

## **Chapter 12**

# **Software Support**

---

## Contents

<b>12.1 Chapter overview .....</b>	<b>12-3</b>
<b>12.2 A Total Life Cycle Approach .....</b>	<b>12-4</b>
12.2.1 Software Support Cost Drivers .....	12-5
12.2.2 Software Support Activities .....	12-6
12.2.3 Software Support Issues .....	12-8
12.2.4 COTS Software Support Issues .....	12-10
<b>12.3 Planning for Support Success .....</b>	<b>12-10</b>
12.3.1 Software Support Cost Estimation .....	12-11
<b>12.4 Software Reengineering .....</b>	<b>12-12</b>
12.4.1 Reengineering Decision .....	12-12
12.4.2 Reengineering Process .....	12-13
<b>12.5 Logistics Support Analysis (LSA) .....</b>	<b>12-14</b>
12.5.1 LSA on the F-22 Program .....	12-16
<b>12.6 Continuous Acquisition and Life Cycle Support (CALC) .....</b>	<b>12-17</b>
<b>12.7 Managing a PDSS Program .....</b>	<b>12-18</b>
12.7.1 Computer Resources Integrated Support Document (CRISD) .....	12-19
<b>12.8 Addressing Software Support in the RFP .....</b>	<b>12-20</b>
12.8.1 Specifying Supportable Software .....	12-22
12.8.1.1 Statement of Objectives (SOO) .....	12-22
12.8.1.2 Specification Practices .....	12-22
12.8.1.3 Documentation .....	12-23
12.8.1.4 Life Cycle Software Support Strategies .....	12-23
<b>12.9 References .....</b>	<b>12-25</b>

---

## 12.1 Chapter overview

In Operation Desert Storm the intensity of battle coupled with large forces using Information Age weaponry and communications created the most intense electronic battlefield ever witnessed. The E-3 Airborne Warning and Control System (AWACS) was an integral part of the battle serving as the “eye” that tracked all battle space aircraft and directed interceptions while safeguarding our forces from surprise enemy aerial attack. The overwhelming density of diverse electronic signals transmitted and received created such a congested environment that the E-3s’ full mission capability was greatly hindered. This E-3 problem had to be quickly corrected and a dedicated software support team sprung into immediate action. The E-3 radar software was rapidly revised, flight tested, and on its way to deployed aircraft within 96 hours. This quick reaction, modification, and change-out during the heat of battle emphasizes the operational necessity for easily supportable software.

The ability to continuously support our major software-intensive systems is a paramount mission requirement. Supportability is critical because there is always an inevitable need to correct latent defects, modify the system to incorporate new requirements, enhance the existing system to add capability, and alter it to increase performance. The ability to accommodate change is an integral requirement of major software-intensive systems.

Unfortunately, when we have fielded unsupported systems, we have often had to expend considerable time and funds to provide the required support or we have had to abandon them altogether. We learned that it is far more cost-effective to address supportability as we define requirements, design the system, and plan for its operational life. In this chapter you will learn how to reduce the risk of acquiring, managing, and maintaining software-intensive systems by ensuring that they are modifiable, expandable, flexible, interoperable, and portable — i.e., supportable.

Software support, often called redevelopment, addresses the maintenance life cycle phase where major software costs occur. Support planning addresses the development acquisition and entails request for proposal (RFP) development that provides for delivery of full documentation, data rights, and delivery of the software engineering environment (SEE) used by the developer.

When tasked with maintenance responsibility of legacy software which has become technologically obsolete, has deteriorated through years of changes, or must be changed anyway to work with new hardware or other software, it may be cost effective to reengineer it. This involves systematic evaluation and alteration of an existing system to reconstitute it (or its components) into a new form or converting it to Ada to perform within a new operational environment, to improve its performance, or to reduce maintenance costs. This process can combine several subprocesses, such as reverse engineering, restructuring, redocumentation, forward engineering, or retargeting.

---

## 12.2 A Total Life Cycle Approach

With the exception of the B-2 bomber, DoD will not be purchasing any additional bomber aircraft in the foreseeable future. Procurements of new, advanced fighter aircraft [i.e., the F-22 or Joint Advanced Strike Technology (JAST)] will not occur until the early 2000s. Thus, we have to rely on existing aircraft platforms for several years. The recent modification to the B-1B Lancer is a prime example of this. The B-1B is being upgraded to a conventional munitions capability. The bulk of the effort focuses on the enhancement and modification of the B-1B's offensive avionics software component. *These trends indicate that the future capability of our major software-intensive systems is inexorably dependent on our ability to cost-effectively maintain them.*

Software maintenance is really a poor name for the post-deployment software support (PDSS) activity. In other engineering contexts, maintenance implies repairing broken or worn-out parts. But software does not break — nor does it wear out. It is for this reason that PDSS is often called the *redevelopment* phase. As defined by the Institute of Electrical and Electronics Engineers (IEEE), software maintenance is

The process of modifying a software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment. [IEEE90]

Software is alive! Whether it is in production or not, it is always in the process of becoming, evolving, changing. Research on software maintenance shows that user requirements impacting software account for 41% of post-deployment support costs, while hardware changes account for 10%. [BASSETT95] That is to say, over half of all software support is driven by changes in the system's external environment. Because software must evolve in response to its external environment, it is more like a living thing than an inanimate object that only needs to be designed once, and thereafter, infrequently repaired or maintained. With software, development (and redevelopment) is the norm in response to external changes. Therefore, designing for maintenance must be incorporated and unified with development.

Software support is different from but includes the same activities as development. It is different because the developer has no existing system from which to work; whereas, the maintainer must be able to read and understand already existing code and solve problems within an existing framework which constrains the solution set. The developer has no *product knowledge* because the product does not yet exist. The maintainer must have complete product knowledge to do his job well. Support is the same as development because the maintainer must perform the same tasks as the developer, such as define and analyze user requirements, design a solution (within the constraints of the existing solution), convert that design into code, test the revised solution, and update documentation to reflect changes. Figure 12-1 illustrates how support tasks correspond to and mirror the development process. [GLASS92]

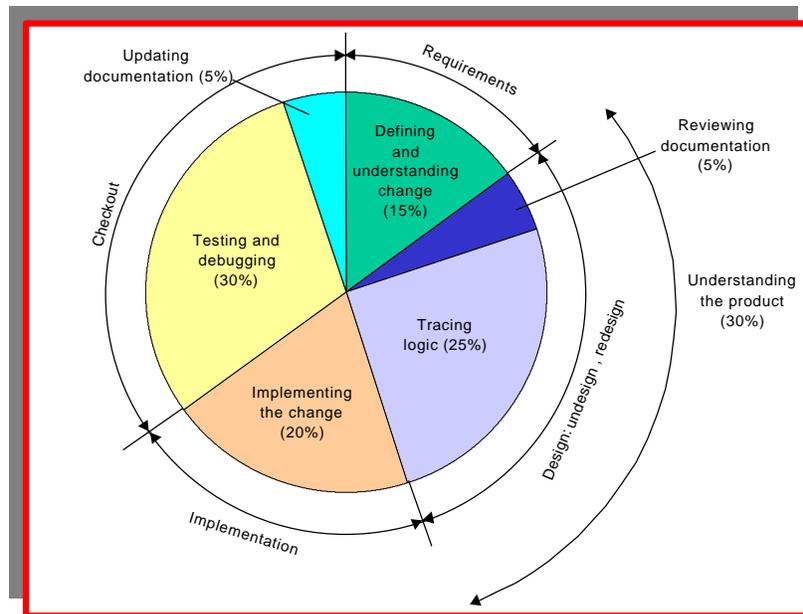


Figure 12-1. Support Tasks Superimposed on the Software Development Phase

## 12.2.1 Software Support Cost Drivers

The demand for delivering high quality software support in time has never been greater. However, software support is by far the biggest life cycle cost driver and the most significant source of system risk for all major DoD software-intensive acquisitions. Although software support occurs during the post-deployment phase, it must be planned for upfront during requirements definition and design. It must also be budgeted for and continuously addressed throughout the system's life. *Developing supportable software is one of the most important criteria for software success.* All the causes of cost and schedule overruns, performance shortfalls, and for programs being thrown off stride are amplified once the system is in the hands of the maintainer. Therefore, the *Software Crisis* has primarily been the *Maintenance Crisis*. According to numerous DoD and industry studies, the typical cost to maintain a software product is from *60% to 80% of total life cycle costs*. Your challenge is to minimize the cost of software maintenance, and to avoid being at the heart of the *Crisis*. These costs are depicted on Figure 12-2.

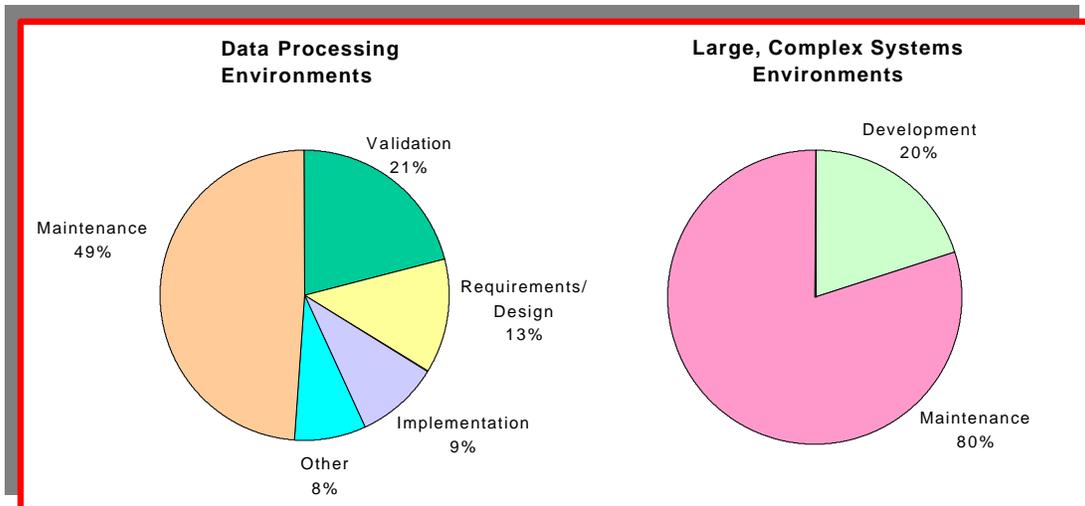


Figure 12-2. Life Cycle Support Costs

These cost increases during the software maintenance phase have historically been caused by dramatic *decreases in productivity* (measured in lines-of-code (LOC)/manmonths or function (feature) points/manmonths.) Productivity drops of 40:1 have been reported during software support. [BOEHM81] For example, what cost \$150/LOC to develop might cost \$1,000/LOC to maintain. This significant increase in system cost demands that basic decisions about how the software will be maintained be made during the concept and design phases. Easy access to the software and an inexpensive medium for distributing enhancements can have significant effects on life cycle costs. A well thought out concept of operations includes hardware provisions for spare connectors, card slots, and memory capacity to facilitate interoperability to new software systems as they are fielded and integrated into the defense inventory. [PIERSALL94] A flexible, modular architecture is also essential for ensuring *understandability, modifiability, interoperability, reusability, expandability, and portability* — all prerequisites for supportable software.

## 12.2.2 Software Support Activities

Figure 12-3 is based on a study of 487 commercial software development organizations and illustrates how software support changes are distributed among support tasks. Most software support dollars are spent on defining, designing, and testing changes. After these activities are performed (whether there is one unit or hundreds of units in the field), subsequent increases in cost are marginal. Support activities include:

- Interacting with users to determine what changes or corrections are needed,
- Reading existing code to understand how it works,
- Changing existing code to make it perform differently,
- Testing the code to make sure it performs both old and new functions correctly, and
- Delivering the new version with sufficiently revised documentation to support the user and the product.

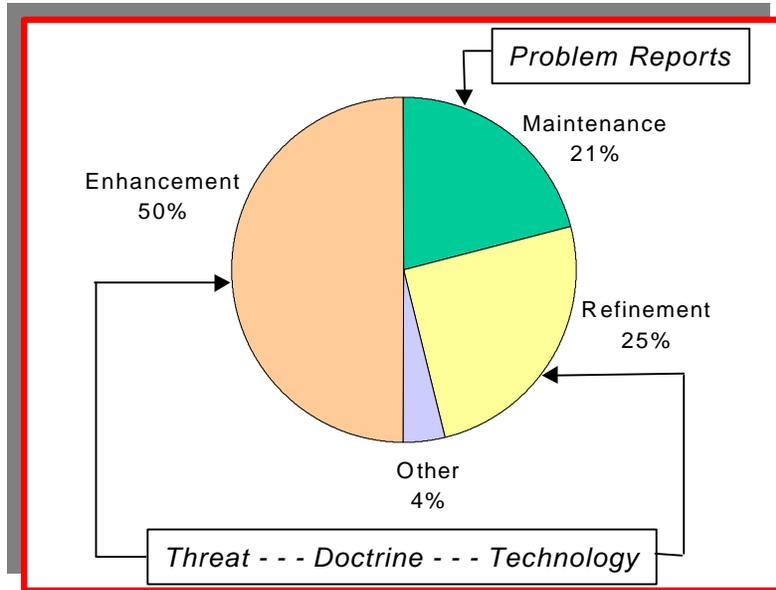


Figure 12-3. Causes of Software Changes [PIERSALL94]

During operational testing, supportability evaluations concentrate on software code, supporting documentation and implementation, computer support resources, and life cycle process planning. Due to its impact on software support, spare computing capacity is also examined. The four areas the Air Force Operational Test and Evaluation Center (AFOTEC) evaluates for supportability are illustrated in Figure 12-4. For example, maintainability evaluations consist of questionnaires that concentrate on the specific characteristics of a maintainable system, such as consistency, modularity, and traceability. Software supportability is evaluated by the developer when the documentation and source code are initially baselined (usually during initial integration test and evaluation) and then periodically until the completion of software development. The information gained during integration testing helps the developer build more maintainable software.

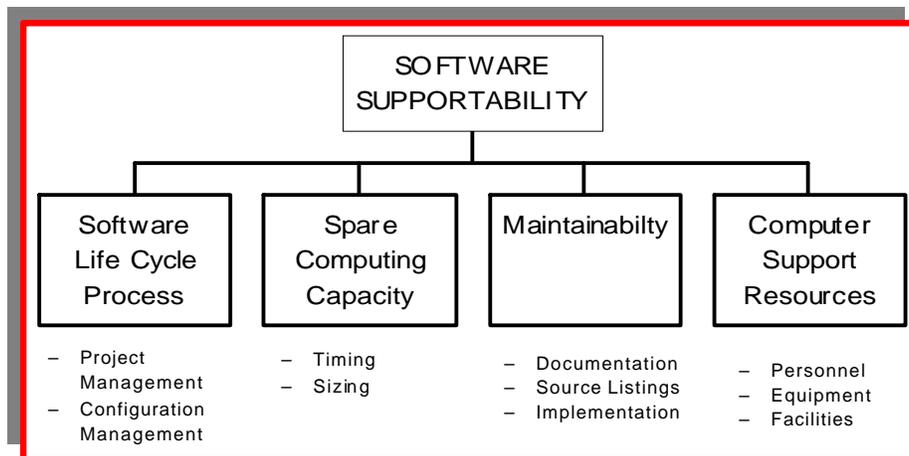


Figure 12-4. AFOTEC Software Supportability Evaluation Areas

### 12.2.3 Software Support Issues

In theory, *software never wears out!* It has none of the physical properties found in hardware which the forces of Nature and the operational environment can play on to cause physical systems to decline in performance. When pieces of hardware begin their life span, they often have a high failure rate (defects per unit time) because of problems created during manufacturing. Those pieces that survive the “infant mortality” period usually have lower failure rates (often for many years) until components begin to wear out. At this point, the failure rate begins to climb again. This trend, called the “*bathtub curve*” by hardware engineers, is true for all hardware systems — whether an automobile, a radio, or a computer.

While software does not wear out in the physical sense, *it does deteriorate!* There are some interesting similarities and differences to be seen when the software failure rate is superimposed on the bathtub curve. Like hardware, new software usually has a fairly high failure rate until the bugs are worked out. At which point failures drop to a very low level. Theoretically, software should stay at that low level indefinitely because it has no tangible components upon which the forces of the physical environment can play. However, after software enters its operational life (during PDSS), it undergoes changes to correct latent defects, to adapt to changing user requirements, or to improve performance. These changes make the software failure rate curve steadily begin an upward journey. Hardware deteriorates for lack of maintenance, whereas *software deteriorates because of maintenance!* [GLASS92] By making changes, software maintainers often inadvertently introduce “*side-effects*” causing the defect rate to rise, as illustrated in Figure 12-5.

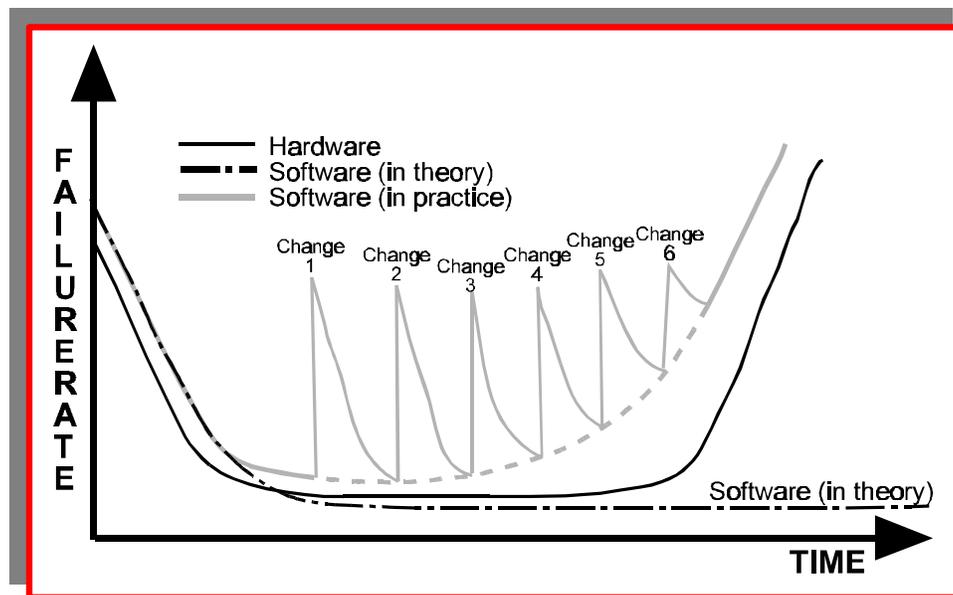


Figure 12-5. Bathtub Curves for Hardware and Software

Although side-effects can be quite complex, most are caused by one thing — *there are no spare parts for software!* When software fails the part causing the failure cannot simply be replaced with a spare. When software fails, from defects inserted during maintenance, often the only way to correct for the cause of failure is through design modification. Every time the design is modified

it weakens the original structure (or how the modules work internally and with each other) and eventually the software begins to fall apart. Undisciplined maintenance (or that performed in the field under stressed conditions) frequently compounds the problem. Maintainers, struggling against time to make corrections, modifications, or adaptations to new requirements, often compound the defects created by the last generation of maintainers. In the rush to get the product to impatient users, they take short cuts — exacerbating the software's deterioration. Problems arise when there is a failure to modify the design when patches are made (causing the design and code to be out of synch). Problems also stem from a failure to update documentation or a failure to use modern concepts of design and programming in the initial development.

Most of the problems associated with software support can be traced directly back to deficiencies in the way the original software product was planned, managed, and designed. *Lack of sound software engineering discipline, control, and attention to the design of modular software architectures during development translates into software support problems resulting in excessive maintenance costs.* Some classic software support issues include:

- Lack of requirements traceability;
- The evolution of software versions or releases that are difficult or impossible to trace [*the evolution of changes that are not documented*];
- Unavailability of the software development toolset (compilers, loaders, etc can have impacts);
- Impossible to understand code [software understandability usually increases as the number of software modules increases];
- Documentation that is nonexistent or of such poor quality that it is useless [documentation must be understandable and consistent with the source code to be of value]; and
- Inflexible software not designed to accommodate change [unless the architecture allows for change, modifications to the software are difficult and defect-prone]. [PRESSMAN92]

This last point is, perhaps, the most critical deficiency. The software architecture should carefully address abstraction, encapsulation, and information hiding to minimize dependencies. By separating computational and operational details from interface calls, and by maximizing use of object-oriented design, the software can be easily modified. Modifications can occur during development and during post-deployment operation with less risk of introducing unwanted side effects.

Many factors play in the supportability equation. An undisciplined, poorly managed development process where design, coding, and testing were conducted with carelessness negatively impact the support task. Design characteristics that affect software supportability include:

- Design complexity (including related attributes of software size, structure, and interrelationships);
- Stability and flexibility of the design itself;
- Adequacy of documentation to support PDSS;
- Completeness of the software development effort; and
- Extent and implementation of configuration management practices for both operational and support software. [SHUMSKAS92]

Other factors within the development environment that impact software supportability include:

- Availability of qualified software personnel,
- System structure understandability,
- Ease of system handling,
- Use of standardized programming languages,
- Documentation structure standardization,
- Test case availability,
- Built-in debugging mechanisms,
- Delivery of the original development SEE to the maintenance organization, and
- Availability of appropriate computer hardware to conduct maintenance activities. [PRESSMAN92]

---

## 12.2.4 COTS Software Support Issues

Software support includes support of government-developed software, contractor-developed software, and commercial-off-the-shelf (COTS) software. Issues to consider when supporting COTS software include:

- The acquisition agent must acquire appropriate documentation and data rights, licensing, and subscription services (such as options to purchase or escrow proprietary information) which allows the Government to support the software if contracted support becomes unfeasible.
- The software support activity (SSA) must maintain appropriate licensing and subscription services (vendor field change orders and software releases) throughout the life of the system.
- COTS resources must not be altered so as to preclude contractor logistics support or void licensing or subscription services.
- The supporting command must provide logistics support and contract for subscription services required to update and maintain COTS assets. It must also evaluate operational and logistic impacts of change due to subscription-related hardware and software upgrades.
- The operating command must provide a technical review of proposed changes during upgrades and changes to COTS assets. It is responsible for evaluating effectiveness and mission impact of changes due to subscription-related software upgrades.

---

## 12.3 Planning for Support Success

In recent years, early planning for software support has become a main DoD acquisition priority. Learning from costly past mistakes, the early F-22 planners wanted to make their weapon system a “*maintenance man’s dream*,” according to Colonel John Borky, former director of ATF Avionics. [BORKY90] Colonel Ron Bischoff, Air Logistics Center (ALC) system program manager for the F-22, remarked, “*We are practicing [with F-22 support and maintenance design] what we always said we were going to do, but never did...[Before] it was a build it, then fix it, way of doing business.*” [BISCHOFF91] In the past, the system program manager responsible for supporting the aircraft was not assigned until late in the design process. Support problems were not addressed until after an aircraft was deployed and maintenance problems occurred. *F-22*

*planners specified support requirements early*, which then became part of the RFP. Colonel Bischoff explained that planning for support success was accomplished by making it *a source selection criterion that support issues be addressed during the design stage*.

Colonel Bischoff remarked that writing and maintaining software for the F-22 will be a much larger task than for any other aircraft in history. He explained, “*The F-22 leads DoD’s list of the most complex software projects, with a projected 7 million lines-of-code.*” [BISCHOFF91] F-22 planners enforced consistency and completeness by mandating the use of Ada for all F-22 software systems. By using Ada, all F-22 software engineers are forced to use common terminology, from ground support systems to operational flight programs. Bischoff claims, “*That was a major step forward. Ada makes the software much more supportable because it is written in much clearer text. Lack of documentation killed us in the past.*” F-22 planners also enforced the use of a common Ada software engineering environment that provides uniform development tools for all the software development team members.

To augment F-22 support success, Air Force and contractor personnel will work together as integrated product teams (IPTs) to maintain F-22 software. To plan for this requirement, the ALC F-22 system program office (SPO) has ALC software personnel involved *shoulder-to-shoulder* with contractors so they will understand what is being done and why. Colonel Bischoff boasted, “*We’re already planning for the first update to the operational flight program within a year or two after the first F-22 rolls off the production line!*” [BISCHOFF91]

As discussed above, decreases in productivity during PDSS can be tied to *increases in software complexity* the longer it is in the support phase. The more modifications made to the software (especially to a poorly engineered product), the more complex it becomes with corresponding increases in the introduction of defects. These exponential increases in effort (and cost) are mainly the product of *poorly engineered software*. [PRESSMAN92] Therefore, *planning for supportability upfront is a major determinant of software development success*. Software not developed with maintenance in mind can end up so poorly designed and documented that total redevelopment is actually cheaper than maintaining the original code. With today’s shrinking defense dollars, *failure to make software maintenance a design priority* would not only be poor management on your part, but could very well result in an inability to support your product.

---

### 12.3.1 Software Support Cost Estimation

The variety and undefined scope of future changes throughout the software life cycle make estimating support costs one of the most difficult — yet most important activities to consider due to its impact on the DoD budget. Most software estimating models estimate software support costs; however, the types of activities and the costs included in their estimates vary significantly from model to model. Most parametrically-based software support estimating models provide a top-level approximation of sustaining engineering and support requirements. *They do not produce estimates that can be reliably used alone as the basis for a software support budget or similar purpose*. Once software has been transferred into a support environment, changes to the software (especially major changes or additions to basic software functionality) must be estimated using software models calibrated to the redevelopment environment.

---

## 12.4 Software Reengineering

The concept of reengineering is relatively new within the software development community. The motivation behind reengineering is to get a handle on the ever-growing software maintenance burden. The rapid evolution of software and hardware technology over the past 20 years has left DoD with a legacy of millions of lines of failure-prone code, written in a conglomeration of languages, running on a hodgepodge of incompatible hardware.

“Reengineering” is defined as the examination and alteration of a software system to reconstitute and re-implement it in a new form. The reengineering process involves recovering the design from an existing application and using that information to reconstitute it to improve its quality and decrease maintenance costs. While reengineering re-implements existing system functions in a better, more efficient manner, new or improved functions are often also added. [PRESSMAN92]

---

### 12.4.1 Reengineering Decision

Reengineering of old, worn-out or obsolete code is often economically justified. The lengthy DoD acquisition process often takes a decade or more for large software-intensive systems to come on line. By industry standards, military software is often obsolete before it enters the field, at which point a 20-year operational life usually lies ahead. The cost of maintaining software over its extended life can be from two to 10 times as much as the cost to initially develop it. The decision to reengineer software is often one based on the premise to “*pay now or pay much more later.*” [PRESSMAN92] There are basically three situations when reengineering is beneficial. These include:

- When the existing system has become technologically obsolete and must be replaced;
- When the existing system has deteriorated to the point where it has severe technical problems; and
- When it might be expedient to upgrade the existing system. [SNEED91]

You may choose to reengineer if you reach the conclusion that it is better to *pay now*, rather than waiting to *pay-much-more-later*. “*Paying now*” is what Perry calls *avoiding the rathole syndrome*. He defines a *rathole* as the dark place where software maintainers throw their money with no possibility of return on investment. He equates the legacy software rathole with the old car rathole. In the short-term, it is cheaper to fix your old car than it is to buy a new one. But over an extended period, the out-of-pocket expense for parts and labor to patch your old clunker will cost you more without increasing its resale value than investing in a new car. He also explains that software maintenance ratholes are like ratholes in the woods. Once you plug one up, the rat digs another. [PERRY93] Reengineering, when cost effective, can provide you with a way to plug up all your ratholes and have a new system with all the bells and whistles your user desires. It may well be the long-term, low-cost solution to your software maintenance problems. The reasons to reengineer include:

- To reduce maintenance costs,
- To decrease defect rate,
- To convert to a better language or hardware platform,
- To lengthen the life-span, and
- To enable changes in the user's environment.

Another reason to reengineer is often based on the logical migration of the system. Since the system has to be dramatically changed anyway, it might as well be upgraded to more current technology. Your reengineering decision must be based on a thorough feasibility analysis of the costs, benefits, and risks involved in continued patching (if possible) versus redevelopment (starting from scratch) versus reengineering. This analysis is based on a calculation of the target system's expected lifetime and the comparison of reengineering costs with the costs of a new development. A rule of thumb is, *reengineering is a viable alternative when the cost to reengineer is not more than 50% of the cost to redevelop*. It may also be determined that it is too expensive to reengineer the entire system. [SNEED91] Studies conducted by major industry software developers indicate that *80% of the problems are caused by 20% of the software*. [JONES91] Therefore, in some cases, only 20% of a system may need reengineering.

Reengineering is only one of several options you have as a maintenance manager in fulfilling your user's needs. These options must be weighed against each other. Factors to consider, in addition to cost, include:

- The added value of reengineering relative to the value of a new system and the value of the old system.
- The risk of reengineering relative to the risk of a new development and the risk of doing nothing.
- The life expectancy of the existing system relative to the time required to reengineer it and the time required to redevelop it. [SNEED91]

---

## 12.4.2 Reengineering Process

Reengineering involves a number of engineering concepts. How these engineering tasks make up the reengineering process and relate to each other is illustrated on Figure 12-6. These methods include:

- **Reverse engineering.** This is the process of examining an existing software system to abstract its design and fundamental requirements. It is also the end-to-end process used to understand the existing software well enough to change it. It is the opposite of *forward engineering* (the traditional way software is developed).
- **Forward engineering.** This is the set of engineering activities that use the products and artifacts derived from legacy software and new requirements to produce a new target system.
- **Restructuring.** This is the process of reorganizing or transforming an existing system from one representation form to another at the same relative abstraction level, while preserving the subject software's external functional behavior. Most commonly applied, restructuring involves taking (perhaps unstructured) software and adding structure.
- **Redocumentation.** This is the process of analyzing the software to produce support documentation in various forms, including users' manuals and reformatting the system's source code listings.

Other software support engineering concepts not illustrated on this figure include: retargeting, the process of transforming and hosting (or porting) existing software to a new hardware configuration; and source code translation, the transformation of source code from one language to another or from one version of a language to another version (e.g., translating COBOL-74 to COBOL-85). [OLSEM93]

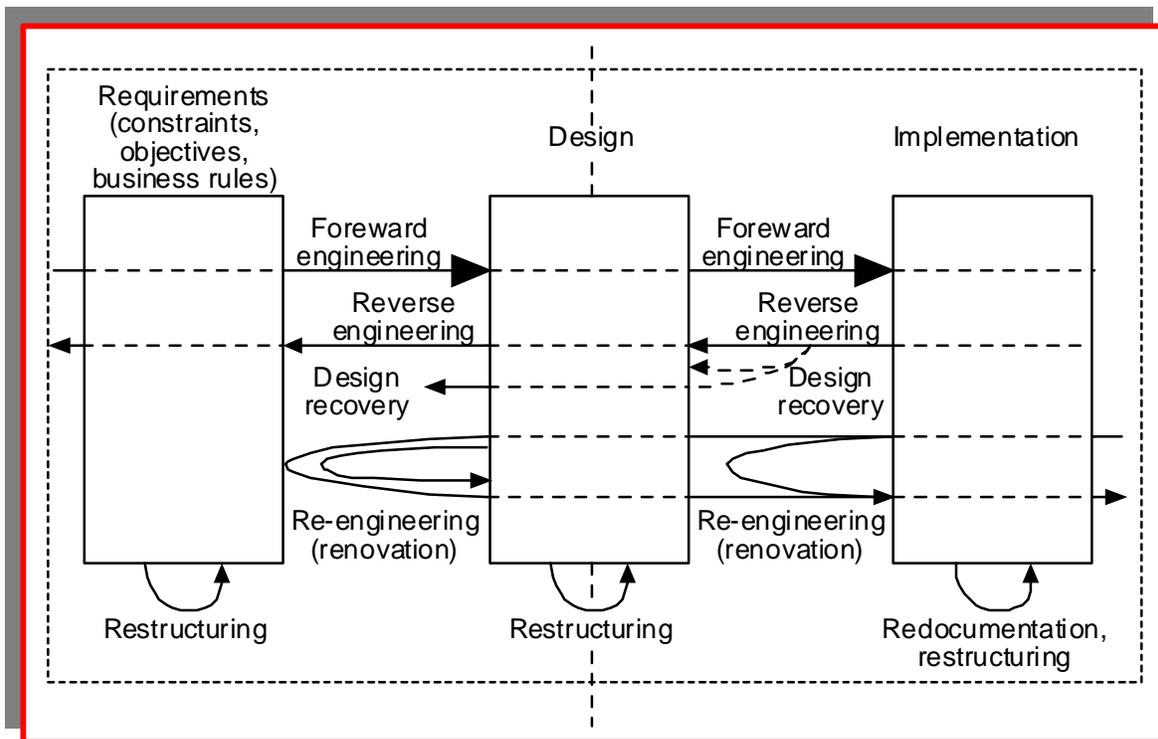


Figure 12-6. Relationship Among Support Engineering Tasks [GLASS92]

Your reengineering strategy can be integrated into your domain engineering approach with profitable results. This may involve looking at reengineering as a total migration plan that can involve a number of incremental steps — rather than as a single event at one point in time. A comprehensive model of the reengineered system can also be developed and maintained while the implementation of the plan is staggered as resources permit. [For more information see Feiler's *Reengineering: An Engineering Problem*, SEI Special Report, 1992.]

## 12.5 Logistics Support Analysis (LSA)

It has not been the practice for contractors to perform formal LSAs for software acquisitions. Even for weapon systems, most LSA is confined to hardware. A complete, well-rounded approach to assuring that software is supportable has not been formally developed. In 1991, at the 26th Annual International Logistics Symposium sponsored by the Society of Logistics Engineers (SOLE), a paper was presented by A.G. Johnson and T.A. Haden, from the United Kingdom Ministry of Defense Army Electronics Branch. This paper included a Software Supportability Checklist, modeled after those used for hardware. It is reproduced in Table 12-1 for the benefit of program managers and contractors who desire to give additional attention to the LSA of their software.

	SOFTWARE SUPPORTABILITY FACTOR	DESCRIPTION
1	Maintainability	Requirement for a Maintenance Task Analysis (MTA)
2	FTA, FMECA	Requirement for Fault Tree Analysis (FTA) and Failure Modes and Effects and Criticality Analysis (FMECA) to be performed to functional death
3	Defect Rate	Requirements to state a contractual target defect rate per lines of code over an agreed period including confidence limits
4	Failure Identification	Design to provide features that achieve failure detection and location times
5	Failure Snapshot	Design to provide features that achieve failure detection and location times
6	Tool Kit	Provision of User/Maintainers software tool kits to aid failure location
7	Loading and Saving Data	Design to allow loading or saving data in specified times
8	Configuration Identification	User/maintainer able to identify the configuration status (version) without accompanying documentation
9	Exception Handling	Design to allow exception handling to preclude failure conditions from aborting software during operations
10	Support Policy Constraints	Use Study to include what the software must do and not do
11	Support Maintenance Policy	Support specific maintenance policies and manpower ceilings and skill level availability to be stated
12	Software Support and Maintenance Categories	Categories of software support and maintenance to be stated
13	Media	Proposed media must: (a) suit the environmental requirements, and (b) be acceptable as a consumable item
14	Media Copying	Simplify copying and distribution
15	Media Marking	To allow physical and internal marking; safety critical items to be separately marked
16	Packaging	Media packaging to be consumable, reusable, and robust
17	Handling	Media to require no special precautions and meet Use Study requirements
18	Storage	Media to require no special precautions or facilities and meet Use Study requirements
19	Transportation	Media and packaging to require no special requirements
20	Training, User	User training required to detect failures and invoke exception handling
21	Training, Support	Support training required to detect and locate failures and invoke exception handling
22	Publications	User and Support publications will be required
23	Definitions	The Requirement must include contractually agreed upon definitions of: incident, fault, failure, defect, reliability, and failure categories
24	Resources	Cost estimates must be sought for software maintenance
25	Test Tools	Contractor-owned and maintained software test tools and documentation must be provided
26	Test Tool Access	Access to test tools to be provided to software support personnel
27	Incident/Failure Reporting	Incident and failure reporting to be available

Table 12-1. Software Supportability Checklist

---

## 12.5.1 LSA on the F-22 Program

From the outset, the F-22 program has enhanced and implemented Integrated Product Development (IPD) and Integrated Weapon System Management (IWSM). Specifically, the program has always integrated software engineers and logistics personnel throughout all Integrated Product Teams (IPTs). In addition, the Life Cycle Software Support (LCSS) IPT was created to influence software design for supportability and to build a specification that describes the software support concepts for the life of the weapon system. Personnel from product centers, support centers, customers, and contractors work together on the IPTs. Thus, program decisions related to software development and support are jointly determined. Since each IPT is composed of representatives from all disciplines, life cycle impact is always considered as are plans for future software support. Because a software support facility is still some years away, support decisions are analyzed to determine future impact. LCSS IPT personnel ensure that decision makers are briefed on the consequences of support decisions.

**NOTE: See master's thesis, Guidelines for Ensuring Software Supportability in Systems Developed Under the Integrated Weapon System Management Concept, by Johndro and Butts, Air Force Institute of Technology, December 1993.**

Instead of the traditional LSA process, the approach the LCSS IPT used was a combination of parametric models, analogy, expert opinion, and top-down analysis. By analogy, they compared the overall size of the effort to past fighter aircraft designs. The F-22 will have at least twice as much software on board the aircraft as any fighter currently in DoD. Also by analogy, they initially estimated that the magnitude of average software block change would be approximately 10% of the total source lines-of-code.

The F-22 also employs data tables to implement highly volatile functions and reduce the magnitude of block changes. Key design decisions were made to move potential areas of change out of the source code and into the lookup tables. Potential change areas are now isolated to easily modifiable code blocks instead of locked in algorithms. For example, most Pilot-Vehicle Interface (PVI) functions have traditionally been hard-coded into the software, but on the F-22, many of these functions will be implemented using data tables. By expert opinion, the IPT leads in charge of software development estimated that the use of data tables would reduce the block change size by about 50%. Once the overall effort was estimated, parametric analyses of each subsystem provided estimates for schedule and personnel requirements. Three software cost estimation models (*SEER*, *REVIC*, and *CostMotio*) indicated varying degrees of schedule and personnel requirements. The IPT leads then selected a single model to continue a top-down analysis of the large subsystems.

Software support facility cost estimates were also based on expert opinion and analogy. Subject matter experts, such as lab managers and integration and test leads, suggested space and equipment requirements based on F-22 development efforts from which equipment cost estimates were derived. Personnel cost estimates were based on the current annual rates for government and contractor software development personnel when applied to parametric analysis results. Similar data, which had been previously collected from the F-14, F-15, F-16, and F/A-18 programs, were used for comparison purposes. The comparative data corroborated facility and personnel cost estimates.

An inherent difference between hardware support and software support is that hardware support is based on the finished product, while software support must mimic the development process. Hardware support must use the tools necessary to repair a finished product, not tools required to build a one. Software support, on the other hand, must use tools functionally identical to those used during the development process. To determine F-22 software support requirements, the LCSS IPT started their LSA program by identifying the tools used to create the software. They then developed a software supportability database based on MIL-STD-1388A. Although traditional LSA process was not used to assess software supportability, LSA Record (LSAR) data items are incorporated in a database. Both software maintainers and developers reviewed and commented on the initial database design, as defined by the LCSS IPT. To populate the analysis database, data are collected from the software development IPTs during each development phase. The database is segregated by computer software configuration items (CSCIs) and by development cycle phase. This data collection relationship will continue throughout production and post-production support. The software supportability database implements the intent of the LSA process at the highest level to accommodate software support requirements.

The LCSS IPT will generate several guidance documents for the F-22 program. Specifically, IPT personnel will also prepare and publish a Post-Deployment Software Support Concept Document (PDSSCD) as an executive summary of the processes, plans, and procedures to be used in post-deployment support. System Program Office (SPO) personnel will update the F-22 Computer Resources Life Cycle Management Plan (CRLCMP) to reflect software support decisions published in the PDSSCD. Contractor personnel will prepare and deliver a Computer Resources Integrated Support Document (CRISD) to define the processes, plans, and procedures for software support. Additionally, contractor personnel will prepare an Integrated Weapons System Support Facility (IWSSF) development specification to define and itemize the resources needed to implement the CRISD.

The LCSS IPT takes a very proactive role in the Software Product Evaluation (SPE) process. Since the SPE process keeps software support personnel closely associated with software development teams, support personnel are able to influence design and improve supportability. For example, LCSS IPT and Charles Stark Draper Laboratory personnel developed Document Evaluation Guidelines to help evaluate hundreds of software documents generated by the program. These guidelines provided developers with criteria to follow during initial product development. They also form the basis for document SPEs. The LCSS IPT personnel also train government and contractor personnel on the SPE process so that documents are prepared according to the same guidelines against which they will be evaluated. This dramatically improves the first-time approval rate of software documents.

---

## 12.6 Continuous Acquisition and Life Cycle Support (CALS)

CALS is a collection of standards for developing, storing, and communicating products, parts specifications, and other engineering technical information electronically. The purpose of CALS is to get *on-line* engineering data and specifications for high-tech equipment in a DoD-wide database for easy retrieval and updating throughout a weapon system's life. *All new weapons systems should include a "delivery-in-place" capability.* This is the electronic capability to

deliver *on-request* all contractually required information. Although the data resides with the contractor, DoD retains the rights to the data and must be provided access to it on a fee-for-service basis.

## 12.7 Managing a PDSS Program

You employ the same tactics for successful management of PDSS as those employed for new-starts and ongoing software developments. The solutions to your PDSS development problems are the also same software engineering practices used throughout other phases of the life cycle. Unfortunately, you are at the mercy of the acquirer and initial developer who may have burdened your program with problems. Planning and execution of software support must begin during the concept exploration phase and continue until the system is removed from the inventory. The key areas that must be addressed are illustrated in Figure 12-7. These key areas consist of processes, products, and support systems.

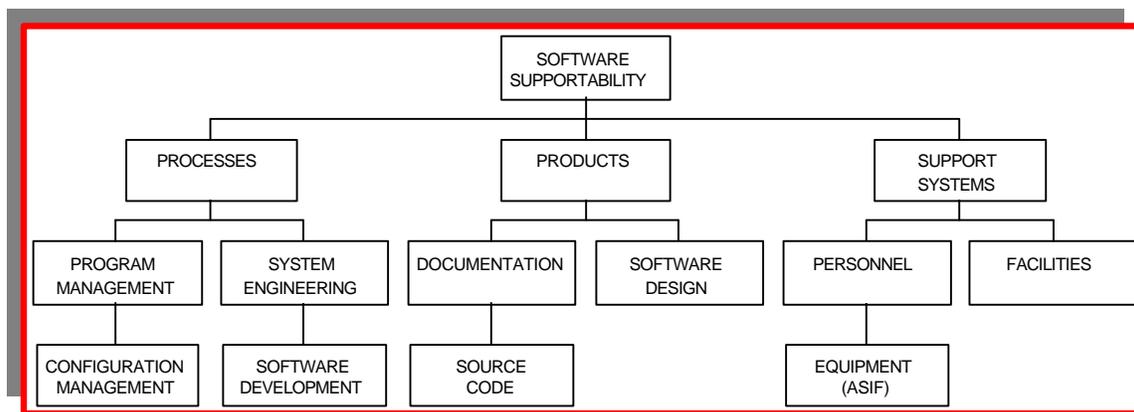


Figure 12-7. Post-Deployment Software Support Key Considerations

Life cycle support strategies typically span the support spectrum from sole source contractor to full government organic, with each strategy presenting different advantages and disadvantages needing evaluation. *A high level IPT consisting of the operational user, the Program Executive Officer (PEO), and the acquisition agent must make the support decision prior to Milestone I.* This focuses attention on the software support process and allows the acquisition agent to begin planning for it earlier in the program.

To effectively manage and control software development and to ensure software supportability requires that we incorporate *measurement* in the developer's decision making and reporting processes. With measurement, we can monitor the development effort, gain early insight into potential problem areas which can negatively impact the PDSS task, and we can ease verification procedures.

Support processes are the most important element for management, control, and improvement of software support. The key processes that must be captured and recorded are *program management*, *configuration management*, *systems engineering*, and *software development*. The key products essential to PDSS are documentation, source code, and a description of the software design and test procedures. The baseline for PDSS activity is the delivered products from the initial development. The effectiveness of PDSS is governed by the usability and descriptiveness of the

delivered documentation. Source documents for these essential products are contract Contract Data Requirements Lists (CDRLs), Contract Line Item Numbers (CLINs), and the CRISD. Support systems include the people, facilities, tools, and equipment needed to perform the maintenance task.

The following are key management activities to remember for PDSS success:

- Determine your life cycle support strategy early,
- Remember that software support is actually software redevelopment,
- Ensure adequacy of contractor software development processes during source selection,
- Identify supportability requirements and objectives in system requirements documents and Statements of Objectives,
- Specify required documentation and verification methods in the appropriate CDRLs,
- Identify necessary software development and support tools in CRISD, and
- Establish a Computer Resource Working Group (CRWG) IPT.

---

## 12.7.1 Computer Resources Integrated Support Document (CRISD)

The CRISD is the key document for software support. It defines facility requirements, specifies equipment and required support software, and lists the number of required personnel, skills, required training. It contains information crucial to the establishment of the SEE, its functionality, and limitations. It is a management tool that accurately characterizes the SEE's evolution over time.

The CRISD is a product of the software development process. As the pyramid in Figure 12-8 illustrates, the bottom tier, "*early analysis and supportability decisions during design*," is the cornerstone in achieving supportable software. Support requirements and characteristics must be specified at the beginning of the design process so that supportability features are an integral part of system development. This permits identification of support resources as they are needed. It also enables the identification and documentation of the software development tools used. The CRISD is a *living document* that reflects the development configuration and test/integration environments. Thus, the CRISD lays the foundation for PDSS and is essential in reducing software life cycle support costs.

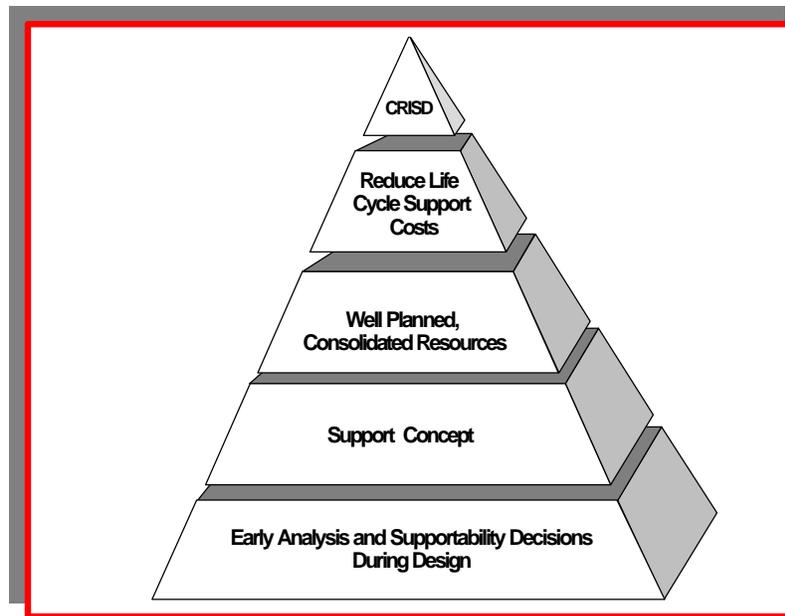


Figure 12-8. Computer Resources Integrated Support Document (CRISD)

**NOTE: See Volume 2, Appendix I for a discussion on the importance of the CRISD and how it is being implemented on the F-22 Program.**

## 12.8 Addressing Software Support in the RFP

Supportability is one of the most important issues to address in the RFP. Your RFP must require that offerors plan for supportability by stipulating that the software be developed with a *supportable architecture that anticipates change, uses accepted protocols and interfaces, and has documentation consistent with the code*. This can only be achieved during initial software development and must be addressed upfront in the development contract. The higher the quality of the initial system, the easier it will be to support. Therefore, the offeror's approach to supportability must be a major source selection criterion.

In 1990, a survey of over 100 businesses and technical people conducted by the Air Force Scientific Advisory Board revealed that contractors do not perceive supportability and maintenance as important factors for winning software development contracts. This study showed that software contractors believe cost, performance, and schedule are the Government's main concern. This perception by contractors must be changed. The primary vehicle to help institute this change, especially for your program, will be the emphasis given to supportability in your RFP. [PDSS90]

One method to emphasize the importance of supportability is to require pre-award competitive software exercises (e.g., prototypes and demonstrations). These compute-offs can be followed by multiple awards for design demonstrations. The design demonstrations are based on evolving, value-added prototypes that ultimately converge into a fully supported product at the end of the initial procurement. To make this acquisition strategy effective, the developing contractor(s) must be required to support previous, but evolving, versions of the product the same way a PDSS maintainer would. The prototype developers are required to select design(s) that promulgates a

low-cost, efficient solution with minimal side effects on software maintenance. The subsequent Engineering and Manufacturing Development (EMD) development contract is awarded to the most supportable design.

Whether a contractor maintains the software, or it is transitioned to in-house government maintainers, the maintainer must have the original developer's SEE and other essential tools for proper code maintenance. The following deliverables must be required:

- Data rights to make and install changes,
- Source code and documentation adequate to understand the code,
- Computer resources (SEE, computers, compilers, etc.) needed to modify the source code and produce object code,
- Equipment and support software to test the subject code, to diagnose problems, and to test solutions, enhancements, and modifications,
- Equipment needed to distribute and install the new software,
- A workable system to identify problems, resolve new requirements, and manage the support workload, and
- Skilled personnel to perform required maintenance tasks. [ALC89]

The way you structure the RFP to acquire and develop your initial software can profoundly impact the availability and usefulness of the required support environment. Therefore, *you must require that all offerors describe their plans for supportability as part of their proposal submission.* To ensure a prospective offeror's systems engineering and software development processes adequately address the supportability of software, it is imperative you carefully evaluate the offeror's software development processes during source selection. To do so, three major areas must be addressed:

- **Software Development Plan (SDP).** Require the submission of a SDP with offerors' proposals that states how they intend to ensure their development process addresses supportability relative to the systems engineering process. This plan is evaluated during source selection.
- **Capability Evaluation.**
- **Instructions to Offerors (ITO).** The ITO and source selection evaluation criteria must specifically address those areas you consider critical processes. The evaluation criteria should describe what is required of the offerors' proposal and how it will be evaluated. The Aeronautical Systems Center has developed an RFP template which provides general and specific guidance on preparing the RFP for software-intensive systems. [You might also consider requiring that offerors address the software supportability instructions contained in Volume 2, Appendix S, *Software Source Selection* as part of their proposal response. In addition, Appendix S provides a shopping list of RFP statements, definitions of software supportability metrics, and a sample "Instructions to Offerors" that addresses supportability.]

## 12.8.1 Specifying Supportable Software

Acquiring supportable software also requires the specification of software product performance requirements. The major instruments contained within the RFP are illustrated in Figure 12-9.

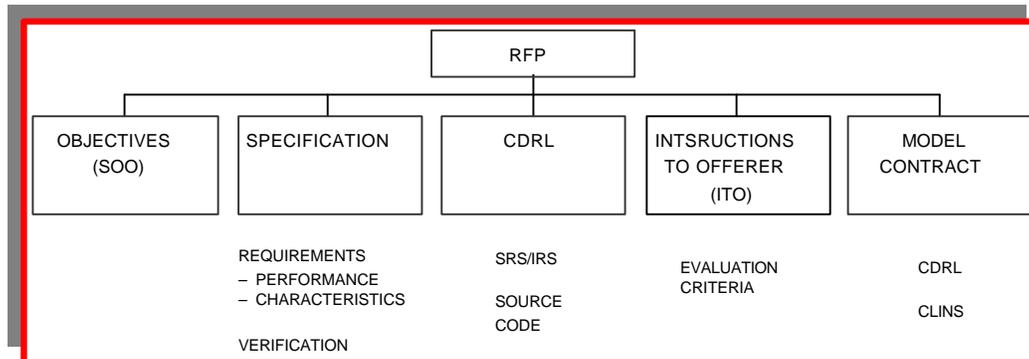


Figure 12-9. Acquisition Instruments

### 12.8.1.1 Statement of Objectives (SOO)

The SOO defines an objective for efficient, life cycle software support consistent with total system requirements. The SOO states that software supportability requirements and support characteristics are to be managed as an integral part of system development.

### 12.8.1.2 Specification Practices

In accordance with the Perry Memo, your RPF must describe *what you want* to procure — not how to design or build it. You can provide top-level system specifications or requirements documents to satisfy the “*what you want.*” These specifications can only contain performance requirements and key system characteristics — they can not contain design solutions or detailed design requirements. You can describe the methods you intend to use to verify that system requirements have been achieved. For each performance requirement, a corresponding method of verification should be provided. Therefore, specify *key software supportability characteristics* along with corresponding verification methods in the system specification or requirements document. You should specify the following characteristics to ensure your software acquisition is supportable:

- **Module size.** Module size affects software supportability. Module size [a typical computer software component (CSC)] should generally not exceed 100 source lines-of-code (SLOC).
- **Complexity.** Application complexity affects software supportability. One generally accepted complexity measure is McCabe’s Cyclomatic Complexity Measure, which should not exceed 10 for a given module.
- **Programming language.** The use of widely-accepted, higher order programming languages to develop software enhances software supportability.

- **Spare memory.** The availability of installed spare memory improves software supportability. Spare memory permits the incorporation of enhancements and the correction of latent deficiencies. The effect of spare memory on supportability was calculated for the E-3 AWACS where two similar radars were delivered with 9% spare and 34% spare memory, respectively for the APY-1 and the APY-2. Measurements revealed a 3 to 1 difference in cost and schedule impact when making the same change to both E-3 radars.
- **Spare computer throughput.** The availability of installed spare throughput affects the software supportability by permitting the incorporation of enhancements and the correction of latent deficiencies.
- **Spare computer system input/output.** The availability of installed spare input/output affects software supportability.
- **Other parameters.** These include Halstead Metrics, SAIC, Inc.'s *Quality Profile Metrics for Supportable Maintainable Software*, the IEEE's software reliability concepts as they may apply to specifying a required level of software supportability, and Rome Laboratory's, *Framework Implementation Guidebook, RL-TR-94-146*, August 1994.

### 12.8.1.3 Documentation

Because software is unlike any other product, the only way to visualize and understand it is through its documentation. Without accurate, high-quality documentation, software cannot be understood. In essence, documentation is the most important aspect of software support. Documentation delivery requirements specified in CDRLs include:

- Software and Interface Requirements Specifications,
- Software and Interface Design Descriptions,
- Database Descriptions,
- Software Product Specifications,
- Source Code Listings,
- Test plans/descriptions/reports,
- Software Development Plans,
- Software programming manuals,
- Software users manuals, and
- Software maintenance manuals.

The specific criteria for government acceptance of software design information should be clearly specified in the appropriate CDRLs (DD Form 1423) items. This includes the verification methodology, composition of the verification teams, and quantitative thresholds that must be met or exceeded. Offerors should be encouraged to provide alternative verification approaches.

### 12.8.1.4 Life Cycle Software Support Strategies

To ensure the contractor's process for developing the software addresses information and documentation management, quality, and verification procedures, typical life cycle support strategies available for source selection include the following.

- **Sole source (original contractor).** The original contractor is awarded the software support contract. The processes, products, and support system are already in place at the contractor's facility and typically are the same as those used during the development.
- **Competitive (support equipment provided).** A competitive contract is awarded and the processes, products, and support systems are either transferred from the original contractor facility to the competing contractor or the items are duplicated. The original contractor can also be a competitor.
- **Organic/contractor mix.** The Government and the contractor share responsibility for software support. Each agent is assigned a percentage of the software to be supported. Typically the Government and contractor are collocated. The processes, products, and support system are relocated to a government support center or the items are duplicated. Manning of the effort is shared by the Government and either the original contractor or a competitive contractor.
- **Organic.** The Government assumes responsibility for software CSCIs. The processes, products, and support systems are relocated to a government support center or duplicated. Support processes are executed by government organic personnel.

---

## 12.9 References

- [ALC89] *Supportable Software Acquisition Guide*, First Edition, San Antonio Air Logistics Center, October 1989
- [BASSETT95] Bassett, Paul, "Maintenance is a Misnomer," *Software Magazine*, December 1995
- [BISCHOFF91] Bischoff, Col Ron, as quoted in "Design and Planning Make High-Tech F-22 Easy to Maintain and Support," *Aviation Week & Space Technology*, July 15, 1991
- [BOEHM81] Boehm, Barry W., *Software Engineering Economics*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981
- [BORKY90] Borky, Col John M., as quoted in "ATF Avionics Met Dem/Val Goals, Providing Data for Flight Tests," *Aviation Week & Space Technology*, September 24, 1990
- [GLASS92] Glass, Robert L., *Building Quality Software*, Prentice Hall, Englewood Cliffs, New Jersey, 1992
- [IEEE90] Institute of Electrical and Electronic Engineers, Inc., *IEEE Standard Glossary of Software Engineering Technology*, IEEE STD 610.12-1990, New York NY, December 10, 1990
- [JONES91] Jones, Capers, *Applied Software Measurement: Assuring Productivity and Quality*, McGraw-Hill, Inc., New York, 1991
- [OLSEM93] Olsem, Michael R. and Chris Sittenauer, "Terms in Transition: Reengineering Terminology," *CrossTalk*, Software Technology Support Center, Special Edition, 1993
- [PDSS90] "Report of the Ad Hoc Committee on Post-deployment Software Support," US Air Force Scientific Advisory Board, December 1990
- [PERRY93] Perry, William E., "Don't Pour Money Down Rat Holes that Infest Your Budget," *Government Computing News*, December 6, 1993
- [PIERSALL94] Piersall, COL James, "The Importance of Software Support to Army Readiness," Army Research, Development, and Acquisition Bulletin, January-February 1994
- [PRESSMAN92] Pressman, Roger S., *Software Engineering: A Practitioner's Approach*, Third Edition, McGraw-Hill, Inc., New York, 1992
- [SHUMSKAS92] Shumskas, Anthony F., "Software Risk Mitigation," G. Gordon Schulmeyer and James I. McManus, eds., *Total Quality Management for Software*, Van Nostrand Reinhold, New York, 1992
- [SNEED91] Sneed, Harry M., "Economics of Software Reengineering," *Software Maintenance: Research and Practice*, Volume 3, John Wiley and Sons, Ltd., 1991