

DEVELOPMENT OF TECHNICAL SOFTWARE Case Study Notes

Written by	<u>L.Tuohey, School of Computer Applications Dublin City University</u>
Reference & Version	<u>DCU.LT-G1.001-WPAPER-001v1.1</u>
Date of issue	<u>November 10th 2000</u>

CONTENTS

1. Introduction	6
2. Case Study A: Simulation SW for HIPPARCOS Calibration	7
2.1 Context and User Requirements	7
2.2 Software Development History	11
2.2.1 Specification of software requirements	11
2.2.1.1 Introduction	11
2.2.1.2 Immediate transition to design.....	11
2.2.1.3 User requirements document	11
2.2.1.4 Software requirements document	12
2.2.1.5 Approach adopted for present case.....	15
2.2.2 Design.....	15
2.2.3 Implementation.....	17
2.2.4 Test.....	17
2.2.5 Delivery and exploitation	18
2.3 Analysis : Notable Features, Lessons Learned	19
3. Case Study B: Critical Real-Time SW for ISO AOCS.....	23
3.1 Context and User Requirements	23
3.2 Software Development History	27
3.2.1 Familiarisation and prototyping.....	27
3.2.2 Specification of software requirements	28
3.2.3 Architecture	33
3.2.4 Detailed design	36
3.2.5 Test execution.....	37
3.3 Analysis : Notable Features, Lessons Learned	38
4. Case Study C : Critical Real-time SW for SOHO AOCS.....	40
4.1 Context and User Requirements	40
4.2 Software Development History	40
4.2.1 Familiarisation.....	40
4.2.2 Specification of software requirements	41
4.2.3 Architecture	46
4.2.4 Detailed design and test execution.....	47
4.3 Analysis: Notable Features , Lessons Learned	47
5. Phases of Software Development - Technical Content.....	49
5.1 Introduction	49
5.2 Requirement Definition	51
5.2.1 User Requirements	51
5.2.1.1 General	51
5.2.1.2 ESA SW engineering standards.....	52
5.2.2 Software Requirements.....	53
5.2.2.1 General	53
5.2.2.2 ESA SW engineering standards.....	55
5.3 Design.....	56
5.3.1 Architecture	56
5.3.1.1 General	56
5.3.1.2 ESA SW engineering standards.....	57
5.3.2 Detailed Design	58
5.3.2.1 General	58
5.3.2.2 ESA SW engineering standards.....	59
5.4 Implementation.....	59
5.4.1 Coding	59

5.4.1.1 General	59
5.4.1.2 ESA SW engineering standards.....	59
5.4.2 Test Execution	59
5.4.2.1 General	59
5.4.2.2 ESA SW engineering standards.....	61
6. Phases of Software Development - Review Mechanisms.....	62
6.1 Introduction	62
6.2 Key Formal Reviews	62
6.2.1 Identification and Purpose of each review	62
6.2.2 Review Procedure.....	63
6.3 Intermediate Review Mechanisms.....	64
6.3.1 Identification and Purpose	64
6.3.2 Procedures	64
7. Quality and Configuration Management	66
7.1 Introduction	66
7.2 Software Quality Assurance	66
7.3 Software Configuration Management.....	67
7.3.1 Documentation.....	67
7.3.2 Software.....	68
8. Project Management	69
8.1 Introduction	69
8.2 Project Acquisition	69
8.3 Project Planning and Initiation	69
8.4 On-going Project Management.....	70
8.5 Cost Considerations.....	72
8.6 Schedule Considerations.....	73
References	76

1. Introduction

In this work, a number of case studies in software development are presented. Then, aspects of software development in general are discussed in the light of these cases.

The purpose of this work is not to provide comprehensive coverage of all aspects of software development - there are several software engineering books which have this aim. This work is intended to be complementary to such texts.

The intention is to provide, in a digestible form, information that will be of practical value. The focus of the discussion is on projects with a fairly high engineering and mathematical content but much of the material presented is applicable to software developments of any kind.

It should be borne in mind that the software life-cycle models illustrated and discussed here will require tailoring to meet the specific needs of a given project. Moreover, it is important to be aware that there is much research and new development in the area of software engineering, and to be open to changes of approach resulting from this.

Some attention is given to the context from which software projects arise, including some remarks on the more general topic of systems engineering. Clearly, constraints on software development may often be due to properties of this broader context so that insight on these should lead to a more satisfactory solution for both user and software developer.

Chapters 2, 3 and 4 present case studies A, B and C, respectively. In case A, which deals with development of 'simulation' software, considerable attention is given to the context out of which the software project arose. In cases B and C, which both deal with critical real-time software, the focus is more directly on the software development process. Chapter 5 provides an overview of the various elements of technical work that must be undertaken by the development team in the various phases of a software life-cycle. Chapter 6 identifies the review mechanisms to be applied in the course of a software development. Chapter 7 discusses briefly the topics of software quality assurance and configuration management. Finally, Chapter 8 presents some aspects of the software project management task.

Conclusions and lessons learned are noted within the various chapters. However, it is useful to look ahead to a few major points that emerge:

1. Importance of the software requirements definition phase
2. Need for a clear strategy to handle user requirements that are incomplete or immature
3. Willingness to apply new methods and tools, <u>but</u> in an orderly and planned manner
4. Ensuring the development schedule accommodates time for adequate review

2. Case Study A: Simulation SW for HIPPARCOS Calibration

2.1 Context and User Requirements

This case study is presented as representative of software developed to support or augment scientific or engineering analysis, through simulation of physical phenomena, but not intended to serve as operational software. The term "simulation software" is a convenient if imprecise label for this kind of software; it is imprecise in that, for example, there are many operational or commercial simulation software products.

The software to be discussed was produced as a sub-task in a contract for the definition and assessment of in-orbit payload calibration methods for the European Space Agency's astrometry mission, HIPPARCOS/1/. Specifically, the software was developed as an aid to the definition and analysis of the principal photometric calibration methods, and as the primary means of assessment of these methods.

For present purposes, attention is confined to the task of calibration of photometric sensitivity/1/ in the primary field of view of the HIPPARCOS payload. In fact, the software also simulated calibration of the primary field of view's modulation factors and of the photometric sensitivity of the secondary (star-mapper) field of view. The core of the payload is a grid of alternate clear and opaque slits. The grid is at the focal plane of a telescope so that as stars pass through the telescope's fields of view their images cross the grid and are modulated by it. Behind the grid the photon count rate is detected and measured. The secondary grid is located at the edge of the main grid and has a corresponding secondary detector.

This case is of interest outside the purely software context in that it illustrates the process of formulation and development, starting from high-level system specifications, of user requirements on the software. The progression is shown through A, B and C, below.

Determine the absolute photometric sensitivity within 5% accuracy, its spatial variations over the field of view to within 5% accuracy and its non-linearity to within 1% accuracy. In addition, determine the instrument relative spectral response (accuracy unspecified).

Figure 2.1-1: A - System Level Specification

This prescription is sufficient at system level but, in order to establish a 'determination' procedure, it is necessary to define a mathematical model whose parameters can be directly related to the individual elements of A. Then, a procedure whereby these model parameters are determined to sufficient accuracy will guarantee that the specification is met. Thus, the calibration task consists of establishment of the appropriate mathematical model followed by definition of a procedure for determining the parameters of that model.

The basic calibration model, relating input to measured output, is

$$I_o = I_{pf} \times C_o \quad (2.1-1)$$

where

$$\begin{aligned} I_o &= \text{Stellar photoelectron counting rate(output)} \\ I_{pf} &= \text{Stellar photon flux within assumed instrument bandpass(input)} \\ C_o &= \text{Instrument photometric sensitivity} \end{aligned}$$

In fact, the measured output is

$$I_o' = I_o + I_b \quad (2.1-2)$$

where I_b is the average sky background counting rate.

Then, on noting the dependencies implied by specification A, (2.1-1) yields

$$I_o' = I_{pf} \times C_o(x, y, B-V, I_o') + f(I_b) \quad (2.1-3)$$

where spatial variation is indicated by coordinates (x, y) (origin at centre of square field of view), relative spectral response by $B-V$ (star colour), non-linearity by I_o' and dependence on sky background by $f(I_b)$.

Analysis, including use of the software under discussion, yielded the conclusion that the following form of (2.1-3) was sufficient,

$$I_o' = I_{pf} \times \{ \sum_g \sum_h r_{0gh} \delta_{gh}(x, y) + \sum_m r_{1m} (I_o')^m + \sum_n r_{2n} ((B-V) - 0.5)^n \} + r_{31} \quad (2.1-4)$$

where g and h range from 1 to 3, and m and n from 1 to 2.

Spatial variation is modeled by dividing the field of view into 9 cells, cell (2,2) being the central cell and $\delta_{gh}(x, y) = \text{If}[(x, y) \text{ is in cell } (g, h), 1, 0]^1$. $B-V = 0.5$ for a star of average colour.

In view of (2.1-4) it is possible to replace specification A by

¹ Notation as in *MATHEMATICA*, i.e. $x = \text{If}[\text{condition}, a, b]$ means $x = a$ if condition is true and $x = b$ if condition is false.

- B1** Specify a feasible procedure to
- Determine the absolute photometric sensitivity r_{022} to within 5% accuracy
 - Determine spatial variations r_{011} , r_{012} , r_{013} , r_{021} , r_{023} , r_{031} , r_{032} and r_{033} to within 5% accuracy
 - Determine non-linearity $r_{11}xI_o' + r_{12}xI_o'xI_o'$ to within 1% accuracy
 - Determine star colour parameters r_{21} and r_{22} ; from these, changes in the instrument's spectral response may be monitored/1/.
- B2** Specify the (data collection) time required to achieve the accuracies prescribed above.

Trace from B to A:

- B1** traces to A (assuming sufficiency claim re (2.1-4) is justified)
- B2** is untraced (a necessary operational requirement)

Figure 2.1-2: B - Refinement of System Level Specification

As noted above, part of the calibration task is to define a procedure for determining the parameters introduced in (2.1-4). This includes /1/

- identification of input data
- statement of equations of observation
- definition of method of solution of these equations

Identification of input data: One set of required data consists of measured values of I_o' . In addition, a set of a priori estimates of I_{pf} must be available for specific "calibrating" stars. These are stars for which a priori estimates of spectral photon flux $F(\lambda, B, B-V)$ are available. Then, on taking an assumed payload spectral response $R(\lambda)$, one may calculate

$$I_{pf} = \int R(\lambda)F(\lambda, B, B-V)d\lambda \quad (2.1-5)$$

for a given calibrating star. $(F(\lambda, B, B-V))$ is essentially a look-up table indexed on both star magnitude (B) and colour (B-V)).

Equations of Observation: These are derived from model (2.1-4) as

$$I_{oij}'/I_{pfi} = \sum_g \sum_h r_{0gh} \delta_{gh}(x_{ij}, y_{ij}) + \sum_m r_{1m} (I_{oij}')^m + \sum_n r_{2n} ((B-V)_i - 0.5)^n + r_{31}/I_{pfi} + \epsilon_{ij} \quad (2.1-6)$$

for observation j of calibrating star i. The error ϵ_{ij} is composed of error in I_{oij}' (mainly photon noise) and error in I_{pfi} (mainly due to a priori error in the star catalogue magnitude and colour, and in the star models $(F(\lambda, B, B-V))$ used to derive I_{pf} from B and B-V).

Solution Method: The equations of observation (2.1-6) are processed through a weighted least-squares method, in which the weights are inversely proportional to the estimated error variances.

On identifying the calibration designer to be the software user, the following high-level user requirements on the simulation software emerge:

- U1** Simulate the scanning motion of the satellite over the sky such that a representative time-tagged stream of stellar positions on the instrument's field of view is generated. The stream should be generated for a prescribed data collection period (typically a minimum of 3 to a maximum of 24 hours) and the stars may be assumed to be uniformly distributed in the sky.
- U2** The set of calibrating stars derived through requirement U1 should be appropriately distributed in magnitude and colour.
- U3** The measurement process should be modeled, that is, I_{oij} and I_{pfi} should be determined taking account of observational error ϵ_{ij} (see (2.1-6)).
This entails
 - (a) Generation of an a priori I_{pfi} based on a simulated pre-launch $R(\lambda)$ and a priori B_i and $(B-V)_i$
 - (b) Generation of a 'true' I_{pfi} based on a simulated post-launch ('true') $R(\lambda)$ and 'true' B_i and $(B-V)_i$. Then, multiplication of the 'true' I_{pfi} by a 'true' sensitivity (not the same form as in (2.1-4)) to give a 'true' I_{oij} . Finally, addition of noise (particularly photon noise - proportional to duration of the observation) to derive the observed I_{oij} from the 'true' value.
- U4** The photometric sensitivity should be modeled to support the form specified in (2.1-4) but the implementation should be such that alternative models can easily be incorporated.
- U5** The equations of observation (2.1-6) should be solved according to the specified method (weighted least squares).
- U6** Specification of input to the software should be "user-friendly" and should, so far as possible, not require knowledge of implementation details.
- U7** Results should be clearly displayed. Post-processing should be built in so that the task of analysing results is facilitated as much as possible.

Trace (approximate) from C to B:

- U1** traces to B2
- U2** traces to B2
- U3** "calculation of I_{pf} " part traces to B1
"remainder" part traces to B2
- U4** "first part" traces to B1
"second part" traces to B2
- U5** traces to B1
- U6** untraced (but significant in software terms)
- U7** untraced (but significant in software terms)

Figure 2.1-3: C - User requirements on the software

Note: To check that no higher level requirement has been forgotten i.e. that U1 to U7 are complete one can simply sort the above 'tracing' matrix on field beginning 'B'.

2.2 Software Development History

2.2.1 Specification of software requirements

2.2.1.1 Introduction

The user requirements U1...U7 are a clear high level statement of the functions to be provided by the software from the user's point of view. However, for the software developer they are far from sufficient and need to be specified in much greater detail. The technical details are already available - their establishment is the essence of the calibration designer's task - but are distributed through a variety of documents, rough working notes, memoranda, technical notes, meeting minutes and actions, faxes, reports, published papers and so on.

A number of possible ways of specifying the detailed software requirements in a form suitable for subsequent design suggest themselves, and are discussed in the following sections.

2.2.1.2 Immediate transition to design

This approach assumes that further specification is unnecessary and that software design can proceed immediately.

(a) The approach is possible when each team member contributes to the high-level software design and is familiar with all aspects of the project including the variety of detailed documentation mentioned above.

(b) Alternatively, if the team structure is such that there is a chief designer responsible for the overall software design then, provided that this individual is familiar with all the detailed documentation, the high-level design can proceed immediately. The chief designer can then make each subsystem or component designer aware of the particular documentation that specifies the detailed requirements for that subsystem or component.

2.2.1.3 User requirements document

In this approach, a single document called the User Requirements Document (URD) is prepared. The purpose of this document is to collect together all the detailed user requirements.

(a) In its simplest form the URD would consist of a statement of U1 to U7 with added references to sources of detailed information. For example,

U2 The set of calibrating stars derived through requirement U1 should be appropriately distributed in magnitude and colour colour (see Reference A).

where Reference A is a fax, say, which specifies the number of calibrating stars in each magnitude and colour range.

(b) A simple but substantial enhancement of (a) would be to attach to the URD, as an appendix, copies of all detailed documentation.

(c) The documentation specifying detailed user requirements on the software comes from a variety of sources and has varying degrees of maturity. Therefore, it is inevitable that there will be different notations used and that inconsistencies will exist. Accordingly, a more ambitious form of URD may be envisaged in which all the distributed detailed information is restated in a uniform and consistent manner.

2.2.1.4 Software requirements document

In this approach, a document called the Software Requirements Document (SRD) is prepared. The purpose of this document is to state precisely all the requirements to be satisfied by the software, both those specified explicitly by the user and those arising from other considerations.

The SRD, unlike the URD, is written from the software developer's point of view (though not necessarily that of the software designer). For the case under discussion, there are two possibilities,

(a) Prepare the SRD directly on the basis of the distributed documentation containing the detailed user requirements.

(b) First prepare a URD as indicated in section 2.2.1.3 and from this basis prepare an SRD.

It is important to clarify how an SRD may differ from a URD, particularly a detailed URD prepared in accordance with approach (c) of section 2.2.1.3. The following examples illustrate typical differences.

Example 1: Both URD and SRD might contain the requirement

- "R1.1 The set of calibrating stars shall be distributed in magnitude and colour according to the proportions given in the following table:

	Colour class 1	...	Colour class N
Magnitude class 1	n_{11}	...	n_{1N}
		...	
Magnitude class M	n_{M1}	...	n_{MN}

where n_{st} is the number of stars in magnitude class s and colour class t from the total population of calibrating stars."

In addition, an SRD might contain

- "R1.2 The distribution in magnitude and colour shall be performed by assigning each magnitude class and colour class pair a sub-interval of the interval (0,1), such that the sub-interval length is proportional to the value of the corresponding entry of the table of R1.1. Then, a uniformly distributed random number shall be generated in the interval (0,1) and magnitude and colour classes shall be assigned according to the sub-interval the random number falls in."

Finally, an SRD might also contain

- "R1.3 The (pseudo-) random number generator used for R1.2 shall be a standard one provided by the computer system on which the software is developed."

It will be apparent from this example that the cutoff between URD and SRD is not always clear-cut and may be quite subjective.

Example 2: Refinement of Requirement U6

The statement of this requirement in section 2.1 is

U6 Specification of input to the software should be "user-friendly" and should, so far as possible, not require knowledge of implementation details.

This places the onus on the software developer to detail the requirements on input mechanisms and format - a quite substantial task.

The resulting SRD requirements might include, for example,

"R2.1 Input to the software shall be specified in two successive steps:

- (a) Off-line preparation of separate data files for each of
- R(λ) (separate files for 'pre- and post-launch')
 - F(λ , B, B-V)
 - Calibrating star magnitude and colour distributions (see **R1.1** of Example 1)
- (b) Off-line or interactive preparation of a control data file whose contents are as follows :
- Sequence control parameters
 - Duration of calibration run (hours)
 - Name of file containing R(λ) (pre-launch)
 - Name of file containing R(λ) (post-launch)
 - ...

If "interactive" then the software should prompt the user for each required input, offering defaults where possible."

"R2.2 An input file for R(λ) shall consist of N+2 lines of data, formatted as follows :

```
Header
Data line 0
Data line 1
.
.
Data line N
```

where

Header specifies a title for R(λ) (up to 80 characters)

Data line 0 specifies the total number N of (abscissa, ordinate) pairs

Data line 1 specifies pair 1, (λ_1 , R(λ_1))

.

Data line N specifies pair N, (λ_N , R(λ_N))

In addition it shall be possible for the user to add descriptive comments to the file either as complete lines or in-line. These comments shall be enclosed between the symbols "/*" and "*/".

There would be similar requirements on the other input files. The detailed definition of the requirements for the control data file content could well be particularly lengthy.

Example 3: Refinement of Requirement U1

This requirement as stated in section 2.1,

- U1** Simulate the scanning motion of the satellite over the sky such that a representative time-tagged stream of stellar positions on the instrument's field of view is generated. The stream should be generated for a prescribed data collection period (typically a minimum of 3 to a maximum of 24 hours) and the stars may be assumed to be uniformly distributed in the sky.

is a quite complex one. However, in the event it was possible to take advantage of work already performed elsewhere on the HIPPARCOS project by a different contractor. Thus, the SRD requirement could read

"**R3.1** A representative time-tagged stream of stellar positions on the instrument's field of view shall be generated by means of software X produced by developer Y. Software X shall be used on the same basis as a commercial 'off the shelf' product; in particular, it shall require no special testing."

Example 4: Other miscellaneous examples of SRD requirements that might not appear in a URD are

"**R4.1** The software shall be developed on a VAX computer model xxx using FORTRAN 77 supplemented where necessary by VAX VMS control language."

"**R4.2** The software shall be coded in accordance with a defined standard, covering subprogram header layout, policy on comments, naming conventions, any restrictions on use of language constructs, ..."

"**R4.3** The software shall be delivered in the following format and media ..."

"**R4.4** The documentation provided with the software shall be (a) User Manual, (b) Programmer Guide and (c) Installation Instructions.

The content of the User Manual shall be ...

The content of the Programmer Guide shall be ...

The content of the Installation Instructions shall be ..."

2.2.1.5 Approach adopted for present case

The approach adopted in the present case was (b) of section 2.2.1.2 (Immediate transition to design). This will be discussed further in section 2.3 (below).

2.2.2 Design

A number of clearly defined functions emerge from the requirements:

1. Acquire, through simulation of the satellite's scanning motion, a complete set of time tagged positional data (x_{ij}, y_{ij}, t_{ij}) (observation j of star i).
2. Assign magnitude and colour $(B_i, (B-V)_i)$ to each calibrating star.
3. Compute 'true' photon flux for star i .
4. Compute a priori photon flux for star i together with estimated error variance.

5. Model observed count rate for observation j of star i ; also, calculate estimated error variance.
6. Set up equations of observation and solve them for estimated sensitivity parameters (r_{0gh} , r_{1m} , r_{2n} , r_{31}) together with associated error covariance matrix.

There are a number of sequencing requirements in performance of these functions. For example, 1 defines the total number of stars and so is a pre-requisite for 2. Function 2 is a pre-requisite for both 3 and 4 while 3 is pre-requisite for 5. Both 4 and 5 are pre-requisites for 6.

On the other hand, it is possible to modify later functions without invalidating earlier steps. For example, the model of estimated sensitivity in the observation equations of function 6 may be replaced without affecting the results of previously performed functions.

It is important for the user to examine results of intermediate functions, for check and test purposes but also for their intrinsic interest. For example, the dependence of I_{pf} on B and $B-V$ is by no means a simple one so that the results of both functions 3 and 4 are of considerable interest to the user.

In the light of the above considerations, it is seen to be desirable to provide the user with flexibility in control of execution of the functions subject to the underlying sequencing requirements.

Accordingly, the software design incorporated distinct modules corresponding to each of functions 1 to 6, say $M1$ to $M6$, respectively. In addition, a control module ($M0$) was needed to provide a flexible and friendly user interface, and to verify and implement user-specified function execution sequences.

One may relate these design components to the high-level user requirements of section 2.1 through the following tracing matrix:

Design Component	traces to	User requirement
M0		U6
M0		implicit sequencing requirements
M1		U1
M2		U2
M3		U3(b) (I_{pf} part)
M4		U3(a)
M5		U3(b) (I_{oij} part)
M6		U4
M6		U5
M6		U7

Figure 2.2.2-1: Trace from design components to user requirements

Note: (a) Had separate software requirements, as illustrated in section 2.2.1.4, been prepared for the project then the design components should be traced to the software requirements which would be traced, in turn, to the user requirements.

(b) By sorting on the "user requirement" column one can check that no requirement has been forgotten - a completeness check.

In order to allow the user the possibility of modifying source code of individual functions - this was especially important for modules M5 and M6 - without engaging in large re-compilation or linking exercises, each of functions 1 to 6 was designed to be implemented as a separate FORTRAN program. The control module (M0) was designed to be implemented partly in VAX VMS control language - the top-level procedure, mainly for function sequence verification and execution - and partly in a FORTRAN program - mainly for receipt and interpretation of individual items of a control data file.

The interface between the programs took the form of data files; two versions of each such file were produced, a packed, machine-readable form for efficient inter-program communication and a human-readable form for examination by the user. An essential pre-requisite before implementation could proceed was, of course, that this interface between modules be specified; that is, that the content of the data files be defined in detail. This task was made easier by ensuring so far as possible that the data files corresponded very directly to entities in the 'user domain'.

Following specification of interfaces between the high level modules, the next step was to detail the internal design of these modules. The process was continued to the level at which coding could proceed.

2.2.3 Implementation

The software design sketched in the previous section enabled the software to be implemented in a de-coupled manner by different programmers and was such that modifications in user requirements could be accommodated in a flexible, localised way. It also implied that each module could be tested separately, without the necessity of a substantial integration test phase, and that problematic or error-prone modules could be easily isolated for repair.

2.2.4 Test

Unit testing was not formally controlled and documented and, as has been noted in section 2.2.3, there was little need of special integration tests to check the inter-module interface. However, as a minimum, it was necessary to test that each file making up this interface was read from and/or written to correctly; in particular, that each file created had the content specified by the design.

Thus, attention was focussed on testing at the requirement level, that is on verification that the requirements on the software were in fact satisfied.

A (relatively small) number of high-level tests based on the requirements were formulated. These were usually cases in which it was possible to define analytically the correct outcome. For example, in function 4, if the uncertainties on both B and B-V are specified to be very small then the estimated uncertainty on a priori photon flux I_{pf} should also be very small. Again, in setting up the model for I_{oij} in function 5, it is possible to arrange that the 'true' sensitivity has a very simple form; then the estimated sensitivity should have a correspondingly simple form. Similarly, one can isolate estimation of colour terms (r_{2n}) and note whether expected behaviour is exhibited when a priori and 'true' $R(\lambda)$ are mutually shifted to the blue or red ends of the spectrum /1/. As a final example of this kind of test, it is possible to derive analytical results for estimated parameters and their error covariance matrix in the simplified case that all calibrating stars have the same magnitude and colour, and their observations are all of the same duration; then, the analytic and computed results may be compared.

It is best, when implementing verification tests of the above kind, to set up the test data in the manner specified by the Software User Manual. This provides a check on both software and manual.

Verification tests should, where possible, be developed in a systematic way. This activity can commence from the formulation of requirements, provided that the formulation is testable. Design of tests and specification of actual test data should proceed in parallel to the software development. With reference to the high level requirements U1 to U7 of section 2.1 it can be seen that U6 does not meet, except in a very general way, the criterion of testability - and could be a source of disagreement between user and software developer - whereas its suggested refinements (R2.1, R2.2) in Example 2 of section 2.2.1.3 are far more satisfactory from this point of view. On the other hand U2 is quite testable and one can immediately specify the test,

"Examine a number of simulated sets of calibrating stars to verify that the distribution in each case is sufficiently close to the given distribution in magnitude and colour. Cases of the smallest possible set (say 3 hours) and largest possible set (say 24 hours) should be included."

It will be seen that more precise details can be added as the development proceeds.

2.2.5 Delivery and exploitation

The first software delivery of this case study was to the primary user i.e. the calibration designers. However, as the latter also contributed to the software development and were part of the same team this first delivery was not a formal one. Subsequently, the software was delivered in an agreed form and medium to the higher level contractor, together with its documentation (User Manual, Programmer Guide, Installation Instructions). This second delivery was a formal one but as the software was not intended for operational use in processing actual satellite data the degree of formality was not great.

The calibration method designers performed the main "exploitation" of the software for two purposes.

(a) Refining the calibration methods (e.g. the appropriate form for the estimated photometric sensitivity; the benefits and disadvantages of extending the set of calibrating stars to include stars for which a priori knowledge of magnitude and colour is much poorer; etc).

(b) Assessment of whether the specified system level performances (see A of section 2.1) were achievable and, if so, prediction of how long i.e. of what volume of observed data would be required to achieve them.

2.3 Analysis : Notable Features, Lessons Learned

This case study illustrates notable aspects, both of the context in which a technical software development takes place and of the software development process itself. The software of the case study was developed as a sub-task within a wider project, of which the primary product was a definition of in-orbit calibration methods together with a demonstration of their feasibility. Therefore, the main project effort was directed to achieving delivery of this product and the project team members were chosen with this in mind. However, the software sub-task discussed here required a significant proportion of the total project resources. Therefore, it was necessary to ensure that it did not impinge adversely, through overrunning its allocated resources, on performance of other sub-tasks and therefore on the project as a whole.

Formulation of the calibration methods was largely an exercise in mathematical modeling so that experience of this, particularly of the parameter estimation field, was extremely helpful. Similarly, the scheme embodied in the software for assessment of the calibration method applies general principles and techniques which are well established for problems of this kind /2/. Thus, the importance for both the requirement specifier and software developer of identifying similar or analogous situations and of seeking to classify a given problem within a more general setting is stressed.

A good deal of attention was given, in section 2.1, to the process that ultimately results in a definition of user requirements on the software. The formulation of such requirements, and of the higher level requirements from which they are derived, is of continuing interest in software engineering research and practice, and will be considered again in section 5.2.1. This is a matter of great importance for software development as the manner in which user requirements are stated can have considerable software impact. Moreover, cost and schedule constraints on software development are often dictated by wider constraints either on the immediate software user or on higher levels. In summary, an appreciation of the wider context is an important element in forming a plan for software development that is satisfactory for both developer and user.

The use of tracing matrices in making correspondence from a lower (or later) level to the previous higher (or earlier) level was illustrated and its importance in ensuring that no higher level item is forgotten was noted. This is a straightforward but powerful

technique; clearly, it requires that the elements of each level, individual requirement or design component, for example, be assigned a unique label (U1, R1.2, M0, etc).

It is possible to display the successive levels of specification through to coding, together with corresponding test and verification levels, in the form of a "V" (for verification) diagram. A diagram for the case under study is,

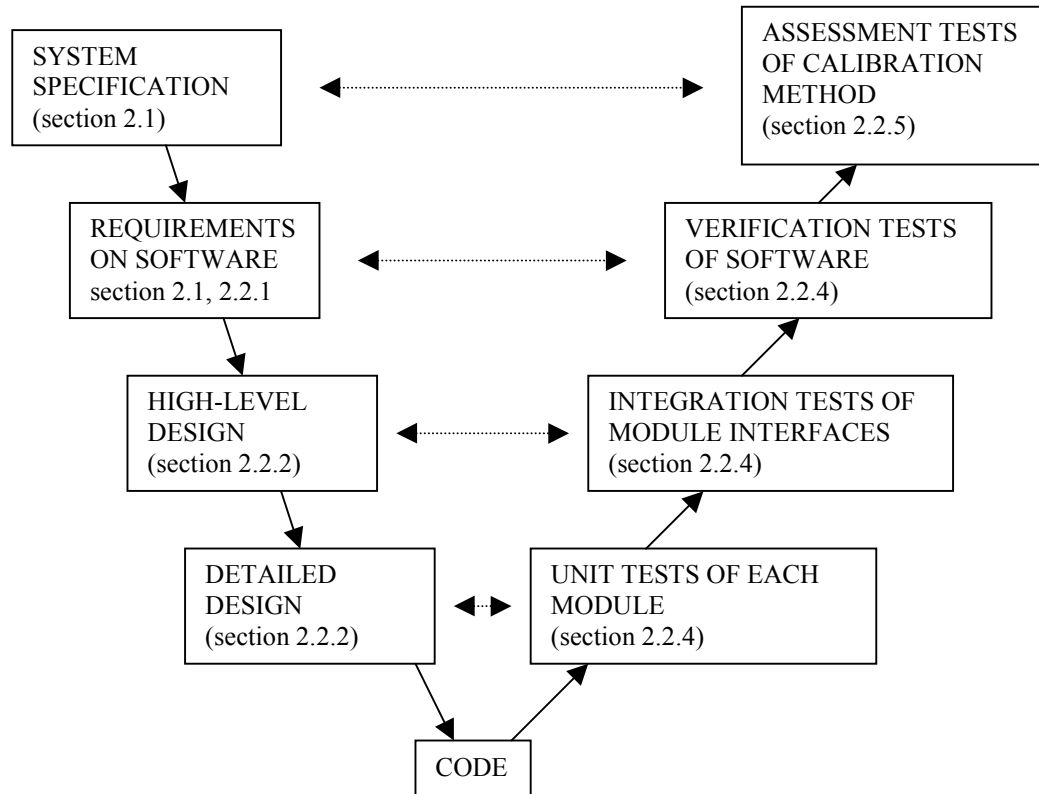


Figure 2.3-1: "V" diagram for Case Study A

Note: (a) This diagram refers to the system specification for photometric sensitivity only. Naturally, in HIPPARCOS, there were a great many other system specifications and for each of them a verification method (not necessarily in software) had to be defined.

(b) This diagram combines, under the heading "REQUIREMENTS ON SOFTWARE", both user and software requirements as no distinction was made for this case.

In section 2.2.1.5 it was noted that the software was proceeded with on the basis of a relatively informal high level statement of requirements, supported by miscellaneous detailed documentation, without first preparing a URD and/or SRD. This was considered to be feasible and cost effective for a number of reasons including the project's relatively limited size (total effort of roughly 2 man years), the possibility and intention from the outset of designing the software in modules which handled well-defined, de-coupled subsets of the user requirements, and the presence in the

project team of personnel experienced in calibration, in formulation of relevant mathematical models and in implementation of similar software.

However, there are obvious disadvantages in this approach. For example, the absence of a clear, detailed definition of the user interface gave rise to much effort being devoted to this aspect of the software, perhaps a disproportionate effort in terms of the project as a whole. Again, the absence of a URD and/or SRD precluded specification of a complete set of verification tests, a process that would certainly have assisted in refining and focussing the requirements on the software. However, it must be borne in mind that preparation of a URD or an SRD or a complete specification of verification tests requires resources so that there was a trade-off between what would be the ideal and what was sufficient for the project, given overall constraints on schedule and personnel.

The software testing, particularly at unit level, was performed in an informal way, the responsibility for module correctness being placed on the implementor of the unit (= module). In general, there is a risk that if the confidence placed on an implementor is not justified then low level errors may not be uncovered at an appropriate level (refer to foregoing "V" diagram, Figure 2.3-1), where they are relatively easy to detect and repair. Instead, they may cause failure at a higher level so that diagnosis is much more difficult. For example, suppose that an error remains undetected in module M2 and later results in a failure in integration of M2 with, say, M3. Then, diagnosis of the integration test failure must admit the possibility that both M2 and M3 are suspect. An even more serious situation would obtain if the error persisted to the level of "CALIBRATION METHOD ASSESSMENT" (refer to "V" diagram, Figure 2.3-1) for then the failure could be suspected to be due to any element of the software, defects in the mathematical formulation of the calibration method or of its assessment procedure, or a (incorrect) conclusion that the system specification was not achievable (which might have serious impact on the HIPPARCOS mission as a whole).

Execution of software tests is not the only mechanism of checking correctness of a piece of software. A second and perhaps more important mechanism is that of review, whereby the work is checked and corrected as necessary at regular intervals. With reference to the "V" diagram above it is clearly desirable to schedule review activities at the completion of each phase (e.g. SYSTEM SPECIFICATION, REQUIREMENTS ON SOFTWARE) or part of a phase (e.g. detailed design of a module), prior to commencement of the next phase. For the case under study, review activities took the form of frequent working meetings within the project team, and regular formal meetings with the higher level contractor (i.e. the customer). The process of software review, as will emerge in chapter 6, may be quite formal and, for critical software developments, must be allocated a significant proportion of the total available resources in terms of both schedule and personnel.

The main output from the calibration project as a whole was a "Calibration Report", which presented the recommended calibration methods, together with details of their assessment. In addition, a Software Requirements Document for the operational calibration software, that is the software later developed to process the actual satellite

data, was produced. Thus, the software of this case study fulfilled the additional role of aiding formulation of the requirements on the corresponding software, and served as a prototype for that software.

The HIPPARCOS satellite was launched in 1989 and has since successfully completed its mission (despite post-launch problems due to failure of an apogee boost motor). The quality of the data obtained has proved to be very high. The calibration methods, including the subject of this case study, have been successfully applied during the satellite's commissioning period, at the beginning of its operational life.

3. Case Study B: Critical Real-Time SW for ISO AOCS

3.1 Context and User Requirements

This case study is presented as the first of two representatives of on-board real-time control software for satellites. In both cases, the software discussed is that which is resident in the controlling processor of the satellite's attitude and orbit control subsystem (AOCS). The role of the AOCS is to maintain the desired attitude (axial pointing) and orbit for the satellite. Hence, the software is highly critical in that any software failure will directly impact on the total mission performance, and if the software fails irrecoverably then the complete mission will be lost.

The degree of criticality of this software is far higher than that of case study A and therefore requires a far greater degree of formality and control in its development. A second important difference is that URD and SRD preparations are carried out by separate contractors so that, in contrast to A, the present case is focussed purely on the software development.

This case study is concerned specifically with the AOCS of the European Space Agency's Infrared Space Observatory (ISO) satellite /3². The specifications for ISO's AOCS, and consequently the corresponding requirements on software, are highly demanding. In particular, a high degree of autonomy (from ground control) is required of the subsystem, entailing substantial self-checking and reconfiguration functions.

The AOCS is one of several subsystems of the satellite but can essentially be considered in isolation for this case study. The AOCS consists of a number of sensors, two sets of actuators, a control unit (ACU) made up of a processor and its resident software, and an interface with the on-board data handling subsystem (OBDH). The OBDH's function is to centrally manage all communications with ground so, from the AOCS view-point, it is the source of telecommands (TCs) from ground and the destination of telemetry (TM) to ground. In addition, the OBDH provides independent external timing signals to the AOCS.

The basic operational principle of the AOCS is that actual positional data are read from the sensors and are compared with desired positional data specified via telecommand by ground; then, positional corrections based on this comparison are calculated and commands to give them effect are transmitted to the actuators. Positional data consist of both information on actual position (of sun, earth and stars depending on sensor) and rate of change of position. The derivation of positional corrections is dictated by control laws.

The operational mode of the AOCS is a convenient shorthand for describing the state of the subsystem at a given time. Stating that the AOCS is in a given mode is equivalent to stating that data from specific sensors are being used for control, that a

²For present purposes, various simplifications and modifications to actual ISO specifications have been made.

specific control law is being used and that a specific actuator set is being used. In all, there are of the order of 7 operational modes. The chief differences between them include the positional accuracies achieved and the degree of autonomy from ground. A transition from the current mode to a different one may be made autonomously or on ground command; not all transitions are permitted.

Apart from the basic ones described above, there are several other functions to be performed by the AOCS software. Self-checking functions, generally based on consistency of data with previous readings, are performed on receipt of sensor data. If inconsistencies are detected then a suspect unit must be switched out and an equivalent redundant one switched in - the AOCS specification requires that there be no single point failures so that this redundancy has to be provided. Functions must be provided for checking the criteria for mode transitions and for effecting such transitions when required. Functions must also be provided for receiving, verifying, interpreting and executing all telecommands, and for gathering and transmitting all required telemetry.

All functions provided by the AOCS software are executed in a continuous, repetitive manner. The execution is regulated by a clock in that each clock pulse signals the start of a time slot and each time slot is associated with the execution of specific functions. The time slot length and the control laws are designed to be compatible. For example, for a given mode, one might have

Slot 1+4n	Read and self-check sensors and reconfigure if necessary. Perform control law calculations to derive positional corrections
Slot 2+4n	Command actuators. Gather telemetry
Slot 3+4n	Process telecommands
Slot 4+4n	Check for and, where necessary, effect mode transitions

where $n=0,1,2,\dots$

Apart from the functional requirements discussed above the AOCS software must satisfy other types of requirements. For example, it will be clear from the preceding paragraph that there is a limited amount of time available for performance of functions so that the software has to satisfy timing constraints. Similarly, there are constraints on the available memory. It is essential that the software be re-programmable by ground so that, for example, amendments can be made should operating conditions be not as anticipated. A further attribute required of the software is that it have a three layer structure such that layer 1 interfaces to hardware (including interrupt handling), layer 3 implements high level application functions while layer 2 provides the interface between the other two layers.

The split between layer 1 and the other two layers is particularly important in this case study because one industrial contractor had responsibility for development of layer 1 (referred to as the Operating System (OS)) and a second contractor had responsibility for layers 2 and 3 (together referred to as the Application Software (ASW)). The application software is the main focus of attention in this study. The OS developer also had responsibility for provision of the control processor (hardware). This

contractual arrangement made it imperative that the interface between OS and ASW was clearly defined and rigorously controlled, and that the implementation was such that the two parts were capable of being combined, to form the complete software, in an easy and reliable manner.

The overall industrial structure of the ISO project is that of a prime contractor, with responsibility for the complete system, supported by lower level contractors with responsibility for individual subsystems. In turn, each subsystem contractor has subcontractors with responsibility for individual elements of the subsystem. For the AOCS subsystem there are the software contractors mentioned above but also subcontractors responsible for provision of hardware items, including individual sensor and actuator units. It is the responsibility of the AOCS contractor to coordinate the work of all subcontractors, including specification of requirements (with delivery dates) and planning of test and integration activities. For software, the AOCS contractor had responsibility for preparation of the User Requirements Document (URD) covering both OS and ASW. Note that the URD included the definition of any mathematical modeling required, including particularly the control law definitions.

The following "V" diagram gives an overview of the AOCS software development context,

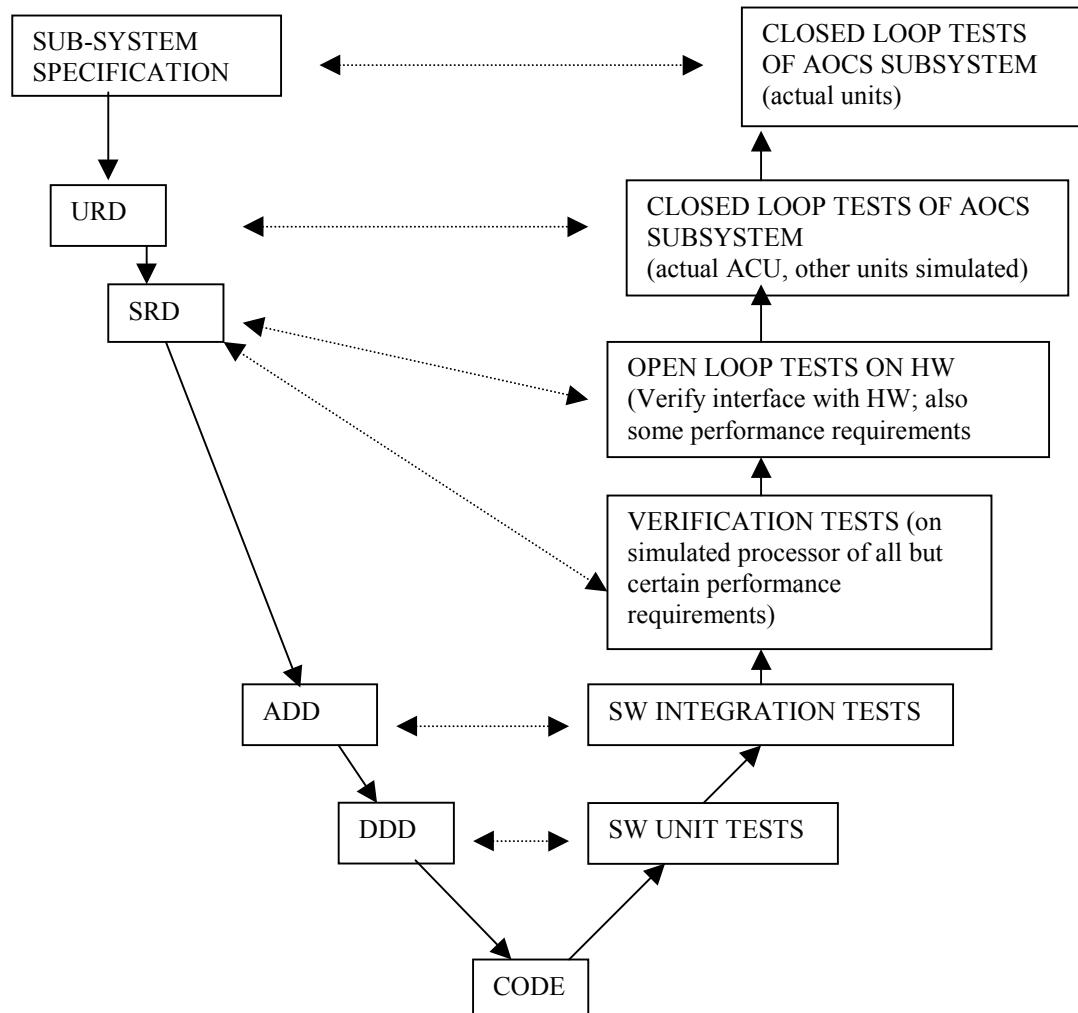


Figure 3.1-1: "V" diagram for Case Study B

where HW refers to the control processor, "closed loop" means that the dynamics of the satellite are simulated, ADD stands for Architectural Design Document and DDD stands for Detailed Design Document.

The diagram presents a simplified picture in that it does not reflect that the software was developed in two separate contracts, giving rise to separate SRDs, ADDs and so on, and with deliveries at different times. Moreover, it does not reflect that the ASW was delivered in a phased manner. Similarly, other units were delivered in a phased way and were integrated into the subsystem as they became available. In addition, the diagram shows "classic" test and verification phases for the software - unit to integration to verification - but it will be seen that this view was somewhat modified in practice.

Both OS and ASW parts of the software were developed in software environments within which the target processor was simulated. Then, the OS was integrated with the actual processor as produced by the same (OS) contractor. Later the ASW was

delivered, in phases, and integrated with the OS and actual processor hardware. Subsequently, the ACU as a whole was tested and integrated with other units at the AOCS contractor's premises. It is clear that, to be effective, these activities called for close cooperation between personnel from different contractors so that it was essential that project management provided an organisational framework to facilitate this.

It is characteristic of satellite projects (and no doubt of projects in all fields) that specifications are not mature at project start, nor even at the point in the project at which detailed design and production must begin. Consequently, one is faced with the problem of how to proceed with development on the basis of changing specifications. Inevitably, this issue arose for the software (ASW) under discussion so that the case study gives some light on resulting difficulties and on approaches to handling them.

The URD provided as input to the software developers specified separately the requirements for OS and ASW. For the ASW, the document was divided according to broad functional headings such as self-checking functions, reconfiguration functions and control laws. Under each heading very detailed individually numbered requirements were specified (as they became known), expressed in pseudo-code as necessary and sometimes supported by Nassi-Schneiderman diagrams. The document appeared in successive issues, thus enabling details to be added and amendments to be made in a controlled manner.

3.2 Software Development History

3.2.1 Familiarisation and prototyping

Prior to the development proper, before formal issue of user requirements, there was a preliminary phase for familiarisation, and for identification and resolution of key technical issues. In particular, the choices of processor and of development language were made. The desirability of using a high level language and the existence of a compiler, with supporting software development environment, for the chosen processor suggested that Ada be selected. However, in view of Ada's novelty (at the time), a feasibility assessment in the form of a prototype development was decided upon.

Realistic user requirements for a particular AOCS mode were specified so that the prototype results would be representative of the actual operational software. Attention was focussed on the Ada system's capacity to implement key requirements, particularly the performance requirements on memory and timing. Prototype OS and ASW software were developed in parallel, an interface in the form of Ada package specifications having been agreed beforehand.

The results of the prototyping exercise confirmed that Ada (without tasking) was a feasible choice. The ease with which the OS and ASW parts of the prototype were combined was particularly satisfactory, as was the speed with which Ada code could be written.

Reference /4/ gives an account of the project's use of Ada, from the user's (AOCS contractor's) point of view.

3.2.2 Specification of software requirements

As indicated in section 3.1, user requirements were grouped under broad functional headings within which individually numbered requirements were specified in detail. This arrangement facilitated the user in specifying certain topics early (control laws, for example) while postponing others until later (reconfiguration functions, for example). It also provided flexibility in incorporating contributions from individuals, expert in different aspects of the system. The detailed level of user requirements implied that it was unnecessary for the software developer to expand them, initially at least; instead the software developer could concentrate on analysis of the requirements, checking particularly on their consistency and completeness. This process of analysis resulted in a software requirements document (SRD) in which the user requirements, complemented by requirements arising from implementation constraints, were restated in a form suitable for input to the software design process.

The URD organisation of requirements had, *from the software developer's perspective*, the disadvantage of being somewhat dispersed and disconnected. For example, steps in processing data from a particular sensor might appear separately under sections on self-checking functions, control law calculations, telemetry gathering and mode transition criteria. Nevertheless, it was the software developer's responsibility to ensure that these various processing activities were implemented in a consistent manner and in proper sequence. Hence, it was very important that these aspects be reflected clearly and explicitly in the SRD. With this objective in mind it was decided to apply the Data Flow Method /5/, extended to include control flow /6/, in analysing the URD and in structuring the SRD. The following paragraphs illustrate some of the key features of the method.

The basic elements of the Data Flow method are data flows, and processes acting on these flows. These elements are represented in diagrams by arrows and bubbles, respectively. In addition, there are data stores (enclosed within parallel horizontal lines), and special processes (rectangles) external to the system which act as data flow sources or sinks. Control flows and processes are represented similarly except that dashed rather than continuous lines and curves are used.

The method proceeds by first establishing a top-level context diagram of the system showing its interaction, in terms of data flows, with external sources and sinks. For example, for the software under study, one might have

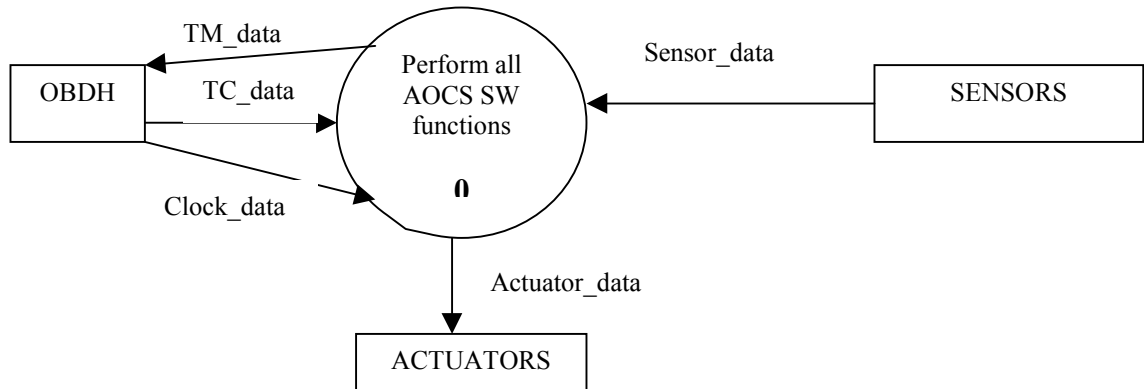


Figure 3.2.2-1: Possible data flow context diagram (case study B)

The next step is to decompose the top level process (0). For example,

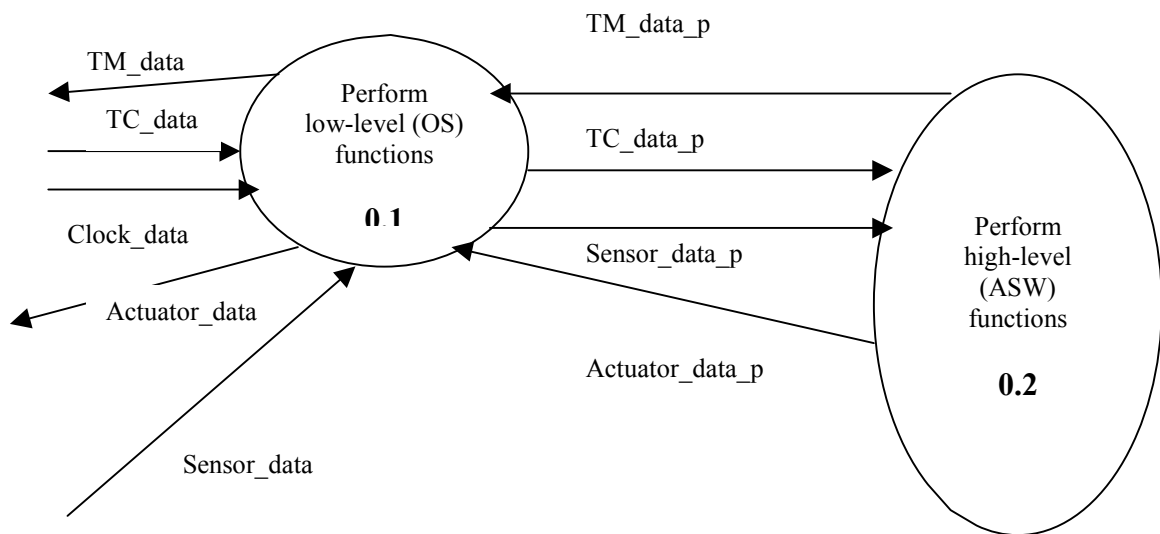


Figure 3.2.2-2: Possible data flow Level 1 diagram (case study B)

Note: '_p' stands for 'processed'.

Continuing in the same manner, and concentrating on the ASW (process 0.2), one might obtain

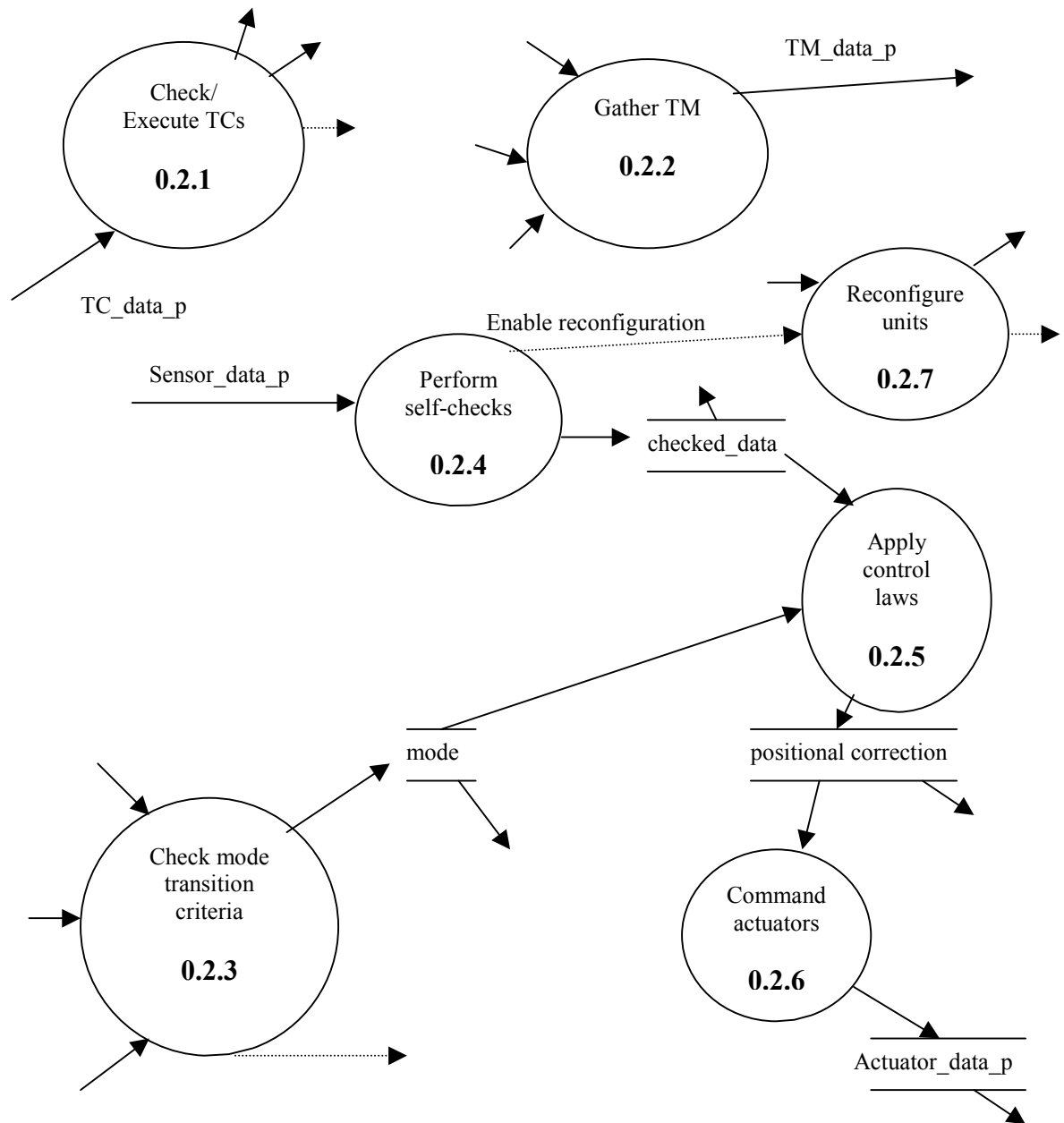


Figure 3.2.2-3: Possible data flow Level 2 diagram (case study B)

No attempt at completeness has been made in this diagram. However, it is clear that there will be more interaction between processes than shown. For example, a telecommand to cause a mode transition would give rise to a control or data flow from 0.2.1 to 0.2.3. Also, it is likely that there would be data flows from all other processes to 0.2.2. The output from process 0.2.3 should certainly condition other processes besides 0.2.5.

The procedure is continued to the next level in a similar manner. For example, one might have

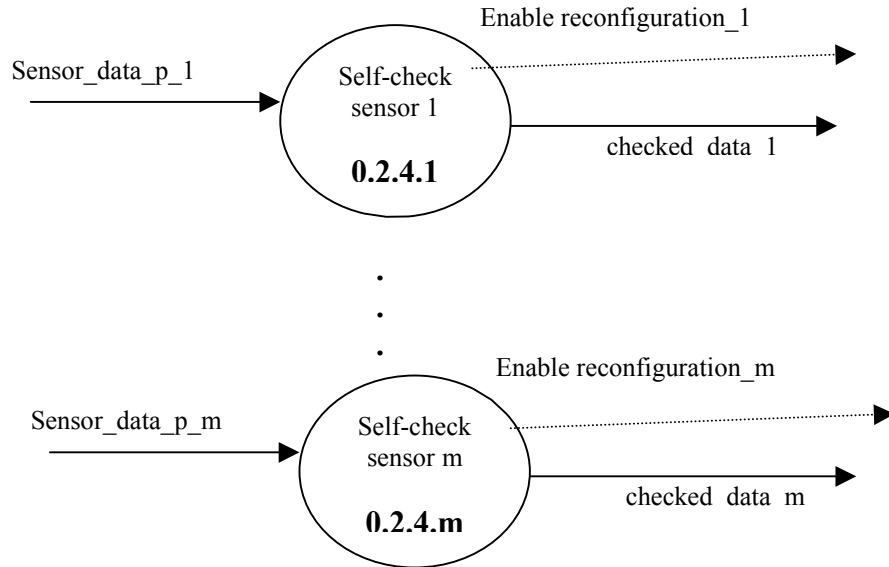


Figure 3.2.2-4: Possible data flow Level 3 diagram (case study B)

where, in this case, data flows as well as processes have been decomposed, that is

$$\begin{aligned} \text{Sensor_data_p} &= \{\text{Sensor_data_p_1} | \dots | \text{Sensor_data_p_m}\} \\ \text{Enable_reconfiguration} &= \{\text{Enable_reconfiguration_1} | \dots | \text{Enable_reconfiguration_m}\} \\ \text{checked_data} &= \{\text{checked_data_1} | \dots | \text{checked_data_m}\} \end{aligned}$$

Similarly, one might have,

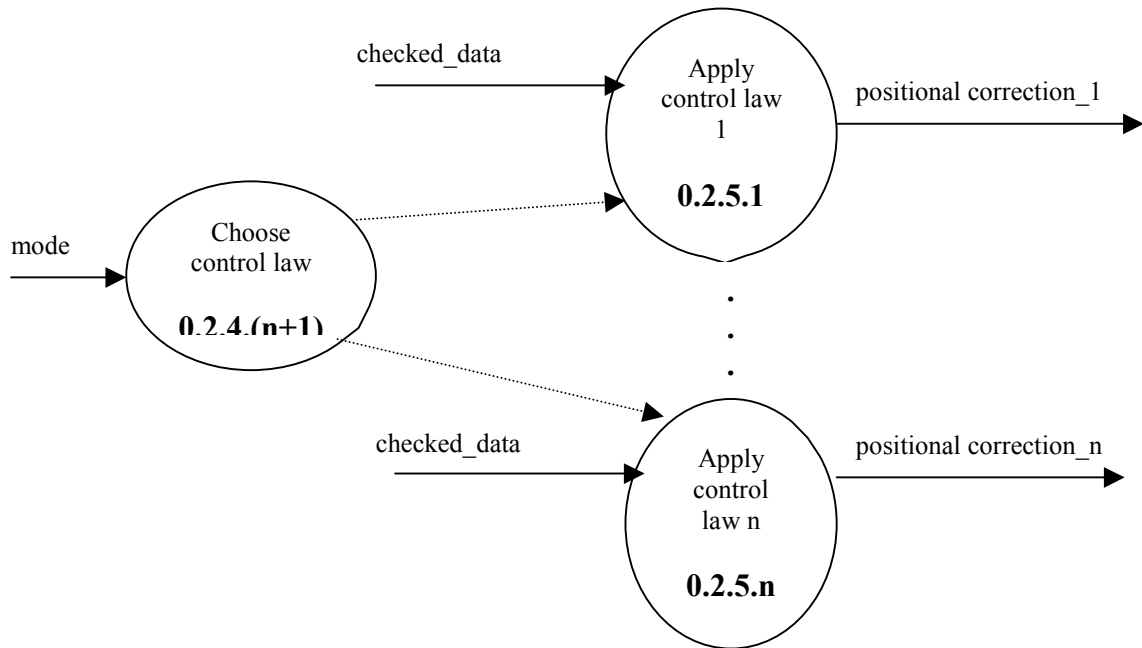


Figure 3.2.2-5: Possible data flow Level 3 diagram (case study B)

The procedure of decomposing processes and flows may be continued to any appropriate depth. However, at some stage a lowest level process is reached. Such a process is detailed in a 'mini-spec' using pseudo-code. For example, suppose 0.2.4.i (Figure 3.2.2.4) is a lowest level process then its mini-spec might read,

```

If (|Sensor_data_p_i - Sensor_data_p_i(previous)| < tolerance_i)
Then
    Sensor_data_p_i(previous) = Sensor_data_p_i
    checked_data_i = F_i(Sensor_data_p_i)
    checked_data_i(previous) = checked_data_i
    Enable_reconfiguration_i = false
Else
    checked_data_i = checked_data_i(previous)
    Enable_reconfiguration_i = true
Endif

```

Figure 3.2.2.6: Possible mini-spec: "MS 0.2.4.i : Self-check sensor i"

The individual mini-specs should, ideally, constitute the actual functional software requirements. These should then be complemented as necessary by requirements covering performance, interfacing, reliability, maintainability and so on.

It may be noted that there are software tools available which implement the Data Flow method. Such tools include the capability of checking consistency of the data flows. Thus, the objective of ensuring consistency in the software requirements is achieved to this extent at least. An automatic tool was used for the case under study.

Having established the software requirements, it is essential to indicate the user requirement(s) in the URD to which each software requirement is traced. From this information, one may generate a tracing matrix from software to user requirements (cf section 2.1) which, on sorting by user requirement, allows a check on completeness, that is, on whether all user requirements have been traced and none forgotten. Of course, it is a quite separate task to check that a claimed trace is actually justified.

It is emphasised that the above illustrations of the data flow approach are much simplified compared with the actual situation for the case under study. For example, the actual self-checking process involved complex interactions of data from different sensors. Consequently, it often proved difficult to generate mini-specs of the above simple form. This in turn could cause difficulty in tracing from software requirement to design (cf. section 2.2.2) in that a requirement might have to be implemented through a number of software operations or, instead, might give rise to a single, undesirably large software operation; in either case, the task of verifying the requirement could prove difficult. In general, also, control flow was an aspect of the actual project that proved difficult to represent in the Data Flow method without introducing substantial complexity.

A number of other tasks, besides definition of software requirements, had to be carried out during the software requirements phase. It was essential that an effort be made to assess the requirements from the point of view of likely resource (memory size and CPU load) needs. Also, it was important that a plan be produced which

specified the proposed method of verification of each software requirement. The method of verification might be through software unit test (e.g. in the case of a requirement implemented within a single software unit), through software integration test (e.g. a requirement implemented over a number of software units), through verification tests of the complete software in the simulated environment, through verification tests of the complete software on the actual processor (e.g. checking that critical timing requirements are met) or through analysis. The use of software unit and integration tests in verification is a departure from the 'V' diagram (Figure 3.1-1), but was essential given the level of detail to which requirements were specified. Both resource estimation and verification plan production could be commenced in parallel to SRD preparation but obviously could not be finished until some time after SRD completion. It was necessary also that plans for configuration management and quality assurance of the software be put in place during this phase, and that the overall plan for development of the software be progressed.

3.2.3 Architecture

The objective in the architectural design phase is to establish basic components of the software and interfaces between them; then, in the detailed design phase, the internal structure of each component is specified in detail. However, in order to establish interfaces between components, that is to specify what each component provides to and requires from other components, it is clear that the internal structure of each component must already have been considered in the architectural phase. Thus, the boundary between architecture and detailed design is not that clear-cut; the difference lies in the degree of formality used in documenting the design, within the two phases, as well as in the level of detail. Much depends also on the size of the software under development. For a large software the architectural phase may result in a decomposition which is not yet complete, that is, the lowest level components so far identified require further decomposition in the detailed design phase. On the other hand, it may be possible to completely decompose a smaller software in the architectural phase; this was the situation for the case under study.

The design approach was informally based on ESA's Hierarchical Object Oriented Design (HOOD) method /7/. The design components were objects representing, for example, data stores or physical entities such as sensors or abstract entities such as buffers. The HOOD method admits two types of relationship between objects, namely parent-child in which an object (parent) is decomposed into lower level objects (children), and senior-junior in which an object (senior) uses facilities provided by another object (junior). In HOOD, the design is presented by means of special diagrams and by detailed object design skeletons. The latter provide, for each object,

- (a) object description
- (b) definition of its provided interface in terms of types, data and operations
- (c) definition of its required interface in terms of types, data and operations
- (d) description of its internal structure (either identification of its child objects in the case of a parent object or definition of internal types, data and operations in the case of a (terminal) child object)

Figure 3.2.3-1: Content of HOOD object design skeleton

There are software tools available to support HOOD which are useful in controlling the large volume of documentation associated with the method as well as in maintaining consistency. However, for the case under study, such a tool was not used.

A key reason for the choice of HOOD was that it is designed to facilitate subsequent coding in Ada, the selected project language. HOOD objects are implemented as Ada packages (apart from the 'main' procedure), an object's provided interface being represented by a package specification while an object's internal structure is represented by a package body. An object's required interface is represented by Ada 'with' clauses which indicate the objects used by the given object; thus, the detail of required types, data and operations is not maintained from design to code (apart from being implicit in the package body).

It is not the intention here to present the actual case design, but an indication is given of the kind of objects identified and of key issues and constraints to be taken account of. The software's top-level decomposition was constrained to consist of an OS (junior) object and an ASW (senior) object. For the ASW, the ideal would be to have had some automatic process of generating design components from the requirements as stated in the SRD. This is, of course, not possible but the Data Flow method does suggest some candidate objects. In particular, several objects were based on significant data stores. For example, objects could be designed to represent all aspects of each of the data stores,

```
mode
checked_data
positional_correction
```

Figure 3.2.3-2: Typical design objects based on Data stores

identified in section 3.2.2. Each of the sensors and actuators provided candidate objects also. In various parts of the design a need for a general buffering mechanism was identified so that this also was a possible object. The Data Flow method also offered, of course, in its processes, candidates for operations within objects. Among the key issues to be addressed in the design process were those of sequencing of operations, of allocation of operations within time slots (see section 3.1), and of providing suitable mechanisms for reprogramming the software.

As noted above the separation, from the software design point of view, between architectural and detailed design may not be clear-cut. In particular, the HOOD

approach covers the architectural phase but also extends to the detailed design phase. The design may be considered to be complete when individual operations have been defined in detailed pseudo-code. From a planning and management point of view, end of the architectural design phase should be the point at which the interface between components is fully defined and baselined; by baselining is meant that the interface is placed under change control so that it cannot be modified without consultation and agreement of all affected parties. For the case under study it was essential, in view of the participation of separate contractors, that the interface between OS and ASW be baselined. It would have been beneficial, also, had the interface between ASW components been baselined; however, this was not possible until quite late in the project because of the need to facilitate incorporation of wide-ranging user requirement amendments. The main advantages of baselining the interface are,

- (a) Detailed design of components can be performed in parallel and independently by different team members
- (b) Test preparation (including planning, test case design and specification, and test harness development) can proceed on a sound basis in parallel to the detailed design

It was noted that the HOOD method entails a substantial level of documentation that must be maintained and, in particular, must be kept consistent. In the absence of specific tools for this purpose, it is possible to make use of the Ada compiler. The idea is to implement the interface between objects through Ada package specifications (provided interfaces) and 'with' clauses (required interfaces). Then the Ada compiler can be used to check the interface syntactically. However, this will not provide a detailed check on 'required interfaces' - for this, it would be necessary to introduce Ada package bodies with suitable (minimal) functionality. It is clear that if the component interface has been implemented in Ada as described then this represents a very significant advance not only in terms of checking interface correctness but also as a point of departure for detailed design and for test preparation. However, there is a serious danger in proceeding to Ada too early in that to do so requires a substantial investment of effort in implementation detail. This may be lost if the design has to be revised and, perhaps more seriously, may deflect attention from broader design considerations. This objection is true, particularly, if the interface is complex (which, of course, it should not be, ideally). For the case under study, the interface provided by the OS to the ASW was represented in Ada. However, within the ASW, it was not possible to use Ada to represent interfaces until late in the project as they could not be baselined until then, as noted above.

It was essential that the architectural design provide a trace (cf. section 2.2.2) from software requirements to design in order to demonstrate completeness, that is, that no requirements had been forgotten. As for the trace from user to software requirements, it is a separate task to check that a claimed trace is, in fact, valid.

There were a number of other activities to be performed during the architectural design phase. These included refinement of the overall software development plan,

and finalisation of the plan for configuration management of the software; the components identified by the architectural design constituted the software configuration items which would, from now on, be controlled separately through detailed design, coding and unit testing. The key technical activities to be completed (following completion of the architectural design) were revised estimation of memory and CPU usage, and preparation of a plan for integration of the software. This plan specified the order in which units (packages) were to be integrated together so that it directly constrained the order of their implementation. There are three basic integration strategies, bottom-up, top-down or a mixture (see section 5.2.1). For the case under study, the last mentioned strategy was used in order that an early impression of the performance of the system as a whole could be formed as well as that of key low level units. The primary purpose of software integration testing is to check that the interface between components is correct. For the case under study, this interface was implemented in Ada package specifications so that its syntactic correctness was assured. Moreover, any component stubs required during unit testing were implemented by means of modified package bodies but with actual package specifications. This meant that to a large extent the component interface was also checked semantically during unit test. Accordingly, the software integration test phase was shorter than anticipated with attention being focussed on larger scale integration tests of several components in combination, and on verification of relevant software requirements.

3.2.4 Detailed design

The purpose of the detailed design phase is to complete specification of the design of each of the software components identified during the architectural phase. In HOOD terms, this essentially means completing the description of the internal structure part of each of the object design skeletons. There may, of course, be a need to revise part of the baselined interface between components; however, such a revision would require formal authorisation before being implemented. The main documents produced during the architectural and detailed design phases are, respectively, an Architectural Design Document (ADD) and a Detailed Design Document (DDD). For the case under study, it was convenient and logical to treat the latter as a more detailed re-issue of the former.

In the detailed design phase, each individual operation is specified in pseudo-code. For the case under study Ada-like conventions were used in the pseudo-code; of course, design nomenclature preserved that of the SRD and URD so far as possible. This choice of pseudo-code meant that the design was largely self-documenting as quite high-level definition can be expressed clearly in Ada. It also had the consequence of blurring the distinction between design and code (cf. Figure 3.1-1 - 'V' diagram), i.e. of reducing the effort required for generating the code. This is a very significant advantage of Ada over lower-level languages but some thought is needed in order to maximise the gain.

The 'classic' picture illustrated in the 'V' diagram (Figure 3.1-1) views unit testing as a process of checking that the code is a correct implementation of the detailed design. Obviously, if there is no difference between detailed design and code then there is

nothing to check. This is an extreme situation but it does highlight that the various unit test strategies described in section 5.2.2 should be applied intelligently in order to achieve the test goal of maximising the number of errors detected. One implication appears to be that tests should be based on higher-level design than previously. Another is that emphasis should be placed on test cases that verify any software requirements traced to the software unit under test. A further point to note is that the software units of this case study were objects, incorporating a number of operations, data and data types. Hence, within each object, tests were required to check individual features with further tests needed to check integration of features.

During the detailed design phase, configuration management and quality assurance activities became increasingly prominent according as baselining of unit design and code proceeded, and testing commenced. In addition, it was necessary to refine estimates of memory and CPU usage, and to prepare a unit test plan. This plan specified the features to be tested, and any internal integration testing to be performed, for each unit. Subsequently, unit test case design and specification, and unit test harness preparation proceeded; these activities took up a very significant proportion of the available resources, in terms of both personnel and schedule. A first formal issue of the Software User's Manual (SUM) also appeared during this phase. This was an extremely important document for the case under study, as indeed it (or its equivalent) is for in most projects.

3.2.5 Test execution

Each coded software component had to be successfully exercised through a set of unit test cases before entry³ to a 'master library' of baselined components. The material required for unit test included the source code of the item under test, any required test driver and stubs, and a build procedure to perform all necessary compilation and linking. It had to be ensured that each item was copied from the correct computer directory in its correct version, and that following the test all materials and results were properly marked and stored. This task clearly required careful control and was facilitated by the use of special configuration management software. It was also essential that problems arising during test were properly recorded and followed up, and that re-testing and regression testing (see section 5.3.2) were facilitated.

Integration testing of baselined components of the ASW was performed on the simulated environment; also, Ada specifications of the baselined OS (delivered by that software's contractor) were integrated with the ASW. Verification tests of software requirements were carried out, so far as possible, on the simulated environment. On completion of these tests, the software was delivered to the AOCS contractor for testing on the real hardware environment.

It is important to note that delivery of software may not be a simple process. First, it is obviously necessary that the customer must accept the software, that is must agree that the software meets certain criteria. It is essential that these acceptance criteria be discussed and agreed by customer and software developer as early as possible in the

³ Should not be interpreted, necessarily, as a physical movement of data.

project's lifetime. For example, it might be agreed that the software should be accepted if it successfully executes all but a certain proportion of the specified verification tests. As part of the process of delivery, it is necessary to place the software on agreed media and in agreed format. This may not be simple; for example, for the software of the case study, it was necessary to place certain parts at specific memory locations, so that a complex linking mechanism was needed. Also, it is usually necessary to prepare documentation as a guide to installation. Finally, the product is not complete unless all its associated documentation - SRD, ADD, DDD, SUM and so on - is complete and up to date.

Following delivery to the AOCS contractor, the complete software (ASW and OS) was installed on the real processor, thus enabling verification of the ASW to be completed. This entailed performing some additional tests to verify requirements that were difficult or impossible to verify on the simulated environment. In particular, certain critical timing requirements were verified. These tests were performed in 'open-loop', that is, the spacecraft dynamics were not simulated.

On completion of the 'open-loop' tests, the software and processor (together making up the attitude control unit) were linked with a general software for simulating spacecraft dynamics. Then, tests were performed in which the unit's reactions to simulated operating conditions were checked. Essentially, these tests were checks on the URD, especially on control algorithms. Initially, these 'closed-loop' tests were performed with software simulations of other AOCS units - sensors, actuators, and so on. Then, the simulated units were replaced progressively by actual units, and thus the complete AOCS subsystem was built up. Clearly, throughout this process, modifications had to be made to the software - not so much because of errors in the software but to accommodate requirement changes or adjustments needed to take account of differences between designed and built hardware units.

3.3 Analysis : Notable Features, Lessons Learned

The ISO satellite was launched in 1993 and has since completed its mission with success (see /15/ for example).

This case study illustrates development of a highly critical piece of software under exacting technical, quality, cost and schedule constraints. While it can be fairly stated that the software development as a whole was successful there appear, nevertheless, to be some areas in which improvement may be possible.

A notable feature of the case study was that the user requirement for the software underwent very significant change in the course of the development. At the same time, it was not possible to wait until these requirements were completely mature before commencing software development - indeed, it was necessary to provide feedback to the user before some requirements could be finalised. However, a very substantial part of this feedback was generated during the analysis of requirements rather than in design and implementation. This was clearly advantageous, both in providing early feedback to the user and in allowing, for the software developer, a better distribution of resources over the project lifetime. In addition, it was possible to

constrain the software design such that the OS and ASW could be developed by different contractors and delivered at different times. On the other hand, it did not prove possible to segregate the ASW design into de-coupled components until quite late in the project, with consequent difficulties in terms of peaking of effort during coding and testing, and in terms of schedule. It would seem that this issue could be addressed in two ways,

(a) Starting from a single SRD, the software designer(s) place an absolute priority on achieving, as early as possible, a design which is very strongly de-coupled, with as simple an interface between components as possible. The achievement of this goal may be at the cost of optimality in use of resources and loss of flexibility in accommodating change.

OR

(b) Attempt, during requirements analysis, to segregate areas which are incomplete and provisional from those which are mature. Then, draw up separate SRDs for each area, including requirement (rather than design) level interface definitions. Softwares should then be developed quite separately for each SRD. The approach of separate SRDs was followed for OS and ASW in the case under study. It has the advantage of being achieved early in the project, before implementation detail has been introduced. It has the disadvantages of imposing far-reaching constraints, which may emerge later as severely penalising, on the software design.

It was noted that the standard 'V' diagram (Figure 3.1-1) did not adequately model some aspects of the project: the verification of software requirements during software unit and integration test, the abbreviation of software integration testing due to the use of Ada specifications, and the almost identity of detailed design and code with its implications for unit testing as well as in acceleration of the coding phase. The great importance of early planning of test and verification is stressed as well as the need for clear consensus between user (customer) and software developer on what constitutes 'software acceptance'.

The merit of the data flow method in providing a synthesis of functions required in the software is stressed. In practice, however, difficulties were encountered in reflecting (using the data flow method) the complex data processing and control mechanisms of the project. Moreover, the method required expenditure of a level of effort that was difficult to estimate for and to justify in the context of the project as a whole. However, if a somewhat more relaxed schedule were possible, with scope for greater emphasis on requirement analysis, then the method should certainly be seriously considered. In general, for a project with very tight constraints it seems essential - or at least prudent - to use techniques and tools that offer as little risk as possible from technical, cost and schedule perspectives. This is not at all a recommendation to avoid innovative methods and tools - it is essential to assimilate these in order to remain competitive.

In conclusion, during performance of an actual project, tradeoffs have to be made between what would be ideal and what is sufficient. For the case under study, for example, there would have been disastrous, and unacceptable consequences for the entire AOCS project had the first software delivery been delayed until it was felt to be absolutely impeccable in terms of the quality of its design.

4. Case Study C : Critical Real-time SW for SOHO AOCS

4.1 Context and User Requirements

This case study, like case B, is representative of on-board real-time software for satellites. As for case B, the software discussed is that which is resident in the controlling processor of a satellite's attitude and orbit control system (AOCS). It follows that the context and terminology of the two cases are very similar. This facilitates contrast and comparison of the two cases, allowing a briefer presentation of case C.

Note: Case C is only considered as far as completion of the architectural design phase.

Case study C is concerned specifically with the AOCS of the European Space Agency's Solar and Heliospheric Observatory (SOHO), part of the agency's Solar Terrestrial Programme /8/. The SOHO mission, in general, and its AOCS design, in particular, are such that the requirements on software were somewhat less complex than for case B. Of course, the basic operational principle of the AOCS was the same for both cases, and provision had to be made for similar types of functions. Also, the software in each case was required to have the same basic three-layer structure.

An important difference between cases B and C was that, in case C, a single contractor was responsible for development of all three software layers while the user (AOCS contractor) had responsibility for the processor hardware. There were, however, requirements, for a phased delivery of the software, beginning with layer 1, in order to comply with the overall AOCS development plan. The 'V' diagram (Figure 3.1-1) remains applicable as an overview of the software development context of case C.

The URD defined by the user (AOCS contractor) was structured somewhat differently than that of case B, a significant level of implementation detail being specified, particularly for layers 1 and 2. The URD was divided into sections within each of which numbered requirements were stated. The document appeared in successive issues, allowing material to be added and amendments to be made; thus, the question of how best to proceed with software development subject to changing requirements came up in this case also.

4.2 Software Development History

4.2.1 Familiarisation

As for case B, provision was made for a period of familiarisation prior to the main development. However, because of the experience gained in case B and due to a very tight overall schedule, this preliminary phase was quite short. In particular, it was decided that the same processor and language (Ada) would be used without need for an extended evaluation period.

4.2.2 Specification of software requirements

The requirements specified in the URD were in great detail and, particularly for levels 1 and 2, prescribed directions to be taken in implementation. Inevitably, however, it proved necessary to amend and refine the URD considerably, largely as a result of a process of dialogue between user and software developer but also due to maturing of the overall AOCS design. A key element of this dialogue was preparation of successive drafts of an SRD. As for case B, the main purpose of the SRD, given the detailed nature of the URD, was to restate, following analysis, the user requirements in a form suitable as input to the design process. Thus, it was necessary

- to separate functional from, for example, performance requirements
- to ensure so far as possible that each requirement was unique, unambiguous and verifiable
- to ensure that requirements were consistent and complete

Figure 4.2.2-1: Key objectives in requirement analysis and specification

At the same time it was very important to preserve URD notation and terminology as much as possible in the interest of reliability in checking the transition from URD to SRD.

It was decided to produce a single SRD for the complete software. This made the task of ensuring consistency and completeness easier. However, it was based on the assumption that it would be possible to freeze the user requirements prior to the main software development. This proved not to be the case so that, in retrospect, it might have been advantageous, but by no means trivial, to define separate SRDs for different parts of the software.

The time-scale for SRD preparation was quite short. Hence, it was decided to adopt an approach which had low risk from the point of view of schedule but at the same time would allow substantive dialogue between user and software developer, and would lead to an SRD with the desired properties as listed above. Moreover, it was important that the SRD should be such as to facilitate subsequent tracing from software requirements to design.

The URD specified to the software developer already defined a decomposition of the requirements by function. Thus, the high level decomposition into functions of layers 1, 2 and 3 was reflected and, within each layer, further levels of decomposition were specified. For example, in layer 1, interrupt handling was identified as a general function and was then decomposed into the functions required for the individual interrupts. However, at the individual requirement level functional, performance and other aspects were intermixed, and there was no definite way of ensuring consistency between requirements; this is not a criticism of the format of the user requirements whose role is not the same as that of software requirements.

The approach, then, to defining software requirements was to refine the functional decomposition specified in the URD, to separate functional from other requirements and, with consistency in mind, to make explicit data items (including control entities such as flags) appearing in requirements and to standardise their nomenclature. In

addition, a flexible, robust and precise way of tracing to user requirements was devised, thus enabling completeness checks to be made. Finally, some progress was made towards facilitating design of test cases for requirement verification.

It was decided to implement the approach using the facilities of a standard word processor (Microsoft WORD) rather than by means of a dedicated software tool. The key step in the implementation was to lay out the requirements in the following format,

FIELD 1	FIELD 2	FIELD 3
<SW requirement number>	<Software requirement text>	<Traced user requirements>

Figure 4.2.2-1: Format for software requirement specification

For example, the following might be a typical SRD fragment:

1.4	--Check if new oper. mode is commanded	
1.4.1	Read mode_data from hardware	U04.03(s1)
1.4.2	If mode_data implies allowed mode Then	U06.01
1.4.2	Derive op_mode from mode_data --Note25	
1.4.2	Endif	
	...	
	...	
2.1	--Initialise level 2 data items as needed	
	...	
2.1.5	op_mode=start_up	U03.11
	...	
	...	
3.5	--Apply control law	
3.5	Case op_mode is	
3.5.1	start_up :	U12.06(p1)
3.5.1	positional_correction=0	
3.5.2	coarse_attitude :	U13.04(p3)
3.5.2	positional_correction=...	
	...	
	...	
3.6	--Command actuators	
3.6	If op_mode is one of start_up , coarse_attitude , ...	
3.6	Then	
3.6	Perform A	
3.6	Else	
3.6	Perform B	
3.6	Endif	
3.6.1	--A Thruster control	
3.6.1	actuator_data_p=...	U17.01
3.6.2	B Reaction wheel control	
3.6.2	actuator_data_p=...	U20.05
	...	
	...	
4	--Performance requirements	
4.1	--Timing requirements	
4.1.1	If op_mode = start_up or coarse_attitude	U12.06(p2)
4.1.1	functions 3.5 , 3.6 shall be performed	U13.04(p4)
4.1.1	every 300ms	U17.02
4.1.1	otherwise ...	
	...	
	...	
4.4	--Accuracy requirements	
4.4.1	32 bit floating point arithmetic shall be used in	U12.06(p3)
4.4.1	function 3.5 (TBC50)	U13.04(p5)
	...	
	...	

Figure 4.2.2-2: Example of software requirement specification

It can be seen, in field 1, that the software requirements are numbered in ascending order, with provision for hierarchical decomposition.

It is clear from this example that by sorting on field 3 one obtains an ordering by user requirement reference; this enables a check to be made that no user requirement has been forgotten, that is, it provides a completeness check. As noted previously it is, of course, a separate task to check that a claimed trace is actually valid.

Bolding is used to highlight items of data but also other keywords of interest. It is possible (in MS WORD) to produce automatically a sorted list of keywords together with the corresponding requirement number. For example, for the above SRD fragment one obtains,

SW Req't No.	Keyword
4.1.1	3.5
4.4.1	3.5
4.1.1	3.6
3.6.1	actuator_data_p
3.6.2	actuator_data_p
3.5.2	coarse_attitude
3.6	coarse_attitude
4.1.1	coarse_attitude
1.4.1	mode_data
1.4.2	mode_data
1.4.2	Note25
1.4.2	op_mode
2.1.5	op_mode
3.5	op_mode
3.6	op_mode
4.1.1	op_mode
3.5.1	positional_correction
3.5.2	positional_correction
2.1.5	start_up
3.5.1	start_up
3.6	start_up
4.1.1	start_up
4.4.1	TBC50

Figure 4.2.2-3: Example of trace of "keywords" to software requirements

This list is an instrument that enables a consistency check to be made between requirements; data items that are common to requirements can be inspected for consistency in their use. In particular, it can be checked that provision has been made, where necessary, for initialisation of data items. The same list can be used as a convenient cross-reference in adding explanatory notes (e.g. **Note25**) or remarks on points of uncertainty (e.g. **TBC50** - TBC = To Be Confirmed).

The requirements on performance of functions may be linked to the functional requirements by the same bolding mechanism. Thus, the foregoing list identifies that 4.1.1 and 4.4.1 specify performance constraints on function **3.5**.⁴

Evidently, the SRD requirements should be stated in a clear and consistent way. In particular, a standard should be specified for the pseudo-code to be used. It is desirable that such a standard be simple but flexible. For the given example the only constructs introduced are

- simple statements (preferably imperative if part of a functional requirement)
- comments (prefixed by --)
- If .. Endif
- Case

Figure 4.2.2-4: Some pseudo-code constructs for requirement specification

There is some attraction in introducing further conventions; for example, to indicate whether a data item is read, or written to, or both in a particular requirement. Such a convention would yield, for the data item **op_mode**,

op_mode (R)	1.4.2
op_mode (R)	2.1.5
op_mode (R)	3.5
op_mode (W)	3.6
op_mode (W)	4.1.1

Clearly, such a convention would "show up" an item that was read from but never written to.

A basic objective in verifying a functional software requirement should be to test that all its logical paths are implemented correctly. For example, in requirement 3.6 above, it would be necessary to test all values of **op_mode**. It is desirable to specify the required test cases to the test implementor in a simple form. A decision table is such a form. For example, for requirement 3.6 above a decision table would be,

⁴ It is interesting to note that the process of bolding keywords and subsequently producing a sorted list is similar to that used by Dr. Johnson in preparing his dictionary. Thus, from /9,p249/,

'He began his task by devoting his first care to a diligent perusal of all such English writers as were most correct in their language, and under every sentence which he meant to quote, he drew a line, and noted in the margin the first letter of the word under which it was to occur. He then delivered these books to his clerks, who transcribed each sentence on a separate slip of paper, and arranged the same under the word referred to. By these means he collected the several words and their different significations, and when the whole arrangement was alphabetically formed, he gave the definitions of their meanings, and collected their etymologies ...'

Incidentally, this note is a reminder that it is important to include in the SRD a data dictionary in which each data item is defined.

	TEST CASE	1	2	...
CONTROLS	op_mode=start_up	T	F	
	op_mode=coarse_attitude	F	T	
	...			
RESULTS	Use thruster control	T	F	
	Use wheel control	F	T	

Figure 4.2.2-5: Fragmentary example of a decision table to facilitate verification

This is a rather simple illustration but it is not difficult to see that a small amount of additional logic would cause considerable complication and would make the task of generating the decision table both time-consuming and error prone. It is possible (in MS WORD) to create semi-automatically a blank decision table for a given piece of logic and this, in itself, reduces the effort required for the task considerably. However, it would be very desirable to have automatic generation of complete decision tables - this would seem mainly to require building a simple interpreter to process pseudo-code fragments.

4.2.3 Architecture

The design approach for this case was again based on ESA's HOOD method. In this case all three software layers were the responsibility of one contractor so that the design process was more unified. However, it was still essential that inter-component interfaces be established early, particularly between layer 1 and the rest of the software, in order to accommodate the phased software deliveries required by the customer.

For this case study, it was decided to acquire a software tool to assist in implementing HOOD. The tool was a very simple one, providing the facility to prepare HOOD diagrams and then to generate automatically outline object design skeletons. It provided also a rather limited level of consistency checking. It can be concluded that the tool was definitely of benefit in the design process, mainly in terms of its diagramming functions but also in helping to control the volume of documentation associated with the method. There are more ambitious and much more expensive tools available but these would not have been justified for a project of this scale.

The format of the SRD was found to lend itself quite well to the task of tracing software requirements to design components. In particular, the software requirement numbering system allowed precise tracing, for those cases where design considerations dictated splitting up of a requirement, to where each part of the requirement was implemented. Obviously, cases arose where it was seen at design time that requirements should be moved from one software layer to another; these could be accommodated by orderly change of the SRD (subject to the constraints of phased software deliveries).

From the experience of the design process for this case study successive steps emerge, as follows:

- (1) Specify architectural design (in detail but not in machine-readable form)
- (2) Implement the component interfaces by means of Ada package specifications (and possibly with Ada package bodies of minimal, but sufficient functionality to check 'required' interfaces). Check with the Ada compiler
- (3) Expand Ada package bodies to complete detailed design
- (4) Complete transition to Ada code

Figure 4.2.3-1: Steps in the software design & coding processes

It should be noted that, in practice, these steps overlap to some extent. Also, the distinction between (3) and (4), for some components, may be negligible or non-existent.

It was noted in section 4.2.2 that, in fact, the user requirements underwent greater change than had been assumed in making the decision to base the development on a single SRD. Accordingly, this evolution of the URD had to be accommodated in the architectural design phase. Essentially, the first delivery of the software (layers 1 and 2) was to be based on one issue of the URD and a second delivery on a subsequent URD re-issue. The strategy adopted to accommodate these constraints was to implement only very mature level 1 and 2 requirements in the first delivery, and postpone implementation of all other requirements to the second delivery. There were drawbacks in this approach, particularly in movement of functions, which should from a technical point of view have been in layer 2, to layer 3 with some consequent distortion of the software design as a whole.

There was an appreciable difference, between cases B and C, in the design for scheduling function execution. For both cases, it was required to ensure that functions appropriate to the current operational mode were scheduled, in proper sequence and time-slot. In case B the approach adopted was to employ a small number of tables (in particular to distinguish between initialisation and normal operation) through which slot-level functions were scheduled and, within each slot, to implement the controlling logic by means of a hierarchy of procedure calls. In case C, on the other hand, tables were used within each slot to schedule execution of functions, the choice of table being dictated by current conditions, particularly the operational mode).

4.2.4 Detailed design and test execution

This document does not address these later project phases. In brief, the same general approach applied as in case B except that the verification of many software requirements at unit and integration level was systematically planned from the outset.

4.3 Analysis: Notable Features , Lessons Learned

The most notable innovation in this case study was the method of treatment of software requirements and, particularly, the format of the SRD. It is believed that this approach is a very effective one while, at the same time, being quite simple to apply. It lacks, perhaps, a facility to present an overview of the requirements (such as provided by the data flow method). It is certainly a very good starting point for forming a data flow representation of the requirements.

It would, perhaps, have been more efficient to base the development on a number of separate SRDs, corresponding to the required number of phased deliveries, as this might have accommodated changes in the user requirements more conveniently. However, it was in fact possible to accommodate these changes during the design phase.

Finally, it was recognised from the outset, and planned accordingly, that many software requirements should be verified at software unit and integration level.

Remark: Following successful recovery from a temporary loss of communication with ground (in 1998), the SOHO satellite continues to perform satisfactorily (see /16/).

5. Phases of Software Development - Technical Content

5.1 Introduction

The purpose of this chapter is to summarise, on the basis of the test cases studied, the main technical work to be performed during each phase of a software development, and to supplement this information with further detail on approach, methods and tools. In particular, elements of each phase, as specified in/10/, are noted; /10/ is a standard which has been applied to several projects over many years so that it represents a widespread and matured view of software development.

It is clear from the case studies of chapters 2 to 4 that a software development may be considered to consist of a number of successive, possibly overlapping, phases. These are, in chronological order,

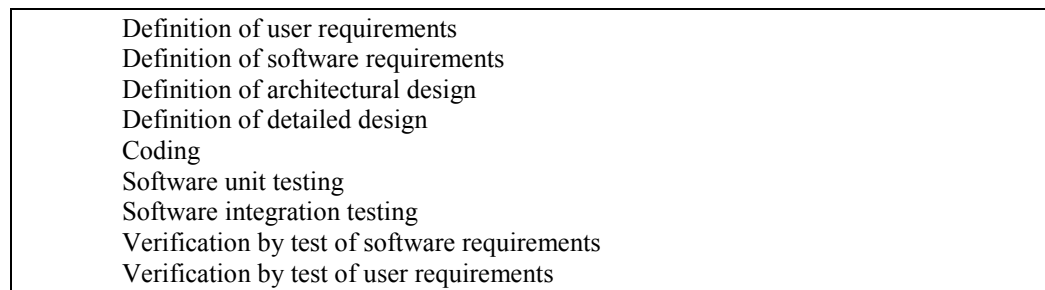


Figure 5.1-1: Phases in software development

The cases have illustrated that different contractors may have responsibility for different phases. Moreover, within the scope of a given contractor's tasks, separate teams or individuals may have responsibility for different phases.

It was seen to be useful to display the different phases in the form of a 'V' diagram. However, such a diagram may need to be modified to accurately reflect particular situations such as that much of the phase 'verification by test of software requirements' should, for the cases studied, be incorporated in the software unit and integration test phases. Also, it was seen that choice of the point of separation between phases is not always clear-cut.

The phase titles given above reflect the basic activities engaged. In addition, there are several associated activities that must be performed. Chief among these are test planning and preparation, resource estimation, quality assurance, configuration management and, of course, project management.

The point of separation between development phases has been acknowledged to be somewhat arbitrary and, indeed, the whole phased approach may be criticised. It is not the intention here to make a detailed evaluation and comparison of approaches to software development. However, some relevant points from the case studies are noted.

1. In case study A no distinction was made between user and software requirements nor was a **formal requirement specification** written; this proved to be feasible, due to the nature of the development and the team composition, and therefore efficient in schedule and cost. However, it meant that detailed specification of requirements and design proceeded in parallel, with the risk of loss of overview and of spending disproportionate effort on some aspects of the software. Also, the lack of a formal specification inhibited early preparation of a complete set of requirement verification tests.

2. In case studies B and C, the division into **separate phases** was strictly maintained, due to the high criticality of the software. In particular, separate user and software requirement documents were prepared; this was essential as the user and software developer were separate contractors. Verification of the software requirement document, prepared by the software developer, served as a basis for acceptance of the software (but see section 5.2.1.2).

3. In case study B, a **prototype** software was developed prior to the main development. The prototype development was also phased with, in particular, user requirement, software requirement, and design documents. No part of the prototype was used directly in the actual software, nor was it intended to be.

4. The distinction between **architectural and detailed design** is not a clear-cut one, as illustrated in cases B and C. A primary purpose of the architectural design is to baseline the interface between software components. This means that, after issue of the architectural design document, changes in the interface can only be made with the consent of all interested parties. Thus, there is a natural reluctance on the part of the software designer not to issue the document until the design of each component is essentially complete; this is because, subsequent to the issue, the design will be constrained by the baselined interface and therefore by the views of other parties. On the other hand, it is important in a project of any significant size, to receive views and obtain agreement on the high level design before committing resources to detailed design. Moreover, from a management point of view, it is essential to have a controlled interface between components so that detailed design and implementation of separate components can be carried on in parallel by different team members, and that associated test preparations can proceed on a sound basis.

5. Case B illustrates that it is possible to baseline an interface between components early and successfully. This was in the situation that two parts (OS and ASW) of the same software were developed by separate contractors. In fact, however, the separation between the two parts was made earlier in that **separate software requirement documents**, based on distinct parts of the user requirement document, were produced. It was possible to proceed with and complete the development of one part (OS), for which the user requirements were mature, in advance of the other part (ASW). Of course, this required that the interface between OS and ASW be defined early thus imposing constraints on the, later, ASW design.

6. In general, there is a **difficulty in the phased development approach when user requirements are not mature**. It is clearly wasteful to devote resources, whether in preparation of a user requirement document or in software development work, to requirements that are quite uncertain. However, as in cases B and C, there may be areas of the requirements that are firm and for which it is necessary to proceed to development in order to meet overall project constraints. It would seem that the least risky approach to handling this situation is to prepare separate software requirement documents corresponding to different parts of the user requirements. Then, independent (apart from strict interface control) software developments could proceed in accordance with the phased approach. There might be just two software requirement documents and corresponding software developments, as for OS and ASW in case B, but there could be more than this. It is clear that this strategy would force early high-level design decisions to be taken. However, it could well be more effective to do so at this point rather than later when issues might be clouded by implementation concerns. Essentially, this was the approach adopted in case A, albeit in an informal way.

7. It was found, in case B, that the distinction between detailed design, expressed in pseudo-code with Ada syntax, and actual code was sometimes blurred. This is, essentially, a benefit of Ada's high level features. Planning should be such that this benefit is maximised to reduce both the **detailed design and coding phases**, and to maintain a clear criterion for unit testing.

8. The importance of maintaining a **distinction between the different test phases** was emphasised in case A, but holds true for all developments. The main point is that software should be thoroughly tested at one level before proceeding to the next, higher level; this is in order to minimise the probability of low level errors when testing for high level errors - otherwise it becomes extremely difficult to debug, as there are so many possible sources for an observed failure, and the process of re-testing on correction of an error may be quite complex and time-consuming.

5.2 Requirement Definition

5.2.1 User Requirements

5.2.1.1 General

Definition of user requirements, while not part of the software development process proper, is its essential prerequisite. The user requirements, from the software developer's perspective, must be such as to allow the development to proceed effectively and, from the user's point of view, must encapsulate all the needs which the software is expected to fulfill.

The user requirements may already have been drafted prior to the software developer's involvement in the project. Thus, it may not be possible for the software developer to influence the user requirement content nor indeed (perhaps) should it be. However, it is important for the software developer that all areas of uncertainty or immaturity in the user requirements be clearly indicated. Also, it is very desirable that the user

requirements be stated clearly and concisely, with a minimum of duplication, and that they be checked for consistency, so far as possible.

It is pointed out that the method of specifying software requirements, illustrated in section 4.2.2, may also be applied to the definition of user requirements, perhaps with less formality.

In case study A, the process of formulation of user requirements is illustrated. Tracing matrices are used to demonstrate that higher level needs are covered by the user requirements. Some illustrations are given in the case study to distinguish between user and software requirements. The distinction is not always clear-cut but, in general, user requirements should exclude implementation detail as much as possible.

It is of some interest to refer to /11/ in which organisation of higher level requirements is discussed. That paper is concerned with formulation of system requirements in a structured way such that their status can be used as a measure of progress throughout the whole lifetime of a project, not just in the early creation and late verification stages. It is emphasised that adequate time and resources should be devoted to the requirements creation stage as this is when the system can be improved enormously at little cost; on the other hand, errors persisting from this stage are the most expensive to correct. The author regards it as essential that requirements reflect the fundamental needs of the customer and not be concerned with solutions, that requirements should be compact in size, stable, uncomplicated and available early. The basic model used in the paper is that of first creating the user requirements, then describing the functional system and then making a design to meet that functionality. (cf. sections 5.2.2.1 and 5.2.2.2). The author goes on to classify requirements under the headings, operations, functional, performance and constraints. Then, it is indicated how the requirements can be used as a measure of progress throughout the project lifetime, with emphasis on qualification (before) over acceptance (after). Next, statement of individual requirements and organisation of the requirements as a whole is discussed. It is considered that certain CASE (computer-aided software engineering) tools, while of some benefit, are by no means adequate for systems engineering purposes; none can handle all the elements necessary, not just functional aspects but also constraints (such as design rules, standards, environmental conditions), partitioning among subcontractors, milestone and review planning, and so on).

Some further material of interest, for both user and software requirement definition, is given in /13/.

5.2.1.2 ESA SW engineering standards

The user requirement definition phase is considered in chapter 2 (part 1) of /10/. There, it is considered that the user requirement document defines the basis for acceptance of the software, contrary to case studies B and C. In fact, this point is a matter for contractual agreement between user and developer - case studies B and C were, perhaps, unusual as a result of being the subject of a complex test process at different contractor's premises.

In /10/ it is stressed that the nature of exchanges between the software and external systems be specified by the user and controlled from the outset, perhaps by means of a separate interface control document (ICD).

Two types of user requirements are distinguished in /10/, capability requirements describing functions and operations needed by the user, and constraint requirements on how the software is to be built.

Each user requirement must, according to /10/, have a number of attributes, namely

Identifier (for later tracing to software requirements)
Need (essential or not)
Priority (if incremental delivery)
Stability
Source (trace back to higher level needs)
Clarity
Verifiability

Figure 5.2.1.2-1: Attributes of user requirements (ESA standards)

5.2.2 Software Requirements

5.2.2.1 General

Definition of software requirements is the first, and arguably the most important phase of the software development process proper. The chief output of the phase is a software requirements document, prepared by the software developer. The software requirements document is the culmination of a process of familiarisation by the software developer, including dialogue with the user; this process may well result in modifications to the user requirements document also. The software requirements document is an important interface between user and developer; it is the detailed statement of the developer's view of what is required and so allows the user an opportunity, before the design phase, either to state agreement with this view or to require any deficiencies in it to be made good.

Definition of software requirements is a process of analysis in contrast to the **synthesis of software design and implementation**. During the software requirements phase, the user requirements are analysed and expanded in such detail that no uncertainty as to what is required remains for the design phase.

The software requirements document should be prepared by the software developer even where, as in case studies B and C, the user requirements are very detailed. It is essential that there be a precise input to the design phase and this may mean re-organisation and rewriting of the requirements. However, it is extremely important that the software requirement document remain accessible to as wide a user readership as possible so that the step from user to software requirements can be subject to a thorough and fruitful review. Such a review is facilitated both by the tracing matrix from user to software requirements, and by avoidance of over-specialised, software-specific terminology and notation.

A central feature of a software requirements document should be its separation of requirements of different types. Most importantly, functional requirements are separated from other aspects such as performance, operations, quality, documentation and test. The case studies illustrate the usual approach of defining functional requirements in a hierarchical, top-down manner. This process of definition should result, at the lowest level, in requirements that embody simple elementary functions, likely to map onto design without need for further decomposition. However, so far as possible, care should be taken to avoid requirement formulations that may constrain design unnecessarily. The definition of requirements process will identify items of data, and care must be taken to define each item carefully and to ensure that data are used consistently throughout all requirements.

There are three important characteristics which software requirements should have, namely completeness, consistency and verifiability.

- Completeness can be checked with the aid of the tracing matrix from user to software requirements.
- Consistency can be checked, to a large extent, by careful definition of data and control, and by analysis of data and control usage throughout the requirements. The results of analysis of data item usage is also an important input to the design process.
- Verifiability may be tackled by defining a number of verification methods (for example, review, analysis, software unit test, software integration test, software system test) and then stating by which method each requirement will be verified. If a requirement is found for which none of the methods apply then either some other method must be defined or the requirement should be rewritten. The table of software requirements and associated methods of verification is called a verification control document (VCD) or matrix (VCM); later, this table is augmented by addition of references to actual test or analysis reports.

It is important that actual verification of requirements should be commenced as soon as possible. In particular, as in cases B and C, it may be necessary to produce a first analysis, based on the software requirement document, to verify resource requirements. Also, it should be possible to design system level tests in some detail; there is no simple, comprehensive approach to system testing /12/ but, typically, it may cover testing of timing requirements, testing of typical scenarios such as execution of individual operational modes and mode transitions in cases B and C, and volume testing such as simulation of the maximum set of calibrating stars in case A. For the design of tests to verify requirements at software unit and integration level it should be possible to draw on the strategies noted in section 5.3.2.1 for unit test design.

There are a variety of methods and software tools available to assist in the process of analysis and software requirement definition. The data flow method (with enhancements for control flow) was illustrated in case study B. Among others may be mentioned the structured analysis and design technique (SADT) and the Vienna Design Method (VDM). Finally, the approach illustrated in case study C is suggested as a straightforward, quite powerful and general aid to requirement analysis and definition; moreover, it allows incorporation of the VCD mentioned above, and of subsequent tracing to design. Thus, it can be used to enable a considerable degree of requirement monitoring throughout the project lifetime (cf. discussion on system requirements in section 5.2.1.1).

5.2.2.2 ESA SW engineering standards

The software requirement definition phase is considered in chapter 3 of /10/. It is noted there that sometimes prototyping may be necessary to clarify or verify certain requirements. The importance of constructing a 'logical' (implementation-independent) model of the requirements is emphasised; such a model is usually a top-down decomposition of the main function. Rules to be observed for a good logical model are identified as,

<p>Functions should have a single definite purpose Functions should be appropriate to their level in the hierarchy Interfaces between functions should be minimised Functions should be decomposed into no more than 7 sub-functions Implementation information should be omitted Performance attributes of each function should be stated Critical functions should be identified</p>
--

Figure 5.2.2.2-1: Rules for a good logical model (ESA standards)

In /10/, several types of requirements, functional, performance, interface, operational, resource, verification, acceptance testing, documentation, security, portability, quality, reliability, maintainability and safety, are identified and discussed.

As for the user requirements (see Figure 5.2.1.2-1) each software requirement should, according to /10/, have the seven attributes of

<p>Identifier (for later tracing to design) Need (essential or not) Priority (if incremental delivery) Stability Source (trace back to user requirements) Clarity Verifiability</p>

Figure 5.2.2.2-2: Attributes of software requirements (ESA standards)

In addition, /10/ specifies the need for software requirements to be complete, consistent and, so far as possible, non-duplicated. Completeness involves ensuring that no user requirements are forgotten but also that an activity has been defined for each input. A number of types of inconsistency are noted: the use of different terms

for the same thing, the same term for different things, incompatible activities happening at the same time and activities happening in the wrong order.

5.3 Design

5.3.1 Architecture

5.3.1.1 General

The primary purpose of this phase of software development is to achieve definition of a high-level software design which satisfies the requirements specified in the software requirements document. The main software components and the interfaces between them are identified and described. It is necessary to specify the provided and required interface, including operations, data and data types, for each component.

Essentially, the software design is a synthesis of the functions and data specified in the functional requirements, such that all aspects of the software requirements document are satisfied. There are various design approaches of which cases B and C illustrate a hierarchical, object-oriented approach. It is generally agreed that a top-down approach should be used although it should be borne in mind that the design process is an iterative one. A very useful element of the design documentation is a tracing matrix, from software requirements to design components, which indicates where each requirement is implemented.

It is important to decide how the design is to be documented. The possibilities range from an informal level, preferably complemented by diagrams, to formal, machine-checkable documentation. The advantages of the former are that it can be produced with relatively limited time and effort, is accessible for review to a non-specialised readership, and can be modified following review without much additional effort. The advantages of the latter are that there is greater assurance that it is comprehensive, correct and consistent and, moreover, it allows easy and rapid transition to the next phase. In fact, as illustrated in case studies B and C, the best approach is probably to plan explicitly to have a number of documentation stages, proceeding from the informal to the formal level.

There are a variety of design methods and associated tools that are useful, particularly in managing and maintaining consistency of the details of the design. Cases B and C have illustrated the use of Ada and HOOD in this regard.

An important complementary activity to architectural design is planning of software integration tests to check whether the interface between components has been correctly implemented. As discussed for case B (section 3.2.3) there are two 'pure' strategies for software integration, bottom-up and top-down, but a mixed strategy is also possible; see /12/ for further details. There are arguments for and against each of these strategies, for example that top-down is likely to lead to early detection of any fundamental high-level errors whereas with a bottom-up approach such errors would persist until very late in the integration process. On the other hand, it could be argued that bottom-up is more efficient in that it reduces the number of component stubs that

have to be written. The significant advantage of using Ada package specifications to define the component interface, and hence to provide an early syntactical check on that interface is illustrated in case B. This led to very substantial reduction in the overall software integration test phase for that case. Finally, it may be noted that there are software tools available to assist in the process of integration test case definition.

5.3.1.2 ESA SW engineering standards

The architectural design phase is covered in chapter 4 (part 1) of /10/. It is considered there that the architectural design is complete when the level of definition is sufficient to enable individuals or small groups to work independently in the detailed design phase; see above for similar remarks (e.g. sections 3.2.3 and 5.1).

In /10/, the concept of a 'physical' model, which describes the software design, is used. This is in contrast to the 'logical' model of the software requirements phase. The design process is seen, essentially, as the transformation of the 'logical' to the 'physical' model.

It is mandatory, in /10/, that the design be derived in a top-down manner as this is considered vital for controlling complexity.

Desirable properties of a software design are, according to /10/, adaptability, efficiency and understandability. It is considered that these goals are more likely to be achieved by aiming for simplicity of function and form in all parts of the design. Simplicity of function is achieved by maximising 'cohesion' of each component, while simplicity of form is achieved by

Minimising coupling Components should be appropriate to their level in the hierarchy Matching software and data structures Maximising the number of components which use a given component Components should be decomposed into no more than 7 child components Removing duplication between components by creating new components

Figure 5.3.1.2-1: Design rules to ensure simplicity of form (ESA standards)

It is interesting to compare the above with the rules for a good 'logical' model quoted in Figure 5.2.2.2-1.

In /10/ it is suggested that the level of detail in the architectural design document will be such as to show which functions each component provides but not necessarily how this is done; again, that it will be stated what information is exchanged in the interface between components but not how this is done. This view would correspond to the informal level of documentation discussed in section 5.3.1.1, above. However, the absence of implementation aspects does not imply that a considerable level of detail is not called for.

It is stated in /10/ that specification of the architectural design should cover

Functional definition of the components Definition of the data structures (that are part of the interface between components) Definition of the control flow (between components) Definition of the computer resource utilisation
--

Figure 5.3.1.2-2: What architectural spec. should contain - ESA standards

Finally, in /10/, it is suggested that the programming language be selected during this phase. However, in many projects as in cases B and C, this decision will be made in an earlier phase.

5.3.2 Detailed Design

5.3.2.1 General

In this phase, the design of each component is detailed to the extent that coding can commence. There is scope for considerable variation in the choice of separation point between architectural and detailed design; much depends on the total size of the software, the chosen design method including the pseudo-code used, the implementation language, and the schedule constraints on the development. For example, in case study C (Figure 4.2.3-1) four design and coding steps were identified of which the first and third should certainly be in the architectural and detailed design phases, respectively, but the appropriate phase for the second step is arguable.

It was found, in case B particularly, that the use of Ada both to specify design and as implementation language led to a situation where there was sometimes little distinction between detailed design and code. However, in general, it is desirable to maintain a definite distinction, by early termination of the design phase if necessary, in order to avoid unnecessary repetition of effort (especially in review) and to ensure that there is a clear criterion for unit testing.

The purpose of unit testing is to check whether the detailed design of each component has been correctly implemented in the code. There are a number of strategies that may be used to design unit tests. These are presented in detail in /12/ and are briefly mentioned here. First a distinction is made between white and black box testing. In the former case the code is visible to the test designer and attention is concentrated on generating test cases which exercise all its logical paths. In the latter case, the test designer has visibility only on the input and output of the item under test. A basic strategy in this case is to focus on exercising boundary values of the input and (if possible) of the output. It is noted that there are software tools available that assist in the preparation of test data.

For case studies B and C, the design components were implemented as Ada packages that generally contained more than one procedure. In these cases, test data sets had to be prepared for each individual procedure as well as for any integration of procedures internal to the package; also, the correct usage of any static data internal to the package had to be monitored.

An important complementary activity that should be well advanced at the end of this phase, if not earlier, is preparation of a software user manual.

5.3.2.2 ESA SW engineering standards

In /10/, the detailed design phase is described in chapter 5 (part 1), together with coding and testing, under the heading 'The Detailed Design and Production Phase'. First, it is foreseen that the process of top-down decomposition, if not already complete in the architectural phase, is continued until lowest level components are identified. Details of each component's design are then seen as proceeding by a process of stepwise refinement.

5.4 Implementation

5.4.1 Coding

5.4.1.1 General

In this phase, each component is implemented in the chosen language. While this should be a relatively automatic process, the level of effort required depends on what has already been achieved in the detailed design phase, and on the implementation language. For case B, much of the detailed design was already very close to the Ada code so that the effort was mainly to make minor adjustments to render it compilable, and to add in some remaining implementation details. On the other hand, if much or all of the implementation is to be at assembly level (to take an extreme case), then it is clear that the coding phase will require very considerable effort.

On completion of coding, each component should be submitted for unit test. Therefore, all necessary test drivers, stubs and build procedures should be prepared, so far as possible, in parallel to coding.

5.4.1.2 ESA SW engineering standards

In /10/ chapter 5 (part 1), it is noted that coding conventions and standards should be established. In fact, it is important that appropriate standards be defined and agreed for all phases of the software development. The importance of producing consistent and structured code is stressed in /10/. The use of language sensitive editors is advised. It is pointed out that compilation is part of the coding process and that use should be made of compile-time statistics.

5.4.2 Test Execution

5.4.2.1 General

On completion of coding, each component is submitted for unit test. Then, the units undergo integration testing according to a specified strategy. Finally, the software as a whole is submitted for system test. This general scheme may be adapted to meet particular circumstances, as illustrated in cases B and C, or may be prepared with more or less formality as illustrated by case A.

It is extremely important from the point of view of schedule that all test preparations be completed in a timely manner so that unit testing can proceed as soon as possible after completion of coding. This includes test case design and generation, preparation of all test harness, and specification of all required test procedures. This is necessary because test execution may, in itself, require a very significant proportion of the total time available to the project.

On successful completion of its unit test, a component must be placed under strict configuration control. This means that the component 'source' file and any required test harness - drivers, stubs, build procedures, execution procedures - must be placed (not necessarily implying physical movement of data) in a secure library from which they can be efficiently retrieved if needed for re-testing or for integration testing. Also, any associated documentation, particularly the test report, must be similarly configured. Thus, it is essential that the configuration management system be in place and operational in advance of test execution; software configuration management tools are extremely useful in this area. If a unit test is unsuccessful then the component must be returned to its coder for repair. The procedure for handling this must be carefully considered to ensure that proper control is maintained while avoiding introduction of excessive formality and consequent delay.

Integration tests are performed by retrieving relevant units from the configured library, and by combining them with the appropriate integration test harness. If an integration test is successful then the corresponding harness must be placed under configuration control. If an integration test is unsuccessful then diagnosis is necessary to determine which of the units under integration (or perhaps the test harness) is at fault. Then, the unit in question must be repaired and resubmitted for unit test. Also, any previous integration tests of which it was a part must be repeated, to some extent at least - it may not be feasible to repeat all tests. Finally, the original integration test must be repeated. Thus, it will be seen that there is a need for careful control of procedural aspects of the process to ensure, for example, that all problems have been resolved, that the correct versions of all software are used, and that all associated documentation is up to date.

On completion of integration testing, the software undergoes system testing. As for the earlier test phases, there is need for proper and efficient control of problem reporting, re-testing and configuration.

Depending on the particular project, it may be necessary to have independent monitoring of some or all of the test phases by quality assurance personnel. These personnel may have particular responsibility for checking the verification control document (see section 5.2.2.1).

It will be seen from the above that test execution must be carefully planned and controlled, and may entail a very substantial degree of procedural formality. The level of formality depends on the nature of the project and, obviously, the greater the formality the greater the time needed.

5.4.2.2 ESA SW engineering standards

In chapter 5 (part 1) of /10/ some of the goals and methods of unit, integration and system testing are stated. Configuration control, including aspects relevant to the test execution phase, are discussed in chapter 3 (part 2) of /10/.

6. Phases of Software Development - Review Mechanisms

6.1 Introduction

As noted in section 2.3, execution of software tests is not the only means of assuring correctness of a piece of software. A different means is provided by the mechanism of reviews. Essentially, reviews take place in order to check whether the work performed in one phase or sub-phase of the development is sufficiently complete and correct to enable the next phase or sub-phase to commence.

Reviews occur at major project milestones, with participation of customer (user) representatives, or at intermediate points in which the participants are members of the development team only. Both the major formal reviews and the intermediate reviews are identified and discussed in the following sections.

Reviews are described in this separate chapter in order to highlight their importance within a project. They are important as a very effective means of checking correctness and completeness of work done. In addition, they are key elements in project planning, they require considerable resources to be properly carried out, and have significant impact on schedule.

6.2 Key Formal Reviews

6.2.1 Identification and Purpose of each review

The major project reviews correspond to the different development phases. Thus, there is a user requirements review (UR/R), software requirements review (SR/R), architectural design review (AD/R), and detailed design reviews (DD/Rs). There may be more than one DD/R - this would depend on the total size of the development and possibly on constraints such as incremental deliveries. In addition, it is usual to have a period of acceptance testing prior to software delivery. It is essential that there be early agreement between user and developer as to acceptance criteria and format and procedural aspects of acceptance testing. Normally, a test readiness review (TR/R) is held prior to commencement of formal acceptance testing, while a test acceptance review (TA/R) is held afterwards.

The basic purpose of each review is relatively self-explanatory.

Thus, a UR/R must check that the user requirements document correctly reflects all the customer needs, including adherence to any high level specifications, and provides a sufficient input to the software developer. Depending on the particular project, the UR/R may also consider and agree acceptance criteria.

The purpose of the SR/R is, primarily, to check that the software requirements document reflects the user requirements correctly and completely, and that the software requirements are unambiguous, consistent and verifiable. It must check also that the software requirements document complies with agreed standards. In addition,

the SR/R may review the initial verification control document, test plans, resource estimates, plans for quality assurance and configuration management, and project management documents.

The purpose of the AD/R is to check that the architectural design correctly reflects all the software requirements and that it adheres to agreed standards. It is likely that associated material will also be reviewed particularly resource estimates, test documents and project management documents.

The precise purpose of a DD/R will depend on the particular project, including the available resources. With reference to case study C, section 4.2.3, a suitable subject for review might be the implemented and compiled component interfaces. Later, a subject for review could be the detailed design of a set of components. It might also be considered worthwhile to review the completed and compiled code of a set of components. In each case, the purpose is to check that the material under review correctly reflects the previous level and is adequate to allow the next level to proceed, and that agreed design or coding standards have been followed. As before, there may well be associated material to be reviewed covering, particularly, resource estimation, test, software user manual and project management.

At a TR/R, it is checked whether preparation for acceptance testing is complete. As noted above, it is very important that this be clearly defined early in the project as there are several possibilities. For example, if verification of software requirements is the basis for acceptance then it will be necessary to review the results of all corresponding software unit, integration and system tests, as well as analysis reports, as given in the verification control document; then, it might be decided to observe execution of selected tests, perhaps with modified data or, within previously agreed limits, to specify further tests. On the other hand, if verification of the user requirements is the basis for acceptance then it is likely that agreement on the test cases to be run will have been reached earlier in the project and that the developer will already have informally executed those cases. In this case, the informal runs can be reviewed, some or all can be selected for formal observation, and variations of test data, within previously agreed limits, can be specified.

The purpose of a TA/R is to review the results of the formal acceptance tests, and to decide whether or not the software should be accepted. It is likely that this review will also check whether all required software documentation is up to date and available for delivery.

6.2.2 Review Procedure

The basic procedure is that the material for review is delivered by its originator to the reviewers at an agreed date, some time in advance of the review meeting.

Then, the reviewers check whether the material meets the required objectives - the reviewers may work from a pre-defined checklist (see /10/, for example). Any discrepancies found are noted in an agreed form - these are often referred to as RIDs (Review Item Discrepancies) (again, see /10/).

Notes on all discrepancies are returned to the originator, sufficiently in advance of the review meeting to allow time for them to be examined and for answers to be prepared.

The purpose of the actual review meeting is to examine each noted discrepancy and to agree or otherwise on the given answer. The meeting should also agree on a schedule for repair of all discrepancies. Finally, the reviewers should decide whether authorisation to proceed to the next phase can be given - in the language of /17/ have the *transition criteria* been satisfied? Strictly and preferably, work should not commence on the next phase until such authorisation has been given.

6.3 Intermediate Review Mechanisms

6.3.1 Identification and Purpose

Participation is usually confined, in intermediate reviews, to members of the development team. However, the customer may - depending on the project - request visibility on the process. The purpose of such reviews is, in essence, the same as that for the formal reviews, namely to check whether performance of a particular task is sufficiently complete to enable its successor task to commence.

If the size of the project demands it, intermediate reviews should be made during the requirements analysis phase; for example, if a hierarchical decomposition of the functional requirements is made then intermediate reviews may be held at various stages of the decomposition. Similarly, in the architectural phase, reviews should be held at various levels of decomposition. In the detailed design phase, there should be a (peer) review of each component's design before coding is commenced. During coding, each compiled component should be the subject of a code inspection before submission for unit test.

In the same way associated activities, particularly design and specification of test cases and test harness preparation, should be the subject of intermediate review.

All formally delivered documentation associated with a software development should be the subject of a technical check. Then, it should be presented for agreement of quality assurance personnel and for approval by project management.

6.3.2 Procedures

There are various ways of performing intermediate reviews, and the most appropriate for a given project may depend on team size and members, available tools, resources and schedule.

As for the major reviews, the review item is distributed and then comments are returned to the author. It may or may not be decided to have a meeting but, in any case, some comment 'close-out' mechanism is necessary.

At the level of code review, it is commonly considered that meetings should take the form of code inspections or walkthroughs. In the latter case, the code is manually

exercised with pre-defined test cases. Clearly, this could be of benefit for other phases of the development also. It is noted that there are software tools available that will allow automatic inspection of code for adherence, for example, to pre-defined standards including adherence to limits on certain software metrics.

7. Quality and Configuration Management

7.1 Introduction

This work is not intended to give a comprehensive treatment of all aspects of software development. In particular, the topics of software quality assurance and configuration management are not covered in any depth. However, for completeness, a brief discussion of these is included.

7.2 Software Quality Assurance

It can be considered that the primary means of assuring software quality is by implementation of good practices throughout a development lifetime. This would include use of sound method at each stage of the work, use of appropriate tools, and scheduling of a series of formal and intermediate reviews.

It should be borne in mind that, in practice, there will be constraints due to resource limitations on, for example, the possible program of reviews - except in the case that the project has no cost and schedule constraints! (see section 8, following).

The issue arises as to how it is checked whether the planned good practices are, in fact, being adhered to. Whether, for example, the agreed standards for analysis, design or coding are being satisfied or whether all necessary aspects have been checked at each review. The actual work of checking is greatly facilitated by, for example, tracing matrices, by the clarity of output from carefully specified methods and associated software tools, and by the use of dedicated 'quality' tools such as code inspection tools. However, the question remains as to who has responsibility for the work of checking.

This question may be answered in a number of ways, depending on the overall policy of the software developer and on the needs of a particular project. One possibility is that the development team - technical management, requirement analysts, designers, coders, testers - have complete responsibility for all checking, subject only to customer participation at major reviews. A different possibility is that there are quality assurance personnel, quite independent of the development team, with this responsibility. There are different levels at which quality assurance personnel may participate and some of these are discussed below. However, it is quite important that there be a clear understanding from the outset as to the overall quality program for the project, as dictated by all the customer's requirements including those on cost and schedule, and on the powers and responsibilities of development and quality personnel.

Quality assurance personnel may, and frequently do, function in a non-technical, procedural manner. Thus, their function is to check, for example, that all documentation used and produced is at correct issue, that all entries are present in tracing matrices between phases, that agreed standards have been adhered to, that change control of documentation and software is properly managed, and that all

software deliveries are made in a proper, orderly way. It is apparent that some of these functions could and, so far as possible, should be performed by automatic tools.

A different level of involvement for quality assurance personnel would be if they were not only to check for presence of all entries in tracing matrices but also to check that these entries were technically correct. Somewhat similarly, one can distinguish between a quality assurance function to check that the verification control document contains a verification method and corresponding report reference for every software requirement, and a function to actually check the content of each report.

It is worth pointing out, regardless of the level of function, the key place of the verification control document in demonstrating quality, if quality is defined as 'satisfying the requirements'.

Finally, it could be envisaged to have a completely independent verification of the software - these issues are raised in /17/ but discussion of /17/ is outside the scope of the present document. In this case an independent team would, on the basis of the software requirements document (or possibly the user requirements document), prepare test data and associated test software in parallel to the actual software development. Then, at the end of the development, the actual software would be exercised with these tests. This approach could be thought of more as product rather than quality assurance.

7.3 Software Configuration Management

7.3.1 Documentation

It is essential that each piece of documentation produced during a project is uniquely identified. For example

<company acronym>.<project acronym>.<seq. no.>.<doc. type>.

It may be that the customer will define an identification system for all formally delivered documents; this was so for all 3 case studies as they were part of wider projects. It is almost inevitable that re-issues or revisions will be required for at least some of the documentation produced. Therefore, it is important that the changes from one issue or revision to the next be clearly described and marked, and that the updated document has a unique means of identification. For example

<company acronym>.<project acronym>.<seq. no.>.<doc. type>. Issue 1, Revision 3

Finally, it should be possible, at any time, to report the current status of all documents including their titles, identifiers, last (or planned) dates of issue, issue numbers and revision numbers.

These three subjects of identification, change control and status accounting are key elements of configuration management. Related to change control is another major process, namely that of problem handling. In fact, it is important that status accounting should cover not only all documentation produced by the software developer but also all documentation supplied to the developer (with note made of receipt dates).

7.3.2 Software

The same elements of identification, change control and status accounting apply equally to software as to documentation, as does that of problem reporting. The software configuration items are established during the architectural design phase (perhaps extending to the detailed design phase for low level components). The headers of configuration items may be used to record their change history; however, it may well be more effective and efficient to use a {configuration management, problem management} tool-set for such recording. The same status report as for documentation may be extended to include software, although the latter may entail more detail per configuration item.

An indication has been given in section 5.4.2, on test execution, of the central role of configuration management during that part of a project life-time. However, it may be borne in mind that a good configuration management system should not be intrusive and should, ideally, be transparent to its user.

The importance of configuration management for a software product which exists in several versions and for which there are several customers can hardly be over-stated.

It is emphasised that software tools for configuration management are extremely useful. In fact, they should be regarded as essential for all but the smallest projects or organisations.

Note: It is stressed that the above treatment is quite cursory as, for example, reference to the relevant chapter of /17/ will reveal.

8. Project Management

8.1 Introduction

The purpose of this chapter is to identify some of the key issues to be addressed in management of a software development project. The chapter has separate sections addressing project acquisition, project planning and initiation, management during the course of a project, cost and schedule. In particular, the schedule implications of periods of review are noted.

Note: Some of the terminology use here (as in the other chapters) may vary with circumstances but the basic concepts should still apply

8.2 Project Acquisition

A software developer's decision to acquire or attempt to acquire a particular project should be based on a general policy as to the kind of projects to be undertaken by the organisation and on a clear view of the organisation's future. This is of importance as a guide to consistency in staff recruitment and in acquisition of equipment.

Project acquisition may occur in several ways. For example, a company that produces commercial software for a general market might decide on a new product on the basis of market research or on customer feedback through its sales department. On the other hand, a user organisation might invite software developers to submit tenders to satisfy a perceived need. It is the latter category to which the three case studies belong and it is the assumed context of what follows. However, the same general principles apply regardless of how a project is acquired.

The central part of the invitation to tender issued by a user organisation is a statement of the work to be performed by the software developer. This consists not only of a definition of the technical requirements (possibly in a preliminary form) but also identifies any schedule constraints, and specifies any standards or practices to be adhered to.

In reply to the invitation to tender, a software developer prepares a proposal specifying how it is intended to meet the user's needs and at what cost. If there are any user needs that the proposal does not *comply* with then this should be stated and justified. It should be noted that a proposal is, in effect, a first iteration of a project plan.

The user organisation selects a 'winning' proposal from those submitted and, usually, there then follows a period of negotiation with the chosen software developer. The purpose of this negotiation is to finalise both the statement of work and the proposal.

8.3 Project Planning and Initiation

The basic step for planning a project, whether in proposal preparation or later, is derivation of a work breakdown structure (WBS). This is a process of decomposing the work to be performed into logical work packages (WP) in a top-down manner.

The decomposition process may have to be continued to a number of sub-levels. At the lowest level, each work package is split into a number of individual tasks.

For each work package, it is necessary to specify the individual responsible for its management and completion, its start date (also, the event which allows it to start), its end date (also, any constraints on its end date), the inputs it requires (for example, documents, software or equipment), the sub-packages it is composed of (for a parent work package) or its individual tasks (for a terminal work package), and the outputs it produces (for example, documents or software). It can be seen that the process of work breakdown is analogous to those of decomposition of functional requirements or of design decomposition and, as for those activities, it is essential to specify the interactions between components. In particular, all sequencing constraints between work packages must be clearly stated.

If a software development is planned in the terms defined by the preceding chapters then its work breakdown structure is automatic, at least in high-level outline. Thus, the top-level work packages might be project management, quality assurance, configuration management and engineering. Each of these could then be broken down further as necessary. In particular, engineering could be decomposed into analysis, synthesis, review and test. Then, analysis could be split into decomposition level 1, decomposition level 2 and so on, while synthesis could be split into architectural design, detailed design (perhaps with sub-packages) and coding (perhaps with sub-packages). Similarly, review could be split (see chapter 5) into UR/R, SR/R, AD/R, DD/R, TR/R and TA/R (with perhaps intermediate reviews also), while test would divide naturally into system, integration and unit components within each of which there would be planning, design, specification, harness preparation and execution sub-packages.

Having established a work breakdown structure it is necessary to form an estimate of the total cost of the project. Essentially, this means forming an estimate of the labour and other costs of terminal work packages, and then combining these to get the total estimate. The process of estimation is not a simple one - it is touched on again in section 8.5.

An essential part of the work breakdown process is allocation of responsibilities to individuals. Thus, it is at this stage that personnel selection takes place and that team structure is decided on. It is not intended to give guidelines on this matter except to emphasise the great advantages of having a committed and mutually supportive team, and the belief that one of project management's most important tasks is to facilitate this.

8.4 On-going Project Management

The main on-going project management activities include monitoring of the status of the work including comparison of actual against planned progress, modifying plans to take account of any problems arising or of new circumstances, and reporting to and liaising with the customer (user).

There are various methods and tools that may be used both for initial planning (see section 8.3) and for on-going monitoring and adjustment. The most basic, but still very useful, is a bar-chart showing each work package (or task) in calendar time, for example



Figure 8.4-1: Illustrative simple bar-chart (or Gantt chart)

This can be extended to indicate both planned and actual durations, inputs and outputs (or deliveries). However, care must be taken to avoid over-cluttered diagrams.

A major problem with bar-charts is that it is difficult to show inter-dependence of activities in a satisfactory way. This problem is overcome by the use of what are usually referred to as PERT/CPM techniques.

There are software tools available which support both bar-charts and PERT/CPM, as well as providing facilities to show, for example, the rate of labour spend as a function of time (allowing overloading to be identified), to report the work-load of each individual, and to build in holiday periods. The potential advantages of such tools are clear but there can be pit-falls, especially too much concentration on detail with consequent loss of overview. The onus remains on the planner to prepare a logical, consistent and meaningful work breakdown structure.

Remark: It *should* be possible (but often is not) for the planner to specify basic data and constraints (for example, on sequencing) to the tool in a simple way; the tool should then report the implications of these inputs to the planner by means of graphic and other output, thus allowing adjustments to be made and further detail to be added in an orderly way.

It will be seen from section 8.3 that a work breakdown structure and corresponding work package descriptions specify considerable detail. For example, in the suggested 'synthesis' work package there is a 'detailed design' sub-package which, in turn, could have sub-packages. For definiteness, suppose WP4 denotes 'engineering', WP4.2 denotes 'synthesis' and WP4.2.2 denotes 'detailed design'. Then, at a further level of decomposition, there could exist a work package WP4.2.2.3 whose description is,

WORK PACKAGE DESCRIPTION

WP4.2.2.3 Component family **X**
 Summary: Perform detailed design of **X**
 Start date: ... End date: ...
 .
 .
 .
 Tasks:
 WP4.2.2.3-1 Perform detailed design of **X**₁
 WP4.2.2.3-2 Perform detailed design of **X**₂
 .
 .
 .

Figure 8.4-2: Sample Work package description (fragment)

At the same time, in the design documentation, there will be a trace from each X_i to a corresponding requirement (say R_{yy}). Then, by bolding X_i as shown, one could envisage a search over both design and planning documents to locate all occurrences of X_i (cf section 4.2.2, above), thus connecting a planning task (WP4.2.2.3-1) to a software requirement (R_{yy}). More generally, it could be worthwhile to have such a scheme covering all project documentation in order to ensure a high level of consistency and cohesion.

8.5 Cost Considerations

It was noted in section 8.3 that the basic estimation procedure is to estimate the individual work packages separately and then to combine these estimates. The actual estimation of effort, especially labour, is rather inexact; it may, for example, be based on experience from previous similar work or from sample, calibrating exercises.

Quite often the estimation problem appears in a different way - an overall effort level is assigned to the project as a whole and is to be shared out between work packages. One then has the task of determining what can reasonably be achieved for the allocated level of effort.

It is important that the rate of expenditure of effort be as close to uniform as possible throughout a project. In the past, it was usually planned that by far the greatest effort would be spent during the implementation stages of a project. However, as has been pointed out in earlier chapters, there is now (or should be!) more emphasis on the requirements analysis phase. This, together with the diminution of effort in detailed design, coding and software integration phases (observed in case study B), permits a better overall spread to be achieved.

Work has been carried out on software project cost estimation, see /14/ for example. In /14/, a model is presented that allows the project effort to be estimated based on code size, while taking account of such factors as experience of personnel, nature of software, and kind of implementation language (for example, high level or assembler). This model would distinguish, for example, between the simulation

software of case study A and the critical, real-time software of case studies B and C. A comparison of A with B and C shows that, in the former, there was a relatively informal level of specification, review and test whereas, for the latter, the contrary was true. Clearly, this goes far in accounting for the considerable cost difference between the projects.

It is possible to observe areas where savings, in both cost and schedule, may be made by experienced, skilled and reliable personnel. Clearly, such personnel will be able to perform tasks more quickly and correctly than inexperienced staff. In turn, the program of intermediate reviews will run more smoothly and may, perhaps, be safely curtailed.

8.6 Schedule Considerations

Very often there are tight schedule constraints on software development. For example, in the cases studied, the software developments were constrained by the wider projects of which they were part. Similarly, one could envisage a tight schedule for development of a commercial software product in order to preserve competitiveness.

It is possible to make substantial savings on schedule in various ways. The most obvious, perhaps, is in the use of a modern high-level language supported by a good development environment. The re-use of software components, if feasible, also gives considerable savings.

It can be expected that the use of various software engineering tools - for analysis, design, test, quality and so on - will also result in shorter schedules as well as in better quality software. However, this will not be true unless the chosen tools are appropriate for the work and unless team members are sufficiently familiar with them (time may be needed for learning); otherwise, the tools could have an effect opposite to that intended.

As noted in section 8.5, deployment of experienced and skilled personnel will result in schedule reductions. However, the software developer must look beyond the current project and so may wish to have some inexperienced staff involved also.

Careful planning is needed in order to comply with a tight project schedule. Sometimes, it may be found that the schedule constraints are incompatible with technical or quality needs. In such cases, some compromise must be found between the technical or quality ideal and what is feasible - obviously, it would be preferable to have the schedule constraints relaxed if at all possible.

Schedule constraints may refer not only to the final software delivery date(s). There may also be constraints on the date(s) of receipt of inputs to the software development process. Most desirable is that the user requirement document be available at the start of the process but, as discussed for case studies B and C, this may not always be possible. In such cases, it is essential to have a clear strategy such as defined in section 5.1.

It is sometimes planned to have a degree of overlap between development phases. This should not pose difficulty subsequent to architectural design in that one could have (and, indeed, would expect to have) a given component or family of components at a different stage of development than another; this pre-supposes that the interface between components has been agreed at the AD/R and has been baselined. However, any overlapping of software requirement and architectural design phases, apart from informal feasibility studies, is very undesirable.

Thus, it is important to have the following successive and non-overlapping stages⁵:

- (A) Preparation of software requirements and associated documents
- (B) SR/R process
- (C) Preparation of architectural design and associated documents
- (D) AD/R process

On completion of (D), the detailed design process may be commenced.

There may be feedback to the main document from the process of preparation of the associated documents in (A) and (C). This should, ideally, be incorporated prior to delivery of the main document. Thus, it should be possible, and is certainly desirable, to deliver the main and associated documents at the same time.

A major disadvantage of phase overlapping is that the later phase is then based on an informal level of information and will have to be revised when the information is formalised. This could entail a considerable degree of configuration control. Moreover, it is very desirable that personnel from one phase be among the reviewers of the work of a neighbouring phase - this may be impossible with overlap.

At the end of a project, the following successive and non-overlapping stages are identified,

- (W) TR/R process
- (X) Performance of acceptance tests
- (Y) TA/R process
- (Z) Delivery

On the basis of the foregoing, one can conclude that legitimate savings may be made in schedule in a variety of ways. However, there are certain components of schedule which are, or which should be, irreducible. These components are highlighted in the following example where, for simplicity, a single DD/R is assumed.

Example: Let the following successive and non-overlapping stages have the indicated durations:

⁵ At least, justification should be made if these stages are not possible and corresponding risk mitigation measures should be taken.

STAGE	DURATION
(A) Preparation of software requirements and associate documents	w_1
(B) SR/R process	r_1
(C) Preparation of architectural design and associate documents	w_2
(D) AD/R process	r_2
(E) Preparation of detailed design and associate documents	w_3
(F) DD/R process	r_3
(G) Coding and test (unit, integration, system) processes	w_4
(H) TR/R process	r_4
(I) Performance of acceptance tests	w_5
(J) TA/R process	r_5
(K) Delivery	d

Figure 8.6-1: Example of project work stages and reviews

Thus, one has, made up of technical work + review + delivery,

$$\text{Total time} = \sum w_j + \sum r_j + d \quad (8.6-1)$$

Usually, review processes (B,D,F,H) consist of successive steps,

- (i) reviewers prepare comments on review items
- (ii) authors prepare replies to comments
- (iii) reviewers and authors meet
- (iv) authors make agreed updates

Typical values (by no means over-estimates) for (i), (ii), (iii) and (iv) would be 2.5, 1, 0.5 and 2 weeks, respectively. This yields $w_j = 6$ weeks for $j = 1, 4$. Steps (i) and (ii) might be omitted for review process J so that it would require 2.5 weeks. Hence, (8.6-1) evaluates to,

$$\text{Total time} = \sum w_j + 26.5 \text{ weeks} + d \quad (8.6-2)$$

The intermediate review component of w_j ($j=1,5$) could be included in a more detailed and realistic analysis.

References

1. Tuohey, W.G., Campbell, J.G., O'Mongain, E., Gardelle, J.P., Wills, R.D., *In-Orbit Scientific Calibration of Hipparcos*, ESA Journal, Vol.11, No.1, 1-17 (1987)
2. Beck, J.V., Arnold, K.J., *Parameter Estimation in Engineering and Science*, John Wiley and Sons (1977)
3. Ximenez de Ferran, S., *The ISO Spacecraft*, ESA Bulletin, No.67, 17-24 (1991)
4. Wiggers, J.J.M., Spruit, M.E.M., *Use of Ada for the ISO AOCS: the User's Viewpoint*, Ada Europe Conference (1991)
5. Birrell, M.D., Ould, M.A., *A Practical Handbook for Software Development*, Cambridge University Press (1988)
6. Ward, P.T., Mellor, S.J., *Structured Development for Real-Time Systems - Volumes I, II and III*, Prentice-Hall, (1985)
7. Robinson, P.J., *Hierarchical Object Oriented Design*, Prentice-Hall, (1994)
8. Wenzel, K.-P., Domingo, V., Schmidt, R., *The Solar- Terrestrial Science Programme*, ESA Bulletin, No.50, 8-16 (1987)
9. Bates, W.J., *Samuel Johnson*, The Hogarth Press (1984)
10. *ESA Software Engineering Standards*, ESA PSS-05-0, Issue 2, ESA Publications Division (1991)
11. Stevens, R.J., *Some Considerations on Organising Requirements for Systems Management*, ESA Journal, Vol. 15, No.1, 35-48 (1991)
12. Myers, G.J., *The Art of Software Testing*, Wiley-Interscience (1979)
13. *Special Section on Requirements Engineering*, IEEE Trans. Software Eng., Vol.17, No.3, 210-258 (1991)
14. Boehm, B., *Software Engineering Economics*, Prentice_Hall (1981)
15. Kessler, M.F., Clavel, J., Faelker, J., *Looking back at ISO operations*, ESA Bulletin, No. 95, 87-97 (1998)
16. Vandenbussche, F.C., *SOHO's Recovery - An Unprecedented Success Story*, ESA Bulletin, No. 97, 39-47 (1999)
17. *Software considerations in airborne systems and equipment certification*, RTCA/DO-178B (1992)