

## **Chapter 11**

# **Understanding Software Development**

---

## Contents

<b>11.1 Software Development from the Acquirer View Point .....</b>	<b>11-4</b>
11.1.1 Requirements .....	11-4
11.1.1.1 Requirements Determination .....	11-4
11.1.1.2 Requirements Management .....	11-5
11.1.1.3 Prototyping .....	11-6
11.1.1.3.1 Prototyping Benefits .....	11-6
11.1.1.3.2 Cautions About Prototypes .....	11-7
11.1.2 Hardware .....	11-7
11.1.2.1 Hardware Requirements .....	11-7
11.1.2.2 Hardware Selection .....	11-8
11.1.3 Software Documentation .....	11-9
11.1.4 Project Planning .....	11-11
11.1.5 Solicitation .....	11-11
11.1.6 Project Tracking and Oversight .....	11-11
11.1.7 Acceptance Testing .....	11-11
11.1.7.1 Government Testing .....	11-11
11.1.8 AFOTEC Testing Objectives .....	11-12
11.1.8.1 Usability .....	11-12
11.1.8.2 Effectiveness .....	11-13
11.1.8.3 Software Maturity .....	11-13
11.1.9 AFOTEC Software Evaluation Tools .....	11-14
11.1.10 AFOTEC Lessons-Learned .....	11-14
<b>11.2 Software Development from the Supplier View Point .....</b>	<b>11-17</b>
11.2.1 Requirements Analysis .....	11-17
11.2.1.1 Analysis .....	11-17
11.2.1.2 Software Requirements Specification (SRS) .....	11-18
11.2.1.3 Interface Requirements Specification (IRS) .....	11-19
11.2.1.4 Prototyping .....	11-19
11.2.2 Project Planning .....	11-20
11.2.3 Test Planning .....	11-20
11.2.4 Preliminary Design .....	11-21
11.2.4.1 Design .....	11-21
11.2.5 Design Simplicity .....	11-22
11.2.6 Architectural Design .....	11-23
11.2.6.1 Preliminary Design Review (PDR) .....	11-25
11.2.6.2 Detailed Design .....	11-27
11.2.6.3 Functional Design .....	11-27

11.2.6.4 Data-Oriented Design .....	11-27
11.2.6.5 Object-Oriented Design .....	11-28
11.2.7 Problem Domains and Solution Domains .....	11-28
11.2.8 Critical Design Review (CDR) .....	11-29
11.2.9 Coding .....	11-31
11.2.10 Testing.....	11-31
11.2.10.1 Testing Objectives .....	11-32
11.2.10.1.1 Defect Detection and Removal .....	11-32
11.2.10.1.2 Defect Removal Strategies .....	11-34
11.2.10.2 Unit Testing .....	11-34
11.2.10.3 Integration Testing .....	11-36
11.2.10.4 System Testing .....	11-36
<b>11.3 Building Secure Software .....</b>	<b>11-36</b>
11.3.1 Security Planning.....	11-36
11.3.2. Operations Security (OPSEC).....	11-37
<b>11.4 The Bottom Line.....</b>	<b>11-41</b>
<b>11.5 References.....</b>	<b>11-42</b>

Software development, from the viewpoint of an acquisition organization, includes the following phases:

- requirements determination,
- project planning,
- solicitation,
- project tracking and oversight,
- acceptance testing.

The phases the supplier follows are similar:

- requirements analysis,
- project planning,
- preliminary design,
- test planning,
- detailed design,
- coding,
- unit testing,
- integration testing,
- system testing.

Software acquirers need to be familiar not only with their own process phases, but also with the supplier's phases.

**NOTE: IEEE/EIA 12207.0-1996, "IEEE Standard for Industry Implementation of International Standard ISO/IEC 12207: 1995 (ISO/IEA 12207) Standard for Information Technology – Software Life Cycle Processes" and IEEE/EIA 12207.1, "Guide for ISO/IEC 12207, Standard for Information Technology – Software life cycle processes – Life cycle data," were adopted on May 27, 1998 for use by the Department of Defense. They contain additional information regarding software development and should be reviewed in conjunction with this chapter.**

---

## 11.1 Software Development from the Acquirer View Point

---

### 11.1.1 Requirements

#### 11.1.1.1 Requirements Determination

Paul Paulson, president of Doyle, Dane and Bernbach, a large New York brokerage firm, was quoted in the *New York Times* as saying,

*"You can learn a lot from the client. Some 70% doesn't matter, but that 30% will kill you."* —  
Paul J. Paulson [PAULSON79]

The first task an acquisition organization must undertake is to determine the needs of the client or user. This is best accomplished by involving the user in defining the needs. One must take care to separate needs from wants. Needs are those items essential to mission accomplishment. Wants are the bells and whistles that, while making life easier, are not essential to mission accomplishment. The requirements should be conveyed to the developer in a manner that clearly emphasizes that the developer must satisfy all the needs, while potential satisfaction of wants will be evaluated using cost/benefit analysis. Requirements should be stated in terms of performance, i.e., what the system is to accomplish, not how the system is to accomplish the task. Requirements must be clearly documented and able to be implemented. They must also be well stated so they are easily understood by the designer and programmer. One measure of clarity is that requirements must be *testable*. If requirements are satisfied, you should be able to quantitatively test them. Classic examples of requirements that cannot be tested are those that state the system must be “*user friendly*” and provide “*rapid response*.” Such requirements are ambiguous to the designer, and are a potential source of endless arguments when the software is delivered.

### 11.1.1.2 Requirements Management

*Management of technical requirements is the most important, and often overlooked, software management issue.* Ideally, requirements should be fully identified before the statement of work (SOW) is written. In practice, this is almost never the case. In large, complex software-intensive systems, requirements continually evolve throughout the system’s life. Therefore, they must be constantly managed because they significantly impact total system development cost and schedule. “Requirements creep” often occurs on long procurements. The users have time to see the possibility of all the features they can have, or want changes to their original vision of the product. The operational environment changes or technology advances. The software contractor may want to accommodate the user, but through requirements creep, may lose control of the cost, schedule, and product. It is your job to hold the line on requirements. Failure to draw a line in the sand on user requirements can be fatal for your program. Freezing requirements through firm baselines is essential. It does not, however, make it impossible for the user to make changes. Evolutionary/incremental buildup of functionality is possible if it is planned and budgeted to occur at milestone decision points where requirements are re-baselined. Version control and tracking, including updating documentation, are other essential parts of the requirements configuration management task.

It is the acquisition organization’s responsibility to manage the requirements given to the contractor. The contractor should never be asked to implement requirement changes that have not been evaluated and approved by the acquisition organization’s configuration control board. This approval process includes requesting the contractor to provide cost and schedule impacts of the proposed change. If cost and/or schedule is negatively impacted, the contract may need to be modified. This process must also identify major stakeholders (or viewpoints) of the system. You must ensure that individual stakeholder needs are consistently collected, analyzed, and documented. A formalized process must also be used to make all stakeholders part of the requirements team.

Your contractor’s Software Development Plan (SDP) should address their understanding of the requirements stability issue, how well they will manage the requirements change process and evolutionary requirements. Your contractor’s management of requirements must stress a

commitment to an iterative process that utilizes structured requirements methods and appropriate tracking and analysis tools. Traceability from original, identified needs to their derived requirements, designs, and implementations must be assured. A two-way requirements traceability matrix should be used to ensure completeness and consistency of requirements among all levels of development (from top-level down to code-level). A requirements team includes multiple working groups, such as:

- An Operational Requirements Document Support Team,
- A Program Requirements Team, and
- An Operational Requirements Working Group.

### 11.1.1.3 Prototyping

Prototyping is a shortcut for demonstrating software functions/capabilities and for eliciting user buy-in. It is a *quick-and-dirty* way to evaluate whether the proposed design meets user needs and is generally produced with *throwaway code*. Prototypes are also not developed with supportability, readability, and usability in mind, and bypass normal configuration management, interface controls, technical documentation, and supportability requirements. Quality control and assurance (testing) and supportability issues (e.g., technical documentation) are seldom addressed, as these activities negate the benefits of the prototype.

#### 11.1.1.3.1 Prototyping Benefits

Major prototyping benefits include: clearer understanding of requirements, particularly if a user-interface prototype is demonstrated; quicker identification of design options and how they may be implemented into code; and resolution of high-risk technical issues in areas where the system may be pushing software state-of-the-art. Prototyping also has a high impact on a certain class of defects and can be used as an effective defect prevention technique. Although with large, complex software developments it is usually not possible to derive all the functional requirements up front, there is evidence that software developments using prototyping tend to reach functional stability quicker than those that do not. With prototyping, on the average, only 10% or fewer functions are added after the requirements phase; whereas, without prototyping 30% or more functions are added after requirements analysis. This leads to unanticipated and unfunded cost and schedule overruns. Also, defect correction costs associated with these late, rushed functions, exacerbate the problem, as they are more than twice as high as those made in earlier phases of development.

A pre-award prototype can be used to determine the offerer's understanding of the requirement, which in turn helps the offeror project a more realistic estimate of system development cost and schedule. While prototyping involves time and resources, *experience shows that the lead-time to a fully operational system is generally less when prototyping is used*. The prototype allows users and designers to know what is wanted, and having already built a simplified version, the fully developed system is less expensive and time-consuming. The final product is also more likely to perform as desired with fewer surprises when delivered.

You must make sure your contractor's SDP addresses coordinating prototype development with the system end-user to ensure a realistic requirements validation process occurs. As a minimum, the user must review and approve all prototypes of critical components. Reiteration of this

process is often necessary to include additional requirements analysis, specification, and validation if the prototyping exercise falls short of user expectations. The approval of the prototype(s) constitutes a baseline for system requirements to be incorporated in the Software Requirements Specification (SRS).

### 11.1.1.3.2 Cautions About Prototypes

Do not mistake a prototype for more than what it is — a shortcut for demonstrating proof-of-concept. Deming talked about the “*burn-and-scrape*” method of quality control for toast, comparing it to getting it right the first time. [DEMING82] The requirements for toast are certainly easier to understand than the requirements for most software systems, so some scraping is understandable. However, uncontrolled prototyping can result in an endless, unproductive sequence of *burn-and-scrape* developments. Because prototypes are not produced within normal development constraints, you must refrain from expanding a prototype without baselines, interfaces, capacity studies, and thorough documentation.

**NOTE: Refrain from forcing coding standards on prototype development as they can adversely impact the benefits of prototypes.**

Another caution about prototypes is they must be well planned and designed to address significant sources of risks you have thoroughly identified and documented in your Risk Management Plan. *You must make sure every effort has been made to understand the requirements before building any prototype, and then ensure that the prototyping effort is converging on a requirement(s) validation.* To benefit from the prototyping exercise, require that each prototyping effort concludes with the delivery of a written report stating what was done, the results, their implications, and the degree to which the prototype met stated objectives.

If supportability (or reliability, portability, interoperability, etc.) are high-risk drivers, these capabilities can be included in your functional description of offerors' prototype demonstrations. However be aware, without a sound software engineering process to back it up, source selection prototype demonstrations can be deceptive with false-positive results.

---

## 11.1.2 Hardware

A major concern during software development is how to get the most advanced computer hardware technology available. The goal is to get the most *crunch* (computer power) *-for-the-buck*. The question to be answered is how to have the most efficient, advanced equipment possible throughout the system life cycle. Because computer hardware improves at an exponential rate, and user requirements grow and change with technology's leading edge, hardware technology is a major source of change over the system's life.

### 11.1.2.1 Hardware Requirements

Computer hardware requirements definition (e.g., digital systems, digital line replacement units/modules, digital circuit cards, complex digital components) is often a considerable management challenge. The translation of hardware requirements through the design specification down to

gate-level schematics requires that designers work across a wide range of abstraction. *The rush to lock into hardware designs before completing essential tradeoffs is often a source of substantial program risk.*

### 11.1.2.2 Hardware Selection

Ideally, computer hardware selection should be delayed until completing sufficient requirements analysis and prototyping to predict the processing power and throughput necessary for successful execution of the planned software. Delaying hardware selection might be feasible if contract support is provided through a systems integrator on a cost-reimbursable basis, and flexibility is allowed in timing and selection. More importantly, selecting computer hardware late in the software development process encourages the development of portable software that can be easily migrated among different hardware platforms.

In reality, the recommended hardware is often not only part of the winning contractor's proposal, but an integral part of their cost estimate. If you receive a proposal keyed to a specific hardware set, this can be considered a *weak* proposal. Studies have shown that in major software-intensive systems acquisitions (e.g., weapons systems and command, control, communications and intelligence (C3I) systems) the cost of developing software can be as much as 80% of the cost of the hardware and software combined. If an offeror bases their cost estimates on a specific hardware set, they may not have a very good understanding of the proposed system.

**NOTE: Modifying a system performance requirement is not necessarily a bad thing or a sign of failure. Lessons-learned show that it is often not worth paying 30% more to get the last 5% of originally-specified performance. In software terms, it may be frivolous to spend another million dollars on hardware to reduce terminal response time from 2.1 seconds to the specified 2.0 seconds.**

Too often hardware acquisition is conducted separately from the software development process. In this case, the software effort cannot be delayed and completed out of context from its eventual operating environment. Hardware selection must proceed in concert with the software effort, which must be completed within hardware environmental constraints (e.g., centralized versus distributed environment, specific database management system, compilers, etc.).

**NOTE: In recent years, lessons-learned have shown that hardware is procured *too early*. Hardware sits around and waits for the software to be developed, and is effectively obsolete when finally implemented. Another common occurrence is that hardware is often budgeted too early. If hardware is not purchased within the fiscal period for which it was budgeted, the funds are removed from the program.**

Factors to consider in hardware selection are quantitative performance requirements (QPRs), especially if command, control, and communication (C3) requirements are being defined. To properly determine/simulate loading for a QPR measurement, an assessment of how the proposed software/hardware will perform together is essential. Operating system upgrades (projected by commercial-off-the-shelf (COTS) hardware vendors) must also be considered as they affect future system growth needs. There are three principles to follow in the initial stages of computer hardware selection (which also apply to software architecture design):

- Follow standards, either *de facto* or specifically defined,
- Follow an open systems architecture, and
- Plan for evolutionary change over the software life cycle.

In integrated airborne avionics environments, severe physical and connectivity constraints may exist. Nevertheless, every effort must be made to use standard computer hardware configurations with well-understood performance characteristics. Although not similarly affected by physical constraints, some intelligence systems, command and control (C2) systems, and MISs must operate with large existing suites of hardware and software. The technical and cost benefits/penalties of compatibility with these pre-existing systems must be assessed. Even when analysis indicates continuing a sole source, proprietary environment is cost-effective, DoD's preference is an open systems architecture.

**CAUTION! “Every vendor with an open mouth claims to have an open system.” [THOMPSON91] Unless vendors follow industry/government-approved standards, the system is not truly open. On the other hand, the considerable time it takes to develop and validate industry standards often leads vendors to use de facto standards. THE POINT IS TO SELECT SYSTEMS THAT ARE “COMPATIBLE” AND “INTERCHANGEABLE” WITH PRODUCTS FROM A WIDE VARIETY OF VENDORS!**

### 11.1.3 Software Documentation

Documentation must support the entire life cycle. Most importantly, it provides fundamental guidance for post deployment software support (PDSS). Documentation can be categorized as being either technical or managerial. Technical (or engineering) documentation is necessary as it records the engineering process and helps software engineers know where they are, what they have produced, and how it was done. It also helps maintainers and other engineers understand the code developed by others. Management documentation is that produced for the Government or the development organization to aid in monitoring the contractor's development progress in achieving program milestones and in fulfilling performance requirement specifications.

Although it often represents the foundation of a successful software development, documentation can also represent a significant source of cost and schedule risk. *Excessive paperwork will ruin software development.* Overloading your contractor with excessive documentation requirements takes away from engineering activities — costing valuable time and money.

Conversely, too few requirements for technical documentation may cause loss of program visibility and control. Design documentation that does not adequately reflect the delivered software's current state is worse than no documentation at all. It translates into high maintenance costs when attempts to enhance or upgrade the system are hampered by insufficient information on the delivered source code. Allocating operational functional requirements to configuration items should be both a management and a technical decision as it establishes the framework for collecting information on software requirements, design, code, and testing. *Shortcuts on maintaining/ updating technical documentation should be avoided at all costs.* Whenever the developer makes changes to data flow, the design architecture, module procedures, or any other software artifact, supporting technical documentation must be periodically updated to reflect those changes. This requirement must be clearly stated in the contract data requirements lists (CDRLs). No

matter how well your software product performs in fulfilling user requirements, if its supporting technical documentation is inadequate, your system is not a quality product. Without quality documentation the product can neither be adequately used nor maintained. Documentation (either in paper copy or electronic format) that confuses, overwhelms, bores, or generally irritates the users is of little or no value to you or your customers. [DENTON92]

Documentation is one of those activities that requires experience to determine a proper balance between too much and too little. Too little technical documentation can create the proverbial “*maintenance man’s nightmare*,” whereas, too much effort expended on producing unnecessary management documentation can waste precious development time and dollars.

Even where program management documentation is kept to a minimum, management and quality metrics reporting is essential and should be a contract requirement. Metrics reports describe the contractors’ progress against their plan. They reveal trends in current progress, give insights into early problem detection, indicate realism in plan adjustments, and can be used to forecast future progress. If not required, software developers are often reluctant to commit to paper their deficiencies and/or accomplishments. Your contractor may be agreeable to your suggestions and direction early in the game. However, as the development progresses and problems are encountered, this agreeability can deteriorate and the contractor may increasingly ask, “*Where in the contract (or other documentation) does it say the software has to do that?*” Once the honeymoon is over, the *documented word* (either in the contract, through delivered metrics documentation, or on-line access) has the most influence on contractor actions. By stressing the importance of metrics reporting early, you can avoid many problems later on.

From the user’s perspective, the software is only as good as the documentation (both written and interactive) that tells how to use it. Failure to include in the user’s documentation changes made to the executable software before delivery can have profoundly negative effects. For example, changes in the order or format of the interactive input to a management information system (MIS), if not documented, can cause significant problems through confusing error messages — or even system crashes.

Software documentation can provide a vehicle for *early user involvement* in the development process. User visibility is necessary to ensure that user requirements are addressed early, rather than added later at much greater expense. Specification and design documents give the user the opportunity to review requirements before the system is designed or coded. If user documentation is not available for user review until program completion, design changes resulting from that review can cause significant schedule slips and additional costs. Software requirements and designs must be clearly documented so they can be evaluated and deficiencies can be discovered and corrected. Remember, the best software design is of little value if it is incomprehensible to those who must translate it into code.

**NOTE: Touch-and-feel demonstrations are more effective mediums for user review of requirements than written specifications and design documents.**

Technical documentation should never be produced after the fact, nor for bureaucratic reasons. Like metrics, documentation should be an outgrowth of the normal development process, not an end in itself. It must be produced to capture the engineering process so that you and others can understand and benefit from what has occurred. Clear documentation prevents developers from getting lost in production activities, and helps maintainers in understanding what the software

does and how it was built. Documentation must be prepared throughout the development process to capture the results of each engineering activity. Documentation, used as a reference tool to aid new personnel, users, and maintainers in becoming familiar with the software product, must be kept up to date. If not kept current, it will impede operational and maintenance efforts resulting in a needless waste of time, effort, and money. (Robust Ada code with its narrative characteristics is almost self-documenting. However, the architecture and concept of operations must be clearly described.) *Online access to (or delivery of) technical documentation in electronic format saves development time and dollars.*

**CAUTION! The time and money saved by delaying or foregoing immediately needed documentation actually wastes time and money later in the program.**

---

## 11.1.4 Project Planning

Project planning is begun once requirements have been sufficiently identified and validated to establish an acquisition program. Planning (or replanning) continues throughout the life of the system. [See Chapter 7, *Acquisition Planning*].

---

## 11.1.5 Solicitation

Solicitation is the act of providing requirements to prospective contractors, evaluating their responses, and awarding a contract. [See Chapter 8, *Contracting for Success*].

---

## 11.1.6 Project Tracking and Oversight

Project tracking and oversight covers the activities of the acquisition organization from the time of project inception through project termination.

---

## 11.1.7 Acceptance Testing

### 11.1.7.1 Government Testing

Operational system testing of major Air Force software-intensive systems is conducted by an interdisciplinary, independent testing agency. The Air Force Operational Test and Evaluation Center (AFOTEC) is a separately operated agency that reports directly to the Chief of Staff of the Air Force. The Center is comprised of a headquarters at Kirtland AFB, New Mexico, detachments at operational locations, and AFOTEC test teams at designated test sites. AFOTEC plans, directs, controls, independently evaluates, and reports on the operational test and evaluation (OT&E) of all major Air Force weapon systems, weapon system support systems, C2, and MISs. It supports system development and production decisions by providing operational assessments and initial OT&E to determine operational effectiveness (how well the system performs) and suitability (including reliability, maintainability, and supportability). Table 11-1 lists the AFOTEC publications you should consult for guidance on software OT&E procedures.

PAMPHLET	TITLE
AFOTTECP 99-102, Volume 1	Management of Software Operational Test and Evaluation
AFOTTECP 99-102, Volume 2	Software Support Life Cycle Process Evaluation Guide
AFOTTECP 99-102, Volume 3	Software Maintainability Evaluator's Guide
AFOTTECP 99-102, Volume 4	Software Usability Evaluation Guide
AFOTTECP 99-102, Volume 5	Software Support Resources Evaluation Guide
AFOTTECP 99-102, Volume 6	Software Maturity Evaluation Guide
AFOTTECP 99-102, Volume 7	Software Reliability Evaluation Guide
AFOTTECP 99-102, Volume 8	Software Operational Assessment Guide

Table 11-1. AFOTTEC Software OT&amp;E Pamphlets

**NOTE: AFOTTECP 99-102 Volumes 2, 4, 5, 6, and 8 are available on the Defense Acquisition Deskbook CD and web site.**

## 11.1.8 AFOTTEC Testing Objectives

AFOTTEC cites six objectives in testing system software.

- Usability,
- Effectiveness,
- Software maturity,
- Reliability
- Safety and
- Supportability.

**NOTE: *Reliability* and *Safety* are discussed in Chapter 9, *Engineering Software-Intensive Systems*, and *Supportability* is discussed in Chapter 12, *Software Support*.**

### 11.1.8.1 Usability

Usability evaluations concentrate on the operator's interaction with a software-intensive system. Observation of test events should reveal strengths and limitations of the system's operator-machine interface and its supporting software. A usability questionnaire is used to assess the usability characteristics of conformability, controllability, workload suitability, descriptiveness, consistency, and simplicity. [See AFOTTECP 99-102, Volume 4, *Software Usability Evaluator's Guide*.]

### 11.1.8.2 Effectiveness

Effectiveness evaluations concentrate on ensuring all critical software is exercised in operationally representative scenarios. Software effectiveness is determined by (and dependent on) system effectiveness.

### 11.1.8.3 Software Maturity

Software maturity [as opposed to software development maturity discussed in Chapter 10, *Software Development Maturity*] is a measure of the software's evolution towards satisfying all documented user requirements, as illustrated in Figure 11-1. [Refer to AFOTTECP 99-102, Volume 6, *Software Maturity Evaluation Guide*.] The main AFOTTEC indicator of software maturity is the trend in accumulated software changes to correct deficiencies, provide modifications, and accommodate hardware changes. The software maturity test objective considers software fault trends, severity, and test completeness while taking into account planned software modifications.

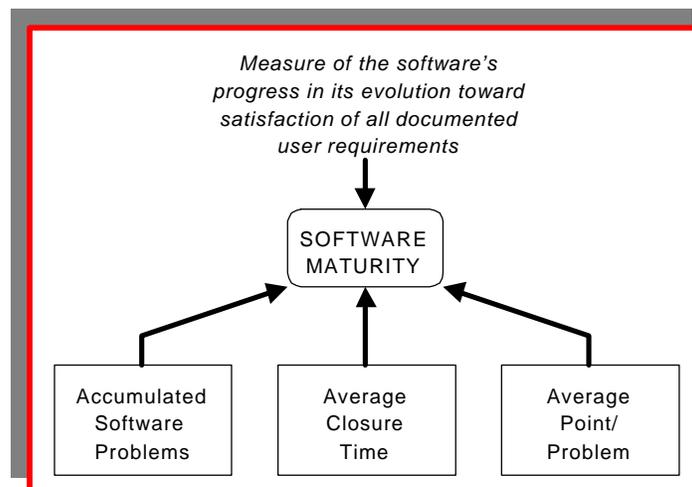


Figure 11-1. OT&E Process for Software Maturity

Software maturity uses a severity point system to track unique problems. A weighted value is assigned based on the severity of the failure as defined in the Data Management Plan. Software faults of higher severity are assigned a higher value than those of less severity. As the test progresses and new fault data are collected, they are plotted against a time line. Ideally, the slope of the curve should decrease with time. This maturity assessment method is illustrated in Figure 11-2.

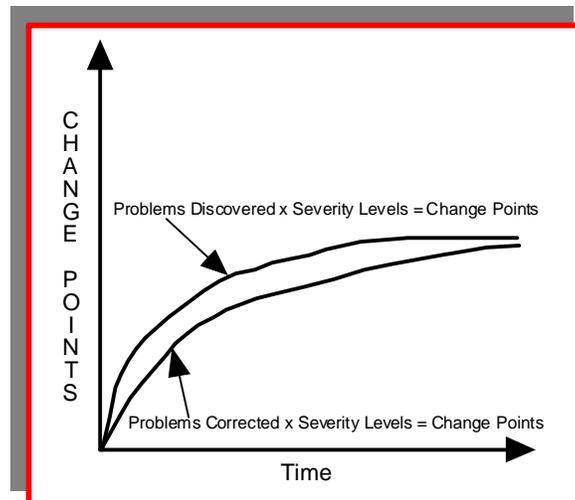


Figure 11-2. Software Maturity

### 11.1.9 AFOTEC Software Evaluation Tools

The AFOTEC software evaluation tools [AFOTEC 99-102, Volumes 1-8] should be used throughout the acquisition and development phases of major systems software. They are based on COTS software metrics, ensure credible evaluations, and help to reduce life cycle costs and schedule. Software evaluation approaches differ among programs; however, the AFOTEC mission is to evaluate software as an integral part of the overall system (as opposed to evaluating it as a separate entity). It uses the same fundamental OT&E processes for MIS and embedded weapon systems software based on the premise that *if the system works, the software works!*

**NOTE: To be effective, software operational test planning must take place throughout the development process. Often, the developer and the SPO are reluctant to provide the operational tester with the documentation and materials needed to perform an effective evaluation of software maturity, reliability, supportability, usability, and effectiveness. Achieving cooperation among the system developer, the SPO, and the operational tester is an essential management prerequisite.**

### 11.1.10 AFOTEC Lessons-Learned

AFOTEC has provided a list of lessons-learned based on the experiences of programs having completed the OT&E process.

- **Deputy for Software Evaluation (DSE).** A DSE should be assigned as early as possible to the SPO to become familiar with the system and to assist in detailed software OT&E planning. The DSE should be onboard at least 6 months prior to the first OT&E test event. (Larger programs may require even more lead-time.) The DSE, a software systems engineer, serves as the software evaluation team leader, is assigned to the test program, and coordinates and controls the completion of OT&E test plan objectives pertaining to software and software support resources.
- **Documentation.** The DSE and software evaluators must be provided current documentation and source code listings in time to perform evaluations. Promised deliveries not received can cause problems; therefore, it is to everyone's benefit to deliver requested documentation on time. You must also identify early the requirement for special sorties, equipment, and analysis support so that test requirements are accommodated.
- **Testing terminology.** Your team must be conversant with terminology and definitions (software faults, defects, errors, bugs, etc.). Industry-accepted definitions of software errors and defects (faults), found in the ANSI/IEEE, *Glossary of Software Engineering Terminology*, are listed in Table 11-2. [IEEE90] A failure is an observable event (an effect). A fault is the cause of a failure. A fault may be caused by software or hardware. A software defect is the human-created element that caused the fault, such as those found in specifications, designs, or code. The bottom line with AFOTEC is, *if an action is required of the operator due to a failure — it must be documented.*

Category	Definition
Error	<ul style="list-style-type: none"> <li>The difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition</li> </ul>
Fault	<ul style="list-style-type: none"> <li>An incorrect step, process, or data definition in a computer program</li> </ul>
Debug	<ul style="list-style-type: none"> <li>To detect, locate, and correct faults in a computer program.</li> </ul>
Failure	<ul style="list-style-type: none"> <li>The inability of a system or component to perform its required functions within specified performance requirements. It is manifested as a fault.</li> </ul>
Testing	<ul style="list-style-type: none"> <li>The process of analyzing a software item to detect the differences between existing and required conditions (that is, bugs) and to evaluate the features of the software items.</li> </ul>
Dynamic analysis	<ul style="list-style-type: none"> <li>The process of evaluating a system or component based on its behavior during execution.</li> </ul>
Static analysis	<ul style="list-style-type: none"> <li>The process of evaluating a system or component based on its form, structure, content, or documentation.</li> </ul>
Correctness	<ul style="list-style-type: none"> <li>The degree to which a system or component is free from faults in its specification, design, and implementation</li> <li>The degree to which software, documentation, or other items meet specified requirements</li> <li>The degree to which software, documentation, or other items meet user needs and expectations, whether specified or not.</li> </ul>
Verification	<ul style="list-style-type: none"> <li>The process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase.</li> <li>Formal proof of program correctness.</li> </ul>
Validation	<ul style="list-style-type: none"> <li>The process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements.</li> </ul>

Table 11-2. IEEE Software Engineering Terminology [IEEE90]

- Final test reports.** Striving for correct technical content, software test teams often write final test reports but are frustrated when OT&E headquarters personnel rewrite them for format and content. Time constraints and pride-of-authorship can strain tensions between the test team and headquarters. One solution is to have the two teams work together to review final report drafts early in the process. Also make sure that the report is written for a wide spectrum of readers, *computerese* is kept to a minimum, and that it is tailored for a senior officer audience with emphasis on results and recommendations.

**NOTE: “Timing” is more important in real-time systems than in any other software development. The Software Engineering Institute (SEI) has developed a method, Rate Monotonic Analysis (RMA), for managing the scheduling and execution of tasks for real-time systems.**

---

## 11.2 Software Development from the Supplier View Point

---

### 11.2.1 Requirements Analysis

#### 11.2.1.1 Analysis

The development team must approach the requirements analysis task with strong leadership that emphasizes risk reduction through evolutionary development and prototyping to ensure quality issues are translated into functional requirements. The software system must, in addition, be analyzed within its *environmental framework*. This analysis may be performed in accordance with one, or several, structured analysis techniques (such as functional decomposition, hierarchy diagrams, object-oriented analysis, data flow analysis, or state transition charts). Methods include: object-oriented (data-oriented) [COAD90], process-oriented (functional or structured analysis) [YOURDON90], and behavior-oriented (temporal, state-oriented, or dynamic; e.g., essential systems analysis) [McMENAMIN84]. Each of these techniques view the system being developed from a different perspective.

The approach selected by the development team depends on the type of software system being defined, and the approach that most clearly states the problem. Further analysis involving user scenarios, transaction modeling, performance modeling, and consistency checking among viewpoints must also be performed. This ensures overall requirements consistency. Requirements so derived must then be validated with the users prior to development to guarantee that the system can, and will in fact, be built. Validation approaches include performance modeling and prototyping of those software components deemed critical to software success. Another good litmus test for the validity of a requirements package, used on the F-22 Program, is to check whether designers from two different development team organizations have identical understanding of a set of bottom-level elements in the requirements hierarchy. If not, your team process is flawed, and the chances the pieces produced by various team members will integrate smoothly is close to zero.

Many techniques have been developed to assist in specifying and documenting requirements, such as integrated computer-aided manufacturing definition language (IDEF) and computer-aided software engineering (CASE) tools. Whatever the tools or methods used, the analysis should include a basic series of requirements activities.

- If requirements are uncertain, build a prototype, or model the information domain,
- Create a behavioral model that defines the process and control specializations,
- Define performance, constraints, and validation criteria,
- The Software Requirements Specification must be written or depicted, and
- Conduct regular formal technical reviews. [PRESSMAN93]

Figure 11-3 illustrates the requirements definition and analysis process performed for the F-22. A joint relationship among all stakeholders must continue throughout development. Eventually,

this effort will result in documentation or data that directly *cross-references test cases to requirements and code*. At the same time, developers and testers should independently plan, design, develop, inspect, execute, and analyze software test results.

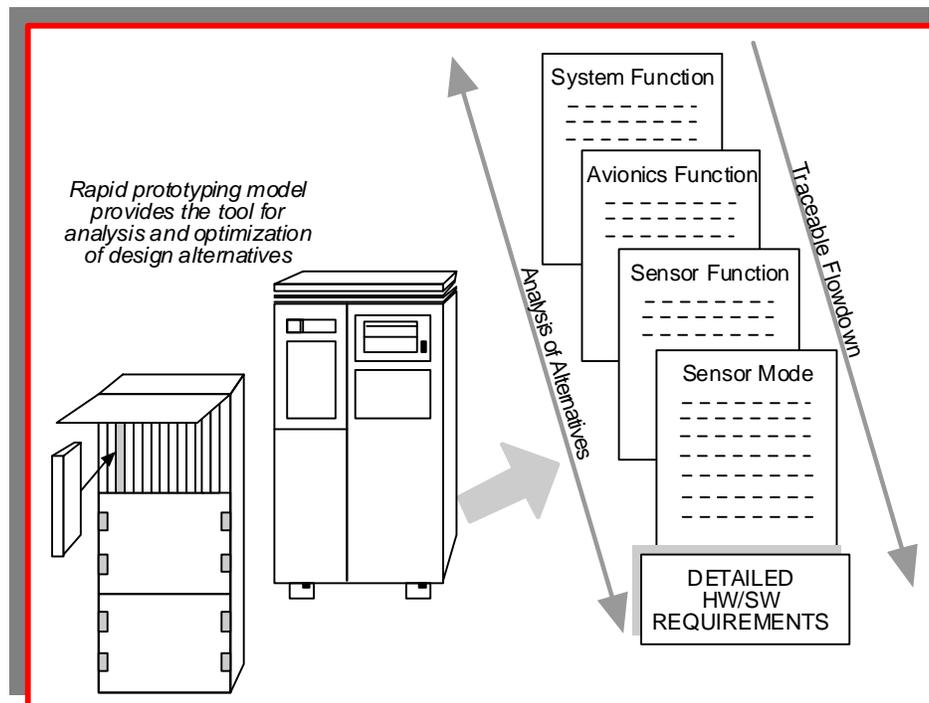


Figure 11-3. F-22 Requirements Process

### 11.2.1.2 Software Requirements Specification (SRS)

The successful completion of the requirements phase results in a Software Requirements Specification (SRS). Common sense must be used when writing these specifications so that they are realistic, achievable, and not just *bells-and-whistles*. It must be kept simple and short. Quality attributes should be defined such that the designer knows how to go about achieving them, and the user knows whether they have been achieved when the software is delivered. [GLASS92] Remember, *quality must be testable and measurable*. To achieve this, there must be an open, honest, and cooperative free exchange of information between the Government and the developer (contractor) as reflected in the SRS.

The specification process is one of *representation*. Requirements must be represented in such a way as to facilitate their successful implementation into software. The characteristics of a good specification are:

- Functionality is separated from the implementation. Specifications must be expressed entirely in the “*what*” form, rather than the “*how*.”
- The specification language must be process-oriented — the process to be automated and the environment in which it is to function and interact must be defined.
- The specification must describe the software within the context of the entire system.
- The specification must be an empirical representation, rather than a design or implementation representation, of the system.

- A specification must be comprehensive and formal enough to determine if the implementation fulfills randomly selected test case requirements.
- The specification must be flexible, enhanceable, and never totally complete.
- The specification must be localized, loosely coupled, and dynamic. [PRESSMAN93]

Dobbins claims that as long as developers insist on writing software requirements in prose form, requirements will continue to be the source of expensive software defects. He recommends the acquisition and use of one or more of the emerging specification generation techniques, many of which require the use of CASE tools. The tools selected should be based on ease-of-use and the ability to perform comprehensive real-time analysis and evaluation of the requirements package as it is being developed. [DOBBINS92]

### 11.2.1.3 Interface Requirements Specification (IRS)

Never lose sight of the fact that hardware and software development are intimately related. Although they are developed in unison, for major programs, software is always on the system's critical path. Early consideration of how the software is to interface with the system and other software is necessary to achieve the benefits of cohesive, interoperable systems. Proper integration of hardware and software can be assured through carefully identified interface requirements and prudently planned reviews, audits, and peer inspections. Such systems provide improved accuracy, currency, and quality. Early identification of integration and interface requirements also prevents redundancy.

Software interface requirements are documented in the IRS. In complex system developments, with multiple developers, each contractor must have a baselined IRS to ensure interface discipline. There must also be a requirement that each contractor's software system interface with other designated systems. Otherwise, each contractor can change their interface at will, affecting other contractors' efforts. Not baselining Interface Control Documents (ICDs) also gives contractors a mechanism to shift the schedule continuously. While ICD changes can lead to additional expense, uncontrolled change is even more dangerous. A more acceptable method is to develop to a given version of an ICD while still having the contractor maintain and update that ICD. The contractor then assumes the responsibility of maintaining current ICDs and of meeting the requirement. Program milestones should be used to determine which ICD is being used to develop any given phase of the system.

**NOTE: Refer to Requirements Determination Process by EDS.**

### 11.2.1.4 Prototyping

Prototyping, along with a structured analysis process and performance modeling, is an effective means to evolve and clarify user expectations. They can be used to resolve conflicts among cost, schedule, performance, and supportability; to ensure users and developers have the same understanding; and to validate requirements. Prototypes provides a better way for resolving the statement, "*I'll know it when I see it,*" than documenting requirements in English, with all its ambiguities. Prototypes, which include the results from rapid prototyping techniques, executable models, and quantitative descriptions of behavior (such as structured prototyping languages or graphical representations), are powerful tools for deriving correct hardware/software partitioning,

for performance testing, and for eliminating significant sources of risk. Remember, *prototypes must be useful, not just demonstrations or models of the system.*

Prototyping involves the early development and exercise of critical software components (e.g., user interfaces, network operating systems, resource managers, and key algorithm processors). They are comprised of the user interface, its interaction details with the proposed system, and executable functional models of critical algorithms. They are different from demonstration systems that provide usable evolutionary increments. Normally, in MIS and C3 systems, prototypes demonstrate screens and limited functions — *not actual software that works!*

One method for developing a prototype to is build it from reusable components. Because the components already exist, a prototype built from reusable parts is the easiest, cheapest, and quickest to build. It can provide rapid functionality since it is built from previously coded and tested components. Another way to build a working prototype is through a tool such as UNAS, which can generate a demonstrable level of functionality in Ada code. An ability to *plug-in, plug-out* COTS products can also greatly speed up the prototyping process. The least desirable prototyping approach is one which uses HOLs and/or rapid prototyping tools that only build a *quick-and-dirty* skeleton of the system. While the external facade (e.g., front-end screens with no code behind them) may give the user a *touch-and-feel* for what the final system will be like, there is nothing behind that front-end prototype (i.e., no functionality that the user can execute to determine if it will do something useful). [YOURDON92]

---

## 11.2.2 Project Planning

Project planning begins during requirements analysis, but the majority of project planning cannot be accomplished until the requirements are understood. A thorough understanding of the requirements leads to better estimates of the skills, effort, and time required to satisfy the requirements. Project planning is never really finished. A software development plan and project plan may be developed early in the project, but they should be reviewed and updated as required throughout the project life cycle.

---

## 11.2.3 Test Planning

A description of requirements tests (or measures) must be included in the software requirements specification. Testing must demonstrate that, if successfully completed, the delivered software will satisfy the requirement. The need for testable requirements demands that testing issues are addressed early in the program. Software test personnel (in addition to the developers, users, and maintainers) must take an active role in the requirements definition, analysis, and software design phases. The formal assessment of quality objectives should be an integral part of this effort. Unless a user need is correctly and completely stated, it is unlikely that either quality code will be written or a test can be performed to determine if the software satisfies the need and quality requirements.

---

## 11.2.4 Preliminary Design

### 11.2.4.1 Design

The importance of software design can be stated simply: *design is where the quality goes in*. This is the critical activity where your choice of a developer pays off. Skills, experience, ingenuity, and a professional commitment to process improvement and excellence are necessary to ensure your product has quality built-in.

Software design is the pad from which development and maintenance activities are launched. *Software design is the process through which requirements are mapped to the software architecture*. Design is also divided into two phases so architecture and requirements allocations are in place before components are detailed. Partitioning the process into two (or more) phases provides the Government with an opportunity to formally review the design as it evolves [e.g., Preliminary Design Review (PDR) and Critical Design Review (CDR)]. Remedial actions can be taken before the design becomes too detailed. The two-phase process also gives you a chance to subject the high-level design to external review (e.g., by systems and hardware engineering team members). This ensures compatibility with other system software and hardware with which the software must interact.

The architectural design defines the highest-level relationship among major software structural components, representing a holistic view of the system. Refinement of the architecture gives top-level detail, leading to an architectural (preliminary) design representation where computer software units (CSUs) are identified. Further refinement produces a detailed design representation of the software, very close to the final source code. [Bottom-up design is this process in reverse.] Detailed design involves refinements of the architecture CSUs leading to algorithmic representations, controls, and data structures for each architectural component. It may be possible to produce poor code from a good design — but seldom is it possible to produce good code from a poor design. Design is the “*make-it-or-break-it*” phase of software development. [GLASS92]

Within the context of architectural and detailed design, a number of activities occur. All the information gathered and analyzed during requirements definition flows into the design activities. The software requirements expressed in the form of information, functional, and behavioral models are synthesized into the design. The design effort produces an architectural design and a detailed design (comprised of a procedural design, a data design, and an interface design). The procedural design translates structural components into a procedural representation of the software. The data design transforms the domain model (created during requirements definition) into the data structures required to implement the software. Interface design not only defines how the software is to interface with other system software and hardware, but with the human-machine interface. Figure 11-4 illustrates how information about the software product, defined during requirements analysis, flows into the design which in turn flows into the coding and testing phases.

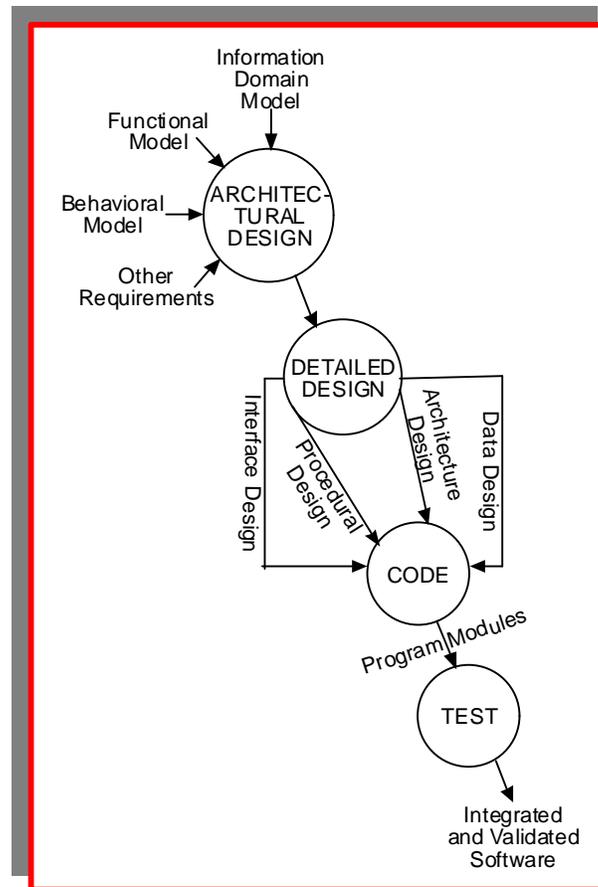


Figure 11-4. Ingredients of Software Design

For software to achieve a high degree of excellence, it must be defect free; i.e., reliable. Adding the dimension of reliability to the quality equation (especially for weapon system software) translates into a design requirement for fault-tolerance. Software designed to be fault-tolerant possesses the following characteristics:

- Defect confinement. Software must be designed so that when a defect occurs, it cannot contaminate other parts of the application beyond the module where it occurred.
- Error detection. Software must be designed so that it tests itself for, and reacts to, errors when they occur.
- Error recovery. Software must be designed so that after a defect is detected, sufficient internal corrective actions are taken to allow successful execution to continue.
- Design diversity. Software and its data must be designed so fallback versions are accessible following defect detection and recovery. [GLASS92]

## 11.2.5 Design Simplicity

When your developer decomposes the solution from the high-level design to lower-levels of detail, it must be kept *simple*. Whatever design methods employed, the underlying issue is to keep it within human limitations for managing complexity. Since automatic software design has still to evolve as a practical reality, you must apply sound engineering discipline to support your

quality design goals. Thus, the design solution must be broken down into *intellectually manageable pieces* (modules). Ideally, modules should contain no more than 100-200 lines-of-code. Remember,

Throughout the design process, the quality of your developer's evolving design should be assessed through a series of informal in-house walkthroughs, formal technical reviews, and peer inspections. To evaluate design quality, *the criteria for a good design must be established in the SRS*. Applying software engineering to the design process encourages the use of fundamental quality design principles, a systematic methodology, and thorough review of the connections between modules and with the external environment.

During design, the skills of your software developer are put to the acid test. Quality software must achieve the goals of software engineering by fulfilling the quantifiable principles of well-engineered. To refresh your memory, these include:

- Abstraction and information hiding,
- Modularity and localization, and
- Uniformity, completeness, and confirmability.

---

## 11.2.6 Architectural Design

A good software architecture should reflect technical certainties and be independent of variants, such as performance, cost, and the specific hardware selection. It must also address higher-level concepts and abstractions. Lower-level details are dealt with during the detailed design phase, which defines the particular modules built under the architecture at the software engineering level. By defining only essentials (or certainties), rather than incidentals (or variants), a good architecture provides for the evolution of the system and for the incremental or evolutionary upgrading of components. A sound approach for the software architect is to address (and to commit to) certain essentials and to be independent of variable incidentals. *The hallmark of a good architecture is the extent to which it allows freedom and flexibility for its implementers.*

Architectures must address the relationships among system components (i.e., the interfaces between them). Standardization of data interfaces, their implementation, access, and communication improves the quality and consistency of data and the overall effectiveness of the system. Data and system interfaces for MIS and C3 systems should be compliant with DISA's Technical Architecture Framework for Information Management (TAFIM). A standards-based architecture reflects a managed environment (based on defined standard interfaces) that describes the characteristics of each architectural component. It is depicted through classes of architectural platforms that are, by definition, modular, highly reusable, and inherently flexible. It provides a high degree of interoperability in that the architecture is owned by the user — not the vendor. [DMR91]

In building a standards-based architecture you should also make sure your software architecture is built using *lateral vision* (i.e., from an agency, command, and user perspective). Once standards-based architectures are built, they must then be integrated into reuse. A rule of thumb is to use the standards that are out there; e.g., Government Open Systems Interconnect Profile (GOSIP), Portable Operating System Interface for Unix Computer Environments (POSIX), Structure Query

Language (SQL), etc. You should also ensure that your developer makes sound decisions about user interface standards. For MISs, a framework should be picked from the TAFIM. The “*Command Center Store*” of Electronic Systems Command provides a generic architecture and reusable components common to many C2 systems.

A user interface, such as OSF/Motif for XWindows, should also be considered. Client/server roles, a migration strategy application (or model layer), and binary application portability with purchased software are also important factors. National Institute of Standards and Technology (NIST), Federal Information Processing Standard (FIPS), and commercial standards should be used, when appropriate. Figure 11-5 illustrates the standards-based architecture planning process. [DMR91]

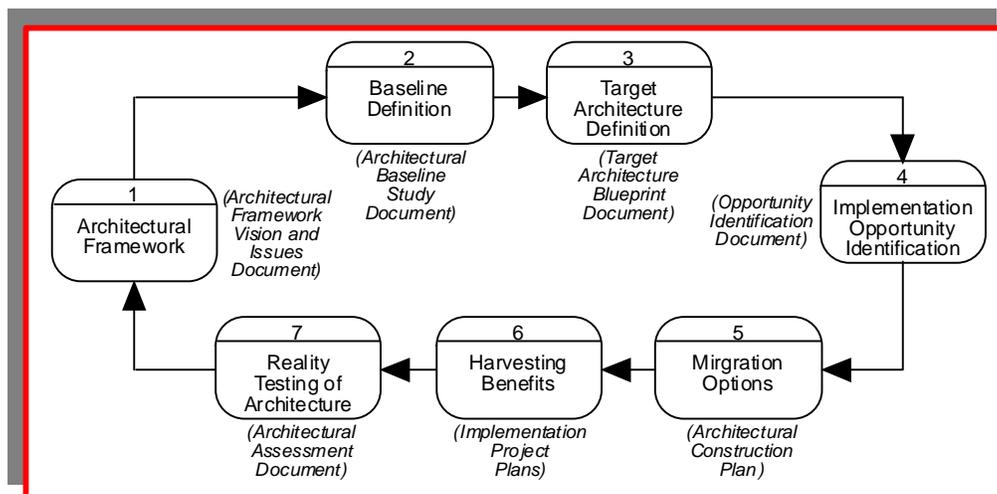


Figure 11-5. Standards-Based Architecture Planning Process [DMR91]

The baselined architecture and plans for the system’s evolution impact your developing application in significant and important ways. *You must pay close attention to the architectural design process because it is critical to the success of your program.* Management considerations include:

- Creating a clear vision of requirements and functionality early,
- Relentlessly eliminating unnecessary complexity from systems and software, and
- Careful control of requirements.

During architectural design, requirements are allocated to a number of components, such as objects or functional requirements. Derived requirements are also defined and allocated. Derived requirements may reflect the need to reuse existing components, to design components for reuse, to take advantage of available COTS software, or other factors such as security and safety design constraints. While not originally stated in the requirements specification, derived requirements impact on quality and performance and must be reflected as functions (or objects), mapped from the SRS to the architecture.

The product of the architectural design phase is the software architecture. The architecture reflects two characteristics of the software solution: (1) the hierarchical structure of procedural components and objects (modules), and (2) the structure of data. It evolves from partitioning related elements of the software solution. To achieve openness to COTS, government-off-the-

shelf (GOTS), and non-developmental item (NDI) solutions, two forms of system partitioning should be achieved. First, systems interfaces, functionality, and data schema must be partitioned within the software architecture such that there are no barriers to the inclusion of the best available technology which requires an awareness of available technology and probable technological progress. Second, the architecture must be partitioned such that those modules that will not change are divorced from the path of those modules slated for evolutionary improvements. In addition, the requirement for an open systems architecture requires that designers possess an early and knowledgeable awareness of market evolution (and indeed revolution) in the burgeoning software technology arena. The architecture should reflect a tradeoff between needs and currently available technology, whereas interfaces must be designed such that a change within one element of the architecture has minimal impact on other elements.

These related software elements were derived from related elements of the real-world problem domain, implicitly defined during requirements analysis. The architecture also defines the underlying software structure. It gives a representation of how the software is partitioned (not allocated) into components and their interconnection. It provides a picture of the flow of data (including database schema, message definitions, input and output parameters, etc.) and the flow of control (including execution sequencing, synchronization relationships, critical timing constraints or events, and error handling). It outlines the rules for integration of system components which involves timing and throughput performance attributes and interconnection layers, standards, and protocols. The architecture also distinguishes between hardware structure and software structure, and provides for the future allocation of software components to hardware components. [PAYTON92]

**NOTE: See Volume 2, Appendix F, Tab 1, “The Importance of Architecture in DoD Software,” and Tab 2, “A New Process for Acquiring Software Architecture.”**

In unprecedented, major software-intensive systems, certain software components often must be custom designed to meet specialized requirements. CASE tools should be used to partition and layer the architecture to isolate those functions which are necessarily unique. One of the biggest issues in integrating uniquely developed software with multiple COTS software packages is interoperability. A solution to this problem is the introduction of a “*middle layer*” of software that isolates the interface between the unique infrastructure and the COTS. The advantage to using such a middle layer is that it encapsulates unique protocols so new COTS products can be *plugged-in* as they become available. Mission-unique software must be designed such that the components of the software infrastructure are transparent to the application code. An analogy to the software middle layer can be made to adapting an electrical appliance to a wall plug. If your radio has a 3-prong plug but your wall circuit only accepts a 2-prong, you can place a 3-prong/2-prong adapter between the two without having to modify either the wall circuit or the radio.

### 11.2.6.1 Preliminary Design Review (PDR)

The PDR is a formal government/contractor review conducted for each computer software configuration item (CSCI) to verify that the top-level software architecture design is consistent with defined software requirements and suitable for the detailed design. The following topics are covered during the PDR:

- The overall software structure to the computer software component (CSC) but not in all cases to the lowest unit level in the software hierarchy [*structure charts are one method for depicting the software architecture*];
- Functional flow showing how SRS allocated requirements are accomplished.
- Control function descriptions explaining how the executive control of CSCs will be accomplished. Start, recovery, and shutdown features are described for each major function or operating mode.
- Storage allocation, timing, and resource utilization information describing how the loading, response, and scheduling requirements will be met by the selected digital hardware and the software design.
- Software development facilities and tools that will be used for the detailed design, coding, and testing of the software. These facilities and tools include compilers, simulators, data reduction software, diagnostic software, a host computer, and test benches.
- Plans for software testing, with emphasis on integrating CSCs in a phased manner. In particular, top-down, bottom-up, or combination strategies are considered, and an effective strategy for the hierarchical software design selected.
- Human engineering of software-controlled control and display functions. Preliminary versions of user's manuals are reviewed to verify that human factor and training considerations are correctly addressed.

The contractor should answer following questions at the Preliminary Design Review (PDR):

- What is the software design structure, the resulting major input/output flows, and the relationships between CSCs?
- Is the overall software structure consistent with a structured, top-down, object or other design and implementation concept?
- Are all common functions identified, and units or subroutines designed to implement these functions?
- Is the interface between CSCs and the operating system or executive clearly defined? Are the methods for invoking each CSC's execution described?
- Has a CSC been designed to satisfy every system requirement?
- Is the traceability relating each CSC to specific software requirements documented?
- Is software being designed in a manner that provides for ease of modification as planned for in the SDP?
- How will the software be integrated with the hardware during full-scale engineering development?
- When will the system and software designs be baselined?
- Are sufficient memory and timing growth capacity being incorporated in the system and software design?
- How will software testing be performed? What levels of testing will be employed? Will an independent analysis and evaluation be accomplished?
- How will testing be used to clearly identify deficiencies as either software or hardware related? How will it be determined if errors/defects are caused by either the hardware or software? How will regression testing be performed?
- How will the software be supported in the field? What hardware and software will be needed for the support base? How will it be procured?

### 11.2.6.2 Detailed Design

The detailed design is a description of how to logically fulfill allocated requirements. The level of detail in the design must be such that software coding can be accomplished by someone other than the designer. The design of each functional unit (module) is performed based on the software requirements specification and the software test plan. The unit's function, its inputs and outputs, plus any constraints (such as memory size or response time) are defined. The detailed design specifies the logical, static, and dynamic relationships among units. It also describes module and system integration test specifications and procedures.

Software engineering techniques can then be used to evaluate and make tradeoffs among the different approaches, which are eventually narrowed down to an optimum solution. As the exploratory process proceeds, the design process becomes more formal. From a quality perspective, the design approach used by your developer must be determined by the nature of the application problem. Your design architecture might be based on functions, data, objects, or a combination thereof.

### 11.2.6.3 Functional Design

For heavily logic-oriented applications (such as real-time systems) where the problem involves algorithms and logic, a function-oriented approach is often used. Function-oriented design depicts information (data and control) flow and content, and partitions the system into functions and behaviors. From the single highest-level system description, the system is partitioned into functions and sub-functions with ever increasing levels of detail. The value of a function-oriented design is that it provides manageable levels of complexity at each phase of the design process. It also identifies and separates the functions performed at each phase of application execution. However, this hierarchical decomposition by function leaves the question as to what is the most abstract description of the system.

The design focuses on those requirements most likely to change (i.e., around functionality). But, if the specification is poorly written, designers are faced with the problem of having to deal with a top-down design for which they are unable to locate the top. Another problem with hierarchical methods is, as decomposition occurs by defining one level at a time, it can delay the discovery of feasibility problems lurking at lower levels. This can be dealt with by using an iterative process in which low-level problems are addressed with a redesign starting from the top-down. [GLASS92] Another drawback with functional design methods is they have limited software reuse benefits. They can lead to the redundant development of numerous partial versions of the same modules — *decreasing productivity* and creating *configuration management overloads*. [AGRESTI86]

### 11.2.6.4 Data-Oriented Design

For heavily data-oriented applications (such as MISs) where the problem involves a database or collection of files, a data-oriented design approach is often used. This approach focuses on the structure and flow of information, rather than the functions it performs. A data-oriented design is a clear cut framework: data structure is defined, data flow is identified, and the operations that enable the flow of data are defined. A problem often encountered with a data-oriented approach is a “*structure-clash*,” where the data structures to be processed are not synchronized (e.g., input file is sorted on rank, whereas output file is sorted on time-in-grade). Solutions to the clash

problem can be the creation of an intermediate file or the conversion of one structure processor into a subroutine for the other. [GLASS92]

### 11.2.6.5 Object-Oriented Design

A variety of object-oriented (OO) methodologies and tools are available for software development. Each approach emphasizes different phases and activities of the software life cycle using various terminologies, products, processes, and implementation techniques. The impact of a methodology on the conduct and management of a software development effort can be extensive. Therefore, if you decide to employ an OO approach, you should encourage your developer to investigate and select the OO approach that best fits your specific program needs. An object-oriented design (OOD) method focuses on interconnecting data objects (data items) and on processing operations in a way that modularizes information and processing rather than processing alone. The software design becomes de-coupled from the details of the data objects used in the system. These details may be changed many times without any effect on the overall software structure. Instead of being based on functional decomposition or data structure or flow, the system is designed in terms of its component objects, classes of objects, subassemblies, and frameworks of related objects and classes. Strassmann explains that component-level software objects can be quickly combined to build new applications. These objects are then candidates for *reuse on multiple applications* — lowering development costs, shortening the development process, and improving testing. Because objects are responsible for a specific function, they can be individually upgraded, augmented, or replaced — leaving the rest of the system unchanged. [STRASSMANN93]

Object-oriented technology lets software engineers take a kind of *Velcro* (or *rip-and-stick*) approach to software development. The idea is to encase software code into objects that reflect real-world entities, such as airplanes, crew chiefs, or engineering change orders. The internal composition of objects is hidden from everyone but the programmer of the object. Once molded into objects, the encapsulated code can be stored in repositories that are network-accessible by other designers. As needed, component-level objects can be quickly grafted with other objects to create new applications.

Using a familiar graphical user interface, such as windows and icons, the object-oriented approach lets developers visualize and design applications by *pointing-and-clicking* on the objects they wish to use. This approach *cultivates reuse* because objects can be used in multiple applications, lowering development costs, speeding up the development process, and improving testing. Because objects are responsible for a specific function, they can be individually upgraded, augmented, or replaced, leaving the rest of the application unaffected. [STRASSMANN93] *OOD has the added benefit of allowing users to participate more closely in the development process.* It is very difficult to describe in writing what a software application is supposed to do, whereas a graphical representation is easy to visualize and manipulate. Objects help all involved in the development process (the systems/software engineers, programmers, and users) to understand what the application should do.

---

## 11.2.7 Problem Domains and Solution Domains

Object-oriented development pioneer, Grady Booch, explains how OOD methodology facilitates developers in solving real-world problems through the creation of complex software solutions. The problem domain has a set of real-world objects, each with its own set of appropriate operations.

These objects can be as simple as a baseball bat or as complicated as the Space Shuttle. Also in the problem domain are real-world algorithms that operate on the objects, resulting in transformed objects. For example, a real-world result may be a course change for the Space Shuttle. When developing software, either the real-world problem is modeled entirely in software, or for example in embedded software, real-world objects are transformed into software and hardware to produce real-world results. *No matter how the solution is implemented, it must parallel the problem domain.* Programming languages provide the means for abstracting objects in the problem domain by implementing them into software. Algorithms, which physically map some real-world action (such as the movement of a control surface), are then applied to the software object to transform it. The closer the solution domain maps your understanding of the problem domain, the closer you get to achieving the goals of modifiability, reliability, efficiency, and understandability.

OOD differs fundamentally from traditional development, where the primary criterion for decomposition is that each software module represents a major step in the overall process. With OOD, each system module stands for an object or class of objects in the problem domain. [BOOCH94] Of course, you will not always have perfect knowledge of the problem domain; instead, it may be an iterative discovery process. As the design of the solution progresses into greater states of decomposition, it is likely new aspects of the problem will be uncovered that were not initially recognized. However, if the solution maps directly to the problem, any new understanding of the problem domain will not radically affect the architecture of the solution. With an object-oriented approach, developers are able to limit the scope of change to only those modules in the solution domain that represent changing objects in the problem domain. *[The Space Shuttle mission will always be fulfilled by a space vehicle (constant); how that vehicle is propelled (variable) may change as technology advances.]*

The OOD method supports the software engineering principles of abstraction and information hiding, since the basis of this approach is the mapping of a direct model of reality into the solution domain. This strategy also provides a method for decomposing a software system into modules where design decisions can be localized to match our view of the real world. It provides a uniform means of notation for selecting those objects and operations that are part of the design. With Ada as the design language, the details of operations can be physically hidden, as well as, the representation of objects.

---

## 11.2.8 Critical Design Review (CDR)

The purpose of CDR is to verify that the detailed software design is complete, correct, internally consistent, satisfies all requirements, and is a suitable basis for coding. The CDR follows the Detailed Design phase, and the successful completion of CDR marks the completion of the Detailed Design phase. The CDR is performed to establish the integrity of a computer program design before coding and testing begins. When a given software system is so complex that a large number of software modules will be produced, the CDR may be accomplished in increments during the development process corresponding to periods during which different software units reach their maturity. For less complex products, the entire review may be accomplished at a single meeting. The primary product of CDR is the formal review of specific software documentation, which will be approved and released for use in coding and testing. CDR covers the following topics:

- Description of how the top-level design, presented at the PDR, has been refined and elaborated upon to include the software architecture down to the lowest-level units.
- The assignment of CSCI requirements to specific lower-level CSCs and units.
- The detailed design characteristics of the CSCs. These detailed descriptions shall include data definitions, control flow, timing, sizing, and storage allocation. Where the number of units is large and the time for the CDR limited, the description concentrates on those units performing the most critical functions.
- Detailed characteristics of all interfaces, including those between CSUs, CSCs, and CSCIs.
- Detailed characteristics of all databases, including file and record format and content, access methods, loading and recovery procedures, and timing and sizing.
- Human engineering considerations.
- Life cycle support considerations that include a description of the software tools and facilities used during development that will be required for software maintenance.

The contractor should answer the following questions at CDR:

- Are each unit's inputs/outputs clearly defined? Are the units, size, frequency, and type of each input/output parameter stated?
- Is the processing for each unit defined in sufficient detail, via flow charts, programming design language (PDL), structured flow charts, or other design language so that the unit can be coded by someone other than the original designer of the unit?
- What simulations, models, or analyses have been performed to verify that the design presented satisfies system and software requirements?
- Has machine dependency been minimized (e.g., not overly dependent on word size, peripherals, or storage characteristics)? Have machine dependent items been segregated into independent units?
- Has the database been designed and documented? Has it been symbolically defined and referenced (e.g., was a central data definition used)?
- Have the software overall timing and sizing constraints been subdivided into timing and sizing constraints for individual units? Are the required timing and sizing constraints still met?
- Have all support tools specified for coding and debugging (i.e., pre- and post-processor) been produced? If not, are they scheduled early enough to meet the needs of the development schedule?
- Are the software test procedures sufficiently complete and specific so that the test can be conducted by someone else?
- Do the test procedures include input data at the limits of required program capability? Do test procedures contain input that will cause the maximum permitted values and quantities of output?
- Do test procedures exercise representative examples of all possible combinations of both legal and illegal input conditions?
- Are there any potential software errors that cannot be detected by the test runs in accordance with the test procedures? If so, why? What will be done to make certain the software does not have those errors?

- How will detected errors be documented? How will corrective actions be recorded and verified?
- What progress has been made in developing or acquiring the simulations and test data needed for testing? Will they be available to support these testing efforts? How will they be controlled during the test effort?

---

## 11.2.9 Coding

The purpose of this phase is to put the design produced during the design phase into practical effect. Programmers translate the design into a language the computer can understand and execute. Of course, as units are coded, further decomposition may be required. In this regard, the design/code/test phases lose their distinction and should form an iterative process at each stage of the solution. Peer reviews or other forms of formal inspection are an important part of the coding phase. Jones reports:

Most forms of testing are less than 30 percent efficient, in that they will find less than one bug out of every three actually present. Formal design and code inspections tend to be the most efficient, and they alone can exceed 60 percent in defect removal efficiency. [JONES91]

---

## 11.2.10 Testing

Testing has been the most labor-intensive activity performed during software development. Testing often requires more effort than the combined total for requirements analysis and design by as much as 15%. It has also been a significant source of risk, often not recognized until too late into cost and schedule overruns. There are two basic reasons why testing is risky. First, testing traditionally occurs so late in software development that defects are costly and time consuming to locate and correct. Second, test procedures are *ad hoc*, not defined and documented, and thus, not repeatable with any consistency across development programs. We enter testing without a clear idea of what is to be accomplished and how. Testing can be a major source of wheel spinning that can lead from one blind alley to another.

Historically, software testing has been a process that *checks software execution* against requirements agreed upon in the SRS. The goal of software testing was to demonstrate correctness and quality. Today, we know this definition of testing is imprecise. Testing cannot produce quality software — nor can it verify correctness. Testing can only confirm the presence (as opposed to the absence) of software defects. The testing of source code alone cannot ensure quality software, because testing only finds faults. It cannot demonstrate that faults do not exist. Therefore, correcting software defects is a fix, not a solution. *Software defects are usually symptoms of more fundamental problems in the development process.* Development process problems might be the failure to follow standard procedures, the misunderstanding of a critical process step, or a lack of adequate training.

Thus, the role of software testing has evolved into an integrated set of *software quality activities* covering the entire life cycle. Software tests apply to all software artifacts. To engineer quality into software, you must inspect, test, and remove errors and defects from requirements, design, documentation, code, test plans, and tests. You must institute an effective defect prevention

program that engages in accurate defect detection and analysis to determine how and why they are inserted. Remember, “*Error is discipline through which we advance.*” [CHANNING92] Although testing cannot prevent defects, *it is the most important activity for generating the defect data necessary for process improvement.*

Developmental testing must not interfere with, nor stand apart from, daily development activities; it must be embedded within your development process. Furthermore, given the uniqueness of each DoD software development program, the embedded testing methodologies you apply must be customized to your environment. If testing standards are instituted and the testing process is properly planned, the time and effort required for testing can be significantly reduced. [MOSLEY93]

### 11.2.10.1 Testing Objectives

Because testing is not limited to the testing phase, but spans the entire software development, your developer’s Test Plan must state general objectives for the overall testing process and specific objectives for each development phase. The primary objective should be to assess whether the system meets user needs. Other objectives depend on the software domain and the environment in which the system will operate. Testing objectives also focus on verifying the accomplishment of quality attributes, as discussed in Chapter 13, *Software Estimation, Measurement, and Metrics*. The bottom line with testing is *test early, test often, and use a combination* of testing strategies and techniques. Also, *automate every testing activity* economically and technically feasible.

#### 11.2.10.1.1 Defect Detection and Removal

Defect detection and removal is the most basic testing objective and the one aspect of quality that can be measured in a tangible and convincing way. Defects (and their removal) can be measured with great precision, and their measurement is one of the fundamental parameters to include in every testing and measurement program. Programs performing well in defect removal normally perform well in other aspects also. Jones reports:

“Interestingly, a cumulative defect removal efficiency of 95 percent appears to be a powerful nodal point for software projects. Projects which achieve overall removal efficiencies approximating or exceeding 95 percent tend to be optimal in three other aspects as well: (1) they have the shortest schedules for projects of their size and type; (2) they have the lowest quantity of effort in terms of person-months or person-hours; and (3) they have the highest levels of user satisfaction after release.” [JONES91]

It is important to understand that “*errors*” relate to early phases of development: requirements definition and design specification. An error in requirements or design will cause the insertion of one or more “*defects*” in the code. However, a defect may not be visible during code execution — neither during testing nor operation. If a defect is executed, it may result in a tangible fault, or it may not. Programmers *debug* code to correct defects by testing for tangible failures. But the lack of failures cannot guarantee the absence of defects. Even if the defect executes, it may not be visible as output. Furthermore, defect correction does not necessarily imply that the error (source of the defect) causing the defect has been corrected.

There are three broad classifications of defects, named after the development phase where they are found: unit/component defects, integration defects, and system defects. Unit defects are the

easiest to find and remove. When a test is failed during system testing, you may not be able to tell if the failure is caused by a unit, integration, or system defect. It is only after the failure is resolved that we know from where it came. As discussed above, system testing is more expensive than unit testing and any unit defect remaining during system testing translates into costly scrap and rework. Integration defects are more difficult to detect and prevent because they occur from interaction among otherwise correct components. Component interactions are *combinatorial*— i.e., they grow as  $n^2$  (the square of the number of components integrated) or worse (e.g.,  $n!$ — that number factorial). An integration testing objective is to assure that few, if any, harmful component interaction defects remain before going to system testing. During system testing, we have the added complexity of multitasking, i.e., the order in which things happen can no longer be predicted with certainty. This uncertainty and the issue of timing is rich soil forever more complex system defects. [BESIER95]

You might ask, if the defect cannot be detected and does not show itself as output, why bother removing it? With mission and safety critical software operating under maximum stressed conditions, the chance of a latent defect-related software failure often increases beyond acceptable limits. This dichotomy amplifies the need to detect, remove, and ultimately prevent the causes of errors before they become illusive software defects. Latent, undetected defects have the tendency to crop up when the software is stressed beyond the scope of its developmental testing. It is at these times, when the software is strained to its maximum performance, that defects are the most costly or even fatal.

The number of errors (unintentionally injected into software by requirements analysts and designers) and defects (injected by programmers while interpreting designs) can be quite large. For complex software systems they can number in the tens-of-thousands. [PUTNAM92] Most of these, however, are removed before delivery by the self-checking of analysts and programmers, by design reviews, peer inspections, walkthroughs, and module and integration testing. Jones estimates the pre-delivery defect removal rate using these techniques to be at about 85%.

For systems where failure to remove defects before delivery can have catastrophic consequences in terms of failed missions or the loss of human life, defect removal techniques must be decidedly intense and aggressive. For instance, because the lives of astronauts depend implicitly on the reliability of Space Shuttle software, the software defect removal process employed on this program has had a near perfect record. Of the 500,000 lines-of-code for each of the six shuttles delivered before the Challenger, there was a zero-defect rate of the mission-unique data tailored for each shuttle mission, and the source code software had 0.11 defects per thousand lines-of-code (KLOC). [KOLKHORST88]

**NOTE: This is not to imply the Challenger disaster was caused by software defects. It was merely the cutoff point for the report upon which this example is based.**

These impressive figures reflect a formal software engineering process that concentrates on learning from mistakes. Finding and correcting mistakes must be a team effort where no individual is held responsible or singled out. Figure 11-6 illustrates the steps performed for every software defect found on the Space Shuttle program, regardless of significance. Process improvement is relentlessly achieved by performing feedback during steps 2 and 3. Much credit for this achievement is attributable to peer inspection techniques [*discussed below*], pioneered by IBM-Houston.

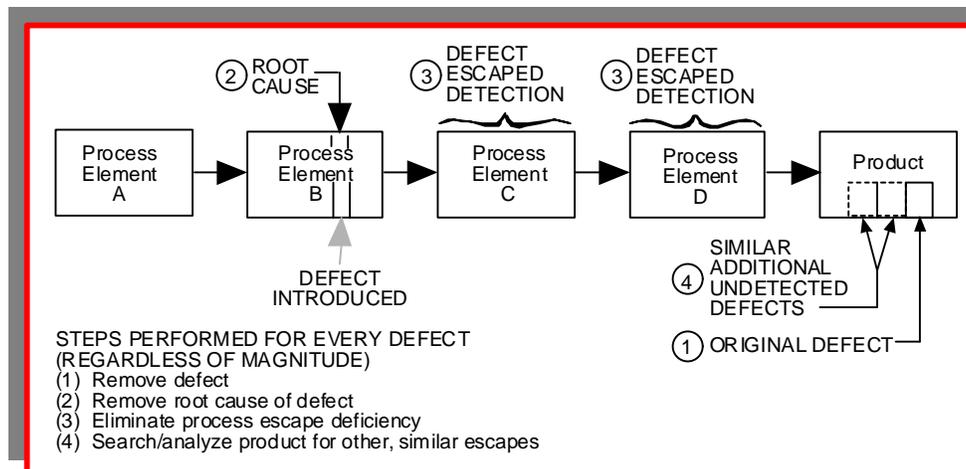


Figure 11-6. Space Shuttle Defect Removal Process Improvement [KELLER93]

### 11.2.10.1.2 Defect Removal Strategies

Given the magnitude of errors associated with requirements and design, it is obvious that these huge sources of errors must be included in your quality control/assurance strategies. PAT teams, demonstrations, prototypes, and peer inspections are all necessary to control front-end sources of errors. Testing and peer inspections are also necessary for discovering inserted defects. It is important to recognize the up-front costs of inspections and testing, as well as the expected downstream cost, quality, and schedule benefits. [BRYKCZYNSKI93]

Finding and removing defects is the most expensive activity in which the software industry invests. Organizations who engage in quality control and defect prevention have an enormous competitive advantage over those who do not. Given the low average efficiencies of most defect removal methods, it is imperative that your developer use a variety of removal techniques to achieve a high cumulative removal efficiency. Special attention must be given to the defects accidentally introduced as the by-products of fixing previous defects. *The total quantity of bad fixes averages about 5% to 10%, which directly relates to the complexity of the product being repaired.* Leading commercial and DoD software developers, for example, may include as many as 20 to 25 different defect removal activities. Serious quality control requires a combination of many techniques each aimed at a class of defects for which its efficiency is the highest. The bottom line in choosing your defect prevention and removal strategy is to choose the combination of methods which will achieve the highest overall efficiency and quality gains for the lowest total life cycle cost.

### 11.2.10.2 Unit Testing

Testing is usually divided into three activities — unit testing, integration testing, and systems testing. A unit is a component. A component is an aggregate of one or more components that can be tested as an aggregate, such as subroutines, functions, macros, the application and the subroutines it calls, communicating routines, or an entire software system. Unit testing is usually performed by the programmer who created the unit. Unit testing is often accomplished in an incremental design/code/test fashion, where more and more of the completed system is progressively tested during each increment. Test cases are selected to force the occurrence of defects. The results of unit tests are then analyzed to see if any defects have occurred, and a debugging process is performed to remove them. A description of the type, cause, and correction

of defects is then placed in a database for future process improvement analysis. The purpose of unit testing is to remove all defects from the component under test. The easiest way to accomplish this is to *begin as early as possible* with requirements testing of the component. Component requirements are easily tested as they represent but a small subset of the requirements for the whole software product. Structure-driven, statistic-driven, and risk-driven testing are also performed during unit testing. [GLASS92] There are two basic types of testing performed at the unit and system level: structural testing (also called glass-box or white-box testing) and behavioral testing (also called functional or black box testing).

Structural testing, or *testing-in-the-small*, ideally involves exhaustively execution of all paths of control flow in a module or system. In reality, exhaustive path testing is usually impossible because the number of potential paths can be infinite. Also, path testing cannot detect missing paths and cannot detect data sensitivity defects. Thus, structural test case design must be based on random and/or selective testing of control flow. Structural testing techniques include:

- Statement coverage,
- Decision coverage,
- Condition coverage,
- Decision/condition coverage,
- Multiple decision/condition coverage,
- Independent path coverage, and
- Structured tableau. [MOSLEY93]

Behavioral testing, or *testing-in-the-large*, focuses on requirements. For example, testing consists of testing all features mentioned in the specification. Behavioral testing be performed, in theory but not in practice, with total ignorance of how the object under test is constructed. It is not concerned with the internal structure of behavior of the module or system, but only with the instances when the program or module does not behave as indicated in its functional specifications. In contrast with exhaustive path testing, behavioral testing focuses on exhaustive input testing, which is also an impossible task. The number of possible valid inputs approaches infinity, as does the number of all possible invalid inputs. Thus, behavioral test case design must be based on random and/or selective testing of inputs. Behavioral testing techniques include:

- Equivalence partitioning,
- Boundary analysis,
- Cause effect graphing,
- Structured tableau, and
- Error guessing. [MOSLEY93]

Neither testing approach alone is enough. Behavioral testing should be used throughout development, while structured methods are best used later in the process. Both methods are complementary; however, some redundancy of test case design exists between certain techniques within the two approaches. The tester should select and use a combination that maximizes yield and minimizes redundancy. Again, automated tools that build test cases are a sound investment.

### 11.2.10.3 Integration Testing

While unit testing is performed by programmers on the modules they develop, integration testing is performed to determine how the individual modules making up a subsystem component perform together as an integrated unit. With large software developments, integration testing often involves the software of many developers where individually developed modules are combined into various software subsystems and tested as integrated units. As a manager, you must be aware of the political problems associated with integration testing of multiple vendor products. Often, when a defect occurs on the interface between two supposedly pre-tested and correct components, neither developer wishes to take the blame for the defect and finger pointing occurs. Each developer believes he has a perfect module and that the defect must have been caused by — and thus must be fixed by — the other person. This situation takes a mix of tact and diplomacy on the integration test manager's part to resolve these problems and get the defects corrected. [GLASS92]

### 11.2.10.4 System Testing

System testing (conducted by the systems developer) usually begins after integration testing is successfully completed. Some redesign and tweaking of both the hardware and software is performed to achieve maximum levels of performance and to iron out bugs. System testing is very much like integration testing where components are integrated into whole parts, but not necessarily whole *software* parts. As with integration testing, the system tester tries to invoke defects while the individual component developers are responsible for their repair. The system tester is also responsible for resolving any political problems that arise. *[Remember, it is essential to perform adequate end-to-end testing prior to signing-off on standard form DD-250s for software.]*

---

## 11.3 Building Secure Software

Security is an essential element of many major DoD software-intensive systems. Adversaries actively collect information about our new systems and software to negate their combat effectiveness and eliminate our advantage of surprise. You must actively plan for and apply OPSEC measures to protect crucial information throughout your acquisition process. The OPSEC process helps guide the development of OPSEC measures. The process asks: what needs to be protected, from whom, is there a potential for exposure of critical information, what are the risks, and how is protection to be accomplished? This team effort must be revisited as your program matures and parameters change. A well thought out plan of protection and its judicious application will ensure the integrity and combat effectiveness of new systems and help us attain our mission objectives.

---

### 11.3.1 Security Planning

Security is a crucial aspect of strategic system and software planning often overlooked. DoD contingency strategists (wargamers) envision the objectives of war in the 21st century, not as attacks to destroy enemy lives, but as maneuvers to gain control of those invisible, more vulnerable, more significant and consequential software-driven systems. Weapons systems dependent on satellite communications for target positioning, global financial systems, highly distributed military

logistics and air traffic control systems, and secure telecommunication networks are “soft” because they are highly pregnable. [BLACK93] Due to its vulnerability, failure to plan for software security could prove catastrophic. Today, reports abound of hackers gaining unauthorized access to software systems, sometimes creating serious damage. Other software-related security problems have resulted in severe financial loss or even loss of life. *The protection of your software must be a major element in your strategic planning process.*

The common objective of acquisition activities is the production of combat-ready weapon systems and/or support for those systems to further our national defense. The advantage we seek, the success of our defensive efforts, is often expressed in the element of surprise. *Surprise*, in this instance, means that our systems, when deployed, can operate in hostile environments and do the job for which they were designed. Lack of surprise means that an adversary already knows enough about our systems to counter them and/or to render them ineffective once deployed. Lieutenant General V.A. Reznichenko, authoritative tactician for former Soviet Union ground forces, explained why security is so important.

*“Surprise makes it possible to take the enemy unawares, to cause panic in his ranks, to paralyze his will to resist, to drastically reduce his fighting efficiency, to contain his actions, to disrupt his troops’ control, and to deny him the opportunity to take effective countermeasures quickly. As a result, this makes it possible to successfully rout even superior enemy forces with the least possible losses to friendly forces.”* — Col General V.G. Reznichenko [REZNICHENKO84]

Some program managers consciously omit security (and safety) requirements from their plans, as they believe such considerations will significantly increase software development costs. As in risk abatement, the benefits of including software security requirements up front must be weighed against life cycle costs. Not planning for security up front and having to address these requirements after development is underway (or the system is deployed) can severely impact the cost of your software development (and system life cycle costs) as they constitute significant cost drivers.

It is imperative that your new software (and hardware) be fully protected commensurate with your program requirements and sensitivities throughout the development life cycle to ensure it is fully combat effective at IOC and that the element of surprise is retained. It makes little sense to expend valuable resources (manpower, money, and time) on software that is compromised before it can fulfill design and mission objectives. Software protection must be an integral and normal part of all acquisition activities.

Building preemptive defenses into your software is one way to fight the software security war against hackers and enemy access to our vital information resources. Another method is to build in the ability to bounce back quickly if penetration is accomplished. You must, therefore, plan for non-lethal warfare risks to be prepared, through prevention and circumvention, for today’s *software-versus-software* battlefield.

---

### 11.3.2. Operations Security (OPSEC)

OPSEC is specifically designed to control and protect information of intelligence value to an adversary. This information is called critical information. Critical information includes the specific facts about our intentions, capabilities, limitations, and activities needed by adversaries to guarantee the failure of our mission. It is the key information about our programs, activities,

hardware, and software, which if lost to an adversary, would compromise that program. Critical information may be either classified or unclassified. It is not only the classification of the information that is important, but also the *value* of the information to an adversary. It requires the safeguarding of all classified information and protection from tampering for unclassified information.

OPSEC is implemented by the development of an OPSEC Plan. The plan is based on a thorough analysis of the important and sensitive aspects of your program (or software system) and of the environment for which the software is being developed. OPSEC planning follows the OPSEC process, a logical method of information analysis and evaluation guiding protection and control. The OPSEC process can be applied to virtually any software development activity, and is as simple or complicated as the situational environment warrants. The steps in the OPSEC process are:

- Identify critical information,
- Describe the intelligence collection threat,
- Identify OPSEC vulnerabilities,
- Perform risk analysis, and
- Identify countermeasures to control and protect the information.

The plan summarizes the results of this analysis process and becomes the framework for subsequent software protection measures.

- **Critical information.** Because you have to know what to protect, the first step is to identify critical information and the indicators that point to it or that may divulge it. The first listing of critical information is in the Operational Requirements Document (ORD) developed by the user, which is very broad and general. As your program proceeds, this list must constantly be reviewed and refined. As your program matures, the list of critical information will become more specific and detailed.
- **Threat.** The threat is specific information about an adversary's capabilities and intentions to collect critical information. It begins with the identification of the adversary(ies). The adversary's resources/assets available to collect critical information and the degree of the intent to collect is then assessed. The threat assessment must be specific (e.g., geographical location, facility, program office, software system, laboratory, or contractor facility). Threat information must be obtained in coordination with the OPSEC officer or through local liaison officers and organizations (e.g., the Air Force Office of Special Investigations, Air Intelligence Agency, or the National Air Intelligence Center. Intelligence collection of threat information is also included in the System Threat Assessment Report (STAR) validated by the Defense Intelligence Agency (DIA).
- **Vulnerability.** Critical information and indicators of critical information are compared with the threat to determine if an OPSEC vulnerability exists. For an OPSEC vulnerability to exist, critical information must be potentially open and available to an adversary, and that adversary must have some type of collection platform in place to obtain the information (e.g., a spy satellite, an agent, an intelligence gathering ship, or communications network access). *If sensitive information is available and an adversary can collect it, then an OPSEC vulnerability exists.*

- **Risk assessment.** A risk assessment is a cost/benefits analysis of proposed protective measures and the mission imperative. Several factors drive this assessment. First, no system can be 100% secure unless it is sealed off from all outside influences. Second, whatever protective measures are used, they must not unduly hinder or prevent mission accomplishment or the attainment of program objectives. Finally, a balance must be found that provides the maximum possible protection while maintaining program integrity.
- **OPSEC measures.** Various methods must be developed that best meet operational protection requirements while mitigating the identified OPSEC vulnerability. OPSEC measures are program specific and must be tailored to the identified vulnerability. OPSEC measures include:
  - Action control measures. These are actions that can be executed to prevent detection and avoid exploitation of critical information. You should avoid stereotyped procedures which can be exploited by an adversary. Examples of action control include making preparations inside buildings rather than outside, conducting activities at night, and adjusting schedules or delaying public affairs releases.
  - Countermeasures. These are methods to disrupt adversary information gathering sensors and associated data links or to prevent the adversary from obtaining, detecting, or recognizing critical information. Examples include jamming, masking, encryption, interference, camouflage, and diversions.
  - Counteranalysis. These are methods to affect the observation and/or interpretation of adversary analysts. They do not prevent detection, but enhance the probability that the detectable activity is overlooked or its significance is misinterpreted. Counteranalysis measures provide uncertainty and alternative answers to adversary questions. Deceptions, including covers and diversions, are in this category of OPSEC measures. Detailed planning of deceptions are separate from protection planning. However, close coordination between OPSEC and deception planners will facilitate the desired result.
  - Protective measures. These measures can and should include the use of all established security disciplines.

Although security is ultimately your responsibility, program protection is not a one-person job. OPSEC measures run the gamut of possibilities and there is ample help available. Each Major Command (MAJCOM), product center, logistics center, test range, and laboratory has an identified OPSEC point-of-contact. Indeed, each security discipline has a point-of-contact. Software protection is, thus, a coordinated team effort — the same as other program activities.

Historically, it has been difficult and expensive to design and build secure/trusted data systems. The traditional way of building secure systems has been to use logical and physical separation (i.e., an “*air gap*”) based on providing a physically secure facility for each system, with everyone in the facility cleared to the level of the most sensitive data. This method is not only expensive, but very inefficient, and has several undesirable properties such as the cost of duplicating facilities, and multiple sets of hardware and software. There is also an inability to share personnel talent and skills due to the need for separation and the number of expensive clearances for people who have no access to the data itself. Possibly the most serious issue is the inability to share data. This creates serious data concurrency problems as duplicated data in the myriad of systems are updated at different frequencies — greatly increasing the probability of error as the number of instances increases. This was a major problem during the Gulf War. Virtually all the data needed was in theater, but it was not accessible in a way that allowed coherent data fusion and integration.

This problem should soon be totally eradicated. All necessary COTS components for building operational systems [i.e., hardware/operating systems, networks, and relational database management systems (RDBMS’)] are National Computer Security Classification (NCSC) evaluated at the CB/B1 and B2 levels. The old quandary that “*COTS products are not secure*” and “*secure products are not COTS*” is no longer true. Today it is possible to get a hardware/operating system-network-RDBMS combination that was evaluated together, which greatly reduces the accreditation effort of the developer and the user.

The RDBMS is the most critical portion of the secure solution. The first, and most important, concern should be a vendor’s overall philosophy and commitment to developing secure products. Some build a minimally compliant product so they appear to have complete secure and non-secure product suites. Serious secure product vendors meet the extreme assurance requirements required at the B2 level and above, while others have layered C2/B1 level features that meet the minimal assurance requirements at B1 and below. Vendors who are serious about the secure products market also view security as an attribute of their product — not as a 150% to 200% premium over the price of their standard product.

Compatibility of the vendors’ products at various levels is a major development security issue. Compatibility has many benefits such as the ability to partition data and applications across different levels without having to duplicate the applications. For example, the ability to access untrusted administrative systems and secure operational systems in the same application is useful. Also desirable is the ability to separate very sensitive SCI data into a B2 assurance RDBMS engine, routine operational data into a B1 assurance RDBMS, and other administrative data into a C2 assurance RDBMS. This makes data and security administration easier while retaining the usability and functionality of one logical database with joins and other transaction management capabilities. In addition to only having to develop one set of applications, this capability has several performance advantages. Joins are required only in those less normal scenarios where multiple kinds of data are required in a single transaction. Otherwise, a single server is used, increasing the apparent network bandwidth for users at different levels. The RDBMS’ distributed capabilities should make the data partitioning invisible to the client-user so the only relevant issue is the client security level, not specialized knowledge of the physical data schema.

All secure products are evaluated against the Trusted Computer System Evaluation Criteria (TCSEC) defined in DoD 5200.28.Std (the Orange Book) and its various interpretation guides. The Orange Book is a statement of the DoD basic security policy and relies on the Bell-Lapadula Security Policy Model. The principle of the Bell-Lapadula model is access mediation based on the relative values of a user’s (subject) clearance level and the data (object) classification level as conveyed by appropriately assigned security labels. The salient features of the model are a “*subject*” may access “*objects*” at its session (login) level and below, and “*may-write objects*” at its session level only.

This policy has some onerous implications for RDBMS’. The most serious of which are that implementation of this policy dictates that: (1) uniqueness of a primary key is only guaranteed within a single security level; (2) an index on a table exists at a single security level; and (3) referential integrity is guaranteed only at a single security level. Related issues are the serious covert channels in databases centered around the physical storage of labels in each row and the serialization of row IDs. The management of data integrity locking mechanisms at different security levels is also a problem. (INFORMIX uses a unique security metadata approach which

eliminates all these covert channel issues by avoiding the need to physically store labels in each row.) To support complex and sophisticated application development, most secure RDBMS' provide a means for mitigating these problems.

The crucial item is the safety and granularity of these mechanisms. Most secure RDBMS' support the simple, but coarse, method of using a configuration option to set it to either "on" or "off." A more sophisticated approach is to support a set of discrete privileges granted and revoked selectively by the Information Systems Security Officer (ISSO) to facilitate a specific task. These discrete privileges are manageable at a granularity no greater than a transaction boundary, and deal with the granularity of indices, uniqueness of primary keys, referential integrity across levels, and locks at multiple levels.

The selection of a secure RDBMS should not lock a developer into a particular hardware environment. A committed secure products vendor will support mainstream hardware platforms and operating systems (e.g., HP, Sun, IBM, DEC, Harris, AT&T, and SCO). They will also support all applicable standards [such as FIPS 127, FIPS-156, XOPEN, RDA, ANSI XXX, and *de facto* standards (e.g., DRDA, ODBC, and TCP/IP)] in their standard and secure products. A secure product should not have a significant performance degradation over an equivalent non-secure product. Vendors should publish official audited benchmarks of both secure and non-secure products.

---

## 11.4 The Bottom Line

The bottom line for successful software development is adherence to the *software engineering discipline* discussed throughout these Guidelines for its stabilizing effects on the development process. No sounder advice can be given. As General George Washington explained in a letter of instructions to the captains of his Virginia regiments in 1759,

Discipline is the soul of an army. It makes small numbers formidable; procures success to the weak and esteem to all. [WASHINGTON59]

---

## 11.5 References

- [AGRESTI86] Agresti, William W., ed., New Paradigms for Software Development, IEEE Computer Society Press, Washington, D.C., 1986
- [BESIER95] Besier, Boris, Black-Box Testing: Techniques for Functional Testing of Software and Systems, John Wiley & Sons, Inc., New York, 1995
- [BLACK93] Black, Peter, "The Next Generation of Weapons: Dependency on Electronic Systems Make Us Vulnerable," *Washington Technology*, December 2, 1993
- [BOOCH94] Booch, Grady, Software Engineering With Ada, Third Edition, The Benjamin/Cummings Publishing Company, Inc., Menlo Park, California, 1994
- [BRYKCZYNSKI93] Brykczynski, Bill and David A. Wheeler, "An Annotated Bibliography on Software Inspections," Institute of Defense Analysis, Alexandria, Virginia, January 1993
- [CHANNING92] Channing, William Ellery, as quoted by Lowell Jay Arthur, Improving Software Quality: An Insider's Guide to TQM, John Wiley & Sons, Inc., New York, 1993
- [COAD90] Coad, Peter and Edward Yourdon, Object-Oriented Analysis, Yourdon Press, Prentice Hall, Englewood Cliffs, New Jersey, 1990
- [DEMING82] Deming, W. Edward, *Out of Crisis*, Massachusetts Institute for Technology, Center for Advanced Engineering Study, Cambridge, Massachusetts, 1982
- [DENTON92] Denton, Lynn and Jody Kelly, Designing, Writing & Producing Computer Documentation, McGraw-Hill, Inc., New York, 1992
- [DMR91] "Strategies for Open Systems," briefing presented by DMR Group, Inc. to SAF/AQK, March 14, 1991
- [DOBBINS92] Dobbins, James H., "TQM Methods in Software," G. Gordon Schulmeyer and James I. McManus, eds., Total Quality Management for Software, Van Nostrand Reinhold, New York, 1992
- [GLASS92] Glass, Robert L., Building Quality Software, Prentice Hall, Englewood Cliffs NJ, 1992
- [IEEE90] IEEE Standard Glossary of Software Engineering Terminology, IEEE Std 610.12-1990, Institute of Electrical and Electronic Engineers, Inc., New York, NY, December 10, 1990
- [JONES91] Jones, Capers, Applied Software Measurement, McGraw-Hill, Inc, New York, 1991
- [KELLER93] Keller, Ted, briefing "Providing Man-Rated Software for the Space Shuttle," IBM, Houston, Texas, 1993
- [KOLKHORST88] Kolkhorst, Barbara G., and A.J. Macina, "Developing Error-Free Software," Fifth International Conference on Testing Computer Software, US Professional Development Institute, Silver Springs, Maryland, June 1988
- [McMENAMIN84] McMenamin, Steve and John Palmer, Essential Systems Analysis, Yourdon Press, Englewood Cliffs, New Jersey, 1984
- [MOSLEY93] Mosley, Daniel J., The Handbook of MIS Application Software Testing: Methods, Techniques, and Tools for Assuring Quality Through Testing, Yourdon Press, Englewood Cliffs, New Jersey, 1993
- [PAULSON79] Paulson, Paul J., as quoted in the *New York Times*, May 4, 1979
- [PAYTON92] Payton, Teri F., briefing, "Reuse Context," presented at the STARS/Air Force Reuse Orientation, October 14, 1992
- [PRESSMAN93] Pressman, Roger S., "Understanding Software Engineering Practices: Required at SEI Level 2 Process Maturity," briefing prepared for the Software Engineering Process Group, July 30, 1993
- [PUTNAM92] Putnam, Lawrence H., and Ware Myers, Measures for Excellence: Reliable Software On Time, Within Budget, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1992
- [REZNICHENKO84] Reznichenko, Col General V.G., Taktika, 1884

- [STRASSMANN93] Strassmann, Paul A., "Information Warfare for Low-Intensity Conflicts," briefing presented to the Army Executives for Software (ARES), West Point, New York, July 15, 1993
- [THOMPSON91] Thompson (SCXS), "Guidelines" comments from SCXS, March 15, 1991
- [WASHINGTON59] Letter to the captains of the Virginia regiments, July 29, 1759, The Writings of George Washington, John C. Fitzgerald, ed., Washington DC, 1931-41
- [YOURDON90] Yourdon, Edward N., Modern Structured Analysis, Prentice Hall, New Jersey, 1990
- [YOURDON92] Yourdon, Edward N., Decline and Fall of the American Programmer, Yourdon Press, Englewood Cliffs, New Jersey, 1992