Chapter 13

Software Estimation, Measurement, and Metrics

Contents

13.1 Chapter Overview

Poor size estimation is one of the main reasons major software-intensive acquisition programs ultimately fail. Size is the critical factor in determining cost, schedule, and effort. The failure to accurately predict (usually too small) results in budget overruns and late deliveries which undermine confidence and erode support for your program. Size estimation is a complicated activity, the results of which must be constantly updated with actual counts throughout the life cycle. Size measures include source lines-of-code, function points, and feature points. Complexity is a function of size, which greatly impacts design errors and latent defects, ultimately resulting in quality problems, cost overruns, and schedule slips. Complexity must be continuously measured, tracked, and controlled. Another factor leading to size estimate inaccuracies is requirements creep which also must be baselined and diligently controlled.

Any human-intensive activity, without control, deteriorates over time. It takes constant attention and discipline to keep software acquisition and development processes from breaking down — let alone improving them. If you do not measure, there is no way to know whether processes are on track or if they are improving. Measurement provides a way for you to assess the status of your program to determine if it is in trouble or in need of corrective action and process improvement. This assessment must be based on up-to-date measures that reflect current program status, both in relation to the program plan and to models of expected performance drawn from historical data of similar programs. If, through measurement, you diagnose your program as being in trouble, you will be able take meaningful, effective remedial action (e.g., controlling requirements changes, improving response time to the developer, relaxing performance requirements, extending your schedule, adding money, or any number of options). [See Chapter 14, *The Management Challenge*, for a discussion on remedial actions for troubled programs.] Measurement provides benefits at the strategic, program, and technical levels.

A good measurement program is an investment in success by facilitating early detection of problems, and by providing quantitative clarification of critical development issues. Metrics give you the ability to identify, resolve, and/or curtail risk issues before they surface. Measurement must not be a goal in itself. It must be integrated into the total software life cycle — not independent of it. To be effective, metrics must not only be collected — they must be used! Campbell and Koster summed up the bottom line for metrics when they exclaimed:

> *"If you ain't measurin,' you ain't managin' — you're only along for the ride (downhill)!"* [CAMPBELL95]

13.2 Software Estimation

Just as we typically need to determine the weight, volume, and dynamic flight characteristics of a developmental aircraft as part of the planning process, you need to determine how much software to build. One of the main reasons software programs fail is our inability to accurately estimate software size. Because we almost always estimate size too low, we do not adequately fund or allow enough time for development. Poor size estimates are usually at the heart of cost and schedule overruns.

The key to credible software sizing is to use a variety of software sizing techniques, and not to rely on a single source or method for the estimate. Reliance on a single source or technique represents a major contribution to program cost and schedule risk, especially if it is based exclusively on a contractor's bid. There are two common types of size inaccuracies for which you can compensate to some degree.

1. Normal statistical inaccuracies can be dealt with by using multiple data sources and estimating methodologies, or by using multiple organizations to do the estimating and check and analyze results.
2. The earlier the estimate is made — the less is known about the software to be developed — and the greater the estimating errors. [See Figure 13-1]

Basing your estimates on more than one source is sound advice for both types of discrepancies. Because accuracy can be improved if estimates are performed with smaller product elements, base your estimates on the smallest possible unit of each component. Then compile these calculations into composite figures. [HUMPHREY89]

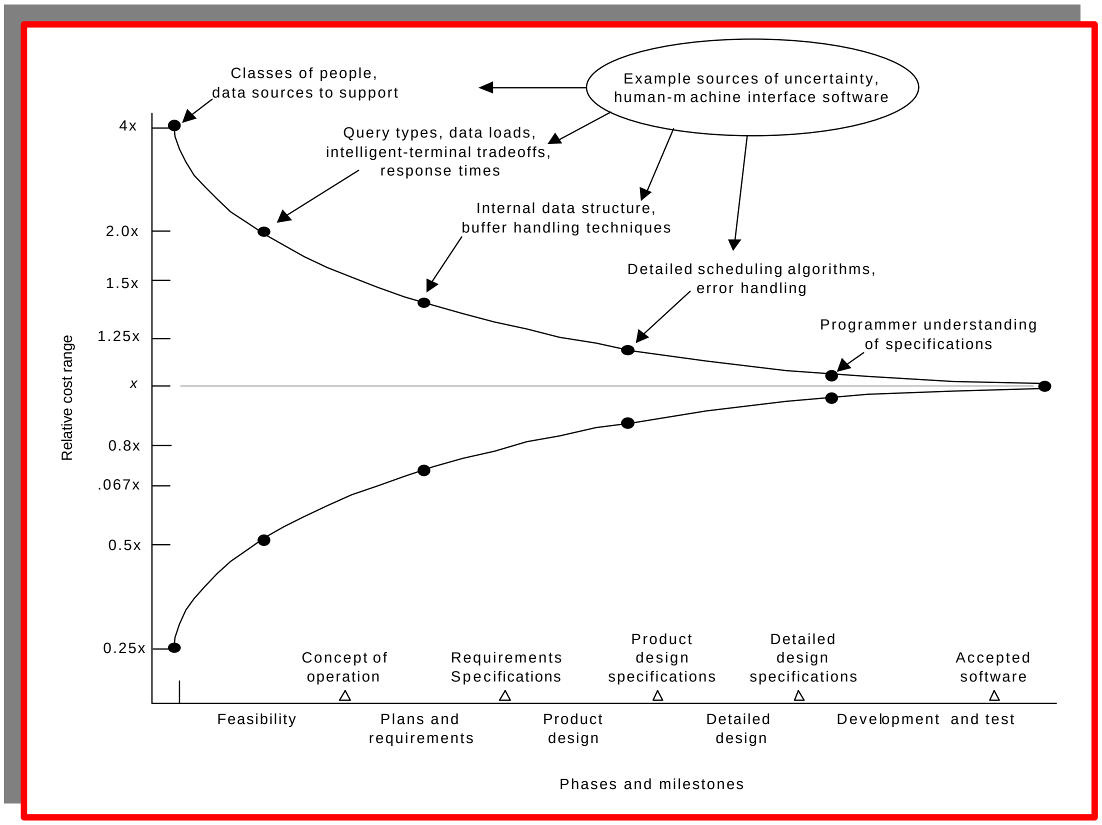

Figure 13-1. Software Cost Estimation Accuracy Versus Phase [BOEHM81]

Given our shortcomings in size estimation, it is absolutely critical that you measure, track, and control software size throughout development. You need to track the actual software size against original estimates (and revisions) both incrementally and for the total build. Analysis is necessary to determine trends in software size and functionality progress. Data requirements for these measures are stated in Contract Data Requirements List (CDRL) items and should include:

- The number of distinct functional requirements in the Software Requirement Specification (SRS) and Interface Requirement Specification (IRS),
- The number of software units contained in the Software Development Plan (SDP) or Software Design Description (SDD), and
- Source lines-of-code (SLOC) or function point estimates for each computer software configuration item (CSCI) and build compared to the actual source code listing for each software unit.

Software size has a direct effect on overall development cost and schedule. Early significant deviations in software size data indicate problems such as:

- Problems in the model(s), logic, and rationale used to develop the estimates,
- Problems in requirements stability, design, coding, and process,
- Unrealistic interpretation of original requirements and resource estimates to develop the system, and
- Faulty software productivity rate estimates.

Significant departures from code development estimates should trigger a risk assessment of the present and overall effort. Size-based models should be revisited to compare your development program with those of similar domain, scope, size, and complexity, if possible.

13.2.1 Measuring Software Size

There are three basic methods for measuring software size. Historically, the primary measure of software size has been the number SLOC. However, it is difficult to relate software functional requirements to SLOC, especially during the early stages of development. An alternative method, function points, should be used to estimate software size. Function points are used primarily for management information systems (MISs), whereas, feature points (similar to function points) are used for real-time or embedded systems. [PUTNAM92] SLOC, function points, and feature points are valuable size estimation techniques. Table 13-1 summarizes the differences between the function point and SLOC methods.

FUNCTION POINTS	SOURCE LINES-OF-CODE
Specification-based	Analogy-based
Language independent	Language dependent
User-oriented	Design-oriented
Variations a function of counting conventions	Variations a function of languages
Expandable to source lines-of-code	Convertible to function points

Table 13-1. Function Points Versus Lines-of-Code

13.2.2 Source Lines-of-Code Estimates

Most SLOC estimates count all executable instructions and data declarations but exclude comments, blanks, and continuation lines. SLOC can be used to estimate size through analogy — by comparing the new software's functionality to similar functionality found in other historic applications. Obviously, having more detailed information available about the functionality of the new software provides the basis for a better comparison. In theory, this should yield a more credible estimate. The relative simplicity of the SLOC size measure facilitates automated and consistent (repeatable) counting of actual completed software size. It also enables recording size data needed to prepare accurate estimates for future efforts. The most significant advantage of SLOC estimates is that they directly relate to the software to be built. The software can then be measured after completion and compared with your initial estimates. [HUMPHREY89] If you are using SLOC with a predictive model (e.g., Barry Boehm's COnstructive COst MOdel (COCOMO)), your estimates will need to be continually updated as new information is available. Only through this constant re-evaluation can the predictive model provide estimates that approximate actual costs.

A large body of literature and historical data exists that uses SLOC, or thousands of source lines-of-code (KSLOC), as the size measure. Source lines-of-code are easy to count and most existing software estimating models use SLOCs as the key input. However, it is virtually impossible to estimate SLOC from initial requirements statements. Their use in estimation requires a level of detail that is hard to achieve (i.e., the planner must often estimate the SLOC to be produced before sufficient detail is available to accurately do so.) [PRESSMAN92]

Because SLOCs are language-specific, the definition of how SLOCs are counted has been troublesome to standardize. This makes comparisons of size estimates between applications written in different programming languages difficult even though conversion factors are available. From SLOC estimates a set of simple, size-oriented productivity and quality metrics can be developed for any given on-going program. These metrics can be further refined using productivity and quality equations such as those found in the basic COCOMO model.

13.2.3 Function Point Size Estimates

Function points, as defined by A.J. Albrecht, are the weighted sums of five different factors that relate to user requirements:

- Inputs,
- Outputs,
- Logic (or master) files,
- Inquiries, and
- Interfaces. [ALBRECHT79]

The International Function Point Users Group (IFPUG) is the focal point for function point definitions. The basic definition of function points provided above has been expanded by several others to include additional types of software functionality, such as those related to embedded weapons systems software (i.e., feature points).

Function points are counted by first tallying the number of each type of function listed above. These unadjusted function point totals are subsequently adjusted by applying complexity measures to each type of function point. The sum of the total complexity-adjusted function points (for all types of function points) becomes the total adjusted function point count. Based on prior experience, the final function point figure can be converted into a reasonably good estimate of required development resources. *[For more information on function point counting, see the "Counting Practices Manual" available from the IFPUG administrative office in Westerville, Ohio for a nominal charge, (614) 895-3170 or Fax (614) 895-3466.]*

Table 13-2 illustrates a function point analysis for a nominal program. First you count the number of inputs, outputs, inquiries, logic files, and interfaces required. These counts are then multiplied by established values. The total of these products is adjusted by the degree of complexity based on the estimator's judgment of the software's complexity. Complexity judgments are domain-specific and include factors such as data communications, distributed data processing, performance, transaction rate, on-line data entry, end-user efficiency, reusability, ease of installation, operation, change, or multiple site use. This process for our nominal program is illustrated in Figure 13-2.

	Simple	Average	Complex	Total
Inputs	3X	4X *2*	6X *2*	20
Outputs	4X *1*	5X *3*	7X	19
Inquiries	3X	4X	6X	0
Files	7X	10X *1*	15X	10
Interfaces	5X	7X	10X	10
UNADJUSTED FUNCTION POINTS =59				

Table 13-2. Function Point Computation

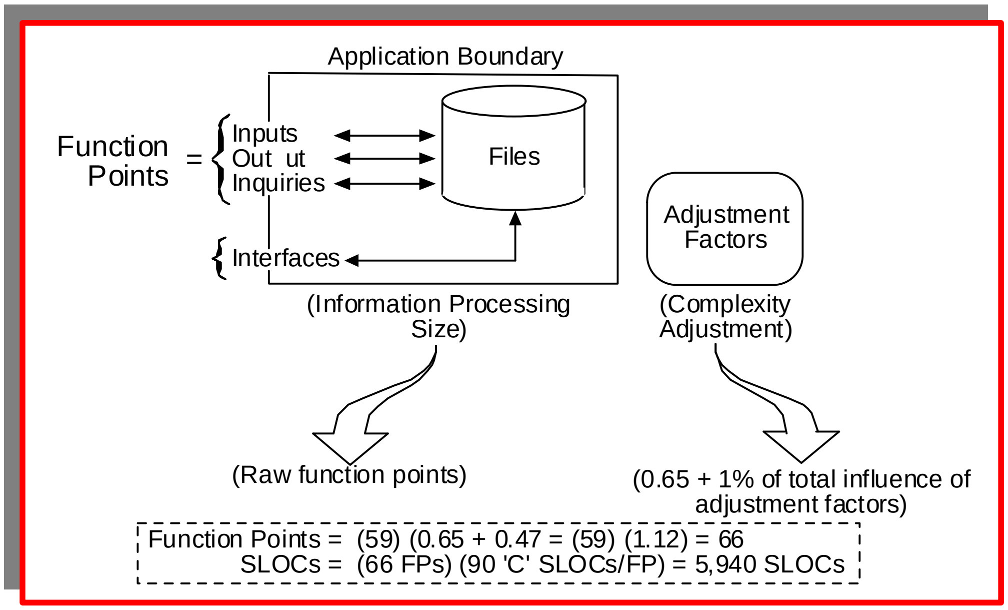
Application Boundary
Function Points =
Inputs
Out ut
Inquiries
Files
Interfaces
Adjustment Factors
(Information Processing Size)
(Complexity Adjustment)
(Raw function points)
(0.65 + 1% of total influence of adjustment factors)
Function Points = (59) (0.65 + 0.47 = (59) (1.12) = 66
SLOCs = (66 FPs) (90 'C' SLOCs/FP) = 5,940 SLOCs

Figure 13-2. Function Point Software Size Computational Process

While function points aid software size estimates, they too have drawbacks. At the very early stages of system development, function points are also difficult to estimate. Additionally, the complexity factors applied to the equation are subjective since they are based on the analyst/engineer's judgment. Few automated tools are available to count either unadjusted or adjusted function points, making comparisons between or among programs difficult, and making the function point counts for any single program inconsistent when calculated by different analysts. However, function points are valuable in making early estimates, especially after the SRS has been completed. Like SLOC, they too are affected by changes in system and/or software requirements. Also, as a relatively new measure of software size, there are few significant, widely-available databases for estimating function points by comparison (analogy) to functionally similar historic software applications.

13.2.4 Feature Point Size Estimates

A derivative of function points, feature points were developed to estimate/measure real-time systems software with high algorithmic complexity and generally fewer inputs/outputs than MISs. Algorithms are sets of mathematical rules expressed to solve significant computational problems. For example, a square root extraction routine, or a Julian date conversion routine, are algorithms.

In addition to the five standard function point parameters, feature points include an algorithm(s) parameter which is assigned the default weight of 3. The feature point method reduces the empirical weights for logical data files from a value of 10 to 7 to account for the reduced significance of logical files in real-time systems. For applications in which the number of algorithms and logical data files are the same, function and feature point counts generate the same numeric values. But, when there are more algorithms than files, feature points produce a greater total than function points. Table 13-3 illustrates the ratio of function point to feature point counts for selected applications. [For a more detailed explanation of feature points, see Capers Jones, *Applied Software Measurement*.] [JONES91]

APPLICATION	FUNCTION POINTS	FEATURE POINTS
Batch MIS projects	1	0.80
On-line MIS projects	1	1.00
On-line database projects	1	1.00
Switching systems projects	1	1.20
Embedded real-time projects	1	1.35
Factory automation projects	1	1.50
Diagnostic and prediction projects	1	1.75

Table 13-3. Ratios of Feature Points to Function Points [JONES91]

NOTE: See Volume 2, Appendix H, *"Counting Rules for Function Points and Feature Points."*

13.2.5 Cost and Schedule Estimation Methodologies/Techniques

Most estimating methodologies are predicated on analogous software programs. Expert opinion is based on experience from similar programs; parametric models stratify internal data bases to simulate environments from many analogous programs; engineering builds reference similar experience at the unit level; and cost estimating relationships (like parametric models) regress algorithms from several analogous programs. Deciding which of these methodologies (or combination of methodologies) is the most appropriate for your program usually depends on *availability of data*, which is in turn depends on where you are in the life cycle or your scope definition.

- **Analogies**. Cost and schedule are determined based on data from completed similar efforts. When applying this method, it is often difficult to find analogous efforts at the total system level. It may be possible, however, to find analogous efforts at the subsystem or lower level computer software configuration item/computer software component/computer software unit (CSCI/CSC/CSU). Furthermore, you may be able to find completed efforts that are more or less similar in complexity. If this is the case, a *scaling factor* may be applied based on expert opinion (e.g., CSCI-x is 80% as complex). After an analogous effort has been found, associated data need to be assessed. It is preferable to use effort rather than cost data; however, if only cost data are available, these costs must be normalized to the same base year as your effort using current and appropriate inflation indices. As with all methods, the quality of the estimate is directly proportional to the credibility of the data.
- **Expert (engineering) opinion**. Cost and schedule are estimated by determining required effort based on input from personnel with extensive experience on similar programs. Due to the inherent subjectivity of this method, it is especially important that input from several independent sources be used. It is also important to request only effort data rather than cost data as cost estimation is usually out of the realm of engineering expertise (and probably dependent on non-similar contracting situations). This method, with the exception of rough orders-of-magnitude estimates, is rarely used as a primary methodology alone. Expert opinion is used to estimate lower-level, low cost, pieces of a larger cost element when a labor-intensive cost estimate is not feasible.

- **Parametric models**. The most commonly-used technology for software estimation is parametric models, a variety of which are available from both commercial and government sources. The estimates produced by the models are *repeatable*, facilitating sensitivity and domain analysis. The models generate estimates through statistical formulas that relate a dependent variable (e.g., cost, schedule, resources) to one or more independent variables. Independent variables are called *"cost drivers"* because any change in their value results in a change in the cost, schedule, or resource estimate. The models also address both the development (e.g., development team skills/experience, process maturity, tools, complexity, size, domain, etc.) and operational (how the software will be used) environments, as well as software characteristics. The environmental factors, used to calculate cost (manpower/effort), schedule, and resources (people, hardware, tools, etc.), are often the basis of comparison among historical programs, and can be used to assess on-going program progress.

Because environmental factors are relatively subjective, a rule of thumb when using parametric models for program estimates is to *use multiple models* as checks and balances against each other. Also note that parametric models are not 100 percent accurate. Boehm states:

> *"Today, a software cost estimation model is doing well if it can estimate software development costs within 20% of the actual costs, 70% of the time, and on its own home turf (that is, within the class of projects to which it is calibrated…. This means that the model's estimates will often be much worse when it is used outside its domain of calibration."* [BOEHM81]

Boehm's assertion is still valid today as reported by Ferens and Christensen. They report:

> *"…in general, model validation showed that the accuracy of the models was no better than within 25 percent of actual development cost or schedule, about one half of the time, even after calibration."* [FERENS00]

In all fairness, the databases used to achieve the results reported in [FERENS00] were not from a single entity, but a compilation of data from several software developers. One would assume as developers increase their maturity levels and gather data, their ability to estimate required resources would improve. However, if dealing with developers that do not have a good estimation database, the wise project manager will plan for cost and schedule to be 30% to 50% higher than the estimates provided by the parametric cost models.

- **Engineering build** (*grass roots*, or *bottoms-up* build). Cost and schedule are determined by estimating effort based on the effort summation of detailed functional breakouts of tasks at the lowest feasible level of work. For software, this requires a detailed understanding of the software architecture. Analysis is performed at the CSC or CSU level and associated effort is predicted based on unit level comparisons to similar units. Often, this method is based on a notional system of *government estimates of most probable cost* and used in source selections before contractor solutions are known. This method is labor-intensive and is usually performed with engineering support; however, it provides better assurance than other methods that the entire development scope is captured in the resulting estimate.
- **Cost Performance Report (CPR) analysis**. Future cost and schedule estimates are based on current progress. This method may not be an optimal choice for predicting software cost and schedule because software is generally developed in three distinct phases (requirements/ design, code/unit test, integration/test) by different teams. Apparent progress in one phase may not be predictive of progress in the next phases, and lack of progress in one phase may not show up until subsequent phases. Difficulty in implementing a poor design may occur without warning, or problems in testing may be the result of poor test planning or previously undetected coding defects. CPR analysis can be a good starting point for identifying problem areas, and problem reports included with CPRs may provide insight for risk assessments.
- **Cost estimating relationships (CERs)/factors**. Cost and schedule are estimated by determining effort based on algebraic relationships between a dependent (effort or cost) variable and independent variables. This method ranges from using simple factor, such as cost per line-of-code on similar program with similar contractors, to detailed multi-variant regressions based on several similar programs with more than one causal (independent) variable. Statistical packages are commercially available for developing CERs, and if data are available from several completed similar programs (which is not often the case), this method may be a worthwhile investment for current and future cost and schedule estimating tasks. Parametric model developers incorporate a series of CERs into an automated process by which parametric inputs determine which CERs are appropriate for the program at hand.

Of these techniques, the most commonly used is parametric modeling. There is currently no list of recommended or approved models; however, you will need to justify the appropriateness of the specific model or other technique you use in an estimate presented for DAB and/or MAISARC Review. As mentioned above, determining which method is most appropriate is driven by the availability of data. Regardless of which method used, a thorough understanding of your software's functionality, architecture, and characteristics, and your contract is necessary to accurately estimate required effort, schedule, and cost.

NOTE: Refer to "A Manager's Checklist for Validating Software Cost and Schedule Estimates," CMU/SEI-95-SR-04, and "Checklists and Criteria for Evaluating the Cost and Schedule Estimating Capabilities of Software Organizations," CMU/SEI-95-SR-05.

13.2.6 Ada-Specific Cost Estimation

Using Ada-specific models is necessary because Ada developments do not follow the classic patterns included in most traditional cost models. As stated above, the time and effort required during the design phase are significantly greater (50% for Ada as opposed to 20% for non-Ada software developments). [Ada/C++91] Another anomaly with Ada developments is productivity rates. Traditional non-Ada developments have historically recorded that productivity rates decrease as program size increases. With Ada, the opposite is often true. Due in large to Ada reusability, the larger the program size — the greater the productivity rate.

13.3 Software Measurement

You cannot build quality software, or improve your process, without measurement. Measurement is essential to achieving the basic management objectives of prediction, progress, and process improvement. An oft-repeated phrase by DeMarco holds true; "You can't manage what you can't measure!" [DeMARCO86] All process improvement must be based on measuring where you have been, where you are now, and properly using the data to predict where you are heading. Collecting good metrics and properly using them always leads to process improvement!

13.3.1 Measures and Metrics, and Indicators

A software measurement is a quantifiable dimension, attribute, or amount of any aspect of a software program, product, or process. It is the raw data which are associated with various elements of the software process and product. Table 13-4 gives some examples of useful management measures.

AREA	MEASURES
Requirements	• CSCI requirements • CSCI design stability
Performance	• Input/output bus throughout capability • Processor memory utilization • Processor throughput put utilization
Schedule	• Requirements allocation status • Preliminary design status • Code and unit test status • Integration status
Cost	• Person-months of effort • Software size

Table 13-4. Example Management Indicators

Metrics (or indicators) are computed from measures. They are quantifiable indices used to compare software products, processes, or projects or to predict their outcomes. With metrics, we can:

- **Monitor** requirements,
- **Predict** development resources,
- **Track** development progress, and
- **Understand** maintenance costs.

Metrics are used to compare the current state of your program with past performance or prior estimates and are derived from earlier data from within the program. They show *trends* of increasing or decreasing values, relative only to the previous value of the same metric. They also show containment or breeches of pre-established limits, such as allowable latent defects.

Metrics are also useful for determining a *"business strategy"* (how resources are being used and consumed). For example, in producing hardware, management looks at a set of metrics for scrap and rework. From a software standpoint, you will want to see the same information on how much money, time, and manpower the process consumes that does not contribute to the end product. One way a software program might consume too many resources is if errors made in the requirements phase were not discovered and corrected until the coding phase. Not only does this create rework, but the *cost to correct an error during the coding phase that was inserted during requirements definition is approximately 50% higher to correct than one inserted and corrected during the coding phase*. [BOEHM81] The key is to catch errors as soon as possible (i.e., in the same phase that they are induced).

Management metrics are measurements that help evaluate how well the acquisition office is proceeding in accomplishing their acquisition plan or how well the contractor is proceeding in accomplishing their Software Development Plan. Trends in management metrics support forecasts of future progress, early trouble detection, and realism in plan adjustments. Software product attributes are measured to arrive at product metrics which determine user satisfaction with the delivered product or service. From the user's perspective, product attributes can be reliability, ease-of-use, timeliness, technical support, responsiveness, problem domain knowledge and understanding, and effectiveness (creative solution to the problem domain). Product attributes are measured to evaluate software quality factors, such as efficiency, integrity, reliability, survivability, usability, correctness, maintainability, verifiability, expandability, flexibility, portability, reusability, and interoperability. Process metrics are used to gauge organizations, tools, techniques, and procedures used to develop and deliver software products. [PRESSMAN92] Process attributes are measured to determine the status of each phase of development (from requirements analysis to user acceptance) and of resources (dollars, people, and schedule) that impact each phase.

NOTE: Despite the SEI's Software Capability Evaluation (SCE) methods, process efficiency can vary widely within companies rated at the same maturity levels, and from program to program.

There are five classes of metrics generally used from a commercial perspective to measure the quantity and quality of software. During development technical and defect metrics are used. After market metrics are then collected which include user satisfaction, warranty, and reputation.

- **Technical metrics** are used to determine whether the code is well-structured, that manuals for hardware and software use are adequate, that documentation is complete, correct, and up-to-date. Technical metrics also describe the external characteristics of the system's implementation.
- **Defect metrics** are used to determine that the system does not erroneously process data, does not abnormally terminate, and does not do the many other things associated with the failure of a software-intensive system.
- **End-user satisfaction** metrics are used to describe the value received from using the system.

- **Warranty metrics** reflect specific revenues and expenditures associated with correcting software defects on a case-by-case basis. These metrics are influenced by the level of defects, willingness of users to come forth with complaints, and the willingness and ability of the software developer to accommodate the user.
- **Reputation metrics** are used to assess perceived user satisfaction with the software and may generate the most value, since it can strongly influence what software is acquired. Reputation may differ significantly from actual satisfaction:
  - Because individual users may use only a small fraction of the functions provided in any software package; and
  - Because marketing and advertising often influence buyer perceptions of software quality more than actual use.

13.3.2 Software Measurement Life Cycle

Effective software measurement adds value to all life cycle phases. Figure 13-3 illustrates the primary measures associated with each phase of development. During the requirements phase, function points are counted. Since requirements problems are a major source of cost and schedule overruns, an error and quality tracking system is put into effect. Once requirements are defined, the next phase of measurement depends on whether the program is a custom development, or a combination of newly developed applications, reusable assets, and COTS. The risks and values of all candidate approaches are quantified and analyzed.

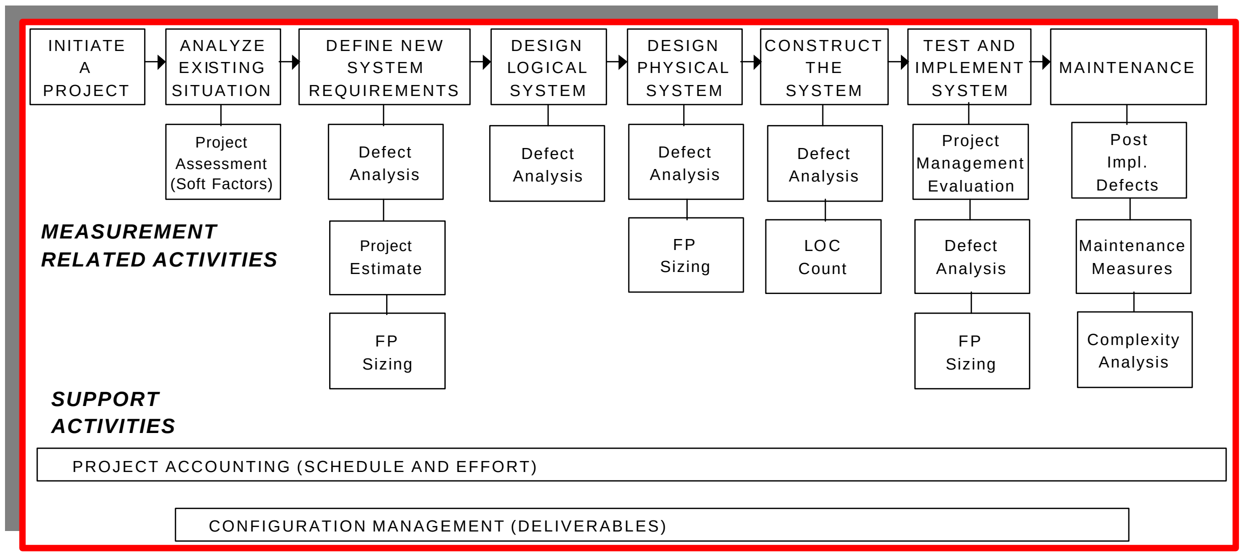

Figure 13-3. Software Measurement Life Cycle [JONES91]

If the solution is a custom development, from the logical system design on, defect removal can be a very costly activity. Therefore, design reviews and peer inspections are used to gather and record defect data. The data are analyzed to determine how to prevent similar defects in the future. During physical design, reviews and peer inspections continue to be valuable and defect data are still collected and analyzed. The coding phase can be anything from very difficult to almost effortless, depending upon the success of the preceding phases. Defect and quality data, as well as code complexity measures, are recorded during code reviews and peer inspections.

One of the most important characteristics of your software development is *complexity*. With the first executable architecture, throughout the design phases, and subsequently, as the code is developed and compiled, evaluate the complexity of your proposed development. This will give you important insight into the feasibility of bringing your program to a successful conclusion (i.e., to provide the desired functional performance on the predicted schedule, at the estimated cost).

The testing phase will range from a simple unit tests the programmers conduct, to a full-scale, multi-staged formal test suite, including function tests, integration tests, stress tests, regression tests, independent tests, field tests, system tests, and final acceptance tests. During all these activities, in-depth defect data are collected and analyzed for use in subsequent defect prevention activities such as changing the development process to prevent similar defects in future developments. Retrospective analyses are performed on defect removal efficiencies for each specific review, peer inspection, and test, and on the cumulative efficiency of the overall series of defect removal steps.

During the maintenance phase, both user satisfaction and latent defect data are collected. For enhancements or replacements of existing systems, the structure, complexity, and defect rates of the existing software are determined. Further retrospective analysis of defect removal efficiency is also be performed. *A general defect removal goal is a cumulative defect removal efficiency of 95% for MIS, and 99.9% (or greater) for real-time weapon systems*. This means that, when defects are found by the development team or by the users, they are summed after the first (and subsequent) year(s) of operations. Ideally, the development team will find and remove 95% to 100% of all latent defects. [JONES91]

13.3.3 Software Measurement Life Cycle at Loral

Table 13-5 illustrates how Loral Federal Systems collects and uses metrics throughout the software life cycle. The measurements collected and the metrics (or in-process indicators) derived from the measures change throughout development. At the start of a program (during the proposal phase or shortly after contract start), detailed development plans are established. These plans provide planned start and completion dates for each CDRL (CSU or CSCI). From these plans, a profile is developed showing the planned percent completion per month over the life of the program for all the software to be developed. Also at program start, a launch meeting is conducted to orient the team to a common development process and to present lessons-learned from previous programs.

PROGRAM PHASE	MEASUREMENT COLLECTION	IN-PROCESS INDICATORS (METRICS)
At Start	Establish SPD (plan star/completion for each phase)	Initial plan-complete profiles
During Design, Coding, Unit Testing	Weekly: Update and complete plan items Regularly: Inspect Defect Data Bi-monthly: Casual analysis meeting Quarterly: Update measurements	% plan verus % casual Inspection effectiveness (defects found ÷defects remaining); Defects ÷ KSLOC Process improvements identified Product quality (projected); Staffing profile
During Integration Testing	Program Trouble Reports (PTRs) Development team survey transferred to integration and test team survey	PTR density (PTR ÷ KSLOC) Open PTRs versus time % PTRs satisfied and survey comments
At End	Document lessons learned Customer survey PTRs during acceptance	% satisfied and customer comments used for future improvements Product quality (actual)

Table 13-5. Collection and Use of Metrics at Loral

During software design, coding, and unit testing, many different measurements and metrics are used. On a weekly basis, actual percent completion status is collected against the plan established at program start. Actuals are then plotted against the plan to obtain early visibility into any variances. The number of weeks (early or late) for each line item is also tracked to determine where problems exist. Source statement counts (starting as estimates at program start) are updated whenever peer inspections are performed. A plot of code change over time is then produced. Peer inspections are conducted on a regular basis and defects are collected. Peer inspection metrics include peer inspection efficiency (effectiveness) (percent of defects detected during inspections versus those found later) and expected product quality [the number of defects detected per thousand source lines-of-code (KSLOC)]. Inspection data are used to project the expected latent defect rate (or conversely, the defect removal efficiency rate) after delivery. At the completion of each development phase, defect causal analysis meetings are held to examine detected defects and to establish procedures, which when implemented, will prevent similar defects from occurring in the future.

During development, program measurements are updated periodically (either monthly or quarterly) in the site metrics database. These measurements include data such as cost, effort, quality, risks, and technical performance measures. Other metrics, such as planned versus actual staffing profiles, can be derived from these data. During integration and test, trouble report data are collected. Two key metrics are produced during this phase: the Program Trouble Report (PTR) density (the number of defects per KSLOC) and Open PTRs over time. An internal development team survey is transferred to the integration and test team so they can derive the team's defect removal efficiency and PTR satisfaction for the internal software delivery.

At the end of the program, lessons-learned for process improvement are documented for use in future team launches. If possible, a customer survey is conducted to determine a customer satisfaction rating for the program. Delivered product quality is measured during acceptance testing and compared to the earlier projected quality rate. These actuals are then used to calibrate the projection model for future and on-going programs.

13.4 Software Measurement Process

The software measurement process must be an objective, orderly method for quantifying, assessing, adjusting, and ultimately improving the development process. Data are collected based on known, or anticipated, development issues, concerns, and questions. They are then analyzed with respect to the software development process and products. The measurement process is used to assess quality, progress, and performance throughout all life cycle phases. The key components of an effective measurement process are:

- Clearly defined software development issues and the measure (data elements) needed to provide insight into those issues;
- Processing of collected data into graphical or tabular reports (indicators) to aid in issue analysis;
- Analysis of indicators to provide insight into development issues; and,
- Use of analysis results to implement process improvements and identify new issues and problems.

Your normal avenue for obtaining measurement data from the contractor is via contract CDRLs. A prudent, SEI Level 3 contractor will implement a measurement process even without government direction. This measurement process includes collecting and receiving actual data (not just graphs or indicators), and analyzing those data. To some extent the government program office can also implement a measurement process independent of the contractor's, especially if the contractor is not sufficiently mature to collect and analyze data on his own. In any case, it is important for the Government and the contractor to meet and discuss analysis results. *Measurement activities keep you actively involved in, and informed of, all phases of the development process.* Figure 13-4 illustrates how measurement was integrated into the organizational hierarchy at IBM-Houston. It shows how a bottoms-up measurement process is folded into corporate activities for achieving corporate objectives.

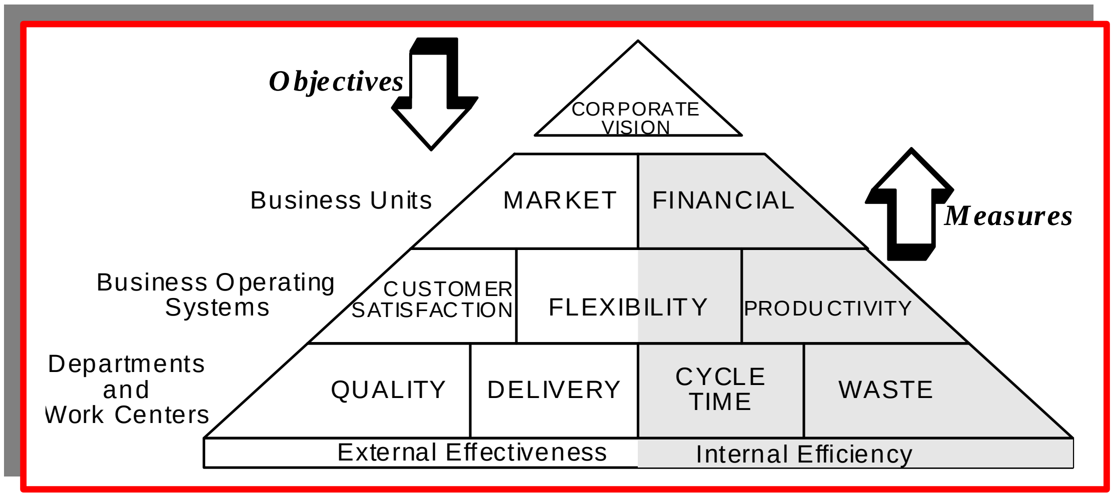

Figure 13-4. Organizational Measurement Hierarchy

13.4.1 Metrics Plan

NOTE: Practical Software Measurement: A Foundation for Objective Project Management, available at www.psmsc.com, and Practical Software Measurement: Measuring for Process Management and Improvement (CMU/SEI-97-HB-003), available at www.sei.cmu.edu should be reviewed in conjunction with the remainder of this chapter.

For measurement to be effective, it must become an integral part of your decision-making process. Insights gained from metrics should be merged with process knowledge gathered from other sources in the conduct of daily program activities. It is the entire measurement process that gives value-added to decision-making, not just the charts and reports. [ROZUM92] Without a firm Metrics Plan, based on issue analysis, you can become overwhelmed by statistics, charts, graphs, and briefings to the point where you have little time for anything other than ingestion. Plan well! Not all data is worth collecting and analyzing. Once your development program is in-process, and your development team begins to design and produce lines-of-code, the effort involved in planning and specifying the metrics to be collected, analyzed, and reported upon begins to pay dividends. Figure 13-5 illustrates examples of life cycle measures and their benefits collected on the Space Shuttle program.

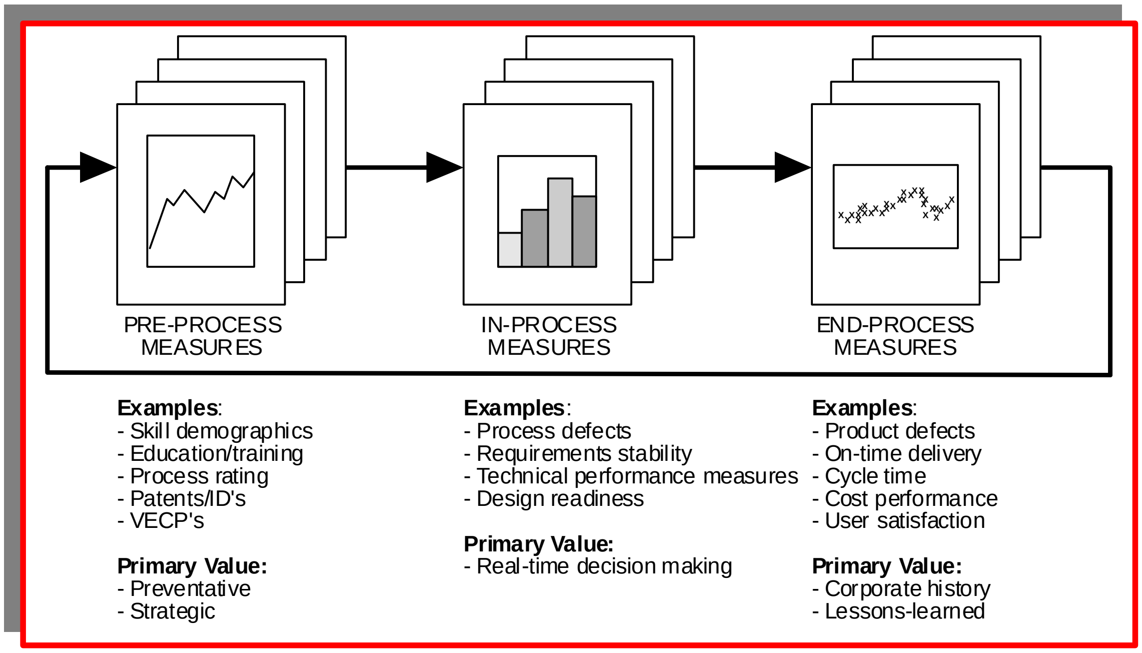

Figure 13-5. Space Shuttle Life Cycle Measurements

The ground rules for a Metrics Usage Plan are that:

- **Metrics must be understandable to be useful**. For example, lines-of-code and function points are the most common, accepted measures of software size with which software engineers are most familiar.
- **Metrics must be economical**. Metrics must be available as a natural by-product of the work itself and integral to the software development process. Studies indicate that approximately 5% to 10% of total software development costs can be spent on metrics. The larger the software program, the more valuable the investment in metrics becomes. Therefore, do not waste programmer time by requiring specialty data collection that interferes with the coding task. Look for tools which can collect most data on an unintrusive basis.
- **Metrics must be field tested**. Beware of software contractors who offer metrics programs that appear to have a sound theoretical basis, but have not had practical application or evaluation. Make sure proposed metrics have been successfully used on other programs or are prototyped before accepting them.
- **Metrics must be highly leveraged**. You are looking for data about the software development process that permit management to make significant improvements. Metrics that show deviations of .005% should be relegated to the trivia bin.
- **Metrics must be timely**. Metrics must be available in time to effect change in the development process. If a measurement is not available until the program is in deep trouble it has no value.
- **Metrics must give proper incentives for process improvement**. High scoring teams are driven to improve performance when trends of increasing improvement and past successes are quantified. Conversely, metrics data should be used very carefully during contractor performance reviews. A poor performance review, based on metrics data, can lead to negative government/industry working relationships. *Do not use metrics to judge team or individual performance.*

- **Metrics must be evenly spaced throughout all phases of development**. Effective measurement adds value to all life cycle activities. [JONES91]
- **Metrics must be useful at multiple levels**. They must be meaningful to both management and technical team members for process improvement in all facets of development.

13.4.2 Activities That Constitute a Measurement Process

Measurement can be expressed in terms of a fairly simple process. Simple is not to be confused with easy. There are innumerable challenges to the successful implementation of an effective measurement process. The following is a recommended measurement process logically grouped by activity type (planning, implementation, and evaluation). During measurement planning, information needs drive the selection of indicators. Indicators and analysis methods drive the selection of specific measures. The specific measures drive the selection of collection processes. During measurement implementation, the collection process provides measurement data that is analyzed and formed into indicators that are used to satisfy information needs. The evaluation activity is where metrics are judged to determine if they are providing the information required for decision making and process improvement.

13.4.2.1 Planning

There are four activities to measurement planning (discussed below). The results of these four activities are documented in the Measurement Plan.

13.4.2.1.1 Define Information Needs

The need for information is derived from goals and objectives. These goals and objectives may come from project planning, project monitoring, process compliance, pilot process improvements, and/or risk mitigation, etc. There are technical needs, project needs, managerial needs, and organizational needs.

13.4.2.1.2 Define Indicators and Analysis Methods to Address Information Needs

Once the information needs have been prioritized, an appropriate number of indicators are selected. Indicators are derived from the analysis of “two or more sets of measurement data”. Once the indicators are selected, the appropriate analysis methods are defined to derive indicators from the measurement data.

It is important during the initial growth of a measurement program that information needs be limited. This is a change management issue. Do not overwhelm the measurement staff. On the other hand, a little advanced planning will serve to develop measurements useful as organizations mature.

13.4.2.1.3 Define the Selected Measures

The selected indicators and analysis methods will drive the selection of specific measures. Once the measures are selected, the details of the measure must be documented. This includes definitions, terms, data requirements, security requirements, data types etc.

Once data collection has commenced, it is important that there is consistency in data definitions. Changing a definition in midstream during data collection produces variations in data trends that can skew the analysis of performance, quality, and related issues. If definitions do change through process-knowledge, it is critically important that you understand each change and how these changes will affect already collected data. [Changes in entry/exit definitions should be reflected in an updated SDP.]

13.4.2.1.4 Metrics Selection

Marciniak and Reifer proclaim that: "Software projects don't get into trouble all at once; instead they get into trouble a little at a time." [MARCINIAK90] Metrics must be selected to ascertain your program's trouble threshold at the earliest phase of development. Basic measures should be tailored by defining and collecting data that address those trouble (risk) areas identified in the Risk Management Plan. A rule of thumb for metrics is that they must provide insight into areas needing process improvement! Which metrics to use depends on the maturity level of the organization. Table 13-6, based on the Army's STEP metrics process, illustrates how metrics are used to obtain program knowledge. [See Army policy on Preparation for Implementing Army Software Test and Evaluation Panel (STEP) Metrics Recommendations, Volume 2, Appendix B.]

METRIC	OBJECTIVE
Schedule	Track progress versus schedule
Cost	Track software expenditures
Computer resource utilization	Track planned versus actual size
Software engineering environment	Rate contractor environment
Design stability	Rate stability of software design and planned release or block. Design stability = [(total modules+deleted module+modules with design changes)/total modules]. Design thresholds: >=0.9 to >0.85 (warning), <=0.85 (alarm)
Requirements traceability	Track rqmts to code by showing % of software rqmts traceable to developer specifications, coding items (CI), CSCI, or CSC. Traceability thresholds are: <=99% to <99.5% (warning), <99.5% (alarm)
Requirements stability	Track changes to requirements
Fault profiles	Track open versus closed anomalies
Complexity	Assess code quality
Breadth testing	Extent to which rqmts are tested. 4 measures are: coverage (ration or rqmts tested to total rqmts); test success (ratio of rqmts passed to rqmts tested); overall success (ratio rqmts passed to total rqmts); deferred (total deferred rqmts). Overall success thresholds: >99% (satisfactory), <99% to <95% (warning), <=95% (alarm)
Depth of testing	Track testing of code
Reliability	Monitor potential downtime due to software

Table 13-6. How Metrics Are Used for Program Management

ATTENTION! Metrics selection must focus on those areas you have identified as sources of significant risk for your program.

NOTE: Examples of minimum measures include: quarterly collation and analysis of size (counting source statements and/or function/feature points), effort (counting staff hours by task and nature of work; e.g., hours for peer inspection), schedule (software events over time), software quality (problems, failures, and faults), rework (SLOC changed or abandoned), requirements traceability (percent of requirements traced to design, code, and test), complexity (quality of code), and breadth of testing (degree of code testing).

Contact the Global Transportation Network (GTN) Program for a copy of the GTN Software Development Metrics Guidelines which discusses the metrics selected and gives examples of how to understand and interpret them. See Volume 2, Appendix A for information on how to contact the GTN program office.

13.4.3 Typical Software Measurements and Metrics

A comprehensive list of industry metrics is available for software engineering management use, ranging from high-level effort and software size measures to detailed requirements measures and personnel information. *Quality*, not quantity, should be the guiding factor in selecting metrics. It is best to choose a small, meaningful set of metrics that have solid baselines in a similar environment. A typical set of metrics might include:

- Quality,
- Size,
- Complexity,
- Requirements,
- Effort,
- Productivity,
- Cost and schedule,
- Scrap and rework, and
- Support.

Some industry sources of historical data are those listed in Volume 2, Appendix A. A good source of information on optional software metrics and complexity measures are the SEI technical reports listed in Volume 2, Appendix D. Also see Rome Laboratory report, RL-TR-94-146, Framework Implementation Guidebook, for a discussion on software quality indicators. Another must-read reference is the Army Software Test and Evaluation Panel (STEP) Software Metrics Initiatives Report, USAMSAA, May 6, 1992 and policy memorandum, Preparation for Implementing Army Software Test and Evaluation Panel (STEP) Metrics Recommendations, Volume 2, Appendix B.

13.4.3.1 Quality

Measuring product quality is difficult for a number of reasons. One reason is the lack of a precise definition for quality. Quality can be defined as the degree of excellence that is measurable in your product. The IEEE definition for software quality is the degree to which a system, component, or process meets:

- Specified requirements
- Customer or user needs or expectations. [IEEE90]

Quality is in the eye of the user! For some programs, product quality might be defined as reliability [i.e., a low failure density (rate)], while on others maintainability is the requirement for a quality product. [MARCINIAK90] Your definition of a quality product must be based on measurable quality attributes that satisfy your program's specified user requirements. Because requirements differ among programs, quality attributes will also vary. [MARCINIAK90]

13.4.3.2 Size

Software size is indicative of the effort required. Software size metrics have been discussed above as SLOC, function points, and feature points.

13.4.3.3 Complexity

Complexity measures focus on designs and actual code. They assume there is a direct correlation between design complexity and design errors, and code complexity and latent defects. By recognizing the properties of each that correlate to their complexity, we can identify those high-risk applications that either should be revised or subjected to additional testing.

Those software properties which correlate to how complex it is are size, interfaces among modules (usually measured as *fan-in*, the number of modules invoking a given application, or *fan-out*, the number of modules invoked by a given application), and structure (the number of paths within a module). Complexity metrics help determine the number and type of tests needed to cover the design (interfaces or calls) or coded logic (branches and statements).

There are several accepted methods for measuring complexity, most of which can be calculated by using automated tools.

NOTE: See Appendix M, *Software Complexity*, for a discussion on complexity analysis.

13.4.4 Requirements

Requirements changes are a major source of software size risk. If not controlled and baselined, requirements will continue to grow, increasing cost, schedule, and fielded defects. If requirements evolve as the software evolves, it is next to impossible to develop a successful product. Software developers find themselves shooting at a moving target and throwing away design and code faster than they can crank it out *[scrap and rework is discussed below]*. Colonel Robert Lyons, Jr., former co-leader of the F-22 System Program Office Avionics Group, cites an "*undisciplined requirements baselining process*" as the number one cause of "*recurring weapons system problems.*" An undisciplined requirements process is characterized by:

- Inadequate requirements definition,
- Late requirements clarification,
- Derived requirements changes,
- Requirements creep, and
- Requirements baselined late. [LYONS91]

Once requirements have been defined, analyzed, and written into the System Requirements Specification (SRS), they must be tracked throughout subsequent phases of development. In a system of any size this is a major undertaking. The design process translates user-specified (or explicit) requirements into derived (or implicit) requirements necessary for the solution to be turned into code. This multiplies requirements by a factor of sometimes hundreds. [GLASS92]

Each implicit requirement must be fulfilled, traced back to an explicit requirement, and addressed in design and test planning. *It is the job of the configuration manager to guarantee the final system meets original user requirements.* As the requirements for a software solution evolve into a design, there is a snowball effect when converting original requirements into design requirements needed to convert the design into code. Conversely, sometimes requirements are not flowed down and get lost during the development process (dropped through the developmental crack) with a resulting loss in system performance or function. When requirements are not adequately tracked, interface data elements can disappear, or extra interface requirements can be introduced. Missing requirements may not become apparent until system integration testing, where the cost to correct this problem is exponentially high.

13.4.5 Effort

In the 1970s, Rome Air Development Center (RADC) collected data on a diverse set of over 500 DoD programs. The programs ranged from large (millions of lines-of-code and thousands of months of effort) to very small (a one month effort). The data was sparse and rather primitive but it did include output KLOC and input effort months and duration. At that time most program managers viewed effort and duration as interchangeable. When we wanted to cut completion time in half, we assigned twice as many people! Lessons-learned, hard knocks, and program measures such as the RADC database, indicated that the relationships between duration and effort were quite complex, nonlinear functions. Many empirical studies over the years have shown that manpower in large developments builds up in a characteristic way and that it is a complex power function of software size and duration. Many estimation models were introduced, the best known of which is Barry Boehm's COnstructive COst MOdel (COCOMO). [HETZEL93]

13.4.6 Productivity

Software productivity is measured in the number of lines-of-code or function/feature points delivered (i.e., SLOC that have been produced, tested, and documented) per staff month that result in an acceptable and usable system.

Boehm explains there are three basic ways to improve software development productivity.

- Reduce the cost-driver multipliers,
- Reduce the amount of code; and,
- Reduce the scalable factor that relates the number of instructions to the number of man months or dollars.

His model for measuring productivity is:

Productivity = Size/Effort where
Effort = Constant x Size $^{\textit{Sigma}}$ ***x Multipliers*** [BOEHM81]

In the equation for effort, multipliers are factors such as efficiency of support tools, whether the software must perform within limited hardware constraints, personnel experience and skills, etc. Figure 13-6 lists the various cost drivers that affect software development costs. Many of these factors (not all) can be modified by effective management practices. The weight of each factor as a cost multiplier (on a scale of 1 to 4.18, with larger numbers having the greater weight) reflects the relative effect that factor has on total development costs. Boehm's studies show that *"employing the right people" has the greatest influence on productivity*. (Reliability and complexity are also important multipliers.)

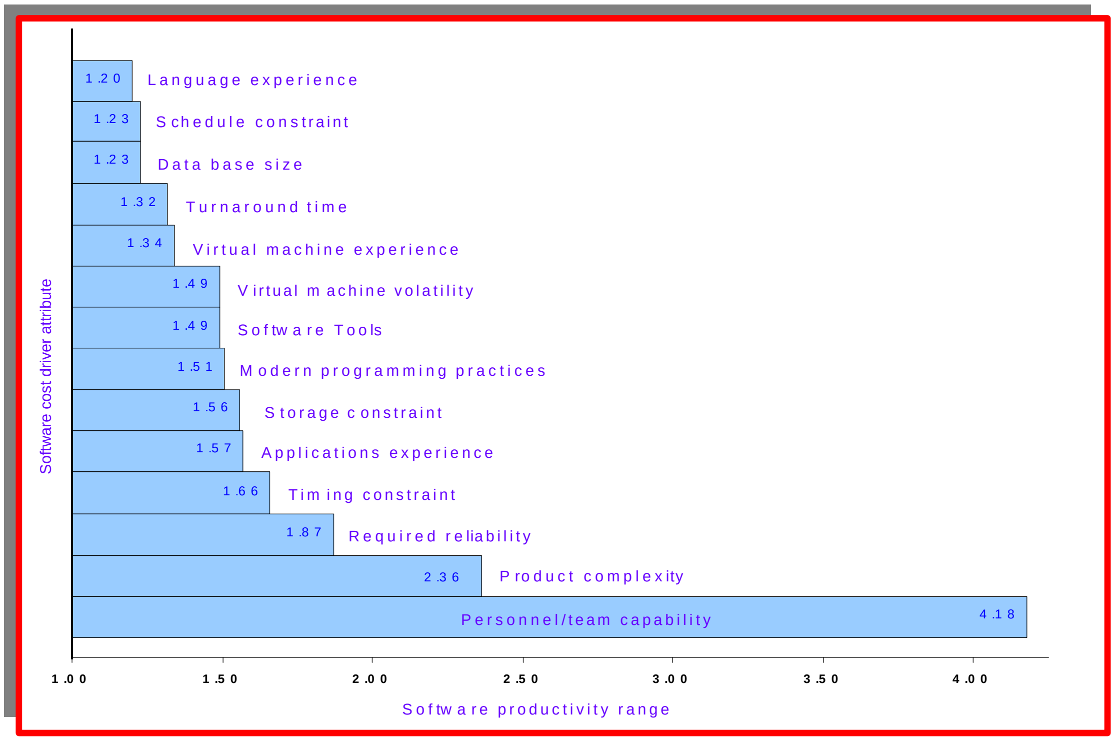

Figure 13-6. Software Productivity Factors (Weighted Cost Multipliers) [BOEHM81]

The third element in the equation, the exponent *Sigma*, is 1.2 for embedded software in Boehm's COCOMO model.

NOTE: COCOMO II, an updated version of COCOMO that allows methodologies other than the classic waterfall (such as object oriented, iterative, incremental), is available.

For large aerospace systems the value amounts to a fairly costly exponent. When you double the size of software, you multiply the cost by $2^{1.2}$, which is 2.3. In other words, the cost is doubled, plus a 30% penalty for size. The size penalty, according to Boehm, results from inefficiency influences that are a product of size. The bigger the software development, the bigger the integration issues, the bigger the team you need, the less efficient they become.

> *"You bring a bigger team on board. The people on the team spend more time talking to each other. There is more learning curve effect as you bring in people who don't know what you are building. There is a lot of thrashing in the process any time there is a proposed change of things that haven't been determined that people are trying to determine. Any time you do have a change, there are ripple effects that are more inefficient on big programs than on small programs."* [BOEHM89]

13.4.7 Cost and Schedule

As previously discussed, the driving factors in DoD software development have always been cost, schedule, or both. A typical DoD scenario has been for the software development schedule to be accelerated to support the overall program schedule, increasing the cost of the software and reducing quality. Because *cost is the key issue in any development program*, it must be reviewed carefully as part of the program review and approval process. As Benjamin Franklin explained:

> *"I conceive that the great part of the miseries of mankind are brought upon them by the false estimates they have made of the value of things."* — Benjamin Franklin [FRANKLIN33]

To avoid the miseries of a runaway program, *you must carefully plan for and control the cost and schedule of your software development effort*. These measures are important for determining and justifying the required funding, to determine if a specific proposal is reasonable, and to insure that the software development schedule is consistent with the overall system schedule. Cost measures should also be used to evaluate whether the developer has the appropriate mix and quantity of assigned staff, and to develop a reasonable program cost and schedule baseline for effective and meaningful program management.

While size is by far the most significant driver of cost and schedule, other factors impact them as well. These factors are usually more qualitative in nature and address the development and operational environments as well as the software's characteristics. Most software cost estimating models use these factors to determine environmental and complexity factors which are in turn used in computations to calculate effort and cost.

Multi-source cost and schedule estimation is the use of multiple, independent organizations, techniques, and models to estimate cost and schedule, including analysis and iteration of the differences between estimates. Whenever possible, multiple sources should be used for estimating any unknowns, not just cost and schedule. Errors or omissions in estimates can often be identified by comparing one with another. Comparative estimates also provide a sounder set of *"should-costs"* upon which to control software development. As with size estimates, assessment from alternate sources (such as program office software technical staff, prime or subcontractors, or professional consulting firms) is advisable for cost and schedule. Reassessments throughout the program life cycle improve the quality of estimates as requirements become better understood and refined. The following summarizes the resources you should consider when costing software development.

- **Human resources**. This includes the number and qualifications of the people required, as well as their functional specialties. Boehm asserts that *human resources are the most significant cost drivers on a software development effort*. [BOEHM81] Development personnel skills and experience (reflected in their productivity) have the greatest effect on cost and schedule.
- **Hardware resources**. This includes development (host) and target computers, and compilers. Hardware resources used to be major cost drivers when development personnel needed to share equipment with multiple constituencies. Now that virtually everyone has a PC or workstation on his or her desk, the issue is whether the target computer significantly differs from the development computer. For instance, if the target machine is an air or spaceborne system, the actual CPU may be technology-driven and not usable for all required development activities.

- **Software resources**. Software is also used as a tool to develop other software. CASE tools needed for development, test, and code generation must be considered. Your toolset might include: business systems planning tools, program management tools, support tools, analysis and design tools, programming tools, integration and test tools, prototyping and simulation tools, maintenance tools, cost/schedule estimating tools, and architectural tools.
- **Reusable resources**. Reusable assets are a valuable resource that must be considered in determining your cost requirements. This includes the assets you will develop for future reuse by other programs, as well as searching the reuse repositories for existing code that can be integrated into your development. Reusable assets will have significant impact on your program cost and schedule.

Schedule measurements track the contractor's performance towards meeting commitments, dates, and milestones. Milestone performance metrics give you a graphical portrayal (data plots and graphs) of program activities and planned delivery dates. It is essential that what constitutes progress slippage and revisions is understood and agreed upon by both the developer and the Government. Therefore, entry and exit criteria for each event or activity must be agreed upon at contract award. A caution in interpreting schedule metrics is to keep in mind that many activities occur simultaneously. Slips in one or more activities usually impact on others. Look for problems in process and *never, never sacrifice quality for schedule!*

13.4.8 Scrap and Rework

A major factor in both software development cost and schedule is that which is either scrapped or reworked. The costs of conformance are the normal costs of preventing defects or other conditions that may result in the scrapping or reworking of the software. The costs of nonconformance are those costs associated with redoing a task due to the introduction of an error, defect, or failure on initial execution (including costs associated with fixing failures that occur after the system is operational, i.e., scrap and rework cost).

NOTE: Good planning requires consideration of the "*rework cycle.*" For iterative development efforts, rework can account for the majority of program work content and cost!

Rework costs are very high. Boehm's data suggest rework costs are about 40% of all software development expenditures. Defects that result in rework are one of the most significant sources of risk in terms of cost, delays, and performance. You must encourage and demand that your software developer effectively measures and controls defects. Rework risk can be controlled by:

- Implementing procedures to identify defects as early as possible;
- Examining the root causes of defects and introducing process improvements to reduce or eliminate future defects; and
- Developing incentives that reward contractors/developers for early and comprehensive defect detection and removal.

There are currently no cost estimating models available that calculate this substantial cost factor. However, program managers must measure and collect the costs associated with software scrap and rework throughout development. First, it makes good sense to monitor and track the cost of defects, and thereby to incentivize closer attention to front-end planning, design, and other defect preventive measures. Second, by collecting these costs across all software development programs, parametric models can be designed to better help us plan for and assess the acquisition costs associated with this significant problem.

NOTE: See Volume 2, Appendix O, *"Swords & Plowshares; The Rework Cycles of Defense and Commercial Software Development Projects."*

13.4.9 Support

Software supportability progress can be measured by tracking certain key supportability characteristics. With these measures, both the developer and the acquirer obtain knowledge which can be focused to control supportability.

- **Memory size**. This metric tracks spare memory over time. The spare memory percentage should not go below the specification requirement.
- **Input/output**. This metric tracks the amount of spare I/O capacity as a function of time. The capacity should not go below the specification requirement.
- **Throughput**. This metric tracks the amount of throughput capacity as a function of time. The capacity should not go below specification requirements.
- **Average module size**. This metric tracks the average module size as a function of time. The module size should not exceed the specification requirement.
- **Module complexity**. This metric tracks the average complexity figure over time. The average complexity should not exceed the specification requirement.
- **Error rate**. This metric tracks the number of errors compared to number of errors corrected over time. The difference between the two is the number of errors still open over time. This metric can be used as a value for tested software reliability in the environment for which it was designed.
- **Supportability**. This metric tracks the average time required to correct a deficiency over time. The measure should either remain constant or the average time should decrease. A decreasing average time indicates supportability improvement.
- **Lines-of-code changed**. This metric tracks the average lines-of-code changed per deficiency corrected when measured over time. The number should remain constant to show the complexity is not increasing and that ease of change is not being degraded.

13.4.9.1 Define the Collection Process of the Measurement Data

From the definition of the specific measures, collection process requirements can be identified. These requirements are used to guide the development of the measurement data collection process.

13.4.9.2 Implementation

Measurement process implementation is controlled by the measurement plan. The implementation activities are discussed below.

13.4.9.3 Collect the Measurement Data

This activity is simply the execution of the measurement data collection process.

As stated above, metrics are representations of the software and the software development process that produces them — the more mature the software development process, the more advanced the metrics process. A well-managed process, with a well-defined data collection effort embedded within it, provides better data and more reliable metrics. Accurate data collection is the basis of a good metrics process. Figure 13-7 illustrates the variety of software quality metrics and management indicators that were collected and tracked for all F-22 weapon systems functions.

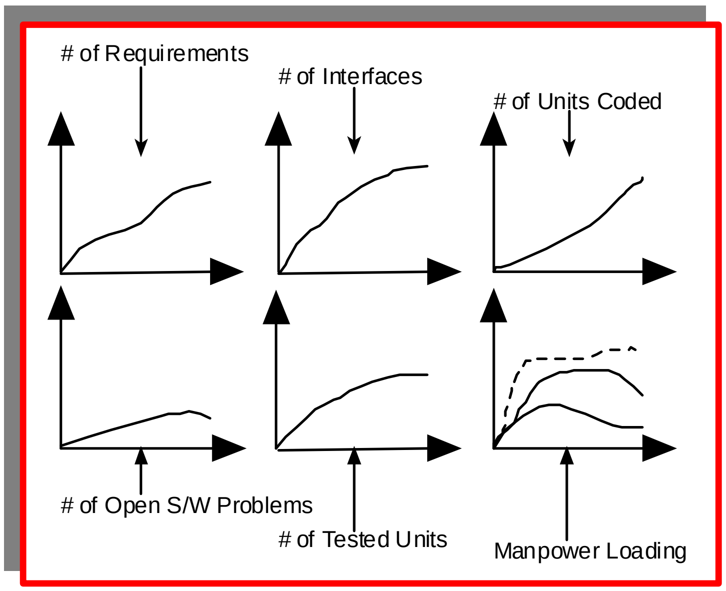

Figure 13-7. F-22 Data Collection for Software Development Tracking

To summarize, data collection must be woven into the developer's process. For instance, count the number of software units to be built by looking at the Software Design Description (SDD) and limit the use of *out-of-range* data. Also, avoid collecting data that are derived, ill-defined, or cannot be traced directly back to the process. As the development progresses, new tools or data collection techniques may emerge. If new data collection methods are employed, the data collection efforts must be tailored to these techniques. In addition, as the data collection changes, your understanding of what those data mean must also change. [ROZUM92]

13.4.9.4 Analyze the Measurement Data to Derive Indicators

Indicators are derived from the analysis performed on the measurement data. The quality of the indicator is tied to the rigor of the analysis process.

Both objective and subjective measures are important to consider when assessing the current state of your program. Objective data consists of actual item counts (e.g., staff hours, SLOC, function points, components, test items, units coded, changes, or errors) that can be independently verified. Objective data are collected through a formal data collection process. Subjective data are based on an individual's (or group's) feeling or understanding of a certain characteristic or condition (e.g., level of problem difficulty, degree of new technology involved, stability of requirements). Objective and subjective data together serve as a system of checks and balances throughout the life cycle. If you are a resourceful manager, you will depend on both to get an accurate picture of your program's health. Subjective data provide critical information for interpreting and validating objective data; while objective data provide true counts that may cause you to question your subjective understanding and investigate further.

Analysis of the collected data must determine which issues are being addressed, and if new issues have emerged. Before making decisions and taking action from the data, you must thoroughly understand what the metrics mean. To understand the data, you must:

- Use multiple sources to validate the accuracy of your data and to determine differences and causes in seemingly identical sets of data. For instance, when counting software defects by severity, spot check actual problem reports to make sure the definition of severity levels is being followed and properly recorded.
- Study the lower-level data collection process, understand what the data represent, and how they were measured.
- Separate collected data and their related issues from program issues. There will be issues about the data themselves (sometimes negating the use of certain data items). However, do not get bogged down in data issues. You should concentrate on program issues (and the data items in which you have confidence) to provide the desired insight.
- Do not assume data from different sources (e.g., from SQA or subcontractors) are based on the same definitions, even if predefined definitions have been established. You must re-verify definitions and identify any variations or differences in data from outside sources when using them for comparisons.
- Realize development processes and products are dynamic and subject to change. Periodic reassessment of your metrics program guarantees that it evolves. Metrics are only meaningful if they provide insight into your current list of prioritized issues and risks.

The availability of program metrics is of little or no value unless you also have access to models (or norms) that represent what is expected. The metrics are used to collect historical data and experience. As their use and program input increases, they are used to generate databases of information that become increasingly more accurate and meaningful. Using information extracted from these databases, you are able to gauge whether measurement trends in your program differ from similar past programs and from expected models of optimum program performance within your software domain. Databases often contain key characteristics upon which models of performance are designed. Cost data usually reflect measures of effort. Process data usually reflect information about the programs (such as methodology, tools, and techniques used) and information about personnel experience and training. Product data include size, change, and defect information and the results of statistical analyses of delivered code.

Figure 13-8 illustrates the possibilities for useful comparison by using metrics, based on available program histories. By using models based on completed software developments, the initiation and revision of your current plans and estimates will be based on "*informed data.*" As you gather performance data on your program, you should compare your values with those for related programs in historical databases. The comparisons in Figure 13-8 should be viewed collectively, as one component of a *feedback-and-control* system that leads to revisions in your management plan. To execute your revised plans, you must make improvements in your development process which will produce adjusted measures for the next round of comparisons.

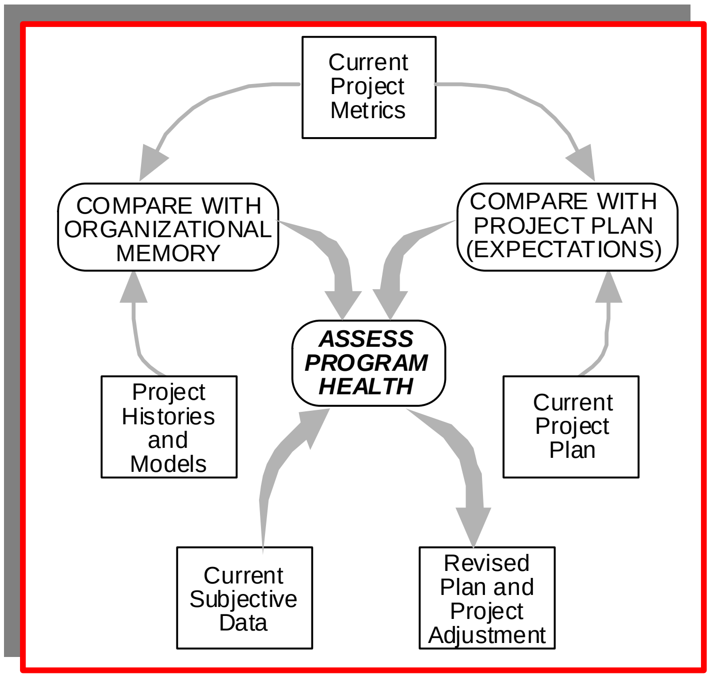
Current Project Metrics
COMPARE WITH ORGANIZATIONAL MEMORY
COMPARE WITH PROJECT PLAN (EXPECTATIONS)
ASSESS PROGRAM HEALTH
Project Histories and Models
Current Project Plan
Current Subjective Data
Revised Plan and Project Adjustment

Figure 13-8. Management Process with Metrics

13.4.9.5 Manage the Measurement Data and Indicators

The measurement data and the indicators must be properly managed according to the requirements identified in the measurement plan.

13.4.9.6 Report the Indicators

Once the indicators are derived, they are made available to all those affected by the indicators or by the decisions made because of the indicators.

13.4.9.7 Evaluation

The value of any measurement process is in direct proportion to the value the indicators have to the users of the indicators.

13.4.9.8 Review Usability of Indicators

Initially, the selection of indicators, analysis methods, and specific measurement data may be a ‘best guess’ whether or not they meet the information needs specified. Over time, through a review of the usefulness of the indicators, the selection can be refined such that there is a high correlation between the indicators selected and the information needs. This will be an iterative process.

13.4.9.9 Begin Measurement Activities Early

Measurement, as a practice, must begin at the project level. Other activities such as project planning, project monitoring, etc. rely on the information gathered by the measurement process.

As the organization begins to focus on process standardization and improvement, the measurement program grows. Additional "information needs" are identified and fed to the measurement process. More mature processes such as quantitative process management, process innovation deployment rely on indicators to determine effectiveness, efficiency, and productivity, etc.

13.5 Cautions About Metrics

Software measures are valuable for gaining insight into software development; however, they are not a solution to issues in and of themselves. To implement a metrics program effectively, you must be aware of limitations and constraints.

- **Metrics must be used as indicators, not as absolutes**. Metrics should be used to prompt additional questions and assessments not necessarily apparent from the measures themselves. For instance, you may want to know why the staff level is below what was planned. Perhaps there is some underlying problem, or perhaps original manpower estimates need adjusting. Metrics cannot be applied in a vacuum, but must be combined with program knowledge to reach correct conclusions.
- **Metrics are only as good as the data that support them**. Input data must be timely, consistent, and accurate. A deficiency in any of these areas can skew the metrics derived from the data and lead to false conclusions.
- **Metrics must be understood to be of value**. This means understanding what the low-level measurement data represent and how they relate to the overall development process. You must look beyond the data and measurement process to understand what is really going on. For example, if there is a sharp decrease in defect detection and an increase in defect resolution and close out, you might conclude that the number of inserted defects is decreasing. However, in a resource-constrained environment, the defect discovery rate may have dropped because engineering resources were temporarily moved from defect detection (e.g., testing) to defect correction.
- **Metrics should not be used to judge your contractor (or individual) performance**. Measurement requires a team effort. While it is necessary to impose contractual provisions to implement software measurement, it is important not to make metrics a controversial issue between you and your contractor. *Support of the measurement process will be jeopardized if you "shoot-the-messenger."* Measurements should be used to identify problem areas and for improving the process and product. While metrics may deal with personnel and organizational data, these data must be used for constructive, process-oriented decision-making, rather than for placing blame on individuals or teams.
- **Metrics cannot identify, explain, or predict everything**. Metrics must be used in concert with sound, hands-on management practice. They are only valuable if used to augment and enhance intimate process knowledge and understanding.

- **Analysis of metrics should NOT be performed exclusively by the contractor**. Ideally, the contractor you select will already have a metrics process in place. As mentioned above, you should implement your own independent metrics analysis process because:
  - Metrics analysis is an iterative process reflecting issues and problems that vary throughout the development cycle;
  - The natural tendency of contractors is to present the program in the best light; therefore, independent government analysis of the data is necessary to avoid misrepresentation; and
  - Metrics analysis must be issue-driven and the government and contractor have inherently different issue perspectives.
- **Direct comparisons of programs should be avoided**. No two programs are alike; therefore, any historical data must be tailored to your program specifics to derive meaningful projections. *[Conversely, do not tailor your data to match historical data.]* However, metrics from other programs should be used as a means to establish *normative values* for analysis purposes.
- **A single metric should not be used**. No single metric can provide the insight needed to address all program issues. Most issues require multiple data items to be sufficiently characterized. Because metrics are interrelated, you must correlate trends across multiple metrics. [ROZUM92]

13.6 Addressing Measurement in the Request for Proposal (RFP)

Your RFP should define the indicators and metrics the Government needs to track progress, quality, schedule, cost, and maintainability. What you should look for when analyzing an offeror's Metrics Usage Plan is "*control*." Through measurement, the process's internal workings are defined and assessed. If an effective process improvement plan is executed (which requires appropriate measurements be taken) data are collected and analyzed to predict process failures. Therefore, the offeror must have a corporate mechanism implemented in a systematic manner that performs orderly process control and methodical process improvement. This can be identified by the measurement methods the company uses to assess the development process, analyze the data collected, and feed back corrections for problems within the process. [CAREY92]

After you have identified your program issues (and before contract award) you and your future contractor must agree on entry and exit criteria definitions for the proposed software development process and products. Entry and exit criteria must also be defined for all data inputs, standards of acceptance, schedule and progress estimation, and data collection and analysis methods. For instance, there must be an agreement on the definition of source lines-of-code and how and when SLOC will be estimated or counted. *The entire collection and analysis process — all definitions, decisions, and agreements — should be written into the contract*

Make sure the software quality metrics and indicators they employ include a clear definition of component parts (e.g., SLOC), are accurate and readily collectible, and span the development spectrum and functional activities. They must identify metrics early and apply them at the beginning of the system engineering and software implementation process. They should also develop a software Metrics Usage Plan before contract award.

13.7 References

[Ada/C++91] *Ada and C++: A Business Case Analysis*, Office of the Deputy Assistant Secretary of the Air Force, Washington, DC, June 1991

[ALBRECHT79] Albrecht, A.J., "Measuring Application Development Productivity," Proceedings of the IBM Applications Development Symposium, Monterey, California, October 1979

[BOEHM81] Boehm, Barry W., Software Engineering Economics, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981

[BOEHM89] Boehm, Barry W., as quoted by Ware Myers, "Software Pivotal to Strategic Defense," IEEE Computer, January 1989

[CAMPBELL95] Campbell, Luke and Brian Koster, "Software Metrics: Adding Engineering Rigor to a Currently Ephemeral Process," briefing presented to the McGrummwell F/A-24 CDR course, 1995

[CAREY92] Carey, Dave and Don Freeman, "Quality Measurements in Software," G. Gordon Schulmeyer and James I. McManus, eds., Total Quality Management for Software, Van Nostrand Reinhold, New York, 1992

[DeMARCO86] DeMarco, Tom, Controlling Software Projects, Yourdon Press, New York, 1986

[FERENS00] Ferens, Daniel V. and David S. Christensen, "Does Calibration Improve the Predictive Accuracy of Software Cost Models?", CrossTalk, April 2000

[FRANKLIN33] Franklin, Benjamin, Poor Richard's Almanac, 1733

[GLASS92] Glass, Robert L., Building Quality Software, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1992

[HETZEL93] Hetzel, Bill, Making Software Measurement Work: Building an Effective Measurement Program, QED Publishing Group, Boston, 1993

[HUMPHREY89] Humphrey, Watts S., Managing the Software Process, The SEI Series in Software Engineering, Addison-Wesley Publishing Company, Inc., 1989

[IEEE90] IEEE Standard Glossary of Software Engineering Terminology, IEEE Std 610.12-1990, Institute of Electrical and Electronic Engineers, Inc., New York, NY, December 10, 1990

[JONES91] Jones, Capers, Applied Software Measurement, McGraw-Hill, New York, 1991

[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

[MARCINIAK90] Marciniak, John J. and Donald J. Reifer, Software Acquisition Management: Managing the Acquisition of Custom Software Systems, John Wiley & Sons, Inc., New York, 1990

[PRESSMAN92] Pressman, Roger S., Software Engineering: A Practitioner's Approach, Third Edition, McGraw-Hill, Inc., New York, 1992

[PUTNAM92] Putnam, Lawrence H., and Ware Myers, Measures for Excellence: Reliable Software On Time, Within Budget, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1992

[ROZUM92] Rozum, James A., Software Measurement Concepts for Acquisition Program Managers, Technical Report CMU/SEI-92-TR-11/ESD-TR-92-11, Carnegie-Mellon University, Software Engineering Institute, Pittsburgh, Pennsylvania, June 1992