SOFTWARE ENGINEERING LABORATORY SERIES SEL-81-305

Recommended Approach to Software Development

Revision 3

JUNE 1992

National Aeronautics and
Space Administration

Goddard Space Flight Center
Greenbelt, Maryland 20771

FOREWORD

The **Software Engineering Laboratory** (SEL) is an organization sponsored by the National Aeronautics and Space Administration/Goddard Space Flight Center (NASA/GSFC) and created to investigate the effectiveness of software engineering technologies when applied to the development of applications software. The SEL was created in 1976 and has three primary organizational members:

NASA/GSFC, Software Engineering Branch

University of Maryland, Department of Computer Science

Computer Sciences Corporation, Software Engineering Operation

The goals of the SEL are (1) to understand the software development process in the GSFC environment; (2) to measure the effects of various methodologies, tools, and models on this process; and (3) to identify and then to apply successful development practices. The activities, findings, and recommendations of the SEL are recorded in the Software Engineering Laboratory Series, a continuing series of reports that includes this document.

The previous version of the *Recommended Approach to Software Development* was published in April 1983. This new edition contains updated material and constitutes a major revision to the 1983 version. The following are primary contributors to the current edition:

Linda Landis, Computer Sciences Corporation
Sharon Waligora, Computer Sciences Corporation
Frank McGarry, Goddard Space Flight Center
Rose Pajerski, Goddard Space Flight Center
Mike Stark, Goddard Space Flight Center
Kevin Orlin Johnson, Computer Sciences Corporation
Donna Cover, Computer Sciences Corporation

Single copies of this document can be obtained by writing to

Software Engineering Branch
Code 552
Goddard Space Flight Center
Greenbelt, Maryland 20771

ACKNOWLEDGMENTS

In preparation for the publication of this document and the *Manager's Handbook for Software Development*, teams of technical managers from NASA/GSFC and Computer Sciences Corporation (CSC) met weekly for many months to resolve issues related to flight dynamics software development. It was through their efforts, experience, and ideas that this edition was made possible.

NASA/GSFC Team Members

Sally Godfrey
Scott Green
Charlie Newman
Rose Pajerski
Mike Stark
Jon Valett

CSC Team Members

Linda Esker
Jean Liu
Bailey Spence
Sharon Waligora
Linda Landis

ABSTRACT

This document presents guidelines for an organized, disciplined approach to software development that is based on studies conducted by the Software Engineering Laboratory (SEL) since 1976. It describes methods and practices for each phase of a software development life cycle that starts with requirements definition and ends with acceptance testing. For each defined life cycle phase, this document presents guidelines for the development process and its management, and for the products produced and their reviews.

This document is a major revision of SEL-81-205.

NOTE: The material presented in this document is consistent with major NASA/GSFC standards.

NOTE: The names of some commercially available products cited in this document may be copyrighted or registered as trademarks. No citation in excess of fair use, express or implied, is made in this document and none should be construed.

CONTENTS

Section 1 — Introduction 1
Section 2 — The Software Development Life Cycle 5
Section 3 — The Requirements Definition Phase 21
Section 4 — The Requirements Analysis Phase 41
Section 5 — The Preliminary Design Phase 63
Section 6 — The Detailed Design Phase 85
Section 7 — The Implementation Phase 107
Section 8 — The System Testing Phase 135
Section 9 — The Acceptance Testing Phase 161
Section 10 — Keys to Success 179
Acronyms 185
References 187
Standard Bibliography of SEL Literature 189
Index 201

LIST OF FIGURES

Figure **Page**

1-1 The SEL Software Engineering Environment 1
2-1 Activities by Percentage of Total Development Staff Effort 6
2-2 Reuse Activities Within the Life Cycle 16
2-3 Graph Showing in Which Life-Cycle Phases Each Measure Is Collected 19
3-1 Generating the System and Operations Concept 23
3-2 Developing Requirements and Specifications 24
3-3 SOC Document Contents 33
3-4 Requirements and Specifications Document Contents 34
3-5 SCR Format 35
3-6 SCR Hardcopy Material Contents 36
3-7 SRR Format 37
3-8 SRR Hardcopy Material Contents 38
4-1 Analyzing Requirements 43
4-2 Timeline of Key Activities in the Requirements Analysis Phase 46
4-3 Effort Data Example - ERBS AGSS 53
4-4 Requirements Analysis Report Contents 55
4-5 SDMP Contents (2 parts) 56
4-6 SSR Format 59
4-7 SSR Hardcopy Material 60
5-1 Developing the Preliminary Design 65
5-2 Preliminary Design Phase Timeline 67
5-3 Extent of the Design Produced for FORTRAN Systems During the Preliminary and Detailed Design Phases 72
5-4 Level of Detail Produced for Ada Systems During Preliminary Design 73
5-5 Preliminary Design Report Contents 81
5-6 PDR Format 82
5-7 PDR Hardcopy Material 83
6-1 Generating the Detailed Design 87
6-2 Timeline of Key Activities in the Detailed Design Phase 88
6-3 Checklist for a Unit Design Inspection 94
6-4 Example of the Impact of Requirements Changes on Size Estimates - the UARS Attitude Ground Support System 98
6-5 Detailed Design Document Contents 100
6-6 CDR Format 103
6-7 CDR Hardcopy Material 104
7-1 Implementing a Software Build 109
7-2 Phases of the Life Cycle Are Repeated for Multiple Builds and Releases 110

LIST OF FIGURES (cont.)

Figure **Page**

7-3 Timeline of Key Activities in the Implementation Phase 112
7-4 Sample Checklist for Code Inspection 118
7-5 Integration Testing Techniques 121
7-6 Development Profile Example 126
7-7 Example of CPU Usage - ERBS AGSS 128
7-8 Generalized Test Plan Format and Contents 131
7-9 BDR Format 133
7-10 BDR Materials 134
8-1 System Testing 136
8-2 Timeline of Key Activities in the System Testing Phase 138
8-3 Sample Software Failure Report Form 148
8-4 EUVEDSIM System Test Profile 152
8-5 SEL Discrepancy Status Model 152
8-6 User's Guide Contents 154
8-7 System Description Contents 156
8-8 ATRR Format 158
8-8 ATRR Materials 158
9-1 Acceptance Testing 163
9-2 Timeline of Key Activities in the Acceptance Testing Phase 164
9-3 Sample Error-Rate Profile, UARS AGSS 175
9-4 Software Development History Contents 178

LIST OF TABLES

Table **Page**

2-1 Measures Recommended by the SEL 18
3-1 Objective Measures Collected During the Requirements Definition Phase 31
4-1 Objective Measures Collected During the Requirements Analysis Phase 51
5-1 Objective Measures Collected During the Preliminary Design Phase 78
6-1 Objective Measures Collected During the Detailed Design Phase 97
7-1 Objective Measures Collected During the Implementation Phase 125
8-1 Objective Measures Collected During the System Testing Phase 151
9-1 Objective Measures Collected During the Acceptance Testing Phase 174

SECTION 1

INTRODUCTION

This document presents a set of guidelines that constitute a disciplined approach to software development. It is intended primarily for managers of software development efforts and for the technical personnel (software engineers, analysts, and programmers) who are responsible for implementing the recommended procedures. This document is neither a manual on applying the technologies described here nor a tutorial on monitoring a government contract. Instead, it describes the methodologies and tools that the Software Engineering Laboratory (SEL) recommends for use in each life cycle phase to produce manageable, reliable, cost-effective software.

THE FLIGHT DYNAMICS ENVIRONMENT

The guidelines included here are those that have proved effective in the experiences of the SEL (Reference 1). The SEL monitors and studies software developed in support of flight dynamics applications at the National Aeronautics and Space Administration/Goddard Space Flight Center (NASA/GSFC). Since its formation in 1976, the SEL has collected data from more than 100 software development projects. Typical projects range in size from approximately 35,000 to 300,000 delivered source lines of code (SLOC) and require from 3 to 60 staff-years to produce.

Flight dynamics software is developed in two distinct computing environments: the Flight Dynamics Facility (FDF) and the Systems Technology Laboratory (STL). (See Figure 1-1.) Mission support software is engineered and operated in the mainframe environment of the FDF. This software is used in orbit determination, orbit adjustment, attitude determination, maneuver planning, and general mission analysis. Advanced concepts for flight dynamics are developed and studied in the STL. Software systems produced in this facility include simulators, systems requiring special architectures (e.g., embedded systems), flight

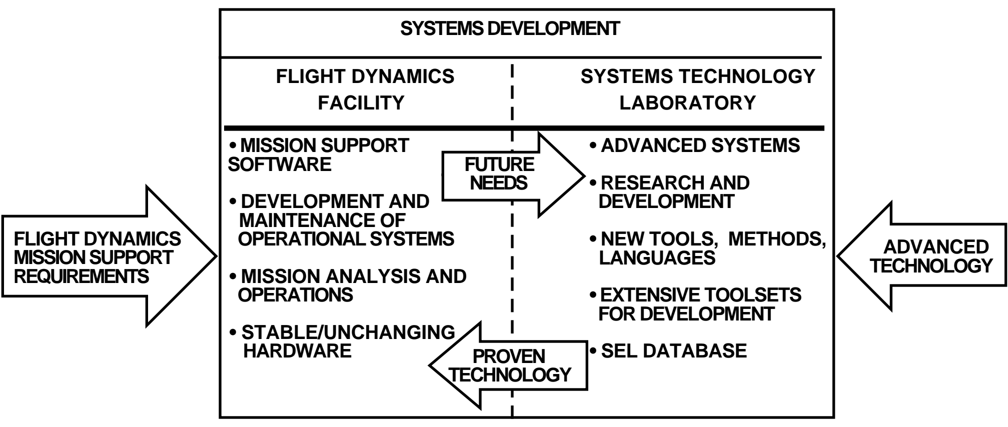


Figure 1-1. The SEL Software Engineering Environment

dynamics utilities, and projects supporting advanced system studies. The STL also hosts the SEL database and the entire set of SEL research tools.

This revised edition of the *Recommended Approach to Software Development* reflects the evolution in life cycle, development methodology, and tools that has taken place in these environments in recent years. During this time, Ada and object-oriented design (OOD) methodologies have been introduced and used successfully. The potential for reuse of requirements, architectures, software, and documentation has been, and continues to be, studied and exploited. Ongoing studies also include experiments with the Cleanroom methodology (References 2 through 4), formal inspection, and computer-aided software engineering (CASE) tools.

Because the SEL's focus is process improvement, it is a catalyst for this evolution. The SEL continuously conducts experiments using the actual, production environment. The lessons learned from these experiments are routinely fed back into an evolving set of standards and practices that includes the *Recommended Approach.*

As these studies are confined to flight dynamics applications, readers of this document are cautioned that the guidance presented here may not always be appropriate for environments with significantly different characteristics.

DOCUMENT OVERVIEW

This document comprises 10 sections. Sections 3 through 9 parallel the phases of the software development life cycle through acceptance testing, and discuss the key activities, products, reviews, methodologies, tools, and metrics of each phase.

Section 1 presents the purpose, organization, and intended audience for the document.

Section 2 provides an overview of the **software development life cycle.** The general goals of any software development effort are discussed, as is the necessity of tailoring the life cycle to adjust to projects of varying size and complexity.

Section 3 provides guidelines for the **requirements definition** phase**.** Generation of the system and operations concept and the requirements and specifications documents are covered. The purpose and format of the system concept and requirements reviews are outlined.

Section 4 discusses the key activities and products of the **requirements analysis** phase: requirements classifications, walk-throughs, functional or object-oriented analysis, the requirements analysis report, and the software specifications review.

Section 5 presents the recommended approach to **preliminary design**. The activities, products, and methodologies covered include structured and object-oriented design, reuse analysis, design walk-throughs, generation of prolog and program design language, the preliminary design report, and the preliminary design review.

Section 6 provides comparable material for the **detailed design** phase. Additional topics include the build test plan, completion of prototyping activities, the critical design review, and the detailed design document.

Section 7 contains guidelines for **implementation** of the designed software system. Coding, code reading, unit testing, and integration are among the activities discussed. The system test plan and user's guide are summarized.

Section 8 addresses **system testing**, including test plans, testing methodologies, and regression testing. Also covered are preparation of the system description document and finalization of the acceptance test plan.

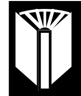
Section 9 discusses the products and activities of the **acceptance testing** phase: preparing tests, executing tests, evaluating results, and resolving discrepancies.

Section 10 itemizes key **DOs and DON'Ts** for project success.

A list of **acronyms**, **references**, a **bibliography** of SEL literature, and an **index** conclude this document.

Although the **maintenance and operation** phase is beyond the scope of the current document, efforts are now underway in the SEL to study this important part of the life cycle. The results of these studies will be incorporated into a future edition.

REFERENCE

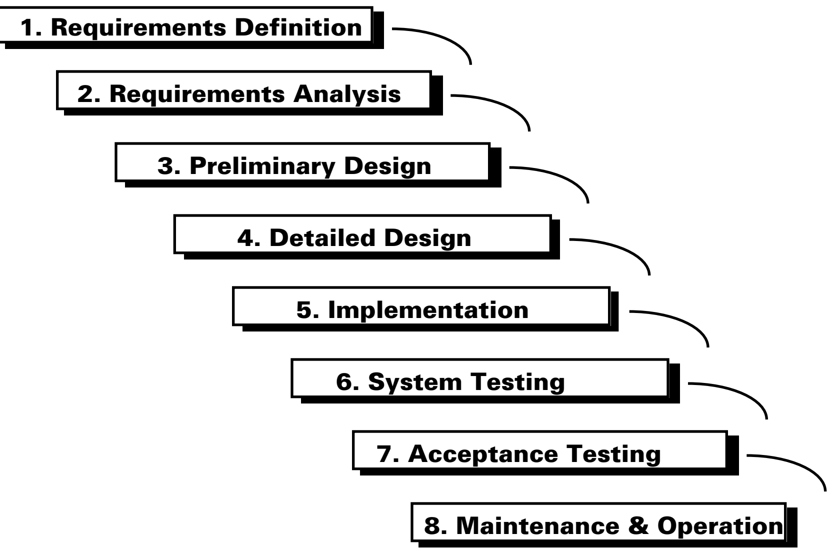


Recent SEL papers on software maintenance include "Measurement Based Improvement of Maintenance in the SEL," and "Towards Full Life Cycle Control, " both by Rombach, Ulery, and Valett. See References 5 and 6.

SECTION 2

THE SOFTWARE DEVELOPMENT LIFE CYCLE

The flight dynamics software development process is modeled as a series of eight sequential phases, collectively referred to as the *software development life cycle*:



Each phase of the software development life cycle is characterized by specific activities and the products produced by those activities.

As shown in Figure 2-1, these eight phases divide the software life cycle into consecutive time periods that do not overlap. However, the *activities* characteristic of one phase may be performed in other phases. Figure 2-1 graphs the spread of activities throughout the development life cycle of typical flight dynamics systems. The figure shows, for example, that although most of the work in analyzing requirements occurs during the requirements analysis phase, some of that activity continues at lower levels in later phases as requirements evolve.

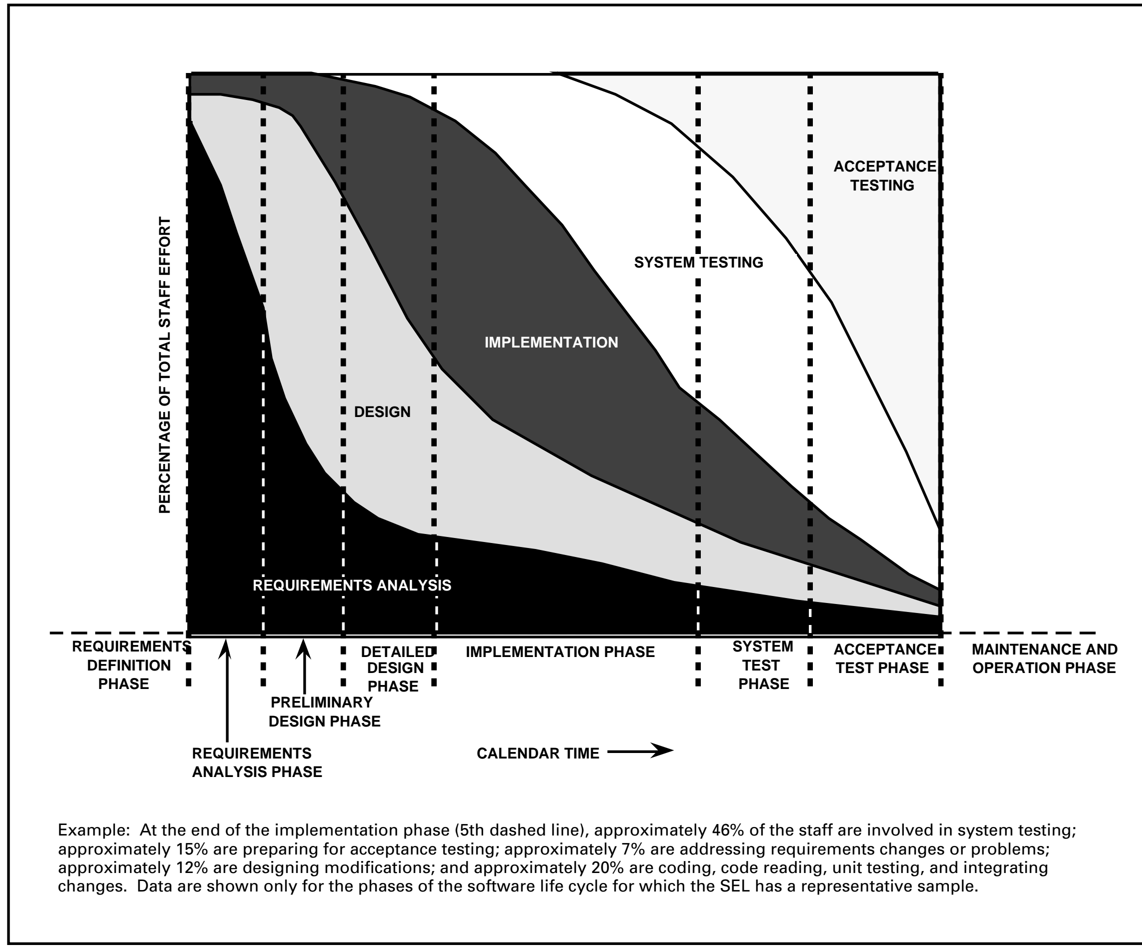


Figure 2-1. Activities by Percentage of Total Development Staff Effort

PHASES OF THE LIFE CYCLE

The eight phases of the software development life cycle are defined in the following paragraphs.

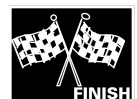
Requirements Definition

Requirements definition is the process by which the needs of the customer are translated into a clear, detailed specification of what the system must do. For flight dynamics applications, the requirements definition phase begins as soon as the mission task is established. A team of analysts studies the available information about the mission and develops an operations concept. This includes a timeline for mission events, required attitude maneuvers, the types of computational processes involved, and specific operational scenarios. The functions that the system must perform are defined down to the level of a subsystem (e.g., a telemetry processor).

NOTE

In this document, the term *analyst* refers to those specialists in flight dynamics (astronomers, mathematicians, physicists, and engineers) who determine the detailed requirements of the system and perform acceptance tests. For these activities, analysts work in teams (e.g., the requirements definition team) and function as agents for the end users of the system.

NOTE



In each phase of the life cycle, certain milestones must be reached in order to declare the phase complete. Because the life cycle is sequential, these *exit criteria* are also the *entry criteria* for the following phase. In this document, entry and exit criteria are shown in the summary tables on the first page of Sections 3 through 9. A brief discussion of the phase's exit criteria is provided at the conclusion of each section.

Working with experienced developers, analysts identify any previously developed software that can be reused on the current project. The advantages and disadvantages of incorporating the existing components are weighed, and an overall architectural concept is negotiated. The results of these analyses are recorded in the *system and operations concept (SOC)* document and assessed in the *system concept review (SCR)*.

Guided by the SOC, a requirements definition team derives a set of system-level requirements from documents provided by the mission project office. A draft version of the requirements is then recast in terms suitable for software design. These specifications define what data will flow into the system, what data will flow out, and what steps must be taken to transform input to output. Supporting mathematical information is included, and the completed *requirements and specifications* document is published. The conclusion of this phase is marked by the *system requirements review (SRR)*, during which the requirements and specifications for the system are evaluated.

Requirements Analysis

The requirements analysis phase begins after the SRR. In this phase, the development team analyzes the requirements and specifications document for completeness and feasibility. The development team uses structured or object-oriented analysis and a requirements classification methodology to clarify and amplify the document. Developers work closely with the requirements definition team to resolve ambiguities, discrepancies, and to-be-determined (TBD) requirements or specifications.

The theme of reuse plays a prominent role throughout the requirements analysis and design phases. Special emphasis is placed on identifying potentially reusable architectures, designs, code, and approaches. (An overview of reuse in the life cycle is presented later in this section.)

When requirements analysis is complete, the development team prepares a summary *requirements analysis report* as a basis for preliminary design. The phase is concluded with a *software specifications review (SSR),* during which the development team presents the results of their analysis for evaluation. The

requirements definition team then updates the requirements and specifications document to incorporate any necessary modifications.

Preliminary Design

The baselined requirements and specifications form a contract between the requirements definition team and the development team and are the starting point for preliminary design. During this phase, members of the development team define the software architecture that will meet the system specifications. They organize the requirements into major subsystems and select an optimum design from among possible alternatives. All internal and external interfaces are defined to the subsystem level, and the designs of high-level functions/objects are specified.

The development team documents the high-level design of the system in the *preliminary design report*. The preliminary design phase culminates in the *preliminary design review (PDR)*, where the development team formally presents the design for evaluation.

Detailed Design

During the detailed design phase, the development team extends the software architecture defined in preliminary design down to the unit level. By successive refinement techniques, they elaborate the preliminary design to produce "code-to" specifications for the software. All formalisms for the design are produced, including the following:

- Functional or object-oriented design diagrams
- Descriptions of all user input, system output (for example, screen, printer, and plotter), and input/output files
- Operational procedures
- Functional and procedural descriptions of each unit
- Descriptions of all internal interfaces among units

The development team documents these design specifications in the *detailed design document* that forms the basis for implementation. At the *critical design review (CDR),* which concludes this phase, the detailed design is examined to determine whether levels of detail and completeness are sufficient for coding to begin.

DEFINITIONS

Throughout this document, the term *unit* is used to designate any set of program statements that are logically treated as a whole. A main program, a subroutine, or a subprogram may each be termed a *unit*. A *module* is a collection of logically related units. *Component* is used in its English language sense to denote any constituent part.

Implementation

In the implementation (code, unit testing, and integration) phase, the developers code new components from the design specifications and revise existing components to meet new requirements. They integrate each component into the growing system, and perform unit and integration testing to ensure that newly added capabilities function correctly.

In a typical project, developers build several subsystems simultaneously from individual components. The team repeatedly tests each subsystem as new components are coded and integrated into the evolving software. At intervals, they combine subsystem capabilities into a complete working system for testing end-to-end processing capabilities. The sequence in which components are coded and integrated into executable subsystems and the process of combining these subsystems into systems are defined in an implementation plan that is prepared by development managers during the detailed design phase.

The team also produces a *system test plan* and a draft of the *user's guide* in preparation for the system testing phase that follows. Implementation is considered complete when all code for the system has been subjected to peer review, tested, and integrated into the system.

System Testing

During the system testing phase, the development team validates the completely integrated system by testing end-to-end capabilities according to the system test plan. The system test plan is based on the requirements and specifications document. Successfully completing the tests specified in the test plan demonstrates that the system satisfies the requirements.

In this phase, the developers correct any errors uncovered by system tests. They also refine the draft user's guide and produce an initial *system description* document. System testing is complete when all tests specified in the system test plan have been run successfully.

Acceptance Testing

In the acceptance testing phase, the system is tested by an independent acceptance test team to ensure that the software meets all requirements. Testing by an independent team (one that does not have the developers' preconceptions about the functioning of the system) provides assurance that the system satisfies the intent of the

original requirements. The acceptance test team usually consists of analysts who will use the system and members of the requirements definition team.

The tests to be executed are specified in the *acceptance test plan* prepared by the acceptance test team before this phase. The plan is based on the contents of the requirements and specifications document and approved specification modifications.

During acceptance testing, the development team assists the test team and may execute acceptance tests under its direction. Any errors uncovered by the tests are corrected by the development team. Acceptance testing is considered complete when the tests specified in the acceptance test plan have been run successfully and the system has been formally accepted. The development team then delivers final versions of the software and the system documentation (user's guide and system description) to the customer.

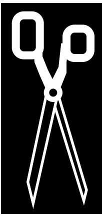
Maintenance and Operation

At the end of acceptance testing, the system becomes the responsibility of a maintenance and operation group. The activities conducted during the maintenance and operation phase are highly dependent on the type of software involved. For most flight dynamics software, this phase typically lasts the lifetime of a spacecraft and involves relatively few changes to the software. For tools and general mission support software, however, this phase may be much longer and more active as the software is modified to respond to changes in the requirements and environment.

The maintenance and operation phase is not specifically addressed in this document. However, because enhancements and error corrections also proceed through a development life cycle, the recommended approach described here is, for the most part, applicable to the maintenance and operation phase. The number and formality of reviews and the amount of documentation produced during maintenance and operation vary depending on the size and complexity of the software and the extent of the modifications.

NOTE

Recent SEL studies have shown that most of the effort in initial maintenance of flight dynamics systems is spent in enhancing the system after launch to satisfy new requirements for long-term operational support. Such enhancements are usually effected without radically altering the architecture of the system. Errors found during the maintenance and operation phase are generally the same type of faults as are uncovered during development, although they require more effort to repair.



TAILORING THE LIFE CYCLE

One of the key characteristics that has shaped the SEL's recommended approach to software development is the homogeneous nature of the problem domain in the flight dynamics environment. Most software is designed either for attitude determination and control for a specific mission, for mission-general orbit determination and tracking, or for mission planning. These projects progress through each life cycle phase sequentially, generating the standard documents and undergoing the normal set of reviews.

Certain projects, however, do not fit this mold. Within the STL, experiments are conducted to study and improve the development process. Advanced tools are developed. For these development efforts — prototypes, expert systems, database tools, Cleanroom experiments, etc. — the life cycle and the methodologies it incorporates often need adjustment. Tailoring allows variation in the level of detail and degree of formality of documentation and reviews, which may be modified, replaced, or combined in the tailoring process. Such tailoring provides a more exact match to unique project requirements and development products at a lower overall cost to the project without sacrificing quality.

RULE

The software development/ management plan (SDMP) must describe how the life cycle will be tailored for a specific project. See Section 4 for more details.

The following paragraphs outline general guidelines for tailoring the life cycle for projects of varying size and type. Additional recommendations may be found throughout this document, accompanying discussions of specific products, reviews, methods, and tools.

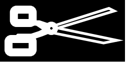
Builds and Releases

The sizes of typical flight dynamics projects vary considerably. Simulators range from approximately 30 thousand source lines of code (KSLOC) to 160 KSLOC. Attitude ground support systems for specific missions vary between 130 KSLOC and 300 KSLOC, while large mission-general systems may exceed 1 million SLOC. The larger the project, the greater the risk of schedule slips, requirements changes, and acceptance problems. To reduce these risks, the implementation phase is partitioned into increments tailored to the size of the project.

Flight dynamics projects with more than 10 KSLOC are implemented in builds. A *build* is a portion of a system that satisfies, in part or completely, an identifiable subset of the specifications. Specifications met in one build also are met in all successor builds. The last build, therefore, is the complete system.

> **NOTE**
>
>
>
> **Reviews are recommended for each build. The suggested format and contents of build design reviews are provided in Section 7.**

A *release* is a build that is delivered for acceptance testing and subsequently released for operational use. Projects of fewer than 300 KSLOC are usually delivered in a single release, unless otherwise dictated by scheduling (e.g., launch) considerations or by TBD requirements. Large projects (more than 300 KSLOC) are generally delivered in multiple releases of 300 to 500 KSLOC each.

> **NOTE**
>
> **Guidelines for tailoring the development approach (including reviews, documentation, and testing) for projects of differing scope and function are provided throughout this document. Look for the scissors symbol in the margin.**
>
> 

Builds within large projects may last up to 6 months. Builds within small projects may be only 2 to 3 months in duration.

Reviews

Reviews are conducted to ensure that analysts and developers understand and fulfill customer needs. Because reviews are designed to assist developers, not to burden them unnecessarily, the number of reviews held may vary from project to project. For tools development, the requirements, requirements analysis, and preliminary design might be reviewed together at PDR. For small projects spanning just several months, only two reviews may be applicable — the SRR and CDR. For very large projects, a CDR could (and should) be held for each major release and/or subsystem to cover all aspects of the system and to accommodate changing requirements.

The criteria used to determine whether one or more reviews can be combined depend on the development process and the life cycle phase. In the requirements analysis phase, for example, answers to the following questions would help determine the need for a separate SSR:

- Are there outstanding analysis issues that need to be reviewed?
- How much time will there be between the start of requirements analysis and the beginning of design?
- How stable are the requirements and specifications?

On small projects, technical reviews can be no more formal than a face-to-face meeting between the key personnel of the project and the customer technical representative. On typical flight dynamics projects, however, reviews are formalized and follow specific formats. Guidelines for these reviews are provided in Sections 3 through 9.

Documentation

On small projects, technical documentation is less formal than on medium or large projects, and fewer documents are published. Documents that would normally be produced separately on larger projects are combined. On a small research project, a single design document may replace the preliminary design report, detailed design document, and system description.

Testing and Verification

Independent testing is generally not performed on small-scale, tool-development efforts. Test plans for such projects can be informal. Although code reading is always performed on even the smallest project, units are often tested in logically related groups rather than individually, and inspections are usually conducted in informal, one-on-one sessions.

Configuration Management and Quality Assurance

Configuration management encompasses all of the activities concerned with controlling the contents of a software system. These activities include monitoring the status of system components, preserving the integrity of released and developing versions of a system, and governing the effects of changes throughout the system. *Quality assurance* activities ensure that software development processes and products conform to established technical requirements and quality standards.

All software and documentation that are developed for delivery are generally subject to formal configuration management and quality assurance controls. Tools developed exclusively for internal use are exempt, unless the tool is required to generate, run, or test a deliverable system.

On medium and small projects, configuration control may be performed by a designated member of the development team — a practice that is strongly discouraged on a large project. Similarly, the quality assurance function may be assigned to a team member with other responsibilities or may be handled by the technical leader.

Prototyping

A *prototype* is an early experimental model of a system, system component, or system function that contains enough capabilities for it to be used to establish or refine requirements or to validate critical design concepts. In the flight dynamics environment, prototypes are used to (1) mitigate risks related to new technology (e.g., hardware, language, design concepts) or (2) resolve requirements issues. In the latter case, entire projects may be planned as prototyping efforts that are designed to establish the requirements for a later system.

Unless the end product of the entire project is a prototype, prototyping activities are usually completed during the requirements analysis and design phases. The prototyping activity has its own, usually informal, life cycle that is embedded within the early phases of the full system's life cycle. If any portion of the prototype is to become part of the final system, it must be validated through all the established checkpoints (design reviews, code reading, unit testing and certification, etc.). As a rule, such prototyping activities should require no more than 15 percent of the total development effort.

RULE

All prototyping activities must be planned and controlled. The plan must define the purpose and scope of the prototyping effort, and must establish specific completion criteria. See Section 4 for more details.

WHEN TO PROTOTYPE

As a rule of thumb, use prototyping whenever

- **the project involves new technology, e.g., new hardware, development language, or system architecture**
- **the requirements are not understood**
- **there are major, unresolved issues concerning performance, reliability, or feasibility**
- **the user interface is critical to system success or is not clearly understood**

For projects in which the end product is a prototype, however, an iterative life cycle may be preferable. This is particularly true when a new user interface is a significant component of the system. An initial version of the prototype is designed, implemented, and demonstrated to the customer, who adds or revises requirements accordingly. The prototype is then expanded with additional builds, and the cycle continues until completion criteria are met.

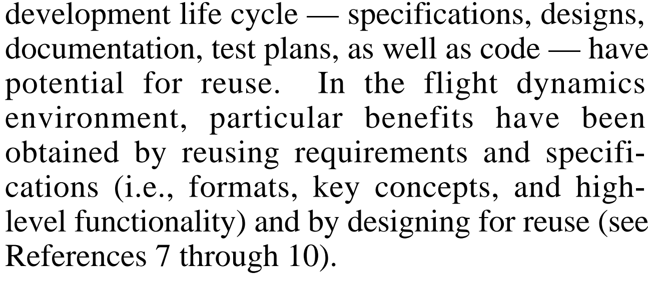
Tailoring the life cycle for any type of prototyping requires careful planning. The more new technology that is to be used on a project, the greater the prototyping effort. The larger the prototyping effort, the more formalized must be its planning, development, and management. The results of even the smallest prototyping effort must always be documented. Lessons learned from the prototype are incorporated into plans for subsequent phases and are included in the project history. See Section 4 for additional guidance on planning and documenting prototyping activities.

REUSE THROUGHOUT THE LIFE CYCLE

From the beginning to the end of the life cycle, the approach to software development recommended by the SEL stresses the principle of *reuse*. Broadly speaking, the reuse of existing experience is a key ingredient to progress in any area. Without reuse, everything must be relearned and re-created. In software development, reuse eliminates having to "reinvent the wheel" in each phase of the life cycle, reducing costs and improving both reliability and productivity.

Planning for reuse maximizes these benefits by allowing the cost of the learning curve in building the initial system to be amortized over the span of follow-on projects. Planned reuse is a primary force behind such recent technologies as object-oriented design and Ada.

All experience and products of the software development life cycle — specifications, designs, documentation, test plans, as well as code — have potential for reuse. In the flight dynamics environment, particular benefits have been obtained by reusing requirements and specifications (i.e., formats, key concepts, and high-level functionality) and by designing for reuse (see References 7 through 10).



KEY REUSE ELEMENTS

Analyze these key elements of a project for possible reuse:
- **requirements characteristics**
- **software architecture**
- **software development process**
- **design architecture or concepts**
- **test plans and procedures**
- **code**
- **user documentation**
- **staff**

Figure 2-2 shows how reuse activities fit into the software development life cycle. The top half of the figure contains activities that are conducted to enable future reuse. The lower half shows activities in which existing software is used in the system under development. These activities are outlined in the following paragraphs.

Activities That Enable Future Reuse

domain analysis

Domain analysis is the examination of the application domain of the development organization to identify common requirements and functions. It is usually performed during the requirements definition and analysis phases, but it may also be conducted as a separate activity unconnected to a particular development effort. Domain analysis produces a standard, general architecture or model that incorporates the common functions of a specific application area and can be tailored to accommodate differences between individual projects. It enables *requirements generalization,* i.e., the preparation of requirements and specifications in such a way that they cover a selected "family" of projects or missions.

requirements generalization

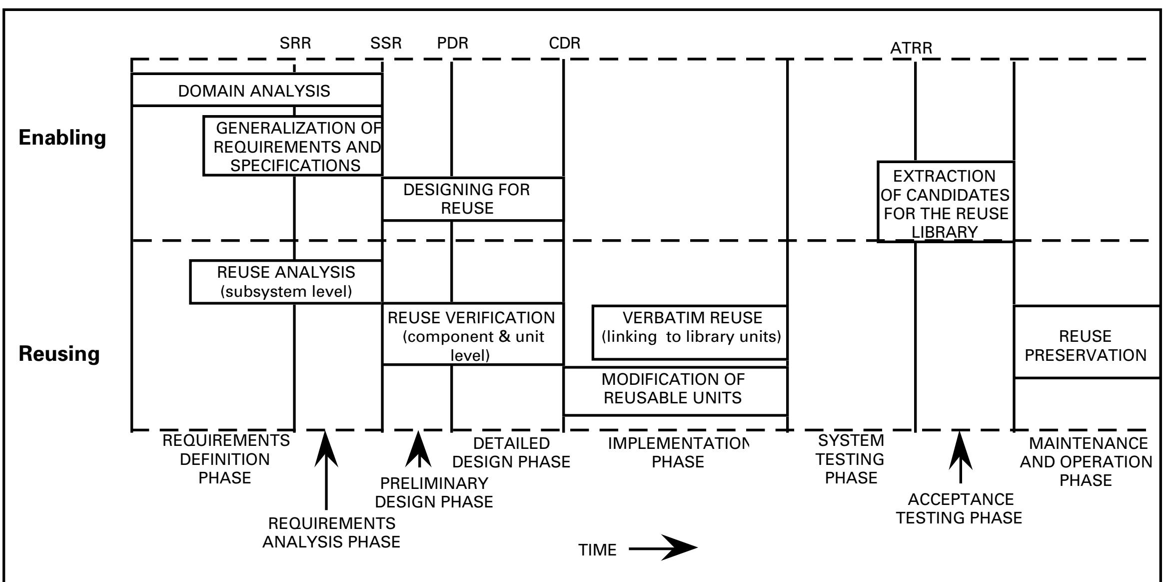


Figure 2-2. Reuse Activities Within the Life Cycle

designing for reuse

Software not originally intended for reuse is more difficult to incorporate into a new system than software explicitly designed for reuse. *Designing for reuse* provides modularity, standard interfaces, and parameterization. Design methods that promote reusability are described in References 9 and 11.

reuse libraries

Reuse libraries hold reusable source code and associated requirements, specifications, design documentation, and test data. In addition to storing the code and related products, the library contains a search facility that provides multiple ways of accessing the software (e.g., by keyword or name). On projects where reuse has been a design driver, *extraction of candidate software* for inclusion in the reuse library takes place after system testing is complete.

Reuse on Current Projects

reuse analysis and verification

During the requirements definition and analysis phases, *reuse analysis* is performed to determine which major segments (subsystems) of existing software can be used in the system to be developed. In the design phases, developers *verify* this analysis by examining each reusable element individually. During the preliminary design phase, developers evaluate major components to determine whether they can be reused verbatim or must be modified; individual units from the reuse library are examined during the detailed design phase.

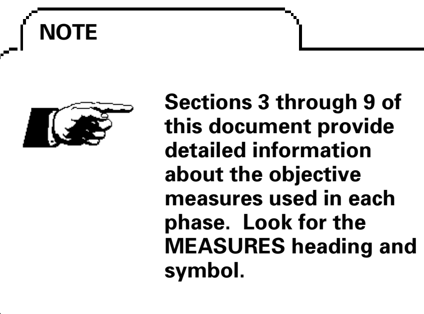
Software may be *reused verbatim* or may be *modified* to fit the needs of the current project. During the implementation phase, developers integrate existing, unchanged units into the developing system by linking directly to the reuse library. Modified software, on the other hand, must be subjected to peer review and unit testing before being integrated.

reuse preservation

A final reuse activity takes place during the maintenance and operation phase of the life cycle. Through the changes that it implements, the maintenance team can positively or negatively affect the reusability of the system; "quick fixes", for example, may complicate future reuse. *Reuse preservation* techniques for maintenance use many of the same practices that promote reuse during the analysis, design, and implementation phases.

MEASURES

Measures of project progress and viability are key to the effective management of any software development effort. In each phase of the life cycle, there are certain critical metrics that a manager must examine to evaluate the progress, stability, and quality of the development project.



Both *objective* and *subjective* data are measured. Objective data are actual counts of items (e.g., staff hours, SLOC, errors) that can be independently verified. Subjective data are dependent on an individual's or group's assessment of a condition (e.g., the level of difficulty of a problem or the clarity of requirements). Together, these data serve as a system of checks and balances. Subjective data provide critical information for interpreting or validating objective data, while objective data provide definitive counts that may cause the manager to question his or her subjective understanding and to investigate further.

Objective measures can be further classified into two groups: those that measure progress or status and those that measure project quality (e.g., stability, completeness, or reliability). Progress measures, such as the number of units coded or the number of tests passed, are evaluated against calculations of the total number of items to be completed. Quality measures, on the other hand, are

Table 2-1. Measures Recommended by the SEL

MEASURES CLASS	MEASURE	SOURCE	FREQUENCY	MAJOR APPLICATION
ESTIMATES	Estimates of: •Total SLOC (new, modified, reused) • Total units • Total effort • Major dates	Managers	Monthly	• Project stability • Planning aid
RESOURCES	• Staff hours (total & by activity) • Computer use	Developers Automated tool	Weekly Weekly	• Project stability • Replanning indicator • Effectiveness/impact of the development process being applied
STATUS	• Requirements (growth, TBDs, changes, Q&As) • Units designed, coded, tested • SLOC (cumulative) • Tests (complete, passed)	Managers Developers Automated Developers	Biweekly Biweekly Weekly Biweekly	• Project progress • Adherence to defined process • Stability and quality of requirements
ERRORS/ CHANGES	• Errors (by category) • Changes (by category) • Changes (to source)	Developers Developers Automated	By event By event Weekly	• Effectiveness/impact of the development process • Adherence to defined process
FINAL CLOSE-OUT	Actuals at completion: • Effort • Size (SLOC, units) • Source characteristics • Major dates	Managers	1 time, at completion	• Build predictive models • Plan/manage new projects

only useful if the manager has access to models or metrics that represent what should be expected.

In the SEL, measurement data from current and past projects are stored in a *project histories database*. Using information extracted from such a database, managers can gauge whether measurement trends in the current project differ from the expected models for the development environment. (See Section 6 of Reference 12.)

The management measures recommended by the SEL are listed in Table 2-1. Figure 2-3 shows in which phases of the life cycle each of these measures is collected.

As Table 2-1 shows, developers are responsible for providing many of the measures that are collected. In the SEL, developers use various *data collection forms* for this purpose. The individual forms are discussed in the sections of this document covering the life-cycle phases to which they apply.

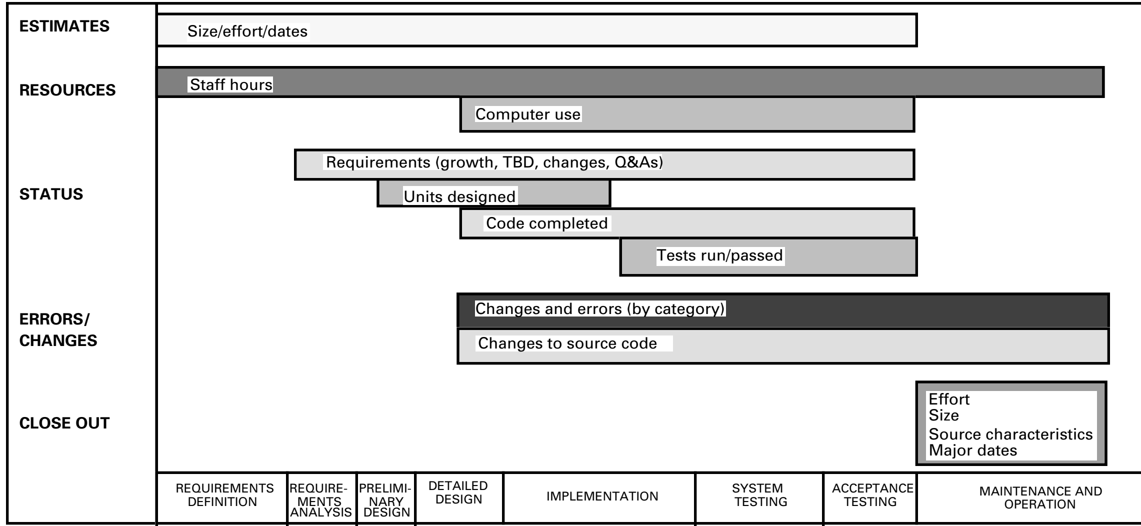
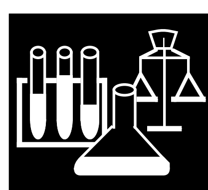


Figure 2-3. Graph Showing in Which Life-Cycle Phases Each Measure Is Collected



EXPERIMENTATION

Measurement is not only essential to the management of a software development effort; it is also critical to software process improvement. In the SEL, process improvement is a way of life. Experiments are continually being conducted to investigate new software engineering technologies, practices, and tools in an effort to build higher-quality systems and improve the local production process. The SEL's ongoing measurement program provides the baseline data and models of the existing development environment against which data from experimental projects are compared.

For several years, the SEL has been conducting experiments and measuring the impact of the application of the Cleanroom methodology (References 2, 3, and 4), which was developed in the early 1980s by Harlan Mills. The goal of the Cleanroom methodology is to build a software product correctly the first time. Cleanroom stresses disciplined "reading" techniques that use the human intellect to verify software products; testing is conducted for the purpose of quality assessment rather than as a method for detecting and repairing errors.

The Cleanroom methodology is still in the early stages of evaluation by the SEL. Although some of the methods of Cleanroom are the same as existing methods in the SEL's recommended approach — e.g., code reading — other aspects remain experimental.

Consequently, the Cleanroom methodology is used throughout this document as an example of the integral aspect of experimentation and process improvement to the SEL's recommended approach. Variations in life cycle processes, methods, and tools resulting from the application of Cleanroom will be highlighted. Look for the experimentation symbol.

DEFINITION

The term *Cleanroom* was borrowed from integrated circuit production. It refers to the dust-free environments in which the circuits are assembled.

LIFE CYCLE PHASES	REQUIREMENTS DEFINITION	REQUIRE-MENTS ANALYSIS	PRELIMI-NARY DESIGN	DETAILED DESIGN	IMPLEMENTATION	SYSTEM TESTING	ACCEPTANCE TESTING

SECTION 3

THE REQUIREMENTS DEFINITION PHASE

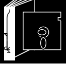
PHASE HIGHLIGHTS

ENTRY CRITERIA

- Problem/project description completed
- Project approved

EXIT CRITERIA

- System and operations concept completed
- SRR completed
- Requirements and specifications baselined



PRODUCTS

- System and operations concept document
- Requirements and specifications document

MEASURES

- Staff hours
- Number of requirements defined vs. estimated total requirements
- Percentage of requirements with completed specifications

METHODS AND TOOLS

- Structured or object-oriented analysis
- Walk-throughs
- Prototyping

KEY ACTIVITIES

Requirements Definition Team

- Develop a system concept
- Prepare the reuse proposal
- Develop an operations concept
- Define the detailed requirements
- Derive the specifications
- Conduct the SCR and SRR

Management Team

- Develop a plan for the phase
- Staff and train the requirements definition team
- Interact with the customer
- Evaluate progress and products
- Control major reviews

OVERVIEW

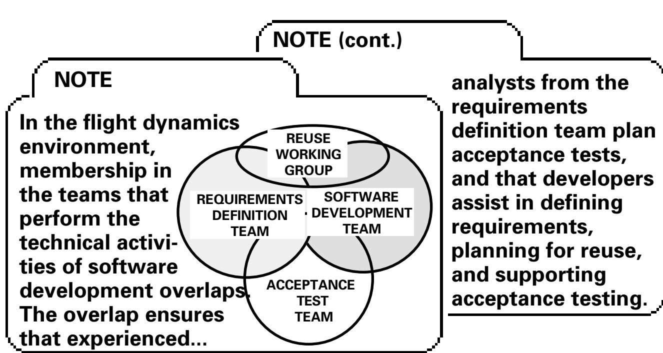
The purpose of the requirements definition phase is to produce a clear, complete, consistent, and testable specification of the technical requirements for the software product.

Requirements definition initiates the software development life cycle. During this phase, the requirements definition team uses an iterative process to expand a broad statement of the system requirements into a complete and detailed specification of each function that the software must perform and each criterion that it must meet. The finished requirements and specifications, combined with the system and operations concept, describe the software product in sufficient detail so that independent software developers can build the required system correctly.

The starting point is usually a set of high-level requirements from the customer that describe the project or problem. For mission support systems, these requirements are extracted from project documentation such as the system instrumentation requirements document (SIRD) and the system operations requirements document (SORD). For internal tools, high-level requirements are often simply a list of the capabilities that the tool is to provide.

In either case, the requirements definition team formulates an overall concept for the system by examining the high-level requirements for similarities to previous missions or systems, identifying existing software that can be reused, and developing a preliminary system architecture. The team then defines scenarios showing how the system will be operated, publishes the system and operations concept document, and conducts a system concept review (SCR). (See Figure 3-1.)

Following the SCR, the team derives detailed requirements for the system from the high-level requirements and the system and operations concept. Using structured or object-oriented analysis, the team specifies the software functions and algorithms needed to satisfy each detailed requirement.



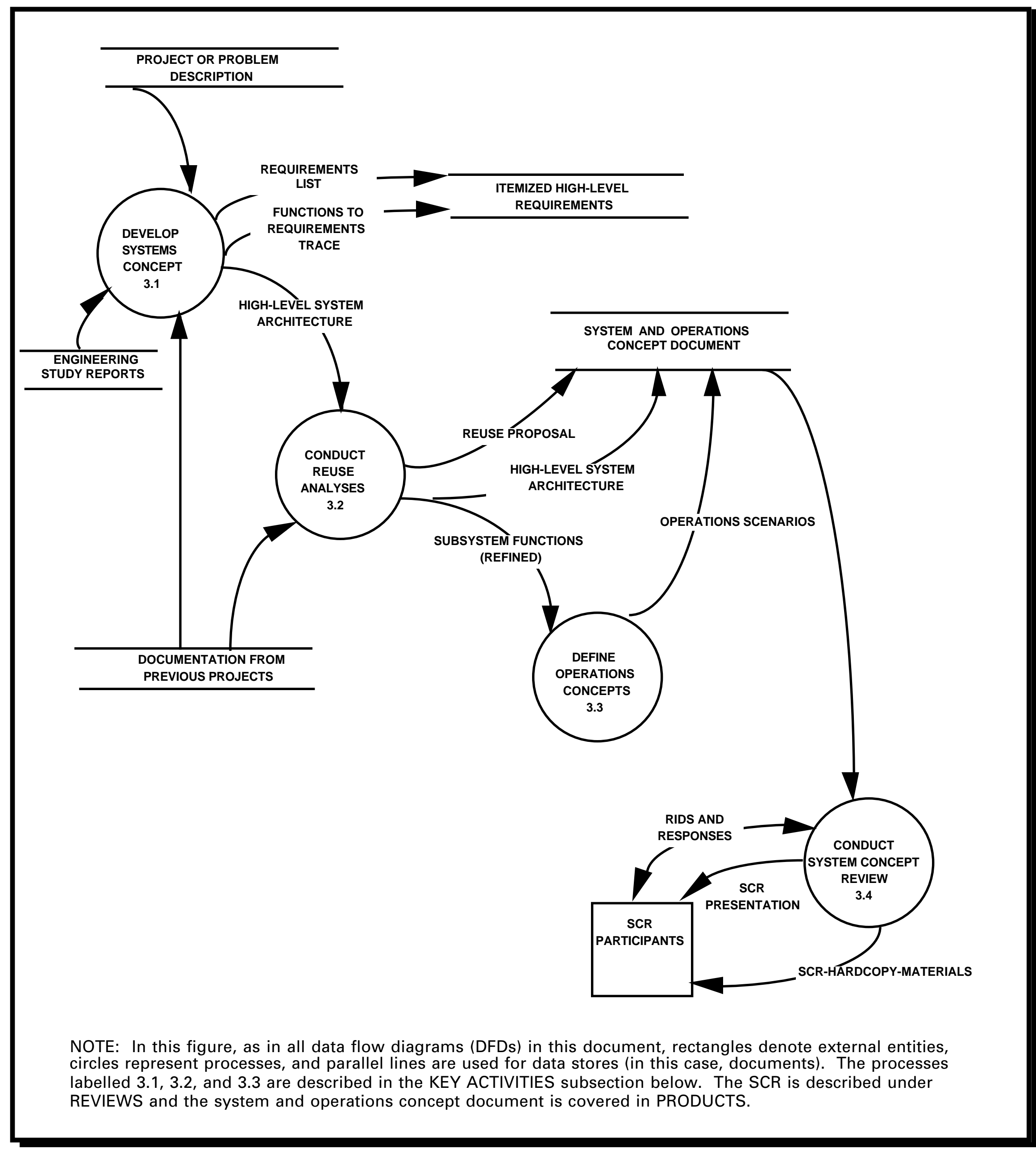


Figure 3-1. Generating the System and Operations Concept

When the specifications are complete, the requirements definition team publishes the requirements and specifications document in three parts: (1) the detailed requirements, (2) the functional or object-oriented specifications, and (3) any necessary mathematical background information. At the end of the phase, the team conducts a system requirements review (SRR) to demonstrate the completeness and quality of these products. (See Figure 3-2.)

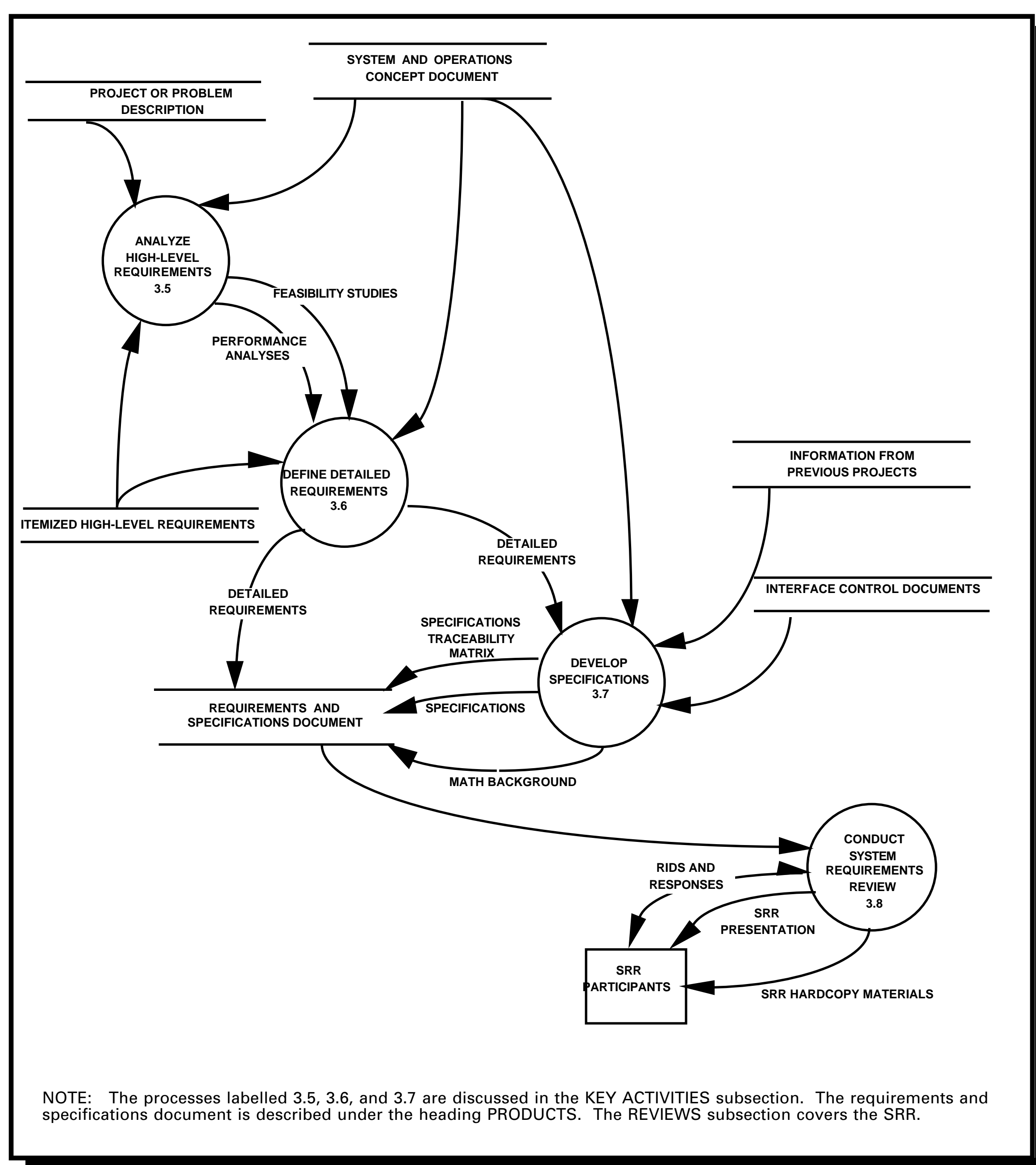


Figure 3-2. Developing Requirements and Specifications

KEY ACTIVITIES

The key technical and managerial activities of the requirements definition phase are itemized below.

Activities of the Requirements Definition Team

- **Develop a system concept.** Collect and itemize all high-level requirements for the system. Describe the basic functions that the system must perform to satisfy these high-level requirements. Address issues such as system lifetime (usage timelines), performance, security, reliability, safety, and data volume.

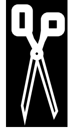
  From this functional description, generate an ideal, high-level system architecture identifying software programs and all major interfaces. Allocate each high-level requirement to software, hardware, or a person. Specify the form (file, display, printout) of all major data interfaces.

- **Prepare the reuse proposal.** Review the requirements and specifications, system descriptions, user's guides, and source code of related, existing systems to identify candidates for reuse. For flight dynamics mission support systems, this involves reviewing support systems for similar spacecraft. Select strong candidates and estimate the corresponding cost and reliability benefits. Determine what compromises are necessary to reuse software and analyze the tradeoffs.

  Adjust the high-level architecture to account for reuseable software. Record the results of all reuse analysis in a *reuse proposal* that will be included in the system and operations concept document.

- **Develop an operations concept.** This clearly defines how the system must operate within its environment. Include operational scenarios for all major modes of operation (e.g., emergency versus normal). Be sure to include the end-user in this process. Conduct an SCR.

TAILORING NOTE

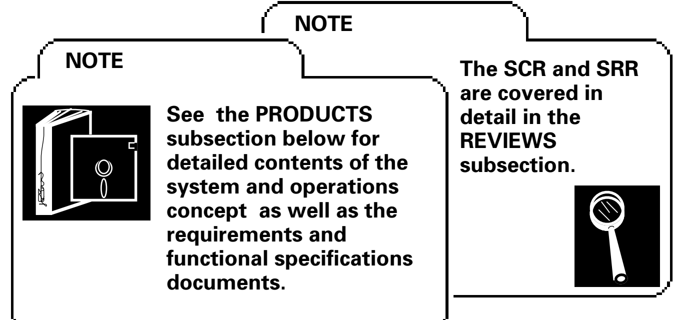


On small projects that are developing tools or prototypes, requirements definition and analysis are often combined into a single phase. On such projects, developers generally perform all requirements definition activities.

REUSE NOTE

Although use of existing software can reduce effort significantly, some compromises may be necessary. Ensure that all tradeoffs are well understood. Avoid these two pitfalls:
- **Failing to make reasonable compromises, thus wasting effort for marginal improvement in quality or functionality**
- **Making ill-advised compromises that save development effort at the cost of significantly degrading functionality or performance**

- **Define the detailed requirements.** Based on the high-level requirements and the system concept and architecture, define all software requirements down to the subsystem level. If the system is large (with many subsystems) or if it will interface with other systems, explicitly define all external interfaces.



  Determine system performance and reliability requirements. If certain acceptance criteria apply to a requirement (e.g., meeting a particular response time), specify the test criteria with the requirement. Identify all intermediate products needed to acceptance test the system.

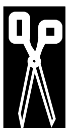
- **Derive the functional specifications** for the system from the requirements. Identify the primary input and output data needed to satisfy the requirements. Use structured or object-oriented analysis to derive the low-level functions and algorithms the software must perform. Define all reports and displays and indicate which data the user must be able to modify.

  Keep the specifications design-neutral and language-neutral; i.e., concentrate on *what* the software needs to do, rather than *how* it will do it. Create a traceability matrix to map each low-level function or data specification to the requirements it fulfills.

  Ensure that all requirements and specifications are given a thorough peer review. Watch for interface problems among major functions and for specifications that are duplicated in multiple subsystems. Ensure compatibility and consistency in notation and level of accuracy among the specified algorithms.

  Prepare the requirements and specifications document, including any necessary mathematical background information, as a basis for beginning software development.

TAILORING NOTE



On very large or complex projects, it is generally advisable to hold a preliminary system requirements review (PSRR) as soon as a draft of the requirements document is complete. This allows end-users and key developers to raise critical issues before requirements are finalized. See the REVIEWS subsection for additional information on the PSRR.

- **Conduct the SRR** and incorporate approved changes into the requirements and specifications. Place the document under configuration management as the system baseline.

Activities of the Management Team

The management activities performed during this phase pave the way for all future phases of the project's life cycle. Specifically, managers must accomplish the following:

- **Develop a plan for the phase.** (Detailed planning of the entire development effort is deferred to the requirements analysis phase, after system specifications have been defined.) Address the staffing of the teams that will perform the technical work, the groups and individuals that will interface with the teams, the technical approach, milestones and schedules, risk management, and quality assurance. List the reviews to be conducted and their level of formality.

- **Staff and train the requirements definition team.** Ensure that the team contains the necessary mix of skills and experience for the task. For mission support systems, the team should include analysts with strong backgrounds in mission analysis, attitude and orbit determination, and operations. The reuse working group must include key software developers as well as experienced analysts. Ensure that staff members have the necessary training in the procedures, methods, and tools needed to accomplish their goals.

- **Interact with the customer** to assure visibility and resolution of all issues. Conduct regular status meetings and ensure communications among team members, managers, customers, and other groups working on aspects of the project.

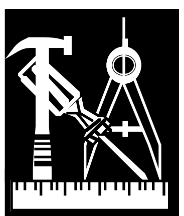
- **Evaluate progress and products**. Review the system and operations concept and the requirements and specifications. Collect progress measures and monitor adherence to schedules and cost.

- **Control major reviews**. Ensure that key personnel are present at reviews, both formal and informal. Participate in the SCR and SRR.

DEFINITION

The key developers who participate in reuse analysis and other requirements definition activities have special technical roles throughout the life cycle. The value of these *application specialists* lies in their specific knowledge and experience. On mission support projects, for example, the application specialist will not only have developed such software previously, but also will understand the complex mathematics and physics of flight dynamics. The application specialist often acts as a "translator," facilitating communications between analysts and developers.

METHODS AND TOOLS



The methods and tools used during the requirements definition phase are

- Structured or object-oriented analysis
- Walk-throughs
- Prototyping

Each is discussed below.

Analysis Methodologies

Structured analysis and *object-oriented analysis* are techniques used to understand and articulate the implications of the textual statements found in the requirements definition. The requirements definition team uses analysis techniques to derive the detailed specifications for the system from the higher-level requirements. The analysis methodology selected for the project should be appropriate to the type of problem the system addresses.

structured analysis

Functional decomposition is currently the most commonly used method of structured analysis. Functional decomposition focuses on processes, each of which represents a set of transformations of input to output. Using this method, the analyst separates the primary system function into successively more detailed levels of processes and defines the data flows between these processes. Authors associated with structured analysis include E. Yourdon, L.Constantine, and T. DeMarco (References 13 and 14). S. Mellor and P. Ward have published a set of real-time extensions to this method for event-response analysis (Reference 15).

object-oriented analysis

Object-oriented analysis combines techniques from the realm of data engineering with a process orientation. This method defines the objects (or entities) and attributes of the real-world problem domain and their interrelationships. The concept of an object provides a means of focusing on the persistent aspect of entities — an emphasis different from that of structured analysis. An object-oriented approach is appropriate for software designed for reuse because specific objects can be readily extracted and replaced to adapt the system for other tasks (e.g., a different spacecraft). Details of the object-oriented approach may be found in References 11, 16, and 17.

In structured analysis, functions are grouped together if they are steps in the execution of a higher-level function. In object-oriented analysis, functions are grouped together if they operate on the same data abstraction. Because of this difference, proceeding from functional specifications to an object-oriented design may necessitate

recasting the data flow diagrams. This is a significant amount of effort that can be avoided by assuming an object-oriented viewpoint during the requirements definition phase.

> **NOTE**
>
> **CASE tools can greatly increase productivity, but they can only aid or improve those activities that the team or individual knows how to perform manually. CASE tools cannot improve analysis, qualify designs or code, etc., if the user does not have have a clear definition of the manual process involved.**

The diagramming capabilities of CASE tools facilitate application of the chosen analysis methodology. The tools provide a means of producing and maintaining the necessary data flow and object-diagrams online. They usually include a centralized repository for storing and retrieving definitions of data, processes, and entities. Advanced tools may allow the specifications themselves to be maintained in the repository, making it easier to trace the requirements to design elements.

Selected tools should be capable of printing the diagrams in a form that can be directly integrated into specifications and other documents. Examples of CASE tools currently used in the flight dynamics environment include System Architect and Software Through Pictures.

Walk-throughs

In all phases of the life cycle, peer review ensures the quality and consistency of the products being generated. The SEL recommends two types of peer review — walk-throughs and inspections — in addition to formal reviews such as the SRR and CDR.

walk-throughs

Walk-throughs are primarily conducted as an aid to understanding, so participants are encouraged to analyze and question the material under discussion. Review materials are distributed to participants prior to the meeting. During the meeting, the walk-through leader gives a brief, tutorial overview of the product, then walks the reviewers through the materials step-by-step. An informal atmosphere and a free interchange of questions and answers among participants fosters the learning process.

inspections

Inspections, on the other hand, are designed to uncover errors as early as possible and to ensure a high-quality product. The inspection team is a small group of peers who are technically competent and familiar with the application, language, and standards used on the project. The products to be reviewed (e.g., requirements, design diagrams, or source code) are given to the inspection team several days before the meeting. Inspectors examine these materials closely, noting all errors or deviations from

standards, and they come to the review meeting prepared to itemize and discuss any problems.

In both walk-throughs and inspections, a designated team member records the minutes of the review session, including issues raised, action items assigned, and completion schedules. Closure of these items is addressed in subsequent meetings.

In the requirements definition phase, walk-throughs of the requirements and specifications are conducted to ensure that key interested parties provide input while requirements are in a formative stage. Participants include the members of the requirements definition team, representatives of systems that will interface with the software to be developed, and application specialists from the development team.

Prototyping

During the requirements definition phase, prototyping may be needed to help resolve requirements issues. For mission support systems, analysts use prototyping tools such as MathCAD to test the mathematical algorithms that will be included in the specifications. For performance requirements, platform-specific performance models or measurement/monitoring tools may be used.

MEASURES

Objective Measures

Three progress measures are tracked during the requirements definition phase:

- Staff hours — i.e., the cumulative effort hours of the project staff
- Number of requirements with completed specifications versus the total number of requirements
- Number of requirements defined versus the total number of estimated requirements

The sources of these data and the frequency with which the data are collected and analyzed are shown in Table 3-1.

Table 3-1. Objective Measures Collected During the Requirements Definition Phase

MEASURE	SOURCE	FREQUENCY (COLLECT/ANALYZE)
Staff hours (total and by activity)	Requirements definition team and managers (time accounting)	Weekly/monthly
Requirements status (percentage of completed specifications; number of requirements defined)	Managers	Biweekly/biweekly
Estimates of total requirements, total requirements definition effort, and schedule	Managers	Monthly/monthly

Evaluation Criteria

staff hours

Effort should be gauged against estimates based on historical data from past projects of a similar nature. Monitor staff hours separately for each major activity. If schedules are being met but hours are lower than expected, the team may not be working at the level of detail necessary to raise problems and issues.

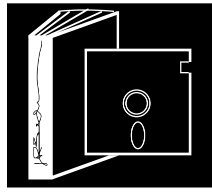
completed specifications

To judge progress following the SCR, track the number of requirements for which specifications have been written as a percentage of the total number of requirements. ("Total requirements" includes those for which a need has been identified, but for which details are still TBD.)

defined requirements

Monitor requirements growth by tracking the number of requirements that have been defined against an estimated total for the project. If requirements stability is an issue, consider tracking the number of changes made to requirements as well. Excessive growth or change to specifications point to a need for greater management control or to the lack of a detailed system operations concept.

PRODUCTS



The key products of the requirements definition phase are the system and operations concept (SOC) document and the requirements and specifications document. The content and form of these products are addressed in the following paragraphs.

System and Operations Concept Document

The SOC document lists the high-level requirements, defines the overall system architecture and its operational environment, and describes how the system will operate within this environment. The document provides a base from which developers can create the software structure and user interface. The format recommended for the document is shown in Figure 3-3.

The SOC is not usually updated after publication. During the requirements analysis phase, developers refine the reuse proposal contained in the document and publish the resulting reuse plan in the requirements analysis report. Similarly, developers refine the operational scenarios and include them in the requirements analysis, preliminary design, and detailed design reports. Because these and other pieces of the SOC are reworked and included in subsequent development products, it may not be necessary to baseline or maintain the SOC itself.

Requirements and Specifications Document

This document is produced by the requirements definition team as the key product of the requirements definition phase. It is often published in multiple volumes: volume 1 defines the requirements, volume 2 contains the functional specifications, and volume 3 provides mathematical specifications. The document is distributed prior to the SRR, updated following the review to incorporate approved review items, and then baselined.

The requirements and specifications document contains a complete list of all requirements —including low-level, derived requirements — and provides the criteria against which the software system will be acceptance tested. The functional or object specifications, which identify the input data, output data, and processing required to transform input to output for each process, provide the basis for detailed design and system testing. The document also includes the mathematical background information necessary to evaluate specified algorithms and to design the system correctly.

The recommended outline for the requirements and specifications document is presented in Figure 3-4.

SYSTEM AND OPERATIONS CONCEPT DOCUMENT

This document provides a top-down view of the system from the user's perspective by describing the behavior of the system in terms of operational methods and scenarios. Analysts should provide the document to the development team by the end of the requirements definition phase. The suggested contents are as follows:

1. **Introduction**
   a. Purpose and background of the system
   b. Document organization

2. **System overview**
   a. Overall system concept
   b. System overview with high-level diagrams showing external interfaces and data flow
   c. Discussion and diagrams showing an ideal, high-level architecture for the system

3. **Reuse proposal**
   a. Summary of domain and reuse analysis performed
   b. Description of potential candidates for reuse — architectural components, designs, operational processes, and test approaches — and associated trade-offs
   c. Discussion and diagrams of the proposed high-level architecture, as adjusted to incorporate reusable elements

4. **Operational environment** — description and high-level diagrams of the environment in which the system will be operated
   a. Overview of operating scenarios
   b. Description and high-level diagrams of the system configuration (hardware and software)
   c. Description of the responsibilities of the operations personnel

5. **Operational modes**
   a. Discussion of the system's modes of operation (e.g., critical versus normal and launch/early mission versus on-orbit operations)
   b. Volume and frequency of data to be processed in each mode
   c. Order, frequency, and type (e.g., batch or interactive) of operations in each mode

6. **Operational description of each major function or object** in the system
   a. Description and high-level diagrams of each major operational scenario showing all input, output, and critical control sequences
   b. Description of the input data, including the format and limitations of the input. Sample screens (i.e., displays, menus, popup windows) depicting the state of the function before receiving the input data should also be included.
   c. Process — high-level description of how this function will work
   d. Description of the output data, including the format and limitations of the output. Samples (i.e., displays, reports, screens, plots) showing the results after processing the input should also be included.
   e. Description of status and prompt messages needed during processing, including guidelines for user responses to any critical messages

7. **Requirements traceability matrix** mapping each operational scenario to requirements

Figure 3-3. SOC Document Contents

REQUIREMENTS AND SPECIFICATIONS DOCUMENT

This document, which contains a complete description of the requirements for the software system, is the primary product of the requirements definition phase. In the flight dynamics environment, it is usually published in three volumes: volume 1 lists the requirements, volume 2 contains the functional specifications, and volume 3 provides the mathematical specifications.

1. **Introduction**
   a. Purpose and background of the project
   b. Document organization

2. **System overview**
   a. Overall system concept
   b. Expected operational environment (hardware, peripherals, etc.)
   c. High-level diagrams of the system showing the external interfaces and data flows
   d. Overview of high-level requirements

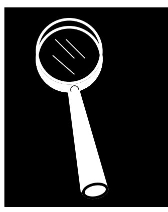
3. **Requirements** — functional, operational (interface, resource, performance, reliability, safety, security), and data requirements
   a. Numbered list of high-level requirements with their respective derived requirements (derived requirements are not explicitly called out in source documents such as the SIRD or SORD, but represent constraints, limitations, or implications that must be satisfied to achieve the explicitly stated requirements)
   b. For each requirement:
      (1) Requirement number and name
      (2) Description of the requirement
      (3) Reference source for the requirement, distinguishing derived from explicit requirements
      (4) Interfaces to other major functions or external entities
      (5) Performance specifications — frequency, response time, accuracy, etc.

4. **Specifications**
   a. Discussion and diagrams showing the functional or object hierarchy of the system
   b. Description and data flow/object diagrams of the basic processes in each major subsystem
   c. Description of general conventions used (mathematical symbols, units of measure, etc.)
   d. Description of each basic function/object, e.g.:
      (1) Function number and name
      (2) Input
      (3) Process — detailed description of what the function should do
      (4) Output
      (5) Identification of candidate reusable software
      (6) Acceptance criteria for verifying satisfaction of related requirements
      (7) Data dictionary — indicating name of item, definition, structural composition of the item, item range, item type

5. **Mapping of specifications to requirements** — also distinguishes project-unique requirements from standard requirements for the project type (AGSS, dynamics simulator, etc.)

6. **Mathematical specifications** — formulas and algorithm descriptions to be used in implementing the computational functions of the system
   a. Overview of each major algorithm
   b. Detailed formulas for each major algorithm

Figure 3-4. Requirements and Specifications Document Contents



REVIEWS

Two key reviews are conducted during the requirements definition phase: the system concept review and the system requirements review. The purpose, participants, scheduling, content, and format of these reviews are discussed in the subsections that follow.

System Concept Review

The SCR gives users, customer representatives, and other interested parties an opportunity to examine and influence the proposed system architecture and operations concept before detailed requirements are written. It is held during the requirements definition phase after system and operations concepts have been defined. In the flight dynamics environment, a full SCR is conducted for large, mission support systems. For smaller development efforts without complex external interfaces, SCR material is presented informally. The SCR format is given in Figure 3-5.

SCR FORMAT

Presenters — requirements definition team

Participants
- Customer representatives
- User representatives
- Representatives of systems and groups that will interface with the system to be developed
- Senior development team representatives (application specialists)
- Quality assurance (QA) representatives
- System capacity/performance analysts

Schedule — after the system and operations document is complete and before requirements definition begins

Agenda — summary of high-level requirements (e.g., from SIRD and SORD) and presentation of system and operations concepts; interactive participation and discussion should be encouraged.

Materials Distribution
- The system and operations concept document is distributed 1 to 2 weeks before the SCR.
- Hardcopy material is distributed a minimum of 3 days before SCR.

Figure 3-5. SCR Format

SCR Hardcopy Material

The hardcopy materials distributed for use at the review should correspond to the presentation viewgraphs. A suggested outline for the contents of SCR hardcopy material is presented in Figure 3-6.

HARDCOPY MATERIAL FOR THE SCR

1. **Agenda** — outline of review material
2. **Introduction** — purpose of system and background of the project
3. **High-level requirements**
   a. Derivation of high-level requirements — identification of input (such as the SIRD and SORD) from project office, support organization, and system engineering organization
   b. Summary of high-level requirements
4. **System concept**
   a. Assumptions
   b. Overall system concept
   c. List of major system capabilities
5. **Reuse considerations**
   a. Existing systems reviewed for possible reuse
   b. Reuse trade-offs analyzed
   c. Proposed reuse candidates
6. **High-level system architecture**
   a. Description and high-level diagrams of the proposed system architecture (hardware and software), including external interfaces and data flow
   b. Diagrams showing the high-level functions of the system — their hierarchy and interaction
7. **System environment**
   a. Computers and peripherals
   b. Communications
   c. Operating system limitations and other constraints
8. **Operations concept**
   a. Assumptions
   b. Organizations that provide system and support input and receive system output
   c. System modes of operation (e.g., critical versus normal and launch versus on-orbit operations)
   d. Order, frequency, and type (e.g., batch or interactive) of operations in each mode
   e. Discussion and high-level diagrams of major operational scenarios
   f. Performance implications
9. **Issues, TBD items, and problems** — outstanding issues and TBDs and a course of action to handle them

Figure 3-6. SCR Hardcopy Material Contents

System Requirements Review (SRR)

When the requirements and specifications document is distributed, the requirements definition team conducts an SRR to present the requirements, obtain feedback, and facilitate resolution of outstanding issues. The SRR format, schedule, and participants are given in Figure 3-7.

SRR FORMAT

Presenters — requirements definition team

Participants
- Customer representatives
- User representatives
- Configuration Control Board (CCB)
- Senior development team representatives
- System capacity/performance analysts
- Quality assurance representatives

Schedule — after requirements definition is complete and before the requirements analysis phase begins

Agenda — selective presentation of system requirements, highlighting operations concepts and critical issues (e.g., TBD requirements)

Materials Distribution
- The requirements and specifications document is distributed 1 to 2 weeks before SRR.
- Hardcopy material is distributed a minimum of 3 days before SRR.

Figure 3-7. SRR Format

TAILORING NOTE

For very large or complex projects, hold a preliminary SRR to obtain interim feedback from users and customers. The format of the PSRR is the same as the SRR. Hardcopy material contains preliminary results and is adjusted to reflect work accomplished to date.

DEFINITION

The *Configuration Control Board (CCB)* is a NASA/GSFC committee that reviews, controls, and approves FDD systems, internal interfaces, and system changes. Among its duties are the approval of system baseline reviews (e.g., SRR, PDR) and baseline documents (e.g., requirements and specifications, detailed design document).

SRR Hardcopy Material

Much of the hardcopy material for the review can be extracted from the requirements and specifications document. An outline and suggested contents of the SRR hardcopy material are presented in Figure 3-8.

HARDCOPY MATERIAL FOR THE SRR

1. **Agenda** — outline of review material
2. **Introduction** — purpose of system and background of the project
3. **Requirements summary** — review of top-level (basic) requirements developed to form the specifications
   a. Background of requirements — overview of project characteristics and major events
   b. Derivation of requirements — identification of input from project office, support organization, and system engineering organization used to formulate the requirements: e.g., the SIRD, SORD, memoranda of information (MOIs), and memoranda of understanding (MOUs)
   c. Relationship of requirements to level of support provided — typical support, critical support, and special or contingency support
   d. Organizations that provide system and support input and receive system output
   e. Data availability — frequency, volume, and format
   f. Facilities — target computing hardware and environment characteristics
   g. Requirements for computer storage, failure/recovery, operator interaction, diagnostic output, security, reliability, and safety
   h. Requirements for support and test software — data simulators, test programs, and utilities
   i. Overview of the requirements and specifications document — its evolution, including draft dates and reviews and outline of contents
4. **Interface requirements** — summary of human, special-purpose hardware, and automated system interfaces, including references to interface agreement documents (IADs) and interface control documents (ICDs)
5. **Performance requirements** — system processing speed, system response time, system failure recovery time, and output data availability
6. **Environmental considerations** — special computing capabilities, e.g., graphics, operating system limitations, computer facility operating procedures and policies, support software limitations, database constraints, and resource limitations
7. **Derived system requirements** — list of those requirements not explicitly called out in the SIRD, SORD, MOIs, and MOUs, but representing constraints, limitations, or implications that must be satisfied to achieve the explicitly stated requirements
8. **Operations concepts**
   a. High-level diagrams of operating scenarios showing intended system behavior from the user's viewpoint
   b. Sample input screens and menus; sample output displays, reports, and plots; critical control sequences
9. **Requirements management approach**
   a. Description of controlled documents, including scheduled updates
   b. Specifications/requirements change-control procedures
   c. System enhancement/maintenance procedures
10. **Personnel organization and interfaces**
11. **Milestones and suggested development schedule**
12. **Issues, TBD items, and problems** — a characterization of all outstanding requirements issues and TBDs, an assessment of their risks (including the effect on progress), and a course of action to manage them, including required effort, schedule, and cost

Figure 3-8. SRR Hardcopy Material Contents

EXIT CRITERIA

Following the SRR, the requirements definition team analyzes all RIDs, determines whether requirements changes are necessary, and revises the requirements and specifications document accordingly. The updated document is sent to the configuration control board (CCB) for approval. Once approved, it becomes a controlled document — the requirements *baseline*.

Use the following questions to determine whether the requirements and specifications are ready to be given to the development team for analysis:

- Do specifications exist for all requirements for which information is available? Have TBD requirements been minimized?
- Have external interfaces been adequately defined?
- Are the specifications consistent in notation, terminology, and level of functionality, and are the algorithms compatible?
- Are the requirements testable?
- Have key exit criteria been met? That is, has the requirements and specifications document been distributed, has the SRR been successfully completed, and have all SRR RIDs been answered?

When the answer to these questions is "yes," the requirements definition phase is complete.

NOTE

During and following formal reviews, review item disposition forms (RIDs) are submitted by participants to identify issues requiring a written response or further action. Managers are responsible for ensuring that all RIDs are logged and answered and resulting action items are assigned and completed.

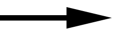
LIFE CYCLE PHASES	REQUIREMENTS DEFINITION	REQUIRE-MENTS ANALYSIS	PRELIMI-NARY DESIGN	DETAILED DESIGN	IMPLEMENTATION	SYSTEM TESTING	ACCEPTANCE TESTING

SECTION 4

THE REQUIREMENTS ANALYSIS PHASE

PHASE HIGHLIGHTS

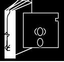
ENTRY CRITERIA



- System and operations concept completed
- SRR completed
- Requirements and specifications baselined

EXIT CRITERIA

- Requirements analysis report completed
- Software specification review (SSR) completed
- SSR RIDs resolved



PRODUCTS

- Requirements analysis report
- Software development/management plan
- Updated requirements and specifications

MEASURES

- Staff hours
- TBD requirements
- Requirements questions/answers
- Requirements changes
- Estimates of system size, effort, and schedule

METHODS AND TOOLS

- Requirements walk-throughs *
- Requirements classification *
- Requirements forms
- Requirements analysis methods and CASE tools
- Prototyping
- Project library

KEY ACTIVITIES

Requirements Definition Team
- Resolve ambiguities, discrepancies, and TBDs in the specifications
- Participate in the SSR

Development Team
- Analyze and classify requirements
- Refine the reuse proposal
- Identify technical risks
- Prepare the requirements analysis report
- Conduct the SSR

Management Team
- Prepare the software development/ management plan
- Staff and train the development team
- Interact with analysts and customers to facilitate resolution of requirements issues
- Review the products of the requirements analysis process
- Plan the transition to preliminary design

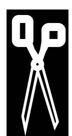
* The guiding philosophy of the Cleanroom method is intellectual control and reliance on the human mind to build a quality product the first time. Walk-throughs and requirements classifications help intensify scrutiny and analysis of requirements early in the life cycle so that requirements modifications during subsequent phases are minimized.

OVERVIEW

The purpose of the requirements analysis phase is to ensure that the requirements and specifications are feasible, complete, and consistent, and that they are understood by the development team.

Requirements analysis begins after the requirements definition team completes the requirements and specifications and holds the SRR. During requirements analysis, members of the development team carefully study the requirements and specifications document. They itemize and categorize each statement in the document to uncover omissions, contradictions, TBDs, and specifications that need clarification or amplification. The development team takes the analysis that was performed in the requirements definition phase to the next level of detail, using the appropriate analysis methodology for the project (e.g., structured or object-oriented analysis). When analysis is complete, the team presents its findings at an SSR.

TAILORING NOTE



On large projects, requirements analysis begins at the PSRR. Key members of the development team examine the review materials, participate in the review itself, and begin classifying requirements shortly thereafter.

The development team works closely with the requirements definition team during the entire phase. The requirements definition team participates in walk-throughs, answers questions, resolves requirements issues, and attends the SSR. Meanwhile, the project manager plans the approaches to be used in developing the software system and in managing the development effort, obtains and trains the necessary staff, and reviews the products produced during the phase.

NOTE

A typical development team comprises
- **the project manager, who manages project resources, monitors progress, and serves as a technical consultant**
- **the project (or task) leader, who provides technical direction and daily supervision**
- **the programmers and application specialists who perform the technical work**
- **a quality assurance representative**
- **a project librarian (see METHODS & TOOLS)**

Figure 4-1 is a high-level data flow diagram of the requirements analysis process.

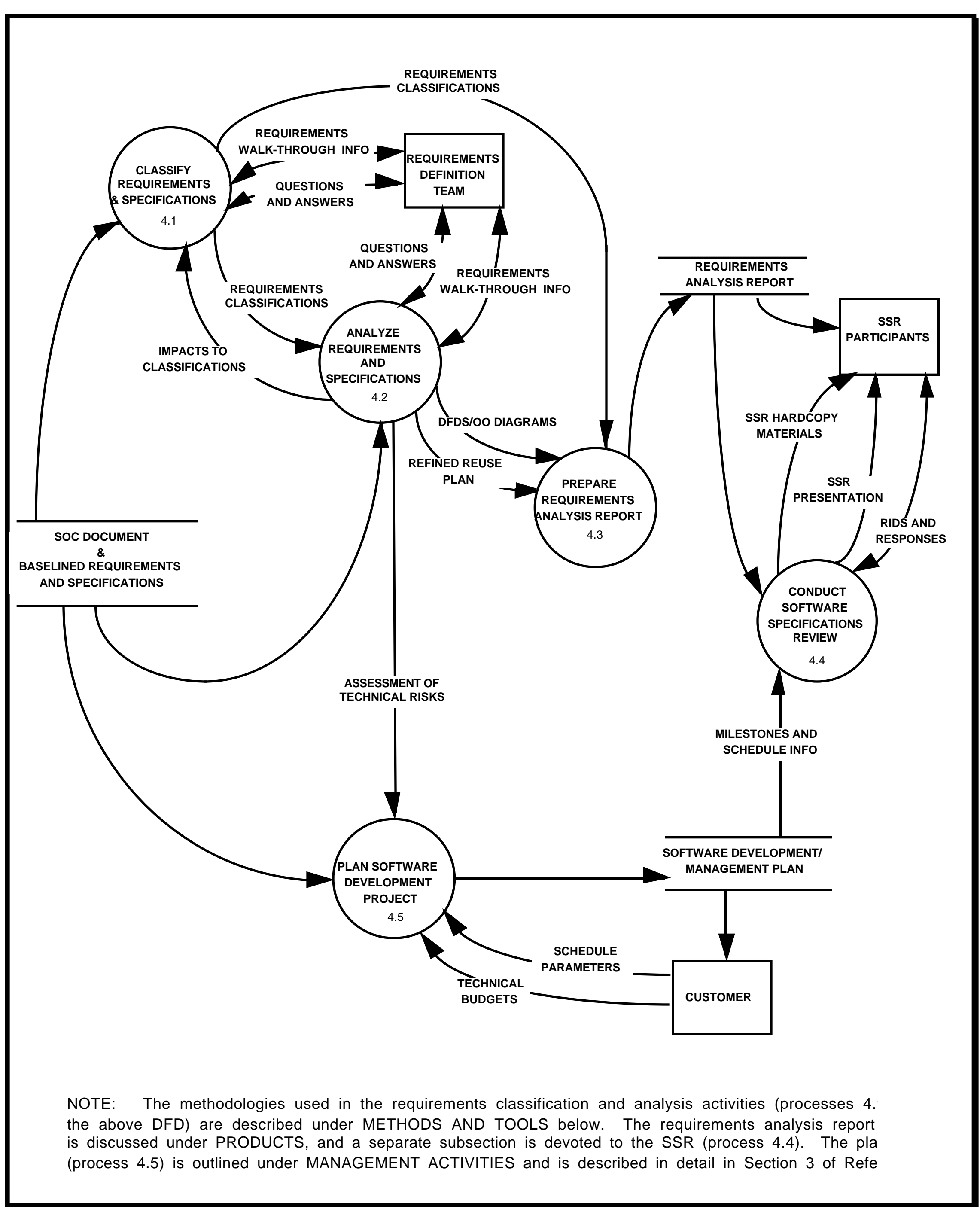


Figure 4-1. Analyzing Requirements

KEY ACTIVITIES

In the requirements analysis phase, activities are divided primarily among the requirements definition team, the development team, and software development managers. The key activities that each performs during the requirements analysis phase are itemized in the following subsections. Figure 4-2 is a sample timeline showing how these activities are typically scheduled.

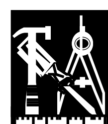
Activities of the Requirements Definition Team

- Resolve ambiguities, discrepancies, and TBDs in the specifications. Conduct the initial walk-throughs of the requirements and specifications for the developmentteam and participate in later walk-throughs. Respond to all developer questions.

  Resolve the requirements issues raised by the development team. Incorporate approved modifications into the requirements and specifications document.

- Participate in the SSR.

Activities of the Development Team



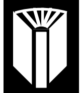
See METHODS AND TOOLS below for more information about walk-throughs, requirements classifications, and analysis methodologies.

- Analyze and classify requirements. Meet with the requirements definition team to walk through and clarify each requirement and specification. Identify requirements and specifications that are missing, conflicting, ambiguous, or infeasible. Assign a classification of *mandatory, requires review, needs clarification, information only,* or *TBD* to each item in the requirements and specifications document.

  Use structured or object-oriented analysis to verify the specifications. Expand the high-level diagrams in the requirements and specifications document to a lower level of detail, and supply missing diagrams so that all specifications are represented at the same level. Ensure that user interactions and major data stores (e.g., attitude history files) are completely specified.

  Determine the feasibility of computer capacity and performance requirements in view of available resources. Establish initial performance estimates by comparing specified

NOTE



The contents of the requirements analysis report and the software development/management plan are covered under PRODUCTS below. The SSR is discussed separately at the end of this section.

REFERENCE

See the *Manager's Handbook for Software Development* and the *Approach to Software Cost Estimation* (References 12 and 18, respectively) for guidance in estimating project size, costs , and schedule.

functions/algorithms with those of existing systems. Use the estimates to model overall performance (CPU, I/O, etc.) for the operational scenarios described in the SOC. Adjust the SOC scenarios to take these results into account.

Walk through the results of classification and analysis with the requirements definition team.

- Refine the reuse proposal into a realistic plan. Analyze the software reuse proposal in the SOC in light of the existing software's current operational capabilities and any changes to the requirements baseline.

- Identify areas of technical risk. Plan and conduct prototyping efforts or other appropriate techniques to minimize these risks.

- Prepare the requirements analysis report and distribute it before the SSR.

- Conduct the SSR and resolve all RIDs.

Activities of the Management Team

- Prepare the software development/management plan (SDMP). Review the histories of related, past projects for applicable size, cost, and schedule data as well as lessons learned. Determine what resources are needed, develop a staffing profile, and estimate project costs. Identify project risks and plan to minimize them. Document the technical and management approaches that will be used on the project.

- Staff and train the development team. Bring staff onto the project as soon as possible following SRR (or, on large projects, PSRR). Ensure communications among development team members, managers, and the requirements definition team.

  Also make certain that the requirements definition team is adequately staffed following SRR, so that TBD and changing requirements can be given prompt and thorough analysis.

- Interact with analysts and customers to facilitate resolution of requirements issues. Work with team leaders to assess the

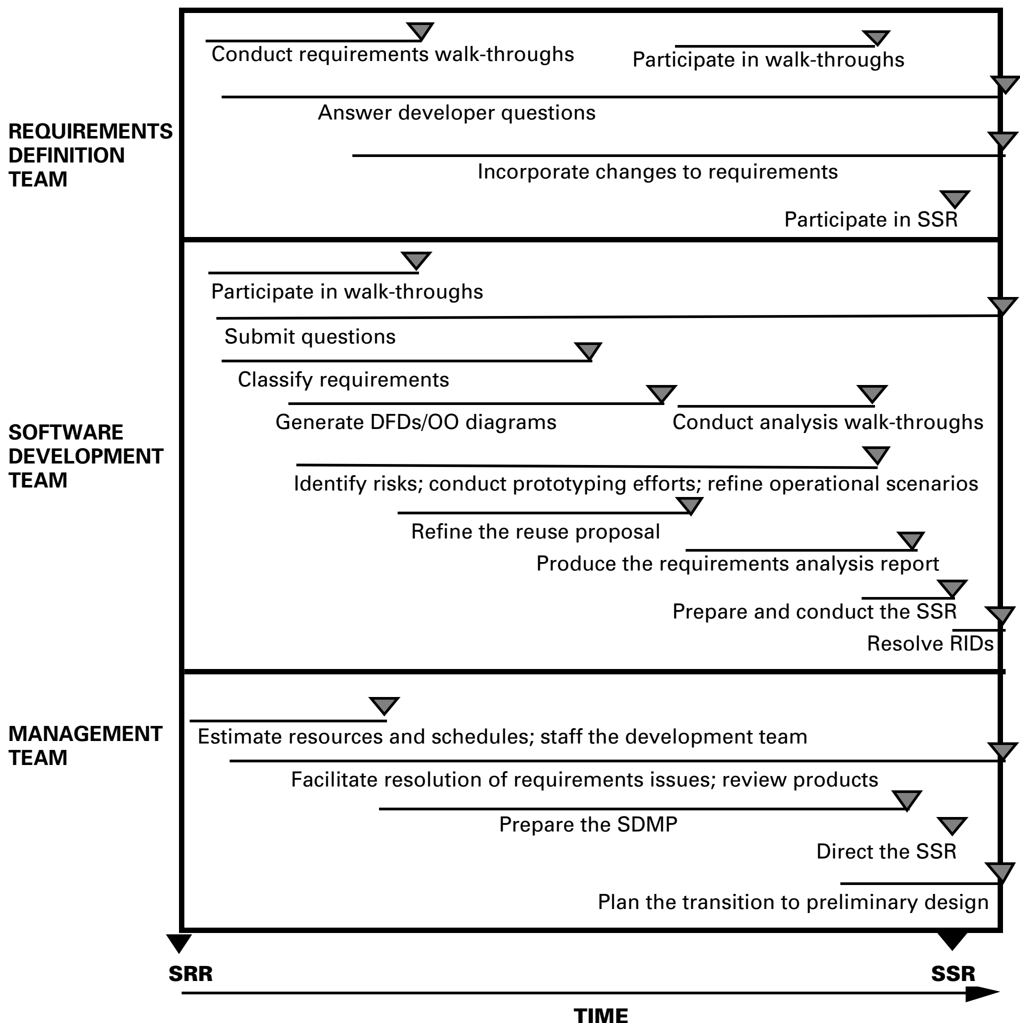
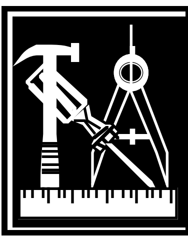


Figure 4-2: Timeline of Key Activities in the Requirements Analysis Phase

feasibility of proposed requirements changes and to estimate their impact on costs and schedules.

- Review the products of the requirements analysis process. Look at requirements classifications, data-flow or object-oriented diagrams, the data dictionary, the requirements analysis report, and the SSR hardcopy materials. Schedule the SSR and ensure participation from the appropriate groups.

- Plan an orderly transition to the preliminary design phase. Convey to the development team members the parts of the software development plan that apply to preliminary design (e.g., design standards and configuration management procedures) and instruct them in the specific software engineering approach to use during design. While the key team members are preparing for SSR, have the remainder of the development team begin preliminary design activities.



METHODS AND TOOLS

The following methods, techniques, and tools are used to support the activities of the requirements analysis phase:

- Requirements walk-throughs
- Requirements classifications
- Requirements forms
- Structured and object-oriented requirements analysis
- CASE tools
- Prototyping
- The project library

Each is discussed below.

Requirements Walk-Throughs

At the beginning of the requirements analysis phase, developers meet informally with analysts of the requirements definition team to go through the requirements and specifications. During these initial walk-throughs, analysts discuss each of the specifications, explain why certain algorithms were selected over others, and give developers the opportunity to raise questions and issues.

After developers have analyzed and classified the requirements and specifications, they conduct walk-throughs of their results for the requirements definition team. One walk-through should be held for each major function or object in the system. During these later walk-throughs, both teams review all problematic specification items and discuss any needed changes to the requirements and specifications document.

To ensure that all problem areas and decisions are documented, one member of the development team should record the minutes of the walk-through meeting. Developers will need the minutes to fill out requirements question-and-answer forms for any issues that require confirmation, further analysis, or other action by the requirements definition team.

Requirements Classification

When the development team is thoroughly familiar with the requirements and specifications document, they take each passage (sentence or paragraph) in the requirements and specifications document and classify it as either *mandatory, requires review, needs clarification, information only*, or *TBD.*

- An item is *mandatory* if it is explicitly defined in project-level requirements documents such as the SIRD or SORD, or if it has been derived from analysis of the project-level requirements. If mandatory items are removed from the specifications, the system will fail to meet project-level requirements.

- If (on the basis of project-level requirements and the system and operations concept) there is no apparent need for a particular requirement or specification, then that item *requires review* (i.e., further analysis by the requirements definition team). The item must be deleted from the specification (by means of a specification modification) or moved into the mandatory category before CDR.

- An item *needs clarification* when it is ambiguous, appears infeasible, or contradicts one or more of the other requirements or specifications.

- An item is labelled as *information only* if it contains no requirement or specification per se. Such an item may provide background information, helpful hints for the software developer, etc.

- A requirement or specification item is *TBD* if (1) the item contains a statement such as "the process is TBD at this time," or (2) information associated with the item is missing or undefined.

HINT

If the requirements and specifications are available in a database, enter the classifications and supporting commentary into the database online. Otherwise, summarize each requirement or specification item, create a list of the summaries, and use the lists to assign classifications.

Requirements Forms

During the requirements analysis and subsequent phases, question-and-answer forms are used to communicate and record requirements issues and clarifications. Specification modifications are used to document requirements changes.

question-and-answer forms

The development team uses question-and-answer forms to track questions submitted to the requirements definition team and to verify their assumptions about requirements. Managers of the requirements definition team use the forms to assign personnel and due dates for their team's response to developers. Responses to questions submitted on the forms must be in writing.

The question-and-answer form cannot be used to authorize changes to requirements or specifications. If analysis of the submitted

specification modifications

question or issue reveals that a requirements change is needed, members of the requirements definition team draft a *specification modification.* Proposed specification modifications must be approved by the managers of both the requirements definition and the development teams and by the CCB. The requirements definition team incorporates all approved specification modifications into the requirements and specifications document.

Analysis Methods and CASE Tools

The methods and tools applicable for requirements analysis are the same as those recommended for the requirements definition phase in Section 3. The development team will generally use the same method as was used by the requirements definition team to take the analysis down to a level of detail below that provided in the specifications. This allows the development team to verify the previous analysis and to fill in any gaps that may exist in the document. If CASE tools were used in the requirements definition phase to generate data flow or object diagrams, it is important to use the same tools in the requirements analysis phase. The value of a CASE tool as a productivity and communication aid is greatly reduced if developers must re-enter or reformat the diagrams for a different tool.

If the requirements definition team has used functional decomposition for their analysis and the development team needs to generate an object-oriented design, then extra analysis steps are required. The development team must diagram the details of the specification at a low level, then use the diagrams to abstract back up to higher-level requirements. This allows the team to take a fresh, object-oriented look at the system architecture and to restructure it as needed.

Prototyping

During the requirements analysis phase, prototyping activities are usually conducted to reduce risk. If unfamiliar technology (e.g., hardware or new development language features) will be employed on the project, prototyping allows the development team to assess the feasibility of the technology early in the life cycle when changes are less costly to effect. If system performance or reliability is a major, unresolved issue, the team can prototype critical operations or algorithms.

On projects where the requirements for the user interface must be prototyped — either because the interface is critical to system success or because users are uncertain of their needs — a tool that

allows the developer to set up screens and windows rapidly is often essential. With such a tool, the developer can give the user the "look and feel" of a system without extensive programming and can obtain early feedback. The tool should be able to generate menus, multiple screens, and windows and respond to input. One such tool that has been successfully used in the SEL is Dan Bricklin's Demo Program.

> **RULE**
>
> **Caution must be exercised to ensure that any prototyping activity that is conducted is necessary, has a defined goal, and is not being used as a means to circumvent standard development procedures. See PRODUCTS in Section 4 for additional guidance on how to plan a prototyping effort.**

Project Library

the librarian

In each software development project, one team member is assigned the role of librarian. During a project, the librarian (sometimes called the software configuration manager) maintains the project library, which is a repository of all project information. The librarian also maintains configured software libraries and operates various software tools in support of project activities.

The librarian establishes the project library during the requirements analysis phase. In general, the project library contains any written material used or produced by the development team for the purpose of recording decisions or communicating information. It includes such items as the requirements and specifications document, requirements question-and-answer forms, approved specification modifications, and the requirements analysis summary report. Necessary management information, such as the software development/management plan, is also included.

MEASURES

The following paragraphs describe the measures and evaluation criteria that managers can use to assess the development process during the requirements analysis phase.

Objective Measures

The progress and quality of requirements analysis are monitored by examining several objective measures:

- Staff hours — actual, cumulative hours of staff effort, as a total and per activity
- Requirements questions and answers — the number of question-and-answer forms submitted versus the number answered

Table 4-1. Objective Measures Collected During the Requirements A

MEASURE	SOURCE	FREQUENCY (COLLECT/ANALYZE	DATA COLLECTION	
			CONTINUED	BEGUN
Staff hours (total and by activity)	Developers and managers (via Personnel Resources Forms (PRFs))	Weekly/monthly	*	
Requirements (changes and additions to baseline)	Managers (via Development Status Forms (DSFs))	Biweekly/biweekl		*
Requirements (TBD specifications)	Managers	Biweekly/biweekl	*	
Requirements (Questions/answers)	Managers (via DSFs)	Biweekly/biweekl		*
Estimates of total SLOC, total effort, schedule, and reuse	Managers (via Project Estimates Forms (PEFs))	Monthly/monthly		*

NOTE

The SEL uses 3 hardcopy forms to collect metrics during the requirements analysis phase. The Personnel Resources Form is used by the development team to record weekly effort hours. The Project Estimates Form is used by managers to record their monthly size and effort estimates The Development Status Form is used to record the number of requirements changes, and number of requirements questions vs. answers. See Reference 19 for detailed information about SEL data collection forms and procedures.

- TBD requirements — the number of requirements classified as TBD versus the total number of requirements
- Requirements changes — the total cumulative number of requirements for which specification modifications have been approved
- Estimates of system size, reuse, effort, and schedule — the total estimated number of lines of code in the system; the estimated number of new, modified, and reused (verbatim) lines of code; the total estimated staff hours needed to develop the system; and estimated dates for the start and end of each phase of the life cycle.

For each of these measures, Table 4-1 shows who provides the data, the frequency with which the data are collected and analyzed, and whether data collection is continued from the requirements definition phase or newly initiated.

Evaluation Criteria

staff hours

Staff hours are usually graphed against a profile of estimated staff effort that is generated by the software development manager for the SDMP (Figure 4-5). This early comparison of planned versus actual staffing is used to evaluate the viability of the plan.

In the flight dynamics environment, hours that are lower than expected are a serious danger signal — even if schedules are being met — because they indicate the development team is understaffed. If too few developers perform requirements analysis, the team will not gain the depth of understanding necessary to surface requirements problems. These problems will show up later in the life cycle when they are far more costly to rectify.

requirements questions and changes

A growing gap between the number of questions submitted and the number of responses received or a large number of requirements changes may indicate problems with the clarity, correctness, or completeness of the requirements as presented in the requirements and specifications document. Data from similar past projects should be used to assess the meaning of the relative sizes of these numbers.

TBD requirements

Because unresolved TBD requirements can necessitate severe design changes later in the project, the number of TBD requirements is the most important measure to be examined during this phase. Categorize and track TBD requirements according to their severity. TBD requirements concerning external interfaces are the most critical, especially if they involve system input. TBDs affecting internal algorithms are generally not so serious.

A TBD requirement is considered *severe* if it could affect the functional design of one or more subsystems or of the high-level data structures needed to support the data processing algorithms. Preliminary design should not proceed until all severe TBD requirements have been resolved. A TBD requirement is considered *nominal* if it affects a portion of a subsystem involving more than one component. Preliminary design can proceed unless large numbers of nominal TBD requirements exist in one functional area. An *incidental* TBD requirement is one that affects only the internals of one unit. Incidental TBD requirements should be resolved by the end of detailed design.

For each TBD requirement, estimate the effect on system size, required effort, cost, and schedule. Often the information necessary to resolve a TBD requirement is not available until later, and design must begin to meet fixed deadlines. These estimates will help determine the uncertainty of the development schedule.

MORE MEASURES

Consider tracking these additional measures of progress during the requirements analysis phase:

- **number of requirements classified vs. total requirements**
- **number of requirements diagrams completed vs. estimated total diagrams**

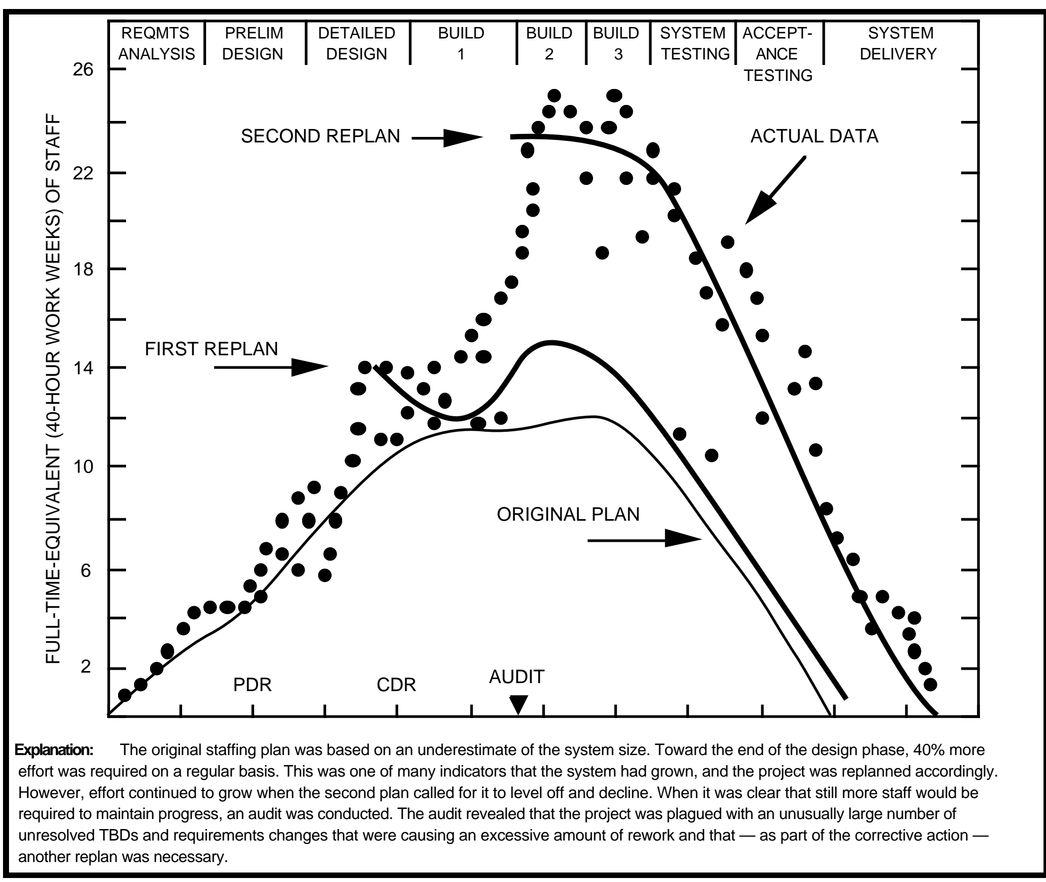
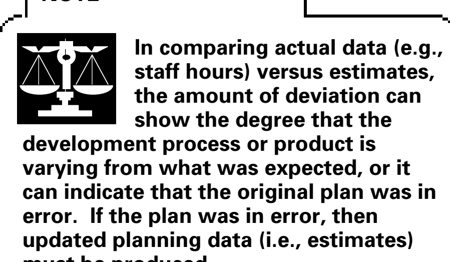


Figure 4-3. Effort Data Example — ERBS AGSS

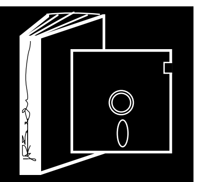
system size estimates

The growth of system size estimates is another key indicator of project stability. Estimating the final size of the system is the first step in the procedure for determining costs, schedules, and staffing levels (Section 3 of Reference 12). Make the initial estimate by comparing current requirements with information from past projects within the application environment. Update and plot the estimate each month throughout the life cycle.



As the project matures, the degree of change in the estimates should stabilize. If requirements growth pushes the system size estimate beyond expected limits, it may be necessary to implement stricter change control procedures or to obtain additional funding and revise project plans. See Section 6 of Reference 12 for additional guidance in evaluating size estimates.

PRODUCTS



The following key products are produced during this phase:

- The requirements analysis report
- The software development/management plan
- Updated requirements and specifications document
- Prototyping plans (as needed)

These products are addressed in the paragraphs that follow.

The Requirements Analysis Report

The requirements analysis report establishes a basis for beginning preliminary design and is, therefore, a key product of the requirements analysis phase. This report includes the following:

- The updated reuse plan (The original reuse proposal was developed during the requirements definition phase and recorded in the systems and operations concept document. It is adjusted to reflect approved requirements changes and the results of analysis of the reusable software's current capabilities.)
- Updates to operational scenarios (in view of prototyping results, performance analyses, requirements changes, and functional reallocations)
- The DFDs or object-oriented diagrams generated to analyze and complete the specifications
- A summary of the results of requirements analysis, highlighting problematic and TBD requirements, system constraints, and development assumptions
- An analysis of the technical risks of the project, as well as cost and schedule risks resulting from TBD requirements or other factors

Figure 4-4 presents the format and contents of the requirements analysis report.

The Software Development/Management Plan

The SDMP provides a detailed exposition of the specific technical and management approaches to be used on the project. In the SDMP, the development team manager discusses how the recommended approach will be tailored for the current project and provides the resource and schedule estimates that will serve as a baseline for comparisons with actual progress data.

REQUIREMENTS ANALYSIS REPORT

This report is prepared by the development team at the conclusion of the requirements analysis phase. It summarizes the results of requirements analysis and establishes a basis for beginning preliminary design. The suggested contents are as follows:

1. **Introduction —** purpose and background of the project, overall system concepts, and document overview

2. **Reuse proposal** — key reuse candidates and overall architectural concept for the system

3. **Operations overview** — updates to system and operations concepts resulting from work performed during the requirements analysis phase
   a. Updated operations scenarios
   b. Operational modes, including volume and frequency of data to be processed in each mode, order, and type of operations, etc.
   c. Updated descriptions of input, output, and messages

4. **Specification analysis**
   a. Summary of classifications (mandatory, requires review, information only, needs clarification, or TBD) assigned to requirements and specifications
   b. Problematic specifications — identification and discussion of conflicting, ambiguous, infeasible, untestable, and TBD requirements and specifications
   c. Unresolved requirements/operations issues, including the dates by which resolutions are needed
   d. Analysis of mathematical algorithms

5. **System constraints**
   a. Hardware availability — execution, storage, peripherals
   b. Operating system limitations
   c. Support software limitations

6. **Performance estimates and models**

7. **Development assumptions**

8. **Risks**, to both costs and schedules. (These should include risks related to TBD or changing requirements, as well as technical risks.)

9. **Prototyping efforts** needed to resolve technical risks, including the goals and completion criteria for each prototyping effort

10. **Data flow or object-oriented diagrams** — results of all structured or object-oriented analysis of the requirements performed during the requirements analysis phase

11. **Data dictionary** — for the updated processes, data flows, and objects shown in the diagrams

Figure 4-4. Requirements Analysis Report Contents

The manager prepares the SDMP during the requirements analysis phase and keeps it up to date throughout the development life cycle. Because of the primary importance of this plan, it is described in detail in the *Manager's Handbook for Software Development* (Reference 12).

The SDMP includes a software development approach; a description of risks and risk mitigation; an initial estimate of the system's size; and estimates of the schedule, staffing, resources, and cost of the project. Figure 4-5 outlines the contents of the SDMP.

SOFTWARE DEVELOPMENT/MANAGEMENT PLAN

In some sections of the plan, material (shown in italics) is to be regularly added during the life of the project. Other sections should be revised and reissued if circumstances require significant changes in approach.

1. INTRODUCTION

- **1.1 Purpose** — brief statement of the project's purpose.
- **1.2 Background** — brief description that shows where the software products produced by the project fit into the overall system.
- **1.3 Organization and Responsibilities**
  - **1.3.1 Project Personnel** — explanation and diagram of how the development team will organize activities and personnel to carry out the project: types and numbers of personnel assigned, reporting relationships, and team members' authorities and responsibilities (see Reference 12 for guidelines on team composition).
  - **1.3.2 Interfacing Groups** — list of interfacing groups, points of contact, and group responsibilities.

2. STATEMENT OF PROBLEM — brief elaboration of the key requirements, the steps numbered) to be taken to accomplish the project, and the relation (if any) to other projects.

3. TECHNICAL APPROACH

- **3.1 Reuse Strategy** — high-level description of the current plan for reusing software from existing systems.
- **3.2 Assumptions and Constraints** — that govern the manner in which the work will be performed.
- **3.3 Anticipated and Unresolved Problems** — that may affect the work and the expected effect on each phase.
- **3.4 Development Environment** — development computer and programming l anguages.
- **3.5 Activities, Tools, and Products** — for each phase, a matrix showing (a) the major activities to be performed, (b) the development methodologies and tools to be applied, and (c) the products of the phase. Includes discussion of any unique approaches or activities.
- **3.6 Build Strategy** — which portions of the system will be implemented in which builds and the rationale for this. *Determined during the preliminary design phase. Updated at the end of detailed design and after each build.*

Figure 4-5. SDMP Contents (1 of 2)

4. MANAGEMENT APPROACH

4.1 Assumptions and Constraints — that affect the management approach, including project priorities.

4.2 Resource Requirements — tabular lists of estimated levels of resources required, including estimates of system size (new and reused SLOC), staff effort (managerial, programmer, and support) by phase, training requirements, and computer resources. Includes estimation methods or rationale used. *Updated estimates are added at the end of each phase.*

4.3 Milestones and Schedules — list of work to be done, who will do it, and when it will be completed. Includes development life cycle (phase start and finish dates); build/release dates; delivery dates of required external interfaces; schedule for integration of externally developed software and hardware; list of data, information, documents, software, hardware, and support to be supplied by external sources and delivery dates; list of data, information, documents, software, and support to be delivered to the customer and delivery dates; and schedule for reviews (internal and external). *Updated schedules are added at the end of each phase.*

4.4 Measures — a table showing, by phase, which measures will be collected to capture roject data for historical analysis and which will be used by management to monitor progress and product quality (see Reference 12). If standard measures will be collected, references to the relevant standards and procedures will suffice. Describes any measures or data collection methods unique to the project.

4.5 Risk Management — statements of each technical and managerial risk or concern and how it is to be mitigated. *Updated at the end of each phase to incorporate any new concerns.*

5. PRODUCT ASSURANCE

5.1 Assumptions and Constraints — that affect the type and degree of quality control and configuration management to be used.

5.2 Quality Assurance — table of the methods and standards that will be used to ensure the quality of the development process and products (by phase). Where these do not deviate from published methods and standards, the table should reference the appropriate documentation. Methods of ensuring or promoting quality that are innovative or unique to the project are described explicitly. Identifies the person(s) responsible for quality assurance on the project, and defines his/her functions and products by phase.

5.3 Configuration Management — table showing products controlled, as well as tools and procedures used to ensure the integrity of the system configuration (when is the system under control, how are changes requested, who makes the changes, etc.). Unique procedures are discussed in detail. If standard configuration management practices are to be applied, references to the appropriate documents are sufficient. Identifies the person responsible for configuration management and describes this role. *Updated before the beginning of each new phase with detailed configuration management procedures for the phase, including naming conventions, directory designations, reuse libraries, etc.*

6. APPENDIX: PROTOTYPING PLANS — collected plans for each prototyping effort to be conducted on the project.

7. PLAN UPDATE HISTORY — *lead sheets from each update of the SDMP, indicating which sections were updated and when the update was made.*

Figure 4-5. SDMP Contents (2 of 2)

Updated Requirements and Specifications

During this phase, the requirements definition team prepares updates to the requirements and specifications document on the basis of approved specification modifications. Additional specification modifications may be approved as a result of discussion at the SSR or the RID process. The requirements definition team must ensure that the updated requirements and specifications document is republished shortly after the SSR, so that it will be available to the developers as they generate the software design.

Prototyping Plans

Managing a prototype effort requires special vigilance. Progress is often difficult to predict and measure. A prototyping effort may continue indefinitely if no criteria are established for evaluating the prototype and judging completion. Writing a plan for each prototyping activity, no matter how brief, is vital to establishing control.

The length of the plan and the time to prepare it should be proportional to the size and scope of the prototyping effort. A one-page plan may be all that is required for small prototyping efforts. A brief plan need not be published separately but may, instead, be incorporated into the SDMP (Figure 4-5).

The following items should be included in the plan:

- Objective of the prototype — its purpose and use
- Statement of the work to be done and the products to be generated
- Completion criteria
- Assessment methods — who will evaluate the prototype and how it will be evaluated
- Technical approach
- Resources required — effort and size estimates, staff, hardware, software, etc.
- Schedule

The SDMP should contain summaries of the detailed prototyping plans. Each summary should describe the general approach, discuss the items to be prototyped, include effort estimates, provide a schedule, and discuss the rationale for the schedule.

SOFTWARE SPECIFICATION REVIEW

At the conclusion of requirements analysis, the development team holds an SSR. This is a high-level review conducted for project management and the end users of the system. The SSR format, schedule, and participants are itemized in Figure 4-6.

SSR FORMAT

Presenters — software development team

Participants
- Requirements definition team
- Customer interfaces for both the requirements definition and software development teams
- User representatives
- Representatives of interfacing systems
- Quality assurance representatives for both teams
- System capacity/performance analysts
- CCB

Schedule — after requirements analysis is complete and before the preliminary design phase is begun

Agenda — selective presentation of the results of requirements analysis, directed primarily toward project management and the users of the system

Materials Distribution
- The requirements analysis report and software development/ management plan are distributed 1 to 2 weeks before SSR.
- Hardcopy material is distributed a minimum of 3 days before SSR.

Figure 4-6. SSR Format

SSR Hardcopy Material

The hardcopy materials for the review will contain some of the same information found in the requirements analysis report. Keep in mind that there is some flexibility in selecting the most appropriate information to include in the presentation. The contents suggested in Figure 4-7 are intended as a guideline.

HARDCOPY MATERIAL FOR THE SSR

1. **Agenda** — outline of review material
2. **Introduction** — background of the project and purpose of system
3. **Analysis overview** — analysis approach, degree of innovation required in analysis, special studies, and results
4. **Revisions since SRR** — changes to system and operations concepts, requirements, and specifications effected following the SRR
5. **Reusable software summary**
   a. Key reuse candidates — identification of existing software components that satisfy specific system specifications exactly or that will satisfy the specifications after modification
   b. Overall architectural concept for the system
   c. Matrix of requirements to be fulfilled by reused components
6. **Requirements classification summary**
   a. List of requirements and specifications with their assigned classifications (mandatory, requires review, needs clarification, information only, or TBD)
   b. Problematic specifications — identification and discussion of conflicting, ambiguous, infeasible, and untestable requirements and specifications
   c. Unresolved requirements/operations issues, including the dates by which resolutions to TBDs are needed (NOTE: This is a key element of the SSR.)
7. **Functional or object-oriented specifications**
   a. Object diagrams or high-level data flow diagrams showing input, transforming processes, and output
   b. Data set definitions for external interfaces to the system
8. **Performance model** — key estimates and results of modeling system performance
9. **Development considerations**
   a. System constraints — hardware availability, operating system limitations, and support software limitations
   b. Utility, support, and test programs — list of auxiliary software required to support development (e.g., data simulators, special test programs, software tools, etc.)
   c. Testing requirements
   d. Development assumptions
10. **Risks**, both to costs and schedules — includes how risks are identified, their potential impact, and how they will be managed. Covers risks related to TBD or changing requirements as well as technical risks
11. **Summary of planned prototyping efforts** needed to resolve technical risks, including the goals and schedule for each effort and a summary of any prototyping conducted to date
12. **Key contacts —** leaders of technical teams, application specialists, and other key project members
13. **Milestones and schedules** — includes size estimates, development life cycle (phase start and finish dates), schedule for reviews (internal and external), build/release requirements, delivery dates of required external interfaces, schedule for integration of externally developed software and hardware

Figure 4-7. SSR Hardcopy Material

LIFE CYCLE PHASES	REQUIREMENTS DEFINITION	REQUIRE-MENTS ANALYSIS	PRELIMI-NARY DESIGN	DETAILED DESIGN	IMPLEMENTATION	SYSTEM TESTING	ACCEPTANCE TESTING

SECTION 5

THE PRELIMINARY DESIGN PHASE

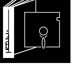
PHASE HIGHLIGHTS

ENTRY CRITERIA

- Requirements analysis report generated
- SSR completed
- SSR RIDs answered

EXIT CRITERIA

- Preliminary design report generated
- PDR completed
- PDR RIDs answered



PRODUCTS

- Preliminary design report

MEASURES

- Units identified/designed
- Requirements Q&As, TBDs, and changes
- Staff hours
- Estimates of system size, effort, schedule, and reuse

METHODS AND TOOLS

- Functional decomposition and object-oriented design **
- Prologs and PDL
- Software engineering notebooks
- Design walk-throughs***
- Design inspections***
- Reuse verification
- Analysis methods

KEY ACTIVITIES

Development Team*
- Prepare preliminary design diagrams
- Prepare prologs and PDL for high-level functions/objects
- Document the design in the preliminary design report
- Conduct the PDR

Management Team
- Reassess schedules, staffing, training, and other resources
- Plan, coordinate, and control requirements changes
- Control the quality of the preliminary design process and products
- Plan the transition to detailed design

Requirements Definition Team
- Resolve outstanding requirements issues
- Participate in design walk-throughs and PDR

With the Cleanroom methodology:
* All test activities are conducted by a separate test team established at the beginning of preliminary design. During this phase, the test team develops a usage profile for statistical testing, derives a list of functions to be tested, and drafts a preliminary system test plan.
** Box structures (References 20,21) are used to develop the design and examine alternatives.
***Cleanroom relies upon design walk-throughs for understanding and upon inspections for ensuring quality products. Both methods have been used extensively on SEL-monitored projects and are key elements of the SEL's recommended approach.

OVERVIEW

The purpose of the preliminary design phase is to define the high-level software architecture that will best satisfy the requirements and specifications for the system.

During preliminary design, the development team uses the updated requirements and specifications document to develop alternative designs and to select an optimum approach. The team partitions the system into major subsystems, specifies all system and subsystem interfaces, and documents the design using *structure charts* or annotated *design diagrams*. Developers use an iterative design process that proceeds somewhat differently, depending on whether a functional decomposition or object-oriented design approach is chosen.

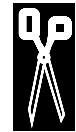
Early in this phase, developers examine the software components that are candidates for reuse and verify their compatibility with overall system requirements and the emerging design. Prototyping activities begun during the requirements analysis phase may continue and new prototyping efforts may be initiated. Developers also define error-handling and recovery strategies, determine user inputs and displays, and update operational scenarios.

During this phase, the development team continues to work closely with analysts of the requirements definition team to resolve requirements ambiguities and TBDs. To ensure that the emerging design meets the system's requirements, developers send formal requirements questions to analysts for clarification, conduct walk-throughs, and subject all design products to peer inspection.

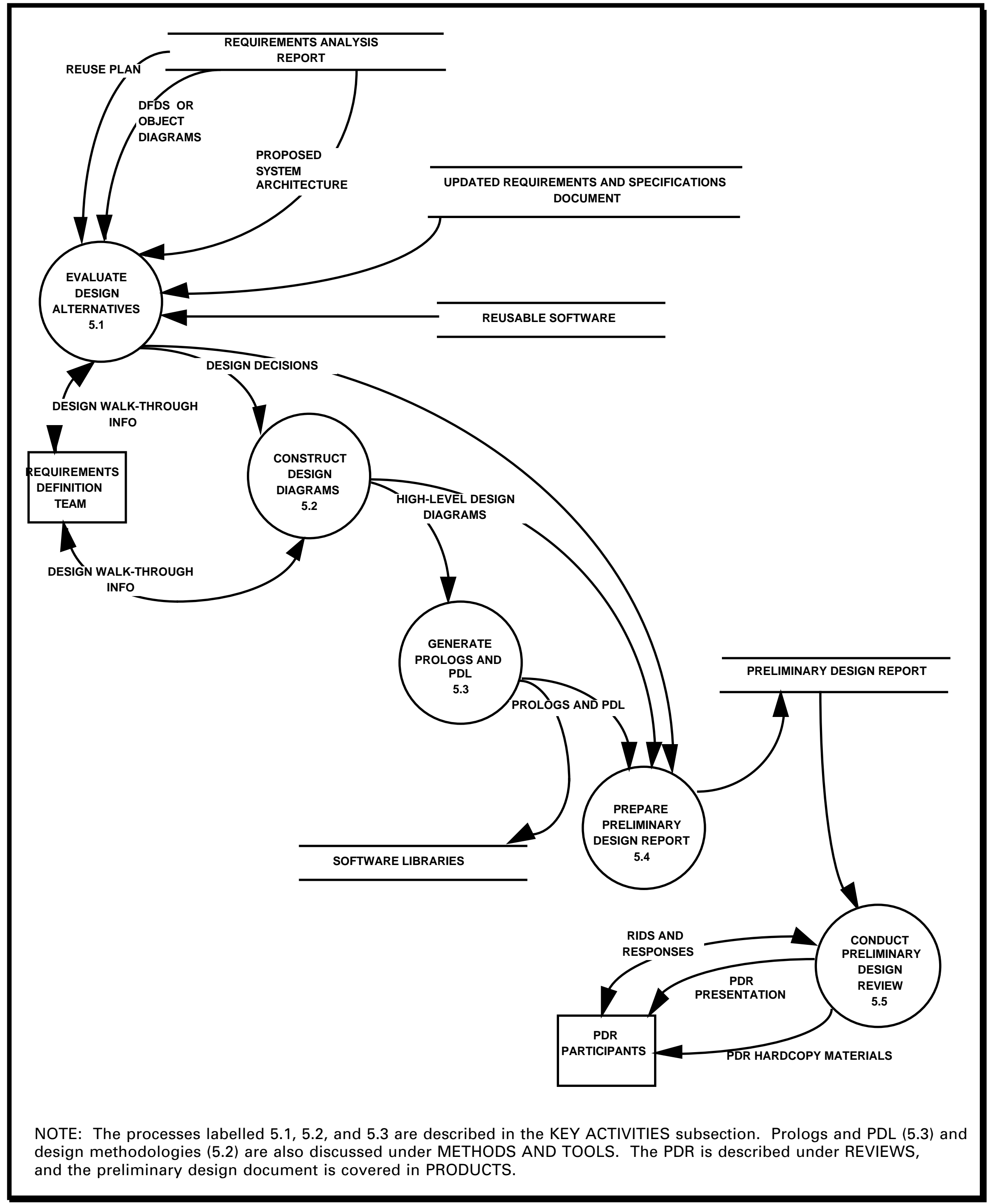
The preliminary design phase culminates in the *preliminary design review (PDR)*, during which developers present the rationale for selecting the high-level system design. The preliminary design report documents the initial system design and is distributed for review prior to the PDR.

Figure 5-1 presents a high-level data flow diagram of the preliminary design process.

TAILORING NOTE



On projects with a high degree of reuse, the preliminary and detailed design phases may be combined. In that case, both preliminary and detailed design activities are conducted, but developers hold only a CDR and produce only the detailed design report.



NOTE: The processes labelled 5.1, 5.2, and 5.3 are described in the KEY ACTIVITIES subsection. Prologs and PDL (5.3) and design methodologies (5.2) are also discussed under METHODS AND TOOLS. The PDR is described under REVIEWS, and the preliminary design document is covered in PRODUCTS.

Figure 5-1. Developing the Preliminary Design

KEY ACTIVITIES

The following are the key activities of the development team, the management team, and the requirements definition team during the preliminary design phase. The development team performs the principal activities of the phase. The requirements definition team concentrates on resolving outstanding TBD requirements and providing support to the development team. The relationships among the major activities of these groups are shown in Figure 5-2.

Activities of the Development Team

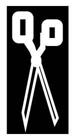
- **Prepare preliminary design diagrams.** Using functional decomposition or object-oriented techniques, expand the preliminary software architecture that was proposed in earlier phases. Idealize the expanded architecture to minimize features that could make the software difficult to implement, test, maintain, or reuse.

  Evaluate design options. Weigh choices according to system priorities (e.g., optimized performance, ease of use, reuse considerations, reliability, or maintainability). Use prototyping and performance modeling to test alternatives, especially in risk areas.

  Examine the software components that are candidates for reuse. If system response requirements are especially stringent, model the performance of the reusable components. Update the reuse plan to reflect the results of these reuse verification activities.

  Generate high-level diagrams of the selected system design and walk the analysts of the requirements definition team through them. Use the high-level design diagrams to explain the system process flow from the analyst's perspective. Focus on system and subsystem interfaces. Explain refinements to the operations scenarios arising from analysis activities and include preliminary versions of user screen and report formats in the walk-through materials.

TAILORING NOTE



Design diagrams, prologs, and PDL are required for all systems, regardless of the design methodology applied. METHODS AND TOOLS discusses ways these items are represented.

NOTE

Reuse verification encompasses designs, documentation, and test plans and data as well as code. See METHODS AND TOOLS for more details.

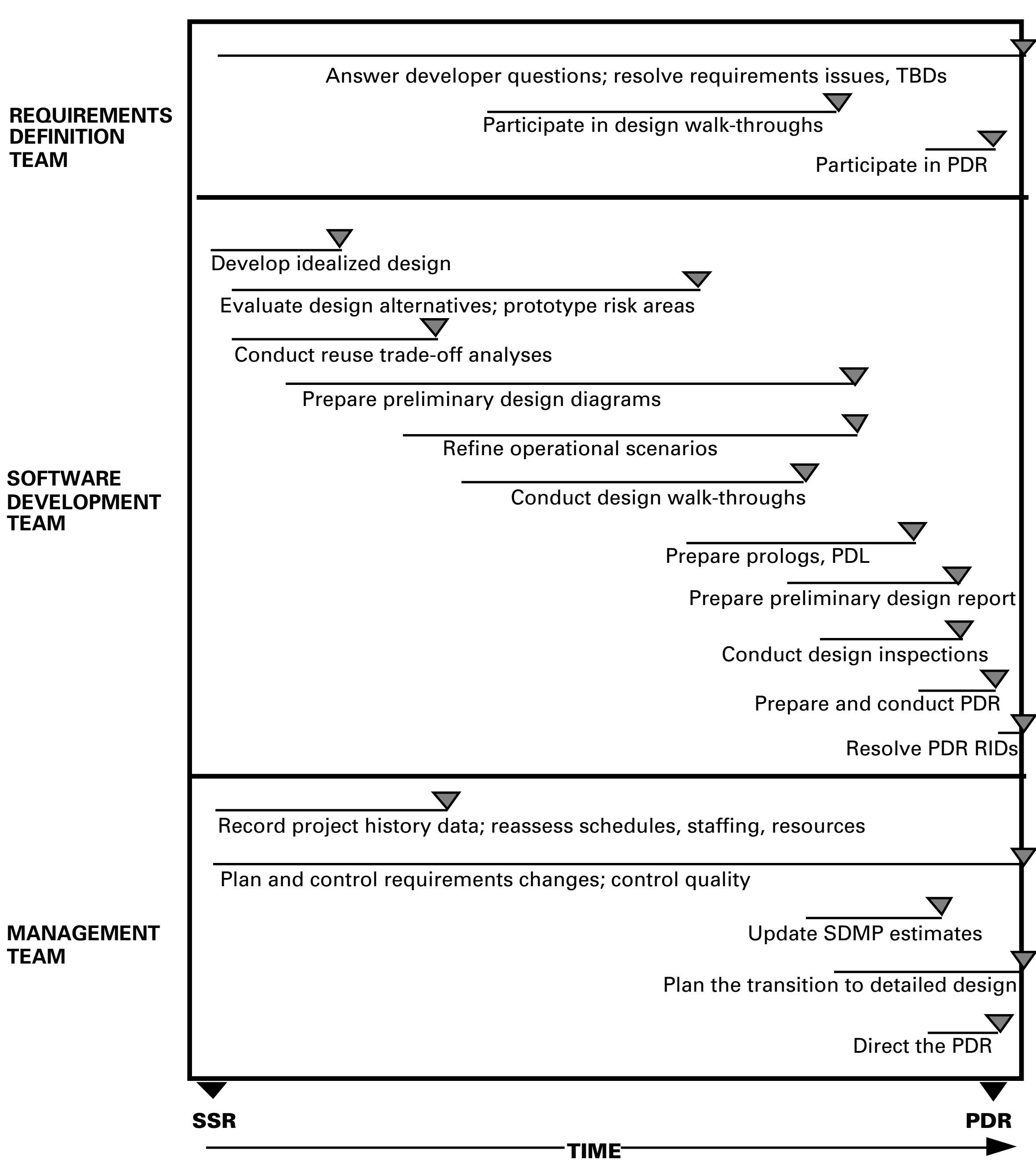


Figure 5-2. Preliminary Design Phase Timeline

- **Prepare prologs and PDL for the high-level functions/objects.** For FORTRAN systems, prepare prologs and PDL to one level below the subsystem drivers. For Ada systems, prepare and compile specifications for the principal objects in the system and construct sufficient PDL to show the dependencies among packages and subprograms. (See Figure 5-4.)

  Provide completed design diagrams, unit prologs/package specifications, and PDL to other members of the development team for independent inspection and certification.

> NOTE
>
>
>
> **Contents of the preliminary design report and PDR materials are provided under the PRODUCTS and REVIEW headings, respectively, in this section. Inspection and certification procedures are covered under METHODS AND TOOLS.**

- **Document the selected design in the preliminary design report.** Include the reuse plan, alternative design decisions, and external interfaces.

- **Conduct the PDR** and resolve RIDs. Record design changes for use in the detailed design phase, but do not update the preliminary design report.

Activities of the Management Team

During preliminary design, the manager's focus changes from planning to control. The following are the major management activities of this phase:

> NOTE
>
> **Material for the software development history (SDH) is collected by the management team throughout the life of the project. See Section 9 for an outline of SDH contents.**

- **Reassess schedules, staffing, training, and other resources.** Begin the software development history by recording lessons learned and project statistics from the requirements analysis phase. Include percentages of project effort and schedule consumed, growth of system size estimates, and team composition.

  Ensure that the development team contains a mix of software engineers and personnel experienced in the application area. If the project is large, partition the development team into groups, usually by subsystem, and appoint group leaders. Adjust staffing levels to compensate for changes in requirements and staff attrition, and ensure the team obtains the training it needs to meet specific project demands.

- **Plan, coordinate, and control requirements changes**. Interface with analysts and the customer to facilitate resolution of requirements issues and TBDs. Monitor the number and severity of requirements questions submitted to analysts and the timeliness of responses. Ensure that analysts and the customer know the dates by which TBDs need to be resolved and understand the impact of changing or undetermined requirements items.

  Assess the technical impact and cost of each specification modification. Constrain modifications that necessitate extensive rework and non-critical enhancements.

- **Control the quality of the preliminary design process and its products** during day-to-day management activities. Ensure that design walk-throughs, inspections, and reviews are scheduled and conducted. Attend walk-throughs and, optionally, inspections and oversee the reporting, tracking, and resolution of the design issues that arise. Make certain that all requisite software documentation is generated and review the preliminary design document. Ensure that the team adheres to project standards and configuration management procedures.

- **Plan an orderly transition to the detailed design phase**. Consider the impacts of TBDs, specification modifications, and schedule or team adjustments. Revise project estimates of effort, duration, and size and update corresponding sections of the SDMP. Develop the project build strategy and prepare a preliminary build plan reflecting prototyping results, project risks, and remaining TBDs.

  Increase team size if necessary to begin detailed design and address the training needs of additional personnel. Oversee the establishment of online libraries to store unit prologs, PDL, and reused code. While project and group leaders prepare for PDR, start the rest of the team on detailed design activities.

  Control the PDR and ensure that all exit criteria have been met before declaring the phase complete.

Activities of the Requirements Definition Team

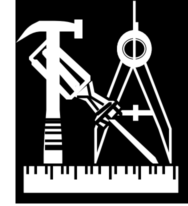
During the preliminary design phase, the requirements definition team provides support to software developers through the following activities:

- **Continue to resolve requirements issues and TBDs.** Clarify ambiguous, conflicting, or incomplete requirements. Provide prompt, written replies to developers' requirements questions and discuss these responses with developers. Respond to changes in high-level system requirements, evaluate the impact of each change, and prepare specification modifications accordingly.

- **Participate in design walk-throughs and the PDR.** Thoroughly analyze the proposed design. Work with developers to refine the operational scenarios and preliminary user interface. Follow up with developers to address issues raised during the walk-throughs.

  Review the preliminary design report and all supporting hardcopy materials before PDR. Pose questions and provide critiques of the initial, high-level system design during the review meeting, and use RIDs to document serious discrepancies.

METHODS AND TOOLS



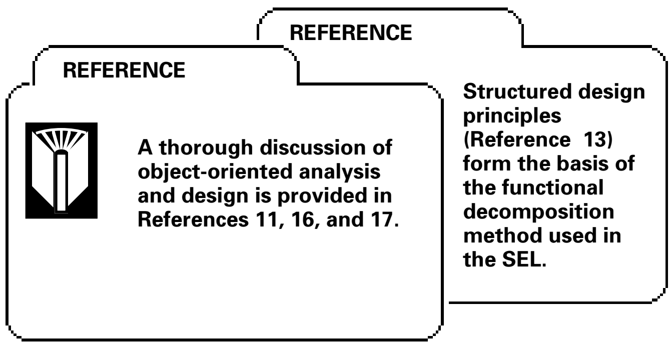
The primary methods and tools used during the preliminary design phase are

- Functional decomposition and object-oriented design
- Prologs and PDL
- Software engineering notebooks (SENs)
- Design walk-throughs
- Design inspections
- Reuse verification
- Analysis methods: prototyping, performance modeling, and code analysis

Functional Decomposition and Object-Oriented Design

Design technologies are methods by which software developers define the major components of a software system, describe the interrelationships among components, and create a foundation for implementation. Design diagrams, structure charts, and documentation support these methods. Through these tools, developers demonstrate that a chosen design approach incorporates

> **NOTE**
>
> **During the design and implementation phases, the software development and requirements definition teams continue to use question-and-answer forms and specification modifications to record and resolve requirements issues. See METHODS AND TOOLS, Section 4, for more details.**

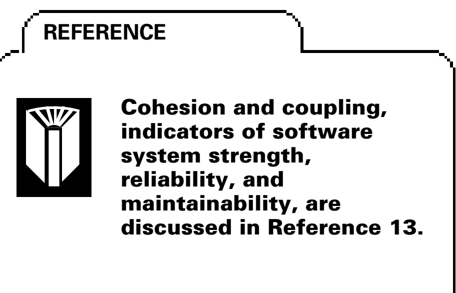


each capability and interface specified in the requirements and specifications document. The two principal design technologies used on SEL-monitored projects are functional decomposition and object-oriented design (OOD).

When using a *functional decomposition* design method, developers identify the major functions of a system and successively refine them into smaller and smaller functionally oriented components. High levels in the design define the algorithmic abstraction (the "what" of the process), and the lower levels provide primitive operations that implement the higher level actions.

In the flight dynamics environment, functional decomposition is normally used for the development of FORTRAN systems. When using this design approach, functional baseline diagrams (tree charts) are generated during preliminary design for all components to two levels below the subsystem drivers (as shown in Figure 5-3). The remaining design diagrams (levels 3 to N) are completed during detailed design. Separate structure charts may augment the diagrams; alternatively, interface information may be added directly to the tree charts.

The SEL also recommends that functionally oriented designs employ the principles of information hiding, data abstraction, loose coupling, and cohesion. Components below the heavy line in the functional decomposition hierarchy of Figure 5-3 denote low-level routines or utilities whose details are deferred to the detailed design phase. However, developers must still understand the total system architecture to produce a correct preliminary design.



When using an *object-oriented* approach, designers identify the abstract objects and their attributes that model the real-world system, define operations on those objects, and establish the interfaces between them. By focusing primarily on the *objects* (the "things" of the system) rather than on the *actions* that affect those objects, object-oriented techniques

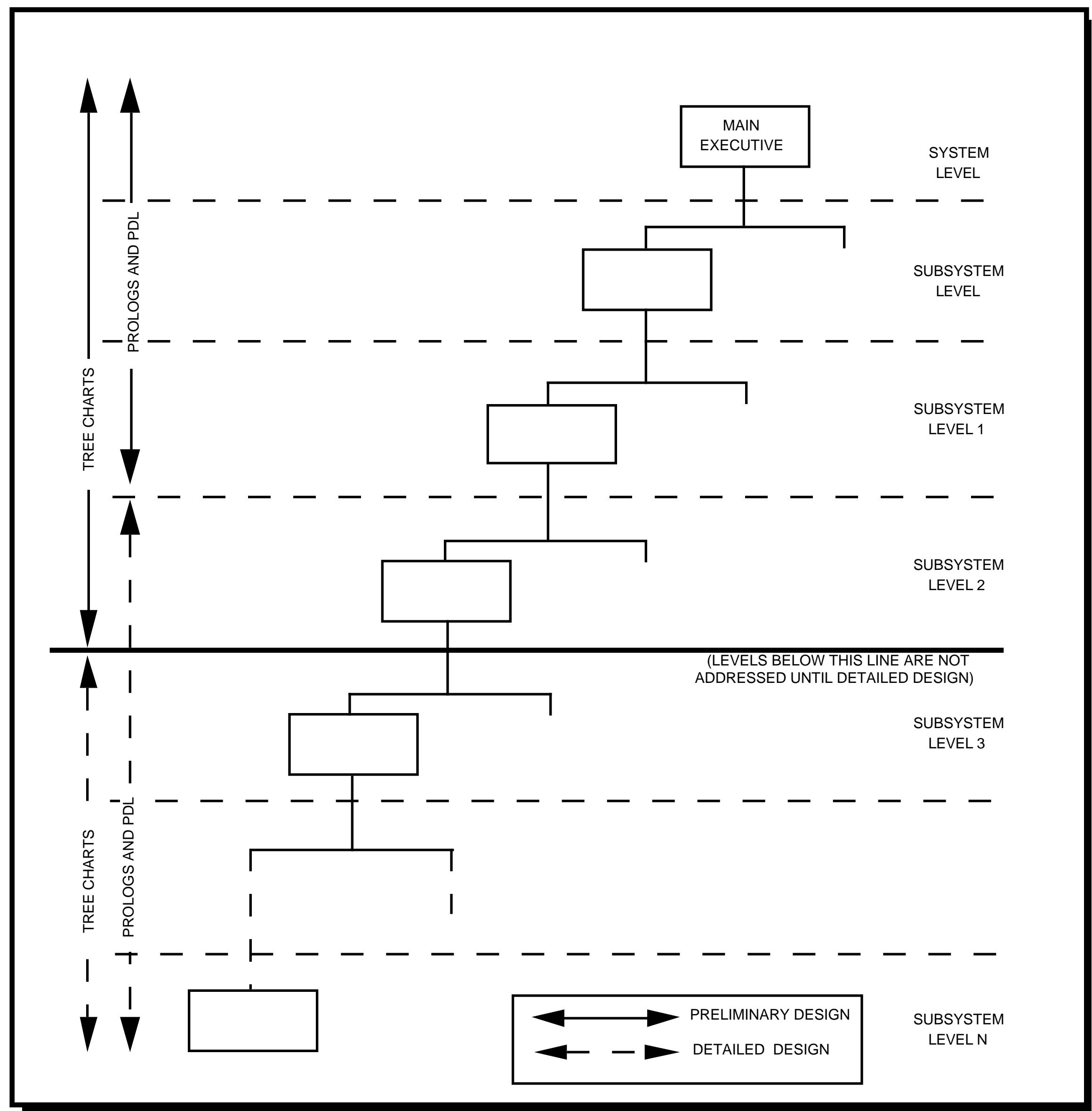


Figure 5-3. Extent of the Design Produced for FORTRAN Systems During the Preliminary and Detailed Design Phases

allow designers to map their solutions more directly to the problem.

In the flight dynamics environment, Ada is typically the language involved when OOD techniques are chosen. In the preliminary design of Ada systems, all packages and subprograms that address elements of the problem domain are identified. This includes all high-level objects necessary to implement the system capabilities, the functions and procedures affecting these objects, externally visible data, and all interfaces and dependencies.

Developers use object-oriented, stepwise refinement until all subsystems, all packages, and all visible subprograms within those packages have been identified. Design of package bodies (the hidden elements shaded in Figure 5-4) is reserved until the detailed design phase. A generalized preliminary design diagram appropriate for an Ada system is shown in Figure 5-4.

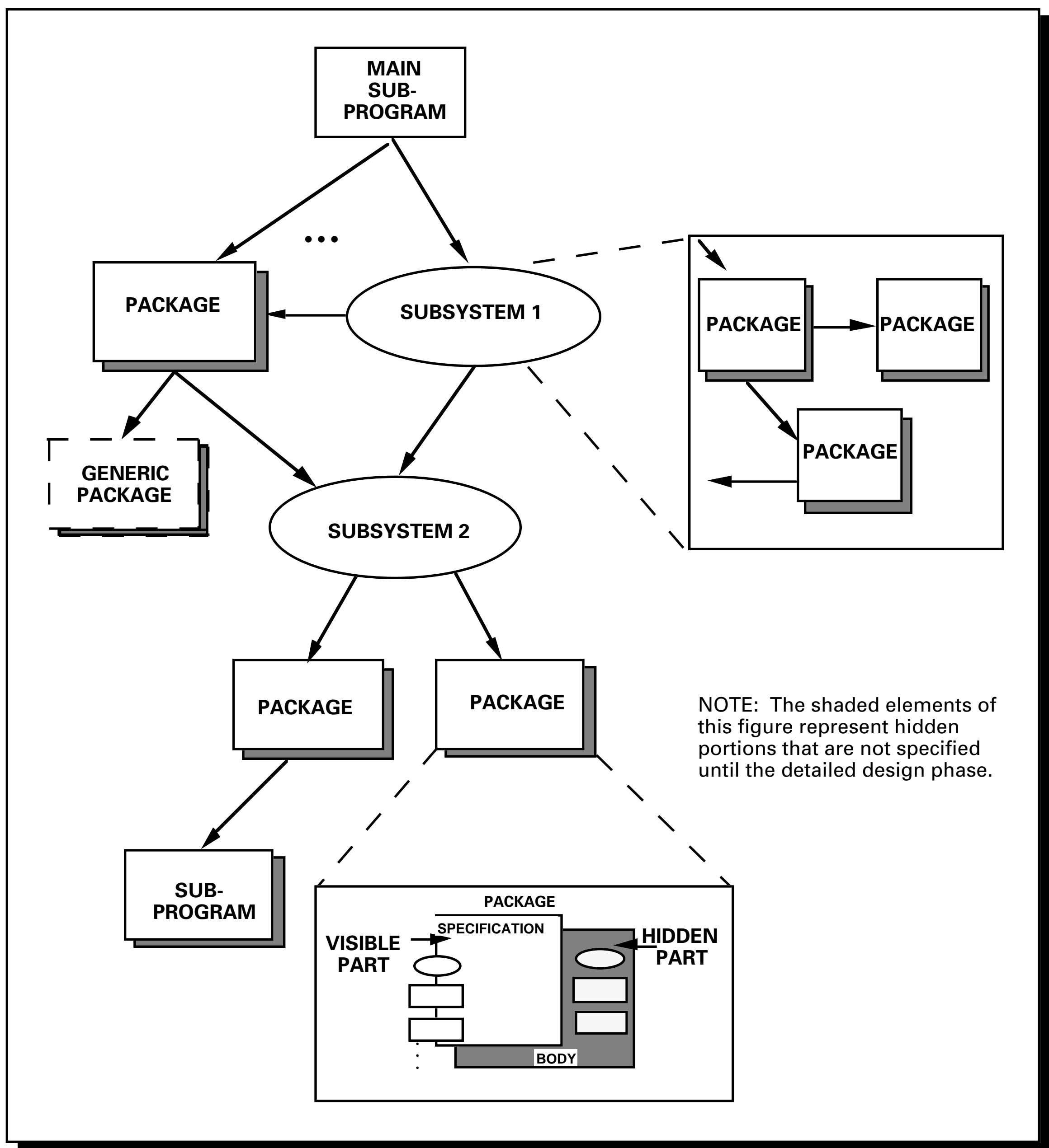
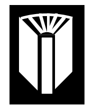


Figure 5-4. Level of Detail Produced for Ada Systems During Preliminary Design

Prologs and PDL

Comparable to the blueprint in hardware systems, prologs and PDL communicate the concept of the design to the level of detail necessary for implementation. The prolog provides a textual explanation of the unit's purpose and variables; PDL provides a formal, algorithmic specification for the unit. By using prologs and PDL, the developer is able to communicate the exact intent of the design to reviewers and coders.

REFERENCE



SEL conventions for prologs and PDL are found in References 22 (Ada conventions) and 23 (FORTRAN conventions).

The SEL views the use of prologs and PDL during design as beneficial and cost-effective. Regardless of the design methodology chosen, developers are required to generate unit prologs and high-level PDL (consisting of such items as call dependencies, major logic branches, and error-handling strategies) to complete the preliminary design.

TAILORING NOTE

For large Ada systems with broad, flat designs, creating PDL for externally visible elements of all packages and all visible subprograms entails extensive effort. When this situation exists, software managers should adjust effort and schedule allocations to allow sufficient time to complete this work in preliminary design.

For FORTRAN systems, prologs and PDL are produced to one level below the subsystem drivers. For object-oriented systems, package specifications (which serve as prologs in Ada systems) and high-level PDL are generated for all program elements depicted in the design diagrams (see Figure 5-4). To identify interface errors, Ada PDL is always compiled; developers use templates and a language-sensitive editor (LSE) to standardize PDL structure.

Software Engineering Notebooks

A software engineering notebook (SEN) is a workbook (i.e., a file folder or special notebook) that facilitates quality assurance and configuration management by consolidating printed information pertinent to a software component. This information becomes more detailed as the life cycle progresses. When printed documentation is combined with online source files in later phases, the SEN provides a baseline history of an element's evolution through the development process.

During preliminary design, developers initiate SENs at a subsystem or logical function level. Into these SENs they collect notes documenting design decisions, design diagrams, structure charts, and signed design inspection checklists for the subsystem or function. SENs are usually maintained by developers until after unit testing, when they are turned over to the project librarian.

Design Walk-Throughs

Developers conduct design walk-throughs to ensure that both the requirements definition and development teams understand the system design as it takes shape. Developers distribute materials prior to the walk-through to participants, who include other developers, analysts, user representatives, representatives of systems that will interface with the software under development, and managers. During the meeting, developers carefully explain how operational aspects of the system (processing sequences and interfaces, screen formats, and user inputs) are reflected in the emerging design. Participants comment on how completely and accurately developers have interpreted the system requirements. A recording secretary notes discrepancies, errors, and inconsistencies and records action items. If significant issues remain at the end of the walk-through, a follow-up session may be scheduled.

The initial walk-through typically presents the overall, system level approach and is later followed by walk-throughs of subsystems and their major parts. When a project is large and the development team has been partitioned into groups, it is important that the group leader of any subsystem that interfaces with the subsystem being presented attend the session.

Design Inspections

NOTE

Design inspections are conducted during both the preliminary and detailed design phases because different units are designed in each phase. If a unit design was inspected and certified during the preliminary design phase, it is not re-inspected during the detailed design phase unless it has been revised. See METHODS AND TOOLS in Section 6 for a detailed description of design inspection procedures.

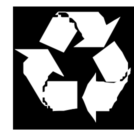
Whereas design walk-throughs focus on explaining operational aspects of the system design, design inspections are independent, in-depth, technical reviews of the design diagrams, prologs, and PDL that are performed by software team peers. Inspections are held as logically related parts of the system design become clear and complete; they are scheduled when unit prologs and PDL have been generated to the required level. Developers distribute these materials for study by the inspection team prior to holding a working meeting.

An inspection team consists of three or more members of the development team who are versed in the project's standards, development language, and system requirements. One member of the team acts as the moderator for the meeting.

Inspection team participants certify that unit logic and interfaces accurately represent the system requirements and that developers have applied correct design principles. Reviewers document their assessment and note weaknesses or interface conflicts on *design inspection checklists* (Figure 6-3). The signed checklist also certifies that the unit follows prescribed project standards and conventions.

design inspection checklists

Reuse Verification



Reuse verification is the process of determining which of the existing software components specified in the reuse proposal should be integrated into the new system design. During reuse verification, developers examine code and documentation from sources such as the FDF's Reusable Software Library (RSL). Developers draw on their experience in considering the integrity of the overall system architecture, the clarity of the design, and system performance.

When the reuse plan recommends using large portions of existing systems, developers must assess the trade-offs of compromising an optimum system design to make it compatible with existing software. They must consider long-term impacts on total software development costs, design clarity, performance requirements, system size, maintainability, and reliability when weighing design options.

When factors (such as incompatible language versions or a high incidence of operating system-specific calls) prohibit an existing component from being reused, developers can study the software and its documentation to understand its function, organization, and data structures before designing a component compatible with the new system. Thus, reused experience is shared across projects even when explicit design and code cannot be.

Analysis Methods: Prototyping, Performance Modeling, and Code Analysis

During preliminary design, the development team uses prototyping to validate design concepts and to test the trade-offs of design options. Developers compose prototype drivers (or *scaffolding code*) to exercise components planned for reuse in the new system. They also prototype user screens and report formats. To confirm that existing components will meet the new system's performance requirements, developers couple prototyping with performance and source code analysis, as described below.

Performance modeling is a means of predicting how efficiently a program will use resources on a given computer. To generate these predictions, developers use such parameters as the number of units anticipated, the volume and frequency of the data flow, memory usage estimates, and the amount of program I/O. Analogy with existing, similar systems may also be used. Results of performance modeling assist developers in deciding among design options.

If executable code exists (e.g., scaffolding code or reused units), developers can run *performance analyzers*, such as Problem Program Evaluator (PPE) or the VAX Performance and Coverage Analyzer, concurrently with the code. These dynamic analyzers generate information such as CPU time, I/O counts, and page fault data that help developers identify areas of the code (e.g., inefficient loop structures or calculations) that need to be redesigned.

Static code analyzers (e.g., VAX Source Code Analyzer) are tools that read the source code of existing components and generate information describing their control structure, symbol definitions, and variable occurrences. A developer can examine the call trees produced by a static code analyzer to assess the scope of a proposed change to a reusable component's interfaces. The designer can then decide which approach is better — to modify and test all affected units or to design a new component.

MEASURES

During preliminary design, managers continue to use the objective measures of the requirements analysis phase. They also begin to monitor additional progress data. The following measures are collected:

- The number of units designed versus the number identified
- Requirements questions and answers, TBDs, and changes
- Staff hours
- Estimates of system size, effort, schedule, and reuse

The source of these data and the frequency of data collection and evaluation are shown in Table 5-1.

Evaluation Criteria

units designed

Project leaders estimate the number of units that will be needed to represent the preliminary system design. Against this estimate, they track the number of unit designs (prolog and PDL) that have been generated and the number certified. Management plots assist in

Table 5-1. Objective Measures Collected During the Preliminary Design Phase

MEASURE	SOURCE	FREQUENCY (COLLECT/ANALYZE)	DATA COLLECTION	
			CONTINUED	BEGUN
Staff hours (total and by activity)	Developers and managers (via PRFs)	Weekly/monthly	*	
Requirements (changes and additions to baseline)	Managers (via Development Status Forms (DSFs))	Biweekly/biweekly	*	
Requirements (TBD specifications)	Managers (via DSFs)	Biweekly/biweekly	*	
Requirements (questions/answers)	Managers (via DSFs)	Biweekly/biweekly	*	
Estimates of total SLOC (new, modified, reused), total units, total effort, and schedule	Managers (via PEFs)	Monthly/monthly	*	* (total units)
Status (units planned/ designed/ certified)	Managers (via DSFs)	Biweekly/biweekly		*

discovering trends or plateaus in this progress data that signal impending difficulties. The graph of units certified should closely follow and parallel the graph for units designed in a fairly smooth curve. Sharp increases (a "stair-step" graph) will appear when many units are hurriedly certified together in an effort to meet schedules.

> **RULE**
>
> **The number of units designed should not be used as a measure of unit completion. The correct completion measure is the number of certified unit designs.**

requirements questions and answers

During preliminary design, a widening gap between the number of requirements questions submitted and the number of answers received can be an early warning signal of impending rework and eventual schedule slippage. A high number of questions and answers when compared with systems of similar size and complexity is interpreted the same way.

TBD requirements

Tracking the number of TBD requirements that persist into the preliminary design phase, especially those concerning external interfaces and hardware changes, is crucial because TBDs represent

incompleteness in the design and increase the potential for rework. TBDs should diminish quickly in the early life cycle phases.

requirements changes

Specification modifications may be issued in response to development team questions or external project influences such as hardware changes. The number and severity of specification modifications resulting from developers' questions reflect the quality of the requirements and specifications document. These changes are normally addressed by the development and requirements definition teams. The number and severity of specification modifications from external causes have more far-reaching implications and should alert managers to anticipate long-term perturbations in schedule and effort estimates.

NOTE

Numbers of Q&As, TBDs, and specification modifications can change rapidly. Plots of these factors help managers to assess project status and highlight problem areas.

questions

answers

staff hours

Significant deviations between planned and actual staff effort hours warrant close examination. When a high level of effort is required to meet schedules, the development team's productivity may be low or the problem may be more complex than originally realized. Low staff hours may indicate delayed staffing on the project or insufficient requirements analysis. The effects of these last conditions are not always immediately obvious in preliminary design, but will surface dramatically as design deficiencies during subsequent phases.

NOTE

SEL managers update effort, schedule, and size estimates approximately once a month and organize these estimates on the Project Estimates Form (PEF). Data from these forms are used to generate system growth and progress plots.

estimates

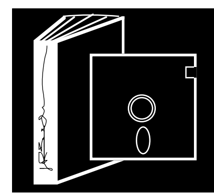
Managers can now use the number of units planned and average unit size to refine estimates of system size. The number of reused units in the system also becomes firmer; managers identify units to be reused without change versus those to be modified and revise effort and schedule projections accordingly.

Estimates are a key management measure; widely varying estimates or sharp increases always warrant investigation. In the preliminary design phase, excessive growth in system size estimates is generally seen when requirements are unstable or change control mechanisms are ineffective.

NOTE

Although a number of measures of design quality (strength, etc.) have been proposed, the SEL has not yet found objective measures that provide significant insight into the quality of a design.

PRODUCTS



The primary product of the preliminary design phase is the high-level design of the system, as documented in the preliminary design report. The report incorporates the results of reuse verification and prototyping and forms the basis for the detailed design document. Because of their size, unit prologs and PDL are normally published in a separate volume that is identified as an addendum to the report.

An outline of the preliminary design report, showing format and content, is provided as Figure 5-5.

PRELIMINARY DESIGN REVIEW

The phase concludes with the preliminary design review (PDR), during which the development team presents the high-level system design and the rationale for choosing that design over alternatives. Also highlighted in the PDR are explanations of external system interfaces, revisions to operations scenarios, major TBDs for resolution, and issues affecting project quality and schedule.

Materials presented at PDR do not necessarily convey the technical depth that the development team has achieved during preliminary design; details of this technical effort are documented in the preliminary design report. Developers limit the PDR presentation to the nominal operating scenarios and significant contingency cases.

The presentation is directed to project managers, users, the requirements definition team, and the CCB. Applications specialists, analysts, quality assurance personnel, and managers perform a detailed technical review of the preliminary design material prior to attending the formal review. They evaluate the system design and provide comments and critiques to the development team during and immediately after the presentation. RIDs are used by participants to record any issues that still need to be resolved. The PDR format and schedule are shown in Figure 5-6, followed by an outline of the PDR hardcopy materials (Figure 5-7).

PRELIMINARY DESIGN REPORT

This report is prepared by the development team as the primary product of the preliminary design phase. It presents the high-level design of the system and forms the basis for the detailed design document. The suggested contents are as follows:

1. **Introduction** — purpose and background of the project, overall system concepts, and document overview

2. **Design overview**
   a. Design drivers and their order of importance (e.g., performance, reliability, hardware, memory considerations, operating system limitations, language considerations, etc.)
   b. Results of reuse tradeoff analyses; the reuse strategy
   c. Critique of alternative designs
   d. Discussion and high-level diagrams of the selected system design, showing hardware interfaces, external data interfaces, interconnections among subsystems, and data flow
   e. A traceability matrix of the subsystems against the requirements
   f. Design status
      (1) List of constraints, concerns, and problem areas and their effects on the design
      (2) List of assumptions and possible effects on design if they are wrong
      (3) List of TBD requirements and an assessment of their effect on system size, required effort, cost, and schedule
      (4) ICD status
      (5) Status of prototyping efforts
   g. Development environment (i.e., hardware, peripheral devices, etc.)

3. **Operations overview**
   a. Operations scenarios/scripts (one for each major product that is generated). Includes the form and volume of the product and the frequency of generation. Panels and displays should be annotated to show what various selections will do and should be traced to a subsystem
   b. System performance considerations
   c. Error recovery strategies (automatic fail-over, user intervention, etc.)

4. **Design description** for each subsystem or major functional breakdown:
   a. Discussion and high-level diagrams of subsystem, including interfaces, data flow, and communications for each processing mode
   b. High-level description of input and output
   c. High-level description of processing keyed to operator-specified input and actions in terms of points of control, functions performed, and results obtained (both normal and abnormal, i.e., error processing and recovery)
   d. Structure charts or object-oriented diagrams expanded to two levels below the subsystem driver
   e. Prologs (specifying the unit's purpose, operation, calling sequence arguments, external references, etc.) and program design language (PDL) for each identified unit (Prologs and PDL are normally published in a separate volume.)

5. **Data interfaces** for each internal and external interface:
   a. Description, including name, function, frequency, coordinates, units, and computer type, length, and representation
   b. Format
      (1) Organization, access method, and description of files (i.e., data files, tape, etc.)
      (2) Layout of frames, samples, records, and/or message blocks
      (3) Storage requirements

Figure 5-5. Preliminary Design Report Contents

PDR FORMAT

Presenters — software development team

Participants
• Requirements definition team
• Quality assurance representatives from both teams
• Customer interfaces for both teams
• User representatives
• Representatives of interfacing systems
• System capacity/performance analysts
• CCB

Schedule — after the preliminary design is complete and before the detailed design phase begins

Agenda — selective presentation of the preliminary design of the system

Materials Distribution
• The preliminary design report is distributed at least 1 week before PDR.
• Hardcopy material is distributed a minimum of 3 days before PDR.

Figure 5-6. PDR Format

EXIT CRITERIA

To determine whether the development team is ready to proceed with detailed design, managers should ask the following questions:

- Have all components that are candidates for reuse been analyzed? Have the trade-offs between reuse and new development been carefully investigated?

- Have developers evaluated alternate design approaches and chosen the optimum design?

- Have all design diagrams, prologs and PDL (or package specifications, if applicable) been generated to the prescribed level? Have they been inspected and certified?

- Have the key exit criteria been met? That is, has the preliminary design report been produced and distributed, has the PDR been successfully completed, and have all PDR RIDs been answered?

When the manager can answer "yes" to each of these questions, the preliminary design phase is complete.

HARDCOPY MATERIAL FOR THE PDR

1. **Agenda** — outline of review material
2. **Introduction** — background of the project and system objectives
3. **Design overview**
   a. Design drivers and their order of importance (e.g., performance, reliability, hardware, memory considerations, programming language, etc.)
   b. Results of reuse tradeoff analyses (at the level of subsystems and major components)
   c. Changes to the reuse proposal since the SSR
   d. Critique of design alternatives
   e. Diagram of selected system design. Shows products generated, interconnections among subsystems, external interfaces. Differences between the system to be developed and existing, similar systems should be emphasized
   f. Mapping of external interfaces to ICDs and ICD status
4. **System operation**
   a. Operations scenarios/scripts — one for each major product that is generated. Includes the form of the product and the frequency of generation. Panels and displays should be annotated to show what various selections will do and should be traced to a subsystem
   b. System performance considerations
   c. Error recovery strategies
5. **Major software components** — one diagram per subsystem
6. **Requirements traceability matrix** mapping requirements to subsystems
7. **Testing strategy**
   a. How test data are to be obtained
   b. Drivers/simulators to be built
   c. Special considerations for Ada testing
8. **Design team assessment** — technical risks and issues/problems internal to the software development effort; areas remaining to be prototyped
9. **Software development/management plan** — brief overview of how the development effort is conducted and managed
10. **Software size estimates** — one slide
11. **Milestones and schedules** — one slide
12. **Issues, problems, TBD items** beyond the control of the development team
    a. Review of TBDs from SSR
    b. Other issues
    c. Dates by which TBDs/issues must be resolved

Figure 5-7. PDR Hardcopy Material

LIFE CYCLE PHASES	REQUIREMENTS DEFINITION	REQUIRE-MENTS ANALYSIS	PRELIMI-NARY DESIGN	DETAILED DESIGN	IMPLEMENTATION	SYSTEM TESTING	ACCEPTANCE TESTING

SECTION 6

THE DETAILED DESIGN PHASE

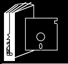
PHASE HIGHLIGHTS

ENTRY CRITERIA

- Preliminary design report generated
- PDR completed
- PDR RIDs answered

EXIT CRITERIA

- Detailed design document generated
- CDR completed*
- CDR RIDs answered



PRODUCTS

- Detailed design document
- Build plan*
- Build/release test plan*

MEASURES

- Units designed/identified
- Requirements Q&As, TBDs, and changes
- Staff hours
- Estimates of system size, effort, schedule, and reuse
- CPU hours

METHODS AND TOOLS

- Functional decomposition and object-oriented design
- Reuse verification
- Analysis methods
- Design walk-throughs
- Design inspections
- Prologs and PDL
- SENs

KEY ACTIVITIES

Development Team

- Prepare detailed design diagrams
- Conduct design walk-throughs
- Refine the operational scenarios
- Complete prologs and PDL for all units
- Provide input to the build plan and begin the build/release test plan*
- Prepare the detailed design document
- Conduct the CDR

Management Team

- Assess lessons learned from the preliminary design phase
- Control requirements changes
- Control the quality of the design process
- Prepare the build plan
- Coordinate the transition to implementation
- Direct the CDR

Requirements Definition Team

- Resolve remaining requirements issues
- Participate in design walk-throughs and CDR

Acceptance Test Team

- Begin work on the acceptance test plan
- Prepare portions of analytical test plan needed for Build 1

*On projects using the Cleanroom methodology, each build consists of a detailed design phase followed by an implementation phase. Statistically selected tests are planned and performed by a separate test team, which compiles, integrates, and tests each build as soon as it has passed coding inspection. An informal review is conducted after the design phase in each build.

OVERVIEW

The purpose of the detailed design phase is to produce a completed design specification that will satisfy all requirements for the system and that can be directly implemented in code.

Unless comments expressed at the PDR indicate serious problems or deficiencies with the preliminary design, detailed design begins following the PDR. The detailed design process is an extension of the activities begun during preliminary design. The development team elaborates the system architecture defined in the preliminary design to the unit level, producing a complete set of *code-to* specifications. The primary product of the phase is the detailed design document, which contains the design diagrams, prologs, and PDL for the software system.

During this phase, the development team conducts walk-throughs of the design for the requirements definition team and subjects each design specification to peer inspection. At the conclusion of the phase, the completed design is formally reviewed at the CDR.

Figure 6-1 shows the major processes of the detailed design phase.

KEY ACTIVITIES

While the development team is generating the detailed design, the requirements definition team continues to resolve the remaining requirements issues and TBDs. As soon as requirements questions and changes level off, an additional team is formed to prepare for acceptance testing. This *acceptance test team* usually consists of the analysts who will use the system, along with some of the staff who prepared the requirements and specifications document.

The activities of the development team, the management team, the requirements definition team, and the acceptance test team are itemized below. A suggested timeline for the performance of these activities is given in Figure 6-2.

Activities of the Development Team

- **Prepare detailed design diagrams** to the lowest level of detail, i.e., the subroutine/subprogram level. Successively refine each subsystem until every component performs a single function and can be coded as a single unit.

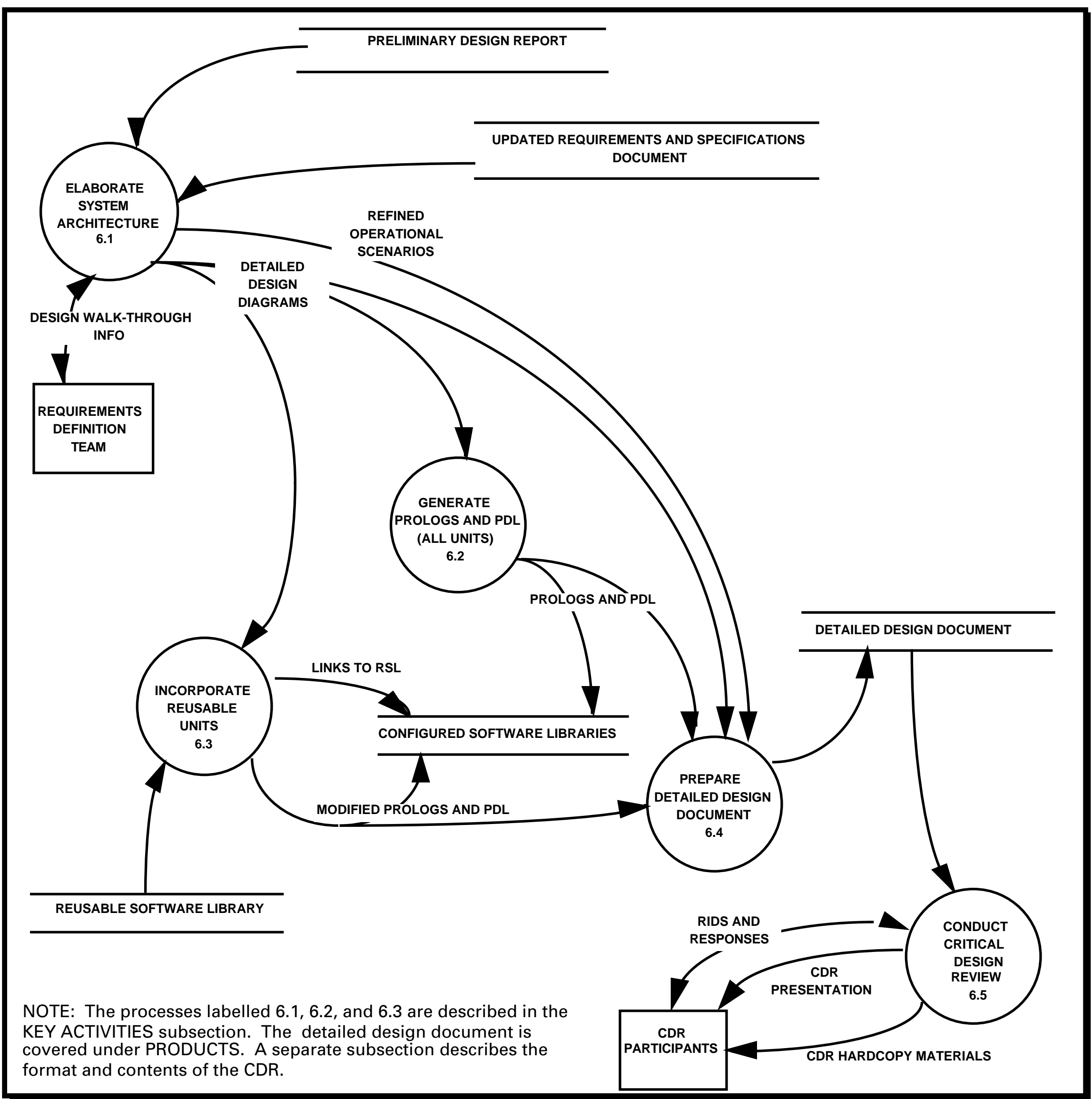


Figure 6-1. Generating the Detailed Design

- **Conduct design walk-throughs.** Walk through each major function or object with other developers and the requirements definition team. On small projects (e.g., simulators), conduct one walk-through per subsystem. On systems of 150 KSLOC or more (e.g., AGSSs), hold two walk-throughs per subsystem — one in the early stages of detailed design and one later in the phase.

- **Refine the operations scenarios** for the system in light of the results of prototyping, performance modeling, and design activities. Finish specifying detailed input and output formats for each subsystem, including displays, printer and plotter output, and data stores.

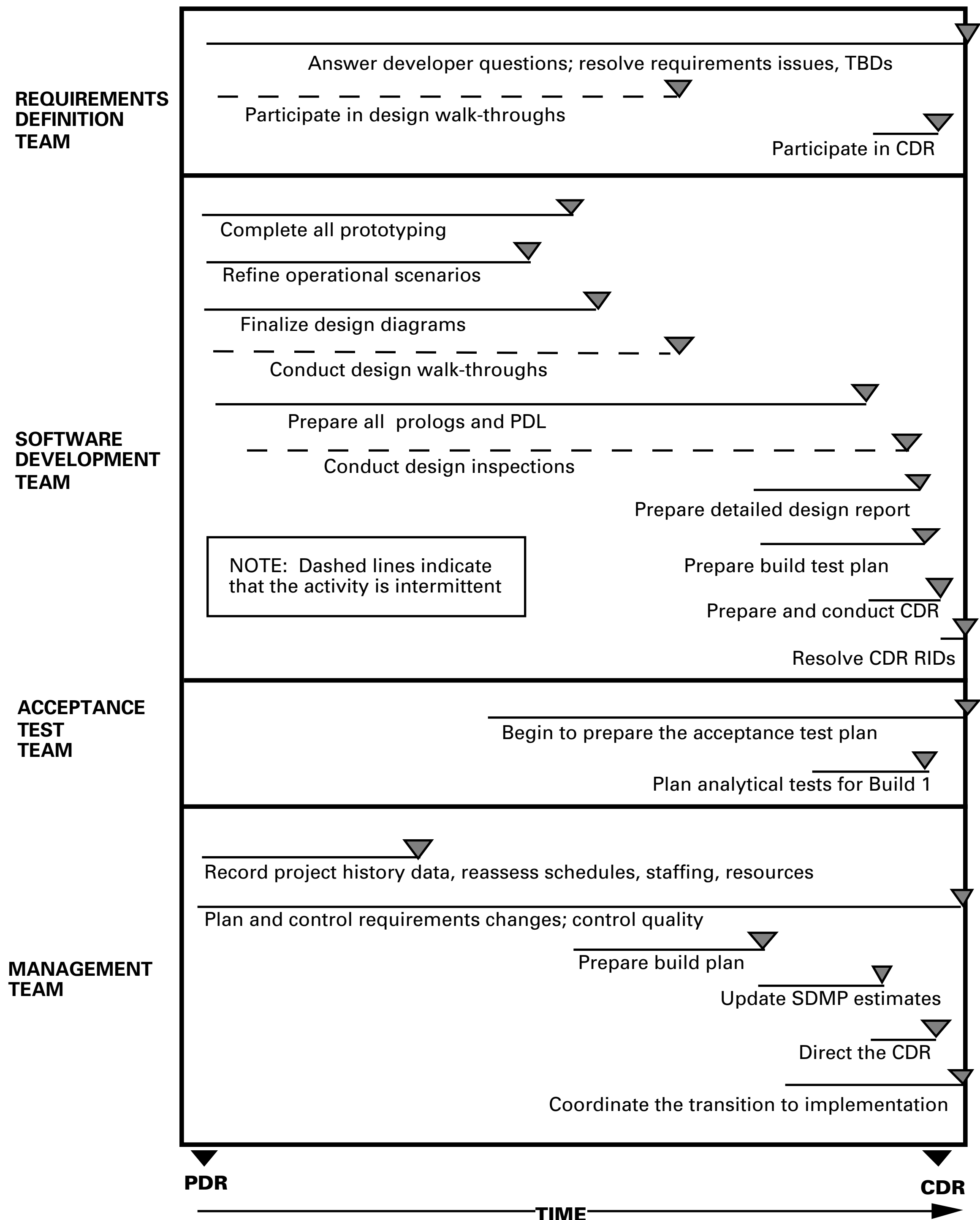


Figure 6-2. Timeline of Key Activities in the Detailed Design Phase

Complete all prototyping efforts. The detailed design presented at CDR should contain no uncertainties that could have been resolved through prototyping.

- **Complete prologs and PDL for all units**. Generate prologs and PDL for the new units specified in the design diagrams. On Ada projects, package specifications and PDL should also be compiled. This means that package bodies are compiled, and that, at a minimum, subprogram bodies each contain a null statement.

  Identify all units (from the RSL or other sources) that will be reused, either verbatim or with modifications. Transfer existing units that require modification into online libraries, and revise the prologs and PDL of these units as necessary.

  Ensure that all unit designs are formally inspected and certified (see *Methods and Tools* below). File the completed checklist in the SEN for the unit.

- **Provide input to the build plan and begin the build test plan**. Provide the technical information needed for the build plan to the management team; include the order in which units should be implemented and integrated. Prepare the test plan for the first build and review it at CDR. (See Section 7 for the plan's format and contents.)

- **Prepare the detailed design document** as a basis for the CDR. Ensure that the team librarian adds all documentation produced during the phase to the project library. (The only exceptions are SEN materials, which are maintained by individual developers until the unit has been coded and tested.)

- **Conduct the CDR** and respond to all CDR RIDs.

Activities of the Management Team

With a few exceptions, the activities of the management team during the detailed design phase parallel those of the previous phase.

- **Assess lessons learned from the preliminary design phase** and record information for the software development history, including schedule data and project statistics. Reassess schedule, staffing, and resources in view of these data.

- **Control requirements changes**. Continue to monitor the number and scope of requirements questions and answers.

Ensure that analysts and the customer understand the potential impact of each requirement TBD and proposed specification modification.

- **Control the quality of the detailed design process** and its products. Ensure adherence to design standards, configuration management procedures (especially change control), reporting procedures, data collection procedures, and quality assurance procedures. Review the design produced and participate in design walk-throughs.

  Ensure that all facets of the project are completely visible and that there is close cooperation between the development team and the other groups with which they must interact.

- **Prepare the build plan**. Use unit dependency information provided by the technical leads and application specialists of the development team to specify the portions of the system that will be developed in each stage of the implementation phase. Document the capabilities to be included in each build and prepare a detailed milestone schedule, factoring in external constraints and user needs.

- **Coordinate the transition to the implementation phase.** It is usually necessary to increase the size of the development team to simultaneously implement multiple subsystems within a build. Inform the development team of the software engineering approaches to be used during implementation and provide the necessary training. Ensure that members of the development team understand the code and testing standards, and the quality assurance and configuration management procedures to be followed.

  Ensure that online project libraries are established, that strict change-control procedures concerning these libraries are followed, and that the necessary software for building and testing the system is made available so that developers can begin implementation immediately after the CDR.

- **Direct the CDR**. Schedule and participate in the review, and ensure that all pertinent groups take part.

Activities of the Requirements Definition Team

- **Resolve any outstanding requirements issues** and TBDs, preparing specification modifications as necessary. Warn developers of any impending changes to requirements so that

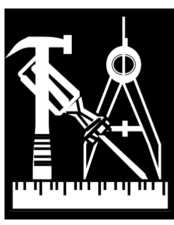
they can plan design activities accordingly. Respond to developers' questions.

- **Participate in design walk-throughs and the CDR.** Review the detailed design document before CDR. During the review, provide a critique of the design. Use RIDs to document issues and discrepancies.

Activities of the Acceptance Test Team

- **Begin work on the acceptance test plan** (Section 8) as soon as requirements have stabilized. Requirements can be considered as stable when developers are no longer submitting new requirements questions and the number of TBDs has declined to a low level.

- **Prepare the portions of the analytical test plan that are needed for Build 1** (see Section 7). Meet with the development team to determine which analytical tests will be needed during the first build of the implementation phase, and complete those portions of the analytical test plan.



METHODS AND TOOLS

The same methods and tools used during the preliminary design phase continue in use during detailed design:

- Functional decomposition and object-oriented design (OOD)
- Reuse verification
- Analysis methods: prototyping, performance modeling, and code analysis
- Design walk-throughs
- Design inspections
- Prologs and PDL
- SENs

During the detailed design phase, the development team uses functional decomposition or OOD techniques to take the design diagrams down to the lowest level of detail (see Figures 5-3 and 5-4). Prototyping and performance modeling efforts that were undertaken to address design issues are completed. The team also completes its examination of the code and documentation of reusable components to determine whether each unit can be incorporated into the system as planned.

The other methods and tools that continue in use from the preliminary design phase — design walk-throughs, design inspections, prologs and PDL, and SENs — have new aspects or applications that are introduced during detailed design. These are discussed in the paragraphs that follow.

Design Walk-throughs

During the detailed design phase, one or two design walk-throughs are held for each subsystem. Participants include members of the development team, the requirements definition team, representatives of systems that will interface with the software under development, and managers. At these walk-throughs, members of the development team step through the subsystem's design, explaining its algorithms, interfaces, and operational aspects. Participants examine and question the design at a detailed level to uncover such issues as mis-matched interfaces, contradictions between algorithms, or potential performance problems.

Design walk-throughs and inspections are initiated during the preliminary design phase. Section 5 defines both inspections and walk-throughs and explains the differences between the two methods.

TAILORING NOTE

Separate, formalized design walk-throughs are not always held on small tool development efforts. On these projects, walk-throughs may be combined with design reviews. The reviews are generally informal, since the products being examined will not be put under CCB control, and focus on system operability and the user interface.

The design of the user interface is specifically addressed in one or more additional walk-throughs. These sessions are attended by the future users of the system and concentrate on the users' point of view. The users learn how the system will look to them under representative operational scenarios, give feedback to developers, and provide a "reality check" on the updated requirements and specifications. Problems encountered with the specifications are documented on question-and-answer forms and submitted to the requirements definition team for action.

Design Inspections

Design inspections are the key methodology of the detailed design phase. Because the CDR is conducted primarily for the benefit of management and users, a detailed, component-by-component review of the design takes place only during design inspections. It is during these inspections that each separate design product is examined for correctness, completeness, and comprehensibility.

Design inspections are always conducted, regardless of the size or type of the project.

At a minimum, the inspection team consists of the moderator, the design's author, and another member of the development team. Units that require detailed knowledge of the application (such as the physics of flight dynamics) are inspected by the development team's application specialist.

On large projects, two or more of the author's peers will inspect the design and a representative from the quality assurance office may be included. Leaders of teams that are developing subsystems that interface with the components being inspected should also attend.

Each inspection session covers a set of logically related units (5 to 10 on average), for which design diagrams and unit-level PDL and prologs have been completed. Copies of the materials to be inspected are distributed to members of the inspection team several days before the session.

Each member individually reviews the materials for technical content and adherence to design standards, and comes to the inspection session prepared to comment on any flaws he or she has uncovered. The moderator records the errors, resolves disagreement, and determines whether reinspection is necessary. The author answers questions and takes action items to resolve the flaws that are identified. All participants are responsible both for finding errors and for ensuring that designs are traceable to requirements.

RULE

Every unit is important to the developing system and each new or modified unit must be reviewed with equal thoroughness. Neglecting a unit because it is reused software, it is not part of a "critical" thread, or its functions are "trivial" can be disastrous.

Design inspection checklists, such as the one shown in Figure 6-3, are distributed to reviewers with the inspection materials to remind them of key review criteria. A master checklist is compiled by the moderator and is used to certify the unit when inspection is complete.

Compiled Prologs and PDL

During the detailed design phase of an Ada project, the development team generates and compiles all PDL for the system. For this work, and for the code and test activities of subsequent phases, developers in the STL use sets of tools and utilities that are part of an Ada software development environment.

UNIT DESIGN INSPECTION CHECKLIST

Unit Name_______________________System__________Build/Release______________________
Task Number_____________________________________Initial Inspection Date______________
Inspection Moderator____

KEY INSPECTION QUESTIONS	Yes	No	Corrected
1. Does the design present a technically valid way of achieving the unit's assigned function?	[ ]	[ ]	[ ]
2. Is all data required by the unit available and defined?	[ ]	[ ]	[ ]
3. Is the dependency between each input and output argument and the processing apparent in the design?	[ ]	[ ]	[ ]
4. Is it clear from the design where the outputs from the unit are generated?	[ ]	[ ]	[ ]
5. Is the dependency between each external reference (e.g., unit file, record) and the processing apparent in the design?	[ ]	[ ]	[ ]
6. Does the design include the necessary error detection and recovery logic?	[ ]	[ ]	[ ]
7. Is the design, as written, sufficient to handle the upper and lower bounds associated with unit inputs (especially arguments)?	[ ]	[ ]	[ ]
ADDITIONAL INSPECTION QUESTIONS			
8. Are the prolog and PDL consistent with the unit's design diagram?	[ ]	[ ]	[ ]
9. Does the PDL define logic as opposed to code?	[ ]	[ ]	[ ]
10. Does the prolog contain enough information to describe the unit clearly to the unfamiliar reader?	[ ]	[ ]	[ ]
11. Do both the prolog and PDL conform to applicable standards?	[ ]	[ ]	[ ]

ACTION ITEMS AND COMMENTS
(List on a separate sheet. Refer to questions above by number.)

INSPECTION RESULTS

1. If all answers to 1-11 were "Yes, " the unit's design passes. Check here and sign below. [ ]
2. If there are serious deficiencies in the design (e.g., if more than one key question was answered "No") the author must correct the unit design and the moderator must schedule a reinspection.
   Scheduled date for reinspection:____
3. If there are minor deficiencies in the design, the author must correct the unit design and hold a followup meeting with the moderator.
   Scheduled date for followup meeting:____

Moderator's signature certifies that this unit meets all applicable standards and satisfies its requirements, and that any identified deficiencies have been resolved (applicable at initial inspection, followup meeting, or reinspection).

Moderator signature:_________________________________ Date_____________________

Figure 6-3. Checklist for a Unit Design Inspection

DEFINITION

In Ada, the term program library refers not to a library of programs, but to a library of compilation units that comprise one or more programs. To detect consistency errors before an entire program is constructed, the Ada compiler cross-checks information from separately compiled units. The program library is the mechanism by which the compiler retains the information needed to perform these checks efficiently.

The Ada development environment provides a *program library manager* that functions as the user interface to the Ada compiler and the linker. The program library manager keeps track of which compilation units are in the library, when they were compiled, which ones have been made obsolete by more recent compilations, and the dependencies among units.

The development environment also includes a *language-sensitive editor (LSE)*. The LSE provides a template for the Ada language that allows developers to enter and edit prologs and PDL interactively.

Ada developers use an additional tool to expedite the generation of package bodies. This utility reads a "bare-bones" package specification, enhances it to conform to local Ada styles (see Reference 22), and uses the specification to build templates for the package body and subprograms.

The other components of the Ada development environment are used primarily during the implementation phase of the life cycle and are described in Section7.

Software Engineering Notebooks (SENs)

DEFINITION

Throughout this document, the term "module" is used to denote a collection of logically related units. In the flight dynamics environment, a module usually consists of 5 to 10 units.

TAILORING NOTE

On Ada projects, one SEN is used to store documentation for each package. That is, the current listings, inspection checklists, and relevant design diagrams for the package's specification, body, and subprograms are maintained together in a single notebook.

By the end of the detailed design phase, the development team's librarian will have created a SEN for each unit and/or module in the system. The developer uses the SEN to store all documentation that pertains to a unit, including the current listing of the unit's prolog and PDL, the unit design inspection checklist, design diagrams, and notes documenting design decisions. Through unit code, test, and integration, each developer retains the SENs for the units for which he or she is responsible. When the units in a module have been coded, tested, and certified, they are ready to be placed under configuration management; at this point, the developer gives the SENs to the librarian who files them in the project library.

MEASURES

Objective Measures

The objective measures used during the preliminary design phase are also the yardsticks used for detailed design, with one addition — CPU hours. The measures to be collected are as follows:

- The number of unit designs that have been certified versus the number identified
- Requirements questions and answers, TBDs, and changes
- Staff hours
- Estimates of system size, effort, schedule, and reuse
- Total CPU hours used to date

The source of these data and the frequency of data collection and evaluation are shown in Table 6-1. The paragraphs that follow provide specific recommendations for evaluating these measures during detailed design, and supplement the guidance given in Section 5.

Evaluation Criteria

TBD requirements

The number of TBD requirements is a vital metric in this phase. Ideally, *all* TBD requirements are resolved by the end of the phase. If this goal is impossible to achieve, the management team must assess how the remaining TBDs will affect system size, system design, staff hours, costs, and schedule, and then evaluate the feasibility of continuing. In the flight dynamics environment, implementation should be postponed if "mission critical" requirements remain unresolved or if more than 10 percent of the total number of requirements are still TBD.

requirements changes

A large number of specification modifications in the detailed design phase is usually an indication that requirements and specifications are unstable or erroneous. The management team must assume that the level of specification modifications will remain high throughout the implementation and system test phases, and should reevaluate system size estimates and schedules accordingly (Figure 6-4).

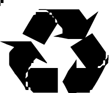
size and effort estimates

By the end of preliminary design, managers know the projected number of units in the system. By the end of the *detailed* design phase, all units to be reused will have been identified, along with the degree of modification needed to incorporate them into the new system. Managers can combine these new figures with productivity rates from past projects to reestimate the effort hours and staffing

Table 6-1. Objective Measures Collected During the Detailed Design Phase

MEASURE	SOURCE	FREQUENCY (COLLECT/ANALYZE)	DATA COLLECTION	
			CONTINUED	BEGUN
Staff hours (total and by activity)	Developers and managers (via PRFs)	Weekly/monthly	*	
Computer use (CPU hours and runs)	Automated tool	Weekly/biweekly		*
Requirements (changes and additions to baseline)	Managers (via DSFs)	Biweekly/biweekly	*	
Requirements (TBD specifications)	Managers (via DSFs)	Biweekly/biweekly	*	
Requirements (questions/answers)	Managers (via DSFs)	Biweekly/biweekly	*	
Estimates of total SLOC (new, modified, reused), total units, total effort , and schedule	Managers (via PEFs)	Monthly/monthly	*	
Status (units planned/ designed/certified)	Managers (via DSFs)	Biweekly/biweekly	*	

levels necessary to complete development. (See Table 3-2 of Reference 12 for guidance in computing these estimates.)

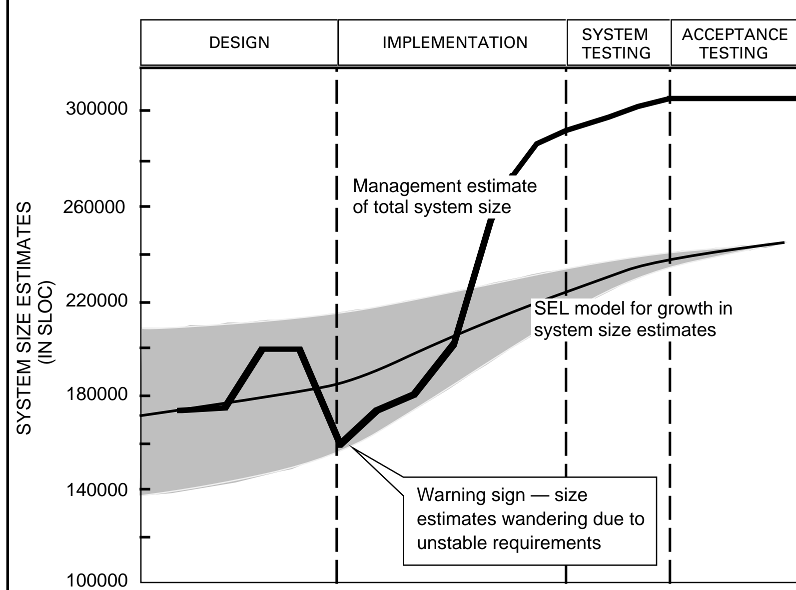


Recent SEL studies have shown that the relative cost of reusing existing units, as a percentage of the cost to develop the unit newly, is 20 percent for FORTRAN projects and 30 percent for Ada projects. The higher percentage for Ada is correlated to a significantly greater proportion of reused to new units on these projects as compared with FORTRAN efforts.

Near the end of the phase, size estimates can balloon unexpectedly if many units are moved from the "reused" to the "new" category. Managers need to ensure that decisions to create new units rather than reuse existing ones are justified and not merely a manifestation of the NIH ("not invented here") syndrome.

computer use

Computer use, as expressed in CPU hours or the number of sessions/job runs, is a key indicator of progress during design and implementation. On a typical flight dynamics project, a small amount of CPU time should be recorded during the design phases as the development team conducts prototyping efforts and enters PDL. Because Ada PDL is compiled, more CPU time should be logged on Ada projects.



Symptom: Size estimates increase then drop during detailed design.

Cause: Excessive requirements changes and ineffective change control mechanisms. Requirements that were deleted from specifications had to be restored during the implementation phase.

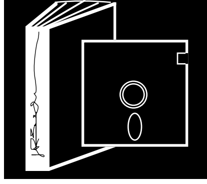
Corrective Actions: Assume that the instability of requirements will lead to a high level of specification modifications during implementation and testing. Analyze risks, replan size estimates accordingly, and request additional budget.

NOTE: In the SEL environment, a large number of TBDs in the requirements and specifications, combined with a substantial number of requirements changes, typically cause a system to grow up to 40 percent larger than is estimated at the time of PDR. As the details of the unknown portions of the system become clear, the size estimate grows more rapidly. The range of accepted growth (shown in grey) narrows as the system becomes more defined.

Figure 6-4. Example of the Impact of Requirements Changes on Size Estimates — the UARS Attitude Ground Support System

A lack of CPU hours on a project that is three-quarters of the way through the detailed design phase should raise a red flag. The management team should investigate to determine whether the team is avoiding using the computer because of inadequate training or is mired in redesign as a result of specification modifications.

PRODUCTS



The development team's primary product for this phase is the completed design for the system, including unit prologs and PDL, as recorded in the detailed design document. In addition, the management team produces a build plan for implementing the design, and the development team begins to develop build test plans. These products are described in the paragraphs that follow.

Detailed Design Document

During the detailed design phase, the preliminary design report is expanded and refined to reflect the results of detailed design activities. Before CDR, this *detailed design document* is completed and distributed for review. The format and contents of the document are shown in Figure 6-5.

The Build Plan

The *build plan* describes the strategy that will be applied in constructing the system during the implementation phase. The plan defines the sequence in which software components are coded and integrated into executable subsystems and the order in which these subsystems are combined into systems.

NOTE

See SECTION 2 for definitions of the terms build and release and for guidance in determining the number of builds and releases that are needed as a function of project size.

The plan usually contains three parts:

- An itemization of the capabilities that will be provided in each build or release
- The rationale, including relevant constraints, for providing specified capabilities in a particular build
- The implementation schedule

NOTE

Additional builds are required on projects using Cleanroom methodology. On Cleanroom projects, a build should consist of portions of the software that can be readily tested and integrated together and should last no more than 2 or 3 months.

A preliminary build plan is usually generated during the preliminary design phase. During detailed design, the build strategy is expanded and refined until, at the conclusion of the phase, the updated strategy is included in the SDMP and presented for evaluation at the CDR.

Builds must be planned to accommodate user needs for operational capabilities or intermediate products. Plans must also allow for fluctuating and TBD requirements. Initially, the development team determines the optimum plan for implementing the system from a technical viewpoint. The management team then analyzes the plan and decides how it must be perturbed to accommodate the user, external (e.g., mission) schedules, specification modifications, and unknowns. Both the optimum and adjusted plans are presented at CDR.

DETAILED DESIGN DOCUMENT

This document is the primary product of the detailed design phase. To complete the document, the development team updates similar material from the preliminary design report and adds greater detail. The suggested contents are as follows:

1. **Introduction** — purpose and background of the project, overall system concepts, and document overview

2. **Design overview**
   a. Design drivers and their order of importance
   b. Reuse strategy
   c. Discussion and high-level diagrams of the selected system design, showing hardware interfaces, external data interfaces, interconnections among subsystems, and data flow
   d. Traceability matrix of major components against requirements and functional specifications
   e. Design status
      (1) List of constraints, concerns, and problem areas and their effects on the design
      (2) List of assumptions and possible effects on design if they are wrong
      (3) List of TBD requirements and an assessment of their effect on system size, required effort, cost, and schedule
      (4) ICD status
      (5) Results of prototyping efforts
   f. Development environment

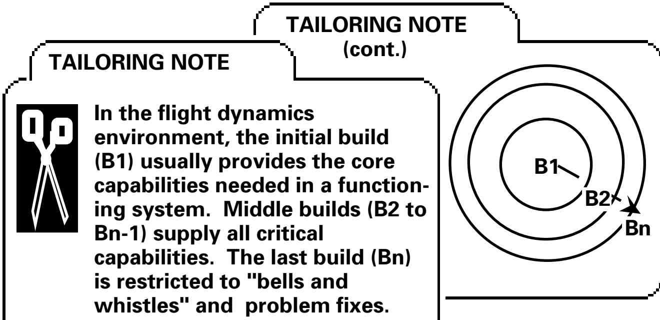
3. **Operations overview**
   a. Operations scenarios/scripts
   b. System performance considerations

4. **Design description** for each subsystem or major functional breakdown:
   a. Overall subsystem capability
   b. Assumptions about and restrictions to processing in each mode
   c. Discussion and high-level diagrams of subsystem, including interfaces, data flow, and communications for each processing mode
   d. High-level description of input and output
   e. Detailed description of processing keyed to operator-specified input and actions in terms of points of control, functions performed, and results obtained (both normal and abnormal, i.e., error processing and recovery)
   f. Structure charts or object-oriented diagrams expanded to the unit level, showing interfaces, data flow, interactive control, interactive input and output, and hardcopy output
   g. Internal storage requirements, i.e., description of arrays, their size, their data capacity in all processing modes, and implied limitations of processing
   h. Detailed input and output specifications
      (1) Processing control parameters, e.g., NAMELISTs
      (2) Facsimiles of graphic displays for interactive graphic systems
      (3) Facsimiles of hardcopy output
   i. List of numbered error messages with description of system's and user's actions
   j. Description of COMMON areas or other global data structures
   k. Prologs and PDL for each unit (normally kept in a separate volume because of size)

5. **Data interfaces**—updated from description in preliminary design report (see Figure 5-5)

Figure 6-5. Detailed Design Document Contents

Each build should address a coherent subset of the requirements and specifications and take three to five months to implement. Each build should cover a set of completed units; that is, the build plan should not require modifications or enhancements to individual units during later builds.



The first build must be kept simple, particularly if the development team is unfamiliar with the development environment or programming language being used. The next builds should address high-risk specifications and critical software capabilities, such as performance requirements, major control functions, and system and user interfaces. The build strategy must ensure that capabilities that can have a major impact on the software design are completed early, so that any problems can be handled while there is still time to recover.

NOTE

The build plans for flight dynamics projects also include the dates by which developers need to receive unit-level, analytic test plans from the acceptance test team. See SECTION 7 for detailed information about the purpose and content of these plans.

Since the last build will grow to include the implementation of specification modifications and the resolution of problems remaining from earlier builds, it should be kept small in initial planning. The next-to-last build, therefore, must supply all crucial system capabilities. If specification modifications that add new features to the system are received during the implementation phase, additional builds that extend the phase may be needed to ensure that existing builds can proceed on schedule.

Build Test Plans

As soon as a build has been defined, the development team can begin to specify the tests that will be conducted to verify that the software works as designed and provides the capabilities allocated to the build or release. The development team executes the *build test plan* immediately following integration of the build.

The test plan for the first build is defined during the detailed design phase. Any modifications to the overall testing strategy that are made as a result of defining this first test plan are presented at the CDR.

Test plans for the remaining builds are generated during the implementation phase.

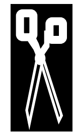
The format and content of build test plans are described in Section 7.

CRITICAL DESIGN REVIEW

The detailed design phase culminates in the CDR. This review is attended by the development team and its managers, the requirements definition team and its managers, quality assurance representatives, user representatives, the CCB, and others involved with the system. Participants evaluate the detailed design of the system to determine whether the design is sufficiently correct and complete for implementation to begin. They also review the build plan to ensure that the implementation schedule and the capabilities allocated to the builds are feasible.

The emphasis at CDR is on modifications — to requirements, high-level designs, system operations, and development plans — made since the PDR. Speakers should highlight these changes both on the slides and during their presentations, so that they become the focus of the review. The CDR also provides an opportunity for the development team to air issues that are of concern to management, the mission project office, quality assurance personnel, and the CCB.

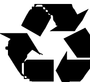
TAILORING NOTE



For very large projects , a CDR should be held for each major subsystem and/or release in order to cover all aspects of the system and to accommodate changing requirements. On such projects, it is vital to have one review, i.e., a System Design Review, that covers the entire system at a high level.

Figure 6-6 shows the recommended CDR format. An outline and suggested contents of the CDR hardcopy material are presented in Figure 6-7. Note that material that was covered at PDR is not presented again, except as needed to contrast changes. For this concise format to be effective, participants must be familiar with the project background, requirements, and design. They should have attended the PDR and studied the detailed design document before the meeting.

REUSE NOTE



At the CDR, developers present statistics showing the number and percentage of components to be reused, and which of these are drawn from the RSL. They also present key points of the detailed reuse strategy, identify any changes to the reuse proposal that have been made since PDR, and describe new/revised reuse tradeoff analyses.

Reviewers should address the following questions:

- *Does the design satisfy all requirements and specifications?*

CDR FORMAT

Presenters — software development team

Participants
- Requirements definition team
- Quality assurance representatives from both teams
- Customer interfaces for both teams
- User representatives
- Representatives of interfacing systems
- System capacity/performance analysts
- CCB

Attendees should be familiar with the project background, requirements, and design.

Schedule — after the detailed design is completed and before implementation is begun

Agenda — selective presentation of the detailed design of the system. Emphasis should be given to changes to the high-level design, system operations, development plan, etc. since PDR.

Materials Distribution
- The detailed design report is distributed at least 10 days before the CDR.
- Hardcopy material is distributed a minimum of 3 days before the review.

Figure 6-6. CDR Format

- *Are the operational scenarios acceptable?*
- *Is the design correct? Will the transformations specified produce the correct output from the input?*
- *Is the design robust? Is user input examined for potential errors before processing continues?*
- *Have all design guidelines and standards been followed? How well have data usage and access been localized? Has coupling between units (i.e., interunit dependency) been minimized? Is each unit internally cohesive (i.e., does it serve a single purpose)?*
- *Is the design testable?*
- *Is the build schedule structured to provide early testing of end-to-end system capabilities? Is the schedule reasonable and feasible for implementing the design?*

HARDCOPY MATERIAL FOR THE CDR

1. **Agenda** — outline of review material

2. **Introduction** — background of the project, purpose of the system, and an agenda outlining review materials to be presented

3. **Design overview** — major design changes since PDR (with justifications)
   a. Design diagrams, showing products generated, interconnections among subsystems, external interfaces
   b. Mapping of external interfaces to ICDs and ICD status

4. **Results of prototyping efforts**

5. **Changes to system operation** since PDR
   a. Updated operations scenarios/scripts
   b. System performance considerations

6. **Changes to major software components** since PDR (with justifications)

7. **Requirements traceability matrix** mapping requirements to major components

8. **Software reuse strategy**
   a. Changes to the reuse proposal since PDR
   b. New/revised reuse tradeoff analyses
   c. Key points of the detailed reuse strategy, including software components to be reused in future projects
   d. Summary of RSL contributions — what is used, what is not, reasons, statistics

9. **Changes to testing strategy**
   a. How test data are to be obtained
   b. Drivers/simulators to be built
   c. Special considerations for Ada testing

10. **Required resources** — hardware required, internal storage requirements, disk space, impact on current computer usage, impacts of compiler

11. **Changes to the SDMP** since PDR

12. **Implementation dependencies (Ada projects)** — the order in which components should be implemented to optimize unit/package testing

13. **Updated software size estimates**

14. **Milestones and schedules** including a well-thought-out build plan

15. **Issues, risks, problems, TBD items**
    a. Review of TBDs from PDR
    b. Dates by which TBDs and other issues must be resolved

Figure 6-7. CDR Hardcopy Material

EXIT CRITERIA

To determine whether the development team is ready to proceed with implementation, the management team should consider the following questions:

- Are all design diagrams complete to the unit level? Have all interfaces — external and internal — been completely specified?
- Do PDL and prologs exist for all units? Have all unit designs been inspected and certified?
- Have all TBD requirements been resolved? If not, how will the remaining TBDs impact the current system design? Are there critical requirements that must be determined before implementation can proceed?
- Have the key exit criteria for the phase been met? That is, has the detailed design document been completed, has the CDR been successfully concluded, and have responses been provided to all CDR RIDs?

When all design products have been generated and no critical requirements remain as TBDs, the implementation phase can begin.

LIFE CYCLE PHASES	REQUIREMENTS DEFINITION	REQUIRE-MENTS ANALYSIS	PRELIMI-NARY DESIGN	DETAILED DESIGN	IMPLEMENTATION	SYSTEM TESTING	ACCEPTANCE TESTING

SECTION 7

THE IMPLEMENTATION PHASE

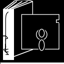
PHASE HIGHLIGHTS

ENTRY CRITERIA

- Detailed design document generated
- CDR completed
- CDR RIDs answered

EXIT CRITERIA

- All system code and supporting data generated and tested
- Build test plans successfully executed
- System test plan completed
- User's guide drafted



PRODUCTS

- System code and supporting data
- Build test plans and results
- System and analytical test plans

MEASURES

- Units coded/ code-certified/ test-certified vs. units identified
- Requirements Q&As, TBDs, and changes
- Estimates of system size, effort, schedule, and reuse
- Staff hours
- CPU hours
- SLOC in controlled libraries (cumulative)
- Changes and errors (by category)

METHODS AND TOOLS

- Code reading
- Unit testing
- Module integration testing
- Build testing
- Configuration management
- SENs
- CASE

KEY ACTIVITIES

Requirements Definition Team

- Resolve any remaining requirements issues
- Participate in build design reviews (BDRs)

Development Team

- Code new units and revise existing units
- Read new and revised units
- Test and integrate each unit/module*
- Plan and conduct build tests*
- Prepare the system test plan*
- Draft the user's guide
- Conduct build design reviews

Management Team

- Reassess schedules, staffing, training, and other resources
- Organize and coordinate subgroups within the development team
- Control requirements changes
- Ensure quality in processes and products
- Direct build design reviews
- Coordinate the transition to system testing

Acceptance Test Team

- Complete the acceptance test plan draft
- Complete the analytical test plan

*Projects that use the Cleanroom methodology have an independent test team. Developers prepare and verify each other's code, then submit the code (in small builds) to the test team for compilation and testing.

OVERVIEW

The purpose of the implementation phase is to build a complete, high-quality software system from the "blueprint" provided in the detailed design document. The implementation phase begins after CDR and proceeds according to the build plan prepared during the detailed design phase. For each build, individual programmers code and test the units identified as belonging to the build, integrate the units into modules, and test module interfaces.

At the same time, the application specialists on the development team prepare plans designed to test the functional capabilities of the build. Build regression tests — a selection of tests already conducted in previous builds — are included in each build test plan to ensure that newly added capabilities have not affected functions implemented previously. All build test plans are reviewed for correctness and completeness by the management team.

When all coding, unit testing, and unit integration testing for the build are complete, selected members of the development team build the system from the source code and execute the tests specified in the build test plan. Both the management team and the development team review the test results to ensure that all discrepancies are identified and corrected.

As build testing progresses, the development team begins to put together the user's guide and the system description documents. A draft of the user's guide must be completed by the end of the implementation phase so that it can be evaluated during system testing. Material from the detailed design document is updated for inclusion in the system description document, which is completed at the end of the system test phase.

build design review

Before beginning the next build, the development team conducts a *build design review (BDR)*. The formality of the BDR depends on the size of the system. Its purpose is to ensure that developers, managers, and customer representatives are aware of any specification modifications and design changes that may have been made since the previous review (CDR or BDR). Current plans for the remaining builds are presented, and any risks associated with these builds are discussed.

The plans for testing the completed system (or release) are also generated during the implementation phase. Application specialists from the development team prepare the system test plan, which is the basis for end-to-end testing during the next life cycle phase. At the same time, members of the independent acceptance test team

prepare the test plan that they will use during the acceptance test phase.

The implementation process for a single build is shown in Figure7-1. Figure 7-2 shows that these implementation processes are repeated for each build and that a larger segment of the life cycle — extending from the detailed design phase through acceptance testing — is repeated for each release.

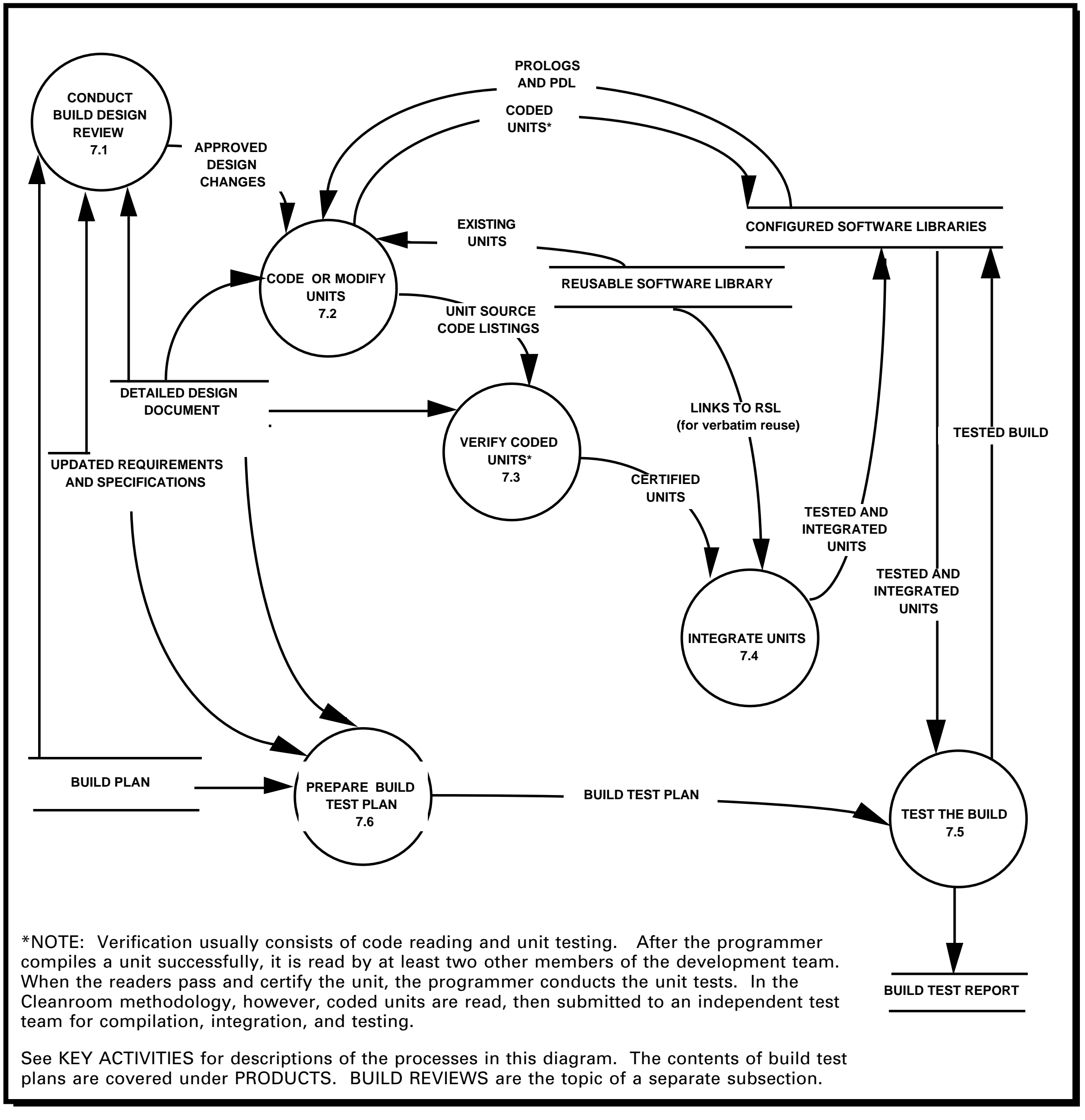


Figure 7-1. Implementing a Software Build

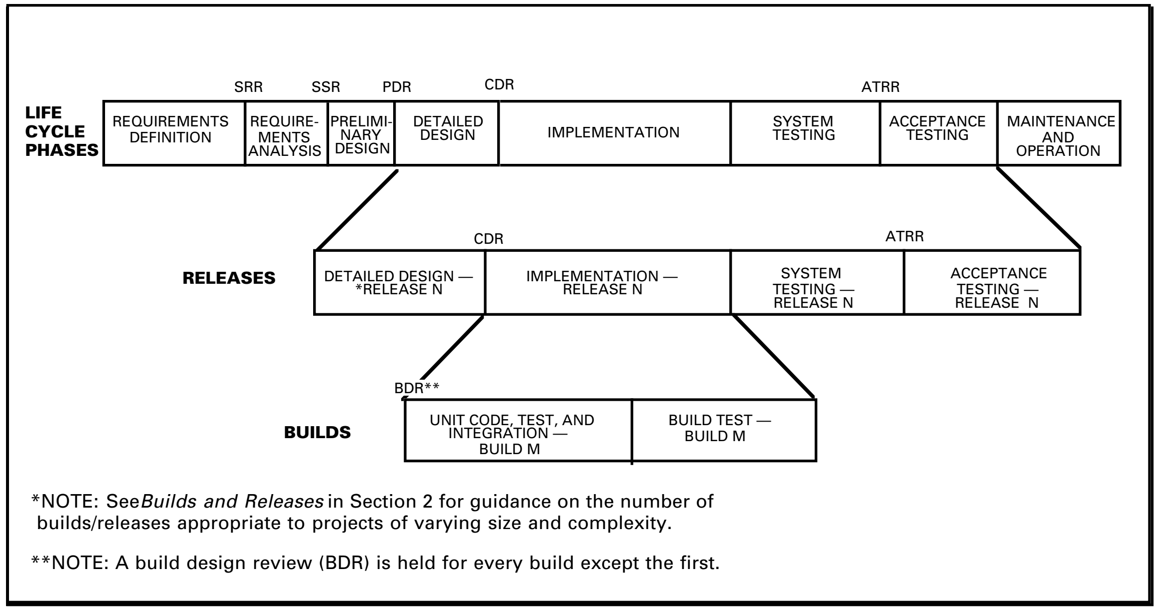


Figure 7-2. Phases of the Life Cycle Are Repeated for Multiple Builds and Releases

KEY ACTIVITIES

Although the activities of coding, code reading, unit testing, and integrating a single module are conducted sequentially, the development team implements each module in parallel with others. Unless the project is very small, the development team is partitioned into multiple groups during the implementation phase. Each group is assigned to develop one or more subsystems, and group members are each assigned one or more modules. During a given build, the group will code, read, and test the modules of its subsystem that are scheduled for the build. Therefore, coding, code reading, unit testing, and module testing activities may be conducted simultaneously at any given time.

During the implementation phase, the application specialists in the development team do not code and test units. Instead, they apply their expertise by reading the code of selected units (such as those with complex, application-specific algorithms) and inspecting unit and module test results. They also prepare the system test plan.

The key technical and managerial activities of the implementation phase are summarized below and shown on a timeline in Figure 7-3.

Activities of the Development Team:

- **Code new units** from the detailed design specifications **and revise existing units** that require modification. Code units so that each PDL statement can be easily matched with a set of coding statements. Use structured coding principles and local coding conventions (References 22 and 23). Prepare the command language (e.g., JCL or DCL) procedures needed to execute the units.

- **Read new and revised units.** Ensure that each unit is read by a minimum of two members of the development team who are not the unit's authors. Correct any errors that are found, reinspecting the unit as necessary. Certify all unit code.

> **DEFINITION**
>
> **Code reading is a systematic procedure for inspecting and understanding source code in order to detect errors or recommend improvements. It is described in detail in this section under METHODS AND TOOLS. Certification is a part of the quality assurance process wherein an individual signs a checklist or form as an independent verification that an activity has been successfully completed.**

- **Test each unit and module.** Prepare unit test procedures and data, and conduct unit tests. If the acceptance test team has provided an analytical test plan for the unit, complete the test cases specified in the plan and verify that the computational results are as expected. Have an experienced member of the development team review and certify all unit test procedures and results.

  Integrate logically related units into modules and integrate the modules into the growing build. Define and run enough tests to verify the I/O generated by the module and the interfaces among units within the module. Ensure that the results of module testing are reviewed and certified.

> **NOTE**
>
>
>
> **See METHODS AND TOOLS in this section for more detailed information on coding standards and unit testing. Test plans — analytic test plans, build test plans, and the system test plan — are described under PRODUCTS.**

- **Plan and conduct build tests.** Prepare the test plan for the build, and complete the preparation of command procedures and data needed for build testing.

> **NOTE**
>
> **On most projects, unit and module tests need not be performed separately. As more units for a particular module are developed, they are unit tested in the context of the units previously developed within the module.**
>
> **Units may be either executable or data. On Ada projects, the module takes the form of a package.**

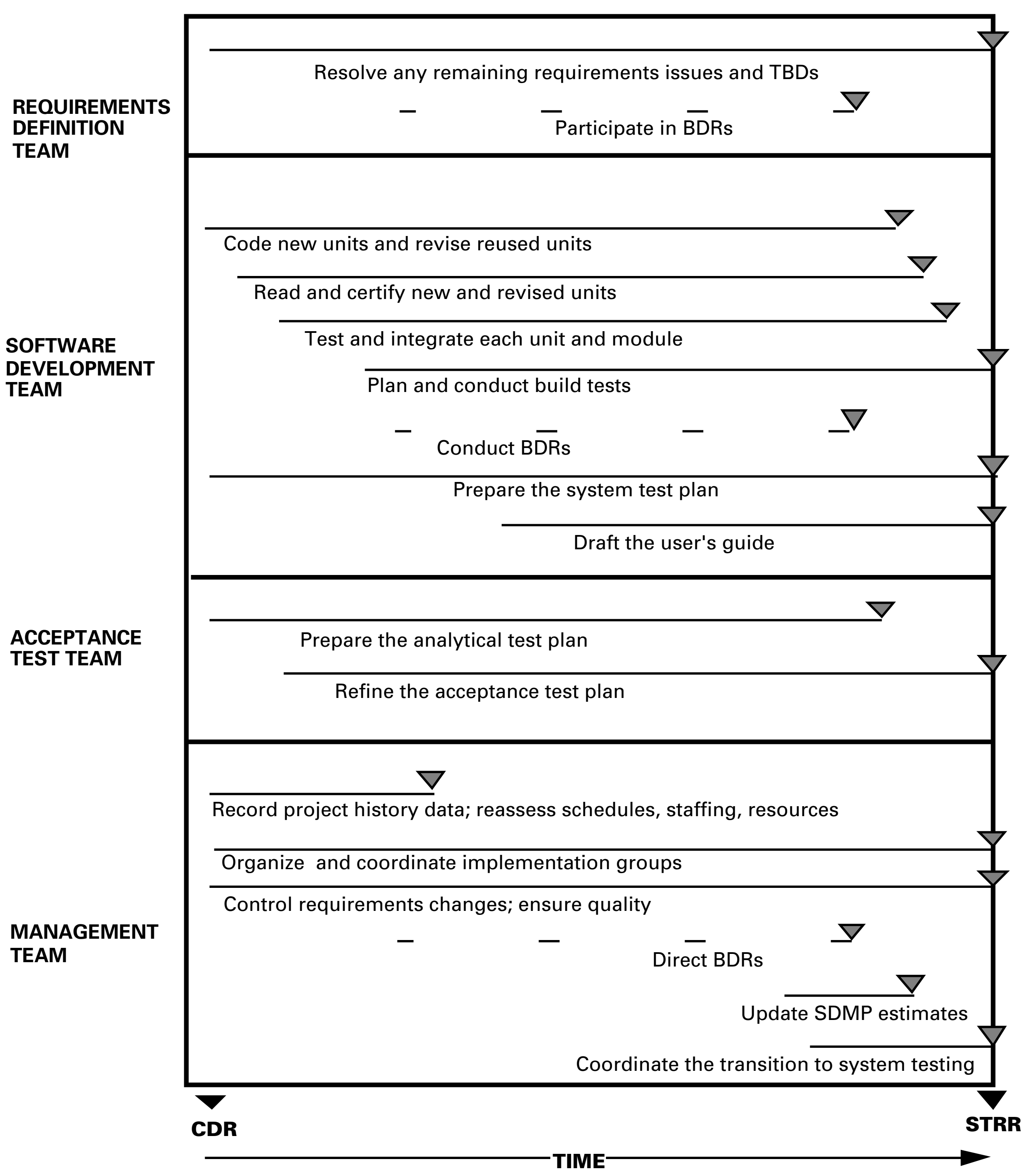


Figure 7-3. Timeline of Key Activities in the Implementation Phase

The load module (or *executable image)* for the build is created by the project librarian. When the load module is prepared, execute the tests specified by the test plan for the build. Ensure that all output needed for test evaluation is generated. Record any discrepancies between the results specified in the plan and actual results.

Correct all discrepancies that are found. When the affected units are repaired and tested, file a report of the changes with the project librarian. The librarian ensures that the configured libraries and test executables are updated with the revised units.

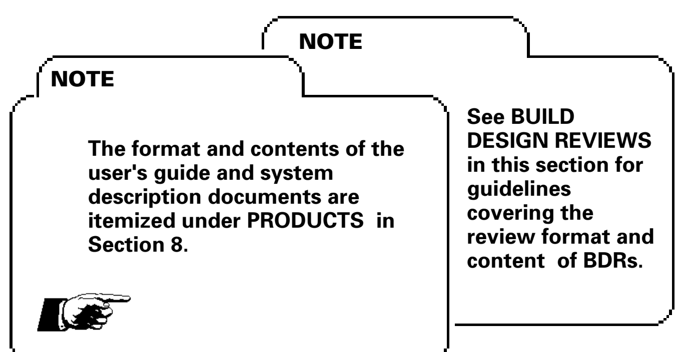
Rerun any tests that failed, and verify that all errors have been corrected. When all build tests have been successfully completed, prepare a written report of the test results.

- **Prepare the system test plan** for use during the system testing phase. Begin to develop the plan immediately after CDR, so that it will be ready by the end of the phase. Prepare the command procedures and input data needed for system testing.

- **Prepare a draft of the user's guide**, using sections of the detailed design document (the operations overview and design description) as a foundation.

  Begin work on the system description document by updating data flow/object diagrams and structure charts from the detailed design document.

- **Conduct a BDR** before every build except the first (changes to the design and/or build plan that apply to the first build are covered during the CDR). Ensure that all design changes are communicated to development team members, users, and other participants. Present the key points of the build plan, making certain that all participants understand their roles in the build, the schedule, and the interfaces with other groups or activities.

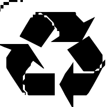


Activities of the Management Team

- **Reassess schedules, staffing, training, and other resources**. At the beginning of the phase, record measures and lessons learned from the detailed design phase and add this information to the draft of the software development history. As implementation progresses, use the size of completed units to refine estimates of total system size. Use measures of actual resources expended and progress during the implementation phase to update cost and resource estimates. (See *Measures* subsection and Reference 12.)

  Reestimate system size, effort required to complete, schedules, and staffing each time a build is completed. Toward the end of the implementation phase, update the SDMP with effort, schedule, and size estimates for the remaining phases.

REUSE NOTE



Managers should make frequent checks throughout the design and implementation phases to ensure that reuse is not being compromised for short-term gains in schedule or budget. Managers must actively promote the reuse of existing software and stress the importance of developing software that is reusable in the future.

- **Organize and coordinate subgroups within the development team.** At the beginning of the phase, organize the development team into small, three- to five-person groups. Assign each group a cohesive set of modules to implement. Regardless of the size of a module set, the same group should code and test the units and the integrated modules. As much as possible, ensure that the designer of a unit is also responsible for its coding and verification.

  Ensure that any new personnel joining the project during this phase are adequately trained in the standards and procedures being followed (including data collection procedures) and in the development language and toolset. Make experienced personnel available to direct new and/or junior personnel and to provide on-the-job training.

- **Control requirements changes.** Thoroughly evaluate the impacts of any specification modifications received during this phase. Report the results of this analysis to the requirements definition team and the customer.

  Ensure that customers and users of the system agree on the implementation schedule for any specification modifications that are approved. To minimize their impact on the build in progress, schedule the implementation of new features for later builds.

- **Ensure quality in processes and products.** Make spot checks throughout the phase to ensure adherence to configuration management procedures, quality assurance procedures, coding standards, data collection procedures, and reporting practices. Configuration management procedures — especially change control on the project's permanent source code libraries — are critical during the implementation phase when the staff is at its peak size and a large amount of code is being produced.

  Monitor adherence to the build plan. Know at all times the status of development activities and the detailed plans for development completion.

  Review the draft user's guide and system test plans. Participate in the inspection of test results for each build and assist the development team in resolving any discrepancies that were identified.

- **Direct all BDRs,** and ensure that any issues raised during the reviews are resolved.

- **Coordinate the transition to the system testing phase.** Staff the system test team with application specialists, and include one or two analysts to take responsibility for ensuring the mathematical and physical validity of the test results. (This will also guarantee that some analysts are trained to operate the software before acceptance testing.) Assign a lead tester to direct the system testing effort and to act as a final authority in determining the success or failure of tests.

  Ensure that the data and computer resources are available to perform the steps specified in the system test plan. Inform personnel of the configuration management and testing procedures to be followed and provide them with the necessary training.

  At the conclusion of the phase, hold an informal *system test readiness review (STRR)*. Use this meeting to assess whether the software, the system test team, and the test environment are ready to begin testing. Assign action items to resolve any outstanding problems and revise schedules accordingly.

Activities of the Requirements Definition Team

- **Resolve any remaining requirements issues**. If implementation is to proceed on schedule, all TBD requirements must be

resolved early in the phase. If any requirements cannot be defined because external, project-level information (e.g., spacecraft hardware specifications) is incomplete, notify upper management of the risks and the potential impact to development schedules. Obtain deadlines by which the missing information will be supplied, and work with the development team to adjust schedules accordingly. Prepare a plan to mitigate these risks and reduce the possible schedule delays and cost overruns.

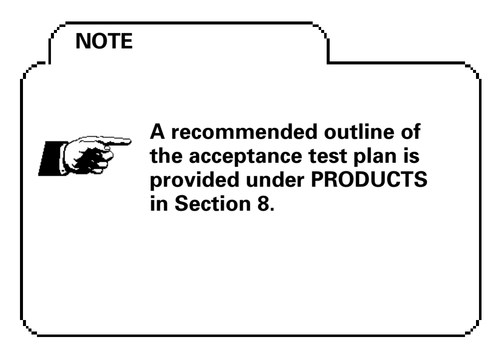
Ensure that changes to requirements that are of external origin (e.g., changes to spacecraft hardware) are incorporated into specification modifications without delay. Submit all specification modifications to the management team for technical evaluation and costing.

- **Participate in all BDRs.** Warn the development team of potential changes to requirements that could impact the design or otherwise affect current or remaining builds.

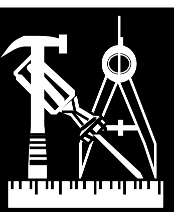
Activities of the Acceptance Test Team

- **Complete the draft of the acceptance test plan** that was begun during the detailed design phase. The draft should be provided to the development team before the start of system testing.

- **Prepare the analytical test plan.** At the beginning of a build, supply the development team with the parts of the analytical test plan that they will need during the build to verify the results of complex mathematical or astronomical computations.



METHODS AND TOOLS



The key methods and tools of the implementation phase are

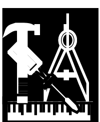
- Code reading
- Unit testing
- Module integration testing
- Build testing
- Configuration management
- SENs
- CASE

Each is discussed below.

Code Reading

The first step in the unit verification process is *code reading,* a systematic procedure for examining and understanding the operation of a program. The SEL has found code reading to be more cost effective in uncovering defects in software than either functional or structural testing and has formalized the code reading process as a key implementation technique (References 25 and 26).

NOTE



The SEL requires the use of structured coding principles and language-dependent coding standards. SEL coding standards are documented in References 22 (Ada) and 23 (FORTRAN). Reference 24 is one of the many sources of information on structured programming.

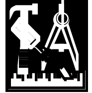
Code reading is designed to verify the logic of the unit, the flow of control within the unit, and boundary conditions. It is performed before unit testing, not afterwards or concurrently. Only code that has compiled cleanly should be presented for code reading.

Every new or modified unit is read by two or more team members. Each reader individually examines and annotates the code, reading it line by line to uncover faults in the unit's interfaces, control flow, logic, conformance to PDL, and adherence to coding standards. A checklist that is used by code readers on SEL-monitored projects is shown in Figure 7-4; its use fosters consistency in code reading by ensuring that the reader has a list of typical errors to look for and specific points to verify.

NOTE

The SEL's recommendation that at least two code readers examine each unit stems from a Cleanroom experiment (Reference 3). This project discovered that an average of only 1/4 of the errors in a unit were found by both readers. That is, 75% of the total errors found during code reading were found by only one of the readers.

NOTE



Some compilers allow the user to generate a cross-reference listing showing which variables are used in the unit and their locations. Code readers should use such listings, if available, to verify that each variable is initialized before first use and that each is referenced the expected number of times. Unreferenced variables may be typos.

The readers and the unit's developer then meet as a team to review the results of the reading and to identify problems that must be resolved. They also inspect the test plan for the unit. If errors have been discovered in the unit, the reader who is leading the meeting (the moderator) returns the unit to the implementor for correction. The unit is then reexamined. When all errors have been resolved, the moderator certifies that the code is satisfactory and signs the checklist. The implementor files the certified checklist in the SEN for the unit.

UNIT CODE INSPECTION CHECKLIST

Unit Name_______________________System__________Build/Release_____________________
Task Number___________________________________Initial Inspection Date_____________
Inspection Moderator____

KEY INSPECTION QUESTIONS	Yes	No	Corrected
1. Is any input argument unused? Is any output argument not produced?	[ ]	[ ]	[ ]
2. Is any data type incorrect or inconsistent?	[ ]	[ ]	[ ]
3. Is any coded algorithm inconsistent with an algorithm explicitly stipulated in PDL or in requirements/specifications?	[ ]	[ ]	[ ]
4. Is any local variable used before it is initialized?	[ ]	[ ]	[ ]
5. Is any external interface incorrectly coded? That is, is any call statement or file/database access incorrectly coded? Also, for an Ada unit, is any external interface not explicitly referenced/with'd-in?	[ ]	[ ]	[ ]
6. Is any logic path incorrect?	[ ]	[ ]	[ ]
7. Does the unit have multiple entry points or multiple, normal (non-error) exits?	[ ]	[ ]	[ ]
ADDITIONAL INSPECTION QUESTIONS			
8. Is any part of the code inconsistent with the unit design specified in the prolog and PDL?	[ ]	[ ]	[ ]
9. Does the code or test plan contain any unauthorized deviations from project standards?	[ ]	[ ]	[ ]
10. Does the code contain any error messages that might be unclear to the user?	[ ]	[ ]	[ ]
11. If the unit was designed to be reusable, has any hindrance to reuse been introduced in the code?	[ ]	[ ]	[ ]

ACTION ITEMS AND COMMENTS
(List on a separate sheet. Refer to questions above by number.)

INSPECTION RESULTS

1. If all answers to 1-11 were "No, " the unit's code passes. Check here and sign below. [ ]
2. If there are serious deficiencies in the code (e.g., if more than one key question was answered "Yes") the author must correct the unit design and the moderator must schedule a reinspection.
   Scheduled date for reinspection:____
3. If there are minor deficiencies in the code, the author must correct the unit design and hold a followup meeting with the moderator.
   Scheduled date for followup meeting:____

Moderator's signature certifies that this unit meets all applicable standards and satisfies its requirements, and that any identified deficiencies have been resolved (applicable at initial inspection, followup meeting, or reinspection).

Moderator Signature:_________________________________ Date:_____________________

Figure 7-4. Sample Checklist for Code Inspection

Each member of the development team should be assigned units to read. If only one or two developers are appointed to read all the units, the other team members will lose an opportunity to gain expertise and increase their understanding of the system.

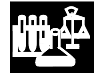
The code reader for a particular unit should not be selected by the unit's developer, but by the task leader. The choice of code reader should be appropriate to the character, complexity, and criticality of the unit. For example, units that contain physical or astronomical calculations should be read by application specialists who are familiar with the requirements and able to uncover analytical errors. Likewise, control units that use operating system services should be read by an operating system expert, and those that interface with a DBMS should be examined by a database specialist.

Unit Testing

Unit testing is the second step in verifying the logic, functionality, computations, and error handling of a unit. The intent of unit testing is to confirm that the unit provides the capability assigned to it, correctly interfaces with other units and data, and is a faithful implementation of the unit design.

In general, the developer who coded the unit executes the tests identified in the unit test plan; independent testers are not required unless the unit must comply with stringent safety or security requirements.

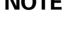
NOTE



On projects employing the Cleanroom methodology, no testing is conducted at the unit level. When a unit has been read and certified, it is submitted to an independent test team for compilation, integration, and functional testing. The tests that are conducted are a statistically selected subset of system tests.

The test plan should be tailored for the type of unit being tested. Structural (path) testing is critical for units that affect the flow of control through the system. The test plan for such a unit is generated by the developer from the unit's design and should include a sufficient number of test cases so that each logic path in the PDL is executed at least once. For units whose function is primarily computational, the developer may execute an analytical test plan. Analytical test plans are prepared by the acceptance test team to assist developers in verifying the results of complex mathematical, physical, and astronomical calculations (see *Products*). Units that are part of the user interface are tested using yet another approach — one that ensures that each of the user options on the screen is thoroughly exercised.

When unit testing is complete, the test results are reviewed by the developer's team leader or application specialist. The reviewer certifies the completeness and correctness of the test. That is, he or she checks the results against the test plan to ensure that all logic paths have been tested and verifies that the test results are accurate. As with code reading, use of a checklist is recommended to assist reviewers and maintain consistency.



The use of a symbolic debugger can greatly improve the efficiency of unit testing. The output generated by the symbolic debugger is filed in the SEN for the unit.

The unit test plan and test results are maintained in the SEN for the unit. If extensive changes are made to the unit at a later time, the unit code must be reread, and the unit must be retested and certified.

The management team determines the level of rigor in unit testing that is most cost effective for the project. For example, in some projects it may be more efficient to conduct testing at the module level than to test individual units. Indeed, for Ada projects, unit testing should generally be conducted within the context of the module (i.e., Ada package).

Module Integration Testing

Developers integrate individual, tested units into modules, then integrate these modules into the growing build. The method of integration testing that is used should be appropriate to the design of the system. Menu-driven systems, for example, lend themselves to either top-down or thread testing (Figure 7-5). In contrast, systems with complex, computational utilities may benefit from a bottom-up approach. As in unit testing, integration test plans and results are reviewed and certified by other members of the development team.

In the SEL environment, modules are verified using the existing, previously tested build as a test bed. Units not yet implemented exist in the module as stubs; that is, they contain no executable instructions except to write a message that the unit was entered and has returned control to the calling unit. This approach tests both the module's integration into the growing system and the internal code of the units that it comprises. Test drivers, which must themselves be coded and tested, are thus eliminated, and higher-level modules are exercised more frequently.

component origination form (COF)

When a module has been integrated, tested, and certified, the developer completes a *component origination form (COF)* for each of the units in the module. This SEL form has a dual function. The information provided on the form is stored in the SEL database and

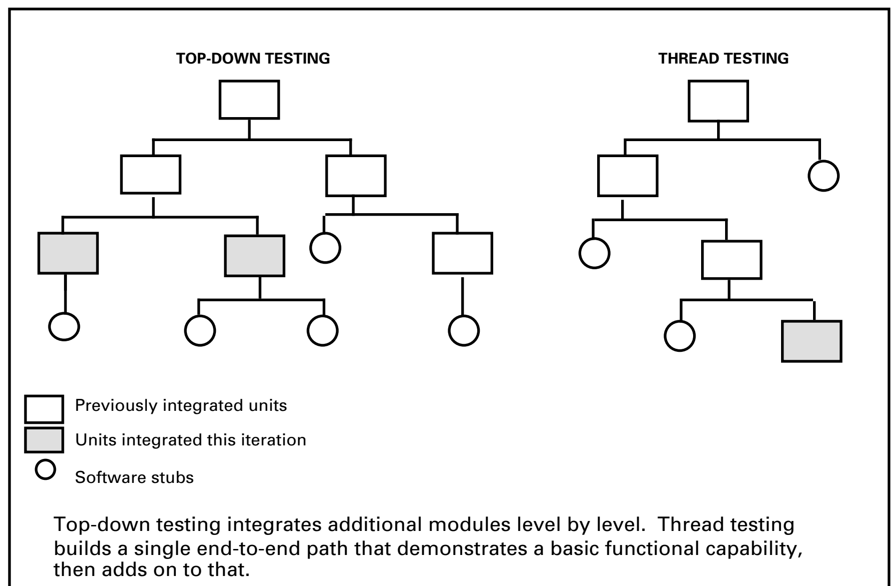


Figure 7-5. Integration Testing Techniques

used to track system composition, growth, and change throughout the project's life cycle. The form is also a key configuration management tool for the project librarian, who uses the source file information on the form to enter completed units into the project's controlled software library.

Build Testing

After all modules in the build have been coded and tested, the development team conducts *build tests* on the software in the controlled library. The purpose of build testing is to verify that the software provides the functionality required of the build and is a correct implementation of the design. Build regression tests are also conducted to ensure that functions provided by previous builds have not been adversely affected by the new components.

Build tests are executed by selected members of the development team following a formal build test plan (as described under *Products* in this section). The project librarian builds the executable images of the software from the tested modules in the configured library. The testers generally use a test checklist or report form to record the results of each test case as it is executed.

The results of tests are rigorously evaluated by developers, application specialists, and the management team. Operational difficulties, abnormal terminations, and differences between actual

and expected test results are recorded on special report forms. These *discrepancy reports* are used to ensure that each problem that is observed is resolved.

discrepancy reports

The development team locates the cause of each discrepancy and corrects the appropriate units. For each logical change that is made to controlled software, developers submit a *change report form (CRF).* This SEL form is used to gather information about the character of the changes made, their source, the effort required, and the number of changes due to errors.

change report form (CRF)

Configuration Management and SENs

During the implementation phase, adherence to configuration management procedures becomes critical. Source code is generally placed under configuration control one module at a time. When the units in the module have been coded, tested, and certified, the developer submits COFs for the units to the project librarian. The librarian moves the units into the project's configured source code libraries and files the units' SENs. Any further changes to these units must be approved by the development team leader or designated application specialist. The developer must check out the appropriate SENs, update the unit(s), fill out one or more CRFs, and update and return the SENs to the project library. Changes to configured libraries are made solely by the project librarian, who replaces configured units with the updated versions.

NOTE

For large systems, the number of discrepancies that must be rectified can be substantial. Managers must track these discrepancies, assign personnel to resolve them, set dates for resolution, and verify that all discrepancies have been corrected. Use of a tracking tool, such as CAT (Reference 27) or a PC-based DBMS, makes this task easier.

The project librarian maintains the central project library, adding to it all documentation produced during the implementation phase: SENs for completed units/modules, test plans and results for each build, drafts of the user's guide and system test plan, and system description information. The librarian also catalogs and stores CRFs for any changes made to software under configuration management, and files specification modifications and updates to design documents.

The management of a project's controlled source code libraries can be greatly facilitated by the use of an online configuration management tool. In the flight dynamics environment, DEC's Code Management System (CMS) is used to manage software developed

in the STL's VAX environment. PANVALET and CAT are used for systems developed, operated, and maintained in the IBM environment of the FDF (Reference 27).

Configuration management tools can be used to store all code, test drivers, data, and executable images; to track changes from one version to the next; and, most importantly, to provide access control. CMS, for example, allows the developer or librarian to reconstruct any previous version of a library element, tracks who is currently working on the element, and maintains a record of library access. With a configuration management tool, the project librarian can readily maintain multiple versions of the system, called *baselines*, each of which represents a major stage in system development. Baselines are generally established for each build, for system testing, for acceptance testing, and for operational use.

CASE

Use of CASE tools can yield significant benefits during the implementation phase. The following tools are those that have been found to be most beneficial in the SEL's development environment.

language-sensitive editors

Language-sensitive editors, such as VAX LSE, provide language-specific templates that help the programmer to enter and compile code efficiently, to review resultant diagnostic messages, and to correct errors — all within a single editing session. *Debuggers* allow the developer to suspend execution of a program, locate and correct execution errors, and return to program execution interactively.

static code analyzers

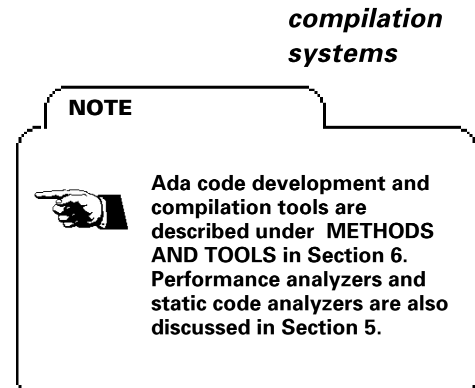
Static code analyzers, such as the VAX Source Code Analyzer (SCA), the RXVP80 static analyzer, and the Static FORTRAN Source Code Analyzer Program (SAP), provide cross-referencing capabilities among source files. They allow the developer to locate subprograms, variables, and data references and to answer questions such as "in which units is variable X used?". Additional functions provided by some analyzers include the display of call-trees and the extraction of design information.

performance analyzers

Performance analyzers (e.g., the VAX Performance and Coverage Analyzer or Boole & Babbage's TSA/PPE) help the developer examine the run-time behavior of software to locate inefficiencies and bottlenecks. They collect data and statistics during the execution of a program and can generate histograms, tables, and call-trees from the data. Performance analyzers can also help locate portions of the software that have not been executed during testing.

compilation systems

Compilation systems (e.g., Alsys' UNIX Ada Compilation System or the VAX DEC/Module Management System) automate and simplify the process of building complex software applications. Compilation systems access source files in the program library and follow the sequence of dependencies among the files to automatically build the system from current versions. This allows a developer or project librarian to rebuild a system using only components that were changed since the previous system build.



software development environment

In the FDF, a tailored *software development environment* called SDE gives developers access to a variety of tools and utilities. SDE (Reference 27) integrates editors, compilers, and file allocators under a single, menu-driven framework. It is a customization of IBM's Interactive System Productivity Facility (ISPF) that provides additional tools for the FDF environment. The basic ISPF capabilities include a screen-oriented editor; utilities for file allocation, copy, display, and code comparison; and both foreground and background processing functions. Customization has added such features as a file translation utility, a system tape generator, specialized print utilities, and transfer functions for software moving between the STL and FDF environments. SDE also provides access to the PANVALET text management system, to the PANEXEC library management system, to Configuration Analysis Tool (CAT), to the RSL, and to source code analyzers.

MEASURES

Many of the same measures used during detailed design continue to be collected and analyzed during the implementation phase. In addition, source code generation and the use of configured libraries provide the manager with new yardsticks of system growth and change.

Objective Measures

The following measures are collected during the implementation phase:

- The number of units coded/read/tested versus the number identified
- Requirements questions and answers, TBDs, and changes
- Estimates of system size, effort, schedule, and reuse

- Staff hours
- Total CPU hours used to date
- Source code growth
- Errors and changes by category

Table 7-1 lists each measure, the frequency with which the data are collected and evaluated, and the sources from which the data are obtained.

Table 7-1. Objective Measures Collected During the Implementation Phase

MEASURE	SOURCE	FREQUENCY (COLLECT/ANALYZE)	DATA COLLECTION	
			CONTINUED	BEGUN
Staff hours (total and by activity)	Developers and managers (via PRFs)	Weekly/monthly	*	
Changes (by category)	Developers (via CRFs)	By event/monthly		*
Changes (to source files)	Automated tool	Weekly/monthly		*
Computer use (CPU hours and runs)	Automated tool	Weekly/biweekly	*	
Errors (by category)	Developers (via CRFs)	By event/monthly		*
Requirements (changes and additions to baseline)	Managers (via DSFs)	Biweekly/biweekly	*	
Requirements (TBD specifications)	Managers (via DSFs)	Biweekly/biweekly	*	
Requirements (questions/answers)	Managers (via DSFs)	Biweekly/biweekly	*	
Estimates of total SLOC (new, modified, and reused), total units, total effort, and schedule	Managers (via PEFs)	Monthly/monthly	*	
SLOC in controlled libraries (cumulative)	Automated tool	Weekly/monthly		*
Status (units identified/ coded/code-certified/ test-certified)	Managers (via DSFs) (The number of completed units is also reported by developers via COFs and by automated tools)	Weekly/biweekly	(Status data differ from those collected during design phases)	*

Evaluation Criteria

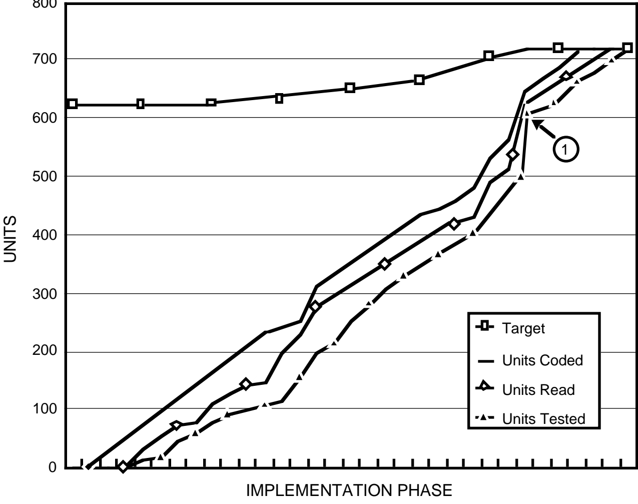
development status

The number of units coded, code-certified, and unit-test-certified, versus the total number of units to be implemented, are the measures of development status collected during the phase. By tracking each of these measures on a single graph, SEL managers can see whether all activities are progressing smoothly and in parallel. Sudden increases or convergences, such as those shown in Figure 7-6, should raise a red flag. When the development team is under pressure to meet schedules, code reading and unit testing can become hurried and superficial. If time is not taken to verify each unit properly, the effort needed to complete system testing will be increased substantially.

In the SEL, the growth in the number of units in the project's configured library is also tracked against the number of COFs. This helps managers ensure that configuration management procedures are being followed and that the data needed to track the origin and types of system components are being collected.

requirements TBDs and changes

Requirements TBDs and changes continue to be tracked during the implementation phase. Because designing a system based on best guesses can lead to extensive rework, a system should not pass CDR with requirements missing. However, if major changes or additions to requirements are unavoidable, the design of that portion of the system should be postponed and presented in a BDR at a later date. One corrective measure for late specifications is to split the development effort into two releases, with the late specifications included in the second release.



Analysis: For most of the implementation phase, code reading and unit testing activities followed unit coding at a steady rate. However, near the end of the phase, nearly three times the normal number of units were completed in a single week (1). This "miracle finish" was due to short cuts in code reading and unit testing that were taken in an effort to meet schedules.

Result: Project entered the system testing phase with poor quality software. To bring the software up to standard, the system test phase took 100% longer than expected.

Figure 7-6. Development Profile Example

estimates

As implementation progresses, managers can obtain more accurate estimates of the total number of units and lines of code in the system. They can use this data to determine whether enough effort has been allocated to complete development.

NOTE

Section 6 of *The Manager's Handbook for Software Development* (Reference 12) contains additional information on the procedures for reestimating system size, cost, and schedule during the implementation phase.

Managers can compute productivity rates to further refine project estimates and to compare the pace of implementation with that of previous projects. Factors that should be considered when measuring productivity include the number of lines of source code in configured libraries, the number of units in the libraries, and the number of completed pages of documentation per staff hour.

staff hours

Staff hours are tracked throughout the phase. If more effort is being required to complete a build than was planned, it is likely that the remaining builds (and phases) will require proportionally more effort as well. After investigating why the deviation has occurred, the manager can decide whether to increase staffing or schedule and can replan accordingly.

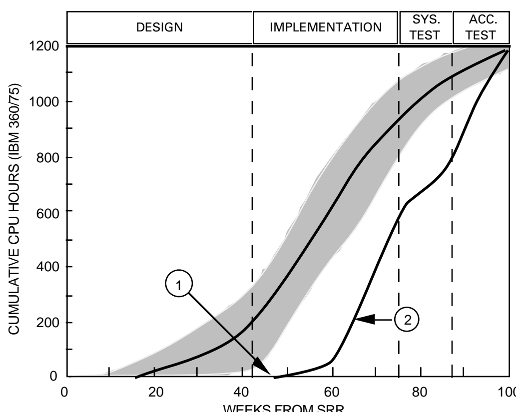
CPU usage

The profile of computer usage on any project is heavily dependent on both the development process and environment. The manager must use models of CPU usage from previous, similar projects for comparison. In the flight dynamics environment, projects that are developing AGSSs show a steep upward trend in CPU usage early in the implementation phase. This trend continues during system testing, but declines in acceptance testing, when testers conduct extensive off-line analysis of numerical results.

CPU hours that differ substantially from the local model can be caused by insufficient testing or by requirements changes that necessitate redesign (Figure 7-7).

source code growth

The amount of source code in the project's configured library is a key measure of progress during the implementation phase. As with CPU usage, the pattern of growth is heavily dependent on the development process. On projects with multiple builds, periods of sharp growth in configured SLOC will often be separated by periods of more moderate growth, when the development team is engaged in testing the build.



Symptom: Computer usage zero midway through imple-mentation (1).

Cause: Redesign in response to excessive requirements changes instead of imple-mentation.

Corrective Action: Replan project based on new scope of work (2).

Note: The local model is shown in gray.

Figure 7-7. Example of CPU Usage — ERBS AGSS

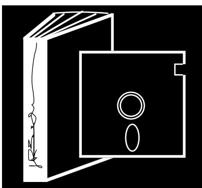
change and error rates

Managers begin to track change and error data as soon as there are units in the configured libraries. These data are usually graphed as the cumulative number of changes or errors per thousand SLOC over time.

Developers complete a CRF for each logical change made to the software, recording which units were revised as a result of the change, the type of change, whether the change was due to error, and the effort required. This information is compared with change data generated from the configuration management system (e.g., CMS) to ensure that the data are consistently reported.

The rate of software change is a key indicator of project stability. Comparative models for change rates should be based on historical data from earlier, similar projects. The SEL model, for example, reflects a steady, even growth of software changes from the middle of the implementation phase through the middle of acceptance testing. Exaggerated flat spots in the graph or sudden jumps in the change rate should always spur investigation. Excessively high rates can result from requirements changes, inadequate design specifications, or insufficient unit testing.

Error rates are generally at their highest level during the implementation phase. Error rates in the system test phase should be significantly lower, and should show a further decline during acceptance testing. The SEL has found that error rates are reduced by approximately half in each phase after implementation, and that this decline is independent of the actual values involved. Higher error rates than expected usually mean that the quality of the software has suffered from inadequate effort at an earlier stage, although such rates may also be found when the development project is exceptionally complex.



PRODUCTS

The key products of the implementation phase are

- System code and supporting data
- A set of build test plans and results
- The system test plan
- The analytical test plan

These products are discussed in the paragraphs that follow. In addition, the development team generates a draft of the user's guide, while the acceptance test team produces an updated version of the acceptance test plan. Since both of these documents are finalized during the next phase, they are described in detail in Section8.

System Code, Supporting Data, and System Files

By the end of the last build, the project's configured libraries will contain all the source code for the completed system (or release), the control and command procedures needed to build and execute the system, and all supporting data and system files.

Included in the supporting files are all input data sets needed for testing. Appropriate test data are obtained or generated for each set of build tests. By the end of the implementation phase, a full suite of input data sets must be ready for use in system testing. If testing is to be effective, these test data must be realistic. Test data are created using a simulator or data generation tool or by manual data entry. Input data for complex calculations are provided to the development team with the analytical test plan (see below).

Build Test Plans

The use of a formal test plan allows build testing to proceed in a logically organized manner and facilitates agreement among managers and developers as to when the testing is satisfactorily completed. The development team prepares the test plan for the first build before the end of the detailed design phase. The test plan for each subsequent build is prepared before the beginning of implementation for the build, and highlights of the plan are presented at the BDR.

RULE

Effective testing depends on the timely availability of appropriate test data. The software management team must ensure that the activity of test data generation is begun well in advance of testing so that neither schedules nor testing quality are compromised because of deficient data.

Build test plans follow the general outline shown in Figure 7-8. Build test plans should always identify a set of build regression tests — key tests that can be rerun to ensure that capabilities previously provided remain intact after corrections have been made or a new build has been delivered.

System Test Plan

The *system test plan* is the formal, detailed specification of the procedures to be followed while testing the end-to-end functionality of the completed system. This test plan follows the same general pattern as the build and acceptance test plans (Figure 7-8). It is reviewed by the system test team and the management team prior to the STRR.

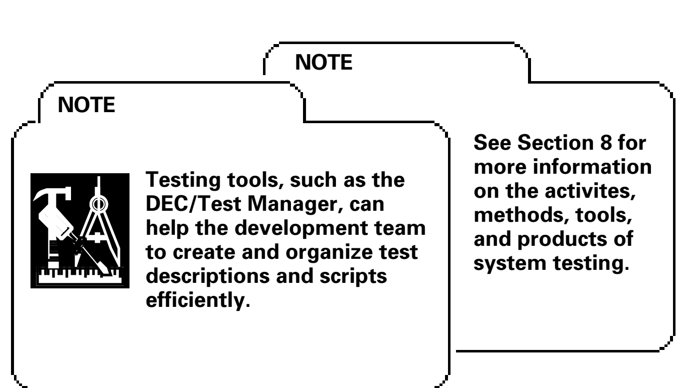
NOTE

In the Cleanroom methodology, tests are statistically selected from a hierarchy of possible user operations. Build tests are scaled-back versions of system tests, with input restrictions. Because test cases are based on the expected use of the system, continuous feedback on the reliability of the software is obtained with each build test.

The system test plan must be designed to verify that the software complies with all requirements and specifications. It should concentrate on the user's view of the system and should probe for any weaknesses that might not have been uncovered during build testing.

The testing prescribed by the plan should be functional rather than structural. In functional testing, the system is treated like a black box. Input is supplied and the output of the system is observed. The system test plan identifies the functional capabilities specified in the requirements and specifications and prescribes a set of input values that exercise those functions. These inputs must include boundary values and error conditions as well as the main processing stream.

System tests should cover multiple operational scenarios, not merely the nominal case (e.g., when the spacecraft is at a particular attitude and orbit position). The test plan must include tests designed to ensure that interfaces among subsystems are thoroughly demonstrated, as well as tests that challenge the robustness of the system by examining its performance under load and its response to user or data errors.



TEST PLAN OUTLINE

The recommended outline for build, system, and acceptance test plans is given below. Interdependencies among tests should be minimized to allow testing to proceed while failures are analyzed and corrected.

1. **Introduction**
   a. Brief overview of the system
   b. Document purpose and scope

2. **Test Procedures**
   a. Test objectives — purpose, type, and level of testing
   b. Testing guidelines — test activity assignments (i.e., who builds the executables and who conducts the tests), test procedures, checklists/report forms to be used, and configuration management procedures
   c. Evaluation criteria — guidelines to be used in determining the success or failure of a test (e.g., completion without system errors, meets performance requirements, and produces expected results) and the scoring system to be followed
   d. Error correction and retesting procedures, including discrepancy report forms to be completed (see Section 8)

3. **Test Summary**
   a. Environmental prerequisites — external data sets and computer resources required
   b. Table summarizing the system or build tests to be performed
   c. Requirements trace — matrix mapping the requirements and functional specifications to one or more test items

4. **Test Descriptions** (Items a to f are repeated for each test)
   a. Test name
   b. Purpose of the test — summary of the capabilities to be verified
   c. Method — step-by-step procedures for conducting the test
   d. Test input
   e. Expected results — description of the expected outcome
   f. Actual results *(added during the testing phase)* — description of the observed results in comparison to the expected results

5. **Regression Test Descriptions** (Repeat items 4a to 4f for each regression test)

Figure 7-8. Generalized Test Plan Format and Contents

System test plans must specify the results that are expected from each test. Plans should refer to specific sections of the requirements and specifications if these documents already contain expected results. Where possible, each test should be defined to minimize its dependence on preceding tests, so that the testing process can adapt to inevitable, unplanned contingencies.

regression tests

The system test plan must also define the set of *regression tests* that will be used to ensure that changes made to software have had no unintended side effects. The regression test set should be short enough to be actually used when needed, yet should exercise a

maximum number of critical functions. As a rule of thumb, select the key 10 percent of the total number of tests as the regression test set.

> **NOTE**
>
> **See METHODS AND TOOLS in Section 8 for additional information on regression testing.**

Analytical Test Plan

In the flight dynamics environment, an additional *analytical test plan* is generated during the implementation phase to assist testers in verifying the results of complex mathematical and astronomical algorithms.

The analytical test plan is produced by members of the acceptance test team and is provided to the development team in two phases. Test procedures for verifying computations that are performed at the unit level are delivered before the start of the build containing the relevant units. The second portion of the analytical test plan, which contains tests of end-to-end functionality, is provided to developers before the start of system testing and is executed along with the system test plan.

> **NOTE**
>
> **If a complete analytical test plan is available early in the implementation phase, the system test plan can be written to incorporate the analytical tests. Otherwise, the analytical test plan is conducted in parallel with the system test plan. In the latter case, the test team must work to efficiently coordinate both sets of tests, minimizing the effort spent in setup, execution, and analysis.**

Analytical test plans are only useful to the development team if input data and output results are explicitly specified. Ideally, test data sets containing analytically robust, simulated data should be supplied to the development team with the plan. The test plan must itemize the expected, numerical input and output for each test as well as any intermediate results that are needed to verify the accuracy of calculations.

BUILD DESIGN REVIEWS

Reviews are recommended for each build. At BDRs, the development team and its managers cover important points of information that need to be transmitted before the next phase of implementation. Such information includes any changes to the system design, the contents of the build, and the build schedule.

The focus of a BDR is on status and planning. Strategies for handling TBDs, risk-management plans, and lessons learned from previous builds should also be covered.

BDR FORMAT

Presenters — software development team

Participants
- Requirements definition team
- Acceptance test team representatives
- Quality assurance representatives
- Customer interfaces
- User representatives
- System capacity/performance analysts

Attendees must be familiar with the project background, requirements, and design.

Schedule — before implementation of each build of a system or release, except the first

Agenda — presentation of changes to the detailed design of the system and to the build plan, emphasizing lessons learned in previous builds and risk mitigation strategies

Materials — distributed a minimum of 3 days before the review.

Figure 7-9. BDR Format

The formality of the review depends on the size and complexity of the project. Large projects may find that a slide presentation and hardcopy handouts are necessary. On smaller projects, developers may simply meet with analysts, customers, and users around a conference table.

A synopsis of the BDR format is shown in Figure 7-9, and a suggested outline for the contents of BDR materials is provided in Figure 7-10. If a formal presentation is made, the materials distributed at the review should be hardcopies of the presentation viewgraphs or slides.

EXIT CRITERIA

At the end of the final build, the software development manager should answer the following questions:

- Have all components of the system passed each stage of verification, inspection, and certification? Are all components organized into configured libraries?

- Have all build test plans been completed? Have all critical discrepancies been resolved successfully?

- Has the system test plan been completed and reviewed? Are data files and procedures in place for system testing?

- Are documentation products complete? That is, have all SENs been checked and systematically filed in the project library? Is the draft of the user's guide ready for evaluation by the system test team?

When the manager can comfortably answer "yes" to each question, the implementation phase is complete.

MATERIALS FOR THE BDR

1. **Agenda** — outline of review material

2. **Design changes** since CDR or the previous BDR (with justifications)
   a. Revised design diagrams
   b. Changes to system operation; updated operations scenarios/scripts, displays, reports, and screens

3. **Build contents**
   a. Requirements to be met in the build
   b. Units and data objects to be included in the build
   c. Unresolved problems in earlier builds to be resolved in this build

4. **Testing strategy** — sequence of build tests, test data, drivers/simulators, etc.

5. **Changes to remaining builds and releases**
   a. Changes in the distribution of requirements among remaining builds/releases
   b. Changes in the distribution of software units and data objects

6. **Updated software size estimates**

7. **Milestones and schedules**
   a. Schedule for implementing and testing the current build
   b. Schedule for the remaining builds/releases

8. **Issues, risks, problems, TBD items**
   a. Review of any remaining TBDs
   b. Risks associated with the build

Figure 7-10. BDR Materials

LIFE CYCLE PHASES	REQUIREMENTS DEFINITION	REQUIRE-MENTS ANALYSIS	PRELIMI-NARY DESIGN	DETAILED DESIGN	IMPLEMENTATION	SYSTEM TESTING	ACCEPTANCE TESTING

SECTION 8

THE SYSTEM TESTING PHASE

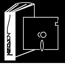
PHASE HIGHLIGHTS

ENTRY CRITERIA

- All system code and supporting data generated and tested
- Build test plans successfully executed
- System test plan completed
- User's guide drafted

EXIT CRITERIA

- System and analytical test plans successfully executed*
- Acceptance test plan finalized
- User's guide and system description completed
- Configuration audits and ATRR conducted



PRODUCTS

- Tested system code and supporting files
- System test results
- User's guide
- System description document
- Acceptance test plan

MEASURES

- System tests planned/executed/passed
- Discrepancies reported/resolved
- Staff hours
- CPU hours
- SLOC in controlled libraries (cumulative)
- Changes and errors (by category)
- Requirements Q&As, TBDs, and changes
- Estimates of system size, effort, schedule, and reuse

METHODS AND TOOLS

- System test plan
- Regression testing
- Configuration management
- Configuration audits
- Test tools
- Test logs
- Discrepancy reports
- IV&V

KEY ACTIVITIES

System Test Team
- Prepare for system testing*
- Execute all items in the system and analytical test plans
- Analyze and report test results
- Control the testing configuration
- Evaluate the user's guide
- Conduct an ATRR

Development Team
- Correct discrepancies found during testing
- Tune system performance
- Complete system documentation
- Identify candidates for the RSL
- Prepare for acceptance testing

Management Team
- Reassess schedules, staffing, and resources
- Ensure the quality and progress of testing
- Control requirements changes
- Conduct configuration audits
- Coordinate the transition to acceptance testing

Acceptance Test Team
- Finalize the acceptance test plan
- Prepare for acceptance testing

* In the Cleanroom methodology, there is no separate system testing phase. All testing is conducted by an independent test team in parallel with software development activities. Test cases are generated statistically based on operational scenarios.

OVERVIEW

The purpose of the system testing phase is to verify the end-to-end functionality of the system in satisfying all requirements and specifications.

During this phase, the system test team executes the tests specified in the system and analytical test plans. The results obtained during test execution are documented, and the development team is notified of any discrepancies. Repairs to software are handled by members of the development team in accordance with stringent configuration management procedures. Corrected software is retested by the system test team, and regression tests are executed to ensure that previously tested functions have not been adversely affected. (See Figure 8-1.)

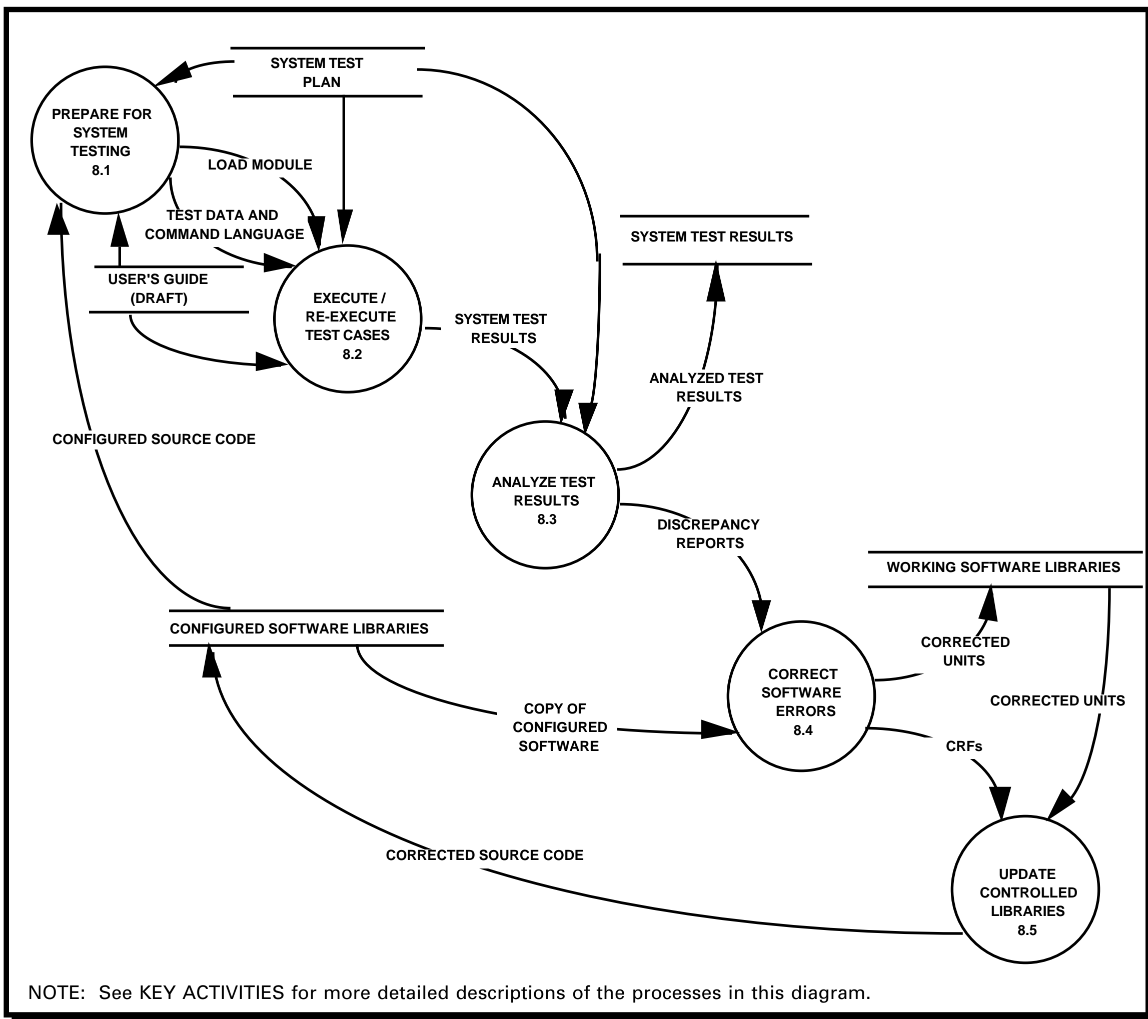


Figure 8-1. System Testing

During the system testing phase, the software development team prepares the documentation for the completed system. The user's guide is evaluated and an updated version is published by the end of the phase. The development team also records the final design of the system in a system description document.

System testing is complete when all tests in both the system test and analytical test plans have either executed successfully or have been waived at the authorization of the customer. Near the end of the phase, the system and its documentation are audited for completeness. The system is then demonstrated to the acceptance test team and an *acceptance test readiness review (ATRR)* is held to determine whether the system test phase can be concluded and the acceptance test phase begun.

KEY ACTIVITIES

System tests are planned and executed by a subset of the development team that includes the application specialists. In the flight dynamics environment, one or more analysts are added to the system test team to ensure the mathematical and physical validity of the test results. They also learn to operate the software in preparation for acceptance testing.

The senior application specialist is usually designated as the leader of the system test team. He or she is responsible for ensuring that test resources are scheduled and coordinated, that the appropriate test environment is established, and that the other members of the team understand the test tools and procedures. During testing, this leader directs the actions of the team, ensures that the test plan is followed, responds to unplanned events, and devises workarounds for failures that threaten the progress of testing.

NOTE

When reliability requirements are particularly stringent, system testing may be conducted by an independent test team. See METHODS AND TOOLS for more information on independent verification and validation (IV&V) procedures.

The key activities of the system test team, the development team, the management team, and the acceptance test team are summarized in the paragraphs that follow. A suggested timeline for the performance of these activities is given in Figure 8-2.

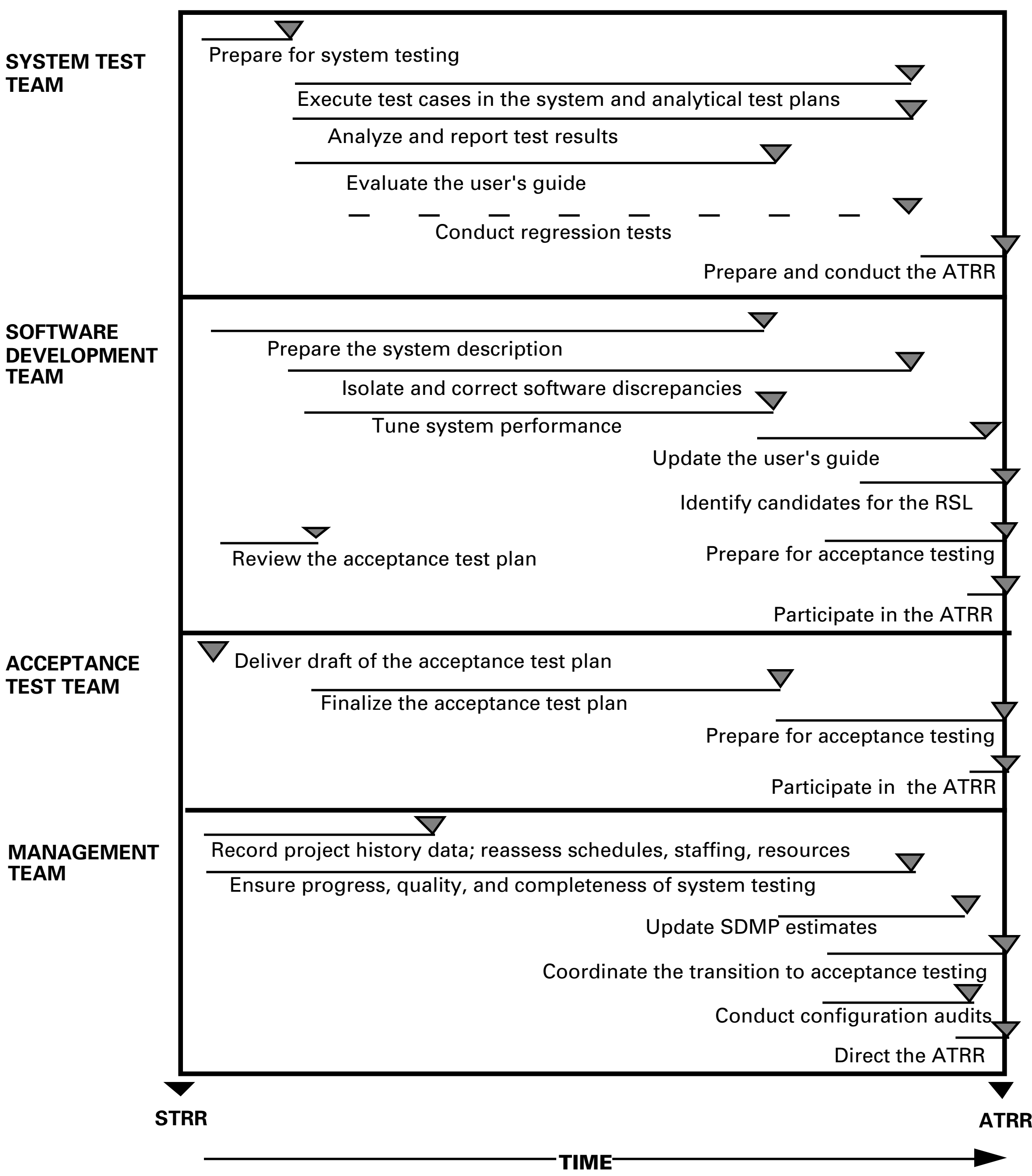


Figure 8-2. Timeline of Key Activities in the System Testing Phase

Activities of the System Test Team

- **Prepare for system testing.** Establish the system test environment. System testing generally takes place on the development computer rather than the eventual operational computer; however, it may be necessary to rehost the system to the target computer if there are critical performance requirements. Ensure that any test tools that will be used are available and operational.

  Ensure that computer resources and operations personnel are scheduled and available. Resolve resource conflicts early; they can seriously affect schedules, particularly when real-time testing is involved.

  Effective testing depends on the timely availability of appropriate test data. Collect and use real data for testing whenever possible. When real data cannot be obtained, generate and use simulated data. Before testing is begun, ensure that all test data and command and parameter files (e.g., JCL and NAMELISTs) are physically reasonable, reliable, and analytically robust. Maintain test data and control files under configuration management.

- **Execute all test items in the system and analytical test plans,** running the regression test set each time a replacement load module is installed. Follow the procedures prescribed in the test plan. Keep printed output for each test execution and maintain a test log so that events can be accurately reconstructed during post-test analysis. Document any discrepancies between expected and actual test results.

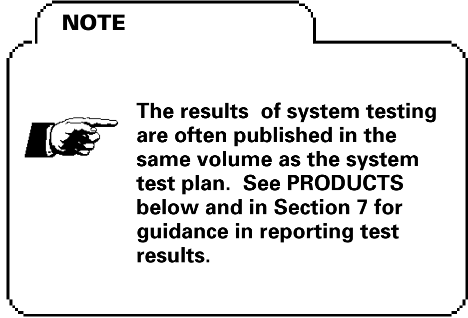
  The amount of diagnostic output generated during testing can either help or hamper analysis. Tailor diagnostics and debug output to the test and analysis approach so that errors can be isolated expeditiously.

RULE

A developer should not be asked to system test his or her own code. Developers on the system test team should test sections of the system implemented by members other than themselves.

- **Analyze and report test results.** Test results must be analyzed and interpreted to determine if they correspond to those that were expected. Where possible, use automated tools in the analysis process. Keep good records of analysis procedures and retain all relevant materials.

Determine whether discrepancies are caused by software or by incorrect operations. Rerun any tests that failed because of errors in data, setup, or procedures as soon as these have been corrected. When software is implicated, report the problem using discrepancy report forms. Discrepancy reports are reviewed by the management team and assigned to members of the development team for correction.



When all system tests are complete, compile a final report documenting the results of testing in both detailed and summary form. Address any problems encountered and their solutions.

- **Control the testing configuration.** System testing must yield reproducible results. Moreover, when test results do not match those that are expected, it must be possible to vary the input parameters to find those to which the unexpected results are sensitive. During this process, the exact system configuration must be kept constant.

  To reduce the possibility of configuration errors, only the project librarian should change configured source libraries and build the executable image that will be used during system testing.

- **Evaluate the user's guide.** Employ the user's guide during test preparation and execution. Annotate a copy of the guide, noting any errors or discrepancies between the information it contains and actual operational conditions. Suggest ways in which the clarity and usability of the guide could be improved.

- **Conduct an ATRR.** Meet with management, customers, and members of the acceptance test and development teams to assess the results of system testing and the state of preparations for acceptance testing. Identify and discuss any outstanding problems that may impose technical limits on the testing or may affect schedules. The system test and development teams must be certain that the system is complete and reliable before it is sent to the acceptance test team.

RULE

When completing a CRF form, be sure to correctly note the original source of an error. Changes to code may be caused by incorrect requirements and specifications or by design errors. Such changes should not be labelled as code errors, even though code was revised.

NOTE

The contents of the user's guide, system description, and acceptance test plan are discussed under the PRODUCTS heading in this section.

Activities of the Development Team

- **Correct discrepancies found during testing.** Assist the test team in isolating the source of discrepancies between expected and actual test results. If the error is in the software design, thoroughly analyze the ramifications of any design changes. Update the affected design diagrams and structure charts before proceeding with corrections to code.

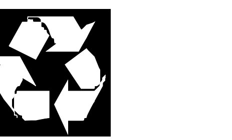
  Verify all corrections using code reading, unit testing, and integration testing. Update the SENs for the revised units and fill out a CRF describing the modifications.

- **Tune the performance of the system.** Analyze the performance of the system, using tools such as the VAX Performance and Coverage Analyzer or TSA/PPE (see *Methods and Tools*, Section 7). Locate and correct any bottlenecks found.

- **Complete the system documentation.** Update the user's guide, correcting any errors that were found by the system test team and incorporating the team's recommendations for improving the document's quality. Ensure that the user's guide reflects any modifications to software or operational procedures that are made as a result of system testing. Deliver the updated version of the guide to the acceptance test team at the ATRR.

  Complete the system description. Update the draft begun during the implementation phase so that the document reflects the final state of the system. By the end of the system testing phase, deliver the completed draft to the acceptance test team.

- **Identify candidates for the RSL.** If reuse has been a determinant in the system design, decide which of the final software components are sufficiently generalized and robust to be candidates for the RSL. Also identify any nongeneric components that could be reused in similar systems. Document this analysis and forward the candidates for inclusion in the RSL.



- **Prepare for acceptance testing.** Review the draft of the acceptance test plan. Ensure that the plan tests only what is in the requirements and specifications document. Any additional performance tests or tests that require intermediate results not specified in the requirements must be negotiated and approved by the management team.

  Work with acceptance testers to obtain the computer resources needed for acceptance testing and to prepare the necessary input data sets and command procedures (JCL/DCL). If the system will be operated in an environment that is different from the development environment, rehost the system to the target computer. Demonstrate the system to the acceptance test team and participate in the ATRR.

Activities of the Management Team

- **Reassess schedules, staffing, and resources**. At the beginning of system testing, record measures and lessons learned from the implementation phase and add this information to the draft of the software development history. Use measures of effort and schedule expended to date to reestimate costs and staffing levels for the remainder of the project.

- **Ensure the quality and progress of system testing.** Coordinate the activities of the system test and development teams, and ensure that team members adhere to procedures and standards. Place special emphasis on enforcing configuration management procedures; the control of software libraries is critical during the system and acceptance testing phases.

  On a weekly basis, analyze summaries of testing progress and examine plots of test discrepancies. Ensure that the development team corrects software errors promptly so that testing does not lose momentum. Assist developers in resolving technical problems when necessary.

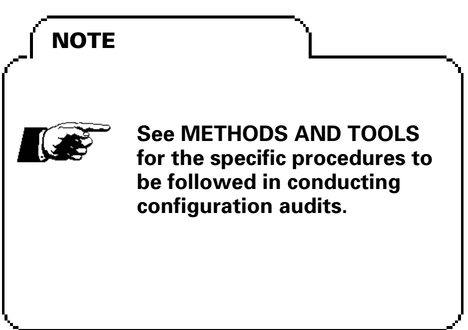
  Ensure that error corrections are thoroughly tested by the development team before revised software is promoted to controlled libraries for regression testing. Review all test results and system documentation.

NOTE

The key to success in system testing is the system test plan. A complete and detailed system test plan results in a precise understanding of the functionality that the tester needs to verify, and provides an easy tracking mechanism for monitoring weekly testing status.

- **Control requirements changes.** Although requirements changes are not frequent this late in the life cycle, when they do occur, the same formal recording mechanisms (i.e., requirements question-and-answer forms and specification modifications) are used as in preceding life cycle phases.

  Challenge any specification modifications that are received after the beginning of system testing. In general, new specification modifications that add requirements or enhance the system should not be accepted unless they are critical for mission support. Moreover, unless mission-critical changes can be incorporated with little or no impact to budget and schedule, they should be scheduled into a later release or implemented during maintenance and operations.



- **Conduct configuration audits.** When system testing is complete, select one or more quality assurance personnel or developers to audit the system configuration. Determine if test records demonstrate that the system meets all requirements and functional specifications, and verify that the system documentation completely and accurately describes the actual software in controlled libraries. Develop action items for the solution of any problems found.

- **Coordinate the transition to the acceptance testing phase and direct the ATRR.** Designate the developers and application specialists who will provide support to the acceptance test team during the next phase. They will be responsible for setting up and running tests with members of the acceptance test team present. Supervise demonstrations of the system for the benefit of the acceptance test team.

  Schedule the ATRR. Ensure that appropriate representatives attend and that the agenda covers any and all issues that could affect the success of acceptance testing. Make certain that all members of both the acceptance test and development teams understand and approve the procedures that will be followed during testing. Assign action items resulting from the meeting and oversee their resolution.

Activities of the Acceptance Test Team

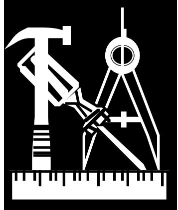
- **Finalize the acceptance test plan**. At the start of the system test phase, provide the development team with a draft of the acceptance test plan. Update and complete the plan during the remainder of the phase.

- **Prepare for acceptance testing.** Prepare all test data, control language (JCL/DCL), and parameter files needed for testing. Generate or request any simulated data that will be needed. Verify test data for completeness and accuracy, and place them under configuration management.

  Schedule the resources needed for testing, including personnel, terminals, and disk space. Optimize resource usage and avoid conflicts; careful scheduling is crucial to testing success.

  Attend the system demonstrations conducted by the development team. Practice running the system with developer support. Participate in the ATRR.

METHODS AND TOOLS



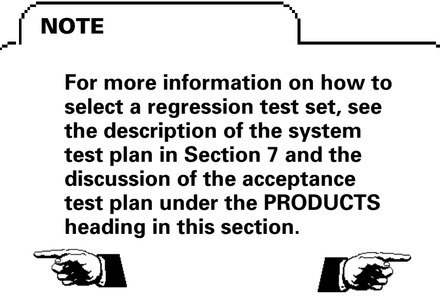
The key methods and tools of the system testing phase are

- The system test plan
- Regression testing
- Configuration management
- Configuration audits
- Test tools
- Test logs
- Discrepancy reports
- IV&V

The system test plan, which is the primary "tool" used during the phase, is a product of the implementation phase and was discussed in Section 7. The other methods and tools in this list are elaborated in the paragraphs that follow.

Regression Testing

Regression testing is the testing that must be performed after functional improvements or corrections have been made to the system to confirm that the changes have created no unintended side effects. Regression testing is an essential element of build and acceptance testing as well as of system testing. In the



implementation phase, build regression tests are run to ensure that a new build has not impaired the functioning of previous builds. During system and acceptance testing, regression tests are conducted by the test team each time the load module is replaced. These regression tests are a selected subset of the full set of system or acceptance tests, and are specified in the test plan.

Regression tests also help assure that configuration management procedures are followed. If regression tests reveal that an outdated or untested unit or module was included in the test configuration, the management team should immediately investigate to determine which configuration management procedures (described below) were violated.

Configuration Management

During the system test phase, strict adherence to configuration management procedures is essential. Because the software is under configuration control at this time, any changes to the code in the permanent source code libraries must be made according to established procedures and recorded on CRFs. Configuration management procedures must ensure that the load modules being tested correspond to the code in the project's libraries.

During system testing, configuration management problems can be avoided by following certain basic principles:

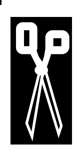
- Limit update access to controlled libraries and restrict rebuilding of test configurations to a single configuration manager (usually the project librarian). Whenever possible, use automated tools, such as CMS control lists, to enforce this restricted access.

- Periodically rebuild the test configuration from the controlled source code, eliminating lower-level working libraries, so that the system test team can start from an updated, configured system on each round of testing. If possible, use a two-level library structure. The top level contains the official system test executable image built from source code in the controlled library; all system testing is performed from this library. The lower-level library is used by developers for testing changes when system tests have failed. When developers' changes are

promoted into the controlled source code, the top-level library is rebuilt and the lower-level library is emptied. Failed tests are then rerun by the test team from the top level.

- Restrict the frequency with which new load modules/executable images are constructed to minimize the amount of regression testing that must be conducted. A new load module can be built whenever a predetermined number of changes to the controlled source have been made or can be scheduled on a regular basis, e.g., every two weeks.

TAILORING NOTE



On Ada projects, management of a third library, the Ada compilation library, is critical to keeping a controlled system configuration. When updated source code is promoted to controlled libraries, the compilation library must be rebuilt before the new executable image is constructed.

SENs must also be updated to reflect changes made to source code during this phase. The developer obtains the SENs from the project librarian before correcting the software and returns them, along with the CRF(s), when changes are completed. The project librarian checks each returned SEN to ensure that the required checklists and listings have been included.

Configuration Audits

After system testing is complete, configuration audits are performed to confirm that the system meets all requirements and specifications, that it is accurately described in the documentation, and that it does not include any unauthorized changes.

FCA

In the *functional configuration audit (FCA)*, selected staff members spot-check the system test results against the expected results in the test plan(s) to determine if the tests were performed properly and completely. They also examine all waivers, verifying that any uncorrected discrepancies have been reviewed and approved by the customer.

When the FCA is complete, the auditors provide a list of the items they examined and their findings to the management team, along with their assessment of the readiness of the system for acceptance testing.

PCA

In the *physical configuration audit (PCA)*, auditors compare the design of the system (as documented in the system description and SENs) against listings of the software in configured libraries to verify that these descriptions match the actual, tested system. The auditors also check the user's guide against the system description to

ensure the two documents are consistent. In a report to the management team, they summarize their activities, list the items audited, itemize any conflicts found, and recommend action items.

RULE

The staff members selected to conduct the FCA should not have implemented or tested the system being audited. Developers, QA, or CM personnel may conduct the PCA.

Test Tools

Many of the automated tools used to facilitate the implementation of the system continue to be employed during the system testing phase. These tools — configuration management systems, symbolic debuggers, static code analyzers, performance analyzers, compilation systems, and code-comparison utilities — are discussed in Section 7.

In addition, test management tools (e.g., DEC/Test Manager) can provide an online mechanism for setting up the test environment, for testing interactive applications in batch mode, and for regression testing. Use of such a tool allows testers to examine test result files interactively and to generate summary reports of test runs automatically.

NOTE

The system test plan must describe the specific procedures to be followed in recording and evaluating system tests and while correcting test discrepancies. The contents of the system test plan are discussed under PRODUCTS in Section 7.

Test Logs

The test log is an ordered collection of the individual report forms or checklists that are completed by testers during actual test execution.

Use of a standard test report form or checklist is essential to the accurate recording of test results. Each report form should identify the test, the tester, the load module/executable image being tested, and the date and time of testing. The form should provide space to summarize the test case and to record whether the test results matched those expected. If a discrepancy is found, it is noted on the form and described in detail in a discrepancy report (see below).

Discrepancy Reports

Testers fill out a discrepancy report form for any errors that they uncover that could not be immediately traced to an incorrect test setup or operational mistake. Discrepancy reports are also known as *software problem reports (SPRs), software trouble reports (STRs)*, or *software failure reports (SFRs).* A sample SFR form is shown in Figure 8-3.

SOFTWARE FAILURE REPORT

Originator:____________________Location:___________________Phone:____________________
SFR#:_________________________Date:_______________________Time:_____________________
Test ID:___________________________Subsystem:_______________________________________
Load Module:____
Source of Failure:__________________________________Failure Level: ____________________
Summary (40 characters): ____

==

Problem Description: ____

Impact, Analysis, and Suggested Resolution:____

(SFR Originator — Do Not Write Below This Line)

==

Disposition (Circle One): Accept Reject Date Resolution Is Required:___________________
Assigned to: ___________________________________Date Completed:______________________
Notes:____

__Figure 8-3. Sample Software Failure Report Form__

Discrepancy reports are reviewed by the leader of the system test team before being passed to the software development team for correction. The priority given to a discrepancy report depends on the severity level of the failure. One grading system for failures that has been used in the flight dynamics environment defines three levels:

- Level 1 (highest severity): A major error occurred and no workaround exists. The test cannot be evaluated further. Abnormal termination of the program, numerical errors, or requirements that are not implemented are considered level1 discrepancies.
- Level 2: A major error occurred but a software workaround exists. Abnormal terminations that can be worked around with different user input, small errors in final results, or minor failures in implementing requirements are classed as level 2 discrepancies.
- Level 3 (lowest severity): A cosmetic error was found. An incorrect report format or an incorrect description in a display or message are classified as cosmetic errors, as are deficiencies that make the software difficult to use but that are not in violation of the requirements and specifications. If testing schedules are tight, the correction of cosmetic errors may be waived at the authorization of the customer.

The status of discrepancy reports must be monitored closely by the system test and management teams. A discrepancy report is closed at the authorization of the test team leader when the source code has been corrected and both the previously-failed test case(s) and the regression test set have been executed successfully.

IV&V

Independent verification and validation (IV&V) is recommended whenever high reliability is required in a particular mission-critical application, such as manned-flight software. In the flight dynamics environment, IV&V implies that system testing is planned and conducted by a team of experienced application specialists who are independent and distinct from the development team. The system test plan that the team develops must contain additional tests designed to determine robustness by stressing the system.

The management team must identify the need for IV&V in the SDMP. An additional 5 to 15 percent of total project costs should be budgeted to cover the additional effort required.

MEASURES

Objective Measures

During the system test phase, managers continue to collect and analyze the following data:

- Staff hours
- Total CPU hours used to date
- Source code growth
- Errors and changes by category
- Requirements questions and answers, TBDs, and changes
- Estimates of system size, effort, schedule, and reuse

They also begin to monitor measures of testing progress:

- The number of tests executed and the number of tests passed versus the number of tests planned
- The number of discrepancies reported versus the number of discrepancies resolved

Table 8-1 lists each of these measures, the frequency with which the data are collected and analyzed, and the sources from which the data are obtained. Aspects of the evaluation of these measures that are unique to the system testing phase are discussed in the paragraphs that follow.

Evaluation Criteria

test status

By tracking the number of system tests executed and passed as a function of time, the management team gains insight into the reliability of the software, the progress of testing, staffing weaknesses, and testing quality. Figure 8-4 shows a sample system test profile of a project monitored by the SEL.

test discrepancies

The management team also monitors plots of the number of discrepancies reported during system testing versus the number repaired. If the gap between the number of discrepancies identified and the number resolved does not begin to close after the early rounds of testing, the management team should investigate. One or more application specialists may need to be added to the development team to speed the correction process.

NOTE

Managers should use a tool to assist them in tracking the progress of system testing. The tool should provide standardized formats for entering and storing testing status data, and should generate plots of discrepancies found and discrepancies remaining unresolved, as a function of time.

Table 8-1. Objective Measures Collected During the System Testing Phase

MEASURE	SOURCE	FREQUENCY (COLLECT/ANALYZE)	DATA COLLECTION	
			CONTINUED	BEGUN
Staff hours (total and by activity)	Developers and managers (via PRFs)	Weekly/monthly	*	
Changes (by category)	Developers (via CRFs)	By event/monthly	*	
Changes (to source files)	Automated tool	Weekly/monthly	*	
Computer use (CPU hours and runs)	Automated tool	Weekly/biweekly	*	
Errors (by category)	Developers (via CRFs)	By event/monthly	*	
Requirements (changes and additions to baseline)	Managers (via DSFs)	Biweekly/biweekly	*	
Requirements (TBD specifications)	Managers (via DSFs)	Biweekly/biweekly	*	
Requirements (questions/answers)	Managers (via DSFs)	Biweekly/biweekly	*	
Estimates of total SLOC (new, modified, and reused), total units, total effort, and schedule	Managers (via PEFs)	Monthly/monthly	*	
SLOC in controlled libraries (cumulative)	Automated tool	Weekly/monthly	*	
Status (tests planned/executed/ passed)	Managers (via DSFs)	Weekly/biweekly		*
Test discrepancies reported/resolved	Managers (via DSFs)	Weekly/biweekly		*

Figure 8-5 shows the SEL model for discrepancy status against which current projects are compared. The model is generally applicable for any testing phase.

If the number of discrepancies per line of code is exceptionally high in comparison with previous projects, the software has not been adequately tested in earlier phases. Schedule and budget allocations should be renegotiated to ensure the quality of the product.

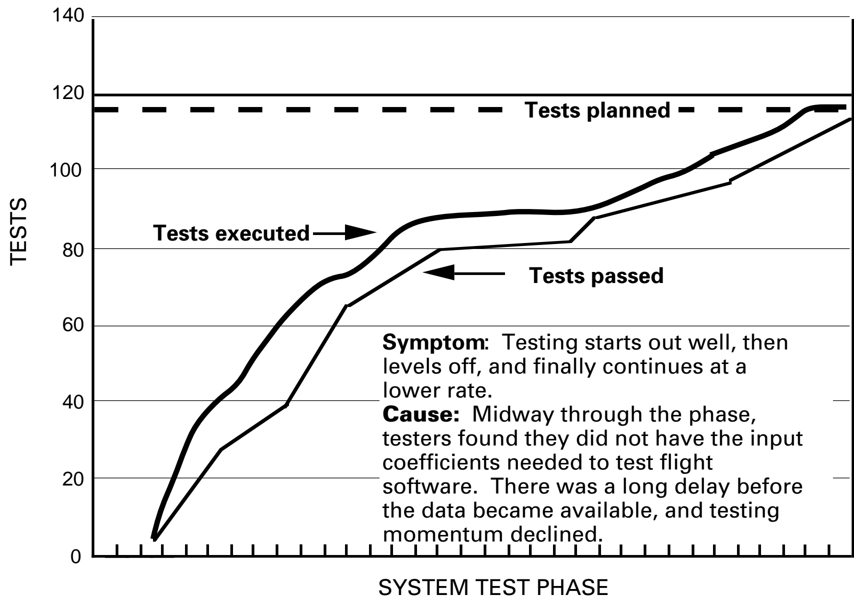


Figure 8-4. EUVEDSIM System Test Profile

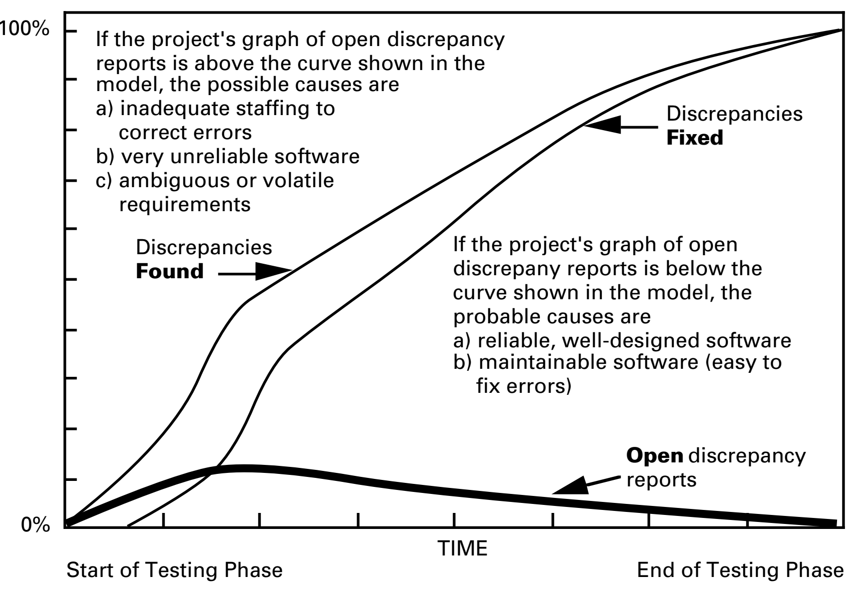
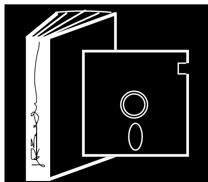


Figure 8-5. SEL Discrepancy Status Model

error rates

Error rates should be significantly lower during system testing than they were during build testing. High error rates during this phase are an indicator of system unreliability. Abnormally low error rates may actually indicate that system testing is inadequate.



PRODUCTS

The key products of the system testing phase are

- Completed, tested system code and supporting files
- System test results
- The updated user's guide
- The system description
- The acceptance test plan

System Code, Supporting Data, and System Files

By the end of system testing, the project's configured libraries contain the final load module, the source code for the system (including changes made to correct discrepancies), the control procedures needed to build and execute the system, and all supporting data and command files.

System Test Results

When system testing is complete, test results are published either as a separate report or in an update of the system test plan. The latter method lets the reader compare actual with expected results more easily and eliminates some redundancy in describing test objectives (see Figure 7-8). The individual report forms or checklists used during testing to log results are collected in a separate volume.

System test results are reviewed by the test team leader, the management team, and the CCB; they are also audited as part of the FCA. Approved test results are controlled using configuration management procedures.

User's Guide

The user's guide contains the information that will be needed by those who must set up and operate the system. A recommended outline for the user's guide is shown in Figure 8-6.

During system testing, the draft of the user's guide that was produced by the development team during the implementation phase is evaluated and updated. The system test team uses the guide during test preparation and execution and provides written comments back to developers. By the end of system testing, the development team publishes an updated, corrected version of the document that reflects the state of the system at the completion of

USER'S GUIDE

The development team begins preparation of the user's guide during the implementation phase. Items 1 and 2, and portions of item 3, represent updated material from the detailed design document, although some rewriting is expected to make it more accessible to the user. A draft is completed by the end of the implementation phase and is evaluated during system testing. At the beginning of the acceptance test phase, an updated version is supplied to the acceptance test team for evaluation. Corrections are incorporated, and a final revision is produced at the end of the phase. The suggested contents are as follows:

1. **Introduction**
   a. Overview of the system, including purpose and background
   b. Document organization
   c. Discussion and high-level diagrams of system showing hardware interfaces, external data interfaces, software architecture, and data flow

2. **Operations overview**
   a. Operations scenarios/scripts
   b. Overview and hierarchy of displays, windows, menus, reports, etc.
   c. System performance considerations

3. **Description** for each subsystem or major functional capability
   a. Overall subsystem capability
   b. Assumptions about and restrictions to processing in each mode
   c. Discussion and high-level diagrams of subsystems, including interfaces, data flow, and communications for each processing mode
   d. High-level description of input and output
   e. Detailed description of processing keyed to operator-specified input and actions in terms of points of control, functions performed, and results obtained (both normal and abnormal, i.e., error processing and recovery)
   f. For interactive subsystems, facsimiles of displays in the order in which they are generated
   g. Facsimiles of hardcopy output in the order in which it is produced, annotated to show what parameters control it
   h. List of numbered messages with explanation of system's and user's actions, annotated to show the units that issue the message

4. **Requirements for execution**
   a. Resources — discussion, high-level diagrams, and tables for system and subsystems
      (1) Hardware
      (2) Data definitions, i.e., data groupings and names
      (3) Peripheral space considerations — data storage and printout
      (4) Memory considerations — program storage, array storage, and data set buffers
      (5) Timing considerations
         (a) CPU time in terms of samples and cycles processed
         (b) I/O time in terms of data sets used and type of processing
         (c) Wall-clock time in terms of samples and cycles processed
   b. Run information — control statements for various processing modes
   c. Control parameter information — by subsystem, detailed description of all control parameters (e.g., NAMELISTs), including name, data type, length, representation, function, valid values, default value, units, and relationship to other parameters

Figure 8-6. User's Guide Contents

testing. This version is delivered to the acceptance test team at the ATRR for use during the next phase.

TAILORING NOTE

In the system description, briefly discuss the error-handling philosophy that has been incorporated into the software. For Ada systems, the discussion should summarize the approach used in raising, reporting, and recovering from exceptions.

TAILORING NOTE

Tailor the content of the user's guide to highlight the key information needed by users. Keep the writing style succinct and easily understandable.

System Description

The system description document records the final design of the system. It contains detailed explanations of the internal structure of the software and is addressed to those who will be responsible for enhancing or otherwise modifying the system in the future. Figure 8-7 gives the outline for the system description recommended by the SEL.

Acceptance Test Plan

The acceptance test plan describes the steps that will be taken during the acceptance testing phase to demonstrate to the customer that the system meets its requirements. The plan details the methods and resources that will be used in testing, and specifies the input data, procedures, and expected results for each test. In the plan, each test is mapped to the requirements and specifications to show which requirements it demonstrates. The requirements verified by a particular test are called *test items.*

test items

The acceptance test plan is prepared by members of the acceptance test team following the generalized test plan outline shown previously (Figure 7-8). To ensure consistency between the requirements documents and the acceptance test plan, members of the requirements definition team join the acceptance test team to begin working on the plan as soon as the initial requirements and specifications have been delivered. As TBD requirements are resolved and as specification modifications are approved, the draft of the acceptance test plan is updated. At the beginning of system testing, the completed draft is provided to the development team for review.

SYSTEM DESCRIPTION

During the implementation phase, the development team begins work on the system description by updating data flow/object diagrams and structure charts from the detailed design. A draft of the document is completed during the system testing phase and a final version is produced by the end of acceptance testing. The suggested contents are as follows:

1. **Introduction** — purpose and background of the project, overall system concepts, and document overview

2. **System overview**
   a. Overview of operations scenarios
   b. Design drivers (e.g., performance considerations) and their order of importance
   c. Reuse strategy
   d. Results of prototyping efforts
   e. Discussion and high-level diagrams of the selected system design, showing hardware interfaces, external data interfaces, interconnections among subsystems, and data flow
   f. Traceability matrix of major components against requirements and functional specifications
   g. Error-handling strategy

3. **Description** of each subsystem or major functional breakdown
   a. Overall subsystem capability
   b. Assumptions about and restrictions to processing in each mode
   c. Discussion and high-level diagrams of subsystem, including interfaces, data flow, and communications for each processing mode
   d. High-level description of input and output
   e. Structure charts or object-oriented diagrams expanded to the subprogram level, showing interfaces, data flow, interactive control, interactive input and output, and hardcopy output

4. **Requirements for creation**
   a. Resources — discussion, high-level diagrams, and tables for system and subsystems
      (1) Hardware
      (2) Support data sets
      (3) Peripheral space considerations — source code storage, scratch space, and printout
      (4) Memory considerations — program generation storage and data set buffers
      (5) Timing considerations
         (a) CPU time in terms of compile, build, and execute benchmark test
         (b) I/O time in terms of the steps to create the system
   b. Creation information — control statements for various steps
   c. Program structure information — control statements for overlaying or loading

5. **Detailed description of input and output** by step — source code libraries for system and subsystems, object code libraries, execution code libraries, and support libraries

6. **Internal storage requirements** — description of arrays, their size, their data capacity in all processing modes, and implied limitations of processing

7. **Data interfaces** for each internal and external interface
   a. Description, including name, function, frequency, coordinates, units, computer type, length, and representation
   b. Format — organization (e.g., indexed), transfer medium (e.g., disk), layout of frames (samples, records, blocks, and/or transmissions), and storage requirements

8. **Description of COMMON blocks,** including locations of any hard-coded physical constants

9. **Prologs/package specifications and PDL** for each unit (separate volume)

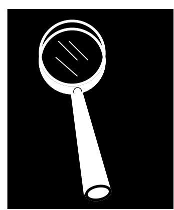
10. **Alphabetical list of subroutines from support data sets**, including a description of each subroutine's function and a reference to the support data set from which it comes

Figure 8-7. System Description Contents

Acceptance tests must be designed to verify that the system meets existing, documented requirements. All tests for robustness and performance must be based on stated requirements. The plan must explicitly specify the results that are expected from each test and any debug data or intermediate products that testers will require in order to evaluate the software.

The basic subset of tests that are to be run during regression testing are also identified in the test plan. The regression test suite should include those tests that verify the largest number of critical requirements in a single run, i.e., those that are the most comprehensive, yet efficient, tests to execute.

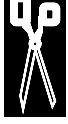
In general, the acceptance test plan should be constructed so that tests can be conducted independently. Multiple tests may be performed simultaneously, and testing should be able to continue after a problem is found. Tests must also be repeatable; if two different testers execute the same test case, the results should be consistent.



ACCEPTANCE TEST READINESS REVIEW

When the system is ready to be given to the acceptance test team, a formal "hand-over" meeting of developers and testers is held. The purpose of this *acceptance test readiness review* (ATRR) is to identify known problems, to establish the ground rules for testing, and to assist the acceptance test team in setting up the testing environment.

TAILORING NOTE



The formality of the ATRR should be tailored to the project. On large projects with many interfacing groups, the ATRR should be held as a formal review with hardcopy materials and slide/viewgraph presentations. On small projects, the ATRR may be an informal meeting held around a conference table.

The ATRR is an opportunity to finalize procedural issues and to reach agreement on the disposition of any unresolved problems. To facilitate this, the development team should conduct demonstrations of the system for the acceptance test team prior to the meeting. At the meeting, the discussion should cover the status of system tests, specification modifications, system test discrepancies, waivers, and the results of configuration audits.

The ATRR format and schedule are shown in Figure 8-8. An outline of recommended materials is provided in Figure 8-9.

ATRR FORMAT

Presenters
- System test team
- Acceptance test team
- Software development team

Other participants
- Quality assurance representatives
- Customer interfaces
- User representatives
- System capacity/performance analysts

Attendees must be familiar with the project background, requirements, and design.

Schedule — after completion of system testing and before the beginning of the acceptance testing phase

Agenda — establishment of ground rules and procedures for acceptance testing, identification of known problems, and discussion of the test environment

Materials — distributed a minimum of 3 days before the review.

Figure 8-8. ATRR Format

MATERIALS FOR THE ATRR

1. **Introduction** — outline of ATRR materials, purpose of the review, and system overview

2. **System test results**
   a. Summary of test results
   b. Status of discrepancies and waivers
   c. Unresolved issues

3. **Acceptance testing overview** — summary of testing approach, including any major changes to the acceptance test plan
   a. Organizational responsibilities during the acceptance testing phase
   b. Test configuration and environment, including facilities and computer resources
   c. Configuration management procedures
   d. Test tools, test data, and drivers/simulators
   e. Test reporting, analysis, and discrepancy-handling procedures
   f. Test milestones and schedules

4. **Test readiness assessment**
   a. Results of configuration audits
   b. System readiness
   c. Staff training and preparedness
   d. Status of the acceptance test plan
   e. Action items and schedule for their completion

Figure 8-9. ATRR Materials

EXIT CRITERIA

To determine whether the system and staff are ready to begin the acceptance testing phase, the management team should ask the following questions:

- Have the system and analytical test plans been executed successfully? Have all unresolved discrepancies either been eliminated as requirements or postponed to a later release in a formal waiver?
- Have the FCA and PCA been conducted? Have all resulting action items been completed?
- Are the user's guide and system description accurate, complete, and clear? Do they reflect the state of the completed system?
- Has the acceptance test plan been finalized?

When the management team can answer "Yes" to each question, the system testing phase can be concluded.

LIFE CYCLE PHASES	REQUIREMENTS DEFINITION	REQUIRE-MENTS ANALYSIS	PRELIMI-NARY DESIGN	DETAILED DESIGN	IMPLEMENTATION	SYSTEM TESTING	ACCEPTANCE TESTING

SECTION 9

THE ACCEPTANCE TESTING PHASE

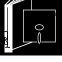
PHASE HIGHLIGHTS

ENTRY CRITERIA →

- System and analytical test plans successfully executed
- Acceptance test plan finalized
- User's guide and system description completed
- Configuration audits and ATRR conducted

EXIT CRITERIA

- Acceptance test plan successfully executed
- User's guide and system description finalized
- System formally accepted and delivered
- Software development history completed

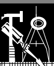


PRODUCTS

- System delivery tape
- Finalized user's guide and system description
- Acceptance test results
- Software development history

MEASURES

- Staff hours
- CPU hours
- SLOC in controlled libraries (cumulative)
- Changes and errors (by category)
- Requirements Q&As, TBDs, and changes
- Estimates of system size, effort, schedule, and reuse
- Acceptance test items planned/ executed/passed
- Discrepancies reported/resolved
- SEL project completion statistics



METHODS AND TOOLS

- Acceptance test plan
- Regression testing
- Configuration management
- Test evaluation and error-correction procedures
- Discrepancy reports, test logs, and test management tools

KEY ACTIVITIES

Acceptance Test Team

- Complete the preparations for acceptance testing
- Execute the tests items in the acceptance test plan
- Analyze and report test results
- Evaluate the user's guide
- Formally accept the system

Development Team

- Support the acceptance test team
- Correct discrepancies found during testing
- Train the acceptance test team to operate the system
- Refine system documentation
- Deliver the completed system

Management Team

- Allocate team responsibilities and finalize testing procedures
- Ensure the quality and progress of testing
- Ensure cooperation among teams
- Prepare the software development history

In the Cleanroom methodology, acceptance testing is complete when the mean time between failure (MTBF) of statistically generated tests exceeds a pre-determined threshold value.

OVERVIEW

The purpose of the acceptance testing phase is to demonstrate that the system meets its requirements in the operational environment.

The acceptance testing phase begins after the successful conclusion of an ATRR. The testing conducted during this phase is performed under the supervision of an independent acceptance test team (on behalf of the customer) and follows the procedures specified in the acceptance test plan. The development team provides training and test execution and analysis support to the test team. They also correct any errors uncovered during testing and update the system documentation to reflect software changes and corrections.

Acceptance testing is complete when each test item specified in the acceptance test plan has either been successfully completed or has been waived at the authorization of the customer, and the system has been formally accepted. The phase is concluded when the system delivery tape and final system documentation have been delivered to the customer for operational use.

Figure 9-1 is a flow diagram that shows the process of acceptance testing.

KEY ACTIVITIES

During this phase, the acceptance test team and the development team work closely together to set up and execute the tests specified in the acceptance test plan.

The acceptance test team is composed of the analysts who will use the system and several members of the team that prepared the requirements and specifications document. The acceptance test team supplies test data, executes the tests, evaluates the test results, and, when acceptance criteria are met, formally accepts the system.

Members of the development team correct any discrepancies arising from the software and finalize the system documentation. Often, the development team sets up and executes the first round of acceptance tests. This provides training for the acceptance test team, which then assumes responsibility for test execution during the subsequent rounds of acceptance testing. In any case, one member of the development team is always present during each testing session to offer operational and analysis support.

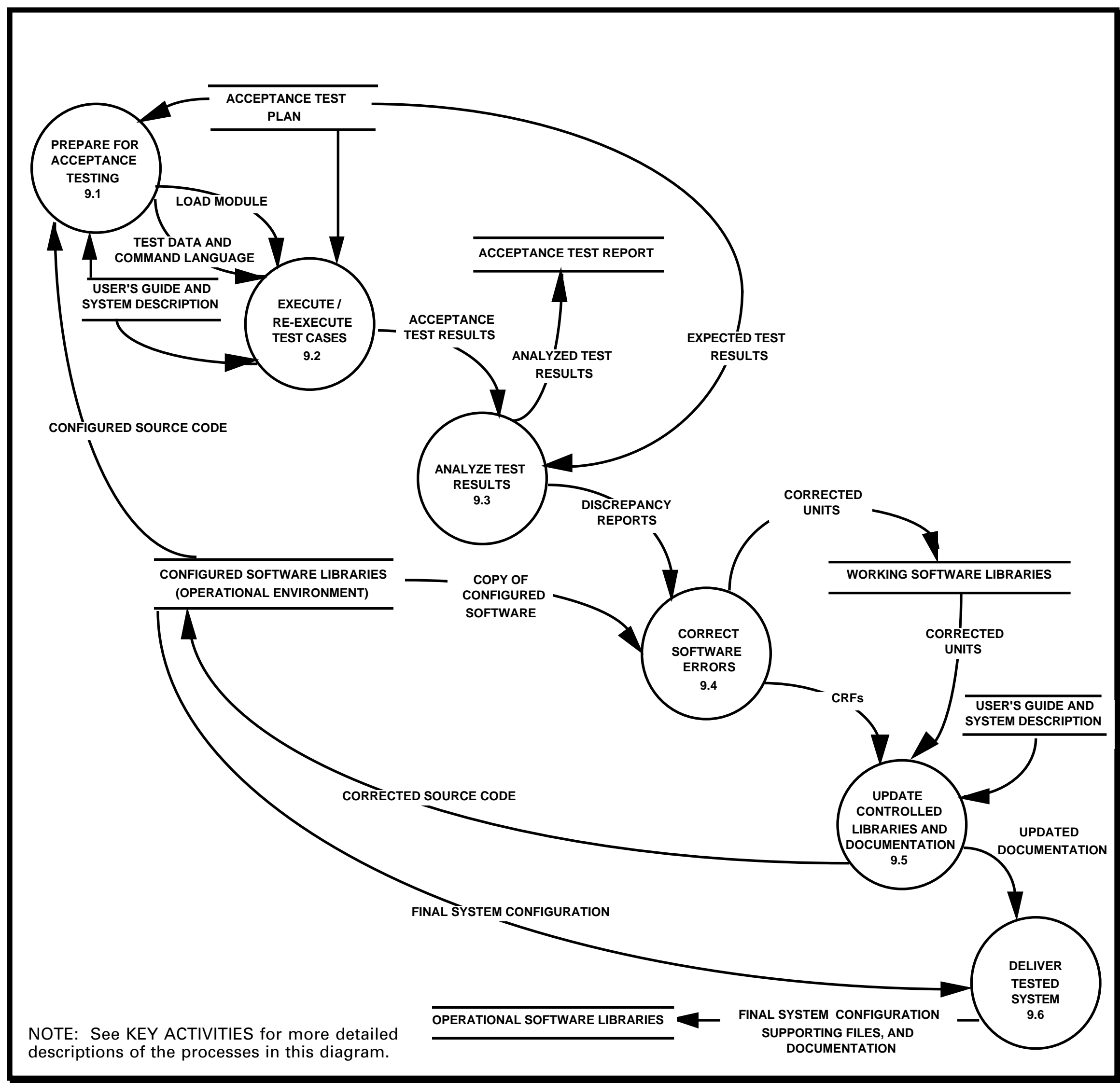


Figure 9-1. Acceptance Testing

Some acceptance testing activities, such as test set up and execution, generation of test data, and formal discrepancy reporting, can be performed by either the development team or by the acceptance test team. At the ATRR, the assignment of each of these activities to a particular team is negotiated by members of the management team. The management team, which includes managers of both the software development and acceptance test teams, bases these assignments on the specific needs and drivers of the project.

The key technical and managerial activities of the acceptance test phase are outlined in the following paragraphs. A recommended timeline for the execution of these activities is shown in Figure 9-2.

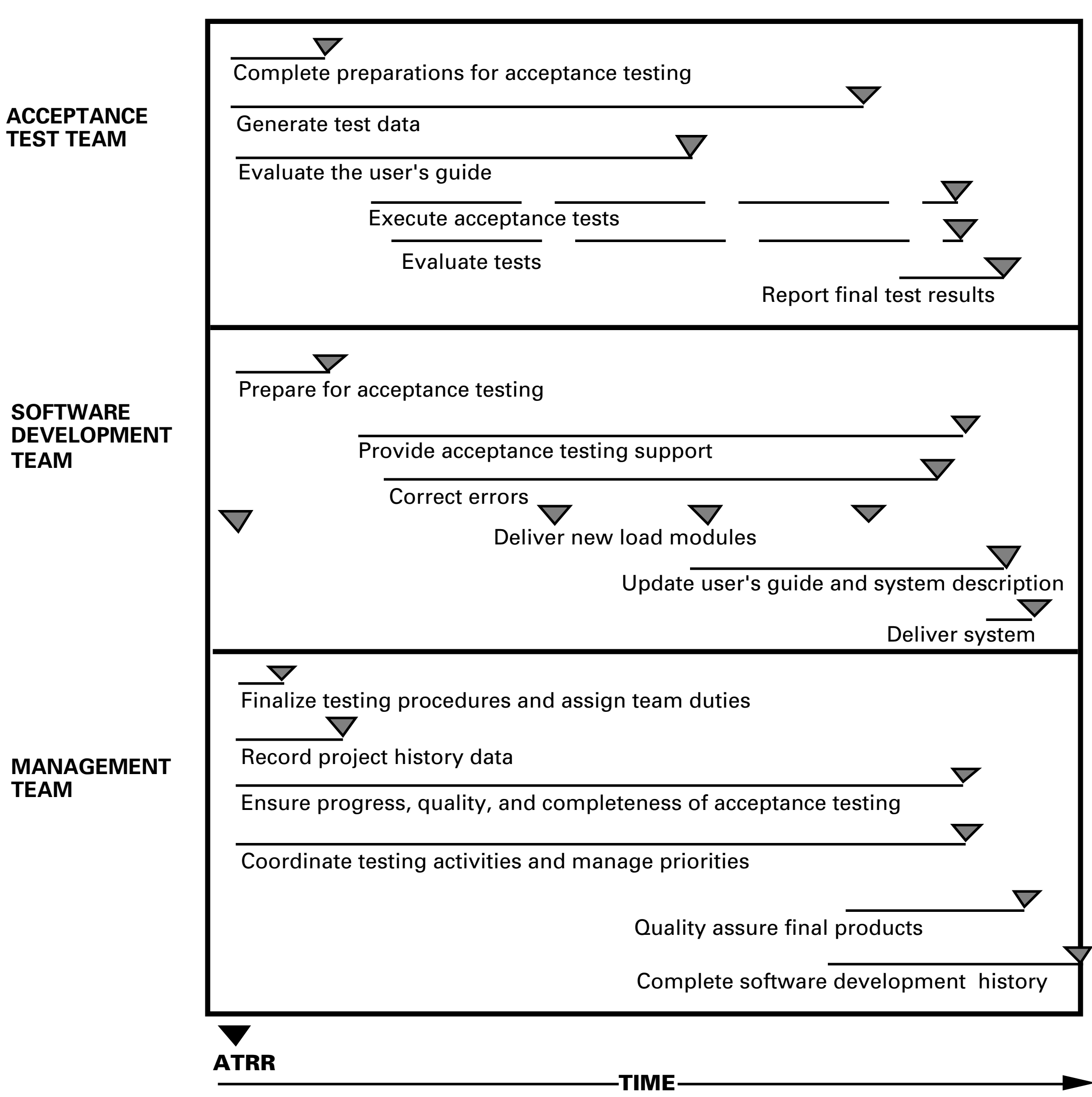


Figure 9-2. Timeline of Key Activities in the Acceptance Testing Phase

Activities of the Acceptance Test Team

- **Complete the preparations for acceptance testing.** Establish the acceptance test environment. Acceptance testing, unlike system testing, always takes place on the computer targeted for operational use; therefore, it is critically important to schedule computer resources well in advance.

  Finish generating the simulated data to be used in testing and allocate the necessary data sets. Quality check the simulated data to ensure that the required test data have been produced. Bear in mind that generating simulated data requires intensive effort and that, because of the complexity of flight dynamics test data, simulated data must often be generated several times before the desired data set is achieved. Plan ahead for adequate time and resources.

  Refer to the user's guide to determine the parameters needed to set up the test cases. Work with the development team to set up the tests accurately and to assure the quality of test preparations. Check the control-language procedures (JCL/DCL) for executing tests.

  Obtain the expected results from the acceptance test plan and review them for completeness. Calculate any expected results that are missing.

- **Execute the test items in the acceptance test plan.** Early in the phase, attempt to execute all tests in the order given in the acceptance test plan, executing each item of each test in sequence. This "shake-down" ensures that any major problems are found early, when they can be most easily corrected.

  If a basic misunderstanding of a requirement is found during initial test runs, such that further testing would be unproductive, suspend testing until a new load module can be produced. Negotiate with the development team to determine a schedule for the delivery of a corrected load module and the resumption of testing.

  Control the testing configuration. To relate test results to a particular load module unambiguously, run the entire set of tests designated for a round of testing on a specific load module before introducing any corrections. Continue testing until the number of errors encountered make further testing unproductive.

Introduce new load modules according to the test configuration management procedures. When a new, corrected load module is received, first rerun those tests that previously failed because of software errors. If those tests succeed, proceed with regression testing.

Execute the regression tests to demonstrate reproducible results and ensure that corrections have not affected other parts of the system. The basic subset of tests to be run during regression testing, as identified in the acceptance test plan, includes those that are the most comprehensive yet efficient tests to execute. The regression test suite may be modified to expedite testing or to provide different coverage as testers gain more experience with the system.

During each test session, review test results with the developers who are supporting the tests to ensure that the test was executed properly. At the end of the test session, produce a preliminary report listing the test cases that were executed and recording these initial observations.

- **Analyze and report test results.** Evaluate test results within one week of execution. Wherever possible, use automated tools in the analysis process. Record analysis procedures and keep all relevant materials. Remember that records and reports must give complete accounts of the procedures that were followed; if a test cannot be evaluated, note the fact and report the reasons for it.

  Compare test results with expected results and document all errors and discrepancies found, regardless of how minor they appear or whether they will be corrected.

  Prepare a detailed evaluation report for each test. This report should include a test checklist and a categorization of the results for each item of each test. Specific test evaluation procedures, including a five-level test categorization scheme, are discussed under *Methods and Tools*.

  Near the end of the phase, when all acceptance tests have been completed, prepare a final report documenting the results of testing. Include the final detailed report on each test, and provide an overall summary that gives an overview of the testing process and records how many test items passed during each round of testing.

- **Evaluate the user's guide.** During acceptance testing, evaluate the user's guide for accuracy, clarity, usability, and completeness. Provide this feedback to the development team so that they can update the document and issue a finalized version by the end of acceptance testing.

- **Formally accept the system.** When the system meets all acceptance criteria, except those waived by the customer, prepare a formal memorandum declaring the system accepted.

Activities of the Development Team

- **Support the acceptance test team.** Assist the acceptance test team in setting up and executing tests. At least one developer should be available during each test session to answer questions, to provide guidance in executing the system, and to review initial test results.

  Work with the acceptance test team to isolate the sources of discrepancies between expected and actual test results. Classify failed test items according to their causes, showing which items failed because of the same software discrepancies and which items could not be evaluated because a preceding test failed. Help the testers prepare formal discrepancy reports; typically, because a single factor may cause more than one failure, there are fewer discrepancy reports than failed test items.

> **NOTE**
>
> **The degree to which the development team assists the acceptance test team in setting up and executing tests, generating test data, or preparing discrepancy reports varies with the project. Management defines each team's duties at the start of the phase and documents them in the acceptance test procedures.**

- **Correct discrepancies found during testing.** Correct those software errors that occur because of faults in the design or code. Errors in the specifications are corrected as negotiated by the acceptance test manager, the development manager, and the customer. Because the requirements and specifications are under configuration control, the CCB must also approve any corrections to specifications. Incorporate software updates that result from approved corrections into a future load module.

> **RULE**
>
> **No specification modifications that add requirements or enhancements can be accepted during acceptance testing. Instead, these should be carefully considered for implementation during the maintenance phase.**

Use detailed test reports, test output, and the specifications to isolate the source of each error in the software. If an error is uncovered in the software design, the effect of the repair on costs and schedules may be significant. Look for a workaround and notify management immediately so that the impact and priority of the error can be determined.

Verify all corrections, using code reading, unit testing, and integration testing. Update the SENs for the revised units and fill out a CRF describing each correction. Strictly follow the same configuration management procedures for revised units as were used during system testing. To reduce the possibility of configuration errors, only the project librarian should change configured source libraries and build the executable image.

Deliveries of new acceptance test load modules should include a memorandum detailing which discrepancy reports have been resolved and which are still outstanding.

- **Train the acceptance test team to operate the system.** Ensure that members of the acceptance test team steadily gain expertise in running the system. Training may be completed at the beginning of acceptance testing or spread out over the phase. The schedule of training depends on when the acceptance test team is to assume full responsibility for executing the tests.

> **NOTE**
>
> **In the flight dynamics environment, training is limited to executing or operating the system. The development team does not train the operations team on how to use the system to support the mission.**

- **Refine system documentation.** Finalize the user's guide and the system description documents to reflect recent software changes and user comments.

- **Deliver the completed system.** After the system is accepted, prepare the final system tape. Verify its correctness by loading it on the operational computer, building the system from it, and using the test data on the tape to execute the regression test set specified in the acceptance test plan.

Activities of the Management Team

- **Allocate team responsibilities and finalize testing procedures.** Negotiate to determine the appropriate team structure, division of duties, testing procedures, and contingency plans for the project.

Define a realistic testing process that can be used on this project and adhere to it. Tailor the standard process to address drivers specific to the project, such as system quality, schedule constraints, and available resources.

Define the roles and responsibilities of each team clearly. Specify which teams are to generate simulated data, set up test cases, execute tests, and prepare formal discrepancy reports. Where teams share responsibility, clearly define the specific activities that each is expected to perform.

Stipulate contingency plans. Specify what will happen if a major error is encountered. Define the conditions that warrant suspension of testing until a correction is implemented. Define when and how testing should circumvent a problem.

- **Ensure the quality and progress of testing.** Ensure that team members adhere to procedures and standards. Analyze summaries of testing progress and examine plots of test discrepancies weekly. Use measures of effort and schedule expended to date to reestimate costs and staffing levels for the remainder of the project.

  Place special emphasis on enforcing configuration management procedures. The control of software libraries and test configurations is critical during the acceptance testing phase.

- **Ensure cooperation among teams.** Coordinate the activities of the acceptance test and the development teams. Ensure that the acceptance test team evaluates tests quickly and that the development team corrects software errors promptly so that the pace of testing does not decline. Assist in resolving technical problems when necessary.

  Ensure that error corrections are thoroughly tested by the development team before revised units are promoted to controlled libraries and delivered for retesting. Review all test results and system documentation.

- **Prepare the software development history.** At the beginning of acceptance testing, record measures and lessons learned from the system testing phase and

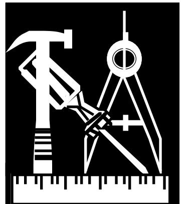
NOTE

The contents of the software development history are outlined under the PRODUCTS heading in this section.

add this information to the software development history.

Complete the phase-by-phase history at the end of acceptance testing and prepare the remainder of the report. Summarize the project as a whole, emphasizing those lessons learned that may be useful to future projects and process-improvement efforts. Include summary charts of key project data from the SEL database.

METHODS AND TOOLS



The key methods and tools of the acceptance testing phase are

- The acceptance test plan
- Regression testing
- Configuration management
- Test evaluation and error-correction procedures
- Discrepancy reports
- Test logs
- Test management tools

The final acceptance test plan, which describes the procedures to be followed in recording and evaluating acceptance tests and while correcting test discrepancies, is a product of the system testing phase and was discussed in Section 8.

Many of the automated tools used in the acceptance testing phase were introduced to facilitate the implementation of the system. These tools — configuration management systems, symbolic debuggers, static code analyzers, performance analyzers, compilation systems, and code-comparison utilities — are discussed in Section 7.

Test logs, discrepancy reports, test management tools, and regression testing procedures are the same as those used for system testing; they are discussed in detail in Section 8.

The other methods and tools in this list are elaborated in the paragraphs that follow.

Configuration Management

During the acceptance test phase, as in the other test phases, strict adherence to configuration management procedures is essential. Any changes to the code in the permanent source code libraries must be made according to established procedures and recorded on CRFs.

Configuration management procedures must ensure that the load modules being tested correspond to the code in the project's libraries.

Developers should follow the guidelines given in Section 8 for managing configured software as they repair discrepancies, test the corrections, and generate new load modules for acceptance testing.

Acceptance testers must be able to relate test results to particular load modules clearly. Therefore, the load modules must be under strict configuration control, and, at this point in the life cycle, the entire set of tests designated for a round of testing should be run on a specific load module before any corrections are introduced. Similarly, changes in the test setup or input data must be formally documented and controlled, and all tests should be labeled for the module and environment in which they were run.

Test Evaluation and Error-Correction Procedures

The acceptance test team is responsible for evaluating each acceptance test and forwarding the results to the development team.

Each test consists of a number of items to be checked. The results of analyzing each test item are categorized according to the following five-level evaluation scheme:

- Level 1: *Item cannot be evaluated or evaluation is incomplete.* This is normally the result of an incorrect test setup or an unsatisfied dependency. If this category is checked, the reason should be stated. If the test could not be evaluated because the test was inadequately specified, the test plan needs to be corrected.

- Level 2: *Item does not pass, and an operational workaround for the problem does not exist.*

- Level 3: *Item does not pass, but an operational workaround for the problem exists.*

- Level 4: *Only cosmetic errors were found in evaluating the item.*

- Level 5: *No problems were found.*

Typically, items categorized as Level 1 or 2 are considered to be high priority and should be addressed before items at Levels 3 and 4, which are lower priority. The intention is to document all errors and discrepancies found, regardless of their severity or whether they will ultimately be corrected.

Tests and test items are evaluated according to these criteria:

- All runs must execute correctly without encountering abnormal terminations or any other runtime errors.
- Numerical results must be within acceptable limits of the expected values.
- Tabular output must meet the criteria outlined in the specifications, with proper numerical units and clear descriptions of parameters.

The acceptance test team documents test results using two kinds of report:

preliminary test reports

- *Preliminary reports*, prepared at the end of each test session, provide a rapid assessment of how the software is working and early indications of any major problems with the system. They are prepared by the acceptance test team on the basis of a "quick-look" evaluation of the executions.

detailed test reports

- *Detailed reports*, prepared within a week of test execution, describe problems but do not identify their sources in the code. They are prepared by the acceptance test team by analyzing results obtained during the test sessions, using hand calculations and detailed comparisons with the expected results given in the acceptance test plan.

The testers and developers then work together to identify the software problems that caused the erroneous test results directly or indirectly. These software problems are documented and tracked through formal discrepancy reports, which are then prioritized and scheduled for correction.

The development team then corrects errors found in the software subject to the following guidelines:

- Software errors that occur because of a violation of the specifications are corrected, and the corrections are incorporated into an updated load module for retesting.
- Corrections resulting from errors in the specifications or analytical errors require modifications to the specifications. The customer and the CCB must approve such modifications before the necessary software updates are incorporated into an updated load module.

MEASURES

Objective Measures

During the acceptance testing phase, managers continue to collect and analyze the following data:

- Staff hours
- Total CPU hours used to date
- Source code growth
- Errors and changes by category
- Requirements questions and answers, TBDs, and changes
- Estimates of system size, effort, schedule, and reuse
- Test items planned, executed, and passed
- Discrepancies reported and resolved

The source of these measures and the frequency with which the data are collected and analyzed are shown in Table 9-1.

NOTE

A complete description of the PCSF and SEF can be found in *Data Collection Procedures for the SEL Database* (Reference 19).

At the completion of each project, SEL personnel prepare a *project completion statistics form (PCSF)*. The management team verifies the data on the PCSF, and fills out a *subjective evaluation form (SEF)*. The data thus gathered helps build an accurate and comprehensive understanding of the software engineering process. Combined with the software development history report, the SEF supplies essential subjective information for interpreting project data.

Evaluation Criteria

test status

The management team can gauge the reliability of the software and the progress of testing by plotting the number of acceptance test items executed and passed in each round of testing as a function of time. An exceptionally high pass rate may mean that the software is unusually reliable, but it may also indicate that testing is too superficial. The management team will need to investigate to determine which is the case. If the rate of test execution is lower than expected, additional experienced staff may be needed on the test team.

Table 9-1. Objective Measures Collected During the Acceptance Testing Phase

MEASURE	SOURCE	FREQUENCY (COLLECT/ANALYZE)	DATA COLLECTION	
			CONTINUED	BEGUN
Staff hours (total and by activity)	Developers and managers (via PRFs)	Weekly/monthly	*	
Changes (by category)	Developers (via CRFs)	By event/monthly	*	
Changes (to source files)	Automated tool	Weekly/monthly	*	
Computer use (CPU hours and runs)	Automated tool	Weekly/biweekly	*	
Errors (by category)	Developers (via CRFs)	By event/monthly	*	
Requirements (changes and additions to baseline)	Managers (via DSFs)	Biweekly/biweekly	*	
Requirements (TBD specifications)	Managers (via DSFs)	Biweekly/biweekly	*	
Requirements (Questions/answers)	Managers (via DSFs)	Biweekly/biweekly	*	
Estimates of total SLOC (new, modified, and reused), total units, total effort , and schedule	Managers (via PEFs)	Monthly/monthly	*	
SLOC in controlled libraries (cumulative)	Automated tool	Weekly/monthly	*	
Status (test items planned/executed/ passed)	Managers (via DSFs)	Weekly/biweekly	*	
Test discrepancies reported/resolved	Managers (via DSFs)	Weekly/biweekly	*	

Note: Test status is plotted separately for the system testing and the acceptance testing phases. Test discrepancies are also plotted separately for each phase.

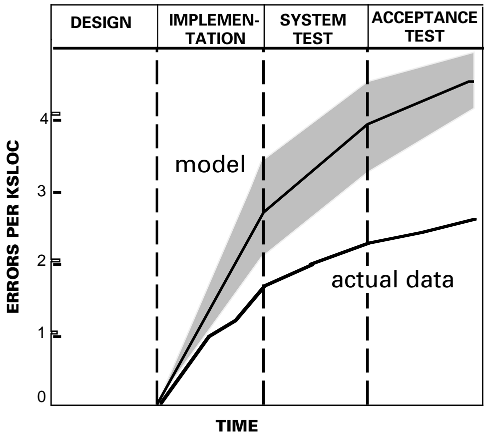
test discrepancies

The management team also monitors plots of the number of discrepancies reported versus the number repaired (see *Measures* in Section 8). If the gap between the number of discrepancies identified and the number resolved does not begin to close after the early rounds of acceptance testing, more staff may be needed on the development team to speed the correction process.

The trend of the error rates during the acceptance testing phase is one indication of system quality and the adequacy of testing. Based on SEL projects from the late 1980s and early 1990s, the SEL model predicts a cumulative error rate of 4.5 errors per KSLOC by the end of acceptance testing. Typically, the SEL expects about 2.6 errors per KSLOC in the implementation phase, 1.3 in system testing, and 0.65 in acceptance testing; i.e., error detection rates decrease by 50 percent from phase to phase.

error rates

An error rate above model bounds often indicates low reliability or misinterpreted requirements, while lower error rates indicate either high reliability or inadequate testing. In either case, the management team must investigate further to understand the situation fully. Figure 9-3 shows an example of an error-rate profile on a high-quality project monitored by the SEL.



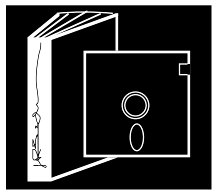
Symptom: Lower error rate and lower error detection rate.

Cause: Early indication of high quality. Smooth progress observed in uncovering errors, even between phases (well-planned testing).

Result: Proved to be one of the highest-quality systems produced in this environment.

Figure 9-3. Sample Error-Rate Profile, UARS AGSS

PRODUCTS



The key products of the acceptance testing phase are

- System delivery tape
- Finalized user's guide and system description
- Acceptance test results
- Software development history

System Delivery Tape

At the end of acceptance testing, a system delivery tape is produced that contains the accepted system code and the supporting files from the project's configured libraries. The system tape holds all of the information required to build and execute the system, including the following:

- The final load module
- The source code for the system
- The control procedures needed to build and execute the system
- All supporting data and command files
- Test data for regression tests

Two copies of the system tape are delivered to the customer.

Finalized User's Guide and System Description

The user's guide is updated to clarify any passages that the testers found confusing and to reflect any changes to the system that were made during the acceptance testing phase. The system description is also updated; system changes are incorporated into the document and any discrepancies found during the configuration audits are corrected. Final versions of both documents are delivered at the end of the project.

Acceptance Test Results

When acceptance testing is complete, test results are published either as a separate report or in an update of the acceptance test plan. (The latter method allows the reader to easily compare actual with expected results and eliminates some redundancy in describing test objectives; see Figure 7-8.) The individual report forms or checklists used during testing to log results are collected in a separate volume.

Software Development History

During the acceptance testing phase, the management team finalizes the software development history for the project. This report, which should be delivered no later than one month after system acceptance, summarizes the development process and evaluates the technical and managerial aspects of the project from a software engineering point of view. The purpose of the report is to help software managers familiarize themselves with successful and unsuccessful practices and to give them a basis for improving the development process and product. The format and contents of the software development history are shown in Figure 9-4.

EXIT CRITERIA

To determine whether the system is ready for delivery, the management team should ask the following questions:

- Was acceptance testing successfully completed?
- Is all documentation ready for delivery? Does the user's guide clearly and completely describe how to execute the software in its operational environment?
- Has the acceptance test team, on behalf of the customer, formally accepted the software?
- Have final project statistics been collected and have subjective evaluation forms been submitted? Is the software development history complete?

When the management team can answer "Yes" to each question, the acceptance testing phase is concluded.

SOFTWARE DEVELOPMENT HISTORY

Material for the development history is collected by the management team throughout the life of the project. At the end of the requirements analysis phase, project data and early lessons learned are compiled into an initial draft. The draft is expanded and refined at the end of each subsequent phase so that, by the end of the project, all relevant material has been collected and recorded. The final version of the software development history is produced within 1 month of software acceptance. The suggested contents are as follows:

1. **Introduction** — purpose of system, customer of system, key requirements, development machines and language

2. **Historical overview** by phase — includes products produced, milestones and other key events, phase duration, important approaches and decisions, staffing information, and special problems
   a. Requirements definition — if requirements were produced by the software development team, this section provides an historical overview of the requirements definition phase. Otherwise, it supplies information about the origin and documentation of the system's requirements and functional specifications
   b. Requirements analysis
   c. Preliminary design
   d. Detailed design
   e. Implementation — coding through integration for each build/release
   f. System testing
   g. Acceptance testing

3. **Project data**
   a. Personnel and organizational structure — list of project participants, their roles, and organizational affiliation. Includes a description of the duties of each role (e.g., analyst, developer, section manager) and a staffing profile over the life of the project
   b. Schedule — table of key dates in the development of the project and a chart showing each estimate (original plus reestimates at each phase end) vs. actual schedule
   c. Project characteristics
      (1) Standard tables of the following numbers: subsystems; total, new, and reused units; total, new, adapted and reused (verbatim) SLOC, statements, and executables; total, managerial, programmer, and support hours; total productivity
      (2) Standard graphs or charts of the following numbers: project growth and change histories; development effort by phase; development effort by activity; CPU usage; system test profile; error rates; original size estimate plus each reestimate vs. final system size; original effort estimate plus each reestimate vs. actual effort required
      (3) Subjective evaluation data for the project — copy of the SEL subjective evaluation form (SEF) or report of SEF data from the project data base (see Reference 19)

4. **Lessons learned** — specific lessons and practical, constructive recommendations that pinpoint the major strengths and weaknesses of the development process and product, with particular attention to factors such as the following:
   a. Planning — development plan timeliness and usefulness, adherence to development plan, personnel adequacy (number and quality), etc.
   b. Requirements — completeness and adequacy for design, change history and stability, and clarity (i.e., were there misinterpretations?), etc.
   c. Development — lessons learned in design, code and unit testing
   d. Testing — lessons learned in system and acceptance testing
   e. Product assurance — adherence to standards and practices; QA and CM lessons learned
   f. New technology — impact of any new technology used by the project on costs, schedules, quality, etc. as viewed by both developers and managers; recommendations for future use of the technology

Figure 9-4. Software Development History Contents

SECTION 10

KEYS TO SUCCESS

HIGHLIGHTS

KEYS TO SOFTWARE DEVELOPMENT SUCCESS

- Understand your environment
- Match the process to the environment
- Experiment to improve the process
- Don't attempt to use excessively foreign technology

DOs AND DON'Ts FOR PROJECT SUCCESS

DO...

- Develop and adhere to a software development plan
- Empower project personnel
- Minimize the bureaucracy
- Establish and manage the software baseline
- Take periodic snapshots of project health and progress
- Reestimate system size, staff effort, and schedules regularly
- Define and manage phase transitions
- Foster a team spirit
- Start the project with a small senior staff

DON'T...

- Allow team members to proceed in an undisciplined manner
- Set unreasonable goals
- Implement changes without assessing their impact and obtaining proper approval
- Gold plate
- Overstaff
- Assume that intermediate schedule slippage can be absorbed later
- Relax standards
- Assume that a large amount of documentation ensures success

KEYS TO SOFTWARE DEVELOPMENT SUCCESS

The recommended approach to software development, described in the preceding sections, has been developed and refined over many years specifically for the flight dynamics environment. The methodology itself is a product of the SEL's learning experience. The SEL has found the following to be the keys to success for any software development organization.

- **Understand the environment.** Before defining or tailoring a methodology for your project or organization, determine the nature of the problem, the limits and capabilities of the staff, and the support software and hardware environment. Collect baseline measurements of staff effort, system size, development schedule, source code growth, software changes and errors, and computer usage.

- **Match the process to the environment**. In planning a project or defining a standard methodology, use your understanding of the people, environment, and problem domain to select and tailor the software process. There must be a match. Be sure that the elements of the process have a rationale and can be enforced. If you don't intend to or cannot enforce a standard or a procedure, do not include it in the plan. Make data collection, analysis, and reporting integral parts of the development methodology.

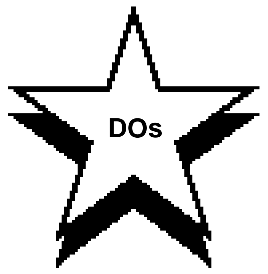
- **Experiment to improve the process**. Once a comfortable match between the process and environment is defined, stretch it, a little at a time, to improve it continuously. Identify candidate areas for improvement, experiment with new techniques or extensions to the process, and measure the impact. Based on a goal of improving the process and/or product, select candidate extensions with a high likelihood of success in the environment. There should always be an expectation that the change will improve the process. Be careful not to change too many things at once so that the results from each change can be isolated. Be aware that not all changes will lead to improvement, so be prepared to back off at key points.

- **Don't attempt to use excessively foreign technology.** Although minor improvements to the standard process can be easily shared from one organization to another, carefully consider major changes. Do not select and attempt a significantly different technology just because it was successful in other situations. The technology, and the risk that accompanies its adoption, must suit the local environment.

DOs AND DON'Ts FOR PROJECT SUCCESS

As standard practice, the SEL records lessons learned on successful and unsuccessful projects. The following DOs and DON'Ts, which were derived from flight dynamics project experience, are key to project success.

Nine DOs for Project Success



- **Develop and adhere to a software development plan.** At the beginning of the project, prepare a software development plan that sets project goals, specifies the project organization and responsibilities, and defines the development methodology that will be used, clearly documenting any deviations from standard procedures. Include project estimates and their rationale. Specify intermediate and end products, product completion and acceptance criteria, and mechanisms for accounting progress. Identify risks and prepare contingency plans.

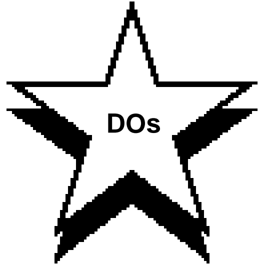
  Maintain and use the plan as a "living" document. Record updates to project estimates, risks, and approaches at key milestones. Provide each team member with the current software development plan. Periodically, audit the team for adherence to the plan.

- **Empower project personnel.** People are a project's most important resource. Allow people to contribute fully to the project solution. Clearly assign responsibilities and delegate the authority to make decisions to specific team members. Provide the team with a precise understanding of project goals and limitations. Be sure the team understands the development methodology and standards to be followed on the project. Explain any deviations from the standard development methodology to the team.

- **Minimize the bureaucracy.** Establish the minimum documentation level and meeting schedule necessary to fully communicate status and decisions among team members and management. Excessive meetings and paperwork slow progress without adding value. Don't try to address difficulties by adding more meetings and management. More meetings plus more documentation plus more management does not equal more success.

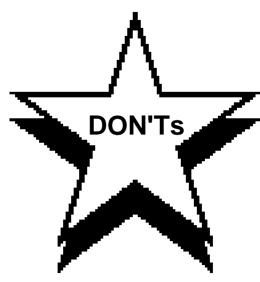
- **Establish and manage the software baseline.** Stabilize requirements and specifications as early as possible. Keep a detailed list of all TBD items — classified by severity of impact in terms of size, cost, and schedule — and set priorities for their resolution. Assign appropriate personnel to resolve TBD items; follow their progress closely to ensure timely resolution. Estimate and document the impact to costs and schedules of each specifications change.
- **Take periodic snapshots of project health and progress, replanning when necessary.** Compare the project's progress, product, and process measures against the project's plan. Also compare the current project with similar, past projects and with measurement models for the organization. Replan when the management team agrees that there is a significant deviation. Depending on project goals and limitations, alter the staffing, schedule, and/or scope of the project. Do not hesitate to reduce the scope of the work when project parameters dictate.

  When the project significantly exceeds defined limits for the local environment, stop all project activity, audit the project to determine the true project status, and identify problem areas. Examine alternatives and devise a recovery plan before proceeding again.
- **Reestimate system size, staff effort, and schedules regularly.** As the project progresses, more information is learned about the size and complexity of the problem. Requirements change, the composition of the development team changes, and problems arise. Do not insist on maintaining original estimates. Each phase of the life cycle provides new and refined information to improve the estimates and to plan the project more effectively. There is nothing wrong with realizing that the size has been underestimated or that the productivity has been overestimated. It is wrong not to be doing something to detect this and take the appropriate action. As a rule, system size, effort, and schedule should be reestimated monthly.
- **Define and manage phase transitions.** Much time can be lost in the transition from one phase to the next. Several weeks before the start of each new phase, review progress to date, and set objectives for the next phase. See that the developers receive training in the activities of the next phase, and set intermediate goals for the team. Clarify any changes that have been made to the development plan. While senior developers are finishing up the current phase, get junior members of the team started on the next phase's activities.

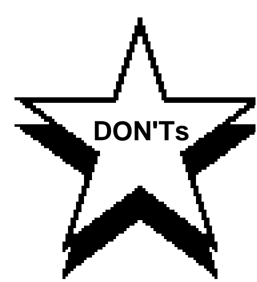


- **Foster a team spirit.** Projects may include people from different organizations or companies. Maximize commonality and minimize differences among project members in all aspects of organization and management. Clearly define and communicate the roles of individuals and teams, but provide an overall project focus. Cross-train as much as possible. Hold combined team meetings so that everyone gets the same story. Have everyone follow the same rules. Report progress on a project level as well as a team level. Struggle through difficulties and celebrate successes together as a unit, helping and applauding each other along the way.
- **Start the project with a small senior staff.** A small group of experienced senior people, who will be team leaders during the remainder of the project, should be involved from the beginning to determine the approach to the project, to set priorities and organize the work, to establish reasonable schedules, and to prepare the software development plan. Ensure that a plan is in place before staffing up with junior personnel.

Eight DON'Ts for Project Success



- **Don't allow team members to proceed in an undisciplined manner.** Developing reliable, high-quality software at low cost is not a creative art; it is a disciplined application of a set of defined principles, methods, practices, and techniques. Be sure the team understands the methodology they are to follow and how they are to apply it on the project. Provide training in specific methods and techniques.
- **Don't set unreasonable goals.** Setting unrealistic goals is worse than not setting any. Likewise, it is unreasonable to hold project personnel to commitments that have become impossible. Either of these situations tends to demotivate the team. Work with the team to set reasonable, yet challenging, intermediate goals and schedules. Success leads to more success. Setting solid reachable goals early in the project usually leads to continued success.
- **Don't implement changes without assessing their impact and obtaining proper approval.** Estimate the cost and schedule impact of each change to requirements and specifications, even if the project can absorb it. Little changes add up over time. Set priorities based on budget and schedule constraints. Explore options with the change originator. In cases where changes or corrections are proposed during walk-throughs, document the proposed changes in the minutes but do not implement them until a formal approved specification modification is received.



- **Don't "gold plate".** Implement only what is required. Often developers and analysts think of additional "little" capabilities or changes that would make the system better in their view. Again, these little items add up over time and can cause a delay in the schedule. In addition, deviations from the approved design may not satisfy the requirements.

- **Don't overstaff, especially early in the project.** Bring people onto the project only when there is productive work for them to do. A small senior team is best equipped to organize and determine the direction of the project at the beginning. However, be careful to provide adequate staff for a thorough requirements analysis. Early in the project, when there are mostly 'soft' products (e.g., requirements analysis and design reports), it is often hard to gauge the depth of understanding of the team. Be sure the project has enough of the right people to get off to a good start.

- **Don't assume that an intermediate schedule slippage can be absorbed later.** Managers and overly optimistic developers tend to assume that the team will be more productive later on in a phase. The productivity of the team will not change appreciably as the project approaches completion of a phase, especially in the later development phases when the process is more sequential. Since little can be done to compress the schedule during the later life cycle phases, adjust the delivery schedule or assign additional senior personnel to the project as soon as this problem is detected.

  Likewise, don't assume that late pieces of design or code will fit into the system with less integration effort than the other parts of the system required. The developers' work will not be of higher quality later in the project than it was earlier.

- **Don't relax standards in an attempt to reduce costs.** Relaxing standards in an attempt to meet a near-term deadline tends to lower the quality of intermediate products and leads to more rework later in the life cycle. It also sends the message to the team that schedules are more important than quality.

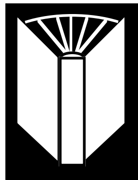
- **Don't assume that a large amount of documentation ensures success.** Each phase of the life cycle does not necessarily require a formally produced document to provide a clear starting point for the next phase. Determine the level of formality and amount of detail required in the documentation based on the project size, life cycle duration, and lifetime of the system.

ACRONYMS

AGSS	attitude ground support system
ATR	Assistant Technical Representative
ATRR	acceptance test readiness review
BDR	build design review
CASE	computer-aided software engineering
CCB	configuration control board
CDR	critical design review
CM	configuration management
CMS	Code Management System
COF	component origination form
CPU	central processing unit
CRF	change report form
DBMS	database management system
DCL	Digital Command Language
DEC	Digital Equipment Corporation
DFD	data flow diagram
DSF	development status form
FCA	functional configuration audit
FDF	Flight Dynamics Facility
GSFC	Goddard Space Flight Center
IAD	interface agreement document
ICD	interface control document
I/O	input/output
ISPF	Interactive System Productivity Facility
IV&V	independent verification and validation
JCL	job control language
KSLOC	thousand source lines of code
LSE	language-sensitive editor
MOI	memorandum of information
MOU	memorandum of understanding
NASA	National Aeronautics and Space Administration
OOD	object-oriented design
PC	personal computer
PCA	physical configuration audit
PCSF	project completion statistics form

ACRONYMS (cont.)

PDL	program design language (pseudocode)
PDR	preliminary design review
PEF	project estimates form
PRF	personnel resources form
PSRR	preliminary system requirements review
QA	quality assurance
Q&A	questions and answers
RID	review item disposition
RSL	reusable software library
SCR	system concept review
SDE	Software Development Environment
SDMP	software development/management plan
SEF	subjective evaluation form
SEL	Software Engineering Laboratory
SEN	software engineering notebook
SFR	software failure report
SIRD	support instrumentation requirements document
SLOC	source lines of code
SME	Software Management Environment
SOC	system and operations concept
SORD	systems operations requirements document
SPR	software problem report
SRR	system requirements review
SSR	software specifications review
STL	Systems Technology Laboratory
STR	software trouble report
STRR	system test readiness review
TBD	to be determined



REFERENCES

1. Software Engineering Laboratory, SEL-81-104, *The Software Engineering Laboratory*, D. N. Card et al., February 1982

2. P.A. Currit, M. Dyer, and H.D. Mills, "Certifying the Reliability of Software, " *IEEE Transactions on Software Engineering,* Vol. SE-12, No. 1, January 1986, pp. 3-11

3. Software Engineering Laboratory, SEL-90-002, *The Cleanroom Case Study in the Software Engineering Laboratory: Project Description and Early Analysis,* S. Green et al., March 1990

4. —, SEL-91-004, *The Software Engineering Laboratory (SEL) Cleanroom Process Model,* S. Green, November 1991

5. H.D. Rombach, B.T. Ulery, and J. D. Valett, "Measurement Based Improvement of Maintenance in the SEL," *Proceedings of the Fourteenth Annual Software Engineering Workshop*, SEL-89-007, November 1989

6. H.D. Rombach, B.T. Ulery, and J. D. Valett, "Towards Full Life Cycle Control: Adding Maintenance Measurement to the SEL," *Journal of Systems and Software;* scheduled for publication in 1992

7. F. McGarry and R. Pajerski, "Towards Understanding Software— 15 Years in the SEL," *Proceedings of the Fifteenth Annual Software Engineering Workshop*, SEL-90-006, November 1990

8. J.W. Bailey and V. R. Basili, "Software Reclamation: Improving Post-Development Reusability," *Proceedings of the Eighth Annual National Conference on Ada Technology,* March 1990. Also published in *Collected Software Engineering Papers: Volume VIII*, SEL-90-005, November 1990

9. M. Stark, "On Designing Parametrized Systems Using Ada," *Proceedings of the Seventh Washington Ada Symposium*, June 1990. Also published in *Collected Software Engineering Papers: Volume VIII*, SEL-90-005, November 1990

10. Flight Dynamics Division Code 550, NASA FDD/552-90/083, *Extreme Ultraviolet Explorer (EUVE) Attitude Ground Support System (AGSS) Software Development History*, B.Groveman et al., October 1990

11. G. Booch, *Object-Oriented Design (with Applications)*, Benjamin/Cummings: Redwood City, CA, 1991

12. Software Engineering Laboratory, SEL-84-101, *Manager's Handbook for Software Development (Revision 1),* L. Landis, F. McGarry, S. Waligora, et al., November 1990

13. E. Yourdon and L. L. Constantine, *Structured Design*, Yourdon Press: NY, 1978

14. T. DeMarco, *Structured Analysis and System Specification*, Yourdon, Inc.: NY, 1978

15. P. Ward and S. Mellor, *Structured Development for Real-Time Systems,* Prentice-Hall: Englewood Cliffs, NJ, 1985

16. P. Coad and E. Yourdon, *Object-Oriented Analysis*, Yourdon Press: NY, 1991

17. Software Engineering Laboratory, SEL-86-002, *General Object-Oriented Software Development,* E. Seidewitz and M. Stark, August 1986

18. —,SEL-83-001, *An Approach to Software Cost Estimation,* F. E. McGarry, G. Page, D. N. Card, et al., February 1984

19. —,SEL-92-002, *Data Collection Procedures for the Software Engineering Laboratory (SEL) Database,* G. Heller, J. Valett, M.Wild, March 1992

20. IBM, Systems Integration Division, TR. 86.00002, *A Design Method for Cleanroom Software Development*, M. Dyer, August 1983

21. H. Mills, "Stepwise Refinement and Verification in Box Structured Systems, " *IEEE Computer*, June 1988

22. Software Engineering Laboratory, SEL-87-002, *Ada Style Guide (Version 1.1)*, E. Seidewitz et al., May 1987

23. —, SEL-86-001, *Programmer's Handbook for Flight Dynamics Software Development*, R. Wood and E. Edwards, March 1986

24. R. C. Lingen, H. D. Mills, and B. I. Witt, *Structured Programming: Theory and Practice,* Addison-Wesley: Reading, Mass., 1979

25. Software Engineering Laboratory, SEL-85-001, *A Comparison of Software Verification Techniques,* D. Card, R. Selby, F. McGarry, et al., April 1985

26. —, SEL-85-005, *Software Verification and Testing,* D. Card, B. Edwards, F. McGarry, et al., December 1985

27. Flight Dynamics Division Code 550, 552-FDD-92/001R1UD0, *Flight Dynamics Software Development Environment (FD/SDE): SDE Version 4.2.0 User's Guide: Revision 1,* M. Durbeck and V. Hensley, February 1992

INDEX

A

Ada 15, 72
- compilation 124
- CPU time and 97
- design 73
  - Fig. 5-4 73
- detailed design phase and 93
- environment 93, 95
- implementation phase and 95
- libraries 95
- LSE 95
- package specifications 68, 74, 89, 93
- PDL 74, 89, 93, 97
- SEL environment and 2
- specifications 68
- style 95
- testing and 120

Analysis
- code 70, 76, 77, 91
- domain 15
- object-oriented *see* OOD
- report *see under* Documents
- requirements *see under* Requirements
- structured 22, 28, 44
  - development team and 7, 42
- tools 49, 70, 76, 91, 139

Analysts
- *notecard* 7
- acceptance test team 10, 86, 162
- communication and 64, 66, 69
  - *notecard* 27
- management team and 45
- requirements analysis phase 47
- requirements definition team 27, 64, 66
- reuse and 7
- reviews and 12, 75, 80, 133
- system test team 115, 137

Analyzers
- code
  - source 124
  - static 77, 147, 170
    - FORTRAN SAP 123
    - RXVP80 123
    - VAX SCA 77, 123
- performance 147, 170
  - Problem Program Evaluator 77
  - TSA/PPE (Boole & Babbage) 123, 141
  - VAX Performance and Coverage Analyzer 123, 141

Application specialists 30
- *notecard* 27
- acceptance testing phase and 143
- change approval and 122
- code reading 119
- development team 90, 93, 108, 110, 137, 150
- documentation and 108, 110
- implementation phase and 110
- inspection team 93
- reviews and 120, 121
- system test team 115, 137
- testing and 121, 149

Audits, configuration 143, 146, 157, 176
- FCA (functional configuration audit) 146
- PCA (physical configuration audit) 146

B

Baseline
- builds and 123
- configuration management tools and 123
- diagrams 71
- libraries and 123
- requirements and specifications document and 8, 27, 39, 45

Baseline *continued*
SDMP and 54
SEL 19
SEN and 74
specific environment and 180, 182
Boole & Babbage *see* TSA/PPE, *under* Analyzers
Builds 11, 12, 90, 99, 101, 111
activities of 110
baselines for 123
changes to 114, 116, 124, 130
development team and 121
documentation and 69
estimates and estimation 114, 127
implementation phase
Fig. 7-1 109
life-cycle phases and
Fig. 7-2 110
load module 113
errors 172
final 176
librarian and 140, 168
libraries and 153
test report forms and 147
testing *see under* Testing
plan *see* plan, build, *under* Documentation
programmers and 108
reviews of 108, 113, 115, 132
notecard 12
see also BDR, *under* Reviews
schedule 103, 132
SEN and 122
system testing phase 152
test bed 120
test plan *see under* plan, *under* Documents
testing *see under* Testing

C

CASE (computer-aided software engineering) 49, 123
SEL environment and 2, 123
CAT *see under* Libraries (for software)
CCB (configuration control board) 49, 153
CDR and 102
PDR and 80
SRR and 39
Central processing unit *see* CPU
Change report form *see under* Forms
Cleanroom methodology 19
notecard 19
SEL environment and 2
see also Phase Highlights tables at beginnings of sections
CMS (Code Management System) 122, 145
Code
Ada PDL 93
analysis *see under* Analysis
analyzers *see* Analyzers
configuration management and 123
detailed design phase 86
estimates and estimation 127
executable 77
experience and 76
libraries 69, 115
reading 110, 117, 120, 126, 141
application specialists and 119
SEL 117
reviews
Fig. 7-4 118
scaffolding *see* drivers, *under* Prototypes and prototyping
source 108
configuration management and 122
libraries 122, 127, 129, 145, 146
measure 124, 127
standards 90
unit 111
defined *notecard* 8
Code Management System *see* CMS
COF *see under* Forms
Components 70, 71
defined *notecard* 8
FDF environment and 71
origination form (COF) *see under* Forms
reuse of *see under* Reuse
reviews 92
see also Reviews
SEL and 71
SEN and 74
Computer-aided software engineering *see* CASE
Configuration
Analysis Tool *see* CAT, *under* Libraries
audits *see* Audits, configuration
control board *see* CCB
management 13, 69, 74, 90, 115, 116, 122, 170
acceptance test team and 144
COF and 121, 126

Configuration *continued*
CRF and 128
implementation phase 115
implementation team and 13
management team and 142
SEN and 95, 122
source code and 122
system test team and 139
system testing phase 136, 145
tools 123
CPU (central processing unit) 45
time 77, 96, 97, 98, 125, 127, 150, 173
CRF *see under* Forms
Criteria
acceptance 10, 26, 181
requirements and specifications document and 32
entry 61
notecard 7
see also Phase Highlights tables at beginnings of sections
evaluation
acceptance testing phase 172, 173
detailed design phase 96
implementation phase 126
preliminary design phase 77
prototyping and 58
requirements analysis phase 52
requirements definition phase 31
system testing phase 150
exit
notecard 7
acceptance testing phase 177
detailed design phase 105
implementation phase 133
preliminary design phase 69, 82
requirements analysis phase 61
requirements definition phase 39
system testing phase 159
see also Phase Highlights tables at beginnings of sections
review 12, 93
SSR 12
test 26
Critical design review *see* CDR, *under* Reviews
Customer
delivery to 10
needs and reviews 12, 69, 108, 114, 133, 140, 146, 149, 155
prototypes and 14
requirements definition and 6, 114, 137
requirements definition team managers and 27
technical representative 13, 35, 108

D

Dan Bricklin's Demo Program 50
Data
Table 4-1 51
abstraction of 71
collection of 1, 51, 77, 139, 150, 180
Table 5-1 77
forms for 18
dictionary 46
externally visible 72
flow 77
Fig. 5-1 64
historical 44, 45, 52
page fault 77
requirements analysis process 42
structures 76
test 139, 140, 144, 155
Database
project histories 18
SEL 2, 120
specialist 119
Debuggers 123, 147
Decomposition, functional 28, 70, 71
Design
approaches and 64
detailed phase *see under* Phase
deviations from 184
documentation of
see Diagrams, Documents
inspections *see* Inspections
object-oriented *see* OOD
preliminary
review of *see* PDR, *under* Reviews
preliminary phase *see under* Phase
requirements and specifications document and 58
reuse and 15, 141
Developers
acceptance testing phase and 143
ATRR and 157
CMS and 123
COF and 120, 122
communication and 90, 92, 129

Developers *continued*
- *notecard* 27
- configuration audits and 143
- CRF and 122, 128
- design phases and 16
- errors and 9, 64
- estimates and estimation and 77
- "gold-plating" habit of 184
- implementation phase 9, 17
- measures and 18
- methods and tools of 70, 71, 73, 74, 95, 123, 124
- overly optimistic 184
- PDL and 74, 95
- practice runs and 144
- preliminary design phase 64
- prologs and 74, 95
- prototypes and prototyping 76
- requirements analysis phase 47
- requirements definition team and 7, 69
- reuse and 7, 27, 32, 76
- reviews and 12, 64, 80, 82, 108, 117, 119
- SEN and 74, 89, 95, 122, 146
- SOC and 32
- STL 93
- testing and 119, 120, 121, 129, 132, 145, 153
- training of 182
- walk-throughs and 75, 87

Diagrams, design 64, 66, 68, 70, 71, 74, 75, 86, 89, 91, 93, 95, 105, 141
- OOD and 66

Dictionary, data 46

Discrepancies *see under* Forms

Documents
- checklists
  - code inspection 117, 118
  - code reading 120
  - design inspection 74, 76, 89, 93, 95
    - Fig. 6-3 94
  - SEN and 117, 146
  - test review 120, 121, 147, 153, 176
- configuration management and 13
- detailed design 8, 68, 80, 86, 89, 91, 98, 99, 102, 105, 108, 113
  - generating Fig. 6-1 87
  - system description and 113
  - user's guide and 113
- developers and 32
- forms *see* Forms
- plan
  - acceptance test 10, 91, 116, 129, 144
    - Fig. 7-8 130
    - contents of 142, 155, 157
    - exit criteria and 159
    - requirements and specifications document and 10, 155
    - system testing phase and 153
  - analytical test 91, 111, 116, 119, 129, 132, 137, 167
    - contents of 132
    - exit criteria and 159
  - build 69, 89, 90, 102, 108, 113, 115
    - contents of 99, 101
    - management team and 98, 99
    - preliminary 99
  - build test
    - Fig. 7-8 130
    - development team and 101, 108, 111, 121, 129
    - exit criteria and 133
    - load module and 113
    - regression tests and 108, 130
  - development 46, 102, 181, 183
    - changes to 182
  - implementation 9
  - integration test 120
  - management *see* SDMP, *hereunder*
  - phase 14, 27
  - prototyping 54, 58
  - reuse 32, 54, 68, 76
    - updates to 66
  - system test 9, 137, 144, 149, 153, 167
    - contents of 130, 131
    - execution of 132
    - exit criteria and 134, 159
    - generating 9, 108, 110, 113
    - libraries and 122
    - management team and 115
    - requirements and specifications document and 9
    - small projects and 13
  - test 89, 98, 122
    - Fig. 7-8 131
    - FCA and 146
    - reuse and 15
    - unit 117, 119

see also name of specific document, hereunder
prototyping and 14
quality assurance and 13
report
audit 147
detailed test 166, 168
discrepancy *see under* Forms
preliminary acceptance test 166
preliminary design 8, 64, 68, 70, 80, 82, 99
addenda to 80
contents of 80
Fig. 5-5 81
updates to 68
requirements analysis 7, 45, 46, 50, 184
contents of 54
Fig. 4-4 54
distribution of 61
SSR and 59
summary 147
test report forms *see under* Forms
requirements and specifications
acceptance test plan and 10, 142, 155
baseline of requirements 39
clarity of 52, 79
classification of items 44
contents of 7, 32
Fig. 3-4 34
development team and 7, 44, 71
errors and 149
exit criteria and 39
generating 182
Fig. 3-2 24
libraries and 50
preliminary design phase and 64
requirements analysis phase and 42, 47, 54
requirements definition phase and 32
requirements definition team and 42
reuse and 45
SOC and 22
SRR and 36, 42
system test plan and 9, 146
updates to 8, 39, 44, 47, 49, 58
reuse and 15
SCR and 35
SDMP (software development/management plan) 45, 50, 54, 69
Fig. 4-5 52
notecard 11
contents 55
Fig. 4-5 56
IV&V and 149
managers as authors 55
prototypes and 58
updates to 99, 114
SEN (software engineering notebook) 74
audits and 146
baseline 74
configuration management 95, 122
contents of 74, 89, 95, 117, 120
detailed design phase and 91
developers and 89, 95
exit criteria and 134
implementation phase and 116, 122
libraries 74, 122, 146
preliminary design phase and 70
updates to 95, 141, 146, 168
SIRD (system instrumentation requirements document) 22, 48
small projects and 13
SOC (system and operations concept document) 32, 45
generating Fig. 3-1 23
requirements and specifications and 22
requirements definition phase and 32
reuse proposal included 25
updates to 32, 45
software development history 68, 89, 114, 142
SORD (system operations requirements document) 22, 48
structure charts *see* Diagrams, design
system description 9, 108, 137, 146
audits and 146
contents of 155
Fig. 8-7 156
detailed design document and 113
exit criteria and 159
final version of 10, 141
system testing phase and 153
tools to produce diagrams for 29
user's guide 9
audits and 146
contents of 153
Fig. 8-6 154
detailed design document and 113

Documents *continued*
draft of 9, 108, 113, 115, 122, 129, 134, 153
exit criteria and 159
final version of 10, 141, 168, 176, 177
system testing phase and 137, 153
Domain analysis *see* Analysis, domain

E

Editor *see* LSE
Entry criteria *see* Criteria, entry
Environments
acceptance testing 157
Ada 93
constraints of 61, 180
CPU use and 127
development team and 101
FDF 1, 52, 71, 72, 96, 122, 124, 149, 180
analysts and 137
analytical test plans in 132
SDE 124
SEL 120, 123
Fig. 1-1 1
STL 1, 124
test 115, 137, 139, 147
transfer among 124, 142, 180
understanding 180
Estimates and estimation 53
code 127
cost 25, 45, 46, 52, 54, 56, 114, 142
documentation of 181
effort *see* staff, *hereunder*
measures and 77, 79, 96, 124
Table 6-1 97
performance 44
performance modeling and 77
requirements 30, 31
reuse 51, 97, 150
schedule 46, 51, 52, 54, 56, 150
specification modifications and 79, 96, 182, 183
staff 31, 51, 52, 56, 58, 150
notecard 10
system 51, 52, 53, 56, 58, 97, 150
updates to 79
TBD requirements and 52, 54, 61
unstable requirements and
Fig. 6-4 98
updates to 68, 69, 79, 96, 114, 127, 181, 182
Exit criteria *see* Criteria, exit

F

FCA *see under* Audits, configuration
FDF (Flight Dynamics Facility) 1
RSL 76
see also under Environment
Forms
COF (component origination form) 120, 126
CRF (change report form) 122, 145
developers and 128
librarian and 122
data collection 18
discrepancy report 122, 140, 147, 169, 170
acceptance test team and 163
closure of 149
development team and 163, 167, 168
exit criteria and 159
management team and 149
SEL model 152
system test team and 149
PCSF (project completion statistics form) 173
requirements question-and-answer 47, 48, 50, 78, 92, 96, 124, 143, 150
RID (review item disposition form) 45, 58
notecard 37
CDR and 89, 105
exit criteria and 39, 61, 82, 105
PDR and 68, 80
requirements definition team and 70, 91
SRR and 39
SEF (subjective evaluation form) 173
test report 121, 147

FORTRAN
FDF environment and 71
PDL 68, 74
preliminary and detailed design phases and
Fig. 5-3 72
prologs 68, 74
see also SAP, *under* Analyzers

G
H

I

Images, executable *see* load module, *under* Builds
Implementation phase *see under* Phase
Inspections 29, 133
  design 92
Interactive System Productivity Facility *see* ISPF
Interfaces 8, 25, 26
  Ada and 72, 74
  detailed design phase 92
  errors 74, 117
  exit criteria and 39, 105
  implementation phase 108, 111
  inspection team and 76
  OOD and 71, 72
  PDR and 80
  preliminary design phase 8, 64, 66, 68, 70, 71, 75
  requirements analysis phase 49
  reuse and 16, 77
  smaller projects 35
  TBD requirements and 52, 78
  user 101, 119
    detailed design phase 92
ISPF (Interactive System Productivity Facility) 124
Items
  review 32, 37, 48
    see also RID, *under* Forms
  test 165
    evaluation of 171, 172
    reports and 166, 167
IV&V (independent verification and validation) 149

J

K

L

Language-sensitive editor *see* LSE
Libraries
  (for documents)
    contents of 74
    program library manager 95
    project 50, 122
      contents of 89, 95, 122
      development team and 113
      exit criteria and 134
      version control 123
    SEN and 122, 146
  (for software) 115, 122, 142, 145, 169
    CAT (Configuration Analysis Tool) 123, 124
    CDR and 90
    changes to 122, 140, 168, 170
    compilation systems and 124
    contents of 69, 129, 153
    control of 142, 145
    documentation and 143, 146
    exit criteria and 133
    measures and 127, 128
    PANEXEC 124
    PANVALET 123, 124
    system delivery and 176
    tools for 122, 123, 124
    usage as measure 124, 126
  development team and 89
  librarian 50, 89, 122
    COF and 121, 122
    CRF and 122
    executable images and 121
    load module and 113
    SEN and 95
  management team and 69, 90
  reuse and 16
  RSL 76, 89, 124, 141
Life cycle, software development
  activities of 5
  builds *see* Builds
  estimates and estimation and 182
  measures of *see* Measures
  milestones *notecard* 7
  phases of 5, 51
    see also Phases
  Recent SEL paper on *notecard* 3
  releases *see* Releases
  reuse and 15, 16
    Fig. 2-2 15
  reviews *see* Reviews
  SDMP and 55
    notecard 11
  tailoring 11, 169
    notecard 12, 25
  see also Phase Highlights tables at beginnings of sections

Load module *see under* Builds
LSE (language-sensitive editor) 95
    PDL and 74
    VAX 123

M

Measures
    acceptance testing phase 169, 173
        Table 9-1, 173
    detailed design phase 96, 124
        Table 6-1 97
        recording of 114
    developers responsible for 18
    estimates and estimation as 79
    implementation phase 114, 124, 126
        Table 7-1 125
    library usage as 126, 127
    life-cycle phases and 17
        Fig. 2-3 18
    preliminary design phase 77, 96
        Table 5-1 78
    project histories database 18
    requirements analysis phase 50, 77
        Table 4-1 51
    requirements definition phase 30
        Table 3-1 31
    SEL 19, 126, 128
        Fig. 2-3 18
        notecard 3
        Table 2-1 18
    software development history and 169
    source code as 127
    system testing phase 142, 150
        Table 8-1 151
    see also Phase Highlights tables at beginnings of sections
Methods 28, 47
    acceptance testing phase 170
    development plan and 181
    discipline and 183
    elements of 180
    environment and 180
    integration testing 120
    prototype assessment 58
    see also Analysis
    see also name of particular method; Phase Highlights tables at beginnings of sections
Metrics *see* Measures
Mills, Harlan 19
Models and modeling
    performance 77
    SEL discrepancy status Fig. 8-5 152
    SEL error 175
Modules
    configuration management and 95
    defined *notecard* 8
    implementation of 114
    integration of 108, 110, 111, 120
    load *see under* Builds
    management system *see under* VAX
    SEL 120
    SEN and 95, 122
    stubs 120
    testing of 110, 111, 120

N

O

OOD (object-oriented design) 8, 15, 28, 42, 44, 49, 64
    Ada and 72
    design diagrams and 66, 91
    detailed design phase and 87, 91
    implementation phase and 113
    PDL and 68
    preliminary design phase and 70, 71, 73, 74
    prologs and 68
    SEL environment and 2

P

PANEXEC *see under* Libraries (for software)
PANVALET *see under* Libraries (for software)
PCA *see under* Audits, configuration
PCSF *see under* Forms
PDL (program design language)
    Ada 74, 93, 97
    coding statements and 111
    development team and 68, 89
    documentation of 80, 86, 95, 98, 119
    evaluation criteria and 77
    exit criteria and 82, 105
    FORTRAN 74
    implementation phase and 74
    management team and 69

methods and tools 70, 91
quality assurance and 117
reviews of 75, 93
SEL and 74
PDR *see under* Reviews
Phase
Note that exit criteria of one phase are the entry criteria of the next phase; see also Phase Highlights tables at beginnings of sections
acceptance testing
acceptance test plan and 155
entry criteria 159
evaluation criteria 171, 172, 173
exit criteria 177
flow diagram Fig. 9-1 163
products 176
transition to 143
detailed design
Ada and 93
design diagrams and 71, 86
entry criteria 82
evaluation criteria 96
exit criteria 105
flow diagram Fig. 6-1 87
formalisms produced during 8
FORTRAN and
Fig.5-3 72
key activities 86, 89
Fig. 6-2 88
measures of 96, 114
methods and tools 91, 92
OOD and 91
package bodies and 73
preliminary design report and 68
products 86, 101, 108
reuse analysis and 96
TBD requirements and 52, 96
transition to 69, 86
walk-throughs 92
implementation 9
configuration management and 115
CPU time and 127
detailed design document and 8
documentation and 141, 144, 153
entry criteria 105
estimates and estimation and 127
evaluation criteria 126
exit criteria 9, 133
flow diagram Fig. 7-1 109
key activities 110
Fig. 7-3 112
lessons learned 142
libraries and 127
measures of 114, 124
Table 7-1 125
methods and tools 116, 147
PDL and 74
products 108, 122, 129, 132
prologs and 74
TBD requirements and 115
transition to 70, 105, 108
maintenance and operations 10
notecard 10
changes deferred to 143
phase not specifically addressed in this document 10
reuse and 17
preliminary design
Ada and 72
diagram of Ada systems Fig. 5-4 73
entry criteria 61
evaluation criteria 77
exit criteria 69, 82
flow diagram Fig. 5-1 65
key activities 66
lessons learned 89
measures of 77, 96
Table 5-1 78
methods and tools 70
products 71, 80, 99
Fig. 5-5 81
requirements analysis report and 54
requirements and specifications document and 64
TBD requirements and 52, 66
transition to 46
requirements analysis 42, 44, 46, 64, 184
entry criteria 39
evaluation criteria 52
exit criteria 61
flow diagram Fig. 4-1 43
key activities 44
Fig. 4-2 46
lessons learned 68
measures of 50, 77
methods and tools of 47
products 46, 54
prototypes and prototyping and 45

Phase *continued*
reuse analysis and 45
walk-throughs 42, 44, 45, 47
requirements definition 22
evaluation criteria 31
exit criteria 39
flow diagram Fig. 3-1 23
key activities 25
measures of 30
Table 3-1 30
methods and tools 28
products 26, 27, 32
prototyping and 30
reuse analysis and 16
walk-throughs 30
system testing
acceptance test plan and 155
analytical test plan and 136
entry criteria 133
evaluation criteria 150
exit criteria 137, 159
flow diagram Fig. 8-1 136
key activities 137
measures of 142, 150
Table 8-1 151
evaluation of 150
methods and tools 144
products 153
purpose of 136
requirements and specifications document and 146, 149
reuse and 141
system test plan and 136
TBD requirements and 150, 155
user's guide and 137
Plans *see under* Documents
PPE (Problem Program Evaluator) 77
Preliminary design review *see* PDR, *under* Reviews
Preliminary system requirements review *see* PSSR, *under* Reviews
Products
acceptance testing phase 176
detailed design phase 86, 98
diagrams 86
development plan and 181, 182
exit criteria and 105, 134
implementation phase 129, 144
intermediate 26, 99, 157, 181, 184
libraries and 16
preliminary design phase 64, 69, 71, 80
Fig. 5-5 81
prototyping plan *see under* Documents
quality assurance and 13, 29, 69, 90, 115
requirements analysis phase 54
requirements definition phase 32
reuse 15
review 27, 42, 46, 92
soft 184
system testing phase 153
tailoring and 11
see also names of specific products; Phase Highlights tables at beginnings of sections
Program library manager *see under* Libraries
Programmers
builds and 108
CASE and 123
LSE and 123
Prologs 68
Ada *see* package specifications, *under* Ada
development team and 89
documentation of 80, 86, 95, 98
evaluation criteria and 77
exit criteria and 82, 105
FORTRAN 68, 74
implementation phase and 74
LSE and 95
management team and 69
methods and tools 70, 91
reviews of 75, 93
SEL and 74
Prototypes and prototyping 14, 45, 69
customer and 14
detailed design phase and 87, 91, 97
documentation of 14, 58
see also plan, *under* Documents
drivers 68, 71, 74, 76, 120, 123
evaluation 58
flight dynamics environment and 14
guidelines for *notecard*s 14
implementation phase and 120
interface 49
objective of 58
plan *see under* plan, *under* Documents
preliminary design phase and 64, 66, 76, 80
requirements analysis phase and 49
requirements definition phase and 30
SDMP and 58

PSRR *see under* Reviews

Q

Quality assurance 13, 69, 74, 80, 90, 93, 102, 115, 142, 143, 165, 169
  documents and 141

R

Regression testing *see under* Testing
Releases 11, 12, 126
  changes deferred to 143, 159
  documentation of 99, 101, 108, 129
  implementation of 109
  life-cycle phases and
    Fig. 7-2 110
Reports *see under* Documents
Requirements
  analysis phase *see under* Phase
  audits and 143
  changes to 143, 159, 182, 183
    Fig. 6-4 98
    BDR and 126
    CDR and 102
    communication of 90, 114
    CPU time and 127
    detailed design phase and 89, 96
    implementation phase and 114, 116
    measures and 128
    preliminary design phase and 79
  classification of 44, 47
  customer and 22, 155
  definition of 22
    methods and tools for 28
  definition phase *see under* Phase
  discrepancy and 149
  generalization of 15
  misinterpreted 175
  review of system *see* SRR, *under* Reviews
  TBD (to-be-determined) 52, 77, 99
    BDR and 126
    classification of 52
    defined 48
    detailed design phase and 96
    development team and 44, 47
    estimation of risk 54, 61
    exit criteria and 39, 61, 105
    interfaces and 52, 78
    management team and 45
    measure 51, 124
    PDR and 80
    preliminary design phase and 78
    requirements analysis and 42
    requirements analysis report and 54
    requirements definition 7
    requirements definition team and 44, 86, 90
    resolution of 52, 64, 69, 70, 86, 90, 96, 115, 155
    smaller projects 12
    total requirements and 31
  testing and 136, 162
  total 31
Requirements definition of 31
Requirements definition phase *see under* Phase
Reusable Software Library *see* RSL, *under* Libraries
Reuse
  acceptance test phase and 173
  activities enabling 15
    Fig. 2-2 15
  analysis and verification 16, 66, 70, 76, 80, 91
    see also candidate software, *hereunder*
  applications specialists and *notecard* 27
  candidate software 16, 22, 64, 66, 76, 82, 89, 96, 141
  current projects and 16
  design and 15, 16, 66
  developers and 17
  documentation and 15, 32, 45, 66, 68, 76
  estimates and estimation 77, 79, 96, 97, 124
  key elements of *notecard* 15
  libraries 16, 69
  life cycle and 15, 16
    Fig. 2-2 15
  performance analyzers and 77
  pitfalls *notecard* 25
  preliminary design phase and 76
  preservation techniques 17
  prototypes and prototyping and 76
  requirements analysis and design phases and 7
  specifications and 15
  verbatim or with modifications 17
Reviews
  BDR (build design review) 108, 113, 115, 132

Reviews *continued*
build test plan and 129
format of
Fig. 7-9 133
hardcopy materials for
Fig. 7-10 134
requirements definition team and 116
TBD requirements and 126
CDR (critical design review) 102
conduct of 8, 86, 89, 90, 113
exit criteria and 105
format
Fig. 6-6 103
hardcopy materials for 99, 104
implementation phase and 108
libraries and 90
project size and 12
prototypes and prototyping and 89
requirements definition team and 48, 91
RIDs and 89, 105
TBD requirements and 126
test plan and 101, 113
criteria for 12, 93
format and contents recommended *notecard* 12
PDR (preliminary design review) 8, 64, 69
CDR and 102
detailed design phase and 86
exit criteria and 80, 82
format of
Fig. 5-6 82
hardcopy materials for
Fig. 5-7 83
preliminary design phase and 80
requirements definition team and 70
RIDs and 68
tool development and 12
product *see under* Product
PSRR (preliminary system requirements review) 45
notecard 26, 37
SCR (system concept review) 7, 22, 35
Fig. 3-1 22
customer and 35
format of Fig. 3-5 35
hardcopy materials for Fig. 3-6 36
SRR (system requirements review) 7, 36, 42, 45
notecard 37
conduct of Fig. 3-7 37
exit criteria and 39
generating
Fig.3-2 23
hardcopy materials for Fig. 3-8 38
requirements and specifications document and 32, 36
small projects and 12
SSR (software specifications review) 7, 59
conduct of 42, 44, 45, 59
Fig. 4-6 59
criteria for 12
hardcopy materials for 46
Fig. 4-7 59
scheduling 46
STRR (system test readiness review) 115
tailoring the life cycle and 12
RID *see under* Forms
Rombach, Dieter *notecard* 3
RSL *see under* Libraries

S

SAP *see under* Analyzers
SCA *see under* Analyzers
SCR *see under* Reviews
SDE *see under* FDF, *under* Environment
SDMP *see under* Documents
SEF *see under* Forms
SEL (Software Engineering Laboratory) 50, 71, 74, 117, 120
baseline developed by 19, 170
checklist for code readers Fig. 7-4 118
COF and 120
CRF 122
database 120
environment 123
Fig. 1-1 1
error rate model 175
evaluation forms 177
experiments in 19, 150
Fig. 8-5 152
history of 1
keys to success 180
lessons learned 181
measures 126, 128
models and modeling Fig. 8-57 152
system description recommended by 155
SOC *see under* Documents

Software
  configuration manager *see* librarian, *under* Libraries
  development/management plan (SDMP) *see* SDMP, *under* Documents
  specifications review *see* SSR, *under* Reviews
Software Engineering Laboratory *see* SEL
Software Through Pictures 29
Source Code Analyzer Program *see under* Analyzers
Specifications
  completed 31
  reuse and 15
  review of *see* SSR, *under* Reviews
SRR *see under* Reviews
SSR *see under* Reviews
Staff hours *see under* Estimates and estimation
STL (Systems Technology Laboratory) 1
Structure charts *see* Diagrams, design
Stubs *see under* Modules
System
  concept review *see* SCR, *under* Reviews
  description document *see under* Documents
  instrumentation
    requirements document *see* SIRD, *under* Documents
  operations concept document *see* SOC, *under* Documents
  requirements review *see* SRR, *under* Reviews
  requirements *see* Requirements
  test plan *see* plan, *under* Documents
  test *see* Testing, system
System Architect 29
System concept review *see* SCR, *under* Reviews
Systems Technology Laboratory *see* STL

T

Tailoring *see under* Life cycle
TBD *see under* Requirements
Teams
  acceptance test 9, 10, 111
    acceptance test plan and 162
    analytical test plan and 119, 132
    assumption of responsibility 168
    ATRR and 137, 155, 157
    composition of 86, 162
    demonstrations and 143, 144, 157
    development team and 144, 162, 163, 165, 167
    documentation and 129, 144, 155, 172
    implementation phase and 108
    key activities 91, 116, 137, 144, 162, 165
    test evaluation and 171
    training of 168
  communication among 42, 45, 64, 90, 113, 114, 116, 167, 169
  development 7, 8, 45, 49, 52
    acceptance test plan and 116
    acceptance test team and 141, 142, 162, 165, 167, 168
    acceptance testing and 10, 162, 163
    analytical test plan and 91, 116, 129, 132
    application specialists on 108, 110, 137
    ATRR and 141, 142
    BDR and 108, 116, 132
    build test plan and 101, 108, 129
    build tests and 121, 127
    builds and 99
    CASE and 49
    CDR and 102
    composition of 68, 114
    configuration management and 69, 136
    demonstrations and 157
    detailed design phase and 86
    discrepancy reports and 149
    documentation and 8, 10, 46, 47, 49, 50, 108, 115, 129, 137, 141, 153, 155
    error correction and 172
    exit criteria and 82, 105
    inspection team and 75, 93
    integration testing and 120
    key activities 66, 86, 110, 111, 137, 141, 162, 167
    librarian 50
    methods and 49
    PDL and 93
    PDR and 8, 80
    preliminary design phase and 61, 64, 68
    products 98
    prototypes and prototyping and 49, 76, 97
    quality assurance and 69
    question-and-answer forms and 48

Teams *continued*
- requirements analysis phase and 42, 44, 47
- requirements and specifications document and 47
- requirements changes and 122
- SDMP and 54
- SSR and 46, 59
- system test plan and 108
- system test team and 136, 141
- testing and 9, 126
- training of 68
- walk-throughs and 75, 86, 92

functional decomposition and 91
implementation
- configuration management and 13
- large projects and 13

inspection 76
- composition of 75, 93
- design inspections and 93

maintenance 17
management 45
- acceptance test team and 143
- acceptance testing and 163
- audits and 146, 147
- builds and 89, 99, 108, 121
- communication and 181, 183
- composition of 68, 69, 163
- configuration management and 69, 142
- customer and 27
- development team and 68, 142, 169
- discrepancies and 150
- documentation and 115, 173, 177
- exit criteria and 105, 159, 177
- IV&V and 149
- key activities 68, 89, 114, 137, 142, 168
- libraries and 69
- products 98
- quality assurance and 69, 173
- requirements changes and 96, 98, 116, 142, 182
- system evaluation and 150
- system test plan and 130
- system test team and 142
- TBD requirements and 96
- testing and 120, 153

membership *notecard* 22
OOD and 91
operations concept 6
requirements definition 7, 8, 27, 45, 48, 49, 66
- baselined requirements and 8
- CDR and 102
- detailed design phase and 86
- documentation and 42, 49, 58, 155
- exit criteria and 39
- key activities 66, 90, 115
- PDR and 80
- preliminary design phase and 64
- question-and-answer forms and 48
- requirements analysis phase and 42, 44, 47
- requirements changes and 114
- requirements definition phase and 22
- requirements question-and-answer forms and 92
- SRR and 39, 42, 44
- SSR and 58
- walk-throughs and 75, 86, 87, 92

system test 115, 136
- analysts on 115, 137
- application specialists on 115
- composition of 115, 137
- development team and 136
- discrepancy reports and 149
- documentation and 153
- exit criteria and 134
- key activities 137
- system test plan and 130
- testing and 153

see also Phase Highlights tables at beginnings of sections

Testing 74
- acceptance 86, 115, 123, 127, 128
  - completion of 10, 162
  - flow diagram Fig. 9-1 163
  - purpose of 162
  - releases and 12
  - *see also under* Phase
- acceptance plan *see* plan, *under* Documents
- build 101, 108, 111, 113, 116, 121, 127, 129, 130, 171
- development team and 9
- drivers 120
- integration
  - Fig. 7-5 121
- load module 145, 146, 165, 166, 167, 168, 171, 172

module 110, 111, 116, 120
plan *see* plan, *under* Documents
regression
build 108, 121
build test plan and 130
small projects and 13
system 115, 123, 126, 127, 128, 129, 130, 134
completion of 9
test plan *see under* Documents
see also under Phase
unit 77, 95, 108, 110, 111, 114, 116, 119, 126, 128
Tools 28, 47, 49, 70, 166
libraries and 50
see also Phase Highlights tables at beginnings of sections
TSA/PPE (Boole & Babbage) *see under* Analyzers

U

Ulery, B. *notecard* 3
Units
central processing *see* CPU
correction of 122
defined *notecard* 8
TBD requirements in 52
see also Testing
User's guide *see under* Documents

V

Valett, Jon *notecard* 3
VAX *see name of particular product*
Verification
exit criteria and 133
independent and validation *see* IV&V
reuse *see under* Reuse
unit 117

W

Walk-throughs 29
Walk-throughs *see also under name of phase, under* Phase

X

Y

Z