	0	1992	First issue
1	1	1995	Minor revisions 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, 10th June 1992
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 detailed design and production phase described in the top level document ESA PSS-05-0.

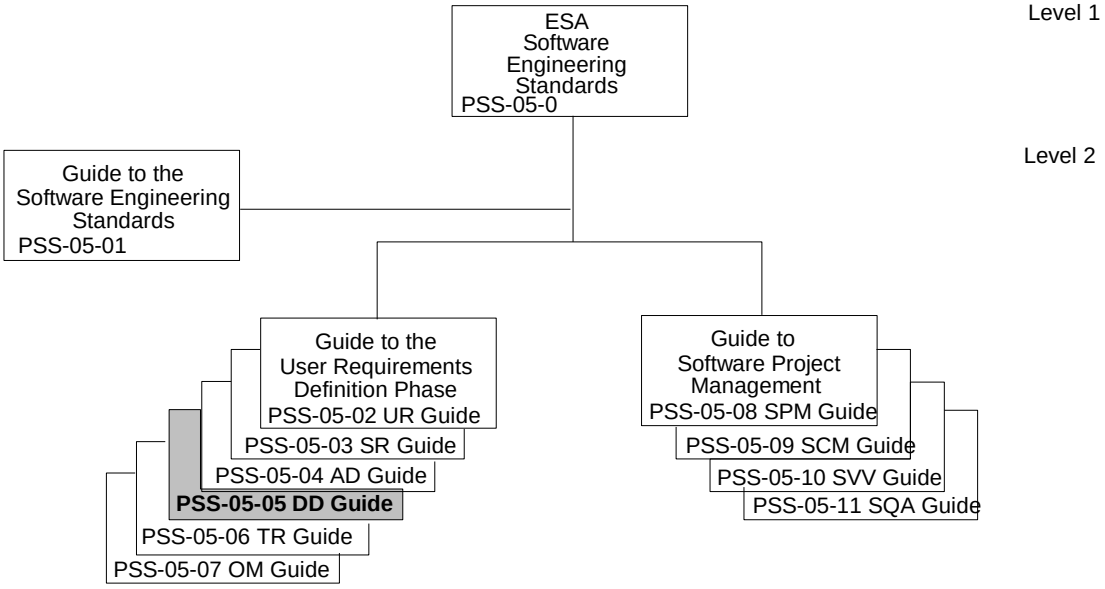

Figure 1: ESA PSS-05-0 document tree

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), Bryan Melton, Daniel de Pablo, Adriaan Scheffer and Richard Stevens.

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 C Mazza
ESOC
Robert Bosch Strasse 5
D-64293 Darmstadt
Germany

BSSC/ESTEC Secretariat
Attention of Mr B Melton
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 second phase of the software development life cycle as the 'Architectural Design Phase' (AD phase). The output of this phase is the Architectural Design Document (ADD). The third phase of the life cycle is the 'Detailed Design and Production Phase' (DD phase). Activities and products are examined in the 'DD review' (DD/R) at the end of the phase.

The DD phase can be called the 'implementation phase' of the life cycle because the developers code, document and test the software after detailing the design specified in the ADD.

This document provides guidance on how to produce the Detailed Design Document (DDD), the code and the Software User Manual (SUM). This document should be read by all active participants in the DD phase, e.g. designers, programmers, project managers and product assurance personnel.

1.2 OVERVIEW

Chapter 2 discusses the DD phase. Chapters 3 and 4 discuss methods and tools for detailed design and production. Chapter 5 and 6 describe how to write the DDD and SUM, starting from the templates. Chapter 7 summarises the life cycle management activities, which are discussed at greater length in other guides.

All the mandatory practices in ESA PSS-05-0 relevant to the DD phase are repeated in this document. The identifier of the practice is added in parentheses to mark a repetition. No new mandatory practices are defined.

This page is intentionally left blank.

CHAPTER 2
THE DETAILED DESIGN AND PRODUCTION PHASE

2.1 INTRODUCTION

Software design is the 'process of defining the architecture, components, interfaces, and other characteristics of a system or component ' [Ref 2]. Detailed design is the process of defining the lower-level components, modules and interfaces. Production is the process of:

- programming - coding the components;
- integrating - assembling the components;
- verifying - testing modules, subsystems and the full system.

The physical model outlined in the AD phase is extended to produce a structured set of component specifications that are consistent, coherent and complete. Each specification defines the functions, inputs, outputs and internal processing of the component.

The software components are documented in the Detailed Design Document (DDD). The DDD is a comprehensive specification of the code. It is the primary reference for maintenance staff in the Transfer phase (TR phase) and the Operations and Maintenance phase (OM phase).

The main outputs of the DD phase are the:

- source and object code;
- Detailed Design Document (DDD);
- Software User Manual (SUM);
- Software Project Management Plan for the TR phase (SPMP/TR);
- Software Configuration Management Plan for the TR phase (SCMP/TR);
- Software Quality Assurance Plan for the TR phase (SQAP/TR);
- Acceptance Test specification (SVVP/AT).

Progress reports, configuration status accounts, and audit reports are also outputs of the phase. These should always be archived.

The detailed design and production of the code is the responsibility of the developer. Engineers developing systems with which the software

interfaces may be consulted during this phase. User representatives and operations personnel may observe system tests.

DD phase activities must be carried out according to the plans defined in the AD phase (DD01). Progress against plans should be continuously monitored by project management and documented at regular intervals in progress reports.

Figure 2.1 is an ideal representation of the flow of software products in the DD phase. The reader should be aware that some DD phase activities can occur in parallel as separate teams build the major components and integrate them. Teams may progress at different rates; some may be engaged in coding and testing while others are designing. The following subsections discuss the activities shown in Figure 2.1.

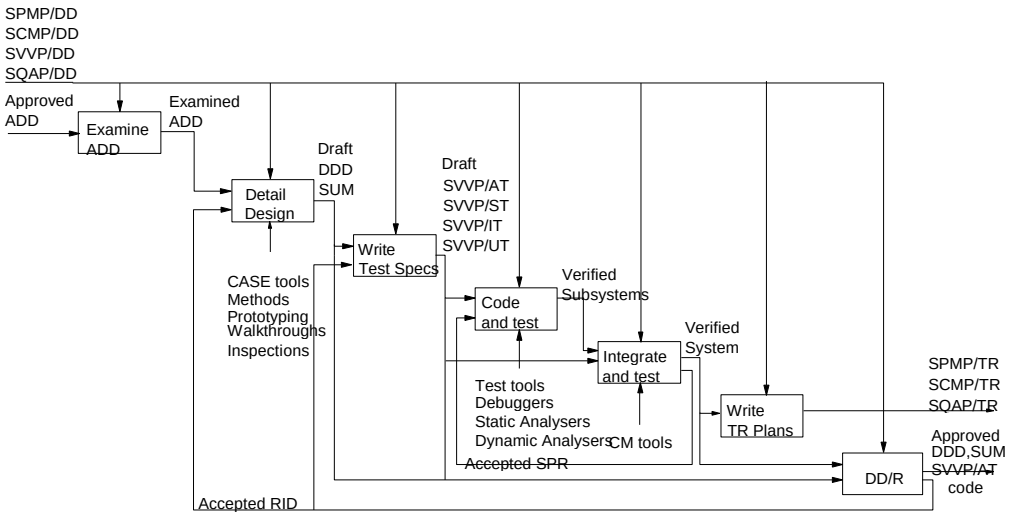

Figure 2.1: DD phase activities

2.2 EXAMINATION OF THE ADD

If the developers have not taken part in the AD/R they should examine the ADD and confirm that it is understandable. Developers should consult ESA PSS-05-04, 'Guide to the Software Architectural Design Phase' for help on ADDs. The examination should be carried out by staff familiar with the architectural design method. The developers should also confirm that adequate technical skill is available to produce the outputs of the DD phase.

2.3 DETAILED DESIGN

Design standards must be set at the start of the DD phase by project management to coordinate the collective efforts of the team. This is especially necessary when development team members are working in parallel.

The developers must first complete the top-down decomposition of the software started in the AD phase (DD02) and then outline the processing to be carried out by each component. Developers must continue the structured approach and not introduce unnecessary complexity. They must build defences against likely problems.

Developers should verify detailed designs in design reviews, level by level. Review of the design by walkthrough or inspection before coding is a more efficient way of eliminating design errors than testing.

The developer should start the production of the user documentation early in the DD phase. This is especially important when the HCI component is significantly large: writing the SUM forces the developer to keep the user's view continuously in mind.

The following subsections discuss these activities in more detail.

2.3.1 Definition of design standards

Wherever possible, standards and conventions used in the AD phase should be carried over into the DD phase. They should be documented in part one of the DDD. Standards and conventions should be defined for:

- design methods;
- documentation;
- naming components;
- Computer Aided Software Engineering (CASE) tools;
- error handling.

2.3.2 Decomposition of the software into modules

As in architectural design, the first stage of detailed design is to define the functions, inputs and outputs of each software component. Whereas architectural design only considered the major, top-level

components, this stage of detailed design must specify all the software components.

The developer starts with the major components defined in the ADD and continues to decompose them until the components can be expressed as modules in the selected programming languages.

Decomposition should be carried out using the same methods and tools employed in the AD phase. CASE tools and graphical methods employed in architectural design can still be used.

Component processing is only specified to the level of detail necessary to answer the question: 'is further decomposition necessary?'. Decomposition criteria are:

- will the module have too many statements?
- will the module be too complex?
- does the module have low cohesion?
- does the module have high coupling?
- does the module contain similar processing to other modules?

ESA PSS-05-04 'Guide to the Software Architectural Design Phase', discusses complexity, cohesion and coupling.

2.3.3 Reuse of the software

Software reuse questions can arise at all stages of design. In the AD phase decisions may have been taken to reuse software for all or some major components, such as:

- application generators;
- database management systems;
- human-computer interaction utilities;
- mathematical utilities;
- graphical utilities.

In the DD phase developers may have to:

- decide which library modules to use;
- build shells around the library modules to standardise interfaces (e.g. for error handling) and enhance portability;

- define conventions for the library modules to use (more than one combination of library modules might do the same job).

Developers should resist the temptation to clone a library module and make minor modifications, because of the duplication that results. Instead they should put any extra processing required in a shell module that calls the library module.

A common reason for reusing software is to save time at the coding and unit testing stage. Reused software is not normally unit tested because this has already been carried out in a previous project or by an external organisation. Nevertheless developers should convince themselves that the software to be reused has been tested to the standards appropriate for their project. Integration testing of reused software is always necessary to check that it correctly interfaces with the rest of the software. Integration testing of reused software can identify performance and resource problems.

2.3.4 Definition of module processing

The developer defines the processing steps in the second stage of detailed design. The developer should first outline the module processing in a Program Design Language (PDL) or pseudo-code and refine it, step-by-step, into a detailed description of the processing in the selected programming language.

The processing description should reflect the type of programming language. When using a procedural language, the description of the processing should contain only:

- sequence constructs (e.g. assignments, invocations);
- selection constructs (e.g. conditions, case statements);
- iteration constructs (e.g. do loops)

The definition of statements that do not affect the logic (e.g. i/o statements, local variable declarations) should be deferred to the coding stage.

Each module should have a single entry point and exit point. Control should flow from the entry point to exit point. Control should flow back only in an iteration construct, i.e. a loop. Branching, if used at all, should be restricted to a few standard situations (e.g. on error), and should always jump forward, not backward, in the control flow.

Recursion is a useful technique for processing a repeating data structure such as a tree, or a list, and for evaluating a query made up of arithmetic, relational, or logical expressions. Recursion should only be used if the programming language explicitly supports it.

PDLs and pseudo-code can be included in the code as comments, easing maintenance, whereas flowcharts cannot. Flowcharts are not compatible with the stepwise refinement technique, and so PDLs and pseudo-code are to be preferred to flowcharts for detailed design.

2.3.5 Defensive design

Developers should anticipate possible problems and include defences against them. Myers [Ref. 14] describes three principles of defensive design:

- mutual suspicion;
- immediate detection;
- redundancy.

The principle of mutual suspicion says that modules should assume other modules contain errors. Modules should be designed to handle erroneous input and error reports from other modules.

Every input from another module or component external to the program (e.g. a file) should be checked. When input data is used in a condition (e.g. CASE statement or IF... THEN.. ELSE...), an outcome should be defined for every possible input case. Every IF condition should have an ELSE clause.

When modules detect errors and return control to the caller, they should always inform the caller that an error has occurred. The calling module can then check the error flag for successful completion of the called module. It should be unnecessary for the caller to check the other module outputs.

It is possible for a subordinate to fail to return control. Modules should normally set a timeout' when waiting for a message, a rendezvous to be kept, or an event flag to be set. Modules should always act appropriately after a timeout (e.g. retry the operation). For a full discussion of error recovery actions, see reference 14.

The principle of immediate detection means that possible errors should be checked for immediately. If the reaction is not immediate, the error should be flagged for later action.

Conventions for taking action after detection of an error should be established. Normally error control responsibilities of a module should be at the same level as normal control responsibilities. Error logging, if done at all, should be done at the point where action is taken, and not at the point of error detection, since the significance of the error may not be apparent at the point of detection. It may be appropriate to insert diagnostic code at the point of detection, however.

Only the top level module should have responsibility for stopping the program. Developers should always check that there are no 'STOP' statements lurking in an otherwise apparently useful module. It may be impossible to reuse modules that suddenly take over responsibility for the control flow of the whole system. Halting the program and giving a traceback may be acceptable in prototype software (because it helps fault diagnosis), but not in operational software.

Redundancy has been discussed in ESA PSS-05-04, 'Guide to the Software Architectural Design Phase'. In detailed design, redundancy considerations can lead designers to include checksums in records and identity tags to confirm that an item is really what it is assumed to be (e.g. header record).

Myers also makes a useful distinction between 'passive fault detection' and 'active fault detection'. The passive fault detection approach is to check for errors in the normal flow of execution. Examples are modules that always check their input, and status codes returned from system calls. The active fault detection approach is to go looking for problems instead of waiting for them to arise. Examples are 'monitor' programs that continuously check disk integrity and attempts to violate system security.

Often most system code is dedicated to error handling. Library modules for error reporting should be made available to prevent duplication of error handling code. When modules detect errors, they should call error library modules to perform standard error handling functions such as error logging and display.

Defensive design principles have influenced the design of most modern languages. Strong type-checking languages automatically check that the calling data type matches the called data type, for example. Ada

goes further and builds range checking into the language. The degree to which a language supports defensive design can be a major factor in its favour.

2.3.6 Optimisation

Conventionally, optimisation means to make the best compromise between opposing tendencies. Improvement in one area is often associated with degradation in another. Software performance is often traded-off against maintainability and portability, for example.

The optimisation process is to:

- define the attributes to change (e.g. execution time);
- measure the attribute values before modifying the software;
- measure the attribute values after modifying the software;
- analyse the change in attribute values before deciding whether to modify the software again.

Optimisation can stop when the goals set in the SRD have been met. Every change has some risk, and the costs and benefits of each change should be clearly defined.

The 'law of diminishing returns' can also be used to decide when to stop optimisation. If there are only slight improvements in the values of attribute values after optimisation, the developers should stop trying to seek improvements.

Failure to get a group of people to agree about the solution to an optimisation problem is itself significant. It means that the attribute is probably optimised, and any improvement in one attribute results in an unacceptable degradation in another.

The structured programming method discourages optimisation because of its effect on reliability and maintainability. Code should be clear and simple, and its optimisation should be left to the compiler. Compilers are more likely to do a better job of optimisation than programmers, because compilers incorporate detailed knowledge of the machine. Often the actual causes of inefficiency are quite different from what programmers might suspect, and can only be revealed with a dynamic analysis tool (see Section 4.3.9).

In summary, developers should define what they are trying to optimise and why, before starting to do it. If in doubt, remember Jackson's two rules of optimisation [Ref. 6]:

- don't do it, but if you must:
- don't do it yet.

2.3.7 Prototyping

Experimental prototyping can be useful for:

- comparing alternative designs;
- checking the feasibility of the design.

The high-level design will normally have been identified during the AD phase. Detailed designs may have to be prototyped in the DD phase to find out which designs best meet the requirements.

The feasibility of a novel design idea should be checked by prototyping. This ensures that an idea not only works, but also that it works well enough to meet non-functional requirements for quality and performance.

2.3.8 Design reviews

Detailed designs should be reviewed top-down, level by level, as they are generated during the DD phase. Reviews may take the form of walkthroughs or inspections. Walkthroughs are useful on all projects for informing and passing on expertise. Inspections are efficient methods for eliminating defects before production begins.

Two types of walkthrough are useful:

- code reading;
- 'what-if?' analysis.

In a code reading, reviews trace the logic of a module from beginning to end. In 'what-if?' analysis, component behaviour is examined for specific inputs.

Static analysis tools evaluate modules without executing them. Static analysis functions are built in to some compilers. Output from static analysis tools may be input to a code review.

When the detailed design of a major component is complete, a critical design review must certify its readiness for implementation (DD10). The project leader should participate in these reviews, with the team leader and team members concerned.

2.3.9 Documentation

The developers must produce the DDD and the SUM. While the ADD specifies tasks, files and programs, the DDD specifies modules. The SUM describes how to use the software, and may be affected by documentation requirements in the SRD.

The recommended approach to module documentation is:

- create the module template to contain headings for the standard DDD entries:

  n Component identifier

  n.1 Type
  n.2 Purpose
  n.3 Function
  n.4 Subordinates
  n.5 Dependencies
  n.6 Interfaces
  n.7 Resources
  n.8 References
  n.9 Processing
  n.10 Data

- detail the design by filling in the sections, with the processing section containing the high-level definition of the processing in a PDL or pseudo-code;
- assemble the completed templates for insertion in the DDD.

A standard module template enables the DDD component specifications to be generated automatically. Tools to extract the module header from the source code and create an entry for the DDD greatly simplify maintenance. When a module is modified, the programmer edits, compiles and verifies the source module, and then runs the tool to generate the new DDD entry.

The SUM contains the information needed by the users of the software to operate it. The SUM should be gradually prepared during the DD

phase. The developers should exercise the procedures described in the SUM when testing the software.

2.4 TEST SPECIFICATIONS

The purpose of testing is to prove empirically that the system, or component under test, meets specified requirements. Normally it is not possible to prove by testing that a component contains no errors.

According to Myers, an important aspect of testing is to execute a program with the intention of finding errors [Ref. 14]. Testers should adopt this falsificationist' approach because it encourages testers to devise tests that the software fails, not passes. This idea originates from Karl Popper, the influential philosopher who first proposed the falsificationist' approach as a scientific method.

Testers must be critical and objective for testing to be effective. On large projects, system and acceptance test specifications should not be written by the analysts, designers and programmers responsible for the SRD, ADD, DDD and code. On small and medium-size projects, it is acceptable for the developer to write the test specifications and for the user or product assurance representatives to review them. Users should run the acceptance tests, not the developers.

2.4.1 Unit test planning

Unit test plans must be generated in the DD phase and documented in the Unit Test section of the Software Verification and Validation Plan (SVVP/UT). The Unit Test Plan should describe the scope, approach and resources required for the unit tests, and take account of the verification requirements in the SRD (see Section 2.5.3 and Chapter 7).

2.4 Test designs, test cases and test procedures

The developer must write specifications for the:

- acceptance tests,
- system tests,
- integration tests,
- unit tests

in the DD phase and document them in the SVVP. The specifications should be based on the test plans, and comply with the verification and acceptance testing requirements in the SRD. The specifications should define the:

- test designs (SVV19);
- test cases (SVV20);
- test procedures (SVV21);

see Section 2.6 below and Chapter 7.

When individual modules have been coded and unit tested, developers have to integrate them into larger components, test the larger components, integrate the larger components and so on. Integration is therefore inextricably linked with testing.

The Software Project Management Plan for the DD phase (SPMP/DD) should contain a delivery plan for the software based on the life cycle approach adopted. The delivery plan influences the integration tests defined in the Software Verification and Validation Plan (SVVP/IT). For a given delivery plan, the SVVP/IT should make integration testing efficient by minimising the number of test aids (e.g. drivers and stubs) and test data files required.

2.5 CODING AND UNIT TESTING

Coding is both the final stage of the design process and also the first stage of production. Coding produces modules, which must be unit tested. Unit testing is the second stage of the production process.

2.5.1 Coding

The transition from the detailed design stage to the coding stage comes when the developer begins to write modules that compile in the programming language. This transition is obvious when detailed design has been performed with flowcharts, but is less so when the developer has used a PDL or pseudo-code.

Coding must be based on the principles of:

- structured programming (DD03);
- concurrent production and documentation (DD04).

Every module should be understandable to a reviewer or maintenance programmer, moderately familiar with the programming language and unfamiliar with the program, the compiler and operating system. Understandability can be achieved in a variety of ways:

- including an introductory header for each module;
- declaring all variables;
- documenting all variables;
- using meaningful, unambiguous names;
- avoiding mixing data types;
- avoiding temporary variables;
- using parentheses in expressions;
- laying out code legibly;
- adding helpful comments;
- avoiding obscuring the module logic with diagnostic code;
- adhering to structured programming rules;
- being consistent in the use of the programming language;
- keeping modules short;
- keeping the code simple.

2.5.1.1 Module headers

Each module should have a header that introduces the module. This should include:

- title;
- configuration item identifier (SCM15);
- original author (SCM16);
- creation date (SCM17);
- change history (SCM18).

If tools to support consistency between source code and DDD are available, the introduction should be followed by explanatory comments from the component description in the DDD part 2.

The header usually comes after the module title statement (e.g. SUBROUTINE) and before the variable declarations.

The standard header should be made available so that it can be edited, completed and inserted at the head of each module.

2.5.1.2 Declarations of variables

Programmers should declare the type of all variables, whatever the programming language. The possible values of variables should either be stated in the variable declaration (as in Ada), or documented. Some languages, such as FORTRAN and Prolog, allow variables to be used without explicit declaration of their type.

Programmers using weakly typed languages should declare all variables. Where strong typing is a compiler option, it should always be used (e.g. IMPLICIT NONE in some FORTRAN compilers). Declarations should be grouped, with argument list variables, global variables and local variable declarations clearly separated.

2.5.1.3 Documentation of variables

Programmers should document the meaning of all variables. Documentation should be integrated with the code (e.g. as a comment to the declaration). The possible values of variables should either be stated in the variable declaration (as in Ada), or documented.

2.5.1.4 Names of variables

Finding meaningful names for variables exercises the imagination of every programmer, but it is effort well worth spending. Names should reflect the purpose of the variable. Natural language words should be used wherever possible. Abbreviations and acronyms, if used, should be defined, perhaps as part of the comment to a variable's declaration.

Similar names should be avoided so that a single typing error in the name of a variable does not identify another variable. Another reason not to use similar names is to avoid ambiguity.

2.5.1.5 Mixing data types

Some programming languages prevent the mixing of data types by the strong type checking feature (e.g. Ada). Mixing data types in expressions should always be avoided, even if the programming language allows it (e.g. FORTRAN). Examples of mixing data types include:

- mixing data types in expressions;

- mismatching data types in an argument list;
- equivalencing different data types.

2.5.1.6 Temporary variables

The temptation to define temporary variables to optimise code should be avoided. Although the use of temporary variables can simplify a statement, more statements are required, and more variables have to be declared. The net effect is that the module as a whole appears more complex.

2.5.1.7 Parentheses

Parentheses should be used in programming language expressions to avoid ambiguity, and to help the reader identify the sequences of operations to be performed. They should not be used solely to override the precedence rules of the language.

2.5.1.8 Layout and presentation

The layout of the code should allow the control logic to be easily appreciated. The usual technique is to separate blocks of sequential code from other statements by blank lines, and to indent statements in condition or iteration blocks. Statements that begin and end sequence, iteration and condition constructs should be aligned vertically (e.g. BEGIN... END, DO... ENDDO, and IF... ELSEIF... ELSE... ENDIF). Long statements should be broken and continued on the next line at a clause in the logic, not at the end of the line. There should never be more than one statement on a line.

2.5.1.9 Comments

Comments increase understandability, but they are no substitute for well-designed, well-presented and intelligible code. Comments should be used to explain difficult or unusual parts of the code. Trivial comments should be avoided.

PDL statements and pseudo-code may be preserved in the code as comments to help reviewers. Comments should be clearly distinguishable from code. It is not adequate only to use a language keyword; comments should be well separated from code by means of blank lines, and perhaps written in mixed case (if upper case is used for the code).

2.5.1.10 Diagnostic code

Diagnostic code is often inserted into a module to:

- make assertions about the state of the program;
- display the contents of variables.

Care should be taken to prevent diagnostic code obscuring the module logic. There is often a temptation to remove diagnostic code to present a clean 'final' version of the source code. However routine diagnostic code that allows verification of correct execution can be invaluable in the maintenance phase. It is therefore recommended that routine diagnostic code be commented out or conditionally compiled (e.g. included as 'debug lines'). Ad hoc diagnostic code added to help discover the cause of a particular problem should be removed after the problem has been solved.

2.5.1.11 Structured programming

The rules of structured programming are given in section 3.2.3. These rules should always be followed when a procedural language is used (such as FORTRAN, COBOL, Pascal or Ada). It is easy to break the rules of structured programming when the older procedural languages (e.g. FORTRAN, COBOL) are used, but less so with the more modern ones (Pascal, Ada).

2.5.1.12 Consistent programming

Language constructs should be used consistently. Inconsistency often occurs when modules are developed and maintained by different people. Coding standards can help achieve consistency, but it is not realistic for them to cover every situation. Modifications to the code should preserve the style of the original.

2.5.1.13 Module size

Modules should be so short that the entire module can be seen at once. This allows the structure, and the control logic, to be appreciated easily. The recommended maximum size of modules is about 50 lines, excluding the header and diagnostic code.

2.5.1.14 Code simplicity

Modules should be 'simple'. The principle of 'Occam's razor', (i.e. that an idea should be expressed by means of the minimum number of

entities), should be observed in programming. Simplicity can be checked formally by applying complexity measures [Ref. 11]. Simplicity can be checked informally using the rule of seven: the number of separate things that have to be held in mind when examining a part of the module should not exceed seven. Whatever method of evaluation is used, all measurements of simplicity should be confirmed by peer review.

2.5.2 Coding standards

Coding standards should be established for all the languages used, and documented or referenced in the DDD. They should provide rules for:

- presentation, (e.g. header information and comment layout);
- naming programs, subprograms, files, variables and data;
- limiting the size of modules;
- using library routines, especially:
  - operating system routines;
  - commercial library routines (e.g. numerical analysis);
  - project-specific utility routines;
- defining constants;
- defining data types;
- using global data;
- using compiler specific features not in the language standard;
- error handling.

2.5.3 Unit testing

Unit tests verify the design and implementation of all components from the lowest level defined in the detailed design up to the lowest level in the architectural design (normally the task level). Test harnesses, composed of a collection of drivers and stubs, need to be constructed to enable unit testing of modules below the top level.

Unit tests verify that a module is doing what it is supposed to do ('black box' testing), and that it is doing it in the way it was intended ('white box' testing). The traditional technique for white box testing is to insert diagnostic code. Although this may still be necessary for testing real-time behaviour, debuggers are now the preferred tool for white box testing. The usual way to test a new module is to step through a few test cases with the

debugger and then to run black box tests for the rest. Later, black box tests are run to fully exercise the module. The input test data should be realistic and sample the range of possibilities. Programmers revert to 'white box' testing mode when they have to debug a problem found in a black box test.

Before a module can be accepted, every statement shall be successfully executed at least once (DD06). Coverage data should be collected during unit tests. Tools and techniques for collecting coverage data are:

- debuggers;
- dynamic analysers;
- diagnostic code.

The inclusion of diagnostics can clutter up the code (see Section 2.5.1.10) and debuggers and dynamic analysers are much preferable. Coverage should be documented in a debug log, a coverage report, or a printout produced by the diagnostic code.

Unit testing is normally carried out by individuals or teams responsible for producing the component.

Unit test plans, test designs, test cases, test procedures and test reports are documented in the Unit Test section of the Software Verification and Validation Plan (SVVP/UT).

2.6 INTEGRATION AND SYSTEM TESTING

The third stage of the production process is to integrate major components resulting from coding and unit testing into the system. In the fourth stage of production, the fully integrated system is tested.

2.6.1 Integration

Integration is the process of building a software system by combining components into a working entity. Integration of components should proceed in an orderly function-by-function sequence. This allows the operational capabilities of the software to be demonstrated early, and thus gives visible evidence that the project is progressing.

Normally, the first components to be integrated support input and output. Once these components have been integrated and tested, they can be used to test others. Whatever sequence of functions is used, it should

minimise the resources required for testing (e.g. testing effort and test tools), while ensuring that all source statements are verified. The Software Verification and Validation Plan section for the integration tests (SVVP/IT) should define the assembly sequence for integration.

Within the functional groupings, components may be integrated and tested top-down or bottom-up. In top-down integration, 'stubs' simulate lower-level modules. In bottom-up integration, 'drivers' simulate the higher-level modules. Stubs and drivers can be used to implement test cases, not just enable the software to be linked and loaded.

The integration process must be controlled by the software configuration management procedures defined in the SCMP (DD05). Good SCM procedures are essential for correct integration. All deliverable code must be identified in a configuration item list (DD12).

2.6.2 Integration testing

Integration testing is done in the DD phase when the major components defined in the ADD are assembled. Integration tests should verify that major components interface correctly.

Integration testing must check that all the data exchanged across an interface comply with the data structure specifications in the ADD (DD07). Integration testing must confirm that the control flows defined in the ADD have been implemented (DD08).

Integration test designs, test cases, test procedures and test reports are documented in the Integration Test section of the Software Verification and Validation Plan (SVVP/IT).

2.6.3 System testing

System testing is the process of testing an integrated software system. This testing can be done in the development or target environment, or a combination of the two. System testing must verify compliance with system objectives, as stated in the SRD (DD09). System testing should include such activities as:

- passing data into the system, correctly processing and outputting it (i.e. end-to-end system tests);
- practice for acceptance tests (i.e. verification that user requirements will be met);
- stress tests (i.e. measurement of performance limits);

- preliminary estimation of reliability and maintainability;
- verification of the Software User Manual.

Trends in the occurrence of defects should be monitored in system tests; the behaviour of such trends is important for the estimation of potential acceptability.

For most embedded systems, as well as systems using special peripherals, it is often useful or necessary to build simulators for the systems with which the software will interface. Such simulators are often required because of:

- late availability of the other systems;
- limited test time with the other systems;
- desire to avoid damaging delicate and/or expensive systems.

Simulators are normally a separate project in themselves. They should be available on time, and certified as identical, from an interface point of view, with the systems they simulate.

System test designs, test cases, test procedures and test reports are documented in the System Test section of the Software Verification and Validation Plan (SVVP/ST).

The SUM is a key document during system (and acceptance) testing. The developers should verify it when testing the system.

2.7 PLANNING THE TRANSFER PHASE

Plans of TR phase activities must be drawn up in the DD phase. Generation of TR phase plans is discussed in Chapter 7. These plans cover project management, configuration management, verification and validation and quality assurance. Outputs are the:

- Software Project Management Plan for the TR phase (SPMP/TR);
- Software Configuration Management Plan for the TR phase (SCMP/TR);
- Software Quality Assurance Plan for the TR phase (SQAP/TR).

2.8 THE DETAILED DESIGN PHASE REVIEW

The development team should hold walkthroughs and internal reviews of a product before its formal review. After production, the DD Review (DD/R) must consider the results of the verification activities and decide whether to transfer the software (DD11). This should be a technical review. The recommended procedure is described in ESA PSS-05-10, which is based closely on the IEEE standard for Technical Reviews [Ref. 4].

Normally, only the code, DDD, SUM and SVVP/AT undergo the full technical review procedure involving users, developers, management and quality assurance staff. The Software Project Management Plan (SPMP/TR), Software Configuration Management Plan (SCMP/TR), and Software Quality Assurance Plan (SQAP/TR) are usually reviewed by management and quality assurance staff only.

In summary, the objective of the DD/R is to verify that:

- the DDD describes the detailed design clearly, completely and in sufficient detail to enable maintenance and development of the software by qualified software engineers not involved in the project;
- modules have been coded according to the DDD;
- modules have been verified according to the unit test specifications in the SVVP/UT;
- major components have been integrated according to the ADD;
- major components have been verified according to the integration test specifications in the SVVP/IT;
- the software has been verified against the SRD according to the system test specifications in the SVVP/ST;
- the SUM explains what the software does and instructs the users how to operate the software correctly;
- the SVVP/AT specifies the test designs, test cases and test procedures so that all the user requirements can be validated.

The DD/R begins when the DDD, SUM, and SVVP, including the test results, are distributed to participants for review. A problem with a document is described in a 'Review Item Discrepancy' (RID) form. A problem with code is described in a Software Problem Report (SPR). Review meetings are then held that have the documents, RIDs and SPRs as input. A review meeting should discuss all the RIDs and SPRs and decide an action for each. The review meeting may also discuss possible solutions to the

problems raised by them. The output of the meeting includes the processed RIDs, SPRs and Software Change Requests (SCR).

The DD/R terminates when a disposition has been agreed for all the RIDs. Each DD/R must decide whether another review cycle is necessary, or whether the TR phase can begin.

CHAPTER 3
METHODS FOR DETAILED DESIGN AND PRODUCTION

3.1 INTRODUCTION

This chapter summarises a range of design methods and programming languages. The details about each method and programming language should be obtained from the references. This guide does not make any particular method or language a standard, nor does it define a set of acceptable methods and languages. Each project should examine its needs, choose appropriate methods and programming languages, and justify the selection in the DDD.

3.2 DETAILED DESIGN METHODS

Detailed design first extends the architectural design to the bottom level components. Developers should use the same design method that they employed in the AD phase. ESA PSS-05-04, 'Guide to the Software Architectural Design Phase', discusses:

- Structured Design;
- Object Oriented Design;
- Jackson System Development;
- Formal Methods.

The next stage of design is to define module processing. This is done by methods such as:

- flowcharts;
- stepwise refinement;
- structured programming;
- program design languages (PDLs);
- pseudo coding;
- Jackson Structured Programming (JSP).

3.2.1 Flowcharts

A flowchart is 'a control flow diagram in which suitably annotated geometrical figures are used to represent operations, data, equipment, and arrows are used to indicate the sequential flow from one to another '[Ref. 2]. It should represent the processing.

Flowcharts are an old software design method, dating from a time when the only tools available to a software designer were a pencil, paper, and stencil. A box is used to represent process steps and diamonds are used to represent decisions. Arrows are used to represent control flow.

Flowcharts predate structured programming and they are difficult to combine with a stepwise refinement approach. Flowcharts are not well supported by tools and so their maintenance can be a burden. Although directly related to module internals, they cannot be integrated with the code, unlike PDLs and pseudo-code. For all these reasons, flowcharts are no longer a recommended technique for detailed design.

3.2.2 Stepwise refinement

Stepwise refinement is the most common method of detailed design. The guidelines for stepwise refinement are:

- start from functional and interface specifications;
- concentrate on the control flow;
- defer data declarations until the coding phase;
- keep steps of refinement small to ease verification;
- review each step as it is made.

Stepwise refinement is closely associated with structured programming (see Section 3.2.3).

3.2.3 Structured programming

Structured programming is commonly associated with the name of E.W. Dijkstra [Ref. 10]. It is the original 'structured method' and proposed:

- hierarchical decomposition;
- the use of only sequence, selection and iteration constructs;
- avoiding jumps in the program.

Myers emphasises the importance of writing code with the intention of communicating with people instead of machines [Ref.14].

The Structured Programming method emphasises that simplicity is the key to achieving correctness, reliability, maintainability and adaptability. Simplicity is achieved through using only three constructs: sequence, selection and iteration. Other constructs are unnecessary.

Structured programming and stepwise refinement (see Section 3.2.3) are inextricably linked. The goal of refinement is to define a procedure that can be encoded in the sequence, selection and iteration constructs of the selected programming language.

Structured programming also lays down the following rules for module construction:

- each module should have a single entry and exit point;
- control flow should proceed from the beginning to the end;
- related code should be blocked together, not dispersed around the module;
- branching should only be performed under prescribed conditions (e.g. on error).

The use of control structures other than sequence, selection and iteration introduces unnecessary complexity. The whole point about banning 'GOTO' was to prevent the definition of complex control structures. Jumping out of loops causes control structures only to be partially contained within others and makes the code fragile.

Modern block-structured languages, such as Pascal and Ada, implement the principles of structured programming, and enforce the three basic control structures. Ada supports branching only at the same logical level and not to arbitrary points in the program.

The basic rules of structured programming can lead to control structures being nested too deeply. It can be quite difficult to follow the logic of a module when the control structures are nested more than three or four levels. Three common ways to minimise this problem are to:

- define more lower-level modules;
- put the error-handling code in blocks separate to the main code.
- branching to the end of the module on detecting an error.

3.2.4 Program Design Languages

Program Design Languages (PDL) are used to develop, analyse and document a program design [from Ref. 7]. A PDL is often obtained from the essential features of a high-level programming language. A PDL may contain special constructs and verification protocols.

It is possible to use a complete programming language (e.g. Smalltalk, Ada) as a PDL [Ref. 7]. ANSI/IEEE Std 990-1987 'IEEE Recommended Practice for Ada As a Program Design Language', provides recommendations 'reflecting the state of the art and alternative approaches to good practice for characteristics of PDLs based on the syntax and semantics of Ada' [Ref. 9]. Using Ada as a model, it says that a PDL should provide support for:

- abstraction;
- decomposition;
- information hiding;
- stepwise refinement;
- modularity;
- algorithm design;
- data structure design;
- connectivity;
- adaptability.

Adoption of a standard PDL makes it possible to define interfaces to CASE tools and programming languages. The ability to generate executable statements from a PDL is desirable.

Using an entire language as a PDL increases the likelihood of tool support. However, it is important that a PDL be simple. Developers should establish conventions for the features of a language that are to be used in detailed design.

PDLs are the preferred detailed design method on larger projects, where the existence of standards and the possibility of tool support makes them more attractive than pseudo-code.

3.2.5 Pseudo-code

Pseudo-code is a combination of programming language constructs and natural language used to express a computer program design [Ref. 2]. Pseudo-code is distinguished from the code proper by the presence of statements that do not compile. Such statements only indicate what needs to be coded. They do not affect the module logic.

Pseudo-code is an informal PDL (see Section 3.2.4) that gives the designer greater freedom of expression than a PDL, at the sacrifice of tool support. Pseudo-code is acceptable for small projects and in prototyping, but on larger projects a PDL is definitely preferable.

3.2.6 Jackson Structured Programming

Jackson Structured Programming (JSP) is a program design technique that derives a program's structure from the structures of its input and output data [Ref. 6]. The JSP dictum is that the program structure should match the data structure'.

In JSP, the basic procedure is to:

- consider the problem environment and define the structures for the data to be processed;
- form a program structure based on these data structures;
- define the tasks to be performed in terms of the elementary operations available, and allocate each of those operations to suitable components in the program structure.

The elementary operations (i.e. statements in the programming language) must be grouped into one of the three composite operations: sequence, iteration and selection. These are the standard structured programming constructs, giving the technique its name.

JSP is suitable for the detailed design of software that processes sequential streams of data whose structure can be described hierarchically. JSP has been quite successful for information systems applications.

Jackson System Development (JSD) is a descendant of JSP. If used, JSD should be started in the SR phase (see ESA PSS-05-03, Guide to the Software Requirements Definition Phase).

3.3 PRODUCTION METHODS

Software production involves writing code in a programming language, verifying the code and integrating it with other code to make a working system. This section therefore discusses programming languages and integration methods.

3.3.1 Programming languages

Programming languages are best classified by their features and application domains. Classification by 'generation' (e.g. 3GL, 4GL) can be very misleading because the generation of a language can be completely unrelated to its age (e.g. Ada, LISP). Even so, study of the history of programming languages can give useful insights into the applicability and features of particular languages [Ref. 13].

3.3.1.1 Feature classification

The following classes of programming languages are widely recognised:

- procedural languages;
- object-oriented languages;
- functional languages;
- logic programming languages.

Application-specific languages based on database management systems are not discussed here because of their lack of generality. Control languages, such as those used to command operating systems, are also not discussed for similar reasons.

Procedural languages are sometimes called 'imperative languages' or 'algorithmic languages'. Functional and logic programming languages are often collectively called 'declarative languages' because they allow programmers to declare 'what' is to be done rather than 'how'.

3.3.1.1.1 Procedural languages

A 'procedural language' should support the following features:

- sequence (composition);
- selection (alternation);
- iteration;
- division into modules.

The traditional procedural languages such as COBOL and FORTRAN support these features.

The sequence construct, also known as the composition construct, allows programmers to specify the order of execution. This is trivially done by placing one statement after another, but can imply the ability to branch (e.g. GOTO).

The sequence construct is used to express the dependencies between operations. Statements that come later in the sequence depend on the results of previous statements. The sequence construct is the most important feature of procedural languages, because the program logic is embedded in the sequence of operations, instead of in a data model (e.g. the trees of Prolog, the lists of LISP and the tables of RDBMS languages).

The selection construct, also known as the condition or alternation construct, allows programmers to evaluate a condition and take appropriate action (e.g. IF... THEN and CASE statements).

The iteration construct allows programmers to construct loops (e.g. DO...). This saves repetition of instructions.

The module construct allows programmers to identify a group of instructions and utilise them elsewhere (e.g. CALL...). It saves repetition of instructions and permits hierarchical decomposition.

Some procedural languages also support:

- block structuring;
- strong typing;
- recursion.

Block structuring enforces the structured programming principle that modules should have only one entry point and one exit point. Pascal, Ada and C support block structuring.

Strong typing requires the data type of each data object to be declared [Ref. 2]. This stops operators being applied to inappropriate data objects and the interaction of data objects of incompatible data types (e.g. when the data type of a calling argument does not match the data type of a called argument). Ada and Pascal are strongly typed languages. Strong typing helps a compiler to find errors and to compile efficiently.

Recursion allows a module to call itself (e.g. module A calls module A), permitting greater economy in programming. Pascal, Ada and C support recursion.

3.3.1.1.2 Object-oriented languages

An object-oriented programming language should support all structured programming language features plus:

- inheritance;
- polymorphism;
- messages.

Examples of object-oriented languages are Smalltalk and C++. Reference 14 provides a useful review of object-oriented programming languages.

Inheritance is the technique by which modules can acquire capabilities from higher-level modules, i.e. simply by being declared as members of a class, they have all the attributes and services of that class.

Polymorphism is the ability of a process to work on different data types, or for an entity to refer at runtime to instances of specific classes. Polymorphism cuts down the amount of source code required. Ideally, a language should be completely polymorphic, so the need to formulate sections of code for each data type is unnecessary. Polymorphism implies support for dynamic binding.

Object-oriented programming languages use 'messages' to implement interfaces. A message encapsulates the details of an action to be performed. A message is sent from a 'sender object' to a 'receiver object' to invoke the services of the latter.

3.3.1.1.3 Functional languages

Functional languages, such as LISP and ML, support declarative structuring. Declarative structuring allows programmers to specify only 'what' is required, without stating how it is to be done. It is an important feature, because it means standard processing capabilities are built into the language (e.g. information retrieval) .

With declarative structuring, procedural constructs are unnecessary. In particular, the sequence construct is not used for the program logic. An underlying information model (e.g. a tree or a list) is used to define the logic.

If some information is required for an operation, it is automatically obtained from the information model. Although it is possible to make one operation depend on the result of a previous one, this is not the usual style of programming.

Functional languages work by applying operators (functions) to arguments (parameters). The arguments themselves may be functional expressions, so that a functional program can be thought of as a single expression applying one function to another. For example if DOUBLE is the function defined as DOUBLE(X) = X + X, and APPLY is the function that executes another function on each member of a list, then the expression APPLY(DOUBLE, [1, 2, 3]) returns [2, 4, 6].

Programs written in functional languages appear very different from those written in procedural languages, because assignment statements are absent. Assignment is unnecessary in a functional language, because all relationships are implied by the information model.

Functional programs are typically short, clear, and specification-like, and are suitable both for specification and for rapid implementation, typically of design prototypes. Modern compilers have reduced the performance problems of functional languages.

A special feature of functional languages is their inherent suitability for parallel implementation, but in practice this has been slow to materialise.

3.3.1.1.4 Logic programming languages

Logic programming languages implement some form of classical logic. Like functional languages, they have a declarative structure. In addition they support:

- backtracking;
- backward chaining;
- forward chaining.

Prolog is the foremost logic programming language.

Backtracking is the ability to return to an earlier point in a chain of reasoning when an earlier conclusion is subsequently found to be false. It is especially useful when traversing a knowledge tree. Backtracking is incompatible with assignment, since assignment cannot be undone

because it erases the contents of variables. Languages which support backtracking are, of necessity, non-procedural.

Backward chaining starts from a hypothesis and reasons backwards to the facts that cause the hypothesis to be true. For example if the fact A and hypothesis B are chained in the expression IF A THEN B, backwards chaining enables the truth of A to be deduced from the truth of B (note that A may be only one of a number of reasons for B to be true).

Forward chaining is the opposite of backward chaining. Forward chaining starts from a collection of facts and reasons forward to a conclusion. For example if the fact X and conclusion Y are chained in the expression IF X THEN Y, forward chaining enables the truth of Y to be deduced from the truth of X.

Forward chaining means that a change to a data item is automatically propagated to all the dependent items. It can be used to support 'data-driven' reasoning.

3.3.1.2 Applications

Commonly recognised application categories for programming languages are:

- real-time systems, including control systems and embedded systems;
- transaction processing, including business and information systems;
- numerical analysis;
- simulation;
- human-computer interaction toolkits (HCI);
- artificial intelligence (AI);
- operating systems;

The application area of language strongly influences the features that are built into it. There is no true 'general purpose language', although some languages are suitable for more than one of the application categories listed above.

3.3.1.3 FORTRAN

FORTRAN was the first widely-used high-level programming language. Its structuring concepts include the now-familiar IF... THEN

alternation, DO iteration, primitive data types such as integer, and procedure composition. FORTRAN does not support recursion.

FORTRAN is familiar to generations of programmers who have learned to use it safely. It is a simple language, compared with Ada, and can be compiled rapidly into efficient object code. Compilers are available for most hardware, but these are not always compatible, both because of the addition of machine-dependent features, especially input/output, and because of successive versions of FORTRAN itself. However, good standards exist.

The language includes some features now generally considered risky, such as:

- the GOTO statement, which allows the rules of structured programming to be easily broken;
- the EQUIVALENCE statement, which permits different names to be used for the same storage location;
- local storage, which permits possibly undesired values to be used from a previous call;
- multiple entry and exit points.

FORTRAN remains the primary language for scientific software. Most compilers allow access to operating system features, which, at the expense of portability, allows FORTRAN to be used for developing real-time control systems.

3.3.1.4 COBOL

COBOL remains the most widely used programming language for business and administration systems. As a procedural language it provides the usual sequence, selection (IF... OTHERWISE...) and iteration constructs (PERFORM...). COBOL has powerful data structuring and file handling mechanisms (e.g. hierarchically arranged records, direct access and indexed files). Although its limited data manipulation facilities severely restricts the programmers' ability to construct complex arithmetic expressions, it does allow fixed point arithmetic, necessary for accurate accounting.

COBOL was conceived as the first 'end-user' programming language. It relies on verbal expressions, not symbols (e.g. ADD, SUBTRACT, MULTIPLY, DIVIDE instead of +, -, x and). It has therefore been criticised as forcing a verbose programming style. While this does help

make COBOL programs self-explanatory, COBOL programs can still be difficult to understand if they are not well-structured. Jackson's Principles of Program Design are explained with the aid of examples of good and bad COBOL programming [Ref. 6].

3.3.1.5 Pascal

Pascal was conceived by a single designer (Niklaus Wirth) to teach the principles of structured programming. It is accordingly simple and direct, though perhaps idiosyncratic. It contains very clear constructs: IF... THEN... ELSE, WHILE... DO, REPEAT... UNTIL, CASE... OF, and so on. Sets, procedures, functions, arrays, records and other constructs are matched by a versatile and safe data typing system, which permits the compiler to detect many kinds of error.

Wirth omitted several powerful features to make Pascal compile efficiently. For example, functions may not return more than one argument, and this must be a primitive type (although some extensions permit records to be returned). String handling and input/output functions are simplistic and have been the source of portability problems.

Pascal handles recursive data structures, such as lists and trees by means of explicit pointers and (sometimes) variant records. These defeat the type checking mechanism and are awkward to handle.

Pascal is now well-defined in an ISO standard, and is very widely available with good compilers and supporting tools. To a considerable extent the language enforces good programming practice.

3.3.1.6 C

C is a language in the same family as Pascal, with its roots in Algol. The language is less strongly typed than Pascal or Algol. It is easy and efficient to compile, and because of its low-level features, such as bit manipulation, can exploit machine-dependent facilities.

C has a rich variety of operators that allow programmers to adopt a very terse style. This can unfortunately make programs difficult to read and understand. Discipline is required to produce code that is portable and maintainable. Coding standards are especially necessary to make C code consistent and readable.

One advantage of C is its close connection with the Unix operating system, which guarantees C a place as a (or the) major systems

programming language. A very wide range of tools and library modules are available for the language.

3.3.1.7 Modula-2

Modula-2 is derived from Pascal. It offers a few new constructs and considerably more power and generality. Two important features are modularity, including incremental compilation, and simulation of concurrency and inter-process communication (via co-routining).

Niklaus Wirth, the inventor of both Pascal and Modula-2, intended that Modula-2 replace Pascal. However Modula-2 is not widely used because of the scarcity of development tools for popular platforms. In contrast to Ada, its main competitor, it is a 'small' language (its compiler is only about 5000 lines, as opposed to Ada's several hundred thousand lines [Ref. 13]).

3.3.1.8 Ada

Ada is a powerful language well suited for the creation of large, reliable and maintainable systems. Unlike most languages it was systematically developed (from a US Department of Defense specification [Ref. 7]). Its many features include tasks (processes) able to communicate asynchronously with each other, strong typing and type checking, and generic programming.

Ada derives ideas from many sources, including several of the languages mentioned here. Although it is a 'large' language, and therefore difficult to master, it can seem relatively familiar to C and Pascal programmers. Applications written in Ada may often be slower than comparable applications written in other languages, perhaps largely due to the immaturity of the compilers. This may become less of a problem in time (as has been seen with other languages).

Ada provides features normally called from the operating system by other languages, providing a valuable degree of device independence and portability for Ada programs. However Ada real time control features may not be adequate for some applications, and direct access to operating system services may be necessary.

Ada does not support inheritance, so it is not an object-oriented language. Ada allows programmers to create 'generic packages' which act as classes. However it is not possible to structure generic packages and inherit package attributes and services.

3.3.1.9 Smalltalk

Smalltalk is the leading member of the family of object-oriented languages. The language is based on the idea of classes of object, which communicate by passing messages, and inherit attributes from classes to which they belong.

Smalltalk is inherently modular and extensible; indeed, every Smalltalk program is an extension of the core language. Large libraries are available offering a wide range of predefined classes for graphics, data handling, arithmetic, communications and so on. Programs can often be created by minor additions to, or modifications of, existing objects.

The language is thus very suitable for rapid iterative development and prototyping, but it is also efficient enough for many applications. It is the language of choice for educating software engineers in object-oriented programming.

Smalltalk differs from other programming languages in being a complete programming environment. This feature is also the language's Achilles heel: using Smalltalk is very much an 'all or nothing' decision. Interfacing to software written in other languages is not usually possible.

3.3.1.10 C++

C++ is an object-oriented and more strongly typed version of C. It therefore enforces greater discipline in programming than C. It is a noticeably more modern language, exploiting the strongest concepts of Pascal and Smalltalk, while providing all the features of its parent language C. The rapid emergence of C++ has been helped by the appearance of tools and standards.

3.3.1.11 LISP

LISP was the first list processing language. It is mostly a functional language but with some procedural features. Because a program is itself a list, it can readily be treated as data by another program. Hence, LISP is an excellent language for writing interpreters and expert systems, and much pioneering work in artificial intelligence has been done in the language.

LISP is now a mature language and accordingly has a rich set of predefined tools (functions) and environments available for it. LISP compilers are available on most architectures, but these are not well standardised; so-called Common LISP is a widespread dialect with

commercial backing. LISP is often run on specialised workstations with language-sensitive editors, interpreters and debuggers, and sometimes with dedicated hardware. Interfaces to C and Prolog, and to procedural components like graphics packages, are frequently provided.

The language was once heavily criticised for inefficiency and illegibility. List manipulation is inherently more costly than, say, array manipulation, but careful optimisation and good compilers have greatly reduced the overheads. The style of LISP programs is more verbose and less legible than that of modern functional languages such as ML. However, recent versions of LISP have again become more purely functional and have updated their syntax. The problem of understanding heavily bracketed expressions (lists of lists) has largely been solved with pretty-printers and automatic bracket checkers.

3.3.1.12 ML

ML (short for Meta-Language) is the most widely used of a family of functional (i.e. non-procedural) programming languages that include Hope and Miranda. Like Prolog, it is declarative and admirably free of side-effects. Functions are similar to those in Pascal, except that they may take any desired form and return essentially any type of result as a structure. ML is notably modular and polymorphic, permitting creation of reusable components in a style analogous to that of Ada. ML lacks the quirkiness of some functional languages, and permits procedural constructs for efficiency.

3.3.1.13 Prolog

Prolog was the first of a family of logic programming languages. Full predicate logic is too complex to implement efficiently, so Prolog compromises by implementing a restricted but useful subset, namely Horn Clause logic. This permits a single affirmative conclusion, but from any number of premises. If a conclusion can be reached in more than one way, more than one clause is provided for it. Thus, groups of clauses act as the analogues of IF... THEN or CASE statements, while a single clause is effectively a procedure or function.

Because a conclusion may fail to be reached by one route, but the same, or another conclusion may be reached by another route, Prolog permits backtracking. This provides a powerful search mechanism. The need to maintain an extensive stack means, however, that Prolog requires a lot of run-time storage.

The language is essentially declarative, not procedural, as the clauses relate one fact or conclusion to another: thus, if something is true then it is always true. Prolog is very well suited to systems embodying rules, deduction, and knowledge (including databases), though it is capable of graphics, arithmetic and device control on a range of machines. It is not ideally suited for highly numerical applications. Prolog itself is (virtually) typeless and freely polymorphic, making it insecure; many of its daughter languages embody varieties of type checking. The language is inherently extensible. With its interactive environment, it is very suitable for prototyping; it has also been used for specification. Most Prolog compilers permit modules of C to be called, typically as if they were built-in predicates.

3.3.1.14 Summary

Table 3.3.1.14 lists the programming languages, their types, primary applications and standards.

Language	Type	Primary Applications	Standard
FORTRAN	Procedural	Numerical analysis Real-time systems	ISO 1539
COBOL	Procedural	Transaction processing	ISO 1989
Pascal	Procedural	Numerical analysis Real-time systems	ISO 7185
C	Procedural	Real-time systems Operating systems	ANSI
Modula-2	Procedural	Real-time systems Numerical analysis	
Ada	Procedural	Real-time systems Simulation	MIL STD 1815A-1983
Smalltalk	Object-oriented	Simulation HCI toolkits	Smalltalk-80
C + +	Object-oriented	Real-time systems Simulation HCI toolkits	AT&T
LISP	Functional	AI	Common LISP
ML	Functional	AI	
Prolog	Logic programming	AI	Edinburgh syntax

Table 3.3.1.14 Summary of Programming Languages

3.3.2 Integration methods

A function-by-function integration method should be used that:

- establishes the infrastructure functions first;
- harmonises with the delivery plan.

It is necessary to establish the infrastructure functions to minimise the amount of test software needed. Examples of infrastructure functions are those that provide data input and output. Building the infrastructure saves the effort of providing drivers and stubs for the components that use it. The infrastructure provides the kernel system from which the rest of the system can grow.

In an incremental development, the delivery plan can constrain the integration method by forcing functions to be delivered in a user-specified order, not one that minimises the integration effort.

Each function will be provided by a group of components that may be integrated 'top-down' and 'bottom-up'. The idea of integrating all the components at one level before proceeding to the next is common to both methods. In top-down integration, the next level to integrate is always the next lower one, while in bottom-up integration the next higher level is integrated.

The integration methods are described in more detail below.

3.3.2.1 Function-by-function

The steps in the function-by-function method are to:

1) select the functions to be integrated;

2) identify the components that carry out the functions;

3) identify the component dependencies (i.e. input data flows or control flows).

4) order the components by the number of dependencies (i.e. fewest dependencies first);

5) when a component depends on another later in the order, create a driver to simulate the input of the component later in the order;

6) introduce the components with the fewest dependencies first.

This approach minimises the number of stubs and drivers required. Stubs and drivers simulate the input to a component, so if the 'real' tested components provide the input, effort does not need to be expended on producing stubs and drivers.

Data flow diagrams are useful for representing component dependencies. Output flows that are not input to any other component should be omitted from the diagram.

When the incremental delivery life cycle approach is being used, the basic procedure above must be modified:

a) define the functions;

b) define when the functions are required;

c) for each release;

do steps 1) to 6) above.

Dependencies with components in a previous or a future release are not counted. If one component depends upon another in a previous release, the existing software can satisfy the dependency. If the component depends upon a component in a future release then a driver to simulate the input must be provided.

3.3.2.2 Top-down integration

The top-down approach to integration is to use 'stub' modules to represent lower-level modules. As modules are completed and tested, they replace the stubs. Stubs can be used to implement test cases.

3.3.2.3 Bottom-up integration

The bottom-up approach to integration is to use 'driver' modules to represent higher-level modules. As modules are completed and tested, they replace the drivers. Drivers can be used to implement test cases.

CHAPTER 4
TOOLS FOR DETAILED DESIGN AND PRODUCTION

4.1 INTRODUCTION

This chapter discusses the tools for detailing the design and producing the software. Tools can be combined to suit the needs of a particular project.

4.2 DETAILED DESIGN

4.2.1 CASE tools

In all but the smallest projects, CASE tools should be used during the DD phase. Like many general purpose tools (e.g. such as word processors and drawing packages), CASE tools should provide:

- windows, icons, menu and pointer (WIMP) style interface for the easy creation and editing of diagrams;
- what you see is what you get (WYSIWYG) style interface that ensures that what is created on the display screen closely resembles what will appear in the document.

Method-specific CASE tools offer the following features not offered by general purpose tools:

- enforcement of the rules of the methods;
- consistency checking;
- easy modification;
- automatic traceability of components to software requirements;
- configuration management of the design information;
- support for abstraction and information hiding;
- support for simulation.

CASE Tools should have an integrated data dictionary or another kind or 'repository'. This is necessary for consistency checking. Developers should check that a tool supports the parts of the method that they intend to use. ESA PSS-05-04, 'Guide to the Software Architectural Design Phase' contains a more detailed list of desirable tool capabilities. The STARTs

Guide provides a good survey of tools for requirements analysis and configuration management [Ref. 3].

4.2.2 Configuration managers

Configuration management of the physical model is essential. The model should evolve from baseline to baseline as it develops in the DD phase, and enforcement of procedures for the identification, change control and status accounting of the model are necessary. In large projects, configuration management tools should be used for the management of the model database.

4.2.3 Precompilers

A precompiler generates code from PDL specifications. This is useful in design, but less so in later stages of development unless software faults can be easily traced back to PDL statements.

4.3 PRODUCTION

A range of production tools are available to help programmers develop, debug, build and test software. Table 4.3 lists the tools in order of their appearance in the production process.

Tool	Purpose
CASE tools	generate module shells
code generators	translate formal relationships into source code
editors	create and modify source code and documentation
language-sensitive editors	create syntactically correct source code
static analysers	examine source code
compilers	translate source code to object code
linkers	join object modules into executable programs
debuggers	locate errors during execution
dynamic analysers	examine running programs
test tools	test modules and programs
word processors	document production
documentation generators	derive user documentation from the source code
configuration managers	store and track versions of modules and files, and record the constituents of each build

Table 4.3: Production tools

4.3.1 Modelling tools

Many modelling tools automatically generate constant, type, and variable declarations for inclusion in the source code of every module. Some modelling tools can translate diagrams showing the calling tree of modules into fully commented function or procedure calls, though these may lack values for actual parameters.

If modelling tools are used, coding begins by completing the skeleton of the module. All the calls are completed; then iteration constructs (WHILE, REPEAT, LOOP, etc) are entered; then alternation constructs (IF, CASE, etc) are inserted. Finally, low-level details such as arithmetic, input/output, and other system calls are filled in.

4.3.2 Code generators

Numerous code generator packages are now available, claiming to take the work out of design and coding. They can help reduce the workload in some areas, such as database management and human-computer interaction. These areas are characterised by repetitive code, and the need to perform numerous trivial but essential operations in set sequences. Such tasks are best automated for accuracy and efficiency.

As code generators become more closely integrated with design methods, it will be possible to code a larger proportion of the components of any given system automatically. Current design methods generally provide limited code generation, for example creating the data declarations and module skeletons; module bodies must then be coded by hand.

Even if parts of the system are to be coded manually, there are still advantages in using a code generator. Changes in software requirements can result in automatic changes to data and module declarations, preserving and checking consistency across the phases of the life-cycle.

4.3.3 Editors

Largely replaced by the word processor for documentation, basic text editors are still widely used for source code creation and modification. Although language-sensitive editors (see Section 4.3.4) offer greater functionality, basic text editors provide all the facilities that many programmers require.

4.3.4 Language-sensitive editors

Language-sensitive editors are now available for many programming languages. They contain an interpreter that helps the user to write syntactically correct code. For example, if the user selects ‘open a file', a menu appears of the known files of the right type. Language-sensitive editors are not ideal for some programming languages, because of the richness of their syntax. Skilled programmers often find them restrictive.

Another approach provides the reserved words of a language from a keypad, which may be real (on a keyboard or tablet), or virtual (on a screen panel). The selection of a keyword like WHILE with a single action is a convenience; the editor may also insert associated syntactic items, such as brackets and comments such as /* END WHILE */ . This does not prevent errors, but is at least a useful guide and the freedom of the programmer is not sacrificed.

The simplest language-sensitive editors do little more than recognise brackets and provide indentation to match. For example, the Pascal reserved words BEGIN and END bracket blocks in IF... THEN... ELSE statements. Automatic indentation after these words have been typed helps to make programs readable and reduces mistakes.

It is also possible to provide templates for program construction, containing standard headers, the obligatory sections for constant and type declarations, and so on. These may be generated automatically by CASE tools. Editors which recognise these templates (and the general form of a legal module), can speed development and help prevent errors. There is considerable scope for the further integration with CASE tools and code generators.

4.3.5 Static analysers

Static analysis is the process of scanning the text of a program to check for faults of construction, dangerous or undesirable features, and the application of software standards. They are especially useful for relatively unstructured languages such as assembler and FORTRAN. Static analysers may:

- check for variables that are not used, or are used before being assigned a value;
- check that the ranges of variables stay within bounds;
- provide a view of the structure of an application;

- provide a view of the internal logical structure of each module;
- measure the complexity of the code with respect to a metric, such as cyclomatic complexity [Ref. 11];
- translate the source code to an intermediate language for formal verification;
- symbolically execute the code by substituting algebraic symbols for program variables;
- measure simple attributes of the code, such as the number of lines of code, and the maximum level of nesting.

Although they can be used to expose poorly structured code, static analysers should not be relied upon to improve poor programming. Structuring should be done in architectural and detailed design, not after implementation.

Most compilers provide some simple static analysis features, such as checking for variables that are not used (see Section 4.3.6). Dedicated static analysis tools usually provide advanced static analysis functions, such as analysis of code structure.

Static analysis tools are no substitute for code review. They just support the review process. Some bad programming practices, such as the choice of meaningful identifiers, evade all known static analysis tools, for example.

4.3.6 Compilers

The choice of compiler on a given computer or operating system may be limited. However, compilers vary widely in speed, thoroughness of checking, ease of use, handling of standard syntax and of language extensions, quality of listings and code output, and programming support features. The choice of compiler is therefore crucial. Trade-offs can become complex, leading to selection by reputation and not by analysis. Users should assess their compilation needs and compare the available options on this basis.

Speed of compilation affects the ease and cost of developing, debugging, and maintaining the product, whereas code quality affects the runtime performance of the product itself. Benchmark source code should therefore reflect the type of code likely to be found in the product to be developed. Compilers should be compared on such data by measuring

compilation time, execution time, and object code size. Runtime stack and heap sizes may also be critical where memory is scarce.

Full checking of compiler compliance with language standards is beyond the means of essentially all ESA software projects. Standards organisations such as ISO, ANSI and BSI certify compilers for a few languages including FORTRAN, COBOL and Pascal. The US Department of Defense certifies Ada compilers. Manufacturers' documentation should be scanned for extensions to, or omissions from, the standards.

Compilers vary widely in programming support features. Good compilers offer much help to the developer, for example:

- full listings;
- cross-reference;
- data on module sizes;
- diagnostics;
- thorough checking;
- switches (e.g. array bounds checking; strict language or extensions).

Some compilers now offer many of the features of development environments, including built-in editors and debuggers, trace tools, version control, linkers, and incremental compilation (with a built-in interpreter). These can substantially speed development.

Where there are switchable checks, project programming standards (DDD, section 2.4) should state which options are applicable.

The most advanced compilers perform a range of optimisations for sequential and parallel machines, attempting to discover and eliminate source code inefficiencies. These may be switchable, for example by directives in the source code. Ada explicitly provides for such directives with the þragma' statement. Users should investigate whether the optimisations they require are implemented in candidate compilers.

4.3.7 Linkers

The linker may be provided with the machine, operating system, or compiler; or, as with Ada, may be integrated with the compiler/interpreter and runtime system. The user therefore has little control over the choice of linker. When there is a choice, it should be considered carefully, as modern

linkers vary considerably in performance, affecting especially the speed of debugging and maintenance.

It is convenient if the linker can automatically discover the appropriate libraries and directories to use, and which modules or components to link. Most linkers can be controlled by parameters, which can be created by a build or make utility; some are closely integrated with such utilities, or indeed with compilers.

In a large system, linking can take significantly longer than compilation, so users should compare linker performance on realistic benchmarks before selecting one for a project. Speed is not the only parameter to consider: some linkers may generate better quality executable code than others.

4.3.8 Debuggers

The use of interactive symbolic debuggers is strongly encouraged, especially for verification. A good debugger is integrated with an editor as well as a compiler/interpreter, and permits a range of investigative modes. Convenient debugging modes include step-by-step execution, breakpoint (spypoint) tracing, variable value reporting, watch condition setting (e.g. a variable beyond a limit value), and interaction.

For graphics, windows, menus, and other software involving cursor control, the debugger must be properly windowed to avoid confusion with the software being debugged. On some devices, illegal calls, e.g. graphics calls outside the frame area, cause processes to abort; debuggers should be able to trap such calls and permit interactive recovery action, or at least diagnosis.

The debugging of real-time software, where it is not possible to step through code, is a special challenge. The traditional diagnostic log is still useful here. The requirement is, as with all debugging, to view the software in a realistic environment. This may be possible with a simulator; if not, hardware-style techniques, in which bus or circuit signals are monitored in real time, may be necessary. Implementers of real-time projects should consider how they may debug and verify their software, and should allocate resources for this purpose.

4.3.9 Dynamic analysers

Dynamic analysis is the process of measuring how much machine resources (e.g. CPU time, i/o time, memory) each module, and line of code,

consumes. In contrast to static analysis (see Section 4.3.5), dynamic analysis is performed on a running program.

Dynamic analysis tools are used for measuring test coverage, since lines of code that are not executed consume no resources. The verification of test coverage, i.e. that all statements have been executed during testing, is a requirement of ESA PSS-05-0. Test coverage is best measured automatically.

Dynamic analysis enables a variety of optimisations to be carried out to tune' the system. Optimisation can be a very inefficient process without detailed knowledge of the system performance. Dynamic analysis tools can locate the parts of the system that are causing performance problems. Source-level modifications can often yield the desired performance gains.

Where there are precise performance requirements, developers should use a dynamic analyser to verify that the system is satisfactorily tuned, and to direct optimisation effort. If there are no precise performance requirements, and a system runs satisfactorily, dynamic analysis can still help detect some types of coding errors (e.g. unnecessary initialisations).

If a dynamic analyser is not available, resource consumption can be measured by means of timing routines. An interactive debugger can be used to observe coverage.

4.3.10 Test tools

Test tools may support one or more of the following functions:

- test data generation;
- test driving and harnessing;
- automated results checking;
- fault diagnosis and debugging;
- test data management.

General purpose test tools can sometimes generate large quantities of input based on simple assumptions. System-specific simulators and modelling tools are needed if realistic test data is required.

Test drivers and harnesses normally have to be provided by the developers. The integration plan should identify what test drivers and stubs are required for testing, and these normally have to be specially developed.

Interpreter systems permit test drivers and harnesses to be created interactively. Short test drivers, consisting of little more than such a call, can be prepared and called as interpreted commands. Such test drivers do not need to be compiled into the final version. Of the languages mentioned in Section 3.3.1, Pascal, ML, Prolog, LISP and Smalltalk are known to be runnable interpretatively.

Tools that provide automated results checking can greatly increase the efficiency of regression testing, and thereby its depth and scope.

If the comparison reveals a discrepancy between past and present results, support for fault diagnosis, via tracebacks of the execution, reports of the contents of variables, can ease the solution of the problem identified in the test.

Perhaps of all the functions, support for the management of test data is the most important. Management of the input and output test data is required if regression testing is to be meaningful.

4.3.11 Word processors

A word processor or text processor should be used. Tools for the creation of paragraphs, sections, headers, footers, tables of contents and indexes all ease the production of a document. A spelling checker is essential, and a grammar checker is desirable. An outliner may be found useful for creation of sub-headings, for viewing the document at different levels of detail and for rearranging the document. The ability to handle diagrams is very important.

Documents invariably go through many drafts as they are created, reviewed and modified. Revised drafts should include change bars. Document comparison programs, which can mark changed text automatically, are invaluable for easing the review process.

Tools for communal preparation of documents are now beginning to be available, allowing many authors to comment and add to a single document.

4.3.12 Documentation generators

Documentation generators allow the automatic production of help and documentation from the information in the code. They help maintain consistency between code and documentation, and make the process of documentation truly concurrent with the coding.

Code generators (see Section 4.3.2) may include tools for automatically generating documentation about the screens, windows and reports that the programmer creates.

4.3.13 Configuration managers

Configuration management is covered in ESA PSS-05-09, Guide to Software Configuration Management'. Implementers should consider the use of an automated configuration manager in the circumstances of each project.

A variety of tools, often centred on a database, is available to assist in controlling the development of software when many modules may exist in many versions. Some tools allow the developer to specify a configuration (m modules in n versions), and then automatically compile, link, and archive it. The use of configuration management tools becomes essential when the number of modules or versions becomes large.

CHAPTER 5
THE DETAILED DESIGN DOCUMENT

5.1 INTRODUCTION

The purpose of a DDD is to describe the detailed solution to the problem stated in the SRD. The DDD must be an output of the DD phase (DD13). The DDD must be complete, accounting for all the software requirements in the SRD (DD15). The DDD should be sufficiently detailed to allow the code to be implemented and maintained. Components (especially interfaces) should be described in sufficient detail to be fully understood.

5.2 STYLE

The style of a DDD should be systematic and rigorous. The language and diagrams used in a DDD should be clear and constructed to a consistent plan. The document as a whole must be modifiable.

5.2.1 Clarity

A DDD is clear if it is easy to understand. The structure of the DDD must reflect the structure of the software design, in terms of the levels and components of the software (DD14). The natural language used in a DDD must be shared by all the development team.

The DDD should not introduce ambiguity. Terms should be used accurately.

A diagram is clear if it is constructed from consistently used symbols, icons, or labels, and is well arranged. Important visual principles are to:

- emphasise important information;
- align symbols regularly;
- allow diagrams to be read left-to-right or top-to-bottom;
- arrange similar items in a row, in the same style;
- exploit visual symmetry to express functional symmetry;
- avoid crossing lines and overlaps;
- avoid crowding.

Diagrams should have a brief title, and be referenced by the text which they illustrate.

Diagrams and text should complement one another and be as closely integrated as possible. The purpose of each diagram should be explained in the text, and each diagram should explain aspects that cannot be expressed in a few words. Diagrams can be used to structure the discussion in the text.

5.2.2 Consistency

The DDD must be consistent. There are several types of inconsistency:

- different terms used for the same thing;
- the same term used for different things;
- incompatible activities happening simultaneously;
- activities happening in the wrong order.

Where a term could have multiple meanings, a single meaning should be defined in a glossary, and only that meaning should be used in the DDD.

Duplication and overlap lead to inconsistency. Clues to inconsistency are a single functional requirement tracing to more than one component. Methods and tools help consistency to be achieved.

Consistency should be preserved both within diagrams and between diagrams in the same document. Diagrams of different kinds should be immediately distinguishable.

5.2.3 Modifiability

A DDD is modifiable if changes to the document can be made easily, completely, and consistently. Good tools make modification easier, although it is always necessary to check for unpredictable side-effects of changes. For example a global string search and replace capability can be very useful, but developers should always guard against unintended changes.

Diagrams, tables, spreadsheets, charts and graphs are modifiable if they are held in a form which can readily be changed. Such items should be prepared either within the word processor, or by a tool compatible with the

word processor. For example, diagrams may be imported automatically into a document: typically, the print process scans the document for symbolic markers indicating graphics and other files.

Where graphics or other data are prepared on the same hardware as the code, it may be necessary to import them by other means. For example, a screen capture utility may create bitmap files ready for printing. These may be numbered and included as an annex. Projects using methods of this kind should define conventions for handling and configuration management of such data.

5.3 EVOLUTION

The DDD should be put under change control by the developer at the start of the Transfer Phase. New components may need to be added and old components modified or deleted. If the DDD is being developed by a team of people, the control of the document may be started at the beginning of the DD phase.

The Software Configuration Management Plan defines a formal change process to identify, control, track and report projected changes, as soon as they are first identified. Approved changes in components must be recorded in the DDD by inserting document change records and a document status sheet at the start of the DDD.

5.4 RESPONSIBILITY

Whoever writes the DDD, the responsibility for it lies with the developer. The developer should nominate people with proven design and implementation skills to write the DDD.

5.5 MEDIUM

The DDD is usually a paper document. The DDD may be distributed electronically when participants have access to the necessary equipment.

5.6 CONTENT

The DDD is the authoritative reference document on how the software works. Part 2 of the DDD must have the same structure and identification scheme as the code itself, with a 1:1 correspondence between

sections of the documentation and the software components (DD14). The DDD must be complete, accounting for all the software requirements in the SRD (DD15).

The DDD should be compiled according to the table of contents provided in Appendix C of ESA PSS-05-0. This table of contents is derived from ANSI/IEEE Std 1016-1987 'IEEE Recommended Practice for Software Design Descriptions' [Ref. 5]. This standard defines a Software Design Description as a 'representation or model of the software system to be created. The model should provide the precise design information needed for the planning, analysis and implementation of the software system'. The DDD should be such a Software Design Description.

The table of contents is reproduced below. Relevant material unsuitable for inclusion in the contents list should be inserted in additional appendices. If there is no material for a section then the phrase 'Not Applicable' should be inserted and the section numbering preserved.

Service Information:
a - Abstract
b - Table of Contents
c - Document Status Sheet
d - Document Change Records made since last issue

Part 1 - General Description

1 Introduction
- 1.1 Purpose
- 1.2 Scope
- 1.3 Definitions, acronyms and abbreviations
- 1.4 References
- 1.5 Overview

2 Project Standards, Conventions and Procedures
- 2.1 Design standards
- 2.2 Documentation standards
- 2.3 Naming conventions
- 2.4 Programming standards
- 2.5 Software development tools

Part 2 - Component Design Specifications

n [Component identifier]

n.1 Type
n.2 Purpose
n.3 Function
n.4 Subordinates
n.5 Dependencies
n.6 Interfaces
n.7 Resources
n.8 References
n.9 Processing
n.10 Data

Appendix A Source code listings
Appendix B Software Requirements vs Components Traceability matrix

A component may belong to a class of components that share characteristics. To avoid repeatedly describing shared characteristics, a sensible approach is to reference the description of the class.

References should be given where appropriate, but a DDD should not refer to documents that follow it in the ESA PSS-05-0 life cycle.

Part 1 of the DDD must be completed before any coding is started. The component design specification in Part 2 must be complete (i.e. no TBCs or TBDs) before coding is started.

5.6.1 DDD/Part 1 - General description

5.6.1.1 DDD/Part 1/1 Introduction

This section should describe the purpose and scope, and provide a glossary, list of references and document overview.

5.6.1.1.1 DDD/Part 1/1.1 Purpose (of the document)

This section should:

(1) describe the purpose of the particular DDD;

(2) specify the intended readership of the DDD.

5.6.1.1.2 DDD/Part 1/1.2 Scope (of the software)

This section should:

(1) identify the software products to be produced;

(2) explain what the proposed software will do (and will not do, if necessary);

(3) define relevant benefits, objectives, and goals as precisely as possible;

(4) be consistent with similar statements in higher-level specifications, if they exist.

5.6.1.1.3 DDD/Part 1/1.3 Definitions, acronyms and abbreviations

This section should define all terms, acronyms, and abbreviations used in the DDD, or refer to other documents where the definitions can be found.

5.6.1.1.4 DDD/Part 1/1.4 References

This section should list all the applicable and reference documents, identified by title, author and date. Each document should be marked as applicable or reference. If appropriate, report number, journal name and publishing organisation should be included.

5.6.1.1.5 DDD/Part 1/1.5 Overview (of the document)

This section should:

(1) describe what the rest of the DDD contains;

(2) explain how the DDD is organised.

5.6.1.2 DDD/Part 1/2 Project standards, conventions and procedures

5.6.1.2.1 DDD/Part 1/2.1 Design standards

These should usually reference methods carried over from the AD phase and only describe DD phase specific methods.

The Detailed Design Standard might need to be different if more than one method or programming language is involved: for example, if some C language design and programming takes place in an Ada project.

5.6.1.2.2 DDD/Part 1/2.2 Documentation standards

This section should describe the format, style, and tools adopted by the project for DD and code documentation. Headers, footers, section formats and typefaces should be specified. They may be prepared as word processor template files, for automatic inclusion in all project documents. If the formats are new, they should be prototyped and reviewed (with users in the case of the SUM).

This section should contain the standard module header and contain instructions for its completion (see Section 2.3.9).

5.6.1.2.3 DDD/Part 1/2.3 Naming conventions

This section should explain all naming conventions used, and draw attention to any points a maintenance programmer would not expect. A table of the file types and the permitted names or extensions for each is recommended for quick reference. See the examples in Table 5.6.1.2.3 and Section 5.6.2.1.

File Type	Name	Extension
document	< <mnemonic> >	.DOC
Ada source code	IDENTIFIER	.ADA
Fortran source code	IDENTIFIER	.FOR
diagram	< <mnemonic> >	.PIC

Table 5.6.1.2.3: Names and extensions

Conventions for naming files, programs, modules, and possibly other structures such as variables and messages, should all be documented here.

5.6.1.2.4 DDD/Part 1/2.4 Programming standards

This section should define the project programming standards. Whatever languages or standards are chosen, the aim should be to create a convenient and easily usable method for writing good-quality software. Note especially the guidelines in Section 2.3.

If the programming language is described in an ESA PSS-05 level 3 Guide, then the guidelines described in that document should be adopted.

Additional restrictions on the use of the language may be imposed if necessary: such additions should be justified here.

When programming in any other language, a standard for its use should be written to provide guidance for programmers. This standard may be referenced or included here.

In general, the programming standard should define a consistent and uniform programming style. Specific points to cover are:

- modularity and structuring;
- headers and commenting;
- indenting and layout;
- library routines to be used;
- language constructs to use;
- language constructs to avoid.

5.6.1.2.5 DDD/Part 1/2.5 Software development tools

This section should list the tools chosen to assist software development. Normally the list will include:

- a CASE tool;
- a source code editor;
- a compiler;
- a debugger;
- a linker;
- a configuration manager / builder;
- a word processor for documentation;
- a tool for drawing diagrams.

Many projects will also use a configuration management system to store configuration items, such as documentation, code and test data.

Prototyping projects might make use of an interpretative tool, such as an incremental compiler/interpreter/debugger.

Other tools that may be helpful to many projects include:

- static analysers;

- dynamic analysers;
- network and data communication tools;
- graphics packages;
- statistics and mathematical packages.

Real-time and embedded systems may require special development tools, including:

- cross-compilers;
- diagnostic, logging, and probe tools.

5.6.2 DDD/Part 2 - Component design specifications

The descriptions of the components should be laid out hierarchically. There should be subsections dealing with the following aspects of each component:

n Component identifier

n.1 Type
n.2 Purpose
n.3 Function
n.4 Subordinates
n.5 Dependencies
n.6 Interfaces
n.7 Resources
n.8 References
n.9 Processing
n.10 Data

The number 'n' should relate to the place of the component in the hierarchy.

5.6.2.1 DDD/Part 2/5.n Component Identifier

Each component should have a unique identifier (SCM06) for effective configuration management. The component should be named according to the rules of the programming language or operating system to be used. Where possible, a hierarchical naming scheme should be used that identifies the parent of the component (e.g. ParentName_ChildName)

The identifier should reflect the purpose and function of the component and be brief yet meaningful. If abbreviation is necessary, abbreviations should be applied consistently and without ambiguity.

Abbreviations should be documented. Component identifiers should be mutually consistent (e.g. if there is a routine called READ_RECORD then one might expect a routine called WRITE_RECORD, not RECORD_WRITING_ROUTINE).

A naming style that clearly distinguishes objects of different classes is good programming practice. In Pascal, for instance, it is traditional to use upper case for user-defined types, mixed case for modules, and lower case for variables, giving the following appearance:

- procedure Count_Chars; {a module}
- type SMALL_INT = 1..255; {a type}
- var count: SMALL_INT; {a variable}

Other styles may be appropriate in other languages. The naming style should be consistent throughout a project. It is wise to avoid styles that might confuse maintenance programmers accustomed to standard industrial practices.

5.6.2.1.1 DDD/Part 2/5.n.1 Type

Component type should be defined by stating its logical and physical characteristics. The logical characteristics should be defined by stating the package, library or class that the component belongs to. The physical characteristics should be defined by stating the type of component, using the implementation terminology (e.g. task, subroutine, subprogram, package, file).

The contents of some component-description sections depend on the component type. For this guide the categories: executable (i.e. contains computer instructions) or non-executable (i.e. contains only data) are used.

5.6.2.1.2 DDD/Part 2/5.n.2 Purpose

The purpose of a component should be defined by tracing it to the software requirements that it implements.

Backward traceability depends upon each component description explicitly referencing the requirements that justify its existence.

5.6.2.1.3 DDD/Part 2/5.n.3 Function

The function of a component must be defined in the DDD. This should be done by stating what the component does.

The function description depends upon the component type. Therefore it may be a description of the:

- process;
- information stored or transmitted.

Process descriptions may use such techniques as Structured English, Precondition-Postcondition specifications and State-Transition Diagrams.

5.6.2.1.4 DDD/Part 2/5.n.4 Subordinates

The subordinates of a component should be defined by listing the immediate children. The subordinates of a program are the subroutines that are 'called by' it. The subordinates of a database could be the files that 'compose' it. The subordinates of an object are the objects that are 'used by' it.

5.6.2.1.5 DDD/Part 2/5.n.5 Dependencies

The dependencies of a component should be defined by listing the constraints placed upon its use by other components. For example:

- 'what operations must have taken place before this component is called?'
- 'what operations are excluded while this operation is taking place?'
- 'what operations have to be carried out after this one?'.

5.6.2.1.6 DDD/Part 2/5.n.6 Interfaces

Both control flow and data flow aspects of each interface need to be specified in the DDD for each 'executable' component. Data aspects of 'non-executable' components should be defined in Subsection 10.

The control flow to and from a component should be defined in terms of how execution of the component is to be started (e.g. subroutine call) and how it is to be terminated (e.g. return). This may be implicit in the definition of the type of component, and a description may not be

necessary. Control flows may also take place during execution (e.g. interrupt) and these should be defined, if they exist.

The data flow input to and output from each component must be detailed in the DDD. Data structures should be identified that:

- are associated with the control flow (e.g. call argument list);
- interface components through common data areas and files.

One component's input may be another's output and to avoid duplication of interface definitions, specific data components should be defined and described separately (e.g. files, messages). The interface definition should only identify the data component and not define its contents.

The interfaces of a component should be defined by explaining 'how' the component interacts with the components that use it. This can be done by describing the mechanisms for:

- invoking or interrupting the component's execution;
- communicating through parameters, common data areas or messages;

If a component interfaces to components in the same system, the interface description should be defined in the DDD (if not already in the ADD). If a component interfaces to components in other systems, the interface description should be defined in an Interface Control Document (ICD).

5.6.2.1.7 DDD/Part 2/5.n.7 Resources

The resources a component requires should be defined by itemising what the component needs from its environment to perform its function. Items that are part of the component interface are excluded. Examples of resources that might be needed by a component are displays, printers and buffers.

5.6.2.1.8 DDD/Part 2/5.n.8 References

Explicit references should be inserted where a component description uses or implies material from another document.

5.6.2.1.9 DDD/Part 2/5.n.9 Processing

The DDD should describe in detail how processing is carried out. Algorithms that are fully described elsewhere may be summarised briefly, provided their sources are properly referenced.

The processing a component needs to do should be defined by defining the control and data flow within it. For some kinds of component (e.g. files) there is no such flow. In practice it is often difficult to separate the description of function from the description of processing. Therefore a detailed description of function can compensate for a lack of detail in the specification of the processing. Techniques of process specification more oriented towards software design are Program Design Language, Pseudo-code and Flow Charts.

Software constraints may specify that the processing be performed using a particular algorithm (which should be stated or referenced).

5.6.2.1.10 DDD/Part 2/5.n.10 Data

The data internal to a component should be defined. The amount of detail required depends strongly on the type of component. The logical and physical data structure of files that interface components should have been defined in the DDD (files and data structures that interface major components will have been defined in the ADD). The data structures internal to a program or subroutine should also be specified (contrast the ADD, where it is omitted).

Data structure definitions must include the:

- description of each element (e.g. name, type, dimension);
- relationships between the elements (i.e. the structure);
- range of possible values of each element;
- initial values of each element.

5.6.3 DDD/Appendix A: Source code listings

This section must contain either:

- listings of the source code, or
- a configuration item list identifying where the source code can be found (DD12).

5.6.4 DDD/Appendix B: Software requirements traceability matrix

This section should contain a table that summarises how each software requirement has been met in the DDD (DD16). The tabular format permits one-to-one and one-to-many relationships to be shown. A template is provided in Appendix D.

CHAPTER 6
THE SOFTWARE USER MANUAL

6.1 INTRODUCTION

The purpose of the Software User Manual (SUM) is to describe how to use the software. It should both educate users about what the software does, and instruct them how to do it. The SUM should combine tutorials and reference information to meet the needs of both novices and experts. A Software User Manual (SUM) must be an output of the DD phase (DD17).

The rules for the style and content of the Software User Manual are based on ANSI/IEEE Std 1063-1987, 'Software User Documentation'.

6.2 STYLE

The author of the SUM needs to be aware of the basic rules of clear writing:

- keep sentences short (e.g. 20 words or less);
- avoid using long words (e.g. 10 letters or more);
- keep paragraphs short (e.g. 10 sentences or less);
- avoid using the passive tense;
- use correct grammar;
- make each paragraph express one point;
- tell the reader what you going to say, say it, and then tell the reader what you have said.

A grammar checker can help the author obey the first five rules.

The concepts of clarity, consistency and modifiability apply to the SUM just as much as to the DDD (see Section 5.2). The rest of this section presents some other considerations that should influence the style of the SUM.

The SUM should reflect the characteristics of the users. Different types of users, e.g. end users and operators, will have different needs. Further, the user's view may change from that of a trainee to expert. The

authors of the SUM should study the characteristics of the users. The URD normally contains a summary.

The SUM should reflect the characteristics of the interface between he system and the user. Embedded software for controlling an instrument on board a satellite might have no interface to humans. At the opposite extreme, a data reduction package might require a large amount of user interaction, and the SUM might be absolutely critical to the success of the package.

6.2.1 Matching the style to the user characteristics

Different viewpoints that may need to be addressed are:

- the end user's view;
- the operator's view;
- the trainee's view.

An individual user may hold more than one view.

6.2.1.1 The end user's view

The end user's view takes in every aspect of the use of the system. The SUM should describe the purpose of the system and should reference the URD.

The SUM should contain an overview of the process to be supported, perhaps making reference to technical information, such as the underlying physics and algorithms employed. Information should be presented from the end user's view, not the developer's. Algorithms should be represented in mathematical form, not in a programming language, for example. Descriptions should not stray into the workings of the system; they are in the SUM only to help the end user understand the intention of the system.

The end user's view is an external view, unconcerned with details of the implementation. Accordingly, the SUM should give an external view of the system, with examples of input required from end users, and the results that would occur (e.g. output). The SUM should place the system in its working context.

6.2.1.2 The operator's view

The operator's view focuses on successfully controlling the software, whether in normal conditions, or when recovering from errors.

From the point of view of the operator, the SUM should contain clear instructions about each task that has to be performed. In particular:

- instructions should be easy to find;
- instructions should be easy to understand.

Ease of location and understanding are important in emergencies. Instructions will be easy to find and understand if a regular structure is adopted for the description of each task.

Writers should ensure that wrong actions are not accidentally selected, and that correct or undo actions are possible if the operator makes a mistake. The SUM should document such corrective actions with examples.

Any irreversible actions must be clearly distinguished in the SUM (and in the system). Typically, the operator will be asked to confirm that an irreversible action is to be selected. All confirmation procedures must be explained with examples in the SUM.

6.2.1.3 The trainee's view

A particularly important view is that of the trainee. There are always users who are not familiar with a system, and they are especially likely to need the support of a well-written guide or tutorial. Since the first experience with a system is often decisive, it is just as vital to write a helpful tutorial for the trainee.

Every SUM should contain a tutorial section, which should provide a welcoming introduction to the software (e.g. 'Getting Started').

6.2.2 Matching the style to the HCI characteristics

The SUM should reflect the Human Computer Interaction (HCI) characteristics of the software. This section presents some guidelines for:

- command-driven systems;
- menu-driven systems;
- GUI, WIMPs and WYSIWYG systems;

- natural language and speech interface systems;
- embedded systems.

6.2.2.1 Command-driven systems

Command-driven systems require the user to type in commands. The keyboard input is echoed on a screen. Command driven systems are simple to implement, and are often preferred by expert users.

Command-driven systems can take some time to learn, and the SUM has a critical role to play in educating the users of the software. A well thought-out SUM can compensate for a difficult user interface.

The alphanumeric nature of the interaction can be directly represented in the SUM. For example:

Commentary and instructions,

```
> user input

system output
```

should be clearly distinguished from each other.

6.2.2.2 Menu-driven systems

Traditional menu-driven systems present the user with a fixed hierarchy of menus. The user starts at the top of the tree and moves up and down the tree. The structure of the menus normally follows the natural sequence of operations (e.g. OPEN the file first, then EDIT it). Often the left-most or the top-most item on the menu is what is usually done first. Sometimes this temporal logic is abandoned in favour of ergonomic considerations (e.g. putting the most frequently used command first).

The SUM should describe the standard paths for traversing the menus. The structure of the SUM should follow the natural sequence of operations, which is normally the menu structure. An index should be provided to give alphabetical access.

6.2.2.3 Graphical User Interface systems

Graphical User Interfaces (GUI) include Windows, Icons, Mouse and Pointer (WIMP) and What You See Is What You Get (WYSIWYG) style interfaces. An aim of their inventors was to make the operation of a system

so simple and intuitive that the reading of a user manual is unnecessary. Unfortunately, making information about the system only available through the system can be very restrictive, and may not support the mode of inquiry of the user. SUMs separate to the software are always required, no matter how sophisticated the interface.

SUMs for a GUI should not assume basic GUI skills. They should instruct the user how to operate the interface and should describe what the user looks at and feels'. Pictures and diagrams of the behaviour should be given, so that the reader is left in no doubt about what the intended behaviour is. This is especially important when the user does not have access to the system when reading the tutorial.

6.2.2.4 Natural-language and speech interface systems

Natural-language and speech interfaces are coming into use. Unlike menu-driven systems and WIMPs, WYSIWYG and GUI type systems, the options are not normally defined on a screen. There may be many options.

For natural language and speech interface systems, the SUM should provide:

- a full list of facilities, explaining their purposes and use;
- examples showing how the facilities relate to each other;
- a list of useful command verbs and auxiliary words for each facility;
- a clear statement of types of sentence that are NOT recognised.

6.2.2.5 Embedded systems

Software that is embedded within a system may not require any human interaction at the software level. Nevertheless a SUM must be supplied with the software and provide at least:

- an overview;
- error messages;
- recovery procedures.

6.3 EVOLUTION

The development of the SUM should start as early as possible. Establishing the potential readership for the SUM should be the first step.

This information is critical for establishing the style of the document. Useful information may be found in the section 'User Characteristics' in the URD.

The Software Configuration Management Plan defines a formal change process to identify, control, track and report projected changes when they are first identified. Approved changes to the SUM must be recorded by inserting document change records and a document status sheet at the start of the document.

The SUM is an integral part of the software. The SUM and the rest of the software must evolve in step. The completeness and accuracy of the SUM should be reviewed whenever a new release of the software is made.

6.4 RESPONSIBILITY

Whoever writes the SUM, the responsibility for it must be the developer's. The developer should nominate people with proven authorship skills to write the SUM.

6.5 MEDIUM

Traditionally, manuals have been printed as books. There are substantial advantages to be gained from issuing manuals in machine-readable form. Implementers should consider whether their system would benefit from such treatment. Two possibilities are:

- online help;
- hypertext.

There should be specific requirements in the URD and SRD for online help and hypertext, since their implementation can absorb both computer resources (e.g. storage) and development effort.

6.5.1 Online help

Information in the SUM may be used to construct an 'online help' system. Online help has the advantages of always being available with the system, whereas paper manuals may be mislaid. Three possibilities, in order of increasing desirability are:

- a standalone help system, not accessible from inside the
- application;

- an integrated help system, accessible from within the application;
- an integrated, context-sensitive help system, that gives the help information about the task being carried out.

In the last option, the user should be able to continue the search for help outside the current context (e.g. to get background information).

6.5.2 Hypertext

A hypertext is a structure of pages of text or graphics (and sometimes of other media). Pages are linked by keywords or phrases. For example, the word 'file' might give access to a page containing the words 'store', 'search', and 'retrieve'; in turn, each of these might give access to a page explaining a command within the documented system. Keywords may be displayed as text (usually highlighted), or may be indicated by icons or regions of a diagram which respond to a pointing device (such as a mouse).

Hypertext readily accommodates the hierarchical structure of most programs, but it can equally represent a network of topics linked by arbitrary connections.

This gives hypertext the advantage of a richer pattern of access than printed text, which is inherently linear (despite hypertext-like helps, such as indexes).

Hypertext is also good for guided tours, which simulate the operation of the program, while pointing out interesting features and setting exercises.

The quality of graphics within a hypertext must enable tables, charts, graphs and diagrams to be read quickly and without strain. Older terminals and personal computers may not be suitable for this purpose. Modern bit-mapped workstations, and personal computers with higher-resolution graphics cards, rarely cause legibility problems.

Stand-alone hypertext has the advantage of being simple and safe; it does not increase the complexity or risk of system software, because it is not connected directly to it.

A disadvantage of stand-alone hypertext is that it is not automatically context-sensitive. On a multitasking operating system, it can be made context-sensitive by issuing a message from the (documented)

system to the hypertext mechanism, naming the entry point (for example, file names' or 'data display').

Reliable proprietary hypertext packages are available on many processors; alternatively, simple hypertext mechanisms can be implemented by means of a database, a specialised tool, or if necessary a programming language.

6.6 CONTENT

The recommended table of contents for a SUM is given below. If it is felt necessary to depart from this structure, implementers should justify any differences briefly, and if possible preserve the standard section numbering for easy reference.

Service Information:
- a - Abstract
- b - Table of Contents
- c - Document Status Sheet
- d - Document Change Records made since last issue.

1 INTRODUCTION
- 1.1 Intended readership
- 1.2 Applicability statement
- 1.3 Purpose
- 1.4 How to use this document
- 1.5 Related documents (including applicable documents)
- 1.6 Conventions
- 1.7 Problem reporting instructions

2 [OVERVIEW SECTION]

(The section ought to give the user a general understanding of what parts of software provide the capabilities needed)

3 [INSTRUCTION SECTION]

(For each operation, provide...

- (a) Functional description
- (b) Cautions and warnings
- (c) Procedures, including,
  - Set-up and initialisation

- Input operations
- What results to expect

(d) Probable errors and possible causes)

4 [REFERENCE SECTION]

(Describe each operation, including:

(a) Functional description
(b) Cautions and warnings
(c) Formal description, including as appropriate:
- required parameters
- optional parameters
- default options
- order and syntax

(d) Examples
(e) Possible error messages and causes
(f) Cross references to other operations)

Appendix A Error messages and recovery procedures
Appendix B Glossary
Appendix C Index (for manuals of 40 pages or more)

In the instruction section of the SUM, material is ordered according to the learning path, with the simplest, most necessary operations appearing first and more advanced, complicated operations appearing later. The size of this section depends on the intended readership. Some users may understand the software after a few examples (and can switch to using the reference section) while other users may require many worked examples.

The reference section of the SUM presents the basic operations, ordered for easy reference (e.g. alphabetically). Reference documentation should be more formal, rigorous and exhaustive than the instructional section. For example a command may be described in the instruction section in concrete terms, with a specific worked example. The description in the reference section should describe all the parameters, qualifiers and keywords, with several examples.

The overview, instruction, and reference sections should be given appropriate names by the SUM authors. For example, an orbit modelling system might have sections separately bound and titled:

- Orbit Modelling System Overview;
- Orbit Modelling System Tutorial;

- Orbit Modelling System Reference Manual.

At lower levels within the SUM, authors are free to define a structure suited to the readership and subject matter. Particular attention should be paid to:

- ordering subjects (e.g. commands, procedures);
- providing visual aids.

'Alphabetical' ordering of subjects acts as a built-in index and speeds access. It has the disadvantage of separating related items, such as 'INSERT' and 'RETRIEVE'. Implementers should consider which method of structuring is most appropriate to an individual document. Where a body of information can be accessed in several ways, it should be indexed and cross-referenced to facilitate the major access methods.

Another possible method is 'procedural' or 'step-by-step' ordering, in which subjects are described in the order the user will execute them. This style is appropriate for the instruction section.

Visual aids in the form of diagrams, graphs, charts, tables, screen dumps and photographs should be given wherever they materially assist the user. These may illustrate the software or hardware, procedures to be followed by the user, data, or the structure of the SUM itself.

Implementers may also include illustrations for other purposes. For example, instructional material may contain cartoons to facilitate understanding or to maintain interest.

Icons may be used to mark differing types of section, such as definitions and procedures. All icons should be explained at the start of the SUM.

6.6.1 SUM\Table of Contents

A table of contents is an important help and should be provided in every SUM. Short manuals may have a simple list of sections. Manuals over 40 pages should provide a fuller list, down to at least the third level of subsections (1.2.3, etc). Where the SUM is issued in several volumes, the first volume should contain a simple table of contents for the entire SUM, and each volume should contain its own table of contents.

There is no fixed style for tables of contents, but they should reference page as well as section numbers. Section headings should be

quoted exactly as they appear in the body of the manual. Subsections may be indented to group them visually by containing a section.

A convenient style is to link the title to the page number with a leader string of dots:

1.2.3 Section Title 29

Lists of figures and tables should be provided in manuals with many of them (e.g. more than twenty).

6.6.2 SUM\1 Introduction

6.6.2.1 SUM\1.1 Intended readership

This section should define the categories of user (e.g. end user and operator) and, for each category:

- define the level of experience assumed;
- state which sections of SUM are most relevant to their needs.

6.6.2.2 SUM\1.2 Applicability statement

This section should define the software releases that the issue of the SUM applies to.

6.6.2.3 SUM\1.3 Purpose

This section should define both the purpose of the SUM and the purpose of the software. It should name the process to be supported by software and the role of the SUM in supporting that process.

6.6.2.4 SUM\1.4 How to use this document

This section should describe what each section of the document contains, its intended use, and the relationship between sections.

6.6.2.5 SUM\1.5 Related documents

This section should list related documents and define the relationship of each document to the others. All document trees that the SUM belongs to should be defined in this section. If the SUM is a multivolume set, each member of the set should be separately identified.

6.6.2.6 SUM\1.6 Conventions

This section should summarise symbols, stylistic conventions, and command syntax conventions used in the document.

Examples of stylistic conventions are boldface and Courier Font to distinguish user input. Examples of syntax conventions are the rules for combining commands, keywords and parameters, or how to operate a WIMP interface.

6.6.2.7 SUM\1.7 Problem Reporting Instructions

This section should summarise the procedures for reporting software problems. ESA PSS-05-0 specifies the problem reporting procedure [Ref. 1, part 2, section 3.2.3.2.2]. The SUM should not refer users to ESA PSS-05-0.

6.6.3 SUM\2 Overview Section

This section should give an overview of the software to all users and summarises:

- the process to be supported to by software;
- the fundamental principles of the process;
- what the software does to support the process;
- what the user needs to supply to the software.

The description of the process to be supported, and its fundamental principles, may be derived from the URD. It should not use software terminology.

The software should be described from an external 'black box' point of view. The discussion should be limited to functions, inputs and outputs that the user sees.

The overview can often become much clearer if a good metaphor is used for the system. Some GUIs, for example, use the 'office system' metaphor of desk, filing cabinets, folders, indexes etc. Often a metaphor will have been defined very early in the system development, to help capture the requirements. The same metaphor can be very useful in explaining the system to new users.

6.6.4 SUM\3 Instruction Section

This section should aim to teach new users how to operate the software. The viewpoint is that of the trainee.

The section should begin by taking the user through short and simple sessions. Each session should have a single purpose, such as to enable the user to edit a data description record'. The sessions should be designed to accompany actual use of the system, and should contain explicit references to the user's actions and the system's behaviour. The trainee should derive a sense of achievement by controlling a practical session that achieves a result.

Diagrams, plans, tables, drawings, and other illustrations should be used to show what the software is doing, and to enable the novice to form a clear and accurate mental model of the system.

The tutorial may be structured in any convenient style. It is wise to divide it into sessions lasting 30-40 minutes. Longer sessions are difficult to absorb.

A session could have the goal of introducing the user to a set of 'advanced' facilities in a system. It may not be practical or desirable to present a whole session. It may be more appropriate to give a tutorial tour'. Even so, it is still helpful to fit all the examples into a single framework (e.g. how to store, search and retrieve).

For each session, this section should provide:

(a) Functional description

A description of what the session is supposed to achieve, in the user's terms.

(b) Cautions and warnings

A list of precautions that may need to be taken; the user is not concerned with all possibilities at this stage, but with gaining a working understanding of some aspect of the system.

(c) Procedures

- set-up and initialisation operations

A description of how to prepare for and start the task;

- input operations

A step-by-step description what the user must do and a description of the screens or windows that the system shows in response;

- what results to expect

A description of the final results that are expected.

(d) Likely errors and possible causes

An informal description of the major errors that are possible in this task, and how to avoid them. The aim is to give the trainee user the confidence to carry on when problems arise. This section should not simply provide a list of errors, but should set them in their context.

6.6.5 SUM\4 Reference Section

This section should give comprehensive information about all the software capabilities. The viewpoint is that of the operator or expert user.

The section should contain a list of operations ordered for easy access. The order may be alphabetical (e.g. for command-driven systems) or correspond directly to the structure of the user interface (e.g. for menu-driven systems).

In contrast to the informal style of the instruction section, the reference section should be formal, rigorous and exhaustive.

For each operation, this section should provide:

(a) Functional description

A concise description of what the operation achieves.

(b) Cautions and warnings

A list of cautions and warnings that apply to the operation.

(c) Formal description, including as appropriate:

- required parameters
- optional parameters
- defaults
- syntax & semantics

Describe precisely what the operation does and how it is used. The means of doing this should be decided in advance. Syntax may be defined formally using the Backus-Naur Form (BNF). Semantics may be described by means of tables, diagrams, equations, or formal language.

(d) Examples

Give one or more worked examples to enable operators to understand the format at a glance. Commands should be illustrated with several short examples, giving the exact form of the most common permutations of parameters and qualifiers in full, and stating what they achieve.

(e) Possible errors and their causes

List all the errors that are possible for this operation and state what causes each one. If error numbers are used, explain what each one means.

(f) References to related operations

Give references to other operations which the operator may need to complete a task, and to logically related operations (e.g. refer to RETRIEVE and DELETE when describing the INSERT operation).

The operations should be described in a convenient order for quick reference: for example, alphabetically, or in functionally related groups. If the section is separately bound, then it should contain its own tables of error messages, glossary, and index; otherwise they should be provided as appendices to the body of the SUM. These appendices are described below.

6.6.6 SUM\Appendix A - Error messages and recovery procedures

This section should list all the error messages. It should not simply repeat the error message: referral to this section means that the user requires help. For each error message the section should give a diagnosis, and suggest recovery procedures. For example:

file 'TEST.DOC' does not exist.

There is no file called 'TEST.DOC' in the current directory and drive. Check that you are in the correct directory to access this file.

If recovery action is likely to involve loss of data, remind the user about possible backup or archiving procedures.

6.6.7 SUM\Appendix B - Glossary

A glossary should be provided if the manual contains terms that operators are unlikely to know, or that are ambiguous.

Care should be taken to avoid redefining terms that operators usually employ in a different sense in a similar context. References may be provided to fuller definitions of terms, either in the SUM itself or in other sources.

Pairs of terms with opposite meanings, or groups of terms with associated meanings, should be cross-referenced within the glossary. Cross-references may be indicated with a highlight, such as italic type.

List any words used in other than their plain dictionary sense, and define their specialised meanings.

All the acronyms used in the SUM should be listed with brief explanations of what they mean.

6.6.8 SUM\Appendix C - Index

An index helps to make manuals easier to use. Manuals over 40 pages should have an index, containing a systematic list of topics from the user's viewpoint.

Indexes should contain major synonyms and variants, especially if these are well known to users but are not employed in the SUM for technical reasons. Such entries may point to primary index entries.

Index entries should point to topics in the body of the manual by:

- page number;
- section number;
- illustration number;
- primary index entry (one level of reference only).

Index entries can usefully contain auxiliary information, especially cross-references to contrasting or related terms. For example, the entry for INSERT could say 'see also DELETE'.

Indexes can provide the most help to users if attention is drawn primarily to important keywords, and to important locations in the body of

the manual. This may be achieved by highlighting such entries, and by grouping minor entries under major headings. Indexes should not contain more than two levels of entry.

If a single index points to different kinds of location, such as pages and illustration numbers, these should be unambiguously distinguished, e.g.

- page 35
- figure 7

as the use of highlighting (35, 7) is not sufficient to prevent confusion in this case.

This page is intentionally left blank.

CHAPTER 7
LIFE CYCLE MANAGEMENT ACTIVITIES

7.1 INTRODUCTION

DD phase activities must be carried out according to the plans defined in the AD phase (DD01). These are:

- Software Project Management Plan for the DD phase (SPMP/DD);
- Software Configuration Management Plan for the DD phase (SCMP/DD);
- Software Verification and Validation Plan for the DD phase (SVVP/DD);
- Software Quality Assurance Plan for the DD phase (SQAP/DD).

Progress against plans should be continuously monitored by project management and documented at regular intervals in progress reports.

Plans of TR phase activities must be drawn up in the DD phase. These plans should cover project management, configuration management, quality assurance and acceptance tests.

7.2 PROJECT MANAGEMENT PLAN FOR THE TR PHASE

By the end of the DD review, the TR phase section of the SPMP (SPMP/TR) must be produced (SPM13). The SPMP/TR describes, in detail, the project activities to be carried out in the TR phase.

Guidance on writing the SPMP/TR is provided in ESA PSS-05-08, Guide to Software Project Management.

7.3 CONFIGURATION MANAGEMENT PLAN FOR THE TR PHASE

During the DD phase, the TR phase section of the SCMP (SCMP/TR) must be produced (SCM48). The SCMP/TR must cover the configuration management procedures for deliverables in the operational environment (SCM49).

Guidance on writing the SCMP/TR is provided in ESA PSS-05-09, Guide to Software Configuration Management.

7.4 QUALITY ASSURANCE PLAN FOR THE TR PHASE

During the DD phase, the TR phase section of the SQAP (SQAP/TR) must be produced (SQA10). The SQAP/TR must describe, in detail, the quality assurance activities to be carried out in the TR phase until final acceptance in the OM phase (SQA11).

SQA activities include monitoring the following activities:

- management;
- documentation;
- standards, practices, conventions, and metrics;
- reviews and audits;
- testing activities;
- problem reporting and corrective action;
- tools, techniques and methods;
- code and media control;
- supplier control;
- record collection, maintenance and retention;
- training;
- risk management.

Guidance on writing the SQAP/TR is provided in ESA PSS-05-11, Guide to Software Quality Assurance.

The SQAP/TR should take account of all the software requirements related to quality, in particular:

- quality requirements;
- reliability requirements;
- maintainability requirements;
- safety requirements;
- verification requirements;
- acceptance-testing requirements.

The level of monitoring planned for the TR phase should be appropriate to the requirements and the criticality of the software. Risk analysis should be used to target areas for detailed scrutiny.

7.5 ACCEPTANCE TEST SPECIFICATION

The developer must construct an acceptance test specification in the DD phase and document it in the SVVP (SVV17). The specification should be based on the acceptance test plan produced in the UR phase. This specification should define the acceptance test:

- designs (SVV19);
- cases (SVV20);
- procedures (SVV21).

Guidance on writing the SVVP/AT is provided in ESA PSS-05-10, Guide to Software Verification and Validation.

This page is intentionally left blank.

APPENDIX A
GLOSSARY

Terms used in this document are consistent with ESA PSS-05-0 [Ref. 1] and ANSI/IEEE Std 610.12-1990 [Ref. 2].

A.1 LIST OF ACRONYMS

AD	Architectural Design
ADD	Architectural Design Document
AD/R	Architectural Design Review
ANSI	American National Standards Institute
AT	Acceptance Test
BSI	British Standards Institute
BSSC	Board for Software Standardisation and Control
CASE	Computer Aided Software Engineering
DD	Detailed Design and production
DDD	Detailed Design Document
DD/R	Detailed Design and production Review
ESA	European Space Agency
HCI	Human-Computer Interaction
IEEE	Institute of Electrical and Electronics Engineers
ISO	International Standards Organisation
IT	Integration Test
ICD	Interface Control Document
JSD	Jackson System Development
JSP	Jackson Structured Programming
OOA	Object-Oriented Analysis
OOD	Object-Oriented Design
PA	Product Assurance
PDL	Program Design Language
PERT	Program Evaluation and Review Technique
PSS	Procedures, Specifications and Standards
RID	Review Item Discrepancy
SADT	Structured Analysis and Design Technique
SCM	Software Configuration Management
SCMP	Software Configuration Management Plan
SCR	Software Change Request
SPM	Software Project Management
SPMP	Software Project Management Plan
SPR	Software Problem Report

SQA	Software Quality Assurance
SQAP	Software Quality Assurance Plan
SRD	Software Requirements Document
ST	System Test
SUM	Software User Manual
SVVP	Software Verification and Validation Plan
TBC	To Be Confirmed
TBD	To Be Defined
TR	Transfer
URD	User Requirements Document
UT	Unit Test

APPENDIX B
REFERENCES

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

2. IEEE Standard Glossary for Software Engineering Terminology, ANSI/IEEE Std 610.12-1990.

3. The STARTs Guide - a guide to methods and software tools for the construction of large real-time systems, NCC Publications, 1987.

4. IEEE Standard for Software Reviews and Audits, IEEE Std 1028-1988.

5. IEEE Recommended Practice for Software Design Descriptions, ANSI/IEEE Std 1016-1987.

6. Principles of Program Design, M. Jackson, Academic Press, 1975.

7. Software Engineering with Ada, second edition, G.Booch, Benjamin/Cummings Publishing Company Inc, 1986.

8. Petri Nets, J.L.Petersen, in ACM Computing Surveys, 9(3) Sept 1977.

9. IEEE Recommended Practice for Ada as Program Design Language, ANSI/IEEE Std 990-1987.

10. Structured Programming, O.-J. Dahl, E.W. Dijkstra and C.A.R.Hoare, Academic Press, 1972.

11. T.J.McCabe, A Complexity Measure, IEEE Transactions on Software Engineering, VOL SE-2, No 4, December 1976.

12. Structured Development for Real-Time Systems, P.T.Ward and S.J.Mellor, Yourdon Press, 1985 (Three Volumes).

13. Computer Languages, N.S.Baron, Penguin Books, 1986.

14. Software Reliability, G.J.Myers, Wiley, 1976

15. Object-Oriented Design, P.Coad and E.Yourdon, Prentice-Hall, 1991.

This page is intentionally left blank.

APPENDIX C
MANDATORY PRACTICES

This appendix is repeated from ESA PSS-05-0 appendix D.5.

DD01 DD phase activities shall be carried out according to the plans defined in the AD phase.

The detailed design and production of software shall be based on the following three principles:

DD02
- top-down decomposition;

DD03
- structured programming;

DD04
- concurrent production and documentation.

DD05 The integration process shall be controlled by the software configuration management procedures defined in the SCMP.

DD06 Before a module can be accepted, every statement in a module shall be executed successfully at least once.

DD07 Integration testing shall check that all the data exchanged across an interface agrees with the data structure specifications in the ADD.

DD08 Integration testing shall confirm that the control flows defined in the ADD have been implemented.

DD09 System testing shall verify compliance with system objectives, as stated in the SRD.

DD10 When the design of a major component is finished, a critical design review shall be convened to certify its readiness for implementation.

DD11 After production, the DD Review (DD/R) shall consider the results of the verification activities and decide whether to transfer the software.

DD12 All deliverable code shall be identified in a configuration item list.

DD13 The DDD shall be an output of the DD phase.

DD14 Part 2 of the DDD shall have the same structure and identification scheme as the code itself, with a 1:1 correspondence between sections of the documentation and the software components.

DD15 The DDD shall be complete, accounting for all the software requirements in the SRD.

DD16 A table cross-referencing software requirements to the detailed design components shall be placed in the DDD.

DD17 A Software User Manual (SUM) shall be an output of the DD phase.

This page is intentionally left blank.

APPENDIX D
REQUIREMENTS TRACEABILITY MATRIX

REQUIREMENTS TRACEABILITY MATRIX **DDD TRACED TO SRD**		DATE: <YY-MM-DD> PAGE 1 OF <nn>
PROJECT: <TITLE OF PROJECT>		
SRD IDENTIFIER	DDD IDENTIFIER	SOFTWARE REQUIREMENT

This page is intentionally left blank.

APPENDIX E
INDEX