ESA PSS-05-03 Issue 1 Revision 1
March 1995

Guide to the software requirements definition phase

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-05-03 Guide to the software requirements definition phase**			
2. ISSUE	3. REVISION	4. DATE	5. REASON FOR CHANGE
1	0	1991	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 1st February 1992
Telematics Supervisory Board

Issue 1 approved by:
The Inspector General, ESA

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 requirements definition phase described in the top level document ESA PSS-05-0.
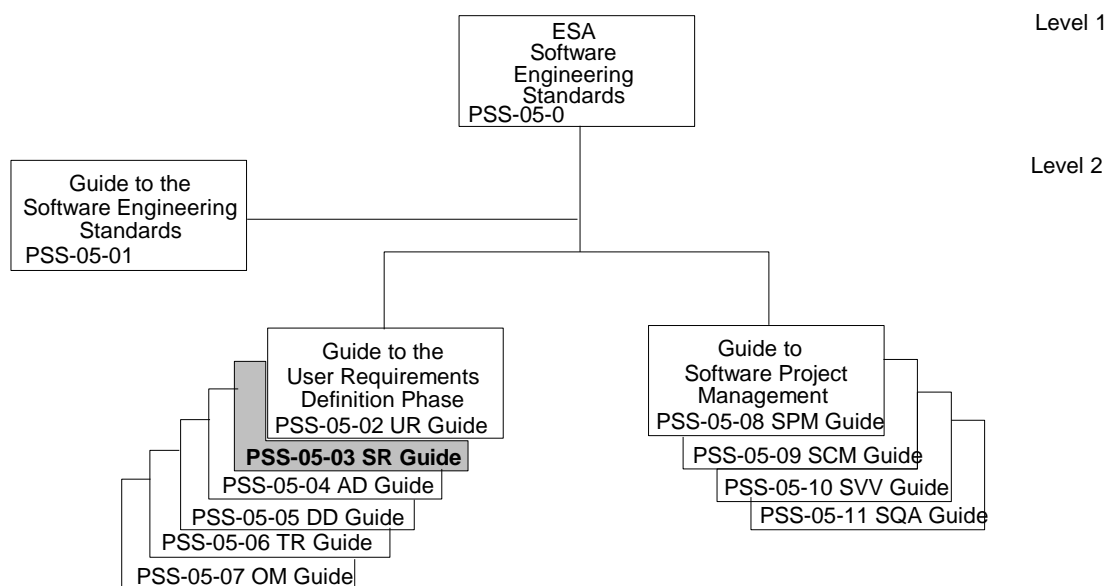
Figure 1: ESA PSS-05-0 document tree

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.

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 a preliminary phase to the software development life cycle called the 'User Requirements Definition Phase' (UR phase). The first phase of the software development life cycle is the 'Software Requirements Definition Phase' (SR phase). Activities and products are examined in the 'SR review' (SR/R) at the end of the phase.

The SR phase can be called the 'problem analysis phase' of the life cycle. The user requirements are analysed and software requirements are produced that must be as complete, consistent and correct as possible.

This document provides guidance on how to produce the software requirements. This document should be read by all active participants in the SR phase, e.g. initiators, user representatives, analysts, designers, project managers and product assurance personnel.

1.2   OVERVIEW

Chapter 2 discusses the SR phase. Chapters 3 and 4 discuss methods and tools for software requirements definition. Chapter 5 describes how to write the SRD, starting from the template. Chapter 6 summarises the life cycle management activities, which are discussed at greater length in other guides.

All the SR phase mandatory practices in ESA PSS-05-0 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 SOFTWARE REQUIREMENTS DEFINITION PHASE

2.1   INTRODUCTION

In the SR phase, a set of software requirements is constructed. This is done by examining the URD and building a 'logical model', using recognised methods and specialist knowledge of the problem domain. The logical model should be an abstract description of what the system must do and should not contain implementation terminology. The model structures the problem and makes it manageable.

A logical model is used to produce a structured set of software requirements that is consistent, coherent and complete. The software requirements specify the functionality, performance, interfaces, quality, reliability, maintainability, safety etc., (see Section 2.4). Software requirements are documented in the Software Requirements Document (SRD). The SRD gives the developer's view of the problem rather than the user's. The SRD must cover all the requirements stated in the URD (SR12). The correspondence between requirements in the URD and SRD is not necessarily one-to-one and frequently isn't. The SRD may also contain requirements that the developer considers are necessary to ensure the product is fit for its purpose (e.g. product assurance standards and interfaces to test equipment).

The main outputs of the SR phase are the:
- Software Requirements Document (SRD);
- Software Project Management Plan for the AD phase (SPMP/AD);
- Software Configuration Management Plan for the AD phase(SCMP/AD);
- Software Verification and Validation Plan for the AD Phase (SVVP/AD);
- Software Quality Assurance Plan for the AD phase (SQAP/AD);
- System Test Plan (SVVP/ST).

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

Defining the software requirements is the developer's responsibility. Besides the developer, participants in the SR phase should include users,

systems engineers, hardware engineers and operations personnel. Project management should ensure that all parties can review the requirements, to minimise incompleteness and error.

SR phase activities must be carried out according to the plans defined in the UR phase (SR01). Progress against plans should be continuously monitored by project management and documented at regular intervals in progress reports.

Figure 2.1 summarises activities and document flow in the SR phase. The following subsections describe the activities of the SR phase in more detail.
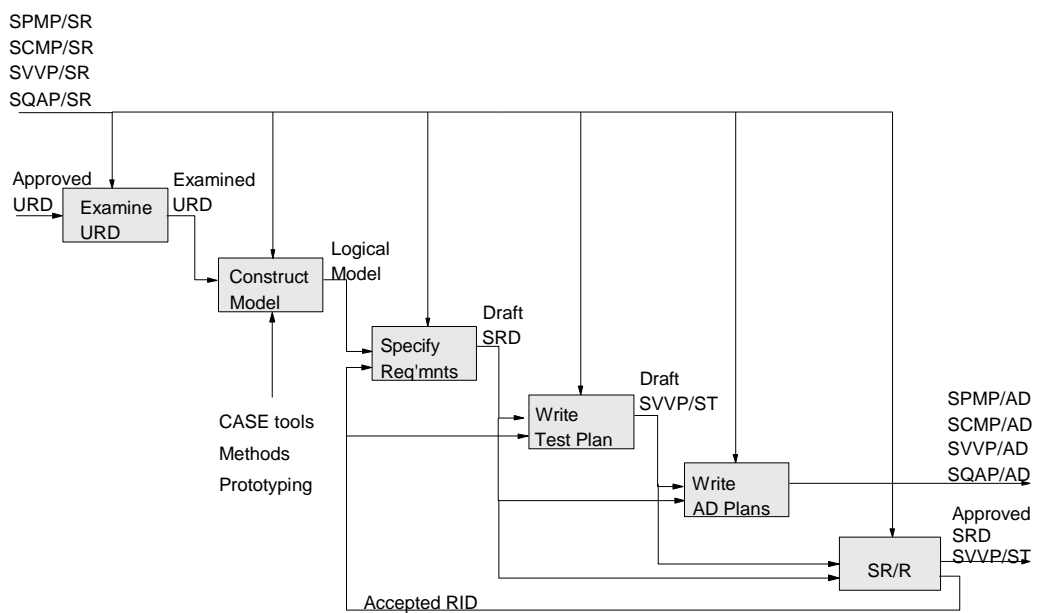
Figure 2.1: SR phase activities

2.2   EXAMINATION OF THE URD

If the developers have not taken part in the User Requirements Review, they should examine the URD and confirm that it is understandable. ESA PSS-05-02, 'Guide to the User Requirements Definition Phase', contains material that should assist in making the URD understandable. Developers should also confirm that adequate technical skills are available for the SR phase.

2.3   CONSTRUCTION OF THE LOGICAL MODEL

A software model is:

- a simplified description of a system;

- hierarchical, with consistent decomposition criteria;

- composed of symbols organised according to some convention;

- built using recognised methods and tools;

- used for reasoning about the software.

A software model should be a 'simplified description' in the sense that it describes the high-level essentials. A hierarchical presentation also makes the description simpler to understand, evaluate at various levels of detail, and maintain. A recognised method, not an undocumented ad-hoc assembly of 'common sense ideas', should be used to construct a software model (SR03). A method, however, does not substitute for the experience and insight of developers, but helps developers apply those abilities better.

In the SR phase, the developers construct an implementation-independent model of what is needed by the user (SR02). This is called a 'logical model' and it:

- shows what the system must do;

- is organised as a hierarchy, progressing through levels of abstraction;

- avoids using implementation terminology (e.g. workstation);

- permits reasoning from cause-to-effect and vice-versa.

A logical model makes the software requirements understandable as a whole, not just individually.

A logical model should be built iteratively. Some tasks may need to be repeated until the description of each level is clear and consistent. Walkthroughs, inspections and technical reviews should be used to ensure that each level of the model is agreed before proceeding to the next level of detail.

CASE tools should be used in all but the smallest projects, because they make clear and consistent models easier to construct and modify.

The type of logical model built depends on the method selected. The method selected depends on the type of software required. Chapter 3

summarises the methods suitable for constructing a logical model. In the following sections, functional decomposition concepts and terminology are used to describe how to construct a logical model. This does not imply that this method shall be used, but only that it may be suitable.

2.3.1   Functional decomposition

The first step in building a logical model is to break the system down into a set of basic functions with a few simple inputs and outputs. This is called 'functional decomposition'.

Functional decomposition is called a 'top-down' method because it starts from a single high-level function and results in a set of low-level functions that combine to perform the high-level function. Each level therefore models the system at different levels of abstraction. The top-down approach produces the most general functions first, leaving the detail to be defined only when necessary.

Consistent criteria should be established for decomposing functions into subfunctions. The criteria should not have any 'implementation bias' (i.e. include design and production considerations). Examples of implementation bias are: 'by programming language', 'by memory requirements' or 'by existing software'.

ESA PSS-05-0 defines the guidelines for building a good logical model. These are repeated below.

(1)   Functions should have a single definite purpose.

(2)   Functions should be appropriate to the level they appear at (e.g. 'Calculate Checksum' should not appear at the same level as 'Verify Telecommands').

(3)   Interfaces should be minimised. This allows design components with weak coupling to be easily derived.

(4)   Each function should be decomposed into no more than seven lower-level functions.

(5)   Implementation terminology should be absent (e.g. file, record, task, module, workstation).

(6)   Performance attributes of each function (capacity, speed etc) should be stated wherever possible.

(7)   Critical functions should be identified.

(8)   Function names should reflect the purpose and say 'what' is to be done, not 'how' it must be done.

(9)   Function names should have a declarative structure (e.g. 'Validate Telecommands').

The recommendation to minimise interfaces needs further qualification. The number of interfaces can be measured by the number of:

- inputs and outputs;

- different functions that interface with it.

High-level functions may have many inputs and outputs. In the first iteration of the logical model, the goal is a set of low-level functions that have a few simple interfaces. The processing of the low-level functions should be briefly described.

In the second and subsequent iterations, functions and data are restructured to reduce the number of interfaces at all levels. Data structures should match the functional decomposition. The data a function deals with should be appropriate to its level.

A logical model should initially cope with routine behaviour. Additional functions may be added later to handle non-routine behaviour, such as startup, shutdown and error handling.

Functional decomposition has reached a sufficient level of detail when the model:

- provides all the capabilities required by the user;

- follows the nine guidelines describe above.

When the use of commercial software looks feasible, developers should ensure that it meets all the user requirements. For example, suppose a high-level database management function is identified. The developer decides that decomposition of the function 'Manage Database' can be stopped because a constraint requirement demands that a particular DBMS is used. This would be wrong if there is a user requirement to select objects from the database. Decomposition should be continued until the user requirement has been included in the model. The designer then has to ensure that a DBMS with the select function is chosen.

2.3.2   Performance analysis

User requirements may contain performance attributes (e.g. capacity, speed and accuracy). These attributes define the performance requirements for a function or group of functions. The logical model should be checked to ensure that no performance requirements conflict; this is done by studying pathways through the data flow.

Decisions about how to partition the performance attributes between a set of functions may involve implementation considerations. Such decisions are best left to the designer and should be avoided in the SR phase.

2.3.3   Criticality analysis

Capability requirements in the URD may have their availability specified in terms of 'Hazard Consequence Severity Category' (HCSC). This can range from 'Catastrophic' through 'Critical' and 'Marginal' to 'Negligible'. If this has been done, the logical model should be analysed to propagate the HCSC to all the requirements related to the capability that have the HCSC attached. Reference 24 describes three criticality analysis techniques:

- Software Failure Modes, Effects and Criticality Analysis (software FMECA);

- Software Common Mode Failure Analysis (Software CFMA);

- Software Fault Tree Analysis (Software FTA).

2.3.4   Prototyping

Models are usually static. However it may be useful to make parts of the model executable to verify them. Such an animated, dynamic model is a kind of prototype.

Prototyping can clarify requirements. The precise details of a user interface may not be clear from the URD, for example. The construction of a prototype for verification by the user is the most effective method of clarifying such requirements.

Data and control flows can be decomposed in numerous ways. For example a URD may say 'enter data into a form and produce a report'.

Only after some analysis does it become apparent what the contents of the form or report should be, and this ought to be confirmed with

the user. The best way to do this to make an example of the form or report, and this may require prototype software to be written.

The development team may identify some requirements as containing a high degree of risk (i.e. there is a large amount of uncertainty whether they can be satisfied). The production of a prototype to assess the feasibility or necessity of such requirements can be very effective in deciding whether they ought to be included in the SRD.

2.4   SPECIFICATION OF THE SOFTWARE REQUIREMENTS

ESA PSS-05-0 defines the following types of software requirements:

Functional Requirements	Documentation Requirements
Performance Requirements	Security Requirements
Interface Requirements	Portability Requirements
Operational Requirements	Quality Requirements
Resource Requirements	Reliability Requirements
Verification Requirements	Maintainability Requirements
Acceptance-Testing Requirements	Safety Requirements

Functional requirements should be organised top-down, according to the structure of the logical model. Non-functional requirements should be attached to functional requirements and can therefore appear at all levels in the hierarchy, and apply to all functional requirements below them. They may be organised as a separate set and cross-referenced, where this is simpler.

2.4.1   Functional requirements

A function is a 'defined objective or characteristic action of a system or component' and a functional requirement 'specifies a function that a system or system component must be able to perform [Ref. 2]. Functional requirements are matched one-to-one to nodes of the logical model. A functional requirement should:

- define 'what' a process must do, not 'how' to implement it;

- define the transformation to be performed on specified inputs to generate specified outputs;

- have performance requirements attached;

- be stated rigourously.

    Functional requirements can be stated rigourously by using short, simple sentences. Several styles are possible, for example Structured English and Precondition-Postcondition style [Ref. 7, 12].

    Examples of functional requirements are:

- 'Calibrate the instrument data'

- 'Select all calibration stars brighter than the 6th magnitude'

2.4.2   Performance requirements

    Performance requirements specify numerical values for measurable variables used to define a function (e.g. rate, frequency, capacity, speed and accuracy). Performance requirements may be included in the quantitative statement of each function, or included as separate requirements. For example the requirements:

    'Calibrate the instrument data'

    'Calibration accuracy shall be 10%'

can be combined to make a single requirement:

    'Calibrate the instrument data to an accuracy of 10%'

    The approach chosen has to trade-off modifiability against duplication.

    Performance requirements may be represented as a range of values [Ref. 21], for example the:

- acceptable value;

- nominal value;

- ideal value.

    The acceptable value defines the minimum level of performance allowed; the nominal value defines a safe margin of performance above the acceptable value, and the ideal value defines the most desirable level of performance.

2.4.3   Interface requirements

Interface requirements specify hardware, software or database elements that the system, or system component, must interact or communicate with. Accordingly, interface requirements should be classified into software, hardware and communications interfaces.

Interface requirements should also be classified into 'internal' and 'external' interface requirements, depending upon whether or not the interface coincides with the system boundary. Whereas the former should always be described in the SRD, the latter may be described in separate 'Interface Control Documents' (ICDs).

This ensures a common, self-contained definition of the interface.

An interface requirement may be stated by:

- describing the data flow or control flow across the interface;

- describing the protocol that governs the exchanges across the interface;

- referencing the specification of the component in another system that is to be used and describing:
  - when the external function is utilised;
  - what is transferred when the external function is utilised.

- defining a constraint on an external function;

- defining a constraint imposed by an external function.

Unless it is present as a constraint, an interface requirement should only define the logical aspects of an interface (e.g. the number and type of items that have to be exchanged), and not the physical details (e.g. byte location of fields in a record, ASCII or binary format etc.). The physical description of data structures should be deferred to the design phase.

Examples of interface requirements are:

- 'Functions X and Y shall exchange instrument data';

- 'Communications between the computer and remote instruments shall be via the IEEE-488 protocol'.

- 'Transmit the amount of fuel remaining to the ground control system every fifteen seconds'.

2.4.4   Operational requirements

Operational requirements specify how the system will run (i.e. when it is to be operated) and how it will communicate with human operators (e.g. screen and keyboards etc.).

Operational requirements may describe physical aspects of the user interface. Descriptions of the dialogue, screen layouts, command language style are all types of operational requirements.

Operational requirements may define ergonomic aspects, e.g. the levels of efficiency that users should be able to attain.

The user may have constrained the user interface in the URD. A function may require the input of some data, for example, and this may be implemented by a keyboard or a speech interface. The user may demand that a speech interface be employed, and this should be stated as an operational requirement.

2.4.5   Resource requirements

Resource requirements specify the upper limits on physical resources such as processing power, main memory, disk space etc. They may describe any requirements that the development or operational environment place upon the software. A resource requirement should state the facts about the resources, and not constrain how they are deployed.

At a system design level, specific allocations of computer resources may have been allocated to software subsystems. Software developers must be aware of resource constraints when they design the software. In some cases (e.g. embedded systems) resource constraints cannot be relaxed.

Examples of resource requirements are:

- 'All programs shall execute with standard user quotas.'

- '5 Mbytes of disk space are available.'

2.4.6   Verification requirements

Verification requirements constrain the design of the product. They may do this by requiring features that facilitate verification of system functions, or by saying how the product is to be verified.

Verification requirements may include specifications of any:

- simulations to be performed;

- requirements imposed by the test environment;

- diagnostic facilities.

Simulation is 'a model that behaves or operates like a given system when provided a set of controlled inputs' [Ref. 2]. A simulator is needed when a system cannot be exercised in the operational environment prior to delivery. The simulator reproduces the behaviour of the operational environment.

2.4.7   Acceptance-testing requirements

Acceptance-testing requirements constrain the design of the product. They are a type of verification requirement, and apply specifically to the TR phase.

2.4.8   Documentation requirements

Documentation requirements state project-specific requirements for documentation, in addition to those contained in ESA PSS-05-0. The format and style of the Interface Control Documents may be described in the documentation requirements, for example.

Documentation should be designed for the target readers (i.e. users and maintenance personnel). The URD contains a section called 'User Characteristics' that may help profile the readership.

2.4.9   Security requirements

Security requirements specify the requirements for securing the system against threats to confidentiality, integrity and availability. They

should describe the level and frequency of access allowed to authorised users of the software. If prevention against unauthorised use is required, the type of unauthorised user should be described. The level of physical protection of the computer facilities may be stated (e.g. backups are to be kept in a fire-proof safe off-site).

Examples of security requirements are protection against:

- accidental destruction of the software;

- accidental loss of data;

- unauthorised use of the software;

- computer viruses.

2.4.10   Portability requirements

Portability requirements specify how easy it should be to move the software from one environment to another. Possible computer and operating systems, other than those of the target system, should be stated.

Examples of the portability requirements are:

- 'it shall be possible to recompile this software to run on computer X without modifying more than 2% of the source code';

- 'no part of the software shall be written in assembler'.

Portability requirements can reduce performance of the software and increase the effort required to build it. For example, asking that the software be made portable between operating systems may require that operating system service routines be called from dedicated modules. This can affect performance, since the number of calls to execute a specific operation is increased. Further, use of computer-specific extensions to a programming language may be precluded.

Portability requirements should therefore be formulated after taking careful consideration of the probable life-time of the software, operating system and hardware.

The portability requirements may reflect a difference in the development and target environment.

2.4.11   Quality requirements

Quality requirements specify the attributes of the software that make it fit for its purpose. The major quality attributes of reliability, maintainability and safety should always be stated separately. Where appropriate, software quality attributes should be specified in measurable terms (i.e. with the use of metrics). For example requirements for the:

- use of specific procedures, standards and regulations;

- use of external software quality assurance personnel;

- qualifications of suppliers.

Any quality-related requirements stated by the user in the UR phase may be supplemented in the SR phase by the in-house standards of the development organisation. Such standards attempt to guarantee the quality of a product by using proper procedures for its production.

2.4.12   Reliability requirements

Software reliability is 'the ability of a system or component to perform its required functions under stated conditions for a specified period of time' [Ref. 2]. The reliability metric, 'Mean Time Between Failure' (MTBF), measures reliability according to this definition.

Reliability requirements should specify the acceptable Mean Time Between Failures of the software, averaged over a significant period. They may also specify the minimum time between failures that is ever acceptable. Reliability requirements may have to be derived from the user's availability requirements. This can be done from the relation:

$$\text{Availability} = \text{MTBF} / (\text{MTBF} + \text{MTTR}).$$

MTTR is the Mean Time To Repair (see Section 2.4.13). MTBF is the average time the software is available, whereas the sum of MTBF and MTTR is the average time it should be operational.

Adequate margins should be added to the availability requirements when deriving the reliability requirements. The specification of the Reliability Requirements should provide a classification of failures based on their severity. Table 2.3.12 provides an example classification, based on the 'able-to-continue?' criterion.

Failure Class	Definition
SEVERE	Operations cannot be continued
WARNING	Operations can be continued, with reduced capability
INFORMATION	Operations can be continued

Table 2.3.12: Failure classification example

Two Examples of reliability requirements are:

- 'the MTBF for severe failures shall be 4 weeks, averaged over 6 months.'

- 'the minimum time between severe failures shall be in excess of 5 days.'

2.4.13   Maintainability requirements

Maintainability is 'the ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment' [Ref. 2]. All aspects of maintainability should be covered in the specification of the maintainability requirements, and should be specified, where appropriate, in quantitative terms.

The idea of fault repair is used to formulate the more restricted definition of maintainability commonly used in software engineering, i.e. 'the ability of an item under stated conditions of use to be retained in, or restored to, within a given period of time, a specified state in which it can perform its required functions'. The maintainability metric, Mean Time To Repair (MTTR), measures maintainability according to this second definition.

Maintainability requirements should specify the acceptable MTTR, averaged over a significant period. They may also specify the maximum time to repair faults ever acceptable. Maintainability requirements may have to be derived from the user's availability requirements (see Section 2.4.12).

Adaptability requirements can effect the software design by ensuring that parameters that are likely to vary do not get 'hard-wired' into the code, or that certain objects are made generic.

Examples of maintainability requirements are:

- 'the MTTR shall be 1 day averaged over a 1 year.'

- 'the time to repair shall never exceed 1 week.'

2.4.14   Safety requirements

Safety requirements specify any requirements to reduce the possibility of damage that can follow from software failure. Safety requirements may identify critical functions whose failure may be hazardous to people or property. Software should be considered safety-critical if the hardware it controls can cause injury to people or damage to property.

While reliability requirements should be used to specify the acceptable frequency of failures, safety requirements should be used to specify what should happen when failures of a critical piece of software actually do occur. In the safety category are requirements for:

- graceful degradation after a failure (e.g. warnings are issued to users before system shutdown and measures are taken to protect property and lives);

- continuation of system availability after a single-point failure.

2.5   SYSTEM TEST PLANNING

System Test Plans must be generated in the SR phase and documented in the System Test section of the Software Verification and Validation Plan (SVVP/ST). See Chapter 6 of this document. The System Test Plan should describe the scope, approach and resources required for the system tests, and address the verification requirements in the SRD.

2.6   THE SOFTWARE REQUIREMENTS REVIEW

The SRD and SVVP/ST are produced iteratively. Walkthroughs and internal reviews should be held before a formal review.

The outputs of the SR phase must be formally reviewed during the Software Requirements Review (SR09). This should be a technical review. The recommended procedure, described in ESA PSS-05-10, is based closely on the IEEE standard for Technical Reviews [Ref. 8].

Normally, only the SRD and System Test Plans undergo the full technical review procedure involving users, developers, management and quality assurance staff. The Software Project Management Plan (SPMP/AD), Software Configuration Management Plan (SCMP/AD), Software Verification and Validation Plan (SVVP/AD), and Software Quality Assurance Plan (SQAP/AD) are usually reviewed by management and quality assurance staff only.

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

- SRD states the software requirements clearly, completely and in sufficient detail to enable the design process to be started;

- SVVP/ST is an adequate plan for system testing the software in the DD phase.

The documents are distributed to the participants in the formal review process for examination. A problem with a document is described in a 'Review Item Discrepancy' (RID) form that may be prepared by any participant in the review. Review meetings are then held which have the documents and RIDs as input. A review meeting should discuss all the RIDs and either accept or reject them. The review meeting may also discuss possible solutions to the problems raised by the RIDs.

The output of a formal review meeting includes a set of accepted RIDs. Each review meeting should end with a decision whether another review meeting is necessary. It is quite possible to proceed to the AD phase with some actions outstanding, which should be relatively minor or have agreed solutions already defined.

2.7   PLANNING THE ARCHITECTURAL DESIGN PHASE

Plans of AD phase activities must be drawn up in the SR phase. Generation of the plans for the AD phase is discussed in chapter 6 of this document. These plans should cover project management, configuration management, verification and validation and quality assurance. Outputs are the:

- Software Project Management Plan for the AD phase (SPMP/AD);

- Software Configuration Management Plan for the AD phase (SCMP/AD);

- Software Verification and Validation Plan for the AD phase (SVVP/AD);

- Software Quality Assurance Plan for the AD phase (SQAP/AD).

CHAPTER 3
METHODS FOR SOFTWARE REQUIREMENTS DEFINITION

3.1   INTRODUCTION

Analysis is the study of a problem, prior to taking some action. The SR phase may be called the 'analysis phase' of ESA PSS-05-0 life cycle. The analysis should be carried out using a recognised method, or combination of methods, suitable for the project. The method selected should define techniques for:

- constructing a logical model;

- specifying the software requirements.

This guide does not provide an exhaustive, authoritative description of any particular method. The references should be consulted to obtain that information. This guide seeks neither to make any particular method a standard nor to define a complete set of acceptable methods. Each project should examine its needs, choose a method and define and justify the selection in the SRD. To assist in making this choice, this chapter summarises some well-known methods and indicates how they can be applied in the SR phase. Possible methods are:

- functional decomposition;

- structured analysis;

- object-oriented analysis;

- formal methods;

- Jackson System Development;

- rapid prototyping.

Although the authors of any particular method will argue for its general applicability, all of the methods appear to have been developed with a particular type of system in mind. It is necessary to look at the examples and case histories to decide whether a method is suitable.

3.2   FUNCTIONAL DECOMPOSITION

Functional decomposition is the traditional method of analysis. The emphasis is on 'what' functionality must be available, not 'how' it is to be

implemented. The functional breakdown is constructed top-down, producing a set of functions, subfunctions and functional interfaces. See Section 2.3.1.

The functional decomposition method was incorporated into the structured analysis method in the late 1970's.

3.3   STRUCTURED ANALYSIS

Structured analysis is a name for a class of methods that analyse a problem by constructing data flow models. Members of this class are:

- Yourdon methods (DeMarco and Ward/Mellor);

- Structured Systems Analysis and Design Methodology (SSADM);

- Structured Analysis and Design Technique (SADT™ [1]).

Structured analysis includes all the concepts of functional decomposition, but produces a better functional specification by rigourously defining the functional interfaces, i.e. the data and control flow between the processes that perform the required functions. The ubiquitous 'Data Flow Diagram' is characteristic of structured analysis methods.

Yourdon methods [Ref. 7, 12] are widely used in the USA and Europe. SSADM [Ref. 4, 9] is recommended by the UK government for 'data processing systems'. It is now under the control of the British Standards Institute (BSI) and will therefore become a British Standard. SADT [Ref. 17, 20] has been successfully used within ESA for some time. Structured analysis methods are expected to be suitable for the majority of ESA software projects.

According to its early operational definition by DeMarco, structured analysis is the use of the following techniques to produce a specification of the system required:

- Data Flow Diagrams;

- Data Dictionary;

- Structured English;

- Decision Tables;

---

[1] ™ trademark of SoftTech Inc, Waltham, Mass., USA.

- Decision Trees.

These techniques are adequate for the analysis of 'information systems'. Developments of structured analysis for 'real-time' or 'embedded systems', have supplemented this list with:

- Transformation Schema;

- State-Transition Diagrams;

- Event Lists;

- Data Schema;

- Precondition-Postcondition Specifications.

SSADM, with its emphasis on data modelling, also includes:

- Entity-Relationship Diagrams (or Entity Models);

- Entity Life Histories.

The methods of structured analysis fall into the groups shown in Table 3.3 according to whether they identify, organise or specify functions or entities.

Activity	Technique
Function Identification	Event Lists Entity Life Histories
Function Organisation	Data Flow Diagrams Transformation Schema Actigrams
Function Specification	Structured English Decision Tables Decision Trees State-Transition Diagrams Transition Tables Precondition-Postconditions
Entity Identification	"Spot the nouns in the description"
Entity Organisation	Data Structure Diagrams Data Schema Entity-Relationship Diagrams
Entity Specification	Data Dictionary

Table 3.3: Structured Analysis Techniques

Structured analysis aims to produce a 'Structured Specification' containing a systematic, rigourous description of a system. This description is in terms of system models. Analysis and design of the system is a model-making process.

3.3.1   DeMarco/SSADM modelling approach

The DeMarco and SSADM methods create and refine the system model through four stages:

- current physical model;

- current logical model;

- required logical model;

- required physical model.

In ESA projects, the goal of the SR phase should be to build a required logical model. The required physical model incorporates implementation considerations and its construction should be deferred to the AD phase.

The DeMarco/SSADM modelling approach assumes that a system is being replaced. The current physical model describes the present way of doing things. This must be rationalised, to make the current logical model, and then combined with the 'problems and requirements list' of the users to construct the required logical model.

This evolutionary concept can be applied quite usefully in ESA projects. Most systems have a clear ancestry, and much can be learned by studying their predecessors. This prevents people from 'reinventing wheels'. The URD should always describe or reference similar systems so that the developer can best understand the context of the user requirements. ESA PSS-05-0 explicitly demands that the relationship to predecessor projects be documented in the SRD. If a system is being replaced and a data processing system is required, then the DeMarco/SSADM approach is recommended.

The DeMarco/SSADM modelling approach is difficult to apply directly when the predecessor system was retired some time previously or does not exist. In such cases the developer of a data processing system should look at the DeMarco/SSADM approach in terms of the activities that must be done in each modelling stage. Each activity uses one or more of the techniques described in Table 3.3. When it makes no sense to think in terms of 'current physical models' and 'current logical models', the developer should define an approach that is suitable for the project. Such a tailored approach should include activities used in the standard modelling stages.

3.3.2   Ward/Mellor modelling approach

Ward and Mellor describe an iterative modelling process that first defines the top levels of an 'essential model'. Instead of proceeding immediately to the lower levels of the essential model, Ward and Mellor recommend that the top levels of an 'implementation model' are built. The cycle continues with definition of the next lower level of the essential model. This stops the essential model diverging too far from reality and prevents the implementation model losing coherence and structure.

The essential model is a kind of logical model. It describes what the system must do to be successful, regardless of the technology chosen to implement it. The essential model is built by first defining the system's environment and identifying the inputs, outputs, stimuli and responses it must handle. This is called 'environmental modelling'. Once the environment is defined, the innards of the system are defined, in progressive detail, by relating inputs to outputs and stimuli to responses. This is called 'behavioural modelling' and supersedes the top-down functional decomposition modelling approach.

ESA real-time software projects should consider using the Ward/Mellor method in the SR phase. The developer should not, however, attempt to construct the definitive implementation model in the SR phase. This task should be done in the AD phase. Instead, predefined design constraints should be used to outline the implementation model and steer the development of the essential model.

3.3.3   SADT modelling approach

There are two stages of requirements definition in SADT:

- context analysis;
- functional specification.

The purpose of context analysis is to define the boundary conditions of the system. 'Functional specification' defines what the system has to do. The output of the functional specification stage is a 'functional architecture', which should be expressed using SADT diagrams.

The functional architecture is a kind of logical model. It is derived from a top-down functional decomposition of the system, starting from the context diagram, which shows all the external interfaces of the system. Data should be decomposed in parallel with the functions.

In the SR phase, SADT diagrams can be used to illustrate the data flow between functions. Mechanism flows should be suppressed. Control flows may be necessary to depict real-time processing.

3.4   OBJECT-ORIENTED ANALYSIS

Object-oriented analysis is the name for a class of methods that analyse a problem by studying the objects in the problem domain. For example some objects that might be important in a motor vehicle simulation problem are engines and gearboxes.

Object-oriented analysis can be viewed as a synthesis of the object concepts pioneered in the Simula67 and Smalltalk programming languages, and the techniques of structured analysis, particularly data modelling. Object-oriented analysis differs from structured analysis by:

- building an object model first, instead of the functional model (i.e. hierarchy of data flow diagrams);

- integrating objects, attributes and operations, instead of separating them between the data model and the functional model.

OOA has been quite successful in tackling problems that are resistant to structured analysis, such as user interfaces. OOA provides a seamless transition to OOD and programming in languages such as Smalltalk, Ada and C++, and is the preferred analysis method when object-oriented methods are going to be used later in the life cycle. Further, the proponents of OOA argue that the objects in a system are more fundamental to its nature than the functions it provides. Specifications based on objects will be more adaptable than specifications based on functions.

The leading OOA methods are:

- Coad-Yourdon;

- Rumbaugh et al's Object Modelling Technique (OMT);

- Shlaer-Mellor;

- Booch.

OOA methods are evolving, and analysts often combine the techniques of different methods when analysing problems. Users of OOA methods are recommended to adopt such a pragmatic approach.

3.4.1   Coad and Yourdon

Coad and Yourdon [Ref. 18] describe an Object-Oriented Analysis (OOA) method based on five major activities:

*   finding classes and objects;

*   identifying structures;

*   identifying subjects;

*   defining attributes;

*   defining services.

These activities are used to construct each layer of a 'five-layer' object model.

Objects exist in the problem domain. Classes are abstractions of the objects. Objects are instances of classes. The first task of the method is to identify classes and objects.

The second task of the method is to identify structures. Two kinds of structures are recognised: 'generalisation- specialisation structures' and 'whole-part structures'. The former type of structure is like a family tree, and inheritance is possible between members of the structure. The latter kind of structure is used to model entity relationships (e.g. each motor contains one armature).

Large, complex models may need to be organised into 'subjects', with each subject supporting a particular view of the problem. For example the object model of a motor vehicle might have a mechanical view and electrical view.

Attributes characterise each class. For example an attribute of an engine might be 'number of cylinders'. Each object will have value for the attribute.

Services define what the objects do. Defining the services is equivalent to defining system functions.

The strengths of Coad and Yourdon's method are its brief, concise description and its use of general texts as sources of definitions, so that the definitions fit common sense and jargon is minimised. The main weakness of the method is its complex notation, which is difficult to use without tool support. Some users of the Coad-Yourdon method have used the OMT diagramming notation instead.

3.4.2   OMT

Rumbaugh et al's Object Modelling Technique (OMT) [Ref 25] transforms the users' problem statement (such as that documented in a User Requirement Document) into three models:

- object model;

- dynamic model;

- functional model.

The three models collectively make the logical model required by ESA PSS-05-0.

The object model shows the static structure in the real world. The procedure for constructing it is:

- identify objects;

- identify classes of objects;

- identify associations (i.e. relationships) between objects;

- identify object attributes;

- use inheritance to organise and simplify class structure;

- organise tightly coupled classes and associations into modules;

- supply brief textual descriptions on each object.

Important types of association are 'aggregation' (i.e. is a part of) and 'generalisation' (i.e. is a type of).

The dynamic model shows the behaviour of the system, especially the sequencing of interactions. The procedure for constructing it is:

- identify sequences of events in the problem domain and document them in 'event traces';

- build a state-transition diagram for each object that is affected by the events, showing the messages that flow, actions that are performed and object state changes that take place when events occur.

The functional model shows how values are derived, without regard for when they are computed. The procedure for constructing it is not to use functional decomposition, but to:

- identify input and output values that the system receives and produces;

- construct data flow diagrams showing how the output values are computed from the input values;

- identify objects that are used as 'data stores';

- identify the object operations that comprise each process.

The functional model is synthesised from object operations, rather than decomposed from a top level function. The operations of objects may be defined at any stage in modelling.

The strengths of OMT are its simple yet powerful notation capabilities and its maturity. It was applied in several projects by its authors before it was published. The main weakness is the lack of techniques for integrating the object, dynamic and functional models.

3.4.3   Shlaer-Mellor

Shlaer and Mellor begin analysis by identifying the problem domains of the system. Each domain 'is a separate world inhabited by its own conceptual entities, or objects' [Ref 26, 27]. Large domains are partitioned into subsystems. Each domain or subsystem is then separately analysed in three steps:

- information modelling;

- state modelling;

- process modelling.

The three modelling activities collectively make the logical model required by ESA PSS-05-0.

The goal of information modelling is to identify the:

- objects in the subsystem

- attributes of each object;

- relationships between each object.

The information model is documented by means of diagrams and definitions of the objects, attributes and relationships.

The goal of state modelling is to identify the:

- states of each object, and the actions that are performed in them;

- events that cause objects to move from one state to another;

- sequences of states that form the life cycle of each object;

- sequences of messages communicating events that flow between objects and subsystems.

State models are documented by means of state model diagrams, showing the sequences of states, and object communication model diagrams, showing the message flows between states.

The goal of process modelling is to identify the:

- operations of each object required in each action;

- attributes of each object that are stored in each action.

Process models are documented by means of action data flow diagrams, showing operations and data flows that occur in each action, an object access model diagrams, showing interobject data access. Complex processes should also be described.

The strengths of the Shlaer-Mellor method are its maturity (its authors claim to have been developing it since 1979) and existence of techniques for integrating the information, state and process models. The main weakness of the method is its complexity.

3.4.4   Booch

Booch models an object-oriented design in terms of a logical view, which defines the classes, objects, and their relationships, and a physical view, which defines the module and process architecture [Ref. 28]. The logical view corresponds to the logical model that ESA PSS-05-0 requires software engineers to construct in the SR phase. The Booch object-oriented method has four steps:

- identify the classes and objects at a given level of abstraction;

- identify the semantics of these classes and objects;

- identify the relationships among these classes and objects;

- implement the classes and objects.

The first three steps should be completed in the SR phase. The last stage is performed in the AD and DD phases. Booch asserts that the process of object-oriented design is neither top-down nor bottom-up but something he calls 'round-trip gestalt design'. The process develops a system incrementally and iteratively. Users of the Booch method are advised to bundle the SR and AD phases together into single 'modelling phase'.

Booch provides four diagramming techniques for documenting the logical view:

- class diagrams, which are used to show the existence of classes and their relationships;

- object diagrams, which are used to show the existence of objects and their behaviour, especially with regard to message communication;

- state-transition diagrams, which show the possible states of each class, and the events that cause transitions from one state to another;

- timing diagrams, which show the sequence of the objects' operations.

Booch's books on object-oriented methods have been described by Stroustrup, the inventor of C++, as the only books worth reading on the subject. This compliment reflects the many insights into good analysis and design practise in his writings. However Booch's notation is cumbersome and few tools are available.

3.5   FORMAL METHODS

A Formal Method should be used when it is necessary to be able to prove that certain consequences will follow specified actions. Formal Methods must have a calculus, to allow proofs to be constructed. This makes rigourous verification possible.

Formal Methods have been criticised for making specifications unintelligible to users. In the ESA PSS-05-0 life cycle, the URD provides the user view of the specification. While the primary user of the SRD is the developer, it does have to be reviewed by other people, and so explanatory notes should be added to ensure that the SRD can be understood by its readership.

Like mathematical treatises, formal specifications contain theorems stating truths about the system to be built. Verification of each theorem is done by proving it from the axioms of the specification. This can, unfortunately, lead to a large number of statements. To avoid this problem,

the 'rigourous' approach can be adopted where shorter, intuitive demonstrations of correctness are used [Ref. 11]. Only critical or problematical statements are explicitly proved.

Formal Methods should be considered for the specification of safety-critical systems, or where there are severe availability or security constraints. There are several Formal Methods available, some of that are summarised in the table below. Z, VDM and LOTOS are discussed in more detail. Table 3.5 lists the more common formal methods.

Method	Reference	Summary
Z	10	Functional specification method for sequential programs
VDM Vienna Development Method	11	Functional specification and development method for sequential programs.
LOTOS Language Of Temporal Ordering Specification	IS 8807	Formal Method with an International Standard. Combination of CCS, CSP and the abstract data typing language ACT ONE. Tools are available.
CSP Communicating Sequential Processes	14	Design language for asynchronous parallelism with synchronous communication. Influenced by JSD and CCS.
OBJ	15	Functional specification language and prototyping tool for sequential programs. Objects (i.e. abstract data types) are the main components of OBJ specifications.
CCS Calculus for Communicating Systems	16	Specification and design of concurrent behaviour of systems. Calculus for asynchronous parallelism with synchronous communication. Used in protocol and communications work. See CSP.
Petri Nets	19	Modelling of concurrent behaviour of systems

Table 3.5: Summary of Formal Methods

3.5.1   Z

Z is a model-oriented specification method based on set theory and first order predicate calculus. The set theory is used to define and organise the entities the software deals with. The predicate calculus is used to define and organise the activities the entities take part in, by stating truths about their behaviour.

Z can be used in the SR phase to permit mathematical modelling the functional behaviour of software. Z is suitable for the specification of sequential programs. Its inability to model concurrent processes makes it unsuitable for use in real-time software projects. To overcome this deficiency Z is being combined with another Formal Method, CSP.

Z specifications can be easily adapted to the requirements of an SRD. Z specifications contain:

- an English language description of all parts of the system;

- a mathematical definition of the system's components;

- consistency theorems;

- other theorems stating important consequences.

3.5.2   VDM

The 'Vienna Development Method' (VDM) is a model-oriented specification and design method based on set theory and Precondition-Postcondition specifications. The set theory is used to define and organise the entities the software deals with. The condition specifications are used to define and organise the activities the entities take part in by stating truths about their behaviour.

VDM uses a top-down method to develop a system model from a high-level definition, using abstract data types and descriptions of external operations, to a low-level definition in implementation-oriented terms. This 'contractual' process, whereby a given level is the implementation of the level above and the requirements for the level below, is called 'reification'. The ability to reason from specification through to implementation is a major feature of VDM.

VDM can be used in the SR phase to permit mathematical modelling the functional behaviour of software. VDM specifications can be easily adapted to the requirements of an SRD.

- VDM specifications contain a:

- description of the state of the system (a mathematical definition of the system components);

- list of data types and invariants;

- list of Precondition-Postcondition specifications.

VDM is generally considered to be deficient in some areas, specifically:

- it is not possible to define explicitly operations (i.e the precondition-postcondition technique does not describe how state variables are changed);

- it is not possible to specify concurrency (making it unsuitable for real-time applications);

- the underlying structure is not modular;

- the specification language lacks abstract and generic features.

3.5.3   LOTOS

LOTOS (Language Of Temporal Ordering Specification) is a formal description technique defined by the International Standardisation Organisation (ISO). It is the only analysis method that is an ISO standard. It was originally developed for use on the Open Systems Interconnection (OSI) standards, but is especially suitable for the specification of distributed processing systems.

LOTOS has two integrated components:

- a 'process algebra' component, which is based on a combination of Calculus of Communicating Systems (CCS, ref 16) and Communicating Sequential Processes (CSP, ref 14);

- a 'data type' component that is based on the algebraic specification language ACT ONE.

LOTOS is both 'executable' (i.e., the specified behaviour may be simulated), and amenable to proof techniques (due to its algebraic properties).

LOTOS encourages developers to work in a structured manner (either top-down or bottom-up), and may be used in several different fashions. Various 'LOTOS specification styles' have been identified and

documented, ranging from 'constraint-oriented' where constraints on, or properties of, the system are specified in isolation from any internal processing mechanisms, to 'monolithic' where the system's behaviour is captured as a tree of alternatives.

Each style has its own strengths; the 'constraint-oriented' style provides a high-level specification of the system and is a powerful way to impose separation of concerns. The 'monolithic' style may be viewed as a much lower level system description that can be transformed into an implementation with relative ease. Two or more styles are often applied in concert, and a LOTOS specification may be refined in much the same way as an English language system requirements specification would be refined into an architectural design and then a detailed design. An obvious benefit of using LOTOS in this way is that each refinement may be verified (i.e. preservation of system properties from a high-level specification to a lower-level specification may be proven).

3.6 JACKSON SYSTEM DEVELOPMENT

Jackson System Development (JSD) analysis techniques are [Ref.13]:

- Structure Diagrams;
- Structure Text;
- System Specification Diagrams.

Unlike structured analysis, which is generally known by its techniques, JSD is best characterised by:

- emphasis on the need to develop software that models the
- behaviour of the things in the real world it is concerned with;
- emphasis on the need to build in adaptability by devising a model that defines the possible system functions;
- avoidance of the top-down approach in favour of a subject-matter based approach.

The term 'model' has a specific meaning in JSD. A model is 'a realisation, in the computer, of an abstract description of the real world' [Ref. 13]. Since the real world has a time dimension it follows that the model will be composed of one or more 'sequential' processes. Each process is

identified by the entities that it is concerned with. The steps in each process describe the (sequential) events in the life cycle of the entity.

A JSD model is described in a 'System Specification' and is, in principle, directly executable. However this will only be the case if each process can be directly mapped to a processor. For example there may be a one-to-one relationship between a telescope and the computer that controls it, but there is a many-to-one relationship between a bank customer and the computer that processes customer transactions (many customers, one computer). JSD implementations employ a 'scheduling' process to coordinate many concurrent processes running on a single processor.

Structured analysis concentrates on devising a 'data model' (hence Data Flow Diagrams and Data Dictionaries). While this may be quite proper for information systems it is sometimes not very helpful when building real-time systems. The JSD method attempts to correct this deficiency by concentrating on devising a 'process model' that is more suited to real-time applications. Application of JSD to information systems is likely to be more difficult than with structured methods, but should be considered where adaptability is a very high priority constraint requirement.

Table 3.6 shows what JSD development activities should take place in the SR phase.

JSD Activity			
Level 0	**Level 1**	**Level 2**	**Level 3**
Specification	Specify Model of Reality	Develop	Write Entity-Action List
		Model Abstractly	Draw Entity-Structure Diagrams
		Define initial versions of model processes	
	Specify System Functions	Add functions to model processes	
		Add timing constraints	

Table 3.6: SR phase JSD development activities

3.7   RAPID PROTOTYPING

A prototype is a 'concrete executable model of selected aspects of a proposed system' [Ref. 5]. If the requirements are not clear, or suspected to be incomplete, it can be useful to develop a prototype based on tentative requirements to explore what the software requirements really are. This is

called 'exploratory prototyping'. Prototypes can help define user interface requirements.

Rapid Prototyping is 'the process of quickly building and evaluating a series of prototypes' [Ref. 6]. Specification and development are iterative.

Rapid Prototyping can be incorporated within the ESA PSS-05-0 life cycle if the iteration loop is contained within a phase.

The development of an information system using 4GLs would contain the loop:

repeat until the user signs off the SRD
  analyse requirements
  create data base
  create user interface
  add selected functions
  review execution of prototype with user

Requirements should be analysed using a recognised method, such as structured analysis, and properly documented in an SRD. Rapid Prototyping needs tool support, otherwise the prototyping may not be rapid enough to be worthwhile.

The prototype's operation should be reviewed with the user and the results used to formulate the requirements in the SRD. The review of the execution of a prototype with a user is not a substitute for the SR review.

Rapid Prototyping can also be used with an Evolutionary Development life cycle approach. A Rapid Prototyping project could consist of several short-period life cycles. Rapid Prototyping tools would be used extensively in the early life cycles, permitting the speedy development of the first prototypes. In later life cycles the product is optimised, which may require new system-specific code to be written.

Software written to support the prototyping activity should not be reused in later phases - the prototypes are 'throwaways'. To allow prototypes to be built quickly, design standards and software requirements can be relaxed. Since quality should be built into deliverable software from its inception, it is bad practice to reuse prototype modules in later phases. Such modules are likely to have interfaces inconsistent with the rest of the design. It is permissible, of course, to use ideas present in prototype code in the DD phase.

This page is intentionally left blank.

CHAPTER 4
TOOLS FOR SOFTWARE REQUIREMENTS DEFINITION

4.1 INTRODUCTION

This chapter discusses the tools for constructing a logical model and specifying the software requirements. Tools can be combined to suit the needs of a particular project.

4.2 TOOLS FOR LOGICAL MODEL CONSTRUCTION

In all but the smallest projects, CASE tools should be used during the SR phase. Like many general purpose tools, such as word processors and drawing packages, a CASE tool should provide:

- a windows, icons, menu and pointer (WIMP) style interface for the easy creation and editing of diagrams;

- a what you see is what you get (WYSIWYG) style interface that ensures that what is created on the display screen is an exact image of what will appear in the document.

Method-specific CASE tools offer the following advantages over general purpose tools:

- enforcement of the rules of the methods;

- consistency checking;

- ease of modification;

- automatic traceability of user requirements through to the software requirements;

- built-in configuration management.

Tools should be selected that have an integrated data dictionary or 'repository' for consistency checking. Developers should check that a tool supports the method that they intend to use. Appendix E contains a more detailed list of desirable tool capabilities.

Configuration management of the model description is essential. The model should evolve from baseline to baseline during the SR phase, and the specification and enforcement of procedures for the identification, change control and status accounting of the model description are

necessary. In large projects, configuration management tools should be used.

4.3   TOOLS FOR SOFTWARE REQUIREMENTS SPECIFICATION

4.3.1   Software requirements management

For large systems, a database management system (DBMS) for storing the software requirements becomes invaluable for maintaining consistency and accessibility. Desirable capabilities of a requirements DBMS are:

- insertion of new requirements;
- modification of existing requirements;
- deletion of requirements;
- storage of attributes (e.g. identifier) with the text;
- selecting by requirement attributes and text strings;
- sorting by requirement attributes and text strings;
- cross-referencing;
- change history recording;
- access control;
- display;
- printing, in a variety formats.

4.3.2   Document production

A word processor or text processor should be used for producing a document. Tools for the creation of paragraphs, sections, headers, footers, tables of contents and indexes all facilitate the production of a document. A spell checker is desirable. An outliner may be found useful for creation of subheadings, 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 in a controlled manner.

CHAPTER 5
THE SOFTWARE REQUIREMENTS DOCUMENT

5.1   INTRODUCTION

The purpose of an SRD is to be an authoritative statement of 'what' the software is to do. An SRD must be complete (SR11) and cover all the requirements stated in the URD (SR12).

The SRD should be detailed enough to allow the implementation of the software without user involvement. The size and content of the SRD should, however, reflect the size and complexity of the software product. It does not, however, need to cover any implementation aspects, and, provided that it is complete, the smaller the SRD, the more readable and reviewable it is.

The SRD is a mandatory output (SR10) of the SR phase and has a definite role to play in the ESA PSS-05-0 documentation scheme. SRD authors should not go beyond the bounds of that role. This means that:

- the SRD must not include implementation details or terminology, unless it has to be present as a constraint (SR15);

- descriptions of functions must say what the software is to do,

- and must avoid saying how it is to be done (SR16);

- the SRD must avoid specifying the hardware or equipment, unless it is a constraint placed by the user (SR17).

5.2   STYLE

The SRD should be systematic, rigourous, clear, consistent and modifiable. Wherever possible, software requirements should be stated in quantitative terms to increase their verifiability.

5.2.1   Clarity

An SRD is 'clear' if each requirement is unambiguous and its meaning is clear to all readers.

If a requirements specification language is used, explanatory text, written in natural language, should be included in the SRD to make it understandable to those not familiar with the specification language.

5.2.2   Consistency

The SRD must be consistent (SR14). There are several types of inconsistency:

- different terms used for the same thing;

- the same term used for different things;

- incompatible activities happening at the same time;

- activities happening in the wrong order.

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

An SRD is consistent if no set of individual requirements conflict. Methods and tools help consistency to be achieved.

5.2.3   Modifiability

Modifiability enables changes to be made easily, completely, and consistently.

When requirements duplicate or overlap one another, cross-references should be included to preserve modifiability.

5.3   EVOLUTION

The SRD should be put under formal change control by the developer as soon as it is approved. New requirements may need to be added and old requirements may have to be modified or deleted. If the SRD is being developed by a team of people, the control of the document may need to be started at the beginning of the SR phase.

The Software Configuration Management Plan for the SR phase should have defined a formal change process to identify, control, track and report projected changes as soon as they are initially identified. Approved changes in requirements must be recorded in the SRD by inserting

document change records and a document status sheet at the start of the SRD.

5.4   RESPONSIBILITY

Whoever actually writes the SRD, the responsibility for it lies with the developer. The developer should nominate people with proven analytical skills to write the SRD. Members of the design and implementation team may take part in the SR/R as they can advise on the technical feasibility of requirements.

5.5   MEDIUM

It is usually assumed that the SRD is a paper document. It may be distributed electronically to participants who have access to the necessary equipment.

5.6   CONTENT

The SRD must be compiled according to the table of contents provided in Appendix C of ESA PSS-05-0 (SR18). This table of contents is derived from ANSI/IEEE Std 830-1984 'Software Requirements Specifications'[Ref. 3]. The description of the model is the only significant addition.

Section 1 should briefly describe the purpose of both the SRD and the product. Section 2 should provide a general description of the project and the product. Section 3 should contain the definitive material about what is required. Appendix A should contain a glossary of terms. Large SRDs (forty pages or more) should also contain an index.

References should be given where appropriate. An SRD should not refer to documents that follow it in the ESA PSS-05-0 life cycle. An SRD should contain no TBCs or TBDs by the time of the Software Requirements Review.

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

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

2 GENERAL DESCRIPTION
    2.1 Relation to current projects
    2.2 Relation to predecessor and successor projects
    2.3 Function and purpose
    2.4 Environmental considerations
    2.5 Relation to other systems
    2.6 General constraints
    2.7 Model description

3 SPECIFIC REQUIREMENTS
(The subsections may be regrouped around high-level functions)
    3.1 Functional requirements
    3.2 Performance requirements
    3.3 Interface requirements
    3.4 Operational requirements
    3.5 Resource requirements
    3.6 Verification requirements
    3.7 Acceptance testing requirements
    3.8 Documentation requirements
    3.9 Security requirements
    3.10 Portability requirements
    3.11 Quality requirements
    3.12 Reliability requirements
    3.13 Maintainability requirements
    3.14 Safety requirements

4 REQUIREMENTS TRACEABILITY MATRIX

        Relevant material unsuitable for inclusion in the above 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.

5.6.1   SRD/1 Introduction

5.6.1.1   SRD/1.1 Purpose (of the document)

This section should:

(1)   define the purpose of the particular SRD;

(2)   specify the intended readership of the SRD.

5.6.1.2   SRD/1.2 Scope (of the software)

This section should:

(1)   identify the software product(s) to be produced by name;

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

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

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

5.6.1.3   SRD/1.3 Definitions, acronyms and abbreviations

This section should provide definitions of all terms, acronyms, and abbreviations needed for the SRD, or refer to other documents where the definitions can be found.

5.6.1.4   SRD/1.4 References

This section should provide a complete list of all the applicable and reference documents. Each document should be identified by its 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.

5.6.1.5   SRD/1.5 Overview (of the document)

This section should:

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

(2)   explain how the SRD is organised.

5.6.2   SRD/2 General Description

This chapter should describe the general factors that affect the product and its requirements. It does not state specific requirements; it only makes those requirements easier to understand.

5.6.2.1   SRD/2.1 Relationship to current projects

This section should describe the context of the project in relation to current projects. This section should identify any other relevant projects the developer is carrying out for the initiator. The project may be independent of other projects or part of a larger project.

Any parent projects should be identified. A detailed description of the interface of the product to the larger system should, however, be deferred until Section 2.5.

5.6.2.2   SRD/2.2 Relationship to predecessor and successor projects

This section should describe the context of the project in relation to past and future projects. This section should identify projects that the developer has carried out for the initiator in the past and also any projects the developer may be expected to carry out in the future, if known.

If the product is to replace an existing product, the reasons for replacing that system should be summarised in this section.

5.6.2.3   SRD/2.3 Function and purpose

This section should discuss the purpose of the product. It should expand upon the points made in Section 1.2.

5.6.2.4   SRD/2.4 Environmental considerations

This section should summarise:

- the physical environment of the target system, i.e. where is the system going to be used and by whom? The URD section called 'User Characteristics' may provide useful material on this topic.

- the hardware environment in the target system, i.e. what computer(s) does the software have to run on?

- the operating environment in the target system, i.e. what operating systems are used?

- the hardware environment in the development system, i.e. what computer(s) does the software have to be developed on?

- the operating environment in the development system, i.e. what software development environment is to be used? or what software tools are to be used?

5.6.2.5   SRD/2.5 Relation to other systems

This section should describe in detail the product's relationship to other systems, for example is the product:

- an independent system?

- a subsystem of a larger system?

- replacing another system?

If the product is a subsystem then this section should:

- summarise the essential characteristics of the larger system;

- identify the other subsystems this subsystem will interface with;

- summarise the computer hardware and peripheral equipment to be used.

A block diagram may be presented showing the major components of the larger system or project, interconnections, and external interfaces.

The URD contains a section 'Product Perspective' that may provide relevant material for this section.

5.6.2.6   SRD/2.6 General constraints

This section should describe any items that will limit the developer's options for building the software. It should provide background information and seek to justify the constraints. The URD contains a section called 'General Constraints' that may provide relevant material for this section.

5.6.2.7   SRD/2.7 Model description

This section should include a top-down description of the logical model. Diagrams, tables and explanatory text may be included.

The functionality at each level should be described, to enable the reader to 'walkthrough' the model level-by-level, function-by-function, flow-by-flow. A bare-bones description of a system in terms of data flow diagrams and low-level functional specifications needs supplementary commentary. Natural language is recommended.

5.6.3   SRD/3 Specific Requirements

The software requirements are detailed in this section. Each requirement must include an identifier (SR04). Essential requirements must be marked as such (SR05). For incremental delivery, each software requirement must include a measure of priority so that the developer can decide the production schedule (SR06). References that trace the software requirements back to the URD must accompany each software requirement (SR07). Any other sources should be stated. Each software requirement must be verifiable (SR08).

The functional requirements should be structured top-down in this section. Non-functional requirements can appear at all levels of the hierarchy of functions, and, by the inheritance principle, apply to all the functional requirements below them. Non-functional requirements may be attached to functional requirements by cross-references or by physically grouping them together in the document.

If a non-functional requirement appears at a lower level, it supersedes any requirement of that type that appears at a higher level. Critical functions, for example, may have more stringent reliability and safety requirements than those of non-critical functions.

Specific requirements may be written in natural language. This makes them understandable to non-specialists, but permits inconsistencies, ambiguity and imprecision to arise. These undesirable properties can be

avoided by using requirements specification languages. Such languages range in rigour from Structured English to Z, VDM and LOTOS.

Each software requirement must have a unique identifier (SR04). Forward traceability to subsequent phases in the life cycle depends upon each requirement having a unique identifier.

Essential software requirements have to be met for the software to be acceptable. If a software requirement is essential, it must be clearly flagged (SR05). Non-essential software requirements should be marked with a measure of desirability (e.g. scale of 1, 2, 3).

The priority of a requirement measures the order, or the timing, of the related functionality becoming available. If the transfer is to be phased, so that some parts come into operation before others, each requirement must be marked with a measure of priority (SR06).

Unstable requirements should be flagged. These requirements may be dependent on feedback from the UR, SR and AD phases. The usual method for flagging unstable requirements is to attach the marker 'TBC'.

The source of each software requirement must be stated (SR07), using the identifier of a user requirement, a document cross-reference, or even the name of a person or group. Backwards traceability depends upon each requirement explicitly referencing its source in the URD or elsewhere.

Each software requirement must be verifiable (SR08). Clarity increases verifiability. Clarity is enhanced by ensuring that each software requirement is well separated from the others. A software requirement is verifiable if some method can be devised for objectively demonstrating that the software implements it correctly.

5.6.4   SRD/Appendix A Requirements Traceability matrix

This section should contain a table summarising how each user requirement is met in the SRD (SR13). See Appendix D.

This page is intentionally left blank.

CHAPTER 6
LIFE CYCLE MANAGEMENT ACTIVITIES

6.1   INTRODUCTION

SR phase activities must be carried out according to the plans defined in the UR phase (SR01). These are:

- Software Project Management Plan for the SR phase (SPMP/SR);

- Software Configuration Management Plan for the SR phase (SCMP/SR);

- Software Verification and Validation Plan for the SR phase (SVVP/SR);

- Software Quality Assurance Plan for the SR phase (SQAP/SR).

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

Plans for AD phase activities must be drawn up in the SR phase. These plans cover project management, configuration management, verification and validation, quality assurance and system tests.

6.2   PROJECT MANAGEMENT PLAN FOR THE AD PHASE

During the SR phase, the AD phase section of the SPMP (SPMP/AD) must be produced (SPM05). The SPMP/AD describes, in detail, the project activities to be carried out in the AD phase.

An estimate of the total project cost must be included in the SPMP/AD (SPM06). Every effort should be made to arrive at a total project cost estimate with an accuracy better than 30%. In addition, a precise estimate of the effort involved in the AD phase must be included in the SPMP/AD (SPM07).

Technical knowledge and experience gained on similar projects should be used to produce the cost estimate. Specific factors affecting estimates for the work required in the AD phase are the:

- number of software requirements;

- level of software requirements;

- complexity of the software requirements;

- stability of software requirements;

- level of definition of external interfaces;

- quality of the SRD.

The function-point analysis method may be of use in the SR phase for costing data-processing systems [Ref. 23].

If an evolutionary software development or incremental delivery life cycle is to be used, the SPMP/AD should say so.

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

6.3   CONFIGURATION MANAGEMENT PLAN FOR THE AD PHASE

During the SR phase, the AD phase section of the SCMP (SCMP/AD) must be produced (SCM44). The SCMP/AD must cover the configuration management procedures for documentation, and any CASE tool outputs or prototype code, to be produced in the AD phase (SCM45).

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

6.4   VERIFICATION AND VALIDATION PLAN FOR THE AD PHASE

During the AD phase, the AD phase section of the SVVP (SVVP/AD) must be produced (SVV12). The SVVP/AD must define how to trace software requirements to components, so that each software component can be justified (SVV13). It should describe how the ADD is to be evaluated by defining the review procedures. It may include specifications of the tests to be performed with prototypes.

During the SR phase, the developer analyses the user requirements and may insert 'acceptance-testing requirements' in the SRD. These requirements constrain the design of the acceptance tests. This must be recognised in the design of the acceptance tests.

The planning of the system tests should proceed in parallel with the definition of the software requirements. The developer may identify 'verification requirements' for the software. These are additional constraints on the verification activities. These requirements are also stated in the SRD.

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

6.5 QUALITY ASSURANCE PLAN FOR THE AD PHASE

During the SR phase, the AD phase section of the SQAP (SQAP/AD) must be produced (SQA06). The SQAP/AD must describe, in detail, the quality assurance activities to be carried out in the AD phase (SQA07).

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;
- records collection maintenance and retention;
- training;
- risk management.

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

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

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

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

6.6   SYSTEM TEST PLANS

The developer must plan the system tests in the SR phase and document it in the SVVP (SVV14). This plan should define the scope, approach, resources and schedule of system testing activities.

Specific tests for each software requirement are not formulated until the DD phase. The System Test Plan should deal with the general issues, for example:

- where will the system tests be done?

- who will attend?

- who will carry them out?

- are tests needed for all software requirements?

- must any special test equipment be used?

- how long is the system testing programme expected to last?

- are simulations necessary?

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

APPENDIX A
GLOSSARY

A.1   LIST OF ACRONYMS

AD	Architectural Design
ANSI	American National Standards Institute
BSSC	Board for Software Standardisation and Control
CASE	Computer Aided Software Engineering
CCS	Calculus of Communicating Systems
CSP	Communicating Sequential Processes
DBMS	Database Management System
ESA	European Space Agency
IEEE	Institute of Electrical and Electronics Engineers
ISO	International Standards Organisation
ICD	Interface Control Document
LOTOS	Language Of Temporal Ordering Specification
OOA	Object-Oriented Analysis
OSI	Open Systems Interconnection
PA	Product Assurance
PSS	Procedures, Specifications and Standards
QA	Quality Assurance
RID	Review Item Discrepancy
SADT	Structured Analysis and Design Technique
SCM	Software Configuration Management
SCMP	Software Configuration Management Plan
SPM	Software Project Management
SPMP	Software Project Management Plan
SQA	Software Quality Assurance
SQAP	Software Quality Assurance Plan
SR	Software Requirements
SRD	Software Requirements Document
SR/R	Software Requirements Review
SSADM	Structured Systems Analysis and Design Methodology
ST	System Test
SUM	Software User Manual
SVVP	Software Verification and Validation Plan
TBC	To Be Confirmed
TBD	To Be Defined
UR	User Requirements
URD	User Requirements Document
VDM	Vienna Development Method

This page is intentionally left blank.

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. IEEE Guide to Software Requirements Specifications, ANSI/IEEE Std 830-1984.

4. SSADM Version 4, NCC Blackwell Publications, 1991

5. Software Evolution Through Rapid Prototyping, Luqi, in COMPUTER, May 1989

6. Structured Rapid Prototyping, J.Connell and L.Shafer, Yourdon Press, 1989.

7. Structured Analysis and System Specification, T.DeMarco, Yourdon Press, 1978.

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

9. Structured Systems Analysis and Design Methodology, G.Cutts, Paradigm, 1987.

10. The Z Notation - a reference manual, J.M.Spivey, Prentice-Hall, 1989.

11. Systematic Software Development Using VDM, C.B.Jones, Prentice-Hall, 1986.

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

13. System Development, M.Jackson, Prentice-Hall, 1983.

14. Communicating Sequential Processes, C.A.R.Hoare, Prentice Hall International, 1985.

15. Programming with parameterised abstract objects in OBJ, Goguen J A, Meseguer J and Plaisted D, in Theory and Practice of Software Technology, North Holland, 1982.

16.   A Calculus for Communicating Systems, R.Milner, in Lecture Notes in Computer Science, No 192, Springer-Verlag.

17.   Structured Analysis for Requirements Definition, D.T.Ross and K.E.Schoman, IEEE Transactions on Software Engineering, Vol SE-3, No 1, January 1977.

18.   Object-Oriented Analysis, P.Coad and E.Yourdon, Second Edition, Yourdon Press, 1991.

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

20.   Structured Analysis (SA): A Language for Communicating Ideas, D.T.Ross, IEEE Transactions on Software Engineering, Vol SE-3, No 1, January 1977.

21.   Principles of Software Engineering Management, T.Gilb, Addison-Wesley.

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

23.   Software function, source lines of code, and development effort prediction: a software science validation, A.J.Albrecht and J.E.Gaffney, IEEE Transactions on Software Engineering, vol SE-9, No 6, November 1983.

24.   Software Reliability Assurance for ESA space systems, ESA PSS-01-230 Issue 1 Draft 8, October 1991.

25.   Object-Oriented Modeling and Design, J.Rumbaugh, M.Blaha, W.Premerlani, F.Eddy and W.Lorensen, Prentice-Hall, 1991

26.   Object-Oriented Systems Analysis - Modeling the World in Data, S.Shlaer and S.J.Mellor, Yourdon Press, 1988

27.   Object Lifecycles - Modeling the World in States, S.Shlaer and S.J.Mellor, Yourdon Press, 1992.

28.   Object-Oriented Design with Applications, G.Booch, Benjamin Cummings, 1991.

APPENDIX C
MANDATORY PRACTICES

This appendix is repeated from ESA PSS-05-0, Appendix D.3

SR01   SR phase activities shall be carried out according to the plans defined in the UR phase.

SR02   The developer shall construct an implementation-independent model of what is needed by the user.

SR03   A recognised method for software requirements analysis shall be adopted and applied consistently in the SR phase.

SR04   Each software requirement shall include an identifier.

SR05   Essential software requirements shall be marked as such.

SR06   For incremental delivery, each software requirement shall include a measure of priority so that the developer can decide the production schedule.

SR07   References that trace software requirements back to the URD shall accompany each software requirement.

SR08   Each software requirement shall be verifiable.

SR09   The outputs of the SR phase shall be formally reviewed during the Software Requirements Review.

SR10   An output of the SR phase shall be the Software Requirements Document (SRD).

SR11   The SRD shall be complete.

SR12   The SRD shall cover all the requirements stated in the URD.

SR13   A table showing how user requirements correspond to software requirements shall be placed in the SRD.

SR14   The SRD shall be consistent.

SR15   The SRD shall not include implementation details or terminology, unless it has to be present as a constraint.

SR16   Descriptions of functions ... shall say what the software is to do, and must avoid saying how it is to be done.

SR17   The SRD shall avoid specifying the hardware or equipment, unless it is a constraint placed by the user.

SR18   The SRD shall be compiled according to the table of contents provided in Appendix C.

APPENDIX D
REQUIREMENTS TRACEABILITY MATRIX

REQUIREMENTS TRACEABILITY MATRIX SRD TRACED TO URD		DATE: <YY-MM-DD> PAGE 1 OF <nn>
PROJECT: <TITLE OF PROJECT>		
URD IDENTIFIER	SRD IDENTIFIER	SUMMARY OF USER REQUIREMENT

This page is intentionally left blank.

APPENDIX E
CASE TOOL SELECTION CRITERIA

This appendix lists selection criteria for the evaluation of CASE tools for building a logical model.

The tool should:

1.   enforce the rules of each method it supports;

2.   allow the user to construct diagrams according to the conventions of the selected method;

3.   support consistency checking (e.g. balancing a DFD set);

4.   store the model description;

5.   be able to store multiple model descriptions;

6.   support top-down decomposition (e.g. by allowing the user to create and edit lower-level DFDs by 'exploding' processes in a higher-level diagram);

7.   minimise line-crossing in a diagram;

8.   minimise the number of keystrokes required to add a symbol;

9.   allow the user to 'tune' a method by adding and deleting rules;

10. support concurrent access by multiple users;

11. permit controlled access to the model database (usually called a repository);

12. permit reuse of all or part of existing model descriptions, to allow the bottom-up integration of models (e.g. rather than decompose a high-level function such as 'Calculate Orbit', it should be possible to import a specification of this function from another model);

13. support document generation according to user-defined templates and formats;

14. support traceability of user requirements to software requirements (and software requirements through to components and code);

15.   support conversion of a diagram from one style to another (e.g. Yourdon to SADT and vice-versa);

16.   allow the user to execute the model (to verify real-time behaviour by animation and simulation);

17.   support all configuration management functions (e.g. to identify items, control changes to them, storage of baselines etc);

18.   keep the users informed of changes when part of a design is concurrently accessed;

19.   support consistency checking in any of 3 checking modes, selectable by the user:

   •   interpretative, i.e. check each change as it is made; reject any illegal changes;

   •   compiler, i.e. accept all changes and check them all at once at the end of the session,;

   •   monitor, i.e. check each change as it is made; issue warnings about illegal changes;

20.   link the checking mode with the access mode when there are multiple users, so that local changes can be done in any checking mode, but changes that have non-local effects are checked immediately and rejected if they are in error;

21.   support scoping and overloading of data item names;

22.   provide context-dependent help facilities;

23.   make effective use of colour to allow parts of a display to be easily distinguished;

24.   have a consistent user interface;

25.   permit direct hardcopy output;

26.   permit data to be imported from other CASE tools;

27.   permit the exporting of data in standard graphics file formats (e.g. CGM);

28.   provide standard word processing functions;

29.   permit the exporting of data to external word-processing applications and editors;

30.   describe the format of tool database (i.e. repository) in the user documentation, so that users can manipulate the database using external software and packages (e.g. tables of ASCII data).

This page is intentionally left blank.

APPENDIX F
INDEX