BSSC(98)1 Issue 1
March 1998

Guide to applying the ESA software engineering standards to projects using Object-Oriented Methods

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

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

DOCUMENT STATUS SHEET

DOCUMENT STATUS SHEET			
1. DOCUMENT TITLE: BSSC(98)1			
2. ISSUE	3. REVISION	4. DATE	5. REASON FOR CHANGE
1	0	1998	

Approved, March 10th 1998
Board for Software Standardisation and Control
M. Jones and U. Mortensen, BSSC co-chairmen

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION 1
1.1 PURPOSE 1
1.2 OVERVIEW 1

CHAPTER 2 OBJECT-ORIENTED METHODS 3
2.1 INTRODUCTION 3
2.2 WHAT IS AN OBJECT-ORIENTED METHOD? 3
2.3 WHY USE OBJECT-ORIENTED METHODS? 4

CHAPTER 3 USING OBJECT-ORIENTED METHODS WITH PSS-05 7
3.1 INTRODUCTION 7
3.2 THE SOFTWARE LIFE CYCLE 8
3.3 USER REQUIREMENTS DEFINITION PHASE 8
3.4 SOFTWARE REQUIREMENTS DEFINITION PHASE 9
3.4.1 Construction of the logical model 9
3.4.2 Specification of the software requirements 11
3.5 ARCHITECTURAL DESIGN PHASE 12
3.5.1 Construction of the physical model 12
3.5.2 Reuse 13
3.5.3 Specification of the architectural design 14
3.5.3 Design evaluation criteria 15
3.6 DETAILED DESIGN AND PRODUCTION PHASE 16
3.6.1 Detailed design 16
3.6.2 Coding 16
3.6.3 Code evaluation criteria 17
3.6.4 Testing 17
3.6.4.1 Unit testing 17
3.6.4.2 Integration testing 18
3.6.4.3 System testing 18
3.7 TRANSFER PHASE 18
3.8 OPERATIONS AND MAINTENANCE PHASE 18

CHAPTER 4 OBJECT-ORIENTED METHODS MAPPING 19
4.1 INTRODUCTION 19
4.2 OMT 19
4.2.1 SR Phase 19
4.2.2 AD and DD phases 21
4.3 BOOCH 22
4.3.1 SR phase 23
4.3.2 AD and DD phases 23

4.4 OOSE 23
4.4.1 SR phase 24
4.4.2 AD phase 25
4.4.3 DD phase 25
4.5 UNIFIED MODELING LANGUAGE 25
4.6 COAD-YOURDON 26
4.6.1 SR phase 27
4.6.2 AD and DD phases 27
4.7 SHLAER-MELLOR 29
4.7.1 SR phase 29
4.7.2 AD and DD phases 30
4.8 HOOD 31
4.8.1 SR phase 31
4.8.2 AD and DD phases 31

APPENDIX A GLOSSARY A-1
APPENDIX B REFERENCES B-1
APPENDIX C PRACTICE GUIDELINES C-1
APPENDIX D DOCUMENT TEMPLATES D-1
APPENDIX E SMALL SOFTWARE PROJECTS E-1
APPENDIX F DOCUMENT EXAMPLES F-1

PREFACE

The ESA Software Engineering Standards, ESA PSS-05-0, define the software practises that must be applied in all the Agency's projects. As object-oriented methods are commonly used for developing software, this guide describes how to use them when the ESA Software Engineering Standards are applied.

The guidelines are based upon the experience of developing custom space system software using object-oriented methods. However much of the guidance is likely to be relevant for the development of other types of application.

One of the conclusions of the experience is that object-oriented methods can be used effectively when the ESA Software Engineering Standards are applied. Object-oriented methods can be used in combination with any life cycle approach such as waterfall, incremental and evolutionary. The use of object-oriented methods mainly affects the activities and products of analysis, design and coding.

The following BSSC members have contributed to the production of this guide: Michael Jones (co-chairman), Uffe Mortensen (co-chairman), Gianfranco Alvisi, Bryan Melton, Daniel de Pablo and Lothar Winzer. The following members of the BSSC Working Group on Object-Oriented methods have also contributed to the production of this guide: Per Hemsö, Christoph Kröll, Andrea Baldi, Yvon-Marc Bourguignon, Jorge Amador and Patricia Rodrigues. The BSSC wishes to thank Jon Fairclough for contributing to and editing the guide. The authors wish to thank all the people who contributed ideas for this guide.

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

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

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

This page is intentionally left blank

CHAPTER 1
INTRODUCTION

1.1 PURPOSE

ESA PSS-05-0 describes the software engineering standards to be applied for all deliverable software implemented for the European Space Agency (ESA) [Ref. 1].

This document provides software development organisations and software project managers with guidelines on the use of Object-Oriented (OO) methods in projects applying the ESA Software Engineering Standards. The guidelines apply to all object-oriented methods. Familiarity with object-oriented methods is not required to understand this document.

Additional guidelines on the application of the ESA Software Engineering Standards are provided in the series of documents described in ESA PSS-05-01, Guide to the Software Engineering Standards '[Ref. 2 to 12] and the BSSC Guide to Applying the ESA Software Engineering Standards to Small Software Projects [Ref. 22]. The guidance in this document takes precedence over that in the additional guides.

1.2 OVERVIEW

Chapter 2 provides an overview of object-oriented methods. Chapter 3 describes the software life cycle in terms of the best practices in object-oriented development. Chapter 4 discusses how the main object-oriented methods are mapped into the ESA PSS-05-0 life cycle. Appendix A contains a glossary of acronyms and abbreviations. Appendix B contains a list of references. Appendix C contains guidelines for applying the practices for the SR, AD and DD phases. Appendix D contains document templates for the SRD, ADD and DDD. Appendix E contains additional guidance for small software projects using object-oriented methods. Appendix F contains some examples of documents from large and small OO projects.

This page is intentionally left blank

CHAPTER 2
OBJECT-ORIENTED METHODS

2.1 INTRODUCTION

What are object-oriented methods? Why should they be used? This section gives brief answers to these key questions.

2.2 WHAT IS AN OBJECT-ORIENTED METHOD?

Object-oriented methods originated in the Simula67 and Smalltalk programming languages, and the techniques of structured analysis, particularly data modelling. Object-oriented methods differ from structured methods by:

- starting development with the identification of the objects in the problem domain that the software will store and process information about
- building an object model first, instead of a functional model (represented in data flow diagrams) or a data model (represented in a data dictionary and entity- relationship diagrams)
- integrating functions and data in objects, instead of separating them between the data model and the functional model.

Object-oriented methods support one or more of the analysis, design and programming activities of software development. All object-oriented methods have the following characteristics:

- the software is composed of objects that have defined attributes and operations; an attribute describes a characteristic of an object (e.g. size); an operation (also called method, service or function) describes an action that an object can do (e.g. ChangeSize);
- objects encapsulate data that can only be accessed by means of the objects operations;
- classes of objects can be defined; classes have attributes and operations; objects are instances of classes;
- classes can inherit attributes and operations from superclasses (also called abstract classes);

- the structure of the software is described in an 'object model', which identifies the classes and their relationships, such as inheritance and 'aggregation '(i.e. composition).

The main object-oriented analysis and design methods are the:

- Rumbaugh et al Object Modeling Technique (OMT)
- Booch method
- Object-Oriented Software Engineering (OOSE) method
- Unified Modeling Language (UML)
- Coad-Yourdon method
- Shlaer-Mellor method
- HOOD method.

These methods are discussed in Chapter 4. UML has resulted from the recent amalgamation of the OMT, Booch and OOSE methods by Rumbaugh, Booch and Jacobson and is intended to replace them. UML is expected to become the industry-standard modeling language very soon.

The main object-oriented programming languages are:

- C++
- Smalltalk
- Ada
- Java
- Objective C
- Object Pascal
- Eiffel.

Projects using object-oriented methods may be recognised by the use of the analysis and design methods and programming languages identified above.

2.3 WHY USE OBJECT-ORIENTED METHODS?

The factors that should be considered when deciding whether to use object-oriented methods are presented below. They are classified according to whether they favour the use of object-oriented methods.

The factors that favour the use of object-oriented methods are:

- the easy transitions from analysis, through design, to programming, promotes consistency, understandability and fewer errors;
- the ease of modifying the system - a structure based upon a functional hierarchy will often require restructuring to implement a new function; a structure based upon an object hierarchy is less likely to require restructuring to implement a new function;
- the ease of reuse - classes may be reused either as they are or by making new classes that inherit from existing classes;
- software with rich functionality is made easier to implement because of the ability to inherit code;
- software can be made more reliable and maintainable because information hiding (i.e. encapsulation) is enforced;
- software can be made more usable when the design of the software is driven by use cases or real-world scenarios involving the system and users;
- software can be made more compact by means of the inheritance feature; less application code is required to deliver the same functionality compared with procedural languages.

The factors that may cause problems when using object-oriented methods are:

- difficulty - creating an object model right first time can be difficult, and considerable iteration of the model is often required during development; this has caused problems when object-oriented methods have been used within the waterfall approach;
- understandability - object models with deep inheritance trees containing many classes, each with multiple relationships, attributes and operations, are complex entities, which can make the software more difficult to understand and maintain;
- lack of expression of functionality - users are interested in what the software will do, i.e. functionality; although functional modelling is allowed in some object-oriented methods, it is rarely done; this problem can be solved by means of natural language explanations of the functionality, or the use of structured analysis data flow modelling techniques, or the employment of the use case technique of OOSE [Ref. 19];

- fragility - changes to classes upon which others depend (i.e superclasses) can force changes to the dependent (sub)classes. This problem is commonly called the 'fragile base class problem".

CHAPTER 3
USING OBJECT-ORIENTED METHODS WITH PSS-05

3.1 INTRODUCTION

ESA PSS-05-0 defines a software life cycle model consisting of the following phases:

- user requirements definition phase
- software requirements definition phase
- architectural design phase
- detailed design and production phase
- transfer phase
- operations and maintenance phase.

This chapter:

- discusses the application of object-oriented methods in each of the life cycle phases
- discusses issues common to all the methods
- provides a description of life cycle activities based upon the Object-Oriented Software Engineering (OOSE) method of Jacobson and common practices in space systems projects.

Object-oriented methods may be mixed with other analysis and design methods, for example:

- Structured Analysis and Design Technique (SADT) in the SR phase
- Object Modelling Technique (OMT) in the AD and DD phases
- However, mixing is not recommended because of the increased effort required to transform the outputs of one method into those of another.

The mixing of object-oriented methods, for example Coad-Yourdon and OMT, is common practice, as no single method contains all the techniques needed to analyse, design and program most systems. The recently released Unified Modelling Language (UML) is said to contain a complete set of techniques.

3.2 THE SOFTWARE LIFE CYCLE

A phased approach to development is fundamental to all the ESA PSS-05-0 life cycle approaches: waterfall, incremental and evolutionary. A phased approach must be used when object-oriented methods are applied, because it is essential for:

- evaluating progress
- ensuring communication of progress
- ensuring consensus on the next steps
- avoiding waste of resources.

When applying object-oriented methods, it is sometimes necessary to iterate models and code to optimise the quality of the software. ESA PSS-05-0 allows iteration both within a phase and between phases. Models and code can be revised at any stage of the life cycle to correct them, perfect them, or adapt them to new requirements. However once models and code have become part of a baseline, changes to them must be controlled to ensure the integrity of the software.

3.3 USER REQUIREMENTS DEFINITION PHASE

The user requirements definition phase produces the User Requirements Document (URD). This is the primary input to object-oriented analysis.

Although object-oriented methods are not intended to be used in the UR phase, requirements capture and definition may benefit from the application of the use case technique (see Section 3.4).

Users may include requirements on software reuse in the URD. These may specify how the software to be developed is to be reused.

3.4 SOFTWARE REQUIREMENTS DEFINITION PHASE

3.4.1 Construction of the logical model

The developer should produce an implementation-independent logical model that consists of:

- a requirements model
- an analysis model.

The requirements model aims to capture the functional requirements for the system, and describes the:

- actors, which model the external entities (e.g. users) outside the system
- use cases, which model the transactions between the actors and the system
- external interfaces between the actors and the system
- problem domain objects, which are the entities in the real world that the system is concerned with.

In an airline scheduling system, examples of actors are 'pilots' and 'clerks'. Examples of use cases are 'acknowledge flight', 'check schedule' and 'confirm booking'. The screens and menus presented to the clerks when confirming the booking are examples of external interfaces. Examples of problem domain objects are 'aircraft' and 'airport'.

Use cases should be named in the declarative style appropriate for functional requirements. An external interface description should accompany each use case, and outline how the system appears to the actors. Only the name, attributes and static relationships (e.g. aggregation and inheritance relationships) of each problem domain object should be defined in the requirements model.

The analysis model aims to give the system a robust and adaptable class[1] structure. The analysis model consists of:

- interface classes
- entity classes

[1] Following UML and common practice, we use the term 'class' instead of 'object' (note that Jacobson uses the word 'object' throughout).

- control classes.

Entity classes hold system information. They correspond to the problem domain classes. Interface classes handle transactions between the system and the external world. Control classes co-ordinate the behaviour of the other classes.

The analysis model is produced by partitioning each use case so that:

- use case functionality that is directly dependent on the system interface is allocated to interface classes
- use case functionality that is concerned with storage and handling of information is allocated to entity classes (unless it is already in an interface class)
- the behaviour of the system in response to the use case is controlled by control classes.

The analysis model should be optimised for robustness against change. This may involve defining inheritance relations between classes.

Logical models of systems should be broken down into 'subject areas'[2]. A subject area is a collection of classes related to a particular view of the problem (e.g. mechanical view) or part of the system (e.g. user display). Subject areas should be used to reduce complexity and improve understandability.

The logical model should be described in diagrams using tools. Each diagram should be accompanied with a textual explanation. The OMT/UML modelling notation is very suitable for describing the logical model.

The completion criterion for the modelling activity is 'all capability requirements are traced to classes'.

---

[2] Also called 'class categories'. The terms 'subsystem' and 'package' are often used synonymously with subject area, or used to describe part of a subject area. It is recommended that the terms subsystem and package be used to describe parts of the design.

3.4.2 Specification of the software requirements

The SRD should describe:

- actors
- use cases
- external interfaces
- subject areas
- classes
- class attributes
- class associations
- the tracing of user requirements to use cases

Class attributes are only described. Definition of their data types should be done in the design phases.

The definitions of class operations are not required in the SRD because they are likely to change in the design phases when the class interactions are defined. However class operations may be included in the class descriptions in the SRD to clarify the purpose of the classes. Class operation argument lists should not be put in the SRD.

The model description in the SRD should describe the classes, their attributes and relationships. The functional requirements should be described in terms of use cases. The interface requirements should describe the external interfaces. The detailed mapping of this information into the ESA PSS-05-0 SRD template is described in Appendix D.2.

Specific requirements for software reuse may be included in the SRD. These may specify libraries or off-the-shelf software packages that are to be used.

3.5 ARCHITECTURAL DESIGN PHASE

3.5.1 Construction of the physical model

The developer should produce a physical model that consists of descriptions of:

- subsystems
- classes[3]
- interactions between classes.

The subsystems are derived from the subject areas of the logical model. The classes are derived from the classes in the logical model. Developers should aim to preserve the structure of the logical model in the physical model because this makes the design more robust to change.

The interaction descriptions define the control and data flow sequences between classes. There should be one interaction description for each use case.

The physical model should be described in diagrams using tools. Each diagram should be accompanied with a textual explanation. The OMT/UML modelling notation is very suitable for describing the physical model.

The completion criterion for the physical modelling activity is 'class interactions for all use cases are defined'. It is only required to define sufficient classes to demonstrate that the design is feasible, will meet the requirements and implement the use cases.

---

[3] Jacobson calls a physical model class a 'block'

3.5.2 Reuse

During design, developers should investigate the possibility of reusing software. Possibilities include:

- internal reuse
- class libraries or toolkits
- frameworks
- design patterns.

Internal reuse involves designing an application so that duplication of code is minimised. This is called factoring in structured design. In object-oriented design, objects use the operations of other objects wherever possible.

A class library or toolkit is a set of related classes that provide useful functionality, without constraining the higher level design of the application. Examples are utilities for handling strings, vectors, lists and stacks. Class libraries and toolkits correspond to the 'utility' libraries of conventional programming.

A 'framework' is a set of co-operating classes that make up a reusable design for a specific class of software [24]. A framework is a skeleton around which a system, subsystem or class may be constructed. The framework is customised by creating application-specific subclasses of superclasses from the framework. A framework may also contain class libraries or toolkits for providing services to the framework superclasses and application-specific subclasses. Class libraries/toolkits or frameworks may be Commercial Off-The-Shelf (COTS) software.

A design pattern is a description of communicating objects and classes that are customised to solve a general design problem in a specific context [25]. Design patterns can be used in a wide variety of application domains. An example of a design pattern is the Model/View/Controller for interface construction.

A design pattern differs from a framework in a number of ways:

- a design pattern is a model of a design and has to be adapted for each application; whereas a framework can be used 'as-is'
- a design pattern consists of schematic code, whereas a framework consists of working code
- design patterns are usually smaller than frameworks

- design patterns are less specialised than frameworks.

3.5.3 Specification of the architectural design

The ADD should describe the:

- subsystem functions and interfaces
- class attributes
- class operations
- class associations
- class interactions.

The ADD decomposition description should describe diagrammatically the subsystems, subsystem interfaces, classes and their associations. Interactions should be described for each use case. The descriptions should be arranged subsystem-by-subsystem and define the classes, attributes, operations and associations (in textual form).

The detailed mapping of this information into the ESA PSS-05-0 ADD template is described in Appendix D.3. See also the example in Appendix F.2.2. Note that ESA PSS-05-0 uses the word 'component' to describe any part of the software. In the object-oriented view, the term 'component' is normally applied to a reusable building block out of which the system is made, in analogy with hardware systems. The components exist when the system is defined. However in ESA PSS-05-0, a component should be understood to be a subsystem or class.

The ADD should be as small as possible, and should be a single document. However when the ADD cannot be contained within a few hundred pages it should broken up into a set of documents, one for the whole system and one for each subsystem. The subsystem ADDs may be elaborated into the DDD volumes in the DD phase.

3.5.3 Design evaluation criteria

The following characteristics should be evaluated:

- size
- degree of reuse
- quality.

The following metrics may be used to evaluate the size of an object-oriented system:

- number of subsystems
- number of classes
- total number of operations.

The following metrics may be used to evaluate the degree of reuse of pre-existing software in an object-oriented system:

- number of subsystems reused
- number of subsystems newly developed
- number of classes reused
- number of classes newly developed
- number of operations reused
- number of operations newly developed.

The quality of an object-oriented design may be evaluated from:

- the number, depth and width of the inheritance hierarchies (hierarchies that are shallow and wide, or deep and narrow, indicate poor classification)
- the number of classes inheriting a specific operation (a high number indicates a compact, efficient design)
- the number of classes that a specific class is dependent upon (a low number indicates low coupling, which indicates good design).

3.6 DETAILED DESIGN AND PRODUCTION PHASE

3.6.1 Detailed design

The classes, attributes, operations, associations and interactions defined in the ADD are refined. New classes, attributes, operations, associations and interactions may be defined.

The DDD should be organised subsystem-by-subsystem and provide descriptions of the:

- class attributes
- class operations
- class associations
- class interactions.

The detailed mapping of this information into the ESA PSS-05-0 DDD template is described in Appendix D.4.

The modelling tools used for design should support the conversion of models to source code and vice-versa (round trip engineering), to ensure that models and code are kept consistent. Tools should also be provided to generate detailed design documentation from the model. The development environment should allow the update of the model, the design documents and the source code from a single point.

Reused software should be analysed with the objective of identifying any design assumptions that are no longer valid. The reused software may need to be modified.

3.6.2 Coding

The classes should be implemented in an object-oriented programming language according to an agreed coding standard. An object-oriented programming environment should be used that:

- provides a screen-based editor, compiler, debugger and linker
- includes a class browser to enable navigation through the code
- include analysis tools to check the memory allocation and performance of the code.

Tables of class attributes and services may be automatically generated from the code and used to maintain the DDD.

3.6.3 Code evaluation criteria

The quality of object-oriented code may be evaluated from:

- the cohesion of the operations (every step of a cohesive operation will contribute to its function)
- length of operations in statements
- the number of branches in the code in each operation (i.e. cyclomatic complexity - 1).

The object-oriented design quality metrics described in section 3.5.3 are also relevant to the code.

3.6.4 Testing

3.6.4.1 Unit testing

A unit is a class or subsystem. Unit testing should verify each subsystem by adding classes one at a time, or in small groups, to the subsystem and testing the subsystem after each addition. For each addition/modification, the unit tests should exercise the classes that have just been added/modified, and also check for side-effects.

When unit testing code written using an object-oriented programming language, test cases should be included to verify that:

- objects can be created, initialised and deleted (i.e. constructors and destructors work)
- all class operations work as specified (black-box and white box tests cases should be designed to verify each operation)
- the order of execution of class operations follows the specification, or can work in any order, or handle out-of-order occurrences.

Reused application software should be unit tested if:

- it is to be used in a safety-critical application or
- it has been modified for the new application.

ESA PSS-05-0 requires that every statement be executed during testing (DD06). This is called the 'statement coverage requirement'. A more demanding requirement that is sometimes imposed is the 'branch coverage requirement', which specifies that every branch of the code be covered in testing. The inheritance and polymorphism features reduce the number of

statements and branches in the code. While this makes it easier to achieve statement and branch coverage in testing, additional tests [Ref. 19] may be needed to verify that:

- subclasses work correctly when the code of a class changes
- an operation works upon various types.

3.6.4.2 Integration testing

Software integration and integration testing should be performed use case by use case (corresponding to the function-by-function method described in ESA PSS-05-0). Black box integration tests verify the use cases as defined in the SRD. White box integration tests verify the class interactions specified in the ADD.

3.6.4.3 System testing

System testing should verify all the use cases. Use cases may be executed in parallel.

3.7 TRANSFER PHASE

The acceptance tests should validate that the system implements all the use cases.

3.8 OPERATIONS AND MAINTENANCE PHASE

The analysis, design and coding of changes to software built using object-oriented methods should be carried out using the same methods. Tools should be used that keep the models and code consistent.

CHAPTER 4
OBJECT-ORIENTED METHODS MAPPING

4.1 INTRODUCTION

The leading object-oriented analysis and design methods are the:

- Object Modeling Technique (OMT)
- Booch method
- Object-Oriented Software Engineering (OOSE) method
- Unified Modeling Language (UML)
- Coad-Yourdon method
- Shlaer-Mellor method
- HOOD method

Developers using object-oriented methods may need to mix techniques from different methods.

4.2 OMT

The strengths of Rumbaugh et al's Object Modelling Technique (OMT) [Ref. 16] are its simple yet powerful notation capabilities and its maturity. It was applied in several projects by its authors before it was published. The main weakness is the lack of techniques for integrating the object, dynamic and functional models. OMT has now been superseded by UML. The UML notation is based largely upon that of OMT.

4.2.1 SR Phase

OMT transforms the users' problem statement (such as that documented in a User Requirement Document) into three models:

- object model
- dynamic model
- functional model.

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

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

- identify objects
- identify classes of objects
- identify associations (i.e. relationships) between objects
- identify object attributes
- use inheritance to organise and simplify class structure
- organise tightly coupled classes and associations into modules
- supply brief textual descriptions on each object.

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

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

- identify sequences of events in the problem domain and document them in 'event traces'
- build a state-transition diagram for each object that is affected by the events, showing the messages that flow, actions that are performed and object state changes that take place when events occur.

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

- identify input and output values that the system receives and produces
- construct data flow diagrams showing how the output values are computed from the input values
- identify objects that are used as 'data stores'
- identify the object operations that comprise each process.

The functional model is synthesised from object operations, rather than decomposed from a top level function. The operations of objects may be defined at any stage in modelling. The functional model can be useful for explaining how the required functionality is delivered, and for tracing operations to functional requirements. However construction of the functional model has often been omitted, and this has often resulted in the logical model being hard to understand.

4.2.2 AD and DD phases

OMT contains two design activities:

- system design
- object design.

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

The steps of system design are conventional and are:

- organise the system into subsystems and arrange them in layers and partitions[4]
- identify concurrency inherent in the problem
- allocate subsystems to processors
- define the data management implementation strategy[5]
- identify shared resources and define the mechanism for controlling access to them
- choose an approach to implementing the control flow
- consider initialisation, termination and failure conditions
- establish trade-off priorities between constraints[6].

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

- batch transformation - a data transformation performed once on an entire input set
- continuous transformation - a data transformation performed continuously as inputs change
- interactive interface - a system dominated by external interactions

---

[4] Partitions organise subsystems in different layers into groups.

[5] The data management strategy defines the approach to handling persistent data, for example by means of a database management system.

[6] For example, if the available memory is limited, the system designers may specify that memory should be conserved at the expense of other characteristics, such as performance.

- dynamic simulation - a system that simulates evolving real world objects
- real-time system - a system dominated by strict timing constraints
- transaction manager - a system concerned with storing and updating data.

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

4.3 BOOCH

Booch originated object-oriented design, and continues to play a leading role in the development of the method [Ref. 15, 20]. His books on object-oriented methods have been described by Stroustrup, the inventor of C++, as the only books worth reading on the subject. This compliment reflects the many insights into good analysis and design practise in his writings. However Booch's notation is cumbersome and few tools are available. Booch now recommends the use of UML, of which he is a co-author.

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

- identify the classes and objects at a given level of abstraction
- identify the attributes and operations of these classes and objects
- identify the relationships among these classes and objects
- implement the classes and objects.

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

Booch advises that all modelling activities be combined into a single 'modelling phase'. This is also recommended by ESA for a small software project [Ref. 22].

4.3.1 SR phase

Booch provides four diagramming techniques for documenting the logical view:

- class diagrams, which are used to show the existence of classes and their relationships
- object diagrams, which are used to show the existence of objects and their behaviour, especially with regard to message communication
- state-transition diagrams, which show the possible states of each class, and the events that cause transitions from one state to another
- timing diagrams, which show the sequence of the objects 'operations.

4.3.2 AD and DD phases

Booch provides two diagramming techniques for documenting the physical view:

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

4.4 OOSE

Jacobson et al's Object-Oriented Software Engineering (OOSE) [Ref. 19] method describes a use case driven approach to software development that maps directly to the user requirements driven life cycle model of ESA PSS-05-0. Three OOSE processes are relevant to the ESA PSS-05-0 life cycle:

- analysis, performed in the SR phase
- construction, performed in the AD and DD phases
- testing, performed in the DD phase.

4.4.1 SR phase

Analysis results in:

- requirements model
- analysis model.

The requirements model aims to capture the functional requirements for the system. The requirements model consists of:

- actors
- use cases
- interface descriptions
- problem domain objects.

See section 3.4 for further detail on the requirements model.

The analysis model aims to give the system a robust and adaptable object structure. The analysis model consists of:

- interface objects
- entity objects
- control objects.

The analysis model is produced by partitioning each use case so that:

- use case functionality that is directly dependent on the system interface is allocated to interface objects
- use case functionality that is concerned with storage and handling of information is allocated to entity objects (unless it is already in an interface object)
- the behaviour of the system in response to the use case is controlled by control objects.

The analysis model should be optimised for robustness against change. This may involve defining inheritance relations between classes.

Entity objects usually correspond to objects in the problem domain. OOSE differs from Coad and Yourdon and OMT by recommending the definition of interface and control objects in analysis. Jacobson argues that this allows OOSE analysis models to be more robust.

The requirements and the analysis model constitute the logical model of the system.

4.4.2 AD phase

In the AD phase, construction produces a design model from the analysis model. This is done by defining:

- defining physical objects, called 'blocks', for each object in the analysis model and then
- defining the control and data flow between blocks in interaction diagrams.

Ideally the correspondence should be one object to one block. However implementation considerations may make the correspondence one to many.

Control and data flow sequences between blocks are defined for each use case.

4.4.3 DD phase

In the DD phase, construction produces an implementation model from the design model. The implementation model is composed of the source code, and is produced by writing the code for the blocks.

The testing process then tests the implementation model to produce the test model, i.e. the verified source code. This involves and integrating and testing the system use case by use case (corresponding to the function-by-function method described in ESA PSS-05-0).

4.5 UNIFIED MODELING LANGUAGE

The Unified Modelling Language (UML) is a language for specifying, constructing, visualising, and documenting the artefacts of software intensive systems. UML has resulted from the amalgamation of the Object Modeling Technique (OMT) [Ref. 16], Booch [Ref. 15] and Jacobson methods [Ref. 19].

UML focuses on a standard modeling language, not a standard process. The UML language notation is based upon that of OMT. The authors of UML recommend a use-case driven, architecture-centric, iterative and incremental development process, i.e. an approach similar to that of

OOSE. Further the UML specification does define a mapping to OOSE. Users of UML should therefore adopt the guidance in Section 4.4.

The Unified Modeling Language contains the following techniques:

- use-case diagrams, which are similar in appearance to those in OOSE;
- class diagrams, which are a melding of OMT, Booch, class diagrams of most other OO methods; process-specific extensions (e.g., stereotypes and their corresponding icons) can be defined for various diagrams to support other modeling styles;
- state diagrams, which are substantially based on the Harel diagrams of OMT with minor modifications;
- activity diagrams, which are similar to the work flow diagrams developed by many sources including many pre-OO sources;
- sequence diagrams, which are found in a variety of OO methods under a variety of names such as 'interaction', 'message trace', and 'event trace' and date to pre-OO days;
- collaboration diagrams, which were adapted from Booch ('object diagram'), Fusion [Ref. 23] ('object interaction graph'), and a number of other sources;
- implementation diagrams ('component and deployment diagrams') are derived from Booch's module and process diagrams, but they are now component-centred, rather than module-centred and are far better interconnected;
- stereotypes are one of the extension mechanisms and extend the semantics of the UML metamodel; user-defined icons can be associated with given stereotypes for tailoring the UML to specific processes.

4.6 COAD-YOURDON

The strengths of Coad and Yourdon's method are its brief, concise description and its use of general texts as sources of definitions, so that the definitions fit common sense and jargon is minimised. The main weakness of the method is its graphical notation, which is difficult to use without tool support. A proven solution to this problem is to use the method with the OMT notation.

4.6.1 SR phase

Coad and Yourdon [Ref. 13] describe an Object-Oriented Analysis method based on five major activities:

- finding classes and objects
- identifying structures
- identifying subjects
- defining attributes
- defining services.

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

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

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

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

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

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

4.6.2 AD and DD phases

Coad and Yourdon have published an integrated approach to object-oriented analysis and design [Ref. 14].

An object-oriented design is constructed from four components:

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

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

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

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

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

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

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

4.7 SHLAER-MELLOR

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

4.7.1 SR phase

Shlaer and Mellor begin analysis by identifying the problem domains of the system. Each domain is a separate world inhabited by its own conceptual entities, or objects '[Ref. 17]. Large domains are partitioned into subsystems. Each domain or subsystem is then separately analysed in three steps:

- information modelling
- state modelling
- process modelling.

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

The goal of information modelling is to identify the:

- objects in the subsystem
- attributes of each object
- relationships between each object.

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

The goal of state modelling is to identify the:

- states of each object, and the actions that are performed in them
- events that cause objects to move from one state to another
- sequences of states that form the life cycle of each object
- sequences of messages communicating events that flow between objects and subsystems.

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

The goal of process modelling is to identify the:

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

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

4.7.2 AD and DD phases

Shlaer and Mellor [Ref. 18] describe an Object-Oriented Design Language (OODLE), derived from the Booch and Buhr notation. There are four types of diagram:

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

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

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

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

- client-server
- friends.

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

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

Inheritance diagrams show the inheritance relationships between classes.

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

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

Recursive design can accelerate development because the generic computing processes can be implemented before the object model is complete.

4.8 HOOD

The Hierarchical Object-Oriented Design (HOOD) method [Ref. 21] was developed in 1987 under European Space Agency (ESA) contract A0/1-1890/86/NL/MA by an industrial consortium.

HOOD is a method for hierarchical decomposing a system into objects that correspond to:

- problem domain entities
- other abstract objects in the solution domain.

The HOOD method comprises techniques for textual and diagrammatic representation of the design.

HOOD has been used extensively in Europe for the development of systems ranging from small embedded applications to large distributed systems.

4.8.1 SR phase

HOOD does not support requirements analysis.

4.8.2 AD and DD phases

HOOD may be used in the AD and DD phases of a project for:

- identifying software modules
- identifying the dependencies between software modules by means of the 'use' relationship
- composing hierarchies of modules by means of 'include' relationships.

This page is intentionally left blank.

APPENDIX A GLOSSARY

A.1 LIST OF ACRONYMS AND ABBREVIATIONS

AD	Architectural Design
ADD	Architectural Design Document
ANSI	American National Standards Institute
AT	Acceptance Test
DD	Detailed Design and production
DDD	Detailed Design Document
ESA	European Space Agency
IEEE	Institute of Electrical and Electronics Engineers
OMT	Object Modeling Technique
OO	Object-Oriented
OOSE	Object-Oriented Software Engineering
IT	Integration Test
PHD	Project History Document
PSS	Procedures, Standards and Specifications
SCMP	Software Configuration Management Plan
SPMP	Software Project Management Plan
SQAP	Software Quality Assurance Plan
SR	Software Requirements
SSD	Software Specification Document
SSSD	Subsystem Software Specification Document
ST	System Test
STD	Software Transfer Document
SRD	Software Requirements Document
SUM	Software User Manual
SVVP	Software Verification and Validation Plan
UML	Unified Modeling Language
UR	User Requirements
URD	User Requirements Document
UT	Unit Test
WBS	Work Breakdown Structure

This page is intentionally left blank.

APPENDIX B REFERENCES

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

2. Guide to the ESA Software Engineering Standards, ESA PSS-05-01 Issue 1 October 1991.

3. Guide to the User Requirements Definition Phase, ESA PSS-05-02 Issue 1 October 1991.

4. Guide to the Software Requirements Definition Phase, ESA PSS-05-03 Issue 1 October 1991.

5. Guide to the Software Architectural Design Phase, ESA PSS-05-04 Issue 1 January 1992.

6. Guide to the Software Detailed Design and Production Phase, ESA PSS-05-05 Issue 1 May 1992.

7. Guide to the Software Transfer Phase , ESA PSS-05-06 Issue 1 October 1994.

8. Guide to the Software Operations and Maintenance Phase, ESA PSS-05-07 Issue 1 December 1994.

9. Guide to Software Project Management, ESA PSS-05-08 Issue 1 June 1994.

10. Guide to Software Configuration Management, ESA PSS-05-09 Issue 1 November 1992.

11. Guide to Software Verification and Validation, ESA PSS-05-10 Issue 1 February 1994.

12. Guide to Software Quality Assurance, ESA PSS-05-11 Issue 1 July 1993.

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

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

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

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

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

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

19. Object-Oriented Software Engineering - A User Case Driven Approach. I.Jacobson, Addison-Wesley, 1992.

20. Unified Modelling Language, G. Booch, J.Rumbaugh, I.Jacobson, Rational Software, 1996

21. HOOD reference manual 3.1, B. Delatte, M. Heitz, J. F. Muller, ESTEC

22. Guide to Applying the ESA Software Engineering Standards to Small Software Projects, BSSC(96)2, May 1996

23. Object-Oriented Development: The Fusion Method, B. Coleman, Prentice-Hall, 1994

24. Object-Oriented Application Frameworks, T. Lewis, P. Calder, E. Gamma, W. Pree, L. Rosenstein, K. Schmucker, A. Weigand and J. Vlissades, Manning, 1995

25. Design patterns, E.Gamma, R.Helm, R.Johnson and J.Vlissades, Addison-Wesley, 1995.

APPENDIX C PRACTICE GUIDELINES

C.1 INTRODUCTION

This chapter provides guidance on applying relevant mandatory and recommended practices when object-oriented methods are used. The order of the practices follows their order of appearance in ESA PSS-05-0.

C.2 SR PHASE

Mandatory practices

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

The implementation-independent model is called a 'logical model' and should describe the actors, use cases, external interfaces and classes. The completion criterion for the modelling activity is 'all capability requirements are traced to classes'.

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

Some recognised methods for OOA are listed in references 13 to 21.

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

Functions may be described in terms of use of cases or process diagrams showing sequences of object operations.

SR18 The SRD shall be compiled according to the table of contents provided in Appendix C (of ESA PSS-05-0).

Guidance on completing the template is included in Appendix D.2 of this guide.

Recommended practices

Part 1 Section 3.3.1 Para 4

Functions should have a single definite purpose. Function names should have a declarative structure (e.g. 'Validate Telecommands'), and say 'what' is to be done rather than 'how'. Good naming allows design components with strong cohesion to be easily derived (see Part 1, Section 4.3.1.3).

These recommendations apply to use cases and operations.

Part 1 Section 3.3.1 Para 5

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

These recommendations apply to classes.

Part 1 Section 3.3.1 Para 7

Each function should be decomposed into no more than seven sub-functions.

This recommendation does not apply.

Part 1 Section 3.3.1 Para 8

The model should omit implementation information (e.g. file, record, task, module);

Because of the seamless nature of OO, the object-oriented terms such as 'object', 'class', 'operation', 'service' and 'message' may occur in both the logical model, physical model and code. This recommendation applies to implementation of objects, classes etc.

Part 1 Section 3.3.1 Para 9

The performance attributes of each function (capacity, speed etc) should be stated;

The performance attributes of each use case or high level operation (capacity, speed etc) should be stated.

Part 1 Section 3.3.1 Para 10

Critical functions should be identified.

Critical use cases or high level operations should be identified.

Part 1 Section 3.4.1 para 3

The functional requirements should be structured top-down in the SRD. Non-functional requirements should be attached to functional requirements and therefore can appear at all levels of the hierarchy, and apply to all functional requirements below them (inheritance of family attributes).

Functional requirements should be structured according to the logical model. Functional requirements may be expressed in terms of use cases or high level operations.

C.3 AD PHASE

Mandatory practices

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

Some recognised methods for OOD are listed in references 13 to 21.

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

The 'physical model' should:

- *define the subsystems, their functions and their interfaces*
- *identify the classes within the subsystems that provide the subsystem functions*
- *describe the interactions between classes.*

The completion criterion for the physical modelling activity is 'class interactions for all use cases are defined'.

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

Object-oriented methods allow a top-down approach by means of the decomposition of the system into subsystems and classes. Therefore object-oriented methods conform to this requirement.

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

AD06 • data input;

AD07 • functions to be performed;

AD08 • data output.

With reference to AD06 to AD08, for each subsystem provide:

- *input data*
- *functions*
- *output data*

With reference to AD06 to AD08, for each class, provide for each operation:

- *input arguments*

- *description*
- *output arguments.*

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

Data structure definitions shall include the:

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

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

AD12
- range of possible values of each element;

AD13
- initial values of each element.

With reference to AD09 to AD13, for each class provide:

- *the name of the class*
- *the type of the class (e.g. interface, entity, control)*
- *a list of its attributes with their name, type and, where appropriate, extent*
- *the relationship of the class to other classes*
- *the range of possible values of each attribute*
- *initial value of each attribute.*

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

Describe the control flow between classes in terms of interaction diagrams [Ref. 19] or equivalent. Interaction diagrams show the messages sent between classes and identify the operations activated in the classes.

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

The major components are the:

- *subsystems*
- *classes in each subsystem.*

AD24 The ADD shall be compiled according to the table of contents provided in Appendix C (of ESA PSS-05-0).

Guidance on completing the template is included in Appendix D.3 of this guide.

Recommended practices

Part 1 Section 4.3.1.1 para 1

The software should be decomposed into a hierarchy of components according to a partitioning method. Examples of partitioning methods are functional decomposition and correspondence with real world objects.

Components are subsystems and classes.

Part 1 Section 4.3.1.1 para 5

In multi-tasking systems, the lowest level of the Architectural Design should be the task level.

This recommendation may not apply to object-oriented software. The completion criterion in Chapter 3 should be used to identify when to stop architectural design.

Part 1 Section 4.3.2 para 1

This should contain diagrams showing, at each level of the architectural design, the data flow and control flow between the components.

Interaction diagrams or equivalent should be used to explain control and data flow.

Part 1 Section 4.3.2.1 Para 3

Data inputs and outputs should be defined as data structures (see next section).

Data structures are classes. See the guidance on AD09 to AD13.

C.4 DD PHASE

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

DD02

- top-down decomposition;

  Object-oriented methods allow a top-down approach by means of the decomposition of the system into subsystems (defined in the AD phase) and classes. However object-oriented methods allow a bottom-up approach (e.g. the definition of superclasses from classes), and this should be used as well.

DD03 • structured programming;

Object-oriented programming incorporates the principles of structured programming, which can be directly applied to the programming of operations.

DD04 • concurrent production and documentation.

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

A module may contain the source code for one or more classes. A module can be separately compiled, and is normally the lowest level configuration item.

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

A major component is a subsystem, class, or group of classes as defined in the ADD.

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

Guidance on completing the template is included in Appendix D.4 of this guide.

APPENDIX D DOCUMENT TEMPLATES

D.1 INTRODUCTION

This chapter provides guidance on use the SRD, ADD and DDD templates when object-oriented methods are used.

All documents should contain the following service information:

a - Abstract
b - Table of contents
c - Document Status Sheet
d - Document Change Records made since last issue

If there is no information pertinent to a section, the section should be omitted and the sections renumbered.

Guidelines on the contents of document sections are given in italics. Section titles which are to be provided by document authors are enclosed in square brackets.

D.2 SRD TABLE OF CONTENTS

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

2 General Description
- 2.1 Relation to current projects
  Describe the relationship to other current projects.
- 2.2 Relation to predecessor and successor projects
  Describe the relationship to previous and future projects.
- 2.3 Function and purpose
  Describe the main functions the product must perform.
- 2.4 Environmental considerations
  Describe where the product will be used, who will use it, who will operate it, the hardware it will run on, and the operating system.
- 2.5 Relation to other systems
  Describe related external systems and subsystems.
- 2.6 General constraints
  Describe the main constraints that apply and why they exist.
- 2.7 Model description
  Describe the logical model using recognised analysis methods.

3 Specific Requirements
The specific requirements should be organised according to the structure of the logical model (e.g. subsections for each subject area, use case, class etc). Specific requirements that apply to every part of the logical model should be grouped under the heading 'system requirements'.
- 3.1 Functional requirements
- 3.2 Performance requirements
- 3.3 Interface requirements
- 3.4 Operational requirements
- 3.5 Resource requirements
- 3.6 Verification requirements
- 3.7 Acceptance testing requirements
- 3.8 Documentation requirements

3.9 Security requirements
3.10 Portability requirements
3.11 Quality requirements
3.12 Reliability requirements
3.13 Maintainability requirements
3.14 Safety requirements

4 User Requirements vs Software Requirements Traceability matrix
Give a table cross-referencing software requirements to user requirements.

D.3 ADD TABLE OF CONTENTS

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

2 System Overview
Summarise the system context and system design.

3 System Context
Describe the system context, with diagrams.
Define the external interfaces.

4 System Design
- 4.1 Design method
  Describe or reference the design method.
- 4.2 Decomposition description
  Describe the physical model using recognised design methods.

5 Component Description
Describe each subsystem and class. Structure this section according to the decomposition description. Separate physical documents may be produced to describe the classes in each subsystem.

5.n [Component identifier]
Name the subsystem or class.
- 5.n.1 Type
  Say whether the component is a subsystem or class.
- 5.n.2 Purpose
  Trace the component to the software requirements.
- 5.n.3 Function
  Say what the component does.
  For a subsystem, define the functions.
  For a class, describe each operation.
- 5.n.4 Subordinates
  List the immediate children.
  For each subsystem list the top-level classes.
  For each class list the subclasses.

5.n.5 Dependencies
Describe the preconditions for using this component.
For each subsystem list the subsystems that use it.
For each class list the super classes.
For each class list the classes that use the operations of the class.

5.n.6 Interfaces
For each subsystem define the input and output control and data flow.
For each class provide a list of its attributes with their name, type range of possible values, initial value and, where appropriate, extent.
For each class, provide the input and output arguments of each operation.

5.n.7 Resources
List the resources required, such as CPU capacity, memory, storage, displays and printers.

5.n.8 References
Reference any documents needed to understand the component.

5.n.9 Processing
Describe the control and data flow within the component.
For subsystems, describe the execution threads and identify the mechanisms used.
Outline the processing of each operation of the classes.

5.n.10 Data
Define in detail the data internal to components, such as files, used for interfacing major components. Otherwise give an outline description.

6 Feasibility and Resource Estimates
Summarise the computer resources required to build, operate and maintain the software.

7 Software Requirements vs Components Traceability matrix
Give a table cross-referencing components to software requirements.

8 Class Index (optional)
Relate the class names to the page number of the start of the section where they are described.

9 Class Summary (optional)
Give a table of classes, with a brief description of their purpose.

D.4 DDD TABLE OF CONTENTS

Part 1 - General Description

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

2 Project Standards, Conventions and Procedures
- 2.1 Design standards
  Describe or reference the design method used.
- 2.2 Documentation standards
  Describe the format, style and tools for documentation.
- 2.3 Naming conventions
  Describe the conventions for naming files, modules etc.
- 2.4 Programming standards
  Define and reference the coding standards.
- 2.5 Software development tools
  Define and reference the design and production tools.

Part 2 - Component Design Specifications

Describe each subsystem and class. Structure this section according to the decomposition description in the ADD. Separate physical documents may be produced to describe the classes in each subsystem.

5.n [Component identifier]

Name the subsystem or class.

- 5.n.1 Type
  Say whether the component is a subsystem or class.
- 5.n.2 Purpose
  Trace the component to the software requirements.
- 5.n.3 Function
  Say what the component does.
  For a subsystem, define the functions.
  For a class, describe each operation.
- 5.n.4 Subordinates
  List the immediate children.
  For each subsystem list the top-level classes.
  For each class list the subclasses.

5.n.5 Dependencies
Describe the preconditions for using this component.
For each subsystem list the subsystems that use it.
For each class list the super classes.
For each class list the classes that use the operations of the class.

5.n.6 Interfaces
For each subsystem define the input and output control and data flow.
For each class provide a list of its attributes with their name, type range of possible values, initial value and, where appropriate, extent.
For each class, provide the input and output arguments of each operation.

5.n.7 Resources
List the resources required, such as CPU capacity, memory, storage, displays and printers.

5.n.8 References
Reference any documents needed to understand the component.

5.n.9 Processing
Describe the control and data flow within the component using pseudo-code or a PDL.
For subsystems, describe the execution threads and identify the mechanisms used.
Outline the processing of each operation of the classes.

5.n.10 Data
Define in detail the data internal to components.

Appendix A Source code listings
Insert listings of the code or a configuration item list showing where the code can be found.

Appendix B Software Requirements vs Component Traceability matrix
Give a table cross-referencing components to software requirements.

Appendix C Class Index (optional)
Relate the class names to the page number of the start of the section where they are described.

Appendix D Class Summary (optional)
Give a table of classes, with a brief description of their purpose.

This page is intentionally left blank.

APPENDIX E SMALL SOFTWARE PROJECTS

E.1 INTRODUCTION

The Guide to Applying the ESA Software Engineering Standards to Small Software Projects, BSSC(96)2 [Ref. 22], contains a revised life cycle document templates for small software projects. Guidance is provided on applying the ESA PSS-05-0 mandatory practices to small projects.

This Guide to Applying the ESA Software Engineering to Projects using Object-Oriented Methods is compatible with the small projects guide in the sense that the guidance can be combined when using object-oriented methods in a small software project.

A revised template for Software Specification Document (SSD) for small software projects is given below. Section 5, component descriptions has been modified for projects using object-oriented methods. A revised template is also provided for source code headers.

E.2 SSD TABLE OF CONTENTS

1 Introduction
- 1.1 Purpose of the document
- 1.2 Definitions, acronyms and abbreviations
- 1.3 References
- 1.4 Overview of the document

2 Model description
Describe the logical model using a recognised analysis method.

3 Specific Requirements
The specific requirements should be organised according to the structure of the logical model (e.g. subsections for each subject area, use case, class etc). Specific requirements that apply to every part of the logical model should be grouped under the heading 'system requirements'.
- 3.1 Functional requirements
- 3.2 Performance requirements
- 3.3 Interface requirements
- 3.4 Operational requirements
- 3.5 Resource requirements
- 3.6 Verification requirements
- 3.7 Acceptance testing requirements
- 3.8 Documentation requirements
- 3.9 Security requirements
- 3.10 Portability requirements
- 3.11 Quality requirements
- 3.12 Reliability requirements
- 3.13 Maintainability requirements
- 3.14 Safety requirements

4 System design
- 4.1 Design method
  Describe or reference the design method used.
- 4.2 Decomposition description
  Describe the physical model using recognised design methods.

5 Component Description
Describe each subsystem and class.
Structure this section according to the decomposition description.

5.n [Component identifier]

Name the subsystem or class.

5.n.1 Type

Say whether the component is a subsystem or class

5.n.2 Purpose

Trace the component to the software requirements.

5.n.3 Function

Say what the component does.
For a subsystem, define the functions.
For a class, describe each operation .

5.n.4 Subordinates

List the immediate children.
For each subsystem list the top-level classes.
For each class list the subclasses.

5.n.5 Dependencies

Describe the preconditions for using this component.
For each subsystem list the subsystems that use it.
For each class list the super classes.
For each class list the classes that use the operations of the class.

5.n.6 Interfaces

For each subsystem define the input and output control and data flow.
For each class provide a list of its attributes with their name, type range of possible values, initial value and, where appropriate, extent.
For each class, provide the input and output arguments of each operation.

5.n.7 Resources

List the resources required, such as displays and printers.

5.n.8 References

Reference any documents needed to understand the component.

5.n.9 Processing

Describe the control and data flow within the component.
For subsystems, describe the execution threads and identify the mechanisms used.
Outline the processing of each operation of the classes.

5.n.10 Data

Define in detail the data internal to components, such as files, used for interfacing major components. Otherwise give an outline description.

6 Feasibility and Resource Estimates
Summarise the computer resources required to build, operate and maintain the software.

7 User Requirements vs Software Requirements Traceability matrix
Provide tables cross-referencing user requirements to software requirements and vice versa.

8 Software Requirements vs Components Traceability matrix
Provide tables cross-referencing software requirements to components and vice versa.

9 Class Index (optional)
Relate the class names to the page number of the start of the section where they are described.

10 Class Summary (optional)
Give a table of classes, with a brief description of their purpose.

E.3 SOURCE CODE DOCUMENTATION REQUIREMENTS

Each class should contain a header that defines:

Class Name
Author
Creation Date
Change History
Version/Date/Author/Description
Function
Define the purpose of the class and list the operations.
Interfaces
For each class provide:
- *a list of its attributes with their name, type range of possible values, initial value and, where appropriate, extent*
- *the input and output arguments of each operation.*

Dependencies
Describe the preconditions for using this component.
For each class list the super classes and any classes that use the operations of the class.
Processing
Summarise the processing using pseudo-code or a PDL

APPENDIX F DOCUMENT EXAMPLES

F.1 INTRODUCTION

This appendix provides examples of how the SRD, ADD and DDD can be structured for large and small systems that were developed using object-oriented methods. They are based closely upon actual projects.

F.2 LARGE MISSION CONTROL SYSTEM

The mission control system was developed using the Coad-Yourdon method and the OMT notation.

F.2.1 LARGE MISSION CONTROL SYSTEM SRD

Sections 1 and 2 of this SRD follow the Standard. Section 3 covers the system level specific requirements common to all subsystems, and has the same structure as section 3 of the SRD standard. Section 4 covers the subsystem level specific requirements; only requirements categories relevant to the subsystem are included.

1 Introduction
1.1 Purpose
1.2 Scope
1.3 Definitions, Acronyms, Abbreviations
1.4 References
1.5 Overview

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

3 System level requirements
3.1 Functional requirements
3.2 Performance requirements
3.3 Interface requirements

3.4 Operational requirements
3.5 Resource requirements
3.6 Verification requirements
3.7 Acceptance testing requirements
3.8 Documentation requirements
3.9 Security requirements
3.10 Portability requirements
3.11 Quality requirements
3.12 Reliability requirements
3.13 Maintainability requirements
3.14 Safety requirements

4 Specific requirements
4.1 Introduction
4.2 Modelling technique and notational conventions
4.3 Subject areas
4.4 Subject Area 1
4.4.1 Concept
4.4.2 Diagram
4.4.3 Description and requirements
Table of requirements, with attributes
4.5 Subject Area 2
...

F.2.2 LARGE MISSION CONTROL SYSTEM ADD

Sections 1, 2, 3 and 4 of this ADD follow the Standard. Section 5, subsystem descriptions, describes components, as required by the standard. However component descriptions are limited to relevant topics and a summary of the classes in each component is included. Indexes are included to enable easy location of information about subsystems and classes.

1 Introduction
1.1 Purpose
1.2 Scope
1.3 Definitions, Acronyms, Abbreviations
1.4 References
1.5 Overview

2 System overview

3 System context

4 System design
4.1 Design method
4.2 Decomposition description
The physical model is decomposed into components
The components are decomposed into subject areas
The subject areas are decomposed into subsystems.
4.3 How to read subsystem descriptions

5 Subsystem descriptions

5.1 Problem domain component
5.1.n Subject Area n
5.1.n.m Subsystem m
5.1. n.m.1 Type
5.1. n.m.2 Purpose
5.1. n.m.3 Function
Class diagram, showing relations.
Discussion of diagram.
5.1. n.m.4 Interfaces
Table of interfaces to other subsystems
5.1. n.m.5 Resources
5.1. n.m.6 References
5.1. n.m.7 Class summary
Class diagram, showing attributes and relations
Tables of attributes, with name, description and type

5.2 Human Interaction component
5.3 Data management component
5.4 Task management component

6 Feasibility and resource estimates

7 Software components vs component traceability matrix

A Subsystem page index

B Class page index.

F.2.3 LARGE MISSION CONTROL SYSTEM DDD

DDD part 1 information is located in planning documents. The DDD part 2 is organised into an overview DDD and DDDs for each subsystem

Sections 1 of each DDD follows the Standard. Section 2 of the overview DDD provides an overview of the subsystems. Section 2 of a subsystem DDD contains descriptions of the components used in the subsystem. Component descriptions are limited to relevant topics and a summary of the classes in each component is included. In addition there is a configuration item list and error code list for each subsystem.

F.2.3.1 Overview DDD table of contents

1 Introduction
1.1 Purpose
1.2 Scope
1.3 Definitions, Acronyms, Abbreviations
1.4 References
1.5 Overview

2 Subsystems overview
2.1 Subsystem 1
2.n Subsystem n

F.2.3.2 Subsystem DDD table of contents

1 Introduction
1.1 Purpose
1.2 Scope
1.3 Definitions, Acronyms, Abbreviations
1.4 References
1.5 Overview

2 Subsystem overview
2.1 Type
2.2 Purpose
2.3 Function
2.4 External interfaces
2.5 Resources

3 Component descriptions
3.n Component n
3.n.1 Type
3.n.2 Purpose
3.n.3 Function
3.n.4 Interfaces
3.n.5 Resources

3.n.6 Class summary
Table of attributes, with name, type, access and description
Table of services, with arguments, type, access and description.
Table of messages, with thread and description
3.n.7 Design notes

A Configuration Item List
B Subsystem error codes

F.3 SMALL MISSION PLANNING SYSTEM

The mission planning system was developed using Yourdon (i.e. DeMarco) Structured Analysis method for the SRD, and the OMT method for the Design Document (DD), which is a combined ADD and DDD.

F.3.1 SMALL MISSION PLANNING SYSTEM SRD

Sections 1 and 2 of this SRD follow the Standard. Section 3 is organised into subsystem level specific requirements and system level specific requirements.

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

2 General Description
- 2.1 Relation to current projects
- 2.2 Relation to predecessor and successor projects
- 2.3 Function and purpose
- 2.4 Environmental considerations
- 2.5 Relation to other systems
- 2.6 General constraints
- 2.7 Model description
- 2.7.1 Summary of analysis method
- 2.7.2 Context diagram and data dictionary

3 Specific Requirements
- 3.n Subsystem n
  - *Data flow diagram*

Data dictionary
Table of requirements classified as Functional, Performance, Interface Operational, Resource

3.x System level requirements
Table of requirements classified as verification, acceptance testing, documentation, security, portability, quality, reliability, maintainability, safety

4 User Requirements vs Software Requirements Traceability matrix

F.3.2 SMALL MISSION PLANNING SYSTEM DD

Sections 1 and 2 correspond to the standard DDD Part 1. Section 3 contains a System Overview, based upon the ADD and component descriptions as required by the ADD and DDD part 2. The component descriptions do not follow the standard, but contain equivalent relevant information.

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

2 Project standards, conventions and procedures

- 2.1 Design standards
- 2.1.1 Design method
  - 2.1.1.1 Object model
  - 2.1.1.2 Dynamic model
  - 2.1.1.3 Functional model
- 2.2 Documentation standards
- 2.3 Naming conventions
- 2.4 Programming standards
- 2.5 Software development tools

3 Component design specifications
- 3.1 System overview
- 3.n Class n

Purpose
Superclass
Table of Attributes, with type, access and description
Table of Services, with returns, arguments, access and function
Functional description of how the class works; may include dynamic model.