ESA PSS-05-04 Issue 1 Revision 1
March 1995

Guide to the software architectural design phase

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

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

DOCUMENT STATUS SHEET

DOCUMENT STATUS SHEET			
1. DOCUMENT TITLE: **ESA PSS-05-04 Guide to the software architectural design phase**			
2. ISSUE	3. REVISION	4. DATE	5. REASON FOR CHANGE
1	0	1992	First issue
1	1	1995	Minor updates for publication

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

Issue 1 approved, March 31st 1992
Telematics Supervisory Board

Issue 1 approved by:
The Inspector General, ESA

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

TABLE OF CONTENTS

This page is intentionally left blank

PREFACE

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

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

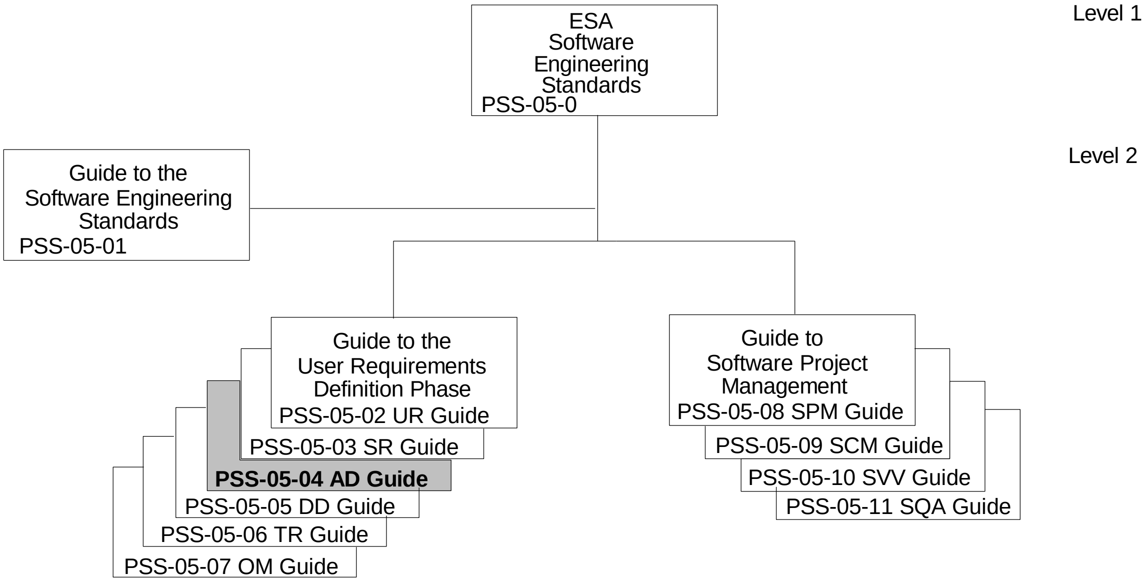

Figure 1: ESA PSS-05-0 document tree

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

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

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

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

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

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

CHAPTER 1
INTRODUCTION

1.1 PURPOSE

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

ESA PSS-05-0 defines the first phase of the software development life cycle as the 'Software Requirements Definition Phase' (SR phase). The output of this phase is the Software Requirements Document (SRD). The second phase of the life cycle is the 'Architectural Design Phase' (AD phase). Activities and products are examined in the 'AD review' (AD/R) at the end of the phase.

The AD phase can be called the 'solution phase' of the life cycle because it defines the software in terms of the major software components and interfaces. The 'Architectural Design' must cover all the requirements in the SRD.

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

1.2 OVERVIEW

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

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

This page is intentionally left blank

CHAPTER 2
THE ARCHITECTURAL DESIGN PHASE

2.1 INTRODUCTION

In the AD phase, the software requirements are transformed into definitions of software components and their interfaces, to establish the framework of the software. This is done by examining the SRD and building a 'physical model' using recognised software engineering methods. The physical model should describe the solution in concrete, implementation terms. Just as the logical model produced in the SR phase structures the problem and makes it manageable, the physical model does the same for the solution.

The physical model is used to produce a structured set of component specifications that are consistent, coherent and complete. Each specification defines the functions, inputs and outputs of the component (see Section 2.4).

The major software components are documented in the Architectural Design Document (ADD). The ADD gives the developer's solution to the problem stated in the SRD. The ADD must cover all the requirements stated in the SRD (AD20), but avoid the detailed consideration of software requirements that do not affect the structure.

The main outputs of the AD phase are the:

- Architectural Design Document (ADD);
- Software Project Management Plan for the DD phase (SPMP/DD);
- Software Configuration Management Plan for the DD phase(SCMP/DD);
- Software Verification and Validation Plan for the DD Phase (SVVP/DD);
- Software Quality Assurance Plan for the DD phase (SQAP/DD);
- Integration Test Plan (SVVP/IT).

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

While the architectural design is a responsibility of the developer, participants in the AD phase also should include user representatives, systems engineers, hardware engineers and operations personnel. In reviewing the architectural design, project management should ensure that all parties are consulted, to minimise the risk of incompleteness and error.

AD phase activities must be carried out according to the plans defined in the SR phase (AD01). Progress against plans should be continuously monitored by project management and documented at regular intervals in progress reports.

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

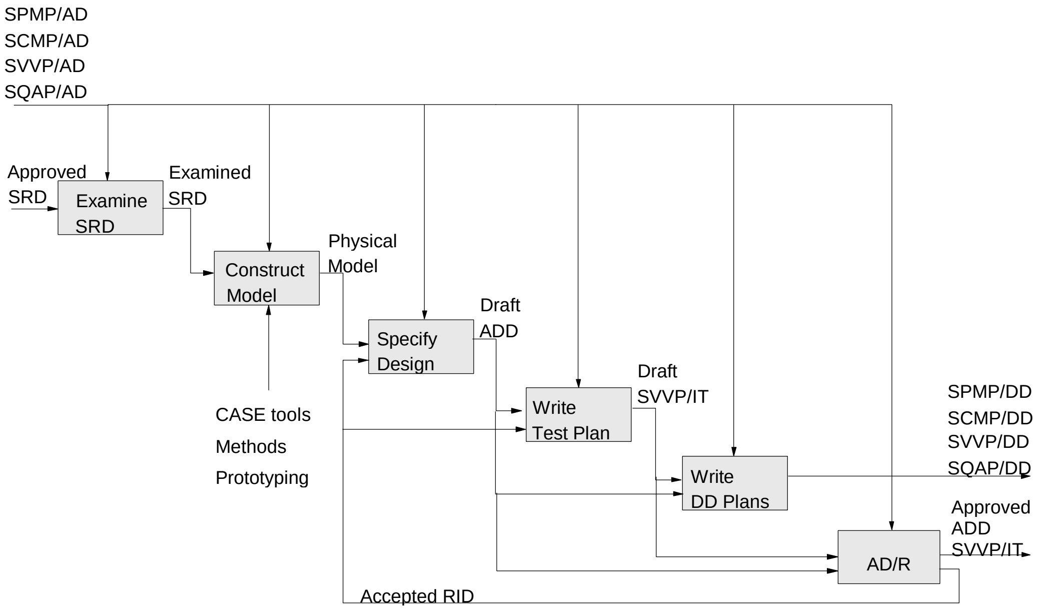

Figure 2.1: AD phase activities

2.2 EXAMINATION OF THE SRD

Normally the developers will have taken part in the SR/R, but if not, they should examine the SRD and confirm that it is understandable. ESA PSS-05-03, 'Guide to the Software Requirements Definition Phase', contains material that should assist this. In particular, if a specific method has been used for the specification of the requirements, the development organisation should ensure that the examination is carried out by staff familiar with that method.

The developers should also confirm that adequate technical skill is available to produce the outputs of the AD phase.

2.3 CONSTRUCTION OF THE PHYSICAL MODEL

A software model is:

- a simplified description of a system;
- hierarchical, with consistent decomposition criteria;
- composed of symbols organised according to some convention;
- built with the aid of recognised methods and tools;
- used for reasoning about the software.

A 'simplified description' is obtained by abstracting the essentials and ignoring non-essentials. A hierarchical presentation makes the model simpler to understand.

A recognised method for software design must be adopted and used consistently in the AD phase (AD02). The key criterion for the 'recognition' of a method is full documentation. The availability of tools and training courses are also important.

A method does not substitute for the experience and insight of the developer but helps to apply those abilities more effectively.

In the AD phase, the developer constructs a 'physical model' describing the design in implementation terminology (AD03). The physical model:

- defines the software components;
- is presented as a hierarchy of control;
- uses implementation terminology (e.g. computer, file , record);
- shows control and data flow.

The physical model provides a framework for software development that allows independent work on the low-level components in the DD phase. The framework defines interfaces that programmers need to know to build their part of the software. Programmers can work in parallel, knowing that their part of the system will fit in with the others.

The logical model is the starting point for the construction of the physical model. The design method may provide a technique for transforming the logical model into the physical model. Design constraints, such as the need to reuse components, may cause departure from the structure of the logical model. The affect on adaptability of each modification of the structure should be assessed. A minor new requirement should not cause a major design change. The designer's goal is to produce an efficient design that has the structure of the logical model and meets all the software requirements.

The method used to decompose the software into its component parts shall permit the top-down approach (AD04). Top-down design starts by defining the top level software components and then proceeds to define the lower-level components. The top-down approach is vital for controlling complexity because it enforces 'information hiding' by demanding that lower-level components behave as 'black boxes'. Even if a design method allows other approaches (e.g. bottom-up), the presentation of the design in the ADD must always be top-down.

The construction of the physical model should be an iterative process where some or all of the tasks are repeated until a clear, consistent definition has been formulated. Walkthroughs, inspections and technical reviews should be used to agree each level of the physical model before descending to the next level of detail.

In all but the smallest projects, CASE tools should be used for building a physical model. They make consistent models that are easier to construct and modify.

The type of physical model that is built depends upon the method selected. Chapter 3 summarises the methods for constructing a physical model. In the following subsections, concepts and terminology of structured methods are used to show how to define the architecture. This does not imply that these methods shall be used, but only that they may be suitable.

2.3.1 Decomposition of the software into components

The design process starts by decomposing the software into components. The decomposition should be done top-down, based on the functional decomposition in the logical model.

The correspondence between functional requirements in the SRD and components in the ADD is not necessarily one-to-one and frequently isn't. This is because of the need to work within the constraints forced by the non-functional requirements (see Section 2.3.2) and the capabilities of the software and hardware technology. Designs that meet these constraints are said to be 'feasible'. When there are two or more feasible solutions to a design problem, the designer should choose the most cost-effective one.

Correctness at each level can only be confirmed after demonstrating feasibility of the next level down. Such demonstrations may require prototyping. Designers rely on their knowledge of the technology, and experience of similar systems, to achieve a good design in just a few iterations.

The decomposition of the software into components should proceed until the architectural design is detailed enough to allow:

- individual groups or team members to commence the DD phase;
- the schedule and cost of the DD and TR phases to be estimated.

In a multitasking real-time system, the appropriate level to stop architectural design is when the processing has become purely sequential. This is the lowest level of the task hierarchy, and is the stage at which the control flow has been fully defined.

It is usually unnecessary to describe the architecture down to the module level. However some consideration of module level processing is usually necessary if the functionality at higher levels is to be allocated correctly.

2.3.2 Implementation of non-functional requirements

The SRD contains several categories of requirements that are non-functional. Such requirements can be found under the headings of:

Performance Requirements
Interface Requirements
Operational Requirements

Resource Requirements
Verification Requirements
Acceptance-Testing Requirements
Documentation Requirements
Security Requirements
Portability Requirements
Quality Requirements
Reliability Requirements
Maintainability Requirements
Safety Requirements

The specification of each component should be compared with all the requirements. This task is made easier if the requirements are grouped with their related functions in the SRD. Some kinds of non-functional requirements do not influence the architecture and their implementation can be deferred to the DD phase. Nevertheless, each requirement must be accounted for, even if this only involves including the non-functional requirement in a component's specification.

2.3.2.1 Performance requirements

Performance requirements may be stated for the whole system, or be attached to high-level functions. Whatever the approach, the designer should identify the components that are used to provide the service implied by the performance requirements and specify the performance of each component.

Performance attributes may be incorporated in the functional requirements in the SRD. Where there is a one-to-one correspondence between a component and a function, definition of the performance of the component is straightforward.

The performance of a component that has not yet been built can only be estimated from knowledge of the underlying technology upon which it is based. The designer may have to confirm the feasibility of the performance specification by prototyping. As an example, the execution time of an i/o routine can be predicted from knowledge of the required data flow volume and i/o rate to a peripheral. This prediction should be confirmed by prototyping the routine.

2.3.2.2 Interface requirements

A component design should satisfy all the interface requirements associated with a functional requirement.

Interface requirements may define internal or external control and data flows. The developer has to design the physical mechanism for the control and data flows. Mechanisms may include subroutine calls, interrupts, file and record formats and so on.

The interfaces to a component should match its level of abstraction. A component that processes files should have files flowing in and out. Processing of records, fields, bits and bytes should be delegated to lower-level components.

Interfaces should be implemented in such a way as to minimise the coupling. This can be done in several ways:

- minimise the number of separate items flowing into and out of the component;
- minimise the number of separate routes into and out of the component.

Minimisation of the number of separate items flowing into the component should not reduce the understandability of the interface. Data items should be aggregated into structures that have the appropriate level of abstraction, but not arbitrarily bunched together to reduce the numbers (e.g. by using an array).

Ideally, a single entry point for both the control and data flows should be provided. For a subroutine, this means providing all the external data the routine needs in the call argument list. Separating the control and data flow routes (e.g. by routing the data into a component via a common area) greatly increases the coupling between components.

External interface requirements may be referenced by the SRD in an Interface Control Document (ICD). The physical mechanism of the interface may be predefined, and may constrain the software design. Distorting effects should be minimised by providing dedicated components to handle external interfaces. Incidentally, this approach enhances portability and maintainability.

Where an interface standard has to be used, software components that provide the interface can be reused from existing libraries or systems. The repercussions of reusing a component should be considered before making the decision to do it.

2.3.2.3 Operational requirements

The operational requirements of the software should be considered in the AD phase. This should result in a definition of the:

- ergonomic aspects;
- screen layouts;
- user interface language;
- help system.

2.3.2.4 Resource requirements

Resource requirements should be an important consideration in the AD phase, as they may affect the choice of programming language and the high-level architecture. For example the design of the software for a single processor system can be quite different from that of a multiple processor system.

2.3.2.5 Verification requirements

Verification requirements may call for test harnesses or even a 'simulator'. The developer must consider the interfaces to external software used for verification.

A simulator may be used to exercise the system under development, and is external to it. The architectural design of a simulator should be entirely separate from the main design.

2.3.2.6 Acceptance-testing requirements

Like verification requirements, acceptance-testing requirements may call for test harnesses or even a 'simulator' (see Section 2.3.2.5).

2.3.2.7 Documentation requirements

Documentation requirements seldom affect the Architectural Design.

2.3.2.8 Security requirements

Security requirements may affect the selection of hardware, the operating system, the design of the interfaces among components, and the design of database components. Decisions about access to capabilities of the system may have to be made in the AD phase.

2.3.2.9 Portability requirements

Portability requirements can severely constrain the architectural design by forcing the isolation of machine-dependent features. Portability requirements can restrict the options for meeting performance requirements and make them more difficult to meet.

The designer must study the portability requirements carefully and isolate the features common to all the intended environments, and then design the software to use only these features. This may not always be possible, so the designer must explicitly define components to implement features not available in all the intended environments.

2.3.2.10 Quality requirements

Quality requirements can affect:

- the choice of design methods;
- the decisions about reuse, e.g: has the reused software been specified, designed and tested properly?
- the choice of tools;
- the type of design review, e.g. walkthroughs or inspections?

2.3.2.11 Reliability requirements

Reliability requirements can only be directly verified after the software has been written. The key to designing a system having a specified level of reliability is to:

- be aware of the design quality factors that are predictive of eventual software reliability (e.g. complexity);
- calibrate the relationship between predictive metrics and the indicative metrics of reliability (e.g. MTBF);
- ensure that the design scores well in terms of the predictive metrics.

For small projects, careful adherence to the design quality guidelines will be sufficient. This is the 'qualitative' approach to satisfying the reliability requirements. For larger projects, or projects with stringent reliability requirements, a 'quantitative' approach should measure adherence to design quality guidelines with reliability metrics. Metrics have been devised (e.g. 39 metrics are listed in Reference 17) to measure reliability at all phases in the life cycle.

An obvious way of meeting reliability requirements is to reuse software whose operational reliability has already been proven.

Engineering features specifically designed to enhance reliability include:

- fault tolerance;
- redundancy;
- voting.

Fault tolerance allows continued operation of a system after the occurrence of a fault, perhaps with a reduced level of capability or performance. Fault tolerance can be widely applied in software (e.g. software will frequently fail if it does not check its input).

Redundancy achieves fault tolerance with no reduction of capability because systems are replicated. There are two types of redundancy:

- passive;
- active.

Passive redundancy is characterised by one system being 'online' at a time. The remaining systems are on standby. If the online system fails, a standby is brought into operation. This is the 'backup' concept.

Active redundancy is characterised by multiple systems being 'online' simultaneously. Voting is required to control an active redundant system. An odd number of different implementations of the system each meeting the same requirements are developed and installed by different teams. Control flow is a majority decision between the different systems. The systems may have:

- identical software and hardware;
- identical hardware and different software;
- different hardware and software.

2.3.2.12 Maintainability requirements

Maintainability requirements should be considered throughout the architectural design phase. However it is only possible to verify directly the Mean Time To Repair (MTTR) in the operations phase. MTTR can be verified indirectly in the architectural design phase by relating it to design metrics, such as complexity.

The techniques used for increasing software reliability also tend to increase its maintainability. Smaller projects should follow the design quality guidelines to assure maintainability. Larger projects should adopt a quantitative approach based on metrics. Maintainability is also enhanced by:

- selecting a high-level programming language;
- minimising the volume of code that has to be maintained;
- providing software engineering tools that allow modifications to be made easily;
- assigning values to parameters at runtime;
- building in features that allow faults to be located quickly;
- provision of remote maintenance facilities.

Some software engineering tools may be for configuration management. The Software Configuration Management Plan should take account of the maintainability requirements, since the efficiency of these procedures can have a marked effect on maintainability.

Binding is 'the assignment of a value or referent to an identifier' [Ref. 2]. An example is the assignment of a value to a parameter. Static binding is 'binding performed prior to the execution of a program and not subject to change during program execution' [Ref. 2]. Contrast with dynamic binding, which is 'binding performed during the execution of a computer program' [Ref. 2]. Increasing dynamic binding and reducing static binding makes the software more adaptable and maintainable, usually at the expense of complexity and performance. The designer should examine these trade-offs in deciding when to bind values to parameters.

Features that allow errors to be located quickly and easily should be built into the software. Examples are tracebacks, error reporting systems and routines that check their input and report anomalies. The policy and procedures for error reporting should be formulated in the AD phase and applied consistently by the whole project from the start of the DD phase.

Remote maintenance facilities can enhance maintainability in two ways. Firstly, operations can be monitored and the occurrence of faults prevented. Secondly, faults can be cured more quickly because it is not always necessary to travel to the site.

2.3.2.13 Safety requirements

Designing for safety ensures that people and property are not endangered:

- during normal operations;
- following a failure.

It would be unsafe, for example, to design a control system that opens and closes a lift door and omit a mechanism to detect and respond to people or property blocking the door.

A system may be very reliable and maintainable yet may still be unsafe when a fault occurs. The designer should include features to ensure safe behaviour after a fault, for example:

- transfer of control to an identical redundant system;
- transfer of control to a reduced capability backup system;
- graceful degradation, either to a useful, but reduced level of capability, or even to complete shutdown.

Systems which behave safely when faults occur are said to be 'fail safe'.

Prevention of accidents is achieved by adding 'active safety' features, such as the capability to monitor obstructions in the lift door control system example. Prevention of damage following an accident is achieved by adding 'passive safety' features, such as an error handling mechanism.

If safety requirements have been omitted from the SRD, the designer should alert management. Analysts and designers share the responsibility for the development of safe systems.

2.3.3 Design quality

Designs should be adaptable, efficient and understandable. Adaptable designs are easy to modify and maintain. Efficient designs make minimal use of available resources. Designs must be understandable if they are to be built, operated and maintained effectively.

Attaining these goals is assisted by aiming for simplicity. Metrics should be used for measuring simplicity/complexity (e.g. number of interfaces per component and cyclomatic complexity).

Simple components are made by maximising the degree to which the activities internal to the component are related to one another (i.e. 'cohesion'). Simple structures can be made by:

- minimising the number of distinct items that are passed between components (i.e. 'coupling');
- ensuring that the function a component performs is appropriate to its place in the hierarchy (i.e. 'system shape');
- ensuring the structure of the software follows the structure of the data it deals with (i.e. 'data structure match');
- designing components so that their use by other components can be maximised (i.e. 'fan-in');
- restricting the number of child components to seven or less (i.e. 'fan-out');
- removing duplication between components by making new components (i.e. 'factoring').

Designs should be 'modular', with minimal coupling between components and maximum cohesion within each component. Modular designs are more understandable, reusable and maintainable than designs consisting of components with complicated interfaces and poorly matched groupings of functions.

Each level of a design should include only the essential aspects and omit inessential detail. This means they should have the appropriate degree of 'abstraction'. This enhances understandability, adaptability and maintainability.

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, understandability can be considerably impaired by permitting unnecessary variety. CASE tools, designs standards and design reviews all help to enforce consistency and uniformity.

As each layer is defined, the design should be refined to increase the quality, reliability, maintainability and safety of the software. When the next lower layer is designed, unforeseen consequences of earlier design decisions are often spotted. The developer must be prepared to revise all levels of a design during its development. However if the guidelines described in this section are applied throughout the design process, the need for major revisions in the design will be minimised.

2.3.3.1 Design metrics

2.3.3.1.1 Cohesion

The parts of a component should all relate to a single purpose. Cohesion measures the degree to which activities within a component are related to one another. The cohesion of each component should be maximised. High cohesion leads to robust, adaptable, maintainable software.

Cohesion should be evaluated externally (does the name signify a specific purpose?) and internally (does it contain unrelated pieces of logic?). Reference 15 identifies seven types of cohesion:

- functional cohesion;
- sequential cohesion;
- communicational cohesion;
- procedural cohesion;
- temporal cohesion;
- logical cohesion;
- coincidental cohesion.

Functional cohesion is most desirable and coincidental cohesion should definitely be avoided. The other types of cohesion are acceptable under certain conditions. The seven types are discussed below with reference to typical components that might be found in a statistics package.

Functionally cohesive components contain elements that all contribute to the execution of one and only one problem-related task. Names carry a clear idea of purpose (e.g. CALCULATE_MEAN). Functionally cohesive components should be easy to reuse. All the components at the bottom of a structure should be functionally cohesive.

Sequentially cohesive components contain chains of activities, with the output of an activity being the input to the next. Names consist of specific verbs and composite objects (e.g. CALCULATE_TOTAL_AND_MEAN). Sequentially cohesive components should be split into functionally cohesive components.

Communicationally cohesive components contain activities that share the same input data. Names have over-general verbs but specific objects (e.g. ANALYSE_CENSUS_DATA), showing that the object is used for a range of activities. Communicationally cohesive components should be split into functionally cohesive components.

Procedurally cohesive components contain chains of activities, with control passing from one activity to the next. Individual activities are only related to the overall goal rather than to one another. Names of procedurally cohesive components are often 'high level', because they convey an idea of their possible activities, rather than their actual activities (e.g. PROCESS_CENSUS_DATA). Procedurally cohesive components are acceptable at the highest levels of the structure.

Temporally cohesive components contain activities that all take place simultaneously. The names define a stage in the processing (e.g. INITIALISE, TERMINATE). Bundling unrelated activities together on a temporal basis is bad practice because it reduces modularity. An initialisation component may have to be related to other components by a global data area, for example, and this increases coupling and reduces the possibilities for reuse. Temporal cohesion should be avoided.

Logically cohesive components perform a range of similar activities. Much of the processing is shared, but special effects are controlled by means of flags or 'control couples'. The names of logically cohesive components can be quite general, but the results can be quite specialised (e.g. PROCESS_STATISTICS). Logical cohesion causes general sounding names to appear at the bottom levels of the structure, and this can make the design less understandable. Logical cohesion should be avoided, but this may require significant restructuring.

Coincidentally cohesive components contain wholly unrelated activities. Names give no idea of the purpose of the component (e.g. MISCELLANEOUS_ROUTINE). Coincidentally cohesive components are often invented at the end of the design process to contain miscellaneous activities that do not neatly fit with any others. Coincidental cohesion severely reduces understandability and maintainability, and should be avoided, even if the penalty is substantial redesign.

Practical methods for increasing cohesion are to:

- make sure that each component has a single clearly defined function;
- make sure that every part of the component contributes to that function;
- keep the internal processing short and simple.

2.3.3.1.2 Coupling

A 'couple' is an item of information passed between two components. The 'coupling' between two components can be measured by the number of couples passed. The number of couples flowing into and out of a component should be minimised.

The term 'coupling' is also frequently used to describe the relative independence of a component. 'Tightly' coupled components have a high level of interdependence and 'loosely' coupled components have a low level of interdependence. Dependencies between components should be minimised to maximise reusability and maintainability.

Reducing coupling also simplifies interfaces. The information passed between components should be the minimum necessary to do the job.

High-level components can have high coupling when low-level information items are passed in and out. Low-level items should be grouped into data structures to reduce the coupling. The component can then refer to the data structure.

Reference 15 describes five types of coupling:

- data coupling;

- stamp coupling;
- control coupling;
- common coupling;
- content coupling.

Data coupling is most desirable and content coupling should be avoided. The other types of coupling are all acceptable under certain conditions, which are discussed below.

Data coupling is passing only the data needed. The couple name is a noun or noun phrase. The couple name can be the object in the component name. For example a component called 'CALCULATE_MEAN' might have an output data couple called 'MEAN'.

Stamp coupling is passing more data than is needed. The couple name is a noun or noun phrase. Stamp coupling should be avoided. Components should only be given the data that is relevant to their task. An example of stamp coupling is to pass a data structure called 'CENSUS_DATA' to a component that only requires the ages of the people in the census to calculate the average age.

Control coupling is communication by flags. The couple name may contain a verb. It is bad practice to use control coupling for the primary control flow; this should be implied by the component hierarchy. It is acceptable to use control coupling for secondary control flows such as in error handling.

Common coupling is communication by means of global data, and causes data to be less secure because faults in components that have access to the data area may corrupt it. Common coupling should normally be avoided, although it is sometimes necessary to provide access to data by components that do not have a direct call connection (e.g. if A calls B calls C, and A has to pass data to C).

Content coupling is communication by changing the instructions and internal data in another component. Communication coupling is bad because it prevents information hiding. Only older languages and assembler allow this type of coupling.

Practical methods of reducing coupling are to:

- structure the data;
- avoid using flags or semaphores for primary control flow and use messages or component calls;
- avoid passing data through components that do not use it;
- provide access to global or shared data by dedicated components.
- make sure that each component has a single entry point and exit point.

2.3.3.1.3 System shape

System shape describes the degree to which:

- the top components in a system are divorced from the physical aspects of the data they deal with;
- physical characteristics of the input and output data are independent.

If the system achieves these criteria to a high degree then the shape is said to be 'balanced'.

The system shape rule is: 'separate logical and physical functionality'.

2.3.3.1.4 Data structure match

In administrative data processing systems, the structure of the system should follow the structure of the data it deals with [Ref. 21]. A payroll package should deal with files at the highest levels, records at the middle levels and fields at the lowest levels.

The data structure match rule is: 'when appropriate, match the system structure to the data structure'.

2.3.3.1.5 Fan-in

The number of components that call a component measures its 'fan-in'. Reusability increases as fan-in increases.

Simple fan-in rules are: 'maximise fan-in' and 'reuse components as often as possible'.

2.3.3.1.6 Fan-out

The number of components that a component calls measures its 'fan-out'. Fan-out should usually be small, not more than seven. However some kinds of constructs force much higher fan-out values (e.g. case statements) and this is often acceptable.

Simple fan-out rules are: 'make the average fan-out seven or less' or 'make components depend on as few others as possible'.

2.3.3.1.7 Factoring

Factoring measures lack of duplication in software. A component may be factorised to increase cohesiveness and reduce size. Factoring creates reusable components with high fan-in. Factoring should be performed in the AD phase to identify utility software common to all components.

2.3.3.2 Complexity metrics

Suitable metrics for the design phases are listed in Table 2.3.3.2. This list has been compiled from ANSI/IEEE Std 982.1-1988 and 982.2-1988 [Ref. 17 and 18]. For a detailed discussion, the reader should consult the references.

2.3.3.3 Abstraction

An abstraction is 'a view of an object that focuses on the information relevant to a particular purpose and ignores the remainder of the information' [Ref. 2]. Design methods and tools should support abstraction by providing techniques for encapsulating functions and data in ways best suited to the view required. It should be possible to view a hierarchical design at various levels of detail.

The level of abstraction of a component should be related to its place within the structure of a system. For example a component that deals with files should delegate record handling operations to lower-level modules, as shown in Figures 2.3.3.3.A and B.

	Metric	Definition
1.	Number of Entries and Exits	measures the complexity by counting the entry and exit points in each component [Ref. 17, Section 4.13]
2	Software Science Measures	measures the complexity using the counts of operators and operands [Ref. 17, Section 4.14]
3.	Graph Theoretic Complexity	measures the design complexity in terms of numbers of components, interfaces and resources [Ref. 17, Section 4.15, and Ref. 23]. Similar to Cyclomatic Complexity.
4.	Cyclomatic Complexity	measures the component complexity from: Number of program flows between groups of program statements - Number of sequential groups of program statements + 2. A component with sequential flow from entry to exit has a cyclomatic complexity of 1-2+1=1 [Ref. 17, Section 4.16, and Ref. 23]
5.	Minimal Unit Test Case Determination	measures the complexity of a component from the number of independent paths through it, so that a minimal number of covering test cases can be generated for unit test [Ref. 17, Section 4.17]
6.	Design Structure	measures the design complexity in terms of its top-down profile, component dependence, component requirements for prior processing, database size and modularity and number of entry and exit points per component [Ref. 17, Section 4.19]
7.	Data Flow Complexity	measures the structural and procedural complexity in terms of component length, fan-in and fan-out [Ref. 17, Section 4.25]

Table 2.3.3.2: Complexity metrics

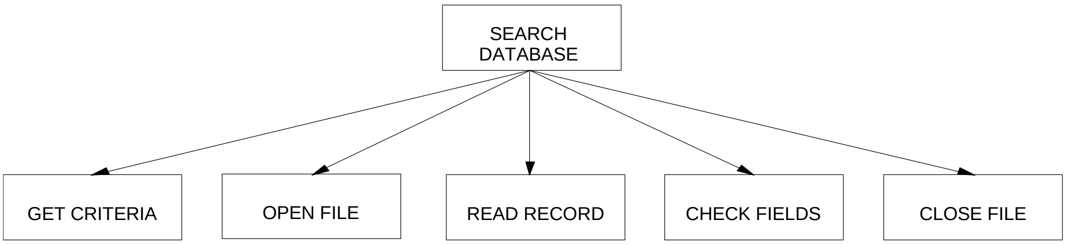

Figure 2.3.3.3A: Incorrectly layered abstractions

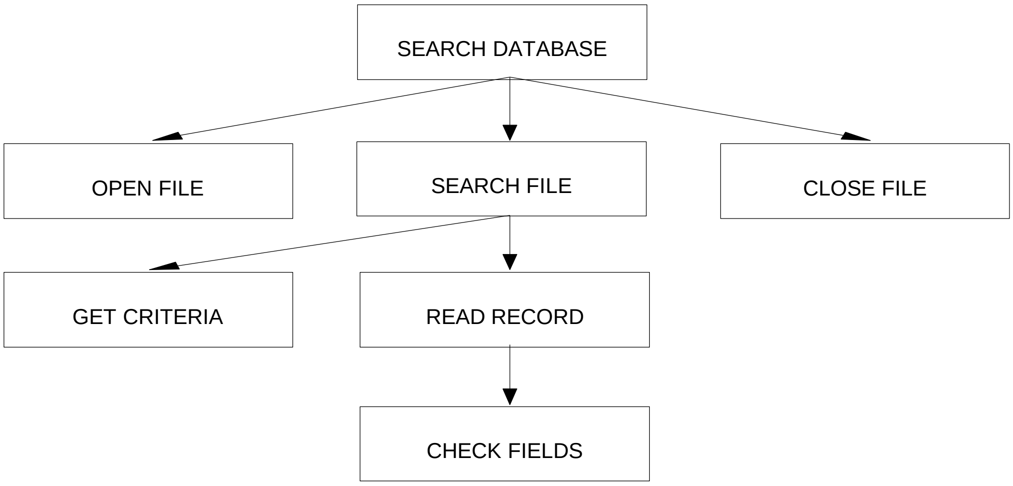

Figure 2.3.3.3B: Correctly layered abstractions.

2.3.3.4 Information hiding

Information hiding is 'the technique of encapsulating software design decisions in components so that the interfaces to the component reveal as little as possible about its inner working' (adapted from Reference 2). Components that hide information about their inner workings are 'black boxes'.

Information hiding is needed to enable developers to view the system at different layers of abstraction. Therefore it should not be necessary to understand any low-level processing to understand the high-level processing.

Sometimes information needs to be shared among components and this goes against the 'information hiding' principle. The 'information clustering' technique restricts access to information to as few components as possible. A library of components is often built to provide the sole means of access to information clusters such as common blocks.

2.3.3.5 Modularity

Modularity is 'the degree to which a system or computer program is composed of discrete components such that a change in one component has minimal impact on other components' [Ref. 2]. Component coupling is low in a modular design.

2.3.4 Prototyping

'Experimental prototypes' may be built to verify the correctness of a technical solution. If different designs meet the same requirements, prototypes can help identify the best choice. High risk, critical technologies should be prototyped in the AD phase. This is essential if a workable design is to be selected. Just as in the requirements phases, prototyping in the AD phase can avoid expensive redesign and recoding in later phases.

Two types of experimental prototypes can be distinguished: 'partial-system' prototypes and 'whole-system' prototypes. Partial-system prototypes are built to demonstrate the feasibility of a particular feature of the design. They are usually programmed in the language to be used in the DD phase. Whole-system prototypes are built to test the complete design. They may be built with CASE tools that permit execution of the design, or off-the-shelf tools that can be adapted for the application. Whole-system prototypes can help identify behavioural problems, such as resource contention.

2.3.5 Choosing a design approach

There is no unique design for any software system. Studies of the different options may be necessary. Alternative designs should be outlined and the trade-off between promising candidates discussed. The choice depends upon the type of system. In a real-time situation, performance and response time could be important, whereas in an administrative system stability of the data base might be more important.

The designer should consider the two major kinds of implementation for each software component:

- generating new components from scratch;
- reusing a component from another system or library.

New components are likely to need most maintenance. The expertise needed to maintain old components might not be available at all, and this can be an equally difficult problem.

Reuse generally leads to increased reliability because software failures peak at time of first release and decline thereafter. Unfortunately the logic of a system often has to be distorted to accommodate components not designed specifically for it, and this reduces adaptability. The structure of the system becomes over-strained and cannot stand the stresses imposed by additional features.

Developers should consider the requirements for maintainability, reliability, portability and adaptability in deciding whether to create new components or reuse old ones.

Only the selected design approach should be reflected in the ADD (and DDD). However the reasons for each major design decision should be recorded in the Project History Document. Modification of the software in later phases may require design decisions to be re-examined.

2.4 SPECIFICATION OF THE ARCHITECTURAL DESIGN

The architectural design is specified by identifying the components, defining the control and data flow between them, and stating for each of them the:

- functions to be performed;
- data input;
- data output;
- resource utilisation.

2.4.1 Assigning functions to components

The function of a component is defined by stating its objective or characteristic actions [Ref. 2].

The functional definition of a component should be derived from the functional requirements in the SRD, and references should be added to the component definition to record this.

When the architectural design is terminated at the task level, each task may be associated with many functional requirements. Nevertheless, there should be a strong association between the functions carried out by a component (i.e. each component should have high cohesion).

Internal processing should be described in terms of the control and data flow to lower-level components. The control flow may be described diagrammatically, using structure charts, transformation schemas or object diagrams. It should describe both nominal and non-nominal behaviour.

At the lowest design level, the processing done by each component should be defined. This may be done in natural language, Structured English or pseudo code. Parts of the functional specification may be reproduced or referenced to define what processing is required (e.g. state-transition diagrams, decision trees etc).

The data processing of each component can be defined by listing the data that the component deals with, and the operations that the component performs (e.g. create, read, update, delete). This information can be gathered from entity life histories [Ref. 4, 9]. The job of functional definition is then to describe how data processing operations are to be done. This approach ensures that no data processing functions are accidentally omitted.

2.4.2 Definition of data structures

Data structure definitions must include the:

- characteristics of each part (e.g. name, type, dimension) (AD10);
- relationships between parts (i.e. structure) (AD11);
- range of possible values of each part (AD12);
- initial values of each part (AD13).

The names of the parts of a data structure should be meaningful and unambiguous. They should leave no doubt about what the part is. Acronyms and obscure coding systems should be avoided in the architectural design.

The data type of each part of the data structure must be stated. Similarly the dimensionality of each item must be defined. Effort should be taken to keep the dimensionality of each part of the data structure as low as possible. Arrays should be homogeneous, with all the cells storing similar information.

The range of possible values of each part of the data structure should be stated. Knowledge about the possible values helps define the storage requirements. In the DD phase, defensive programming techniques use knowledge about the possible values to check the input to components.

The designer must define how data structures are initialised. Inputs that may change should be accessible to the user or operator (e.g. as in 'table-driven' software). Parameters that will never change (e.g.) need to be stored so that all components use the same value (e.g. FORTRAN include files of PARAMETER statements).

The accessibility of the data structures to components is an important consideration in architectural design, and has a crucial effect on the component coupling. Access to a data structure should be granted only on a 'need-to-know' basis and the temptation to define global data should be resisted.

Global data implies some form of data sharing, and even if access to the global data is controlled (e.g. by semaphores), a single component may corrupt the data area. Such failures are usually irrecoverable. It is, however, acceptable to make data global if it is the only way of achieving the necessary performance.

Data structures shared between a component and its immediate subordinates should be passed. When components are separated from one another in the hierarchy this technique becomes clumsy, and leads to the occurrence of 'tramp data', i.e. data that passes through components without being used. The best solution to the 'tramp data' problem is to make an 'information cluster' (see Section 2.3.3.4).

2.4.3 Definition of control flow

Control flow should always be described from initialisation to termination. It should be related to the system state, perhaps by the use of state-transition diagrams [Ref. 11].

The precise control flow mechanism should be described, for example:

- calls;
- interrupts;
- messages.

Control flow defines the component hierarchy. When control and data couples are removed from a structure chart, for example, the tree diagram that is left defines the component hierarchy.

2.4.3.1 Sequential and parallel control flow

Control flow may be sequential, parallel or a combination of the two. Decisions about sequential and parallel control flow are a crucial activity in architectural design. They should be based on the logical model. Functions that exchange data, even indirectly, must be arranged in sequence, whereas functions that have no such interface can execute in parallel.

Observations about sequential and parallel activities can affect how functions are allocated to components. For example, all the functions that execute in sequence may be allocated to a single task (or program) whereas functions that can execute in parallel may be allocated to different tasks.

2.4.3.2 Synchronous and asynchronous control flow

Control flow may be synchronous or asynchronous. The execution of two components is synchronous if, at any stage in the operation of one component, the state of the other is precisely determined. Sequential processing is always synchronous (when one component is doing something, the other must be doing nothing). Parallel control flow may also be synchronised, as in an array processor where multiple processors, each handling a different array element, are all synchronised by the same clock.

'Polling' is a synchronous activity. A component interrogates other lower-level components (which may be external) for information and takes appropriate action. Polling is a robust, easy-to-program mechanism but it can cause performance problems. To get an answer one has to ask the same question of everyone in turn, and this takes time.

Tasks running on separate computers run asynchronously, as do tasks running on a single processor under a multi-tasking operating system. Operations must be controlled by some type of handshake, such as a rendezvous. A 'scheduler' type task is usually necessary to coordinate all the other tasks.

An interrupt is an example of an asynchronous control flow. Interrupts force components to suspend their operation and pass control to other components. The exact component state when the interruption occurs cannot be predicted. Such non-deterministic behaviour makes verification and maintenance more difficult.

Another example of an asynchronous control flow is a 'one-way' message. A component sends a message and proceeds with its own processing without waiting for confirmation of reception. There is no handshake. Designers should only introduce this type of control flow if the rest of the processing of the sender makes no assumptions about the receiver state.

Asynchronous control flows can cause improvements in software performance; this advantage should be carefully traded-off against the extra complexity they may add.

2.4.4 Definition of the computer resource utilisation

The computer resources (e.g. CPU speed, memory, storage, system software) needed for development and operations must be estimated in the AD phase and defined in the ADD (AD15). For many software projects, the development and operational environments will be the same. Any resource requirements in the SRD will constrain the design.

Virtual memory systems have reduced the problem of main memory utilisation. Even so, the packaging of the components into units that execute together may still be necessary in some systems. These issues may have to be addressed in the AD phase, as they influence the division of work and how the software is integrated.

2.4.5 Selection of the programming language

The chosen programming languages should support top-down decomposition, structured programming and concurrent production and documentation. The programming language and the AD method should be compatible.

The relation between the AD method and programming language means that the choice should be made when the AD method is selected, at the beginning of the AD phase. However the process of design may lead to this initial choice of programming language being revised.

The selection of the programming languages to be used should be reviewed when the architectural design is complete.

Other considerations will affect the choice of programming language. Portability and maintenance considerations imply that assembler should be avoided. Portability considerations argue against using FORTRAN for real-time software because operating system services have to be called. The system services are not standard language features. Programming languages that allow meaningful names rather than terse, abbreviated names should be favoured since readable software is easier to maintain.

Standardised languages and high-quality compilers, debuggers, static and dynamic analysis tools are important considerations.

While it may be necessary to use more than one programming language (e.g a mixed microprocessor and minicomputer configuration), maintenance is easier if a single programming language is used.

It is not sufficient just to define the language to be used. The particular language standard and compiler should also be defined during the AD phase. The rules for using non-standard features allowed by a compiler should be formulated. Language extensions should be avoided as this makes the software less portable. However, performance and maintenance requirements may sometimes outweigh those of portability and make some non-standard features acceptable.

2.5 INTEGRATION TEST PLANNING

Integration Test Plans must be generated in the AD phase and documented in the Integration Test section of the Software Verification and Validation Plan (SVVP/IT). The Integration Test Plan should describe the scope, approach and resources required for the integration tests, and take account of the verification requirements in the SRD. See Chapter 6.

2.6 PLANNING THE DETAILED DESIGN PHASE

Plans of DD phase activities must be drawn up in the AD phase. Generation of DD phase plans is discussed in Chapter 6. These plans cover project management, configuration management, verification and validation and quality assurance. Outputs are the:

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

2.7 THE ARCHITECTURAL DESIGN PHASE REVIEW

Production of the ADD and SVVP/IT is an iterative process. The development team should hold walkthroughs and internal reviews of a document before its formal review.

The outputs of the AD phase shall be formally reviewed during the Architectural Design Review (AD16). This should be a technical review. The recommended procedure is described in ESA PSS-05-10, and is based closely on the IEEE standard for Technical Reviews [Ref. 8].

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

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

- ADD describes the optimal solution to the problem stated in the SRD;
- ADD describes the architectural design clearly, completely and in sufficient detail to enable the detailed design process to be started;
- SVVP/IT is an adequate plan for integration testing of the software in the DD phase.

An AD/R begins when the documents are distributed to participants for review. A problem with a document is described in a 'Review Item Discrepancy' (RID) form that may be prepared by any participant. Review meetings are then held that have the documents and RIDs as input. A review meeting should discuss all the RIDs and either accept or reject them. The review meeting may also discuss possible solutions to the problems raised by them. The output of the meeting includes the processed RIDs.

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

This page is intentionally left blank

CHAPTER 3
METHODS FOR ARCHITECTURAL DESIGN

3.1 INTRODUCTION

An architectural design method should provide a systematic way of defining the software components. Any method should facilitate the production of a high-quality, efficient design that meets all the requirements.

This guide summarises a range of design methods to enable the designer to make a choice. The details about each method should be obtained from the references. This guide does not seek either to make any particular method a standard or to define a set of acceptable methods. Each project should examine its needs, choose a method (or combination of methods) and justify the selection in the ADD. To help make this choice, this guide summarises some well-known methods and discusses how they can be applied in the AD phase.

Methods that may prove useful for architectural design are:

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

The AD method can affect the choice of programming language. Early versions of Structured Design, for example, decompose the software to fit a control flow concept based on the call structure of a third generation programming language such as FORTRAN, C and COBOL. Similarly, the use of object-oriented design implies the use of an object-oriented programming language such as C++, Smalltalk and Ada.

3.2 STRUCTURED DESIGN

Structured Design is not a single method, but a name for a class of methods. Members of this class are:

- Yourdon methods (Yourdon/Constantine and Ward/Mellor);

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

- Structured Systems Analysis and Design Methodology (SSADM).

Yourdon methods are widely used in the USA and Europe. SSADM is recommended by the UK government for 'data processing systems'. It is now under the control of the British Standards Institute (BSI). SADT has been successfully used within ESA for some time.

3.2.1 Yourdon methods

Structured Design is 'a disciplined approach to software design that adheres to a specified set of rules based on principles such as top-down design, stepwise refinement and data flow analysis' [Ref. 1]. In the Yourdon approach [Ref. 11, 16, 20], the technique can be characterised by the use of:

- Structure Charts to show hierarchy and control and data flow;

- Pseudo Code to describe the processing.

The well-defined path between the analysis and design phases is a major advantage of structured methods. DeMarco's version of structured analysis defines four kinds of model. Table 3.2.1 relates the models to the ESA life cycle phase in which they should be built.

Model No	Model Name	ESA PSS-05-0 phase
1	current physical model	SR phase
2	current logical model	SR phase
3	required logical model	SR phase
4	required physical model	AD phase

Table 3.2.1: Structured Analysis Models

The required physical model should be defined at the beginning the AD phase since it is input to the Structured Design process.

Structured Design defines concepts for assessing the quality of design produced by any method. It has also pioneered concepts, now found in more recent methods, such as:

- allowing the form of the problem to guide the solution;

[1] Trademark of SoftTech Inc, Waltham, Mass, USA

- attempting to reduce complexity via partitioning and hierarchical organisation;
- making systems understandable through rigorous diagramming techniques.

Structured Design methods arose out of experience of building 'data processing' systems and so early methods could only describe sequential control flow. A more recent extension of structured methods developed by Ward and Mellor catered for real-time systems [Ref. 11] by adding extra features such as:

- control transformations and event flows in DFDs;
- time-continuous and time-discrete behaviour in DFDs;
- Petri Nets, to verify the executability of DFDs [Ref. 13];
- State-Transition Diagrams to describe control transformations.

Ward and Mellor define the purpose of the analysis phase as the creation of an 'essential model'. In the design phase the 'essential model' is transformed to a computer 'implementation model'. The implementation model is progressively designed in a top-down manner by allocating functions to processors, tasks and modules.

Transformation Schemas are used to describe the design at the processor level, where there may be parallelism. Structure Charts are used at the task level and below, where the control flow is sequential.

3.2.2 SADT

Structured Analysis and Design Technique (SADT) uses the same diagramming techniques for both analysis and design [Ref. 3]. Unlike the early versions of Yourdon, SADT has always catered for real-time systems modelling, because it permits the representation of control flows. However SADT diagrams need to be supplemented with State-Transition Diagrams to define behaviour.

SADT diagram boxes should represent components. Each box should be labelled with the component identifier. Control flows should represent calls, interrupts and loading operations. Input and output data flows should represent the files, messages and calling arguments passed between components. Mechanism flows should carry special information about the implementation of the component.

3.2.3 SSADM

SSADM is a detailed methodology with six activities [Ref. 4, 9]:

1. analysis of the current situation;

2. specification of requirements;

3. selection of system option;

4. logical data design;

5. logical process design;

6. physical design.

In ESA projects, activities 1 and 2 should take place in the UR and SR phases respectively. The output of 2 is a required logical model of the software, which should be described in the SRD. Activities 3 to 6 should take place in the AD phase. The ADD should describe the physical design.

SSADM results in a process model and a data model. The models are built in parallel and each is used to check the consistency of the other. This built-in consistency checking approach is the major advantage that SSADM offers. Data modelling is treated more thoroughly by SSADM than by the other methods. SSADM does not support the design of real-time systems.

3.3 OBJECT-ORIENTED DESIGN

Object-Oriented Design (OOD) is an approach to software design based on objects and classes. An object is 'an abstraction of something in the domain of a problem or its implementation, reflecting the capabilities of a system to keep information about it, interact with it or both; an encapsulation of attribute values and their exclusive services' [Ref. 24]. A class describes a set of objects with common attributes and services. An object is an 'instance' of a class and the act of creating an object is called 'instantiation'.

Classes may be entities with abstract data types, such as 'Telemetry Packet' and 'Spectrum', as well as simpler entities with primitive data types such as real numbers, integers and character strings. A class is defined by its attributes and services. For example the 'Spectrum' class would have attributes such as 'minimum frequency', 'centre frequency', 'maximum frequency' and services such as 'calibrate'.

Classes can be decomposed into subclasses. There might be several types of Telemetry Packet for example, and subclasses of Telemetry Packet such as 'Photometer Packet' and 'Spectrometer Packet' may be created. Subclasses share family characteristics, and OOD provides for this by permitting subclasses to inherit operations and attributes from their parents. This leads to modular, structured systems that require less code to implement. Common code is automatically localised in a top-down way.

Object-oriented design methods offer better support for reuse than other methods. The traditional bottom-up reuse mechanism where an application module calls a library module is of course possible. In addition, the inheritance feature permits top-down reuse of superclass attributes and operations.

OOD combines information and services, leading to increased modularity. Control and data structures can be defined in an integrated way.

Other features of the object-oriented approach besides classes, objects and inheritance are message passing and polymorphism. Objects send messages to other objects to command their services. Messages are also used to pass information. Polymorphism is the capability, at runtime, to refer to instances of various classes. Polymorphism is often implemented by allowing 'dynamic binding'.

The object-oriented features such as polymorphism rely on dynamic memory allocation. This can make the performance of object-oriented software unpredictable. Care should be taken when programming real-time applications to ensure that functions that must complete by a defined deadline have their processing fully defined at compile-time, not run-time.

OOD is most effective when implemented through an object-oriented programming language that supports object definition, inheritance, message passing and polymorphism. Smalltalk, C++, Eiffel and Object Pascal support all these features.

Object-oriented techniques have been shown to be much more suitable for implementing event-driven software, such as Graphical User Interfaces (GUIs), than structured methods. Many CASE tools for structured methods are implemented by means of object-oriented techniques.

If object-oriented methods are to be used, they should be used throughout the life cycle. This means that OOD should only be selected for the AD phase if Object-Oriented Analysis (OOA) have been used in the SR phase. The seamless transition from analysis to design to programming is a major advantage of the object-oriented methods, as it facilitates iteration. An early paper by Booch [Ref. 13] describes a technique for transforming a logical model built using structured analysis to a physical model using object-oriented design. In practice the results have not been satisfactory. The structured analysis view is based on functions and data and the object-oriented view is based on classes, objects, attributes and services. The views are quite different, and it is difficult to hold both in mind simultaneously.

Like Structured Design, Object-Oriented Design is not a single method, but a name for a class of methods. Members of this class include:

- Booch;
- Hierarchical Object-Oriented Design (HOOD).
- Coad-Yourdon;
- Object Modelling Technique (OMT) from Rumbaugh et al;
- Shlaer-Mellor.

These methods are discussed below.

3.3.1 Booch

Booch originated object-oriented design [Ref. 22], and continues to play a leading role in the development of the method [Ref. 26].

Booch models an object-oriented design in terms of a logical view, which defines the classes, objects, and their relationships, and a physical view, which defines the module and process architecture. The logical view corresponds to the logical model that ESA PSS-05-0 requires software engineers to construct in the SR phase (see ESA PSS-05-03, Guide to the Software Requirements Definition Phase). The physical view corresponds to the physical model that ESA PSS-05-0 requires software engineers to construct in the AD phase.

Booch provides two diagramming techniques for documenting the physical view:

- module diagrams, which are used to show the allocation of classes and objects to modules such as programs, packages and tasks in the physical design (the term 'module' in Booch's method is used to describe any design component);
- process diagrams, which show the allocation of modules to hardware processors.

Booch's books on Object-Oriented Design contain many insights into good design practise. However Booch's notation is cumbersome and few tools are available.

3.3.2 HOOD

Hierarchical Object-Oriented Design (HOOD) [Ref. 14] is one of a family of object-oriented methods that try to marry object-orientation with structured design methods (see Reference 25 for another example). Hierarchy follows naturally from decomposition of the top-level 'root' object. As in structured design, data couples flow between software components. The main difference between HOOD and structured methods is that software components get their identity from their correspondence to things in the real world, rather than to functions that the system has to perform.

HOOD was originally designed to be used with Ada, although Ada does not support inheritance, and is not an object-oriented programming language. This is not serious problem for the HOOD designer, because the method does not use classes to structure systems.

HOOD does not have a complementary analysis method. The logical model is normally built using structured analysis. Transformation of the logical model to the physical model is difficult, making it hard to construct a coherent design.

HOOD has been used embedded Ada applications. It has a niche, but is unlikely to become as widespread and well supported by tools as, say, OMT.

3.3.3 Coad and Yourdon

Coad and Yourdon have published an integrated approach to object-oriented analysis and design [Ref. 24]. An object-oriented design is constructed from four components:

- problem domain component;
- human interaction component;
- task management component;
- data management component.

Each component is composed of classes and objects. The problem domain component is based on the (logical) model built with OOA in the analysis phase. It defines the subject matter of the system and its responsibilities. If the system is to be implemented in an object-oriented language, the correspondence between problem domain classes and objects will be one to one, and the problem domain component can be directly programmed. Reuse considerations, and the non-availability of a fully object-oriented programming language, may make the design of the problem domain component depart from ideal represented by the OOA model.

The human interaction component handles sending and receiving messages to and from the user. The classes and objects in the human interaction component have names taken from the user interface language, e.g. window and menu.

Many systems will have multiple threads of execution, and the designer must construct a task management component to organise the processing. The designer needs to define tasks as event-driven or clock-driven, as well as their priority and criticality.

The data management component provides the infrastructure to store and retrieve objects. It may be a simple file system, a relational database management system, or even an object-oriented database management system.

The four components together make the physical model of the system. At the top level, all Coad and Yourdon Object-Oriented Designs have the same structure.

Classes and objects are organised into 'generalisation-specialisation' and 'whole-part' structures. Generalisation-specialisation structures are 'family trees', with children inheriting the attributes of their parents. Whole-part structures are formed when an object is decomposed.

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

3.3.4 OMT

The Object Modelling Technique (OMT) of Rumbaugh et al [Ref 26] contains two design activities:

- system design;
- object design.

System design should be performed in the AD phase. The object design should be performed in the DD phase.

The steps of system design are conventional and are:

- organise the system into subsystems and arrange them in layers and partitions;
- identify concurrency inherent in the problem;
- allocate subsystems to processors;
- define the data management implementation strategy;
- identify global resources and define the mechanism for controlling access to them;
- choose an approach to implementing software control;
- consider boundary conditions;
- establish trade-off priorities.

Many systems are quite similar, and Rumbaugh suggests that system designs be based on one of several frameworks or 'canonical architectures'. The ones they propose are:

- batch transformation - a data transformation performed once on an entire input set;
- continuous transformation - a data transformation performed continuously as inputs change;
- interactive interface - a system dominated by external interactions;
- dynamic simulation - a system that simulates evolving real world objects;
- real-time system - a system dominated by strict timing constraints;
- transaction manager - a system concerned with storing and updating data.

The OMT system design approach contains many design ideas that are generally applicable.

3.3.5 Shlaer-Mellor

Shlaer and Mellor [Ref 28] describe an Object-Oriented Design Language (OODLE), derived from the Booch and Buhr notation. There are four types of diagram:

- class diagram;
- class structure chart;
- dependency diagram;
- inheritance diagram.

There is a class diagram for each class. The class diagram defines the operations and attributes of the class.

The class structure chart defines the module structure of the class, and the control and data flow between the modules of the class. There is a class structure chart for each class.

Dependency diagrams illustrate the dependencies between classes, which may be:

- client-server;
- friends.

A client server dependency exists when a class (the client) calls upon the operations of another class (the server).

A friendship dependency exists when one class access the internal data of another class. This is an information-hiding violation.

Inheritance diagrams show the inheritance relationships between classes.

Shlaer and Mellor define a ‘recursive design ’method that uses the OODLE notation as follows:

- define how the generic computing processes will be implemented;
- implement the object model classes using the generic computing processes.

The Shlaer-Mellor design approach is more complex than that of other object-oriented methods.

3.4 JACKSON SYSTEM DEVELOPMENT

Jackson System Development method should only be used in the AD phase if it has been used in the SR phase [Ref. 12]. In this case the SRD will contain:

- definitions of the real-world entities with which the software is to be concerned;
- definitions of the actions of these entities in the real world;
- a description of that part of the real world that is to be simulated in software;
- specifications of the functions that the software will perform, i.e. a selection of real-world actions that the software will emulate.

Table 3.4 shows how the JSD activities map to the ESA PSS-05-0 life cycle.

The first JSD task in the AD phase is matching processes to processors, i.e. deciding what computer runs each process. In JSD there may be a process for each instance of an entity type. The designer must make sure that the computer has enough performance to run all the instances of the process type.

The second JSD task in the AD phase is to transform process specifications. Unless there is one computer available for each process described in the SRD (e.g. a dedicated control system), it will be necessary to transform the specification of each process into a form capable of execution on a computer. Rather than having many individual processes that run once, it is necessary to have a single generic process that runs many times. A computer program consists of data and instructions. To transform a process into a program, the instruction and data components of each process must be separated. In JSD, the data component of a process is called the 'state-vector'. The state-vector also retains information on the stage that processing has got to, so that individual processes can be resumed at the correct point. The state-vector of a process should be specified in the AD phase. A state-vector should reside in permanent storage. The executable portion of a process should be specified in the AD phase in pseudo code. This may be carried over from the SRD without major modification; introduction of detailed implementation considerations should be deferred until the DD phase.

The other major JSD task in the AD phase is to schedule processing. The execution of the processing has to be coordinated. In JSD this is done by introducing a 'scheduling' process that activates and deactivates other processes and controls their execution.

JSD Activity			
Level 0	**Level 1**	**Level 2**	**Level 3**
Specification (SR Phase)	Specify Model of Reality	Develop Model Abstractly	Write Entity-Action List
			Draw Entity-Structure Diagrams
		Define Initial versions of Model processes	
	Specify System Functions	Add functions to model processes	
		Add timing constraints	
Implementation (AD Phase)	Match processes to processors		
	Transform processes		
	Devise Scheduler		
Implementation (DD Phase)	Define Database		

Table 3.4: JSD activities and the ESA PSS-05-0 life cycle

3.5 FORMAL METHODS

Formal methods aim to produce a rigorous specification of the software in terms of an agreed notation with well-defined semantics. Formal methods should only be applied in the AD phase if they have been used in the SR phase. See ESA PSS-05-02, Guide to the Software Requirements Definition Phase, for a discussion of Formal Methods. Formal methods may be used when the software is safety-critical, or when the availability and security requirements are very demanding.

Development of a formal specification should proceed by increasing the detail until direct realisation of the specification in a programming language is possible. This procedure is sometimes called 'reification' [Ref. 10].

The AD phase of a software development using a formal method is marked by the taking of the first implementation decisions about the software. Relevant questions are: 'how are the abstract data types to be represented? and 'how are operations to be implemented? These decisions may constrain subsequent iterations of the specification. When a Formal Method is being used, the AD phase should end when the specification can be coded in a high-level programming language.

This page is intentionally left blank.

CHAPTER 4
TOOLS FOR ARCHITECTURAL DESIGN

4.1 INTRODUCTION

This chapter discusses the tools for constructing a physical model and specifying the architectural design. Tools can be combined to suit the needs of a particular project.

4.2 TOOLS FOR PHYSICAL MODEL CONSTRUCTION

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

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

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

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

CASE Tools should have an integrated data dictionary or another kind or 'repository'. This is necessary for consistency checking. Developers should check that a tool supports the parts of the method that they intend to use. Appendix E contains a more detailed list of desirable tool capabilities.

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

4.3 TOOLS FOR ARCHITECTURAL DESIGN SPECIFICATION

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

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

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

CHAPTER 5
THE ARCHITECTURAL DESIGN DOCUMENT

5.1 INTRODUCTION

The ADD defines the framework of the solution. The ADD must be an output from the AD phase (AD19). The ADD must be sufficiently detailed to allow the project leader to draw up a detailed implementation plan and to control the overall project during the remaining development phases (AD23). It should be detailed enough to define the integration tests. Components (especially interfaces) should be defined in sufficient detail so that programmers, or teams of programmers, can work independently. The ADD is incomplete if programmers have to make decisions that affect programmers working on other components. Provided it meets these goals, the smaller the ADD, the more readable and reviewable it is.

The ADD should:

- avoid extensive specification of module processing (this is done in the DD phase);
- not cover the project management, configuration management, verification and quality assurance aspects (which are covered by the SPMP, SCMP, SVVP and SQAP).

Only the selected design approach should be reflected in the ADD (AD05). Descriptions of options and analyses of trade-offs should be archived by the project and reviewed in the Project History Document (PHD).

5.2 STYLE

The style of an ADD should be systematic and rigorous. The ADD should be clear, consistent and modifiable.

5.2.1 Clarity

An ADD is clear if each component has definite inputs, outputs, function and relationships to other components. The natural language used in an ADD should be shared by all members of the development team.

Explanatory text, written in natural language, should be included to enable review by those not familiar with the design method.

5.2.2 Consistency

The ADD must be consistent (AD22). There are several types of inconsistency:

- different terms used for the same thing;
- the same term used for different things;
- incompatible activities happening simultaneously;
- activities happening in the wrong order;
- using two different components to perform the same function.

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

Duplication and overlap lead to inconsistency. If the same functional requirement can be traced to more than one component, this will be a clue to inconsistency. Methods and tools help consistency to be achieved.

5.2.3 Modifiability

An ADD is modifiable if design changes can be documented easily, completely and consistently.

5.3 EVOLUTION

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

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

5.4 RESPONSIBILITY

The developer is responsible for producing the ADD. The developer should nominate people with proven design and implementation skills to write it.

5.5 MEDIUM

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

5.6 CONTENT

The ADD must define the major software components and the interfaces between them (AD17). Components may be systems, subsystems, data stores, components, programs and processes. The types of software components will reflect the AD method used. At the end of the AD phase the descriptions of the components are assembled into an ADD.

The ADD must be compiled according to the table of contents provided in Appendix C of ESA PSS-05-0 (AD24). This table of contents is derived from ANSI/IEEE Std 1016-1987 'Software Design Descriptions' [Ref. 17]. This standard defines a Software Design Description as a 'representation or model of the software system to be created. The model should provide the precise design information needed for the planning, analysis and implementation of the software system'. The ADD should be such a Software Design Description.

References should be given where appropriate, but an ADD should not refer to documents that follow it in the ESA PSS-05-0 life cycle. An ADD should contain no TBCs or TBDs by the time of the Architectural Design Review.

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

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

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

2 System Overview

3 System Context
3.n External interface definition

4 System Design
4.1 Design method
4.2 Decomposition description

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

6 Feasibility and Resource Estimates

7 Software Requirements vs Components Traceability matrix

5.6.1 ADD/1 INTRODUCTION

This section should provide an overview of the entire document and a description of the scope of the software.

5.6.1.1 ADD/1.1 Purpose (of the document)

This section should:

(1) describe the purpose of the particular ADD;

(2) specify the intended readership of the ADD.

5.6.1.2 ADD/1.2 Scope (of the software)

This section should:

(1) identify the software products to be produced;

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

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

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

5.6.1.3 ADD/1.3 Definitions, acronyms and abbreviations

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

5.6.1.4 ADD/1.4 References

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

5.6.1.5 ADD/1.5 Overview (of the document)

This section should:

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

(2) explain how the ADD is organised.

This section is not included in the table of contents provided in ESA PSS-05-0, and is therefore optional.

5.6.2 ADD/2 SYSTEM OVERVIEW

This section should briefly introduce to the system context and design, and discuss the background to the project.

This section may summarise the costs and benefits of the selected architecture, and may refer to trade-off studies and prototyping exercises.

5.6.3 ADD/3 SYSTEM CONTEXT

This section should define all the external interfaces (AD18). This discussion should be based on a system block diagram or context diagram to illustrate the relationship between this system and other systems.

5.6.4 ADD/4 SYSTEM DESIGN

5.6.4.1 ADD/4.1 Design Method

The design method used should be named and referenced. A brief description may be added to aid readers not familiar with the method. Any deviations and extensions of the method should be explained and justified.

5.6.4.2 ADD/4.2 Decomposition description

The software components should be summarised. This should be presented as structure charts or object diagrams showing the hierarchy, control flow (AD14) and data flow between the components.

Components can be organised in various ways to provide the views needed by different members of the development organisation. ANSI/IEEE 1016-1987 describes three possible ways of presenting the software design. They are the 'Decomposition View', the 'Dependency View' and the 'Interface View'. Ideally, all views should be provided.

The Decomposition View shows the component breakdown. It defines the system using one or more of the identity, type, purpose, function and subordinate parts of the component description. Suitable methods are tables listing component identities with a one-line summary of their purpose. The intended readership consists of managers, for work package definition, and development personnel, to trace or cross-reference components and functions.

The Dependency View emphasises the relationships among the components. It should define the system using one or more of the identity, type, purpose, dependency and resource parts of the component description. Suitable methods of presentation are Structure Charts and Object Diagrams. The intended readership of this description consists of managers, for the formulation of the order of implementation of work packages, and maintenance personnel, who need to assess the impact of design changes.

The Interface View emphasises the functionality of the components and their interfaces. It should define the system using one or more of the identity, functions and interface parts of the component description. Suitable methods of presentation are interface files and parameter tables. The intended readership of this description comprises designers, programmers and testers, who need to know how to use the components in the system.

5.6.5 ADD/3 COMPONENT DESCRIPTION

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

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

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

5.6.5.1 ADD/5.n Component Identifier

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

The identifier should reflect the purpose and function of the component and be brief yet be meaningful. If abbreviation is necessary, abbreviations should be applied consistently and without ambiguity. Abbreviations should be documented. Component identifiers should be mutually consistent (e.g. if there is a routine called READ_RECORD then one might expect a routine called WRITE_RECORD, not RECORD_WRITING_ROUTINE).

5.6.5.1.1 ADD/5.n.1 Type

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

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

5.6.5.1.2 ADD/5.n.2 Purpose

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

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

5.6.5.1.3 ADD/5.n.3 Function

The function of a component must be defined in the ADD (AD07). This should be done by stating what the component does.

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

- the process;
- the information stored or transmitted.

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

5.6.5.1.4 ADD/5.n.4 Subordinates

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

5.6.5.1.5 ADD/5.n.5 Dependencies

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

- 'what operations have to have taken place before this component is called?'
- 'what operations are excluded when this operation is taking place?'
- 'what components have to be executed after this one?'.

5.6.5.1.6 ADD/5.n.6 Interfaces

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

The control flow to and from a component should be defined in terms of how execution of the component is to be started (e.g. subroutine call) and how it is to be terminated (e.g. return). This may be implicit in the definition of the type of component, and a description may not be necessary. Control flows may also take place during execution (e.g. interrupt) and these should be defined, if they exist.

The data flow input to and output from each component must be detailed in the ADD (AD06, AD08, AD09). Data structures should be identified that:

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

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

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

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

If a component interfaces to components in the same system then the interface description should be defined in the ADD. If a component interfaces to components in other systems, the interface description should be defined in an Interface Control Document (ICD).

5.6.5.1.7 ADD/5.n.7 Resources

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

5.6.5.1.8 ADD/5.n.8 References

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

5.6.5.1.9 ADD/5.n.9 Processing

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

The ADD should not provide a complete, exhaustive specification of the processing to be done by bottom-level components; this should be done in the Detailed Design and Production Phase. The processing of higher-level components, especially the control flow, data flow and state transitions should be specified.

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

5.6.5.1.10 ADD/5.n.10 Data

The data internal to a component should be defined. The amount of detail required depends strongly on the type of component. The logical and physical data structure of files that interface major components should be defined in detail. The specification of the data internal to a major component should be postponed to the DD phase.

Data structure definitions must include the:

- description of each element (e.g. name, type, dimension) (AD10);
- relationships between the elements (i.e. the structure, AD11);
- range of possible values of each element (AD12);
- initial values of each element (AD13).

5.6.6 ADD/6 FEASIBILITY AND RESOURCE ESTIMATES

This section should contain a summary of the computer resources required to build, operate and maintain the software (AD15).

5.6.7 ADD/7 SOFTWARE REQUIREMENTS TRACEABILITY MATRIX

This section should contain a table that summarises how each software requirement has been met in the ADD (AD21). The tabular format permits one-to-one and one-to-many relationships to be shown. A template is provided in Appendix D.

This page is intentionally left blank.

CHAPTER 6
LIFE CYCLE MANAGEMENT ACTIVITIES

6.1 INTRODUCTION

AD phase activities must be carried out according to the plans defined in the SR phase (AD01). These are:

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

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

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

6.2 PROJECT MANAGEMENT PLAN FOR THE DD PHASE

By the end of the AD review, the DD phase section of the SPMP (SPMP/DD) must be produced (SPM08). The SPMP/DD describes, in detail, the project activities to be carried out in the DD phase.

An estimate of the total project cost must be included in the SPMP/DD (SPM09). Every effort should be made to arrive at a total project cost estimates with an accuracy better than 10%. The SPMP/DD must contain a Work-Breakdown Structure (WBS) that is directly related to the breakdown of the software into components (SPM10).

The SPMP/DD must contain a planning network showing the relationships between the coding testing and integration activities (SPM11). No software production packages in the SPMP/DD must last longer than 1 man-month (SPM12).

Cost estimation methods can be used to estimate the total number of man-months and the implementation schedule needed for the DD phase. Estimates of the number of lines of code, supplemented by many other parameters, are the usual input. When properly calibrated (using data collected from previous projects done by the same organisation), such models can yield predictions that are 20% accurate 68% of the time (TRW's initial calibration of the COCOMO model). Even if cost models are used, technical knowledge and experience gained on similar projects need to be used to arrive at the desired 10% accurate cost estimate.

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

6.3 CONFIGURATION MANAGEMENT PLAN FOR THE DD PHASE

During the AD phase, the DD phase section of the SCMP (SCMP/DD) must be produced (SCM46). The SCMP/DD must cover the configuration management procedures for documentation, deliverable code, and any CASE tool outputs or prototype code, to be produced in the DD phase (SCM47).

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

6.4 VERIFICATION AND VALIDATION PLAN FOR THE DD PHASE

During the AD phase, the DD phase section of the SVVP (SVVP/DD) must be produced (SVV15). The SVVP/DD must define how the DDD and the code are evaluated by defining the review and traceability procedures. It may include specifications of the tests to be performed with prototypes.

The planning of the integration tests should proceed in parallel with the definition of the architectural design.

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

6.5 QUALITY ASSURANCE PLAN FOR THE DD PHASE

During the AD phase, the DD phase section of the SQAP (SQAP/DD) must be produced (SQA08). The SQAP/DD must describe, in detail, the quality assurance activities to be carried out in the DD phase (SQA09).

SQA activities include monitoring the following activities:

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

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

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

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

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

6.6 INTEGRATION TEST PLANS

The developer must construct an integration test plan in the AD phase and document it in the SVVP (SVV17). This plan should define the scope, approach, resources and schedule of integration testing activities.

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

- where will the integration tests be done?
- who will attend?
- who will carry them out?
- are tests needed for all software requirements?
- must any special test software be used?
- how long is the integration testing programme expected to last?
- are simulations necessary?

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

APPENDIX A
GLOSSARY

A.1 LIST OF ACRONYMS

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

AD	Architectural Design
ADD	Architectural Design Document
AD/R	Architectural Design Review
ANSI	American National Standards Institute
BSSC	Board for Software Standardisation and Control
CASE	Computer Aided Software Engineering
DD	Detailed Design and production
DFD	Data Flow Diagram
ESA	European Space Agency
GUI	Graphical User Interface
HOOD	Hierarchical Object Oriented Design
IEEE	Institute of Electrical and Electronics Engineers
ISO	International Standards Organisation
ICD	Interface Control Document
JSD	Jackson System Development
MTBF	Mean Time Between Failures
MTTR	Mean Time To Repair
OOA	Object-Oriented Analysis
OOD	Object-Oriented Design
OSI	Open Systems Interconnection
PA	Product Assurance
PSS	Procedures, Specifications and Standards
QA	Quality Assurance
RID	Review Item Discrepancy
SADT	Structured Analysis and Design Technique
SCM	Software Configuration Management
SCMP	Software Configuration Management Plan
SPM	Software Project Management
SPMP	Software Project Management Plan
SQA	Software Quality Assurance
SQAP	Software Quality Assurance Plan
SR	Software Requirements
SRD	Software Requirements Document

SR/R	Software Requirements Review
SSADM	Structured Systems Analysis and Design Methodology
ST	System Test
SUM	Software User Manual
SVVP	Software Verification and Validation Plan
TBC	To Be Confirmed
TBD	To Be Defined
UR	User Requirements
URD	User Requirements Document
UR/R	User Requirements Review

APPENDIX B
REFERENCES

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

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

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

4. SSADM Version 4, NCC Blackwell Publications, 1991

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

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

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

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

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

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

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

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

13. Object Oriented Development, G.Booch, in IEEE Transactions on Software Engineering, VOL SE-12, February 1986.

14. Hood Reference Manual, Issue 3, Draft C, Reference WME/89-173/JB HOOD Working Group, ESTEC, 1989.

15. The Practical Guide to Structured Systems Design, M.Page-Jones, Yourdon Press, 1980.

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

17. IEEE Standard Dictionary of Measures to Produce Reliable Software, IEEE Std 982.1-1988.

18. IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software, IEEE Std 982.2-1988.

19. Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design, E. Yourdon and L. Constantine, Yourdon Press, 1978.

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

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

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

23. A complexity Measure, T.J.McCabe, IEEE Transactions on Software Engineering, Vol. SE-2, No. 4, December 1976.

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

25. The Object-Oriented Structured Design Notation for Software Design Representation, A.I. Wasserman, P.A. Pircher and R.J. Muller, Computer, March 1990.

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

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

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

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

APPENDIX C
MANDATORY PRACTICES

This appendix is repeated from ESA PSS-05-0, appendix D.4

AD01 AD phase activities shall be carried out according to the plans defined in the SR phase.

AD02 A recognised method for software design shall be adopted and applied consistently in the AD phase.

AD03 The developer shall construct a 'physical model', which describes the design of the software using implementation terminology.

AD04 The method used to decompose the software into its component parts shall permit a top-down approach.

AD05 Only the selected design approach shall be reflected in the ADD.

For each component the following information shall be detailed in the ADD:

AD06 • data input;

AD07 • functions to be performed;

AD08 • data output.

AD09 Data structures that interface components shall be defined in the ADD.

Data structure definitions shall include the:

AD10 • description of each element (e.g. name, type, dimension);

AD11 • relationships between the elements (i.e. the structure);

AD12 • range of possible values of each element;

AD13 • initial values of each element.

AD14 The control flow between the components shall be defined in the ADD.

AD15 The computer resources (e.g. CPU speed, memory, storage, system software) needed in the development environment and the operational environment shall be estimated in the AD phase and defined in the ADD.

AD16 The outputs of the AD phase shall be formally reviewed during the Architectural Design Review.

AD17 The ADD shall define the major components of the software and the interfaces between them.

AD18 The ADD shall define or reference all external interfaces.

AD19 The ADD shall be an output from the AD phase.

AD20 The ADD shall be complete, covering all the software requirements described in the SRD.

AD21 A table cross-referencing software requirements to parts of the architectural design shall be placed in the ADD.

AD22 The ADD shall be consistent.

AD23 The ADD shall be sufficiently detailed to allow the project leader to draw up a detailed implementation plan and to control the overall project during the remaining development phases.

AD24 The ADD shall be compiled according to the table of contents provided in Appendix C.

APPENDIX D
REQUIREMENTS TRACEABILITY MATRIX

<table>
<tr><td colspan="3"><b>REQUIREMENTS TRACEABILITY MATRIX</b>
<b>ADD TRACED TO SRD</b></td><td>DATE: <YY-MM-DD>
PAGE 1 OF <nn></td></tr>
<tr><td colspan="3">PROJECT: <TITLE OF PROJECT></td><td></td></tr>
<tr><td>SRD IDENTIFIER</td><td>ADD IDENTIFIER</td><td colspan="2">SOFTWARE REQUIREMENT</td></tr>
<tr><td></td><td></td><td colspan="2"></td></tr>
</table>

This page is intentionally left blank

APPENDIX E
CASE TOOL SELECTION CRITERIA

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

The tool should:

1. enforce the rules of each method it supports;

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

3. support consistency checking (e.g. balancing);

4. store the model description;

5. be able to store multiple model descriptions;

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

7. minimise line-crossing in a diagram;

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

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

10. support concurrent access by multiple users;

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

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

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

14. support traceability of software requirements to components and code;

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

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

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

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

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

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

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

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

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

21. support scoping and overloading of data item names;

22. support the production of module shells from component specifications;

23. support the insertion and editing of a description of the processing in the module shell;

24. provide context-dependent help facilities;

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

26. have a consistent user interface;

27. permit direct hardcopy output;

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

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

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

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

This page is intentionally left blank.

APPENDIX F
INDEX