

Chapter 14

The Management Challenge

Contents

14.1 Chapter Overview	14-3
14.2 Seize the Opportunity	14-4
14.2.1 Embrace the Software Vision: Make It Work for You	14-4
14.2.2 Make the Commitment to Excellence	14-6
14.3 Program Management Challenge	14-7
14.3.1 Managing a New-Start Program	14-7
14.3.1.1 Lessons Learned	14-7
14.3.1.2 Earned Value Management System (EVMS)	14-9
14.3.2 Managing an On-going Program	14-11
14.3.3 Managing a PDSS Program	14-12
14.3.4 Determining If Your Program Is In Trouble	14-12
14.3.4.1 What to Do With a Troubled Program	14-17
14.3.4.1.1 Take a Hiatus	14-18
14.3.4.1.2 Increase Your Schedule	14-18
14.3.4.1.3 Reduce the Number of Requirements to be Satisfied	14-18
14.3.4.1.4 Improve Your Process	14-19
14.3.4.2 What To Do With a Program Catastrophe?	14-19
14.3.4.2.1 Abandoning the Catastrophe	14-20
14.4. The Continuous Improvement Challenge	14-20
14.4.1 Measurement	14-20
14.4.2 Baselines	14-21
14.4.3 Benchmarks	14-21
14.5 Your Management Challenge	14-22
14.6 References	14-24

14.1 Chapter Overview

In this chapter, you will learn that, in addition to detailed technical insight, a high-level, big picture perspective is needed for successful software acquisition management. Closely tied to the technical competence needed for good management is the confidence that you are being supported. From the governing documents, sources for schools and tools, through the white papers and acquisition program examples, to the guidelines and philosophical insights on selected subjects found in the Appendices of these guidelines, you have a wealth of practical information to assimilate and digest. The Vision for Software expressed here encompasses the promise that you have a software infrastructure to support your management activities. Your challenge is to make use of these resources (e.g., tools, schools, repositories, programs, technology, professional workforce) to ensure the success of your program as it supports the DoD mission.

There are three categories of acquisition management which apply to DoD software programs. If you are managing a new-start program, your challenge is to follow the advice found in these Guidelines with the objective of attaining customer satisfaction, quality, economy, efficiency, and process improvement. If your program is a smooth running on-going effort, your goal is to improve your process. This is accomplished through rigorous self-assessment and the introduction of new processes, tools, improved methods, and advanced technologies.

If your on-going program is in trouble, you must first assess the extent of your problems. The cure for a troubled program can only be achieved by identifying the causes of your problems, removing them, and preventing their recurrence. While you are focusing on a cure, there are some band-aid efforts you can employ to get back on track until the sources of problems are identified and remedied. As Benjamin Disraeli, former British prime minister, proclaimed, “He who gains time gains everything.” Increasing your schedule will gain you time, productivity, and decrease defects, as will reducing the number of requirements to be satisfied. If you determine, however, that your program is beyond repair through detailed cost/benefit analyses, do not think twice, stop it dead in its tracks!

Throughout these Guidelines the underlying theme has been quality through process improvement. Your program is never so successful that it cannot be made better. Process improvement means there is a definable, measurable process to improve. The bottom line for improving software development is measurement. You must be able to determine where you stand today to determine how to improve for tomorrow. This includes establishing a baseline and measuring progress from that point in time. Measurement should include all facets of your process for which improvement is possible, and for which metrics can be applied as a normal part of everyday activities. Benchmarks are useful for comparing your effort with other successful programs, and for setting realistic goals for improvement.

These Guidelines are your opportunity for success. They provide you with information you can use to enhance and support your management efforts. You will find no secrets here — only better ways of doing business, based on common sense and learning from our mistakes. Remember that success can only be obtained through simultaneous efforts. Your challenge is to take what you have learned here and direct it to your given program. With sustained constancy and sound management decisions, you will help achieve the Vision for Software.

14.2 Seize the Opportunity

In an interview with the *Washington Post*, General Colin L. Powell described how to achieve success.

“There are no secrets to success; don’t waste your time looking for them...Success is the result of perfection, hard work, learning from failure, loyalty to those for whom you work, and persistence. You must be ready for opportunity when it comes.” [POWELL89]

As a software-intensive system acquisition manager, these Guidelines provide you with a significantly improved opportunity for success. Managers must aggressively look for better ways to increase productivity, reduce costs, and improve product quality. This motivation comes by learning from failure, loyalty to those for whom you work (and those who work for you), a determination to achieve quality through persistent work, and a desire for perfection. Software engineering is the basis upon which this opportunity resides. The proven paradigms and methods presented in these Guidelines allow you to take full advantage of this technology.

A software acquisition infrastructure has been established to provide a framework for applying software engineering technology to your program. This infrastructure was designed to be flexible, to take advantage of software state-of-the-art and from management practices that work and will provide you the greatest opportunity for success. However, as Mosemann explains,

“Software problems will not be solved purely by policies, by standards, or even by education. An integrated DoD software technology strategy that includes both software management and technology initiatives will make a much larger difference in resolving DoD’s current and future software problems.” [MOSEMANN93]

Mosemann warns that institutional changes simply do not happen by mandate; there has to be *buy-in* at every level. Your commitment to turn around software acquisition problems is *the most important buy-in of all!* To do this, all of you who are affected by the infrastructure must participate in its evolution. Incentives must be provided to our industrial partners, along with education and training for our managers, practitioners, and team members. Measurement is an integral part of the framework, as cost/benefits must be understood and quantified. Ways to exploit our valuable cache of legacy software assets through reuse and re-engineering must be explored. Our systems must be open and have well-defined generic architectures so they can evolve and endure. Our customers must be enlightened and our suppliers must be certified. If you are ready for success, *the opportunity is yours!*

14.2.1 Embrace the Software Vision: *Make It Work for You*

Although we have turned the tide of failure and experienced some success, we must never be satisfied with the status quo. We must be dedicated to never-ending software process improvement. The Vision is to *continuously improve software quality and predictability through diligent application of engineering discipline*. The way we plan to achieve this Vision is a twofold approach of which you are an integral part. One facet of the Vision encompasses the institutionalization of software engineering practice throughout all software development programs DoD-wide. Having read these Guidelines, you have a solid foundation from which to make your

contribution to this Vision by institutionalizing the practice of software engineering within your program. Because education and training are key to achieving the Vision, you, as a software manager, must place high priority on keeping your software professionals trained and educated in software engineering discipline.

The other facet of the Vision is the establishment of a software engineering infrastructure. As illustrated on Figure 14-1, this infrastructure is based on a concept created by the Japanese some 20 years ago — the *house-of-quality*. Used as a total quality management (TQM) communication tool, the structure shows how all the pieces of a system are needed to build and provide support to the whole. The importance of the pillars to each other in supporting the ceiling (the Vision) is an interrelated and co-related set of methods, techniques, technologies, and organizations. Your side of the equation — using software engineering discipline to build your pillar — needs parallel balance and support from the infrastructure to achieve the Vision for the whole. Here, the purpose is ultimately to help you and other software professionals by actively addressing software issues surfacing within your programs. Part of the infrastructure is the gathering of a software work force within which communication, learning, and education are cultivated and where exchange of corporate knowledge flows freely through technology transfer and the sharing of lessons-learned. Infrastructure resources are dedicated to continuous improvement through working groups and agent (software organizations) support. The infrastructure also brings consistency, repeatability, and currency to software development through the implementation of software policies and management plans.

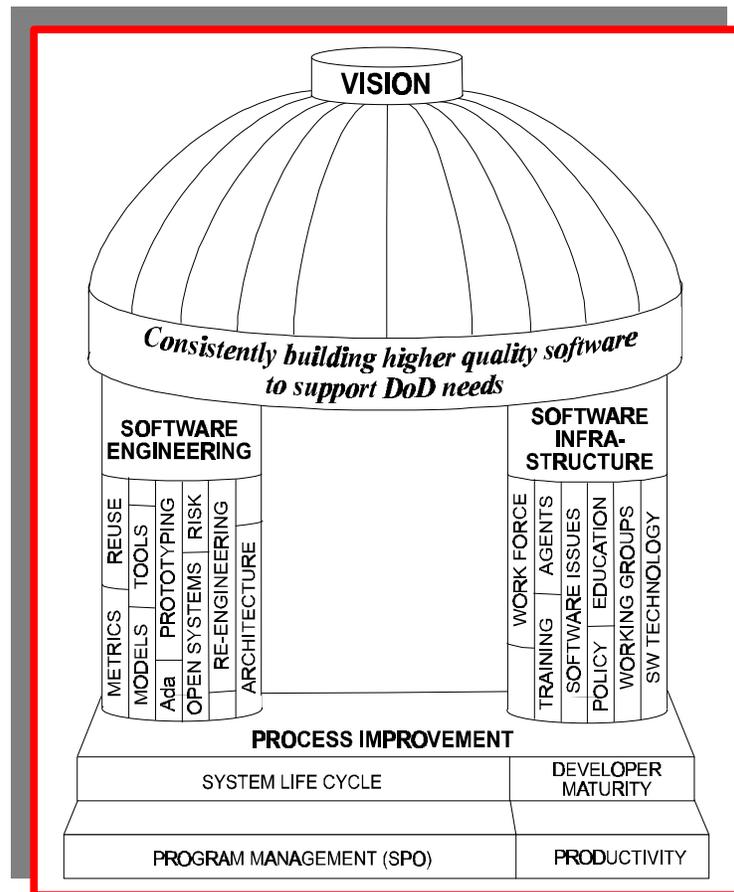


Figure 14-1. Vision for Software

You must realize that the software engineering for which you are responsible is a relatively young discipline. At first it may seem little more than a *hodge podge* of rules, methods, and disparate pieces of information. The Vision provides the unifying theme that brings the ingredients for success into a single software engineering framework. The separate pieces, such as metrics, reuse, models, tools, prototyping, open systems, re-engineering, risk management, and architecture are interrelated and merged into an integrating foundation permitting us to build quality into our software through the application of technology and practical know-how. This discipline provides an understanding of what it is we are trying to do, and how to go about doing it.

At the foundation of the Vision, holding it all together and making it work, is process improvement. The commitment and contribution to this concept must come from your program office, your contractors, your colleagues' programs, and your counterparts within the software infrastructure. The Vision is to select those contractors who have in hand a predictable, mature, software development process with demonstrable, built-in mechanisms for its continuous improvement.

“Nothing is of greater importance in time of war than knowing how to make the best use of a fair opportunity when it is offered.” — Niccolo Machiavelli [MACHIAVELLI21]

In the heat of fighting your daily management battles, remember the Vision. As you are engineering your software, a software infrastructure provides you the opportunity to do your job better, to help you succeed. This infrastructure is comprised of policies to keep you in tune with initiatives to improve the way we develop our software and manage our acquisitions. DoD and Service policies and instructions are there to make sure we build uniformity and predictability into our systems. Organizations within the infrastructure are there to assist in implementing reuse and metrics, to evaluate our tools and our contractors, and to research new technologies to improve the way we do our jobs. Training programs and software courses provide the opportunity to advance our skills, and to increase our understanding of the software engineering discipline. Make use of the tools, the repositories, the education, the programs, the technology, the agents (labs, institutes, and centers), and the software work force discussed throughout these Guidelines. They are offered as your fair opportunity; use them to your best advantage. Remember, you are not on a solo mission — an extensive team is there to back you up.

14.2.2 Make the Commitment to Excellence

Embracing the Vision also means making a commitment to excellence — excellence in management and excellence in your product. People are conditioned to believe defects in software are inevitable. For the foreseeable future, software will continue to be built by humans; however, humans are believed to have a *built-in defect factor*. Most commercial software development organizations allow 20% of sales for scrap, rework, warranty repairs, complaint handling, service, test, and inspection. [SCHULMEYER92] *Human errors* cause this waste. To eliminate waste in software development, we must concentrate on preventing the errors and defects that plague us. There must be a commitment to defect reduction for all programs.

In his book, Quality is Free, Cosby explains that a defect which is prevented has no cost. It needs no repair, no examination, no explanation. [COSBY79] Defect prevention techniques can include peer inspections, process action teams, Cleanroom engineering, software quality assurance (SQA), early testing, commercial-off-the-shelf (COTS), reuse, prototyping, and demonstrations. A serious

defect prevention program is comprised of combinations of techniques, each chosen for its ability to prevent a different class of defects.

14.3 Program Management Challenge

We are aware that all our readers are not at the same stage in their acquisition programs. The issues with which you are challenged and how you deal with them will, therefore, differ. Your program may be a new start, may be many years into a long acquisition cycle, may be running smoothly, or plagued with the problems common to software acquisition and development projects. You might be tasked with the maintenance of newly delivered software, or software that has been in use for 20 years or more. Or you might be supporting a combination of new Ada software that has to run with older non-Ada applications, or a combination of COTS or non-developmental item (NDI). These different management challenges are addressed in the following sections, or in the chapters cited, and are listed as the following:

- Managing a new-start program,
- Managing an on-going program,
- Managing a PDSS program, and
- Managing a troubled program.

14.3.1 Managing a New-Start Program

Every new program can benefit from the lessons learned on previous programs. Additionally, it is important to set up a means to accurately determine program progress. The means required by DoD 5000.2-R for major projects is the Earned Value Management System .

14.3.1.1 Lessons Learned

If you are managing a new development, follow these Guidelines as completely and fully as possible. Your challenge is to apply proven software engineering practices and streamlined procurement methods to your acquisition program. They should reflect the concept that we are interested in not only buying product, but process. We have attempted to assemble a variety of lessons-learned to give you insight into what works and what does not. The following are lessons learned that deal with software acquisition and development from various sources. Don't repeat history. Take the time to review these lessons periodically. See how they may apply to your project. Then take the steps necessary to avoid the problems they describe. The descriptions below contain the outlines only, take time to download and read the entire documents.

James H. Dobbins, a Professor of System Management at the Defense Systems Management College and Course Director for the Management of Software Acquisition Course, wrote an article titled "Software Acquisition Management in a Nutshell" for the January-February 1994 issue of Program Manager magazine (available at www.dsmc.dsm.mil/pubs/pdf/pmpdf94/dobbins.pdf). Though the Mil-STDs cited are out of date, the remainder of the article is as valid today as when it was written. In it, Mr. Dobbins discusses eight cost-proposal blinders that prevent the program manager from recognizing software risks. He then covers twenty-three sources of software risk and uncertainty. These are followed by twenty-nine rules for managing

software acquisition. He also includes seventeen rules to keep software contracting from “biting” you. Software metrics is the next area covered, including eleven implications of the software complexity metric. Dobbins concludes by describing the importance of managing software testing.

The Software Program Managers Network (SPMN) reported twenty-four categories of Lessons Learned – Current Problems in their SPMN Software Development Bulletin Number 3, 31 December 1998. It is available at www.spmn.com. The categories are:

- Systems Engineering
- Safety and Security
- Continuous Risk Management
- Requirements Management
- Planning and Tracking
- Products Required for Delivery
- Interface Management
- Visibility
- Cost Estimation
- Schedule Compression
- Rework
- Reuse
- Architecture
- Quality
- Retaining Technical Staff
- Approach to Achieving Higher SEI Rating
- Integrated Product Teams
- Configuration Management
- Test
- Metrics
- Cost of Maintenance
- Software Development Environment/Tools
- Contract/RFP Management
- Commercial-off-the-Shelf (COTS) Products.

The SPMN also identified 16 Critical Software Practices for Performance-Based Management (available at www.spmn.com/critical_software_practices.html), categorized in three areas. They are:

- Project Integrity
 - Adopt Continuous Project Management
 - Estimate Cost and Schedule Empirically
 - Use Metrics to Manage
 - Track Earned Value
 - Track Defects Against Targets
 - Treat People as the Most Important Resource

- Construction Integrity
 - Adopt Life Cycle Configuration Management
 - Manage and Trace Requirements
 - Use System-Based Software Design
 - Ensure Data and Database Interoperability
 - Define and Control Interfaces
 - Design Twice, Code Once
 - Assess Reuse Risks and Costs
- Product Stability and Integrity
 - Inspect Requirements and Design
 - Manage Testing as a Continuous Process
 - Compile and Smoke Test Frequently.

It is also sound advice to research lessons-learned from programs similar to yours within your domain to arm yourself with as much knowledge as possible. Never forget, software acquisition is one of the toughest management battles you will ever fight. Be armed, prepared, and well-trained. You must always plan, measure, track, and control with quality as your number one goal.

Another major issue to address in your new acquisition is to make sure the new software you are building today is not a maintenance nightmare tomorrow. Well-engineered software must be reliable, understandable, and modifiable. The maintenance burden of tomorrow's legacy software will be lightened by the success of your efforts today.

14.3.1.2 Earned Value Management System (EVMS)

DoD 5000.2-R discusses the EVMS in the section on Cost Performance (3.3.5.3), and in Appendix VI. One of the stated purposes of EVMS is to “Provide an adequate basis for responsible decision making by both contractor management and DoD Component personnel by requiring that contractors’ internal management control systems produce data that: (a) indicate work progress; (b) properly relate cost, schedule, and technical accomplishment; (c) are valid, timely, and able to be audited; and (d) provide DoD Component managers with information at a practical level of summarization.”

The EVMS is more than the formulas often associated with earned value. It includes thirty-two mandatory procedures grouped in five categories:

- Organization
- Planning, Scheduling, and Budgeting
- Accounting Considerations
- Analysis and Management Reports
- Revisions and Data Management

Use of the EVMS is required on significant contracts and subcontracts within all acquisition programs. Significant contracts include research, development, test, and evaluation contracts and subcontracts with a value of \$70 million or more or procurement contracts and subcontracts

with a value of \$300 million or more (in FY 1996 constant dollars). Compliance with EVMS criteria is not required on firm fixed price contracts, time and materials contracts, and contracts which consist mostly of level-of-effort work. However, all program managers may want to review the EVMS criteria and select for implementation the procedures that are important to their program.

The Defense Systems Management College (DSMC) Earned Value Management Gold Card, shown as Figure 14-2, covers what several individuals usually characterize as the EVMS. Additional EVMS information is available on the DoD EVMS home page at www.acq.osd.mil/pm/.

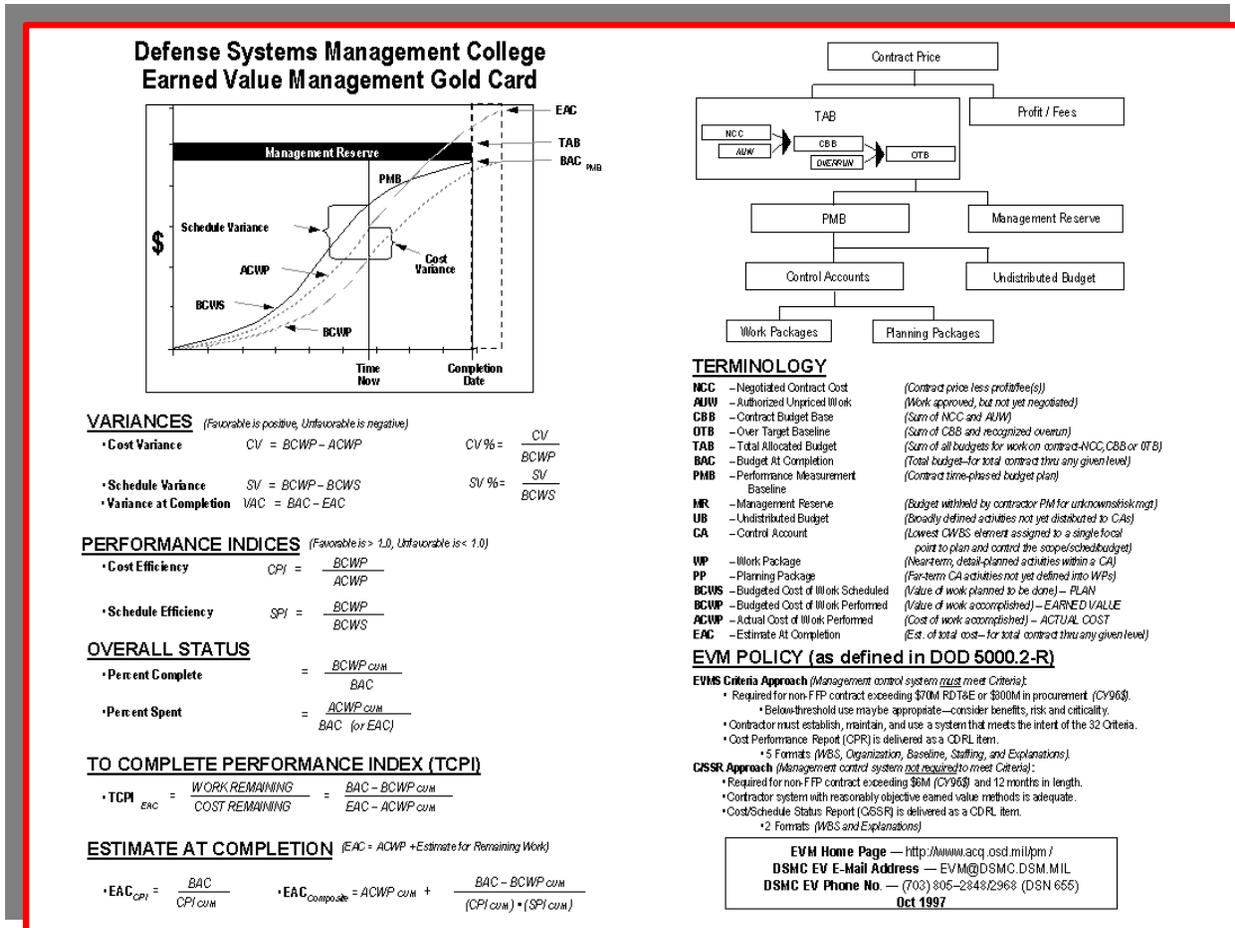


Figure 14-2. DSMC Earned Value Management Gold Card

One caution about using Actual Cost of Work Performed (ACWP), Budgeted Cost of Work Performed (BCWP), and Budgeted Cost of Work Scheduled (BCWS) to determine program status. The summary earned value metrics can be misleading. They may indicate a program is at the half-way point when it is only at the 10 percent point on the critical path. Critical path only earned value metrics must be examined. Earned value metrics can also be misleading at the start of a program, suggesting that the same variance seen on an early process block or work breakdown structure element will reoccur for every remaining process block. Some organizations have modified their tracking to account for these problems.

14.3.2 Managing an On-going Program

Today, there are very few major new-start software-intensive acquisitions in DoD. Therefore, most of the readers of these Guidelines are either managing on-going programs, or programs in post-deployment software support (PDSS) [discussed in Chapter 12, *Software Support*]. If your program is on track, do not be tempted to sit back and rest on your laurels. As Brigadier General Marshall explained:

“Success is disarming. Tension is the normal state of mind and body in combat. When the tension suddenly relaxes through the winning of the first objective, troops are apt to be pervaded by a sense of extreme well-being and there is apt to ensue laxness in all of its forms and with all of its dangers.” [MARSHALL47]

No one has ever reached a state of perfection in software development. If your program has successfully achieved its first objectives, do not become disarmed by success. There is danger in relaxing your management efforts through a sense of well-being. Your challenge is to relentlessly improve your process through an investment in resources and effort to increase and mature your development capabilities.

Old habits, doing things the way they have always been done, are major inhibitors to innovation, growth, and progress. You must relentlessly improve your process and your management skills. The time to initiate improvement is not when things are broken, but when they are working well. Robert J. Kriegel, a performance psychology pioneer explains:

- To ride the wave of change, move before the wave hits you.
- Always mess with success.
- Speed kills quality, performance, and innovation.
- The best time to change is when you don't have to.
- Playing it safe is dangerous.
- Get in the habit of breaking your habits.
- Round up your sacred cows and put them out to pasture.
- Stoke the fire, don't soak it; and,
- If it ain't broke, BREAK IT! [DRAKE93]

Transitioning a software development program into a mature, software production requires sound management practices, an unremitting obsession for process improvement, and a wise use of technology. Elevating your program's software quality and productivity is neither simple nor cheap, but well worth the investment. New methods can include transitioning to Ada, adding new tools, or altering development methods and practices. As you have learned throughout these Guidelines, there are many practices, processes, methods, tools, and technologies that offer improvements. These transitions are not always free and may involve some initial schedule and cost impact. You and your contractor(s) should evaluate together the relative merits of the improved practices that seem to offer the greatest potential for reducing overall cost and schedule risk. They must also be assessed for their ability to decrease defects and increase the quality of your product. Software technology transitions are an opportunity for significant gains in quality and productivity, but poorly planned and executed transitions can result in serious program setbacks.

Successful implementation of “*new ways of doing business*” in on-going programs cannot be the exclusive province of either the contractor or the government program manager. Since these *best practices* were not foreseen at contract award, contract documents will not reflect their use and *may* (or may not) need to be modified. Generally, contractors will need to absorb some initial unplanned cost, and the Government will need to concede to some schedule delays. However, if technology transition planning is performed successfully, cost and schedule investments will reap substantial dividends.

The key is to enlighten your customer — educate your contractor — gain a consensus about “*what to do*” and “*how to do it.*” *Be sure they read these Guidelines!* Take advantage of the infrastructure of support organizations that are doing a lot of the homework for you. They are there to evaluate your needs and advise you on how to proceed. *Remember the Vision; make it work for you and keep on pressing!*

14.3.3 Managing a PDSS Program

If you are managing a PDSS program, you employ the same tactics as new-start and on-going programs. Follow the software engineering discipline discussed in these Guidelines with the ceaseless goal of improving your process. This can include re-engineering part or all of your code to Ada, incorporating reuse and COTS for enhanced functionality, or restructuring your code so it is more maintainable and modifiable.

14.3.4 Determining If Your Program Is In Trouble

You most likely already know if your program is in trouble! Your developer is not providing orderly documentation, the software development plan is inadequate, or not being followed. Your program is over budget, behind schedule, and the user-discovered defect rate in delivered modules is above the acceptable range. These are not uncommon problems where a program is on its way to a near disastrous situation. Programs in trouble can run into delays and budget overruns of 200% to 300%, and, in some cases, must be abandoned. [BENNATAN92]

Most software engineering methodologies focus on *preventing* (not *correcting*) these types of problems. Preventing problems is always easier and less costly than solving them. As you have learned throughout these Guidelines, problems become more expensive the further into the development they are discovered. Once neglected, problems propagate into other areas of the development process, making them more difficult and costly to reverse. Your challenge is to determine if your program can be salvaged by enacting a radical change that adopts the ingredients for success found in these Guidelines.

NOTE: If you are not sure whether your program is in trouble, look at management metrics variances. If the current set looks “*abnormal*,” you are in trouble!

Before you can make a decision about a cure, you must first determine the cause of your program’s sickness and the severity of the disease. You must determine whether your program is so sick it should either be terminated, started over from scratch, or whether upgrading your technology and improving your process will provide sufficient remedy. To make this assessment, apply the

same software engineering discipline used to prevent problems. *The best way to identify and assess the severity of your problems is to go looking for them.* There are a few basic sources of problems common to almost all DoD software programs in trouble. These include:

- Software’s inherent complexity,
- Our inability to estimate cost, schedule, and size,
- Unstable requirements, and
- Poor problem-solving/decision making (which includes reliance on *Silver Bullets*).

Colonel Lyons noted some additional problems:

- Failure to recognize or accept that a software challenge exists,
- Questionable developer capability, capacity, and tools,
- Inadequate development process discipline; and,
- Failure to manage subcontracts. [LYONS91]

Cost, schedule, and quality problems associated with software products are merely symptoms of problems in the process that produced them. Defects, design errors, and major schedule slips are not the causes of problems — they are the *symptoms*. Behind the symptoms, something was done by someone during the creation or evolution of that activity that caused the problem. By analyzing the cause (e.g., of design errors) and concentrating your resources on the software process, you can determine what must be done to improve that process, and thus, to solve your problems. [ARTHUR93] To determine where in your development process the cause of your problems lie, you have to *quantify it*. To accomplish this, you must:

- **Define** your process,
- **Measure** your process and product,
- **Analyze** the metrics to determine deficiencies in your process and the quality of your product; and,
- **Institute** the software engineering practices and methods discussed in these Guidelines.

Process improvement implies there is some definable and measurable *process* to improve. In software engineering, all processes at each development phase are targets for improvement. There are also ancillary processes, such as configuration management, software quality, test and integration, in-process reviews, and formal peer inspections. Each of these ancillary processes supports your overall development process, and each can be improved.

To quantify your process, and thus improve it, you must have a *baseline*. This baseline is used as the measured starting point for each phase of problem solving. You must, therefore, become sufficiently organized to have a definable, quantifiable process that can be measured. [REIFER92] Once measurement data is collected, it must be pondered, analyzed, placed in a larger context, and woven into the fabric of where you have been and where you are going. *Measurement information must be transformed into “insight” for it to be meaningful.*

The following Software Program Managers Network “*Breathalyzer*” questions will give you a *quick-look* into the status of your program’s health. If at any time you cannot answer any of these questions or must answer one or more with a “no,” you should schedule an immediate program review.

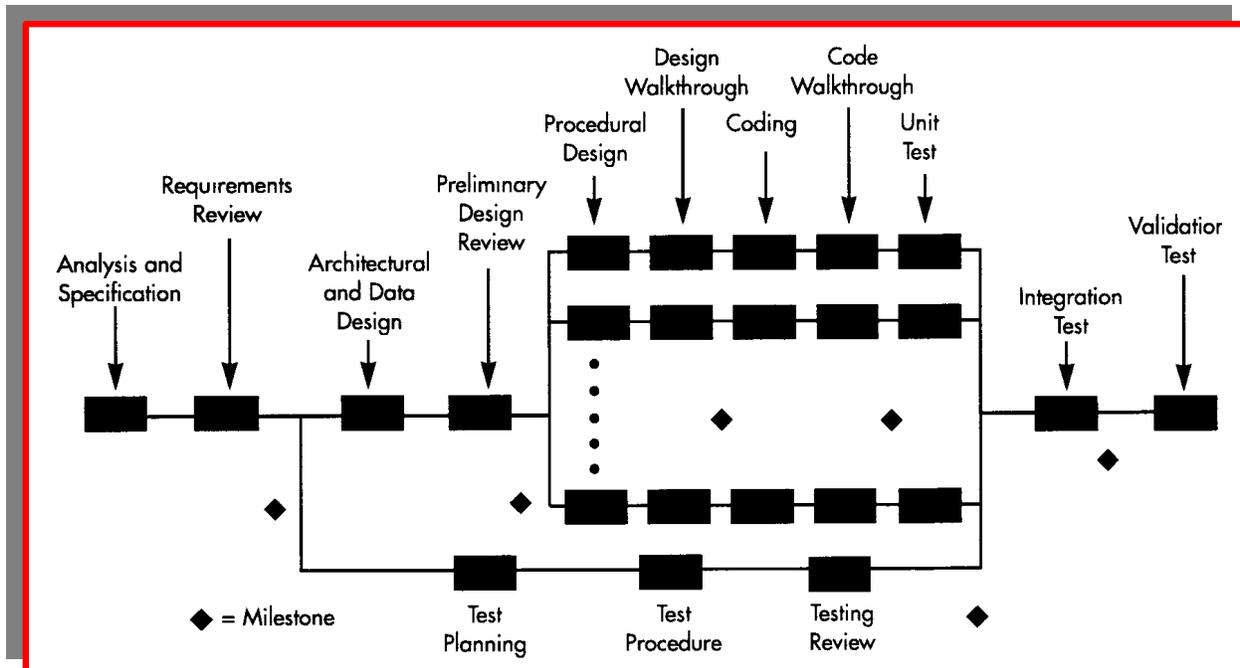


Figure 14-3. Activity Network Example

1. Do you have a current, credible activity network supported by a work breakdown structure (WBS)? As illustrated on Figure 14-3, an activity network is the primary means to organize and allocate work.
 - Have you identified your critical path items?
 - What explicit provisions have you made for work that is not on your WBS?
 - Does the activity network clearly organize, define, and graphically display the work to be accomplished?
 - Does the top-level activity network graphically define the program from start to finish, including dependencies?
 - Does the lowest-level WBS show work packages with measurable tasks of short duration?
 - Are program objectives fully supported by lower-level objectives?
 - Does each task on the network have a well-defined deliverable?
 - Is each work package under budget control (expressed in labor hours, dollars, or other numerical units)?

NOTE: A well-constructed activity network is essential for accurate estimates of program time, cost, and personnel needs, because estimates should begin with specific work packages.

2. Do you have a current, credible schedule?
 - Is the schedule based on a program activity network supported by the WBS?
 - Is the schedule based on realistic historical, quantitative performance estimates?
 - Does the schedule provide time for education, holidays, vacations, sick leave, etc.?
 - Does the schedule provide time for quality assurance activities?
 - Does the schedule allow for all interdependencies?
 - Does the schedule account for resource overlap?
 - Is the schedule for the next 3-6 months as detailed as possible?
 - Is the schedule consistently updated at all levels on Gantt, PERT, and critical path charts every two weeks?
 - Is the budget clearly based on the schedule and required resources over time?
 - Can you perform to the schedule and budget?

3. Do you know what you have to deliver?
 - Are system operational requirements clearly specified?
 - Are definitions of what the software must do to support system operational requirements clearly specified?
 - Are system interfaces clearly specified, and, if appropriate, prototyped?
 - Is the selection of software architecture and design method traceable to system operational characteristics?
 - Are descriptions of the system environment and relationships of the software application to the system architecture specified clearly?
 - Are specific development requirements expertly defined?
 - Are specific acceptance and delivery requirements expertly defined?
 - Are user requirements agreed to by joint teams of developers and users?
 - Are system requirements traceable through the software design?

4. Do you have a list of your Top Ten risk items? If so, what are they? [See Chapter 6, *Risk Management*, for more information on the Top Ten List.]
 - Has a Risk Officer been assigned to the program?
 - Are risks determined through established processes for risk identification, assessment, and mitigation?
 - Is there a database that includes all non-negligible risks in terms of probability, earliest expected visible symptom, and estimated and actual schedule and cost effects?
 - Are all program personnel encouraged to become risk identifiers?
 - Is there an anonymous communications channel for transmitting and receiving bad news?
 - Are correction plans written, followed-up, and reported?
 - Is the database of top-ten risk lists updated regularly?
 - Are transfers of all deliverables/products controlled?
 - Are user requirements reasonably stable?
 - How are risks changing over time?

5. *Do you know your schedule compression?* (Schedule compression is an indication of the percent by which this program is expected to outperform the statistical norm for programs of its size and class.)
- Has the schedule been constructed bottom up from quantitative estimates, not by predetermined end dates?
 - Has the schedule been modified when major modifications in the software take place?
 - Have programmers and test personnel received training in the principal domain area, the hardware, support software, and tools?
 - Have very detailed unit-level and interface design specifications been created for maximum parallel programmer effort?
 - Does the program avoid extreme dependence on specific individuals?
 - Are people working abnormal hours?
 - Do you know the historical schedule compression percentage on similar programs, and the results of those programs?
 - Is any part of the schedule compression based on the use of new technologies?
 - Has the percent of software functionality been decreased in proportion to the percent of schedule compression?

$$\text{ScheduleCompressionPercentage} = \left\{ 1.00 - \left[\frac{\text{CalendarTimeScheduled}}{\text{NormalExpectedTime}} \right] \right\} \cdot 100$$

(Nominal Expected Time is a function of total effort expressed in person months.)

For example, Boehm found that for a class of DoD programs of 500 person months or more:

$$\text{Nominal Expected Time} = 2.14 \cdot [\text{Expected Person Months}]^{.33}$$

(Nominal Expected time was measured from System Requirements Review to System Acceptance Test.) [BOEHM81]

NOTE: Attempts to compress a schedule to less than 80% of its nominal schedule aren't usually successful. New technologies offer additional risk in time and cost.

6. What is the estimated size of your software deliverable? How was it derived?
- Has the program scope been clearly established?
 - Were measurements from previous programs used as a basis for size estimates?
 - Were source lines-of-code (SLOC) used as a basis for estimates?
 - Were function points used as a basis for estimates?
 - What estimating tools were used?
 - Are the developers who do the estimating experienced in the domain area?
 - Were estimates of program size corroborated by estimate verification?
 - Are estimates regularly updated to reflect software development realities?

NOTE: Software size estimation is a process that should continue as the program proceeds.

7. Do you know the percentage of external interfaces that are not under your control?
 - Has each external interface been identified?
 - Have critical dependencies of each external interface been documented?
 - Has each external interface been ranked based on potential program impact?
 - Have procedures been established to monitor external interfaces until the risk is eliminated or substantially reduced?
 - Have agreements with the external interface controlling organizations been reached and documented?
8. Does your staff have sufficient expertise in the key program domains?
 - Do you know what the user needs, wants, and expects?
 - Does the staffing plan include a list of the key expertise areas and estimated number of personnel needed?
 - Does most of the program staff have experience with the specific type of system (business, personnel, weapon, etc.) being developed?
 - Does most of the program staff have extensive experience in the software language to be used?
 - Are the developers able to proceed without undue requests for additional time and cost to help resolve technical problems?
 - Do the developers understand their program role and are they committed to its success?
 - Are the developers knowledgeable in domain engineering — the process of choosing the best model for the program and using it throughout design, code, and test?
 - Is there a domain area expert assigned to each domain?
9. Have you identified adequate staff to allocate to the scheduled tasks at the scheduled time?
 - Do you have sufficient staff to support the tasks identified in the activity network?
 - Is the staffing plan based on historical data of level of effort, or staff months on similar programs?
 - Do you have staffing for the current tasks and all the tasks scheduled to occur in the next two months?
 - Have alternative staff buildup approaches been planned?
 - Does the staff buildup rate match the rate at which the program leaders identify unsolved problems?
 - Is there sufficient range and coverage of skills on the program?
 - Is there adequate time allocated for staff vacations, sick leave, training and education?

14.3.4.1 What to Do With a Troubled Program

The following sections offer suggestions on how to deal with a troubled program. If you decide, after you have thoroughly analyzed your process and identified the root causes of your problems, that your program is salvageable, you might consider a 3-6 month hiatus to institute the guidance found in this book and get your house in order. In addition, there are some *quick-fix strategies* (as opposed to long-term cures) you can employ if you are truly desperate. Quick-fix strategies include the following:

- Increase your schedule, and
- Reduce the number of requirements to be satisfied.

Improving your acquisition or development process can help to bring your project under control. Using a hiatus or quick fixes may bring immediate relief. But if these tactics work, *you must, must implement software engineering discipline to sustain any permanent improvement.* Remember, if *quick-fixes* work in the short-term, whatever in your process was causing your problems in the first place must be identified and rectified to sustain long-term improvement. If the root causes are not dealt with, your process will revert back to the problems you identified in your initial process assessment, on an order of magnitude worse. Of course, improving your process is the ideal solution.

14.3.4.1.1 Take a Hiatus

By following the software engineering practices discussed here, there is a significant probability you will gain back some or all of the hiatus time you invest in rescuing your program. The Air Traffic Control System in Canada is an excellent example. The program was in trouble. The contractor brought in a new manager whose first action was to *educate the customer*. Then, it was agreed that a hiatus would occur. It lasted 8 months. During this time many changes were made, including the adoption of the Rome Laboratory Software Quality Framework, acquisition of the Universal Network Architecture Services (UNAS) tool and the Rational Environment,TM and training of the software development team to a new mindset. As of this writing, the program is on schedule, at cost, and expects to recover most, if not all, of the hiatus time.

14.3.4.1.2 Increase Your Schedule

“More software programs have gone awry for lack of calendar time than for all other causes combined. Why is this cause of disaster so common?” — Frederick P. Brooks, Jr. [BROOKS75]

When you set your schedule to the minimum development time, effort is at its maximum to meet deadlines, but the number of defects is also correspondingly high. For the troubled (but salvageable) program, the temptation is to throw additional manpower at the problem and hold the schedule. *This will not work!* Instead of adding manpower in a desperate attempt to meet unrealistic schedules, extend the development time — without increasing or decreasing manpower. This can substantially reduce the effort (and associated cost) compared to what it would have taken to accomplish the task on the compressed schedule. In addition, the number of defects will drop. Regrettably, this is often not possible once the program is well underway. If your program is in the 12th month of a 12-month schedule, it is just too late to decide you should have planned in terms of a 17-month schedule. [PUTNAM92] Therefore, the sooner you decide to extend your schedule, the more likely it will be viewed as a credible move by those above you.

BEWARE! Adding extra staff to reduce schedule has often not worked. In fact, studies show that it can increase your schedule and increase your defects. Brooks’ well-known observation rings true: “Adding manpower to a late software program makes it later.” [BROOKS75]

14.3.4.1.3 Reduce the Number of Requirements to be Satisfied

If your program is in trouble, reducing the number of requirements to be satisfied will reduce development time, effort, the number of defects, and improve programmer productivity by reducing the size of the software to be developed. Software size can be reduced by paring the less essential functions from your software, or by deferring the development of separate functions not needed

for immediate delivery [i.e., strip the product (with the user's involvement) to the greatest number of essential functions that can be delivered in the time available].

14.3.4.1.4 Improve Your Process

Improving your process will reduce effort, cost, development time, and the number of defects. *This is the ideal solution because all management indicators improve.* Remember, improving your process takes time and should not be considered a *quick-fix*. It takes a long-term strategic commitment. The software development process must be measured for improvements that are both objective and management-oriented. Through measurement, you can determine which are the best strategies to employ for improvement. Choosing a strategy that is, indeed, better will result in software developed in less time, with less effort and money, and increased quality. Improvement requires the ability to answer questions such as:

- When in the software life cycle do errors/defects occur?
- When and how are errors/defects detected?
- What can be done to detect errors/defects earlier?
- When are errors/defects corrected and at what cost?
- What causes the errors/defects, and what can prevent the errors/defects that do occur?

Solving software development problems is not just the application of a set of tools, methods, or motivational campaigns. It requires commitment and a dedication to a *standard-of-excellence*. It is instituting a cultural change, and changing how your team members think and work. *It involves understanding and enhancing the human process that underlies software development at all levels.* Improvements can be achieved by changes in procedures, training of personnel, addition of tools, increased automation, and simulated faults insertion. [KENETT92] However, changing the way people think — cultural change — is the greatest challenge, and the key to your success with process changes.

Improvements only occur when rigorous software engineering discipline is applied to improve the human process. The human process must be organized around improvement objectives, properly supported by technology. Whatever it takes to cure your program, *there must be no turning back to the old ways of doing business!* DoD has seen its share of software fiascos. Your challenge is not to let a fiasco turn into a catastrophe, which occurs when we have not learned from our collective mistakes. [REIFER92] There are many techniques and lessons-learned for solving software problems. A few have been introduced here. Others are being discovered daily. Your challenge is to find out what will work for you and implement them! Remember Vince Lombardi's advice,

"The greatest accomplishment is not in never falling, but in rising again after you fall."
[LOMBARDI68]

14.3.4.2 What To Do With a Program Catastrophe?

A program catastrophe occurs when the only viable solution is program termination. Examples of circumstances leading to program termination are:

- The program appears to be technically infeasible; i.e., the work cannot be completed given the current state of technology.
- The costs to complete the program far exceed the utility of the final system, or the software will be so costly to operate that the user is better off never implementing it.
- The software will never be completed by a critical date, after which it will not be needed (e.g., an old system will be made to *make-do*).
- The performance quality or maintainability of the software is so bad that the software will be useless when completed — the best way to correct the problem is to start over.
- The software development process is so chaotic, and/or its personnel are so lacking in talent, as to provide no expectation of improvement within a reasonable time, at a reasonable cost.

14.3.4.2.1 Abandoning the Catastrophe

If your program is a catastrophe, you must recognize the problem *as soon as possible!* The nature of the catastrophe must be identified, and you should treat all efforts and costs expended to date as sunk. This decision is based on a cost/benefit analysis of completing the program, versus restarting it, versus canceling it. Contracting officials should be called in to see if any penalties or restitution to the Government is possible. Sunk costs must be completely disregarded on the common sense principle of *don't throw good money after bad*. [ROETZHEIM88]

NOTE: If you have to abandon your program, you should be praised for having the wisdom and fortitude to do so! But, remember, we are all still learning. So by all means, document your lessons-learned and send them to us at the address in the Foreword and last page of this Volume. The benefits of your insights may more than offset present financial losses by helping others to better understand the software management challenge.

14.4. The Continuous Improvement Challenge

As discussed throughout these Guidelines, to achieve continuous improvement you must establish a *software improvement culture* within your program. Everyone on the team (not just the software developers) must be committed to attaining the *standard-of-excellence* you set for your program. Because maintaining high standards requires persistent correction, process improvement should be a regular topic of discussion at all in-process reviews and peer inspections. It should also be on the agenda of working group and management meetings held at all levels. Process improvement metrics should be published, discussed, and assessed, the same as budget and schedule status metrics. Your management guidance must support a “*software process first*” philosophy. It is your responsibility to allocate the necessary resources to make improvement happen.

14.4.1 Measurement

The most critical factor in the process improvement equation is the collection of metrics. Software quality metrics must be collected and analyzed throughout software development. Once you specify a desired *standard-of-quality* for each element of importance to your program, achieved levels of quality must be measured at all predefined development milestones. These periodic measures will allow you to assess current quality status, predict the quality level of the final

product, and determine where quality is below desired levels. They give you the ability to zero in on problem areas on which process improvement activities can concentrate.

NOTE: See Chapter 13, *Software Estimation, Measurement and Metrics*, for a discussion on how to set up a measurement program.

14.4.2 Baselines

A key element in a measurement program is the baseline. It gives you a quantitative view of where you are today. It provides a framework for comparing your development program with historical data, and a context for improvement and innovation. It identifies strengths and weaknesses of the existing process, and helps to communicate them to all stakeholders. [HETZEL93] Baselines are usually established at key milestone points. A meaningful baseline for process improvement must go beyond productivity and quality measures. A complete baseline involves all measurable and improvable facets of the process. These include human resources, organizational structure, user environment, software engineering environment (tools, procedures, technology infrastructure), cost, schedule, funding, management practices — all those things that impact your process. [RUBIN93]

14.4.3 Benchmarks

Software benchmarking is a concept borrowed from the hardware manufacturing industry. Measurements (e.g., failure rates, specifications, time-to-market, cost to produce) are compared with those of competitors. Using these measures, understanding that your production process takes, for instance, 30% more time, costs 20% more, or produces 14% more latent defects than your competitors, makes you realize you are doing something wrong. These figures alone do not tell you what is wrong, they just tell you that you are doing something different that affects your competitive marketplace position.

“Benchmarking is a method for establishing baselines by which your development process can be compared and rated against recognized industry leaders. This comparison is used to establish targets and priorities for improving your process to achieve benchmarked levels of performance and quality.” — Walter J. Utz, Jr. [UTZ92]

The quality approach is to fix the process causing the problem rather than fixing the product over and over again. Optimizing your development process can be accomplished by assessing the maturity of your software development capabilities [discussed in Chapter 10, *Software Development Maturity*]. Each time your capabilities are assessed, you will gain insight into those problem areas where you can concentrate your efforts in each subsequent round of process improvement activities. Studies show that process improvement goals continually mature your process, increase quality and productivity, and lower cost. Process improvement and control continues until it is finally time to abandon the process by making a technology transition to a superior process. [UTZ92]

BEWARE! Studies show that programs operating at low levels of maturity tend to abandon long-term improvement plans when faced with short-term crises.

Quantifiable improvement of software development capabilities requires *buy-in* by all stakeholders in the product and by the owners of all aspects of the process. Improvement activities must be continued and sustained over the entire software life cycle. Improvements should be implemented on all DoD programs in a phased-in, incremental, well-planned manner. Incentives and rewards should be budgeted and granted for improving software capabilities. Your continuous improvement efforts should be sustained until the methods and procedures for improvement become so ingrained in your program's culture that they are performed routinely, as an integral part of every day activities.

14.5 Your Management Challenge

Frederick Brooks is one of the true pioneers of software engineering. In a now classic collection of essays, Brooks includes a line drawing of a prehistoric tar pit, where great, now extinct creatures are struggling to pull themselves from the gooey abyss. He explains:

The tar pit of software engineering will continue to be sticky for a long time to come. One can expect the human race to continue attempting systems just within or just beyond our reach; and software systems are perhaps the most intricate and complex of man's handiwork. The management of this complex craft will demand our best use of new languages and systems, our best adoption of proven engineering management methods, liberal doses of common sense, and a God-given humility to recognize our fallibility and limitations. [BROOKS75]

Your challenge as a software manager is to use the information found in these Guidelines, take control of your acquisition, and develop software with predictable cost, schedule, performance, and quality.

Lloyd K. Mosemann, II, while Deputy Assistant Secretary of the Air Force for Communications, Computers and Support Systems, challenged the software community with eight tasks. He remarked that the number *eight* is inadvertently prophetic in that the number eight is the number for new beginnings. There are seven days in a week and on the eighth day you start all over. Your generation of software managers is at a turning point in history as you have the opportunity to start all over with a new order of successful software management. The software community's eight management tasks are:

1. To stimulate infrastructure investment,
2. To accelerate the pace of technology advance,
3. To adopt an architecture mentality,
4. To encourage functional managers to become more involved, and to address the fundamentals of how they do their business,
5. To advocate technology transition,
6. To make greater use of meaningful metrics,
7. To reduce the overhead burdens associated with software development, and
8. To have defined processes and to institutionalize engineering discipline.

Oliver Cromwell, a famous English statesman and soldier, was on the side of Parliament during the English Civil War. He created the New Model Army (the first professional army in British history), defeated the Scots and the Irish, destroyed the monarchy, executed King Charles I, and ruled England. This illustrious military leader's motto was:

"Not only strike while the iron is hot, but make it hot by striking." [CROMWELL47]

The iron is hot! You are equipped with the tools, the repositories, the education, the programs, the technology, the agents (labs, institutes, and centers), and the software infrastructure to help you do your job smarter and better. They are your opportunity to make the iron hot by striking!

14.6 References

- [ARTHUR93] Arthur, Lowell Jay, Improving Software Quality: An Insider's Guide to TQM, John Wiley & Sons, Inc., New York, 1993
- [BENNATAN92] Bennatan, E.M., On Time, Within Budget: Software Project Management Practices and Techniques, QED Publishing Group, Wellesley, Massachusetts, 1992
- [BOEHM81] Boehm, Barry W., Software Engineering Economics, Prentice-Hall, Inc, Upper Saddle River, New Jersey, 1981
- [BROOKS75] Brooks, Frederick P., Jr., The Mythical Man-Month: Essays on Software Engineering, Addison-Wesley, Reading, Massachusetts, 1975
- [COSBY79] Cosby, Philip B., Quality Is Free, New American Library, Inc., New York, 1979
- [CROMWELL47] Cromwell, Oliver, Writings and Speeches of Oliver Cromwell, Harvard University Press, Cambridge, Massachusetts, 1947
- [DRAKE93] Drake, Dick, review of the book If It Ain't Broke, Break It! by Robert J Kriegel, August 18, 1993
- [HETZEL93] Hetzel, Bill, Making Software Measurement Work: Building an Effective Measurement Program, QED Publishing Group, Boston, 1993
- [KENETT92] Kenett, Ron S., "Understanding the Software Process," G. Gordon Schulmeyer and James I. McManus, eds., Total Quality Management for Software, Van Nostrand Reinhold, New York, 1992
- [LOMBARDI68] Lombardi, Vince, as quoted by Jerry Kramer, Instant Replay, 1968
- [LYONS91] Lyons, Lt Col Robert P., Jr., "Acquisition Perspectives: F-22 Advanced Tactical Fighter," briefing presented to Boldstroke Senior Executive Forum on Software Management, October 16, 1991
- [MACHIARELLI21] Machiavelli, Niccolo, from 1421 writings, The Art of War, The Robbs-Merill Co., Inc., Indianapolis, 1965
- [MARSHALL47] Marshall, BGEN S.L.A., Men Against Fire, 1947
- [MOSEMANN93] Mosemann, Lloyd K., II, as quoted in Ada Information Clearinghouse Newsletter, Vol. XI, No. 2, August 1993
- [POWELL89] Powell, GEN Colin L., as quoted in the Washington Post, January 14, 1989
- [PUTNAM92] Putnam, Lawrence H., and Ware Myers, Measures for Excellence: Reliable Software on Time, Within Budget, Yourdon Press, Englewood Cliffs, New Jersey, 1992
- [REIFER92] Reifer, Donald J., "Software Reuse for TQM," G. Gordon Schulmeyer and James I. McManus, eds., Total Quality Management for Software, Van Nostrand Reinhold, New York, 1992
- [ROETZHEIM88] Roetzheim, William H., Structured Computer Project Management, Prentice Hall, Englewood Cliffs, New Jersey, 1988
- [RUBIN93] Rubin, Howard, "Putting a Measurement Program in Place," Jessica Keyes, ed., Software Engineering Productivity Handbook, Windcrest/McGraw-Hill, New York, 1993
- [SCHULMEYER92] Schulmeyer, G. Gordon, "Zero Defect Software Development," G. Gordon Schulmeyer and James I. McManus, eds., Total Quality Management for Software, Van Nostrand Reinhold, New York, 1992
- [UTZ92] Utz, Walter J., Jr., Software Technology Transitions: Making the Transition to Software Engineering, Prentice Hall, Englewood Cliffs, New Jersey, 1992