HUGHES

HUGHES STX CORPORATION

# Introduction to the Software Engineering Guidebook

Pradip Sitaram

**Hughes STX Corporation**

September 14, 1995

HUGHES
HUGHES STX CORPORATION

## Purpose & Scope of this presentation

- Provide some background and historical information
- Describe organizational entities
- Terminology
- High-level overview of the Guidebook
- First step in Software Engineering Training
  - Requirements Engineering
- Overview of the other phases of development
- Overview of Software Project Management activities
- Overview of Software Support activities
- Feedback

Pradip Sitaram

HUGHES
HUGHES STX CORPORATION

## Background Information

- **SWEI -** Software Excellence Initiative (Ken Klenk)
  - Pete Mumford - Chairperson
  - Focus groups' Point's of Contact
  - Division Representatives
  - Business Development Representative
  - Training coordinator
- Focus groups
  - SEPG - Software Engineering Process Group (contact - Pradip Sitaram)
    - » To train and guide software developers in software engineering principles
  - SEL - Software Engineering Laboratory (contact - Temp Johnson)
    - » To provide hardware and software resources
  - CSSE - Center for Software and Systems Excellence (contact - TBD)
    - » Facilitate technology transfer and support special interest groups (SIGs)
  - Metrics
  - Training
  - and other groups as needed

Pradip Sitaram

## Background Information (cont.)

HUGHES
HUGHES STX CORPORATION

- Roles and Responsibilities
  - Division Reps are the formal channel of communication between the projects and the SWEI (and the focus groups)
  - HSTX Representative will interface with other Hughes organizations
  - Software professionals will inform their division reps of their support requirements
- Points of Contact:
  - Rick Dorsey - Space and Earth Sciences Div. - rdorsey@ccmail.stx.com
  - Larry Hogle - Business Development - lhogle@ccmail.stx.com
  - Temp Johnson - Applied Sciences & Tech Div. - tjohnson@ccmail.stx.com
  - Bob Kurtz - Systems Technology Div. - kurtz@mustang.nrl.navy.mil
  - Cathie Meetre - Computing & Data Mgt. Div. meetre@selsvr.stx.com
  - Pete Mumford - Earth Resources Div. - mumf@sioux.sodak.net
  - Pradip Sitaram - SEPG - sitaram@selsvr.stx.com

Pradip Sitaram

## Software Process Maturity

HUGHES
HUGHES STX CORPORATION

**See Attachment 11**

Pradip Sitaram

## Software Process Maturity (cont.)

HUGHES
HUGHES STX CORPORATION

- Assessment Findings
  - Inconsistent/undefined procedures and standards
  - Insufficient Training
  - Customer environment sometimes not suitable for formal processes
  - CM, QA and Testing not integrated
  - Software size, cost, and schedule estimation processes undocumented
  - Metrics are not collected
- Recommendations
  - Develop tailorable guidelines
  - Institute a training program
  - Support staff working at customer environments/sites
  - Effectively integrate QA, CM and Testing
  - Establish guidelines for software size, cost and schedule estimation
  - Identify Metrics and establish a tracking and reporting mechanism

Pradip Sitaram

## Why was the Guidebook developed

HUGHES
HUGHES STX CORPORATION

- Overall understanding of software engineering principles
- Provide a common engineering perspective
- Integrated approach to software development, management and support
- Provide software engineering information
  - Lifecycle Process Models
  - Development methodologies
  - Checklists
  - Tailoring guidelines
- Help improve the software engineering processes
- Help support proposal activities

Pradip Sitaram

## In the context of *cmi* ...

HUGHES
HUGHES STX CORPORATION

A process must first be **manageable** before it can be improved in an orderly and sustained manner. A software process is manageable when it is:

- **Defined and Documented**—Inputs, outputs, work activities, and responsibilities are outlined and delimited.
- **Measured**—Inputs, outputs, work activities, and resources are measured to provide a basis for control and improvement.
- **Controlled**—A predetermined mechanism exists to maintain a process in its desired state.
- **Continuously Improved and Optimized**—A predetermined mechanism exists to improve and optimize the process. Software process management cycles through the following stages:
  - » Process definition
  - » Measurement and feedback from use
  - » Evaluation leading to improvement and optimization

Pradip Sitaram

## Guidebook Developers

HUGHES
HUGHES STX CORPORATION

- Writers:
  - Members of the SEPG and software developers and managers from all across the company
- Reviewers:
  - Over 70 software professionals across HSTX
- Editing and Graphics:
  - SEPG members
  - Publications Resource Center

Pradip Sitaram

## Guidebook References

HUGHES
HUGHES STX CORPORATION

- Information Systems Division (Hughes Aircraft), Division 48's Software Engineering Handbook
- NASA's Software Engineering Laboratory series documents
- DOD documents
- Technical Papers
- Textbooks

Pradip Sitaram

## Intended Audience

HUGHES
HUGHES STX CORPORATION

- Programmers
- Analysts
- Engineers
- Managers
- Quality Assurance staff
- Configuration Management staff

**Distribution note:**
This is a proprietary document, restricted to Hughes STX employees

**Guidebook Abstract:**
Intended for customers and as a marketing document

Pradip Sitaram

## How the Guidebook can Help

HUGHES
HUGHES STX CORPORATION

- Helps to select the appropriate process model setting the baseline from which progress can be measured.
- Helps answer the following:
  - Do you know what your software is doing and what it should do?
  - Does your customer know what the software will do?
  - Do you know what conditions will cause your software to fail?
  - Will successors to your project be able to reproduce your results and continue to develop and modify your software without any significant delay?
  - Are you satisfied with your software development?

Note:

If you did not answer yes to all of these questions, or if you wish you could have developed your software differently, this guidebook can help you. If you did answer yes to any of these questions, we could use your expertise to further upgrade this guidebook.

Pradip Sitaram

## How the Guidebook can Help (cont.)

HUGHES
HUGHES STX CORPORATION

- Provides a quick reference guide to software engineering principles, tools, and techniques. For example:
  - Does your customer wish to know why you are using the spiral model rather than the familiar waterfall model? See the section on lifecycle process models (Section 3).
  - How do you translate a user's needs into software requirements, then into software design to be tested, documented, and turned over for operational use? See the section on the software development process (Section 4).
  - Do you need to ensure that your end products meet their requirements and that outputs fulfill the requirements established during the previous development phase? See the requirements section (Section 4.1).
  - Is your boss concerned about a lack of sophistication in methodologies you are using to manage your software project? See the section on project management (Section 5).
  - How do you identify configuration items within your system at discrete points in time? See the CM section (Section 6.1).
  - How do you ensure that your product meets or exceeds specifications? See the QA section (Section 6.2).

Pradip Sitaram

## Guidebook Organization

HUGHES
HUGHES STX CORPORATION

- Guidebook Outline
  - Background Information
  - Software Engineering Concepts
  - Lifecycle Process Models
  - Software Development Activity
  - Software Project Management Activity
  - Software Support Activity
- Common Features of each section
  - Introductory information
  - General information
  - Process Flow
  - Tailoring guide
  - Summary
  - References
  - Checklists
  - Sample Outlines

Pradip Sitaram

## Tailoring the Guidebook

HUGHES
HUGHES STX CORPORATION

- **Use the information as Guidelines (not gospel)**
- Review the drivers that are specific to your project:
  - Objective of the end-product
  - Your Customer
  - Your Project operating standards
  - Your staff and their skill mix
  - Number of people

**Use the Guidebook as a repository for information and mold whatever you need to best suit your project.**

***There is no silver bullet that will solve all the software engineering problems***

Pradip Sitaram

HUGHES
HUGHES STX CORPORATION

## Software Engineering Concepts

Pradip Sitaram

## Definitions

HUGHES
HUGHES STX CORPORATION

**Software Engineering:**

- "The application of scientific and engineering principles to the:
  - i) orderly transformation of a problem into a working solution, and,
  - ii) subsequent maintenance of that software throughout its useful life."
- "The practical application of computer science, management, and other sciences to analyze, design, construct, and maintain software and its associated documentation."
- "An engineering science that applies the concept of analysis, design, coding, testing, documentation, and management to the successful completion of a large, custom-built computer program."
- "Systematic application of methods, tools, and techniques to achieve a stated requirement or objective for an efficient software system."

Pradip Sitaram

## Overview - The Integrated View

HUGHES
HUGHES STX CORPORATION

The integrated view shows: (see Attachment 1)

- The relationship between the primary activities within the software development process: software project management activity, software development activity, and software support activity
- The various phases of the software development lifecycle and the activities performed during these phases
- The points where documentation and deliverables are (typically) produced throughout the development of the software
- The documentation process, which continues throughout the lifecycle: documentation is used to describe the product and serves as a medium of communication between the various personnel involved in the software development
- The evolution of documentation that is started in one phase as it changes during subsequent phases, until it is available for reference in later stages of development
- The points where reviews are typically held to monitor the quality of the product being developed

**Pradip Sitaram**

## Propagation of Errors

HUGHES
HUGHES STX CORPORATION

see Attachment 2

**Pradip Sitaram**

## Documentation

HUGHES
HUGHES STX CORPORATION

- The specification and design of the system must be clearly understood by the analysts, designers, management, and customers. **Because verbal descriptions are often too ambiguous or vague and are unavailable for future reference, the specification and design must be documented using text and diagrams for clarity and future reference.** A well-documented specification and design provide an excellent reference point to assess the extent of development and greatly reduce the risk of falling into the "I am 90 percent finished" syndrome.

- During the initial phases of the lifecycle, the documentation is the specification and it is the design of the system. If the documentation is bad, the design is bad. If the documentation does not exist, there is no design, only people thinking and talking about a design, which is of some value, but not much.

Pradip Sitaram

## Documentation (cont...1)

HUGHES
HUGHES STX CORPORATION

- **Requirements Specification**—The requirements specification is the communication tool between the developer and the customer. It shows the customer that the developers understand what the customer wants. The software requirements specification is then used as a management tool. By establishing a requirements baseline, managers and developers will be able to control changes by estimating impacts on cost and schedules whenever requirements are modified.

- **Testing**—Requirements can be verified and problems can be analyzed by anyone, not just the person who developed the code, thereby reducing the burden on the developers.

- **Operations**—Without good documentation, only the individuals who developed the software can effectively operate it. With clear documentation, operations personnel can operate the software cheaply and more effectively.

Pradip Sitaram

## Documentation (cont...2)

HUGHES
HUGHES STX CORPORATION

- **Maintenance**—Requests for corrections, changes, and enhancements to the software are more easily addressed when developers can refer to documentation that describes the software being modified.

- **Reusability**—Good documentation will allow developers to identify reusable software components. When good documentation is available, it is possible to modify and enhance the existing software more efficiently for use in another system (if it is not directly reusable). Without documentation, valuable time and effort are lost in trying to determine what the software does (and how it does it), often leading to the software being discarded.

- Documentation provides an **ongoing description** of the system. Document deliverables are used by managers to measure progress and to mark the transitions between lifecycle phases.

Pradip Sitaram

## Reuse

HUGHES
HUGHES STX CORPORATION

- Emphasize the principles of reuse throughout the software development lifecycle. All products generated during the software development lifecycle—requirements, design, code, documentation, and test plans—have the potential to be reused.
- Time and resources are saved in development, testing, and porting.
- Bugs are more likely to be detected (and subsequently corrected) because:
  - Systems are tested each time they are reused.
  - When a bug is detected, all systems reusing a particular component benefit.
- Code developed with reuse in mind is far more maintainable.
- Elimination of redundancies produces smaller, more manageable systems.
- HSTX Software Reuse Repository
  - **URL - http://selsvr.stx.com/Hstx/reuse**

Pradip Sitaram

## Reuse (cont...1)

HUGHES
HUGHES STX CORPORATION

- **These Activities Enable Reuse**
  - **Domain Analysis**—Identifies common requirements across the application domain and helps produce a model that describes common functions of a specific application area. This can later be tailored to accommodate specific differences.
  - **Requirements Generalization**—Covers those requirements that are intended to describe a "family" of systems or functions.
  - **Designing for Reuse**—Provides modularity, standardized interfaces, and extensible and maintainable code.
  - **Reuse Libraries**—Hold reusable source code and associated requirements, designs, documentations, and tests results. These products may be used verbatim or modified to fit the purpose.
  - **Reuse Preservation**—Ensures that changes and enhancements made during the operational phase of the software adhere to the same principles that promote reuse, i.e.., "quick fixes" may complicate future reuse.

**(see Attachment 3)**

Pradip Sitaram

## Reuse (cont...2)

HUGHES
HUGHES STX CORPORATION

- The benefits of reuse can be maximized by planning for reuse early in the development process. For example, to write reusable software, keep in mind the following guidelines:
  - Set in-line documentation standards to increase understandability of code.
  - Set naming constraints for constants, types, and functions.
  - Set usage conventions for functions governing argument order and data type.
  - Encapsulate all data structures.
  - Adhere to industry standards (ANSI, POSIX, etc.).
  - Strive for portability (to UNIXes, VMS, DOS) whenever possible).
  - See Section 4.4 for more details.

Pradip Sitaram

HUGHES
HUGHES STX CORPORATION

## Lifecycle Process Models

Pradip Sitaram

## What are Lifecycle Process Models

HUGHES
HUGHES STX CORPORATION

A Lifecycle Process Model defines:

- the expected sequences of events,
- development and management activities,
- reviews,
- products, and
- milestones

for a project.

Lifecycle Process Models serve as frameworks and provide checklists. They are developed to help, not to restrict. They need not be followed exactly; the important point is to be aware of all the available options and to understand why you are deviating from the model (if you are) it reminds you to make a conscious and informed decision.

Pradip Sitaram

## Why Use Lifecycle Process Models

HUGHES
HUGHES STX CORPORATION

- Assist in planning and provide a common frame of reference and terminology.
- Define sequences of events and phases.
- Identify the activities to be performed.
- Establish reviews to be scheduled.
- Define the interim and end products that need to be produced.
- Provide milestones in the schedule to evaluate the plan and approach.
- Provide the basis for producing the software development plan, cost estimates, and schedules.
- Encourage developers to specify what the system is supposed to do (define the requirements) before building the system.
- Encourage developers to plan how components will interact (design) before building the system.
- Enable managers to track progress more accurately and to uncover slippages early.
- Recommend that the development process generate a series of documents that can later be used to test and maintain the system.
- Reduce development and maintenance costs.
- Enable the development of a more structured and manageable system.

Pradip Sitaram

## Waterfall Model

HUGHES
HUGHES STX CORPORATION

**See Attachment 4**

- Progresses in distinct sequential phases of development
- Has gone through many refinements to deal with increasingly complex software development projects
- Most models are variations of the Waterfall model

Pradip Sitaram

## Spiral Model

HUGHES
HUGHES STX CORPORATION

**See Attachment 5**

- Activities are represented as a spiralling progression of events that moves outward from the center of the spiral.
- Each cycle proceeds through the following 4 quadrants:
  - A: determine objectives, alternatives, and constraints
  - B: Evaluate alternatives; identify and resolve risks
  - C: Develop and verify the next level product
  - D: Plan next phases
- Once detail design is complete, the spiral model proceeds thru coding and unit testing, integration testing and acceptance testing (just like the Waterfall model)
- Advantages:
  - encourages analysis of objectives, alternatives, and risks at every step - this provides an alternative to one big commitment at the start.
  - allows for objectives, to be re-evaluated and refined based on the latest perception of needs
- NOT effective if plans, objectives, & constraints cannot be changed

Pradip Sitaram

## Incremental Development Model

HUGHES
HUGHES STX CORPORATION

**see Attachment 6**

- Incremental development is the process of building software by initially constructing a part of the entire system and progressively adding functionality in successive builds.

- Because the initial capability is achieved quickly, costs normally associated with development prior to the initial release are **seemingly reduced**; these costs are actually spread across a number of builds.

- By providing operational builds of the system more quickly, the possibility that the user's requirements may change during the development of a build is also reduced; changes in requirements may also be deferred to a later build of the software.

Pradip Sitaram

## Incremental Dev. Model (cont...1)

HUGHES
HUGHES STX CORPORATION

- Note that when the incremental development model is used, the software is **intentionally** constructed to (initially) satisfy fewer requirements. However, the software is designed to facilitate the incorporation of new requirements in later builds.

- Advantages of the Incremental Development Approach
  - Initial development time is reduced (because of the reduced functionality).
  - Software can be progressively enhanced for a longer period of time (because it is designed for growth).
  - The operational date is earlier (although at limited functionality).
  - Mechanisms to address/cope with changing requirements are provided.
  - Tradeoffs of functionality and performance between versions are allowed.

Pradip Sitaram

## Incremental Dev. Model (cont...2)

HUGHES
HUGHES STX CORPORATION

- When using the incremental development process model, the **software must be designed carefully to easily support additional functionality and growth**. The functionality that is not being provided in the current build is deferred for a later build, but the plans for adding this functionality must be well thought out and analyzed.

- This approach is different from the evolutionary prototype model because the implication is that in the incremental development model the developers understand most of the requirements but are choosing to provide the functionality in subsets of increasing capability.

Pradip Sitaram

## Prototyping

HUGHES
HUGHES STX CORPORATION

- Prototyping is the technique of constructing a **partial implementation** of a system so that customers, users, or developers can **learn** more about a problem or a solution to that problem. The key word here is **partial**; if you were implementing the complete system, it would no longer be a prototype, it would be the system.
- Prototypes can be developed in the requirements, design, or coding phases of the software development lifecycle.
- Prototyping is not a euphemism for hacking, nor is prototyping an excuse to develop undocumented and unstructured code. Remember, the primary objective in developing a prototype is to learn; a completely undocumented, unstructured, and sloppy prototype will outweigh its usefulness with time wasted by developers attempting to figure out how it was constructed.

Pradip Sitaram

## Prototyping (cont.)

HUGHES
HUGHES STX CORPORATION

- Some Reasons To Develop a Prototype:
  - Demonstrate a capability either internally or to an external customer.
  - Assess a design approach or an algorithm for correctness or efficiency.
  - Evaluate the ability of a software development system to support efficient software production or to support a given number of programmers.
  - Provide a measurement vehicle when estimating user response times, recovery times, transmission times, code expansion factors, etc.
  - Validate requirements by demonstrating that they can be implemented and exploring possible error conditions that requirements must cover.
  - Clarify ambiguous requirements.
  - Provide a vehicle for soliciting end-user input, primarily on the Human-Machine Interface (HMI).
  - Form a basis for the full implementation effort.
  - Serve as an early, concrete milestone in the development schedule.
  - Demonstrate feasibility of new and evolving technology.

Pradip Sitaram

## Planning for Prototypes

HUGHES
HUGHES STX CORPORATION

- Contents of a Prototyping Plan
  - The purpose and use of the prototype
  - Brief description of the work to be done and the products to be generated
  - Technical approach
  - Completion criteria
  - Evaluation criteria and methods
  - Resources required: effort, size, staff, and hardware and software estimates
  - Schedule

**Beware:** A prototyping effort could continue indefinitely if the completion criteria and evaluation guidelines are not established.

Pradip Sitaram

## Throwaway Prototyping

HUGHES
HUGHES STX CORPORATION

**See Attachment 7**

- A throwaway prototype is constructed to learn more about the problem or its solution. **This prototype is discarded once it has been used and the requisite knowledge has been gained.**

- Design and code should be understandable to its developers for the prototype to fully serve its purpose. The throwaway prototype should be delivered quickly - there are no rigorous lifecycle phases to be followed. The advantage lies in quickly gaining additional knowledge about a certain aspect of the system so that the normal development lifecycle of the system can proceed accordingly.

- A throwaway prototype can be developed during the requirements, design, and coding phases of any of the lifecycle process models (waterfall, spiral, incremental build, evolutionary prototyping, etc.).

Pradip Sitaram

## Throwaway Prototyping (cont...1)

HUGHES
HUGHES STX CORPORATION

- During Requirements Analysis, a Prototype May Be Developed To:
  - Determine the feasibility of a requirement.
  - Validate that a particular function is really necessary.
  - Uncover missing requirements.
  - Clarify an ambiguous requirement.
  - Determine the validity of the user interface.
  - Write a preliminary SRS.
  - Implement a prototype based on a preliminary SRS.
  - Achieve user experience with the prototype.

- Beware of a common scenario that occurs when a throwaway prototype is delivered: the customers say they love the prototype and want to make it an operational system- the infamous operational prototype.

- To prevent the prototype from being used as the actual system:
  - Prototype the system in pieces (do not build an end-to-end prototype).
  - Simulate the system's interaction with data.

Pradip Sitaram

## Evolutionary Prototyping

HUGHES
HUGHES STX CORPORATION

**See Attachment 8**

- In this model, the prototype is constructed to learn more about the problem or its solution. Once the prototype has been used and the requisite knowledge has been gained, **the prototype is then adapted to satisfy the now better understood requirements.**

- Evolutionary prototypes cannot be built in a sloppy manner. Because the evolutionary prototype will finally evolve into the final product, it must demonstrate all the quality, maintainability, and reliability associated with the final product. Remember,**It is impossible to retrofit quality, maintainability, and reliability.**

Pradip Sitaram

## Evolutionary Prototyping (cont.)

HUGHES
HUGHES STX CORPORATION

- Approaches:
  - build only the parts of the system that are well understood, leaving the others to later generations of the prototype
  - lowering the importance of performance (to paraphrase Dijkstra, it is easier to make a working program faster than make a fast program work).

- If necessary, you could build a throwaway prototype during an evolutionary prototyping process, especially if it clarifies your understanding of the issues you are addressing with the evolutionary prototype.

Pradip Sitaram

## Throwaway vs. Evolutionary Prototyping

HUGHES
HUGHES STX CORPORATION

| | Throwaway Prototype | Evolutionary Prototype |
|---|---|---|
| **Development Guidelines** | "Quick and dirty", no rigor | Structured, rigorous |
| **What to build** | Build only difficult parts | Build only difficult parts first, build on solid foundation |
| **Design Drivers** | Optimized development time | Optimized modifiability |
| **Ultimate Goals** | Learn from it, and throw it away | Learn from it, and evolve it |

Pradip Sitaram

## Comparison of Lifecycle Process Models

HUGHES
HUGHES STX CORPORATION

| | Characteristics | Advantages | Disadvantages |
|---|---|---|---|
| Waterfall | • Disciplined and sequential approach<br>• Requirements need to be known at the start<br>• Document driven | • Simple model<br>• Well-defined steps | • Big commitment required up front<br>• User problems identified late |
| Spiral | • Risk reduction at every step<br>• Flexible, iterative process<br>• Supports evolving system needs | • Risks addressed, evaluated, and reduced at every step | • Difficult to implement for contract software<br>• Difficult to schedule<br>• Difficult to decide on the "number of turns of the spiral" |
| Incremental Development | • Early (initial) operational capability<br>• Software designed to facilitate growth<br>• Partial capability, with additional capability provided in subsequent builds | • Early availability of initial operational capability<br>• Software designed to be extensible<br>• Problems addressed with each build<br>• Allows tradeoffs between functionality and performance between builds | • Difficult to manage the development, testing, and release of the builds |
| Evolutionary Prototyping | • Builds the difficult parts<br>• Provides increasing capability with each release<br>• Software built to learn from, and then evolved | • Software designed to be extensible<br>• Problems addressed with each release | • Difficult to decide which requirements should be addressed with each release |

Pradip Sitaram

HUGHES
HUGHES STX CORPORATION

## Software Development Activities

Pradip Sitaram

## An Integrated View of Software Development

HUGHES
HUGHES STX CORPORATION

See Attachment 1

Pradip Sitaram

HUGHES
HUGHES STX CORPORATION

## Requirements Analysis

Pradip Sitaram

## ...a reference check

HUGHES
HUGHES STX CORPORATION

**See Attachment 9**

Pradip Sitaram

## Requirements - Definitions

HUGHES
HUGHES STX CORPORATION

- A complete, concise description of the **external behavior** of the software system, including its interfaces to its environment, other software systems, communications ports, hardware, and users.
- This description is recorded in a document called the Software Requirements Specification (SRS).
- To analyze and specify the software requirements, software developers must first analyze the current system (automated or nonautomated) and the problem(s) being addressed.
- The information required to perform this analysis is obtained from:
  - the Statement of Work (SOW)
  - operations concepts documents
  - systems requirements
  - interviews with users and customers

Pradip Sitaram

## Process flow in the Requirements phase

HUGHES
HUGHES STX CORPORATION

**See Attachment 10**

Pradip Sitaram

## Primary Functions of an SRS

HUGHES
HUGHES STX CORPORATION

- Facilitates communication among customers, users, analysts, & designers.
- Establishes the basis for the contractual agreement and provides a standard against which compliance is measured.
- Clearly defines the required functionality of the software: the software must provide all required functions (functions that are not required should not be specified).
- Reduces development costs—only the specified requirements are designed for and built. Reduces the possibility of rework by raising issues early in the development lifecycle.

Pradip Sitaram

## Primary Functions of an SRS (cont.)

HUGHES
HUGHES STX CORPORATION

- Provides the relative necessity (essential, desired, optional, TBD) and the relative volatility (confirmed, changing, unconfirmed, TBD) of the specified requirements.
- Provides basis for verifying compliance; supports system testing activities.
- Provides the foundation and helps control the evolution of the system.
- Facilitates transfer and reuse. The SRS makes it easier to transfer the knowledge about a software product to new users and machines. Potential Users can review the SRS to determine how well the system meets their needs and also gauge the software for compliance to the specified requirements.

Pradip Sitaram

## What Should be in an SRS

HUGHES
HUGHES STX CORPORATION

- A complete, concise description of the entire external interface of the software system with its environment, including other software, communication ports, hardware, and human users. This includes two types of requirements:
  - Behavioral requirements define what the software system does. All the functions to be performed, all the inputs and outputs to and from the software system, and information concerning how the inputs and outputs will interrelate are described.
  - Nonbehavioral requirements define the attributes of the software system as it performs its job. They include a complete description of the software system's required level of efficiency, reliability, security, maintainability, portability, visibility, capacity, and standards compliance.
- Software requirements should not be confused with user needs. It is the software developer's responsibility to interpret the user needs (customers often refer to these needs as requirements) and translate them into the SRS.

Pradip Sitaram

## What Should NOT be in an SRS

HUGHES
HUGHES STX CORPORATION

- Project requirements: staffing, schedules, costs, milestones, activities, phases, and reporting procedures (these belong in the software project management plan)
- Designs (these belong in the design documents)
- Product assurance plans: CM plans, Verification and Validation (V&V) plans, test plans, and QA plans

Pradip Sitaram

## Attributes of a good SRS

HUGHES
HUGHES STX CORPORATION

- **Correct**—Every requirement specified represents something that is required of the system to be built.
- **Unambiguous**—Every requirement specified has only one interpretation.
- **Complete**—Everything the software is supposed to do is included
- **Verifiable**—There is a cost-effective method to check the final software system to ensure that every requirement specified has been met (testable).
- **Consistent**—1) No two parts of any requirement should have conflicting terms, 2) no two requirements should specify the system to exhibit conflicting characteristics, and 3) no two requirements should require the system to respond to conflicting timing patterns.

Pradip Sitaram

## Attributes of a good SRS (cont.)

HUGHES
HUGHES STX CORPORATION

- **Understandable by Noncomputer Specialists**—It should serve as a communication tool between customers and developers.
- **Modifiable**—Requirements will change; the easier these changes are to make the better.
- **Traceable**—Origin of each requirement and its dependents is easily identified.
- **Annotated**—Guidance for development is provided to show the relative necessity and relative volatility of the requirements.
- **Usable**—Most importantly, the requirements should be produced in a manner that allows them to be used and to be of help to the developers.
- **Feasible**—Can this system be built?

Pradip Sitaram

## Advantages

HUGHES
HUGHES STX CORPORATION

- **Myth:** "...the requirements will change anyway, so why bother documenting them...."

  **Fact:** In the early phases of the lifecycle, the (**documented**) software requirements specifications are the requirements. If they haven't been documented, there are no requirements! Requirements must be documented from the very beginning for the very reason that they do change: this is the best way to control and manage changing requirements. The fact is that requirements will change and evolve. The best that we can do as developers is to manage and control their evolution.

Pradip Sitaram

## Advantages (cont.)

HUGHES
HUGHES STX CORPORATION

- Faulty (or unspecified) requirements will lead to errors in the system. Errors can be costly to the project, especially because errors often remain latent and are undetected until well after the stage in which they were made. The later in the development lifecycle a software error is detected, the more expensive it will be to repair. Typically, errors made in requirements specifications are because of incorrect facts, omissions, inconsistencies, and ambiguities. Using formal analysis and specification methods correctly can reduce the incidence of errors in the requirements phase.

Pradip Sitaram

## Formal Techniques

HUGHES
HUGHES STX CORPORATION

- Data Flow Diagrams (DFDs) (see Section 4.1.9.3)
- Entity Relationship Diagrams (ERDs) (see Section 4.1.9.3)
- Finite State Machines (FSMs) (see Section 4.1.9.3)
- Statecharts (see Section 4.1.9.3)
- Data Dictionaries
- Decision Tables and Decision Trees
- Object-Oriented Diagrams (OODs)
- Program Design Language (PDL)
- Requirements Engineering Validation System
- Requirements Language Processor
- Specification and Description Language
- PAISLey
- Petri Nets

Pradip Sitaram

## Tailoring to a small project

HUGHES
HUGHES STX CORPORATION

- Each project is unique. Tailoring the information provided in this section is essential in defining and implementing the requirements analysis function to a specific project.

- **Regardless of project size, the requirements analysis function needs to be performed. Only the level of detail and formality of the process and products vary among projects.**

- Some of the factors to be considered are:
  - Time
  - Resources
  - Complexity
  - Contractual commitments
  - Intended use of the product

Pradip Sitaram

## Tailoring (cont...1)

HUGHES
HUGHES STX CORPORATION

- For small projects where time and resources are very limited, it is impractical to attempt to provide a complete suite of documentation and evaluate the SRS at a formal review. However, it is essential to complete at least the following, in writing, before the software is designed:
  - Briefly describe the objective of the project and include a few statements describing the external behavior of the software; this will help you to control the scope of development.
  - List and briefly describe any constraints (standards, hardware limitations, security, availability of third-party software).
  - List and briefly describe external interfaces for (all applicable):
    - » Other software
    - » User
    - » Operators
    - » Communications
  - Identify, list, and describe the primary functional requirements being addressed by the system.

Pradip Sitaram

## Tailoring (cont...2)

HUGHES
HUGHES STX CORPORATION

- Identify and describe the data flows into and out of the system at the context level and associate the primary data flows to the primary functions. The details provided regarding data flows can be extended according to the resources available and complexity of the problem being described.

- If applicable, identify and describe the primary operating states of the system and the events that the system responds to. Again, the details regarding the description of the states and the events can be extended according to the complexity of the problem being addressed.

Pradip Sitaram

## Tailoring (cont...3)

HUGHES
HUGHES STX CORPORATION

- If a user interface is required, determine whether
  - » it is hierarchical or menubar-driven and whether it has pop-up windows.
  - » Describe events when windows are displayed
  - » describe when windows are displayed concurrently
  - » Describe what the windows look like
  - » what events they respond to
  - » what they do in response to these events

- Note: It is perfectly acceptable to *design* the user interface during the requirements phase, because you are describing **what** the interface looks and feels like (**not how** the interface accomplishes its functions). Remember, the user interface is the external interface of the software

Pradip Sitaram

## Tailoring (cont...4)

HUGHES
HUGHES STX CORPORATION

- Ensure that the issues you are specifying meet the attributes

- Briefly describe your plans (outline your test plan) to test the software after it is built to confirm that the requirements have been satisfied. **Do not specify requirements for which you cannot prescribe a test to verify compliance.**

- Remember that the objective is to specify **What** the system will do. It is essential to obtain the customer's approval on what you have written this will serve as a **common point of reference** during future development activities. The formal SSR can be replaced by an informal discussion about the requirements, culminating in agreement between the developers and the customer on the requirements that will be addressed during the development process.

Pradip Sitaram

## Checklists

HUGHES
HUGHES STX CORPORATION

- Are Requirements complete ?
- Are Requirements consistent ?
- Is implementation feasible ?
- Are Requirements testable ?
- Are Requirements understandable ?
- SRS checklist
- IRS checklist
- SSR checklist

Pradip Sitaram

## Preliminary Design

HUGHES
HUGHES STX CORPORATION

- Primary Functions of a Preliminary Design
- General Methodology for Developing a Preliminary Design
- Preliminary Design Phase Process Flow
- Items to be addressed/described for Each Software Subsystem that is identified
- Selecting a Design Methodology
- Organizing a Software Design Document
- Reviews
- Tailoring to a Small Project
- Checklists
- Sample Tables of Contents
  - Detailed Design Document (Reference: NASA-DID-P400)
  - Software Subsystem Specification (Reference: DOD-STD-1703)
  - Interface Control Document (Reference: DOD-STD-1703)
  - Software Development File (SDF)

Pradip Sitaram

## Detailed Design

HUGHES
HUGHES STX CORPORATION

- General Methodology for Developing the Detailed Design
  - For Each Subsystem of the Software System
  - For Each Database of the Software System (if any)
  - For Each Software Subsystem-to-Software Subsystem Interface Specified in the IRS
  - For Each Module Identified as Part of a Software Subsystem
- Detailed Design Phase Process Flow
- Reviews
- Summary of the Detailed Design phase
- Tailoring to a Small Project
- Suggested Reference Material
- Checklists

Pradip Sitaram

## Coding and Unit Testing

HUGHES
HUGHES STX CORPORATION

- General Methodology for Performing Coding and Unit Testing
- General Guidelines for Developing Code
  - Guidelines for Comments
  - General Guidelines
  - In-Line Comments
- Prologues
  - Function Prologues
  - File Prologues
  - Module Prologues
- Epilogues
- Banners
- Naming Conventions
- Coding Style
- General Philosophy
  - Correctness
  - Understandability
  - Modifiability
  - Reusability
  - Elegance

**Pradip Sitaram**

## Coding and Unit Testing (cont.)

HUGHES
HUGHES STX CORPORATION

- Code Structure
- Code Formatting
- Files
- Functions
- Constants
- Global Variables
- Organizing the Unit Test Documentation
- Reviews
- Summary of the Code and Unit Test Phase
- Tailoring to a Small Project
- Reference Material
- Coding Guidelines for C
- Comments
- Naming Conventions
  - General, Constants, Globals, Types, Functions, Macros
- Coding Style
- Checklists

**Pradip Sitaram**

## Integration and Testing

HUGHES
HUGHES STX CORPORATION

- General Methodology for Subsystem Integration and Testing
- Important Considerations for Subsystem Integration
- Reviews
- Summary of the Integrating and Testing phase
- Tailoring to a Small Project
- Reference Material
- Checklists
- Sample Tables of Contents for Test Plans

Pradip Sitaram

## System Testing

HUGHES
HUGHES STX CORPORATION

- General Methodology for Performing a Systems Test
- Reviews
- Summary of the Systems Testing phase
- Tailoring to a Small Project
- Reference Material
- Checklists

Pradip Sitaram

## Acceptance Testing

HUGHES
HUGHES STX CORPORATION

- General Methodology for System Acceptance Testing
- Tailoring to a Small Project
- Reference Material
- Checklists

Pradip Sitaram

## Operations and Maintenance

HUGHES
HUGHES STX CORPORATION

- Four Categories of Software Maintenance
  - Corrective Maintenance
  - Adaptive Maintenance
  - Perfective Maintenance
  - Performance Maintenance
- General Methodology for Operations and Maintenance
- Tailoring to a Small Project
- Reference Material
- Checklists
- Sample Tables of Contents

Pradip Sitaram

## Software Project Management Activities

HUGHES
HUGHES STX CORPORATION

- Software Project Management Planning (5.1)
- Software Development Planning (5.2)
- Software Cost Estimating (5.3)
- Software Metrics (5.4)
- Scheduling and Tracking (5.5)
- Risk Management (5.6)

- Do's for Project Success (5.7)
- Don'ts for Project Success (5.8)
- Danger Signals and Corrective Measures (5.9)

Pradip Sitaram

## Project Do's...

HUGHES
HUGHES STX CORPORATION

- Use a small senior staff for the early lifecycle phases.
- Develop and adhere to an SDP.
- Write down the SRS.
- Define specific intermediate and end products.
- Examine alternative approaches.
- Perform risk analysis.
- Conduct formal and informal reviews with customers and users.
- Use a defined testing process.
- Use a central repository.
- Keep a detailed list of TBD items.
- Update system size, required effort, cost, and schedule estimates.
- Allocate sufficient time for testing and integration.
- Experiment.

Pradip Sitaram

## Project Don'ts...

HUGHES
HUGHES STX CORPORATION

- Don't overstaff.
- Don't allow an undisciplined development approach.
- Don't delegate technical details to team members.
- Don't assume that a rigid set of project-specific standards and guidelines ensures success.
- Don't assume that a large set of documentation ensures success.
- Don't deviate from the approved design.
- Don't assume that relaxing project-specific standards and guidelines will reduce costs.
- Don't assume that the pace will increase later in the project.
- Don't assume that schedule slippage can be absorbed in later phases.
- Don't assume that introducing new tools will reduce the schedule.
- Don't assume that everything will fit together smoothly at the end.

Pradip Sitaram

## Danger Signals

HUGHES
HUGHES STX CORPORATION

- Scheduled capabilities are delayed to a later build/release.
- Coding is started too early (staff is too large too early).
- Numerous changes are made to the initial SDP.
- Guidelines or planned procedures are de-emphasized or deleted.
- Sudden changes in staffing (magnitude) are suggested and/or made.
- Excessive (irrelevant) documentation and paperwork is being prepared.
- There is a continual increase in the number of TBD items and ECRs.
- A decrease in estimated effort for system testing is suggested and/or made.
- There is reliance on other sources for "soon-to-be-available" software.

Pradip Sitaram

## Corrective Measures

HUGHES
HUGHES STX CORPORATION

- Stop current activities and review the problem activity.
- Decrease staff to a manageable level.
- Assign a senior staff member to assist junior personnel.
- Increase and tighten management procedures.
- Increase the number of intermediate deliverables.
- Decrease the scope of work and define a manageable thread of the system.
- Audit the project with independent personnel and act on their findings.

Pradip Sitaram

## Software Support Activities (1) Configuration Management

HUGHES
HUGHES STX CORPORATION

- Software Configuration Management (6.1)
- Main Functions of SCM
- Configuration Identification
- Functional, Allocated, and Product Baselines
- Configuration Control
- Build Control
- Configuration Status Accounting
- Configuration Auditing
- Phase-Independent SCM
- Continuous Identification of Configuration Items
- Software Development Library
- Configuration Control Board
- Phase-Dependent SCM
- SCM Tools
- Tailoring to a Small Project
- Sample Tables of Contents

Pradip Sitaram

## Software Support Activities (2)
## Quality Assurance

HUGHES
HUGHES STX CORPORATION

- Software Quality Assurance (6.2)
- The specific goals of QA
- Evaluation of:
  - Corrective Action Process
  - Software Plans
  - Software Management Activities
  - Software Configuration Management Activities
  - Software Engineering Activities
  - Software Testing and Qualification Activities
  - Software Development Library
  - Software Storage, Handling and Delivery
  - Software Media and Documentation Distribution
  - Subcontract Management
  - Software Documentation
  - Software
- Phase-Dependent Quality Evaluations
- Tailoring to a Small Project
- Sample Tables of Contents

Pradip Sitaram

## Review

HUGHES
HUGHES STX CORPORATION

- Background and historical information
- Organizational entities
- Terminology
- High-level overview of the Guidebook
- First step in Software Engineering Training
  - Requirements Engineering
- Overview of the other phases of development
- Overview of Software Project Management activities
- Overview of Software Support activities

Pradip Sitaram

## Feedback

HUGHES
HUGHES STX CORPORATION

- Presentation evaluation
- Comments on the Guidebook

Pradip Sitaram

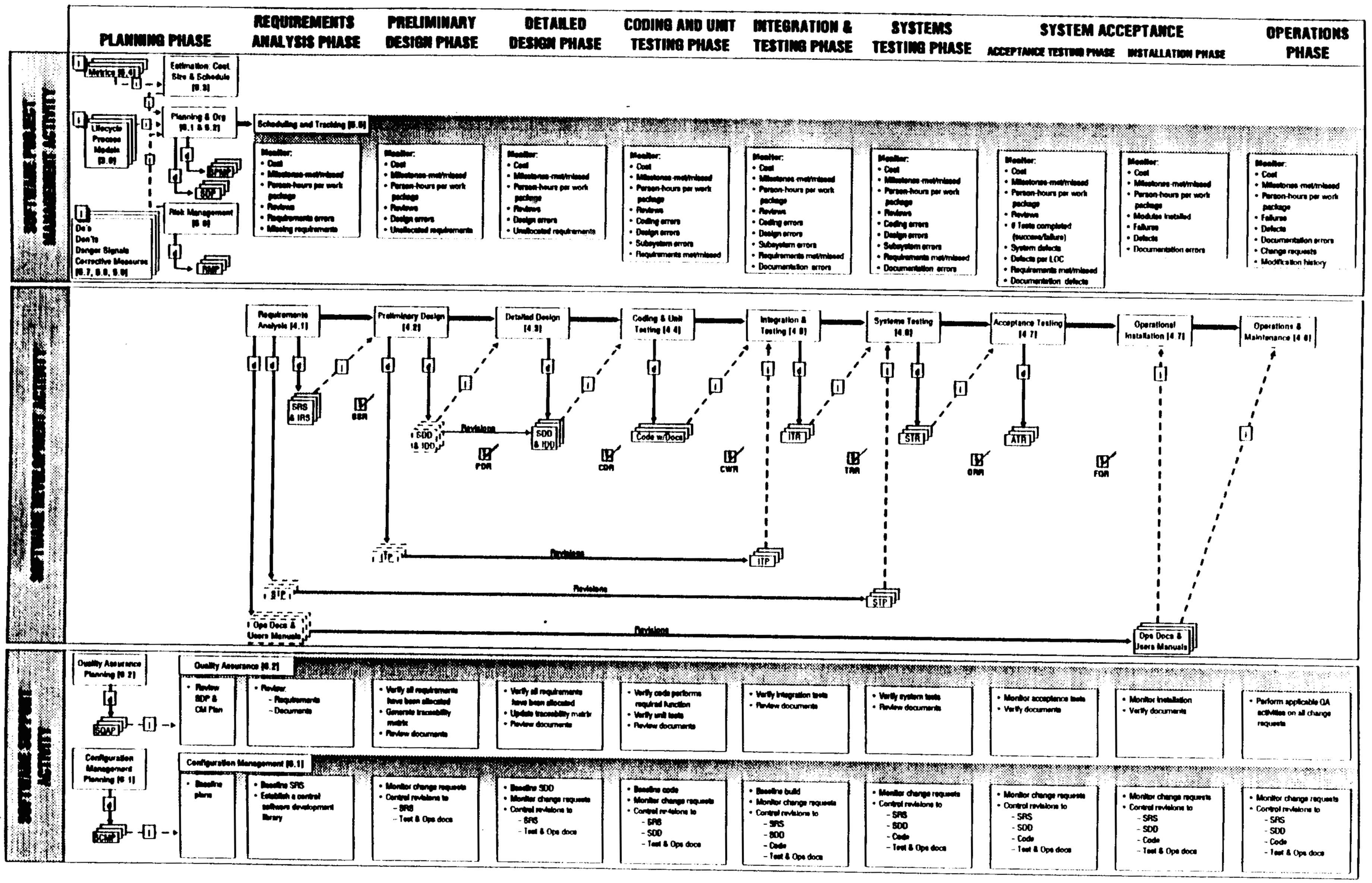
**Introduction**
**to the**
**Software Engineering**
**Guidebook**

# Attachments

Pradip Sitaram
Hughes STX Corporation

September 14, 1995

HUGHES
HUGHES STX CORPORATION


SOFTWARE PROJECT MANAGEMENT ACTIVITY
SOFTWARE DEVELOPMENT ACTIVITY
SOFTWARE SUPPORT ACTIVITY
PLANNING PHASE
REQUIREMENTS ANALYSIS PHASE
PRELIMINARY DESIGN PHASE
DETAILED DESIGN PHASE
CODING AND UNIT TESTING PHASE
INTEGRATION & TESTING PHASE
SYSTEMS TESTING PHASE
SYSTEM ACCEPTANCE
ACCEPTANCE TESTING PHASE
INSTALLATION PHASE
OPERATIONS PHASE
Metrics [6.4]
Estimation: Cost, Size & Schedule [6.3]
Lifecycle Process Models [3.0]
Planning & Org [6.1 & 6.2]
SPMP
SDP
Risk Management [6.6]
RMP
Do's
Don'ts
Danger Signals
Corrective Measures [6.7, 6.8, 9.0]
Scheduling and Tracking [6.5]
Monitor:
• Cost
• Milestones met/missed
• Person-hours per work package
• Reviews
• Requirements errors
• Missing requirements
• Design errors
• Unallocated requirements
• Coding errors
• Subsystem errors
• Requirements met/missed
• Documentation errors
• # Tests completed (success/failure)
• System defects
• Defects per LOC
• Documentation defects
• Modules installed
• Failures
• Defects
• Change requests
• Modification history
Requirements Analysis [4.1]
Preliminary Design [4.2]
Detailed Design [4.3]
Coding & Unit Testing [4.4]
Integration & Testing [4.5]
Systems Testing [4.6]
Acceptance Testing [4.7]
Operational Installation [4.7]
Operations & Maintenance [4.8]
SRS & IRS
SSR
SDD & IDD
PDR
CDR
Code w/Docs
CWR
ITR
TRR
STR
ORR
ATR
FQR
ITP
STP
Ops Docs & Users Manuals
Revisions
Quality Assurance Planning [8.2]
SQAP
Quality Assurance [8.2]
• Review SDP & CM Plan
• Review: - Requirements - Documents
• Verify all requirements have been allocated
• Generate traceability matrix
• Review documents
• Update traceability matrix
• Verify code performs required function
• Verify unit tests
• Verify integration tests
• Verify system tests
• Monitor acceptance tests
• Verify documents
• Monitor installation
• Perform applicable QA activities on all change requests
Configuration Management Planning [8.1]
SCMP
Configuration Management [8.1]
• Baseline plans
• Baseline SRS
• Establish a control software development library
• Monitor change requests
• Control revisions to - SRS - SDD - Code - Test & Ops docs
• Baseline SDD
• Baseline code
• Baseline build

HUGHES
HUGHES STX CORPORATION

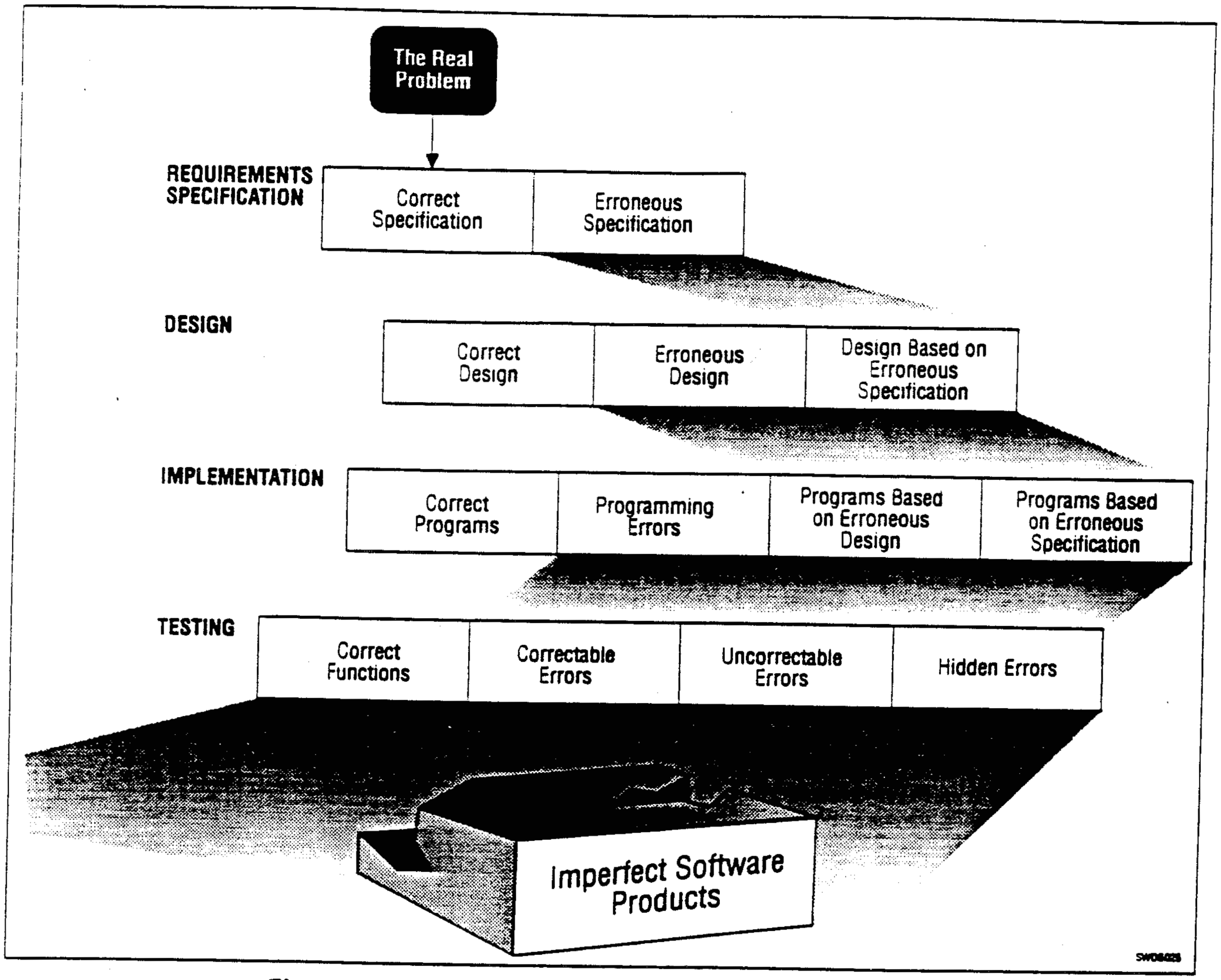


**Figure 2.3-1. The Cumulative Effects of Errors, © IEEE 1983**

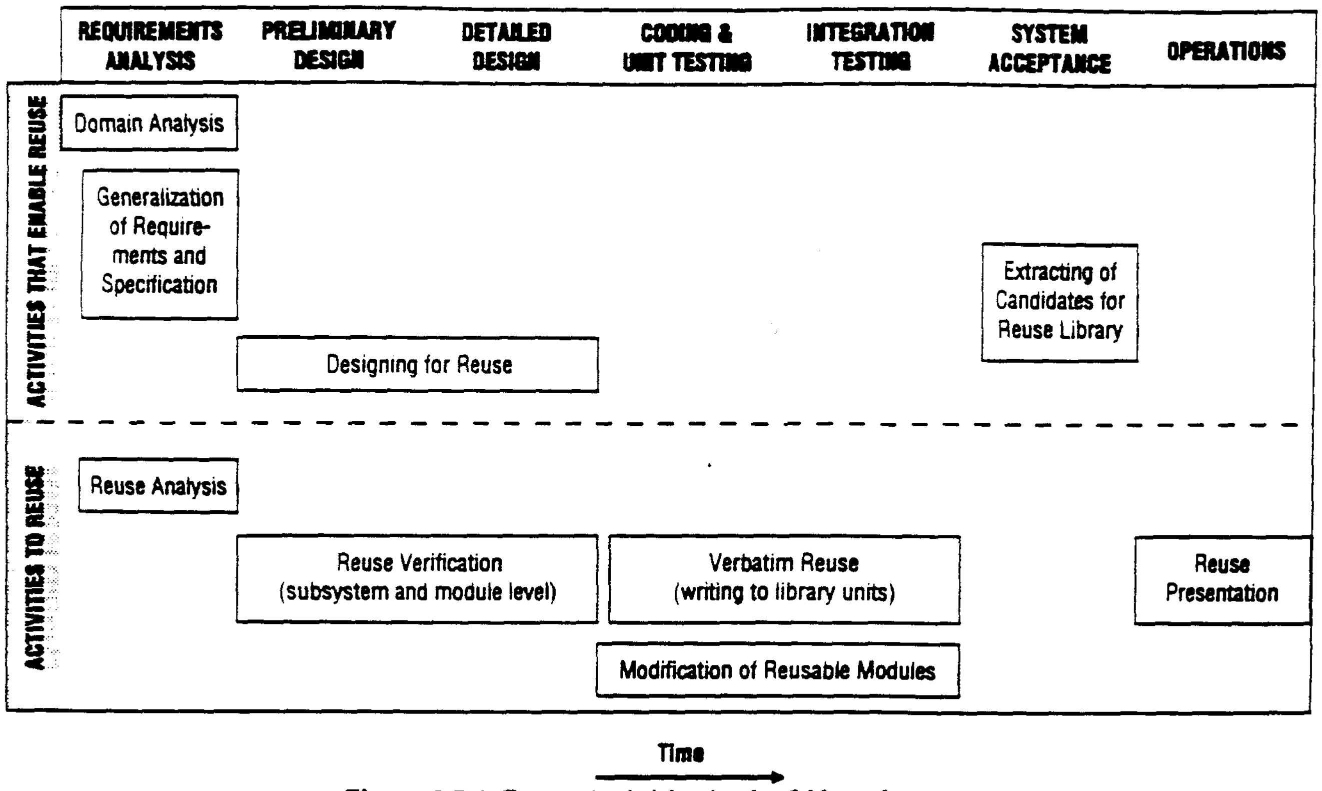

REQUIREMENTS ANALYSIS
PRELIMINARY DESIGN
DETAILED DESIGN
CODING & UNIT TESTING
INTEGRATION TESTING
SYSTEM ACCEPTANCE
OPERATIONS
ACTIVITIES THAT ENABLE REUSE
Domain Analysis
Generalization of Require-ments and Specification
Designing for Reuse
Extracting of Candidates for Reuse Library
ACTIVITIES TO REUSE
Reuse Analysis
Reuse Verification (subsystem and module level)
Verbatim Reuse (writing to library units)
Reuse Presentation
Modification of Reusable Modules
Time

**Figure 2.5-1. Reuse Activities in the Lifecycles**

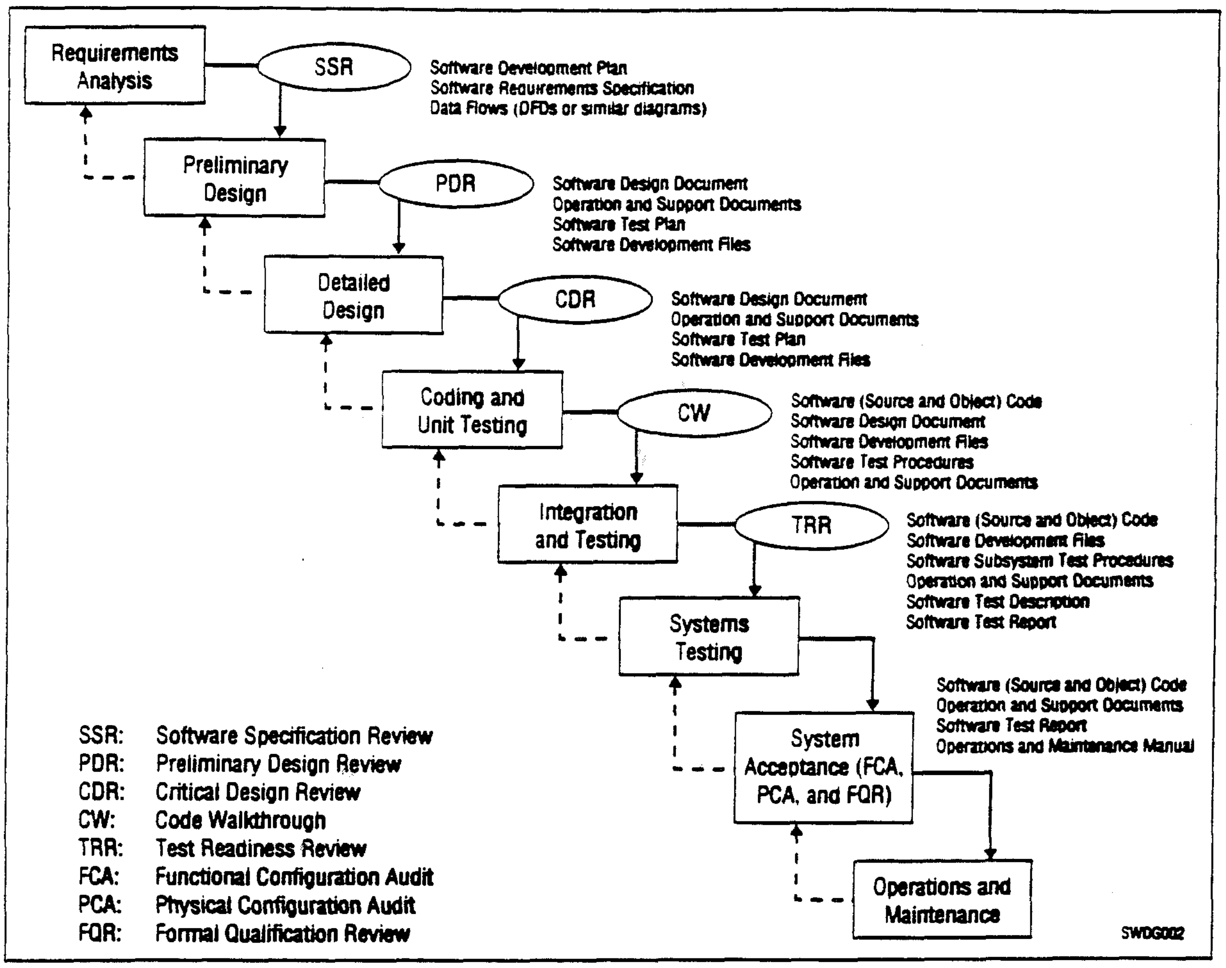

Requirements Analysis
SSR
Software Development Plan
Software Requirements Specification
Data Flows (DFDs or similar diagrams)
Preliminary Design
PDR
Software Design Document
Operation and Support Documents
Software Test Plan
Software Development Files
Detailed Design
CDR
Software Design Document
Operation and Support Documents
Software Test Plan
Software Development Files
Coding and Unit Testing
CW
Software (Source and Object) Code
Software Design Document
Software Development Files
Software Test Procedures
Operation and Support Documents
Integration and Testing
TRR
Software (Source and Object) Code
Software Development Files
Software Subsystem Test Procedures
Operation and Support Documents
Software Test Description
Software Test Report
Systems Testing
Software (Source and Object) Code
Operation and Support Documents
Software Test Report
Operations and Maintenance Manual
System Acceptance (FCA, PCA, and FQR)
Operations and Maintenance
SSR: Software Specification Review
PDR: Preliminary Design Review
CDR: Critical Design Review
CW: Code Walkthrough
TRR: Test Readiness Review
FCA: Functional Configuration Audit
PCA: Physical Configuration Audit
FQR: Formal Qualification Review
SWDG002

**Figure 3.2-1. The Waterfall Model—Phases, Reviews, and Major Products**

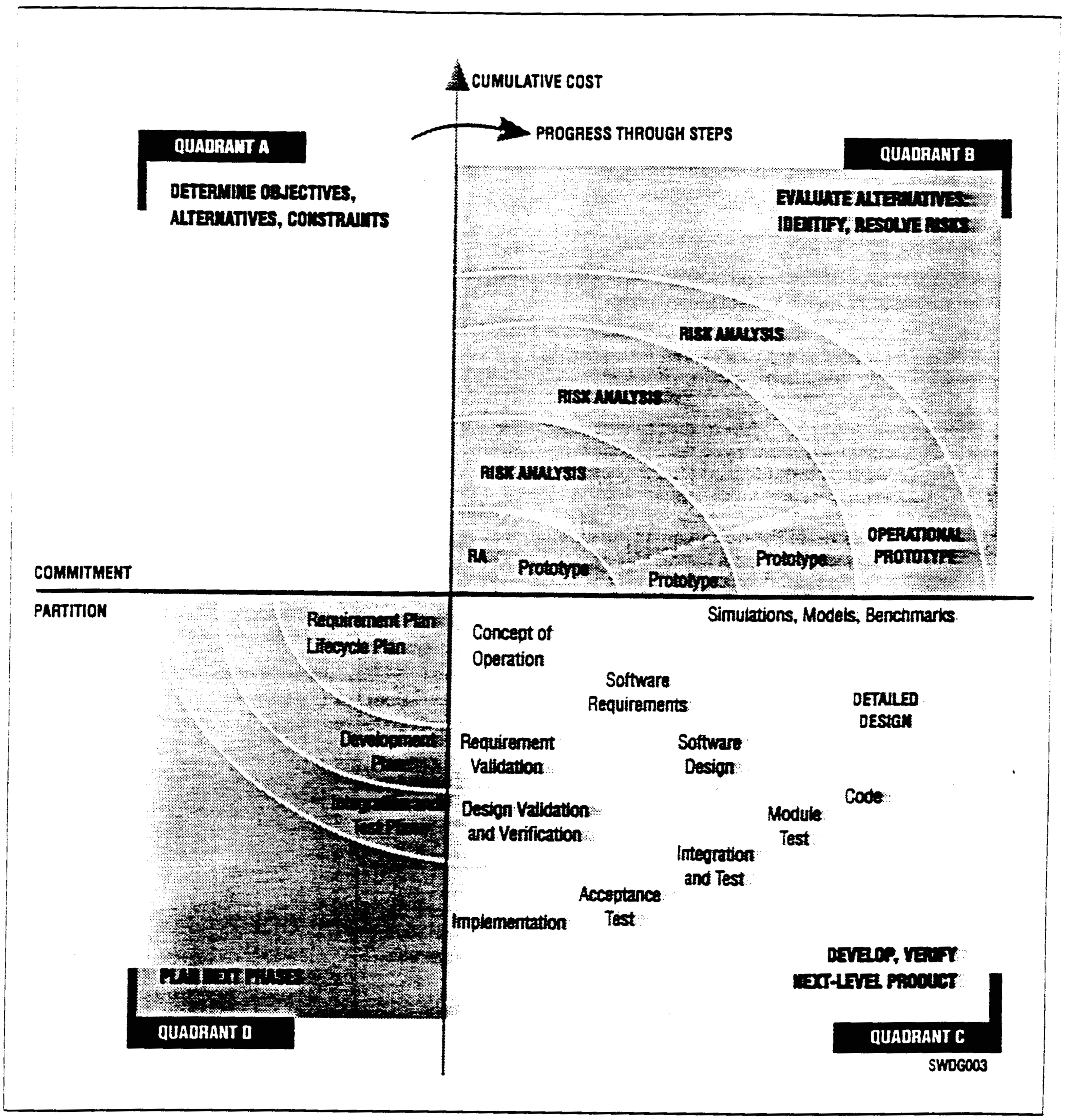


Figure 3.3-1. Spiral Model of the Software Process

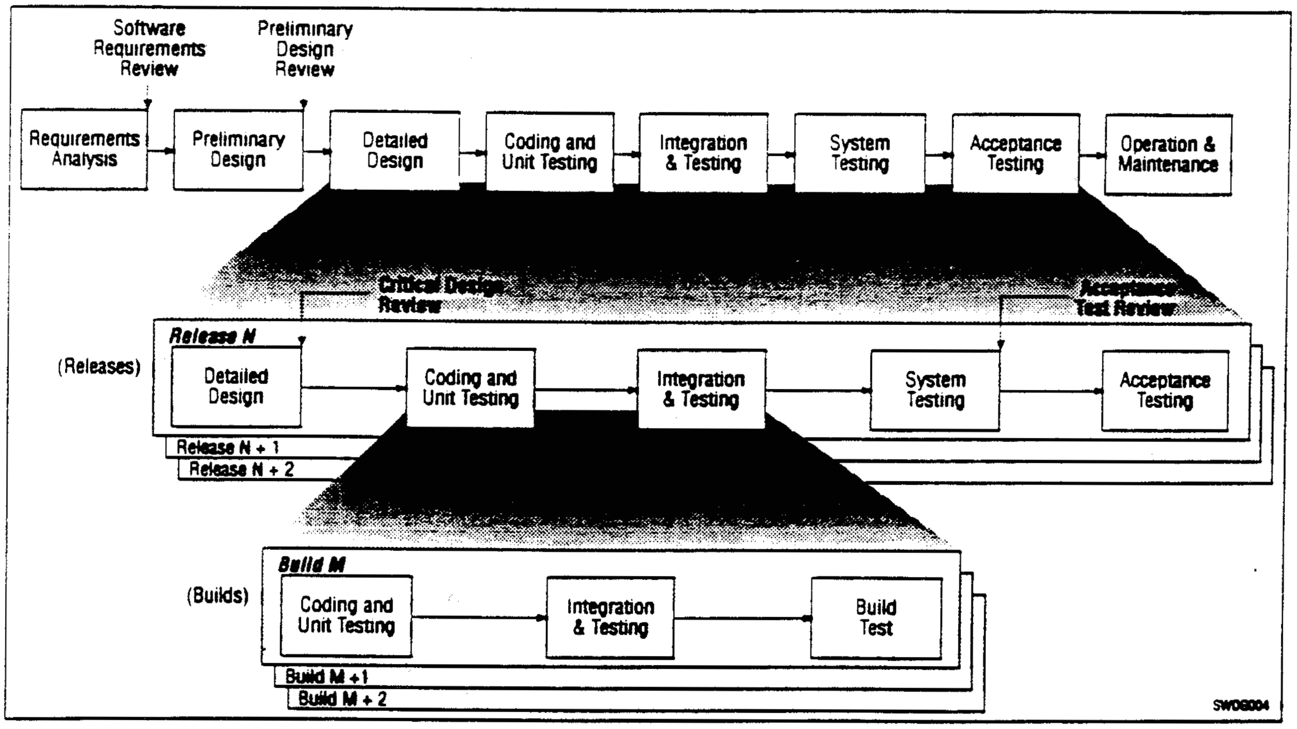


**Figure 3.4-1. Incremental Development Model**

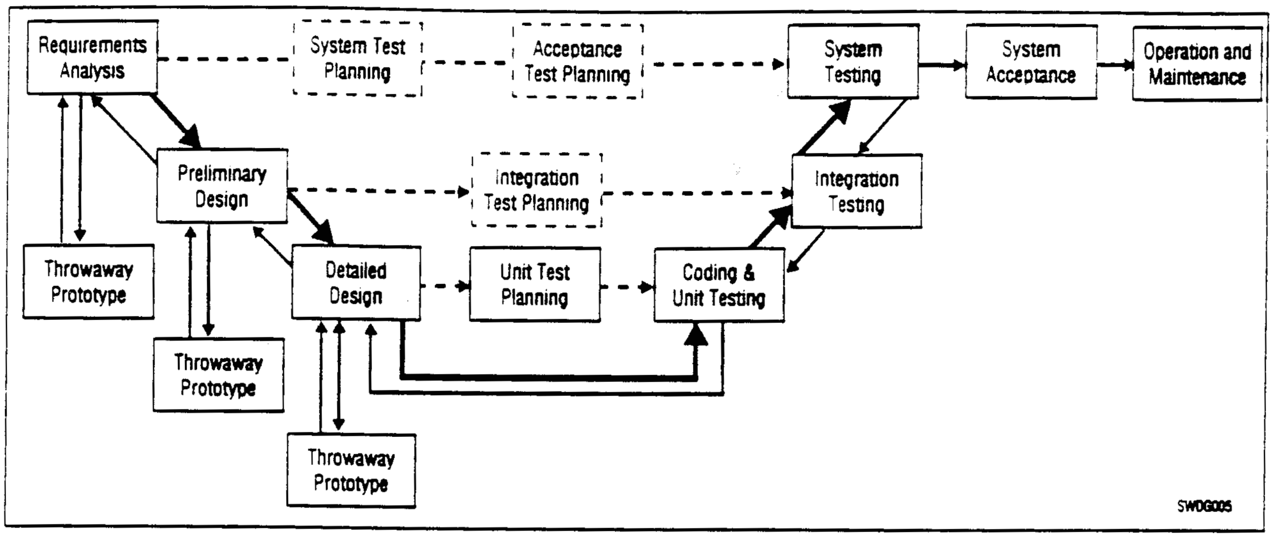


**Figure 3.5.2-1. Throwaway Prototyping During Requirements Analysis, Preliminary Design, and Detailed Design**

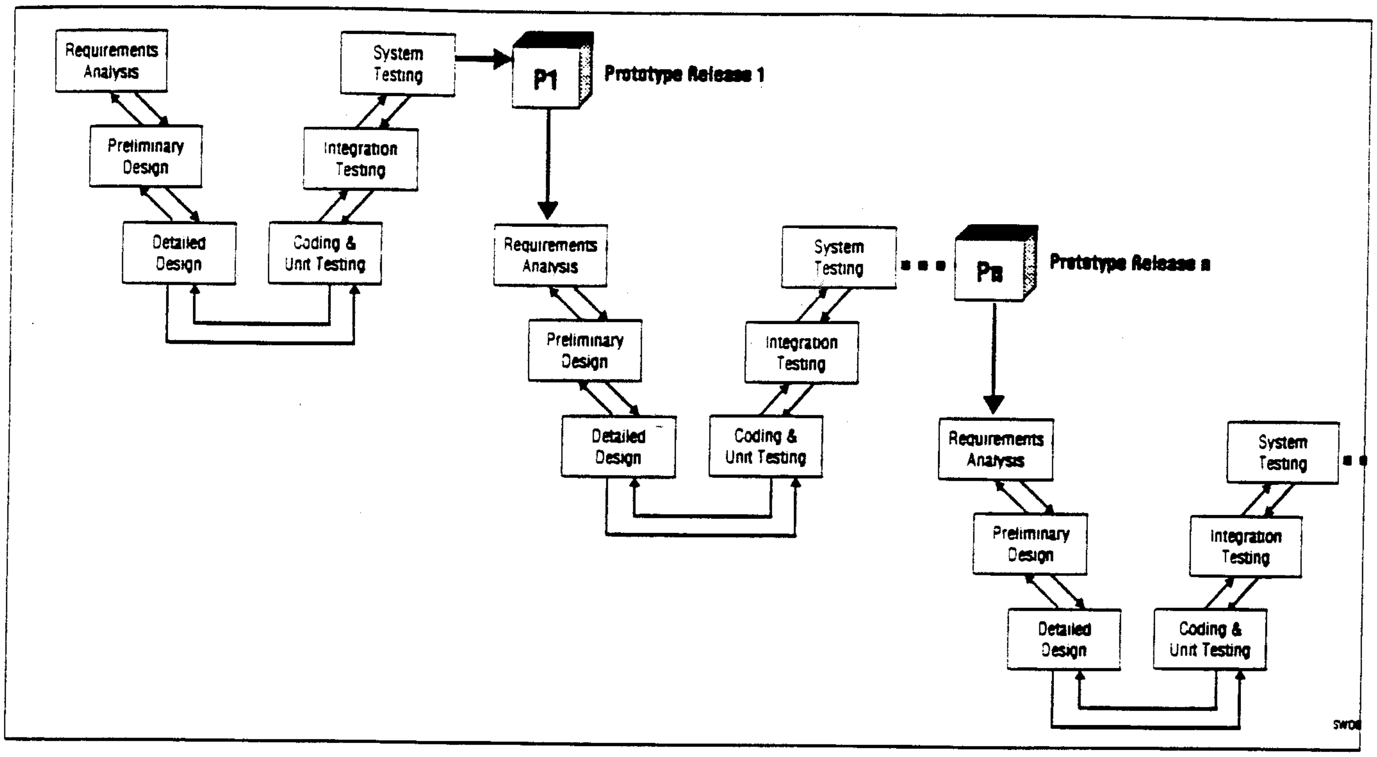
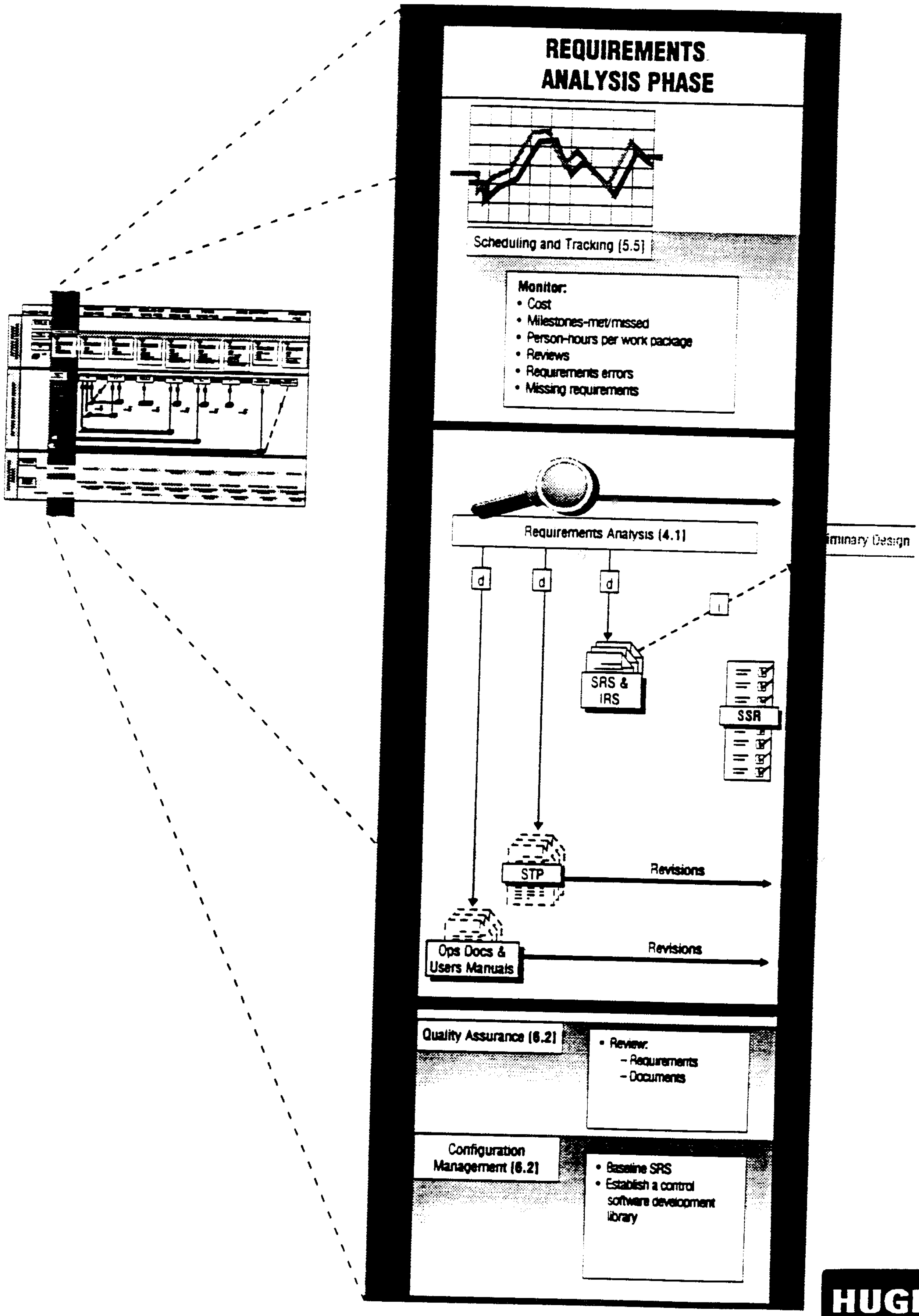


Figure 3.5.3-1. The Evolutionary Prototype Model

HUGHES

HUGHES STX CORPORATION


REQUIREMENTS ANALYSIS PHASE
Scheduling and Tracking [5.5]
Monitor:
• Cost
• Milestones-met/missed
• Person-hours per work package
• Reviews
• Requirements errors
• Missing requirements
Requirements Analysis [4.1]
iminary Design
d
d
d
i
SRS & IRS
SSR
STP
Revisions
Ops Docs & Users Manuals
Revisions
Quality Assurance [6.2]
• Review:
– Requirements
– Documents
Configuration Management [6.2]
• Baseline SRS
• Establish a control software development library

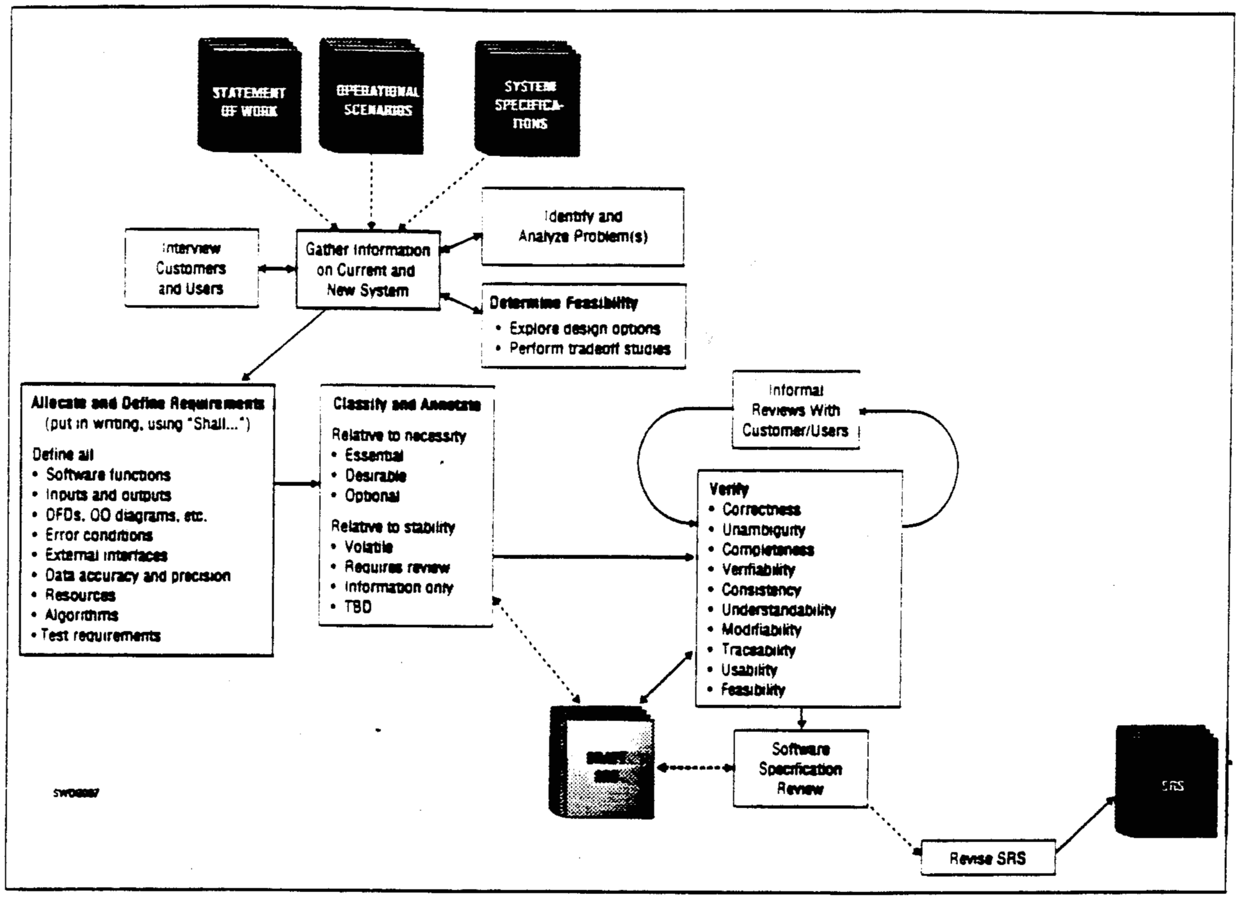
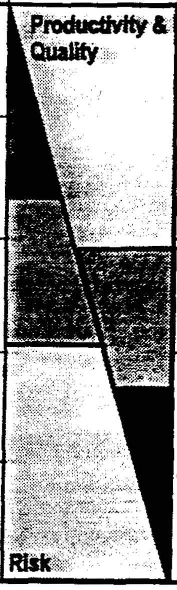


**Figure 4.1.1-1. Requirements Phase Process Flow**

| Level | Characteristic | Key Challenges | Result | People Implications |
|---|---|---|---|---|
| 5<br>**Optimizing** | Improvement fed back into process | • Still human intensive process<br>• Maintain organization at optimizing level |  | • Focus on "Fire prevention"<br>• Highly developed sense of teamwork and sense of interdependencies |
| 4<br>**Managed** | (Quantitative)<br>Measured process | • Changing Technology<br>• Problem Analysis<br>• Problem prevention | | |
| 3<br>**Defined** | (Qualitative)<br>Process defined and Institutionalized | • Process Measurement<br>• Process Analysis<br>• Quantitative Quality problems | | • Transitional state; traces of both low and high maturity<br>• Increased defined process;less ad hoc<br>• Less critical event driven |
| 2<br>**Repeatable** | (Intuitive)<br>Process dependant on individuals | • Training, Tech. Practices (reviews, testing)<br>• Process focus (Stds, process groups) | | • Focus on "Fire fighting"<br>• Effectiveness low/ frustration high<br>• Adversarial relationship across disciplines<br>• "Caste" system |
| 1<br>**Initial** | ad hoc / chaotic | • Project Management<br>• Project Planning<br>• CM<br>• Software QA | | |

HUGHES STX CORPORATION

HSTX Software Engineering Guidebook
Control ID: SES-016

HUGHES
HUGHES STX CORPORATION

# SOFTWARE ENGINEERING GUIDEBOOK

REQUIREMENTS ANALYSIS & SPECIFICATION
PRELIMINARY DESIGN
DETAILED DESIGN
PROTOTYPING
CODING & UNIT TESTING
INTEGRATION & TESTING
SYSTEMS TESTING
OPERATIONS & MAINTENANCE
PROJECT MANAGEMENT
CONFIGURATION MANAGEMENT
QUALITY ASSURANCE
ESTIMATING
LIFECYCLE PROCESS MODELS
REVIEWS
METRICS
REUSE
PLANNING
SCHEDULING

The Hughes STX Software Engineering Guidebook Development Team:

Pradip Sitaram
Guidebook Development Team Lead
Tel #: (301) 441-4184
email: sitaram@selsvr.stx.com

Temp Johnson
Software Excellence Initiative (SWEI) Chairperson
Tel #: (301) 441-4171
email: tjohnson@ccmail.stx.com

Lee Bodden
Dave Niver
Sherry Paquin
Carl Solomon
Mark Solomon

Other contributing writers: Babu Balakrishna, Erik Dorfman, Al Dwyer, Bob Harberts, Jerry Klein, Teresa Larson, Len Olsen, Chris Lynnes, and Ed Wilson.

The Publications Resource Center:

Karen Poxon, Editor
Dannette Lawrence, Format and Layout
John Hazen, Graphics
Ron Bretemps, Reprographics

And, all the other members of the Publications Resource Center who assisted with the development of this guidebook.

# Acknowledgments

This Software Engineering Guidebook is the result of the dedicated effort and support of many individuals throughout Hughes STX Corporation (HSTX). It originated out of a grass-roots effort combined with a corporate vision and desire to better understand and improve the processes and products of our software engineering efforts. With the continued support of HSTX management, current and past members of the HSTX Software Engineering Process Group (SEPG), and other HSTX personnel, this guidebook will continue to evolve and improve our software engineering process.

The SEPG would like to thank Ashok Kaveeshwar, Naren Bewtra, Dick Bishop, Ron Estes, Kit Harvel, Rick Payne, Mirco Snidero, and Dick Tighe for their invaluable support in providing resources (personnel, time, and hardware), all of which made the development of this guidebook possible.

This version was enhanced and improved through the reviews, and valuable comments from the numerous software developers, managers, and software support personnel across HSTX.

This version also benefited greatly from the information in the Hughes, Division 48 (now HITC) *Software Engineering Handbook.* Other major sources of information included NASA GSFC's Software Engineering Laboratory documents, ANSI/IEEE documents and standards, and DoD documents and standards.

# Preface

As part of the HSTX Software Excellence Initiative (SWEI), the Software Engineering Process Group (SEPG) is chartered to train and guide software developers in software engineering principles. The objectives of the SWEI closely correspond with those of our implementation of continuous measurable improvement (*cmi*). The software engineering process is one of three factors affecting software quality; the other two are people and technology. In this guidebook, we will emphasize the *process;* our premise is that the quality of a software system depends upon the quality of the process used to develop the software. By improving our software development process, we will improve the quality of our software.

*cmi* suggests that the quality of our software can be measured in terms of how well it helps our clients accomplish their missions. Our success depends on our ability to help our customers understand how they will benefit from the latest technology in software development. Our challenge, then, is to harness our knowledge of software development methodologies and techniques and mold them to suit our customers' environments. The key to success lies in our ability to be creative and flexible while applying tried and proven methodologies to meet our customers' needs. Quality software products are best attained through proven and repeatable processes. This guidebook is intended to provide a framework from which software developers can meet this challenge.

The purpose of the HSTX Software Engineering Guidebook is to enhance the quality of our software systems and increase the productivity of those individuals responsible for designing, developing, maintaining, and managing these systems. The SEPG has developed this guidebook to:

- Foster an overall, company-wide understanding of software engineering principles.
- Present proven software development processes in a software engineering reference that can serve as an initial foundation for HSTX software engineering training and in support of proposal activities.
- Foster a common engineering perspective with which to plan, develop, implement, maintain, manage, review, and improve HSTX software processes.
- Provide an integrated approach to software engineering activities encompassing software development and maintenance, software support (i.e., Quality Assurance [QA] and Configuration Management [CM]), and software management.
- Provide software engineering information (offering lifecycle models, development methodologies, checklists, and tailoring guidelines) in a concise, easy-to-update format that is practical and tailorable to every HSTX software project or task.

# Table of Contents

Section Page

## Table of Contents (Continued)

# List of Illustrations

# List of Tables

*Section 1*

# Introduction

## Contents

Hughes STX Corporation (HSTX) often conducts business from the challenging position of developing large, complex software systems in a customer environment under tight deadlines and with frequently changing or under-specified requirements. Creating a quality product under these conditions requires considerable planning and careful management. A systematic, structured software development process that is tailored to each project or task will lead to a high quality software product.

Undefined and ad-hoc software development practices often cause problems such as software that does not meet requirements, is unreliable, is difficult to maintain, cannot be reused, and has inadequate documentation and a project that is over budget, difficult to track developmentally, and unable to meet deadlines. By following tried and proven software engineering principles, these problems are significantly reduced, in these cases by the early establishment of requirements, more effective scheduling, and ample documentation. Quality software is thoroughly documented and produces repeatable results, enabling subsequent users to fully understand its design, structure, and operation and have confidence in its products.

## 1.1 Purpose

The purpose of this Software Engineering Guidebook is to identify key aspects of the software development process, such as lifecycle models, development phases, development activities and methodologies, prototyping, tools, Configuration Management (CM), Quality Assurance (QA), and software project management. It offers a collection of methodologies and approaches that HSTX software developers can use as a reference to successfully meet the needs of their customers.

This guidebook is intended to be a living document that will evolve over time; as additional information becomes available regarding effective and successful approaches and methodologies, it will be added to the guidebook. Feedback from software developers, managers, and support staff applying the methodologies and techniques presented in this guidebook will be incorporated into future editions, thus providing a forum for sharing effective software engineering techniques among software developers. In addition, as more and more HSTX staff routinely use this guidebook, it will provide the basis for a commonality to our software engineering approaches.

A primary goal of the HSTX Software Excellence Initiative (SWEI) is the continuous measurable improvement (*cmi*) of our software development process. The following point must be addressed for us to achieve this goal:

- A process must first be manageable before it can be improved in an orderly and sustained manner. A software process is manageable when it is:
  - **Defined and Documented**—Inputs, outputs, work activities, and responsibilities are outlined and delimited.
  - **Measured**—Inputs, outputs, work activities, and resources are measured to provide a basis for control and improvement.
  - **Controlled**—A predetermined mechanism exists to maintain a process in its desired state.
  - **Continuously Improved and Optimized**—A predetermined mechanism exists to improve and optimize the process. Software process management cycles through the following stages:
    - Process definition
    - Measurement and feedback from use
    - Evaluation leading to improvement and optimization.

This guidebook addresses the first three steps leading to a manageable software process; i.e., defining and documenting the best-suited HSTX software development processes, measuring software development progress, and maintaining a process in its desired state. To achieve the fourth step, *cmi* of the software development process, you must *share* your experiences so that future editions of the guidebook can include this information to benefit subsequent software development.

## 1.2 Intended Audience

This guidebook is intended for programmers, analysts, engineers, managers, and software support (QA and CM) staff working in the field of software development. Its objective is to provide a better understanding of the software development process to a wide cross-section of HSTX.

The guidebook includes descriptions of time-tested procedures and methodologies that can be selected and used effectively for a project of any size. For managers, this guidebook presents the information necessary to manage a software development project.

## 1.3 How This Guidebook Can Help You

To be successful, a software development project must be delivered on time and within cost, and it must meet the customer's specified requirements. For this to happen, the management functions of planning, organizing, estimating, monitoring, and controlling must be understood and applied correctly. An important part of this guidebook is the project management section (Section 5), which addresses these concepts and provides useful procedures for their implementation.

The guidebook begins with an overview of the lifecycle process models. Choosing an appropriate model is crucial to the success of a software project; it sets the baseline from which progress will be measured. Selecting the model up front with the customer can prevent misunderstandings during the course of the project.

Do you know what your software is doing and what it should do? Does your customer know what the software will do? Do you know what conditions will cause your software to fail? Will successors to your project be able to reproduce your results and continue to develop and modify your software without any significant delay? Are you satisfied with your software development?

If you did not answer *yes* to all of these questions, or if you wish you could have developed your software differently, this guidebook can help you. If you did answer *yes* to any of these questions, we could use your expertise to further upgrade this guidebook.

To help you understand how to develop software through the entire lifecycle, this guidebook will provide you with a quick reference guide to software engineering principles, tools, and techniques. This is a resource from which selections can be taken and modified as needed. For example:

- Does your customer wish to know why you are using the spiral model rather than the familiar waterfall model? See the section on lifecycle process models (Section 3).
- How do you translate a user's needs into software requirements, then into software design to be tested, documented, and turned over for operational use? See the section on the software development process (Section 4).
- Do you need to ensure that your end products meet their requirements and that outputs fulfill the requirements established during the previous development phase? See the requirements section (Section 4.1).
- Is your boss concerned about a lack of sophistication in methodologies you are using to manage your software project? See the section on project management (Section 5).

- How do you identify configuration items within your system at discrete points in time? See the CM section (Section 6.1).
- How do you ensure that your product meets or exceeds specifications? See the QA section (Section 6.2).

Section 2, Introduction to Software Engineering, provides a broad overview of the various facets of software engineering. This section has a diagram that can be used as a road-map to navigate through the guidebook. Each section in this book begins with introductory information, followed by detailed subject material that is compiled from various sources for easy reference. Each section ends with guidelines on tailoring the presented material to a smaller project and references to books and publications that contain further details on the subjects covered in this document.

## 1.4 Tailoring This Guidebook

This guidebook has already been tailored to the HSTX environment, and its information will require further tailoring to make it relevant to specific projects. The quality of a project's software system depends greatly on how the selected process is tailored to that project.

Remember that this guidebook is intended to provide useful and helpful guidelines. Its objective is to aid software development professionals within HSTX and make them aware of the various options available to them during all phases of software development. You are not being directed to follow *all* the methodologies presented here—*use this book as a repository for information and mold whatever you need to best suit your project*. Remember: There is no silver bullet that will solve all of your software engineering problems.

## Section 2

# Introduction to Software Engineering

# Contents

## 2.1 Definitions

**Software Engineering:**

- "The application of scientific and engineering principles to the:

  i) orderly transformation of a problem into a working solution, and,
  ii) subsequent maintenance of that software throughout its useful life."[*DAV90*]

- "The practical application of computer science, management, and other sciences to analyze, design, construct, and maintain software and its associated documentation."[*THA87*]

- "An engineering science that applies the concept of analysis, design, coding, testing, documentation, and management to the successful completion of a large, custom-built computer program."[*THA87*]

- "Systematic application of methods, tools, and techniques to achieve a stated requirement or objective for an efficient software system."[*THA87*]

## 2.2 Introduction

The software engineering discipline is a complex network of management, engineering and development, and support and control functions. To understand and apply the various software engineering functions effectively, we must first understand how all these functions relate to each other.

Figure 2.2-1 presents the entire software development process and shows:

- The relationship between the primary activities within the software development process: software project management activity, software development activity, and software support activity
- The various phases of the software development lifecycle and the activities performed during these phases
- The points where documentation and deliverables are (typically) produced throughout the development of the software
- The documentation process, which continues throughout the lifecycle; documentation is used to describe the product and serves as a medium of communication between the various personnel involved in the software development
- The evolution of documentation that is started in one phase as it changes during subsequent phases, until it is available for reference in later stages of development
- The points where reviews are typically held to monitor the quality of the product being developed

Rather than trying to digest this diagram all at once, it is recommended that you refer back to it often as you read the other sections in this guidebook. This diagram will help you to identify points of reference in the lifecycle as you proceed through the guidebook. You can also use it to locate topics in the lifecycle that are of interest to you; each major entity is referenced to its corresponding section number in the guidebook. Each major subsection (development phase) in Section 4 begins with a "zoomed-in" view of the part of the diagram that is related to that particular phase.

Figure 2.2-1 shows an integrated view of software project management, software development, and software support activity:

- Software project management activity comprises:
  - Planning and organization (Sections 5.1 and 5.2)
  - Estimation: cost, size, and schedule (Section 5.3)
  - Collecting Software Metrics (Section 5.4)
  - Scheduling and tracking (Section 5.5)
  - Risk management (Section 5.6)
- Software development activity comprises:
  - Requirements analysis and specification (Section 4.1)
  - Preliminary design (Section 4.2)
  - Detailed design (Section 4.3)
  - Coding and module testing (Section 4.4)
  - Integration and testing (Section 4.5)
  - Systems testing (Section 4.6)
  - System acceptance (Section 4.7)
  - Operations and maintenance (Section 4.8)
- Software support activity comprises:
  - Configuration Management (CM) (Section 6.1)
  - Quality Assurance (QA) (Section 6.2)

The software development process depicted in Figure 2.2-1 is based on a waterfall model (see Section 3). This should not be taken to mean that this guidebook recommends using the waterfall model over the others. The primary purpose of the diagram is to show all the interrelationships between the activities—for illustrative purposes, this graphical representation happens to resemble the waterfall model. Because most other process models are variations of the waterfall model, a similar relationship will exist between the primary activities as is depicted in this diagram.

Figure 2.2-1 presents a number of deliverables (documentation or otherwise) throughout the lifecycle. It is important to understand that a deliverable does not necessarily mean a product that is due to the customer—a deliverable is a product created during a particular phase of development that is necessary for other software developers to perform their duties. For example, even though a customer does not ask for a software requirements document, the software requirements document is still a deliverable; it will be used by the software designers to design the software, project managers to monitor and control the evolution of requirements, and testers to test the software.

Developing a quality software product is not easy. Current software development practices often produce software that does not do what the customer had expected and is over budget, unreliable, and extremely difficult to maintain. Most of the problems plaguing software developers stem from improper software development practices. Often, the software being produced is a complex product. The process for creating a complex, quality product should be well thought out and managed.

For instance, consider the process of constructing a building and its similarity with developing software. For a building to be constructed, a number of things must happen:

- An architect meets with the client to learn why the building is required and what it will be used for, who it will be used by, the number of people it will be used by, the times of the day it will be occupied, etc. (identifies the requirements).

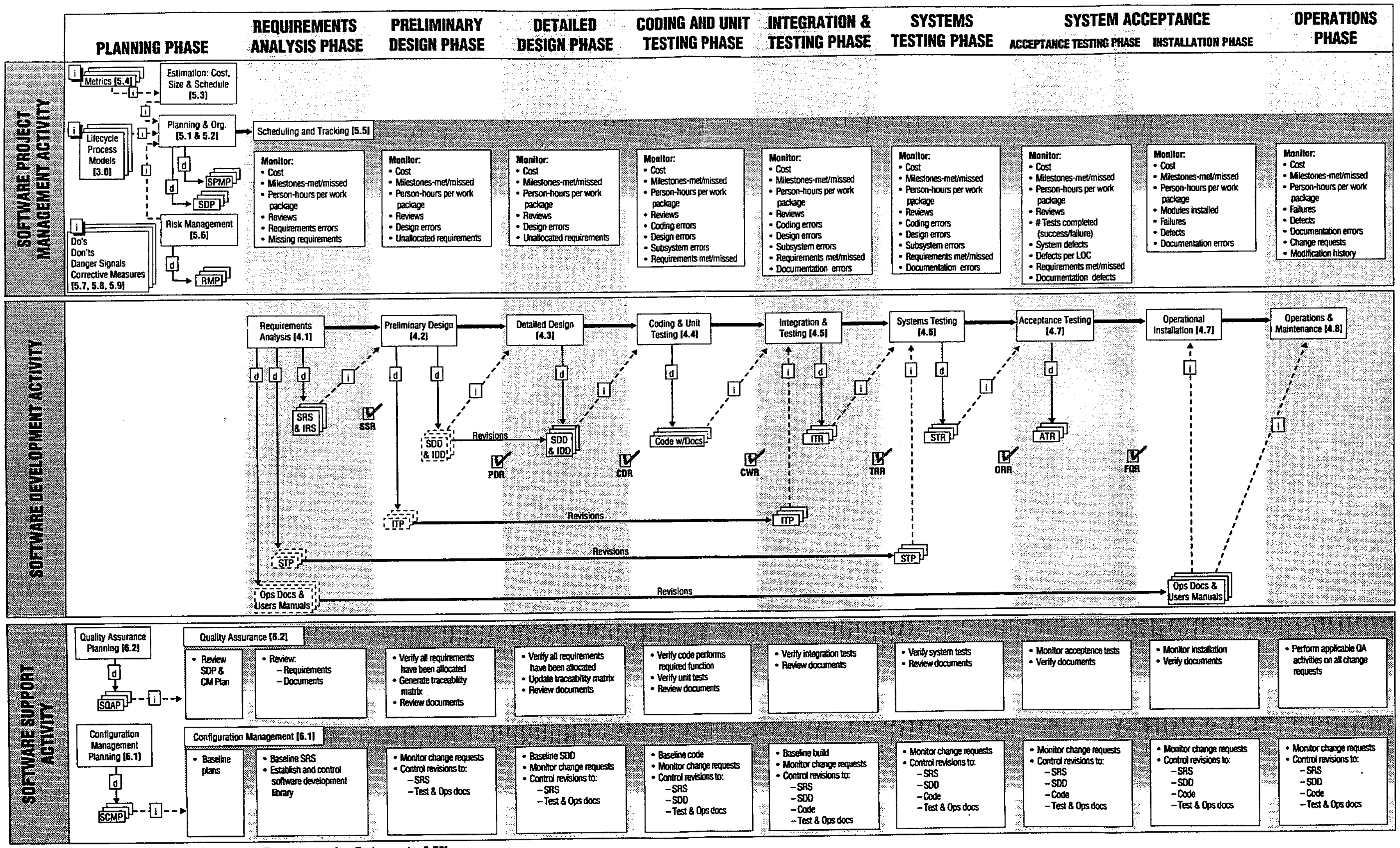

PLANNING PHASE
REQUIREMENTS ANALYSIS PHASE
PRELIMINARY DESIGN PHASE
DETAILED DESIGN PHASE
CODING AND UNIT TESTING PHASE
INTEGRATION & TESTING PHASE
SYSTEMS TESTING PHASE
SYSTEM ACCEPTANCE
ACCEPTANCE TESTING PHASE
INSTALLATION PHASE
OPERATIONS PHASE
SOFTWARE PROJECT MANAGEMENT ACTIVITY
Metrics [5.4]
Estimation: Cost, Size & Schedule [5.3]
Lifecycle Process Models [3.0]
Planning & Org. [5.1 & 5.2]
SPMP
SDP
Do's Don'ts Danger Signals Corrective Measures [5.7, 5.8, 5.9]
Risk Management [5.6]
RMP
Scheduling and Tracking [5.5]
Monitor: • Cost • Milestones-met/missed • Person-hours per work package • Reviews • Requirements errors • Missing requirements
Monitor: • Cost • Milestones-met/missed • Person-hours per work package • Reviews • Design errors • Unallocated requirements
Monitor: • Cost • Milestones-met/missed • Person-hours per work package • Reviews • Design errors • Unallocated requirements
Monitor: • Cost • Milestones-met/missed • Person-hours per work package • Reviews • Coding errors • Design errors • Subsystem errors • Requirements met/missed
Monitor: • Cost • Milestones-met/missed • Person-hours per work package • Reviews • Coding errors • Design errors • Subsystem errors • Requirements met/missed • Documentation errors
Monitor: • Cost • Milestones-met/missed • Person-hours per work package • Reviews • Coding errors • Design errors • Subsystem errors • Requirements met/missed • Documentation errors
Monitor: • Cost • Milestones-met/missed • Person-hours per work package • Reviews • # Tests completed (success/failure) • System defects • Defects per LOC • Requirements met/missed • Documentation defects
Monitor: • Cost • Milestones-met/missed • Person-hours per work package • Modules installed • Failures • Defects • Documentation errors
Monitor: • Cost • Milestones-met/missed • Person-hours per work package • Failures • Defects • Documentation errors • Change requests • Modification history
SOFTWARE DEVELOPMENT ACTIVITY
Requirements Analysis [4.1]
Preliminary Design [4.2]
Detailed Design [4.3]
Coding & Unit Testing [4.4]
Integration & Testing [4.5]
Systems Testing [4.6]
Acceptance Testing [4.7]
Operational Installation [4.7]
Operations & Maintenance [4.8]
SRS & IRS
SSR
SDD & IDD
Revisions
SDD & IDD
PDR
Code w/Docs
CDR
CWR
ITR
TRR
STR
ORR
ATR
FQR
ITP
Revisions
ITP
STP
Revisions
STP
Ops Docs & Users Manuals
Revisions
Ops Docs & Users Manuals
d
i
SOFTWARE SUPPORT ACTIVITY
Quality Assurance Planning [6.2]
SOAP
Quality Assurance [6.2]
• Review SDP & CM Plan
• Review: – Requirements – Documents
• Verify all requirements have been allocated • Generate traceability matrix • Review documents
• Verify all requirements have been allocated • Update traceability matrix • Review documents
• Verify code performs required function • Verify unit tests • Review documents
• Verify integration tests • Review documents
• Verify system tests • Review documents
• Monitor acceptence tests • Verify documents
• Monitor installation • Verify documents
• Perform applicable QA activities on all change requests
Configuration Management Planning [6.1]
SCMP
Configuration Management [6.1]
• Baseline plans
• Baseline SRS • Establish and control software development library
• Monitor change requests • Control revisions to: – SRS – Test & Ops docs
• Baseline SDD • Monitor change requests • Control revisions to: – SRS – Test & Ops docs
• Baseline code • Monitor change requests • Control revisions to: – SRS – SDD – Test & Ops docs
• Baseline build • Monitor change requests • Control revisions to: – SRS – SDD – Code – Test & Ops docs
• Monitor change requests • Control revisions to: – SRS – SDD – Code – Test & Ops docs
• Monitor change requests • Control revisions to: – SRS – SDD – Code – Test & Ops docs
• Monitor change requests • Control revisions to: – SRS – SDD – Code – Test & Ops docs
• Monitor change requests • Control revisions to: – SRS – SDD – Code – Test & Ops docs

**Figure 2.2-1 Software Development Process—An Integrated View**

- The architect develops conceptual drawings of the building that show the client what the building will look like from the outside, along with sketches of the interior.
- The architect may then build a scale model of the building (prototype) being proposed to further clarify ideas presented in the conceptual drawings, to identify requirements that may have been missed, and to check the feasibility of constructing the building.
- When the architect and the client agree on the description of the building being proposed, the architect will generate detailed blueprints (design) and specifications for the building contractor to construct the building.
- In accordance with the blueprints and specifications, the contractor prepares the foundation and starts construction of the building.

At every phase in the construction, the architect and contractor use well-established and accepted techniques. They follow well-defined standards while developing the blueprints and during construction. The objective is to be able to clearly communicate what needs to be done to all the various subcontractors involved in the construction of the building. It is evident that each of these steps is well defined and needs to occur for a successful building construction project; failure to communicate and improper construction practices will cause serious harm and will not meet the client's requirements. There is no reason why the software development process should not be as structured as the building construction process. The software development process should also progress systematically in phases:

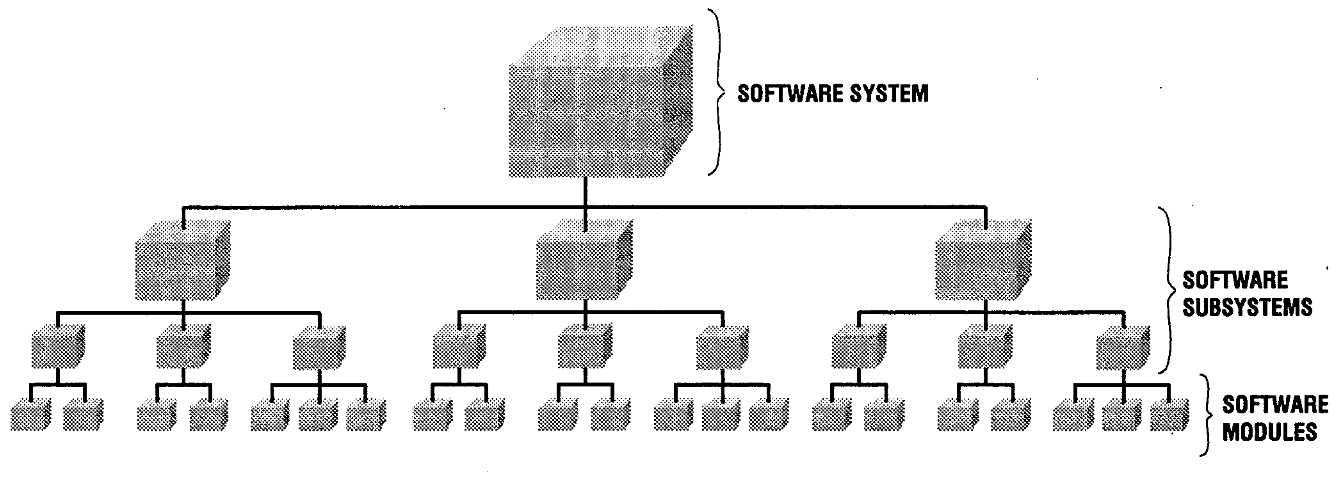
- Analyze and understand the problem.
- Define *what* the software is required to do to solve the problem.
- Describe *how* the software will do *what* is required to solve the problem.
- *Program* the modules to do *what* the software requires to solve the problem.
- *Test* the modules to verify that the software does *what* is required to solve the problem.

Throughout this guidebook we will be referring to various parts of the software using the following terminology: 1) software system, 2) software subsystem, and 3) software module. Figure 2.2-2 is a graphical representation of these terms; keep the definitions of these terms in mind as you read the rest of this book.

## 2.3 The Propagation of Errors

All developers go through the different phases in the software development process with varying degrees of formality at each phase. Products and activities can be evaluated at every phase of the development process. The risk of errors propagating through the products of the later phases diminishes when developers spend the additional effort at each phase to ensure the correctness of the product of that phase. In the real world, errors occur; the ability to catch these errors (and rectify them) early in the development process is the key to developing a successful product.

The activities and products of each phase depend upon the actions taken in the previous phase(s); it is clear that we must define *what* the software will do (software requirements) before we can attempt to describe *how* the software will be built (design). Errors made during the development process can have a major impact on the quality and cost of the software. Following a well-planned, well-defined, and structured software development process will minimize the possibility of errors and control the quality of the software product. Because errors created in one phase are inherited by the next phase, decisions made in later phases may be based on erroneous products of previous phases. Thus, the effects of the errors are magnified in later phases of development. *The later in the development process an error is detected, the more costly it is to fix.* Figure 2.3-1 demonstrates the cumulative effects of errors during the process of software development.



**SOFTWARE SYSTEM:** A "chunk" of software that is separately specified, tested, and delivered. Among other things, this means that each software system has its own requirements specification and test (in DoD terminology this is a Computer Software Configuration Item [CSCI]).

**SOFTWARE SUBSYSTEM:** A software subsystem consists of one or more modules that are logically or functionally related. Software subsystems can be composed of other software subsystems, and there can be several levels of software subsystems between the software system and software module levels. Because of the diversity of design methods and their corresponding design entities, there are no universal criteria for identifying a software subsystem. For example, a software subsystem may be an independent area in a structured design, a package or task in an Ada PDL design, one or more objects in an object-oriented design, or a process in a data flow diagram. In general, however, the software subsystem is the basis for the preliminary design version of the software design document (in DoD terminology this is a Computer Software Component [CSC]).

**SOFTWARE MODULE:** A software module is the lowest level design entity that is implemented in code. Software subsystems are made up of a set of modules that are related to each other in some way (e.g., control relationship in a structure chart or task graph). Software modules may be shared among software subsystems (in DoD terminology this is a Computer Software Unit [CSU]).

SWDG019

**Figure 2.2-2. Software Component Terminology**

## 2.4 Documentation

It is necessary to document your software development activities as part of the software development process. This guidebook will provide you with guidelines for documentation at the various phases of development.

**Why Document?**

The specification and design of the system must be clearly understood by the analysts, designers, management, and customers. Because verbal descriptions are often too ambiguous or vague and are unavailable for future reference, the specification and design must be documented using text and diagrams for clarity and future reference. A well-documented specification and design provide an excellent reference point to assess the extent of development and greatly reduce the risk of falling into the "I am 90 percent finished" syndrome.

During the initial phases of the lifecycle, the documentation *is* the specification and it *is* the design of the system. If the documentation is bad, the design is bad. If the documentation does not exist, there is no design, only people thinking and talking about a design, which is of some value, but not much.

[*ROY87*]

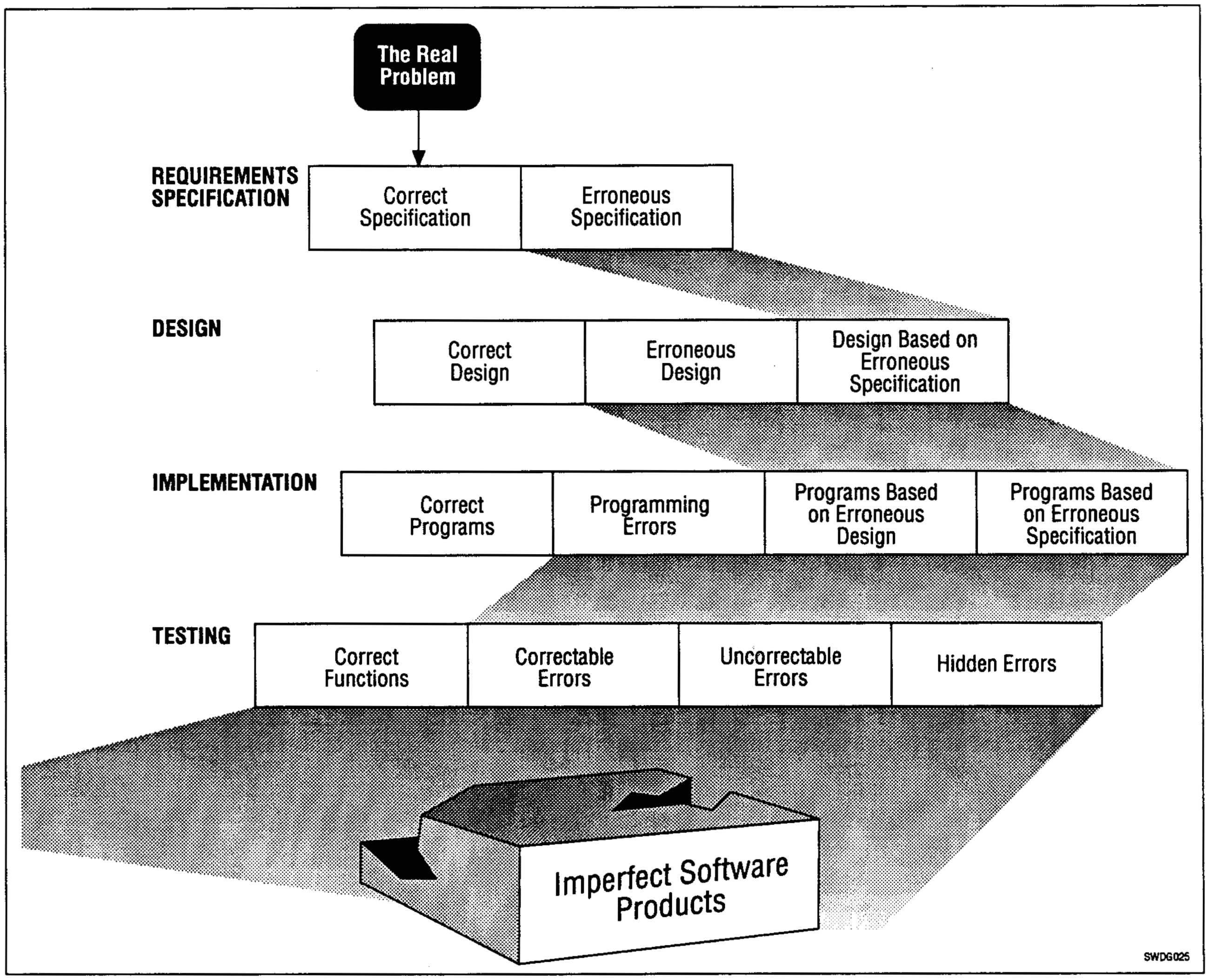


**Figure 2.3-1. The Cumulative Effects of Errors, © IEEE 1983** [*DAV90*]

**Why Document? (Continued)**

The value of good documentation becomes apparent in the following:

- **Requirements Specification**—The requirements specification is the communication tool between the developer and the customer. It shows the customer that the developers understand what the customer wants. The software requirements specification is then used as a management tool. By establishing a requirements baseline, managers and developers will be able to control changes by estimating impacts on cost and schedules whenever requirements are modified.
- **Testing**—Requirements can be verified and problems can be analyzed by anyone, not just the person who developed the code, thereby reducing the burden on the developers.
- **Operations**—Without good documentation, only the individuals who developed the software can effectively operate it. With clear documentation, operations personnel can operate the software cheaply and more effectively.

**Why Document? (Continued)**

- **Maintenance**—Requests for corrections, changes, and enhancements to the software are more easily addressed when developers can refer to documentation that describes the software being modified.
- **Reusability**—Good documentation will allow developers to identify reusable software components. When good documentation is available, it is possible to modify and enhance the existing software more efficiently for use in another system (if it is not directly reusable). Without documentation, valuable time and effort are lost in trying to determine what the software does (and how it does it), often leading to the software being discarded.

Documentation provides an ongoing description of the system. Document deliverables are used by managers to measure progress and to mark the transitions between lifecycle phases.

As software developers, it is our duty to inform and educate our customers of the inherent value in the timely (and usable) documentation of the development activities and products. We should never allow ourselves to fall into the trap of thinking "...we'll document the system after it has been built..."; *the documentation will never get done after the system has been built.* Many developers and customers wrongly assume that by not documenting during development, they are saving development time. Though it might seem as though considerable progress is being made initially, the development will fall apart when changes need to be made and the software must be redesigned and maintained. This is especially true when you consider our work environment of rapidly changing, loosely defined requirements and extremely tight deadlines—good documentation is one of the keys to project success. A favorite argument of anti-documentation proponents is that the software requirements are going to change anyway, so there is no point in wasting time on documentation. *The fact that the requirements change is precisely why they must be documented.*

## 2.5 Reusability

It is important to emphasize the principles of reuse throughout the software development lifecycle. The reuse of existing experience is the principal ingredient for success in any field. Without the ability to reuse, everything must be relearned and rebuilt from scratch. "Reinventing the wheel" in every aspect of software development can be a costly, unreliable, and unproductive venture. All products generated during the software development lifecyle—requirements, design, code, documentation, and test plans—have the potential to be reused. Figure 2.5-1 illustrates reuse activities within the software development lifecycle.

**Why Reuse Software?**

- Time and resources are saved in development, testing, and porting.
- Bugs are more likely to be detected (and subsequently corrected) because:
  - Systems are tested each time they are reused.
  - When a bug is detected, all systems reusing a particular component benefit.
- Code developed with reuse in mind is far more maintainable.
- Elimination of redundancies produces smaller, more manageable systems.

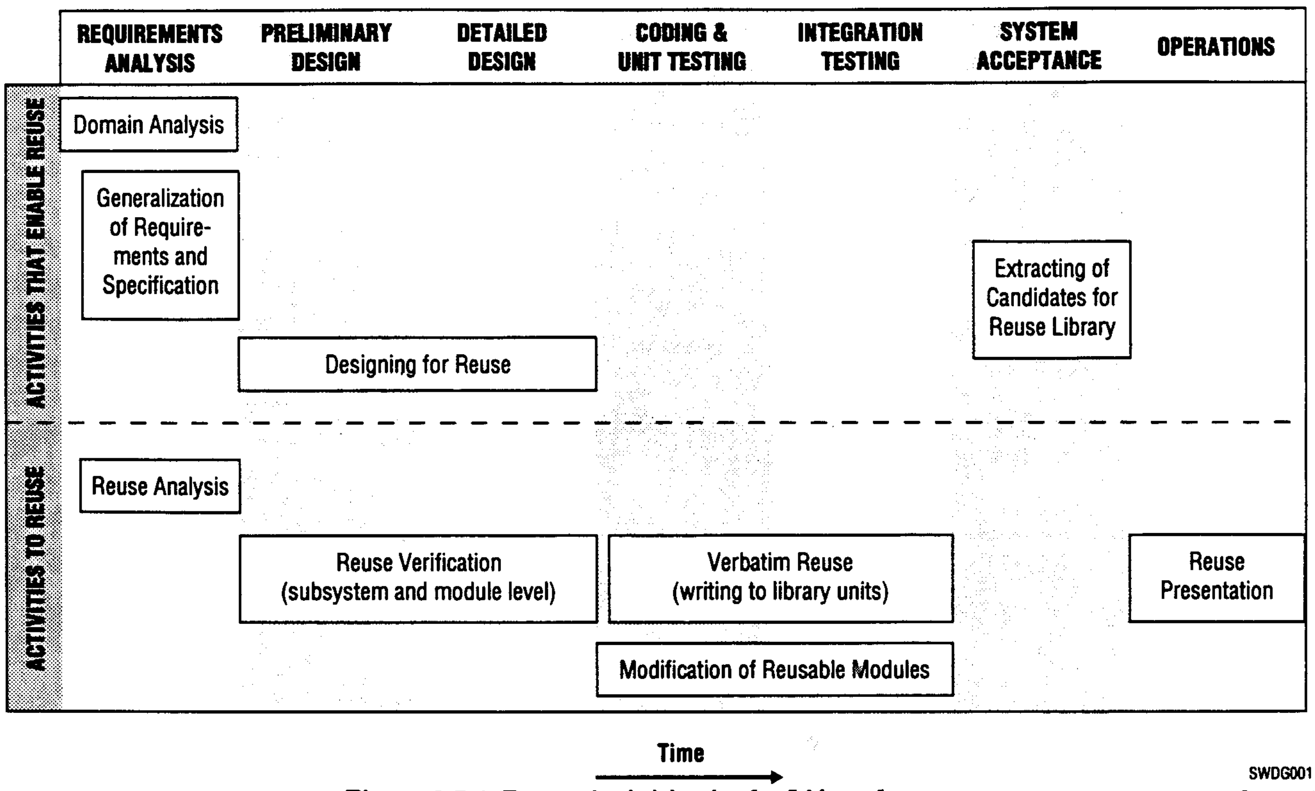

REQUIREMENTS ANALYSIS
PRELIMINARY DESIGN
DETAILED DESIGN
CODING & UNIT TESTING
INTEGRATION TESTING
SYSTEM ACCEPTANCE
OPERATIONS
ACTIVITIES THAT ENABLE REUSE
Domain Analysis
Generalization of Require-ments and Specification
Designing for Reuse
Extracting of Candidates for Reuse Library
ACTIVITIES TO REUSE
Reuse Analysis
Reuse Verification (subsystem and module level)
Verbatim Reuse (writing to library units)
Reuse Presentation
Modification of Reusable Modules
Time
SWDG001

**Figure 2.5-1. Reuse Activities in the Lifecycles** *[SEL-81-305]*

**Activities That Enable Reuse**

- **Domain Analysis**—Identifies common requirements across the application domain and helps produce a model that describes common functions of a specific application area. This can later be tailored to accommodate specific differences.
- **Requirements Generalization**—Covers those requirements that are intended to describe a "family" of systems or functions.
- **Designing for Reuse**—Provides modularity, standardized interfaces, and extensible and maintainable code.
- **Reuse Libraries**—Hold reusable source code and associated requirements, designs, documentations, and tests results. These products may be used verbatim or modified to fit the purpose.
- **Reuse Preservation**—Ensures that changes and enhancements made during the operational phase of the software adhere to the same principles that promote reuse, i.e., "quick fixes" may complicate future reuse.

The benefits of reuse can be maximized by planning for reuse early in the development process.

For example, to write reusable software, keep in mind the following guidelines:

- Set in-line documentation standards to increase understandability of code.
- Set naming constraints for constants, types, and functions.
- Set usage conventions for functions governing argument order and data type.

- Encapsulate all data structures.
- Adhere to industry standards (ANSI, POSIX, etc.).
- Strive for portability (to UNIXes, VMS, DOS) whenever possible).

See Section 4.4 for more details.

## 2.6 Cited References

[*DAV90*] Davis, A., Software Requirements: Analysis and Specification, Englewood Cliffs, New Jersey: Prentice-Hall, 1990, p. 7.

[*THA87*] Thayer, R., Tutorial: Software Engineering Project Management, Ed. R. H. Thayer, Computer Society of the IEEE, Washington, DC, 1987, p. 438.

[*THA87*] Thayer, p. 438.

[*THA87*] Thayer, p. 438.

[*DAV90*] Davis, p. 25.

[*ROY87*] Royce, W., "Managing the Development of Large Software Systems," Tutorial: Software Engineering Project Management, Ed. R. H. Thayer, Computer Society of the IEEE, Washington, DC, 1987, p. 121.

[SEL-81-305] *Recommended Approach to Software Development, Rev. 3*, Software Engineering Laboratory Series, SEL-81-305, NASA Goddard Space Flight Center, June 1992.

## Section 3

# Lifecycle Process Models

## Contents

## 3.1 Introduction

A lifecycle process model assists planning by defining the expected sequences of events, development and management activities, reviews, products, and milestones for a project. It is difficult to manage a software project without understanding the development state of the project, especially since a software product is a relatively abstract entity prior to completion. The phases in a lifecycle process model increase the visibility of individual activities within the complex, intertwined network of events during the development of a software product. The stepwise arrangement of phases provides milestones during the course of the project [*SIT88*].

**Using Lifecycle Phases**

- Establishes verification points at which products are reviewed for completeness, correctness, and consistency
- Forms a stable basis for proceeding into the next activity
- Eliminates many problems and enhances the probability of success in the following phase by ending the current phase with a review of its products

**Why Use Lifecycle Process Models?**

Lifecycle Process Models:

- Assist in planning and provide a common frame of reference and terminology.
- Define sequences of events and phases.
- Identify the activities to be performed.
- Establish reviews to be scheduled.
- Define the interim and end products that need to be produced.
- Provide milestones in the schedule to evaluate the plan and approach.
- Provide the basis for producing the software development plan, cost estimates, and schedules.
- Encourage developers to specify what the system is supposed to do (define the requirements) before building the system.
- Encourage developers to plan how components will interact (design) before building the system.
- Enable managers to track progress more accurately and to uncover slippages early.
- Recommend that the development process generate a series of documents that can later be used to test and maintain the system.
- Reduce development and maintenance costs.
- Enable the development of a more structured and manageable system.

[*DAV88*]

Various types of process models can be used to model the software development lifecycle. This section of the guidebook will present the waterfall, the spiral, and the incremental development models. In addition, the throwaway prototype and the evolutionary prototype model will be described.

Remember, models are prototypical guidelines, not gospel—they serve as frameworks and provide checklists. They are developed to *help*, not to *restrict*. They need not be followed exactly; the important point is to be aware of all the available options and to understand why you are deviating from the model (if you are)—it reminds you to make a conscious and informed decision.

[*ISD48*]

## 3.2 Waterfall Model

The classic waterfall model was first proposed by Winston Royce in 1970 [*ROY70*]. Figure 3.2-1 shows one representation of the waterfall model. The model progresses in distinct phases of development. The boxes represent the various phases of software development. The solid arrows indicate the direction of progress, and the ellipses between the phases are the reviews that typically occur at the end of a particular phase and before the following phase. The dotted arrows show possible paths for back-tracking through the phases in case problems occur. For example, if a new set of requirements is identified during detailed design, all related detailed design activities must be suspended while further requirements analysis is done to address these new requirements.

The waterfall model has gone through many refinements to deal with the increasing complexity of software development projects. Initially, the model did not have the back-tracking arrows to represent paths to retrace through the developmental stages. Most of the models used by contractors and Government agencies are some variation of the waterfall model.

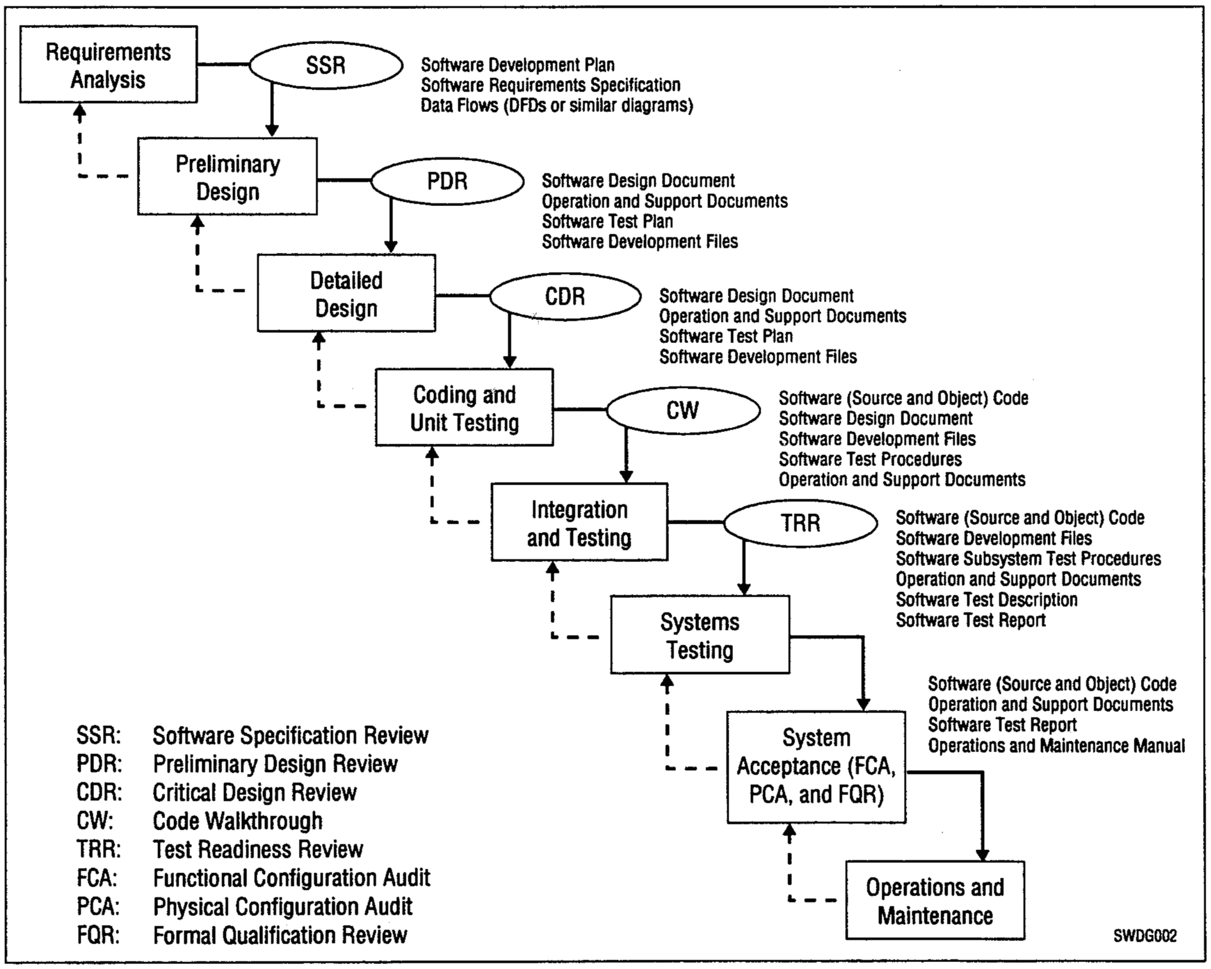

Requirements Analysis
SSR
Software Development Plan
Software Requirements Specification
Data Flows (DFDs or similar diagrams)
Preliminary Design
PDR
Software Design Document
Operation and Support Documents
Software Test Plan
Software Development Files
Detailed Design
CDR
Software Design Document
Operation and Support Documents
Software Test Plan
Software Development Files
Coding and Unit Testing
CW
Software (Source and Object) Code
Software Design Document
Software Development Files
Software Test Procedures
Operation and Support Documents
Integration and Testing
TRR
Software (Source and Object) Code
Software Development Files
Software Subsystem Test Procedures
Operation and Support Documents
Software Test Description
Software Test Report
Systems Testing
Software (Source and Object) Code
Operation and Support Documents
Software Test Report
Operations and Maintenance Manual
System Acceptance (FCA, PCA, and FQR)
Operations and Maintenance
SSR: Software Specification Review
PDR: Preliminary Design Review
CDR: Critical Design Review
CW: Code Walkthrough
TRR: Test Readiness Review
FCA: Functional Configuration Audit
PCA: Physical Configuration Audit
FQR: Formal Qualification Review
SWDG002

**Figure 3.2-1. The Waterfall Model—Phases, Reviews, and Major Products**

### 3.2.1 Lifecycle Phases of the Waterfall Model

**Requirements Analysis Phase**

The requirements analysis phase includes activities that analyze the software problem and develop specifications describing the external behavior of the software system to be built. The requirements specifications are also known as the functional description or functional requirements. The Software Requirements Specification (SRS) demonstrate to the customer that the developers understand the software problem and have defined *what* the software must do to solve it. The software requirements describing the functionality of the software, the external interfaces, and the expected performance are documented in the SRS. The requirements specification is written to be understood by the customer, users, developers, and testers. This phase culminates with the Software Specification Review (SSR).

**Preliminary Design Phase**

The preliminary design phase develops an implementable design from the SRS. The following activities are performed during the preliminary design: 1) refine the software system into smaller software subsystems; 2) allocate requirements to these subsystems; 3) develop the formal test approach; and 4) finalize decisions regarding whether software subsystems should be built, purchased, or reused and select the Database Management System (DBMS), if applicable. The software subsystems are documented in the Software Design Document (SDD) in terms of their inputs, outputs, and functions. The SDD describes *how* the software will meet the requirements specified in the SRS. The preliminary design is also known as the high-level design, architectural design, or functional design. This phase culminates with the Preliminary Design Review (PDR).

**Detailed Design Phase**

The detailed design phase develops the lowest level of the software design. The software subsystems are refined to identify the software modules that will be translated into code. The algorithms and internal logic for these software modules are defined using a design description language. This design information down to the module level is captured in the SDD. The detailed design is also known as program design. This phase culminates with the Critical Design Review (CDR).

**Coding and Unit Testing Phase**

During the coding and unit testing phase, the software modules are coded according to the designs developed during the design phases. After individual modules are coded, they are reviewed using code walkthrough and/or code reading techniques by other developers. Following successful review, the modules are tested (unit tested) to ensure that they perform their functions as required. The software modules are normally placed under configuration management at this time. This phase culminates with the completion of coding and successful testing of all modules.

**Integration and Testing Phase**

During the integration and testing phase, the software modules coded in the previous phase are integrated to form software subsystems. These subsystems are then individually tested. It is recommended that these tests are performed by individuals who are not part of the development team (though the developers may help in the testing process). This phase culminates with the Test Readiness Review (TRR).

**Systems Testing Phase**

During the systems testing phase, the software system is tested in its hardware environment to ensure that it functions as specified in the software requirements. This is the final phase of testing to ensure that all requirements have been satisfied and that the system is ready for the customer.

### System Acceptance Phase

During the system acceptance phase, in addition to formal testing, there are several reviews to verify that the hardware, software, and interfaces of the system are complete and documented for operational installation. These are the Formal Qualification Review (FQR), the Functional Configuration Audit (FCA), and the Physical Configuration Audit (PCA).

### Operations and Maintenance Phase

The operations and maintenance phase occurs when the software is delivered and operational at the end site. It is important that during this phase of the software lifecycle, requested software changes do not adversely affect the operational software. Software changes should be thoroughly tested and regression tested so that original functionality is not degraded by new software. If the requested software changes include changes to the requirements, it may be desirable to perform all of the previous software lifecycle activities again, beginning with requirements analysis.

*[ISD48]*

## 3.3 Spiral Model

The spiral model represents the activities related to software development as a spiralling progression of events that moves outward from the center of the spiral. For each development phase from project conception through preliminary design, this model places great emphasis on defining the objectives and evaluating alternatives and constraints, evaluating the alternatives and their potential risks, developing and verifying the compliance of an interim product (e.g., prototype, document), and planning for the next phase, using knowledge gained from the previous phases. The primary goal is to ensure that most of the development objectives, alternatives, and risks have been identified, addressed, and evaluated before proceeding to the next phase of development.

As illustrated in Figure 3.3-1, *each cycle* in the spiral model proceeds through the following four quadrants (steps):

### Quadrant A—Determine Objectives, Alternatives, Constraints

Each cycle of the spiral begins with Quadrant A, where the following are identified:

- The objectives for the portion of the product being addressed (e.g., performance, functionality, ability to accommodate change)
- The alternative approaches for implementing this portion of the product (e.g., approach A, approach B, reuse, buy)
- The constraints imposed on the application of the alternatives (e.g., cost, schedule, interface)

### Quadrant B—Evaluate Alternatives: Identify, Resolve Risks

The risks associated with each alternative are evaluated using formal risk analysis (see Section 5.6) with respect to the objectives and constraints. This process frequently identifies areas of uncertainty that are often significant sources of risk. Prototypes, simulations, questionnaires, and analytical models may be required to identify cost-effective approaches to resolve the risks. The next step depends on the results from the evaluation of the risks, and could be any of the following:

- Proceed with the next phase.
- Develop a model.
- Change the objectives.

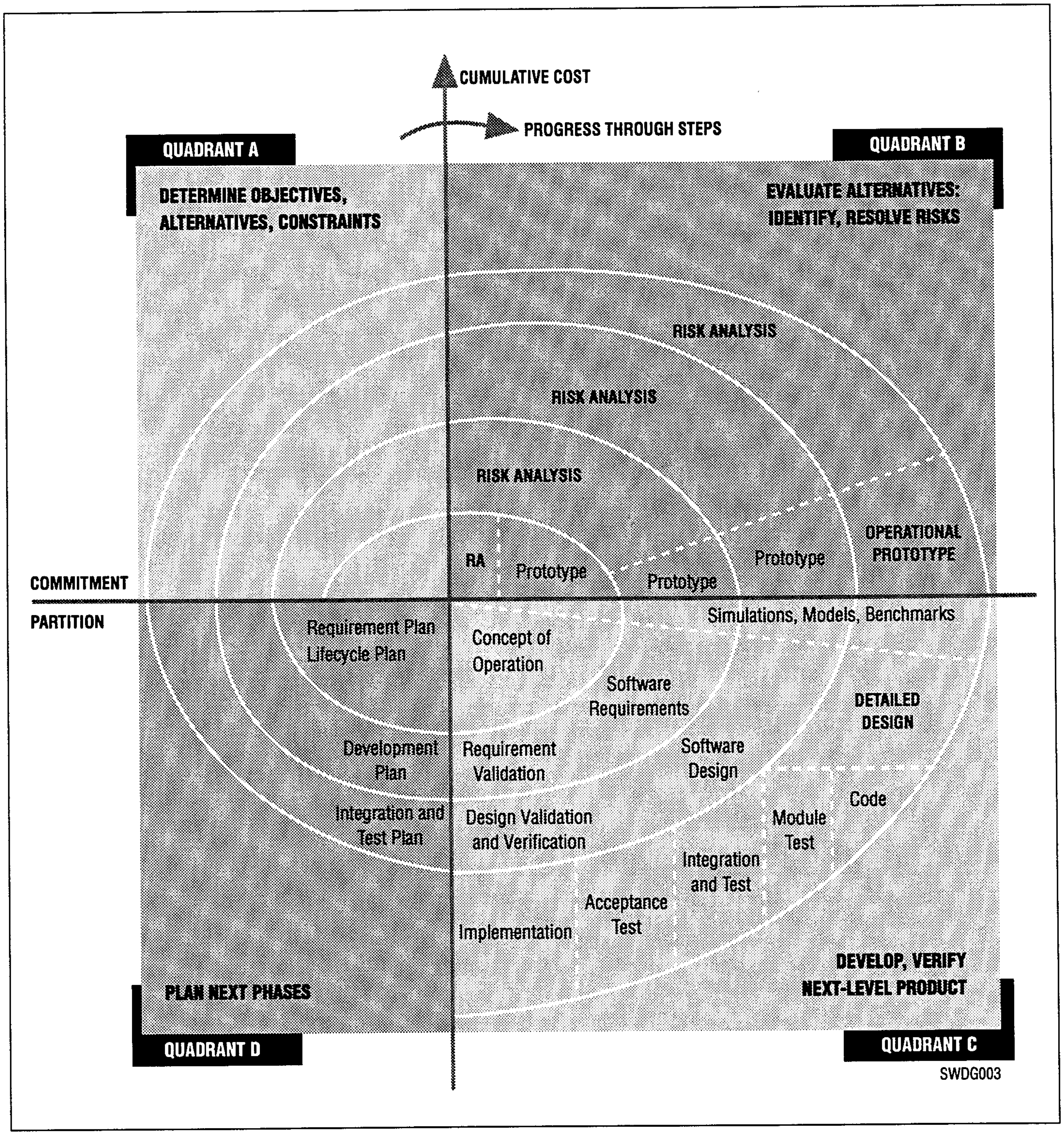


**Figure 3.3-1. Spiral Model of the Software Process**

- Revise the constraints.
- Adopt evolutionary development.
- Stop.

**Quadrant C—Develop, Verify Next-Level Product**

A product is developed. This product may be a plan, software requirements, software design, code, simulation, or prototype to address a specific issue. This product is then verified to ensure that it meets the objectives set in Quadrant A.

**Quadrant D—Plan Next Phases**

The next phase of development is planned. These plans are based on the information and lessons learned from the last completed step.

*Note: Once the detailed design of the software is complete, the spiral model proceeds to code and module testing, integration testing, and acceptance testing, just as the waterfall model does.*

As an example, locate the cycle of the spiral that includes software design. Trace the spiral back (counterclockwise) until you reach Quadrant A. Notice the activities that need to take place for the development of the software design. The spiral model moves through the quadrants, performing the following activities:

- All the objectives of the design are explicitly identified, along with the alternative design approaches and constraints for each approach.
- Risk analysis is performed to determine possible problems (and mitigation plans) that could arise during design (technical problems as well as problems such as staffing). Prototypes are created if needed to investigate risks.
- If necessary, models and simulations are created that address specific portions of the product (cycles of the spiral may be required for some of the issues being addressed). If all risks have been satisfactorily addressed, the software design can be generated. The design is then verified to ensure that it meets the objectives.
- The development plan for the integration and test plan is produced.

**Advantages of the Spiral Development Approach**

- The spiral model encourages analysis of objectives, alternatives, and risks at each step of development, providing an alternative to one big commitment/decision point at the start of the project. In Figure 3-2, the farther one moves away from the intersection of the axes, the greater the cost commitment.
- The spiral model allows for the objectives to be re-evaluated and refined based on the latest perception of needs, resources, and capabilities.

[*ISD48*]

## 3.4 Incremental Development Model

Incremental development is the process of building software by initially constructing a part of the entire system and progressively adding functionality in successive builds. Because the initial capability is achieved quickly, costs normally associated with development prior to the initial release are seemingly reduced; these costs are actually spread across a number of builds. By providing operational builds of the system more quickly, the possibility that the user's requirements may change during the development of a build is also reduced; changes in requirements may also be deferred to a later build of the software.

It must be noted that when the incremental development model is used, the software is intentionally constructed to (initially) satisfy fewer requirements. However, *the software is designed to facilitate the incorporation of new requirements in later builds.* Figure 3.4-1 is an example of the incremental development model.

**Advantages of the Incremental Development Approach**

- Initial development time is reduced (because of the reduced functionality).
- Software can be progressively enhanced for a longer period of time (because it is designed for growth).
- The operational date is earlier (although at limited functionality).
- Mechanisms to address/cope with changing requirements are provided.
- Tradeoffs of functionality and performance between versions are allowed.

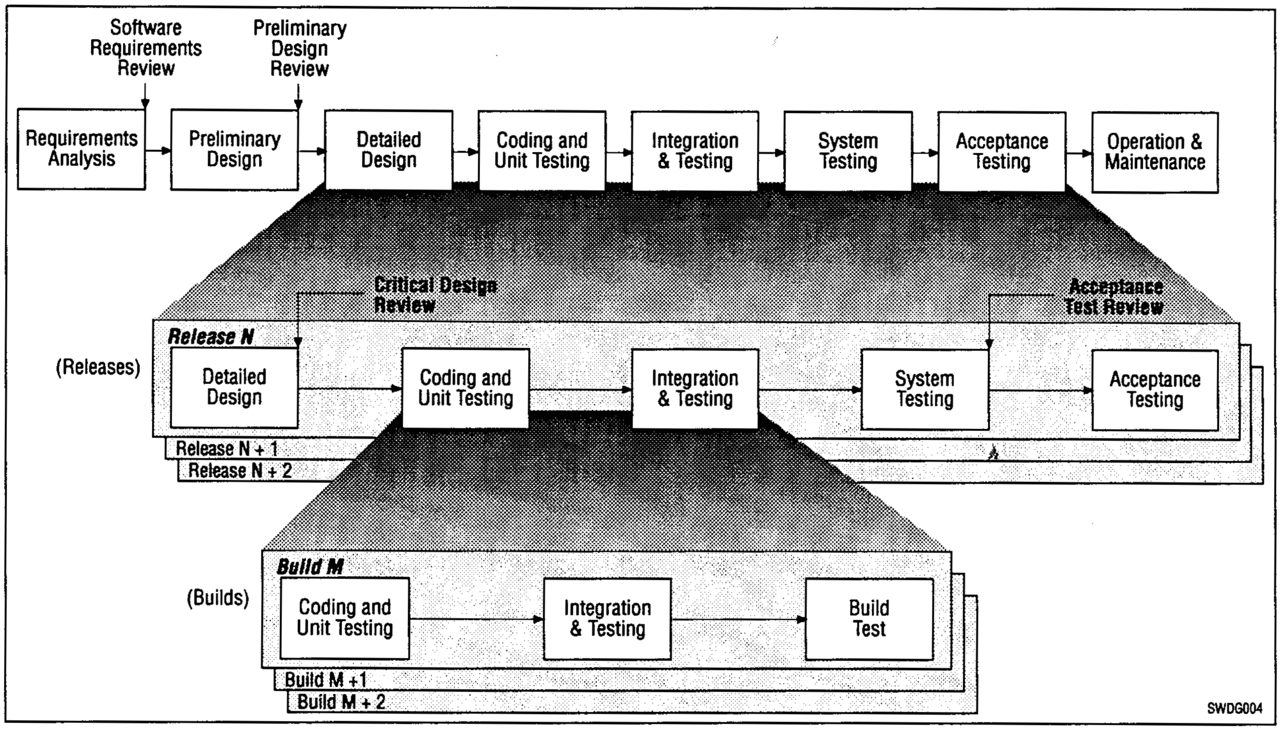


**Figure 3.4-1. Incremental Development Model** *[SEL-81-305]*

This software development approach is different from the evolutionary prototype model (see Section 3.5.2) because the implication is that in the incremental development model the developers understand most of the requirements but are choosing to provide the functionality in subsets of increasing capability.

Remember, when using the incremental development process model, the software must be designed carefully to easily support additional functionality and growth. The functionality that is not being provided in the current build is *deferred* for a later build, but the plans for adding this functionality must be well thought out and analyzed.

## 3.5 Prototyping and Prototyping Models

**Prototyping** is the technique of constructing a partial implementation of a system so that customers, users, or developers can learn more about a problem or a solution to that problem [*DAV90*]. The key word here is "partial"; if you were implementing the complete system, it would no longer be a prototype, it would be *the system*. Prototypes can be developed in the requirements, design, or coding phases of the software development lifecycle.

Prototyping is not a euphemism for "hacking," nor is prototyping an excuse to develop undocumented and unstructured code. Remember, the primary objective in developing a prototype is to *learn*; a completely undocumented, unstructured, and sloppy prototype will outweigh its usefulness with time wasted by developers attempting to figure out how it was constructed.

**Some Reasons To Develop a Prototype**

- Demonstrate a capability either internally or to an external customer.
- Assess a design approach or an algorithm for correctness or efficiency.
- Evaluate the ability of a software development system to support efficient software production or to support a given number of programmers.
- Provide a measurement vehicle when estimating user response times, recovery times, transmission times, code expansion factors, etc.
- Validate requirements by demonstrating that they can be implemented and exploring possible error conditions that requirements must cover.
- Clarify ambiguous requirements.
- Provide a vehicle for soliciting end-user input, primarily on the Human-Machine Interface (HMI).
- Form a basis for the full implementation effort.
- Serve as an early, concrete milestone in the development schedule.
- Demonstrate feasibility of new and evolving technology.

**Prototype Development Guidelines**

- The degree of formality of a prototype should match the intended use of the prototype. This means that the formality of both the processes and the products must be considered. For example, a prototype that is implemented to provide an initial operational capability should have rigorous reviews, because it will be delivered as a part of the final system. A prototype whose purpose is to display screen format to solicit user input may not require such formal reviews if it will be discarded (or used only for further information gathering) and its robustness is not an issue.
- Prototypes must be produced early enough to have an effect. For example, if a critical algorithm is being prototyped to determine whether it will provide the necessary accuracy, the optimum time for the prototype is *not* just prior to the CDR; if the prototype shows that the algorithm is inadequate, there will be effects on requirements and top-level design as well as on detailed design, and the results will not be ready in time for the CDR.
- The following is more of a recommendation: Because a prototype is still a product, it should reflect the same high quality that any other product would have. The ultimate use of a product is sometimes unknown—a prototype that begins as a proof of concept might evolve into operational capability. It can be difficult to convert a questionable quality prototype to a high-quality operational product. In all prototypes, comments should make sense, the design and source code should be structured, inputs and outputs should be consistent, and so on.

[*SEL-81-305*]

### 3.5.1 Planning for Prototype Development

Managing a prototype development effort requires special care and attention. It is often difficult to foresee the progress of the development effort and therefore difficult to measure. *Beware: A prototyping effort could continue indefinitely if the completion criteria and evaluation guidelines are not established.* It is essential to write a plan to monitor and track every prototyping activity. The detail of the contents of the plan should be proportional to the prototyping effort; i.e., a one-page plan would suffice for small efforts as long as the issues in the following table have been addressed.

**Prototyping Plan Contents**

- The purpose and use of the prototype
- Brief description of the work to be done and the products to be generated
- Technical approach
- Completion criteria
- Evaluation criteria and methods
- Resources required: effort, size, staff, and hardware and software estimates
- Schedule

[*SEL-81-305*]

### 3.5.2 The Throwaway Prototype

A throwaway prototype is constructed to learn more about the problem or its solution. *This prototype is discarded once it has been used and the requisite knowledge has been gained* [*DAV88*]. Though these prototypes are throwaway, the design and code should be understandable to its developers for the prototype to fully serve its purpose. The throwaway prototype should be delivered quickly—there are no rigorous lifecycle phases to be followed. The advantage lies in quickly gaining additional knowledge about a certain aspect of the system so that the normal development lifecycle of the system can proceed accordingly. A throwaway prototype can be developed during the requirements, design, and coding phases of any of the lifecycle process models (waterfall, spiral, incremental build, evolutionary prototyping, etc.).

**During Requirements Analysis, a Prototype May Be Developed To:**

- Determine the feasibility of a requirement.
- Validate that a particular function is particularly necessary.
- Uncover missing requirements.
- Clarify an ambiguous requirement.
- Determine the validity of the user interface.
- Write a preliminary SRS.
- Implement a prototype based on a preliminary SRS.
- Achieve user experience with the prototype.

Beware of a common scenario that occurs when a throwaway prototype is delivered: the customers say they love the prototype and want to make it an operational system. It is the responsibility of the developer to explain to the customer that a good prototype does not mean it is a great product. As Davis states, "That's like saying they want to put wings on a prototype of a flight simulator and fly it for real!" [*DAV88*]

The following are two ways to prevent the prototype from being used as the actual system:

- Prototype the system in pieces (do not build an end-to-end prototype).
- Simulate the system's interaction with data.

Figure 3.5.2-1 represents throwaway prototypes being developed while using a waterfall model.

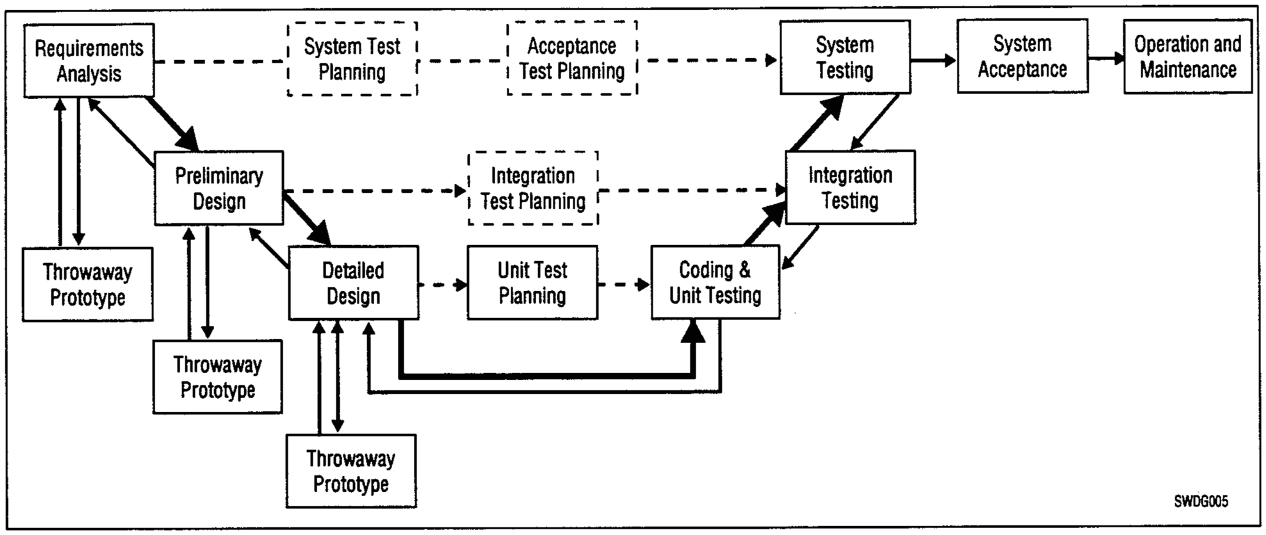


**Figure 3.5.2-1. Throwaway Prototyping During Requirements Analysis, Preliminary Design, and Detailed Design**

### 3.5.3 Evolutionary Prototyping Model

In this model, the prototype is constructed to learn more about the problem or its solution. Once the prototype has been used and the requisite knowledge has been gained, *the prototype is then adapted to satisfy the now better understood requirements.* Evolutionary prototypes cannot be built in a sloppy manner. Because the evolutionary prototype will finally evolve into the final product, it must demonstrate all the quality, maintainability, and reliability associated with the final product. Remember,

**It is impossible to retrofit quality, maintainability, and reliability.** [*DAV88*]

Compromises that can be made while developing an evolutionary prototype are 1) building only the parts of the system that are well understood, leaving the others to later generations of the prototype (these parts could be developed from knowledge gained from a throwaway prototype of the "obscure" module) and 2) lowering the importance of performance (to paraphrase Dijkstra, *it is easier to make a working program faster than make a fast program work*).

If necessary, you could build a throwaway prototype during an evolutionary prototyping process, especially if it clarifies your understanding of the issues you are addressing with the evolutionary prototype.

Figure 3.5.3-1 represents the model for an evolutionary prototype.

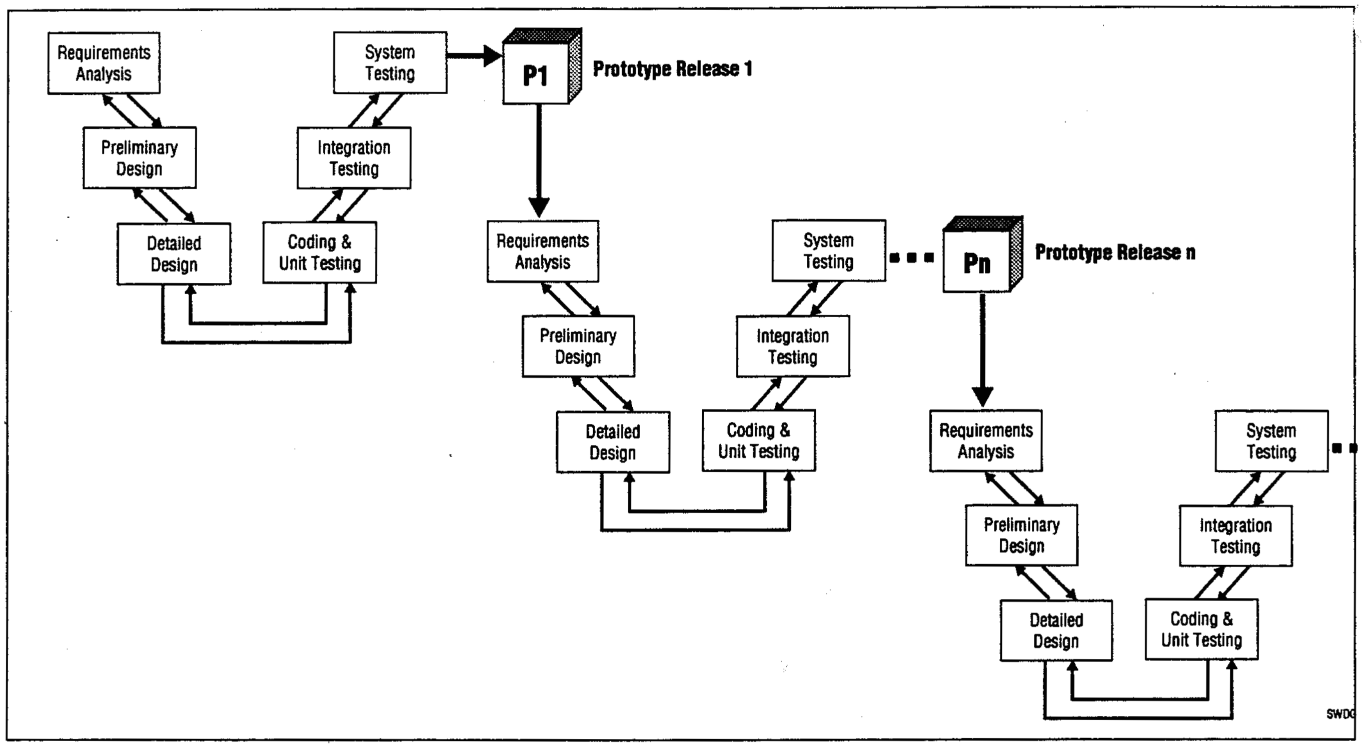


**Figure 3.5.3-1. The Evolutionary Prototype Model**

## 3.5.4 Throwaway Prototyping vs. Evolutionary Prototyping

Table 3.5.4-1 compares the throwaway prototype with the evolutionary prototype:

**Table 3.5.4-1. Throwaway Prototype vs. Evolutionary Prototype**

| | **Throwaway Prototype** | **Evolutionary Prototype** |
|---|---|---|
| **Development Guidelines** | Quick and dirty, no rigor | Structured, rigorous |
| **What To Build** | Build only difficult parts | Build only understood parts first, build on solid foundation |
| **Design Drivers** | Optimized development time | Optimized modifiability |
| **Ultimate Goals** | Learn from it and *throw it away* | Learn from it and *evolve it* |

[*DAV88*]

## 3.5.5 Types of Prototypes

Table 3.5.5-1 compares the various reasons a prototype may be built (i.e., to demonstrate proof of concept of performance, to demonstrate proof of HMI concepts, for rapid implementation, or for [or to demonstrate] initial operational capability), with consideration given to requirements, documentation, validation, management visibility, acceptance, delivery, team size, and cost.

Table 3.5.5-1. Comparison of Prototypes

| | Proof of Concept or Performance | Proof of HMI Concepts | Rapid Implementation | Initial Operational Capability |
|---|---|---|---|---|
| **Characteristic Requirements** | Iterative | Informal | White Paper | Formal with reviews |
| **Documentation** | Notes only | Notes and display format | Notes and users manual | Full complement |
| **Validation** | By analysis | User inspection | Complete but informal | Formal, with plans and procedures |
| **Management Visibility** | Informal status | Informal status | Few milestones, informal | Well-defined milestones and reviews |
| **Acceptance** | Engineering evaluation | Demonstration | Experimental observations | Formal, with customer buyoff |
| **Delivery** | Throwaway | Input for requirements | To customer | To customer |
| **Team Size** | Small | Small/medium | Small | Medium/large |
| **Cost** | Small | Small | Reduced Development | Full Development |
| **Type of Prototype** | Throwaway | Throwaway | Throwaway | Evolutionary |

## 3.6 Selecting a Model

A model should be selected based on the needs of the contract and the task. This usually occurs in the proposal stage before the contract is awarded or at the very beginning of the contract or task. The selection of the model depends on the analysis of the requirements and other contractual constraints and issues. A model is selected to oversee the development of an engineered product and to help engineers and project managers control the development of the product.

In our work, by the time a contract is issued, the model has usually already been selected. If the required task is to define requirements; design, code, and test the software; and deliver the product, we usually select the waterfall model. When requirements are unclear and volatile, we use the evolutionary prototyping model. If operational capability is required in a short period of time for a system with well-defined requirements, we choose the incremental development model. If the task is maintaining existing software, the same software development processes apply, although in smaller pieces (as changes, whether as Engineering Change Requests, Change Orders, or Task Orders). There is usually not enough time to perform a risk analysis to determine whether the system objectives are likely to satisfy user needs.

The theory behind the spiral model is that a program can proceed in steps, with each step leading to well-analyzed decisions for the next step. In the overall field of software development, where up to 50% of software projects are said to lead to no usable products, the spiral model is useful in promoting reasoned analysis during the life of the project. For example, if the risk analysis conducted after the definition of software requirements showed that the system was not feasible, the requirements can be scaled back, or the entire project can be modified before large amounts of resources are wasted.

The spiral model is particularly well suited to internal company projects, where such decision points can exist. In some cases, Government systems are developed in phases, with separate contracts for feasibility studies, design competitions, demonstrations, and full-scale development. When a task includes the flexibility required to implement it, the spiral model is a good choice. It encourages evaluation of alternatives and incremental planning.

*Note: A spiral model cannot be used effectively if the objectives, constraints, or plans cannot be changed.*

Any model can be applied to a small part of any job. For example, an individual change request could be handled in accordance with any model. A simple change could be implemented using the waterfall model. A complicated change could be handled by defining objectives, evaluating alternatives, analyzing risk and identifying contingencies, producing a plan, and developing the product (which would then consist of revised code and documentation); this calls for the basic spiral model. The product could be delivered in increments, which would make the evolutionary spiral model apply. *[ISD48]*

Table 3.6-1 compares the waterfall, spiral, incremental development, and evolutionary prototyping models.

**Table 3.6-1. A Comparison of Lifecycle Process Models**

| | Characteristics | Advantages | Disadvantages |
|---|---|---|---|
| **Waterfall** | • Disciplined and sequential approach<br>• Requirements need to be known at the start<br>• Document driven | • Simple model<br>• Well-defined steps | • Big commitment required up front<br>• User problems identified late |
| **Spiral** | • Risk reduction at every step<br>• Flexible, iterative process<br>• Supports evolving system needs | • Risks addressed, evaluated, and reduced at every step | • Difficult to implement for contract software<br>• Difficult to schedule<br>• Difficult to decide on the "number of turns of the spiral" |
| **Incremental Development** | • Early (initial) operational capability<br>• Software designed to facilitate growth<br>• Partial capability, with additional capability provided in subsequent builds | • Early availability of initial operational capability<br>• Software designed to be extensible<br>• Problems addressed with each build<br>• Allows tradeoffs between functionality and performance between builds | • Difficult to manage the development, testing, and release of the builds |
| **Evolutionary Prototyping** | • Builds the difficult parts<br>• Provides increasing capability with each release<br>• Software built to learn from, and then evolved | • Software designed to be extensible<br>• Problems addressed with each release | • Difficult to decide which requirements should be addressed with each release |

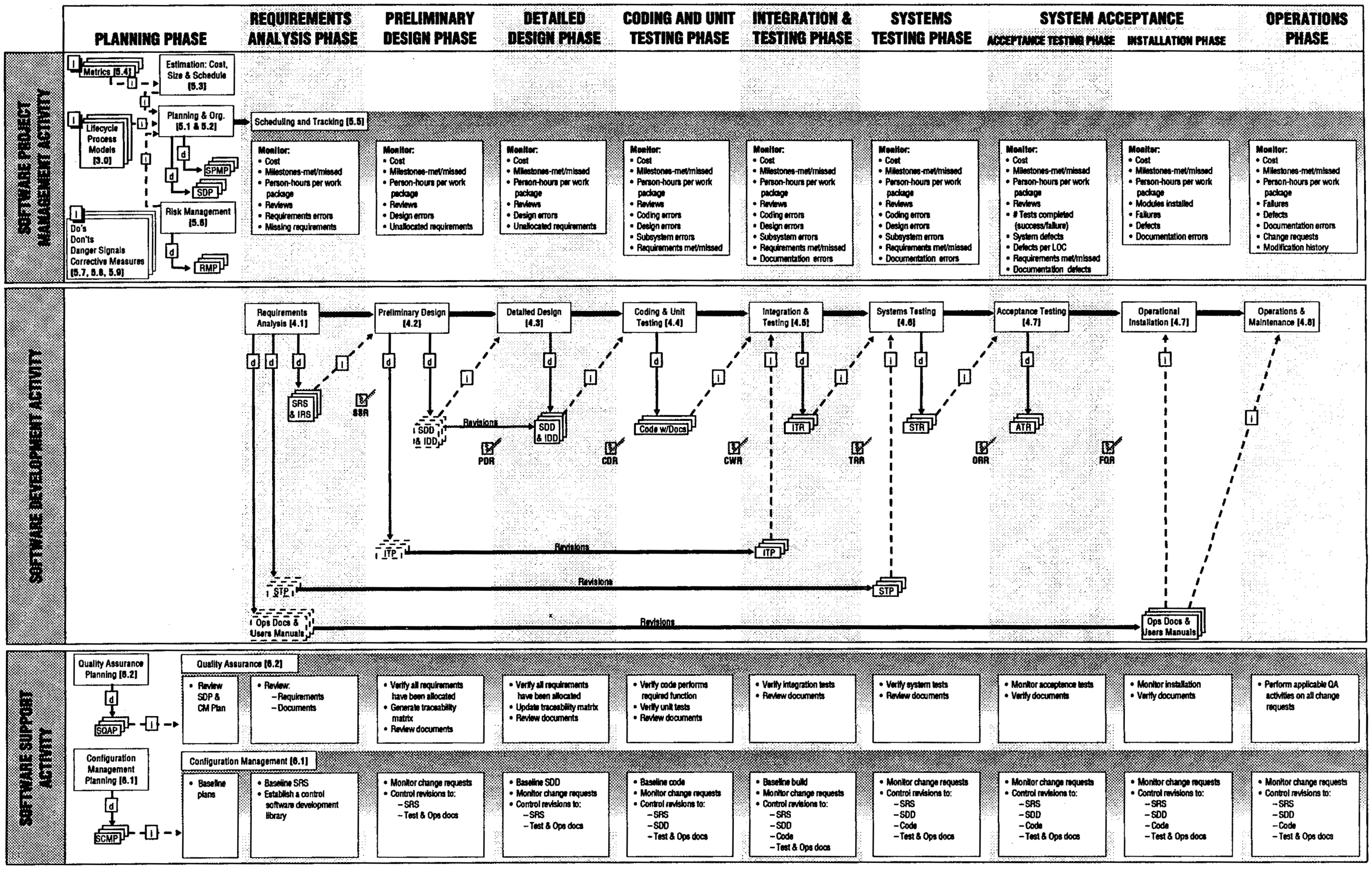
## 3.7 Cited References

| | |
|---|---|
| [*SIT88*] | Sitaram, Pradip, "A Concurrent Process Model for Software Development," Thesis, 1988. |
| [*DAV88*] | Davis, E. Bersoff, and E. Comer, "A Strategy for Comparing Alternative Software Development Life Cycle Models," *IEEE Transactions on Software Engineering, 14, 10,* October 1988. |
| [*ISD48*] | *Software Engineering Handbook, Build* 3, Division 48, Information System Division, Hughes Aircraft Company, March 1992, p. 2-11. |
| [*ISD48*] | *Software Engineering Handbook, Build* 3, pp. 2-4–2-6. |
| [*ISD48*] | *Software Engineering Handbook, Build* 3, pp. 2-6–2-8. |
| [*SEL-81-305*] | *Recommended Approach to Software Development, Rev. 3,* Software Engineering Laboratory Series, SEL-81-305, NASA Goddard Space Flight Center, June 1992. |
| [*DAV90*] | Davis, A.,*Software Requirements: Analysis and Specification,* Englewood Cliffs, New Jersey: Prentice-Hall, 1990, p. 343. |
| [*DAV88*] | Davis, p. 343. |
| [*DAV88*] | Davis, p. 347. |
| [*DAV88*] | Davis, p. 346. |
| [*DAV88*] | Davis, p. 354. |
| [*ISD48*] | *Software Engineering Handbook, Build* 3, pp. 2-10–2-11. |
| [*ROY70*] | Royce, W., "Managing the Development of Large Software Systems," *Proceedings of IEEE WESCON,* 1970, p. 1-9. |

*Section 4*

# Software Development Activities

# Contents


PLANNING PHASE
REQUIREMENTS ANALYSIS PHASE
PRELIMINARY DESIGN PHASE
DETAILED DESIGN PHASE
CODING AND UNIT TESTING PHASE
INTEGRATION & TESTING PHASE
SYSTEMS TESTING PHASE
SYSTEM ACCEPTANCE
ACCEPTANCE TESTING PHASE
INSTALLATION PHASE
OPERATIONS PHASE
SOFTWARE PROJECT MANAGEMENT ACTIVITY
Metrics [5.4]
Estimation: Cost, Size & Schedule [5.3]
Lifecycle Process Models [3.0]
Planning & Org. [5.1 & 5.2]
SPMP
SDP
Risk Management [5.6]
RMP
Do's Don'ts Danger Signals Corrective Measures [5.7, 5.8, 5.9]
Scheduling and Tracking [5.5]
Monitor: • Cost • Milestones-met/missed • Person-hours per work package • Reviews • Requirements errors • Missing requirements
Monitor: • Cost • Milestones-met/missed • Person-hours per work package • Reviews • Design errors • Unallocated requirements
Monitor: • Cost • Milestones-met/missed • Person-hours per work package • Reviews • Design errors • Unallocated requirements
Monitor: • Cost • Milestones-met/missed • Person-hours per work package • Reviews • Coding errors • Design errors • Subsystem errors • Requirements met/missed
Monitor: • Cost • Milestones-met/missed • Person-hours per work package • Reviews • Coding errors • Design errors • Subsystem errors • Requirements met/missed • Documentation errors
Monitor: • Cost • Milestones-met/missed • Person-hours per work package • Reviews • Coding errors • Design errors • Subsystem errors • Requirements met/missed • Documentation errors
Monitor: • Cost • Milestones-met/missed • Person-hours per work package • Reviews • # Tests completed (success/failure) • System defects • Defects per LOC • Requirements met/missed • Documentation defects
Monitor: • Cost • Milestones-met/missed • Person-hours per work package • Modules installed • Failures • Defects • Documentation errors
Monitor: • Cost • Milestones-met/missed • Person-hours per work package • Failures • Defects • Documentation errors • Change requests • Modification history
SOFTWARE DEVELOPMENT ACTIVITY
Requirements Analysis [4.1]
Preliminary Design [4.2]
Detailed Design [4.3]
Coding & Unit Testing [4.4]
Integration & Testing [4.5]
Systems Testing [4.6]
Acceptance Testing [4.7]
Operational Installation [4.7]
Operations & Maintenance [4.8]
SRS & IRS
SSR
SDD & IDD
PDR
Code w/Docs
CDR
ITR
CWR
STR
TRR
ATR
ORR
FQR
ITP
STP
Ops Docs & Users Manuals
Revisions
d
i
SOFTWARE SUPPORT ACTIVITY
Quality Assurance Planning [6.2]
SQAP
Quality Assurance [6.2]
• Review SDP & CM Plan
• Review: –Requirements –Documents
• Verify all requirements have been allocated • Generate traceability matrix • Review documents
• Verify all requirements have been allocated • Update traceability matrix • Review documents
• Verify code performs required function • Verify unit tests • Review documents
• Verify integration tests • Review documents
• Verify system tests • Review documents
• Monitor acceptance tests • Verify documents
• Monitor installation • Verify documents
• Perform applicable QA activities on all change requests
Configuration Management Planning [6.1]
SCMP
Configuration Management [6.1]
• Baseline plans
• Baseline SRS • Establish a control software development library
• Monitor change requests • Control revisions to: –SRS –Test & Ops docs
• Baseline SDD • Monitor change requests • Control revisions to: –SRS –Test & Ops docs
• Baseline code • Monitor change requests • Control revisions to: –SRS –SDD –Test & Ops docs
• Baseline build • Monitor change requests • Control revisions to: –SRS –SDD –Code –Test & Ops docs
• Monitor change requests • Control revisions to: –SRS –SDD –Code –Test & Ops docs
• Monitor change requests • Control revisions to: –SRS –SDD –Code –Test & Ops docs
• Monitor change requests • Control revisions to: –SRS –SDD –Code –Test & Ops docs
• Monitor change requests • Control revisions to: –SRS –SDD –Code –Test & Ops docs

Software development is one of the three major activities performed in the software lifecycle; the other two are software project management and software support (i.e., Configuration Management [CM] and Quality Assurance [QA]). In concert with the other two activities, software development is an ongoing activity throughout the lifecycle; it proceeds through the following phases:

- Requirements Analysis Phase—Section 4.1
- Preliminary Design Phase—Section 4.2
- Detailed Design Phase—Section 4.3
- Coding and Unit Test Phase—Section 4.4
- Integration and Testing Phase—Section 4.5
- System Testing Phase—Section 4.6
- Acceptance Testing Phase—Section 4.7
- Operations and Maintenance Phase—Section 4.8

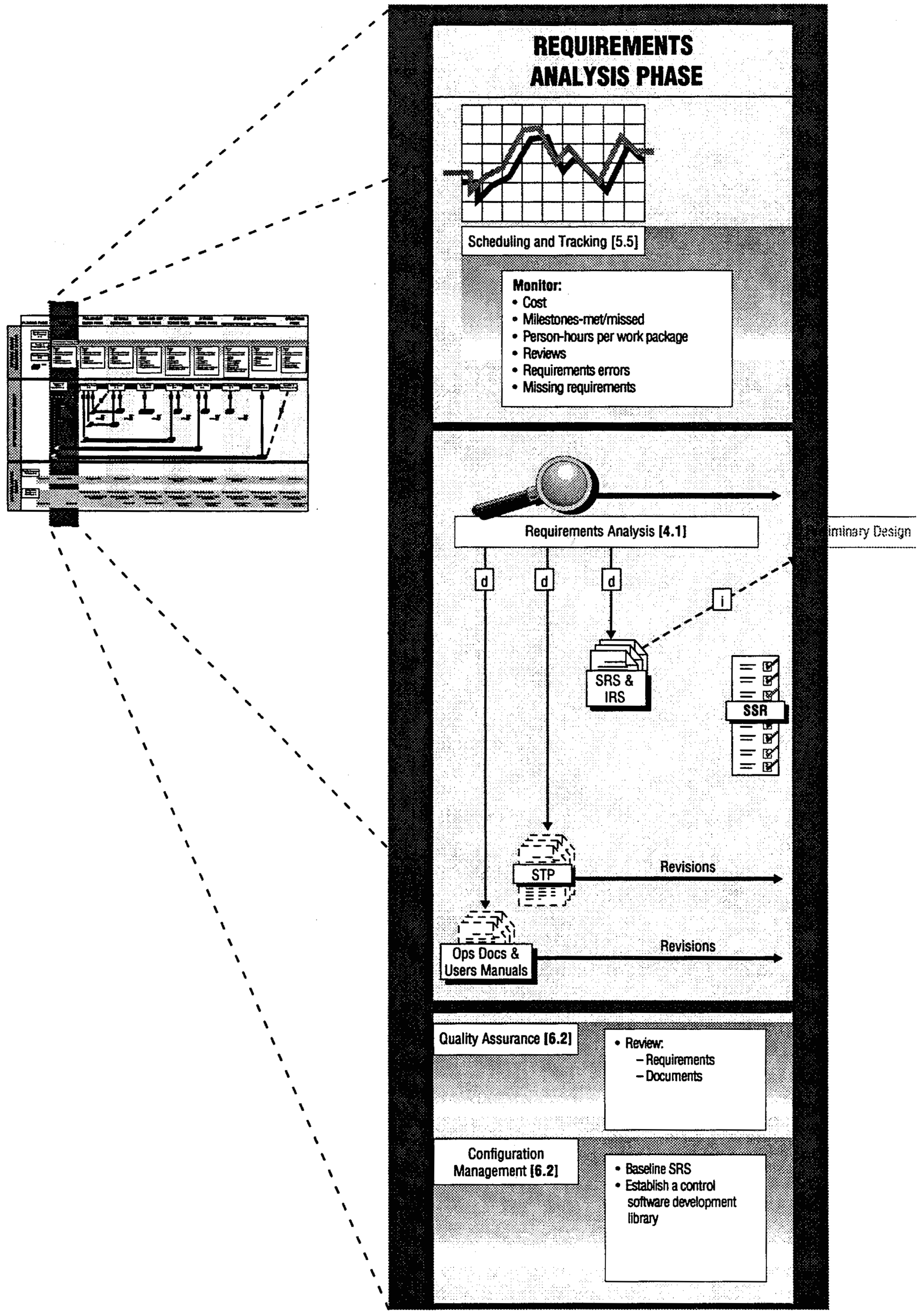
Each section contains a description of all the activities performed during that phase; descriptions of deliverables, documentation, and reviews; and checklists, sample tables of contents for documentation, and references for further information.

The facing page is a photo-reduced copy of Figure 2.2-1 (Software Development Process—An Integrated View) presented in Section 2. Each subsection in Section 4 (i.e., 4.1–4.8) begins with a "zoomed in" view of its respective phase as illustrated in this figure.

*Section 4.1*

# Requirements Analysis Phase

## Contents


REQUIREMENTS ANALYSIS PHASE
Scheduling and Tracking [5.5]
Monitor:
• Cost
• Milestones-met/missed
• Person-hours per work package
• Reviews
• Requirements errors
• Missing requirements
Requirements Analysis [4.1]
iminary Design
d
d
d
i
SRS & IRS
SSR
STP
Revisions
Ops Docs & Users Manuals
Revisions
Quality Assurance [6.2]
• Review:
– Requirements
– Documents
Configuration Management [6.2]
• Baseline SRS
• Establish a control software development library

## 4.1.1 Introduction

A complete, concise description of the external behavior of the software system, including its interfaces to its environment, other software systems, communications ports, hardware, and users is developed during this phase of software development. This description is recorded in a document called the Software Requirements Specification (SRS). To analyze and specify the software requirements, software developers must first analyze the current system (automated or nonautomated) and the problem(s) being addressed. The information required to perform this analysis is obtained from the Statement of Work (SOW), operations concepts documents, and systems requirements. Additionally, vital information and insight should be obtained from interviews with users and customers.

In most HSTX projects, our software engineering staff is responsible for developing the SRS. However, if there is a well-defined systems engineering organization for the project, systems engineering personnel may develop software requirements from the system specification, operational concept documents, and other analyses documents. It must be noted that no matter who produces the SRS, it must be complete, unambiguous, and understood by all related organizations within HSTX and the customer. The SRS is the project manager's key tool for controlling the scope of the software development effort.

The SRS that is produced during this phase is the communication tool between the developer and the customer. It shows the customer that the developers understand what the customer wants. The SRS then serves as a management tool and is used to establish baseline requirements, allowing managers and developers to control changes by monitoring cost and schedule impacts.

Figure 4.1.1-1 presents a graphical representation of the various activities associated with the requirements analysis phase of software development. [*ISD48*]

## 4.1.2 General Methodology for Developing Software Requirements

Various steps are involved in specifying software requirements. Some of the following steps may not apply to your particular project. You can tailor this process to best suit your project's needs.

1. Allocate the requirements that have been specified in the system specification, operational concept document, customer Request for Proposal (RFP), HSTX proposal, or contract decision agreements to the software system's requirements. This should be done formally because it will establish the basis for future testing.

2. Allocate all the system's inputs and outputs to the software system(s). A system input or output is one that is visible from *outside* the system. That is, it is not an internal flow between software systems or within software subsystems. It is an item that could be tested for, during the formal testing, and it relates to an existing requirement on the system.

*Note: Because each software system will have its own SRS (and possibly an Interface Requirements Specification [IRS]), the remaining steps in this methodology are applied to each software system.*

3. Develop an initial *software design concept* to identify the functions of the software system (if required). Generally, SRS's are organized by function. Note that at this stage a function is not actually a design element (because no software design phase has occurred), but it often assumes that identity during the later stages of development. Develop function definitions that make sense in terms of software design. Try to minimize the data flow

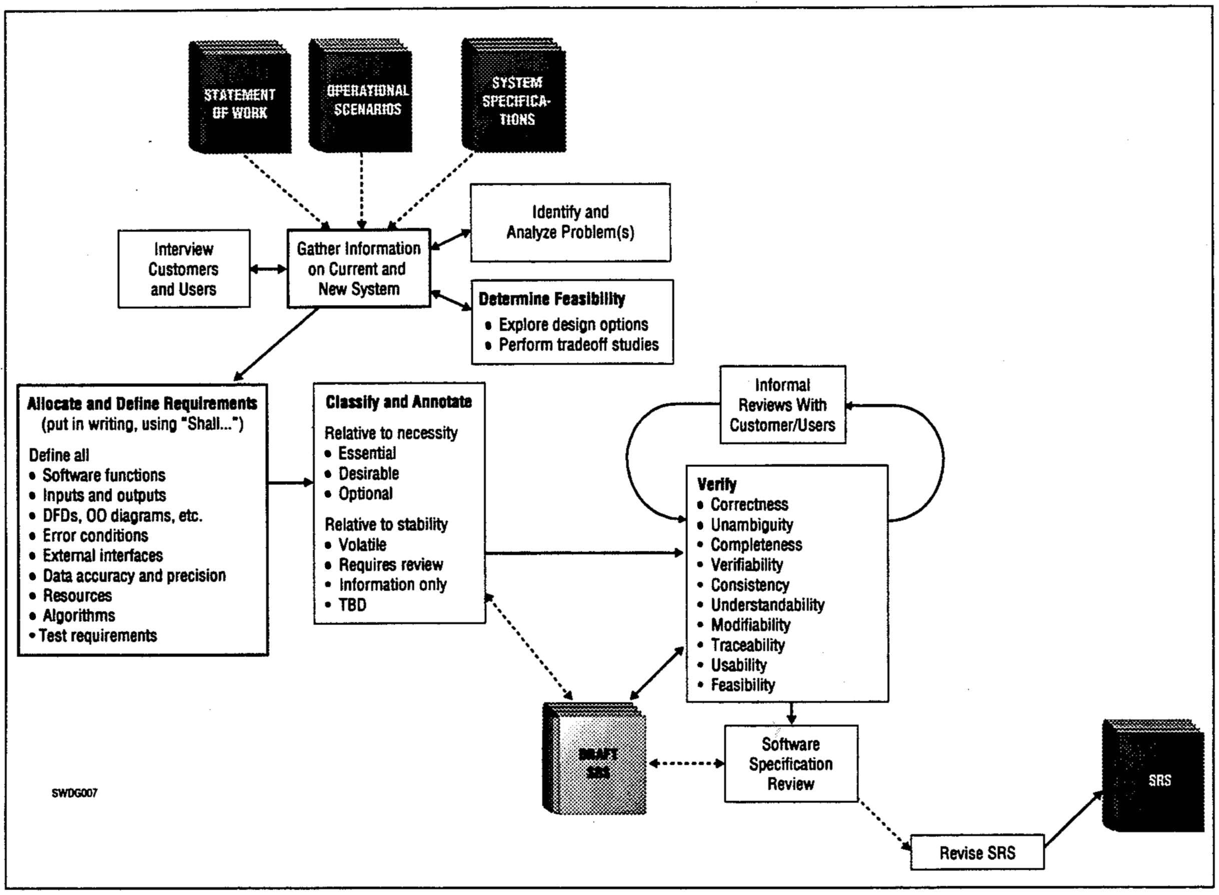


**Figure 4.1.1-1. Requirements Phase Process Flow**

among functions and to keep them cohesive. If data flow diagrams are used in analyzing system requirements, their topology can help identify functions.

4. Allocate requirements, inputs, and outputs to functions. At this point, there is no need to "connect the functions" with data flows internal to the software system. There is often pressure to fabricate those internal connections, but the large scope of that job will obscure the need to refine system requirements into real software requirements. The internal data flows will be constructed in Step 8.

5. For each input to the software system, define the processing requirements. Describe *what* the software system is required to do with that input. This step involves analyzing all requirements dealing with that input. Notice that the question here is "What?", not "How?".

6. For each output from the software system, define the processing requirements. Describe *what* the software system is required to do to produce that output. Again, the question is "What?", not "How?".

7. Connect the inputs to the software system and outputs from the software system. If any input does not relate to an output, determine why that input exists. If any output does not relate to an input, determine how the output was produced. These connections can be made using data flow diagrams, operational flow diagrams, minispecifications, or other techniques.

8. Connect the software system's functions with the data flows after all the software system's requirements, inputs, and outputs have been identified and understood. A consistent and complete SRS should have each input to the software system traceable in some form through the functions to an output from the software system, and vice versa.

9. Produce the detailed requirements. Decompose each requirement into a set of necessary and sufficiently detailed requirements. The goal is to produce a set of requirements that adequately and fully define the required processing of the software system. The SRS should specify everything the software system must do (the software system need not do anything that the SRS does not specify). The crucial factor here is to include only the requirements, *not the design* (i.e., include only *"what,"* not *"how"*). See Section 4.1.9.1, SRS Checklist, for a further discussion of this point.

10. Determine whether any other requirements need to be added. The following is a list of some "nonfunctional" requirements areas to consider:

    a. Interfaces
    b. Security
    c. Adaptation
    d. Performance
    e. Resource Utilization
    f. Safety
    g. Constraints (design, environmental, budget)
    h. Quality Factors
    i. Human Engineering
    j. Traceability
    k. Qualification
    l. Preparation for Delivery

    "Resource utilization" consists of any requirements for CPU utilization, throughput, storage, response time, communication bandwidth, and peripheral device usage. Remember to include only real requirements, not desires or estimates.

11. Tailor a requirements evaluation checklist (see Section 4.1.9.1 for a sample) and conduct an internal review of the requirements. The review should include a search for any of the "potentially bad words"; words such as "usually," "approximately," "clearly," and easily" can indicate requirements problems. Resolve any pending problems or items mentioned as "TBDs." *Keep in mind that the requirements must be testable.*

12. Document the requirements for each software system in an SRS and an IRS. A format for the SRS can be selected and tailored from the samples given in Section 4.1.9.2. Where applicable, the SRS should include a table that maps the software requirements back to the system specification (for traceability). When approved, the SRS and IRS establish the allocated baseline.

13. Review and update the Software Development Plan (SDP) (see Section 5.2) as needed. If required, submit this document to the customer for review.

14. Conduct an Software Specification Review (SSR) (see Section 4.1.4.2). This is generally a formal review with the customer, focusing on completeness, consistency, clarity, and feasibility. Some projects may conduct more than one SSR, with different software systems reviewed at different SSRs. There are at least two situations that might warrant multiple SSRs. First, if requirements analysis progresses at different rates for different software systems, holding a single SSR might delay design work on those software systems that are ready to proceed early. Second, some software systems (or parts of software systems) might be designated "critical, meaning they can affect human safety or are a vital basis to

the rest of the software. Those critical software systems might warrant an early SSR. In any case, an entire software system should be reviewed as a whole. Dividing a software system between two SSRs can make it easy to leave the most difficult parts vague.

15. As the result of the SSR (possibly after completion of action items), obtain the customer's signature approval of the SRS/IRS. The requirements documents are then placed under formal configuration control.

Other methodologies may be used for requirements analysis. Some of the more common are structured systems analysis, user software engineering, and operational sequence diagrams. The choice of methodology is determined by the program.

[ISD48]

## 4.1.3 Organizing a Software Requirements Specification

**Primary Functions of an SRS**

- Facilitates communication among the customers, users, analysts, and designers.
- Establishes the basis for the contractual agreement and provides a standard against which compliance is measured.
- Clearly defines the required functionality of the software: the software must provide *all* required functions (functions that are not required should not be specified).
- Reduces development costs—only the specified requirements are designed for and built. Reduces the possibility of rework by raising issues early in the development lifecycle.
- Provides the relative necessity (essential, desired, optional, TBD) and the relative volatility (confirmed, changing, unconfirmed, TBD) of the specified requirements.
- Provides the basis for verifying compliance by supporting system testing activities.
- Provides the foundation and helps control the evolution of the system.
- Facilitates transfer and reuse. The SRS makes it easier to transfer the knowledge about a software product to new users and machines. Potential Users can review the SRS to determine how well the system meets their needs and also gauge the software for compliance to the specified requirements.

[ISD48]

*Note: Software requirements should not be confused with user needs. It is the software developer's responsibility to interpret the user needs (customers often refer to these needs as requirements) and translate them into the SRS.*

A common excuse for not specifying and documenting requirements is that "...the requirements will change anyway, so why bother documenting them...." Remember, in the early phases of the lifecycle, *the (documented) software requirements specifications are the requirements*. If they haven't been documented, there *are* no requirements! Requirements must be documented from the very beginning for the very reason that they do change: this is the best way to control and manage changing requirements. The fact is that requirements will change and evolve. The best that we can do as developers is to manage and control their evolution.

[DAV90]

**What *SHOULD* Be Included in an SRS**

- A complete, concise description of the entire external interface of the software system with its environment, including other software, communication ports, hardware, and human users. This includes two types of requirements:
  - Behavioral requirements define *what* the software system does. All the functions to be performed, all the inputs and outputs to and from the software system, and information concerning how the inputs and outputs will interrelate are described.
  - Nonbehavioral requirements define the attributes of the software system as it performs its job. They include a complete description of the software system's required level of efficiency, reliability, security, maintainability, portability, visibility, capacity, and standards compliance.

*[DAV90]*

**What *SHOULD NOT* Be Included in an SRS**

- Project requirements: staffing, schedules, costs, milestones, activities, phases, and reporting procedures (these belong in the software project management plan)
- Designs (these belong in the design documents)
- Product assurance plans: CM plans, Verification and Validation (V&V) plans, test plans, and QA plans

*[DAV90]*

**Attributes of a Well-Written SRS**

- **Correct**—Every requirement specified represents something that is required of the system to be built.
- **Unambiguous**—Every requirement specified has only one interpretation.
- **Complete**—Everything the software is supposed to do is included in the SRS.
- **Verifiable**—There should be a cost-effective method to check the final software system to ensure that every requirement specified has been met (testable).
- **Consistent**—1) No two parts of any requirement should have conflicting terms, 2) no two requirements should specify the system to exhibit conflicting characteristics, and 3) no two requirements should require the system to respond to conflicting timing patterns.
- **Understandable by Noncomputer Specialists**—It should serve as a communication tool between customers and developers.
- **Modifiable**—Requirements will change; the easier these changes are to make the better.
- **Traceable**—The origin of each requirement and its dependents is easily identified.
- **Annotated**—Guidance for development is provided to show the relative necessity and relative volatility of the requirements.
- **Usable**—Most importantly, the requirements should be produced in a manner that allows them to be used and to be of help to the developers.
- **Feasible**—Can this system be built?

*[DAV90]*

Remember, faulty (or unspecified) requirements will lead to errors in the system. Errors can be costly to the project, especially because errors often remain latent and are undetected until well after the stage in which they were made. The later in the development lifecycle a software error is detected, the more expensive it will be to repair. Typically, errors made in requirements specifications are because of incorrect facts, omissions, inconsistencies, and ambiguities. Using formal analysis and specification methods correctly can reduce the incidence of errors in the requirements phase. A careful review of the SRS for correctness, completeness, consistency, and the other attributes listed earlier can help errors be detected and addressed before further development has occurred.

Remember, when you feel that a textual or informal description will not suffice and has the possibility of being misunderstood, use a formal technique to specify your requirement.

**Formal Techniques for Specifying Requirements**

- Data Flow Diagrams (DFDs) (see Section 4.1.9.3)
- Entity Relationship Diagrams (ERDs) (see Section 4.1.9.3)
- Finite State Machines (FSMs) (see Section 4.1.9.3)
- Statecharts (see Section 4.1.9.3)
- Data Dictionaries
- Decision Tables and Decision Trees
- Object-Oriented Diagrams (OODs)
- Program Design Language (PDL)
- Requirements Engineering Validation System
- Requirements Language Processor
- Specification and Description Language
- PAISLey
- Petri Nets

There are many ways of organizing an SRS. See Section 4.1.9.2 for sample tables of contents. Any one of these can be modified and used to suit your particular project.

### 4.1.4 Reviews

#### 4.1.4.1 Internal Reviews

The software products developed during the software requirements phase should be reviewed internally before being delivered to the customer. Internal reviews provide early identification of potential problem areas and ensure that requirements and standards are being met. Internal reviews, also ensure that the software developers will receive a complete and usable product as the basis for their development.

Establish a checklist prior to the internal review of the software requirements. At a minimum, the checklist should address completeness, consistency, feasibility, testability, and understandability. A sample checklist for SRR is given in Section 4.1.9.1. The requirements review includes a review of the requirements contained in the SRS and the IRS.

#### 4.1.4.2 Software Specification Review

The objective of the SSR is to review the software requirements, interface requirements, and the operational concept. These are reviewed for technical adequacy, feasibility, and compliance with system requirements. A checklist for the review is given in Section 4.1.9.1.

The following items should be reviewed during the SSR for each software system:

- Functional overview of the software system. This should include inputs, processing, and outputs of each function.
- Overall software system performance requirements, including those for execution time, storage requirements, and similar constraints.
- Control flow and data flow between each of the software functions that comprise the software system.
- All interface requirements between the software system and all other configuration items both internal and external to the system.
- Qualification requirements that identify applicable levels and methods of testing for the software requirements that comprise the software system.
- Any special delivery requirements for the software system.
- Quality factor requirements, i.e., correctness, reliability, efficiency, integrity, usability, maintainability, testability, flexibility, portability, reusability, and interoperability.
- Mission requirements of the system and its associated operational and support environments.
- Functions and characteristics of the computer system within the overall system.
- Milestone schedules (see Section 5.2).

### 4.1.5 Summary

| | |
|---|---|
| **Inputs** | • Contractual Documents—SOW, Task Assignment, Proposal.<br>• SDP.<br>• System Requirements.<br>• Customer and User Interviews. |
| **Software Project Management Activities** | • SDP Review.<br>• Risk Management.<br>• Estimation and Tracking. |
| **Software Development Activities** | • Develop SRS.<br>• Develop IRS.<br>• Conduct SSR. |
| **Software Support Activities** | • CM—Place SRS under CM.<br>• QA—Review SRS, IRS, and SSR materials. |
| **Products** | • SRS.<br>• IRS.<br>• SDP—The SDP is completed at this time and will most likely be updated during later development phases.<br>• Requirements Allocation—System-level requirements are allocated to functions within software systems.<br>• Resource Allocation—Critical resources such as memory and processing time are allocated to software elements. If the allocation is done using a model, the model can be considered a product also, since it may be refined in the next phase. |
| **Review** | • SSR |

### 4.1.6 Tailoring to a Small Project

Each project is unique. Tailoring the information provided in this section is essential in defining and implementing the requirements analysis function to a specific project. Regardless of project size, the requirements analysis function needs to be performed. Only the level of detail and formality of the process and products vary among projects. Some of the factors to be considered are:

- Time
- Resources
- Complexity
- Contractual commitments
- Intended use of the product

For small projects where time and resources are very limited, it is impractical to attempt to provide a complete suite of documentation and evaluate the SRS at a formal review. However, it is essential to complete at least the following, in writing, before the software is designed:

- Briefly describe the objective of the project and include a few statements describing the external behavior of the software; this will help you to control the scope of development.
- List and briefly describe any constraints (standards, hardware limitations, security, availability of third-party software).
- List and briefly describe external interfaces for (all applicable):
  - Other software
  - User
  - Operators
  - Communications
- Identify, list, and describe the primary functional requirements being addressed by the system.
- Identify and describe the data flows into and out of the system at the context level and associate the primary data flows to the primary functions. The details provided regarding data flows can be extended according to the resources available and complexity of the problem being described.
- If applicable, identify and describe the primary operating states of the system and the events that the system responds to. Again, the details regarding the description of the states and the events can be extended according to the complexity of the problem being addressed.
- If a user interface is required, determine whether it is hierarchical or menubar-driven and whether it has pop-up windows. Describe events when windows are displayed and describe when windows are displayed concurrently. Describe what the windows look like, what events they respond to, and what they do in response to these events. Note: It is perfectly acceptable to "design" the user interface during the requirements phase, because you are describing *what the interface looks and feels like* (not *how* the interface accomplishes its functions). Remember, the user interface is the external interface of the software.
- Ensure that the issues you are specifying meet the attributes listed in Section 4.1.3.

- Briefly describe your plans (outline your test plan) to test the software after it is built to confirm that the requirements have been satisfied. Do not specify requirements for which you cannot prescribe a test to verify compliance.

Again, remember that the objective is to specify *What* the system will do. It is essential to obtain the customer's approval on what you have written—this will serve as a common point of reference during future development activities. The formal SSR can be replaced by an informal discussion about the requirements, culminating in agreement between the developers and the customer on the requirements that will be addressed during the development process.

## 4.1.7 Suggested Reference Material

Davis, Alan M., *Software Requirements: Analysis and Specification,* Prentice Hall, 1990.

Thayer, Richard, and Merlin Dorfman, *System and Software Requirements Engineering,* IEEE Computer Press Tutorial, 1990.

Yourdon, Edward, *Modern Structured Analysis,* Yourdon Press.

Coad, Peter, *Object Oriented Analysis,* Yourdon Press.

**Relevant Standards:**

- DOD-STD-2168
- DOD-STD-2167A
- MIL-STD-1521B
- DI-MCCR-80025
- DI-MCCR-80026
- GP 5-0-6 Attachment B
- DFI 5-0-53.3 Attachment C
- DFI-5-0-53.3 Attachment D
- DOD-STD-1703 (NS)
- ANSI/IEEE Std 830-1984

## 4.1.8 Cited References

[*ISD48*] *Software Engineering Handbook, Build 3,* Division 48, Information System Division, Hughes Aircraft Company, March 1992, p. 4-2.

[*ISD48*] *Software Engineering Handbook, Build 3,* March 1992, pp. 4-3–4-6.

[*ISD48*] *Software Engineering Handbook, Build 3,* March 1992, p. 4-1.

[*DAV90*] Davis, A., *Software Requirements: Analysis and Specification,* Englewood Cliffs, New Jersey: Prentice-Hall, 1990, p. 182.

[*DAV90*] Davis, p. 183.

[*DAV90*] Davis, p. 184.

[*DAV90*] Davis, p. 23.

### 4.1.9 Appendixes

#### 4.1.9.1 Checklists

The checklists provided in this section present a list of most of the issues that may need to be reviewed. It may not be necessary to address each of the items in the checklist. The goal of providing these checklists is for you to be aware of all the issues and for you to tailor these checklists to your project by consciously eliminating the items you do not need.

These checklists can be used to assess the completeness and correctness of software requirements and the readiness for an SSR. The checklists are used to assess the requirements themselves, the requirements documents, and the material for a requirements review.

| Are Requirements Complete? | |
|---|---|
| **Y/N** | **Check** |
| | Is all equipment identified (e.g., processors, memories, interface hardware, peripherals)? |
| | Are all requirements required by the SRS and IRS complete? |
| | If there are some TBD requirements, are they scheduled for completion as documented action items? |
| | Is there a data flow diagram (or similar notation) representing the processing sequence of the functional requirements, if required? |
| | Are all required data flows specified, including sources and destinations? |
| | Are any mathematical equations required as constraints on processing given or referenced? |
| | Are the accuracy/precision requirements defined? |
| | Are all required software system inputs and outputs allocated to processing sections? |
| | Are all software functions considered (e.g., loading, prestart tests, startup, modes of operation, operator interactions, normal terminations, restart, abnormal conditions, performance monitoring and tuning, test support features, recording, adaptation)? |
| | Are the processing requirements specified for recognized error conditions (e.g., hardware faults, I/O errors, computational errors, processing overload, buffer overflow, events failing to occur, out-of-sequence events, incorrect manual inputs)? |
| | Are communication conventions defined for each external interface (e.g., message headers, identifiers, sequence numbers, checksums)? |
| | Are all messages on each external interface completely defined (e.g., identification, type, name, description, size, frequency, direction of transfer, transfer rate, format, data units, data unit attributes)? |
| | If an executive is to be developed for this application, are the appropriate requirements specified? |
| | Are the resource requirements specified, including spare capacities? |
| | Are the test requirements defined (e.g., test levels and provisions to inject test data, adjust parameters, control or trace the execution of test runs, and extract test results)? |

| Are Requirements Consistent? | |
|---|---|
| **Y/N** | **Check** |
| | Is each object/function referred to by one unique name? |
| | Is each object/function defined by one set of characteristics that are not in conflict with one another? |
| | Are the requirements free of logical conflicts? |
| | Are the requirements free of timing conflicts? |
| | Is each requirement specified only once? |
| | Are all data and messages specified only once? |
| | Are acronyms and abbreviations defined and used consistently? |
| | Are mathematical equations defined consistently? |
| | Are the data flows consistent with the specified inputs and outputs of the requirement paragraphs? |
| | Are data flow notations used consistently? |
| | Are the order and frequency of messages consistent with the specified processing sequences and response times? |
| | Are the message data attributes consistent with the inputs and outputs of relevant requirement paragraphs? |
| | Are the loads used to allocate resource budgets consistently specified for all functions? |

| Is Implementation Feasible? | |
|---|---|
| **Y/N** | **Check** |
| | Do the data expected from external sources exist there? |
| | Are the data expected by outside destinations available? |
| | Are the data sent to outside destinations expected there? |
| | Are the requirements achievable with available technology? |
| | Are the necessary implementation tools available? |
| | On the basis of available facts or modeling information, are the performance requirements realistic (e.g., response times, accuracies, processing capacities)? |
| | Are the resource budgets realistic (e.g., CPU time, I/O utilization, memory, worst-case loads, data storage)? |
| | Has a specific system load been decided as the basis for performance tests? |
| | If a general-purpose executive is to be used, is it identified and factored into the performance requirements and resource budgets? |
| | Is the scope of requirements consistent with software estimates, schedules, and support facility plans? |

| Are Requirements Testable? | |
|---|---|
| **Y/N** | **Check** |
| | Are all requirements specified against the software (i.e., not against the hardware or the operator)? |
| | Can all requirements be verified by some (implicit, explicit, analytical, or empirical) means? |
| | Can test procedures be written against all requirements, using existing or planned resources? |
| | Can the test results be evaluated against predetermined acceptance criteria? |

| Are Requirements Understandable? | |
|---|---|
| **Y/N** | **Check** |
| | Are the major software functions described in relation to system operation? |
| | Are the requirements clearly stated? |
| | Do the requirements have unique interpretations? |
| | Is the terminology understandable and consistent? |
| | Is all notation defined? |
| | Is the glossary adequate? |
| | Are the data flow naming conventions defined? |
| | Is each requirement checked for clarity using the "potentially bad word list" in DFI 6-0-0.2, Attachment A? |

| SRS Checklist | |
|---|---|
| **Y/N** | **Check** |
| | Does the SRS adhere to the required format (e.g., GP 5-0-6 Attachment B) or contract Data Item Description (DID)? |
| | Is it internally consistent? |
| | Is it consistent with IRSs and higher level specifications? |
| | Will the customer be able to use this document to understand and train others in understanding the software requirements? |
| | Is the document ready to be delivered to the customer? |
| | Was it developed in accordance with the SDP, the software CM plan, and the software QA plan? |
| | Is the document consistent with the operational concept document? |

| IRS Checklist | |
|---|---|
| **Y/N** | **Check** |
| | Does the IRS adhere to the required format (e.g., contract DID)? |
| | Is it internally consistent? |
| | Is it consistent with other IRSs, the SRS, and higher level specifications? |
| | Will the customer be able to use this document to understand and train others in understanding the interface requirements? |
| | Is the document ready to be delivered to the customer? |
| | Was it developed in accordance with GP 5-0-6 and Information Systems Division Instruction DI 5-0-6, SDP, the software cm plan, and the software QA plan? |
| | Is the document consistent with the operational concept document? |

| SSR Presentation Material Checklist | |
|---|---|
| **Y/N** | **Check** |
| | Has MIL-STD-1521 (or other contract requirement) been reviewed to ensure that all required information is complete and available? |
| | Are the requirements ready to be presented at the SSR? |
| | Has the form of information presentation been established? |
| | Are the viewgraphs dated and numbered? |
| | Is the SSR plan complete in terms of agenda, facilities, handouts, recording of minutes, action items, and follow-up? |
| | Have success criteria been agreed upon with the customer? |

### 4.1.9.2 Sample Tables of Contents for SRS

**Example 1: DI-MCCR-80025A**

- 1.0 Scope
  - 1.1 Identification
  - 1.2 Software System Overview
  - 1.3 Document Overview
- 2.0 Applicable Documents
  - 2.1 Government Documents
  - 2.2 Non-Government Documents
- 3.0 Engineering Requirements (for a software system)
  - 3.1 Software System External Interface Requirements
  - 3.2 Software System Capability Requirements
    - 3.2.*x* Capability *x*
  - 3.3 Software System Internal Interfaces
  - 3.4 Software System Data Element Requirements
  - 3.5 Adaptation Requirements
    - 3.5.1 Installation-Dependent Data
    - 3.5.2 Operational Parameters
  - 3.6 Sizing and Timing Requirements
  - 3.7 Safety Requirements
  - 3.8 Security Requirements
  - 3.9 Design Constraints
  - 3.10 Software Quality Factors
  - 3.11 Human Performance/human Engineering Factors
    - 3.11.1 Human Information Processing
    - 3.11.2 Foreseeable Human Errors
    - 3.11.3 Total System Implications (e.g., training support, operational environment)
  - 3.12 Requirements Traceability
- 4.0 Qualification requirements
  - 4.1 Methods (demonstrations vs. test vs. analysis vs. inspection)
  - 4.2 Special (e.g., facilities, formulas, tools)
- 5.0 Preparation for Delivery
- 6.0 Notes (e.g., glossary, formula derivations, abbreviations, background information)

**Example 2: NASA's SFW-DID-08**

- 1.0 Introduction
  - 1.1 Identification
  - 1.2 Scope
  - 1.3 Purpose
  - 1.4 Organization
  - 1.5 Objectives
- 2.0 Applicable Documents
  - 2.1 Reference
  - 2.2 Information
  - 2.3 Parent Documents
- 3.0 User Scenarios
- 4.0 Requirements
  - 4.1 Functional and Performance Requirements
    - 4.1.*x* Function *x*
  - 4.2 Timing and Sizing Requirements
  - 4.3 Design Standards and Constraints
  - 4.4 Interface Requirements
  - 4.5 Programming Requirements
  - 4.6 Adaptation Requirements
    - 4.6.1 System Environment
    - 4.6.2 System Parameters
    - 4.6.3 System Capacities
  - 4.7 Database Requirements
  - 4.8 Quality Factors
    - 4.8.1 Correctness
    - 4.8.2 Reliability
    - 4.8.3 Efficiency
    - 4.8.4 Integrity
    - 4.8.5 Usability
    - 4.8.6 Maintainability
    - 4.8.7 Testability
    - 4.8.8 Flexibility
    - 4.8.9 Portability
    - 4.8.10 Reusability
    - 4.8.11 Interoperability
    - 4.8.12 Additional Factors
- 5.0 Qualification Requirements
  - 5.1 Qualification Methods
  - 5.2 Qualification Levels
  - 5.3 Acceptance Tolerance
  - 5.4 Tools/Facilities
  - 5.5 Special Qualification Requirements
- 6.0 Preparation for Delivery
- 7.0 Notes
- 8.0 Appendixes
- 9.0 Glossary

**Example 3: ANSI/IEEE STD-830-1984**

1.0 Introduction
- 1.1 Purpose of the SRS
- 1.2 Scope of the Product
- 1.3 Definitions, Acronyms, and Abbreviations
- 1.4 References
- 1.5 Overview of the Rest of the SRS

2.0 General Description
- 2.1 Product Perspective
- 2.2 Product Functions
- 2.3 User Characteristics
- 2.4 General Constraints
- 2.5 Assumptions and Dependencies

3.0 Specific Requirements
[Alternatives for 3.0 follow]

Appendixes
Index

**Alternative 1 for Section 3**

3.0 Specific Requirements
- 3.1 Functional Requirements
  - 3.1.1 Functional Requirement 1
    - 3.1.1.1 Introduction
    - 3.1.1.2 Inputs
    - 3.1.1.3 Processing
    - 3.1.1.4 Outputs
  - 3.1.2 Functional Requirement 2
  - ...
  - 3.1.*n* Functional Requirement *n*
  - ...
- 3.2 External interface requirements
  - 3.2.1 User Interfaces
  - 3.2.2 Hardware Interfaces
  - 3.2.3 Software Interfaces
  - 3.2.4 Communications Interfaces
- 3.3 Performance Requirements
- 3.4 Design Constraints
  - 3.4.1 Standards Compliance
  - 3.4.2 Hardware Limitations
  - ...
- 3.5 Attributes
  - 3.5.1 Availability
  - 3.5.2 Security
  - 3.5.3 Maintainability
  - 3.5.4 Transferability/Conversion
  - ...
- 3.6 Other Requirements
  - 3.6.1 Database
  - 3.6.2 Operations
  - 3.6.3 Site Adaptation

**Alternative 2 for Section 3**

- 3.0 Specific Requirements
  - 3.1 Functional Requirements
    - 3.1.1 Functional Requirement 1
      - 3.1.1.1 Specification
      - 3.1.1.1.1 Introduction
      - 3.1.1.1.2 Inputs
      - 3.1.1.1.3 Processing
      - 3.1.1.1.4 Outputs
      - 3.1.1.2 External Interfaces
      - 3.1.1.2.1 User Interfaces
      - 3.1.1.2.2 Hardware Interfaces
      - 3.1.1.2.3 Software Interfaces
      - 3.1.1.2.4 Communication Interfaces
    - 3.1.2 Functional Requirement 2
    - ...
    - 3.1.*n* Functional Requirement *n*
    - ...
  - 3.2 Performance Requirements
  - 3.3 Design Constraints
  - 3.4 Attributes
    - 3.4.1 Availability
    - 3.4.2 Security
    - 3.4.3 Maintainability
    - 3.4.4 Transferability/Conversion
    - ...
  - 3.5 Other Requirements
    - 3.5.1 Database
    - 3.5.2 Operations
    - 3.5.3 Site Adaptation

**Alternative 3 for Section 3**

- 3.0 Specific Requirements
  - 3.1 Functional requirements
    - 3.1.1 Functional Requirement 1
      - 3.1.1.1 Introduction
      - 3.1.1.2 Inputs
      - 3.1.1.3 Processing
      - 3.1.1.4 Outputs
      - 3.1.1.5 Performance Requirements
      - 3.1.1.6 Design Constraints
        - 3.1.1.6.1 Standards Compliance
        - 3.1.1.6.2 Hardware Limitations
        - ...
      - 3.1.1.7 Attributes
        - 3.1.1.7.1 Availability
        - 3.1.1.7.2 Security
        - 3.1.1.7.3 Maintainability
        - 3.1.1.7.4 Transferability/ Conversion
        - ...
      - 3.1.1.8 Other requirements
        - 3.1.1.8.1 Database
        - 3.1.1.8.2 Operations
        - 3.1.1.8.3 Site Adaptation
        - ...
    - 3.1.2 Functional Requirement 2
    - ...
    - 3.1.*n* Functional Requirement *n*
    - ...
  - 3.2 External Interface Requirements
    - 3.2.1 User Interfaces
      - 3.2.1.1 Performance Requirements
      - 3.2.1.2 Design Constraints
        - 3.2.1.2.1 Standards Compliance
        - 3.2.1.2.2 Hardware Limitations
        - ...
      - 3.2.1.3 Attributes
        - 3.2.1.3.1 Availability
        - 3.2.1.3.2 Security
        - 3.2.1.3.3 Maintainability
        - 3.2.1.3.4 Transferability/ Conversion
        - ...
      - 3.2.1.4 Other Requirements
        - 3.2.1.4.1 Database
        - 3.2.1.4.2 Operations
        - 3.2.1.4.3 Site Adaptation
        - ...
    - 3.2.2 Hardware Interfaces
    - ...
    - 3.2.3 Software Interfaces
    - ...
    - 3.2.4 Communications Interfaces

**Alternative 4 for Section 3**

3.0 Specific Requirements
- 3.1 Functional Requirement 1
  - 3.1.1 Introduction
  - 3.1.2 Inputs
  - 3.1.3 Processing
  - 3.1.4 Outputs
  - 3.1.5 External Interfaces
    - 3.1.5.1 User Interfaces
    - 3.1.5.2 Hardware Interfaces
    - 3.1.5.3 Software Interfaces
    - 3.1.5.4 Communications Interfaces
  - 3.1.6 Performance Requirements
  - 3.1.7 Design Constraints
    - 3.1.7.1 Standards Compliance
    - 3.1.7.2 Hardware Limitations
    - ...
  - 3.1.8 Attributes
    - 3.1.8.1 Availability
    - 3.1.8.2 Security
    - 3.1.8.3 Maintainability
    - 3.1.8.4 Transferability/Conversion
    - ...
  - 3.1.9 Other Requirements
    - 3.1.9.1 Database
    - 3.1.9.2 Operations
    - 3.1.9.3 Site Adaptation
    - ...
- 3.2 Functional Requirement 2
- ...
- 3.*n* Functional Requirement *n*

**Sample Outline of a Generic SRS**

1.0 Product Overview and Summary

2.0 Environments
- 2.1 Development
- 2.2 Operations
- 2.3 Maintenance

3.0 External Interfaces and Data Flow
- 3.1 User Displays and Report Formats
- 3.2 User Command Summary
- 3.3 High-Level Data Flow Diagrams
- 3.4 Logical Data Sources and Sinks
- 3.5 Logical Data Stores
- 3.6 Logical Data Dictionary

4.0 Functional Specifications

5.0 Performance Requirements

6.0 Exception Conditions and Exception Handling

7.0 Early Subsets and Implementation Priorities

8.0 Foreseeable Modifications and Enhancements

9.0 Acceptable Criteria
- 9.1 Functional and Performance Tests
- 9.2 Documentation Standards

10.0 Design Guidelines (hints and constraints)

11.0 Sources of Information

12.0 Glossary of Terms

**Sample Outline of a Users Manual**

1.0 Introduction
- 1.1 Product Overview and Rationale
- 1.2 Terminology and Basic Features
- 1.3 Summary of Display and Report Formats
- 1.4 Outline of the Manual

2.0 Getting Started
- 2.1 Sign-on
- 2.2 Help Mode
- 2.3 Sample Run

3.0 Modes of Operation
- 3.1 Commands
- 3.2 Dialogs
- 3.3 Features

4.0 Advanced Features

5.0 Command Syntax and System Options

6.0 Index

**Interface Requirements Specification (IRS)**
**Sample Table of Contents (adapted from DI-MCCR-80026A)**

1.0 Scope
- 1.1 Identification
- 1.2 System Overview
- 1.3 Document Overview

2.0 Applicable Documents
- 2.1 Specifications
- 2.2 Standards
- 2.3 Drawings
- 2.4 Other Publications

3.0 Interface Specification
- 3.x Interface Name
  - 3.x.1 Interface Requirements
    - a) Whether interfacing software systems are to execute concurrently or sequentially. If concurrently, the method of synchronization to be used
    - b) Communication protocol to be used for the interface
    - c) Priority Level of the Interface
  - 3.x.2 Data requirements
    For each data element:
    A Project-unique identifier for the data element
    A brief description of the data element
    The software system, hardware item, or other critical item that is the source of the data element
    The software system, hardware item, or other critical item that is the users of the data element
    Units of measure required for the data element (e.g., seconds, meters, kilohertz, etc.)
    The limit/range of values required for each data element (for constants provide the actual value)
    The accuracy required for the data element
    The precision required for the data element in terms of digits

4.0 Notes

5.0 Appendixes

## 4.1.9.3 Formal Analysis Techniques

### DATA FLOW DIAGRAMS (DFDs)

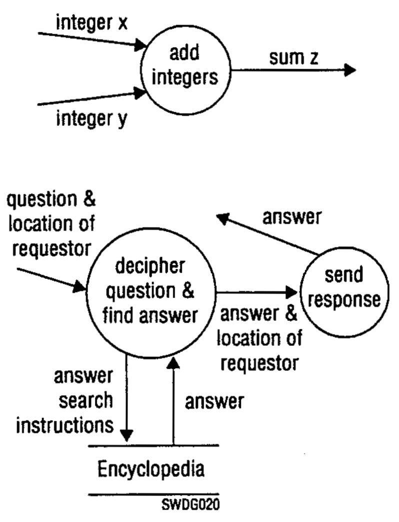
#### NOTATION

| | |
|---|---|
| process name label | A process that acts upon the data that are passed (flow) into it |
| Data source/sink name label | A data source or data destination (terminators) |
| Data flow label | Represents the data and the direction they flow in |
| Data store name label | A data store |
| Signal label | A signal |

#### DIRECTIONS

1. All names used in the Data Flow Diagram (DFD) should be unique. Using unique names makes it easier to refer to items in the DFD.
2. Meaningful names should be used to label the processes and data flows—DO NOT use names like "process data" to label a process or "data" to label a data flow. Remember, you should be able to look at a data flow and understand the relationship between the data and the processes in your system.
3. Arrows in a DFD represent the flow of data; remember, a DFD is NOT a flow chart and therefore does not present the relative order/sequence of events.
4. You cannot represent logical decisions in a DFD. (Drawing a diamond-shaped box with conditional arrows emerging from it implies an ordering of events that does not make sense in a data flow.)
5. The DFD should be developed from the top down; i.e., begin with the data flows into and out of the entire system (treat the system as a process), then define the system as a set of "lower level" processes with the data flows between them and progressively decompose the processes until they can be comfortably described in a process specification.

#### SIMPLE DATA FLOWS


integer x
add integers
sum z
integer y
question & location of requestor
answer
decipher question & find answer
send response
answer & location of requestor
answer search instructions
answer
Encyclopedia
SWDG020

#### COMMENTS

DFDs are used to describe the flow of data through a system. The system is represented as a set of processes connected by the data associated with those processes. DFDs are used to describe the system at various levels of abstraction.

**Advantages**

- Ideal for describing the transformation of data as they flow through the system.
- Simple notation makes it easy to understand.
- The system can be represented in increasing levels of detail with each progressive level in the hierarchy.

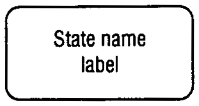
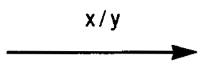
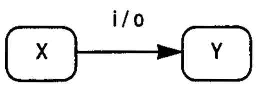
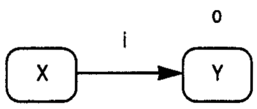
**Disadvantages**

- The order of processing cannot be implied or represented.
- Concurrency cannot be shown easily.
- It is easy to show too much detail—remember to stop when the process is easily understood.

SWDG020

## FINITE STATE MACHINES (FSMs)

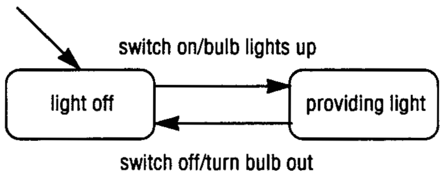
### NOTATION

| | |
|---|---|
|  | A state (mode of behavior) of the machine. |
|  | x: stimulus (input)<br>y: output |
|  | Mealy machine: The output is associated to the stimulus or input.<br>X: State X<br>Y: State Y<br>i: input<br>o: output |
|  | Moore machine: The output is associated to the current state (not the transition).<br>X: State X<br>Y: State Y<br>i: input<br>o: output |

### DIRECTIONS

1. Finite State Machines (FSMs) are also known as State Transition Diagrams (STDs).
2. An FSM is a hypothetical machine that can be in only ONE of a given number of states at a specific time.
3. All names used in the FSM should be unique, which makes it easier to refer to items in the FSM.
4. Meaningful names should be used to label the states and stimuli—DO NOT use names like "running process" to label a state or "signal" to label a stimulus. Remember, you should be able to look at an FSM and recognize the relationship between the states and the stimuli to understand the behavior of your system.
5. Arrows in an FSM represent the stimulus to the system; remember, an FSM is NOT a DFD.
6. The FSM responds to a stimulus (input) by generating an output and changing the state it is in. The output and the next state are functions of the current state and the input.
7. There are two types of FSM: the Moore machine and the Mealy machine (see alongside); our examples show the Mealy machine.

### A SIMPLE FSM



This FSM represents a light bulb. The angled (unlabeled) arrow points to the "light off" state; this is the "default entry state" of our hypothetical light bulb. When this machine receives a stimulus of "switch on," the "bulb lights up" as an "output" and the machine transitions to the "providing light" state. The machine will return to the "light off" state is when it receives a stimulus of "switch off" (while in the "providing light" state); it does so by turning the bulb out.

**Possible Applications**

- Simple user interfaces
- Parsers
- Control systems
- Interprocess communication protocols

### COMMENTS

**Advantages**

- FSMs are very useful in representing the behavior of a system when reacting to external stimuli.
- Unambiguous representation leaves little possibility of misleading the reader.
- FSMs use simple notation and are easy to understand.

**Disadvantages**

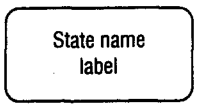
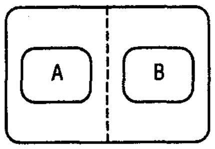
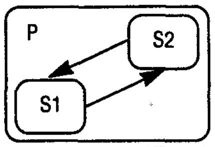
- FSMs can be in only ONE of a given number of states—every state has an "OR" relationship with every other state in the machine. Therefore, it is impossible to represent a machine that exists concurrently in another state.
- FSMs can become complex intertwined diagrams because the states cannot be "decomposed."
- Conditional state transitions are not possible.

*(State charts were developed to overcome many of the restrictions placed upon "traditional" STDs.)*

SWDG022

## STATECHARTS

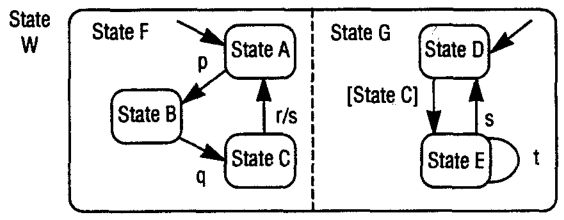
### NOTATION

| | |
|---|---|
|  | A state (mode of behavior) of the machine. |
|  | The dotted line signifies that the system exists in State A and State B concurrently. |
|  | State S1 and S2 are substates of State P (State P is the superstate; S1 and S2 are its subordinates). |
| $\xrightarrow{x / y}$ | x: stimulus (input)<br>y: output |
| $\xrightarrow{[\text{State Q}]}$ | Stimulus is based on the system entering State Q (conditional transition). |
| $\xrightarrow{p \mid q \ \& \ [\text{State m}]}$ | Stimuli may be "OR'd," "AND'd," and combined with state dependencies. |

### DIRECTIONS

1. Statecharts are extensions to FSMs (STDs).
2. A statechart represents the behavior of a system in terms of the states in which the system may exist.
3. All names used in the statechart should be unique; this makes it easier to refer to items in the statechart.
4. Meaningful names should be used to label the states and stimuli—DO NOT use names like "running process" to label a state or "signal" to label a stimulus. Remember, you should be able to look at a statechart and recognize the relationship between the states and the stimuli to understand the behavior of your system.
5. Arrows in a statechart represent the stimulus to the system; remember, a statechart is NOT a DFD.
6. The statechart responds to a stimulus (input) by generating an output (which may be NULL) and changing the state it is in.
7. A statechart should be developed from the top down—moving from higher levels of abstraction to greater levels of detail.

### A GENERIC STATECHART



The system is in State W; it is represented in terms of State F and State G (the dotted line indicates that it exists in these states concurrently). The unlabeled arrows point to the default entry (initial) state(s) of the system. When Stimulus p is received, it transitions to State B. When q is received, it transitions to State C; this in turn causes it also to transition from State D to State E. When r is received, it outputs s and transitions back to State A, with s causing State E to transition to State D.

**Possible Applications**

- Demonstration of concurrent processes
- User interfaces
- Parsers
- Control systems
- Interprocess communication protocols

### COMMENTS

**Advantages**

- Statecharts are very useful in representing the behavior of a system when it reacts to external stimuli.
- Unambiguous representation leaves little possibility of misleading the reader.
- Statecharts use simple notation and are easy to understand.
- It is possible to represent a machine that exists concurrently in another state. States may be "AND'd" as well as "OR'd."
- Complex systems can be represented in a series of statecharts that show progressively greater levels of detail—different parts of a statechart may show varying levels of detail.
- It is possible to show conditional state transitions; dependence on another state and/or combination of stimuli is possible (transitions are not just dependent on external stimuli).

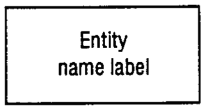
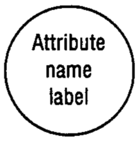
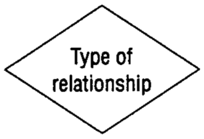
**Disadvantages**

- Diagrams can get complicated when they are not developed correctly.

SWDG023

## ENTITY-RELATIONSHIP DIAGRAMS (ERDs)

### NOTATION

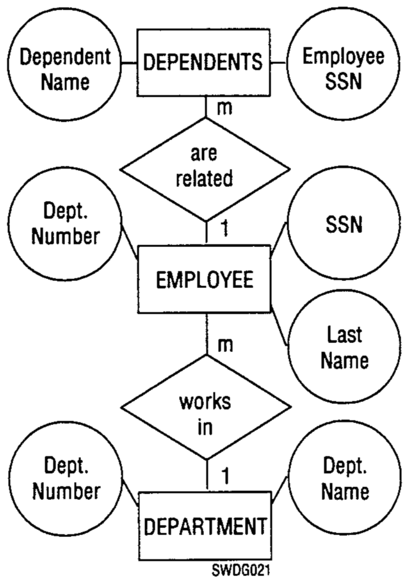
| | |
|---|---|
|  | An entity—a significant item about which information needs to be held |
|  | An attribute of an entity—the specific information that should be held |
|  | Represents the relationship between entities |
| ——— | Connects an entity with its attributes and relationships |
| m ——— 1<br>m ——— m<br>1 ——— 1 | Many-to-one<br>Many-to-many<br>One-to-one |

### DIRECTIONS

1. All names used in the Entity-Relationship Diagram (ERD) should be unique. Using unique names makes it easier to refer to items in the DFD. Use meaningful names when labeling the relationships and entities.
2. Entities:
   - Must have multiple occurrences or instances.
   - Each instance must be uniquely identifiable from other instances.
   - If an entity cannot be uniquely identified, it may not be an entity.
   - All entities are nouns, but not all nouns are entities.
   - If an entity has no attributes, it may be only an attribute.
3. Relationships are two-directional, significant associations between two entities or between an entity and itself. Read a relationship first in one direction and then in the other. There are three types of relationships: many-to-one, many-to-many, and one-to-one.
4. Attributes:
   - Information about an entity that needs to be known or held.
   - Describe an entity by qualifying, identifying, classifying, quantifying, or expressing the state of the entity.
   - Represent a type of description or detail, not an instance.
   - Should always be broken down into their lowest meaningful components.
   - Can have only a single value for each entity instance.
   - Cannot be derived or calculated from the existing value of other attributes.
   - If an attribute has attributes of its own, it is really an entity.

*Note: This example uses the Chen notation.*

### SIMPLE ERD



### COMMENTS

ERDs are used to represent the objects manipulated by a system, showing their attributes and their interrelationships.

**Advantages**

- Simple notation makes it easy to understand.
- Simple notation is easy to develop and refine.
- The system can be represented in increasing levels of detail.
- An ERD is independent of the hardware or software to be used in the implementation. It can be mapped to a hierarchical, network, or relational database.

**Disadvantages**

- Large ERDs are difficult to manage and understand.

**Applications**

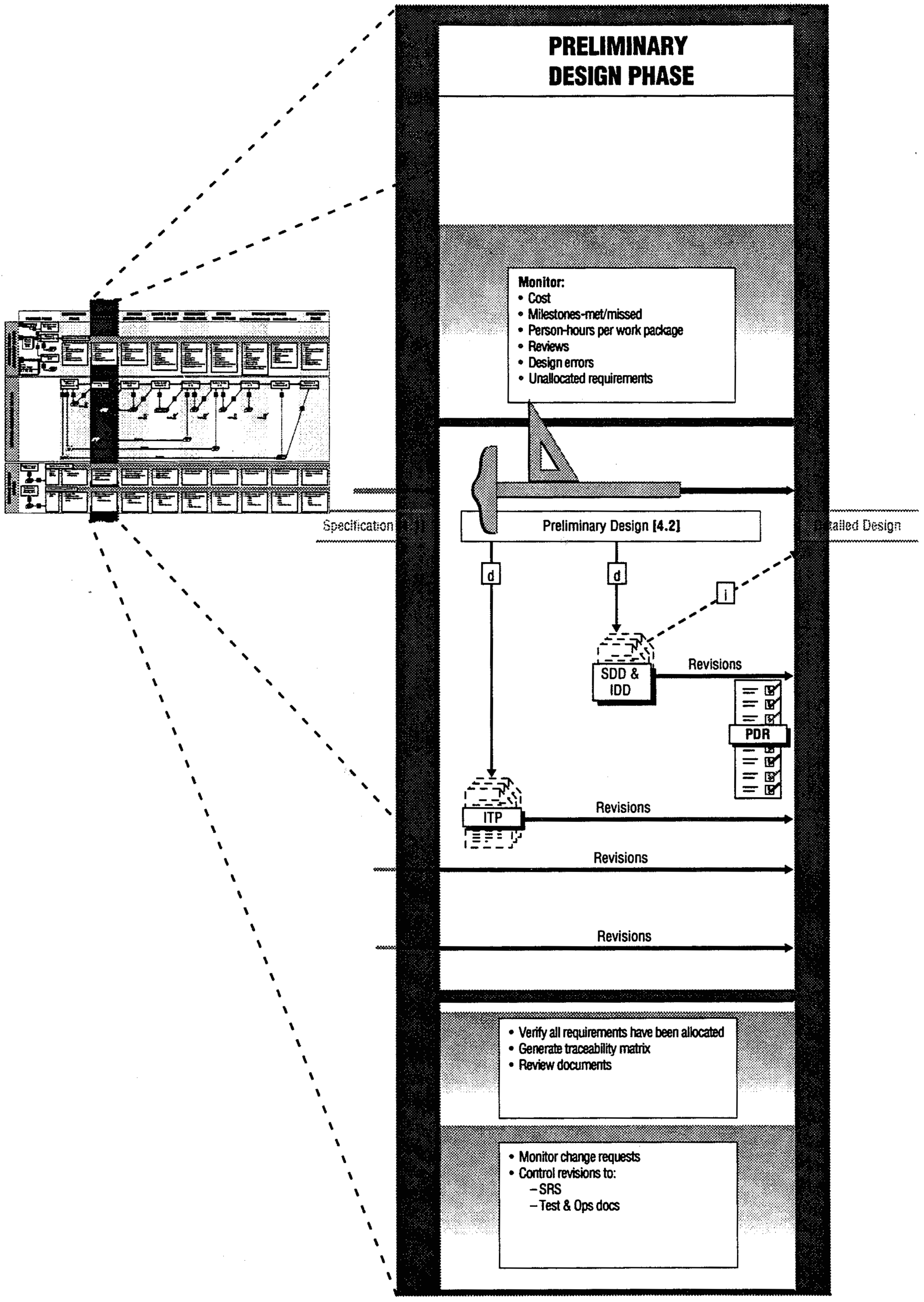
- Widely used for conceptual data modeling
- Used for designing databases—entities translate into tables, the attributes into the columns, etc.
- Used in modeling of real-world entities to ascertain interrelationships

SWDG021

## *Section 4.2*

# Preliminary Design Phase

## Contents


PRELIMINARY DESIGN PHASE
Monitor:
• Cost
• Milestones-met/missed
• Person-hours per work package
• Reviews
• Design errors
• Unallocated requirements
Specification
Preliminary Design [4.2]
Detailed Design
d
d
i
SDD & IDD
Revisions
PDR
ITP
Revisions
Revisions
Revisions
• Verify all requirements have been allocated
• Generate traceability matrix
• Review documents
• Monitor change requests
• Control revisions to:
– SRS
– Test & Ops docs

### 4.2.1 Introduction

The activities performed during the preliminary design phase include high-level design (also known as the architectural design) of the software, initial development of the test plans (see Section 4.6.7.1), documentation to record the software design, and the Preliminary Design Review (PDR). Figure 4.2.1-1 represents the process flow diagram for the preliminary design phase of software development.

In the requirements analysis phase, an SRS was developed to describe *what* the software will do; in the preliminary design phase, the outline of *how* it will be done is established.

The activities during this phase should follow a well-defined, systematic approach as defined in the SDP, (see Section 5.2), no matter which design methodology is used). The preliminary design is the framework for the more detailed design decisions that will follow in the subsequent phase. All future development activity will be based on the work done in this phase.

Test planning is performed in this phase to ensure that the software can be tested. Preliminary versions of the user manual and maintenance manual are developed in this phase to ensure that once built, the software can be used and operated in the manner that was intended.

**Primary Functions of a Preliminary Design**

- Partition the software system into its major structural and functional subsystems.
- Develop the operating procedures.
- Identify and evaluate all alternative design strategies.
- Define and associate all inputs to the system and outputs from the system to specific subsystems.
- Define all inputs and outputs between each of the software subsystems.
- Define the algorithms needed by the subsystems.
- Outline error processing and recovery strategies.
- Ensure that all requirements are being met by the software subsystems.
- Identify all existing software that will be used by this system.

### 4.2.2 General Methodology for Developing a Preliminary Design

The following steps describe a general methodology to develop a preliminary design:

1. Develop a high-level design using a well-defined design methodology. During the design process:

   a. Identify and select the software subsystems.
   b. Allocate the functional requirements to software subsystems.
   c. Allocate all other requirements (interface, design, programming, performance, and quality) to the software subsystems whenever applicable.
   d. Identify and select Commercial Off-the-Shelf (COTS), Government-furnished, proprietary, or reusable software to meet some or all of the allocated requirements.
   e. Identify functional control and data flow among the software subsystems.

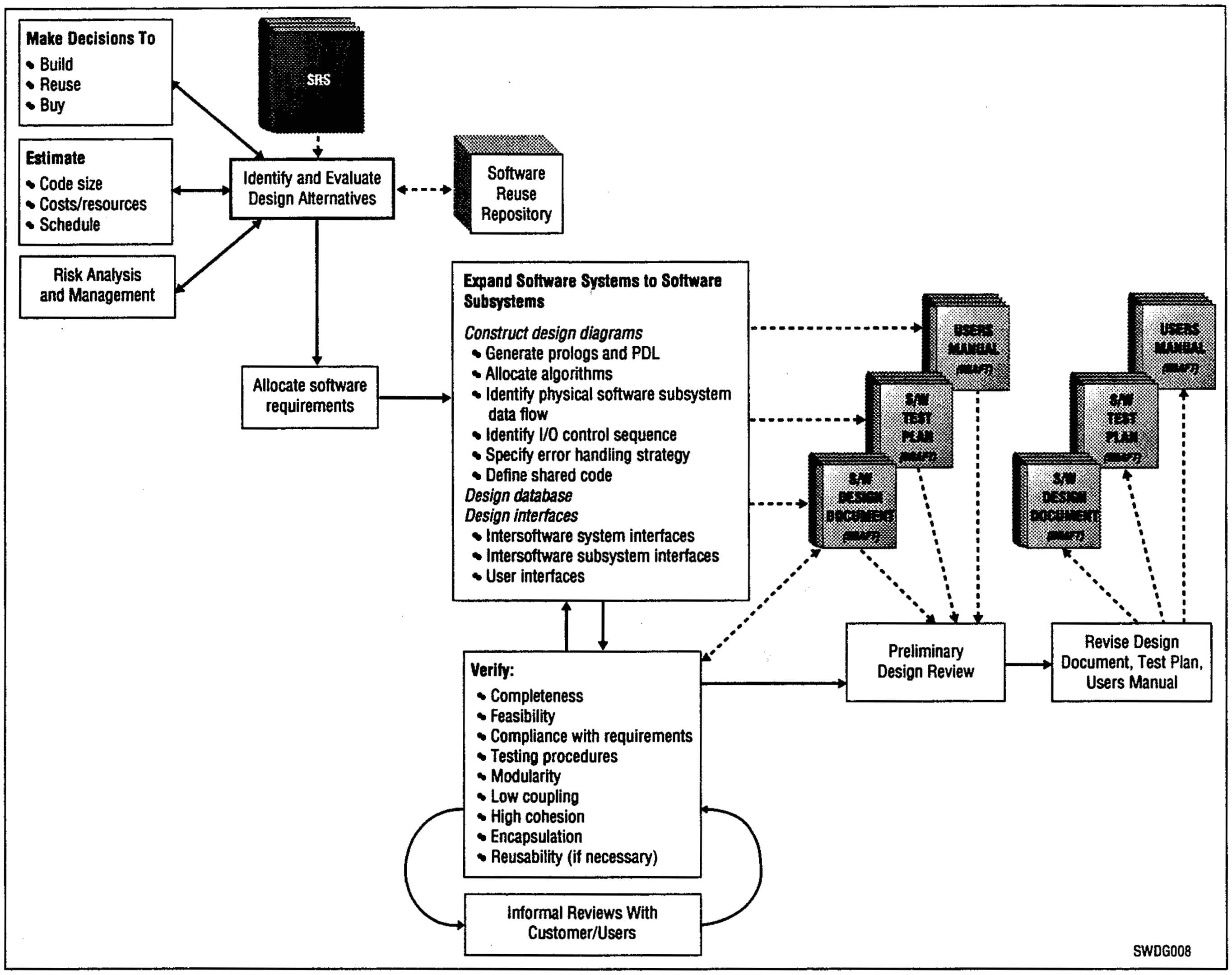

Make Decisions To
• Build
• Reuse
• Buy
Estimate
• Code size
• Costs/resources
• Schedule
Risk Analysis and Management
SRS
Identify and Evaluate Design Alternatives
Software Reuse Repository
Allocate software requirements
Expand Software Systems to Software Subsystems
Construct design diagrams
• Generate prologs and PDL
• Allocate algorithms
• Identify physical software subsystem data flow
• Identify I/O control sequence
• Specify error handling strategy
• Define shared code
Design database
Design interfaces
• Intersoftware system interfaces
• Intersoftware subsystem interfaces
• User interfaces
USERS MANUAL (DRAFT)
S/W TEST PLAN (DRAFT)
S/W DESIGN DOCUMENT (DRAFT)
Verify:
• Completeness
• Feasibility
• Compliance with requirements
• Testing procedures
• Modularity
• Low coupling
• High cohesion
• Encapsulation
• Reusability (if necessary)
Informal Reviews With Customer/Users
Preliminary Design Review
Revise Design Document, Test Plan, Users Manual
SWDG008

**Figure 4.2.1-1. Preliminary Design Phase Process Flow**

f. Select and describe database(s) used by the software subsystems.

| For Each Software Subsystem |
|---|
| • Describe the inputs to and outputs from the software subsystem.<br>• Describe data (local and global) required by the software subsystem.<br>• Describe events processed by the software subsystem.<br>• Describe timing and sequencing conditions that cause the software subsystem to be executed.<br>• Describe algorithms, special control features, error detection, and recovery processing of the software subsystem. |

*Note: It may be useful to use a design description language (i.e., PDL, structure graph, or other formal syntax) when describing the high-level design.*

2. Document the high-level design in a preliminary design version of the Software Design Document (SDD) for each system. If required, submit the design documentation to the customer for review.

3. Develop a preliminary Interface Design Document (IDD) to document the preliminary design for the interfaces external to each software system. If required, submit the design documentation to the customer for review.
4. Document the rationale for key design decisions in a Software Development File (SDF), see Section 4.2.8.1.
5. Plan and document testing procedures for the software system and subsystem in one or more software test plans. Planning includes development of test requirements, responsibilities, and schedules. If required, submit the plans to the customer for review. (Refer to Section 5.5.3, "Scheduling Multiple Builds.")
6. Estimate and/or measure resource utilization; for each software subsystem for each of the budgets allocated in the requirements analysis. Verify the high-level design's implementation of allocated resource budgets using a documented system load. Establish a resource budget plan, including management reserves, for each phase and report actual (and predicted) vs. budgeted utilizations.
7. Develop preliminary versions of the operations and support documents; and, if required, submit them to the customer for review. Normally, the required manuals are:
   a. Computer System Operators Manual
   b. Software Users Manual
   c. Software Programmers Manual
8. Perform a risk analysis; on the developed design plan. Some relevant questions are: What are the plan's weak spots? Have alternatives been determined for each, if they are needed? How much risk is involved? Refer to Section 5.6 for information on risk management.
9. Conduct an internal review and then a PDR of the products developed during this phase at the end of the preliminary design phase.

**Selecting a Design Methodology**

Selecting the "correct" design methodology for a project is difficult; there is no fixed formula that points to a specific methodology. The following is a list of some of the factors affecting the selection of a design methodology:

- If the project consists of modifying existing software, you are encouraged to use the same methodology that was used in the original development, which gives a more consistent design and documentation. However, a different methodology may be used if the new project will make major modifications, implement one or more completely new major functions, or modify software that is expected to undergo many more modifications during a long life. Remember, it is more likely that existing code will need to be modified when using a different methodology.
- If the project is staffed with people predominantly familiar with a specific methodology, it may make sense to use it. No methodology is strictly mechanical; each depends on the intelligence, creativity, and attitude of the implementors. *It is better to do a good job of structured design than a poor job of object-oriented design, and vice versa.*
- If the customer has expressed a preference for (or has required) a specific methodology, or if the proposal is based on a specific methodology, changing to a different one would require a good justification.
- A better set of tools may exist (or may already be in-house) for one methodology than exists for another.

Two popular design methodologies are structured design and object-oriented design, although many others (and many variants) exist.

**Structured design**—uses data flows in developing structure charts that show the interaction of software elements. Evaluation criteria such as coupling, cohesion, information hiding, and scope of effect are used to judge the quality of the charts and to guide the design team in revising or improving the design. The team iterates between data flows and structure charts at successively lower levels.

**Object-oriented design**—encapsulates data and the operations performed on the data into "objects." Objects are viewed functionally by other software and can be used without knowledge of their internal data structures or operations.

In both design approaches, the goal is to encapsulate information about data structures and operations, to prevent errors from propagating through the software. Because objects consist of both data and operations, object-oriented design is promoted as fostering the ability to reuse software. Both methodologies (and others) are described in numerous books, articles, and courses. (Structured design; has been described by Yourdon and Constantine, Hatley and Pirbhai, and DeMarco. Object-oriented design; has been described by Booch, Coad, Meyer, Wasserman, Rumbaugh, et al.)

When descriptions of methodologies are required, as in a proposal or SDP, it may be appropriate to describe techniques such as project notebooks, walkthrough, design standards, and document review and control procedures, as well as the design methodology.

[*ISD48*]

### 4.2.3 Organizing a Software Design Document

There are many ways of organizing an SDD. Section 4.2.8.2 presents sample tables of contents for an SDD.

### 4.2.4 Reviews

#### 4.2.4.1 Internal Review

Internal reviews provide early identification of potential problem areas and ensure that requirements and standards are met. See Section 4.2.8.1 for details.

#### 4.2.4.2 Preliminary Design Review

This section establishes guidelines for planning the presentation of software-related PDR material. "Presentation material" is the set of viewgraphs or other visual media used in conducting a formal review. Presentation material consists of summary data, created during the design process or extracted from software development and management plans (reformatted as necessary), for review by an audience that includes both technical and management representatives. The purpose of a PDR is to formally review, with the contracting agency, the high-level software design, the Software Test Plan (STP), and the preliminary versions of the operation and support documents.

Sample checklists used to assess the results of the preliminary design are provided in Section 4.2.8.1. These checklists address the material to be covered at the PDR. The review procedures in this section are meant to be tailored to individual project needs and requirements. [*ISD48*]

### 4.2.5 Summary

| | |
|---|---|
| Inputs | • SRS<br>• IRS<br>• SDP<br>• Requirements Allocation<br>• Resource Allocation |
| Software Project Management Activities | • Revision of SDP<br>• Risk Management<br>• Estimation and Tracking |
| Software Development Activities | • Development of Preliminary Design<br>• Development of Interface Design Specification |
| Software Support Activities | • CM<br>• QA |
| Products | • SDD<br>• IDD<br>• SDFs<br>• STP<br>• Resource Usage Plan<br>• Requirements Traceability<br>• Design Walkthrough Reports<br>• Operations and Support Documents<br>– Computer System Operators Manual<br>– Software User's Manual<br>– Software Programmers Manual |
| Review | • PDR |

### 4.2.6 Tailoring to a Small Project

Each project is unique. Tailoring the information provided in this section is essential in defining and implementing the preliminary design function to a specific project. Regardless of project size, the preliminary design function needs to be performed. Only the level of detail and formality of the process and products vary among projects. Some of the factors to be considered are:

- Time
- Resources
- Complexity
- Contractual commitments
- Intended use of the product

It is impractical to try produce a complete set of documentation and to conduct formal reviews in a small project where time and resources are very limited. It is essential, however, to describe high-level design in writing and to describe at the high level *how* the software will meet the requirements that were set. At a minimum:

- Describe the overall strategy for implementing the software.
- Determine whether it is based on existing software and identify software you plan on reusing. Describe the modification procedures.

- Determine whether it is procedure based or object oriented; identify the procedures and/or classes (software components).
- Describe how the requirements have been allocated to the software components.
- If a user interface is required, describe how the interface responds to the events that were specified.
- Describe how the primary functions satisfy the requirements they are assigned.
- If necessary (depending on the complexity of the software) describe the functions/classes subordinate to those described in the high-level design.
- Briefly describe how the software will be tested and outline your test plan.

The goal is to communicate the design approach to the customer and other developers. Remember to have the customer approve the design approach before proceeding to the next phase. The formal PDR can be replaced with an informal presentation of the design approach describing the issues above, culminating in agreement between the customer and developers on all the design issues being addressed.

## 4.2.7 Suggested Reference Material

Freeman, Peter, and Anthony Wasserman, *Tutorial on Software Design Techniques*, IEEE Computer Society Press.

Coad, Peter, *Object Oriented Design*, Yourdon Press.

Page-Jones, Meiller, Structured Design.

**Relevant Standards:**

- DOD-STD-2167A
- MIL-STD-1521B
- Data Item Descriptions (DIDs):
  - DI-MCCR-80018—Computer System Operators Manual (CSOM)
  - DI-MCCR-80027—Interface Design Document (IDD)
  - DI-MCCR-80012—Software Design Document (SDD)
  - DI-MCCR-80014—Software Test Plan (STP)
  - DI-MCCR-80019—Software User's Manual (SUM)
- GP 5-0-6 Attachment B
- DFI 5-0-6.1—Software Design Process Practices
- DFI 5-0-6.3—Internal Walk-throughs
- DFI 5-0-6.4—Software Development File (SDF)
- DOD-STD-1703 (NS)—Software Product Standards
- ANSI/IEEE Std 1016-1987—IEEE Recommended Practice for Software Design Descriptions

### 4.2.7.1 Cited References

[*ISD48*] *Software Engineering Handbook, Build 3,* Division 48, Information System Division, Hughes Aircraft Company, March 1992, p. 5-1–5-5.

[*ISD48*] *Software Engineering Handbook, Build 3,* p. 5-8.

[*ISD48*] *Software Engineering Handbook, Build 3,* Appendix A.

## 4.2.8 Appendixes

### 4.2.8.1 Checklists

The checklists provided in this section present a list of most of the issues that may need to be reviewed. It may not be necessary to address each of the items in the checklist. The goal of providing these checklists is for you to be aware of all the issues and for you to tailor these checklists to your project by consciously eliminating the items you do not need.

These checklists can be used to assess the completeness and correctness of preliminary design and the readiness for a PDR. The checklists are used to assess the design itself, the preliminary design documents, and the material for a PDR.

| **Design Walkthrough Checklist** | |
|---|---|
| **Y/N** | **Check** |
| | Are all design materials complete? |
| | Does the design adhere to the project's design standards? |
| | Is the design represented in the required format? |
| | Have all required sections of the design document been produced? |
| | Is a requirements trace available? |
| | Does the design implement all requirements allocated to this software system? |
| | Does the design address capabilities not specified in the requirements? |
| | Have possible error conditions been adequately addressed? |
| | Have interfaces been adequately defined and addressed? |
| | Have requirements other than functional requirements (e.g., quality factors, maintainability) been adequately addressed? |
| | Has usage of key resources been estimated? |

| General Preparation | |
|---|---|
| Y/N | Check |
| | Has the contract-imposed standard (e.g., 1679A, 2167, 490) been reviewed to ensure that all required design information is complete and available? |
| | Have the contract SOW and SDP been reviewed for design review applicability and requirements? |
| | For design information that is incomplete or unavailable, have written waivers been arranged with the program manager/customer? |

| Is the Software Design Complete? | |
|---|---|
| Y/N | Check |
| | Does the design contain all information required for the preliminary design in the SDD? |
| | Does the design contain all information required in the preliminary IDD? |
| | Has a thread for each operational transaction been developed? |
| | Have all operating system/executive interfaces been defined? |
| | Have all I/O techniques and interfaces between the operating system and scheduled elements been defined? |
| | Have all dependencies and interfaces between the operating system and scheduled elements been defined? |
| | Have all design limitations, including technical risk, been addressed and evaluated? |
| | Have all open issues been addressed and documented? |
| | Has the reusable software been identified and the approach for development established? |
| | Are significant analyses and decision rationales documented in the design documentation? |

| Is the Systems Design Complete? | |
|---|---|
| Y/N | Check |
| | Are software requirements complete in written form (i.e., in the SRS)? |
| | Have all hardware components been selected? |
| | Are all hardware components available? |
| | Have all commercial, Government-furnished, and reusable software products been selected, evaluated, and ordered? |

| Is the Systems Design Complete? (Continued) | |
|---|---|
| **Y/N** | **Check** |
| | Have implied requirements, software-imposed requirements, and carryover requirements (in the case of lifted software) been identified, documented, and forwarded to the program manager/ customer? |
| | Has a software requirements analysis internal review been conducted? |
| | Have preliminary operations and support manuals been prepared? |

| Does the Design Meet the Requirements? | |
|---|---|
| **Y/N** | **Check** |
| | Does the design implement all functional, interface, performance, quality, sizing/timing, and adaptation requirements? |
| | Has a *cross-reference index assigning each software requirement* to a preliminary design software element been completed? |
| | Does the design implement all other contractual requirements? |
| | Can all elements of the design can be traced to SRS or IRS requirements? |

| Is the Design Feasible? | |
|---|---|
| **Y/N** | **Check** |
| | Has an operational test scenario been defined and verified? |
| | Have sizing and timing estimates been completed to indicate the software will meet reserve requirements? |
| | Are thread response times within requirements? |
| | Can the design be built within schedule and cost given the project constraints (i.e., project schedule, software budget, established software development environment, and identified operational hardware configuration)? |
| | Have all new algorithms been prototyped? |
| | Is the design based on known, proven principles? |
| | Have test points been identified that indicate the design is testable? |
| | Have the Human-Machine Interface (HMI) features been prototyped? |
| | Does the support software to be used on the project (i.e., compilers, system generators) support the design? |
| | Have technical risk and long-lead items been evaluated and documented? |

**Is the Design Good?**

| Y/N | Check |
|---|---|
| | Is the level of design appropriate for this review? |
| | Does the design comply with standards given in the SDP? |
| | Within a margin of reasonableness, can the design be described as being no more and no less than what is required? |
| | Is the design implemented in accordance with the methodology and tools specified in the SDP? |
| | Is the design internally consistent? |
| | Is the design understandable? |
| | Does the design address all necessary control features (scheduling, sequencing, interrupt processing, special control)? |
| | Does the design address extreme conditions (error processing, startup, recovery, startover)? |
| | Does the design address the identification and description of implementation for recursion/re-entry? |

**Preliminary Design Document Checklist**

| Y/N | Check |
|---|---|
| | Does the design document adhere to the required format? |
| | Is the design document internally consistent? |
| | Is the design document consistent with other SDDs, IDDs, and the parent SRS and IRS? |
| | Will the customer be able to use this document to understand and train others in understanding the software system design? |
| | Is the document ready to be delivered to the customer? |
| | Is the document developed in accordance with the SDP, software CM plan, and the software QA plan? |
| | Is the document consistent with Software Users Manual? |

**Software Test Plan Checklist**

| Y/N | Check |
|---|---|
| | Are the tests feasible? |
| | Can all tests be conducted within budget and time constraints? |
| | Do all tests have objective success criteria? |
| | Are all test scenarios defined and achievable? |

| Software Test Plan Checklist (Continued) | |
|---|---|
| Y/N | Check |
| | Are test facilities planned and achievable? |
| | Are special (nondeliverable) test equipment and software identified, evaluated, and available? |

| STP—Document Quality | |
|---|---|
| Y/N | Check |
| | Does the document adhere to the required format (e.g., GP 5-0-6, contract DID)? |
| | Is it internally consistent? |
| | Is it consistent with the parent SRS, i.e., is it traceable? |
| | Is the document consistent with the preliminary design? |
| | Is the document ready to be delivered to the customer? |
| | Was it developed in accordance with the SDP, the SWHB, the software CM plan, and the software QA plan? |

| Is the STP Complete? | |
|---|---|
| Y/N | Check |
| | Are all requirements allocated to specific tests or phases? |
| | Have test responsibilities been identified? |
| | Do test plans establish adequate test criteria/coverage? |
| | Have test limitations been addressed and documented? |

| Operations and Support Manuals | |
|---|---|
| Y/N | Check |
| | Do the manuals adhere to the required format (e.g., GP 5-0-6, contract DID)? |
| | Are they internally consistent? |
| | Are they consistent with preliminary design, SDD, STP, and other user manuals? |
| | Are they understandable? |
| | Are the documents ready to be delivered to the customer? |
| | Have normal and extreme operational conditions been addressed? |

| Operations and Support Manuals (Continued) | |
|---|---|
| **Y/N** | **Check** |
| | Have normal and extreme support conditions been addressed? |
| | Are the content of the documents appropriate for the end user? |
| | Were the documents developed in accordance with the SDP, the SWHB, the software CM plan, and the software QA plan? |

| PDR—Presentation Material | |
|---|---|
| **Y/N** | **Check** |
| | Have MIL-STD-1521 and other contractual requirements been reviewed to ensure that all required information is complete and available? |
| | Is the preliminary design ready to be presented at PDR? |
| | Has the form of presentation been established? |
| | Have success criteria been agreed upon with the customer? |
| | Is the PDR plan complete in terms of agenda, facilities, handouts, recording of minutes, action items, and follow-up? |

*[ISD48]*

### 4.2.8.2 Sample Tables of Contents

*Note: This appendix contains samples of design documents tables of contents. Please refer to Section 4.6.7.1 for samples of test plans.*

**Adapted from NASA-DID-P300**

1.0 Introduction

2.0 Related documentation

3.0 Design approach and trade-offs
Describe the rationale and trade-offs and other design considerations influencing the major design decisions of the software.

4.0 Architectural design description
Describe the logical and functional design of the software using:
- Logical or functional decomposition
- Description of subordinate software subsystems including their inputs and outputs
- Relationships and interactions between the software subsystems
- Logical data design—the conceptual schema
- Entity/data identification and relationships
- Timing and sequencing
- Implementation constraints

5.0 External interface design
This section will evolve into the IDD
5.1 Describe the design for each interface in the SRS in terms of:
  - Information description
  - Initiation criteria
  - Expected response
  - Protocol and conventions
  - Error identification, handling, and recovery
  - Queueing
  - Implementation constraints

5.2 Interface allocation
This section will allocate the software's external interface requirements to the appropriate lower level elements. Use a table or graphics to increase clarity. Ensure that all external requirements, including performance, are allocated.

6.0 Requirements allocation and traceability
This section documents the allocation of the software requirements to the appropriate software subsystems. Show the traceability of all requirements, including performance and constraints for this software, to the design presented above. Explicitly identify any derived requirements.

7.0 If functionality is to be provided in incremental builds, specify the requirements and functions associated with each build.

8.0 Abbreviations and acronyms

9.0 Glossary

10.0 Notes

11.0 Appendixes

**Design Document**
**Adapted from DI-MCCR-80012A**

1.0 Scope
- 1.1 Identification
- 1.2 System overview
- 1.3 Document overview

2.0 Referenced Documents

3.0 Preliminary Design
- 3.1 Software system overview
  - 3.1.1 Software system architecture
  - 3.1.2 Software system states and modes
  - 3.1.3 Memory processing and time allocation
- 3.2 Software system design specification
  - 3.2.x Software subsystem x
    - 3.2.x.y Sublevel software subsystem

4.0 Detailed Design
- 4.x Software subsystem x
  - 4.x.y Software module
    - 4.x.y.1 Software module design specification/constraints
    - 4.x.y.2 Software module design
      - a. Input/output data elements
      - b. Local data elements
      - c. Interrupts and signals
      - d. Algorithms
      - e. Error handling
      - f. Data conversion
      - g. Use of other elements
        - Other software modules
        - Shared data stored in global memory
        - Input and output buffers, including message buffers
      - h. Logic flow
      - i. Data structures
      - j. Local data files or database
      - k. Limitations

5.0 Software System Data
- a. For data elements internal to the software system
  - i) Name of the data element
  - ii) A brief description
  - iii) The units of measure, such as knots, seconds, meters, and feet
  - iv) The limit range of values required for the data element (for constraints provide the actual value)
  - v) The accuracy required for the data element
  - vi) The precision/resolution in terms of significant digits
  - vii) For real-time systems, the frequency at which the data elements are calculated/refreshed, e.g., 10 KHz, 50 Msec
  - viii) Legality checks performed on the data element
  - ix) The data type, such as integer, ASCII, fixed, real, enumeration
  - x) The data representation format
  - xi) The software module where the data element is set or calculated
  - xii) The software module where the data element is used
  - xiii) The data source from which the data are supplied, e.g., databases or data files, global common, local common, compool, datapool, parameter
- b. For data elements of the software systems external interfaces
  - i) Identify the data element
  - ii) Identify the interface by name and project-unique identifier
  - iii) Reference the IDD in which the external interface is described

6.0 Software System Data Files
- 6.1 Data file to software subsystem/software module reference
- 6.x Data file name

7.0 Requirements Traceability

8.0 Notes

9.0 Appendixes

**Detailed Design Document**
**Reference: NASA-DID-P400**

1.0 Introduction

2.0 Related Documentation

3.0 Detailed Design Approach and Tradeoffs

4.0 Detailed Design Description
- 4.1 Compilation Unit Design and Traceability to Architectural Design
- 4.2 Detailed Design and Compilation Units

5.0 External Interface Detailed Design
- 5.1 Interface Allocation Design
- 5.2 Physical Interface Design

6.0 Coding and Implementation Notes

7.0 Firmware Support Manual

8.0 Abbreviations and Acronyms

9.0 Glossary

10.0 Notes

11.0 Appendixes

**Software Subsystem Specification**
**Reference: DOD-STD-1703**

1.0 General
- 1.1 Purpose of System/Subsystem Specification
- 1.2 Project References
- 1.3 Terms and Abbreviations

2.0 Summary of Requirements
- 2.1 System/Subsystem Description
- 2.2 System/Subsystem Functions
  - 2.2.1 Functional Allocation Description
  - 2.2.2 Functional Requirements Matrix
  - 2.2.3 Accuracy and Validity
  - 2.2.4 Timing
- 2.3 Flexibility

3.0 Environment
- 3.1 Equipment Environment
- 3.2 Support Software Environment
- 3.3 Interfaces
  - 3.3.1 Interface Block Diagram

**Software Subsystem Specification**
**Reference: DOD-STD-1703 (Continued)**

  - 3.3.2 Software Interfaces
  - 3.3.3 Hardware-to-Software interfaces
- 3.4 Security and Privacy
- 3.5 Storage and Processing Allocation

4.0 Design Details
- 4.1 General Operating Procedures
- 4.2 System Logical Flow
  - 4.2.1 Program Interrupts
  - 4.2.2 Control of Computer Program Components
  - 4.2.3 Special Control Features
- 4.3 System Data
  - 4.3.1 Inputs
  - 4.3.2 Outputs
  - 4.3.3 Displays
    - 4.3.3.1 Description of Displays
    - 4.3.3.2 Display Identification
- 4.4 Program Descriptions
  - 4.4.1 Computer Program Identification
    - 4.4.1.1 Computer Program Component No. 1
      - 4.4.1.1.1 Computer Program Component No. 1 Graphical Representation
      - 4.4.1.1.2 Computer Program Component No. 1 Description
      - 4.4.1.1.3 Computer Program Component No. 1 Interfaces
    - 4.4.1.X Computer Program Component No. X
      - 4.4.1.X.1 Computer Program Component No. X Graphical Representation
      - 4.4.1.X.2 Computer Program Component No. X Description
      - 4.4.1.X.3 Computer Program Component No. X Interfaces
  - 4.4.N Computer Program No. N
- 4.5 database

5.0 Test and Qualification

6.0 Notes

**Interface Control Document**
**Reference: DOD-STD-1703**

1.0 General
- 1.1 Purpose of the Interface Control Document
- 1.2 Project References

2.0 Interfaces
- 2.1 Interface Block Diagram
- 2.2 Software Interfaces
  - a) Interface identification
  - b) Functional description
  - c) Direction of data flow and transfer of control
  - d) Formats and volumes of data to be passed
  - e) Types of interface, such as manual or automatic
  - f) Interface procedures, including telecommunications considerations
  - g) Priority level of the interface data interrupt
  - h) Maximum time allowed for the receiving software element to respond to the interface data interrupt and effects of not responding within the allocated time
  - i) Design requirements imposed upon other computer programs as a result of the design of the interface
- 2.3 Hardware-to-Software Interfaces
  - a) Interface identification
  - b) Functional description
  - c) Direction of signal
  - d) Format of signal
  - e) Transfer protocol used for the signal interface
  - f) Frequency of the signal
  - g) Priority of the signal
  - h) Maximum time allowed for responding to the signals, and the effect of not responding within the allocated time
  - i) Design requirements imposed upon the computer program as a result of the design of the interface

### 4.2.8.3 Software Development File (SDF)

This section is a collection of all the documentation describing the development and testing of an individual software module. It serves as a common point of reference for a particular software module with respect to requirements addressed, design issues, code, test procedures and reports, and problem reports. SDFs are also known as Unit Development Files/Folders (UDFs).

At a minimum, the SDF should contain:

- Requirements addressed
- Design considerations and constraints
- Design documentation and data
- Schedule and status information
- Source code listing
- Test requirements
- Test cases
- Test procedures
- Test results
- Software problem reports

A separate SDF is created and maintained for each software module. An SDF is developed for each module as it is identified during the design phase.

The SDF serves not only as a repository of module information, but is used to record and track the status of all work completed to date on that module.

The SDF is also the primary tool for software QA staff to assess requirements traceability, specification compliance, design verification, and coding standard compliance.

The following is an example of the contents of an SDF:

Cover Sheet—Overview of the contents, schedule and module status:

- Module name
- List of contents (sections)
- For each section:
  - Schedule, completion, and review dates
  - Names of developer and reviewer

Section 1—Requirements:

- Software requirements addressed by this module
- Conflicting requirements and their impact
- Deviation or waivers from the requirements in the SRS

Section 2—Design:

- Design description
- Data flows, flowcharts, state machines, statecharts, etc.
- Module's Program Design language (PDL)

Section 3—Functional Capabilities:

- List of testable functions performed by the module

Section 4—Code Listing:

- Printed listing of error-free compilation or assembly of module

Section 5—Test Plan and Procedures:

- Test plan describing the unit tests to be performed on the module
- Test procedures to be followed and a description of test tools and drivers, test data, expected results, and acceptance criteria to be applied

Section 6—Test Results:

- Test results for each test that was performed, including test date, start and stop times, tester's name, pass/fail results, anomalies, problem reports, and discrepancies

Section 7—Notes:

- Any additional notes and comments regarding the module

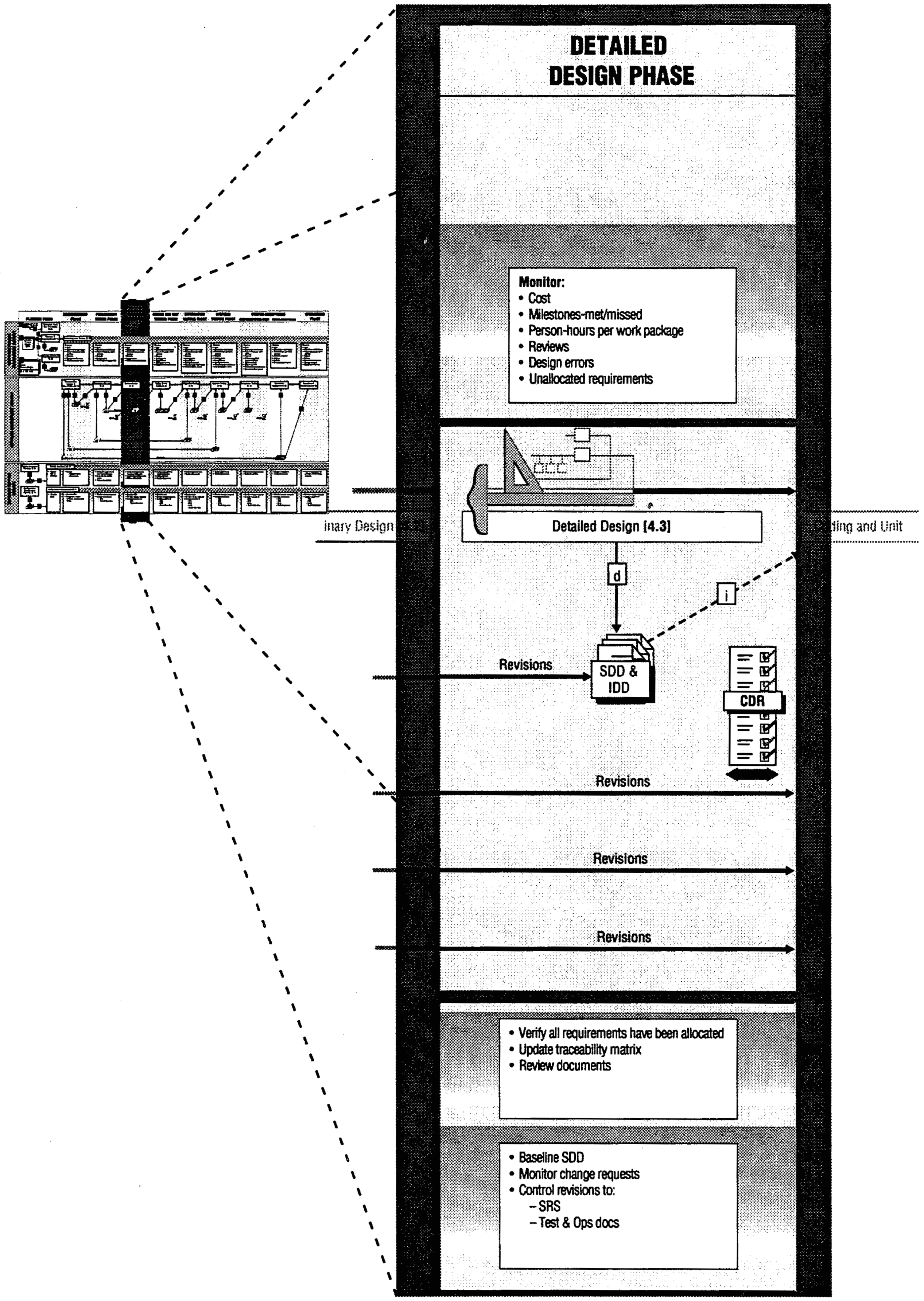
Section 8—Review Comments:

- Comments made by SQA staff during any review of the module

*Section 4.3*

# Detailed Design Phase

## Contents


DETAILED DESIGN PHASE
Monitor:
• Cost
• Milestones-met/missed
• Person-hours per work package
• Reviews
• Design errors
• Unallocated requirements
inary Design
Detailed Design [4.3]
ding and Unit
d
i
Revisions
SDD & IDD
CDR
Revisions
Revisions
Revisions
• Verify all requirements have been allocated
• Update traceability matrix
• Review documents
• Baseline SDD
• Monitor change requests
• Control revisions to:
– SRS
– Test & Ops docs

## 4.3.1 Introduction

This section establishes engineering guidelines for the detailed design phase, which includes software subsystem and module design, performance analysis, test design, design and test documentation, and formal review. Figure 4.3.1-1 represents the process flow diagram for the detailed design phase of software development.

A detailed design is the fully decomposed design of all software subsystems that were established in the preliminary software design. The detailed design identifies all modules and specifies all module design constraints. Also described are the internal logic requirements and interfaces of modules. The detailed design includes the finalized design of software system interfaces and global databases.

Design during this phase should follow a well-defined, disciplined approach that has been determined prior to this phase and documented in the SDP. To ensure awareness of testability in the design, module and software subsystem test case development are included in the detailed design phase.

## 4.3.2 General Methodology for Developing the Detailed Design

The following steps outline the development of the detailed design.

1. Develop a detailed design for all software subsystems using a well-defined design methodology. The detailed design phase usually continues the methodology begun in the preliminary design phase. The following tables identify tasks to be completed for various elements of the software system during detailed design

| For Each Subsystem of the Software System: |
|---|
| • Select and identify the component software modules. |
| • Allocate functional requirements to the software modules. |
| • Define the software module architecture (describe the relationship among software modules). |
| • Define the global data requirements of the software subsystem. |

| For Each Database of the Software System (if any): |
|---|
| • Select and identify files or tables. |
| • Define architecture (relationship among files or tables). |
| • Define file interface with the database management system, if any. |
| • Define the record, field, and item characteristics of each file or table. |

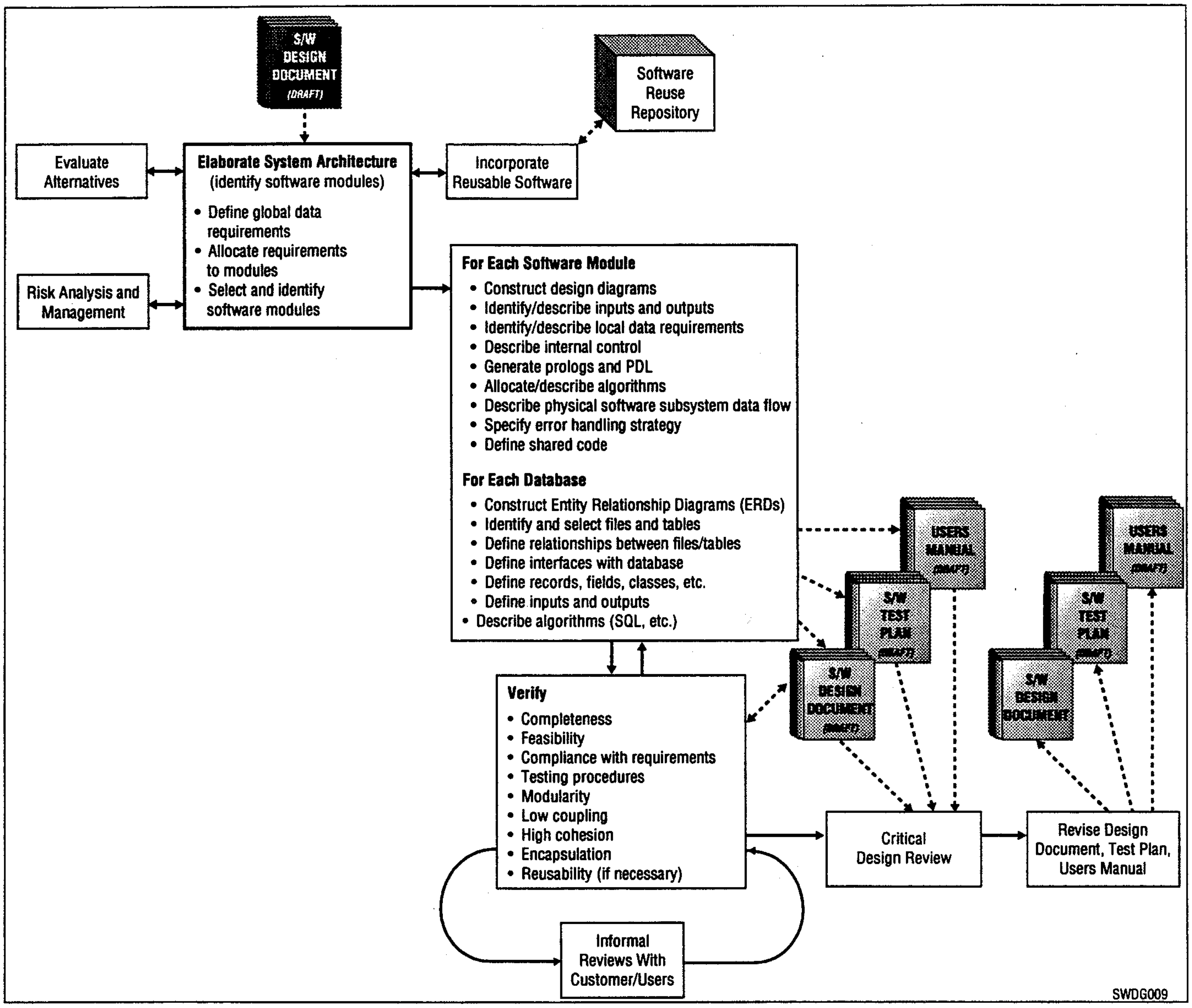

S/W DESIGN DOCUMENT (DRAFT)
Software Reuse Repository
Evaluate Alternatives
Elaborate System Architecture (identify software modules)
• Define global data requirements
• Allocate requirements to modules
• Select and identify software modules
Incorporate Reusable Software
Risk Analysis and Management
For Each Software Module
• Construct design diagrams
• Identify/describe inputs and outputs
• Identify/describe local data requirements
• Describe internal control
• Generate prologs and PDL
• Allocate/describe algorithms
• Describe physical software subsystem data flow
• Specify error handling strategy
• Define shared code
For Each Database
• Construct Entity Relationship Diagrams (ERDs)
• Identify and select files and tables
• Define relationships between files/tables
• Define interfaces with database
• Define records, fields, classes, etc.
• Define inputs and outputs
• Describe algorithms (SQL, etc.)
USERS MANUAL (DRAFT)
S/W TEST PLAN (DRAFT)
S/W DESIGN DOCUMENT (DRAFT)
USERS MANUAL (DRAFT)
S/W TEST PLAN (DRAFT)
S/W DESIGN DOCUMENT
Verify
• Completeness
• Feasibility
• Compliance with requirements
• Testing procedures
• Modularity
• Low coupling
• High cohesion
• Encapsulation
• Reusability (if necessary)
Critical Design Review
Revise Design Document, Test Plan, Users Manual
Informal Reviews With Customer/Users
SWDG009

**Figure 4.3.1-1. Detailed Design Phase Process Flow**

**For Each Software Subsystem-to-Software Subsystem Interface Specified in the IRS:**

- Select and identify the items passed on the interface.
- Identify type, initiation event, and response requirement.
- Identify the field characteristics of each interface item.

**For Each Module Identified as Part of a Software Subsystem:**

- Specify the inputs to and outputs from the software module.
- Identify the local data required by the software module.
- Design the internal control of the software module.
- Specify the algorithms to be used.
- Specify the external references.
- Identify the limitations that constrain design of the software module.

*Note: Using a design description language (e.g., PDL, structure graph, or other formal syntax) makes it easier to describe the detailed design.*

2. Document the detailed design and the supporting engineering analyses and decision rationale in a Software Design Document (SDD) and a detailed IDD. If required, submit the SDD and IDD to the customer for review.

3. Establish and maintain SDFs or new software modules identified in the detailed design.

4. To support software subsystem integration, develop an Integration Test Plan (ITP) for each software system during this phase and document it in the SDFs. The plan should describe the testing of each software build as identified in the SDP. The plan should describe integration procedures, test data sources and simulations, tests for resource utilization, and plans for documenting problems and results. The plan may also allocate requirements to test cases.

5. To ensure awareness of the testability of the design, develop software module and software subsystem test cases and descriptions during the detailed design phase. Enter the results into the corresponding SDFs.

6. For each software system-level test identified in the STP, describe and document test cases in the software test description document for each software system. If required, submit the software test description document to the customer for review.

7. Update the software users manual(s) as required with new information from this phase.

8. Complete the resource utilization; plan that was developed in the preliminary design. If required, submit the resource utilization plan to the customer for review.

9. Continue to develop preliminary versions of the operations and support documents and, if required, submit them to the customer for review. The normally required manuals are the Computer System Operators Manual and the Software Users Manual.

10. Perform a risk analysis on the developed design plan. Some relevant questions are: What are the plans weak spots? Have alternatives been determined for each, if they are needed? How much risk is involved? Refer to Section 5.6 for information on risk management.

11. Conduct an internal review of all products developed during this phase. Refer to Section 4.3.3.1 for more information.

12. At the end of this phase conduct a CDR of the products developed during this phase. [*ISD48*]

### 4.3.2.1 Tailoring a Methodology

A project may handle different categories of software in different ways during detailed design. For complex or critical modules, it may require that detailed design consist of pseudocode and formal reviews. For minor modifications to existing software, it may require a marked-up listing indicating the location and kinds of changes (or the changes themselves). If the contract permits, it may be useful to define categories of software. One such categorization is shown below.

| Category A | Category B | Category C |
|---|---|---|
| Algorithms with unclear, poorly defined approaches | High-risk areas or algorithms | Well-defined algorithms |
| Modules with complex data structures or interfaces | Poorly defined HMI | Well-defined HMI |
| Modification of > 10% of existing code | | Modification of < 10% of existing code |
| Rehosted code with major algorithm change. | | Rehosted code with no major algorithm changes |
| | | Reused code |
| | | Straightforward control modules |

Category A would use the most rigorous and formal development methods, while category C would use the most efficient methods (because of the lower risk involved with category C software). Category B would be in between. Items that could vary by category include degree of formality in walkthroughs, the need for formal walkthroughs vs. informal discussion or review, the timing and detail of documentation, the thoroughness of module testing, and the type of design material produced (charts, PDL, pseudocode, prototype code).

Methodologies can include more than the methodology used to produce the detailed software design. When descriptions of methodologies are required, as in a proposal or SDP, it may be appropriate to describe techniques such as project notebooks, walkthroughs, design standards, and document review and control procedures as well as the methodology for detailed design.

[*ISD48*]

## 4.3.3 Reviews

### 4.3.3.1 Internal Reviews

This section establishes guidelines for the internal review of software products developed during the detailed design phase. Internal reviews provide early identification of potential problem areas and to ensure that requirements and standards are met.

### 4.3.3.2 Critical Design Review

This section establishes guidelines for planning the presentation of software-related CDR material. "Presentation material" is the set of viewgraphs or other visual media used in conducting a formal review. Presentation material consists of summary data, created during the design process or extracted from software development and management plans (reformatted as necessary), for review by an audience that includes both technical and management representatives. The objectives of the CDR are to establish detailed design compatibility (i.e., to verify interfaces) and to review acceptability of the design, performance, test characteristics, and associated documents.

Sample checklists used to assess the results of the detailed design are provided in Section 4.3.7.1. These checklists address the material to be covered at the CDR. The review procedures in this section are meant to be tailored to individual project needs and requirements.

[*ISD48*]

### 4.3.4 Summary

| | |
|---|---|
| Inputs | • SDD<br>• IDD<br>• SDFs<br>• STP<br>• Resource Utilization Plan<br>• Requirements traceability<br>• Design Walkthrough Reports<br>• Operations and Support Documents |
| Software Project Management Activities | • Revision of SDP<br>• Risk Management<br>• Estimation, Monitoring and Tracking |
| Software Development Activities | • Development of Detailed Design<br>• Development of Interface Design Specification |
| Software Support Activities | • CM<br>• QA |
| Products | • Updated Software Design Document<br>• Software Development Files<br>• Resource Utilization Plan<br>• Requirements Traceability<br>• Integration Plan<br>• Design Walkthrough Reports<br>• Software Test Description<br>• Software Module and Software subsystem Test Cases and Descriptions<br>• CDR Material<br>• Interface Design Document<br>• Updated Operations and Support Documents |
| Review | • Critical Design Review |

### 4.3.5 Tailoring to a Small Project

Each project is unique. Tailoring the information provided in this section is essential in defining and implementing the detailed design function to a specific project. Regardless of project size, the detailed design function needs to be performed. Only the level of detail and formality of the process and products vary among projects. Some of the factors to be considered are:

- Time
- Resources
- Complexity

- Contractual commitments
- Intended use of the product

In small projects where time and resources are very limited, the detailed design phase can often be combined with the preliminary design phase. Just as described in Section 4.2.6, at the very least the following should be addressed:

- Describe the overall strategy for implementing the software.
- Determine whether it is based on existing software and identify software you plan on reusing. Describe the modification procedures.
- Determine whether it is procedure based or object oriented; identify the procedures and/or classes (software components).
- Describe how the requirements have been allocated to the software components.
- If a user interface is required, describe how the interface responds to the events that were specified.
- Describe how the primary functions satisfy the requirements they are assigned.
- If necessary (depending on the complexity of the software), describe the functions/classes subordinate to those described in the high-level design.
- Briefly describe how the software will be tested and outline your test plan.

The goal is to communicate the design approach to the customer and other developers. Remember to have the customer approve the design approach before proceeding onto the next phase. The formal PDR and CDR can be replaced with an informal presentation of the design approach describing the issues above, culminating in agreement between the customer and developers on all the design issues being addressed.

### 4.3.6 Suggested Reference Material

Freeman, Peter, and Anthony Wasserman, *Tutorial on Software Design Techniques*, IEEE Computer Society Press.

Coad, Peter, *Object Oriented Design*, Yourdon Press.

**Relevant Standards:**

- DOD-STD-2167—Software Development
- MIL-STD-1521—Technical Reviews and Audits
- Data Item Descriptions (DIDs):
  - DI-MCCR-80018—Computer System Operators Manual (CSCOM)
  - DI-MCCR-80027—Interface Design Document (IDD)
  - DI-MCCR-80012—Software Design Document (SDD)
  - DI-MCCR-80015—Software Test Description (STD)
  - DI-MCCR-80019—Software User's Manual (SUM)
- DOD-STD-1703 (NS)—Software Product Standards
- ANSI/IEEE Std 1016–1987—IEEE Recommended Practice for Software Design Descriptions

### 4.3.6.1 Cited References

[*ISD48*] *Software Engineering Handbook, Build 3,* Division 48, Information System Division, Hughes Aircraft Company, March 1992, pp. 6-2–6-4.

[*ISD48*] *Software Engineering Handbook, Build 3,* March 1992, p. 6-4.

[*ISD48*] *Software Engineering Handbook, Build 3,* March 1992, p. 6-8.

## 4.3.7 Appendix

### 4.3.7.1 Checklists

The checklists provided in this section present a list of most of the issues that may need to be reviewed. It may not be necessary to address each of the items in the checklist. The goal of providing these checklists is for you to be aware of all the issues and for you to tailor these checklists to your project by consciously eliminating the items you do not need.

These checklists can be used to assess the completeness and correctness of detailed design and the readiness for a CDR. The checklists are used to assess the design itself, the detailed design documents, and the material for a CDR.

| Is the Software Design Complete? | |
|---|---|
| Y/N | Check |
| | Does the design contain all the information required in the SDD? |
| | Does the design contain all the information required for the database design? |
| | Does the design contain all the information required in the IDD? |
| | Has a thread for each operational transaction been developed? |
| | Have the modules of the design been allocated to an operational thread where applicable? |
| | Have the module and software subsystem test cases been described in terms of inputs, expected results, and evaluation criteria? |

| Is the Software Design Complete? | |
|---|---|
| Y/N | Check |
| | Have all hardware components been selected? |
| | Have all commercial software products been selected? |
| | Have open design issues been identified and risk assessments performed? |
| | Have user manuals been updated? |
| | Has the STP been completed and approved? |
| | Have the SRS and IRS been completed and approved? |

| **Does the Design Meet the Requirements?** | |
|---|---|
| **Y/N** | **Check** |
| | Does the design implement all functional, interface, performance, quality, sizing/timing, and adaptation requirements? |
| | Does the detailed design implement the top-level design? |
| | Does the design accommodate TBDs in the SRS and IRS? |

| **Is the Design Feasible?** | |
|---|---|
| **Y/N** | **Check** |
| | Has an operational test scenario been defined and verified? |
| | Does the design meet performance within system resources? |
| | Are thread response times within requirements? |
| | Can the design be built within schedule and cost? |
| | Have all new algorithms been prototyped? |
| | Is the design based on known, proven principles? |
| | Is the design testable? |
| | Have HMI features been prototyped? |
| | Does the design incorporate applicable human factors engineering principles? |
| | Have high-risk design areas been identified? |
| | Is the design cross-referenced with functional requirements? |

| **Is the Design Good?** | |
|---|---|
| **Y/N** | **Check** |
| | Is the level of design appropriate for this review? |
| | Does the design comply with standards given in the SDP? |
| | Within a margin of reasonableness, can the design be described as being no more and no less than what is required? |
| | Has reusable code been used in the design whenever feasible? |
| | *Is the design decomposition in accordance with criteria given in the SDP?* |
| | Has each software system been decomposed into modular software subsystems and Modules? |

| Software Design Document—Detailed Design Checklist | |
|---|---|
| **Y/N** | **Check** |
| | Does it adhere to the required format? |
| | Is it internally consistent? |
| | Is it understandable? |
| | Does the document match the design? |
| | Is the document ready to be delivered to the customer? |
| | Are all the SDDs consistent as a set? |

| Database Design Checklist | |
|---|---|
| **Y/N** | **Check** |
| | Does it adhere to the required format? |
| | Is it understandable? |
| | Is it internally consistent? |
| | Does it match the design? |
| | Is it ready to be delivered to the customer? |
| | Is the database design consistent with the functional design? |

| IDD Checklist | |
|---|---|
| **Y/N** | **Check** |
| | Does it adhere to the required format? |
| | Is it internally consistent? |
| | Is it understandable? |
| | Does it match the overall design? |
| | Is the document ready to be delivered to the customer? |
| | Have interface limitations and constraints been identified? |
| | Have inconsistencies in interface requirements been identified and resolved? |
| | Are all the SDDs consistent as a set? |

| Software Test Description Checklist | |
|---|---|
| **Is the STD Feasible?** | |
| **Y/N** | **Check** |
| | Have all operational threads been addressed? |
| | Has a cross-reference between tests and software design been established? |
| | Are the tests consistent with the SRS and operational concept document? |
| | Is the test environment feasible (can it be implemented)? |
| | Have all special test resources been identified? |
| | Have regression test requirements been addressed? |

| Is the STD a Quality Document? | |
|---|---|
| **Y/N** | **Check** |
| | Does it adhere to the required format? |
| | Is it internally consistent? |
| | Is it consistent with the parent STP? |
| | Is it understandable? |
| | Is the document consistent with the detailed design? |
| | Is the document ready to be delivered to the customer? |

| Software Module and Software Subsystem Test Case Checklist | |
|---|---|
| **Y/N** | **Check** |
| | Have module and software subsystem test plans been developed and documented? |
| | Are module and software subsystem test cases consistent with the parent STP? |
| | Are test resources available? |
| | Are test conditions reproducible? |

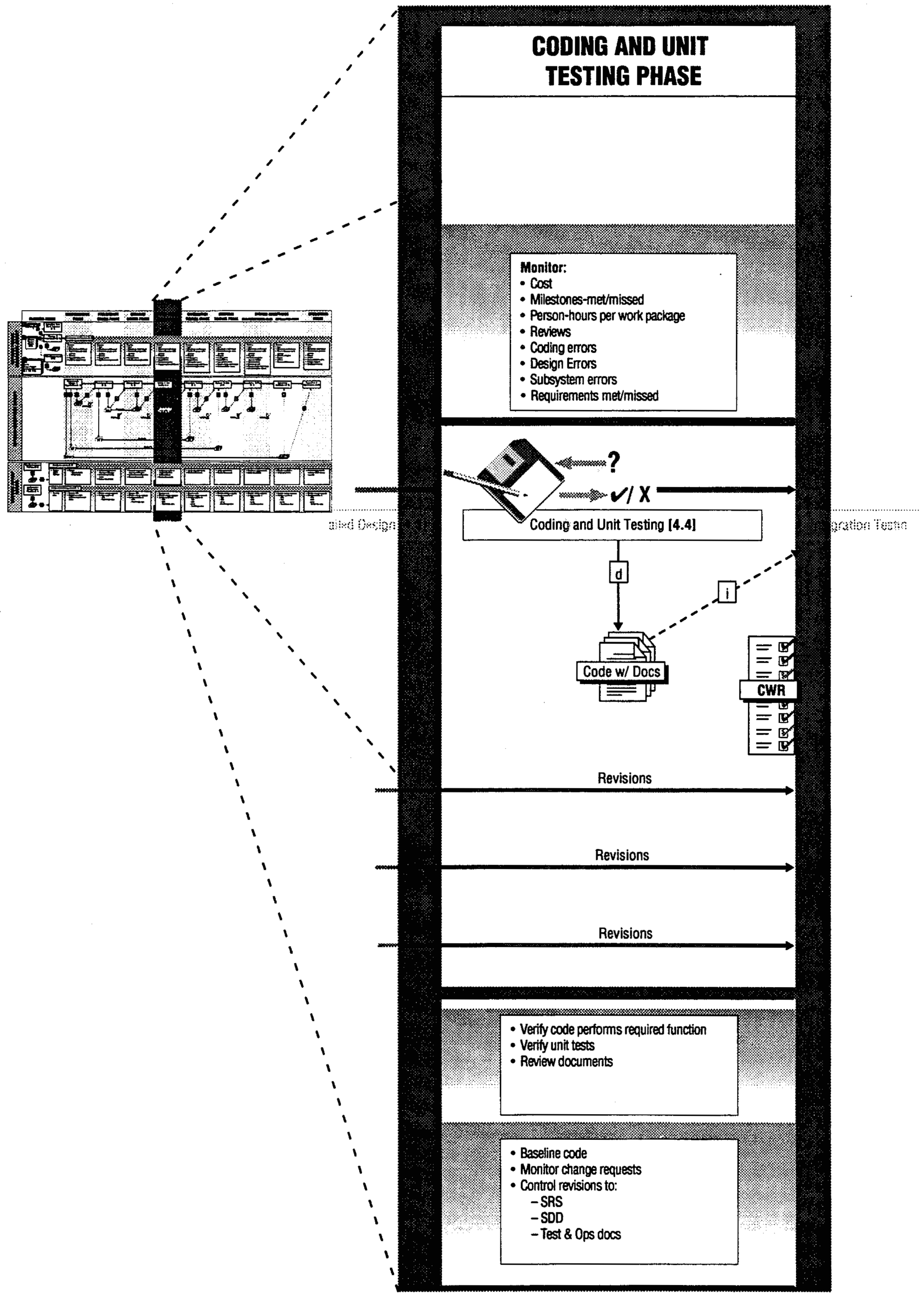
| **Quality of the Test Case Documentation** | |
|---|---|
| **Y/N** | **Check** |
| | Do module and software subsystem test cases comply with the requirements? |
| | Do module and software subsystem test cases define the detailed test constraints required for each module and software subsystem? |
| | Are module and software subsystem test cases consistent with the Software Users Manual? |
| | Are test cases understandable? |
| | Are module and software subsystem test cases consistent with the design? |
| | Have data reduction and analysis techniques been identified? |

| **Quality of the Test Case Documentation** | |
|---|---|
| **Y/N** | **Check** |
| | Is the design ready to be presented at CDR? |
| | Has the form of presentation been established? |
| | Have success criteria been agreed upon with the customer? |
| | Is the CDR plan complete in terms of agenda, facilities, handouts, recording of minutes, action items, and follow-up? |

*Section 4.4*

# Code and Unit Test Phase

## Contents


CODING AND UNIT TESTING PHASE
Monitor:
• Cost
• Milestones-met/missed
• Person-hours per work package
• Reviews
• Coding errors
• Design Errors
• Subsystem errors
• Requirements met/missed
?
✓/ X
Coding and Unit Testing [4.4]
d
i
Code w/ Docs
CWR
Revisions
Revisions
Revisions
• Verify code performs required function
• Verify unit tests
• Review documents
• Baseline code
• Monitor change requests
• Control revisions to:
– SRS
– SDD
– Test & Ops docs

## 4.4.1 Introduction

This section presents the engineering guidelines for the code and unit test phase. This phase includes coding, code reviews, unit testing, preparations for software subsystem and software system testing, and updating the operations and maintenance manuals. The main activities of this phase are the coding, code reviews, and testing of each unit to ensure that it is correct and performs according to the design specifications. Coding consists of implementing the detailed designs in the selected programming language. The products of these activities are retained in each module's Software Development File (SDF).

The order of coding is defined by the project. Two different approaches are top-down and bottom-up. In the top-down development approach, the top-level units are coded first, followed by units of successively lower levels. Bottom-up development starts with coding the lowest-level units first and then proceeds to successively higher levels. A mixture of these approaches is sometimes used, especially for testing new hardware, commercial-off-the-shelf (COTS), and/or external interfaces (this implies that low-level interface routines should be done first). Particularly critical, complex, or poorly defined software units can be coded and tested early regardless of their level in the hierarchy.

Review of the code against the design is done after coding, but before unit testing. Code reviews can be performed by code walkthroughs or code reading. Both techniques review the internal correctness of a unit(s), proper implementation of the detailed design, and the unit's understandability and maintainability and whether it follows the project-specific coding standards. The main purpose for these types of reviews is to detect errors. The earlier errors can be detected, the lower the cost of fixing them.

Unit (or module) testing ensures that the modules perform as specified in the design. Testing covers both the internal workings of the module and the module's external interfaces (i.e., its input and output). Types of testing include logic path testing and data value testing (e.g, correct and incorrect data, boundary values). This type of testing is referred to as "white box testing," i.e., testing the internals of the module.

## 4.4.2 General Methodology for Performing Coding and Unit Testing

Coding and unit testing should follow a well-defined, disciplined approach that has been established for the particular project and agreed upon by project members (managers, developers, and software support staff). This approach should have been documented in the Software Development Plan (SDP).

1. Review the code and unit test approach, procedures, and other specifications given in the SDP. Should changes be advisable or necessary, update the SDP using the CM procedures given in the SDP.

2. Populate each module's SDF with its applicable requirements and design information.

3. Code each new or modified module (i.e., make changes to reusable code) in the specified programming language, in accordance with the coding standards established and documented for the project in the SDP. (Guidelines for coding, which can be used to develop a project's coding standards, are given in Section 4.4.10.1.) The sequence to be followed in coding modules should also be given in the SDP. Compile the code with the specified compiler, and then revise and recompile until all compilation errors are removed. Put a copy of the compiled listing into the module's SDF.

4. Review the code using either the code walkthrough or the code reading technique. (Sample copies of a code reading form and code walkthrough checklists are given in Section 4.4.10.2) Store the review form or checklist in the module's SDF.

5. Make any revisions resulting from the review. Repeat Steps 3 and 4 until all errors are resolved. Store the updated compiled listing and review comments in the SDF.

6. Develop the unit test plan and test procedures for each module and record these in each SDF.

7. Unit test, review, and, if necessary, retest each module until it passes all unit test cases. Record the unit test results in the corresponding SDF.

8. Estimate and/or measure the resource utilization for each of the budgets allocated in the requirements analysis. Verify allocated resource budgets of the code implementation using a documented system load and (if applicable) a calibrated model. Report the actual (predicted or measured) vs. budgeted utilization according to project instructions.

9. To reflect an accurate representation of each module, maintain its SDF throughout this phase.

10. If necessary, update the Integration and Test Plan (ITP) that was created during the detailed design phase. The plan should describe the integration test procedures, test data sources and simulations, tests for resource utilization, and plans for documenting problems and results. The plan may also indicate the allocation of requirements to test cases.

11. Develop subsystem integration and test procedures and document them in the SDFs. (See the checklist for Subsystem/System Test Procedures in Section 4.5.7.1.)

12. To begin planning for system-level testing, develop preliminary system test procedures for each system. Document these in the Software Test Description (STD). If an independent test team/organization (ITO) is to be used on the project, they are the ones to plan and document the system-level tests. In this case, the developer needs to review the plans and procedures being developed by the ITO.

13. Update, as necessary, the Computer Operators Manual, Software Users Manual, and Software Programmers Manual.

14. Update all prior documentation, especially requirements, design, and test plans, to accurately reflect the current software implementation and planned tests, in accordance with the CM procedures and mechanisms described in the SDP.

15. As each module successfully completes unit testing, enter it into the Software Development Library (SDL) to maintain configuration control. This is necessary at this time because the module is now accessible to all members of the project and has the potential of receiving change requests. The module is now available for integration testing.

*[ISD48]*

## 4.4.3 General Guidelines for Developing Code

### 4.4.3.1 Terminology

We will use the following terminology to refer to entities in source code files across different programming languages.

**Function:** The smallest "callable" entity supported in the programming language. Listed below are some programming languages and what "function" will denote in each of them:

- C—functions
- C++—functions, methods
- FORTRAN—main routine, subroutine, function
- Pascal—function, procedures
- Lisp—functions
- Prolog—procedures (collections of rules whose heads all consist of the same predicate with the same arity).

**File:** A disk file containing source code.

**Module:** A collection of one or more conceptually related source code files; for example:

- C++—class header and source files
- Ada—packages
- C—source file(s) and accompanying header file (e.g., stdio, signal, string)

### 4.4.3.2 Guidelines for Comments

#### 4.4.3.2.1 General Guidelines

The purpose of comments is to make the code easily understandable to individuals who need to review or maintain the code.

- Comments should provide "higher level" descriptions of what is going on than would be revealed by inspection of code.
- Comments should be maintained to ensure their correctness; inaccurate comments are more damaging to code than no comments at all, because other developers and maintainers may make erroneous decisions based on those comments.

#### 4.4.3.2.2 In-Line Comments

##### 4.4.3.2.2.1 General

Although different programming languages have different commenting practices, there are some general guidelines that apply across languages. Your in-line comments should:

- Account for a significant percentage of the lines in your source code.
- Appear before every important control construct (loops, if-thens, etc.), where the purpose of the construct is not immediately obvious.

### 4.4.3.2.2.2 Commenting a Single Line of Code

**Above-the-Line Comments**

For above-the-line comments, use the following guidelines to create a strong visual link between the comment and the code:

- Place the comment immediately above the line of code (without separating white space).
- Indent the comment to the same column as the line of code.
- Offset the comment/code pair from any surrounding code with vertical whitespace. For example: Punctuate the comment appropriately to enhance readability. Begin the comment with a capital letter and end it with a colon or ellipsis to direct the reader's eye to the text that follows.

```
...other code...

      /* Here's my comment: */
      statement to be commented;

...other code...
```

Use at least one space between the comment marks and the actual comment text... do not "jam" the text in.

**Right-Margin Comments**

- For short comments (and for languages that support it), you can place the comment to the right of the line of code; leave enough white space to easily distinguish between the two. If you are commenting a block of statements this way, try to line up all the comments:

```
first statement;        // do the first thing
second statement;       // and the second
third statement;        // and the last
```

Punctuation and capitalization are less important with this type of "post-it note" comment.

### 4.4.3.2.2.3 Commenting a Block of Code

To comment a block of statements, you can use the above-the-line approach if the statements have no above-the-line comments of their own:

```
...other code...

      /* If needed, do those things: */
      if (test) {
         do this thing;        /* very important */
         do that thing;
      }

...other code...
```

If the block of code is more complex, use a different style to make the block comment stand out:

```
...other code...

/* AND NOW, A TWO-PART INVENTION... */

/* The first part: */
thing one;
thing two;
thing three;

/* The second part: */
thing a;
thing b;
thing c;

...other code...
```

### 4.4.3.3 Prologues

#### 4.4.3.3.1 What Are They?

Prologues are comments placed before functions (or procedures, or methods) and at the top of files. (Note: Refer to Section 4.4.3.1, Terminology, to select the prologue type that applies to your programming language. For instance, for programming in Fortran, only function prologues and possibly file prologues apply; for C, C++, and Pascal, all prologue types apply.)

- A *function prologue* describes the function that follows it: its purpose, arguments, return values, etc.
- A *file prologue* (at the top of a file) describes the file it is in.
- A *module prologue* is a special file prologue used for one designated file of a module.

#### 4.4.3.3.2 Why Use Them?

Prologues provide "one-stop shopping" for important information. If you need to know what a function's return values are, look at the function prologue. If you need to know the purpose of a file, look at the file prologue.

Uniform prologues make it easier to scan files for information. Project-standard prologues in standard locations are more visually obvious than sporadic comments, making it easier to visually locate functions or other information in a listing.

#### 4.4.3.3.3 General Guidelines

The style of all prologues in a project should be consistent. Uniformity enables developers to easily scan code developed by others for desired information. Variation of styles within a project produces utter chaos.

The prologue should begin with the name of the item being documented (filename, function name, etc.), and it should be offset in some visually obvious way (surrounded by asterisks, newlines, etc.). This allows quick visual scanning through source text for desired functions.

Prologues should be divided into *Sections*; use the following guidelines to make it easier to find desired information within a long prologue:

- Take each section from a standard set of *topics*.
- Present the topics in a *standard order.*
- Omit inapplicable sections entirely or leave the topic name in as a "placeholder."
- Introduce each section by the *topic name,* which should be offset so that it is easy to find (e.g., in all capital letters, on a line by itself, indented several spaces to the left of the text following).

Use "inheritance" of documentation where appropriate: Information in module prologues is understood to pertain to *all* files and to all functions in the module. If necessary, a file or a function can override this information in its prologue, declaring itself to be an exception to the general rule(s).

Avoid excessive detail in your prologues, which can make it more likely that the prologues will become "out of synch" with the in-line comments as the code changes or evolves.

#### 4.4.3.3.4 Function Prologues

Function prologues are always placed immediately before the source code for the function itself.

You may wish to precede them by a page break for more readable printouts (page-breaks are white space characters and are typically ignored by compilers and interpreters).

The following are suggested "standard" section topics, in suggested order of appearance:

- `DESCRIPTION`—What the function does, and possibly who calls it.

- ARGUMENTS OR PARAMETERS—What all/some of the arguments are, in detail. You may choose to omit arguments from this section if they are adequately described by in-line comments or by the file/module prologues.
- RETURNS—Possible return values, and circumstances under which they would be returned (these can be very general; e.g., "returns zero on success, negative on error").
- ALGORITHM—How the function does what it does, if this is important information and not easily deducible from the in-line comments. You would probably use this only for very complex functions.
- NOTES—Miscellaneous notes, for maintainers or serious users of the function.
- WARNINGS—Anything that users of the function should keep in mind.
- BUGS—Any bugs or shortcomings in the function that should be corrected.
- MODIFICATIONS—Modification history of the function. Every entry should have a date, the name or initials of the person who made the modification, and a short description of what was changed.
- OTHER—Any other topics you think are useful (e.g., sample calling sequence).

The following is a sample function prologue:

```
/**************************************************
 * SgMakeLemonade
 **************************************************
 *
 * DESCRIPTION
 *    Make some lemonade.
 *
 * ARGUMENTS
 *    'num_lemons' is the number of lemons to use.
 *
 *    'how_sweet' is either SgSOUR or SgSWEET. If
 *     given as zero, the default is SgSWEET.
 *
 * RETURNS
 *    The number of pints of lemonade made on success,
 *    or negative on failure.
 *
 * WARNINGS
 *    If SgSWEET is specified, the lemonade will be
 *    higher in calories.
 */
```

#### 4.4.3.3.5 File Prologues

File prologues should be placed at the top of each source code file.

The following are suggested "standard" section topics, in suggested order of appearance:

- DESCRIPTION—What the file contains. You may wish this to be a very short description, ending with a "see file X", where file X is the module "header" file.
- MODULE—What module the file belongs to. You may not need this section if the information is self-evident from the filename. For example, a C development project may define that files of the form "X.c", "X.h", "X_p.h" belong by definition to module "X".

- AUTHOR—The author of the file. This is useful information, even if the file author is always the same as the author of the module.

#### 4.4.3.3.6 Module Prologues

Module prologues are a special case of file prologues. For each module, you should designate one file of the module to contain the module header. For example, in a C application, you might decide that for every module X consisting of files X.c and X.h, the file X.h will contain the module header.

The following section topics are suggested:

- DESCRIPTION—What the module does, and perhaps what files or submodules it consists of.
- AUTHOR—The author of the module. This is usually the author of all files in the module.
- NAMESPACE—A description of the names of all entities in the module that are "visible" to the outside world. This information is useful in avoiding naming conflicts in large applications. Note that the best "namespaces" are usually the ones with the shortest descriptions; for example, one module's namespace might be "constants, functions, and variables in mixed case beginning with 'Xt' followed by a capital letter."
- MODIFICATIONS—Modification history of the module. Every entry should have a date, the name or initials of the person who made the modification, and a short description of what was changed. If you use RCS, SCCS, CMS or some other CM system, you may not need this section, because the system will track the history for you.

### 4.4.3.4 Epilogues

Just as all files should begin with a standard file prologue, so they should end with a standard file "epilogue," for example:

```
        . . .
        /*************** end of file "sample.c" ***************/
```

The presence of such an epilogue indicates that the file has not been accidentally truncated. Like the prologue, the epilogue should contain the file name.

### 4.4.3.5 Banners

Often, a source code file may be divided up into several sections, each with a different purpose; for example, constant declarations may be at the top, followed by global variable declarations, followed by "private" helper functions, followed by "public" functions. To make it easier to locate these sections and to assist other developers in adding new entities to the file, you may wish to introduce each section with a "banner" comment.

In a way, banners are, simple prologue comments, except that:

- Because a banner may cover a collection of functions, each of which has its own function prologue, you should choose a banner style that stands out from the prologue style.

- Banners generally contain nothing more than a short description of the section that follows, and for the most part, the same sections will appear over and over in different files. For example, one project might use the following "standard" banners:
- `CONSTANTS`, to introduce the definition of global constants.
- `TYPES`, to introduce the definitions of user-defined types.
- `GLOBALS`, to introduce the definition/declaration of global variables.
- `PRIVATE FUNCTIONS`, to introduce "internal" utility functions.
- `PUBLIC FUNCTIONS`, to introduce functions that are usable by the outside world.

#### 4.4.3.6 Naming Conventions

##### 4.4.3.6.1 Objectives

When developing source code, programmers must frequently decide upon the names of any new entities (functions, types, constants, and global variables) that they are introducing into the global namespace of the system that they are developing.

We introduce the following suggested naming conventions for several reasons:

- To avoid, or at least seriously minimize, the chance of name collisions between HSTX-developed software and system entities or entities in external packages.
- To avoid, or at least seriously minimize, the chance of name collisions between reusable HSTX-developed software originating from separate HSTX development projects.
- To produce entity names that are as brief as possible, so that:
  - Visual bandwidth (the amount of information that the typical reader can absorb at one glance) is not exceeded when reading or scanning for function names.
  - Typing is kept to a minimum.
  - Entities do not take up so much room that single lines of code must be placed on multiple text lines to fit in an 80-character-wide display.
- To produce entity names that are as legible as possible.
- To name entities so that their module of origin is self-evident by examination of the name, which will increase the understandability of the code.

##### 4.4.3.6.2 Name Styles

All entities will have names consisting of one or more "words." How the words are distinguished depends on the conventions and limitations of the individual programming language; here are some examples:

- `ThreeLittleWords` (in "MixedCaseStyle")
- `threeLittleWords` (in "mixedCaseStyle")
- `THREE_LITTLE_WORDS` (in "ALL_CAPS_STYLE")
- `three-little-words` (in "all-lowercase-style")

Every development project should pick one style and stick with it as much as possible for naming all entities, perhaps with slight variations to indicate the nature of the entity being named (e.g., all user-defined typenames might end with a "_t").

For example, the HSTX Software Reuse Repository uses the "MixedCase" style.

### 4.4.3.6.3 The Project Name/Namespace

The first word of an entity should unambiguously identify the *development project* that the entity belongs to, thus preventing name collisions between entities in that project's library and other libraries. Thus, the names of all entities originating from the same development project should begin with the same first word, which we will call the *project name*. Entities whose names begin with a given project name are said to fall within that *project namespace*.

For example, the HSTX Software Reuse Repository uses "`Sr`" as its project name. All entities in the Repository have names of the form "`Sr...`", where "`...`" is in "MixedCase" style. Thus, the function "`SrLogFileOpen`" falls within the "`Sr`" project namespace.

The project name should be:

- Two to four characters, to minimize typing and visual parsing.
- As unique and uncommon a sequence of letters as possible (e.g., `Sr`, `Eg`, `Fst`...)

### 4.4.3.6.4 The Module Name/Namespace

After the first word, a sequence of words (preferably one word) will identify the *module* within the project that the entity belongs to. Thus, the names of all entities originating from the same module should begin with the same sequence of words: the *project name* followed by the module identifier. We will call the complete sequence the *module name*; entity names beginning with a given module name are said to fall within that *module namespace*.

For example, the "message logging" module within the HSTX Software Reuse Repository has "`SrLog`" as its module namespace, which is the project namespace "`Sr`" followed by the identifier "`Log`." All entities in this module have names of the form "`SrLog...`", where "`...`" is in "MixedCase" style.

The module identifier (the portion of the module name after the project name) should be:

- Preferably one word (e.g., "`Logfile`" instead of "`LogFile`").
- Preferably a noun phrase (e.g., "`Log`", "`File`", etc.).
- Unique within the project, to avoid name clashes within a project.
- As few characters as possible, as far as the number of characters, to minimize both the typing and the "visual bandwidth" of entity names within that module. For example, "`Uif`" instead of "`UserInterface`."

### 4.4.3.6.5 Entity Names

Words that follow the project-and-module prefix, are used to distinguish among the different entities in the module. At this point, variations within the naming style may be used so that entity names are "self-commenting." For example, consider the following entities for a hypothetical module "`SrFile`":

```
TYPES:        SrFile, SrFileProt.
FUNCTIONS:    SrFileOpen(), SrFileClose(), SrFileWrite().
CONSTANTS:    SrFileOK, SrFileENOTFOUND, SrFileENOPERM.
GLOBALS:      SrFileGNumOpen, SrFileGErrno.
```

Again, the entity name should be as short as possible to minimize typing and visual bandwidth, but not at the expense of understandability.

### 4.4.3.7 Coding Style

#### 4.4.3.7.1 General Philosophy

Most metrics of good coding style should not be taken too literally. Every language, every task, has the potential for a number of exceptions that violate well-meaning rules of program design.

When evaluating several design possibilities, keep in mind the *spirit* rather than the *letter* of the laws of standard coding practices. The ultimate design objectives are:

- CORRECTNESS—Does the code perform its intended task? Is appropriate error-checking used? If it fails, does it do so gracefully and clean up after itself?
- UNDERSTANDABILITY—Will other developers be able to understand my code when I have left the project? Will I understand my own code a year from now?
- MODIFIABILITY—Can the code be easily adapted to changes in the requirements of the system? Can it be easily extended to cover a wider variety of inputs?
- REUSABILITY—Is the code unnecessarily specific to a particular task, or would minor changes in the code render it useful for other tasks (or other environments) as well?
- ELEGANCE—Does the code perform its task in a simple yet *efficient* manner?

You may have additional priorities for your system, such as portability, adherence to standards, and encapsulation.

(For example, a brief, maintainable, elegant, and easily understandable solution using a "goto" is arguably superior to a long, complicated, and incomprehensible solution that uses a tangle of "ifs" and "whiles.")

Always use common sense when designing your code.

#### 4.4.3.7.2 Code Structure

In the design phase, identify broad classes of tasks for your application (e.g., database access, user interface utilities) and conceptual entities that need to be represented (e.g., satellites, observations, orbital tracks), and then divide your code into modules (and submodules) based on these classes. For each module, sketch out a plausible collection of interface functions by which the "outside world" will "talk" to the module. This basic organizational scheme will greatly simplify code understandability and management.

Modules should be as loosely coupled as possible: changes to the internal implementation of one module should not require changes to any other modules, not even submodules.

Modules should be as strongly cohesive as possible: a module meant to model a specific entity or task should contain *all* and *only* functions/information needed to adequately model the given entity or task.

Beware of excessive nesting of control structures; generally, it makes code harder to read and understand. Deeply nested code is sometimes an indication that a subtask of the current function should be separated out and made a "helper" function in its own right.

In complex statements, use parentheses (or the functional equivalent in your programming language) where allowed. This improves understandability and reduces the risk of errors caused by incorrect assumptions about the precedence of operators.

### 4.4.3.7.3 Code Formatting

Regardless of your editing environment, limit the lines in your source files (and any output files intended to be human-readable) to 80 characters. Longer lines are problematic for developers or users who may not have access to wider terminals or terminal emulators.

Indent code according to the standards or conventions dictated by the language you are using, if any; where more than one acceptable style exists, choose a single style, and use it *throughout* the project. This will greatly enhance the readability of the code.

Where allowed by language conventions, indentation should be as slight as possible (e.g., from 2 to 5 characters per "tabbing" level). Large indentations can make it impossible to fit right-margin comments, and ultimately source code, onto an 80-column page when the code is deeply nested.

Use blank lines to separate "chunks" of code; typically, each such chunk should be preceded by a comment.

Try not to have more than one executable statement per line of source code, unless you are certain that it enhances rather than detracts from readability.

### 4.4.3.7.4 Files

Every file should contain code for either *a single module* or *a portion of a single module*. The only time a file should contain code for more than one module is when it contains code for a main module and for one or more (preferably private) submodules of that module.

Decide on the explicit purpose of a file and stick to it. Do not use files as "dumping grounds" for functions and variables that don't seem to belong anywhere else. (One popular exception is a "utils" module. Just don't let it get out of control.)

Choose a filename to be as close as possible to the name of the module implemented by that file.

### 4.4.3.7.5 Functions

Functions should have only a single entry point.

Functions *preferably* should have only one exit point, with the exception of error exits; as many "error" exits as are needed may be used.

Functions should be as long as they need to be to accomplish the task at hand. Don't let the number of lines drive the implementation of the function—focus instead on the content, and whether or not extracting out a task and placing it in a subfunction would make the code more understandable and maintainable.

Avoid lengthy argument lists; they hamper modifiability of the function. If a function must take a large number of conceptually related arguments, consider instead defining a record structure that can contain most or all of that information and passing that single record structure to the function instead of its component values. This approach allows easy addition or removal of arguments.

Functions must "clean up" after themselves before or on exit, making certain that no unintended side effects have occurred: files opened should be closed, memory allocated should be freed, etc. *Necessary cleanup must be performed for both "success" and "error" exits!*

### 4.4.3.7.6 Constants

Numeric constants should never be used in code, except for the most basic ones (0 and 1). Use symbolic constants to improve understandability of code. For example:

If a number zero is well understood, the number of bits in a byte, you may do this:

```
for (i = 0; i > 8  /* bits per byte */ ;  i++)  {
    }
```

### 4.4.3.7.7 Global Variables

Limit the use of global variables as much as possible without sacrificing clarity of code.

Limit direct access of a module's global variables by users of that module; instead, provide "interface functions" for users to set and access these globals. These interface functions encapsulate the module's internal representation of the global data, allowing the internal representations to change without requiring extensive modification to external code.

Document global variables thoroughly.

Use a naming convention to identify variables as global (e.g., gFooBar)

### 4.4.4 Activities Following Unit Testing

After each module passes unit testing, it is entered into the SDL. Here the module is accessible to all developers and can be used in integration testing. Because it is now widely accessible and may require further changes, management of the module passes from the module's developer to the CM staff. It is now the responsibility of the CM staff to ensure that modules in the SDL are easily accessible to the developers and that all changes to these modules are systematically tracked, authorized, implemented, and tested. CM staff is also responsible for notifying the developer of any changes.

When descriptions of methodologies are required, as in a proposal or SDP, it may be appropriate for this phase to describe such methodologies as project notebooks, code walkthroughs, code reading, coding standards, corrective action system, CM, and unit test standards, as well as the process used to develop the code.

The code and unit test phase is completed when all modules have been written, reviewed, unit tested, and entered into the SDL. In addition, all documentation for each module (e.g., code walkthrough reports, unit test reports) must be up to date. Furthermore, any changes to the requirements and/or software design resulting from this phase must be noted and submitted via the project's corrective action system (e.g., use of engineering change request, software trouble report, software change request). Updates to the preliminary drafts of the operations and maintenance manuals also need to be made during this phase.

## 4.4.5 Organizing the Unit Test Documentation

Unit test documentation is composed of the Unit Test Plan and the Unit Test Summary Report. These can be combined into one document. A proposed outline for each follow:

**Unit Test Plan**

1. Project name
2. Software system name
3. Module name
4. Site(s) where the unit tests will be performed
5. For software runs in multiple modes, specify the modes used in testing
6. Names of persons preparing and approving the test procedures
7. Identification of test tools and drivers
8. List of any called modules which are stubbed and any drivers used
9. For each test case:

    a. List of inputs including data name/location and value, hardware settings, etc.
    b. List of expected outputs including data name/location, value, evaluation criteria
    c. List of all breakpoints, snapshots, etc.

**Unit Test Summary Report**

1. Project name
2. Software system name
3. Module name
4. Date, site, and name of person performing tests
5. List or summary of test results
6. For test failures, corrective actions taken (code modification, reviews); list of retests; name, site, and date of tests and test results
7. Testing approval

## 4.4.6 Reviews

### 4.4.6.1 Internal Reviews

Code reviews are conducted using the code walkthrough or code reading process. Code walkthroughs are performed in a group with the developer of the unit describing or walking through the code with other members of the development team. The objectives are to investigate the internal correctness of the code and validate it against the detailed design. Walkthroughs also serve as a mechanism to ensure that project-specific coding standards have been followed. Walkthroughs are conducted in a thorough manner with presented material reviewed line by line when necessary to improve quality and productivity through early discovery of potential problems. Deficiencies found are noted in the unit's SDF and must be resolved before unit testing begins.

Code reading has the same objectives as the code walkthroughs. However, it is performed individually by another member of the development team. The developer of a unit gives a listing of the code to another member to read and review. Again, deficiencies are noted in the unit's SDF and should be resolved by the developer before unit testing can begin.

#### 4.4.6.2 Formal Review

Usually there is no formal review for the code and unit test phase of development. However, the products of this phase, especially the informal test results, may be reviewed at the Test Readiness Review (TRR) and should be retained in the SDFs.

### 4.4.7 Summary of the Code and Unit Test Phase

| | |
|---|---|
| Inputs | • Software Design Document (SDD)<br>• Interface Design Document (IDD)<br>• Software Module; and Software Sub-system; Test Cases and Descriptions.<br>• SDFs.<br>• Software Test Plan (STP)<br>• ITP<br>• Resource Utilization Plan.<br>• Requirements Traceability<br>• Operations and Support Documents:<br>– Computer Operators Manual<br>– Software User's Manual<br>– Software Programmers Manual. |
| Software Project Management Activities | • SDP<br>• Risk Management<br>• Estimation and Tracking |
| Software Development Activities | • Coding of All Modules<br>• Test of All modules at the Module Level |
| Software Support Activities | • CM<br>• QA |
| Products | • Tested Source Code<br>• Code Walkthrough or Code Reading Reports<br>• Unit Test Results<br>• Documentation Changes (e.g., specification, design)<br>• Detailed Integration Plan<br>• Updated SDFs<br>• SDL |
| Reviews | • Code Reading<br>• Code Walkthroughs |

### 4.4.8 Tailoring to a Small Project

Each project is unique. Tailoring the information provided in this section is essential in defining and implementing the code and unit test function to a specific project. Regardless of project size, the code and unit test function needs to be performed. Only the level of detail and formality of the process and products vary among projects. Some of the factors to be considered are:

- Time
- Resources
- Complexity
- Contractual commitments
- Intended use of the product

If time and resources are very limited, at a minimum, you should adhere to the following guidelines to allow your code to be more maintainable and extensible:

- Decide on a naming convention for the functions, files, and modules in your project.
- Identify a namespace for the modules in your project (Eg, Sr), and stick to it.
- Organize your project into discrete modules based on functionality (e.g., QsStr for string utilities, QsFile for file utilities).
- Make certain that every function has a prologue, explaining, at a minimum, what it does.
- Provide adequate in-line comments in your code. This is the key to being able to easily maintain and extend your code.
- Use the code reading technique for all code. This is an effective and efficient means of detecting errors.
- Perform unit testing on the most critical and complex code. Remember, time spent in unit testing usually reduces the time necessary for integration testing, because errors are easier to locate and retest.

### 4.4.9 Suggested Reference Material

*Software Engineering Handbook, Build 3*, Information System Division, Division 48, Hughes Aircraft Company, March 1992.

*Manager's Handbook for Software Development, Revision 1*, NASA, GSFC, Software Engineering Laboratory Series, SEL-84-101, November 1990.

*Recommended Approach to Software Development, Revision 3*, NASA, GSFC, Software Engineering Laboratory Series, SEL-81-305, June 1992.

**Relevant Standards:**

- ANSI/IEEE Std 1008-1987—IEEE Standard for Software Unit Testing
- ANSI/IEEE Std 829-1983—IEEE Standard for Software Test Documentation
- DOD-STD-2167—Software Development

- Data Item Descriptions (DIDs):
  - DI-MCCR-80027—Interface Design Document (IDD)
  - DI-MCCR-80018—Computer System Operators Manual (CSOM)
  - DI-MCCR-80012—Software Design Document (SDD)
  - DI-MCCR-80021—Software Programmer's Manual (SPM)
  - DI-MCCR-80015—Software Test Description (STD)
  - DI-MCCR-80019—Software User's Manual (SUM)
- DOD-STD-1703 (NS)—Software Product Standards
- NASA-STD-2100-91—NASA Software Documentation Standard

### 4.4.9.1 Cited References

[*ISD48*] *Software Engineering Handbook, Build 3*, Information System Division, Division 48, Hughes Aircraft Company, March 1992, pp. 7-1–7-3.

## 4.4.10 Appendix

### 4.4.10.1 Coding Guidelines for C

#### 4.4.10.1.1 Comments

In-line comment should occupy only one line, *if possible*, with the comment delimiters on that line:

```
/* A good comment: */
some code;
```

Lengthy comments, whether prologues or long in-line comments, should use asterisks as their left border. This allows the source code to be run through reformatters such as "indent" without destroying any formatting within the comment:

```
/*
 * This is a long comment, possibly a portion
 * of a prologue. Separate paragraphs and
 * illustrations, like:
 *
 *     :-)
 *
 * will be nicely preserved because of the
 * asterisks forming the left margin.
 */
some code;
```

Do not nest comments, even if your compiler allows it; comment nesting is not portable. Use preprocessor directives to "comment out" code:

```
#if 0
     /* The code to be commented out */
     ...
#endif
```

### 4.4.10.1.2 Naming Conventions

#### 4.4.10.1.2.1 General

Use the mixed-case style, with a leading capital letter, for all names. For example, "`Eg`" might be a project name, and "`EgLog`" a module under that project.

There are two principal reasons that we depart from the conventional "`lowercase_with_underscores`" that is common in C:

- Mixed case allows for the differentiation of words without having to add extra characters, thus reducing typing and name length.
- Underscores tend to break names up visually and may make them difficult to "parse" as a unit. Mixed case keeps names as single "chunks."

Try to limit all module names to eight characters or less. This will ease typing and will allow the files containing the source code associated with those modules to have names taken *directly* from the module name and yet still be transportable to DOS platforms. For example, module "`SrLog`" might be contained in files "`SrLog.h`" and "`SrLog.c`," both of which are legal filenames under DOS and minimal POSIX implementations.

Although objects with internal linkage (i.e., objects with file scope declared to be "`static`") are generally "private," care must be taken to avoid name clashes with other objects that have external linkage because the results are undefined (ANSI C, 3.1.2.2). Therefore, it is probably not a good idea to depart from these naming conventions, regardless of how "private" the identifiers are.

#### 4.4.10.1.2.2 Constants

Constant names should be composed of the module name followed by all capitals, preferably without underscores. This sets constants apart from other entities, and loosely follows the C coding convention of placing manifest constants in all CAPS.

For example: `EgFileOK, EgFileENOTFOUND, EgFileMAXOPEN.`

Where modules define many constants (return values, options flags, sizes), module authors should use the first character or characters of the uppercase portion of the constant name to underscore its purpose and minimize name clashes.

For example, consider these further partitions of the "`EgFile`" namespace:

```
EgFileO<ALLCAPS>        "Ok" return values
EgFileE<ALLCAPS>        "Error" return values
```

| | |
|---|---|
| `EgFileW<ALLCAPS>` | "Warning" return values |
| `EgFileF<ALLCAPS>` | Options flags |
| `EgFileMAX<ALLCAPS>` | Limits |

### 4.4.10.1.2.3 Globals

Global variables names should consist of the module name, followed by a "G," followed by the remainder of the variable name in mixed case. This clearly identifies an object as a global variable:

```
EgFileGNumOpen
EgFileGBuf
```

An inverted approach is not objectionable, but it loses some of the "up-front" quality:

```
EgFileNumOpenG
EgFileBufG
```

### 4.4.10.1.2.4 Types

Typenames created by "`typedef`" are entirely mixed case.

According to personal taste, typedefs associated with a module may end in a "T," as dictated by some conventional C coding styles: however, context usually serves to distinguish typenames from other entities, and the trailing "T" may be more hindrance than help.

Here are possible types defined by the module "`EgFile`," in both styles:

```
EgFile      EgFileT
EgFileInfo  EgFileInfoT
```

### 4.4.10.1.2.5 Functions

Function names are simply mixed case, beginning with the module name. for example:

```
EgFileDelete()
EgFileGetSize()
```

### 4.4.10.1.2.6 Macros

There is a school of thought that macro names should *always* be obviously different from function names, to avoid a case where side effects change the semantics of the code. For example:

```
#define max(a,b)  ((a) > (b) ? (a) : (b))

/* This would behave differently if max() were a function: */
x = 0;
y = 1;
z = max(++x, ++y);
```

At your discretion you may give macros names composed of the module name followed by all caps, with no underscores; this parallels the "all caps" convention for macro names.

#### 4.4.10.1.3 Coding Style

##### 4.4.10.1.3.1 General

Where possible, write code that complies with the established industry standards of ANSI C and POSIX.

Write code that is as portable as possible within the scope of the intended use of the code. Good software is often used on operating systems other than the one it was originally intended for, so try to use conditional compilation constructs to ensure that your code can be compiled and executed under:

- Both ANSI and Classic C compilers, in addition to the C compiler you are using
- UNIX, if you are not currently developing under UNIX

##### 4.4.10.1.3.2 Conditional Compilation

Perform conditional compilation by *available functionality, not by platform.* You may have to #define your own custom "switches" to do so:

```
/* A good way, using a custom "switch": */
if SrCCF_HAS_STDARG
      extern int SrLog(char *func, char *code, char *fmt,...);
#else
      extern int SrLog();
#endif

/* A bad way: */
#ifdef VAX11C
      extern int SrLog(char *func, char *code, char *fmt,...);
#else
      extern int SrLog();
#endif
```

When #defining conditional compilation switches, define them to a boolean true value (e.g., 1) so that they may be used with either `#if` or `#ifdef`. Since the switches are more flexible, they are easier to use correctly:

```
/* Compilation switch: are function prototypes supported? */
#if defined(__STDC__) || defined(VAXC)
#define SrCCF_HAS_PROTOTYPES 1
#endif
...

/* Do something, based on whether or not we have prototypes: */
#if SrCCF_HAS_PROTOTYPES
...
#endif
```

Try to place all of your #define'd conditional-compilation switches in a single project-wide header file. This will make it easy for developers to consult the file to see if a switch is available before writing their own.

#### 4.4.10.1.3.3 Code Structure

Typically, a module `X` will consist of at least two files, `X.c` and `X.h`. Typically, `X.h` will contain:

- Definitions of preprocessor constants and macros needed by the users of the module.
- Definitions of types needed by the users of the module.
- "Extern" declarations of ALL global variables and functions accessible by users of the module.

X.c will typically contain:

- A #include of `X.h`. Even if unneeded, this is very useful in asserting that there are no mismatches between the declarations in `X.h` and the objects in `X.c`.
- Definitions of preprocessor constants and macros used privately by the module.
- Definitions of types used privately by the module.
- Declarations of global variables.
- Definitions of functions.

Objects (functions and global variables) that are "private" to a "`.c`" file should be declared "`static`."

#### 4.4.10.1.3.4 Code Formatting

Try to indent using some conventional style: BSD, K&R, or the style supported by a language-sensitive editor (Emacs C mode, LSE, etc.).

All binary arithmetic and logical operators (&, +, | |, ==, etc.) should be preceded and followed by a space.

The assignment operator (':=') should be preceded and followed by a space.

Precede "(" by either a "(" or a space, and never precede a ")" by a space.

Never precede a statement-terminating semicolon by a space.

In multiple-statement blocks bracketed by "{" and "}", place the opening { on the same line as the statement introducing the block, and indent the closing } to the same column as that statement. This collapses unnecessary vertical whitespace:

```
/* Good style: */
for (i = 0; i < n; i++) {

   /* More good style: */
      if (a[i] < a[n]) {
      ...
   }
}
```

### 4.4.10.2 Checklists

The checklists provided in this section present a list of most of the issues that may need to be reviewed. It may not always be necessary to address each of the items in the checklist. The goal of providing these checklists is for you to be aware of all the issues, and for you to tailor this checklist to your project by consciously eliminating the items you do not need.

**Code Walkthrough Checklist**

The following code walkthrough checklists are organized by types of errors that may occur.

| Standard Errors Checklist | |
|---|---|
| **Y/N** | **Check** |
| | Are prologue and comments in accordance with software standards and procedures? |
| | Have only standard coding constructs been used? |
| | Does nesting of code comply with established standards? |
| | Has direct code been used only when approved? |
| | Does the size of the module comply with established standards? |
| | Is there only one entry and one exit for each module? |
| | Has top-down format been used with concise statements and orderly development of logic? |
| | Is comment formatting correct? |
| | Is the code in compliance with other coding standards? |
| | Does the code do what the comments say it does? |

| Interface Errors Checklist | |
|---|---|
| **Y/N** | **Check** |
| | Are all modules included and correct? |
| | Are all module call arguments consistent? |
| | Is the database properly used or set? |
| | Are interrupts handled correctly? |

| Data Definition Errors Checklist | |
|---|---|
| **Y/N** | **Check** |
| | Are data properly initialized? |
| | Are data modules or scaling correct? |
| | Are variable types correct? |
| | Are variables defined? |

| Logic Errors Checklist | |
|---|---|
| **Y/N** | **Check** |
| | Are logical operators/operands correct? |
| | Are logic activities in proper sequence? |
| | Are the correct variables checked? |
| | Are logic or condition tests missing? |
| | Are loops separate, without erroneous interaction? |
| | Is common code used when logic must be duplicated? |

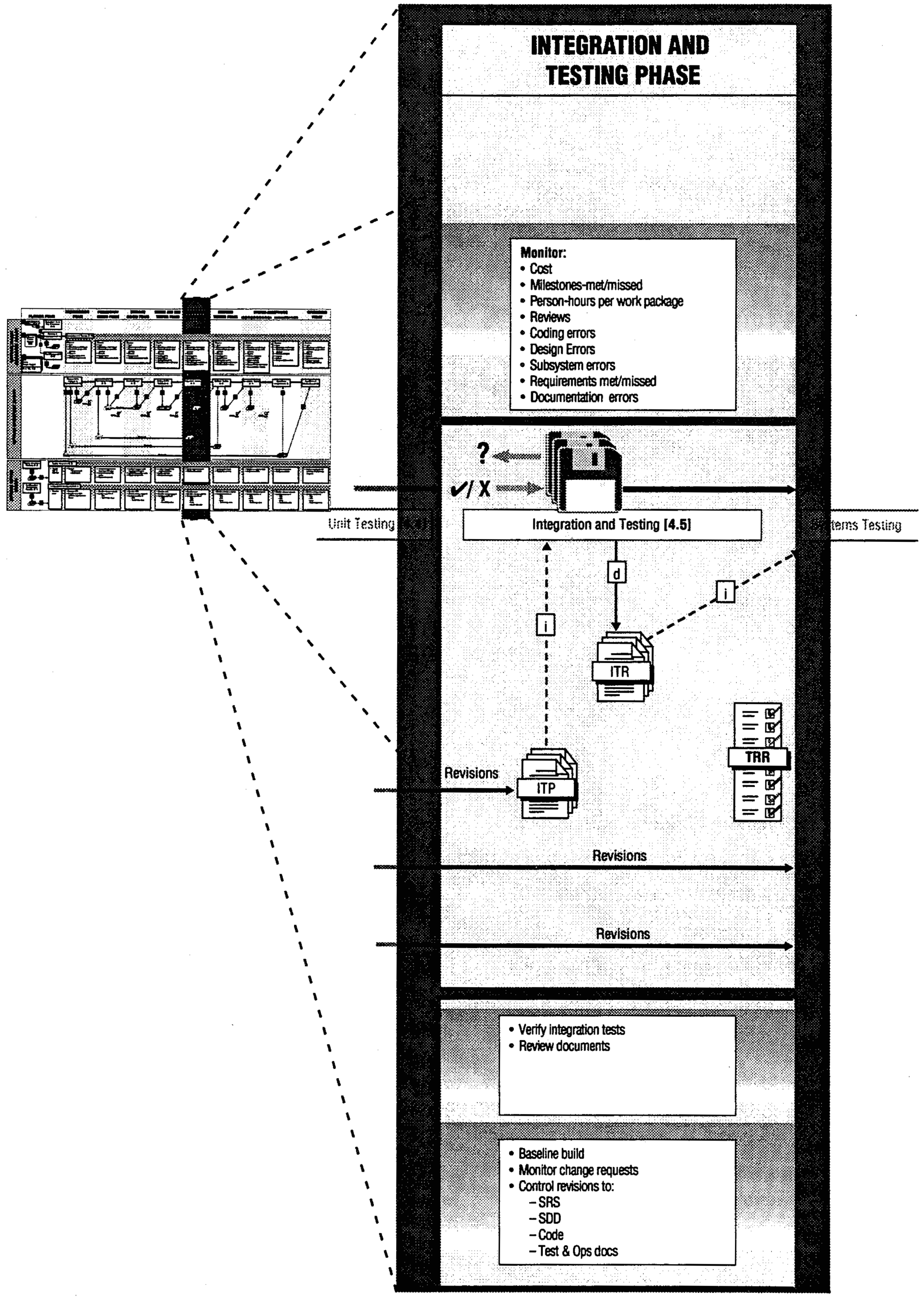
| Data Handling Errors Checklist | |
|---|---|
| **Y/N** | **Check** |
| | Is there proper referencing or storing of data? |
| | Are flags or indexes used properly? |
| | Are bits manipulated correctly? |
| | Are variable types correct? |
| | Is data packing and unpacking done correctly? |
| | Is subscripting used correctly? |

| Computational Errors Checklist | |
|---|---|
| **Y/N** | **Check** |
| | Are operators/operands in equations accurate? |
| | Are sign conventions correct? |
| | Are equations correct? |
| | Is precision maintained in mixed-mode arithmetic? |
| | Are all required computations present? |
| | Is accuracy maintained during rounding or truncation? |

*Section 4.5*

# Integration and Testing Phase

## Contents


INTEGRATION AND TESTING PHASE
Monitor:
• Cost
• Milestones-met/missed
• Person-hours per work package
• Reviews
• Coding errors
• Design Errors
• Subsystem errors
• Requirements met/missed
• Documentation errors
?
✓/ X
Unit Testing
Integration and Testing [4.5]
Systems Testing
d
i
i
ITR
TRR
Revisions
ITP
Revisions
Revisions
• Verify integration tests
• Review documents
• Baseline build
• Monitor change requests
• Control revisions to:
– SRS
– SDD
– Code
– Test & Ops docs

## 4.5.1 Introduction

This section establishes engineering guidelines for software subsystem integration and the subsequent testing generally known as integration testing, including subsystem integration and testing, resource utilization monitoring, SDF maintenance, system test procedure development, and formal review.

There is no formal dividing line between the coding and unit testing phase and this phase. Rather, when the internal review process of the first several modules is completed and their source code is placed into the SDL, personnel assigned to perform the integration can begin to test them according to the integration and test plans. These individuals should be the senior members of the development team; they are in a position to recognize and resolve integration problems more expeditiously than any other group, thus reducing the cost and time needed for integration testing with a potential savings in the later test phases. As more and more software modules are available, they are added to the integration and test configuration. This process continues until the complete software system build is available for system testing.

Integration is the process of aggregating system components into a specific version of the system called a build, and ensuring that these components interact as designed. A system can be composed of one or more subsystems, and a subsystem can be composed of one or more threads. Subsystem integration consists of aggregating modules into threads, verifying each thread according to an informal set of instructions and acceptance criteria, and finally combining the threads to compose a subsystem. All the subsystems are then integrated into the complete system, and each prepared version of the entire system released by development is called a build. The steps in subsystem integration follow the subsystem ITP completed in the detailed design phase.

There can be one or more builds during the development of a complete system. A build normally coincides with a milestone in the software development schedule. A system may be developed and released with only one build, or there may be many builds with additional design and implementation occurring between each build resulting in significant system changes between each build or with just minor problem corrections and enhancements between each build.

## 4.5.2 General Methodology for Subsystem Integration and Testing

One or more methodologies can be used to define the integration and testing approach for the software. When descriptions of methodologies are required, as in a proposal or SDP, it may be appropriate to describe techniques such as when each subsystem will be integrated and how and to include supporting activities and documentation such as project notebooks, document review and control procedures, test standards, the corrective action process, and integration notebooks, as well as test plans and procedures.

For subsystem integration and testing, a well-defined, disciplined approach should be followed. This approach should be documented in the SDP. The general steps are:

1. Review the subsystem integration and test plans and procedures prior to actual integration and testing for test coverage completeness. The plan should describe integration procedures, test data sources and simulations, tests for resource utilization, and plans for documenting problems and results. All interactions between subsystem threads and modules should be identified and included in the test plan. The plan may also allocate requirements to test cases.

2. Establish an integration and test intermediate baseline with the first fully integrated version of the system for each defined build within the developmental CM environment, which will provide for configuration accountability.
3. Aggregate the modules into threads. Test each thread in accordance with the test plans and procedures. A thorough integration approach requires the testing of each possible software path within each thread. However, cost and schedules will sometimes necessitate testing essential and high-risk threads first, and then testing everything else on a time-available basis.
4. Incorporate source code changes necessary to resolve problems found during thread testing. The formal corrective action process may be bypassed during this phase if an informal developmental change control process has been established. On subsequent builds, retest the affected software. Where necessary, make appropriate updates to the documentation (including the software users manual(s) and software programming manual) to reflect the software changes. Record all problems, corrective actions, and test results in the SDFs.
5. Accumulate threads into one or more subsystems; these are then combined to compose the entire system, with each version of the system called a build. The interfaces between threads and later the interfaces between subsystems should all be identified and tested as the system is being integrated. Aggregate the current build with all corrections and enhancements introduced into any previous builds. For each build, develop a version release report identifying the enhancements and other differences from the previous build.
6. Measure resource utilization for each of the budgets allocated in requirements analysis. Verify the integrated implementation of allocated resource budgets using a documented system load and (if applicable) a calibrated model. Report actual (predicted or measured) vs. budgeted use in accordance with command media.
7. Test each build in accordance with the subsystem test plans developed during preliminary design and the subsystem test cases and test procedures developed during detailed design.
8. Incorporate source code changes that were necessary to resolve problems found during subsystem integration testing. On subsequent builds, retest the affected software. Where necessary, make appropriate updates to the documentation (including the software users manual and software programming manual) to reflect the software changes. Record all problems and corrective actions in the SDFs.
9. To record the history of the tests in a standard format, document subsystem test results in a Software Test Report (STR) and enter them into the SDFs.
10. Maintain the SDFs so that all material is up to date.
11. For each build in the development schedule, conduct an Integration Test Readiness Review (ITRR) at the end of the build integration test phase. The final ITRR will be conducted after the last build has completed subsystem and system integration testing and before system testing begins.
12. Continue subsystem testing until all subsystems for the system have been integrated and tested. This includes interface testing between all subsystems that are part of the system.
13. Update (or support the update of) the final system test procedures for the system and document them in the Software Test Description document. If required, submit the Software Test Procedures to the customer for review.

[*ISD48*]

### 4.5.2.1 Important Considerations for Subsystem Integration

The following items should be considered with any methodology:

- Perform subsystem integration and testing in an environment that represents the target hardware and software environment as closely as possible. Pay specific attention to hardware/software switch settings and specific system environmental parameters because these parameters can vary easily from site to site.
- Implement a CM and Control System (CMCS) for the software prior to integration. This system controls the addition of new software or changes to the integration system. When more than one person is using the same set of software, it is essential to know the current version and change status of each element (module and build). The system should also control documentation so that it is updated, when required. It is highly recommended that a CMCS be selected before implementation begins because it could have an impact on the development environment.
- Use a version description to identify the different versions of the modules and subsystems of the software linked together for the release of a build. This description should identify the hardware and commercial software revision levels, where possible.
- Involve the system testers in the later part of the integration test phase to familiarize them with and train them on any new or modified functionality. The system testers can also be using their expertise to informally begin evaluating the integration tests and results.

## 4.5.3 Reviews

### 4.5.3.1 Internal Reviews

Internal reviews are used to provide early identification of potential problem areas and to ensure that requirements and standards are met.

Appropriate internal reviews should be conducted to ensure that the walkthroughs, test documents, and plans are complete and feasible and agree with the software implementation. The best way to review test plans and procedures is to dry run the test procedures, making corrections as necessary. Some contracts may require the procedures to be submitted before there is an opportunity for a dry run. It can be beneficial when the customer recognizes that there will be minor procedural changes, because it gives the customer an early opportunity to comment on the procedures.

The subsystem integration test results, the SDFs, and all other products developed or modified should be reviewed during this phase.

Prior to internal review of the system test procedure, a checklist (Refer to Section 4.5.8, Appendixes) should be established and documented. At a minimum, the checklist should address the technical adequacy criteria. Use this checklist to evaluate the system test procedure.

All ITRRs should be prepared for and conducted. If opted, only the final ITRR is a formal review and the earlier ITRRs can be internal, but this should be worked out with the customer. All ITRRs should discuss the tests conducted, the results, schedules, resources, and any problems still remaining in the system or any workarounds to be used.

### 4.5.3.2 Formal Reviews

Technically, a formal review is not always required to conclude this phase. However, it is recommended that an ITRR be conducted at the end of each software build's integration testing phase, and that at least the last build to be delivered has a formal review that precedes formal system testing (the next phase) and could be considered the culmination of all the former subsystem and system integration test phases. Integration test results are presented by the developers with concurrence from the system testers and the software engineering team to demonstrate to the customer that all software comprising the current build is ready for system testing.

## 4.5.4 Summary

| | |
|---|---|
| **Inputs** | ➧ Master tapes (or other media) containing load subsystems<br>➧ Hardware documentation (e.g., to determine how to set hardware switches)<br>➧ Software documentation (e.g., how to load, start, and reinitialize the system)<br>➧ Procedure on how to use integration/ debug tools<br>➧ Software source listings<br>➧ Declassification procedures for classified software<br>➧ Version description document |
| **Software Project Management Activities** | ➧ Review test plans<br>➧ Monitor test schedule<br>➧ Track critical problems<br>➧ Identify alternatives to high-risk solutions<br>➧ Manage test resources (including personnel) |
| **Software Development Activities** | ➧ Prepare ITPs<br>➧ Senior developers conduct integration testing<br>➧ Development team resolves identified problems |
| **Software Support Activities** | ➧ CM<br>➧ QA<br>➧ Problem report documentation and tracking |
| **Products** | ➧ Set of software that is ready for formal system testing<br>➧ Reports of the integration activity<br>➧ Corrective action system reports<br>➧ Integration notebook<br>➧ STR<br>➧ SDFs<br>➧ Operations and Support Documents<br>➧ STPs<br>➧ Version Description Document (VDD) |
| **Review** | ➧ Tests and results presented by developers<br>• Remaining problems identified by developers<br>➧ Workarounds described by developers<br>➧ System readiness also analyzed by system testers |

## 4.5.5 Tailoring to a Small Project

Each project is unique. Tailoring the information provided in this section is essential in defining and implementing the integration test function to a specific project. Regardless of project size, the integration test function needs to be performed. Only the level of detail and formality of the process and products vary among projects. Some of the factors to be considered are:

- Time
- Resources
- Complexity
- Contractual commitments
- Intended use of the product

A large project may have several development cycles, each with one or more builds and with each build comprising one or more subsystems. A small project may have only one development cycle with only-one build. In both cases, project personnel should follow the steps described in Section 4.5.3 because these steps are scalable for one or more builds.

### 4.5.6 Suggested Reference Material

Myers, Glenford, *The Art of Software Testing*, New York, Wiley-Interscience, 1979.

Mosley, Daniel J., *The Handbook of MIS Application Software Testing: Methods, Techniques, and Tools for Assuring Quality Through Testing*, New Jersey, Yourdon Press, 1993.

The following is a list of applicable standards to be followed during the subsystem integration and test phase. The SDP should indicate which of the following standards will be followed.

- DOD-STD-2167A—Software Development
- MIL-STD-1521C—Technical Reviews and Audits
- Data Item Descriptions (DIDs):
  - DI-MCCR-80015—STD
  - DI-MCCR-80017—STR
- DOD-STD-1703 (NS)—Software Product Standards
- NASA-STD-2100-91—NASA Software Documentation Standard Software Engineering Program
- NASA-DID-A000—Assurance and Test Procedures
- NASA-DID-A200—Test Procedures
- NASA-DID-R009—Test Report

#### 4.5.6.1 Cited References

[*ISD48*] *Software Engineering Handbook, Build 3*, Division 48, Information System Division, Hughes Aircraft Company, March 1992, p. 8-2.

[*ISD48*] *Software Engineering Handbook, Build 3*, Appendix A.

## 4.5.7 Appendix

### 4.5.7.1 Checklists

The checklists provided in this section present a list of most of the issues that may need to be reviewed. It may not always be necessary to address each of the items in the checklist. The goal of providing these checklists is for you to be aware of all the issues and for you to tailor this checklist to your project by consciously eliminating the items you do not need.

These checklists can be used to assess the completeness and correctness of subsystem integration test and the readiness for system test.

| **Source Code Checklist** | |
|---|---|
| **Y/N** | **Check** |
| | Is the source code:<br>Correct? |
| | Consistent? |
| | Accurate? |
| | Understandable? |
| | Complete? |
| | Testable? |
| | Maintainable? |
| | Does the code comply to the project programming standards? |
| | Does the code meet the maintainability requirements? |
| | Does the code fulfill the subsystem requirements? |
| | Is the code consistent with the SDD and IDD? |
| | Have the sizing resources been evaluated? |
| | Have the timing allocations been evaluated? |

| Subsystem Integration Test Results Checklist | |
|---|---|
| **Y/N** | **Check** |
| | Do the test results conform to the expected test results? |
| | Do the test results show complete testing? |
| | Are the test results internally consistent? |
| | Are the test results understandable? |
| | Have the anomalies been evaluated for severity? |
| | Have the anomalies been evaluated for their effect on the subsystem? |
| | Do the test results reflect the completeness of retesting, if any? |
| | Is the subsystem ready to enter system testing? |

| ITRR Checklist | |
|---|---|
| **Y/N** | **Check** |
| | Progress made |
| | Technical adequacy of code |
| | Program plans |
| | System test approach |
| | Source code |
| | Module test results |
| | Accomplishments of subsystem integration |
| | Requirements changes |
| | Design changes |
| | System test plans and descriptions |
| | System test procedures |
| | Subsystem integration test cases, procedures and results |
| | Test resources |
| | Test limitations |
| | Problems |
| | Schedules |
| | Documentation updates |

*[ISD48]*

### 4.5.7.2 Sample Tables of Contents

Below are several examples of the table of contents for an ITP. Selection of the most appropriate test plan template depends on your customer and the desired level of formality

| Test Plan Outline<br>Reference: IEEE | |
|---|---|
| 1.0 | Test-plan Identifier |
| 2.0 | Introduction |
| 3.0 | Test Items |
| 4.0 | Functions To Be Tested |
| 5.0 | Functions Not To Be Tested |
| 6.0 | Approach |
| 7.0 | Item Pass/Fail Criteria |
| 8.0 | Suspension Criteria and Resumption Requirements |
| 9.0 | Test Deliverables |
| 10.0 | Testing Tasks |
| 11.0 | Environmental Needs |
| 12.0 | Responsibilities |
| 13.0 | Staffing and Training Needs |
| 14.0 | Schedule |
| 15.0 | Risks and Contingencies |
| 16.0 | Approvals |

| Test Procedures<br>Reference: NASA-DID-A200 | |
|---|---|
| 1.0 | Introduction |
| 2.0 | Related Documentation |
| 3.0 | Test Identification and Objective |
| 4.0 | Procedures |
| 5.0 | Evaluation Criteria |
| 6.0 | Expected Results |
| 7.0 | Actual Results |
| 8.0 | Abbreviations and Acronyms |
| 9.0 | Glossary |
| 10.0 | Notes |
| 11.0 | Appendixes |

**Test Plan Outline**
**Reference: SEL-81-305, (Recommended Approach to Software Development)**

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

2.0 Test Procedures
   a. Test objectives—purpose, scope 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 CM 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.

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

4.0 Test Descriptions (items a–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.

5.0 Regression Testing Descriptions (repeat items 4a–4f for each regression test)

**Software System Integration and Test Plan**
**Reference: DOD-STD-1703**

1.0 Purpose and Scope
- 1.1 Relationship to Other Test Activities

2.0 Applicable Documents
- 2.1 Development Specifications
- 2.2 Standards
- 2.3 Other Publications

3.0 Integration and Test Identification

4.0 Resources Required
- 4.1 Personnel Requirements
- 4.2 Facilities/Hardware
- 4.3 Interfacing/Support Software

5.0 Test Management
- 5.1 Integration Test Team Organization and Responsibilities
- 5.2 Responsibilities of Other Organizations
- 5.3 Product Control
- 5.4 Test Control
- 5.5 Evaluation and Retest Criteria
- 5.6 Test Reporting
- 5.7 Test Review
- 5.8 Test Data Environment

6.0 Test Structure and Design
- 6.1 Test Levels
- 6.2 Test Approach
- 6.3 Test Inputs
- 6.4 Test Cases/Classes of Tests
- 6.5 Test Identification

7.0 Software Requirements To Be Satisfied Through Integration Testing
- 7.1 Software Requirements
- 7.2 Requirements Verification Traceability

8.0 Schedules

**Test Plan**
**Reference: DOD-STD-7935A**

1.0 General
- 1.1 Purpose of the Test Plan
- 1.2 Project References
- 1.3 Terms and Abbreviations

2.0 Development Test Activity
- 2.1 Statement of Pretest Activity
- 2.2 Pretest Activity Results

3.0 Test Plan
- 3.1 System Description
- 3.2 Testing Schedule
- 3.3 First Location (Identify) Testing
  - 3.3.1 Milestone Chart
  - 3.3.2 Equipment Requirements
  - 3.3.3 Software Requirements
  - 3.3.4 Personnel Requirements
  - 3.3.5 Orientation Plan
  - 3.3.6 Test Materials
    - 3.3.6.1 Deliverable Materials
    - 3.3.6.2 Site Supplied Materials
  - 3.3.7 Security
- 3.4 Second Location (Identify) Testing

4.0 Test Specification and Evaluation
- 4.1 Test Specification
  - 4.1.1 Performance Requirements
  - 4.1.2 System Functions
  - 4.1.3 Test/Function Relationships
- 4.2 Test Methods and Constraints
  - 4.2.1 Test Conditions
  - 4.2.2 Extent of Test
  - 4.2.3 Data Recording
  - 4.2.4 Test Constraints
- 4.3 Test Progression
- 4.4 Test Evaluation
  - 4.4.1 Test Data Criteria
  - 4.4.2 Test Data Reduction

5.0 Test (Identify) Description
- 5.1 Test Description
- 5.2 Test Control
  - 5.2.1 Means of Control
  - 5.2.2 Test Data
    - 5.2.2.1 Input Data
    - 5.2.2.2 Input Commands
    - 5.2.2.3 Output Data
    - 5.2.2.4 Output Notification
- 5.3 Test Procedures
  - 5.3.1 Test Setup
  - 5.3.2 Test Initialization
  - 5.3.3 Test Steps
  - 5.3.4 Test Termination

**Software System DT&E Plan**
**Reference: DOD-STD-1703**

1.0 General
- 1.1 Introduction
- 1.2 Purpose
- 1.3 Criteria for Conducting Software System DT&E
- 1.4 Project References

2. Test Requirements and Acceptance Criteria

3.0 Test Descriptions
- 3.1 Classes of Tests
- 3.2 Test Case Structure

4.0 Software Requirements/Test Specification

5.0 Resources Required
- 5.1 Personnel Requirements
- 5.2 Facilities/Hardware
- 5.3 Support Software

6.0 Database for Software System DT&E

7.0 Test Management
- 7.1 Protocols

8.0 Customer Support Requirements

9.0 Software System DT&E Test Schedules
- 9.1 Master Test Activity Schedule
- 9.2 Activity Network for DT&E Testing

10.0 Software Modification and Retest Criteria

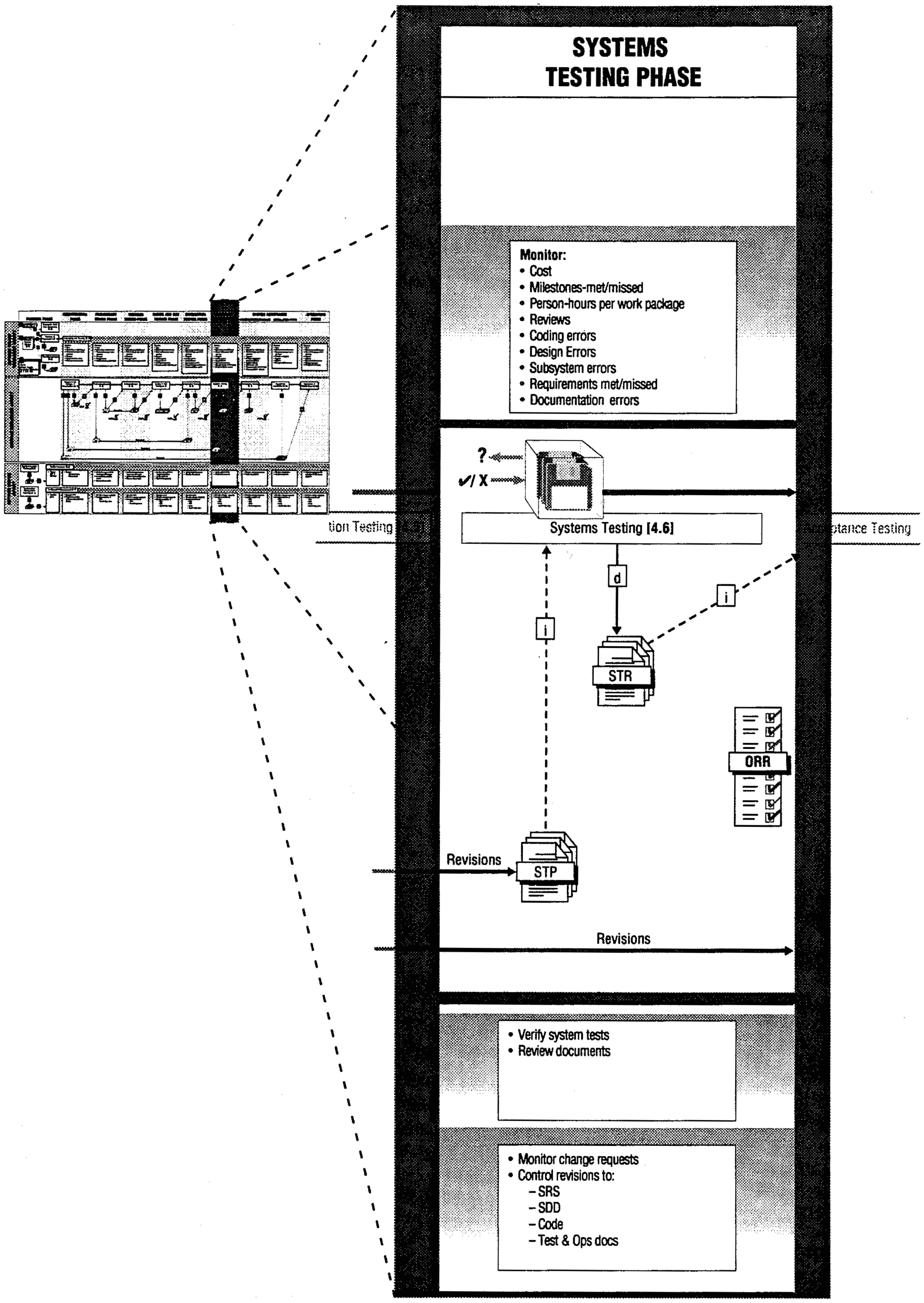
11.0 Notes

;

*Section 4.6*

# Systems Testing Phase

## Contents


SYSTEMS TESTING PHASE
Monitor:
• Cost
• Milestones-met/missed
• Person-hours per work package
• Reviews
• Coding errors
• Design Errors
• Subsystem errors
• Requirements met/missed
• Documentation errors
?
✔/ X
tion Testing
Systems Testing [4.6]
ptance Testing
d
i
i
STR
ORR
Revisions
STP
Revisions
• Verify system tests
• Review documents
• Monitor change requests
• Control revisions to:
– SRS
– SDD
– Code
– Test & Ops docs

## 4.6.1 Introduction

This section establishes engineering guidelines for the system testing software development phase. Within the HSTX methodology, system testing is generally defined as the testing that replicates executing the functions that must be performed, and exercising the capabilities that a system must have in the operational environment and in any interaction with the end users. System testing is most effective when the end users and/or operations personnel are involved in the testing. A system testing environment must replicate the operational and/or user environment as much as possible. System testing activities include SDF maintenance, test plan generation, system test execution, problem reporting, tracking and mitigation, and both informal and formal reviews.

Prior to conducting system testing, the software developers who have completed the integration testing should present an STRR (formal or informal). The STRR is attended by the Independent Test Organization (ITO) that will be conducting the system testing. It should be determined to the satisfaction of the Program Manager, QA, and the system test team that the system has been fully integrated and is ready for system testing with all supporting files, databases, and documentation in place.

The ITO conducting the system testing should consist of a team of end users such as operations personnel and/or scientists. The end user and operations personnel are usually best able to determine whether the system will meet operational needs. Any detected problems, inconsistencies, shortcomings, or misconceptions can be identified at this early point and resolved by software developers in an expeditious manner. Operational problems that are detected early in the test cycle can be resolved before putting the system through extensive testing. This results in lower costs for the overall development process. This is especially true if the system will then undergo a more formal testing cycle such as acceptance testing, which is highly recommended. System testing demonstrates that the system satisfies the following requirements:

- The software supports the full range of operational capabilities required by the Software Requirements Specification (SRS) and the Interface Requirements Specification (IRS); and this should be demonstrable in an operational environment (or as close to an operational environment as possible). If the full range of capabilities is to be provided over several builds, then only the capabilities specified for each build should be tested.
- The software satisfies performance requirements and operational and development constraints; this should be demonstrable in an operational environment.
- The software supports external interface requirements as verified in external testing.
- The software supports Human-Machine Interface (HMI) and system control interfaces.

At the conclusion of system testing, an Acceptance Test Readiness Review (ATRR) will be conducted by the system testing team for the customer and the acceptance testing team. Successful conclusion of the review indicates customer concurrence that the system is ready for acceptance testing.

This is the final phase of software development before acceptance testing. During this phase, the principal activity for the software developers is to support ITO personnel as they formally test the system for compliance with requirements and its operability in an operational environment. ITO personnel have exclusive control over the system during system testing.

### 4.6.2 General Methodology for Performing a Systems Test

A well-defined, disciplined approach should be followed for system testing that should have been documented in the SDP. The emphasis should be on testing the functionality of the system as it will be used operationally. Most of the preparation is done well before the system test phase begins by the ITO rather than by the software development team. However, software developers should understand the process, and may be requested to assist. It is very valuable to have software developers review test procedures early in the process because most requirements are subject to some degree of interpretation at lower levels. For system testing, the general steps are:

1. Review the system test plan and procedures prior to actual testing for test coverage completeness. The plan should describe integration procedures, test data sources and simulations, tests for resource utilization, and plans for documenting problems and results. System tests plans are written against requirements and how the system should run or is anticipated to run in an operational environment. The plan may also allocate requirements to test cases, but this is generally done in acceptance testing.
2. Establish and freeze a baseline configuration. This usually involves obtaining a source copy of the software and building a system from it (compiling, linking, loading, etc.), as well as establishing a hardware configuration. The ITO must be assured that the software under test is a stable system. At the successful conclusion of this phase, this baseline will become the Product Baseline.
3. Obtain approval of procedures and schedules. This is usually a cycle of procedure submittal, customer comment, and revision. Approved procedures are required for a formal test.
4. Document and resolve any problems identified during the system test, must be documented and resolved in compliance with an established software change procedure that is part of a defined configuration management plan (i.e., no "on-the-fly changes" are allowed!).
5. Perform the test with test and QA personnel, at a minimum. QA certifies the results of each step and notes any deviations from or corrections to the procedure.
6. Conduct one or more briefings on test results with management, QA, and software developers. Review the results of the test. Discuss problems and potential solutions. Identify problems in understanding or executing the system. In comparing test results, mark up the official procedure to reflect the "as-run" version (in theory, there should be no changes, but there often are). Explain any anomalies (such as unexpected results or displays caused by operator error, unexpected timing, etc.) and determine the result of the test as passed, conditionally passed, or failed. (A test may pass conditionally if data reduction results must be examined, or anomalies need to be explained.) If necessary, a retest may be scheduled. A test could be rerun immediately, if a failure was caused by incorrect switch setting or an incorrectly set parameter. The approaches chosen by developers may affect test results. For example, if an SRS requires "time-of-day" to be displayed on a screen, a test procedure may specify that a test operator record that time at several points during the test. However, the tester may assume that "time" is displayed to the nearest second, while (absent any other requirement) the developer may have chosen to display it to the nearest minute, which may not be adequate for operational purposes. A more common example is that a test procedure may specify that certain data be recorded for post-test analysis, when there is no capability to record those data without modifying the software under test. Document any problems that cannot be resolved with discussions and/ or input from the development team.
7. Issue one or more test reports with the results of the testing. The report usually includes a list of documented problems, the marked up (and certified) test procedure, and other results of the briefing.

*[ISD48]*

## 4.6.3 Reviews

### 4.6.3.1 Internal Reviews

Internal reviews of the system testing of the software products developed are used to provide early identification of potential problem areas and to ensure that requirements and standards are met. Internal reviews take the form of informal discussions about test procedures and results and possible solutions to problems. Early internal reviews should be conducted to ensure that the test documentation and plans are feasible and agree with the software implementation. Review the system testing results and the SDFs for all other software products tested or modified during this phase.

## 4.6.4 Summary

| | |
|---|---|
| Inputs | • System provided by CM<br>• Hardware documentation (e.g., to determine how to set hardware switches)<br>• Software documentation (e.g., how to load, start, and reinitialize the system)<br>• Procedure on how to use integration/ debug tools<br>• Software source listings<br>• Declassification procedures for classified software<br>• Version description document |
| Software Project Management Activities | • Review test plans and schedule<br>• Monitor test progress<br>• Track problems, identify solutions to high-risk problems<br>• Manage test resources, including personnel |
| Software Development Activities | • Provide support to system testers<br>• Resolve identified problems<br>• Review test procedures |
| Software Support Activities | • CM<br>• Quality Assurance<br>• ITO |
| Products | • Controlled, stable set of software<br>• Reports of the integration activity.<br>• Corrective action system reports<br>• Software STR<br>• SDFs<br>• Operations and Support Documents<br>• Software System Test Procedures<br>• Software Product Specification (SPS)<br>• Version Description Document (VDD) |
| Review | • Discuss results of tests<br>• Identify critical problems<br>• Identify workarounds<br>• Recommend promotion of system to acceptance testing |

### 4.6.5 Tailoring to a Small Project

Each project is unique. Tailoring the information provided in this section is essential in defining and implementing the system test function to a specific project. Regardless of project size, the system test function needs to be performed. Only the level of detail and formality of the process and products vary among projects. Some of the factors to be considered are:

- Time
- Resources
- Complexity
- Contractual commitments
- Intended use of the product

The reviews that are conducted for smaller projects may be very informal, involving a gathering of only a few people in someone's office. Discussions with developers may involve visiting one or two people informally. With larger projects, reviews may involve teams of people as with entire development groups. The level of formality should increase with the number of people and the rigor of standards that to be applied on a project.

### 4.6.6 Suggested Reference Material

Myers, Glenford, *The Art of Software Testing,* New York, Wiley-Interscience, 1979.

Mosley, Daniel J., *The Handbook of MIS Application Software Testing: Methods, Techniques, and Tools for Assuring Quality Through Testing,* New Jersey, Yourdon Press, 1993.

The following is a list of applicable standards to be followed during the system test phase. The STP should indicate which of the following standards will be followed.

- DOD-STD-2167A—Software Development
- MIL-STD-1521C—Technical Reviews and Audits
- Data Item Descriptions (DIDs):
  - DI-MCCR-80015—STD
  - DI-MCCR-80017—STR
- DOD-STD-1703 (NS)—Software Product Standards
- NASA-STD-2100-91—NASA Software Documentation Standard Software Engineering Program
- NASA-DID-A000—Assurance and Test Procedures
- NASA-DID-A200—Test Procedures
- NASA-DID-R009—Test Report

#### 4.6.6.1 Cited References

[*ISD48*] *Software Engineering Handbook, Build 3,* Division 48, Information System Division - Hughes Aircraft Company, March 1992, pp. 9-2–9-3.

## 4.6.7 Appendix

### 4.6.7.1 Checklists

The checklists provided in this section present a list of most of the issues that may need to be reviewed. It may not always be necessary to address each of the items in the checklist. The goal of providing these checklists is for you to be aware of all the issues and for you to tailor this checklist to your project by consciously eliminating the items you do not need.

These checklists can be used to assess the completeness and correctness of system test and the readiness for customer acceptance.

| System Test Procedures Checklist | |
|---|---|
| **Y/N** | **Check** |
| | Are the test procedures adequately detailed? |
| | Do the procedures specify:<br>Test inputs?<br>Expected results?<br>Evaluation criteria? |
| | Are the procedures traceable to the STP? |
| | Do the procedures fulfill all the requirements of the STP? |
| | Are the procedures consistent with the SRS and IRS? |
| | Do the test procedures show that the system correctly implements the allocated requirements? At a minimum:<br>Compliant with design requirements?<br>Timing, sizing, and accuracy assessed?<br>Performance at boundaries and interfaces and under stress and error conditions checked?<br>Test coverage and software reliability and maintainability measured? |

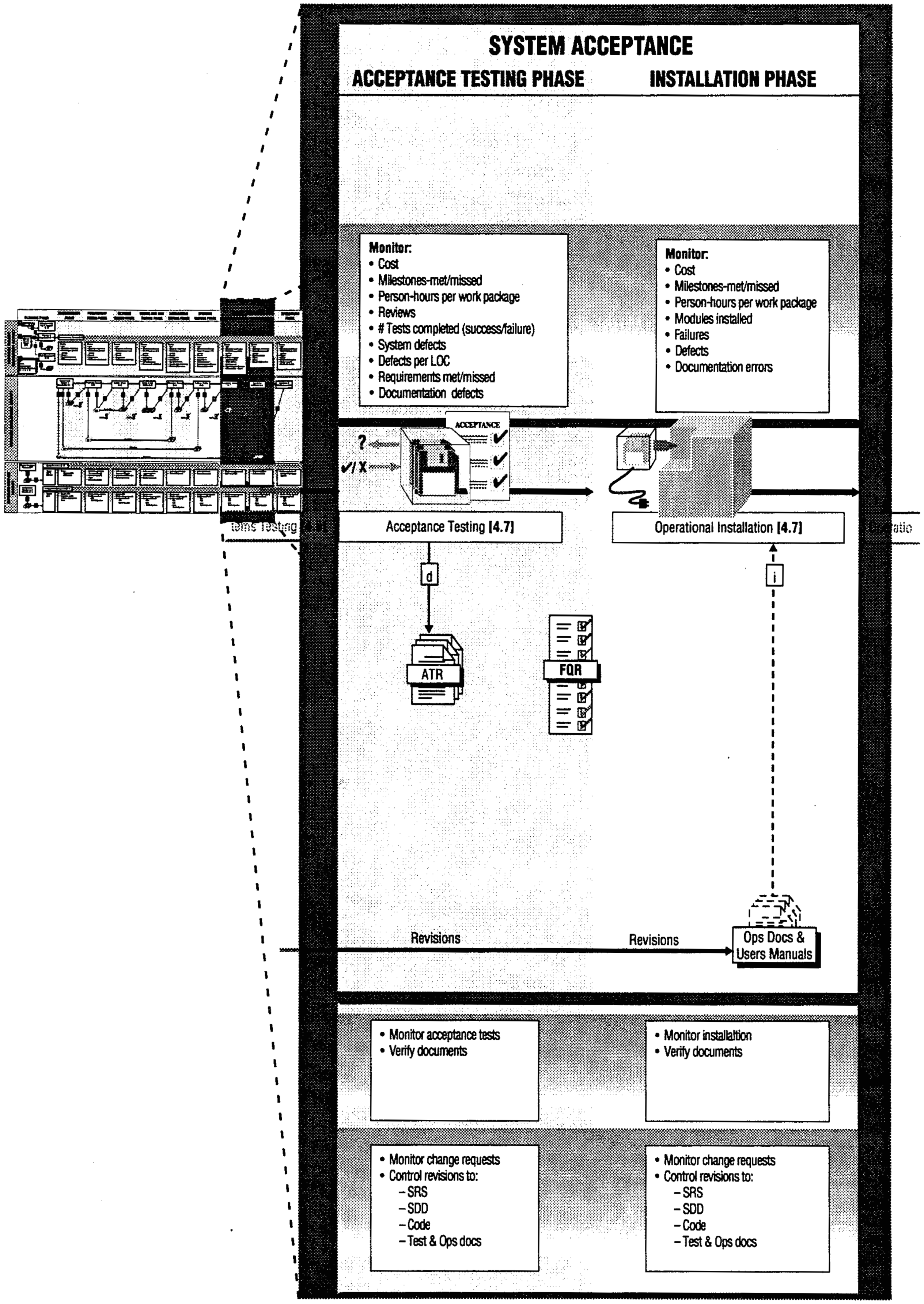
| Operations and Support Documents Checklist | |
|---|---|
| **Y/N** | **Check** |
| | Are these documents consistent with each other? Are they:<br>Understandable?<br>Technically adequate?<br>Presentable?<br>Compliant with project standards? |

| System Test Results Checklist | |
|---|---|
| **Y/N** | **Check** |
| | Do the test results conform to the expected test results? |
| | Do the test results show completeness of testing? |
| | Are the test results internally consistent? |
| | Are the test results understandable? |
| | Was the severity of any anomalies and their effect on the system evaluated? |
| | Do the test results show completeness of retesting, if it was necessary? |
| | Does the system:<br>Support the full range of operational capabilities required by the SRS and IRS?<br>Satisfy performance requirements and operational and development constraints?<br>Support HMI and system control interfaces?<br>Support external interface requirements? |

*Section 4.7*

# Acceptance Testing Phase

## Contents


SYSTEM ACCEPTANCE
ACCEPTANCE TESTING PHASE
INSTALLATION PHASE
Monitor:
• Cost
• Milestones-met/missed
• Person-hours per work package
• Reviews
• # Tests completed (success/failure)
• System defects
• Defects per LOC
• Requirements met/missed
• Documentation defects
Monitor:
• Cost
• Milestones-met/missed
• Person-hours per work package
• Modules installed
• Failures
• Defects
• Documentation errors
ACCEPTANCE
?
✓/X
Acceptance Testing [4.7]
Operational Installation [4.7]
d
i
ATR
FQR
Revisions
Revisions
Ops Docs & Users Manuals
• Monitor acceptance tests
• Verify documents
• Monitor installaltion
• Verify documents
• Monitor change requests
• Control revisions to:
– SRS
– SDD
– Code
– Test & Ops docs
• Monitor change requests
• Control revisions to:
– SRS
– SDD
– Code
– Test & Ops docs

## 4.7.1 Introduction

Acceptance testing is performed by a specialized ITO dedicated to testing. The ITO's mission is to verify that all requirements specified for a system or a unique build of a system have been implemented. This verification is usually accomplished using a suite of very well-defined tests with all of the requirements to be verified mapped into the tests. Acceptance testing is very rigorous and formal. The system must be under CM control during the testing period. Problems encountered are documented, and usually a project Configuration Control Board (CCB) will decide if the problem must be fixed before the system can be accepted. Any changes incorporated into the system will necessitate rerunning at least a designated core set of tests. Part of acceptance testing also involves one or more reviews to verify that the system hardware, software, and interfaces are complete and documented for operational installation. These reviews are the FCA, PCA, and FQR. They can be combined for convenience and efficiency. The readiness of the system to be promoted to an operational status is analyzed during these reviews. Test results and problems are discussed and decisions are made by management on whether or not to promote the system. Some problems may be mitigated and not prevent acceptance, while other problems may require resolution. The customer has the final say on system acceptance based on current system status, the remaining system problems, and the mitigations that have been reached. An installation date is usually set for system promotion.

## 4.7.2 General Methodology for System Acceptance Testing

The FCA is a formal audit that validates that a configured system functions according to the specified requirements. The FCA ensures that the collected test data verify that the configured system has achieved the performance specified and that CM has maintained the configuration identification documents for each configured item. During the FCA, QA reviews and checks for accuracy and completeness the qualification test procedures, results, and data for each configured item of the system.

The PCA is the formal examination of the "as-built" version of the configured system. The system baseline is established for the accepted version of the system against its design documentation. The PCA for each configured item of a system is conducted at the successful conclusion of the configured item's FCA. With the customer's and operational team's approval of the product specifications and at the successful conclusion of the PCA, the product baseline is established. QA also reviews all of the operational and support documents (including the Operator's, User's, and Diagnostics Manuals) during the PCA.

QA reviews and checks the qualification test procedures for completeness and the results for accuracy. Required documentation and the performance of the entire system are also reviewed during the FQR.

## 4.7.3 Tailoring to a Small Project

Each project is unique. Tailoring the information provided in this section is essential in defining and implementing the system's acceptance testing function for a specific project. Regardless of size, the acceptance testing function needs to be performed. Only the level of detail and formality of the process and products vary among projects. Some of the factors to be considered are:

- Time
- Resources

- Complexity
- Contractual commitments
- Intended use of the product

Even on a small project there are desirable steps to follow for system acceptance. The reviews described above can be done on an informal basis with the customer or users.

- List of materials to be reviewed.
- Documentation checked for accuracy and completeness.
- Status of each test case.
- Status of all known software problems documented and maintained in a database.
- Is the implementation of the Software Design consistent with the requirements?
- Are the test results traceable?
- Are the test results summarized?

[*ISD48*]

### 4.7.4 Suggested Reference Material

*Automatic Dependent Surveillance Configuration Management Plan,* Aviation System Division (HSTX), 1991.

*Automatic Dependent Surveillance Quality Control Program Plan,* Aviation System Division (HSTX), 1991.

Paulk, M. C., B. Curtis, M. B. Chrissis, and C. V. Weber, *Capability Maturity Model for Software, Version 1.1,* Software Engineering Institute, Carnegie Mellon University, 1993.

*Software Engineering Handbook,* Division 48, Information Systems Division, Hughes Aircraft Company, 1992.

#### 4.7.4.1 Cited References

[*ISD48*] *Software Engineering Handbook, Build 3,* Division 48, Information System Division, Hughes Aircraft Company, March 1992.

### 4.7.5 Appendix

#### 4.7.5.1 Checklists

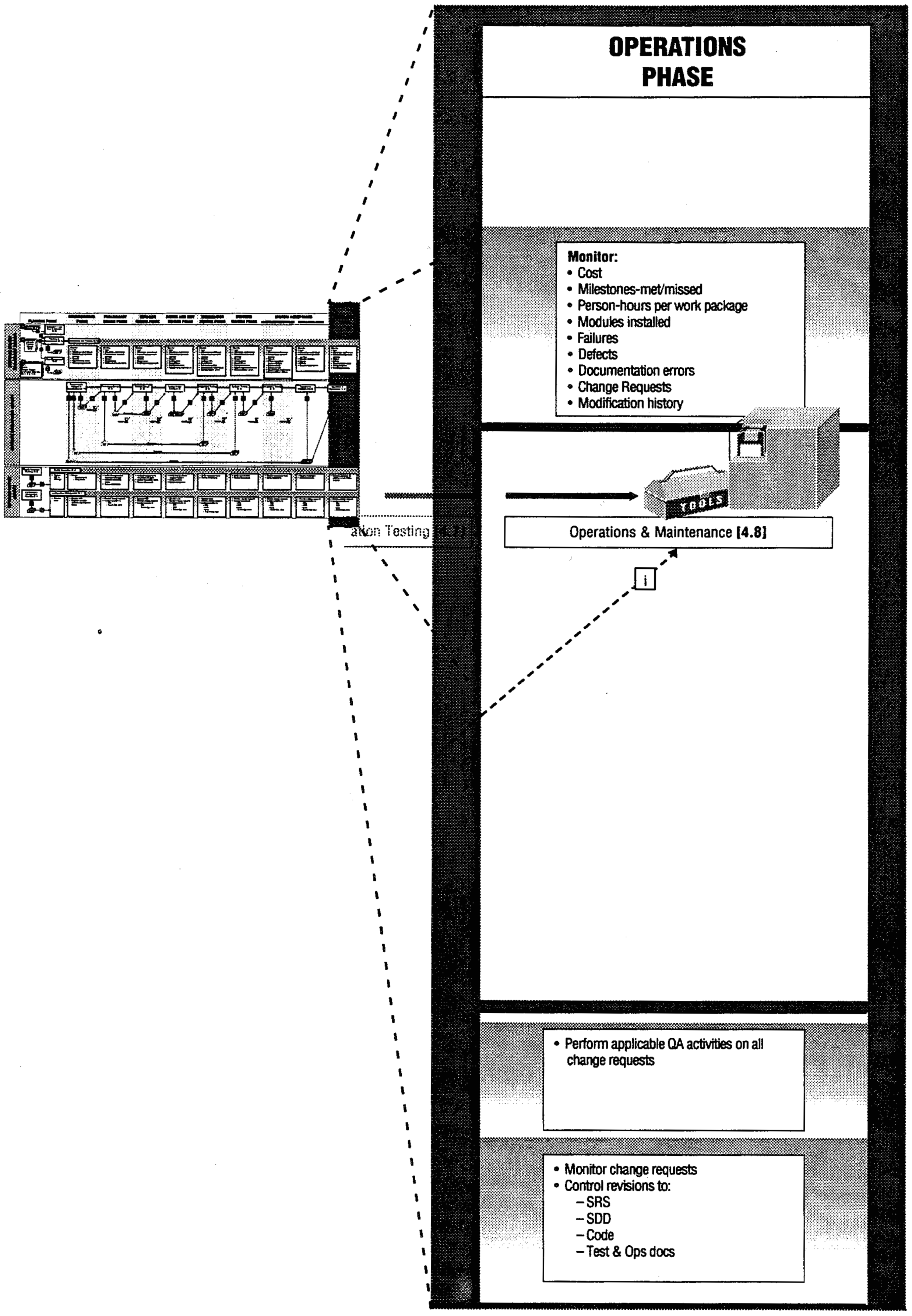
This section presents a checklist for assessing the readiness of materials for a FQR. Most of the issues that may need to be reviewed are listed. It may not always be necessary to address each of the items in the checklist. The goal of providing this checklist is for you to be aware of all the issues to tailor this checklist to your project by consciously eliminating the items you do not need.

| **Formal Qualification Review Checklist** | |
|---|---|
| **Y/N** | **Check** |
| | Were the schedule, agenda, and list of materials to be reviewed established in agreement with the contracting agency? |
| | Was the documentation prepared for the FQR checked for accuracy, completeness, consistency, format, and organization? |
| | Was the status of each test procedure reviewed and summarized? |
| | Are all known software discrepancies documented in accordance with the problem reporting system and ready for review? |
| | Are all test limitations identified and documented? |
| | Does the Software Design Document (SDD) reflect the exact version of the software module? |
| | Are the software module test results traceable to the software module test plan and test procedures? |
| | Are the software module test results summarized relative to the acceptance criteria specified in the Software Test Plan (STP)? |

*Section 4.8*

# Operations and Maintenance Phase

## Contents


OPERATIONS PHASE
Monitor:
• Cost
• Milestones-met/missed
• Person-hours per work package
• Modules installed
• Failures
• Defects
• Documentation errors
• Change Requests
• Modification history
TOOLS
Operations & Maintenance [4.8]
i
• Perform applicable QA activities on all change requests
• Monitor change requests
• Control revisions to:
– SRS
– SDD
– Code
– Test & Ops docs

## 4.8.1 Introduction

Operations and maintenance is that part of the software lifecycle when the software is delivered and operational at the client site. It is important that changes made during this phase of the software lifecycle do not adversely affect the operational software. Thus, software changes should be thoroughly tested in a test environment before they are incorporated into operations. Software changes should also be thoroughly regression tested so that original functionality is not degraded by new software. If the requested software changes include changes to the requirements, it may be desirable to perform all of the software lifecycle activities, beginning with requirements analysis.

During the operations and maintenance phase, changes to the software (software code and/or software documentation) can be proposed by anyone involved with the software. This includes the customer(s), users, computer operations staff, development and maintenance contractor staff, and others. Proposed changes to the software fall into four categories:

**Four Categories of Software Maintenance**

- **Corrective Maintenance**—Changes to fix known deficiencies/errors
- **Adaptive Maintenance**—Changes necessary for the software to operate in a new or modified environment (e.g., different type of computer or peripherals, different or upgraded operating system)
- **Perfective Maintenance**—Changes to enhance the functionality of the software
- **Performance Maintenance**—Changes to improve the performance of the software

A proposed change(s) is documented as either an ECR, if the change necessitates a software requirements change, or a PTR. (See Glossary for different names of this report.) This begins the change process, which continues until the change is not accepted for implementation or a new release of the software, including the updated software code and documentation, is made operational.

An example of the steps in the change process is given in Figure 4.8.1-1. In this case, the customer identifies a problem that does not lead to a requirements change.

The change process may result in repeating many of the activities performed during initial development of the software (i.e., planning, requirements analysis,..., system acceptance). Maintenance staff should refer to the relevant sections for details and tailor these to their maintenance activities.

## 4.8.2 General Methodology for Operations and Maintenance

Software maintenance usually involves repeating some of the same activities performed in the development phases. If SDFs are used during the initial development, they can be very useful during the maintenance phase. Information in the SDFs provides a documentation history of all of the modules in the system. Given below are the unique steps for software maintenance, including the activities of the software support staff (QA and CM).

1. A problem database should be established. Any operational problem or suggested improvement indicated by the customer should be documented in the database. This database would contain both ECRs and PTRs.

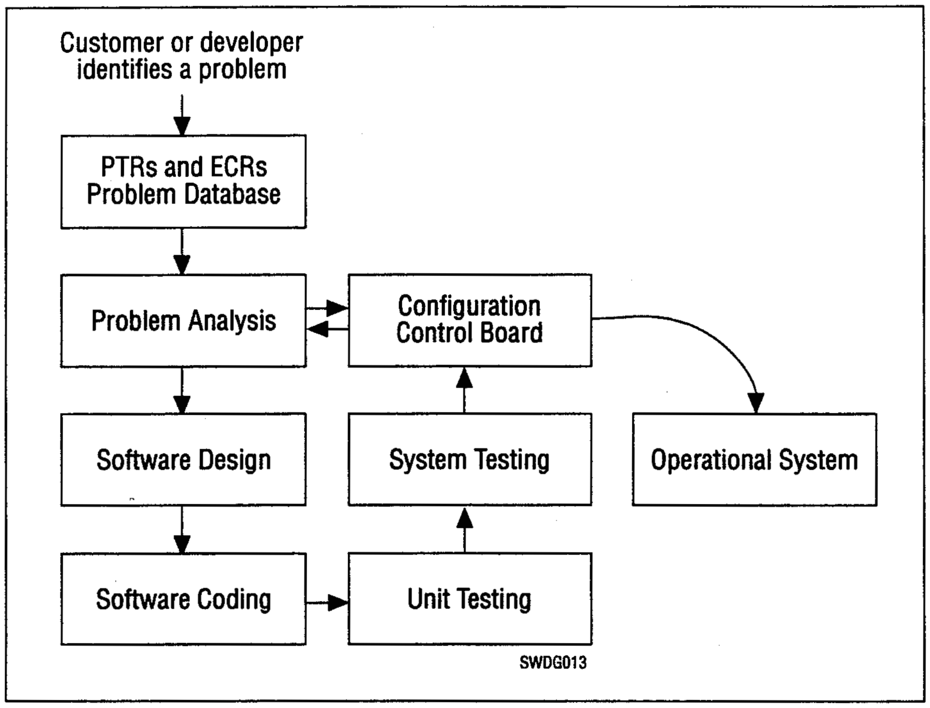

Customer or developer identifies a problem
PTRs and ECRs Problem Database
Problem Analysis
Configuration Control Board
Software Design
System Testing
Operational System
Software Coding
Unit Testing
SWDG013

**Figure 4.8.1-1. The change process may result in repeating many of the activities performed during initial development of the software; i.e., planning, requirement analysis, and system acceptance). Maintenance staff should refer to the prior sections for details and tailor these to their maintenance activities.**

**Important information that needs to be in the problem database includes:**

- A unique identifier of the problem
- A one-line abstract of the problem so that the problems can be listed on a page
- Priority of the problem
- Status of the problem
- Person that documented the original problem
- Suspected software module that needs to be corrected
- Date the problem was originally documented
- Indication whether the problem is an ECR or PTR
- Detailed problem description
- Level of effort required to resolve the problem
- Resolution of the problem (summary and description)
- Author of the fix
- Date of QA witness of fix test
- Pass or fail of fix by QA

2. A CCB comprising representatives from project management, CM, QA, technical leads, and the customer should be established. The CCB determines whether problems are classified as ECRs or PTRs. The CCB also determines the priority and scheduling of the problem resolution.

3. If an ECR involves a change in requirements, the lifecycle should begin with requirements analysis and requirements review and proceed through all the other lifecycle phases, ending at acceptance testing.
4. It is CM's responsibility to ensure that problems are documented in the problem database.
5. It is QA's responsibility to make sure that problems are tracked to closure.
6. Software developers are responsible for the problem analysis. If the analysis indicates that a major design change is required to fix the problem, it may be desirable to bring the issue back to the CCB to determine whether the problem is worth fixing.
7. Software developers are responsible for designing the software fixes for the problems. If the fix for a problem involves a large design change, it may be desirable to have a design review with the customer. The software developers may also be responsible for documentation changes to design documents and user's manuals.
8. The software developers are responsible for coding the software fixes to the problems and for unit testing the fixes.
9. A test team is responsible for system testing with the fixes. This test team is also responsible for regression testing to verify that the fixes do not degrade the functionality of the system. Depending on the customer's requirements, this test team may assist in formal acceptance testing.
10. CM verifies that software baselines are maintained and that fixes are added to the baseline in a controlled manner. CM keeps track of what fixes are contained in what versions. If a fix causes a serious problem, it may be necessary to revert to a previous version of software. CM is also responsible for version control of the software documentation.
11. QA verifies that procedures are followed and that the necessary documentation is completed along with the software changes.
12. The CCB determines which software should be released with which software build.

### 4.8.3 Tailoring to a Small Project

On a small project most of the functions described above should still be done. At a minimum, the following steps are recommended:

1. It is still important to document the problems in some sort of database.
2. The developer is responsible for tracking the problems to closure.
3. The priority of the problems must be agreed on between the developer and the customer. In this case, the CCB might consist only of a developer and the customer.
4. If software changes result in requirements changes, the requirements should be documented by the developer and reviewed with the customer to ensure that the requirements are understood by both the developer and the customer.
5. The developer is responsible for problem analysis.
6. The developer is responsible for designing the software changes. If there are significant design changes, it may also be desirable to have an informal design review with the customer before any software changes are made.
7. The developer is responsible for coding the software changes and unit testing the changes.
8. On a small project the developer would be responsible for system testing in addition to the unit testing described above. The developer would also be responsible for regression testing.
9. The developer would also be responsible for maintaining baselines and controlling software versions.

### 4.8.4 Suggested Reference Material

Paulk, M. C., B. Curtis, M. B. Chrissis, and C. V. Weber, *Capability Maturity Model for Software, Version 1.1*, Software Engineering Institute, Carnegie Mellon University, 1993.

*Software Engineering Handbook*, Information Systems Division, Division 48, Hughes Aircraft Company, 1992.

### 4.8.5 Appendix

#### 4.8.5.1 Checklists

The checklists provided in this section present a list of most of the issues that may need to be reviewed. It may not always be necessary to address each of the items in the checklist. The goal of providing these checklists is for you to be aware of all the issues and for you to tailor this checklist to your project by consciously eliminating the items you do not need.

| Checklist To Verify Operations and Maintenance Setup | |
|---|---|
| Y/N | Check |
| | Are problems being recorded in the problem database? |
| | Will the CCB determine the priority of the problem and determine whether it is a PTR or an ECR? |
| | If an ECR involves a change in requirements, will the lifecycle begin with requirements analysis and requirements review and proceed through all of the other lifecycle phases ending at acceptance testing? |
| | Are software developers responsible for the problem analysis? |
| | Are software developers responsible for designing the software fixes to the problems? |
| | If the fix for a problem involves a large design change, is there a design review with the customer? |
| | Are the software developers responsible for coding and unit testing the software fixes to the problems? |
| | Are test teams responsible for system testing? |

### 4.8.5.2 Sample Tables of Contents

**Sample Table of Contents of a Operational Procedures Manual**
Operational Procedures Manual
Reference: NASA-DID-P700

| | |
|---|---|
| 1.0 | Introduction |
| 2.0 | Related Documentation |
| 3.0 | System Preparation and Setup Procedures |
| 4.0 | Standard Operating Procedures |
| 5.0 | Fault and Recovery Procedures |
| 6.0 | Emergency Procedures |
| 7.0 | Diagnostic Procedures |
| 8.0 | Abbreviations and Acronyms |
| 9.0 | Glossary |
| 10.0 | Notes |
| 11.0 | Appendices |

**Sample Engineering Change Request Outline**
Engineering Change Proposal
Reference: NASA-DID-R005

a. Proposal Identification

b. Originator Identification Including
  1. Name and organization
  2. Address and phone

c. Product (including documents) identification including
  1. Name or title
  2. Version number
  3. If applicable, environment information (e.g., hardware and operating system for a software product)

d. Proposal information including
  1. Title
  2. Date
  3. Classification (e.g., major or minor)
  4. Priority
  5. Description of proposed change
  6. Recommendation (if any)

e. Proposal analysis including
  1. Classification
  2. Resources required to implement change
  3. Effect upon operational personnel and training
  4. Suggested resolution
  5. Reference to associated analysis

f. Change authority including
  1. Disposition
  2. Resolution
  3. Implementation schedule
  4. Authority signature

## Section 5

# Software Project Management Activities

# Contents

Managing a software project is one of the three major activities performed in the software lifecycle; the other two are software development/maintenance and software support (i.e., QA, CM). In concert with the other two activities, managing a software project is an ongoing activity throughout the software lifecycle. It begins in the planning phase and continues through software retirement.

The major software project management activities and the subsections in which they are described are:

- Software Project Management Planning—Section 5.1
- Software Development Planning—Section 5.2
- Software Cost Estimating—Section 5.3
- Software Metrics—Section 5.4
- Scheduling and Tracking—Section 5.5
- Risk Management—Section 5.6

Many of these activities are initially performed during the software planning phase, such as planning, cost estimation, metrics definition, schedule generation, risk definition, and analysis. They are also performed on a periodic and as-needed basis throughout the subsequent phases. For instance, schedule and cost tracking and metrics collection and analysis would be performed on a regular basis. Risk assessment and mitigation, rescheduling, recosting, and replanning would occur as a result of any of a variety of internal or external factors. These factors could include a better understanding of the problem, modifications to the contract, changes in funding or delivery dates, or staff or hardware availability problems.

The last three subsections are tips for project success. These include:

- Do's for Project Success—Section 5.7
- Don'ts for Project Success—Section 5.8
- Danger Signals and Corrective Measures—Section 5.9

*Section 5.1*

# Software Project Management Planning

## Contents

### 5.1.1 Introduction

This section describes the planning involved in preparing to manage a software development/maintenance project. This planning activity is performed during the planning phase of the software lifecycle or earlier during the proposal and contract startup phase (these latter two phases are not described in this document). Involvement by the software development/maintenance personnel and the software support (QA, CM) personnel in the planning activities is necessary in developing a coordinated and realistic plan.

**Benefits of Software Project Management Planning**

- Provides upper management with a high-level summary of the project
- Provides an integrated end-to-end view of the project, including work elements, resources, schedule, and costs
- Provides the basis for risk assessment and the development of risk mitigation strategies
- Provides the customer with the same insight and progress monitoring ability
- Provides the software development/maintenance and software support (QA, CM) staff input and insight into the planning process
- Serves as a vehicle for communication, understanding, and agreement among the software project manager, software developers/maintainers, software support (QA, CM) staff, other contractors, and the customer

Software project management planning culminates in the development of the Software Project Management Plan (SPMP). Whether or not the SPMP is a contractual deliverable, it should nevertheless be produced. The SPMP documents how the software project manager plans to organize and manage the software development and/or maintenance project. Its focus is on the managerial aspects of the project, many of which impact the software development/ maintenance and software support (QA, CM) activities. The SPMP should contain the tasks to be completed, their associated time phasing, and resources necessary to meet the contractual or internal organizational commitments. The plan must clearly define the work that is to be accomplished, the resources and schedules necessary to complete the work, and the management processes needed to direct, monitor, and track progress and costs as well as anticipate and respond to problems.

**Essential Information in the SPMP**

- Project summary description
- Project organization and interfaces with the customer and other contractors
- Managerial processes, including monitoring and controlling mechanisms and risk management
- Work elements, schedule, deliverables, and budget
- Resources, including staffing plan, facilities, and computational and support requirements

### 5.1.2 General Methodology

The following describes the major steps in software project management planning. The order of the steps generally follows the contents in the SPMP. Many of the steps will be repeated at different times in the planning process in order to refine the estimates. Details for cost estimation, scheduling and schedule tracking, software metrics, and risk management methodologies are given in subsequent sections. Only a reference to them will be given.

1. Write a project overview including:
   a. The goals and objectives of the project
   b. Background information
   c. A general description of the work and the deliverable work products
2. Describe the various organizations involved on the project.
   a. Describe the project's organization, roles and responsibilities.
   b. Define the managerial model (hierarchical, matrix) to be used.
   c. Describe the customer and users of the software system.
   d. Describe the customer and users' organizations.
   e. Describe the interfaces between the project organization and the customers and users.
3. Define the managerial processes used to manage the project.
   a. Define the management goals, objectives, and priorities.
   b. Define the assumptions, dependencies, and constraints.
   c. Describe the monitoring, controlling and reporting processes (see Sections 5.4 and 5.5)
   d. Describe the risk management activities (see Section 5.6).
   e. Describe the staffing plan.
4. Define the technical processes to be used on the project.
   a. Define the managerial and technical methods, tools, and techniques to be used.
   b. Describe the site specific technical information.
5. Define the work elements, schedule, and budget.
   a. Define the work packages, deliverables, and dependencies.
   b. Define the schedule (see Section 5.5).
   c. Define the required resources.
   d. Define the budget and resource allocations (see Section 5.3).

### 5.1.3 Tailoring to a Small Project

Each project is unique. Tailoring the information provided in this section is essential in defining and implementing the software project management planning function for a specific project. Regardless of size, the software project management planning function needs to be performed. Only the level of detail and formality of the process and products vary among projects.

Steps in tailoring the software project management planning function include the following:

1. Review the box entitled "Essential Information in the SPMP."
2. Review the "General Methodology" steps, noting whether and how each applies to your project.
3. Review and note applicable information from the software development planning, cost estimation, metrics, scheduling and tracking, and risk management sections.
4. Develop an annotated outline for the SPMP. Decide whether this document should be combined with other planning documents (e.g., the Software Development Plan [SDP]).

5. Write a draft SPMP and have it reviewed by other software managers, developers/ maintainers, and software support staff who will work on the project.

6. Revise the SPMP.

## 5.1.4 Suggested Reference Material

Fairley, Richard E., "A Guide for Preparing Software Project Management Plans," *IEEE Tutorial: Software Engineering Project Management*, Richard H. Thayer, Computer Society Press of the IEEE, 1988, pp. 257–264.

"IEEE Standard for Software Project Management Plans," IEEE-STD-1058.1, ANSI/IEEE Std 1051.1-1987, December 1987.

Automatic Dependent Surveillance (ADS) Project Management Plan, Hughes STX Corp., August 1991.

*Software Engineering Handbook, Build* 3, Division 48, Information Systems Division, Hughes Aircraft Company, 1992.

Work Control Plan, NASA Space and Earth Sciences Contract, Technical Applications Group, Hughes STX Corp.

Program Management Plan, Data Item Description, Backgrounds Data Center Contract, Naval Research Laboratory.

## 5.1.5 Appendixes

### 5.1.5.1 Software Project Management Planning Checklists

| Schedule Checklist | |
|---|---|
| **Y/N** | **Check** |
| | Have the schedules been constructed at the appropriate level of detail? |
| | Have software schedules been coordinated with other schedules (such as hardware delivery, training, CM, QA)? |
| | Has the critical path been identified? |
| | Have project-specific standards for software schedules been established? |
| | Have dates for deliverables and customer reviews been established/planned? |
| | Are any prototypes or models planned early enough to provide useful results? |

| Personnel Checklist | |
|---|---|
| **Y/N** | **Check** |
| | Has the initial organization been devised? |
| | Have staffing levels been projected and compared to the budget? |

| **Personnel Checklist (Continued)** | |
|---|---|
| | Have commitments been received from key personnel? |
| | Is there a plan to acquire the remaining staff at the appropriate times? |
| | Has a design team of senior-level staff been established/planned? |
| | Has a training plan been established? |

| **Facilities Checklist** | |
|---|---|
| **Y/N** | **Check** |
| | Has the usage of a development facility been estimated? |
| | Have adequate development facilities been planned (whether company-owned, borrowed, onsite, etc.) |
| | Has the development system (hardware and software such as editors, compilers, debuggers, and CM tools) been exercised to determine that software can successfully be developed? |
| | Have arrangements been made for any security requirements? |

| **Coordination Checklist** | |
|---|---|
| **Y/N** | **Check** |
| | Have document deliveries been coordinated with supporting organizations (QA, CM, reproduction)? |
| | Has acquisition of COTS items been coordinated with the Purchasing Department? |
| | Have regular meetings been established with necessary parties (including superiors)? |
| | Is there a plan for disseminating information within the project staff (staff meetings, email, bulletin boards, newsletters)? |

| **Checklist for Other Items** | |
|---|---|
| **Y/N** | **Check** |
| | Has a risk management plan been developed and implemented? |
| | Has the proposal been reviewed for commitments regarding software methodology, products, and schedules? |
| | Has a set of metrics been established to measure technical performance? |
| | Has a set of metrics been established to measure progress? |
| | For each metric, has a collection plan and threshold/expected value been established? |
| | Have reporting methods and formats (from subordinates to superiors) been established? |

### 5.1.5.2 Tables of Contents

**Software Project Management Plan (SPMP)—Table of Contents**
(IEEE Standard 1058.1-1987)

1.0 Introduction
- 1.1 Project overview
- 1.2 Project deliverables
- 1.3 Evolution of the SPMP
- 1.4 Reference materials
- 1.5 Definitions and acronyms

2.0 Project Organization
- 2.1 Project model
- 2.2 Organization structure
- 2.3 Organizational boundaries and interfaces
- 2.4 Project responsibilities

3.0 Managerial Process
- 3.1 Management objectives and priorities
- 3.2 Assumptions, dependencies, and constraints
- 3.3 Risk management
- 3.4 Monitoring and controlling mechanisms
- 3.5 Staffing plan

4.0 Technical Process
- 4.1 Methods, tools, and techniques
- 4.2 Site information for computer operations personnel
- 4.3 Project support functions

5.0 Work Elements, Schedule, and Budget
- 5.1 Work packages
- 5.2 Dependencies
- 5.3 Resource requirements
- 5.4 Budget and resource allocation
- 5.5 Schedule

**Work Control Plan—Table of Contents**

1.0 Introduction
- 1.1 Science/technical objectives
- 1.2 Customer organization
- 1.3 General background

2.0 Requirements/statement of work
- 2.1 Stated and derived requirements
- 2.2 Assumptions and constraints

3.0 Work Breakdown Structure (WBS)

4.0 Technical Approach
- 4.1 Applicable software lifecycle model, phases, and reviews
- 4.2 Development/maintenance methodologies, procedures, and tools
- 4.3 Software support (CM, QA) methodologies, procedures, and tools
- 4.4 Project/task-specific software standards
- 4.5 Candidate software reuse and COTS products
- 4.6 Risk assessment and mitigation approach

5.0 Performance Schedule
- 5.1 Activities
- 5.2 Milestones
- 5.3 Monitoring and controlling mechanisms
- 5.4 Reviews

6.0 Deliverables

7.0 Resources
- 7.1 Staffing plan
- 7.2 Subcontractor role
- 7.3 Computer, networks, and other equipment
- 7.4 Government-Furnished Equipment (GFE) and GFI (interface) dependencies
- 7.5 Other

*Section 5.2*

# Software Development Planning

## Contents

## 5.2.1 Introduction

This section describes the activities involved in planning for software development. This planning activity is performed during the planning phase of the software lifecycle or earlier during the proposal and contract startup phase (these latter two phases are not described in this document). Involvement by the software development/maintenance personnel and the software support (QA, CM) personnel in the planning activities is necessary in developing a coordinated and realistic plan.

**Benefits of Software Development Planning**

- Provides an early understanding of how the software will be developed for each phase of development
- Provides a detailed plan for software development before development begins
- Raises questions early in the lifecycle regarding development approaches, procedures, etc.
- Provides the software development/maintenance and software support (QA, CM) staff input into the planning process
- Serves as a vehicle for communication, understanding, and agreement among software project manager, software developers/maintainers, software support (QA, CM) staff, other contractors, and the customer

Software development planning culminates in the development of the SDP. Whether or not the SDP is a contractual deliverable, it should nevertheless be produced. The SDP documents how the software will be developed. Its focus is on the technical development aspects of the project, including both software development and software support (QA, CM) activities. The SDP describes the lifecycle model representing the development phases and contains the methods and procedures to be implemented and followed by software development and software support staff for each phase of development. It lays out the development schedule, indicating development and software support activities and milestones, contractual and informal reviews, and software deliveries. In essence, it is the road map for software development.

**Essential Information in the SDP**

- Project summary description
- Software lifecycle model indicating development phases, reviews, and deliverables
- Software development and software support (QA, CM) functions and organizations
- Descriptions of both development and software support (QA, CM) methods and procedures for each phase

The SDP should be viewed as a working document. That is, as the work becomes clarified, better approaches chosen, etc., the SDP should be updated to reflect those changes. Such changes may occur during any of the software lifecycle phases.

## 5.2.2 General Methodology

The following describes the major steps in software development planning.

**Software Development Scope**

1. Write a system overview (a system context diagram may prove useful) including:

   a. The general nature of the system and software
   b. Summary of the history of system development, operation, and maintenance
   c. Identification of the project sponsor, user, developer, and support agencies
   d. Identification of the current and planned operating sites

**Software Development Planning—General**

2. Write an overview of the work to be accomplished including:

   a. The system and software to be developed
   b. The documentation required
   c. Overview of the system lifecycle and the position of the project within that lifecycle
   d. The software lifecycle model to be used
   e. Project schedules and resources
   f. Other aspects of the project, such as security, privacy, methods, standards to be followed, and testing constraints

3. Define the overall software development process to be used, including:

   a. The lifecycle model for software devlopment, including phases, products, reviews, and deliverables.
   b. A mapping of the activities required by contract provisions onto that portion of the software lifecycle model.

4. Define the general plans for software engineering, including:

   a. The software development methodologies to be used, by phase, including the tools and procedures to be used in support of these methods.
   b. The approach to be followed for identifying, evaluating, and incorporating COTS and reusable software.
   c. The approach to be followed for safety analysis.

5. Define the general plans for software testing, including:

   a. The testing methodologies to be used by phase, including the tools and procedures to be used to support these methods.
   b. The approach for planning, conducting, evaluating tests and responding to test failure.
   c. Achieving the required level of independence, including testing on the target computer system or an equivalent system.

**Software Development Planning—Details**

6. Define the approach to be followed for subsequent planning, including:

   a. Further development of this SDP.
   b. Planning of the software system and software system integration testing.
   c. Performance of or participation in planning system testing.
   d. Planning for transition to software support.
   e. Planning for software installation and training at user sites.

7. Define the approach to be followed for establishing, controlling, and maintaining a software development environment, including descriptions of:

   a. The software engineering environment.
   b. The software test environment.

   c. The Software Development Library (SDL)
   d. The Software Development Files (SDFs)
   e. Design and coding standards to be used
   f. Nondeliverable software to be used
   g. Any other software standards and procedures to be used

8. Define the approach to be followed for performing or participating in system requirements analysis, including:

   a. Analyzing user input
   b. Defining the operational concept
   c. Defining the system requirements

9. Define the approach to be followed for performing or participating in system design analysis, including:

   a. Developing the system behavioral design
   b. Developing the system architectural design

10. Define the approach to be followed for software requirements analysis, including:

    a. Defining the software system engineering (and corresponding qualification) requirements
    b. Defining the software system interface (and corresponding qualification) requirements

11. Define the approach to be followed for performing software architectural design, including:

    a. Developing the software system behavioral design
    b. Developing the software system architectural design
    c. Developing the database logical design

12. Define the approach to be followed for performing software detailed design, including:

    a. Developing the software system detailed design
    b. Developing the software system interface design
    c. Developing the database physical design

13. Define the approach to be followed for coding and unit testing including the programming language(s) to be used, including:

    a. Coding software units
    b. Populating those databases to be populated as part of software development
    c. Preparing for unit testing
    d. Performing unit testin
    e. Revising and retesting based on test results
    f. Recording unit test results

14. Define the approach to be followed for software subsystem integration and testing, including:

    a. Preparing test cases and test data (possible use of simulators)
    b. Preparing test procedures
    c. Performing dry runs of test procedures
    d. Performing software subsystem integration and testing
    e. Revising and retesting based on test results
    f. Analyzing and recording software subsystem integration and test results
    g. Updating software subsystem integration and test cases and procedures

15. Define the approach to be followed for software system testing, including the same items as for software subsystem integration and test.

16. Define the approach to be followed for performing or participating in system acceptance testing, including the same items as for software subsystem integration and test, plus customer-witnessed testing.

17. Define the approach to be followed for preparing for software use and support, including:

    a. Developing software users and maintenance manuals
    b. Developing computer system operator manuals
    c. Performing installation and training at user sites
    d. Transitioning software and environments to the designated support site

18. Define the approach to be followed for preparing for software delivery, including:

    a. Preparing executable code for delivery
    b. Preparing source code for delivery
    c. Developing software product specifications
    d. Developing version descriptions
    e. Supporting Functional Configuration Audit(s) (FCA)
    f. Supporting Physical Configuration Audit(s) (PCA)

19. Define the approach to be followed for performing software process and product evaluations (or reference the QA plan), including:

    a. Performing in-process software process and product evaluations
    b. Performing final software product evaluations

20. Define the approach to be followed for performing software CM (or reference the CM plan), including:

    a. Configuration identification, control, status, and audits of development products
    b. Interface with customer CM, including:
        1) Supporting the baselining of specifications
        2) Using ECPs/ECRs
        3) Configuration status accounting, including the format, content, and purpose of reports to be used
    c. Storage, handling, and delivery of project media

21. Define the approach to be followed for performing corrective action and process improvements.

22. Define the approach to be followed for holding joint (customer/contractor) reviews.

**Software Development Planning—Schedules**

23. Define the schedules for the project (or reference the SPMP), including:

    a. Schedule(s) identifying the activities in each build and showing initiation of each activity, availability of draft and final deliverables and other milestones, and completion of each activity
    b. An activity network (e.g., PERT chart), depicting sequential relationships and dependencies among activities and identifying those activities that impose the greatest time restrictions on the project

**Software Development Planning—Project Organization and Resources**

24. Define the project organization and resources (or reference the SPMP), including:

a. The organizational structure to be used on the project, including the organizations involved, their relationships to one another, and the authority and responsibility of each organization for carrying out required activities
b. The resources to be applied to the project, including:
    1) Personnel resources
    2) Overview of contractor facilities to be used
    3) Customer-furnished items, facilities required, and dates needed
    4) Training needs
    5) Other required resources, a plan for obtaining them, and need/availability dates

### 5.2.3 Tailoring to a Small Project

Each project is unique. Tailoring the information provided in this section is essential in defining and implementing the software development planning function for a specific project. Regardless of size, the software development planning function needs to be performed. Only the level of detail and formality of the process and products vary among projects.

Steps in tailoring the software development planning function include the following:

1. Review the box entitled "Essential Information in the SDP" and note how these will apply to your project.

2. Review the "General Methodology" steps and note what is applicable and how it will be applied to your project.

3. Write a draft SDP. Tips: a) Use references to other documents (e.g., SPMP) rather than duplicating the material, b) possibly combine documents such as the SPMP and SDP, addressing only the applicable information, and c) use TBDs only for sections that you actually intend to update in the future.

4. Have developers/maintainers and software support (QA, CM) personnel review and comment on the draft SDP.

5. Update and finalize the written SDP after consensus is reached by the software manager, developers/maintainers, and software support staff.

### 5.2.4 Suggested Reference Material

"IEEE Standard for Software Development Plans," IEEE-STD-1058.1, ANSI/IEEE Std 1051.1-1987, December 1987.

Automatic Dependent Surveillance (ADS) Development Plan, Hughes STX Corp., August 1991

*Software Engineering Handbook, Build 3*, Division 48, Information Systems Division, Hughes Aircraft Company, 1992.

## 5.2.5 Appendixes

### 5.2.5.1 Checklists

| Project-Specific Standards and Practices | |
|---|---|
| **Y/N** | **Check** |
| | Have methodologies been selected for requirements analysis and preliminary and detailed design? |
| | Have project-specific coding standards been documented? |
| | Has a procedure for walkthrough been established? |
| | Has a procedure for action items been established? |
| | Has a Software CM Board (or equivalent) been established to provide software review of requirements changes and problem reports? |

### 5.2.5.2 Tables of Contents

**Software Development Plan—Table of Contents**
(Iceland Air Defense System [IADS])

1.0 Scope
- 1.1 Identification
- 1.2 Purpose
- 1.3 Introduction

2.0 Referenced Documents
- 2.1 Government Documents
- 2.2 Non-Government Documents
- 2.3 Other Publications

3.0 Resources and Organization
- 3.1 Project Resources
- 3.2 Software Development
- 3.3 Software Configuration Management
- 3.4 Software Quality Evaluation
- 3.5 Other Software Development Functions

4.0 Development Schedule and Milestones
- 4.1 Activities
- 4.2 Activity Network
- 4.3 Procedures for Risk Management
- 4.4 Identification of High-Risk Areas

5.0 Software Development Procedures
- 5.1 Software Standards and Procedures
- 5.2 Software Configuration Management
- 5.3 Software Quality Evaluation
- 5.4 Additional Software Development Procedures
- 5.5 Commercially Available, Reusable, and Government-Furnished Software
- 5.6 Data Rights and Documentation
- 5.7 Nondeliverable Software, Firmware, and Hardware Controls
- 5.8 Software Developed for Hardware Configuration Items
- 5.9 Installation and Checkout
- 5.10 Interface Management

6.0 Notes
- 6.1 Abbreviations and Acronyms
- 6.2 Glossary
- 6.3 Changes Since Last Delivery

**Software Development Plan—Table of Contents**
(Software Engineering Handbook [DIV 48])

1.0 Introduction

2.0 Software Development Management

3.0 Software Engineering

4.0 Testing

5.0 Software Product Evaluations

6.0 Software Configuration Management

7.0 Software Quality Assurance

**Consolidated Software Plan—Table of Contents**
(MIL-STD-SDD, Draft December 1992)

1.0 Scope
- 1.1 Identification
- 1.2 System overview
- 1.3 Document overview

2.0 Referenced Documents

3.0 Software development planning
- 3.1 Overview of the work to be done
- 3.2 General requirements
- 3.3 Detailed requirements
- 3.4 Schedules
- 3.5 Project organization and resources

4.0 Software Installation Planning
- 4.1 Installation overview
- 4.2 Site information for computer operations personnel
  - 4.2.x (Site name)
- 4.3 Site information for user personnel
  - 4.3.x (Site name)

5.0 Software Support Planning
- 5.1 Software support resources
- 5.2 Recommended procedures
- 5.3 Training
- 5.4 Anticipated areas of change
- 5.5 Transition planning

6.0 Notes

Appendixes

*Section 5.3*

# Software Cost Estimating

## Contents

This section presents guidelines for estimating the size, cost, and schedule of software development projects. Figures 5.3.1-1 and 5.3.1-2 are process flow diagrams for determining software costs and schedule. Figure 5.3.1-3 illustrates a process flow for determining software size. The information in this section was derived from that illustration.

*Note: All written estimates of labor, costs, size, and schedules, including rough estimates, normally require approval of an HSTX department manager or above before they can be given to a customer or another contractor.*

## 5.3.1 General Procedure for Cost Estimating

Estimating the size, cost, and schedule for software development projects should follow a well-defined, systematic approach that provides an *auditable trail* from beginning to end. This is especially important on proposal efforts that require a basis of estimate for the software development costs. At the outset of the estimating process, establish a mechanism for tracking and saving (CM) the various estimation products. Bidding and marketing strategies are not included; they are beyond the scope of this guidebook.

The output of the software estimation activity is the cost of all efforts necessary to perform the software development. The software estimation process consists of the following ordered activities:

1. Develop a system design.
2. Define the size of the software system.
3. Define the environmental factors (these are the cost factors that are input to a cost).
4. Execute the software costing model(s).
5. Develop a project estimate.
6. Perform risk analysis.
7. Develop a project bid.
8. Perform dynamic cost projection.

Each of these steps is explained in the next section.

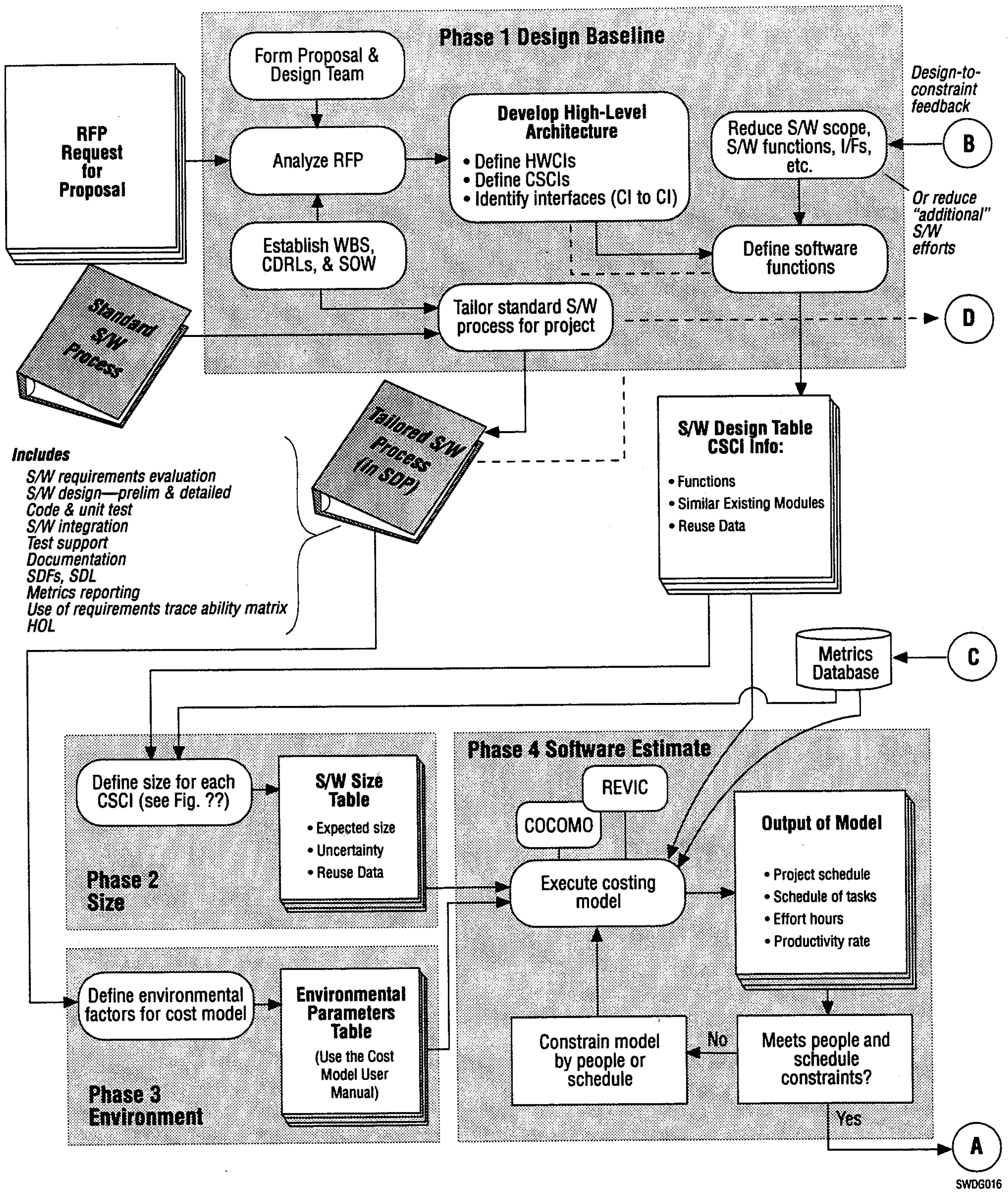
## 5.3.2 Detailed Procedure

The following steps detail the procedures for cost estimating:

1. Develop a system design. The following are the steps for the system design phase of cost estimation:

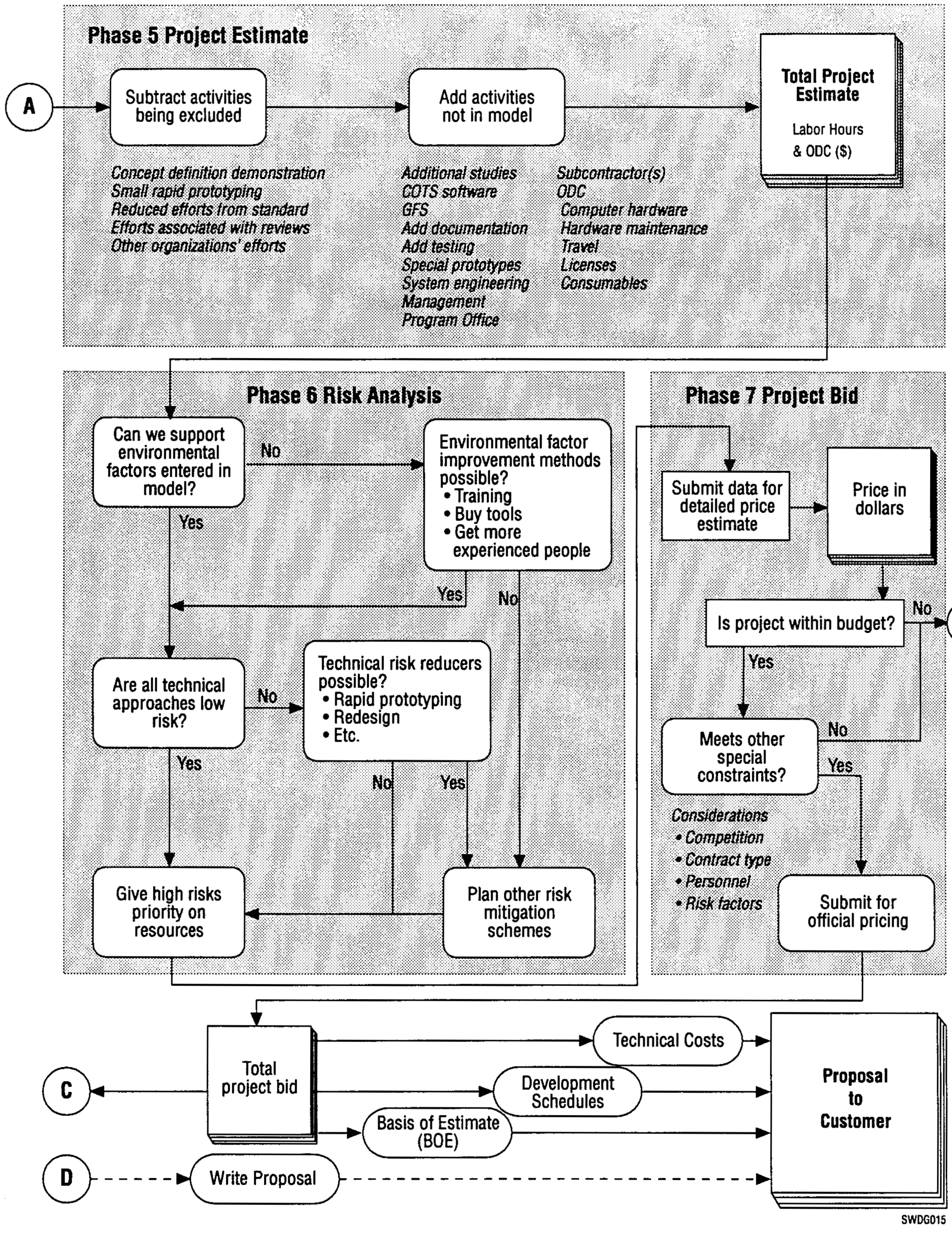
   a. Form a proposal team. The proposal/design team should include a program/project manager, systems engineers, hardware engineers (when necessary), independent test, and software engineers. To provide a mix of experience and viewpoints, inclusion of at least three software engineers, of whom two have had prior software estimation experience, is recommended.

*Note: This procedure is designed for estimating large projects. Smaller projects may call for fewer people. You should always have at least two people on the team, to trade ideas.*


Phase 1 Design Baseline
RFP Request for Proposal
Form Proposal & Design Team
Analyze RFP
Establish WBS, CDRLs, & SOW
Develop High-Level Architecture
• Define HWCIs
• Define CSCIs
• Identify interfaces (CI to CI)
Design-to-constraint feedback
Reduce S/W scope, S/W functions, I/Fs, etc.
B
Or reduce "additional" S/W efforts
Define software functions
Tailor standard S/W process for project
D
Standard S/W Process
Tailored S/W Process (in SDP)
Includes
S/W requirements evaluation
S/W design—prelim & detailed
Code & unit test
S/W integration
Test support
Documentation
SDFs, SDL
Metrics reporting
Use of requirements trace ability matrix
HOL
S/W Design Table CSCI Info:
• Functions
• Similar Existing Modules
• Reuse Data
Metrics Database
C
Define size for each CSCI (see Fig. ??)
S/W Size Table
• Expected size
• Uncertainty
• Reuse Data
Phase 2 Size
Define environmental factors for cost model
Environmental Parameters Table
(Use the Cost Model User Manual)
Phase 3 Environment
Phase 4 Software Estimate
COCOMO
REVIC
Execute costing model
Output of Model
• Project schedule
• Schedule of tasks
• Effort hours
• Productivity rate
Constrain model by people or schedule
No
Meets people and schedule constraints?
Yes
A
SWDG016

[ISD48]

**Figure 5.3.1-1. Software Cost Estimation Process (Page 1 of 2)**



[ISD48]

**Figure 5.3.1-2. Software Cost Estimation Process (Page 2 of 2)**

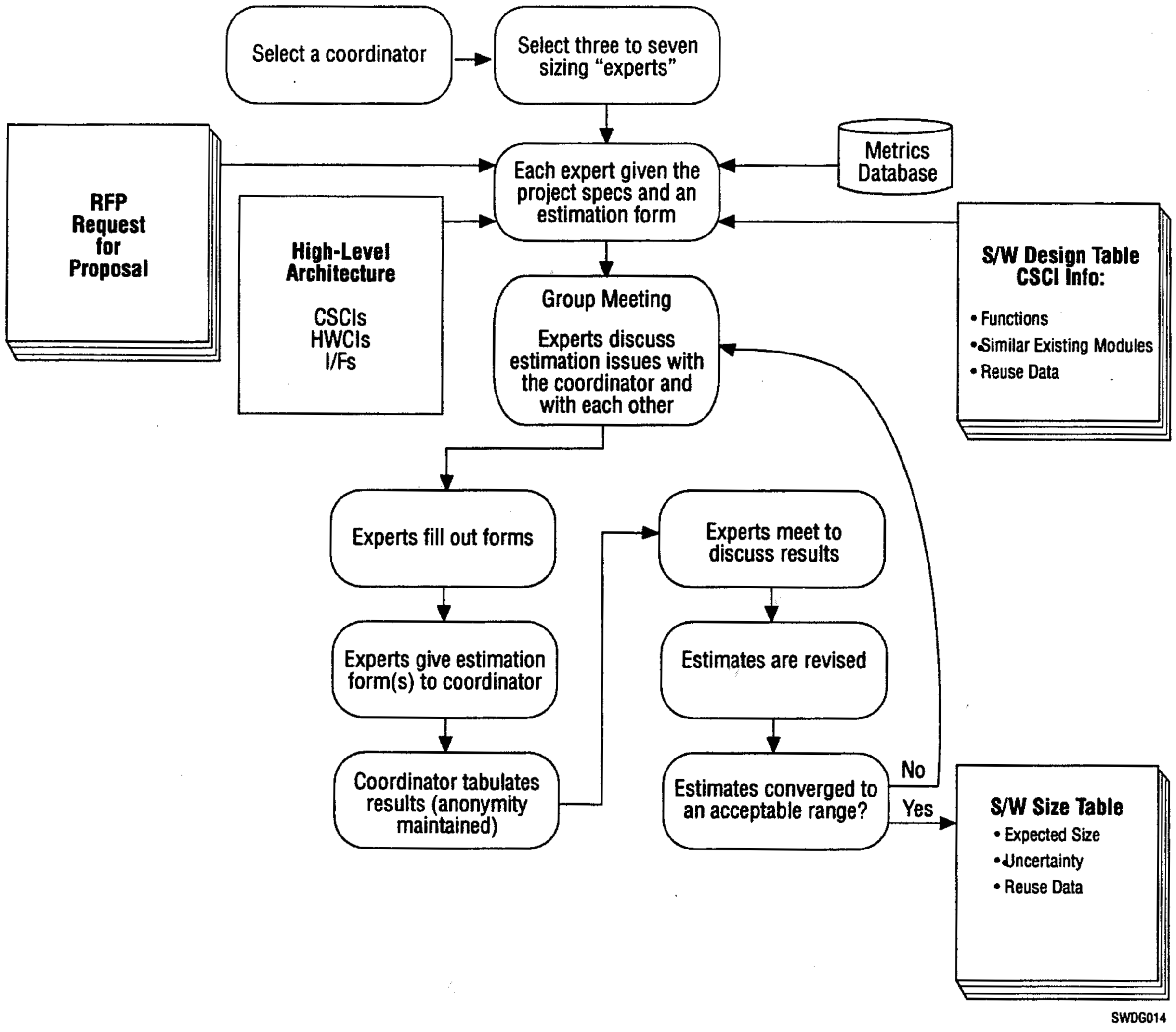

Select a coordinator
Select three to seven sizing "experts"
RFP Request for Proposal
Each expert given the project specs and an estimation form
Metrics Database
High-Level Architecture
CSCIs
HWCIs
I/Fs
S/W Design Table CSCI Info:
• Functions
• Similar Existing Modules
• Reuse Data
Group Meeting
Experts discuss estimation issues with the coordinator and with each other
Experts fill out forms
Experts meet to discuss results
Experts give estimation form(s) to coordinator
Estimates are revised
Coordinator tabulates results (anonymity maintained)
Estimates converged to an acceptable range?
No
Yes
S/W Size Table
• Expected Size
• Uncertainty
• Reuse Data
SWDG014

**Figure 5.3.1-3. Software Size Estimation Process Wideband Delphi Method**

b. Analyze the RFP and other system documents such as the System Specification and the Software Requirements Specification (SRS), if available.

c. Establish the Work Breakdown Structure (WBS), Contract Data Requirements List (CDRL), and SOW; with the Project Manager, if these are not defined in the contract. These elements, the basis for a Project Management Plan, set the bounds on the scope of the problem. On small task order efforts, the single input may be the task's SOW or verbal input from the customer.

d. Develop a high-level system architecture. The architecture consists of:

   1) Selection and identification of hardware configuration items (if you have them)
   2) Selection and identification of software systems and subsystems
   3) Identification of interfaces from software subsystem to software subsystem
   4) Results of trade studies, if necessary

e. Define the functions of each software subsystem in a Software Estimation Design Table. This table includes the names and functions of each of the software subsystems.

f. Develop a tailored software process. This guidebook constitutes a recommended HSTX software process. Tailor the standard process and include it directly or by reference in an SDP. The list below constitutes the minimum for a tailored software development process; this assumes the SRS has been developed.

 1) Evaluation of software requirements
 2) Software preliminary and detailed design phases
 3) An implementation and unit test phase
 4) Support of software integration by the software developer
 5) Software testing by an Independent Test Organization (ITO)
 6) Documentation as described in contract/task requirements
 7) The use of SDFs
 8) The use of an SDL
 9) Software metrics collection and reporting
 10) The use of a requirements traceability matrix
 11) The use of a High-Order Language (HOL)

 If the software organization is responsible for the development of the SRS, additional effort for this activity needs to be included in the software estimate.

2. Define the size of each software system in Source Lines of Code (SLOC). (Refer to Figure 5.3-3, a process flow for determining software size.) This process consists of the following steps, using the Wideband Delphi Technique originated by the Rand Corporation:

[*BOE81, pp. 333-336*]

a. A person who will not be performing detailed sizing is selected as coordinator.

b. Three to seven software experts with experience in software sizing are chosen. Fewer people may be used for smaller projects. (You must use at least two people for this to work at all; three are better.)

c. The coordinator presents each expert with the system specifications, an estimation form, and a list of modules, including their sizes, from past experience that are similar to those being developed. The historical data could be from a metrics database. (A sample of the estimation form is given in Figure 5.3.3-1.)

d. The coordinator calls a group meeting in which the experts discuss estimation issues with the coordinator and with each other.

e. The experts fill out forms (anonymously). A sample is given in Figure 5.3.3-1.

f. The coordinator prepares and distributes a summary of the estimates at the top of the iteration form. This is shown in Figure 5.3.3-1.

g. The coordinator calls a group meeting specifically for the experts to discuss any points where their estimates vary widely.

h. The experts fill out the bottom of the iteration form, again anonymously, and Steps e through g are iterated for as many rounds as appropriate (until the estimates converge to an acceptable range).

i. The coordinator ensures that the output for each software system includes the software system name, expected (estimated) size, the estimation uncertainty (standard deviation or low/high spread), and reuse information (such as previous size, percentage to be redesigned, percentage of code to be changed, and percentage of code to be retested).

3. Define the development environmental factors for each software system. Use the cost model user manual (see next step) to determine the environmental factor settings for the development.

4. Execute the software costing model(s). The models are computer resident models for estimating the effort and schedules for software development projects. Use either the Revised Intermediate Constructive Cost Model (REVIC) *(available in the Software Engineering Laboratory [SEL] at HSTX)* or the Constructive Cost Model (COCOMO) *(also available in the SEL)*, or both. These models produce estimates of project schedule, schedule of phases, labor hours for the software effort, and productivity rate. If the model output meets schedule or staffing constraints, proceed to the next step. Otherwise, rerun the model and constrain either schedule or people, whichever is the more important constraint. (You cannot constrain both schedule and people.)

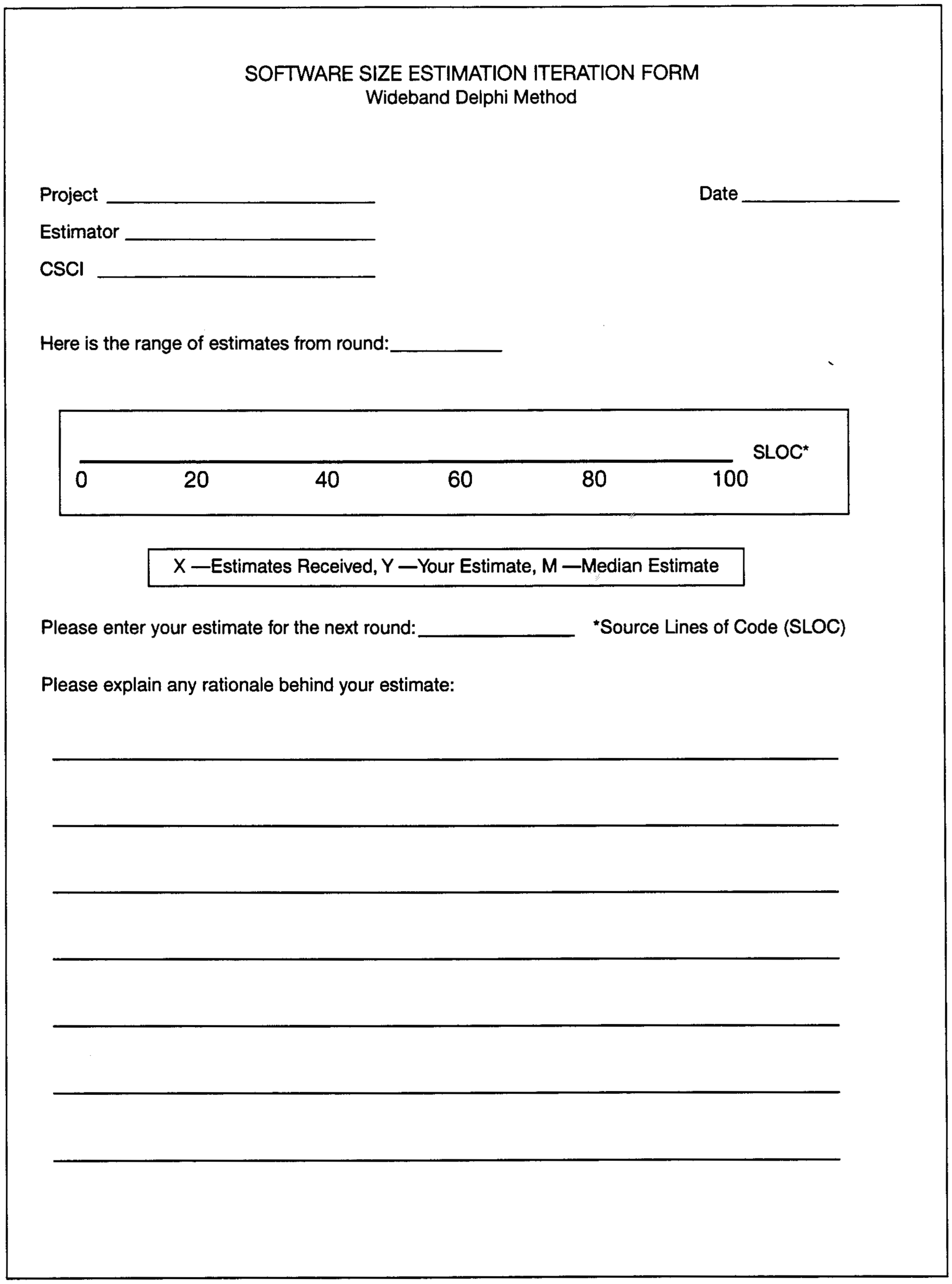
*Note: The REVIC model is based on the COCOMO model, and its user interface is easier to use. The COCOMO model is implemented as a Microsoft Excel spreadsheet and has fewer constraints than REVIC. REVIC has some schedule constraints that make very short development schedules impossible to model. See the REVIC User's Manual for additional details.*

5. Develop a project estimate. Using the output of the costing model, subtract activities included in the model but not in the development process, such as documentation, formal reviews, or efforts by other organizations. Also subtract items not appropriate for a tailored software process, as with a small rapid prototyping effort. Add activities required for the project, but not included in the model. For example:

   a. Additional studies
   b. COTS software
   c. Familiarization with and testing of Government-furnished software
   d. Additional documentation
   e. Additional testing
   f. Special prototypes
   g. Security
   h. Subcontractors
   i. Computer operators
   j. Management *
   k. Program Office *
   l. Systems engineering *
   m Independent Testing *
   n. Software QA *
   o. CM *
   p. Other Direct Costs (ODCs) (nonlabor expense), such as computer hardware, maintenance, travel, licenses, and consumables

   The items marked * are included in some models. The COCOMO model which is implemented as an Excel spreadsheet in the SEL, has these factors included. See the REVIC User's Manual for information about the factors that it includes.

6. Perform risk analysis. Risk analysis is where the "doability" of the project is assessed. Can this project be done as costed? Can the environmental factors be supported? *(Are you really going to use Computer-Aided Software Engineering [CASE] tools? Will the expert programmers be available for this job?)* What are the areas of greatest technical risk? Is there new technology? As a result of the analysis, add any risk mitigation costs, and ensure that high-risk activities get highest priority on resources (people and equipment).



SOFTWARE SIZE ESTIMATION ITERATION FORM
Wideband Delphi Method

Project ____________________ Date ____________

Estimator ____________________

CSCI ____________________

Here is the range of estimates from round: __________

0 20 40 60 80 100 SLOC*

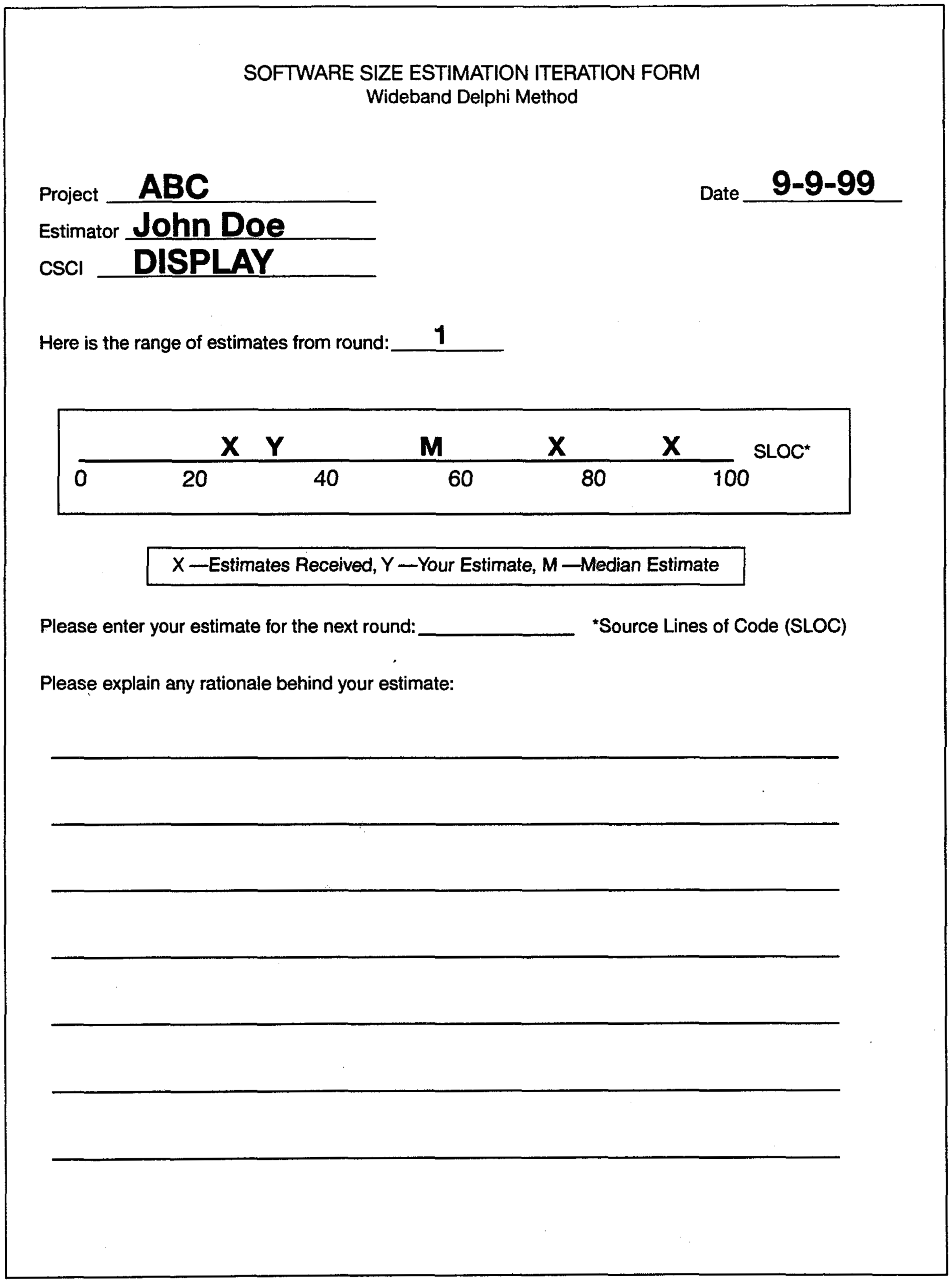
X —Estimates Received, Y —Your Estimate, M —Median Estimate

Please enter your estimate for the next round: __________ *Source Lines of Code (SLOC)

Please explain any rationale behind your estimate:

[ISD48]

Figure 5.3.3-1. Software Size Estimation Iteration Form (Whiteband Delphi Method



SOFTWARE SIZE ESTIMATION ITERATION FORM
Wideband Delphi Method

Project ABC

Estimator John Doe

CSCI DISPLAY

Date 9-9-99

Here is the range of estimates from round: 1

```
                 X  Y          M          X          X    SLOC*
 ____________________________________________________________
0        20        40        60        80        100
```

X —Estimates Received, Y —Your Estimate, M —Median Estimate

Please enter your estimate for the next round: ________ *Source Lines of Code (SLOC)

Please explain any rationale behind your estimate:

[ISD48]

**Figure 5.3.3-1. Software Size Estimation Iteration Form (Whiteband Delphi Method**

7. Develop a project bid. Notice that the project bid is different from the project estimate. The bid depends on many factors, including the project estimate. These factors include such things as the importance of winning this contract and the willingness of the company to invest additional funds in the likelihood of winning follow-on work. The project bid consists of the following steps:

   a. Get a detailed price estimate. "Cost" is transformed into "price." The pricing system uses straight-time labor rates and ODC to build a total price for the project. The pricing system includes overhead, General and Administrative (G&A) costs, cost of money, and profit. Consult the HSTX Contracts Department for details on pricing. Once a price estimate is completed, you may find your costs above a budgeted number. It may then be necessary to start over at Step 1 to reduce the scope of the project.
   b. Consider special constraints and apply associated costs.
   c. Submit the data for official pricing using the HSTX pricing system (see the HSTX Contracts Department).
   d. Assemble and compile the bid with other portions of the proposal effort. Coordinate with the project manager and/or your immediate supervisor before presenting the proposal to the customer.

8. Perform dynamic cost projection throughout the project. Dynamic cost projection is the software costing activity in which the project tracks estimates and factors throughout the project. Tracking the estimates throughout the project provides historical data for future projects.

*Note: The HSTX Software Engineering Process Group (SEPG) is responsible for storing and making available these data for other projects. See the SEPG lead for additional details.*

### 5.3.3 Estimating Contingencies

Take care in estimating software size. Estimating size is critical and is often inaccurate. Sizes are invariably underestimated. Watts S. Humphreys says "Code growth is the most important single factor in cost and schedule overruns." [HUN89] Barry Boehm explains in *Software Engineering Economics* that underestimation of software size is caused by three factors [BOE81, 320–321]:

- People are basically optimistic and desire to please.
- People tend to have incomplete recall of previous experience.
- People are generally not familiar with the entire software job.

The point is to choose estimating experts carefully and to take enough care in estimating sizes.

### 5.3.4 Cited References

[ISD48] *Software Engineering Handbook, Build 3*, Division 48, Information System Division, Hughes Aircraft Company, March 1992.

[HUN89] *Managing the Software Process*, The SEI Series in Software Engineering, Addison-Wesley Publishing Company, Inc., Reading, MA, 1989, pp. 92–96.

[BOE81] Boehm, Barry W., *Software Engineering Economics*, Prentice Hall, Inc., Englewood Cliffs, NJ, 1981, pp. 333–336.

*Section 5.4*

# Software Metrics

## Contents

## 5.4.1 Introduction

This section describes the collection and uses of software metrics. Software metrics are measurable indications or attributes of a software development process, product, or project. These metrics provide management with a means to monitor a project. Both objective and subjective measures are important to consider when assessing the current state of the project [*SEL-81-101*]. Objective data consist of actual counts of items, while subjective data are based on feelings about a characteristic or condition (e.g., level of difficulty of a problem, stability of the requirements, etc.) [*SEL-81-101*]. Software metrics should be collected and reported throughout the software lifecycle, although different software metrics may be required during various phases of software development and maintenance. Metrics are used by senior management, project management, proposal teams, software engineers, software developers, QA individuals, and CM.

*The primary objective for collecting data is ultimately to produce a quality product [MPI92].*

The benefits derived from collecting, analyzing, and reporting metrics include the following:

- Measuring and improving the software development process
- Determining trends and predicting problem areas
- Monitoring and tracking product quality
- Measuring, predicting, and improving software product quality
- Monitoring and tracking project progress
- Measuring and improving productivity
- Calibrating models
- Identifying complex computer program modules
- Producing realistic schedules
- Gathering data for better estimation of future bids

The following are personal benefits of using metrics for the managers [*MPI92*]:

- Improved communication with customers, managers, and fellow employees
- Improved resource management
- Improved employee morale
- Ability to visualize and generate goals
- Increased quality of the software development process
- Ability to assess the process, product, and project
- Assessment of the process and products to help identify specific areas of improvement
- Better visibility of the process

The following are personal benefits of using metrics for engineers, developers, testers, CM, and QA individuals [*MPI92*]:

- Improved communication with customers, managers, and fellow employees
- Historical data to aid in allocating appropriate time and resources for a project
- Improved quality of the product resulting in an improved personal image
- A more consistent and predictable process or product
- Collected data that show where the process needs streamlining

- Collected data to justify the need for tools to streamline the process

A Software Metrics Group (SMG)—a subgroup of the SEPG—should be formed and should document and maintain guidelines for software metrics. These guidelines should apply to all HSTX projects that involve the generation and/or maintenance of software. It is the responsibility of individual project managers to use the software metrics guidelines.

The estimates of progress for current software projects or the estimates of the required effort for future software projects are often essentially crude guesses, if metrics are not collected and used. Metrics collection provides a quantitative method by which progress on a project can be tracked and forecasts generated. Historical data can be collected from several projects to make projections with less risk, quantitative estimates of the risk, and projections with increased schedule and effort estimation accuracy. A historical database should be established and maintained by the SMG to gather metrics data. The metrics data collected can be used to determine trends, calibrate models, and support proposal teams.

Metrics should be used to measure processes and products and refine processes to decrease error density and increase productivity; they should not be used to evaluate people. If metrics were used to evaluate people, it is unlikely that the metrics would be collected in an accurate manner.

## 5.4.2 Metrics Details

*"The best criteria for the value of a metric is the degree to which it helps us make a decision."*

—Barry Boehm [*MMI92*]

### 5.4.2.1 The Goal/Question/Metric Paradigm

Victor Basili has defined the concept of the Goal/Question/Metric (G/Q/M) paradigm. The G/Q/M paradigm is a mechanism for defining and evaluating a set of operational goals, using measurement [*BAS90*]. The G/Q/M model is an approach for deriving goals for a specific organization. After determining the goals of the organization, appropriate metrics can be used to support the process of attaining the goals. This model includes the following basic steps [*MPI92*]:

1. Identify improvement goals for the process, product, or project. This step can be further divided into the following steps [*MPI92*]:
    a. Identify the stakeholders. Stakeholders can include the customer, end users, developers, testers, QA, CM, marketing, and both senior and project management.
    b. Identify the stakeholders' most important issues.
    c. Prioritize the stakeholders' problems, opportunities, and requirements.
    d. Group the related issues.
    e. Validate priorities and groupings with stakeholder representatives.
    f. Formulate goals and subgoals.
2. Identify the questions quantifying the goals.
3. Identify the metrics for determining the answers to the questions.
4. Develop mechanisms for data collection and analysis.
5. Collect, validate, and analyze the data for feedback on projects and corrective action.
6. Analyze in a postmortem fashion to assess conformance and make recommendations for future improvements.

7. Provide feedback data to the project group.

The following is an example from Daskalantonakis' paper [*DAS92*]:

**Goal 1:** Analyze the project planning/tracking phase to update the project plan, with respect to project cost/budget, schedule, and effort from the point of view of the software manager.

**Question 1.1:** How can I increase the accuracy of the effort/schedule estimates obtained for my current project?

**Data Items/Metrics Used:**

- Planned project effort (in person hours) from previous (similar) projects and current project
- Actual project effort (in person hours) from previous (similar) projects
- Planned project schedule (in calendar months) from previous (similar) projects and current project
- Actual project schedule (in calendar months) from previous (similar) projects

**Question 1.2:** Is my project progressing according to schedule? If not, what activities are affected, and how can I get the schedule under control?

**Data Items/Metrics Used:**

- Initially planned project schedule (in calendar months) per current project phase
- Actual project schedule (in calendar months) thus far for each project phase
- For each incomplete phase, the projected completion data for that phase

### 5.4.2.2 Data Metrics

The Software Engineering Institute (SEI) recommends collecting the following metrics at a minimum [*CAR92*]:

- Counts of physical Source Lines of Code (SLOCs) (estimated and actual) to measure size, progress, and reuse
- Counts (estimated and actual) of staff hours expended (per month and cumulative) to measure effort, cost, and resource allocations
- Calendar dates (estimated and actual) to measure schedule
- Counts of software errors and defects to measure quality, readiness for delivery, and improvement trends

The following is a list of metrics that could be collected:

**Project**

- Project characteristics (such as type of application, programming languages used)
- Size in SLOC (new, modified, deleted, reused) converted to Thousand Assembly-Equivalent Lines of Code (KAELOC) (estimates and actuals)
- Effort (estimates and actuals)
- Schedule (estimates and actuals)
- Total errors

- Total defects
- Number of staff
- WBS
- Delivered defects
- Delivered defects per size
- Number of open customer problems

The following items are phase-dependent metrics. Estimates for the next three metrics should be re-estimated and reported during the review associated with each of the following phases.

- Size in SLOC (new, modified, deleted, reused) converted to KAELOC (estimates and actuals)
- Effort (estimates and actuals)
- Schedule (estimates and actuals)

**Requirements Phase**

- Total number of requirements (estimated and actual)
- Number of requirements defined
- Number of requirements questions
- Number of requirements changed
- Requirements inspections completed
- Number of requirements errors found in reviews
- Number of requirements defects found in subsequent phases
- Documentation Page Count (DPC) for requirements documents

**Design Phase** [*ISD48*]

- Software subsystem designs completed
- Number of software modules identified
- Number of software modules designed
- Interface designs completed
- Design walkthroughs completed
- Design inspections completed
- Design errors found in reviews
- Design defects found in subsequent phases
- DPC for high-level design documents
- DPC for detailed design documents
- DPC for Interface Control Documents (ICDs)

**Coding Phase** [*ISD48*]

- Modules coded
- Code walkthrough completed
- Inspections completed

- Coding errors found in reviews
- Coding defects found in subsequent phases
- Modules unit tested
- Modules accepted into integration library

**Testing Phase** [*ISD48*]

- Modules successfully integrated
- Test steps planned
- Test steps executed
- Test steps passed
- Problems opened
- Problems closed
- Modules accepted into controlled library
- SLOC of modules accepted into controlled library (cumulative)

**Maintenance Phase**

- Problems opened
- Problems closed

In determining which of the above metrics should be collected, the G/Q/M paradigm should be kept in mind. This means that only the metrics that support the goals of the project and/or division are collected.

### 5.4.2.3 Useful Computed Metrics

Computed metrics are those that are calculated using the primitive metrics that are directly observed. The following are useful computed metrics from the *Motorola Software Metrics Reference Document* [*MMR91*]:

$$\text{In-Process Faults (IPF)} = \frac{\text{IPF caused by delta software development}}{\text{Assembly-equivalent delta source size (KAELOC)}}$$

See Table 5.4.2.3-1 to determine the assembly-equivalent source size.

$$\text{In-Process Defects (IPD)} = \frac{\text{IPD caused by delta software development}}{\text{Assembly-equivalent delta source size (KAELOC)}}$$

$$\text{Total Released Defects (TRD) total} = \frac{\text{Number of released defects}}{\text{Assembly-equivalent total source size (KAELOC)}}$$

$$\text{TRD delta} = \frac{\text{Number of released defects caused by delta software development}}{\text{Assembly-equivalent total source size (KAELOC)}}$$

$$\text{Customer-Found Defects (CFD) total} = \frac{\text{Number of CFD}}{\text{Assembly-equivalent total source size (KAELOC)}}$$

$$\text{CFD delta} = \frac{\text{Number of CFD caused by delta software development}}{\text{Assembly-equivalent total source size (KAELOC)}}$$

**Table 5.4.2.3-1. Table for Determining the Assembly-Equivalent Source Size ]** *[JON 88]*

| Language | Ratio-Source:Executable |
|---|---|
| Assembler | 1:1 |
| Macro-Assembler | 1:1.5 |
| JCL and Execs | 1:1.5 |
| UNIX Shell Script | 1:1.5 |
| IMS DB Languages | 1:1.5 |
| C | 1:2.5 |
| ALGOL | 1:3 |
| CHILL | 1:3 |
| COBOL | 1:3 |
| FORTRAN | 1:3 |
| JOVIAL | 1:3 |
| CMS2 | 1:3 |
| Pascal | 1:3.5 |
| RPG | 1:4 |
| PL1 and MPL | 1:4 |
| MODULA-2 | 1:4 |
| Ada | 1:4.5 |
| PROLOG | 1:5 |
| LISP | 1:5 |
| FORTH | 1:5 |
| BASIC | 1:5 |
| LOGO | 1:5 |
| STRATEGEM | 1:9 |
| APL | 1:10 |
| C++ | 1:11 |
| OBJECTIVE-C | 1:12 |
| SMALLTALK | 1:15 |
| 4th-GLs | 1:16 |
| Query Languages | 1:25 |
| IEF PAD Lines | 1:45 |
| Spreadsheets | 1:50 |

New Open Problems (NOP) = Total new postrelease problems opened during the month

Total Open Problems (TOP) = Total number of postrelease problems that remain open at the end of the month

$$\text{Age of Open Problems (AOP)} = \frac{\text{Total time postrelease problems remaining open at the end of the month have been open}}{\text{Number of open postrelease problems remaining open at the end of the month}}$$

$$\text{Age of Closed Problems (ACP)} = \frac{\text{Total time postrelease problems closed within the month were open}}{\text{Number of open postrelease problems closed within the month}}$$

Cost To Fix Postrelease Problems (CFP) = Dollar cost associated with fixing postrelease problems within the month

$$\text{Total Defect Containment Effectiveness (TDCE)} = \frac{\text{Defects found prerelease}}{\text{Total prerelease and postrelease defects found}}$$

$$\text{Phase Containment Effectiveness (PCE)} = \frac{\text{Errors}_i}{\text{Errors}_i + \text{Defects}_i}$$

Where: Errors$_i$ is the number of errors found in the reviews of phase i and Defects$_i$ is the number of defects introduced in phase i (found so far) that escaped the formal reviews of phase i.

$$\text{Schedule Estimation Accuracy (SEA)} = \frac{\text{Actual project duration}}{\textit{Estimated}\text{ project duration}}$$

$$\text{Effort Estimation Accuracy (EEA)} = \frac{\text{Actual project effort}}{\text{Estimated project effort}}$$

$$\text{Software Productivity (SP) delta} = \frac{\text{Assembly-equivalent delta source size (KAELOC)}}{\text{Software development effort}}$$

$$\text{SP total} = \frac{\text{Assembly-equivalent total source size (KAELOC)}}{\text{Software development effort}}$$

$$\text{Software reliability} = \text{Failure Rate (FR)} = \frac{\text{Number of failures}}{\text{Time}}$$

McCabe's Cyclomatic Complexity [*McC82*]=V(G)= e – n + 2

Where: e is the number of edges or paths on the control flow graph G and n is the number of nodes on the control flow graph G.

This complexity metric is defined for each module. A value of more than 10 is considered too high for a module. A different cutoff value may be selected based on internal standard and results of data analysis [*MPI92*].

Halstead's Difficulty Metric [*STO92*]:

Observed length:

$N_0 = N_1 + N_2$

Where: $N_1$ = Total usage of all operators (verbs) in a module, and

$N_2$ = Total usage of all operands (nouns) in a module

Calculated length:

$N_C = n_1 \log n_1 + n_2 \log n_2$

Where: $n_1$ = Number of unique operators (verbs) in a module, and

$n_2$ = Number of unique operands (nouns) in a module

Normalized as:

$$1 - \frac{N_C - N_0}{N_C} \quad \text{or}$$

$$0 \text{ if } \frac{N_C - N_0}{N_0} > 1$$

### 5.4.2.4 Tracking Metrics

The following are useful ways to graph and report metrics:

- Total effort (estimated vs. actual)
- Effort per development phase (estimated vs. actual)
- Staffing per development phase (estimated vs. actual)
- Schedule (estimated vs. actual)
- Size of data in SLOC (estimates [most likely, minimum, maximum] vs. actual)
- Errors vs. development phase (include estimated, actual, and closed)
- Defects vs. development phase (include estimated, actual, and closed)
- Requirements vs. development phase (include total, changed, and removed)
- Inspections (number completed vs. planned total)
- Modules tested and integrated (current number vs. planned total)
- Unit test steps (number completed vs. planned total)
- System test steps (number completed vs. planned total)

The following charts are the recommended Motorola software metrics charts that use the computed metrics defined in Section 5.4.2.3 [*MMR91*]:

- IPF and IPD as a function of calendar time
- TRD (total) and TRD (delta) as a function of calendar time
- CFD (total) and CFD (delta) as a function of calendar time
- TOP and NOP per month

- AOP and ACP reported monthly
- CFP reported monthly
- TDCE as a function of calendar time
- PCE per development phase
- SEA and EEA as a function of calendar time
- FR vs. total time of testing

### 5.4.2.5 Procedures

Metrics data should be collected throughout the software development lifecycle and tracked using the recommended graphs described above. The SMG can be useful in determining which metrics to collect and how to present the metrics graphically. Once these are determined, it is important to train all of the appropriate people in their use [*STO92*].

Metrics should be evaluated to track project progress, determine trends, and identify problem areas.

Metrics should be reported to the SMG, at a minimum, by software system and total project. All metrics forms and charts applicable to the current development phase should be generated and presented to the SMG. These forms should contain project characteristics, such as type of application and programming languages used, in addition to the other metrics reported.

One way to facilitate the collection of effort (staff hours) is to set up the WBS system so that each development phase has a different WBS number. The standard accounting reports can then be used to report effort. One way to facilitate the collection of errors and defects is to set up a problems database, which can be set up using any number of database tools. The problems can be classified as errors or defects and assigned to a development phase. Microsoft Excel can be used to create graphs if the data are imported from the problems database. Microsoft Project can be used to display schedules.

## 5.4.3 Tailoring to a Small Project

The G/Q/M paradigm should be used to determine which metrics would be useful for a given project. The goals selected may not be a function of project size. Therefore, the metrics collected on a small project may be the same as those collected on a larger project. The reviews for each development phase may be more informal, but it is still useful to collect the metrics data for each phase.

The following metrics should collected on all projects [*CAR92*:

- Counts of physical SLOC (estimated and actual)
- Counts (estimated and actual) of staff-hours expended (per month and cumulative)
- Calendar dates (estimated and actual)
- Counts of software errors and defects

## 5.4.4 Cited References

[*ISD48*] *Software Engineering Handbook, Build 3,* Division 48, Information System Division, Hughes Aircraft Company, March 1992, pp. 11-20–11-21.

[*SEL-81-305*] *Recommended Approach to Software Development, Revision 3 (SEL-81-305)*, NASA Goddard Space Flight Center, June 1992.

[*MPI92*] *Software Metrics for Process Improvement—Participant Guide*, Motorola University, April 1992.

[*BAS90*] Basili, Victor R., *Software Modeling and Measurement: The Goal/Question/Metric Paradigm*, Draft Technical Report, Institute for Advanced Computer Studies, Department of Computer Science, University of Maryland, College Park, 1990.

[*DAS92*] Daskalantonakis, Michael K., *A Handbook for In-Process Use of Metrics by Software Managers*, Motorola, Inc., March 1992.

[*MMR91*] *Motorola Software Metrics Reference Document*, April 1991.

[*JON88*] Jones, Capers, *Table of Programming Languages and Levels*, Enterprise Software Planning workshop notes by Software Productivity Research, Inc., Version 4.0, March 23, 1988.

[*SEL-81-101*] *Manager's Handbook for Software Development, Revision 1 (SEL-81-101)*, NASA Goddard Space Flight Center, November 1990.

[*CAR92*] Carleton, Anita, Robert Park, Worfhart Goethert, William Florac, Elizabeth Bailey, and Shari Pfleeger, *Software Measurement for DoD Systems: Recommendations for Initial Core Measures*, Technical Report, Software Engineering Institute, CMU/SEI-92-TR-19, September 1992.

[*McC82*] McCabe, Thomas J., *Structured Testing: A Software Testing Methodology Using the Cyclomatic Complexity Metric*, NBS Special Publication 500-99, National Bureau of Standards, December 1982.

[*STO92*] Storch, Richard, *An Introduction to Metrics*, Presentation to HSTX, 1992.[STO92]

## 5.4.5 Appendix

### 5.4.5.1 Checklist

| **Procedure for Establishing and Using Metrics** | |
|---|---|
| **Y/N** | **Check** |
| | Have improvement goals for the process, product, or project been identified? |
| | Have the questions quantifying the goals been identified? |
| | Have the metrics for determining the answers to the questions been identified? |
| | Have mechanisms for data collection and analysis been developed? |
| | Have the data been collected, validated, and analyzed for feedback on projects and corrective action? |
| | Has a postmortem analysis been conducted to assess conformance and make recommendations for future improvements? |
| | Have feedback data been provided to the project group? |

*Section 5.5*

# Scheduling and Tracking

## Contents

### 5.5.1 Introduction

Scheduling is one of the earliest activities in a project. On most HSTX contracts, schedules are originally produced during the proposal phase and refined after contract award. A project's schedule is evolvable to account for factors such as shifts in priority and scope, changes in external dependencies, and changes in project resources. This section discusses approaches to scheduling.

### 5.5.2 What To Schedule

Schedules show activities, milestones, and dependencies. Schedules should show all major activities, as well as milestones. A top-level schedule shows two kinds of milestones:

- **Major Milestones**—These are contractual milestones and other activities with customer visibility, such as major reviews, product deliveries, launch dates, and due dates for Government-furnished property.
- **Milestones Involving Dependencies**—These are milestones that connect to someone else's schedule, or milestones that are fed by someone else's schedule. For example, if you need to produce an ICD in order for another organization to start work on software that interfaces to your system, your schedule should show when you expect to complete the plan (because that drives another organization's work) and when you need the software to be available (because that depends on another organization's work). It may be useful to develop a dependency chart before schedules are attached to it. Showing these milestones allows a project-wide schedule review to coordinate efforts.

An intermediate schedule shows lower level milestones, with a moving window of finer granularity. Within the window, activities are scheduled in detail, perhaps in 1-week increments. Beyond the window, activities might be scheduled by month, reflecting the higher uncertainty in the future. A 2-year schedule might have a 3-month moving window. Note that requirements for financial planning (which can vary by project) might affect scheduling decisions. For example, if the earned value system on a particular project requires definition of planning packages in a 4-month moving window, it is simpler to adopt the same size window for detailed planning. On task order contracts for NASA, HSTX submits Contractor Task Reports (CTRs) that plan the work for typically 6 or 12 months. Monthly reports are required, detailing the schedule status for the past month and the next month.

On detailed or individual schedules, it is best to show all required activities. For example, if the schedule includes production of a document, the schedule might include milestones for development of "ancillary paragraphs" (applicable documents, glossary, table of contents, etc.), review by QA, delivery to the document producer, and review of the final product before copies are made. There are two reasons for such comprehensive schedules. First, it can avoid forgotten steps. If the schedule shows that the developer is to produce a glossary before QA can review a document, the glossary will not be forgotten. Second, it avoids slips in activities scheduled later. Even a tight schedule should show some time for rework after QA reviews a document; otherwise, the almost inevitable rework will cause the next activity to start late.

### 5.5.3 Scheduling Principles

Several principles apply to any kind of schedule construction:

- **Schedule all work.** This sounds simple, but it is often ignored. For example, "design" is not finished without a document, or sometimes a review. A review is not finished without completion of the action items. If this effort is not scheduled, the next activity will probably start late, and there will be effort unaccounted for by the schedule. This can lead to the "90% done" condition, in which most of the visible work has been completed but the product is somehow not ready to be delivered. (For example, the schedule can show that design is "done," but a few unscheduled activities must occur before the design document can be submitted: resolve design review action items; get customer concurrence if action items resulted from a formal review; change the design if necessary in accordance with action item resolution; revise the requirements traceability matrix; complete the document, including all sections, a glossary, notes, etc.; review and correct the document; and so on. Ask "What will signify that we're done with this activity?" and then put the result on the schedule.

- **Use concrete milestones.** Software schedules often consist of milestones for design, code, and test. These milestones are deceptively vague, however. For example, when is "code" complete? It could be when the programmer says it is complete, when the code compiles without error, when a code walkthrough has been done, when action items from the code walkthrough have been resolved, when unit testing is complete, when the code is submitted to the integration library, or when the code is successfully installed in the integration library. "Design" and "test" are similarly vague. Some sample concrete milestones are shown below.

| Design Milestones | Code Milestones | Test Milestones |
|---|---|---|
| Design walkthrough completed | Code walkthrough | Thread executed |
| Walkthrough action items done | Walkthrough action items done | Thread problems fixed |
| Detailed interface design documented | Unit test completed and filed | Procedure revised |
| Design specification paragraph written | Unit accepted into library | Software frozen |

In selecting milestones for your activities, a good rule to follow is that an activity is complete when the result is available to the next activity. For example, a unit's code and test work is complete when the unit is in the integration library and available for use in integration. (After "normal" code and unit test work, more (unscheduled) work could be required before the unit is available for the next step. The unit test may have worked, but the unit may have been submitted with incorrect CM library control commands, causing it to be rejected from the library (and thus be unavailable for the next activity). The unit may have made nonstandard use of some data files, causing it to be rejected. The unit may have a name or some external variables that duplicate something in the library, causing rejection. Multiple versions may already be in the library, causing confusion as to the "latest" version. In other words, the unit may fall into a void between "unit test" ("I'm done with it") and "integration" ("We can't use it"), in which no one seems responsible.

- **Avoid false precision.** Creating a schedule does not create information; it creates a reflection of what you already know. You do not know that a 4-week task will complete on a Wednesday, so you should not show that precision on a schedule. Too many things can go wrong that cause a 1-day slip. For example, a key person may be sick on Tuesday; a snowstorm in Colorado, an earthquake in California, or a hurricane in Virginia could close the facility for a day; or a power, computer, or reproduction failure could cause a delay. (Of course, these are also reasons to avoid pushing work out close to a deadline.) The following is a recommendation for schedule granularity:

| Duration of Effort | Schedule Granularity |
|---|---|
| 2 weeks | Day |
| 3–12 weeks | Week |
| Over 12 weeks | Month |

There are some exceptions to this guideline. If a proposal is due at 10:00 a.m. EST on a date 6 months from now, a schedule can show specific days for delivery, shipment, printing, final galley proofs, etc. (because the deadline is fixed and we know that proposals are never done early). If a contract gives specific dates for availability of Government-furnished property, those dates can be shown on a schedule (because they are contractual dates). Do not include greater precision than your information warrants.

- **Reschedule only after redirection.** "Reschedule" means to throw away the old schedule and produce a new one. Its effect is to wipe away any schedule slips and produce a bright, shiny, unsullied schedule. It is not proper to reschedule unless there has been a redirection; that is, unless there has been a change in the contract, or customer direction, that changes the baseline activities, or direction from the program manager or customer. (A redirection may involve only part of a program; if so, that is the only part to be rescheduled.) Showing a slip does not constitute a "reschedule." It *is* proper to *replan* a schedule to rearrange activities, but the revised schedule should show the changes from the baseline schedule, with some activities slipped and some (with luck) advanced. If the schedule gets messy, it accurately reflects the fact that the planning or development process has been messy. If the customer concurs in rescheduling for "cleanup" purposes, a new schedule can be produced.

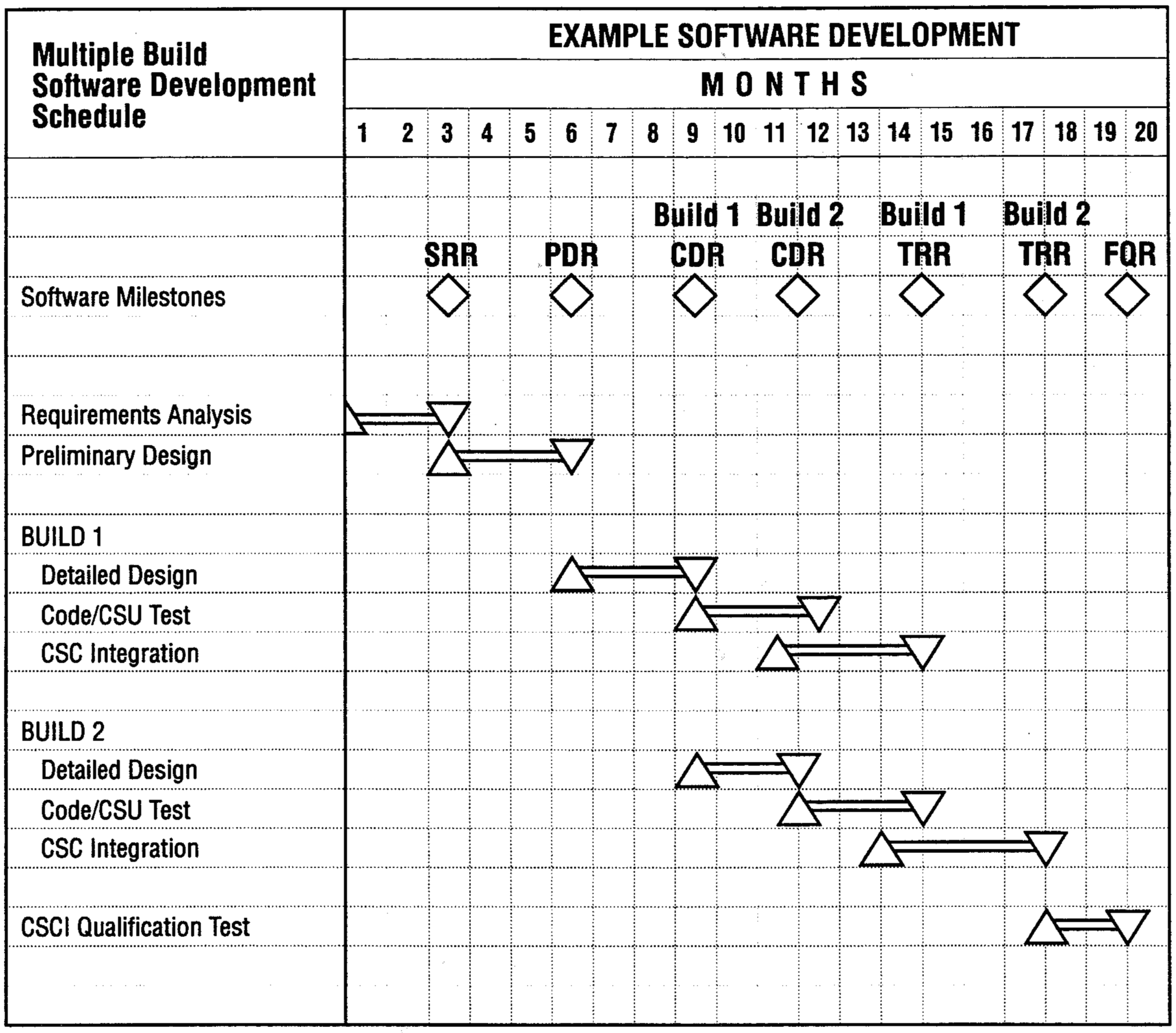
## 5.5.4 Scheduling Multiple Builds

For large development projects, it is often advisable to schedule multiple builds of the software. The incremental build approach enables a large software system and software development team to be divided into smaller, more manageable components. Developers on the second (and succeeding) builds will benefit from the experiences of the teams that develop the preceding builds. Additionally, the software developed during the first build, which will be the key parts of the system, will be developed and tested sooner, longer, and more thoroughly. Refer to Figure 5.5.4-1 for an example of a multiple build software development schedule.

The composition of each build should be completely defined during the software requirements analysis and the preliminary design phases. With incremental builds, each build will have its own set of software development phases. For example, each build may contain a detailed design phase through subsystem integration phase, or each build may contain only a subsystem integration phase.

The following are some guidelines for determining the scheduling of multiple software builds:

- **Identifying Threads**—The build definition essential to program integration and test starts with thread definition. A thread is a sequence of software that accepts a system input and produces a system output. That is, it is a set of software that can be executed with very few or no stubs or drivers providing essential input or output. (Stubs may be necessary to hold the place of units assigned to later threads.) The intent is to aggregate units into a set that can provide a complete, although small, piece of the system's processing. (One way to decompose the software into threads is to use a PERT-type chart or data flow diagram that



SWDG012

**Figure 5.5.4-1. Multiple Build Software Development Schedule**

illustrates the processing of all system inputs and outputs. Threads can be graphically identified by circling or color-coding sequences through the diagram.)

- **Identifying Builds**—Threads should be aggregated into builds. A build is a major collection of software whose integration test will be noted on a milestone schedule. The composition of a build depends on the needs of the program. However, the following guidelines apply:
  - Well-understood core requirements and functions should be implemented in early builds.
  - Higher risk code should be implemented in early builds.
  - Builds should represent complete logical divisions of the software architecture, with simple interfaces between builds.
  - Functions whose implementation is not completely understood or may depend on the implementation of other software should be implemented in later builds.

  - A build should contain software from a single major subsystem when possible.
  - The sizes and expected difficulties of builds should not vary greatly.
  - Software dissimilar in nature or allocated to different CPUs may require a separate build. Builds including software for start-up, initialization, essential operator interface, recording data useful for integration, etc., will occur first. These builds should be kept small to enable the earliest possible progress (which is a confidence and morale builder) and to provide a platform for other builds.
  - Include database integration and testing as an integral part of the total software integration.
  - The build plan should consider any formal delivery schedules or internal schedules. There may be a reason to integrate specific software at specific times in the process to support planned later activities or to coincide with the availability of interfacing hardware.
  - Do not defer "ancillary" functions such as error-checking, recovery, and rollback to separate or late builds. Despite the normal pressure to get "the real stuff" working, these functions can speed up integration and lessen frustration if they are developed early.

- **Milestones**—One or more threads support a functional capability as identified in the SRS. A milestone may consist of one or more capabilities that can be demonstrated by executing part of the software integration and test procedures. A milestone can also be associated with the release of a master integration and test tape (or other media) containing all the software that supports a particular build. One major milestone could be an early operational capability. Depending on hardware, software, and human resource availability, parallel integration and testing may be scheduled to encourage and promote integration and testing efficiency.
- **Intermediate Builds**—Intermediate builds that consist of informal builds may be identified to allow smaller increments of software to be integrated and tested. Smaller increments are encouraged because they usually promote faster problem identification.

## 5.5.5 Scheduling Mechanics

The mechanics of representing a schedule are generally not important. If the customer requires a specific format, the mechanics may become important. A few simple rules apply:

- Use automated programs for neatness and ease of updating. (Microsoft Project is an example of such a program.) However, if many schedules need weekly updates, it may be faster to update them manually.
- Use standard symbols for milestones, progress, and schedule changes. Any nonstandard symbol should be defined on each schedule page. Activities with changes should be highlighted (asterisks are easy) for the regular schedule review.
- Enter major milestones into the chart first. For example, showing a Preliminary Design Review (PDR) on the chart can indicate the amount of flexibility of the schedule being reviewed. After the major milestones are listed, the intermediate milestones can be inserted.
- Before contract award, it is best to develop schedules without actual calendar dates because the entire schedule may move. Use dates After Receipt of Order (ARO) or After Contract Award (ACA).

- **Use the schedule.** A schedule should be a tool for managing, not just for reporting. Refer to it when checking progress. Microsoft Project can be set up based on the WBS and can therefore be used to generate Basis of Estimates (BOEs) and to monitor status.

Refer to Figure 5.5.5-1 for an example of an automated software development schedule. This is sometimes called a Gantt chart.

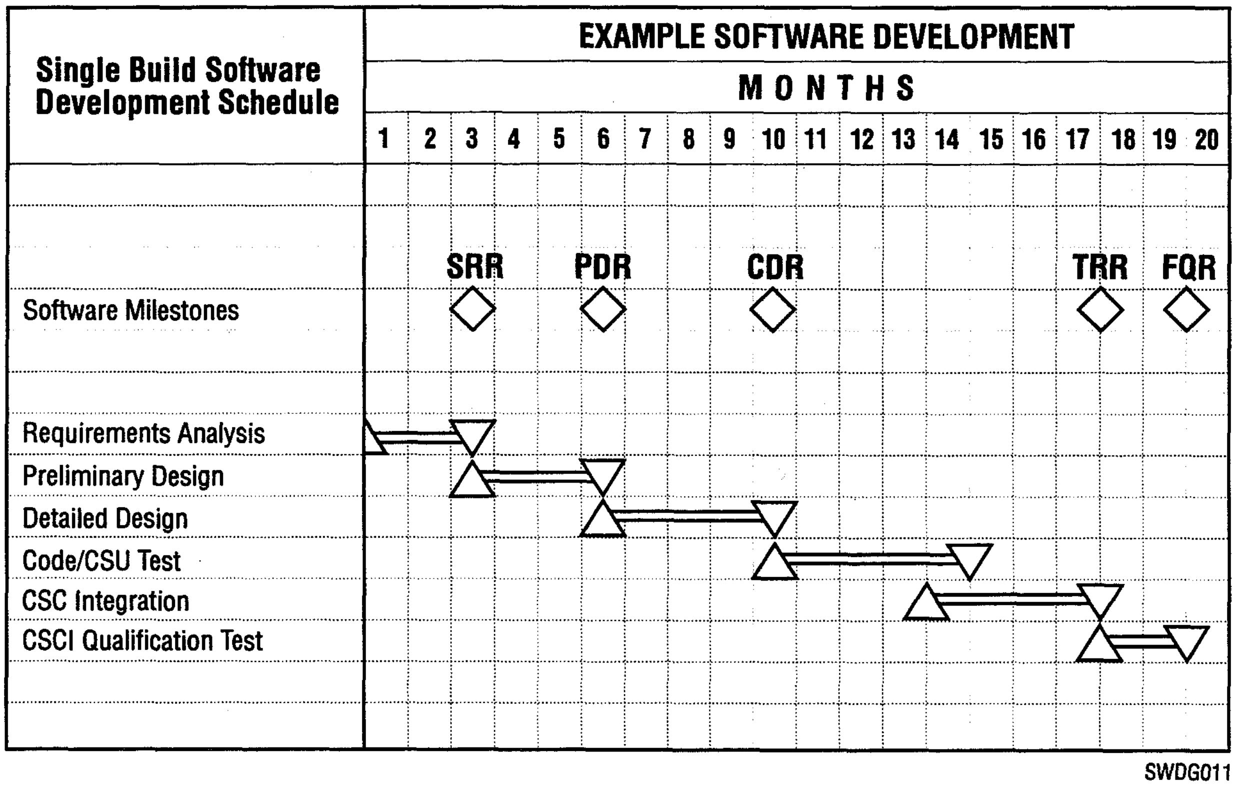


**Figure 5.5.5-1. Software Development Schedule (Single Build)**

## 5.5.6 Activity Networks

Activity networks, or PERT charts, are automated charts that define activities and their relationships in a different form than a Gantt chart. Each activity is assigned a duration (sometimes a minimum and maximum duration) and a relationship with predecessor and successor activities. (Examples: Activity A must complete before Activity B starts. Or Activity B can start after Activity A starts, but before A completes.) Using these dependencies, the activity network program then predicts a completion date, identifies a critical path (the "long pole" thread that determines the completion date), and often calculates a "float" for each activity. The float for an activity is the time (usually the number of days) that the activity can slip without affecting the overall schedule. PERT charts can help in determining possible parallelism of activities and resource levels. PERT charts are very useful in planning (and replanning), but three cautions apply:

- PERT charts require automated tools. The burden of manual calculation is too great.
- PERT charts are most useful when they contain a manageable number of activities. The number varies from tool to tool, but is usually fewer than 200. With too many activities,

the burden of maintaining the network is too high. Review is too time-consuming, physical update time (entering data, printing graphs, etc.) is excessive, and utility decreases. To reduce the number of network activities, consolidate consecutive activities without dependencies into a single activity and use a normal schedule (or a subnetwork) to detail the component parts.

- Beware of relying on probabilities. For example, assume two parallel activities converge into a third activity, and that the duration of each of the first two activities is 75% certain. The probability that one activity will be late is 25%, but the probability that one or the other or both will be late (thus delaying the start of the third activity) is 43.75%. (If the individual probabilities of slipping are 30%, then it is actually likely that one or the other or both will be late.)

## 5.5.7 Schedule Tracking

Constructing a schedule can assist in forming plans, devising an organization, and determining the resources needed. However, to be useful after the initial planning stage, schedules must be tracked and corrective action taken when progress deviates from the plan. There are several steps in schedule tracking.

Schedule tracking usually starts with weekly progress reports from individual developers. These reports should concentrate on progress made, plans for the next week, and current or expected problems. Long reports on "what I did this week" should be discouraged as time-wasters. The items of concern are concrete progress toward milestones, expected progress in the next period, and problems or issues on which the developer needs help from supervision.

These weekly reports are combined into a team, group, or task weekly report. The schedules will be updated by the team leaders, section manager, planning staff, or software manager, depending on the project.

The team reports are aggregated into a software project report. The official program schedules are updated based on this report, which is usually produced monthly. PERT charts are usually updated monthly, although they may be updated more frequently during crises.

Many software activities will not have concrete milestones on a weekly basis. It is important for the software manager or team leaders to have personal contact with the developers, rather than relying completely on written reports. This contact can produce a "feeling" about what progress is being made, as well as a calibration of individual reports. (Some people report no progress until they are certain that the job is 100% complete; others report great progress based on what they expect to accomplish in the following day or two. The software manager should understand the reports he or she receives.) Metrics that are collected on a regular basis could be very useful for determining status in this case.

When progress falls behind the plan, corrective action should be taken. There are many types of possible corrective action, depending on project circumstances. The following are some examples, in addition to "work harder" and "add more people":

- Reallocate effort from areas ahead of schedule to areas behind schedule.
- Reallocate effort from noncritical path items to critical path items.
- Define interim capabilities to reduce risk, provide checkpoints, and establish goals.
- Analyze sources of delay and establish alternate procedures (i.e., establish a *cmi* team).
- Borrow experts or consultants in the areas causing difficulty.

- Re-examine activities to ensure that each activity is essential. (Example: Are people manually creating documentation that could be generated automatically, or almost automatically?)
- Examine whether any activities can occur in parallel rather than in series. (Example: Can developers update requirements traces while their software is being installed in the library, rather than having to do it before submitting their software?)
- Spread people over staggered shifts to reduce competition for resources.
- Establish teams dedicated to and responsible for problem areas. Staff the teams with all skills needed for the job (including dedicated QA and CM staff, if necessary.)
- Re-examine skills and assign people to the areas in which they perform best. (Examples: It may help to assign individuals full time to unit testing, integration, document production, problem tracking, or coding, depending on their individual skills.)

## 5.5.8 Using Graphical Profiles for Schedule Tracking

Graphical profiles provide a useful technique for tracking a project's progress. For example, when a project manager sees a graph of "Test Cases Completed vs. Time," the manager is able to assess the relative progress of testing that week as compared to prior weeks or to assess the time required to complete the testing (see Figure 5.5.8-1).

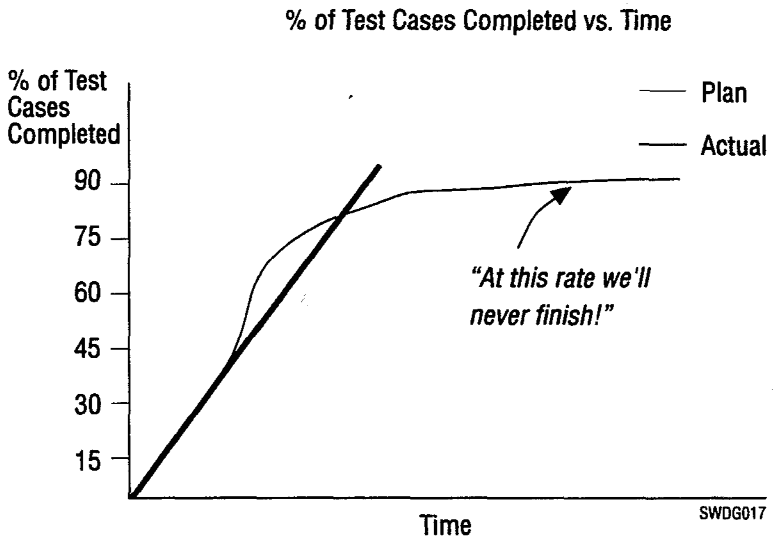


**Figure 5.5.8-1. Example Graphical Profile**

Graphical profiles expose the "90% done" problem. When you are regularly plotting progress on a graph, as in the above figure, management is able to visually see the rate of progress or lack of progress on a project. Showing planned vs. actual measurements sheds light on where you need to be at any given point in order to meet the schedule. Graphical profiles work only when you are collecting the underlying metrics data for the profile. The discussion of metrics in Section 5.4 provides details on what metrics should be collected and how to successfully collect them. Microsoft Excel can be used to store the metrics and to display the graphical profiles.

### 5.5.9 Tailoring to a Small Project

Even on the smallest project it is important to schedule and track activities and milestones. If various reviews are not required by the contract, internal reviews should be scheduled. If documents are not required by the contract, completion of informal documentation should be scheduled. Graphical profiles of planned vs. actual metrics can be useful, even on the smallest project. Microsoft Project is an example of a planning tool that is useful on all projects.

### 5.5.10 Suggested Reference Material

*Software Engineering Handbook, Build 3*, Division 48, Information System Division, Hughes Aircraft Company, March 1992, p. 3-13.

## *Section 5.6*

# Risk Management

## Contents

Software development managers need to address software risks before they become project disasters. Software risk management provides a set of tools to *manage* risks from initial discovery to effective mitigation. This section discusses the fundamentals of software risk management, including risk identification, risk analysis, risk mitigation, and risk exposure calculations.

## 5.6.1 Definitions

*Software Risk Management* is defined by Barry Boehm [BOE89] as "an emerging discipline whose objectives are to identify, address, and eliminate software risk items before they become either threats to successful software operation or major sources of software rework." The key elements of software risk management are the assessment and control of risks. Assessment refers to the discovery, characterization, and prioritization of risks. Control refers to the elimination, avoidance, and reduction of risks or "What can we do about them?".

*Risk Exposure (RE)* or *risk impact* is defined by the following formula:

**RE = Prob(UO) * Loss(UO)**

where *Prob(UO)* is the probability of an Unsatisfactory Outcome (UO) and *Loss(UO)* is the loss (in dollars or a relative numerical scale) to the parties affected if the outcome is unsatisfactory [BOE89].

*Risk Reduction Leverage (RRL),* which provides a comparison metric that measures the relative cost-benefit of implementing possible risk reduction activities, is represented by the following formula defined by Boehm [BOE89]:

**RRL = { RE(before) - RE(after) } / (cost of risk reduction measure)**

Where *RE(before)* is the risk exposure before the risk reduction effort, or what you started with; *RE(after)* is the risk exposure after the implementation of the risk reduction. A higher value of RRL is considered better than a lower value.

## 5.6.2 Introduction

The wide use of computers and the increasing complexity of their software systems increases the probability of a disaster and makes risk management critical.

From the development cost perspective, a large, complex project has many opportunities for problems to occur. However, problems can also occur on medium, small, and very small software development projects. The cumulative losses associated with the medium and smaller sized projects may account for the majority of software disasters that occur. Often, on smaller projects, less attention is paid to problems that might occur in the future (both during the development and during the operations and maintenance phases). Most of us work on medium- to small-sized projects that may have a wide array of potential software disasters waiting to erupt. Boehm surveyed a number of TRW projects during the 1980s and compiled a list of the top-ten software risk items, shown in Table 5.6.2-1.

Table 5.6.2-1. A Prioritized Top-ten List of Software Risk Items

| | |
|---|---|
| People | 1. Personnel shortfalls |
| Resources | 2. Unrealistic schedules and budgets |
| Requirements | 3. Developing the wrong software functions |
| | 4. Developing the wrong user interface |
| | 5. Gold plating |
| | 6. Continuing stream of requirement changes |
| Externals | 7. Shortfalls in externally furnished components |
| | 8. Shortfalls in externally performed tasks |
| Technology | 9. Real-time performance shortfalls |
| | 10. Straining computer-science capabilities |

Does your project have one or more of these potential disasters waiting to happen? How can software risk management techniques mitigate these common risks?

## 5.6.3 Software Risk Management Fundamentals

Many software project managers are already performing risk management. Good managers are always thinking ahead, which is the essence of risk management. The science of software risk management formalizes this process. The remainder of this section discusses the process of software risk management, including software risk management fundamentals; risk identification, analysis, mitigation, and management issues; and recent research and conclusions.

### 5.6.3.1 Types of Risk

In the software industry, risks have been subdivided into a set of categories. The U. S. Air Force leads the way in the development of risk management formal practice. They typically divide risks into the following types:

- **Cost**—The uncertainty in the ability to complete a program within its budget
- **Schedule**—The uncertainty in the ability to complete a program within the allocated schedule
- **Technical**—The uncertainty in the ability to achieve the required technology
- **Operational**—The uncertainty in the ability of the delivered system to meet its operational (mission) requirements
- **Support**—The uncertainty in the ability of the support organization to maintain, change, and/or enhance the deployed system

These risk types are useful in identifying and analyzing the impact of risks. Many software risks result in both cost and schedule risks.

#### 5.6.3.1.1 Risk Reduction Leverage Example

The following is an example of using RRL to compare two options:

**Case Study:** The marketing department at your company promised the customer all kinds of extra "bells" and "whistles" for your software system. You, as software manager, are tracking the progress and costs of this project closely. You discover that you are headed for a potential overrun of $100,000, if you don't do something now. You have two options: spend $20,000 scrubbing requirements to remove the extra features from the software or spend $20,000 trying to implement the extra features. You estimate the *Prob*(*UO*) to be 0.8 and the *Loss*(*UO*) to be $100K, yielding:

**Case 1: Scrub Requirements**

*RE*(*before*) = (0.8)*($100K) = $80K
*RE*(*after*) = (0.4)*($40) = $16K
*RRL*(*scrubbing*) = ($80K–$16K)/$20K = 3.2

In this case, the *RE*(*after*) assumes that you have reduced the probability of an overrun to 0.4 instead of 0.8 and you have reduced the cost of that overrun to $40K.

**Case 2: Work Extra Hard**

*RE*(*before*) = (0.8)*($100K) = $80K
*RE*(*after*) = (0.7)*($100) = $70K
*RRL*(*working hard*) =($80K–$70K)/ $20K = 0.5

In this case, the *RE*(*after*) assumes that you have reduced the probability of an overrun to 0.7 instead of 0.8, but you have not reduced the cost of the overrun.

In this example, the higher value of 3.2 vs. 0.5 indicates to the software manager that scrubbing the requirements would be the less risky alternative.

### 5.6.3.2 Software Risk Management Taxonomy

Boehm has developed a taxonomy for software risk management. Risk management has two primary components: *risk assessment* and *risk mitigation* (also known as risk control). Risk assessment consists of *risk identification, risk analysis,* and *risk prioritization*. Risk mitigation or risk control is composed of *risk management planning, risk resolution,* and *risk monitoring*.

The following sections discuss in further detail the elements of Boehm's software risk management taxonomy. Risk identification and risk analysis (including risk prioritization) are key elements of the process described in the following sections. Section 5.6.2.2.3, Risk Mitigation, will discuss risk management planning, risk resolution, and risk monitoring. Figure 5.6.3.2-1 shows a summary of the risk management taxonomy.

#### 5.6.3.2.1 Risk Identification

The first step in risk assessment is risk identification. The goal of risk identification is to identify those elements of a program that contain risk. The methods commonly used in risk identification include checklists, decision driver analysis, assumption analysis, and decomposition.

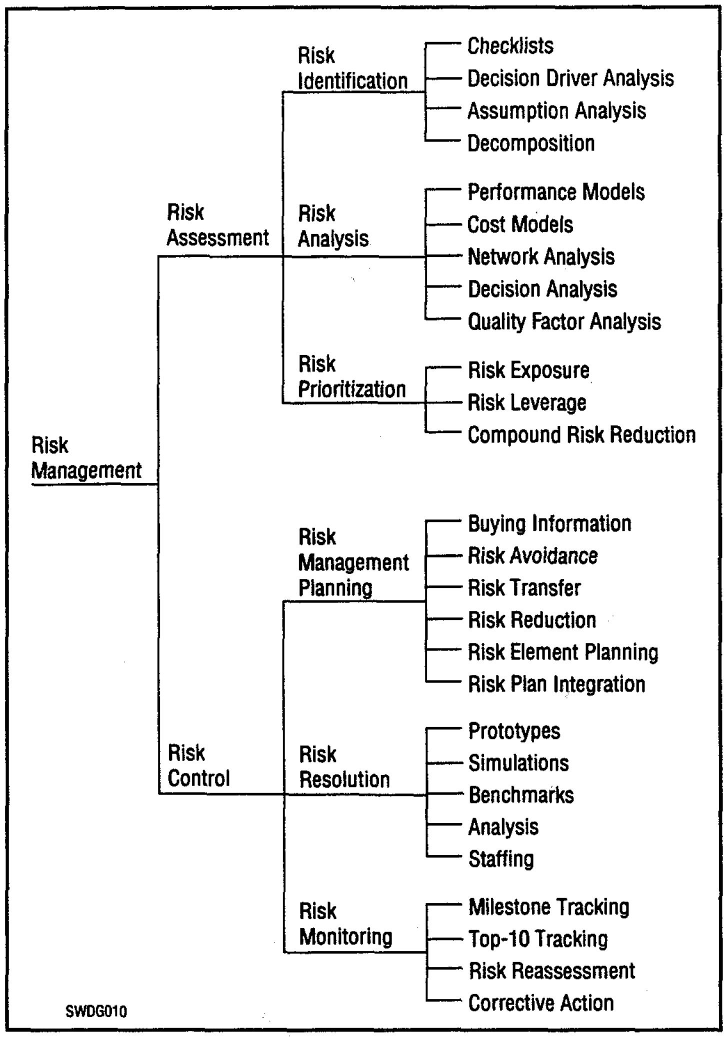


**Figure 5.6.3.2-1. Risk Management Steps**

**Checklists**—Checklists are useful in starting the risk identification process. Organizations should develop their own list of top-ten risks that are common to their business environment. One risk common to today's projects might be "shrinking budgets." Table 5.6.3.2-1 lists the top-ten risks found by Boehm and some possible risk management techniques that can be applied to these risks.

**Decision Driver Analysis**—Risk drivers are those variables that cause cost, schedule, performance, or support risk to fluctuate significantly. Performance drivers can be subdivided into requirements, constraints, technology, and development approach. The technology variables include language, hardware, tools, data rights, and experience.

**Assumption Analysis**—Major software risks are hidden behind assumptions. It is important to look back at the history of the development of the initial scheduling and budgeting performed. The project parameters should be checked against history (i.e., hardware delivery schedule, requirements stability, and external milestone shifts).

**Table 5.6.3.2-1. Top-ten List of Software Risk Items With Risk Management Techniques**

| Risk Item | Risk Management Techniques |
|---|---|
| 1. Personnel shortfalls | Staffing with top talent, job matching, team building, morale building, cross-training, prescheduling key people |
| 2. Unrealistic schedules and budgets | Detailed, multisource cost and schedule estimation; design to cost; incremental development; software reuse; requirements scrubbing |
| 3. Developing the wrong software functions | Organization analysis; mission analysis; operations-concept formulation; user surveys; prototyping; early user's manuals; requirements traceability (if it's not required, don't do it; if it is required, don't forget it) |
| 4. Developing the wrong user interface | Task analysis; prototyping; scenarios; user characterization (functionality, style, workload) |
| 5. Gold plating | Requirements scrubbing; prototyping; cost-benefit analysis; design to cost; requirements traceability |
| 6. Continuing stream of requirement changes | High change threshold; information hiding; incremental development (defer changes to later increments) |
| 7. Shortfalls in externally furnished components | Benchmarking; inspections; reference checking; compatibility analysis |
| 8. Shortfalls in externally performed tasks | Reference checking; preaward audits; award-fee contracts; competitive design or prototyping; team building |
| 9. Real-time performance shortfalls | Simulation; benchmarking; modeling; prototyping; instrumentation; tuning |
| 10. Straining computer-science capabilities | Technical analysis; cost-benefit analysis; prototyping; reference checking |

**Decomposition**—Software risks typically hide in large structures such as subcontracts, user interfaces, database management systems, and utility libraries. Look for oversimplified problems, complex interactions, major sources of change, and unprepared teams of people. These large structures may cover up a lack of knowledge or understanding about that segment of the project.

Decomposition is the process of breaking down these large segments into smaller, more manageable pieces. Task "fan-in" and "fan-out" should be examined. Those tasks that have many tasks fanning into them will be late if any of the parent tasks is late. Those tasks with many tasks fanning out of them will cause all of the child tasks to be late if the parent task is late itself.

### 5.6.3.2.2 Risk Analysis

The goal of risk analysis is to understand a project's risks by gathering data on *Loss*(*UO*) or costs and *Prob*(*UO*) or probability. The software manager is able to estimate REs using these parameters as discussed earlier. The methods or tools used in risk analysis are decision analysis, network analysis, cost risk analysis, and risk prioritization.

**Decision Analysis**—Decision analysis is used to calculate impacts in complex situations. Typically, a decision tree is used to illustrate the options or choices possible when a problem occurs. Each has several possible outcomes with estimated probabilities and costs. The manager computes the expected values and the maximum cost of each option. One option is chosen to represent the impact of the risk item.

**Network Analysis**—Network analysis is primarily a tool for schedule risk analysis. A PERT chart is used to show the dependencies of interrelated tasks. Fan outs, fan ins, and critical paths are easily analyzed on the PERT chart. The existence of multiple parallel critical paths is often a problem on projects. These should be replanned to reduce the number of critical paths.

**Cost Risk Analysis**— Software cost estimation models such as COCOMO are excellent tools for cost risk analysis. The tools are best at determining initial estimates before the project has started. In-process or cost-to-complete estimates are more difficult to develop because of the many changes that can occur during a project, such as personnel changes, requirements changes, and technical disasters. An organization's ability to correctly model its costs is very much based on the existence and accuracy of organization-specific cost accounting data. Those companies that keep track of how each software development dollar was spent on a project are better able to estimate costs on future projects. Setting up a charge structure that maps to the WBS is a way of getting cost data directly from accounting reports.

**Risk Prioritization**—The goal of risk prioritization is to develop an ordered list of risk elements. RE diagrams are important tools in risk prioritization. Risk prioritization should actually be going on during the risk identification process. You should not spend much analysis time on low-priority risk items. When RE calculations become difficult, the betting-odds technique is sometimes useful. To use this method, simply think of how much money you would bet on a risk item occurring. If you are willing to bet a lot of money, then the probability of that risk item occurring is high.

### 5.6.3.2.3 Risk Mitigation

Risk mitigation or risk control addresses how the software manager develops an "attack plan" for the risks identified during the risk assessment process. Risk mitigation consists of three key areas:

- **Risk Management Planning**—Develop a proactive approach to handling the risk.
- **Risk Resolution**—Reduce the RE of undesirable outcomes for assumed risks.
- **Risk Monitoring**—Understand the current risk situation.

Table 5.6.3.2.3-1 summarizes the many techniques available for managing software risks.

## 5.6.4 Software Risk Management Issues

Because of the scope of software risk management, a number of issues result from its application to projects. A few of these issues are listed below.

Historically, too much emphasis has been placed on the quantitative aspects of risk management. RE calculations are highly dependent on the probabilities used, which, of course, are really subjective "estimates" of the chance that a particular unsatisfactory outcome will occur. Too often there is not very much basis in reality of the "numbers" used.

A related issue deals with the calculation of RRL. It is difficult to calculate project RRLs because of the infinite combinations of risks. For example, a seemingly simple UO of "hardware late" could be decomposed into many possibilities such as all hardware late, all hardware very late, all hardware a little late, some hardware late, some hardware very late, or

Table 5.6-3. Risk Control Methods

| Risk Control Area | Methods | |
|---|---|---|
| Risk Management Planning | • Information buying<br>• Risk avoidance<br>• Risk transfer | • Risk reduction<br>• Risk element planning |
| Risk Resolution | • Prototypes<br>• Simulations<br>• Benchmarks<br>• Staffing<br>• Analysis<br>• Staffing and rescheduling of people<br>• Team building<br>• Cost and schedule estimation<br>• Design to cost<br>• Design to schedule<br>• Design techniques | • Incremental development<br>• Requirements scrubbing<br>• Prototyping<br>• Mission analysis<br>• Reference checking<br>• Preaward audits<br>• Performance<br>• Fault tree analysis<br>• Failure modes<br>• Use of spiral model |
| Risk Monitoring | • Milestone tracking<br>• Top-ten tracking<br>• Risk reassessment | • Corrective action<br>• Risk management teams |

some hardware a little late. The "some hardware late" UO divides into many combinations of different hardware that could be late (e.g., memory, CPU, disks, printers, terminals).

Software risk management is not performed in a "cookbook" fashion. Management is the key word in this field. The management of risk requires a good manager making good decisions. No software process will make up for poor decisions. Executive levels of management need to empower their employees to make decisions. However, this empowerment should be limited by the degree of comfort that the executives have with the decision-making ability of their employees.

Traditionally, too much emphasis has been placed on the risk identification segment of risk management. The emphasis would probably be better placed on risk mitigation instead. The key challenges in software risk management today are in the risk mitigation area. The budget constraints of today's economy require extremely creative solutions to the risks and problems encountered in the highly complex systems currently being built.

## 5.6.5 Summary

Software risk management focuses the software team's energy on the important issues. It is a controlled process that empowers the project team to make decisions at the appropriate level using facts and data, not intuition. Most importantly, software risk management increases the probability of a successful program. Remember the following risk management principles [BOE89]:

- If you do not actively attack the risks, they will actively attack you.
- Never make promises you cannot keep, no matter what the pressure.
- Raise and document new issues as they happen.
- If you do not ask for risk information, you are asking for trouble.

The techniques of software risk management will help the software manager (and others on the software team) to make better decisions that reduce or eliminate the unsatisfactory outcomes associated with most projects.

### 5.6.6 Tailoring to a Small Project

Regardless of size, the risk management function needs to be performed. The software manager should be familiar with the concepts of risk identification, risk analysis, and risk mitigation. The manager's project planning should include risk management planning, risk resolution techniques, and risk monitoring. Every manager (or developer responsible for a task) should review the top-ten list of software risk items. If any of these items seems a likely risk, the risk management techniques should be reviewed and the appropriate measures implemented. RRLs can be estimated to determine whether risk reduction measures would be cost effective. With experience, items can be added to the top-ten list of software risks. Then, checklists can be used to determine whether these risks might be present on a project.

### 5.6.7 Suggested Reference Material

Littlewood, B., and L. Strigini, "The Risks of Software," *Scientific American*, November 1992.

[*NEW86*] Newmann, P., "On Hierarchical Design of Computer Systems for Critical Applications," *IEEE Transactions on Software Engineering, SE-12, 9*, September 1986, pp. 905–920.

[*GLA92*] Glass, R., *Building Quality Software*, Prentice Hall, Englewood Cliffs, New Jersey, 1992.

[*HAC91*] "Risk Management Course Materials," Hughes Aircraft Company, CA, 1991.

[*CHAR*] Charette, "Software Risk Management."

[*FIR91*] Firth, Robert, "The Role of Risk," October 1991.

Tate, Paul, "Risk! The Third Factor," *Datamation*, Cathers Publishing Company, April 15, 1988, pp. 58–64.

### 5.6.7 Cited References

[*BOE89*] Boehm, Barry, *Software Risk Management*, IEEE Computer Society Press, Washington, DC, 1989.

*Section 5.7*

# Do's for Project Success

## Contents

- Use a small senior staff for the early lifecycle phases.
- Develop and adhere to an SDP.
- Write down the SRS.
- Define specific intermediate and end products.
- Examine alternative approaches.
- Perform risk analysis.
- Conduct formal and informal reviews with customers and users.
- Use a defined testing process.
- Use a central repository.
- Keep a detailed list of TBD items.
- Update system size, required effort, cost, and schedule estimates.
- Allocate sufficient time for testing and integration.
- Experiment.

[*SEL-81-205*]

## 5.7.1 Do's for Project Success—Details

**Use a small senior staff for the early lifecycle phases.** Begin a project (i.e., planning and requirements phase) with a small group of experienced professionals, especially while preparing the SDP, setting priorities, organizing the work, and establishing reasonable schedules. With a large team there is a tendency to try to keep people busy by beginning design or coding before the problem has been defined.

**Develop and adhere to an SDP.** The SDP defines the following:

- Project organization and responsibilities
- Lifecycle phases
- Approaches
- Intermediate and end products
- Approach guidelines
- Standards
- Product completion and acceptance criteria
- CM procedures
- QA procedures
- Mechanism for accounting status
- Product and progress reviews
- Cost and schedule reviews
- Contingency plans

The SDP must be made available to all the team members and must be adhered to. The SDP must be updated throughout the software development lifecycle as needed.

**Write down the software requirements.** The software requirements must be written down and recorded early in the software development process (during the requirements analysis phase). Recording the software requirements allows project managers to control the scope of the project and serves as a medium of communication between the developers, customers, software support (CM, QA), and other associated personnel.

**Define specific intermediate and end products.** Each lifecycle phase must have specific intermediate and end products that define well-focused, short-term goals for the development team. They provide the team not only with a means to measure and evaluate progress but also with a sense of accomplishment as each product is delivered.

**Examine alternative approaches.** Do not assume that there is only one way of performing a task (especially during design)—ensure that alternative approaches are considered and evaluated in terms of project objectives and constraints, i.e., schedule, cost, team skill mix, resources, and existing software.

**Perform risk analysis.** Perform a risk analysis at the start of the project. This process identifies potential risks to the successful completion of the project within its proposed schedule and cost. Prioritize potential risks and prepare contingency or mitigation strategies to address those risks most likely to occur. Ensure that developers understand the risks and can alert management early if risks begin to materialize. Review risks periodically throughout the development process.

**Conduct both informal and formal reviews with customers and users.** Plan for and conduct both types of reviews throughout the development process. Formal reviews (e.g., SSR, PDR, CDR) and informal reviews (e.g., demonstrations of the user interface, discussions about a specific set of requirements or portions of the design) provide a means of feedback from customers and users, especially as to whether the developing products meet the customers' and users' needs.

**Use a formal testing process.** All of testing (unit, integration, system, and acceptance) makes up 40%–60% of a completed project's effort, cost, and schedule. Avoid haphazard testing, develop a well-organized and efficient test plan, and follow it.

**Use a central repository.** Keep all development and material records in a central location so that the development process and progress are visible to management and staff.

**Keep a detailed list of TBD items.** Classify TBD items in terms of size, required effort, cost, and schedule. Set priorities and assign personnel to them. Monitor progress to ensure a timely resolution.

**Update system size, required effort, cost, and schedule estimates.** Do not insist on maintaining original estimates. Each phase provides new and refined information about the problem that can be used to improve the original estimates and plan more effectively.

**Allocate sufficient time for testing and integration.** Integration and testing are the most sequential phases in the development process. Little can be done to reduce the work in these phases. Avoid the common error of assuming that the integration and testing effort can be compressed to make up for earlier slippages in the schedule.

**Experiment.** Resources are scarce, and technology is advancing faster than ever before; review alternative approaches to identify areas of improvement. Acquire new skills, try new techniques. Assess the risk of using new approaches, methodologies, and tools and plan for increased time spent learning. Prototype risky, yet seemingly advantageous, new technologies. Plan for technology advancement. Apply *cmi* techniques to continually improve the software development process.

### 5.7.2 Cited References

*{SEL-81-205} Recommended Approach for Software Development, SEL-81-205,* NASA Goddard Space Flight Center, April 1993, pp. 5-3–5-6.

*Section 5.8*

# Don’ts for Project Success

## Contents

- Don't overstaff.
- Don't allow an undisciplined development approach.
- Don't delegate technical details to team members.
- Don't assume that a rigid set of project-specific standards and guidelines ensures success.
- Don't assume that a large set of documentation ensures success.
- Don't deviate from the approved design.
- Don't assume that relaxing project-specific standards and guidelines will reduce costs.
- Don't assume that the pace will increase later in the project.
- Don't assume that schedule slippage can be absorbed in later phases.
- Don't assume that introducing new tools will reduce the schedule.
- Don't assume that everything will fit together smoothly at the end.

*[SEL-81-205]*

## 5.8.1 Don'ts for Project Success—Details

Don't overstaff (this is especially dangerous in early development phases). When a large staff is assigned at the beginning of a project, the tendency is to start designing and building the system before the problem has been understood. Managers are frequently reluctant to admit mistakes after a significant amount of the budget has been spent. This unwillingness to discard work and start over will cause further problems because the remainder of the project will be based on an invalid or incomplete set of requirements and/or design.

**Don't allow an undisciplined development approach.** Software development is a very disciplined application of a set of refined principles, methods, practices, and techniques.

**Don't delegate technical details to team members.** First-line managers must know the technical details of the project. Do not delegate this aspect of the project to the members of the development team, especially to those on a junior level.

**Don't assume that a rigid set of project-specific standards and guidelines ensures success.** Project-specific standards and guidelines promote discipline and consistency in the software development process and facilitate walkthroughs, reviews, and evaluation. However, the experienced judgments and decisions of the project manager, development team leader, and other senior personnel are necessary for the project to succeed.

**Don't assume that a large set of documentation ensures success.** Each phase of the lifecycle does not necessarily require a formally produced document to provide a clear starting point for the next phase. The level of formality and amount of detail to be provided in the documentation must be determined by the project size, lifecycle duration, and lifetime of the system. For example, small projects do not require a *formally* produced preliminary design document. By the time the document is prepared (edited, typed, reviewed, etc.), the design is probably obsolete.

**Don't deviate from the approved design.** As development progresses, developers may tend to implement a slightly different design that still satisfies the requirements. The managers must control this tendency by holding design walkthroughs. Modifications by individual developers may be correct in the local sense but not for the system as a whole.

**Don't assume that relaxing project-specific standards and guidelines will reduce costs.** When a failure to meet a deadline seems imminent, managers and developers sometimes attempt shortcuts by relaxing configuration control procedures, data collection procedures, design formalism or coding standards. In the long run, panic actions cause greater problems and added expense, and do not usually succeed in making the deadline anyway.

**Don't assume that the pace will increase later in the project.** When design, implementation, or testing is progressing slower than anticipated, assign additional senior personnel to help and/or make schedule adjustments. The work rate for a given activity is characteristic of the particular development team; it generally does not change within a short period of time. Do not assume that the team will work faster later on.

**Don't assume that schedule slippage can be absorbed in later phases.** It is a common mistake of managers and overly optimistic developers to assume that the team will be more productive later on in the project. Little can be done to compress the schedule during the later lifecycle phases—the managers should analyze the problem and take appropriate action regarding scheduling as soon as the problem is identified.

**Don't assume that introducing new tools will reduce the schedule.** Another common mistake is to assume that using a new tool (e.g., CASE tools) will increase productivity to such an extent that the schedule can be dramatically reduced. Time to learn the new tool and unrealized assumed or promised capabilities detract from schedule benefits. Introduction of new tools to mitigate current or imminent scheduled slippage usually increases rather than decreases schedule slippage.

**Don't assume that everything will fit together smoothly at the end.** People sometimes erroneously assume that pieces of the system will all fit together with minimal integration effort. Problems will occur; plan ahead and schedule time for integration.

## 5.8.2 Cited References

[*SEL-81-205*] *Recommended Approach for Software Development*, Software Engineering Laboratory Series (SEL-81-205), NASA Goddard Space Flight Center, April 1993, pp. 5-7–5-10.

*Section 5.9*

# Danger Signals and Corrective Measures

## Contents

## 5.9.1 Danger Signals

- Scheduled capabilities are delayed to a later build/release.
- Coding is started too early (staff is too large too early).
- Numerous changes are made to the initial SDP.
- Guidelines or planned procedures are de-emphasized or deleted.
- Sudden changes in staffing (magnitude) are suggested and/or made.
- Excessive (irrelevant) documentation and paperwork is being prepared.
- There is a continual increase in the number of TBD items and ECRs.
- A decrease in estimated effort for system testing is suggested and/or made.
- There is reliance on other sources for "soon-to-be-available" software.

*[SEL-81-205]*

### Danger Signals—Details

**Scheduled capabilities are delayed to a later build/release.** What is the root of this problem? Why did this have to happen? Was the customer involved in this decision? Was the cause one of the high-priority risk items? What could be done next time to prevent this?

**Coding is started too early (staff is too large too early).** This is the trap of considering that only the code is the "real" product and hurrying to begin. Usually this occurs before sufficient understanding of the problem, its requirements and design approach has been reached.

**Numerous changes are made to the initial SDP.** If the development plan requires numerous updates, either the plan was not sufficiently throughout or it was too optimistic. Rather than continuing to make small updates, take the time to reassess and rewrite the plan.

**Guidelines or planned procedures are de-emphasized or deleted.** There is no valid rationale for doing this. If you are trying to save time to meet looming schedule milestones, consider the long-term (e.g., next phase, maintenance phase) effects of this action.

**Sudden changes in staffing (magnitude) are suggested and/or made.** Beware of the syndrome of adding more people to meet imminent schedule milestones. More people require more start up time, communication, etc.

**Excessive (irrelevant) documentation and paperwork is being prepared.** This is definitely a personnel demotivator. Who wants to produce a good product knowing that its only purpose is to become a project statistic and shelfware?

**There is a continual increase in the number of TBD items and ECRs.** This is another morale-deflating situation for the developers and maintainers who see the rising tide of problems. It also causes everyone to lose confidence in the initial product.

**A decrease in estimated effort for system testing is suggested and/or made.** Usually this is a result of slippages in the intermediate milestones. Reducing testing to meet the final delivery date only reduces the quality of the product.

**There is reliance on other sources for "soon-to-be-available" software.** This is definitely a risk-prone strategy. If their software remains "vaporware," its immediate impact on your product will be very real.

### 5.9.1.1 Cited References

[*SEL-81-205*] *Recommended Approach for Software Development*, Software Engineering Laboratory Series (SEL-81-205), NASA Goddard Space Flight Center, April 1993, pp. 4-11–4-14.

## 5.9.2 Corrective Measures

- Stop current activities and review the problem activity.
- Decrease staff to a manageable level.
- Assign a senior staff member to assist junior personnel.
- Increase and tighten management procedures.
- Increase the number of intermediate deliverables.
- Decrease the scope of work and define a manageable thread of the system.
- Audit the project with independent personnel and act on their findings.

[*SEL-81-205*]

### Corrective Measures—Details

**Stop current activities and review the problem activity.** Focus your attention to the problem, get it solved and then proceed with the other activities.

**Decrease staff to a manageable level.** It is better for all (management, staff, customers) to have a smaller staff with greater real productivity than a larger, more costly staff with unclear direction.

**Assign a senior staff member to assist junior personnel.** Assess the work currently being asked of the junior personnel. If it is "over-their-heads" for their position, no one benefits; assign a knowledgeable senior staff member to help.

**Increase and tighten management procedures.** Review management's role, view, and participation in the development process.

**Increase the number of intermediate deliverables.** Intermediate deliverables provide smaller, achievable goals. Managers, developers, and customers can assess progress more easily by tracking these intermediate deliverables.

**Decrease the scope of work and define a manageable thread of the system.** This is sometimes necessary as the problem and requirements for the product become clearer. What was once assumed to be possible within the set schedule and cost may now not be achievable.

**Audit the project with independent personnel and act on their findings.** Fresh eyes and minds can sometime help you to see the obvious, provide new approaches, and rethink the project from a fresh perspective.

### 5.9.1.2 Cited References

[*SEL-81-205*] *Recommended Approach for Software Development*, Software Engineering Laboratory Series (SEL-81-205), NASA Goddard Space Flight Center, April 1993, pp. 4-15–4-23.

## Section 6

# Software Support Activities

# Contents

Software support (i.e., CM, QA) is one of the three major activities performed in the software lifecycle, the other two being software development/maintenance and software project management. In concert with the other two activities, software support is an ongoing activity throughout the software lifecycle. It begins in the planning phase and continues through software retirement.

The major software support activities and the subsections in which they are described are:

- Software Configuration Management, Section 6.1
- Software Quality Assurance, Section 6.2

Software Configuration Management (SCM) and Software Quality Assurance (SQA) activities begin in the software planning phase by defining the SCM and SQA activities to be performed in all subsequent phases. Defining these activities early and in concert with management and software development planning facilitates a coordinated approach to building and maintaining software. Planning in this coordinated manner, with input and review by developers and maintainers, managers, and software support staff, fosters understanding of the functions of the others and a coordinated, team approach to development. The full benefits of such an approach can be realized at the beginning and throughout the life of the project.

*Section 6.1*

# Software Configuration Management

## Contents

### 6.1.1 Introduction

During the software lifecycle (see Section 3.3.1), software products are created, revised, tested, reviewed, released, and delivered and may then undergo multiple cycles of modification, test, review, and rerelease. Software, in all its forms (e.g., requirements specification, code), is changed repeatedly during the development and maintenance phases of its lifecycle. At some point in the lifecycle, identified items of software need to be put under formal control, whereby changes to the software are formally managed. This software support discipline is SCM. Its purpose is to establish and maintain the integrity of the software throughout the software lifecycle, without excessively encumbering those involved in the change process.

More specifically, SCM strives to:

- Ensure that changes to software products are systematically controlled, monitored, and tracked, resulting in software products containing only known and authorized changes
- Establish a change process that does not hinder personnel in easily accessing and understanding current and prior versions of the software products and their associated changes

Managing the software and changes to it is critical for large and complex software development and maintenance projects. As the software, environment and number of personnel involved grow larger (e.g., increasing number of requirements, lines of code) and more complex (e.g., interconnections between levels of requirements; between requirements and design; and between development staff, suppliers, customers, and users), managing changes to the software becomes more critical and must be more rigorous.

### 6.1.2 General Methodology for Software Configuration Management

SCM begins in the planning phase of the software lifecycle and continues throughout the development, operations, and maintenance phases. SCM is composed of four main functions.

| Main Functions of SCM |
|---|
| • Configuration Identification |
| • Configuration Control |
| • Configuration Status Accounting |
| • Configuration Auditing |

Some of the functions listed above are performed over the entire software development lifecycle and others are phase specific. This section describes each of the four main functions of SCM, its phase-independent and phase-dependent activities.

#### 6.1.2.1 Configuration Identification

Configuration identification is the process of identifying what software items will be placed under CC, how they are to be uniquely identified (configuration identification), what combination of versions will comprise a release (version description), and how are the releases to be uniquely identified.

The process of identifying what software items are to be identified as software CIs is typically done during the preliminary software design lifecycle phase. As the functional requirements are allocated to hardware and software and the preliminary software design is engineered, the software CIs are designated. Configuration identification is the process of assigning a unique identifier for each software CI. For this purpose, a CI identification scheme is created. For example:

**Configuration Identification Number = IMSV0-CSC-011 v0.1 940303**

In the example above:

- **"IMSV0"** represents the Project Identification field; in this example, it is IMS Version 0.
- **"CSC"** represents the Configuration Item Category field; in this example, CSC is the acronym for Computer System Component (also referred to as a software subsystem).
- **"011"** represents the Identification Index field, a number from 001—999 used to distinguish CSC category CIs.
- **"v0.1"** represents the Configuration Item Version Number field.
- **"940303"** represents the Version Date field in yymmdd format.

Each release of the software constitutes a software baseline and is uniquely identified. Baselines are major points in the development and maintenance process where, in effect, a snapshot of the system configuration is taken (recorded in the configuration identification documentation). In addition to creating scheduled releases, you can create a baseline at logical points in the developmental lifecycle. Baselines must be approved by a controlling organization (e.g., the CCB) and form the basis for further development. In addition to the interim software builds (internal releases) at logical points in the development effort, there are three major baselines typical to the software development lifecycle: the functional, allocated, and product baselines.

**Functional Baseline**—An accepted set of system-level functional requirements that become the basis of the hardware and software requirements (allocated to hardware and software in the allocated baseline). The functional baseline is usually the final output of the System Requirements Review (SRR).

**Allocated Baseline**—An allocation of functional baseline requirements to system elements (hardware and software CIs). It represents approval of the allocation and interpretation of the requirements. The allocated baseline is usually finalized during the Preliminary Design Review (PDR).

**Product Baseline**—An accepted product, including documentation. It is established at the end of development. The product baseline is usually finalized between the systems testing phase and the acceptance testing phase of the development lifecycle.

### 6.1.2.2 Configuration Control

CC is the process used to protect the integrity of the CI configuration. Proposed changes to any CI configuration are coordinated, evaluated, approved (or disapproved), and implemented through a disciplined process.

There are three types of software CC:

- **Change Control**—The process for requesting changes, deciding and authorizing what changes to make, making changes, and recording the changes.

- **Version Control**—The process of assigning initial version numbers to CIs, assigning successive version numbers to CIs as changes are authorized and implemented, and keeping track of the CI version numbers.
- **Build Control**—The process of assembling the correct versions of CIs to form an approved release, incorporating them into the release, and recording the CI versions in the build documentation.

There are two categories of software CC: developmental CC and formal CC. The scope of developmental CC is limited to control of CIs during the development lifecycle (the CI is delivered to the customer for his/her approval before customer acceptance). Formal CC is implemented after a CI has been delivered to the customer.

The developmental CC change process differs from the formal CC process in one very important aspect: developmental CC does not require change authorization from the customer. Developmental CC is the contractor's internal mechanism for change control.

The formal CC process involves the same decision-making body (CCB) as the developmental CC process, with the exception of the final change authority. In developmental CC, the contractor's project manager is typically the chairperson for the CCB and has the final authority for all CI changes. In the formal CC change process, the customer's Contracting Officer's Technical Representative (COTR) is typically the CCB chairperson.

**Examples of Software Change Control**

**Developmental Change Control**

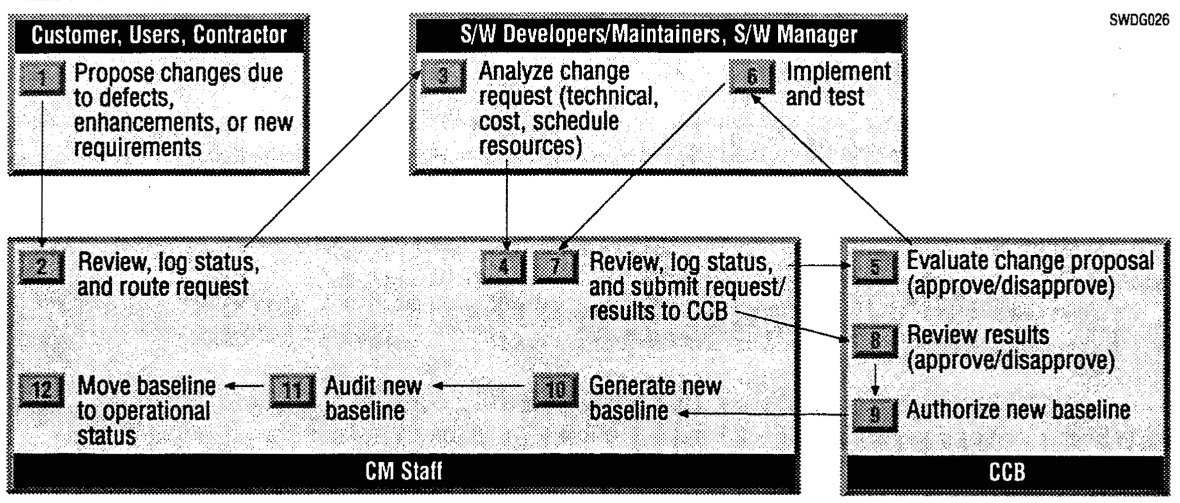
- Changes to CIs resulting from design walkthroughs or code walkthroughs
- Changes to CIs to correct errors found during unit-level testing, integration testing, system testing, or acceptance testing
- Changes to software documentation CIs to reflect changes made to software CIs

**Formal Change Control**

- Changes to delivered CIs to correct errors found during system operation
- Changes to delivered CIs to incorporate software enhancements
- Changes to software documentation CIs to reflect changes made to delivered software CIs
- Changes to system user documentation CIs

The typical change control process entails the following procedural steps. This process is illustrated in Figure 6.1.2.2-1.

1. An SMR is initiated and submitted to the CM staff.
2. CM staff reviews the SMR for completeness, logs it into the tracking system, and routes it to the appropriate manager.
3. The manager determines whether it is a duplicate, is invalid, or requires analysis. Duplicates and invalid SMRs are closed at this point. An analyst is assigned for valid SMRs. The SMR is analyzed for feasibility, technical solution, impacts on requirements, design, estimated time to complete change, cost, and benefit. The manager reviews the


SWDG026
Customer, Users, Contractor
1 Propose changes due to defects, enhancements, or new requirements
S/W Developers/Maintainers, S/W Manager
3 Analyze change request (technical, cost, schedule resources)
6 Implement and test
2 Review, log status, and route request
4 7 Review, log status, and submit request/ results to CCB
12 Move baseline to operational status
11 Audit new baseline
10 Generate new baseline
CM Staff
5 Evaluate change proposal (approve/disapprove)
8 Review results (approve/disapprove)
9 Authorize new baseline
CCB

| CM Function | Steps | Comments |
|---|---|---|
| Configuration Control | 1–12 | |
| Configuration Auditing | 11 | •Ensures correct versions of CIs included<br>•Ensures all change requests incorporated<br>•Ensures documentation reflects baseline |
| Configuration Status Accounting | Status information from Steps 2, 4, 7, 10–12 | • Logs and tracks status of change requests and affected CIs<br>• Generates status reports |

**Figure 6.1.2.2-1. Configuration Control, Status Accounting, and Auditing**

analysis for thoroughness and completeness. If it is adequate, the manager routes SMR back to the CM staff.

4. Upon completion of the analysis, the CM staff updates the request status and prepares it for the next CCB meeting. The CCB makes all preparations for the next CCB meeting.

5. The CCB receives and evaluates the SMRs and approves/disapproves each SMR. Authorized SMRs are routed for change implementation and testing.

6. Software modifications are implemented and tested in a development environment. All changed products, results, and documentation are routed back to the CM staff.

7. CM staff updates the SMR status and prepares it for the next CCB meeting.

8. The CCB reviews the recorded implementation information and approves (or disapproves) the incorporation of the changes into the next baseline.

9. At some point, the CCB authorizes the generation of a new baseline.

10. CM staff incorporates authorized changes into the affected baseline, updates the SMR status records, and updates the version documentation.

11. CM staff audits the new baseline to ensure that only authorized changes have been included and that all documentation has been updated accordingly.

12. CM staff direct and coordinate the transition of the new baseline into the operational environment.

### 6.1.2.3 Configuration Status Accounting

The purpose of configuration status accounting is to maintain the version status records for all CIs and make the current status records available to management in the form of status accounting reports. These reports include a list of the CIs, their current version numbers, and the status of all open SMRs against those CIs.

Status accounting reports are distributed to management on a periodic basis. The report content may vary to suit individual managers' needs, but typically the reports are sorted by CI identifier (to correlate open SMRs to CIs), by SMR number (to review the status of the SMRs in chronological sequence), or by SMR priority.

### 6.1.2.4 Configuration Auditing

An audit of the configuration documentation against the actual CIs is referred to as a configuration audit. There are two types of configuration audits: periodic and product baseline.

Periodic configuration audits are performed by the CM staff regularly to assess the effectiveness of the configuration identification, CC, and status accounting procedures and to find (and correct) configuration documentation inconsistencies.

Product baseline configuration audits are performed immediately before product delivery. The purpose of this audit is to verify that the configuration documentation completely and accurately describes the software product baseline CIs and that all SMRs written against the product baseline CIs have been resolved and closed.

### 6.1.2.5 Phase-Independent SCM

The following activity, library, and organization are performed, used, and function, respectively, throughout many phases of the software lifecycle.

#### 6.1.2.5.1 Continuous Identification of Configuration Items

The initial identification of CIs is done in the planning phase of development. This includes generating a list of CIs and defining the naming conventions, standards, and procedures to be followed throughout development and operations.

In subsequent phases, as a better and more detailed understanding of the software system develops, new CIs are identified or modified, and new or modified conventions, standards, and procedures are necessary.

It is the responsibility of the CM staff to manage and monitor these changes, including any updates to documentation under CC.

#### 6.1.2.5.2 Software Development Library

The SDL is a controlled collection of software (i.e., code and documentation) and associated tools and procedures used to facilitate the development and operational support of the software. The SDL is first established to control the initial documentation placed under developmental or formal CC. These may include the SPMP, SDP, or SRS.

The CM staff is responsible for physical or electronic access to the SDL, and controls the checkin/checkout process for any changes to its contents. For example, in response to an SMR, a programmer may check out a set of software modules to update them. Following final authorization by the CCB that the modifications were correctly made and tested, the CM staff would then check in the updated modules.

The SDL contains all the software that was controlled through both developmental and formal CC.

#### 6.1.2.5.3 Configuration Control Board

The CCB is a group of technical and managerial personnel who approve or disapprove any changes to CIs currently under CC. They also approve or disapprove waivers or deviations to the CIs currently under CC.

The CCB is the authorization organization for both the developmental and formal CC process. For developmental CC, the software project manager is usually the chairman; customer participation may be only as a nonvoting member. The customer, is however, the chairman of the CCB for the formal CC process. As such, he or she has final authority over changes to CIs.

The CM staff are usually board members and are responsible for administratively and technically supporting the board meetings.

### 6.1.2.6 Phase-Dependent SCM

#### 6.1.2.6.1 Planning Phase

Planning for each of the SCM functions is conducted in the project planning phase.

**Planning Activities for Each of the SCM Functions**

- **Configuration identification** planning includes initial identification of baselines and the software items comprising the baselines, the associated development phase when each baseline is to be produced, the review and approval events including acceptance criteria for each baseline, and defined procedures to label and catalog both software code and documentation.
- **Configuration control** planning includes definition of the level of authority for change approval for the lifecycle phases, methods to be used in processing change proposals to established configurations, methods of implementing approved change proposals, procedures for software library control, methods for CC of interfaces with external systems/organizations, and control procedures for associated software (e.g., Commercial Off-the-Shelf (COTS), in-house support software).
- **Configuration status accounting** planning includes defining how status information on CIs will be collected, verified, stored, processed, and reported; identifying what periodic reports are to be provided and distributed to whom; and describing how to implement any special status tracking requirements.
- **Configuration audit** planning includes defining when audits are to take place, what CIs are to be audited, the role of CM staff in these audits, and the procedures to be used in the identification and resolution of problems found from the audits.

The planning results are documented in the Software Configuration Management Plan (SCMP). The SCMP describes the four main functions of SCM as well as the organizations and personnel involved in the change process and their roles and authority.

The SCMP consists of two major sections: management and CM activities. The first section describes the organizations associated with CM (i.e., their authority, responsibility, and relationships). Such organizations are the CCB, the software development group, management, the CM staff itself, the QA staff, and possibly other personnel responsible for external systems. The SCMP also describes the interfaces between these organizations as well as the schedule for CM implementation (e.g., milestones for creating the CM staff, CCB, software baselines). The second section describes the major activities to be performed under each of the four functions of SCM.

At the end of the planning phase the Software Project Management Plan (SPMP), Software Development Plan (SDP), QA Plan and SCMP are placed under developmental CC. The SMR can be used as the vehicle for requesting changes to these plans (and all other controlled documentation), precluding the necessity for creating another form that would follow essentially the same process as the SMR.

### 6.1.2.6.2 Requirements Analysis Through Operations and Maintenance

Table 6.1.2.6.2-1 delineates SCM activities by phase.

**Table 6.1.2.6.2-1. SCM Activities by Phase**

| Lifecycle Phase | Major SCM Activities |
|---|---|
| Planning | • Develop the SCMP.<br>• Place planning documents under developmental CC.<br>• Establish and control the Software Development Library (SDL). |
| Requirements Analysis | • Support requirements analysis and documentation development using developmental CC.<br>• Perform a configuration audit before delivery of Software Requirements Specification (SRS) and Interface Requirements Specification (IRS).<br>• Place the SRS and IRS under formal CC.<br>• Track the status of ECRs, SMRs, and resulting modifications to CIs. |
| Preliminary Design | • Support design and document development using developmental CC.<br>• Establish and support a CCB to review proposed changes to CIs under CC.<br>• Track the status of ECRs, SMRs, and resulting modifications to CIs. |
| Detailed Design | • Support design and documentation development using developmental CC.<br>• Perform a configuration audit before delivery of the SDD.<br>• Place the SDD under operational CC.<br>• Support the CCB in reviewing proposed changes to CIs under CC.<br>• Track the status of ECRs, SMRs, and resulting modifications to CIs. |
| Code and Unit Test | • Place unit-tested modules under developmental CC.<br>• Support the CCB in reviewing proposed changes to CIs under CC.<br>• Track the status of ECRs, SMRs, and resulting modifications to CIs.<br>• Perform configuration audits as necessary. |
| Integration Testing | • Following successful testing, place the software subsystems under developmental CC.<br>• Support the CCB in reviewing proposed changes to CIs under CC.<br>• Track the status of ECRs, SMRs, and resulting modifications to CIs.<br>• Perform configuration audits as necessary. |

Table 6.1.2.6.2-1. SCM Activities by Phase (Continued)

| Lifecycle Phase | Major SCM Activities |
|---|---|
| System Test | • Following successful testing, place the software systems under formal CC.<br>• Perform a configuration audit before delivery of software.<br>• Support the CCB in reviewing proposed changes to CIs under CC.<br>• Track the status of ECRs, SMRs, and resulting modifications to CIs. |
| System Acceptance | • Support the CCB in reviewing proposed changes to CIs under CC.<br>• Track the status of ECRs, SMRs, and resulting modifications to CIs. |
| Operations and Maintenance | • Perform a configuration audit on each new release of the software.<br>• Support the CCB in reviewing proposed changes to CIs under CC.<br>• Track the status of ECRs, SMRs, and resulting modifications to CIs. |

## 6.1.3 SCM Tools

Software tools are available for software CC from "freeware" sources and software vendors. In the freeware category, a Unix utility titled Revision Control System (RCS), distributed by The Free Software Foundation, is available and works quite well. The only drawback is that you must be developing on a Unix platform. A predecessor of RCS, Source Code Control System (SCCS), is also available for Unix platforms. Aegis, distributed under the terms of the GNU General Public License, is characterized as a "project change supervisor" (Unix environment). Commercial tools include:

- CCC/Harvest by Softool
- ClearCase by Atria Software
- CMS by Digital Equipment Corp.
- Endeavor by Legent Corp.
- PVCS by Intersolv

CCC/Harvest and ClearCase operate in the Unix environment. CMS operates in the VMS environment. PVCS works in Windows NT, MS-DOS, OS/2, AIX, Sun OS and Solaris, HP-UX, and SCO Unix environments.

## 6.1.4 Tailoring to a Small Project

Tailoring the information provided in this section is essential for defining and implementing SCM in your project or task. Regardless of project size, the SCM function needs to be performed at some level. Only the level of detail, process rigor, and products vary among projects. Some of the tailoring factors to be considered are:

- Time
- Resources
- Complexity
- Contractual commitments and requirements
- Intended use of the product

One tailoring strategy is to take each of the four elements one at a time, decide what the minimum requirements are for each (for a streamlined, but effective process), and determine the least cumbersome implementation. The results of this tailoring should be documented in the SCMP. Most likely the SCMP itself should be a less formally produced document, yet still capture the essential SCM process, procedures, and activities. The following is an approach for tailoring each of the SCM functions.

#### 6.1.4.1 Tailoring the Configuration Identification Process

Tailoring configuration identification is relatively straightforward. First, determine the categories of the software CIs (e.g., programs, subsystems, modules). Determine the likely maximum number of CIs in each category (e.g., 1–10, 10–100). Next, determine the minimum information required to differentiate between CIs and how much other information is practical to include in the identifier. In the example in Section 6.1.3.1, the project, CI category, an index number, version, and date were included. You may decide that you need only the category, index, and version number. You may find that for documents you need to add a subcategory. The identifier must convey at least the minimum information *required to differentiate between all CIs*. Anything in the identifier beyond that is for convenience.

#### 6.1.4.2 Tailoring the Change Control Process

Tailoring the CC process is more challenging. You need to consider the following and tailor each to your particular project:

- Change authority (CCB)
- CCB process
- Vehicle for documenting and tracking changes (SMR form)
- Process controls (managers at key decision points)
- Status information capture and recording process
- Process standards for change analysis, implementation, testing, and baseline integration
- SMR closure criteria

For example, on your project an actual CCB may be impractical; change authorization may reside with one person. Information capture may be sufficient at the beginning and end of the change process. Problem/enhancement impact analysis may be limited by schedule and resource constraints. SMRs may be produced only for essential, high-priority changes. SMRs may be allowed to go from analysis to implementation and through testing without management review and control.

#### 6.1.4.3 Tailoring the Status Accounting Process

Tailoring the status accounting process is straightforward. The nature of status accounting is, simply, good recordkeeping. First, determine the minimum information necessary to report the status of each CI. Typically, this is the CI name, identifier, version number, version date, the change status, and which open SMRs are written against that CI (if any). This information can be recorded manually or in a data base program. Common sense says that the key to credible status accounting records (as with any records) is the frequency and accuracy of your record updates.

### 6.1.4.4 Tailoring the Configuration Audit Process

SCM audits are essentially the same process regardless of the project development rigor. Auditing new baselines before they are made operational is still required to ensure that the correct versions of the CIs are included in the new baseline. Tailoring of internal audits (i.e., configuration documentation is selected randomly for verification and compared to the actual CI, and the results of the comparison are recorded and reported to project management) is easily achieved by reducing the frequency of the audits and the number of records sampled for each audit. As you would for any process, decide what is practical, write the procedure, implement the process, and modify it over time to correct inadequacies.

## 6.1.5 Suggested Reference Material

"IEEE Standard Glossary of Software Engineering Terminology," IEEE-STD-610, ANSI/IEEE Std 610.12-1990, February 1991.

"IEEE Standard for Software Configuration Management Plans," IEEE-STD-828, ANSI/IEEE Std 828-1983, June 1983.

"Configuration Management," MIL-STD-973, Military Standard 973, April 1992.

"Configuration Management Plan, Data Item Description," NASA-DID-600, NASA-STD-2100-9,1 NASA Software Documentation Standard, Software Engineering Program, July 1992.

## 6.1.6 Appendix

### 6.1.6.1 Sample Tables of Contents

| Configuration Management Plan—Table of Contents (Reference: NASA-DID-M600 from NASA-STD-2100-91, July 1992) | |
|---|---|
| 1.0 | Introduction |
| 2.0 | Related Documentation |
| 3.0 | Configuration Management Process Overview |
| 4.0 | Configuration Control Activities |
| 4.1 | Configuration Identification |
| 4.2 | Configuration Change Control |
| 4.2.1 | Controlled Storage and Release Management |
| 4.2.2 | Change Control Flow |
| 4.2.3 | Change Documentation |
| 4.2.4 | Change Review Process |
| 4.3 | Configuration Status Accounting |
| 5.0 | Abbreviations and Acronyms |
| 6.0 | Glossary |
| 7.0 | Notes |
| 8.0 | Appendix |

| Configuration Management Plan—Table of Contents (MIL-STD-973, 17 April 1992) | |
|---|---|
| 1.0 | Introduction |
| 2.0 | Reference Documents |
| 3.0 | Organization |
| 4.0 | Configuration Management Phasing and Milestones |
| 5.0 | Data Management |
| 6.0 | Configuration Identification |
| 7.0 | Interface Management |
| 8.0 | Configuration Control |
| 9.0 | Configuration Status Accounting |
| 10.0 | Configuration Audits |
| 11.0 | Subcontractor/Vendor Control |

**Software Configuration Management Plan—Table of Contents**
(IEEE Standard 828-1983)

## *Section 6.2*

# Software Quality Assurance

## Contents

## 6.2.1 Introduction

It is important to understand that QA does not automatically guarantee quality software. QA ensures that the project team is developing software according to an approved plan and that the software will satisfy the specified requirements. QA does not determine whether the software requirements are complete and accurate. The entire project team, which includes software engineering, software management, software CM, Independent Test Organization (ITO), and software QA, works together to build quality into the software products.

The degree to which a product meets its specified requirements and to which the project team follows approved methods and procedures is clearly the responsibility of the project managers and functional managers.

An important role that a QA organization can perform is systematically and continuously collecting defect data, analyzing those data, and making recommendation for improvements to project management, using continuous measurable improvement (cmi) techniques. Properly used, defect trend data can help an organization reach new levels of quality while substantially reducing the normal costs associated with the production software.

The overall quality objective is to ensure that the software products are suitable for use by the end user.

The software should satisfy the need for which it was intended. All organizations have responsibility for quality. All organizations should conduct their activities in a consistent manner in support of the quality mission.

The QA organization provides the structure, discipline, methods, and procedures to ensure that the software products meet their specified characteristics in support of the global quality mission of ensuring that the software is suitable for use by the end user.

The specific goals of QA are to:

- Establish and perform the necessary evaluations and procedures for the systematic evaluation of software development processes and products.
- Assure management, both company and project, that the established and approved applicable standards and methods are used for the development, evaluation, control, and delivery of software products (both developmental and nondevelopmental as well as deliverable and applicable nondeliverable).
- Collect relevant quality data consistently across all projects, analyze the data, perform defect/ discrepancy trending analysis, recommend corrective actions, and report the information to management and clients, as required.
- Ensure that product and process-related deficiencies are identified, analyzed, tracked, and appropriately reconciled.
- Develop an efficient and effective Software Quality Assurance Plan implementing all relevant requirements from company policy and customer contract.
- Assist all organizations in the evaluation process of self inspection with the intent of achieving continuous improvement throughout the project.

### 6.2.2 General Methodology for Quality Assurance

Each project develops a project Software Quality Assurance Plan (SQAP) for software if software is being developed or delivered as part of the project requirement. The plan will define QA's role, as part of the project team, in supporting the development of the software product. *The intent of the plan is to tailor an effective and efficient QA process based on company guidelines, the contract, and the needs of the project.* The plan is a living document and must be continuously evaluated for change. External customer needs and project characteristics often influence the need for change to all project plans.

There are two basic types of evaluations that are conducted by the QA function. One is the evaluation of a process or activity that is used in the production of a software product, its validation, or its control. This is a process evaluation. Process evaluations are characterized by the systematic review of processes to ensure that they are conducted in accordance with approved and documented methodologies.

Another is the evaluation of products that are used or developed as part of the product and/ or its operating environment, and products that are used to manufacture, test, or control the product. These evaluations are product evaluations. Product evaluations are characterized by the close examination of a product such as data, computer programs, designs, and documentation.

#### 6.2.2.1 Phase-Independent Quality Evaluations

Some QA activities are performed during all phases of the software lifecycle. These repetitive tasks include monitoring activities for compliance to plans and procedures and evaluating documentation.

##### 6.2.2.1.1 Evaluation of Corrective Action Process

The problem reporting, analysis, and change activity is the responsibility of project management, but the tasks are generally performed by all team members. The QA will verify that the following actions are complete or items correct:

- Ensure that the project has a defined, closed-loop methodology for collecting and analyzing problem reports and that appropriate changes are made to the project products.
- Evaluate the problem data to ensure that they collect the required information, including classification and priority.
- Ensure that the qualification testing of changes is adequate to ensure that no new problems are introduced and that the changes satisfy their intended purpose.
- Ensure that the project defect data are included in the statistical data used to analyze process trends.
- Ensure that the process trend analysis is available to the project management for its review and action.

#### 6.2.2.1.2 Evaluation of Software Plans

It is the responsibility of the software project manager, to ensure that there is an adequate schedule and budget for the proper review and correction of all software plans (e.g., SPMP, SDP, SCMP, SQAP). The QA analyst will verify that the following are complete and/or correct:

- The plan is reviewed and problems are identified in a timely manner prior to the delivery to the customer.
- The document is reviewed for compliance to the contractual requirements.
- The document is internally consistent.
- The document is consistent with other project plans, and all schedules are consistent.
- The plans have been distributed, and project personnel have been informed of the plans and their requirements.

#### 6.2.2.1.3 Evaluation of Software Management Activities

The QA analyst will perform these evaluation tasks at any appropriate time during the software development lifecycle. The QA analyst will verify that the following are complete and/or correct:

- Appropriate management practices are adequately specified by the SPMP.
- Practices are contractually compliant.
- Practices described in the planning documents are implemented.
- Results of the activities are recorded.

#### 6.2.2.1.4 Evaluation of Software Configuration Management Activities

The QA analyst will perform evaluations of SCM tasks at any appropriate time during the software development lifecycle. The QA analyst will verify that the following actions are complete and/or correct:

- Appropriate CM practices are adequately specified by the SCMP.
- CM practices are contractually compliant.
- Practices implemented by the planning documents are routinely followed.
- Results of the CM activities are recorded.

#### 6.2.2.1.5 Evaluation of Software Engineering Activities

These software engineering evaluations will be conducted regularly during the project lifecycle at the discretion of the QA analyst. The QA analyst will verify that the following actions/items are complete and/or correct:

- The engineering practices to be used on the project are consistent with the SDP, applicable company guidelines and contractual requirements.
- All personnel have had adequate training or experience with the standards and methods.

- Practices and procedures defined and incorporated into the project methodology are routinely followed.

#### 6.2.2.1.6 Evaluation of Software Testing and Qualification Activities

The evaluation criteria for software testing and qualification activities are independent of any particular phase and are oriented more to the management of the test and qualification activity rather than the specific execution of phase-dependent tasks. The QA analyst will verify that the following actions/items are complete and/or correct:

- The planning for test activities is timely, defined, and documented in accordance with the management plans. These activities are conducted according to and in total compliance with the contract.
- Reviews of the test activity are conducted in accordance with all plans and schedules.
- Test team members have defined standards and methods that are documented for this project.
- The standards and procedures for test and test documentation are routinely followed.

#### 6.2.2.1.7 Evaluation of Software Development Library

The software management function is responsible and accountable for the proper and contractually compliant operation of the SDL activities in all phases of the program. The QA analyst will verify that the following SDL related actions/items are complete and/or correct:

- The SDL has been systematically planned, the plan has been documented, in the SDP and there are procedures for the operation of the SDL.
- The SDL is operated in accordance with the approved methodologies, and procedures are identified as applicable to the project.
- The approved procedures and methods are adequate in safeguarding the developing product and providing controlled access to the current versions of the configuration items as well as providing the capability to reconstruct any previous version.

#### 6.2.2.1.8 Evaluation of Software Storage, Handling and Delivery

The software management function is responsible and accountable for the proper and contractually compliant operations of storage, handling, and delivery activities in all phases of the program. The QA analyst will verify that the following processes are accurate and/or complete:

- The activities have been systematically planned and documented, and there are procedures for implementation.
- The activities are operated in accordance with the approved methodologies, and procedures are identified as applicable by the project.
- The activities have approved procedures and methods that are adequate in safeguarding the products from physical damage and loss and that ensure contractual compliance.

#### 6.2.2.1.9 Evaluation of Software Media and Documentation Distribution

The software management function is responsible and accountable for the proper and contractually compliant document and media distribution activities in all phases of the program. The QA analyst will verify that the following processes are complete and/or correct:

- Document and media distribution have been systematically planned, the plan has been documented, and there are procedures for implementation.
- Distribution is conducted to the approved methodologies, and procedures identified as applicable by the project.
- Distribution procedures and methods are adequate in ensuring that the distribution of documentation and media is complete and timely and that they are contractually compliant.

#### 6.2.2.1.10 Evaluation of Subcontract Management

Software project management and functional management are responsible and accountable for the proper application of the software engineering processes in all phases of the program. The QA analyst will verify that:

- The contract is complete and is a flowdown of the requirements in the prime contract and/or company procedures to ensure that the subcontracted element is produced in a manner that is consistent with the rest of the system.
- The contracted elements have clearly stated physical, performance, and functional requirements, including timing and sizing requirements.
- The contract clearly defines the schedule of events and activities and includes management reviews.
- The management procedures for oversight of a subcontractor are defined and these procedures are followed.
- The subcontractor's products are evaluated for functional and physical characteristics prior to acceptance.

#### 6.2.2.1.11 Evaluation of Software Documentation

Software project management is responsible and accountable for the proper and complete documentation of its products in accordance with the contractual and company requirements. The QA analyst will verify that the following actions/items are complete and/or correct:

- Each document is compliant with its contractual requirements, including content and format.
- Each document defines the appropriate level of information consistent with its purpose.
- The contents of each document flow from a higher level document and the document does not introduce new requirements.
- The contents of each document are accurate, correct, not redundant, and technically consistent with other documents and with itself.

- Documents are produced in accordance with the processes defined for the project, including reviews, review cycles, and distribution lists.
- Documents are produced in accordance with published schedules and are available within the appropriate phase of the project.
- Documents are placed into formal CC prior to the delivery to the customer, and all changes to approved and controlled documents are correct and approved before they are made.

#### 6.2.2.1.12 Evaluation of Software

Project software development and management are responsible and accountable for the accurate and correct coding of its products in accordance with the contractual and company requirements. The QA analyst will verify that:

- All software that is developed and deliverable is written to conform to the contractual requirements and the company guidelines and specifications approved for the project, including maintainability, timing, and sizing requirements.
- Software that is deliverable, but was not developed by the project, is contractually compliant, adequately documented, qualified for use by examination or test, and appropriately licensed and has the Government data rights defined.
- All software that is developed for use in the product or that is used to test or develop the product is in compliance with its technical definition, including its development folders and design documentation, and is properly used for the function for which it was intended.
- All software has been tested to approved procedures, demonstrates the implementation of all applicable requirements, and is suitable for end-user use.
- All software used on the project is properly classified, controlled, maintained, defined, and protected from intentional and unintentional unauthorized change.
- All software meets all established criteria for the product, such as internal consistency, traceability of code to the design and interface documents, and understandability.

### 6.2.2.2 Phase-Dependent Quality Evaluations

These are the software quality functions that apply to the activities and products specific to a software lifecycle phase.

#### 6.2.2.2.1 System Requirements Analysis/Design

Software project management is responsible and accountable for the proper and complete documentation of their products in accordance with the contractual and company requirements and for the proper application of the software engineering processes in all phases of the program. The QA analyst will verify that:

- System requirements review is supported as specified in the contract.
- System Design Review is supported as specified in the contract.
- System requirements are consistently and completely specified and have been allocated appropriately to the software systems as documented in the System Design Document.

- Preliminary software requirements are defined in the preliminary SRSs.
- Preliminary interface requirements specify each external interface to each software system and are documented in the preliminary IRS.
- The preliminary set of qualification requirements for each software system is defined in the preliminary SRS.
- The qualification requirements are consistent with and traceable to the qualification requirements defined in the system specification and are documented in a cross-reference document.
- All preliminary software documents, the SRS and IRS, are placed under developmental CC.

#### 6.2.2.2.2 Software Requirements Analysis

Software project management is responsible and accountable for the proper and complete documentation of its products in accordance with the contractual and company requirements and for the proper application of the software engineering processes in all phases of the program. The QA analyst will verify that:

- Software Specification Reviews (SSRs) are conducted in accordance with contractual requirements.
- The SRSs and the IRSs are authenticated by the contracting agency to form the allocated baseline.
- A complete set of software requirements has been specified, and the appropriate SRSs have been updated.
- A complete set of requirements for external interfaces for all software systems is specified, and the IRS is completed.
- The final SRSs and the final IRSs are placed into formal CC.
- The qualification requirements are completely defined for each software system and documented in the SRS.
- The requirements are traceable to the qualification requirements defined in the system specification, and this traceability is documented as defined in the management plans.

#### 6.2.2.2.3 Software Preliminary Design

Software project management is responsible and accountable for the proper and complete documentation of its products in accordance with the contractual and company requirements and for the proper application of the software engineering processes in all phases of the program. The QA analyst will verify that:

- Preliminary Design Reviews (PDRs) are conducted in accordance with contract requirements.
- Preliminary designs for each software system are adequately detailed and defined in the preliminary Software Design Document (SDDs).
- The design for external interfaces is adequately detailed in the preliminary IDD.

- Test requirements for subsystem testing are established and recorded in the Software Development Files (SDFs).
- All appropriate engineering information is defined in the SDD.
- The Software Test Plan (STP), preliminary Software Design Document (SDD), and the preliminary Interface Design Documents (IDDs) are placed under developmental CC.
- The formal qualification tests for each software system are documented in the STP.
- The plans are clear, concise, and executable.
- The necessary resources are scheduled, and configurations are identified for each software system qualification test.

#### 6.2.2.2.4 Software Detailed Design

Software project management is responsible and accountable for the proper and complete documentation of its products in accordance with the contractual and company requirements and for the proper application of the software engineering processes in all phases of the program. The QA analyst will verify that:

- Critical Design Reviews (CDRs) are conducted in accordance with contract requirements.
- The detailed designs for the software systems and for the external interfaces for each software system are documented in the SDD and IDD, respectively.
- The detailed designs include the design requirements for subsystems.
- The test responsibilities, test cases, and schedules for module testing and subsystem integration and test are complete and recorded in the SDFs.
- The SDD is updated with additional engineering data, if appropriate.
- Each software system has defined test cases that are documented in the STD document or equivalent as defined by the management plans.
- The test cases for each software system are adequate to test the performance of the software system.
- The final IDDs and the Software Test Descriptions (STDs) are placed under formal CC.

#### 6.2.2.2.5 Software Coding and Module Testing

Software project management is responsible and accountable for the proper and complete documentation of its products in accordance with the contractual and company requirements and for the proper application of the software engineering processes in all phases of the program. The QA analyst will verify that:

- The SDDs have been properly updated with all approved changes.
- Modules are coded, test procedures are developed, modules are tested, and appropriate documentation is updated.
- The module source code, which has been successfully compiled, link edited, executed or examined, tested or analyzed, and evaluated, is placed under developmental CC.
- Subsystem test procedures are developed and documented in the SDFs.

### 6.2.2.2.6 Subsystem Integration and Testing

Software project management is responsible and accountable for the proper and complete documentation of its products in accordance with the contractual and company requirements and for the proper application of the software engineering processes in all phases of the program. The QA analyst will verify that:

- Test Readiness Reviews (TRRs) are conducted in accordance with contract requirements.
- Subsystem integration and testing is accomplished in accordance with the documented procedures and the results documented in the SDFs.
- Appropriate design documentation and code are updated correctly as a result of the testing.
- Each software system test case, previously defined, has documented procedures to invoke the test.
- The tests have been conducted and the results documented, reviewed, and approved to allow the software system to progress to formal qualification testing.
- The properly updated design documents and source code for successfully tested subsystems are placed under developmental CC.

### 6.2.2.2.7 System Testing

Software project management is responsible and accountable for the proper and complete documentation of its products in accordance with the contractual and company requirements and for the proper application of the software engineering processes in all phases of the program. The QA analyst will verify that:

- Version Description Documents (VDDs) are produced for each successfully tested software system.
- All necessary documentation for the Functional Configuration Audit (FCA) and Physical Configuration Audit (PCA) is produced.
- Documentation is revised as necessary in response to the software system testing, and the Software Product Specification (SPS) is appropriately produced.
- Updated source code is produced for delivery.
- Each software system is formally tested, and the results are documented, reviewed, and approved as demonstrating that the software system meets its intended use.
- The source code is appropriately updated and validated and is demonstrated to be consistent with the sum total of its documented technical definition.

### 6.2.2.2.8 Acceptance Testing

Software project management is responsible and accountable for the proper and complete documentation of its products in accordance with the contractual and company requirements and for the proper application of the software engineering processes in all phases of the program. The QA analyst will verify that:

- FCAs are supported in accordance with the contract requirements.
- PCAs are supported in accordance with the contract requirements.

- Physical and functional audit actions items are closed in accordance with the contractual requirements.
- All approved changes to the software and its technical definition are properly made.
- The system, including all CIs, has been formally tested, the results documented, and accepted by the customer as suitable for their use.
- All source code is updated and is consistent with the sum total of its technical definition.
- The Software Product Specifications, after FCA and PCA authentication by the contracting agency, are placed into formal configuration control to form the product baseline.

### 6.2.3 Tailoring to a Small Project

Each project is unique. The quality needs of the project vary depending on many factors. Some of the factors to be considered in planning a quality program are:

- The consequence of failure of the product or any of its discrete elements
- The source of elements of the product such as COTS, customer furnished, contractor furnished, and reusable code elements
- Tools available for use (e.g., dynamic and static code analyzers)
- Relative complexity and size of the project
- Developmental and support staffing on the project

These factors have a direct bearing on the implementation of the tasks to be performed and on the allocation of resources.

Consideration must be given to the frequency of evaluations to be performed on the contract. Few contracts require 100% of all items to be inspected, but that is the most frequent approach used by QA. What effort of inspection and analysis is reasonable to ensure the quality of the product is dependent on all of the other factors being considered.

The key function of QA is to implement *cmi* by collecting defect data, analyzing those data, and working with the project team to implement corrective actions to reduce errors.

### 6.2.4 Suggested Reference Material

Automatic Dependent Surveillance (ADS) Computer Software Quality Program Plan, Hughes STX Corp., August 1991.

"Software Quality Assurance Plan," DOD-STD-1703, Department of Defense Standard-1703, February 1987.

"Software Quality Program Plan," DOD-STD-2168, Department of Defense Standard-2168, April 1988.

"Defense System Software Development," DOD-STD-2167A, Department of Defense, February 1988.

## 6.2.5 Appendix

### 6.2.5.1 Sample Tables of Contents

**Software Quality Program Plan—Table of Contents**
(Reference: Automatic Dependent Surveillance (ADS) Computer Software Quality Program Plan (CSQPP), 14 August 1991)

1.0 Introduction
 1.1 Scope
 1.2 Purpose
 1.3 Applicability
 1.4 Definitions
2.0 Reference Documents
3.0 Requirements
 3.1 Organization
 3.2 Personnel
 3.3 Resources
 3.4 Development Process Flow
 3.5 Audits
 3.6 Standards and Procedures
 3.7 In-Process Controls
 3.8 Configuration Management
 3.9 Library Controls
 3.10 Corrective Action, Reporting, and Control
 3.11 Tools
 3.12 Supplier Control
 3.13 Test Controls
 3.14 Software Independent Verification and Validation
 3.15 Records
 3.16 Reports
 3.17 Installation and Checkout
 3.18 Storage, Handling, and Shipping
Appendixes

**Software Quality Assurance Plan—Table of Contents**
Reference: DOD-STD-1703

1.0 Introduction
 1.1 Purpose
 1.2 Scope
 1.3 Applicable Documents
  1.3.1 Customer Documents
  1.3.2 Developer Documents
  1.3.3 Project-Specific Documents
 1.4 QA Plan Maintenance
 1.5 Project Software Development Cycle
2.0 Quality Assurance Organization
 2.1 QA Operational Responsibilities
 2.2 QA Program Responsibilities
 2.3 QA Reports
3.0 Quality Assurance Functions
 3.1 Quality Standards and Procedures
 3.2 Audits
 3.3 QA Participation in Reviews, Audits, Control Boards
 3.4 Test Monitoring
 3.5 Discrepancy Control Monitoring and Review
 3.6 Tools, Techniques, and Methodologies
4.0 Quality Assurance Application Areas
 4.1 Work Tasking and Authorization
 4.2 Configuration Management
 4.3 Testing
 4.4 Computer Program Design
 4.5 Computer Program Development
 4.6 Software Documentation
 4.7 Library Controls

**Software Quality Program Plan—Table of Contents**
(Reference DOD-STD-2168, DOD-STD-2167A)

# Acronyms

# Software Engineering Guidebook
## List of Acronyms

| | |
|---|---|
| ACA | After Contract Award |
| ACP | Age of Closed Problems |
| ANSI | American National Standards Institute |
| AOP | Age of Open Problems |
| ARO | After Receipt of Order |
| ATRR | Acceptance Test Readiness Review |
| BOE | Basis of Estimate |
| CASE | Computer-Aided Software Engineering |
| CC | Configuration Control |
| CCB | Configuration Control Board |
| CDR | Critical Design Review |
| CDRL | Control Data Requirements List |
| CFD | Customer Found Defects |
| CFP | Cost to Fix Post-lease Problems |
| CI | Configuration Item |
| CM | Configuration Management |
| CMCS | Configuration Management and Control System |
| cmi | continuous measurable improvement |
| COCOMO | Constructive Cost Model |
| COTR | Contracting Officer's Technical Representative |
| COTS | Commercial Off-The-Shelf |
| CPU | Central Processing Unit |
| CSC | Computer Software Component |
| CSU | Computer Software Unit |
| CTR | Contractor Task Report |
| CW | Code Walkthrough |
| DBMS | Data Base Management System |

| | |
|---|---|
| DFD | Data Flow Diagram |
| DID | Data Item Description |
| DoD | Department of Defense |
| DPC | Documentation Page Count |
| DR | Discrepancy Report |
| DT&E | Development, Test, and Evaluation |
| ECB | Engineering Control Board |
| ECR | Engineering Change Request |
| EEA | Effect Estimation Accuracy |
| ERD | Entity-Relationship Diagram |
| EST | Eastern Standard Time |
| FCA | Functional Configuration Audit |
| FQR | Formal Qualification Review |
| FR | Failure Rate |
| FSM | Finite State Machine |
| G&A | General and Administrative |
| GFE | Government Furnished Equipment |
| G/Q/M | Goal/Questions/Metric |
| GSFC | Goddard Space Flight Center |
| HITC | Hughes Information Technology Corporation |
| HMI | Human Machine Interface |
| HOL | High Order Language |
| HSTX | Hughes STX Corporation |
| IADS | Iceland Air Defense System |
| ICD | Interface Control Document |
| IDD | Interface Design Document |
| IEEE | Institute of Electronics and Electrical Engineering |
| I/O | Input/Output |
| IPF | In-Process Faults |
| IRS | Interface Requirements Specification |

| | |
|---|---|
| ITO | Independent Test Organization/Team |
| ITP | Integration Test Plan |
| ITRR | Integration Test Readiness Review |
| KAELOC | Thousand Assembly Equivalent Lines of Code |
| NASA | National Aeronautics and Space Administration |
| NOP | New Open Problems |
| ODC | Other Direct Cost |
| OOA | Object-Oriented Analysis |
| OOD | Object-Oriented Design |
| ORR | Operational Readiness Review |
| PDL | Program Design Language |
| PDR | Preliminary Design Review |
| PCA | Physical Configuration Audit |
| PR/CR | Problem Report/Change Request |
| RE | Risk Exposure |
| RRL | Risk Reduction Leverage |
| REVIC | Revised Intermediate Constructive Cost Model |
| RCS | Revision Control System |
| RFP | Request for Proposal |
| SCCB | Software Configuration Control Board |
| SCCS | Source Code Control System |
| SCM | Software Configuration Management |
| SCMP | Software Configuration Management Plan |
| SCR | Software Change Request |
| SDF | Software Development File |
| SDP | Software Development Plan |
| SDL | Software Development Library |
| SEA | Schedule Estimation Accuracy |
| SEI | Software Engineering Institute |
| SEL | Software Engineering Laboratory |

| | |
|---|---|
| SEPG | Software Engineering Process Group |
| SLOC | Source Lines of Code |
| SMG | Software Metrics Group |
| SMR | Software Modification Request |
| SOW | Statement of Work |
| SP | Software Productivity |
| SPMP | Software Project Management Plan |
| SPS | Software Product Specification |
| SQA | Software Quality Assurance |
| SRR | Software Requirements Review |
| SRS | Software Requirements Specifications |
| SSR | Software Specification Review |
| STD | State Transition Diagram |
| STD | Software Test Description |
| STP | Software Test Plan |
| STR | Software Test Report |
| SwEI | Software Excellence Initiative |
| TDCE | Total Defect Containment Effectiveness |
| TOP | Total Open Problems |
| TPD | To Be Done |
| TRD | Total Released Defects |
| TRR | Test Readiness Review |
| UDF | Unit Development File/Folder |
| UO | Unsatisfactory Outcome |
| V&V | Verification and Validation |
| VDD | Version Description Document |
| WBS | Work Breakdown Structure |
| WCP | Work Control Plan |

# Glossary

***Activity.*** A particular task (e.g., writing the software requirements, designing the software, reviewing the test results).

***Baseline.*** A work product that has been formally reviewed and agreed upon, which then serves as the basis for further development, and that can be changed only through formal change control procedures.

***Builds.*** A logical collection of software representing a predefined version of the system that contains part or all of the functionality of the entire system.

***Commercial Off-the-Shelf (COTS).*** A hardware or software item that is commercially available.

***Configuration Audit.*** The process of verifying that the current version of the Configuration Item (CI) agrees with the current version of the corresponding technical documentation, that the technical documentation accurately describes the CI, and that all Software Modification Requests (SMRs) have been resolved.

***Configuration Baseline.*** A specification or product that has been formally defined, documented, reviewed, and agreed upon at a specific time during the CI lifecycle. The baseline thereafter serves as the basis for further development and can be changed only through formal change control procedures. There are three formally designated configuration baselines in the lifecycle of a CI, namely, the functional, allocated, and product baselines.

***Configuration Control.*** The systematic proposal, justification, evaluation, coordination, and approval (or disapproval) of proposed changes and the implementation of all approved changes into the configuration of a CI after its configuration baseline(s) have been established.

***Configuration Control Board (CCB).*** A group that determines the type of problem (Engineering Change Request [ECR] or Program Trouble Report [PTR]) and the priority for resolving the problem (sometimes known as an Engineering Control Board [ECB]).

***Configuration Identification.*** The process of selecting CIs, determining the type of configuration documentation for each CI, and issuing numbers (and other identifiers) to each CI and its associated documentation.

***Configuration Management (CM).*** The function of selecting project baseline items, controlling the items and changes to them, and recording and reporting status and change activity for these items. Changes to these baseline items are controlled systematically using a defined change control process [Paulk et al., *Capability Maturity Model for Software, Version 1.1*, 1993].

***Configuration Status Accounting.*** The recording and reporting of the information needed to manage CIs effectively, including a listing of the identified CIs and the status of proposed changes to those CIs.

***Defects.*** Problems that are discovered after the review of the software development phase in which they were introduced.

***Documentation Page Count (DPC).*** The number of pages contained in a single copy of each of the documents produced. Page counts are to be gathered per document.

***Effort.*** Reporting of effort should be in labor hours so the data are easily transportable from project to project.

***Engineering Change Request (ECR).*** a) A requested engineering change and the documentation in which the change is described, justified, and submitted to the Government for approval (or disapproval). ECRs are required by the Government for postdelivery changes to contract deliverables. b) A requested change that requires a change in requirements as well as a change in software.

***Engineering Release.*** An action whereby a CI is officially made available for its intended use.

***Error Density.*** The count of errors or defects recorded, as a function of time, from the design phase through contract completion per 1,000 Source Lines of Code (SLOCs). It is expressed as a ratio of number of known unresolved errors over the total SLOCs, expected or actual, at completion times 1,000.

***Error Rate.*** The number of errors discovered each month.

***Errors.*** Problems that are created and discovered within the same software development phase.

***Evaluation.*** The process of determining whether an item or activity meets specified criteria.

***Faults.*** The combination of errors and defects.

***Formal Qualification Review (FQR).*** A formal review used to verify that the group of configured system components that compose the system complies with the hardware, software, and interface requirements. This review is often referred to as an Operational Readiness Review (ORR).

***Functional Configuration Audit (FCA).*** A formal audit used to verify that the configured system's actual performance complies with its hardware, software, and interface requirements. The verification can be demonstrated during the testing and reported during the FQR.

***In-Process Metric.*** Any metric that is collected and analyzed during the course of a project, which is then fed back to improve the process, product, or project during the life of the project [*Software Metrics for Process Improvement—Participant Guide,* Motorola University, April 1992].

***Phase.*** A period of time defined by management at or prior to project initiation through which the project is expected to pass (e.g., planning phase, requirements phase, design phase).

***Physical Configuration Audit (PCA).*** A formal audit of the "as-built" version of a configured system. The configuration system is compared against its design documentation to establish a product baseline. This is often a Quality Assurance (QA) activity.

***Program Trouble Report (PTR).*** The documentation of a software problem that may require a change in the software, but not in the requirements. This documentation also called the Software Change Request (SCR), Discrepancy Report (DR), or Problem Report/Change Request (PR/CR).

***Quality Assurance (QA).*** a) The function of reviewing and auditing software products and activities to ensure that they comply with applicable processes, guidelines, and procedures and providing the staff and managers with the results of the reviews or audits. (Adapted from the CMM.) b) A set of planned activities executed independently to ensure that the project team is developing software as described in its planning documents (e.g., Software Development Plan [SDP], Software Project Management Plan [SPMP], and Software Configuration Management Plan [SCMP]).

***Requirement.*** A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed document.

***Reviews:***

- ***Inspection.*** A formal meeting of several people to discuss the readiness of a software module for integration with other software modules as determined by preparatory source code analysis. Design or code inspections are the process of reviewing the detailed design or code produced with project personnel and peers.
- ***Walkthrough.*** An informal meeting between the programmer and at least one other appropriately knowledgeable person.

***Software Configuration Control Board.*** A board composed of technical and administrative representatives who recommend approval (or disapproval) of proposed software changes to a CI's current approved configuration. The board also recommends approval (or disapproval) of waivers and deviations from a CI's current approved configuration.

***Software Configuration Item.*** A unit of software identified for configuration control and treated as a single entity in the change control process. Software is typically identified hierarchically starting at the top (executable program level), with logical subgroups and modules at the bottom. These levels are sometimes referred to as the Computer System Configuration Item (CSCI), Computer System Component (CSC) and Computer System Unit (CSU), respectively.

***Software Configuration Management.*** A discipline applying technical and administrative direction and monitoring to identify and document the functional characteristics of CIs, control changes to those characteristics, record and report change processing and implementation status, and verify correlation of CI documentation to actual CI configuration.

***Software Development File (SDF).*** A file that contains the applicable requirements, design, code, and unit test information for a single software module.

***Software Lifecycle Process Model.*** A model depicting the phases through which a software system progresses, beginning when the product is conceived and ending when the product is retired. The model shows the relationships between the primary activities, baselines, deliverables, reviews, and milestones throughout the life of the system.

***Software Metric.*** A unit that enables us to quantitatively determine the extent to which a software process, product, or project possesses a certain attribute [*Software Metrics for Process Improvement—Participant Guide,* Motorola University, April 1992].

***Software Modification Request (SMR).*** Documentation of a problem with (or proposed enhancement to) software CIs. The SMR is used to track problem/enhancement analysis, problem correction/enhancement implementation, testing, baseline integration, and validation. The SMR is the vehicle for software CI change authorization. (The SMR is synonymous with a PTR or SCR.)

***Software Module.*** The same as a software unit (also referred to as a CSU).

***Software Problem.*** A discrepancy between a deliverable product of a phase of software development and any of the following: the product documentation, the product of an earlier phase, or the user requirements [*Software Metrics for Process Improvement—Participant Guide,* Motorola University, April 1992].

***Software Quality.*** The ability of a software product to satisfy its specified requirements.

***Software Subsystem.*** One or more modules (units) that are logically or functionally related. There may be one or more levels of subsystems in a system. In other words, a subsystem may contain other subsystems (also referred to as a CSC).

***Software System.*** Sometimes described as a "chunk" of software that is separately contracted for, specified, tested, and delivered. Each software system has its own requirements specification and system test (also referred to as a CSCI).

***Software Unit.*** The lowest level design entity that is implemented in the code (also referred to as a CSU).

***Source Line of Code (SLOC).*** A noncomment, nonblank line of written code defined as all source lines excluding blank lines and lines that contain only comments. The count for metrics reporting should be the number of new, modified, deleted, and total SLOCs.

***Staffing.*** The equivalent head count each month derived from the number of labor hours.

***Stakeholders.*** Individuals, or groups of individuals, who have a vested interest in the process, product, or project [*Software Metrics for Process Improvement—Participant Guide,* Motorola University, April 1992].

***Test Step.*** A numbered step in a documented test procedure.

***Thousand Assembly-Equivalent Lines of Code (KAELOC).*** A metric used to normalize the number of lines of code between different software languages. The source size is multiplied by a factor specific to each programming language [*Motorola Software Metrics Reference Document,* April 1991] (see Table 5.4-1).

***Threads.*** All software required to execute a process from system input through system response.

***Work Breakdown Structure (WBS).*** A decomposition of the work into a list of tasks, subtasks, and associated activities. The list is organized and numbered in a hierarchical manner.

# Comments and Feedback

The members of the SEPG would like to thank you in advance for taking the time to send us comments on the Software Engineering Guidebook. This Guidebook is the first of many steps to improve the software development processes used by HSTX to plan, develop and maintain software. This is a continuous process of improvement; your comments are the vitally important feedback needed to complete the cycle. Please bear in mind that both, positive and negative comments will be appreciated; we want to ensure that we do not undo what we are doing right when we rectify a problem.

Again, we would like to stress that this is a Guidebook, *not a Standard*. Its purpose is to serve as a guide to help your software development/maintenance process, not dictate how software should be developed and/or maintained. There are a multitude of ways to build and maintain software. No one way works best for all or even some projects or individual tasks.

The following are a re-iteration of the goals for the Guidebook:

- To foster an overall understanding, company-wide, of software engineering and the software life cycle,
- To provide useful software engineering information that is tailorable to individual projects/tasks,
- To provide an integrated approach to software engineering encompassing software development/maintenance activities, software support (i.e., quality assurance (QA) and configuration management (CM)) activities and software management activities,
- To provide software engineering information (e.g., life cycle models, development methodologies, checklists, tailoring guidelines) that is both useful and tailorable to every HSTX software development/maintenance project or task,
- To provide a single source of software engineering information that is both concise and easy to update with new information,
- To foster an engineering perspective and common language with which to discuss, plan, implement, manage, review and improve the variety of software and software processes used by HSTX employees,
- To serve as a foundation for HSTX software engineering training.

The following pages provide a list of issues that will be most useful to the developers of the Guidebook; your time in responding to these questions will be greatly appreciated. The questions have been divided into the following categories to help you organize your ideas, i) General Comments, ii) Content, and iii) Organization, Format and Presentation; however, you may provide your feedback in any format/organization you find most convenient. Please feel free to address issues that have not been identified in these questions.

Again, thank you for taking the time to comment on the Guidebook. Although all of your suggestions may not be included in the next version of the Guidebook, be assured that all comments will be considered. Please remember to provide us your name and related information so that we can respond to your suggestions. Remember to make a copy of the following pages before filling them out - this will allow you to reuse these pages for future comments.

## Software Engineering Guidebook Comments and Feedback

### Form General Information:

Name: ______________________ Date: ____________

Telephone: ______________________ Email: ____________

Project: ______________________
(Example: NSSDC, SES, etc.)

Task: ______________________

**Job Category** (circle one):

| | | |
|---|---|---|
| Associate Programmer Analyst | Associate Systems Engineer | Associate Engineer |
| **Programmer Analyst** | **Systems Engineer** | **Engineer** |
| Senior Programmer Analyst | Senior Systems Engineer | Senior Engineer |
| Principle Programmer Analyst | Principle Systems Engineer | Principle Engineer |
| Chief Programmer Analyst | Chief Systems Engineer | Chief Engineer |
| Associate Scientist | Associate Systems Programmer | Associate Technical Specialist |
| **Scientist** | **Systems Programmer** | **Technical Specialist** |
| Senior Scientist | Senior Systems Programmer | Senior Technical Specialist |
| Principle Scientist | Principle Systems Programmer | Principle Technical Specialist |
| Chief Scientist | Chief Systems Programmer | |
| Associate Technician | Associate Data Technician | Associate Administrator |
| **Technician** | **Data Technician** | **Administrator** |
| Senior Technician | Senior Data Technician | Senior Administrator |
| Associate Administrative Asst. | Associate Secretary | Associate Documentation Asst. |
| **Administrative Asst.** | **Secretary** | **Documentation Assistant** |
| Senior Administrative Asst. | Senior Secretary | Senior Documentation Asst. |
| | Executive Secretary | |
| Associate Clerk | Associate Publications Specialist | |
| **Clerk** | **Publications Specialist** | |
| Senior Clerk | Senior Publications Specialist | |

**Primary Function and Duties** (circle all that apply):

| | |
|---|---|
| Software engineer/developer | Task Member |
| Software Support (circle all that apply): | Technical Supervisor (Task Leader) |
| Configuration Management | Section manager |
| Quality Assurance | Department Manager |
| Software Manager | Program Manager |
| Other: | Other: |

## General Comments:

1. Did you find the Guidebook useful... (Complete all that apply)

a) ... as a Software developer/maintainer? (Yes/No):

b) ... as a Software project manager? (Yes/No):

c) ... as Software support staff (CM, QA)? (Yes/No):

2. Were you able to tailor the Guidebook to your specific project/task?

3. How would (did) you use the Guidebook?

a) In which software life cycle phases would (did) you use the Guidebook?

b) What information would (did) you use from the Guidebook?

4. Please rate the Guidebook as an on-the-job reference?

| 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|
| (not Good) | | | | (Good) |

5. What did you *like* about the Guidebook?

6. What did you *dislike* about the Guidebook?

**Content:**

1. Does it clearly explain the functions and inter-relationships of development/maintenance, software support (QA, CM) and software management functions? If not, what is missing?

2. What sections did you find *most* helpful?

3. What sections did you find *least* helpful?

4. List sections with *too much* detail?

5. List sections with *too little* detail?

6. What *additional* information would you like the Guidebook to cover?

## Organization, Format, and Presentation:

1. Is the material organized in a logical, intuitive, and useful manner? If not, what changes would improve it?

2. What about the organization, format or presentation of the material *enhanced* its usefulness?

3. What about the organization, format or presentation of the material *detracted* from its usefulness?

Additional Comments:

And, finally, in which area(s) would you like to help us improve the Guidebook?

**Please send your comments to:**

Pradip Sitaram
Hughes STX
Commerce I, Suite 400
Greenbelt, MD 20770
Tel#: (301)441-4184
email: sitaram@selsvr.stx.com

or

Temp Johnson
Hughes STX
Commerce I, Suite 400
Greenbelt, MD 20770
Tel#: (301)441-4171
email: tjohnson@ccmail.stx.com