BSSC (2002)1 Issue 1.0
14 March 2002

ESA Ground Segment Software Engineering and Management Guide

Part A Software Engineering

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

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

DOCUMENT STATUS SHEET

DOCUMENT STATUS SHEET			
1. DOCUMENT TITLE: ESA Ground Segment Software Engineering and Management Guide : Part A Software Engineering			
2. ISSUE	3. REVISION	4. DATE	5. REASON FOR CHANGE
1	0	March 2002	First Issue

Approved, March 2002
Board for Software Standardisation and Control
M. Jones, BSSC co-chairman U. Mortensen, BSSC co-chairman

Copyright © 2002 by European Space Agency

TABLE OF CONTENTS

DOCUMENT STATUS SHEET II

TABLE OF CONTENTS III

PREFACE X

INTRODUCTION 1
1.1 PURPOSE 1
1.2 OVERVIEW 2
1.3 TAILORING THE GUIDE 2

SOFTWARE LIFE CYCLE PROCESSES FOR GROUND SEGMENT 4
2.1 INTRODUCTION 4
2.2 SPACE PROJECT ENGINEERING 4
2.3 GROUND SEGMENT SYSTEMS 5
2.4 LEVELS OF DECOMPOSITION 7
2.5 SOFTWARE REQUIREMENTS FOR GROUND SYSTEMS 10
2.6 PROCESSES, ACTIVITIES AND TASKS 12
2.7 THE SOFTWARE LIFE CYCLE PROCESSES 12
2.7.1 System Engineering for Software 13
2.7.2 Software Requirements Engineering 14
2.7.3 Software Design Engineering 15
2.7.4 Software Verification and Validation 16
2.7.5 Software Operations Engineering 16
2.7.6 Software Maintenance 17
2.8 GROUND SEGMENT SYSTEM ENGINEERING 17
2.8.1 Ground Segment System Engineering Phases 17
2.8.2 Mapping ECSS-E-40 onto GS System Engineering Phases 20
2.9 SUPPORTING THE SOFTWARE ENGINEERING PROCESS 22
2.9.1 Documentation 22
2.9.2 Configuration Management 22
2.9.3 Software Product Assurance 23
2.9.4 Software Project Management 23

SYSTEM ENGINEERING FOR SOFTWARE 25
3.1 INTRODUCTION 25

3.2 PROCESS INPUTS....26
3.3 ACTIVITIES....26
3.3.1 System Requirements Analysis....27
3.3.1.1 System requirements specification....27
3.3.1.2 Criticality Analysis....30
3.3.2 System Partitioning....33
3.3.2.1 Partition requirements to appropriate subsystems....34
3.3.2.2 Define subsystem interfaces....35
3.3.3 System Level Requirements for Software Verification and Validation 35
3.3.4 System Level Requirements for Software Integration....35
3.3.4.1 Software observability requirements....35
3.3.4.2 Interface requirements....37
3.3.4.3 Development constraints....38
3.3.4.4 System Level Integration of Software....38
3.3.5 System Requirements Review....39
3.4 PROCESS OUTPUTS....39
3.4.1 Requirements Baseline....39
3.4.2 Interface Requirements Document....40
3.4.3 Design Justification File....40

SOFTWARE MANAGEMENT....41
4.1 INTRODUCTION....41
4.2 PROCESS INPUTS....41
4.3 ACTIVITIES....41
4.3.1 Planning....41
4.3.2 Selection of the Software Life Cycle Model....42
4.3.2.1 Standard Waterfall Model....44
4.3.2.2 Incremental delivery model....45
4.3.2.3 Evolutionary development model....46
4.3.3 Technical Budget and Margin Management....48
4.4 PROCESS OUTPUTS....48

SOFTWARE REQUIREMENTS ENGINEERING....50
5.1 INTRODUCTION....50
5.2 PROCESS INPUTS....51
5.3 ACTIVITIES....51

5.3.1 Software Requirements Analysis....52
5.3.1.1 Establishing the software requirements....52
5.3.1.2 Identification of Software Requirements....58
5.3.1.3 Evaluation of software requirements....59
5.3.1.4 Software Requirements Review (SWRR)....60
5.3.2 Software Architectural Design....61
5.3.2.1 Construction of the architectural model....61
5.3.2.2 Definition of the Interfaces....65
5.3.2.3 Software Integration Planning....66
5.3.2.4 Evaluation of the Architecture....67
5.3.2.5 Preliminary Design Review (PDR)....67
5.3.3 Software Verification and Validation....68
5.3.3.1 Project level requirements....68
5.3.3.2 Tasks....69
5.3.3.3 Organisation....69
5.4 PROCESS OUTPUTS....69
5.4.1 Technical Specification....69
5.4.2 Design Justification File....70
5.4.3 Design Definition File....70
5.4.4 Interface Control Document....70

SOFTWARE DESIGN ENGINEERING....71
6.1 INTRODUCTION....71
6.2 PROCESS INPUTS....71
6.3 ACTIVITIES....72
6.3.1 Design of Software Items....72
6.3.1.1 Software Component Design....72
6.3.1.2 Interface Design....73
6.3.1.3 Drafting of Software User Manual....74
6.3.1.4 Software Unit Test Planning....74
6.3.1.5 Software Integration Planning....75
6.3.1.6 Evaluation of Design and Test Specifications....75
6.3.2 Coding and Testing....76
6.3.2.1 Develop and document software units....77
6.3.2.2 Unit Testing....78
6.3.2.3 Software User Manual Updates....79
6.3.2.4 Integration Testing Requirements....79

6.3.2.5 Evaluation of Code and Test Results....79
6.3.3 Integration....79
6.3.3.1 Integration Planning....80
6.3.3.2 Integration Testing....80
6.3.3.3 Software User Manual Update....80
6.3.3.4 Evaluation of the Integration Testing....81
6.3.4 Validation Testing....81
6.3.5 Critical Design Review....82
6.4 PROCESS OUTPUTS....83
6.4.1 Design Definition File (DDF)....83
6.4.1.1 Software Components Design Documents....83
6.4.1.2 Software User Manual....83
6.4.2 Technical Specification (TS)....84
6.4.2.1 Interface Control Document (ICD)....84
6.4.3 Design Justification File (DJF)....84
6.4.3.1 Software Unit Test Plan....84
6.4.3.2 Software Integration Plan....84
6.4.3.3 Software Validation against Technical Specification....84

SOFTWARE VALIDATION AND ACCEPTANCE....85
7.1 INTRODUCTION....85
7.2 PROCESS INPUTS....85
7.2.1 Requirements Baseline....85
7.2.2 Technical Specification....86
7.2.2.1 Interface Control Document....86
7.2.2.2 Software Requirements Specification....86
7.2.3 Design Definition File....86
7.2.3.1 Software User Manual....86
7.3 ACTIVITIES....86
7.3.1 Validation Testing against RB subset-1....86
7.3.2 Qualification Review....86
7.3.3 Delivery and Installation....87
7.3.4 Validation Testing against RB subset-2....87
7.3.5 Validation Testing against RB....87
7.3.6 Software User Manual Updates....88
7.3.7 Acceptance Review....88
7.4 PROCESS OUTPUTS....89

7.4.1 Design Definition File 89
7.4.1.1 Software Installation Plan 89
7.4.1.2 Source Code Files, Build Code Files, Executable Code Files....89
7.4.2 Design Justification File 89
7.4.2.1 Preliminary Acceptance Test Specification 89
7.4.2.2 Preliminary Acceptance Test Results....89
7.4.2.3 Qualification Review Report 89
7.4.2.4 Operational Acceptance Test Specification 90
7.4.2.5 Operational Acceptance Test Results....90
7.4.2.6 Observation Reports....90
7.4.2.7 Compliance Matrix 90
7.4.2.8 Acceptance Review Report 90

SOFTWARE OPERATIONS ENGINEERING....91
8.1 INTRODUCTION 91
8.2 PROCESS INPUTS....92
8.3 PROCESS ACTIVITIES 92
8.3.1 Operational Planning....93
8.3.1.1 Procedures for Anomaly Handling 95
8.3.1.2 Operational Testing Specifications 95
8.3.2 Operational Testing....96
8.3.3 System Operation 96
8.3.4 User Support....97
8.4 PROCESS OUTPUTS....98
8.4.1 Software Operations Plan....98

SOFTWARE MAINTENANCE....99
9.1 INTRODUCTION 99
9.2 PROCESS INPUTS....100
9.3 PROCESS ACTIVITIES 100
9.3.1 Problem and Modification Analysis....101
9.3.2 Modification Implementation 103
9.3.2.1 Test Criteria....103
9.3.2.2 Implementation....104
9.3.3 Maintenance Review/Acceptance 104
9.3.4 Software Migration 104
9.3.5 Software Retirement 106

9.4 PROCESS OUTPUTS....107
9.4.1 Maintenance File (MF)....107
9.4.1.1 Problem Analysis Report....107
9.4.1.2 Software Release Note....107
9.4.2 Maintenance Plan....108
9.4.3 Migration Plan....108
9.4.4 Migration Justification....108

SOFTWARE RE-USE....110
10.1 INTRODUCTION....110
10.2 PROCESS INPUTS....111
10.3 ACTIVITIES....111
10.3.1 Developing Software for Intended Re-use....111
10.3.1.1 Customer Requirements....111
10.3.1.2 Supplier Requirements....112
10.3.2 Re-using Software from Other Projects....112
10.3.3 Use of Third Party COTS Products....113
10.4 PROCESS OUTPUTS....114
10.4.1 Requirements Baseline....114
10.4.2 Technical Specification....115
10.4.3 Software Development Plan....115
10.4.4 Design Justification File....115

MAN-MACHINE INTERFACES....117
11.1 INTRODUCTION....117
11.2 PROCESS INPUTS....117
11.3 PROCESS ACTIVITIES....118
11.3.1 Determine Prototyping Requirements....118
11.3.2 Determine MMI Standards....118
11.3.2.1 General Guidelines....119
11.3.2.2 Information Display Guidelines....121
11.3.3 Supplier Consideration of MMI Aspects....122
11.4 PROCESS OUTPUTS....123
11.4.1 Requirements Baseline....123
11.4.2 Technical Specification....123
11.4.3 Design Justification File....123

APPENDIX A GLOSSARY....1

DEFINITIONS 1
Operational Software 1
Non-operational Software 1
ABBREVIATED TERMS 1

APPENDIX B REFERENCES 1

APPENDIX C DOCUMENT LIFECYCLES 1

PREFACE

This Guide comprises three parts: A, B and C.

This part, Part A, describes the software engineering activities for space system ground segments, and is designed to be applied in all ground segment software engineering projects undertaken by the European Space Agency (ESA). In the past, ground segment software development projects undertaken by ESA and, especially, the European Space Operations Centre (ESOC) have been undertaken according to the ESA Software Engineering Standards, ESA PSS-05-0. ESA now applies the European Co-operation for Space Standardisation (ECSS) E-40 Space Engineering: Software standard for all space software projects. Requirements relating to ground segment software are also specified in the ECSS E-70 Space Engineering: Ground Systems and Operations standard.

This part of the guide describes how to implement the requirements of ECSS-E-40 and ECSS-E-70 on ground segment software projects undertaken by the various parts of ESA. The guide also:

- carries over working practices from ESA PSS-05 and ESA Quality Management System, where they fully implement the requirements of ECSS-E-40 and the other standards
- reflects the lessons learnt in the application of ESA PSS-05.

The PSS-05 standards covered the development and management of software development projects. ECSS-E-40 does not contain project management and configuration management practices, which are defined in the ECSS-M series of standards, nor quality assurance standards, which are defined in ECSS-Q series of standards. Part B of this guide addresses these requirements.

A BSSC Working Group prepared the first draft of Part A of this Guide. The Working Group comprised Yves Doat, Gottlob Gienger, Gianpiero di Girolamo, Angela Head, Michael Jones, Alfio Mantineo, and Eric Perdrix, all from ESA/ESOC with Serge Valera from ESA/ESTEC. Richard Jack was editor and researcher. The Guide was reviewed and revised by the following BSSC

Members: Michael Jones (co-chairman) Uffe Mortensen (co-chairman), Alessandro Ciarlo, Daniel de Pablo and Lothar Winzer, assisted by Eduardo Gomez. The BSSC wishes to thank John Brinkworth and John Barcroft for editing the final version.

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

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

Germany

michael.jones@esa.int

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

The Netherlands

uffe.mortensen@esa.int

CHAPTER 1
INTRODUCTION

1.1 PURPOSE

This Software Engineering and Management Guide concerns the development and maintenance of ground segment software. This guide covers all aspects of software development for ground segment software including requirements definition, design, production, verification and validation, transfer, operations and maintenance.

This guide is a unified and complete description of how to implement the requirements of ECSS-E-40, ECSS-E-70, ECSS-Q-80 and the ECSS-M standards as concerns ground segment software. The guide also:

- carries over working practices from ESA PSS-05 [Ref 24, 25] and ESA Quality Management System, where they fully implement the requirements of ECSS-E-40 and the other standards
- reflects the lessons learnt in the application of ESA PSS-05.

The guide complies with the requirements of ECSS-E-40, Software [Ref. 11], and ECSS-Q-80, Software Product Assurance [Ref. 10], which are themselves based on Information Technology Software Life Cycle Processes 12207:1995 [Ref. 1]. As the software developed according to this guide is specifically used in ground segments, the guide is also compliant with the applicable requirements of ECSS E-70, Ground Systems and Operations [Ref. 13].

Managers, software engineers and assurance specialists applying this guide are thus conformant with the relevant ECSS standards.

1.2 OVERVIEW

Part A provides an overview of the software life cycle process, in accordance with the E-40 and E-70 standards, and their application in the lifecycle of software for ground systems. It also gives advice on applying this guide to a particular project.

Part B of the Guide covers the practices to implement effective management of the development, operation and maintenance of ground segment software. The majority of these practices are defined in the ECSS Management [Ref. 3 to 9] and ECSS Product Assurance [Ref. 10] series of documents, rather than the ECSS E-40 and E-70 standards.

Part C provides proposed templates for documents.

1.3 TAILORING THE GUIDE

This guide describes the software engineering processes to be applied to all deliverable software products developed for ground segments. The processes are characterised in terms of activities and tasks. The tasks contain the requirements to be applied. As described in chapter 2.2, the tasks are organised as mandatory practices, recommended practices and guidelines.

The tailoring process is the deletion of non-applicable processes, activities and tasks [Ref. 1]. The addition of unique or special processes, activities or tasks is permitted, as specified in the contract. Tailoring may also involve the deletion of outputs or the limitation of applicability to certain parts of the system. The existing requirements may be refined or specified. Tailoring guidelines (for informative purposes only) are provided in appendix C of [Ref. 11].

When different procurements are used, each software product can be procured using different tailoring approaches. Some of the criteria that can be considered are:

- Overall space project risk (see part B)

- Characteristics of product, equipment or project such as criticality, longevity, size, operational or non-operational status (see part B), real-time constraints and level of definition of the requirements

- Cost.

The tailoring process is explained in Part B of this Guide section 2.3.2.

CHAPTER 2
SOFTWARE LIFE CYCLE PROCESSES FOR GROUND SEGMENT

2.1 INTRODUCTION

The chapter contains a brief description of the scope of ground systems and explains the concept of decomposing the ground segment to its lower level components. This seeks to establish the relationship of the software development to the overall ground segment development.

This chapter introduces the concept of applying processes to develop ground segment software.

2.2 SPACE PROJECT ENGINEERING

The purpose of a space project is to deliver to a customer a system that includes one or more elements intended for operation in space [Ref. 15]. The engineering process, one of five main domains within space projects, is responsible for the definition of the system, verification that the customer's technical requirements are achieved, and compliance with the applicable project constraints.

A space system is composed of three principal elements:

- Space segment
- Launch service segment
- Ground segment

A specific space system consists of a set of interdependent elements put together to achieve a given objective. The physical form may include any combination of hardware, software and personnel.

This Guide considers only the requirements for the development and maintenance of software for ground segments.

2.3 GROUND SEGMENT SYSTEMS

A brief summary of the scope of the ground segment is presented. A more detailed description of the scope of the engineering activities for the ground segment may be found in ECSS E-00 [Ref. 15] and ECSS E-70 [Ref. 13].

Within a Space System, Mission Operations comprises that subset of mission engineering activities required to operate the space segment. These activities are broadly flight operations, ground operations and logistics engineering. Mission Operations implements the mission in accordance with the stated, implied or re-defined mission objectives in terms of providing plans and services, conducting experiments, producing, providing and distributing mission products.

In this context the domain of ECSS-E-70 [Ref. 13] covers the Ground Segment, i.e. the ground facilities and personnel involved in the preparation and/or execution of Mission Operations for unmanned missions. In addition, it also considers those aspects of the space segment system of relevance to mission operations. The ground segment can be seen as composed of two main components:

- Ground Operations Organisations, comprising the human resources performing the various operational tasks and preparing the mission operations data (e.g. procedures, documentation, mission parameters, mission description data, etc.).
- Ground Segments, consisting of the major ground infrastructure elements that are used to support the preparation activities leading up to mission operations, the conduct of operations themselves and all post-operational activities.

The scope of this guide is the development of software for the Ground Segments. The Ground Segment consists of a number of subsystems [Ref. 15]. A subsystem consists of a set of interdependent components constituted to achieve a given

objective by performing a specified function. A subsystem does not provide sufficient functionality to satisfy the customer's needs.

A Ground Segment normally consists of, but is not restricted to, the following main subsystems:

- The Mission Control System (MCS). This is the system responsible for control of the mission after launch
- The Ground Station System (GSTS). This includes the antenna, receivers, ground converter, communication software, etc.
- The Ground Communication Sub-net (GCS). This is the communications network linking the ground stations and control centres used in any given mission.
- On-Board Software Validation Facility (SVF), used for developing, testing and maintenance of on-board software

The MCS may contain:

- Operations Control System (OCS),
- Payload Control System (PCS) and
- Mission Exploitation System (MES),
- Simulators, including simulators for training and testing

In addition, Electrical Ground Support Equipment (EGSE) will be needed for the check-out of spacecraft or payloads before launch.

With the exception of Simulators and the SVF, these are further defined in ECSS-E-70 [Ref. 13].

Those systems may be grouped together to constitute facilities. There is a direct correspondence between ground systems and operations space organisations. The combination of an operations organisation and its corresponding supporting facility constitutes a Ground Segment Entity. Classical examples of entities are Control Centres, from which the elements of an operations organisation control an element of the mission using the related facilities. Each such entity may contain a number of software

products in the sense of ECSS-E-40 [Ref. 11]. For example an Operations Control System could contain the following software products: control system kernel, mission planning system, file transfer system, data distribution system. This guide is concerned with the software engineering of the Ground systems or the software products contained therein. It is not concerned with the ground operations organisations, but it is clear that their needs will have a profound effect on the software products.

Although they are ground segment elements, the EGSE and the SVF are usually procured along with the space segment. Their development processes are rather closely coupled to the space segment development phases rather than to the ground segment ones discussed in section 2.8.

In general, development of the ground segment elements will be delayed compared to that of the space segment elements. This is because of the design dependencies: for example the training simulator design cannot be completed until the design of the spacecraft elements it is simulating is sufficiently well known. Since the ground segment is used for operating the space segment, the main constraint on its development is that of readiness for operations. This normally means that the ground segment will be available for operations some time before launch. However, in some cases the development software engineering processes may continue after launch, e.g. for a deep space mission for which the mission operations takes place at the end of a long "cruise phase" (~years), it may be decided to develop mission operations software after launch.

2.4 LEVELS OF DECOMPOSITION

Each of the ground segment subsystems is composed of a number of lower-level components. The component is referred to as an equipment when it consists of a hardware element (which may include embedded software) and called a software product when

it comprises software only. A ground segment subsystem may consist of any number of equipments and software products.

The ground segment may be implemented as a single procurement, but more usually several separate procurements are used for the separate software products or equipment of which it is composed.

The levels of decomposition are fully defined in ECSS-E-00 [Ref. 15]. They are summarised, in Table 2.1, in terms of the software engineering terminology used in E-40 [Ref. 11].

Level of Decomposition	**Definition**
Software product	A software product, an item comprising software only, is designed and built to achieve a specific purpose.
Software Component	A software product is composed of components. A component consists of two or more software units joined together to form an item with defined characteristics but which does not by itself achieve a specific purpose.
Software Unit	The lowest level of decomposition. A software unit is any software entity that is discrete and identifiable with respect to compiling, combining with other items and loading.

Table 2.1 Levels of Decomposition

This Guide describes the processes required to develop a software product. In doing so, it covers some activities relating to the integration of software product to the ground segment or overall space system.

The level of integration required will be dependent on the project requirements. A particular project may require the development of a single software product or it may involve the development of a number of software products and equipment.

Particular care should be taken to ensure that the boundary of the software product is clear and that the requirements for integration into the wider system are addressed.

2.5 SOFTWARE REQUIREMENTS FOR GROUND SYSTEMS

ECSS-E-70, Ground Systems and Operations [Ref. 13] standard, provides a number of specific requirements for software products for ground systems.

Software products often constitute critical elements of ground segment in terms of cost, schedule and technical risk. A particular aspect for consideration [Ref. 13] is to maximise re-use of functionality across missions, since in most cases only a small part of the total system needs to be modified to accommodate the mission specific characteristics.

The following general aspects, listed in Table 2.2, should be considered to ensure a cost-effective design, implementation operation and maintenance of ground software, based on the requirements of ECSS-E-70 [Ref. 13]. This Guide provides additional information on how these general requirements are addressed within the ECSS-E-40 [Ref.11].

Aspect	**Meaning**
Configurability	Primarily to be modular and parameter driven to accommodate evolution and to enable re-use across missions. Modularity allows the replacement of parts with different components
Vendor independence	With respect to computer platform and vendor
Scalability	To facilitate expansion of the hardware, software configuration of the system without major re-design

Aspect	**Meaning**
Portability	To reduce cost of migrating the system to a new computer platform and operating system in order to cope with the obsolescence of hardware and software (e.g. in case of long duration missions or long-lived infrastructure software)
Openness	Refers to the capability to interface with other (new) systems or to add functionality without major re-design.
Re-usability	To permit re-usability across missions i.e. mission customisation should not involve massive modifications
Standards	Whenever possible, widely used (often de-facto) standards should be utilised
COTS products	Whenever cost-effective and technically suitable, commercial off-the-shelf (COTS) should be used.
open source software	Open Source software products have the advantage that the source is available, can be changed or adapted and removed dependency on the product release policy (e.g. as concerns platform migration) of COTS vendors.

Table 2.2 ECSS-E-70 Requirements for Ground Systems Software

It should be noted that the ground system operation may include the following aspects of software development related to the space segment:

- On-board software management
- On-board software maintenance

The software development processes defined in this guide also cover the engineering of On-board Software Management software. On-board software maintenance, to maintain the space segment on-board software, shall be performed in accordance with the requirements of space segment software.

2.6 PROCESSES, ACTIVITIES AND TASKS

A process is a set of interrelated activities that transform inputs into outputs [Ref. 1]. This guide describes those processes that transform initial customer requirements into a software product.

Each process is divided into a set of activities. The activities describe the main operations to be carried out in a process. Each activity is further divided into a set of tasks. The tasks are the requirements that are normally applied to all projects developing ground segment software. These requirements are defined in the ECSS-E-40 [Ref. 11] but they can be tailored for the needs of a specific project, as discussed in chapter 3.

2.7 THE SOFTWARE LIFE CYCLE PROCESSES

The processes for developing ground segment software are:

- Systems Engineering for Software
- Software Requirements Engineering
- Software Design Engineering
- Software Verification and Validation

These processes cover the lifecycle of a ground segment software product from the initial customer requirements through to

implementation, installation and acceptance of the software product. Once the software product has been developed, it is required to be

- operated (Software Operations Engineering)
- maintained (Software Maintenance)

throughout its useful life.

The System Engineering process may identify a number of software products to be developed, as described in section 2.4. Each software product will be developed according to the above processes but these individual products will require a process of integration and validation to form the system. The operation of the software products may be dependent on the operational aspects of the overall system.

2.7.1 System Engineering for Software

The System Engineering for Software process is the responsibility of the customer. The customer is responsible for the delivery of a system in which the developed software will be integrated. In practice, however, the supplier may carry out many of the key activities and tasks on behalf of the customer.

In the majority of ground segment cases, a software product is in fact part of a larger ground segment system or element of that system. The requirements for system engineering are fully specified in [Ref. 13]. The System Engineering for Software process is restricted to those aspects of the system engineering process that require the introduction of additional requirements specific to software development. The System Engineering for Software process ensures that the customer requirements for software are complete, unambiguous and properly express the customer's needs.

The documentation of the system level requirements is a prerequisite to the requirements engineering for the software.

This process ensures that the system requirements for software are fully specified and documented. Activities from the software life cycle processes may be required during system requirements analysis, for example the technique of constructing software prototypes is often used to clarify or help the understanding of system requirements.

This process also establishes the top-level partitioning of the system between the various elements e.g. hardware, software and manual operations. All system requirements are allocated to specific items. The systems requirements allocated to software are documented in the requirements baseline. The specific interface requirements are documented in the Interface Requirement Document.

2.7.2 Software Requirements Engineering

This process is concerned with the establishment of the requirements of the software product, along with the elaboration of the interface requirements. Each of the software products identified in the System Engineering process will undergo Software Requirements Engineering. The Software Requirements Engineering process is the responsibility of the supplier.

Software Requirements Engineering is the process that bridges the gap between system level software partitioning and the software design process. Requirements analysis enables the specification of software function and performance, the definition of the software's interface with other software or other parts of the system. It will also establish the design constraints that the software must meet. The process also provides the representation of the information and function that can be translated into data, architectural and detailed design.

The requirements for each of the software products are normally described in a technical specification and an interface control document. The technical specification contains a precise and coherent definition of functional and performance

requirements. The interface control document contains the specification for interaction with external systems, which may be other software products or other systems.

During this process, cost, schedule and implementation plans are written. These are addressed in Part B of this Guide.

All significant trade-offs, feasibility analysis, make/buy decisions and supporting technical assessments are documented in the design justification file.

This process also includes preparing the architectural design of the software product, i.e. top-level structure and the software components meeting the software requirements. These are described in the design definition file. The process will also identify the top-level design for the external interfaces (i.e. to other software or systems) and internal interfaces (i.e. between the software components of the software product). These are described in the interface control document.

2.7.3 Software Design Engineering

Software Design Engineering activities includes the detailed design and the coding of the software product. The process is also concerned with the unit, integration and system testing activities for each software product. Software Design Engineering is the responsibility of the supplier.

This process produces the design for each element in the software product tree. All elements of the software design are documented, in the design definition file (DDF), including the source code.

The documentation produced should also include description of the rationale for design decisions and of the analysis and test approach to show that the design meets the requirements. This will be held in the design justification file (DJF).

For large software developments, in which the software has been partitioned into smaller subsystems, the software design engineering process will also include the integration of the individual subsystems into the complete software system.

2.7.4 Software Verification and Validation

Verification [Ref. 1] determines whether the outputs of an activity fulfil the requirements or conditions imposed on them in the previous activities. Validation determines whether the requirements, and the final as-built software product, fulfil their specific intended use. In essence, this process confirms that the customer's needs are properly expressed as requirements, that all requirements are met and that the design constraints are respected.

The Software Verification and Validation process runs concurrently with the systems engineering, requirements engineering and design engineering processes and activities within the process are included in the descriptions of these processes. At the end of the Software Design Engineering process, however, specific verification and validation activities are undertaken. These activities, described in the Software Validation and Acceptance process, include the transfer of the software to the customer and the subsequent acceptance of the software by the customer. Acceptance requires the formal evaluation of the software product in its operational environment, which is carried out after the software has been transferred.

2.7.5 Software Operations Engineering

The Software Operations process commences after the acceptance of the software product in its operational environment. Since the software product forms an integral part of the ground segment, the phasing and management of operations will be determined by the system level needs. The Software Operations Engineering process is not directly connected to the

overall mission phases but rather is governed by the need to operate the software product at a given time.

2.7.6 Software Maintenance

This process is concerned with the controlled modifications to code and associated documentation due to a problem or the need for improvement or adaptation. The process ends with the retirement of the software product.

2.8 GROUND SEGMENT SYSTEM ENGINEERING

2.8.1 Ground Segment System Engineering Phases

In ECSS-E-70 [Ref. 19], ground segment engineering is partitioned into phases A to F that include the activities described later in this section. The ground segment life cycle phases are not necessarily concurrent with those of the space segment, although there is extensive interaction between the two. Furthermore, other project life cycle models may be used for the development of individual system elements – this section focuses particularly on the implications for elements containing software.

For each ground segment phase, ECSS-E-70 [Ref. 19] identifies the main activities and their inputs and outputs, and the major review(s). The list of activities, major reviews (formal phase transitions) and products are summarised in various tables and, whenever applicable, reference to the relevant Document Requirements Definition (DRD) is also made.

A brief overview of the Ground Segment life cycle phases activities is given below:

- Feasibility Studies and Conceptual Design (Phase 0/A): During this phase, the requests initially expressed by the Space System Customer (SSC) are analysed in order to identify and characterise the ground segment, in terms of operational feasibility and needs, expected performance and RAMS objectives (Reliability, Availability, Maintainability, and Safety).
- Preliminary Design (Phase B): This phase is to achieve a precise definition of the ground segment baseline. During the preliminary design phase the ground segment is decomposed into its main elements.
- Design (Phase C): This phase is to complete the design of the ground segment to element level and to start implementation. In addition, this phase also includes the definition of the operations organisation and the start of production of mission operations data (operational procedures, database) and detailed mission analysis.
- Production (Phase D): This phase is to procure all ground segment facilities and elements and to integrate them into an operational ground segment that is ready to support the in-orbit operations and exploitation of the space segment. It is composed of three main sub-phases that correspond to:

  1. Production/procurement,
  2. Technical verification and validation, the integration and technical validation of its major constituent elements, the main objective of which is to confirm the compliance of the ground segment with the specifications and its compatibility with the space segment and the external entities
  3. Operational validation, complementing the technical validation by involving the operations organisation (i.e. personnel and procedures), in order to verify that the overall ground segment is able to support the mission. Acceptance, which requires the formal evaluation of the software product

in its operational environment, is carried out during operational validation.

- In-orbit Operations (Phase E): This phase is to operate and exploit the mission during all in-orbit operations. For this it is necessary to maintain the ground segment in accordance with the maintenance plans and to correct any satellite anomalies that may occur.
- Mission Termination (Phase F): During this phase the space segment is withdrawn from service, after preparation and planning in liaison with the space segment customer and according to international regulations. This may include transferring its constituting spacecraft to another orbit.

2.8.2 Mapping ECSS-E-40 onto GS System Engineering Phases

The mapping of Engineering phases to the Ground Segment Software development lifecycle is individually defined for each project and is independent of the Space Segment software lifecycle mapping. Figure 2.1 shows a mapping of ECSS-E-40 Software Development Processes [Ref. 11] to ECSS-E-70 Ground Segment Life Cycle Phases [Ref. 13]. The mapping introduces an additional review, the Software Requirements Review (SWRR). This is permitted by ECSS-E-40, since ECSS-E-40 prescribes the minimal set of reviews and does not forbid the insertion of intermediate reviews.

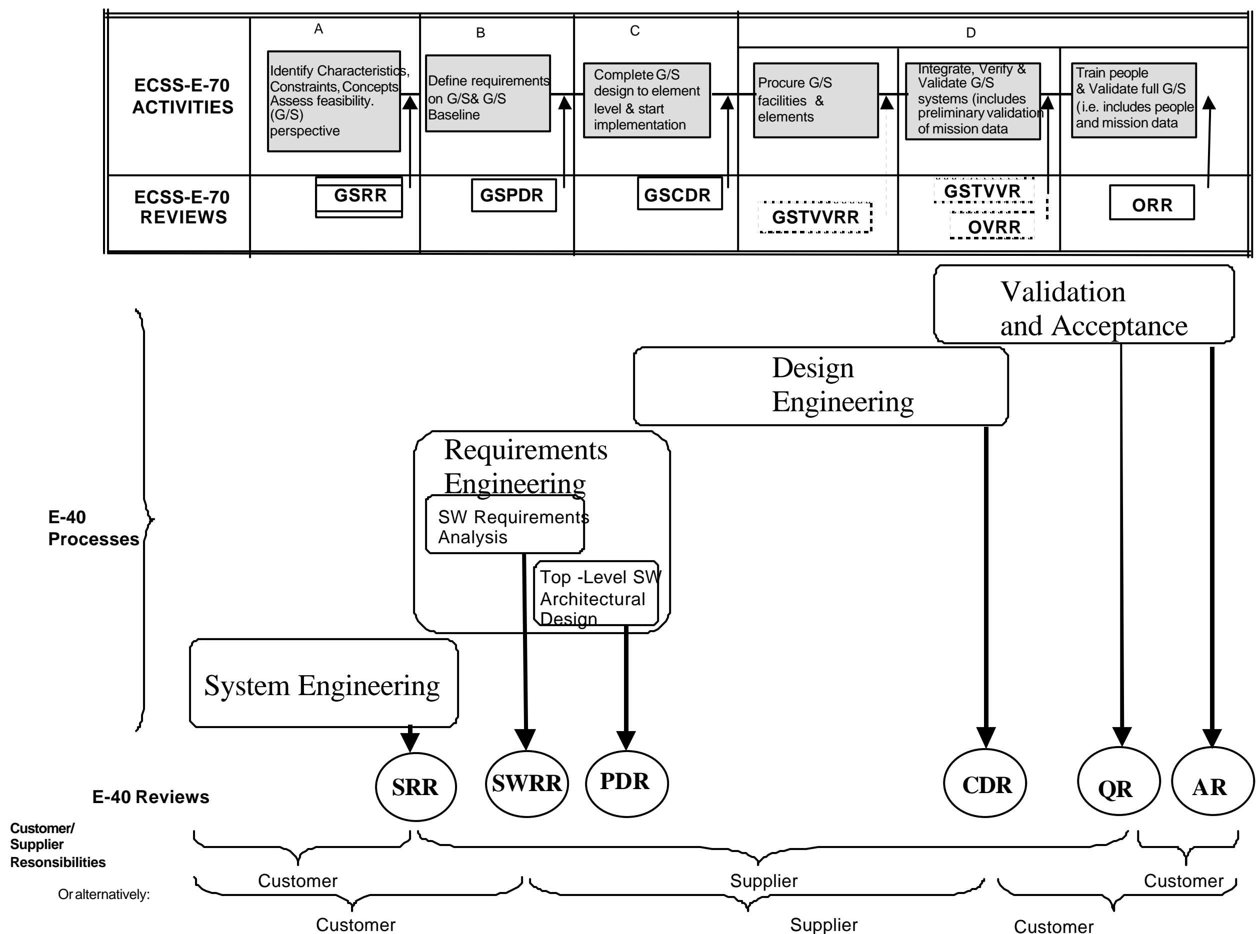

Figure 2.1 Mapping of ECSS-E-40 (Phases A-D) to ECSS-E-70

Figure 2.2 shows a mapping for the remaining phases.

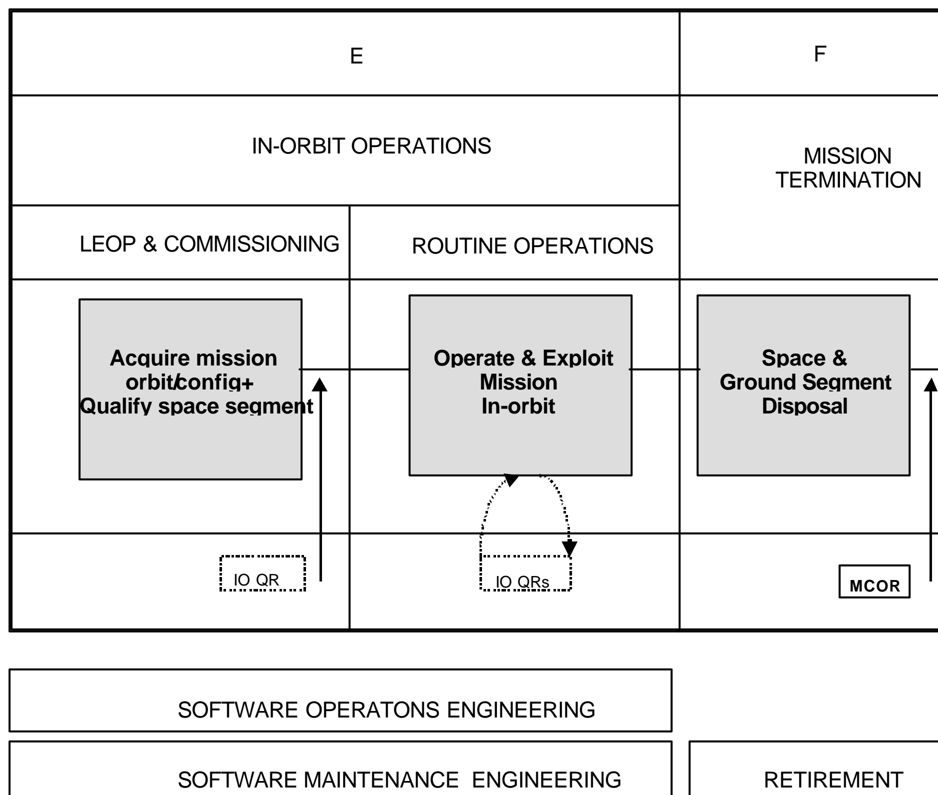

Figure 2.2: Mapping of ECSS-E-40 to ECSS-E-70 (Phases E - F)

Note also that in figure 2.1, operational validation is, in effect, done at system level (see Chapter 9).

Figure 2.1 also shows separation of responsibilities between customer and supplier. This need not imply a single supplier. For example requirements engineering and design engineering may be carried out by different suppliers (although this has the disadvantage that there will be more requirements interpretation problems by the supplier doing the design engineering and possibly less cost awareness on the part of the supplier preparing the requirements).

2.9 SUPPORTING THE SOFTWARE ENGINEERING PROCESS

This Guide covers the engineering processes required to develop ground segment software products, in accordance with the E-40 standard [Ref. 11]. A number of supporting activities to assist in the management and product assurance of the software engineering process are defined in the ECSS-M [Ref. 3 to 9] and ECSS-Q [Ref. 10] series of standards.

Part B of this Guide covers the following aspects of management of the software development.

2.9.1 Documentation

The purpose of the documentation process is to record information produced by a lifecycle process. This process is concerned with the planning, production, distribution and maintenance of all documentation from the development project. The document process takes account of the requirements of ECSS-M-50 [Ref. 8].

2.9.2 Configuration Management

The purpose of the Configuration Management process [Ref. 1] is to apply administrative and technical control throughout the software lifecycle to:

- identify, define and baseline software items in a system
- control modifications and releases of the items
- record and report the status of the items and modification requests
- ensure the completeness, consistency and correctness of the items
- storage, handling and delivery of the items.

The configuration management aspects for the internal control of the software product development will be compatible with the requirements of ECSS-M-40 [Ref.7]. These aspects will also

be used to assist the Software Maintenance process, described in Chapter 10.

The configuration management process is also important to the management of the interfaces to the other system components. Interface management procedures shall be defined in accordance with ECSS-M-40 requirements. The aim is to define procedures that guarantee the consistency of the system interfaces.

The interface management procedures for a particular project are normally documented in the customer requirements baseline (RB). Where appropriate, specific configuration management documentation may be produced.

2.9.3 Software Product Assurance

The purpose of the Software Product Assurance process is to provide adequate assurance that the software product and processes in the project lifecycle conform to their specified requirements and adhere to the established plans. Quality Assurance may be internal or external depending on whether evidence of product or process quality is demonstrated to the management of the supplier or the customer.

The guidance in this section shall be primarily derived from the ECSS-Q series of standards, in particular ECSS-Q-80 [Ref. 10].

2.9.4 Software Project Management

The ECSS-M standards define the requirements to be applied to the management of space projects. Although many of the practices will be directly applicable to ground segment software projects, the standards will be tailored to suit the particular nature of software development.

This page is intentionally left blank

CHAPTER 3
SYSTEM ENGINEERING FOR SOFTWARE

3.1 INTRODUCTION

The System Engineering for Software process may be called the 'problem definition' process. The purpose of the process is to determine the system level aspects of the ground segment that can have a bearing on the software development.

The definition of the system requirements shall be the responsibility of the customer. The expertise of the software engineers, hardware engineers and operations personnel should be used to help refine and review the system requirements.

An output of the process is contained within the requirements baseline (RB). The RB comprises a number of distinct components, as described in Appendix A, but will be referred to as a single document for convenience. The RB will contain the customer requirements. The customer requirements are a critical item for the whole software project because they define the basis upon which the software is accepted.

The other principal output of this process is the Interface Requirements Document (IRD), which specifies the external interfaces for the overall system and the interfaces between the constituent parts of the system. The IRD is conceptually part of the RB but will normally be held as a separate chapter or volume, as appropriate.

The system engineering process terminates with the formal customer approval of the requirements baseline by the System Requirements Review (SRR).

3.2 PROCESS INPUTS

There may be no formal inputs to this process, depending on the definition of the system. In the case where the system under development is part of a higher level system, there is likely to be formal specification describing the key requirements of the components and their interfaces. In the case where no higher level specification exists, the results of interviews, surveys, studies and prototyping exercises are often helpful in formulating the system requirements.

3.3 ACTIVITIES

The main activity of the system engineering process is to capture the customer requirements and document them in the RB. The scope of the system has to be established and the interfaces with the external systems identified. The nature of the system will be dictated by its level in the product hierarchy, as discussed in chapter 2. At the higher levels, the system engineering process will be concerned with the partitioning of functionality between hardware, software and manual operations. At the lower levels of the partitioning, the system engineering process may be concerned with the partitioning and interfacing of purely software components.

Ground segment software is often operated by operations personnel; it follows that the requirements baseline will reflect both system requirements and user requirements. For example the RB for a mission control system will contain requirements relating to (interfaces to) the other system elements (e.g. the ground stations and the communications) and to the operators and spacecraft engineers that use it

The System Engineering process is fully described in ECSS-E-00 [Ref. 15], while the details of the system engineering discipline are described in ECSS-E-10 [Ref. 16]. The purpose of this guide is not to explain all aspects of systems engineering but rather to provide

guidance on the software aspects of the Systems Engineering process. When dealing with software components, the System Engineering process is composed of the following activities:

- System Requirements Analysis
- System Partitioning
- System Level Requirements for Software Verification and Validation
- System Level Requirements for Integration of Software
- System Requirements Review

3.3.1 System Requirements Analysis

The key aspects of the System Requirements Analysis activity are to identify the customer-defined objectives and goals for the product and then proceed to model these requirements in a manner that allocates them to a set of engineering components - software, hardware and people.

Once function, performance, constraints and interfaces are bounded, the next task is allocation, assigning functions to one or more engineering components.

3.3.1.1 System requirements specification

In order to document the systems requirements, it is necessary to analyse the specific intended use of the system.

The key aspect of this task is to establish a set of overall objectives, which the system must meet. These should not be expressed in terms of the systems functionality but rather should define why the system is being procured for a particular environment.

From the objectives of the system, the requirements of the system shall be derived. The following types of requirements are normally addressed:

3.3.1.1.1 Functional Capability requirements

The system shall be defined in terms of the functional capabilities it supplies to its users. Functional capability requirements describe 'what' the users want to do. This requirement addresses the capability of the software to provide functions which meet stated and implied needs when the software is used under specified conditions.

A functional capability requirement should define an operation, or sequence of related operations, that the system will be able to perform. The functional capability requirements should be expressed quantitatively in terms of:

- Capacity - how much of a capability requirement is needed at a given time
- Speed - how fast the complete operation, or sequence of operations, is to be performed
- Accuracy - the difference between what is intended and what happens when an operation is carried out

3.3.1.1.2 Non-functional requirements

The non-functional system properties, such as reliability and safety, shall be defined. Non-functional requirements place restrictions on how the requirements are to be met. There are a number of ways of expressing non-functional requirements but normally the following attributes listed in Table 3.1 would be considered.

Attribute	Meaning
Usability	How the system will run and how it will communicate with human operators.
Efficiency	The upper limits on physical resources such as processing power, main memory, disk space, etc.
Maintainability	The ease by which faults can be corrected and software can be adapted to meet new requirements.
Portability	The ease by which a system can be moved from one environment to another
Reliability	The acceptable mean time interval between failures of the software, averaged over a significant time period.
Security	The capability of the system against threats to its confidentiality, integrity and availability
Safety	The ability to deal with potential problems such as hardware or software faults

Table 3.1 Non-functional characteristics

3.3.1.1.3 Excluded characteristics

The principal objective of the system engineering process is to specify what the system should do, but it is sometimes necessary to specify what the system must not do. Where ever possible, these excluded characteristics should be expressed in terms of capability or non-functional requirements. For example, the system may be developed to operate on a number of different platforms but a specific platform may be excluded. As another example, a system may be developed against a particular standard but certain elements of that standard may not be addressed.

All such excluded characteristics shall be explicitly stated.

3.3.1.1.4 Interface requirements

All external interfaces to the system shall be defined. An interface is a shared boundary between two systems and it may be defined in terms of what is exchanged across the boundary. External interfaces state how interactions with other systems or system components must be done.

The interface requirements are documented as part of the requirements baseline. They are normally documented in an Interface Requirements Document, which is part of the RB.

3.3.1.1.5 Software engineering methods and tools

All software development projects shall follow the process model specified in this guide but, where appropriate, defined methods and coding standards may be specified for the activities and tasks within the process model.

In the case where the customer requires specific software engineering methods and tools to be applied during the development, these shall be defined in the requirements baseline.

3.3.1.2 Criticality Analysis

The overall safety and reliability requirements for the system shall be defined. All critical functional aspects of the software shall be identified and, without introducing undesirable software complexity, the number of critical components shall be minimised. The results of the criticality analysis shall be documented in the requirements baseline.

Criticality for ground software can be considered in the following categories:

1. Software which could cause danger to a spacecraft pre-launch. This applies to EGSE software and to mission software used in System Validation Tests and other test activities with the spacecraft.
2. Software used to control the launcher: this software may be safety critical.
3. Software that could endanger a mission if it does not work correctly, e.g. if it produces incorrect results. This software could also be safety critical (e.g. if used in manned missions).
4. Software that needs to be working correctly during important or critical operations.

Appropriate use should be made of criticality analysis methods, such as:

- Software Failure Modes, Effects and Criticality Analysis (software FMECA)
- Software Common Mode Failure Analysis (Software CFMA)
- Software Fault Tree Analysis (Software FTA)

The supplier shall define and apply measures to assure the reliability of critical component. These measures may include:

- use of software design or methods which have performed successfully in a similar application;
- failure-mode analysis of the software, with the insertion of appropriate features for failure isolation and handling;
- defensive programming and restrictions on language features used;
- use of formal design language for formal proof;
- unit testing shall be re-run on the final code if instrumentation was used;
- test coverage of all decision branches;
- full inspection of source code;
- witnessed or independent testing;
- analysis of failure statistics
- removal of deactivated code.

Some techniques for increasing reliability/availability of the software include:

- Use of a back-up system that can take control without loss of data and within a given time
- Distribution of tasks/functions over platforms
- Separation of real-time and off-line functions

Software connected with commanding can endanger a mission if malfunction occurs. Examples of this kind of software are commanding software, mission planning software, Orbit and attitude manoeuvre software, On-board software management software and automatic command procedure software. The following techniques have been developed to provide safety for this kind of operations:

- Commands are subject to individual pre-transmission checks
- Manually sent commands must be confirmed ("arm and go")
- Critical commands or sequences are subject to a higher level of authorization
- Complex sequences of commands (also automatic command procedures) are subject to verification using a simulator.
- Commands identified as forbidden are filtered out by the system (the filtering system should be time-dependent, since commands may only be dangerous in certain states of the system).

Any automatic command procedures stored as part of the mission database have to be validated like software and put under configuration control. The same procedures apply to any part of the software that is driven by scripts. The possibility of human errors during operation should also be taken into account and the software should be able to detect and reject erroneous input and behave in a reasonable way if errors occur.

3.3.2 System Partitioning

A top-level partitioning of the system shall be established. The partitioning, derived from the system specification above, should include hardware, software and human operations as appropriate to the system. All system requirements shall be allocated to the different system components.

For almost all systems, there are many possible solutions that may be developed. These cover a range of solutions with different combinations of hardware, software and human operations. The solution chosen for further development should be the technical solution that meets the requirements most cost-effectively.

This activity shall ensure that the following tasks are carried out:

- Partition requirements to appropriate subsystems
- Define subsystem interfaces

The partitioning shall be documented in the requirements baseline. The subsystem interfaces are described in the Interface Requirements Document, a part of the requirements baseline.

3.3.2.1 Partition requirements to appropriate subsystems

The requirements should be collected into related groups. The different alternatives should be identified.

The different subsystems that make up the system shall be identified. The subsystem identification should be driven by the requirements but it may also be affected by other organisational and environmental factors.

All requirements shall be assigned to the identified subsystems. Functional requirements shall be allocated to (i.e. executed in) only one subsystem, except for critical functions that need to be implemented redundantly. If appropriate, non-functional requirements may be allocated to all subsystems to ensure, for example, that all subsystems have comparable quality.

The partitioning of the subsystem components allows the identification of hardware and software configuration items.

Traceability of the system requirements to the subsystem partitions shall be demonstrated. This will be retained in the design justification file (DJF).

3.3.2.2 Define subsystem interfaces

The interfaces that are provided and expected by each subsystem shall be defined. Once these interfaces have been agreed, parallel development of the subsystems becomes possible. The subsystem interfaces shall be defined in the Interface Requirement Document (IRD).

3.3.3 System Level Requirements for Software Verification and Validation

The aim of this activity is to ensure that the customer's verification and validation requirements, at the system level, are identified. In particular, the activity shall ensure that the requirements for software acceptance are addressed and documented in the requirements baseline.

In order to ensure that software acceptance can be successfully achieved, each requirement shall be defined such that it is possible to:

- Check that the requirement is incorporated in the software subsystem
- Verify that the software will implement the requirement
- Test that the software does implement the requirement.

3.3.4 System Level Requirements for Software Integration

3.3.4.1 Software observability requirements

The aim of this activity is to ensure that, when a software product is composed of different modules, all the necessary

software observability requirements are addressed and documented in the requirements baseline.

Software observability is the property of a system for which monitoring of visible variables (i.e. those available to the system integrator) is always sufficient to determine the state of the system. In other words, the system integrator should be able to isolate errors by looking at the variables exported at the interface (available to him), without knowing the internal details of each module.

The customer shall ensure that all such requirements are documented to facilitate the integration of the software product into the system.

The following methods are commonly employed in ground segment software to achieve observability:

- Operator interaction: the operator is informed at his operations interface about progress of the executing software and warned about occurrence and location of abnormal conditions.
- Error monitoring and reporting: error conditions are monitored and immediately reported. The system can then start logging and recovery procedures or gracefully terminate the application.
- Exception catching: similar to error monitoring. Abnormal conditions trapped by the hardware or operating system trigger the execution of a piece of code that will record the state of the system and start recovery procedures (or gracefully terminate the application).
- Debug-only code: error-checking code (usually slow to execute) can be activated by running the system in debug mode or by setting specific flags.
- Initialization of subroutines: Some critical variables are put into a known state before starting a subroutine. This can be used to validate the output of the subroutine, to temporarily remove the functionality of the subroutine or to simplify the algorithm that produces the output.
- Keep alive techniques: a module or communications protocol is continuously monitored in order to detect if its execution halts.

3.3.4.2 Interface requirements

The customer shall specify all the interfaces between the software and the system, in order to facilitate integration of the software product into the system. The interfaces shall address both the nominal and degraded modes (i.e. behaviour in case of failure) of operation. These requirements shall be documented in the Interface Requirements Document (IRD).

The customer shall ensure that the data media requirements for each interface are specified.

3.3.4.3 Development constraints

Although the majority of the constraints on the system will normally be identified during the system analysis activity, there may be constraints placed on the software development due to the requirement to integrate the software with the system. These constraints may include:

- The operating system
- The COTs used
- The software development environment

The customer shall document all such constraints in the requirements baseline.

3.3.4.4 System Level Integration of Software

System level integration of ground segment software is particularly important. This ensures that the different elements work together properly. This is ensured in successive system level integration tests, culminating in end-to-end test involving complete chains of elements. Tests should check out:

- flow-control between elements,
- network capacity,
- that proper end-to-end functioning of protocols become evident,
- error recovery, e.g. switching to redundant links in case of link failure; for this it is important to be able to provoke error cases (e.g. reduction of link bandwidth, injection of bit errors or synchronisation errors),
- support of realistic operational scenarios (e.g. combinations of real-time and playback links with tele-command activity),
- duration tests are also important at system level.

3.3.5 System Requirements Review

The output of the System Engineering for Software process shall be formally reviewed during the System Requirements Review (SRR). In the case where the customer's product is an integrated hardware and software product, this review shall be performed in accordance with the ECSS system engineering standards [Ref. 12]. In cases where the customer's product is a software product, this should be a Joint Review. Participants should include the customer, operators, developers (hardware and system engineers) and the managers concerned. Part B of this guide describes the mechanism for conducting a Joint Review.

The successful outcome of the review is the establishment of the requirement baseline. This represents the customer's needs towards the system to be developed.

3.4 PROCESS OUTPUTS

3.4.1 Requirements Baseline

This document, containing a number of components, expresses the customer's needs. The requirements baseline (RB) shall always be produced before a software project is started.

Change control of the RB should be the responsibility of the customer. Change of the RB may well affect the specification of some or all of the software products within the system. The management of the RB and other system level documentation is addressed in Part B of this guide.

3.4.2 Interface Requirements Document

The interface requirements document (IRD) expresses the customer's interface requirements for the software to be produced. It is mandatory in all cases where the software product is intended for integration with the customer's hardware or software products. Depending on the project's size and nature, the IRD can be separate chapters or separate volumes of the RB.

3.4.3 Design Justification File

The design justification file (DJF) is generated and reviewed at all stages of the development and review process. It contains the documents that describe the trade-offs, design choice justifications, test procedures, test results, evaluations and any other documentation called for to justify the design of the product. The DJF is the primary input for the Qualification and Acceptance Reviews and acts as supporting input to other reviews.

At the end of the system engineering process, the DJF would contain the traceability to system partitioning and the results of the SRR milestone review.

CHAPTER 4
SOFTWARE MANAGEMENT

4.1 INTRODUCTION

The practices for addressing software management requirements are covered in Part B of this guide. The majority of the requirements for the management of software projects are covered by the ECSS M series and Part B provides a mapping of these requirements for the effective management of ground segment software development.

This chapter deals with a subset of the project management issues. The primary issue is the selection of an appropriate life cycle model to organise the software processes for a particular project.

The requirements for technical budget and margin management are also defined.

4.2 PROCESS INPUTS

The normal input into this process is the definition of the customer requirements in the requirements baseline.

4.3 ACTIVITIES

4.3.1 Planning

Software engineering activities shall be systematically planned and carried out. Plans shall be developed to cover:

- Development
- Configuration and Document Management
- Verification and Validation
- Maintenance
- Software Quality Assurance on Process and Product.

For further details of these plans please refer to Part B.

4.3.2 Selection of the Software Life Cycle Model

The organisation of the primary processes on a time base is known as a life cycle model. A life cycle model defines a project into a sequence of phases, which relate the processes to a time base. A life cycle model defines the processes, activities and tasks that occur within each phase and the relationships between each of them. A number of life cycle models exist, but they all share the software development processes summarised in chapter 2.

ECSS-E-40 provides a process model with processes that sometimes overlap in time. Establishing a project involves instantiating the process model and identifying dependencies, thus creating project phases.

The life cycle model chosen is specific to the requirements of the project, but it must address the software processes defined in ECSS-E-40. The possible lifecycle models are discussed in more detail in ECSS-E-40-4. [Ref 28] Each life cycle model is characterised by its reviews and deliverables. Reviews are used to mark progress within a project. These reviews normally mark the approval of specific deliverables.

The software engineering processes must occur whatever the size, the application, the hardware, the operating system or programming language used. Each of these factors, however, influences the development approach and the style and content of the deliverable items. The life cycle model chosen is specific to

the requirements of the project but it must address the software processes identified in chapter 2.

Normally the deliverables of each process must be reviewed and approved before proceeding to the next process, although this depends on the life cycle model chosen for the project. For example, the design engineering process should not start before the customer requirements have been defined but it may commence before the architectural design, part of the software requirements engineering process, has been agreed. The project must balance the additional risks introduced by this approach with the possible saving in development time.

There are six major reviews that mark progress in the software life cycle. These reviews are:

- System Requirement Review (SRR), which marks approval of the requirements baseline
- Software Requirements Review (SWRR), which marks the customer's agreement that all requirements with respect to the RB are captured in the software requirements specification
- Preliminary Design Review (PDR), which marks approval of the technical specification and the software architectural design
- Critical Design Review (CDR), which marks approval of the detailed design, source code and the results of testing
- Qualification Review (QR), which marks approval of the software against the technical specification
- Acceptance Review (AR), which marks acceptance of the software against the intended operational environment

These reviews have been selected as the minimum necessary for a workable contractual relationship. They are normally conducted at the completion of a key activity. They must be present in all projects and must include the customer. In long projects, additional milestones should be added to measure the progress of deliverables.

It should be noted that these review points are applicable to a single product. When a system or subsystem development includes a number of software products, each product will undergo the above reviews. There are also likely to be similar reviews that address the integration of the individual products.

The supplier shall define the lower-level software engineering methods and tools to be applied during the development. All software development shall adopt the process model in this guide but lower-level methods and tools, e.g. coding standards, shall be chosen to suit the development approach. These standards shall be approved, by the customer, as being fit for the application under development.

4.3.2.1 Standard Waterfall Model

The Standard Waterfall Model is essentially a once-through, do-each-activity-once approach. The key characteristic of the waterfall model is that the processes defined by this guide are organised in a sequential manner.

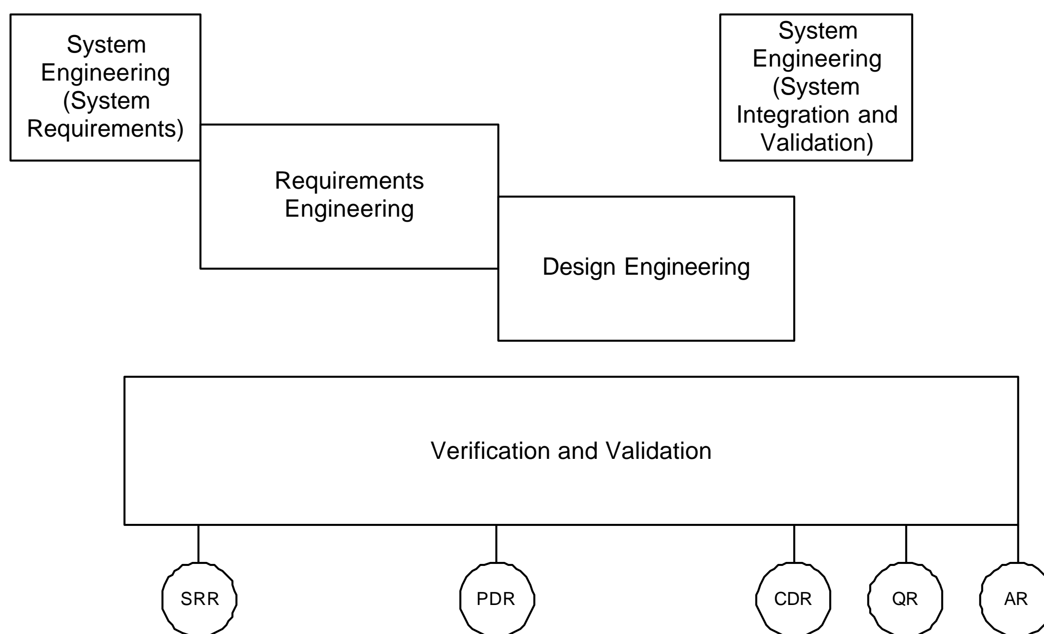

Figure 4.1: The waterfall model

Completion of a phase is achieved by means of a review of the deliverable outputs and their approval for use in the next phase

A phase can start prior to the completion of a previous phase but it must be recognised that this carries an associated risk. Errors or omissions may become visible at a later stage, requiring rework with an associated cost and schedule impact.

The waterfall model allows for a limited amount of iteration between phases, to allow for the correction of defects.

4.3.2.2 Incremental delivery model

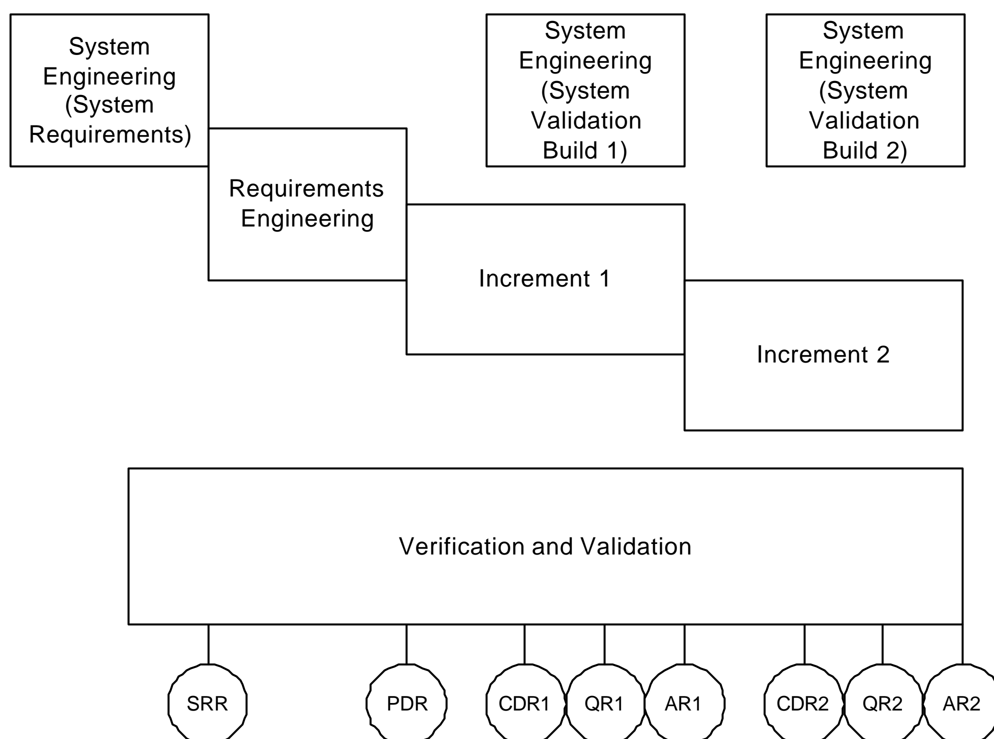

Figure 4.2: The incremental delivery model

The incremental delivery life cycle model starts with a given set of requirements and performs the development in a sequence of builds. The first build incorporates a part of the requirements, the next build adds more requirements and so on until the complete product is built. At each build, the necessary processes, activities

and tasks are performed e.g. software requirements engineering may be performed once, while the design engineering process is performed during each build.

In this approach, as each build is developed, the activities and tasks in the development process are employed in parallel with the operations and maintenance processes.

4.3.2.3 Evolutionary development model

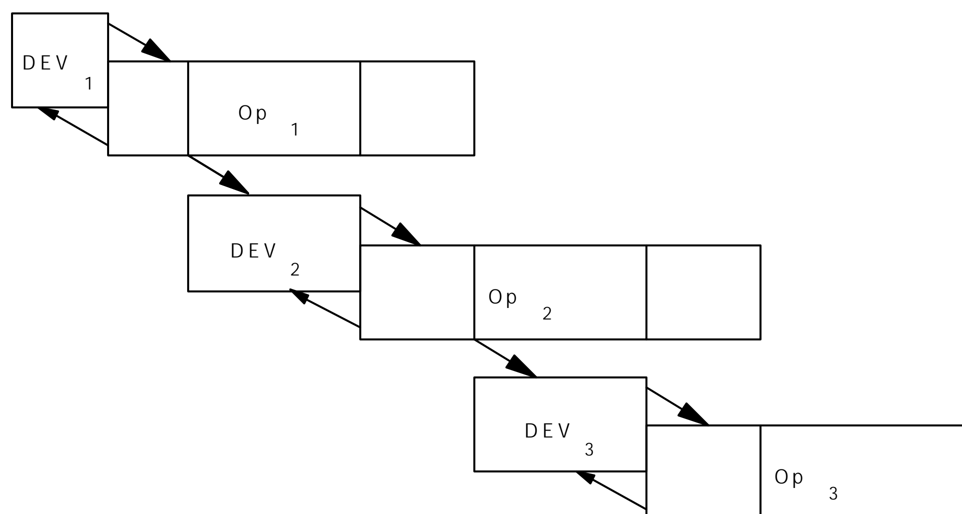

Figure 4.3. The evolutionary development model

The Dev. box is equivalent to Waterfall model in Figure 4.1

The Op boxes represents an operational phase, using a specific release

This approach is characterised by the planned development of multiple releases. All processes of the life cycle are executed to produce a release (this can include System Engineering depending upon whether all system engineering requirements are knowable at the start). Each release incorporates the experience of earlier releases. The evolutionary approach may be used because, for example:

- Some customer experience is required to refine and complete the requirements
- Some parts of the implementation may depend on the availability of future technology
- Some new customer requirements are anticipated but not yet known
- Some new customer requirements may be significantly more difficult to meet than others and it is decided not to allow them to delay a usable delivery

In an evolutionary development, the supplier should recognise the customer's priorities and produce the parts of the software that are both important to the customer and possible to develop with the minimal technical problems or delays.

The disadvantage of the evolutionary approach is that if the requirements are very incomplete at the beginning, the initial software structure may not bear the weight of later evolution. Expensive redesign may be necessary or, even worse, temporary solutions may become embedded in the system and distort its evolution. Customers may become impatient with teething troubles of each new release. In each development cycle, it is important to aim for a complete statement of requirements to reduce risk and to develop an adaptable design to ensure later modifiability. In an evolutionary development, all requirements do not need to be fully implemented in each development cycle but the architectural design should take account of all known requirements.

The Dynamic Systems Development Method (DSDM) is a more formal method for undertaking evolutionary development, which allows a management framework for undertaking the development.

4.3.3 Technical Budget and Margin Management

This section deals with the areas of computer resources (e.g. CPU load, maximum memory requirement) and performance requirements. The technical budget targets and the margin philosophy dedicated to the software shall be specified by the customer in the requirements baseline. The margin philosophy describes the rationale for margins allocated to the performance parameters and computer resources of a development. The margin philosophy also defines how these margins are managed during the execution of a project. The importance of this activity varies from system to system and the relative priority of different requirements.

For on-board systems, resource requirements (e.g. RAM, processor type) are fixed constraints on the design. For the majority of ground segments, there is more flexibility and these issues are normally addressed at the design stage. Potential solutions to performance concerns can be addressed by increasing hardware capacity or de-scoping the software functionality but the issues should be considered as design issues and explicitly addressed.

The supplier shall manage the margins regarding the technical budgets and present their status at each milestone, explaining how the margins were derived. Specific evaluation of these aspects shall be considered as part of the verification and validation issues at each of the key development processes. In particular, performance monitoring should be carried out whenever possible.

The results of the evaluation shall be documented in the design justification file and considered at the formal review points.

4.4 PROCESS OUTPUTS

The following plans are detailed in Part B and are a result of the planning process:

- Software development plan
- Software configuration management plan
- Software verification plan
- Software validation plan
- Software maintenance plan
- Software product assurance plan

The outputs from the review processes are fully defined in the chapters dealing with the appropriate processes.

The development life cycle model chosen for the project and the associated reviews shall be documented by the supplier.

The information shall be retained in the software development plan.

CHAPTER 5
SOFTWARE REQUIREMENTS ENGINEERING

5.1 INTRODUCTION

The Software Requirements Engineering process is concerned with the problem analysis and the top-level solution for the software products defined during the system engineering process. Each software subsystem may be a single software product or may contain a number of software products. This chapter defines the process for generating the software requirements for a software product. For a particular project, there may be a number of instances of this process, corresponding to the partitioning from the system engineering process.

The Software Requirements Engineering process bridges the gap between the system level software allocation and the software design engineering. Requirements engineering specifies software functions and performance and establishes design constraints.

There are two main activities involved in the process: requirements analysis and architectural design. The purpose of requirements analysis is to analyse the statement of system requirements, specified in the requirements baseline, to produce a set of software requirements that are as complete, consistent and correct as possible. The purpose of the architectural design process is to define a collection of software components and their interfaces to establish a framework for developing the software product.

The definition of the software requirements is the responsibility of the supplier. The participants in this activity include software developers, hardware engineers and operations personnel. They all have a different concept of the end product. These concepts must be analysed and then synthesised into a

complete and consistent statement of requirements about which everyone can agree.

The definition of the architectural design is the responsibility of the software engineers. Other kinds of engineers may be consulted during the process and representatives of the users and operations personnel should be involved in the review.

The primary output of the Software Requirements Engineering process is the technical specification (TS). As well as defining what the product must do, it is also the reference against which both the design and the product will be verified. Although 'how' aspects may have to be addressed, they should be eliminated from the TS, except for those aspects that constrain the software. Examples of constraints that may be placed on the development include the use of COTS, existing infrastructure and schedule.

The architectural design is documented in the design definition file (DDF). This should document each component and its relationship with other components. The DDF is complete when the level of definition of components and interfaces is sufficient to enable individuals or small groups to work independently in the Design Engineering process.

5.2 PROCESS INPUTS

The input to the Software Requirements Engineering process is the requirements baseline (RB).

5.3 ACTIVITIES

The Software Requirements Engineering process is composed of the following activities:

- Software Requirements Analysis
- Software Architectural Design
- Software Verification and Validation

5.3.1 Software Requirements Analysis

This activity is used to transform those system requirements specific to the software partitions, specified in the RB, into detailed software requirements specified in the TS. This is achieved by analysing the problem, as stated in the RB, and building a coherent, comprehensive description of what the software is required to do. The TS contains the supplier's view of the problem, rather than the customer's. This view should be based upon a model of the software product, built according to a recognised, documented method.

Software requirements may require the construction of prototypes to clarify or verify them. Requirements which cannot be justified by modelling, or whose correctness cannot be demonstrated in a formal way, may need to be prototyped. Man-machine interface requirements often require this kind of 'exploratory prototyping'. This is discussed further in Chapter 12.

The activity of software requirements analysis is composed of the following tasks:

- Establishing the software requirements
- Identification of each requirement
- Evaluation of the requirements

5.3.1.1 Establishing the software requirements

The supplier shall establish the software requirements. In order to determine the requirements, the supplier shall construct a logical model of the proposed software product. The logical model is an implementation-independent model of what is needed by the customer. In all but the smallest projects, CASE tools should be used for building a logical model.

In some projects, the software may be required to implement a specific algorithm or mathematical relationship. The logical model, in such circumstances, shall provide sufficient information to demonstrate to the customer that the supplier understands the algorithm. The logical model is a simplification of the details described below and, thus, an example of tailoring of this guide.

The logical model aims to capture the functional requirements of the software product and provides details of:

- the external entities outside of the software product
- the transactions between the external entities and the software product

The logical model aims to give the software product a robust and adaptable structure. The model normally has three main components, organised according to the development model used, to identify the following aspects:

- the transactions between the software product and the external environment
- the principal functionality of the software product
- the time-based behaviour of the software product.

The latter point is essential for some software products.

A good quality logical model should obey the following rules:

- Functions or operations should have a single, definite purpose. Names should have a declarative structure (e.g. 'Validate Telecommands') and say 'what' is to be done rather than 'how'. Good naming also allows design components with strong cohesion to be easily derived.
- Functions or operations should be appropriate to the level at which they appear (e.g. 'Calculate Checksum' should not appear at the same level as 'Verify Telecommands')
- Interfaces should be minimised. This allows design with weak coupling to be easily derived.
- The model should omit implementation information (e.g. file, record, task, module).
- The performance attributes of each function or operation (capacity, speed, etc) should be stated.
- Critical functions or operations should be identified. Critical software requirements are described in section 6.3.3.

The software requirements are obtained by examining the logical model and they should be rigorously described. Requirements must be described in text to ensure that they are testable. It is preferable, where possible, to supplement the textual requirements using non-textual means, such as by the use of diagrams. To facilitate this, the specification of the requirements should make use of the logical model, with the aim of minimising the supplementary text. This is particularly beneficial when CASE tools are used to assist in the creation of the logical model, as the output of the tool can usually be imported into the specification document.

The logical model should be consistent with any reuse requirements (e.g. should not conflict with the logical model of any reused software).

The requirements shall be derived from the system level software requirements, as described in Chapter 4, which identify a range of functional and non-functional attributes. The software

requirements should be organised in the following general categories:

- Characteristics of software products
- Interfaces Issues
- Security issues
- Safety issues

5.3.1.1.1 Characteristics of software products

Although there are many ways of describing the characteristics of software products, the use of an internationally agreed standard can assist. The use of the ISO definition of the characteristics of software products, [Ref. 14], leads to the classification shown in Table 5.1 below.

Requirement	**Description**
Functionality	Specify 'what' the software has to do i.e. define the purpose of the software. Specify numerical values for measurable variables (e.g. rate, frequency, capacity, speed). Performance requirements may be incorporated in the quantitative specification of each function or stated as separate requirements. Qualitative performance requirements are unacceptable. Specify the constraints on how the software is to be verified. They may include requirements for simulation, emulation, live tests with simulated and/or real inputs, and interfacing with the testing environment. Specify the constraints on how the function is to be validated.
Usability	Specify how the software will run and how it will communicate with human operators. Usability

Requirement	Description
	requirements include all user interface, usability and human-computer interaction requirements as well as the logistical and organisational requirements.
Efficiency	Specify the upper limits on physical resources such as processing power, main memory, disc space etc. These are especially needed when extension of processing hardware late in the lifecycle becomes too expensive, as in many embedded systems
Maintainability	Specify any project -specific requirements for the documentation. Specify how easy it is to repair faults and adapt the software to new requirements. The ease of performing these tasks should be stated in quantitative terms, such as the mean time to repair a fault (MTTR). They may include constraints imposed by the potential maintenance organisation. Maintainability requirements may be derived from the user's availability and adaptability requirements.
Portability	Specify the ease of modifying the software to execute on other computers and operating systems. Possible computers, other than those of the target system, should be stated.
Reliability	Specify the acceptable mean time interval between failures of the software, averaged over a significant period (MTBF). They may also specify the minimum time between failures that is ever acceptable. Reliability requirements have to be derived from the user's availability requirements.

Table 5.1 Types of software requirements

5.3.1.1.2 Interface Issues

Specify hardware, software or database elements with which the system, or system component, must interact or communicate. Interface requirements should be classified into software, hardware or communications interfaces. Software interfaces could include operating systems, software environments, file formats, database management systems and other software applications. Hardware interface requirements may specify the hardware configuration. Communications interface requirements constrain the nature of the interface to other hardware and software. They may demand the use of a particular network protocol. External interface requirements should be described or referenced in the ICD.

5.3.1.1.3 Security Issues

Specify the requirements for securing the system against threats to confidentiality, integrity and availability. These should describe the level and frequency of access allowed to authorised users of the software product. The level of physical protection of the computer facilities may be stated.

5.3.1.1.4 Safety Issues

Specify any requirements to reduce the possibility of damage that can follow from software failure. The safety requirements for the system components will have been identified from the system level analysis, as described in Chapter 4. Software should be considered safety-critical if the information it presents can lead to injury to people, affect the mission or damage property.

The safety requirements for software, which specify what would happen when failures of a critical piece of software actually do occur, will normally be identified at the system level. In deriving the software requirements, it must be ensured that these actions are traceable throughout the software and that appropriate verification and validation is undertaken.

5.3.1.2 Identification of Software Requirements

Each software requirement must be uniquely identified. Each requirement should include the attributes listed in Table 5.2 below.

Attribute	Description
Identifier	Each software requirement shall include an identifier, to facilitate tracing through subsequent processes.
Need	Essential software requirements shall be marked as such. Essential software requirements are non-negotiable; others may be less vitally important and subject to negotiation.
Priority	For incremental delivery, each software requirement shall include a measure of priority so that the developer can decide the production schedule.
Stability	Some requirements may be known to be stable over the expected lifecycle of the software; others may be more dependent on feedback from the design engineering process or may be subject to change during the software lifecycle. Such unstable requirements should be flagged.
Type	Identify the type of requirement, as specified in Table 5.1
Criticality	Each requirement should identify its criticality category, using the types in chapter 3
Verification method	The method of verification for each requirement should be given, using the guidance in this document
Source	References that trace software requirements back to the RB shall accompany each software requirement.

Table 5.2 Requirement Identification Attributes

As well as the attributes of requirements, as described in Table 6.2, the following aspects should be considered when identifying the requirements of a software product:

- Consistency of the requirements
- Duplication of the requirements

5.3.1.2.1 Consistency of software requirements

A set of requirements is consistent if, and only if, no individual requirement conflicts with another requirement. Clarity assists in ensuring that requirements are consistent. A requirement is clear if it has one, and only one, interpretation. Clarity implies lack of ambiguity. If a term used in a particular context has multiple meanings, the term should be qualified or replaced with a more specific term. There are a number of types of inconsistency, for example:

- Different terms for the same thing
- The same term used for different things
- Incompatible activities happening at the same time
- Activities happening in the wrong order

The achievement of consistency is made easier by using methods and tools.

5.3.1.2.2 Duplication of software requirements

Duplication of software requirements should be avoided, although some duplication may be necessary to make the TS understandable. There is always the danger that a requirement that overlaps or duplicates another will be overlooked when the TS is updated. This leads to inconsistencies. Where duplication occurs, cross-references should be inserted to enhance modifiability.

5.3.1.3 Evaluation of software requirements

The supplier shall evaluate the software requirements to ensure that all the criteria specified above have been addressed.

The software requirements related to safety, security and criticality shall be explicitly addressed. The use of suitably rigorous methods for evaluating these criteria is recommended.

The requirements shall be evaluated for traceability with the system partitioning requirements. For the TS to be complete, each requirement in the RB must be accounted for. A traceability matrix must be inserted into the design justification file (DJF) to prove completeness. Completeness has two aspects:

- No system requirement has been overlooked;
- An output has been specified for every possible set of inputs

The phrase 'To Be Defined' (TBD) indicates incompleteness. The nature of the lifecycle model chosen for the development will impact on the completeness of the requirements. For example, the waterfall model assumes that the requirements are complete before moving onto the next process, so there must be no TBDs in the TS. Other lifecycle models may be chosen when extraction of the requirements is more difficult and, in this case, TBDs may be perfectly acceptable. Subsequent activities in the project would be targeted towards completion of the TBDs.

The requirements shall be evaluated for verifiability. This means that it must be possible to:

- Check that the requirement is incorporated in the design
- Verify that the software will implement the requirement
- Latterly, test that the software does implement the requirement

5.3.1.4 Software Requirements Review (SWRR)

The outputs of the software requirements engineering process shall be formally reviewed during the Software Requirements Review (SWRR). This shall be a technical review and shall include the customer. The principal aim of the SWRR is to agree with the customer that all their requirements with respect to the RB are captured in the Software Requirements Specification within the

technical specification (TS). The successful completion of the review establishes a baseline for the architectural design of the software.

After the start of the software architectural design process, modifications to the software requirements can increase costs significantly. The software architectural design process should not be started if there are still doubts, major open points or uncertainties in the software requirements.

5.3.2 Software Architectural Design

This activity is concerned with development and evaluation of the architectural design of the software and its subsequent documentation in the DDF. The generation of the top-level design generally involves:

- Transformation of the requirements into an architecture
- Development of the interface specification
- Identification of test requirements
- Evaluation of the architecture
- Preliminary Design Review

A recognised method for software design shall be adopted and applied consistently in this activity. Where no single method provides all the capabilities required, a project-specific method may be adopted, which should be a combination of recognised methods.

5.3.2.1 Construction of the architectural model

The supplier shall construct an architectural model that describes the design of the software, using implementation technology. The architectural model shall be derived from the software requirements, described in the TS. In transforming the software requirements to an architectural model, design decisions are made in which requirements are allocated to software components, as defined in chapter 2, and their inputs and outputs defined. Design decisions should also satisfy non-functional

requirements, design quality criteria and implementation technology considerations. Design decisions should be recorded in the design justification file (DJF).

Modelling is an iterative process. Each part of the model needs to be specified and re-specified until a coherent description of each component is achieved. In all but the smallest of projects, CASE tools should be used for building the architectural model. They make consistent models easier to construct and modify.

The architectural model should be decomposed into components (e.g. classes, objects, programs, tasks, files, modules, etc) according to the chosen design method. There should be distinct layers within the architecture, with each component occupying a specified layer. Components in a given layer should provide services to the components in the layer above and use the services of the layer immediately below. Component definition should continue until:

- All software requirements have been allocated to components
- Components may be re-used or are small enough to be designed, code and unit tested

Layering assists the control of complexity by the use of 'information hiding'. By treating the lower layers as essentially 'black boxes', the information necessary to the internal workings of the lower layers can remain hidden.

The task of constructing the architectural model results in a set of components having defined operations and interfaces. The operations of each component will be derived from the RB. The level of design detail will show which requirements are to be met by each component but not necessarily how to meet them: this will only be known when the design engineering process is undertaken. The interfaces between components will be restricted to a definition of the information to exchange and not how to exchange it, unless this contributes to the success or failure of the chosen design.

For each component the following information shall be defined:

- inputs
- functions or operations to be performed
- outputs

The architectural design shall be documented in the design definition file. Although textual descriptions can be used, ideally the model should be represented diagrammatically. The architectural model should show, at each layer of the architecture, the interactions between the components. Descriptions of the design should be included where possible, using an appropriate technique. The diagramming technique used should be documented and referenced.

The proof that the architecture meets all the software requirements shall be retained in the design justification file (DJF).

The method of deriving the architectural model will depend on the design method chosen but attention should be paid to the following aspects.

5.3.2.1.1 Implementation of non-functional requirements

The design of each component should be reviewed against the non-functional requirements given in the TS. While some non-functional requirements may apply to all components in the system, other non-functional requirements may affect the design of only a few components.

5.3.2.1.2 Design quality criteria

Designs should be adaptable, efficient and understandable. Adaptable designs are easy to modify and maintain. Efficient designs make minimal use of available resources but care must be taken to balance this against other criteria such as ease of

understanding. Designs must be understandable if they are to be built, operated and maintained effectively.

Aiming for simplicity in form and function in every part of the design assists the attainment of these goals. There are a number of metrics that can be used for measuring complexity (e.g. number of interfaces per component, number of objects) and their use should be considered.

Simplicity of function is achieved by maximising the 'cohesion' of individual components (i.e. the degree to which the activities internal to the component are related to each other).

Simplicity of form is achieved by:

- Minimising the 'coupling' between components i.e. the number of distinct items that are passed between components
- Ensuring that the function a component performs is appropriate to its level in the architecture
- Matching the software and data structures
- Maximising the number of components that use a given component
- Restricting the number of child components to seven or less
- Removing duplication between components by making new components.

Design should be modular, with minimal coupling between components and maximum cohesion within each component. There is minimal duplication between components in a modular design. Components of a modular design are often described as 'black boxes' as they hide internal information from other components. It is not necessary to know how a black box component works to know what to do with it.

Understandable designs employ terminology in a consistent way and always use the same solution to the same problem. Where teams of designers collaborate to produce a design, permitting

unnecessary variety can considerably impair understandability. CASE tools, design standards and design reviews all help to enforce consistency and uniformity.

5.3.2.1.3 Trade-off between alternative designs

There is no unique design for any software system. Studies of the different options may be necessary. A number of criteria, which will depend on the actual application, will be needed to choose the best option.

There is, however, a cost and time issue when evaluating alternative designs. On a given project, this will effect the amount of time that can be spent on assessing an alternative design. The consideration of design quality, discussed above, will give some indication as to whether a specific approach is appropriate. When a well specified, cohesive and minimally coupled design solution has been derived, alternative designs should only be considered when there is a clear need.

Prototyping may be performed to verify the assumptions in the design or to evaluate alternative design approaches. This is called 'experimental prototyping'. For example, if a program requires fast access to data stored on disc, then various methods of file access could be coded and measured. Different access methods could alter the design approach quite significantly and prototyping the access method would become an essential part of the design process.

Only the selected approach shall be documented in the TS. The need for prototyping, code listings, trade-off criteria, reasons for the chosen solution, etc., should be documented in the DJF.

5.3.2.2 Definition of the Interfaces

The interfaces external to the software product and the interfaces between the components in the architecture shall be defined. The internal and external interfaces shall be documented in the ICD, although the format of this shall depend on the design

method chosen. The interfaces are normally specified as data structures. The definition of each interface shall include the:

- Service provided by the interface
- Description of each element (e.g. name, type, dimension) in the interface
- Range of possible values of each element
- Initial values of each element

Software often has special kinds of interfaces, for example:

- Man Machine Interface (MMI) or Human Computer Interface (HCI)
- Application Programmer Interface (API), which will describe the calls to a system providing services to an application. The ICD will be an API specification, describing the services and the signatures of the various calls to the services, these latter expressed in a suitable computer-readable language
- ASCII file interfaces – typically used for expressing limited amounts of data e.g. orbit state vectors, tele-command parameters for import to a control

The interface components will normally only be defined at a high-level at this point. Care should be taken to ensure that the interface design hides the implementation detail.

5.3.2.3 Software Integration Planning

The developer shall develop preliminary test requirements and plan for the software integration process.

Integration is the task of building a software product by combining its components into a working system. The integration task must be planned to ensure that components are integrated in a useful sequence. Integration testing should verify that the major components interface correctly. The approach to integration planning should be to ensure that the minimum time is spent in

testing, while ensuring that the software product is adequately tested.

The software integration plan should define the scope, approach, resources and scheduling for the integration task. This information shall be contained in the design justification file (DJF).

5.3.2.4 Evaluation of the Architecture

The developer shall evaluate the software architecture and the interface design. In particular, the following aspects shall be explicitly addressed:

- The design is correct, consistent and traceable with respect to the requirements
- The design implements the proper sequence of events, inputs, outputs, interfaces, logic flow, allocation of timing and sizing budgets, and error definition, isolation and recovery
- The design implements safety, security and other critical requirements correctly by the use of suitable rigorous methods

The results of the evaluation process shall be documented in the design justification file.

5.3.2.5 Preliminary Design Review (PDR)

The outputs of the software requirements engineering process shall be formally reviewed during the Preliminary Design Review (PDR). This shall be a technical review and shall include the customer. The principal aim of the PDR is to agree with the customer that all their needs with respect to the RB are captured in the TS. An additional aim of the PDR is to review the software architectural design. The successful completion of the review establishes a baseline for the development of a software item.

After the start of the design engineering process, modifications to the architectural design can increase costs significantly. The design engineering process should not be started if

there are still doubts, major open points or uncertainties in the architectural design.

5.3.3 Software Verification and Validation

All the requirements for verification and validation of the software product shall be documented.

The information regarding project criticality, the tools and techniques for verification and validation, and the items affected, shall be added to the DJF. The DJF shall also include the organisational aspects of verification and validation.

The following aspects of verification and validation shall be considered, and documented, at this point:

- Project level requirements
- Tasks
- Organisation

5.3.3.1 Project level requirements

The effort required for verification and validation shall be determined, as a function of the criticality of the software product. Risk, dependability and safety analysis shall be performed if any of the following aspects are traceable to the software product:

- Potential of an undetected error in the software requirement for causing death or personal injury, mission failure or financial or catastrophic equipment loss or damage
- The maturity of and risks associated with the software technology to be used
- Availability of funds and resources in the project for verification and validation

The evaluation of the criticality of the software product shall be documented in the design justification file.

5.3.3.2 Tasks

The verification and validation tasks suitable for the software product, given the assessment of the criticality of the software product, shall be identified. A number of possible verification and validation tasks may be used on a project, for example document reviews, inspections and testing.

The verification activities for each component, that are appropriate to the reliability and safety requirements, should be defined. For example:

- Design documentation must be internally reviewed
- Reused design components must be inspected

The methods and tools chosen shall be documented in the design justification file.

5.3.3.3 Organisation

The need for independent verification and/or validation shall be established as a function of the criticality of the software. If required, a qualified organisation shall be selected to perform the required processes.

The organisational requirements shall be specified in the technical specification.

5.4 PROCESS OUTPUTS

The main outputs of the requirements engineering process are the TS, the updated DJF, the DDF and the ICD.

5.4.1 Technical Specification

This document contains the developer's response to the requirements baseline.

5.4.2 Design Justification File

This file contains the updated information from the requirements engineering process. At the end of this process, the DJF will include the top-level design trade-offs, the requirement traceability and top-level architecture traceability matrices and the PDR milestone report. The DJF also contains the planning aspects for verification and validation.

5.4.3 Design Definition File

This file documents the results of all design activities. At the end of the requirements engineering process, the DDF will contain the software architectural design and the evidence of the customer approval of the TS.

5.4.4 Interface Control Document

This document is the developer's response to the IRD and is normally part of the TS. At the end of the requirements engineering process, the ICD would normally specify the external interfaces to the software item and the preliminary (top-level) external and internal interface designs.

CHAPTER 6
SOFTWARE DESIGN ENGINEERING

6.1 INTRODUCTION

The Software Design Engineering process can be called the 'implementation' activity of the life cycle. The purpose of this process is to detail the software design and to code, document and test the design.

The software design engineering process is the responsibility of the software engineers. Other kinds of engineers may be consulted during the process. Engineers not responsible for the process may independently verify the software.

Important considerations before starting the code production are the adequacy and availability of computer resources for software development. There is no point in starting the code and test activities if the computers, operating system software and, if applicable, communications network are not available or sufficiently reliable and stable. Productivity can drop dramatically if these resources are not adequate. Failure to invest in software tools and development hardware often leads to bigger development costs.

6.2 PROCESS INPUTS

The principal inputs to the Software Design Engineering process are as follows:

- Technical specification (TS)
- Interface control document (ICD)
- Design justification file (DJF)
- Design definition file (DDF)

6.3 ACTIVITIES

The Software Design Engineering process consists of the following activities:

- Design of software
- Coding and testing
- Integration

6.3.1 Design of Software Items

For each software product identified during the requirements engineering process, this activity consists of the following tasks:

- Design of the software component
- Development and documentation of the interface design
- Drafting of the software user manual
- Software unit test planning
- Software integration planning
- Evaluation of the detailed design and test specification

6.3.1.1 Software Component Design

In the software component design task, components of the architectural model are decomposed until they can be expressed as units in the selected programming language. A software unit is a programming entity that is discrete and identifiable with respect to compiling, combining with other units and loading.

For each of the components in the TS, the design detail is of each component specification is expanded.

The methods and CASE tools used for architectural design should be used in this task.

The format of the software component design will be heavily influenced by the method chosen. In all cases, however, the software component design task shall provide a set of component specifications that are consistent, coherent and complete. Each specification defines the functions, inputs, outputs and internal processing of the component.

The software component design documents must be added to the design definition file (DDF) that was input to the Design Engineering process. It shall be ensured that all components from the architectural design are allocated to software units.

6.3.1.2 Interface Design

The initial interface design produced during the architectural design activity shall be updated to provide the additional information available at this level.

The interfaces external to the software product and the interfaces between the components in the architecture shall be defined. The internal and external interfaces shall be documented in the ICD, although the format of this shall depend on the design method chosen. The interfaces are normally specified as data structures. The definition of each interface shall include the:

- Service provided by the interface
- Description of each element (e.g. name, type, dimension) in the interface
- Range of possible values of each element
- Initial values of each element

The detailed design of the interfaces shall permit coding without the requirement for further information.

The interface control document, part of the TS, shall be updated to include the additional levels of detail for both internal and external interfaces. It should be noted that the format of the ICD should reflect the chosen design method.

6.3.1.3 Drafting of Software User Manual

The software user manual, which is normally contained within the design definition file, shall be drafted based on the information available during the detailed design task.

The purpose of the software user manual is to describe how to use the software product. It should contain sufficient information to describe what the software product does and instructions on how to achieve this.

6.3.1.4 Software Unit Test Planning

The supplier shall define and document the unit test requirements and shall plan for testing the software units. Unit tests verify the design and implementation of all software design components.

Unit tests are organised in two ways. Black box unit testing is used to verify that a unit doing what it is supposed to do. White box testing is used to verify that the unit is operating in the way that it was intended. Part B of this Guide addresses possible ways of unit testing a software product. As a minimum, the unit test plan

requirements include the generation of test programs and associated test data sets.

The unit test plan requirements shall be documented in the software unit test plan section of the DJF.

6.3.1.5 Software Integration Planning

Integration is performed when the major components are assembled to build the system. Integration testing should be directed at verifying that major components interface correctly.

The software should be integrated incrementally. The sequence of integrating the major components to create the system shall be defined. The integration sequence should add components in order of their number of dependencies, those with least dependencies normally being added first. This approach facilitates the diagnosis of problems and minimises the amount of test software that needs to be developed to substitute for components not yet integrated. Detailed guidance on integration planning is given in Part B of this Guide.

The test planning should ensure that adequate tests are developed to ensure that all data exchanged across an interface agrees with the specifications in the ICD.

The preliminary software integration plan, part of the DJF, created during the top-level architectural design task, shall be updated to reflect the greater level of detail now available.

The software integration plan shall define the integration sequence and the integration tests at each step. Each integration test should verify the function and interfaces to be added at that step.

6.3.1.6 Evaluation of Design and Test Specifications

The supplier shall ensure that software component design documents, in the design definition file, and the test specifications

shall be evaluated, with the results of the evaluations documented. The results shall be documented in the design justification file.

The detailed design shall be evaluated in accordance with the following requirements:

- correctness, consistency and traceability to the TS
- correct implementation of the proper sequence of events, inputs, outputs, interfaces
- Feasibility of testing
- Feasibility of operation and maintenance

The test specifications (and by implication the design) shall be evaluated for the appropriateness of integration test methods and standards. The evaluation should ensure that suitable rigorous methods have been used to assess safety, security and other critical methods in the software design.

The results of the evaluation shall be documented in the design justification file, which will be available for use in the Critical Design Review. Guidelines on the approach to the evaluation and the potential tailoring to different types of project are given in Part B of this Guide.

6.3.2 Coding and Testing

For each software item, this activity consists of the following tasks:

- Develop and document software units
- Test software units
- Update the software user manual
- Update integration testing requirements
- Evaluate software code and test results

6.3.2.1 Develop and document software units

This task is concerned with the translation of the software component designs into a software programming language. This source code will, in turn, be translated into machine-dependent object code by the compiler and ultimately into machine-code that executes on the eventual system.

The implementation and testing of the system is made easier if the following guidelines are followed

- Use of consistent programming style
- Re-use of existing material
- Use of automated support environments

A consistent coding style can reduce complexity of the software unit. All code produced should be consistent with suitable coding conventions for the chosen programming language. ESA has developed a coding standards for ADA and C/C++ [Ref. 17,18], that shall be applied when these languages are being used. The general guidelines in these standards is also likely to be applicable when other languages are being used.

The solutions used on previous projects to known problems should be considered for adoption. Changes and modifications to code should follow the style of the original code.

Production of consistent, well-organised code is made easier by using appropriate development tools, which can also reduce the development time. Many programming languages may be acquired with a suite of tools that include: debugging compilers,

source code formatting aids, built-in editing facilities, tools for source code control, extensive programming libraries of functions, cross-compilers for specific development environments.

The coding task includes compilation; not only does this produce the code needed for testing the runtime behaviour of the module, it is the first step in verifying the code. Compilation normally produces statistics that can be used for the static analysis of the module.

Supplementary code included to assist testing should be readily identifiable and easy to disable, or remove, after successful testing. Care should be taken to ensure that such code does not obscure the module logic.

The source code shall be retained, in an appropriate format for the project, in the design definition file (DDF).

6.3.2.2 Unit Testing

Each software unit shall be tested in accordance with the unit test documentation. Guidelines for effective testing are contained in Part B of this Guide. The specific testing for a software unit shall be determined by the project requirements but the following aspects should normally be considered:

- The code is traceable to its design, testable, correct and compliant with the appropriate coding standards
- The code implements the proper event sequence, consistent interfaces, completeness, appropriate allocation of timing and sizing budgets, and error definition, isolation and recovery.

Where the code implements safety, security or other critical aspects, suitable rigorous methods should be used to demonstrate correct implementation.

The results of the verification activities shall be documented in the design justification file (DJF).

6.3.2.3 Software User Manual Updates

The results of the unit test task may require updates to the software user manual, a component of the technical specification (TS). This shall be updated as required.

6.3.2.4 Integration Testing Requirements

The integration test requirements and integration plan shall be updated to be consistent with the results of the detailed design task. The Software Integration Plan, part of the design justification file, shall be updated.

6.3.2.5 Evaluation of Code and Test Results

The results of the unit test activities, defined in 6.3.2.2, shall be evaluated by the supplier. This task requires an assessment of the suitability of the software items to undergo integration testing.

The guidelines for evaluating the code and test results are discussed in Part B of this Guide. The extent of the evaluation will depend on the requirements of the project but an assessment of the following aspects should be considered:

- Test coverage of units
- Feasibility of software integration and testing
- Feasibility of operation and maintenance

The results of the evaluations shall be documented in the design justification file (DJF).

6.3.3 Integration

Integration is the activity of building a software product by combining components into a working entity.

The integration of the software product into the higher level system requires additional integration. This system level integration normally takes place after the completion of the Qualification Review (QR) for the software product, as discussed in Chapter 8.

For each software item, this activity consists of the following tasks:

- Updating of the integration plan
- Integration testing
- Update to the software user manual
- Evaluation of the Integration tasks
- Critical Design Review (CDR)

6.3.3.1 Integration Planning

All integration testing shall be carried out in accordance with the software integration plan, part of the design justification file, produced during the Coding and Testing activity. If necessary, this plan shall be updated to reflect all the lessons learnt from the evaluation of the software code and test results.

6.3.3.2 Integration Testing

Integration testing is performed according to the Software Integration Plan. Integration testing is done when the major components are assembled to build the software product. These major components are identified in the TS. Integration testing should verify that the major components interface correctly.

Integration testing must check that all data exchanged across an interface agree with the specifications.

The results of the integration testing shall be documented in the Integration Test Report, held in the DJF.

6.3.3.3 Software User Manual Update

The integration testing task may require update to the software user manual. This shall be updated as required.

6.3.3.4 Evaluation of the Integration Testing

The supplier shall evaluate the results of the integration activity. The evaluation shall consider whether the software components and units have been completely and correctly implemented into the software product. Guidelines on the evaluation of the activity are defined on Part B of this Guide. The exact nature of the evaluation will depend on the project requirements but consideration should be given to the following aspects:

- Traceability to the software requirements in the TS
- External consistency with the software requirements
- Internal consistency
- Test coverage of the requirements of the software item
- Appropriateness of test standards and methods used
- Conformance to expected results
- Feasibility of software validation testing
- Feasibility of operation and maintenance

The results of the evaluation of the integration testing activity shall be documented in the design justification file.

6.3.4 Validation Testing

There are four stages to Validation Testing. The first stage is performed as part of Software Design Engineering, the other three stages as part of Software Validation and Acceptance (see section 7.3). The stages are shown in Table 6.1 below.

Validation Performed Against	**Name of Testing Stage**	**Location & Platform**	**Testing Reviewed**	**Sometimes known as**
Technical Specification	Validation Testing against Technical Specification	Supplier Premises	Critical Design Review CDR	System Tests (can also be carried out as Factory Acceptance Tests: FAT)
Requirements Baseline Subset	Validation Testing against Requirements Baseline Subset - 1	Supplier Premises	Qualification Review QR	Preliminary Acceptance Tests. Factory Acceptance Tests (FAT)
Requirements Baseline Larger-subset *and optionally a subset of the Technical Specification*(2)	Validation Testing against Requirements Baseline Subset - 2	Customer Premises on Development Environment	Acceptance Review AR	Preliminary Site Acceptance Tests PSAT (1)
Requirements Baseline (ideally all requirements) *and optionally a subset of the Technical Specification*	Validation Testing against Requirements Baseline	Customer Premises on Operational Environment	Acceptance Review AR	(Final) Site Acceptance Tests SAT (1)

(1) PSAT and SAT are known together as Operational Acceptance Tests

(2) Preliminary site acceptance test can also be carried out against the technical specification

Table 6.1 Stages of Validation Testing

6.3.5 Critical Design Review

A Critical Design Review (CDR) shall be held at the end of the design engineering process. The customer shall be invited to attend the CDR. The aim of the CDR is to ensure the completeness

of the DDF, software user manual and DJF. The completeness of the verification and validation plans and the availability of the necessary supporting resources (e.g. test case specification, simulators) are reviewed.

The CDR marks the transition of the software system from the 'Specified State' to the 'Defined State', i.e. the CDR signals the end of the design. For large software projects, all software subsystems shall undergo a CDR before they are integrated into the next highest level in the hierarchy.

The results of the CDR shall be documented in the design justification file.

It should be noted that the lifecycle chosen for the software development shall impact on the CDR. If an incremental or evolutionary lifecycle is chosen, there will be a number of CDRs, reflecting the lifecycle model.

6.4 PROCESS OUTPUTS

The main outputs from the process are the software component design, the code and the software user manual. The documented results of the verification and validation activities are also outputs of the process.

6.4.1 Design Definition File (DDF)

The following sections shall be added to the DDF.

6.4.1.1 Software Components Design Documents

This shall be formatted as shown in Appendix C. The source code will normally be included in this section.

6.4.1.2 Software User Manual

The software user manual shall be generated.

6.4.2 Technical Specification (TS)

The following section shall be updated in the TS.

6.4.2.1 Interface Control Document (ICD)

This shall be updated to include the additional level of detail available for both the internal and the external interfaces.

6.4.3 Design Justification File (DJF)

The following sections shall be added to the DJF.

6.4.3.1 Software Unit Test Plan

This document contains the test requirements and plans for testing the software units.

6.4.3.2 Software Integration Plan

The preliminary plan created during the Software Requirement Engineering process shall be updated to include the aspects for integration planning and testing generated during the Design Engineering process.

6.4.3.3 Software Validation against Technical Specification

The supplier shall develop, for each software item, a set of tests, test cases (inputs, outputs, test criteria) and test procedures for conducting software validation testing. This testing is against the technical specification.

CHAPTER 7
SOFTWARE VALIDATION AND ACCEPTANCE

7.1 INTRODUCTION

This process validates the Software against the requirements baseline. The process comprises two parts:

The first part leads to preliminary acceptance and comprises two activities:

- Preliminary acceptance tests
- Qualification Review

These processes are usually conducted under the control of the Supplier and are normally carried out at his premises.

The second part leads to final acceptance and comprises three activities:

- Delivery and Installation
- Operational Acceptance Tests
- Acceptance Review

These activities are normally controlled by the Customer, and may require the involvement of Operational departments. Final acceptance triggers the Software Operations Engineering and Software Maintenance processes.

7.2 PROCESS INPUTS

7.2.1 Requirements Baseline

This is used to prepare the acceptance test specifications. Each of the functional requirements shall be tested. Requirements that cannot be tested shall be validated by other means – e.g. analysis.

7.2.2 Technical Specification

7.2.2.1 Interface Control Document

This is used to prepare the acceptance test specifications. The document may be used to prepare a test harness if necessary.

7.2.2.2 Software Requirements Specification

The software requirements specification may be used in assembling the acceptance test specifications, if testing in the customer environment is performed against a subset of the technical specification (see table 6.1).

7.2.3 Design Definition File

7.2.3.1 Software User Manual

This is used to prepare the acceptance test specifications. At this level most tests will make use of the user interface, which will be described in the software user manual.

7.3 ACTIVITIES

7.3.1 Validation Testing against RB subset-1

These tests are also sometimes called as Factory Acceptance Tests (FAT). They are performed at the supplier's premises. They comprise a set of tests that check the software product against a subset (as constrained by use of supplier's environment) of its requirements baseline.

7.3.2 Qualification Review

This review checks that the product meets its requirements, and is ready to be installed and undergo acceptance testing at customer premises.

7.3.3 Delivery and Installation

The product is delivered to the customer's site and installed in the operational environment using the installation procedures. Installation should be undertaken from scratch to test the adequacy of the Installation procedure.

The supplier shall train Customer personnel in the use of the software.

7.3.4 Validation Testing against RB subset-2

These tests are also called Preliminary Site Acceptance Tests (PSAT). They are performed by the customer at the customer's premises on the development environment. They comprise a set of tests that check the software product against a subset (as constrained by use of cusomter's development environment) of its requirements baseline. They may include a subset of the tests performed against the technical specification. It is also possible to carry out these tests only against the technical specification.

The tests shall include the re-generation of executable from source code, to ensure that the build process is robust. The Supplier shall support the Acceptance testing process. All problems found shall be reported on a Non Conformance Report.

7.3.5 Validation Testing against RB

These tests are also sometime called (Final) Site Acceptance Tests (SAT). They are performed by the customer at the customer's premises on the operational environment. They comprise a set of tests that check the software product against all of its requirements baseline. They may include a subset of the tests performed against the technical specification.

Particular tests that may be required as part of operational validation are:

- tests with the full ground segment including communications network
- tests under different load conditions, including the maximum expected load (e.g. maximum number of user work stations, high speed telemetry playback, etc.)
- reaction and loading of software in response to on-board conditions or anomalies
- ground segment failure cases, e.g. switch of the software on to redundant equipment, reaction to failures of other equipment.

The following tests are typically carried out during ground segment operations validation:

- Tests with RF suitcase representations of the spacecraft (RF compatibility testing)
- Tests with a space segment simulator
- Space to ground segment compatibility tests (called System Validation Tests in some organizations), involving access to the spacecraft on ground during AIV activities. These ensure compatibility of space segment and ground segment and may reveal anomalies on both sides.

The tests shall include the re-generation of executable from source code, to ensure that the build process is robust. The Supplier shall support the Acceptance Testing process. All problems found shall be reported on a Non Conformance Report.

7.3.6 Software User Manual Updates

The results of the validation testing activities may require updates to the software user manual. This shall be updated as required.

7.3.7 Acceptance Review

The results of the acceptance tests shall be reviewed by the Customer. The Supplier shall support the Acceptance Review

process. At the end of the review, the Customer shall indicate the result of the test in the Acceptance Review Report.

7.4 PROCESS OUTPUTS

7.4.1 Design Definition File

7.4.1.1 Software Installation Plan

This document identifies how the product is to be installed on the operational environment. It should include instructions on installing underlying software products and data files, and on setting up the software environment.

7.4.1.2 Source Code Files, Build Code Files, Executable Code Files

These files provide the mechanism for re-generating the executable files.

7.4.2 Design Justification File

7.4.2.1 Preliminary Acceptance Test Specification

This document provides the specification for Factory Acceptance Tests.

7.4.2.2 Preliminary Acceptance Test Results

This document may be a copy of the test specification, hand marked with test results. The results of each test or group of tests that is witnessed by or on behalf of Product Assurance or Customer Personnel shall be signed to provide evidence of witnessing.

7.4.2.3 Qualification Review Report

This shall provide a record of the Qualification Review process

7.4.2.4 Operational Acceptance Test Specification

This document provides the specification for Preliminary and Final Site Acceptance Tests. (See the validation test specification template in Part C.)

7.4.2.5 Operational Acceptance Test Results

This document may be a copy of the test specification, hand marked with test results. The results of each test or group of tests that is witnessed by or on behalf of Customer Product Assurance shall be signed to provide evidence of witnessing.

7.4.2.6 Observation Reports

Observation Reports cover unforeseen departures from planned tests and/or test outcomes that whilst not being a failure are different from the predicted outcome.

7.4.2.7 Compliance Matrix

The compliance matrix shall be updated to include the compliance of Acceptance Tests to requirements baseline.

7.4.2.8 Acceptance Review Report

This shall provide a record of the Acceptance Review process, and shall include a conclusion on the final result of the test.

CHAPTER 8
SOFTWARE OPERATIONS ENGINEERING

8.1 INTRODUCTION

The Software Operations Engineering Process normally starts after the AR of the software product, although plans for it are normally finalised before AR. The nature of the software operations engineering process is determined by the system level need to operate the software product at a given time. Ground segment software products are in extensive operational use to qualify the ground segment, well before the actual mission operation occurs.

The software product is an integrated component of the overall system. The phasing and management of the operations process is determined by the system level requirement to operate the software product at a given time. The Software Operations Engineering process is, in fact, part of the operations activities for the overall system. In principle, operations engineering starts a the time of the Operational Readiness Review (ORR). In practice, it will start earlier, because ground segment software products are in extensive use to qualify the ground segment and interfaces to the space segment well before operations occur. On the other hand some software may not be operational until long after launch, for example in a deep-space mission. This process is concerned only with providing a software input into the system level activities of space system operations engineering. All activities specified here are, therefore, not specific to software but rather form an integral part of the operation of the space segment.

The Software Operations Engineering process relates to the operator. The customer is responsible for establishing the system requirements for the operation of software products. The customer is responsible for the selection of the operator, i.e. the supplier who performs the operations process.

The Software Operations Engineering process is closely linked to the Software Maintenance process, described in Chapter 9. The operator is responsible for identifying problems, or potential problems, with a software product. Resolution of these problems is described in software maintenance and is the responsibility of the maintainer.

The generic organisational model used in the process descriptions assumes that operators operate the system on behalf of users of the system. In practice, operators and users may be the same, although two different roles are distinguished here.

There is in fact a hierarchy of relationships: software support provides a service to computer/network operations, which provides a service to the operator of the applications, who in turn provides a service to the end users of those applications (see diagram opposite). The precise mapping of the generic roles onto the organisation is dependent on the mission requirements and the operations organisation.

8.2 PROCESS INPUTS

The input into this process is the approved software product from the Acceptance Review.

8.3 PROCESS ACTIVITIES

The Software Operations Engineering process, to provide software specific input into the system level process, is undertaken by participation in the following activities:

- Operational planning
- Operational testing (of new releases)
- System operation
- User support

8.3.1 Operational Planning

This activity is concerned with the development, by the operator, of the software operations plan. The software operations plan contains the operator's standards for operational testing, software operation and user support.

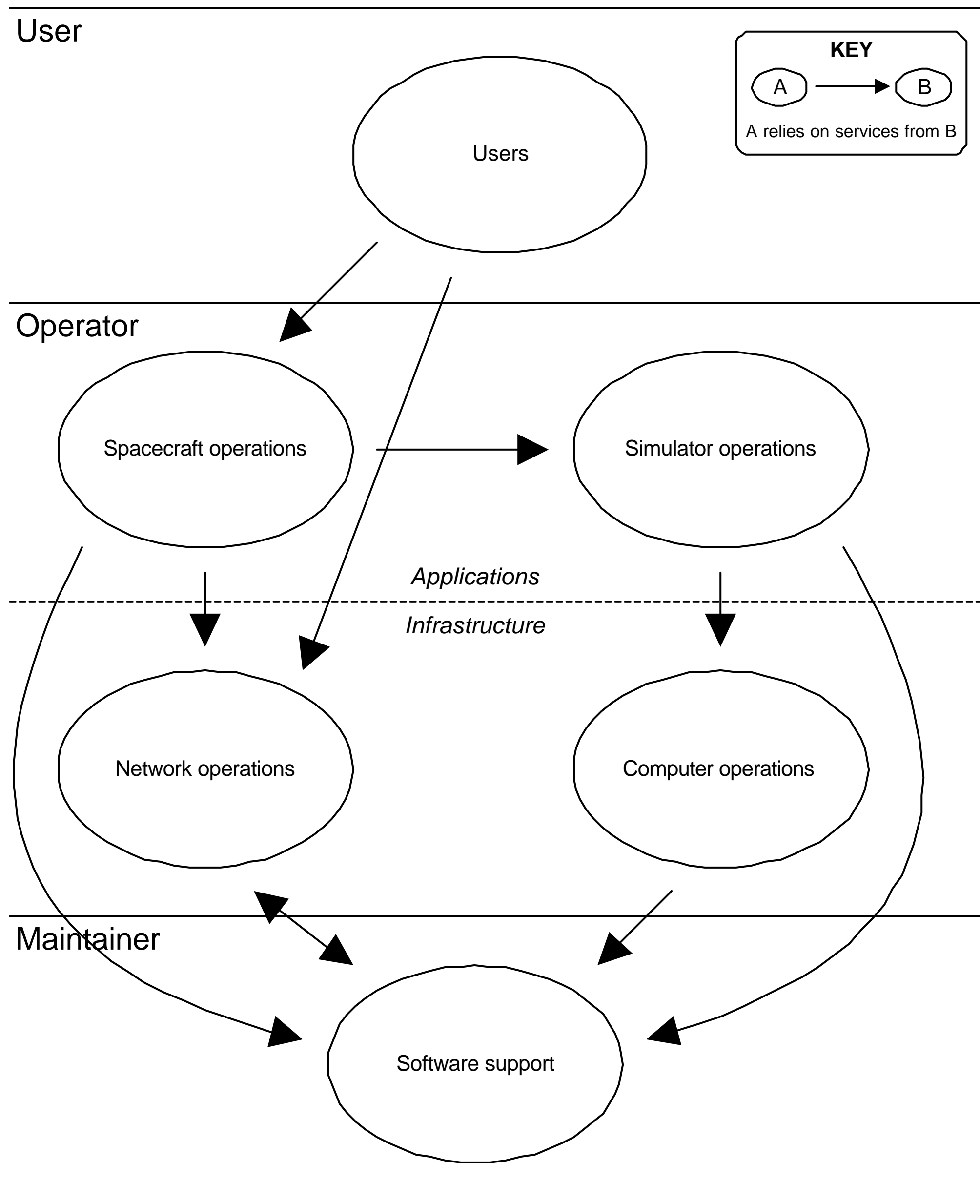

Figure 8.1: User-operator-maintainer relationships

The procedures developed for this plan will be used by the operator of the software product. The plan shall contain the following information:

8.3.1.1 Procedures for Anomaly Handling

The operator shall establish procedures for receiving, recording, resolving and tracking anomalies and for providing feedback. An anomaly may be raised when a departure from expected behaviour occurs. It shall be analysed to determine whether

- the actual behaviour is valid (and thus the expected behaviour is incorrect) or
- alternatively the expected behaviour is correct and the actual behaviour constitutes a problem.

Anomaly handling is addressed in the ECSS-M-40 standard [Ref. 7]. Guidance on the application of these mechanisms is described in Part B – Section 4.4.9 – Software Problem Reporting.

It should be noted that these procedures are similar to those used by the maintainer of a software product. The principal difference is that the operational procedures relate to the identification and tracking of operational anomalies. The maintenance process (Chapter 9) deals with the actual investigation and change of the software product.

8.3.1.2 Operational Testing Specifications

The operator shall where necessary update and amend the existing procedures for testing new releases of the software product in its operational environment. Depending upon the type of maintenance that occurs the testing procedures (usually part of the Acceptance Tests) are used as follows.

- Corrective maintenance – use procedures unchanged
- Perfective maintenance – amend procedures
- Adaptive maintenance – amend procedures

These procedures shall include details of the approach to, and test cases used, in regression testing new releases.

These procedures shall also address the mechanism for releasing the revised software product for operational use, in accordance with the change control mechanism.

The requirements for change control are identified in ECSS-M-40 [Ref. 7]. Guidelines of their application to software operational testing is described in Part B of this Guide.

8.3.2 Operational Testing

For each release of the software product, the operator shall perform operational testing. The software product shall be released for operational use when the operational testing criteria have been satisfied, in accordance with the operations test plan in the software operations plan.

The mechanism for release and evaluation of each software release is described in Part B of this document. In normal practice, the test documentation developed during the acceptance process can be used during this activity.

8.3.3 System Operation

The system is operated in its intended environment according to procedures written by those responsible for the various operations of the system (e.g. computer operations, network operations, simulator operations and spacecraft operations). Feedback information for software maintenance, e.g. appropriate logs or reports, should be identified in the operations procedures.

8.3.4 User Support

The operator shall provide assistance and consultation to the users as requested. These requests and subsequent actions shall be recorded and monitored.

The operator shall record and acknowledge all user requests, forward all appropriate user requests to the Maintenance Process (Chapter 9) for resolution, monitor their resolution to a conclusion and report back to the originator of the request.

Temporary work-arounds may be used subject to the agreement of the originator of the problem report and in accordance with defined plans and procedures. All permanent corrections, releases containing new functionality and system improvements shall be undertaken in accordance with the Maintenance Process (Chapter 9).

In practice user support often takes the form of a help desk, with availability according to the mission needs (e.g. during several working hours with call-out at other times). Typically, members of the same team that carry out software maintenance provide this user support. In this case it maps to what is often called "first-line maintenance".

During critical operations, for example during a Launch and Early Orbit Phase (LEOP), a software support team reporting to a software coordinator will be on shift. They will operate the software in the sense of monitoring its performance, restarting it or reconfiguring it on request and giving advice to the end user.

8.4 PROCESS OUTPUTS

The following are output from the Software Operations Engineering Process:

8.4.1 Software Operations Plan

The software operations plan contains the operational standards for performing the operational process, the procedures for problem handling and operational testing specifications.

CHAPTER 9
SOFTWARE MAINTENANCE

9.1 INTRODUCTION

This process is activated when the software product requires modification to its code and its associated documentation. The modification may be required either due to a problem or by the need for improvement or adaptation. The objective of maintenance is to modify an existing software product while preserving its integrity.

The Software Maintenance process is started after the completion of the Acceptance Review (AR).

As in the case of Software Operations Engineering, the Software Maintenance process is not software-specific but is in fact associated with the system level processes. The outputs of the process are contributions to the system level outputs and the actual format will depend on the system level requirements.

The Software Maintenance process provides the framework for the management of change to a software product. Software maintenance always results in a change to a software product. The actual change to the product may require the initiation of any of the life cycle processes defined in this guide i.e. system engineering, requirements engineering, design engineering and verification and validation. The processes invoked, and the extent to which they are applied, will depend on the nature of the change.

A maintenance organisation, the maintainer, shall be designated for every software product in operational use. In the case where the software development processes are invoked by the maintenance process, the term supplier is taken to mean the maintainer.

The Software Maintenance process may include the migration of the software product to different operational environments. The Software Maintenance process ends with the retirement of the software product.

The Software Maintenance process is closely linked to the Software Operations Engineering process, described in Chapter 9. Problems identified with the operation of a software product are passed, by the operator, to the maintainer for investigation and change. The operator and maintainer must work in collaboration to ensure the effective change to the operational software product.

9.2 PROCESS INPUTS

The following are input to the software maintenance process:

- All software problem reports
- All change requests
- All documentation required to undertake an investigation of a change request
- All documentation requiring change

These problems are normally passed to the maintainer as a result of operator support to the user. The format of the reports will be subject to the agreement of the operator and the maintainer.

9.3 PROCESS ACTIVITIES

The Maintenance Process consists of the following activities:

- Problem and Modification Analysis
- Modification Implementation
- Maintenance Review/Acceptance
- Software Migration
- Software Retirement

The maintainer shall develop documented procedures for undertaking the above activities. These documented procedures are normally referred to as a Maintenance Plan

The maintainer shall include in the Maintenance Plan those document procedures for receiving, recording and tracking problems and modification requests from the operator. A key element of this problem control mechanism is the need to provide feedback to the operator of the product.

More detailed guidance on the scope of these procedures is given in Part B of this document. This addresses aspects such as the format of problem reporting and change control forms and the approval mechanisms for software changes.

9.3.1 Problem and Modification Analysis

All reported software problems or modification requests shall be examined prior to any amendment. Only approved changes shall be implemented.

The key aspect of problem analysis activity is to find the root cause of the problem and the impact of the problem on the operation of the system.

The initial aspect of the activity shall be to determine explicitly the type of problem reported. The following types of request shown in Table 9.1 shall be used.

Type	Meaning

Corrective Maintenance	Changes to correct an error in the operation or a function of the system
Improvement ("perfective maintenance")	Provision of new or improved functionality
Preventive Maintenance	Changes to prevent fault occurrence in the product
Adaptive Maintenance	Migration of product to a new environment

Table 9.1 Maintenance Types

All of the above maintenance types will initially be activated by the operator/maintainer interface. An initial analysis shall be undertaken of the reported problem to determine the scope of the change required. The scope shall consider aspects such as the size of the modification, the cost involved and the time to modify.

All requests shall be prioritised in terms of the following aspects:

- Operations needs
- Risk, e.g. in the case of complex change
- Effort required to implement request

The change should also be assessed against its likely impact on either the software behaviour or the maintenance budget. The importance of the request can be different from the nature of the change. For example, a problem with an important functional aspect of the product may be rectified by a relatively simple change to the code. In the case of reusable or infrastructure software (see chapter 10), the impact on all the projects using the

product must be considered. This may involve obtaining approval of all those involved with the reusable or infrastructure software.

The maintainer shall attempt to reproduce the problem and understand why it happened. This can be used to assist in examining options for changing the product. There may be a variety of options for resolving a problem, each with associated advantages and disadvantages.

The results of this analysis shall be fully recorded in the Maintenance File.

Prior to implementing any modification to the system, the maintainer shall obtain approval from the operator for the selected option. The format of this approval process is discussed in Part B of this Guide.

9.3.2 Modification Implementation

The maintainer shall identify all software items requiring change, the extent of the change and the likely cost. The identification shall include documentation, software units and specific versions. The identification shall be recorded in the Maintenance File (MF).

All modifications are undertaken in accordance with the software engineering processes specified in this document. The software engineering process shall be entered at a point consistent with the scope of the approved change.

The following additional requirements shall be addressed.

9.3.2.1 Test Criteria

Test and evaluation criteria for testing and evaluating the modified and unmodified parts of the product shall be defined and documented.

9.3.2.2 Implementation

Regression testing shall be undertaken to ensure the complete and correct implementation of the system. The regression testing shall ensure that the unmodified requirements were not affected. The results of all regression testing shall be documented.

9.3.3 Maintenance Review/Acceptance

The supplier shall conduct the appropriate joint reviews with the operator to ensure the integrity of the modified system. Agreement of the modification shall constitute a new baseline for the product.

Acceptance testing of the change shall be performed as per section 8.3.2 – Operational Testing.

The baseline for the changes shall be recorded in the maintenance file.

The maintenance review/acceptance mechanism is more fully described in Part B of this document.

9.3.4 Software Migration

Software migration is the activity of transferring a software product from one operational environment to another. This migration may be required due to a change to the underlying hardware platform or to the operating system being used. Other software products or elements within a subsystem or system may also have been changed, thus requiring the software product to be migrated to a new operational environment.

Software migration may require changes to the software product. All such changes shall be undertaken in accordance with the development processes specified in this guide.

All migration activities shall be documented in a migration plan. The plan shall cover the following requirements:

- Requirements analysis and definition of migration
- Development of migration tools
- Conversion of the software product
- Migration execution
- Migration verification
- Support for the old environment
- User and operator involvement in the activities, plus explanation (e.g. for parallel operation extra manpower may be needed).

Users shall be given notification of the migration activities. The migration justification included in the migration plan shall contain the following information:

- Statement of why the old environment can no longer be supported
- Description of the new environment with its date of availability
- Description of other support options available, if any, once support for the old environment has been removed

Parallel operation of the old and new systems, for a specified time, may be required. Any required training for this activity, e.g. for operations, will be defined in the Migration Plan.

All those concerned with the operation of the system shall be informed when the migration takes place. All records relating to the old system shall be archived, subject to the requirements for parallel running.

A review shall be performed to assess the impact of changing to the new environment. The result of the review shall be sent to the appropriate authorities for information, guidance and action. Further guidance on this review is given in Part B of this document.

9.3.5 Software Retirement

Software retirement normally refers to the complete disposal upon customer's request of a software product, but there are situations when a temporary retirement of a software product takes place.

This temporary retirement, known as software mothballing or hibernation, is used for operational reasons, such as when a software product is completed but the system for which it is a part is not completed until a later date. Another instance of mothballing is when the software product is complete but the operations for which it is intended is not available.

When software mothballing is required, the following aspects shall be documented:

- Identification and archiving of all component parts
- Staff training and hand-over requirements, both now and later
- Re-activation mechanism, especially in relation to availability and interface testing of the software
- Testing specifications

An important aspect of this is that testing at the time of mothballing may be required. This testing may require independent staff.

The timescale for software mothballing may have implications on the underlying infrastructure. Changes have occurred to the hardware or the operating system. Any changes to the product as a result of this should be handled in accordance with the Software Migration procedures.

The permanent retirement of a software product shall only start after a decision by the customer on the basis of a Retirement Plan and with the collaboration of all those concerned with the product.

The retirement shall result in the co-ordinated and controlled performance of the operations necessary for the total or partial cessation of use of the software product.

The retirement of the software product shall be executed in accordance with all legal and administrative procedures applicable to the organisation and the product.

9.4 PROCESS OUTPUTS

The following are output from the process:

9.4.1 Maintenance File (MF)

The following sections will be updated in the MF.

9.4.1.1 Problem Analysis Report

All documentation relating to all software change requests shall be recorded. The problem analysis report shall:

- Identify the software (name, version)
- State the criticality of the problem (major/minor)
- Identify the problem (by reference to the problem report)
- Describe the cause of the problem
- Propose actions to rectify the problem
- Describe the resources required to implement the actions

9.4.1.2 Software Release Note

All information relating to each baseline shall be recorded. The following shall be defined when a new baseline is released:

- The identity of the software (name, version)
- Changes implemented in the release (reference to the problem report)
- Configuration items included in the release
- Installation instructions

9.4.2 Maintenance Plan

The maintenance plan describes the procedures for conducting the activities and tasks of the Maintenance Process.

The procedures for receiving, recording and tracking problem reports and modification requests are also defined.

9.4.3 Migration Plan

The migration plan defines the details for moving a software product from one environment to another.

The migration plan is a system level document, not specific to software.

9.4.4 Migration Justification

The migration justification provides users of a software product with the details of why the old environment is no longer to be supported and the details of the new environment. The support requirements during the transition are also recorded.

This page is intentionally left blank

CHAPTER 10
SOFTWARE RE-USE

10.1 INTRODUCTION

Software re-use holds the promise of major cost and time saving on software developments. The other major benefit of re-using software is that the quality of software products should increase by applying previously tested software components.

Software re-use shall be encouraged whenever possible. It should be considered, however, that the application of re-use requires management support. This is particularly true when developing re-usable software, which will probably require a greater cost.

The Software Re-use process establishes the basis of control when:

- It is intended to develop software products for intended re-use on other projects
- It is intended to re-use software products from other projects
- Third party COTS products are to be incorporated into a product
- Public domain or open source software is incorporated into a product

Key elements in the re-use approach are the costs of developing reusable software and the facilities for identifying reusable components. There are likely to be additional costs involved in the verification and validation of potentially re-usable components, while the adoption of re-usable components should have a beneficial effect on project quality, cost and time scale.

10.2 PROCESS INPUTS

There is no specific input into the process. The Software Re-use process is not undertaken in isolation but is an integral part of the software development lifecycle. In particular, the Re-use activities identified here should be applied during the Systems Engineering and Software Requirements Engineering processes.

Re-use may be required either by the customer or the developer. The specification may be for business reasons, such as to develop libraries for future work, or may be required to reduce time, cost or risk on a particular project. It may also be provided to impose a common 'look and feel', or a set of functionality across different projects in order to reduce training costs and promote operator mobility.

10.3 ACTIVITIES

10.3.1 Developing Software for Intended Re-use

This activity is primarily concerned with the identification of software requirements, from the customer domain, that may be used in future applications. The specification of software requirements for intended re-use in this way is normally at the request of the customer. The supplier must ensure that these requests are addressed at each of the key development processes. In addition, the supplier may require the identification of software requirements for which re-use might be possible in future applications.

10.3.1.1 Customer Requirements

The customer shall specify any special re-use constraints, that apply to the development, to enable future re-use of the software. These shall be documented in the requirements baseline.

The aim of this is to identify aspects of the customer's generic application domain that are suitable for potential re-use. This may

include requirements on the software architecture for specific target computers or operating systems.

The supplier should seek ways of identifying generic aspects of an application domain whenever possible, even if not specified by the customer. The use of techniques addressing these aspects is recommended.

10.3.1.2 Supplier Requirements

The supplier shall define procedures, methods and tools to support the development of re-usable software. These procedures, methods and tools must be applied during the software development processes to ensure that all re-use requirements are adequately addressed.

The implementation of the re-use requirements shall be formally evaluated at the PDR and CDR milestones.

The requirements for configuration management and re-use items shall be documented, along with any specific design documentation.

10.3.2 Re-using Software from Other Projects

The supplier shall consider the re-use of previously developed software, including commercial off-the-shelf software, if required by the customer. In the case where the customer has placed no specific requirements for re-use, the supplier should also consider the possible benefits to be obtained through the re-use of previously developed software.

The supplier shall document the specification of intended re-use requirements in the technical specification (TS). The results of the evaluation shall be documented in the design justification file.

There are a number of conditions for considering re-using software from other projects that are described in ECSS-Q0-80 [Ref. 10]. For details of this process please refer to Part B of this Guide.

10.3.3 Use of Third Party COTS Products

Commercial off-the-shelf software (COTS) shall be specified by the customer in the requirements baseline. The customer shall document the acquisition requirements for COTS in the RB.

The supplier shall implement the software acquisition process and document the process in the Software Development Plan.

Where an appropriate COTS product can be shown to meet project requirements, the supplier should consider its use. This is in addition to any requirements placed by the customer. The requirements of the software product may need some revision to suit a chosen COTS product. These changes need to be discussed and agreed with the customer. The supplier shall record the evaluations of all investigated COTS products in the design justification file (DJF), along with the reasons for selection of a particular COTS.

The software procurement process for COTS has to pay special attention to:

- Economic soundness of the supplier
- Past experiences with this supplier
- Supplier 's capability to provide support and maintenance and also the cost of this maintenance
- Access to good quality documentation and source code (this is essential for critical software)
- Licensing and intellectual property rights. Consistency with product assurance and verification and validation requirements is particularly relevant.
- Suitability of the product for its intended use.

[Ref. 10] (section 5.6) provides some guidance regarding product assurance requirements for the purchase of software products. The previous paragraphs also apply to Off-the-Shelf

Software (OTS) and Modified Off-the-Shelf Software (MOTS). Public domain and open source software can be treated in a similar manner bearing in mind that:

- Supplier maintenance may be non-existent.
- The intellectual property issues may be difficult to handle. The GNU General Public License (GPL), for example, requires any product incorporating open source code to be put under the same license (note that this license does not apply to all GNU products).

10.4 PROCESS OUTPUTS

The outputs of this process are additions to the normal contents of the following process documents.

10.4.1 Requirements Baseline

The requirements baseline produced by the customer shall contain the following aspects related to software re-use:

- reuse requirements
- software acquisition process for COTS

The customer organisation may well have a general COTS acquisition process in place, which should be referenced. This general process will typically include aspects such as:

- Identification of specific products
- number of licenses
- type of licenses (development, run-time)

In addition, the following aspects should be considered when acquiring COTS products:

- proprietary, usage, ownership, warranty and licensing rights are satisfied
- future support for the product is planned

10.4.2 Technical Specification

The technical specification produced by the supplier shall contain the following additional item related to software re-use:

- specification to achieve the required re-use

10.4.3 Software Development Plan

The Software Development Plan produced by the supplier shall contain the following additional item related to software re-use:

- software acquisition approach, including justification of selected COTS where appropriate

10.4.4 Design Justification File

The DJF shall incorporate the following additional information for the requirements of software re-use:

- Justification of methods and tools
- Justification of any COTS selected
- Evaluation of re-use potential, when existing software is considered for re-use
- Software user manual aspects relating to possible re-use

This page is intentionally left blank

CHAPTER 11
MAN-MACHINE INTERFACES

11.1 INTRODUCTION

The man-machine interface requirements will vary according to the type of software under consideration. For interactive systems, the users may wish to provide examples of the dialogue that is required, including the hardware to be used (e.g. keyboard, mouse, colour display, etc) and assistance provided by the software (e.g. online help).

Modern MMI technology (e.g. graphical user interfaces, multi-layered choice menus) does not lend itself to conventional software engineering documentation. The use of the technique of software prototyping, as an aid to the development of mock-ups of the MMI and assessment of the suitability of the MMI, is a key element in ensuring that appropriate MMI are provided.

The specific requirements for MMI must be addressed early in the project, utilising the most appropriate techniques and, where necessary, ensuring that suitable MMI expertise is available.

11.2 PROCESS INPUTS

There is no specific input into this process. Consideration of the MMI is not a separate activity but rather a key part of the System Engineering for Software process, by the customer, and the Software Requirements Engineering process by the supplier. The key requirement is of the MMI process is to gain an understanding of the importance of the MMI aspects to the successful operation of the system.

11.3 PROCESS ACTIVITIES

There are three principal activities concerned with this process:

- Determine Prototyping requirements
- Determine MMI standards
- Supplier Consideration of MMI Aspects

11.3.1 Determine Prototyping Requirements

The customer shall evaluate the requirements of the software product to determine the specific need for MMI requirements. In particular, the customer shall consult with the supplier to determine if a software prototype, addressing the specific aspects of the user interface, is required and the scope of such a prototype.

The requirements for such a prototype shall be documented in the RB.

11.3.2 Determine MMI Standards

There are many possible styles and approaches to generating the user interface. The customer, or a particular operational environment, may constrain the approach used.

As in all areas of requirements engineering, it is important to ensure that the user interface requirements are properly addressed. An inappropriate user interface can lead to user rejection of an otherwise perfectly acceptable system.

Consistency of user interfaces is particularly important when different suppliers develop components of a system.

The customer shall ensure that all requirements for the user interface are fully addressed and specified in the requirements baselines. Tables 12.1 and 12.2 give guidelines on aspects that should be considered.

11.3.2.1 General Guidelines

The general guidelines given in Table 11.1 on the MMI should be considered:

Guideline	**Meaning**
Consistency	A consistent format for aspects such as menu selection and command input e.g. consistent use of SAVE.
Meaningful feedback	Operator provided with visual and auditory feedback to assist two-way communication
Checks over destructive tasks	For irreversible tasks, such as file delete, confirmation messages should be used
Easy reversal of actions (undo)	Reversal of operator actions should be available. Interruption of incorrect actions
Reduce user memory requirements	Need for user to remember information should be minimised.
Efficiency of operation	Aspects such as number of keystrokes, mouse travel distances and meaning of messages should be addressed. Minimise number of open windows.
Categorisation	Displays should provide logically related material
Help facilities	Context sensitive help facilities should be provided e.g. specific help on a particular screen.
Minimise commands	Simple action verbs or short verb phrases should be used to name commands

Table 11.1 General MMI Properties

11.3.2.2 Information Display Guidelines

The customer should consider the guidelines given on display options in Table 11.2.

Guideline	Meaning
Relevant	Displayed information should be relevant to the current context
Easily Assimilated	Present data in a format that the operator is able to absorb easily e.g. graphics
Consistency	Labels, abbreviations and colours should be consistent and as expected e.g. red for warnings
Maintain visual context	Graphical representations should be capable of scaling, to allow operator to maintain a relative location of the image
Format of text	Upper and lower case, indentation and grouping should be organised to assist assimilation by the operator
Use of windows	Windows should be used to compartmentalise different types of information
Appropriate displays	In many instances, graphical representations of data are preferential to tables of data

Table 11.2 Display Properties

11.3.3 Supplier Consideration of MMI Aspects

Prototypes are a common engineering practice to test customer reaction and design ideas. A software prototype implements selected aspects of proposed software so that testing, the most direct kind of verification, can be performed. The prototyping approach is very useful for assessing elements of the user interface.

Building a prototype for the MMI aspects, which is also known as a mock-up, allows the assessment of the suitability and appropriateness of the MMI solution.

An MMI prototype is normally built quickly and easily. Such prototype development may relax the quality, reliability, maintainability or safety requirements that are used in producing the final software product. MMI prototype software is therefore normally 'pre-operational' and is not normally part of the delivered system. There may be cases, however, when the MMI prototype will be upgraded to become part of the software product. In these cases, the prototype shall be subjected quality, reliability, maintainability or safety procedures as the rest of the software product.

Where specified in the RB, the supplier shall develop software prototypes in support of The Requirements Engineering process. The aim of the prototype is to ensure that MMI specifications are consolidated and evaluated with respect to human factors and use.

The prototype should ensure:

- Proper consideration of human factors
- The MMI aspects reach an acceptable level of definition
- The technical performance of the MMI is verified.

The supplier shall also consider the guidelines in Tables 11.1 and 11.2 when implementing any aspects of the MMI. The MMI for

the software product shall be defined in the technical specification. The results of the evaluation of the prototype shall be documented in the design justification file.

11.4 PROCESS OUTPUTS

The principal output of the MMI process is the addition of man-machine interface requirements to the outputs of the System Engineering and Requirement Engineering processes.

11.4.1 Requirements Baseline

The customer shall include the specification of man-machine interface requirements in the requirements baseline. In particular, any requirements relating to the use of software prototypes shall be specified.

11.4.2 Technical Specification

The specification of MMI requirements shall be added to the TS.

11.4.3 Design Justification File

The DJF shall contain the results of the evaluation of the software prototype.

APPENDIX A GLOSSARY

DEFINITIONS

Operational Software

Operational software is any software used in operating the space system that can be traced back to requirements in the requirements baseline.

Non-operational Software

Non-operational software is any other software, which could include test scripts, test programs and simulators. Simulators would become operational in the event that they are included in operational processes, for example to check command sequences or operational procedures, as an essential step before carrying out the operations.

ABBREVIATED TERMS

The following additional acronyms are used:

AR	Acceptance Review
BSSC	Board for Software Standardisation and Control
CASE	Computer Aided Software Engineering
CDR	Critical Design Review
COTS	Commercial Off-the-Shelf Software
CPU	Central Processor Unit
DDF	Design Definition File
DJF	Design Justification File
ECSS	European Cooperation for Space Standardisation
EGSE	Electrical Ground Support Equipment
ESA	European Space Agency
ESOC	European Space Operations Centre

FAT	Factory Acceptance Tests
GCS	Ground Communication Sub-net
GSCDR	Ground Segment Critical Design Review
GSPDR	Ground Segment Preliminary Design Review
GSRR	Ground Segment Requirements Review
GSTS	Ground Station System
GSTVVRR	Ground Segment Technical Verification and Validation Review
HCI	Human Computer Interface
ICD	Interface Control Document
IRD	Interface Requirement Document
IOQR	In-Orbit Qualification Review
ISO	International Standards Organisation
LEOP	Launch and Early Orbit Operations
MCOR	Mission Close-Out Review
MCS	Mission Control System
MES	Mission Exploitation System
MMI	Man-Machine Interface
MOTS	Modified Off-The-Shelf Software
OTS	Off-the-Shelf Software
PCS	Payload Control System
PDR	Preliminary Design Review
PSAT	Preliminary Site Acceptance Tests
PSS	Procedures, Standards and Specifications
QR	Qualification Review
RAM	Random Access Memory
RB	Requirements Baseline
RF	Radio Frequency
SAT	Site Acceptance Tests
SPR	Software Problem Report
SRB	Software Review Board
SRR	System Requirements Review
SUM	Software User Manual
SVF	Software Validation Facility
SWRR	Software Requirements Review
TS	Technical Specification

APPENDIX B REFERENCES

1. Information Technology Software Life Cycle Processes ISO/IEC 12207:1995.
2. ECSS-P-001 Glossary of Terms June 1997
3. ECSS-M-00 Policy and Principles April 1996
4. ECSS-M-10 Project Breakdown Structures April 1996
5. ECSS-M-20 Project Organisation April 1996
6. ECSS-M-30 Project Phasing and Planning April 1996
7. ECSS-M-40 Configuration Management April 1996
8. ECSS-M-50 Information/Document Management April 1996
9. ECSS-M-60 Cost and Schedule Management April 1996
10. ECSS-Q-80 Software Product Assurance Issue B
11. ECSS-E-40 Space Engineering: Software, Issue B
12. ISO 9000-3:1997 Guidelines for the application of ISO 9001:1994 to the development, supply, installation and maintenance of computer software
13. ECSS-E-70 Space Engineering : Ground Systems and Operations, Part 1: Principles and Requirements, April 2000
14. ISO 9126 Information Technology – Software product evaluation - Quality characteristics and guidelines for their use
15. ECSS-E-00 Space Engineering: Policy and Principles April 1996
16. ECSS-E-10 Space Engineering: System Engineering April 1996
17. BSSC C/C++ Coding Standard, BSSC99(1), Issue 1
18. BSSC Ada Coding Standard, BSSC98(3), Issue 1

19. ECSS-E-40-3 Space Engineering: Ground Segment Software, Issue 1.0 Draft, September 2000

20. ECSS-E40-DRD Software - Document Requirements Definitions Issue 1 Draft 1 May 2000

21. ECSS-Q-00A Policy and Principles April 1996

22. ECSS-Q-20A Quality Assurance April 1996

23. SPEC/TN3 Issue:3.0 Draft A 5 November1999

24. PSS-05-0 Software Engineering Standards, October 1992

25. PSS-05-0 Software Engineering Guides, May 1995

26. ECSS-M-00-02 A Tailoring of Space Standards, April 2000

27. ECSS-M-00-03 A Risk Management, April 2000

28. ECSS-E-40-4 Space Engineering: Software Lifecycles, Issue 1.0 Draft

APPENDIX C DOCUMENT LIFECYCLES

Documents are shown in the order in which their templates appear in Part C.

File *key at end*	**Document** *italics means template not available*	**Purpose**	**Lifecycle**													
					Sofware Requirements Engineering				**Software Design Engineering**			**Software Validation and Acceptance**				
			System Engineering for Software	**SRR**	**Software Requirements Analysis**	**SWRR**	**Software Architectural Design**	**PDR**	**Design, Coding and Testing, Integration**	**Validation against Technical Specification**	**CDR**	**Preliminary Acceptance Tests**	**QR**	**Delivery, Installation and Operational Acceptance Tests**	**AR**	**Software Operations Engineering / Software Maintenance**
RB	Interface Requirements Document	Customer defines software interface requirements	Completed (then maintained, under change control)													
RB	System Specification	Customer defines software requirements	Completed (then maintained, under change control)													
MGT	Software Development Plan	Supplier addresses management requirements	Drafted		Overall plan updated; detailed plans for architectural design activity		Overall plan updated; detailed plans for software design engineering		Overall plan updated; detailed plans for software validation and acceptance							
DJF	Software Verification Plan	Supplier defines arrangements for review activities	Drafted		Updated		Updated		Updated			Updated				

File *key at end*	**Document** *italics means template not available*	**Purpose**	**Lifecycle**													
					Sofware Requirements Engineering				**Software Design Engineering**			**Software Validation and Acceptance**				
			System Engineering for Software	**SRR**	**Software Requirements Analysis**	**SWRR**	**Software Architectural Design**	**PDR**	**Design, Coding and Testing, Integration**	**Validation against Technical Specification**	**CDR**	**Preliminary Acceptance Tests**	**QR**	**Delivery, Installation and Operational Acceptance Tests**	**AR**	**Software Operations Engineering / Software Maintenance**
DJF	Software Validation Plan	Supplier defines arrangements for validation testing activities (normally against TS)	Drafted		Maintained		Maintained		Finalised							
DJF	Acceptance Test Plan (Software Validation Plan template)	Customer defines how acceptance tests will be performed										Finalised		Maintained		
TS	Interface Control Document	Supplier defines details of software interfaces	Drafted		Finalised		Maintained		Maintained							
TS	Software Requirements Specification	Supplier defines software requirements in response to customer requirements	Drafted		Finalised											
DDF	Software User Manual	Supplier describes what software does and how to achieve it							Created	Maintained		Maintained		Maintained		
DDF	Software Architectural Design	Supplier defines top-level design					Created		Maintained							

File *key at end*	Document *italics means template not available*	Purpose	Lifecycle													
					Sofware Requirements Engineering				**Software Design Engineering**			**Software Validation and Acceptance**				
			System Engineering for Software	**SRR**	**Software Requirements Analysis**	**SWRR**	**Software Architectural Design**	**PDR**	**Design, Coding and Testing, Integration**	**Validation against Technical Specification**	**CDR**	**Preliminary Acceptance Tests**	**QR**	**Delivery, Installation and Operational Acceptance Tests**	**AR**	**Software Operations Engineering / Software Maintenance**
DDF	Software component designs	Supplier defines detailed design							Created							
DDF	Source code								Created	Maintained		Maintained				
DDF	*Software Installation Plan*	Identifies how the software is installed in the operational environment										Finalised				
DDF	Build code files													Delivered		
DDF	Executable code files													Delivered		
DJF	Software Requirements Verification Report	Provides evidence of requirements review			Created											Generated
DJF	*Design Choices and Trade-Offs*	Supplier justifies design decisions					Top-level design choices and trade-offs documented		Maintained	Maintained		Maintained		Maintained		
DJF	*Software Reuse Records*		Created		Created		Created									

File *key at end*	**Document** *italics means template not available*	**Purpose**	**Lifecycle**													
					Sofware Requirements Engineering				**Software Design Engineering**			**Software Validation and Acceptance**				
			System Engineering for Software	**SRR**	**Software Requirements Analysis**	**SWRR**	**Software Architectural Design**	**PDR**	**Design, Coding and Testing, Integration**	**Validation against Technical Specification**	**CDR**	**Preliminary Acceptance Tests**	**QR**	**Delivery, Installation and Operational Acceptance Tests**	**AR**	**Software Operations Engineering / Software Maintenance**
DJF	Requirements Traceability / Compliance matrices	Trace implementation of requirements	System requirements to sub-system partitions		TS to RB completeness		Top-level architecture traceability		Traceability of detailed design to TS			Acceptance test to RB				
DJF	Software Validation Testing Specifications (against TS)	Defines tests, test cases and procedures for validation against TS							Finalised							
DJF	Preliminary Acceptance Test Specification (Software Validation Testing Specification template)	Defines tests, test cases and procedures for FAT										Created				

File *key at end*	**Document** *italics means template not available*	**Purpose**	**Lifecycle**													
			System Engineering for Software	**SRR**	**Sofware Requirements Engineering**			**PDR**	**Software Design Engineering**		**CDR**	**Software Validation and Acceptance**			**AR**	**Software Operations Engineering / Software Maintenance**
					Software Requirements Analysis	**SWRR**	**Software Architectural Design**		**Design, Coding and Testing, Integration**	**Validation against Technical Specification**		**Preliminary Acceptance Tests**	**QR**	**Delivery, Installation and Operational Acceptance Tests**		
DJF	Operational Acceptance Test Specification (Software Validation Testing Specification template)	Defines tests, test cases and procedures for PSAT and SAT												Finalised		
DJF	Software Architecture and Interface Verification Report	Provides evidence of AD review					Created									Created (for design modifications)
DJF	Software Design Verification Report	Provides evidence of DD review							Created							
DJF	Software Code Verification Report	Provides evidence of code review							Created							
DJF	Software Documentation Verification Report	Provides evidence of document review							Created							

File *key at end*	**Document** *italics means template not available*	**Purpose**	**Lifecycle**													
					Sofware Requirements Engineering				**Software Design Engineering**			**Software Validation and Acceptance**				
			System Engineering for Software	**SRR**	**Software Requirements Analysis**	**SWRR**	**Software Architectural Design**	**PDR**	**Design, Coding and Testing, Integration**	**Validation against Technical Specification**	**CDR**	**Preliminary Acceptance Tests**	**QR**	**Delivery, Installation and Operational Acceptance Tests**	**AR**	**Software Operations Engineering / Software Maintenance**
DJF	*Software Unit Test Verification Report*	Provides evidence of unit test results review							Created							
DJF	Software Integration Verification Report	Provides evidence of Integration review							Created							
DJF	Software Integration Plan	Supplier plans the integration task, including integration testing					Preliminary version drafted		Finalised							
DJF	Software Unit Test Plan	Provides details of Unit Testing							Created							
DJF	Software Unit Test Report	Provides results of unit testing							Created							
DJF	Integration Test Report	Provides results of integration testing							Created							
DJF	Software Validation Testing Report	Records results of Validation Testing against the Technical Specification. (Same template used for FAT and (P)SAT too.)								Created						

File *key at end*	Document *italics means template not available*	Purpose	Lifecycle													
					Sofware Requirements Engineering				Software Design Engineering			Software Validation and Acceptance				
			System Engineering for Software	SRR	Software Requirements Analysis	SWRR	Software Architectural Design	PDR	Design, Coding and Testing, Integration	Validation against Technical Specification	CDR	Preliminary Acceptance Tests	QR	Delivery, Installation and Operational Acceptance Tests	AR	Software Operations Engineering / Software Maintenance
DJF	Preliminary Acceptance Test Results (Software Validation Testing Report template)	Records results of FAT										Created				
DJF	Operational Acceptance Test Results (Software Validation Testing Report template)	Records results of PSAT and SAT												Created		
DJF	Test Specification Evaluation Report	Provides evidence of review of a Test Specification								Created		Created		Created		
DJF	Software Design and Test Evaluation Report	Evaluates the detailed design and test							Created							
DJF	Software Budget Report	Reports status of technical budget and margins	Produced		Updated		Updated		Updated			Updated		Updated		

File *key at end*	**Document** *italics means template not available*	**Purpose**	**Lifecycle**														
					Sofware Requirements Engineering				**Software Design Engineering**			**Software Validation and Acceptance**				
			System Engineering for Software	**SRR**	**Software Requirements Analysis**	**SWRR**	**Software Architectural Design**	**PDR**	**Design, Coding and Testing, Integration**	**Validation against Technical Specification**	**CDR**	**Preliminary Acceptance Tests**	**QR**	**Delivery, Installation and Operational Acceptance Tests**	**AR**	**Software Operations Engineering / Software Maintenance**
DJF	Software Configuration File	Provides a definition of the configuration of the software at each milestone	Produced to identify System Engineering documents		Updated to include Requirements Analysis documents		Updated to include Architectural Design documents		Updated to include all software configuration Items	Updated to reflect current configuration status		Updated to reflect current configuration status		Updated to reflect current configuration status		Updated to reflect current configuration status
PAF	Product Assurance Report	Supplier reports on product assurance activities (on same cycle as other management reports)	produced on management report cycle		produced on management report cycle		produced on management report cycle		produced on management report cycle			produced on management report cycle		produced on management report cycle		
PAF	Software Product Assurance Plan	Supplier defines plans for measuring and controlling product and process quality	Drafted		Maintained		Maintained		Maintained			Maintained				
O/M	Software Maintenance Plan	Supplier defines maintenance organisation, processes, etc.	May be drafted now or at agreed later stage						Drafted					Finalised		Maintained
O/M	Software Operations Plan	Operator defines approach to operational testing, operation and user support	Outline may be drafted now or at agreed later stage						Drafted					Finalised		Maintained

File *key at end*	Document *italics means template not available*	Purpose	Lifecycle													
					Sofware Requirements Engineering				Software Design Engineering			Software Validation and Acceptance				
			System Engineering for Software	SRR	Software Requirements Analysis	SWRR	Software Architectural Design	PDR	Design, Coding and Testing, Integration	Validation against Technical Specification	CDR	Preliminary Acceptance Tests	QR	Delivery, Installation and Operational Acceptance Tests	AR	Software Operations Engineering / Software Maintenance
O/M	*Maintenance Records*	Customer maintains documentation related to problems and change requests														Generated
O/M	Migration Plan, including Migration Justification															Created
O/M	Software Retirement Plan															Created
MGT	Software Configuration Management Plan	Supplier defines how control of product configuration will be applied	Drafted		Maintained		Maintained		Maintained			Maintained				
MGT	Software Progress Report	Supplier reports project status (resource, schedule, financial)	produced on management report cycle		produced on management report cycle		produced on management report cycle		produced on management report cycle	produced on management report cycle		produced on management report cycle		produced on management report cycle		
DJF / O/M	Software Problem Report	Records software problems during development or operation							Generated	Generated		Generated		Generated		Generated

File *key at end*	Document *italics means template not available*	Purpose	Lifecycle													
					Sofware Requirements Engineering				Software Design Engineering			Software Validation and Acceptance				
			System Engineering for Software	SRR	Software Requirements Analysis	SWRR	Software Architectural Design	PDR	Design, Coding and Testing, Integration	Validation against Technical Specification	CDR	Preliminary Acceptance Tests	QR	Delivery, Installation and Operational Acceptance Tests	AR	Software Operations Engineering / Software Maintenance
DJF	Non-Conformance Report	Records failure to meet specified requirement			Generated		Generated		Generated	Generated		Generated		Generated		
DJF	*Observation Report*	Records unforeseen departures from planned tests and/or outcomes, but which are not failures								Created		Created		Created		
O/M	Problem Analysis Report	Provides assessment, diagnosis, and approach and estimate for rectification														Created
DJF / O/M	Software Change Request	Records a change required and its cost and schedule impacts			Generated		Generated		Generated	Generated		Generated		Generated		Generated
O/M	Software Release Note	Provides information on a new baseline, including changes applied and installation instructions														Created

File *key at end*	**Document** *italics means template not available*	**Purpose**	**Lifecycle**													
					Sofware Requirements Engineering				**Software Design Engineering**			**Software Validation and Acceptance**				
			System Engineering for Software	**SRR**	**Software Requirements Analysis**	**SWRR**	**Software Architectural Design**	**PDR**	**Design, Coding and Testing, Integration**	**Validation against Technical Specification**	**CDR**	**Preliminary Acceptance Tests**	**QR**	**Delivery, Installation and Operational Acceptance Tests**	**AR**	**Software Operations Engineering / Software Maintenance**
DJF	Request for Waiver	Requests and grants (when signed) formal approval to ignore a requirement			Generated		Generated		Generated	Generated		Generated		Generated		
MGT	*Lessons Learned*	Supplier assesses lessons learned from the project												Finallised		

Key to File identifiers:

DDF Design Definition File
DJF Design Justification File
MGT Management File
O/M Operations File / Maintenance File
PAF Product Assurance File
RB Requirements Baseline
TS Technical Specification