Notes from *Managing the Software Process, by* Watts Humphrey

SEI Series in Software Engineering, Addison-Wesley, 1990.

I*f you don't know where you are, a map won't help*. - W. Humphrey

Chapter 1: A Software Maturity Framework

Fundamentally, software development must be predictable.

The software process is the set of tools, methods, and practices we use to produce a software product. The objectives of software process management are to produce products according to plan while simultaneously improving the organization's capability to produce better products.

The basic principles are those of statistical process control. A process is said to be stable or under statistical control if its future performance is predictable within established statistical limits.When a process is under statistical control, repeating the work in roughly the same way will produce roughly the same result. To obtain consistently better results, it is necessary to improve the process. If the process is not under statistical control, sustained progress is not possible until it is.

Lord Kelvin - “When you can measure what you are speaking about, and express it in numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind; it may be the beginning of knowledge, but you have scarcely in your thoughts advanced the stage of science.” (But, your numbers must be reasonably meaningful.)

The mere act of measuring human processes changes them because of people's fears, and so forth.

Measurements are both expensive and disruptive; overzealous measurements can disrupt the process under study.

SOFTWARE PROCESS IMPROVEMENT

To improve their software capabilities, organizations must take six steps;

- Understand the current status of the development processes
- Develop a vision of the required process
- Establish a list of required process improvement actions in order of priority
- Produce a plan to accomplish the required action
- Commit the resources to execute the plan
- Start over at the first step.

PROCESS MATURITY LEVELS

1. Initial. Until the process is under statistical control, orderly progress in process improvement is not possible. Must at least achieve rudimentary predictability of schedules and costs.

2. **Repeatable**. The organization has achieved a stable process with a repeatable level of statistical control by initiating rigorous project management of commitments, costs, schedules, and changes.

3. **Defined.** The organization has defined the process as a basis for consistent implementation and better understanding. At this point, advanced technology can be introduced.

4. **Managed**. The organization has initiated comprehensive process measurements and analysis. This is when the most significant quality improvements begin.

5. **Optimizing**. The organization now has a foundation for continuing improvement and optimization of the process.

These levels are selected because they

- Represent the historical phases of evolutionary improvement of real software organizations
- Represent a measure of improvement that is reasonable to achieve from a prior level
- Suggest improvement goals and progress measures
- Make obvious a set of intermediate improvement priorities once an organization's status in this framework is known

The Initial Process (Level 1)

Usually ad hoc and chaotic - Organization operates without formalized procedures, cost estimates, and project plans. Tools are neither well intergrated with the process nor uniformly applied. Change control is lax, and there is little senior management exposure or understanding of the problems and issues. Since many problems are deferred or even forgotten, software installation and maintenance often present serious problems.

While organizations at this level may have formal procedures for planning and tracking work, there is no management mechanism to insure they are used. Procedures are often abandoned in a crisis in favor of coding and testing. Level 1 organizations don't use design and code inspections and other techniques not directly related to shipping a product.

Organizations at Level 1 can improve their performance by instituting basic project controls. The most important ones are

- Project management
- Management oversight
- Quality assurance
- Change control

The Repeatable Process (Level 2)

This level provides control over the way the organization establishes plans and commitments. This control provides such an improvement over Level 1 that the people in the organization tend to believe they have mastered the software problem. This strength, however, stems from their prior experience in doing similar work. Level 2 organizations face major risks when presented with new challenges.

Some major risks:

- New tools and methods will affect processes, thus destroying the historical base on which the organization lies. Even with a defined process framework, a new technology can do more harm than good.
- When the organization must develop a new kind of product, it is entering new territory.
- Major organizational change can be highly disruptive. At Level 2, a new manager has no orderly basis for understanding an organization's operation, and new members must learn the ropes by word of mouth.

Key actions required to advance from Repeatable to the next stage, the Defined Process, are:

- Establish a process group: A process group is a technical resource that focuses heavily on improving software processes. In most software organizations, all the people are generally devoted to product work. Until some people are assigned full-time to work on the process, little orderly progress can be made in improving it.
- Establish a software development process architecture (or development cycle) that describes the technical and management activities required for proper execution of the development process. The architecture is a structural decomposition of the development cycle into tasks, each of which has a defined set of prerequisites, functional decompositions, verification procedures, and task completion specifications.

•Introduce a family of software engineering methods and technologies. These include design and code inspections, formal design methods, library control systems, and comprehensive testing methods. Prototying and modern languages should be considered.

The Defined Process (Level 3)

The organization has the foundation for major and continuing change. When faced with a crisis, the software teams will continue to use the same process that has been defined.

However, the process is still only qualitative; there is little data to indicate how much is accomplished or how effective the process is. There is considerable debate about the value of software process measurements and the best one to use.

The key steps required to advance from the Defined Process to the next level are:

- Establish a minimum set of basic process measurements to identify the quality and cost parameters of each process step. The objective is to quantify the relative costs and benefits of each major process activity, such as the cost and yield of error detection and correction methods.
- Establish a process database and the resources to manage and maintain it. Cost and yield data should be maintained centrally to guard against loss, to make it available for all projects, and to facilitate process quality and productivity analysis.

- Provide sufficient process resources to gather and maintain the process data and to advise project members on its use. Assign skilled professionals to monitor the quality of the data before entry into the database and to provide guidance on the analysis methods and interpretation.
- Assess the relative quality of each product and inform management where quality targets are not being met. Should be done by an independent quality assurance group.

The Managed Process (Level 4)

Largest problem at Level 4 is the cost of gathering data. There are many sources of potentially valuable measure of the software process, but such data are expensive to collect and maintain.

Productivity data are meaningless unless explicitly defined. For example, the simple measure of lines of source code per expended development month can vary by 100 times or more, depending on the interpretation of the parameters.

When different groups gather data but do not use identical definitions, the results are not comparable, even if it makes sense to compare them. It is rare when two processes are comparable by simple measures. The variations in task complexity caused by different product types can exceed five to one. Similarly, the cost per line of code for small modifications is often two to three times that for new programs.

Process data must not be used to compare projects or individuals. Its purpose is too illuminate the product being developed and to provide an informed basis for improving the process. When such data are used by management to evaluate individuals or terms, the reliability of the data itself will deteriorate.

The two fundamental requirements for advancing from the Managed Process to the next level are:

- Support automatic gathering of process data. All data is subject to error and omission, some data cannot be gathered by hand, and the accuracy of manually gathered data is often poor.
- Use process data to analyze and to modify the process to prevent problems and improve efficiency.

The Optimizing Process (Level 5)

To this point software development managers have largely focused on their products and will typically gather and analyze only data that directly relates to product improvement. In the Optimizing Process, the data are available to tune the process itself.

For example, many types of errors can be identified far more economically by design or code inspections than by testing. However, some kinds of errors are either uneconomical to detect or almost impossible to find except by machine. Examples are errors involving interfaces, performance, human factors, and error recovery.

So, there are two aspects of testing: removal of defects and assessment of program quality. To reduce the cost of removing defects, inspections should be emphasized. The role of functional and system testing should then be changed to one of gathering quality data on the program. This involves studying each bug to see if it is an isolated problem or if it indicates design problems that require more comprehensive analysis.

With Level 5, the organization should identify the weakest elements of the process and fix them. Data are available to justify the application of technology to various critical tasks, and numerical evidence is available on the effectiveness with which the process has been applied to any given product.

Chapter 2: The Principles of Software Process Change

People:

- The best people are always in short supply
- You probably have about the best team you can get right now.
- With proper leadership and support, most people can do much better than they are currently doing

Design:

- Superior products have superior design. Successful products are designed by people who understand the application (domain engineer).
- A program should be viewed as executable knowledge. Program designers should have application knowledge.

The Six Basic Principles of Software Process Change:

- Major changes to the process must start at the top.
- Ultimately, everyone must be involved.
- Effective change requires great knowledge of the current process
- Change is continuous
- Software process changes will not be retained without conscious effort and periodic reinforcement
- Software process improvement requires investment

Continuous Change:

- Reactive changes generally make things worse
- Every defect is an improvement opportunity
- Crisis prevention is more important than crisis recovery

Software Processes Changes Won't Stick by Themselves

The tendency for improvements to deteriorate is characterized by the term *entrophy* (Webster's: a measure of the degree of disorder in a...system; entrophy always increases and available energy diminishes in a closed system.).

New methods must be carefully introduced and periodically monitored, or they to will rapidly decay.

Human adoption of new process involves four stages:

- Installation - Initial training
- Practice - People learn to perform as instructed
- Proficiency - Traditional learning curve
- Naturalness - Method ingrained and performed without intellectual effort.

It Takes Time, Skill, and Money!

- To improve the software process, someone must work on it
- Unplanned process improvement is wishful thinking
- Automation of a poorly defined process will produce poorly defined results
- Improvements should be made in small steps
- Train!!!!

Some Common Misconceptions about the Software Process

- We must start with firm requirements
- If it passes test it must be OK
- Software quality can't be measured
- The problems are technical
- We need better people
- Software management is different

FIRM REQUIEMENTS - A software perversity law seems to be the more firm the specifications, the more likely they are to be wrong. With rare exceptions, requirements change as the software job progresses. Just by writing a program, we change our perceptions of the task. Requirements cannot be firm because we cannot anticipate the ways the tasks will change when they are automated.

For large-scale programs, the task of stating a complete requirement is not just difficult; it is impossible. Generally, we must develop software incrementally.

However, we must have stability long enough to build and test a system. However, if we freeze requirements too early, later retrofits are expensive.

SOFTWARE MANAGEMENT IS DIFFERENT - Management must not view it as black art. Must insist on tracking plans and reviews.

Champions, Sponsors, and Agents

- Champions - Ones who initiate change. They bring management's attention to the subject, obtain the blessing of a sponsor, and establish the credibility to get the change program launched. The champion maintains focus on the goal, strives to overcome obstacles, and refuses to give up when the going gets tough.
- Sponsors - Senior manager who provides resources and official backing. Once a sponsor is found, the champion's job is done; it is time to launch the change process.

- Agents - Change agents lead the planning and implementation. Agents must be enthusiastic, technically and politically savvy, respected by others, and have management's confidence and support.

Elements of Change

- Planning
- Implementation
- Communication

Chapter 3: Software Process Assessment

Process assessments help software organizations improve themselves by identifying their crucial problems and establishing improvement priorities. The basic assessment objectives are:

- Learn how the organization works
- Identify its major problems
- Enroll its opinion leaders in the change process

The essential approach is to conduct a series of structured interviews with key people in the organization to learn their problems, concerns, and creative ideas.

ASSESSMENT OVERVIEW

A software assessment is not an audit. Audit are conducted for senior managers who suspect problems and send in experts to uncover them. A software process assessment is a review of a software organization to advise its management and professionals on how they can improve their operation.

The phases of assessment are:

- Preparation - Senior management agrees to participate in the process and to take actions on the resulting recommendations or explain why not. Concludes with a training program for the assessment team
- Assessment - The on-site assessment period. It takes several days to two or more weeks. It concludes with a preliminary report to local management.

•Recommendations - Final recommendations are presented to local managers. A local action team is then formed to plan and implement the recommendations.

Five Assessment Principles:

- The need for a process model as a basis for assessment
- The requirement for confidentiality
- Senior management involvement
- An attitude of respect for the views of the people in the organization be assessed
- An action orientation

Start with a process model - Without a model, there is no standard; therefore, no measure of change.

Observe strict confidentiality - Otherwise, people will learn they cannot speak in confidence. This means managers can't be in interviews with their subordinates.

Involve senior management - The senior manager (called *site manager* here) sets the organizations priorities. The site manager must be personally involved in the assessment and its follow-up actions. Without this support, the assessment is a waste of time because lasting improvement must survive periodic crises.

Respect the people in the assessed organization - They probably work hard and are trying to improve. Do not appear arrogant; otherwise, they will not cooperate and may try to prove the team is ineffective. The only source of real information is from the workers.

Assessment recommendations should highlight the three or four items of highest priority. Don't overwhelm the organization. The report must always be in writing.

Implementation Considerations - The greatest risk is that no significant improvement actions will be taken (the "disappearing problem" syndrome). Superficial changes won't help. A small, full-time group should guide the implementation effort, with participation from other action plan working groups. Don't forget that site managers can change or be otherwise distracted, so don't rely on that person solely, no matter how committed.

Chapter 4: The Initial Process

In the Initial Process, professionals are driven from crisis to crisis by unplanned priorities and unmanaged change. Such groups seldom meet commitments. They may deliver on schedule occasionally, but this is the exception and normally due to herculean individual efforts rather than the strength of the organization. Senior management makes unreasonable demands while cutting resources. Schedules are the only priority, but when they do deliver, nobody likes the results. Many first-class professionals leave as soon as they can. Most of the rest have strong local ties or have just given up and collect their pay.

Customers tolerate such organizations only if there are no alternatives.

WHY ORGANIZATIONS ARE CHAOTIC

Simplest explanation: People don't like to admit they don't know. When political pressure amounts for a commitment, it is difficult to say you don't know. So, people tend to promise products they can't deliver.

While lack of a commitment discipline is the common reason for chaotic behavior, there are other forces:

- Under extreme pressure, software managers make a guess instead of a plan. The guess is usually low, so chaos develops.
- When things get rough, there may be a strong temptation to believe in magic. A savior or new technology may be the answer.
- The scale of software projects normally follows an escalating cycle.

- Programs take more code than expected;
- As the programs become larger, new technical and management issues arise.
- Since these are unlike previous experience, they are a surprise.
- Even after a higher maturity level is reached, new management, increased competition, or new technical challenges put pressure on processes; then, an organization may revert to the Initial Process.

UNPLANNED COMMITMENTS

When a new feature seems simple, management may just commit to do it without planning. Often, the change is far more complicated than thought. This results in confusion, changed priorities, and stress. The users and senior management become frustrated and unwilling to listen to excuses when this becomes a pattern.

GURUS

Gurus can make this worse because they run projects out of their heads. With nothing written down, everyone comes to them for guidance. At the breaking point, the limits of this intuitive approach have been reached and there is no simple recovery.

MAGIC

"No silver bullets" - Brooks

PROBLEMS OF SCALE

Software size is insidious because of its impact on the development process. Building small programs does not lead to discipline needed to build large software systems.

As software products become larger, the progressive level of scale are roughly as follows:

- One person knows the details of the program.
- One person understands it but can't remember it so the design must be documented
- One person understands the overall design of the program, but the details of its component modules are each understood by separate experts.

•A large software project is defined and understood at the product management level, but a separate team understands the characteristics and design of each of its component programs.

•With software systems, the high-level design may be well defined and understood by the system management team, but each of the component products is only understood by the respective product management organizations.

•When the system is very large and has evolved through many versions, there may be no one who understands it.

As software knowledge is more widely distributed:

- Common notations are needed for precise communication.These standards must be documented, interpreted, and updated.
- Conflicts in standards must be identified and resolved
- Standards changes must be controlled and distributed

With large-scale software, similar control is needed for requirements, design, code, and test.

As software size increases, prototypes or multiple releases are needed because:

- The total function cannot be implemented at one time.
- Some needs cannot be understood without operational experience on a partial system.
- Some design issues cannot be resolved until a preliminary system has been built and run.
- A release discipline helps sort out user priorities.
- No one successfully builds software in one shot, anyway.

With multiple releases, new complications arise:

- The requirements must be phased to meet end user needs.
- Each software component design must synchronize with these needs.
- Product interdependencies are orchestrated to meet release functional requirements
- The build and integration plans are scheduled around these interdependencies
- Early system drivers are scheduled to meet component test needs.
- The tight schedule requires subsequent release development to start before the prior releases are finished.

SOFTWARE PROCESS ENTROPHY

As we build new systems, we learn what we should have built. This is a good reason for prototypes. This can help avoid chaos.

THE WAY OUT

- Apply systematic project management - the work must be estimated, planned, and managed.
- Adhere to careful change control - changes must be controlled, including requirements, design, implementation, and test.
- Utilize independent software assurance - an independent technical means is required to assure that all essential project activities are properly performed.

THE BASIC PRINCIPLES FOR CONTROLLING CHAOS IN SOFTWARE PRGANIZATIONS ARE:

- Plan the work
- Track and maintain the plan
- Divide the work into individual parts
- Precisely define the requirements for each part
- Rigorously control the relationships among the parts
- Treat software development as a learning process
- Recognize what you don't know
- When the gap between your knowledge and the task is severe, fix it before proceeding
- Manage, audit, and review the work to ensure it is done as planned
- Commit to your work and work to your commitments
- Refine the plan as your knowledge improves

PART TWO:THE REPEATABLE PROCESS

Chapter 5: Managing Software Organizations

The basic principles of project management are:

- Each project has a plan that is based on a hierarchy of commitments;
- A management system resolves the natural conflicts between the projects and between the line and staff organizations;
- An oversight and review system audits and tracks progress against the plans.

The foundation for software project management is the commitment discipline. Commitments are supported by plans, reviews, and so forth, but commitments are met by people.

The elements of an effective commitment are:

- The person making the commitment does so willingly;
- The commitment is not made lightly;
- There is agreement between the parties on what is to be done, by whom, and when;
- The commitment is openly stated;
- The person tries to meet the commitment, even if help is needed;
- Prior to the commitment date, if it is clear that it cannot be met, advance notice is given and a new commitment is negotiated.

The software commitment process:

- All commitments for future software delivery are made personally by the organization's senior executive;
- These commitments are made only after successful completion of a formal review and concurrence process;
- An enforcement mechanism ensures these reviews and concurrences are properly conducted.

The senior manager should require evidence that the following work was done prior to approving a commitment:

- The work has been defined between the developers and the customer;
- A documented plan has been produced, including a resource estimate, a schedule, and a cost estimate;
- All directly involved parties have agreed to this plan in writing;
- Adequate planning has been done to ensure the commitment is a reasonable risk.
- An independent review ensures the planning work was done according to the organization's standards and procedures.
- The groups doing the work have or can acquire the resources needed.

Product and Period Plans

The period (operating) plan deals with technical and business issues in annual and organizational terms. Thus expenses, capital requirements, and product delivery commitments are established for each period by each organizational entity.

Product plans focus on the activities and objectives of each product. The issues are function, cost, schedule, and quality, together with related resources and checkpoints.

The distinction between period and product can be confusing. Project personnel view their work as the fundamental business of the organization and have little appreciation for period information. The project is not set up on an annual basis and often has considerable difficulty in producing annual data on items such as cost, quality, or productivity.

Organizations, however, are generally measured on a period basis, and the project must be translated into period terms for inclusion in annual budgets, plans, or stockholders reports.

The organization's direction is set in its strategy. The strategy is a period instrument that deals with the problems and issues five or more years ahead. It builds the framework for continual organizational improvement.

The Contention Process

An effective management system requires a parallel contention system to encourage the open expression of differences and their rational resolution. The principle is that the best decisions are based on full understanding of the relevant issues.

The principles of the contention system are:

- All major decisions are reviewed with the involved parties in advance, and the parties are requested to agree. Where possible, any issues are resolved before proceeding;
- When the time comes for a decision, all dissenting parties are present and asked to state their views;

- When there is no agreement, senior management determines if there is knowledgeable agreement, if any disagreeing parties are absent, or if more preparation is needed. In the latter two cases, the decision is deferred until the necessary homework is done.

This system does not generate dissention that does not already exist, but it does reveal that dissention and optimize its utility.

Management must develop a system for tracking projects and commitments (a project management system).

Chapter 6: The Project Plan

The project plan is developed at the beginning of the job and is refined as the work progresses. The logic for software project planning is:

- While requirements are initially vague, the project plan starts a mapping process from vague and incomplete to accurate and precise.
- At each subsequent refinement, resource projections, size estimates, and schedules can be more accurate.
- When the requirements are sufficiently clear, a detailed design and implementation strategy can be developed and incorporated into the plan.
- Throughout this cycle, the plan provides the framework for negotiating the time and resources to do the job.

The planning cycle:

- Initial requirements;
- Commitment control (understand a commitment's impact before agreeing);
- Work Breakdown Structure (WBS)--key elements of the plan;
- Estimate size of each product element;
- Project resources needed;
- Produce a schedule.

Project plan contents:

- Goals and objectives What to be done, for whom, by whom, by when, criteria for success
- Work breakdown structure - Quantitative estimates of the code required for each product element
- Product size estimates;
- Resource estimates;
- Project schedule.

Requirements change:

- Implement the product in small steps;
- Select each increment to support succeeding increments and/or improve requirements knowledge;
- Freeze the requirements for each incremental step before starting design;
- When the requirements change during implementation, defer the change to a subsequent increment;
- If the changes cannot be deferred, stop work, modify the requirements, revise the plan, and start again on the design.

Size Measures

- Line-of-code estimates;
- Function points

SUMARY OF FUNCTION POINTS

- Count the number of inputs, outputs, inquiries, master files, and interfaces required;
- Multiply the counts by the following factors:

Inputs	4
Outputs	5
Inquiries	4
Master files	10
Interfaces	10

- Adjust the total of these products by +25%, 0, or -25%, depending on the estimators judgement of the program's complexity

Function points are most helpful when:

- A new program development is being estimated;
- The applications involve considerable input-output or file activity;
- An experienced function point expert is available;
- Sufficient data is on hand to permit reasonably accurate conversion from function points to LOC.

ESTIMATING

Once a size measure has been established, an estimating procedure is needed. The Wideband Delphi Technique can be used:

- A group of experts is given the program's specifications and an estimation form;
- They discuss the product and any estimation issues;
- They each anonymously complete the estimation forms;
- The estimates are given to the coordinator, who tabulates the results and returns them to the experts;
- Only each expert's personal estimate is identified; all others are anonymous;
- The experts meet to discuss the results, revising their estimates as they see appropriate;
- The cycle continues until the estimates converge to an acceptable range.

The earlier the estimate, the less is known about the product and the greater the estimating inaccuracy.

Estimates are usually low. Humphrey's rules of thumb are:

CODE GROWTH BY PROJECT PHASE

Completed Project Phase	**Code Growth Range (%)**
Requirements	100-200
High-level design	75-100
Detailed design	50-100
Implementation	25-50
Function test	10-25
System test	0-10

PRODUCTIVITY FACTORS

It is very difficult to track productivity factors. A major issue is how to measure productivity. It is LOC, $/LOC, errors/ LOC,...? Usually, LOC/time is used, but that still raises issues of language, language level, application type, modified or new code product size, team size, etc.

Consider the following data from a large IBM study.

1. Relative Productivity versus Program Size and Product Class

	Size in KLOC		
Product class	<10	10-50	>50
Language	1.8	3.9	4.0
Control	1.6	1.8	2.4
Communications	1.0	1.6	2.0

Relative Productivity versus Percent of Program New or Changed and Product Class

	%New or Changed		
Product Class	<20%	20-40	40%
Language	3.0	6.0	6.6
Control	1.5	2.3	2.3
Communications	0.4	1.8	1.9

Environmental Factors

Effects of Environment on Performance

Environmental Factor	**Top 25%**	**Bottom 25%**	**All**
Floor space sq ft	78	46	63
% yes			
Quiet workspace	57	29	42
Private workspace	62	29	42
Silence phone	52	10	29
Divert calls	76	19	57
Needless interruptions	38	76	62
Workspace makes them feel appreciated	57	29	45

Also noted that productivity rates range from 1 to 11 for 90% of the projects. Others have report ranges of as high as 56 to 1.

GETTING PRODUCTIVITY DATA

- Identify a set of recently completed programs that are similar to the new program as possible in size, language, application type, team experience, etc.
- Get data on the size, in LOC, of each project. Use the same counting scheme for each program.
- For modified programs, note the percent of code modified and count only the number of new or changed LOC in the productivity calculations.
- Obtain a count of programmer months expended for each project, but be sure to include or exclude the same labor categories for each program. The requirements effort should be excluded because it is highly dependent on customer relationships and application knowledge.

Guidelines for Interpreting the Collected Data

- Even with a large experience base, productivity estimates will be only crude approximations;
- Real productivity data has high standard deviations;
- With enough data, extreme cases should be discarded;
- Most of the factors that affect productivity cannot be determined or compensated for.

SCHEDULING

Once resources needed have been calculated, the project schedule can be developed by spreading these resources over the planed product development phases. This is best done based on an organization's experience with similar projects. Published estimates can also be useful, such as the one below from B. Boehm.

Phase Distribution - Resource and Time

	Product Size			
	Small	Intermediate	Medium	Large
Phase	2K LOC	8K LOC	32 KLOC	128 KLOC
Resource:				
Product design	16%	16%	16%	16%
Programming:				
Detailed design	26	25	24	23
Code & unit test	42	40	38	36
Integration & test	16	19	22	25
Schedule:				
Product design	19	19	19	19
Programming	63	59	55	51
Integration & test	18	22	26	30

Also, some data from Martin Marietta:

Development Effort Per Program Phase

Program Phase	% Effort
Design	3.49
Detailed design	11.05
Code and unit test	23.17
Unit and integration test	27.82
Qualification test	34.47

PROJECT TRACKING

One requirement for sound project management is the ability to determine project status. Should have some check points such as:

- Module specifications complete and approved;
- Module design complete, inspected, and corrections made;
- Module unit test plan complete, reviewed, and approved;
- Module coding complete, and first compilation;
- Module code inspection complete and corrections made;
- Module through unit test and delivered to Software Configuration Management for integration in the baseline

Every checkpoint must be specific and measurable.

Plot the plan versus the actual status weekly to keep track of the schedule. Otherwise, schedules are useless.

Partial credit is not allowed. A 50% module design complete checkpoint means that design is 100% complete for 50% of the modules, not that the average level of completion of the modules is 50%.

THE DEVELOPMENT PLAN

A detailed development plan must be developed and reviewed by all participating parties. Speak now or forever hold your peace.

Chapter 7: Software Configuration Management - Part 1

The most frustrating software problems are often caused by poor software configuration management (SCM). For example, a bug fixed at one time reappears; a developed and tested feature is missing; or a fully tested program suddenly doesn't work.

SCM helps to reduce these problems by coordinating the work products of many different people who work on a common project. With such control, can get problems such as;

- Simultaneous update - When two or more programmers work separately on the same program, the last one to make changes can easily destroy the other work.
- Shared code - Often, when a bug is fixed in code shared by several programmers, some of them are not notified;

- Common code - In large systems, when common program functions are modified, all the users need to know.
- Versions - Most large programs are developed in evolutionary releases. With one release in customer use, one in test, and a third in development, bug fixes must be propagated between them.

These problems stem from a lack of control. The key is to have a control system that answers the following questions:

- What is my current software configuration?
- What is its status?
- How do I control changes to my configuration?
- How do I inform everyone else of my changes?
- What changes have been made to my software?
- Do anyone else's changes affect my software?

SOFTWARE PRODUCT NOMENCLATURE

- System - The package of all the software that meet's the user's requirements.
- Subsystem - Comprise large systems, such as communications, display, and processing;
- Product - Components of subsystems, such as control program, compliers, and utilities of an operating system.
- Component - Components of a product, such as supervisor and scheduler of a control program.
- Module - Lowest level of components is modules. Typically implement individual functions that are relatively small and self-contained, such as queue management and interrupt dispatcher.

During implementation, two things are happening. First, the modules are being developed, enhanced, tested, and repaired from detailed design and implementation through system test. Second, the modules are being assembled into components, products, subsystems, and systems. During this building process, the modules are consistently being changed to add functions or repair problems. This process is supported by a hierarchy of tests:

- Unit test - A separate test for each individual module;
- Integration test - As the modules are integrated into components, products, subsystems, and systems, their interfaces and interdependencies are tested to insure they are properly designed and implemented.
- Function test - When integration results in a functionally operable build, it is tested in a component test, product test, subsystem test, and finally in a system test;

- Regression test - At each integration test (here called a spin), a new product is produced. This is the first tested to insure that it hasn't regressed, or lost functions present in a previous build.

Once an initial product is stabilized, a first baseline is established. Each baseline is a permanent database, together with all the changes that produced it. Only tested code and approved changes are in the baseline, which is fully protected.

The key SCM tasks are:

- Configuration control
- Change management
- Revisions
- Deltas
- Conditional code.

CONFIGURATION CONTROL

The task of configuration control revolves around one official copy of the code. The simplest way to protect every system revision is to keep a separate official copy of each version.

However, when two or more groups work on separate copies of the same or similar versions of common code, they often make different changes to correct the same problem. A good rule of thumb is that no two separate copies of a program can be kept identical. If separate copies exist, they must be assumed to differ; even if they were the same, they will soon diverge.

Only keep one official copy of any code that is used by several groups. Working copies may occasionally be used, but a common library must be the official source for all this common code, and only officially approved changes can be permitted into this library.

REVISIONS

Keep track of every change to every module and test case. There is one latest official version and every prior version is identified and retained; these obsolete copies can be used to trace problems.

A numbering system must separately identify each test, module, component, product, and system.

VERSIONS

Often, several different functions can be implemented by the same module with only modest coding differences. For example, different memory management code may be needed to handle expansion beyond the standard 512K. Then, a different use a standard management module below 512K, and one using a mode switch beyond 512K.

Since these are different programs, they have different designations, such as MEM and MEML. Each would have its own sequence of revisions and revision numbering schemes.

DELTAS

Versions solves the problem of different functional needs for the same module but introduces multiple copies of the same code. The reason is that most of the code in the two modules would be identical. One, however, may have an additional routine to handle memory limits testing and mode switching (in the case of the 512K memory limits problem). Since they are stored as separate modules, however, there is no way to make sure all changes made to one are incorporated into the other.

One way to handle this is with deltas. This involves storing the base module (MEM) with those changes required to make it into MEML When maintenance is required on MEM, these changes can be made directly, so long as they do not interfere with the delta code. Changes to MEML are made to the delta, with MEM left alone.

There are disadvantages to this system. It is possible to have versions of both versions, so tracking get extremely compli-cated. If an element is lost or corrupted in a chain of deltas, it may be difficult to resurrect the entire chain. If deltas live a long time, they could grow into large blocks of code.

An answer is to use deltas only for temporary variations; then incorporate them into the baseline. However, the deltas have to be incorporated separately.

CONDITIONAL CODE

Another way of handling slight variations between modules is to use some form of conditional program construction. For example, a billing program might use different functions depending on the need for a state sales tax. The source program would contain various tax versions, but none would be included in the final system unless called for at system installation.

The use of conditional code simplifies code control because there is only one official copy of each module. The number of version combinations is also minimized.

There are disadvantages, however. The most important is that end users must specify all parameters and then perform a special and perhaps complex installation process. Also, thew system generation process becomes more complex with system growth.

BASELINES

The baseline is the foundation for SCM. It provides the official standard on which subsequent work is based and to which only authorized changes are made. After an initial baseline is established and frozen, every subsequent change is recorded as a delta until the next baseline is set.

While a baseline should be established early in a project, establishing one too early will impose unnecessary procedures and slow the programmers' work. As long as programmers can work on individual modules with little interaction, a code baseline is not needed. As soon as integration begins, formal control is needed.

BASELINE SCOPE.

Items to be included in the implementation phase are:

- The current level of each module, including source and object code
- he current level of each test case, including source and object code
- The current level of each assemble, compiler, editor, or other tool used
- The current level of any special test or operational data
- The current level of all macros, libraries, and files
- The current level of any installation or operating procedures
- The current level of operating systems and hardware, if pertinent

Retain every change, no matter how minor it seems.

BASELINE CONTROL

Controlled flexibility is accomplished by providing the programmers with private working copies of any part of the baseline. They can try new changes, conduct tests, etc. without disturbing anyone else. When ready, new changes can be incorporated into the baseline, after assuring the changes are compatible and no new code causes regressions.

Every proposed change must be tested against a trial version of the new baseline to make sure it does not invalidate any other changes.

CONFIGURATION MANAGEMENT RECORDS

Every change proposal is documented and authorized before being made. The documentation includes the reason for the change, the potential cost in time, the person responsible for the change, and the products affected.

Detailed records are especially crucial when hardware and software changes are made simultaneously. Just because the programs run doesn't mean they will continue to run if hardware changes.

Should always have *problem reports*, which document every problem and the precise conditions that caused it.

The *expect list* details every function and planned feature for every component in each new baseline.

CONFIGURATION MANAGEMENT RESPONSIBILITIES

The configuration manager is the central control point for system changes and has the following responsibilities:

- Develop, document, and distribute the SCM procedures;
- Establish the system baseline, including backup provisions;
- Insure that no unauthorized changes are made to the baseline;
- Insure that all baseline changes are recorded in sufficient detail so the can be reproduced or backed out;
- Insure that all baseline changes are regression tested;
- Provide the focal point for exception resolution.

MODULE OWNERSHIP

To insure integrity of modules, each module should have an owner. Of course, each person usually owns more than one module at a time.

The module owner's responsibilities are:

- Know and understand the module design;
- Provide advice to anyone who works on or interfaces with the module;
- Serve as a technical control point for all module modifications, including both enhancement and repair;
- Insure module integrity by reviewing all changes and conducting periodic regression tests

Module ownership insures design continuity by providing a single focus for all module changes. Its disadvantages are that it depends on the skill and availability of individuals and it only provides design control at the detailed level. These disadvantages can be countered by using a back-up “buddy” system between module owners and by maintaining an overall design responsibility to monitor and control the software structure, interfaces, macros, and conventions.

THE CHANGE CONTROL BOARD

On moderate through large projects, a change control board (CCB) (sometimes called the Configuration Control Board) is needed to insure every change is properly considered and coordinated.

The CCB should include some members from development, documentation, test, assurance, maintenance, and release. The CCB reviews each request for change and approves it, disapproves it, or requests more information.

Depending on project size, several CCBs may be needed, each with expertise authority over a specific area. Some examples are: overall design and module interfaces, the control program, the applications component, user interfaces, and development tools. With multiple CCBs, a system-level CCS is needed to resolve disputes between these lower-level boards.

A CCB typically needs the following information on each proposed change:

- Size - How many new/changed LOC?
- Alternatives - How else can it be done?
- Complexity - Is the change within a single component or does it involve others?
- Schedule - When?
- Impact - What are future consequences?
- Cost - What are potential costs and savings?
- Relationship with other changes - Will another change supersede or invalidate this one, or does it depend on other changes?
- Test - Are there special test requirements?
- Resources - Are the needed people available to do the work?
- System impact - What are the memory, performance

- Benefits - What are the expected benefits?
- Politics - Are there special considerations such as who is requesting the change or whom it will affect?
- Change maturity - How long has the change been under consideration?

If the change is to fix a customer-related problem, other information may be required:

- A definition of the problem the change is intended to fix;
- The conditions under which the problem was observed;
- A copy of the trouble report;
- A technical description of the problem;
- The names of other programs affected.

CCB PROBLEMS

Do not waive a review just because change activity has become a bottleneck. It is precisely at the time of heaviest change and testing activity that loss of control is most likely and CCB review is most needed.

Select CCB members with care. They have the power to block any part of the project so these assignments should not be treated lightly. The project's software development manager should personally chair the highest-level CCB.

THE NEED FOR AUTOMATED TOOLS

Without automated tools, changes can easily be lost or done without proper procedures.

Chapter 8: Software Quality Assurance

“What is not tracked is not done”

In software, so many things need to be done management cannot track all of them. So, some organization needs to do the tracking. That is the role of software quality assurance (SQA).

SQA is designed to insure that officially established processes are being implemented and followed. Specifically, SQO insures that:

- An appropriate development methodology is in place;
- The projects use standards and procedures in their work;
- Independent reviews and audits are conducted;
- Documentation is produced to support maintenance and enhancement;
- Documentation is produced during development and not after development;
- Mechanisms are in place and used to control changes;
- Testing emphasizes all the high-risk product areas;
- Each software task is satisfactorily completed before the next one begins;
- Deviations from standards and procedures are exposed ASAP;

- The project is audible by external professionals;
- The quality control work is itself performed against established standards;
- The SQA plan and the software development plan are compatible.

The goals of SQA are:

- To improve software quality by approximately monitoring both the software and the development process that produces it;
- To insure full compliance with the established standards and procedures for the software and the software process;
- To insure that inadequacies in the product, the process, or the standards are brought to managements' attention so these inadequacies can be fixed.

THE ROLE OF SQA

The people responsible for the software projects are the only ones who can be responsible for quality. The role of SQA is to monitor the way these groups perform their responsibilities. In doing this, there are several pitfalls:

- It is a mistake to assume that SQA staff can do anything about quality;
- The existence of SQA does not insure that standards and procedures will be followed;
- Unless management demonstrates its support for SQA by following their recommendations, SQA will be ineffective;
- Unless line management requires that SQA try to resolve their issues with project management before escalation, SQA and development will not work together effectively.

All SQA can do is alert management to deviations from established standards and practices. Management must then insist that the quality problems be fixed before the software is shipped; otherwise, SQA becomes an expensive bureaucratic exercise.

SQA RESPONSIBILITIES

SQA can be effective when it reports through an independent management chain, when it is properly staffed, and when it sees its role as supporting the development and maintenance personnel in improving product quality. Then, SQA should be given the following responsibilities:

- Review all development and quality plans for completeness;
- Participate as inspection moderators in design and code inspections;
- Review all test plans for adherence to standards;
- Review a significant sample of all test results to determine adherence to plans;
- Periodically audit SCM performance to determine adherence to standards;

- Participate in all project quarterly and phase reviews and register non-concurrence if appropriate standards and procedures have not be reasonably met.

SQA FUNCTIONS

Before establishing an SQA function, the basic organizational framework should include the following:

- Quality assurance practices - Adequate development tools, techniques, methods, and standards are defined and available for Quality Assurance review;
- Software project planning evaluation - If not defined at the outset, they will not be implemented;
- Requirements evaluation - Initial requirements must be reviewed for conformance to quality standards;
- Evaluation of the design process-
- Evaluation of coding practices -

- Evaluation of the software integration and test process -
- In-process evaluation of the management and project control process -

SQA REPORTING

SQA reporting should not be under the software development manager. SQA should report to a high enough management level to have some chance of influencing priorities and obtaining enough resources to fix key problems. However, lower-level reporting normally results in better working relationships with developers, while the ability to influence priorities is reduced.

Some general guidelines are:

- SQA should not report to the project manager;
- SQA should report somewhere within the local company or plant organization;
- There should be no more than one management position between SQA and the senior location manager;
- SQA should always have a “dotted-line” relationship to a senior corporate executive;
- SQA should report to someone having a vested interest in software quality, like the staff head responsible for field services.

SQA CONSIDERATIONS

- SQA organizations are rarely staffed with sufficiently experienced or knowledgable people because such people usually prefer development/design work, and management often wants them in the latter, too;
- The SQA management team often is not capable of negotiating with development. This depends on the caliber of the SQA team;
- Senior management often backs development over SQA on a very large percentage of issues. Development then ignores he SQA issues, and SQA degenerates into a series of low-level, useless debates.

- Many SQA organizations operate without suitably documented and approved development standards and procedures; without such standards, they do not have a sound basis for judging developmental work, and every issue becomes a matter of opinion. Development also wins such generalized debates when schedules are tight.
- Software development groups rarely produce verifiable quality plans. SQA is then trapped into arguments over specific defects rather than overall quality indicators. SQA may win the battle but lose the war.

SQA PEOPLE

Getting good people into SQA can be a problem. Possible solutions include putting new hires there (but must also have experienced people there, too), rotating personnel through SQA (which may result in only poor developers being assigned there), and requiring that all new development managers be promoted from SQA after spending at least 6 months there (which can be very effective).

INDEPENDENT VERIFICATION AND VALIDATION

In DoD contracts, independent verification and validation (IV&V) is often specified. The IV&V organization provides an independent assessment of the quality of the software. However, do not confuse SQA and IV&V. SQA works for the developer; IV&V works for the customer.

Level 3: Major Topics from Humphrey

Chapter 9 - SOFTWARE STANDARDS
Chapter 10 - SOFTWARE INSPECTIONS
Chapter 11 - SOFTWARE TESTING
Chapter 12 - SOFTWARE CONFIGURATION MANAGEMENT (continued)
Chapter 13 - DEFINING THE SOFTWARE PROCESS
Chapter 14 - THE SOFTWARE ENGINEERING PROCESS GROUP

1. Software standards - Standards for languages, coding conventions, commenting, error reporting, software requirements, software reviews and audits, SQA plans, test plans, SCM, documentation, and so forth.

2. Software inspection - Formal review process for defect prevention based on humans.

3. Software testing - Computer-based defect prevention.

4. SCM -

5. Software Processes - Spiral model, waterfall model, process architecture, requirements gathering, and so forth

Chapter 14: The Software Engineering Process Group (SEPG)

If a person's only tool is a hammer, the whole world looks like a nail - Anon

Software development requires people who do the actual development and people who refine the development process. While every software development organization has full-time developers, having full-time process groups is relatively new--but essential in large projects.

Software processes improvement requires the steps:

- Identify the key problems
- Establish priorities
- Define action plans
- Get professional and management agreement
- Assign people
- Provide guidance and training
- Launch implementation
- Track progress
- Fix the inevitable problems

The SEPG has two basic tasks that are done simultaneously:

- Initiating and sustaining process change, and
- Supporting normal operations.

The SEPG may spawn other groups for technology support, such as education, cost estimation, standards, and SAQ.

Change is an essential part of the software process, but it cannot be excessive or it will disrupt projects. For example, changing languages every three months certainly will lead to failure. The software process should be viewed as continuous learning. Future products will be larger and more complex than today's products.

Change is needed to bring the current level of practice up to the current level of knowledge, We typically know how to do our work far better than we are doing it. Too often, we get lost in details of relatively minor problems (when taken individually) and are unable to apply our best knowledge.

THE SEPG AS CHANGE AGENT

A change agent provides the energy, enthusiasm, and direction to overcome resistance and cause change. While management approves action plans, the SEPG takes the lead in launching the required efforts, providing leadership in getting them staffed, and supporting the work with needed information, training, and skills.

When change is essential, it is important to control the pace of change. There must be reasonable stability, but change is necessary to get better.

THE SEPG SUSTAINING ROLES

The continuing role of the SEPG can be divided into six categories:

- Establish process standards
- Maintain the process database
- Serve as the focal point for technology insertion
- Provide key process education
- Provide project consultation
- Make periodic assessments and status reports

ESTABLISHING STANDARDS

- The SEPG recommends the priorities for establishing standards. There are many potential standards, so they must be introduced in an orderly way. New standards should not be started until the developed standards have been approved and adopted.
- The SEPG should insure that prior work on the subject is reviewed--don't reinvent the wheel.
- The standards development team should include available experts and users. Find out what is theoretically pleasing versus what will work.
- The final standard review and approval is the most important step. A comprehensive review by prospective users will expose conflicting opinions and facilitate thoughtful technical and managerial debates needed to produce a competent result.

THE PROCESS DATABASE

The process database is the repository for the data gathered on the software engineering process and the resulting products. These data provide the basis for improving estimating accuracy, analyzing productivity, and assessing product quality.

The types of information retained in this database are:

- Size, cost, and schedule data
- Product metrics
- Process metrics

Considerations for an effective database are:

The reason for gathering each piece of data must be stated. Don't gather it just because it is there.

- Define the exact meaning of each field of data before you gather it, together with all anticipated options and exceptions.
- Define simple, easy to use tools for gathering the data.
- Make sure the data are gathered in a timely way so they will be useful. Omitted relevant data are just as wasteful irrelevant data.
- Make sure the data are accurate.
- Define procedures for entering the data into the database.
- Provide user access to the data
- Protect and maintain the data.

TECHNOLOGY INSERTION FOCAL POINT

Technology support for any reasonably large software engineering organization involves seven activities:

- A process definition is needed to identify the software tasks that are both widely used enough and sufficiently common to warrant automated support;
- A set of requirements is then developed for the needed support;
- These requirements must be approved by the users, the people who will handle installation and support, and management;
- A search is made to identify commercially available tools that meet these needs;

- An overall technology plan is developed that defines long-term technology needs and establishes a strategy for addressing them. This includes an architectural plan for the support environment and a phased introduction plan;
- A technology support group is established to handle tool and environment installation, provide user assistance, and be the interface to the tool suppliers on problem and interfaces;
- Education, training, and consulting must be provided before the tools are installed for general use.

EDUCATION AND TRAINING

The SEPG must serve as the focal point for process education, but must avoid becoming saddled with all of the teaching responsibilities. Ideally, a full-time education group maintains a staff of volunteers, instructors, or consultants to do the job.

Examples of needed courses are:

- Project management methods;
- Software design methods;
- Quality management;
- Design and code inspections.

ESTABLISHING THE SEPG

The key questions are:

- How big should the effort be?
- Where does the staff come from?
- Who should head it?
- Where should it report?
- What tasks are its initial focus?
- What are the key risks?
- How is effectiveness evaluated?

SEPG SIZEAND STAFFING

The SEPG should have about 2% of the software professionals for a large (100 or more) software development staff. For smaller organizations, there should be at least one full-time process guru with part-time support from other working groups.

Staff the SEPG with the best, most experienced professionals you can find. If the SEPG has management backing, the better people will seek these assignments.

The SEPG leader must be a knowledgeable manager with a demonstrated ability to make things happen, the respect of the project professionals, and the support of top management. The key criteria are:

- Agents must be enthusiastic about leading the change process;
- Agents must be both technically and politically capable of understanding the problems and insuring that effective solutions are implemented;
- Agents need the respect of the people they deal with;
- Agents must have management's confidence and support or they will not act with the assurance needed to get wide cooperation and acceptance.

SEPG PRIORITIES AND RISKS

Risk situations for the SEPG:

- It doesn't have enough full-time people to do the work;
- It doesn't have management support to spawn improvement efforts;
- Its manager is not able to obtain the participation of the most competent software professionals.

EVALUATING THE SEPG

Can use the following criteria:

- Level 2 - Does the SEPG have a plan for its work, a tracking system, and means to retain and control its work products?
- Level 3 - Have the SEPG professionals established a basic framework for their own work, including standards, procedures, and a review program;
- Level 4 - Does the SEPG measure the productivity and quality of its own work? This, for example, might include the workload factors for training, consultation, process development, and administration.
- Level 5 - Does the SEPG regularly assess its own activities for improvement opportunities and incorporate them into its working process?

The following section is from "Our Worst Current Development Practices," Caper Jones, *Software*, March 1996, 102-104.

"There are two kinds of failures: those who thought and never did and those who did and never thought." In the software business, the worst failures fall into the second category. Too often we plunge into a major project, without considering the factors that have the strongest bearings on success or failure. Even worse, many software managers refuse to learn from failure, repeating the same destructive behavior project after project, then wondering why Mylanta has become one of their major food groups. -- Roger Pressman

Following are the 10 worst current practices--those factors that often lead to failure and disaster. A software project is a failure if it was:

- Terminated because of cost or schedule overruns;
- Experienced schedule or cost overruns in excess of 50 percent of initial projections; or
- Resulted in client lawsuits for contractual noncompliance.

PRACTICE 1: No historical software-measurement data.

PRACTICE 2: Rejection of accurate estimates - Our industry lacks a solid empirical foundation of measured results. Thus, almost every major software project is subject to arbitrary and sometimes irrational schedule and cost constraints.

Therefore, lack accurate measurement data is the rot cause

for practices three through ten and contributes to a host of other problems such as:

- Inability to perform return-on-investment calculations;
- Susceptibility to false claims by tool and method vendors; and
- Software contracts that are ambiguous and difficult to monitor.

PRACTICES 3 AND 4: Failure to use automated estimating tools and automated planing tools - There are about 50 commercial software-cost estimating tools and more than 100 project-planning tools on the market. Software projects that use such tools currently have a much greater probability of success than those that attempt to estimate and plan by manual means.

PRACTICES 5 AND 6: Excessive, irrational schedule pres-

sure and creep in user requirements. These practices also derive from a lack of solid empirical data. Once management or the client imposes an arbitrary and irrational schedule, they insist on adhering to it, which results in shortcuts to design, specification, and quality control that damage the project beyond redemption

At the same time, the original requirements for the project tend to grow continuously throughout the development cycle. The combination of continuous schedule pressure and continuous growth in unanticipated requirements results in a very hazardous pairing.
Software requirements change at an average rate of about 1% a month. Thus for a project with a 36-month development cycle, a third of the features and functions may be added as afterthoughts.

PRACTICES 7 AND 8: Failure to monitor progress and perform formal risk management. Even a rudimentary checklist of software-risk control factors would be helpful:

- What measurement data from similar projects has been analyzed?
- What measurement data on this project will be collected?
- Have formal estimates and plans been prepared?
- How will creeping user requirements be handled?
- What milestones will be used to indicate satisfactory progress?
- What series of reviews, inspections, and tests will be used?

PRACTICES 9 AND 10: Failure to use design and code inspections. All best-in-class software procedures use software inspections. The measured defect-removal efficiency of inspections is about twice that of most forms of software testing: about 60 percent for inspections versus 30 percent for most kinds of testing.

Chapter 15: Data Gathering and Analysis

The principles of data gathering:

- The data are gathered in accordance with specific objectives and plans;
- The choice of data to be gathered is based on a model or hypothesis about the process being examined;
- The data gathering process must consider the impact of data gathering on the entire organization;
- The data gathering plan must have management support.

The objectives of data gathering are:

- Understanding
- Evaluation
- Control
- Prediction

BUT, BE CAREFUL: George Miller, a famous psychologist once said: “In truth, a good case can be made that if your knowledge is meager and unsatisfactory, the last thing in the world you should do is make measurements. The chance is negligible that you will measure the right things accidentally.”

[NOTE: This is probably in response to the famous quotation from Lord Kelvin: “When you can measure what you are speaking about, and express it in numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind; it may be the beginning of knowledge, but you have scarcely advanced to the stage of science.”]

Mark Twain - “Collecting data is like collecting garbage. You need to know what you’re going to do with it before you collect it.” (paraphrase)

Given enough random numbers and enough time, one can draw inferences from data and events most likely independent, such as the stock market goes up in years when the National League team wins the World Series.

THE IMPACT OF DATA GATHERING ON AN ORGANIZATION

Must consider the effects of measurements on the people and the effects of people on the measurements. When people know they are being measured, their performance will change (the Hawthorn Effect); they will give top priority to improving the measure. In the software business, employees must know the numbers will not be used against them; otherwise, they will make the numbers look good regardless of reality.

Collecting data is tedious and must generally be done by software professionals who are already very busy. Unless they and their immediate managers are convinced that the data are important, they either won't do it or will not be very careful when they do. If professionals are shown how the data will help them, they will be interested in the results and exercise greater care.

For reasons of objectivity and cost, it is important to automate as much as possible. Special data gathering instrumentation is expensive, so it is wise to make data collection a part of the existing software engineering processes whenever possible. If portions of the SCM function are automated, they can provide key data with relatively little extra effort.

Management support id‘s critical. Data gathering must be viewed as an investment. Once useful results are evident, project managers willing eager to support the data collection.

A study at the NASA Goddard Space Flight Center's Software Engineering Laboratory shows that data collection can take 15% of a development budget. The main reason is the work in manual and done by inexperienced people. With improved tools, these costs can go down. Even so, it is crucial that data be defined precisely to insure the right information is obtained and validated and to insure it accurately represents the process.

TYPICAL PROBLEMS AND CAUSES IN DATA GATHERING

Problem	**Typical Cause**
Data are not correct	Raw data entered incorrectly. Data were generated carelessly.
Data not timely	Data not generated rapidly enough.
Data not measured or indexed properly	Raw data not gathered consistently with purpose of analysis.
Too much data needed	Too many coefficients in model.
Needed data do not exist	No one retained data. Data don't exist.

DATA GATHERING PLAN

The plan should be developed by the SEPG with the help of those who will gather the data. It should cover the following topics:

- What data is needed by whom and for what purpose? Make it clear it will not be used for personnel evaluation.
- What are the data specifications? All definitions must be clear.
- Who will support data gathering?
- How will the data be gathered? Appropriate forms and data entry facilities must be available.
- How will the data be validated? Software process data are error-prone, so it should be validated as quickly as possible.
- How will the data be managed? Need a reliable staff with suitable facilities

Basili reported that software process data may be as much as 50% erroneous, even when recorded by the programmers at the time they first found the errors. BE CAREFUL!

SOFTWARE MEASURES

Data Characteristics:

- The measures should be robust - They should be repeatable, precise, and relatively insensitive to minor changes in tools, methods, and product characteristics. Otherwise, variations could be caused by measurement anomalies instead of by software processes.
- The measures should suggest a norm - Defect measures, for example, should have a lower value when best, with zero being desirable.
- The measures should suggest an improvement strategy - Complexity measures, for example, should imply a reduction of complexity.
- They should be a natural result of the process
- The measures should be simple

- They should be both predictable and traceable - Measures are of most value when they are projected ahead of time and then compared with the actual experience. Project personnel can then see better how to change their behavior to improve the result.

Software measures may be classified as:

- Objective/Subjective - Distinguishes ones that count things and those involving human judgement;
- Absolute/Relative - Absolute measures are invariant to the addition of new terms (such as the size of a program). Relative measures change (such as the mean of test scores). Objective measures tend to be absolute, while subjective measures tend to be relative.
- Explicit/Derived - Explicit measures are often taken directly, while derived measures are computed from explicit measures or from other derived measures. Programmer months expended on a project is an explicit measure, while productivity per month in LOC is a derived measure.

- Dynamic/Static - Dynamic measures have a time dimension, as with errors found per month. Static measures remain invariant, as with total defects found during development.
- Predictive/Explanatory - Predictive measures can be obtained in advance, while explanatory measures are produced after the fact.

The following is from an article by S. Pfleeger

Lessons Learned in Building a Corporate Metrics Program

Shari Pfleeger

IEEE Software, May 1993

1. Begin with the process - Developers must understand the need for the metrics. Otherwise, they may not provide accurate data or use the results of the analysis.

2. Keep the metrics close to the developers - Do not form a separate metrics group.

3. Start with people who need help; let then do the advertising for you.

4. Automate as much as possible.

5. Keep things simple and easy to understand

6. Capture whatever you can without burdening developers

7. If the developers don't want to, don't make them.

8. Using some metrics is better than using no metrics.

9. Use different metrics when needed

10. Criticize the process and the product, not the people

SOFTWARE SIZE MEASURES

The assumption is that effort required is directly related to program size. Unfortunately, there is no universal measure of program size because program size is not a simple subject. Do we count new, deleted, reused, etc. lines? What about higher-level languages versus assembly language? What about comment statements?

LOC possibilities:

- Some alternative ways of counting LOC are:
- Executable lines
- Executable lines plus data definitions
- Executable lines, data definitions, and comments
- Executable lines, data definitions, comments, and JCL
- Physical lines on a screen
- Logical delimiters, such as semicolons
- Only new lines
- New and changed lines
- New, changed, and reused lines
- All delivered lines plus temporary scaffold code
- All delivered lines, temporary scaffolding, and support code

ERROR DATA

- Error - Human mistakes. Could be typographical, syntactic, semantic, etc.
- Defect - Improper program conditions as the result of an error. Not all errors produce program defects, and not all defects are caused by programmers (bad packaging or handling, for example).
- Bug (fault) - Program defect encountered in operation, either under test or in use. Bugs result from defects, but all defects do not cause bugs (some are never found).
- Failures - A malfunction of a user's installation. Could result from a bug, incorrect installation, a hardware failure, etc.
- Problems - User-encountered difficulties. may result from failures, misuse, or misunderstanding. Problems are human events; failures are system events.

CLASSES OF DEFECT MEASURE

Defects can be classified along dimensions of:

- Severity
- Symptoms
- Where found
- When found (unit test, system test, etc.)
- How found (inspection, testing, etc.)
- Where caused (what part of software)
- When caused (design, etc.)
- How caused (logical, data definition, etc.)
- Where fixed
- When fixed
- How fixed

Statistical breakdowns of errors are given below (Tables 15.6-15.8).

ANALYSIS OF INSPECTION RATES

Preparation time is the sum of preparation times for all individuals involved in an inspection.

Inspection time is the time spent by the entire team in the inspection process. For a five-person inspection group, one inspection hour equals five programmer inspection hours, while one preparation hour equals one programmer preparation hour. In the following data, it appears the author used an average of the individual preparation times rather than a sum.

(see Figure 15.2)

Errors found per KLOC decline with increasing inspection rate. An upper limit of 300 to 400 LOC of Fortran might be reasonable.

Chapter 16: Managing Software Quality

Motivation is the key to good work. management must challenge developers. If senior management tolerates poor work, sloppiness will pervade the organization. Meetings must start on time, reports must be accurate, etc.

No risk, no heroes/heroines.

Four basic quality principles:

- Without aggressive quality goals, nothing will change;
- If these goals are not numerical, then quality will remain talk;
- Without quality plans, only you are committed to quality;
- Quality plans are just paper unless you track and review them.

Chapter 17: Defect Prevention

Must prevent defects because:

- To meet the escalating needs of society, progressively larger and more complex software will be needed.
- Since these programs will be used in increasingly sensitive applications, software defects will likely become progressively more damaging to society.
- For the foreseeable future, programs will continue to be designed by error-prone humans.
- This means that, with present methods, the number and severity of bugs encountered by systems users will increase.

The principles of defect prevention:

- Programmers must evaluate their own errors.
- Feedback is an essential part of defect prevention. There is no single cure-all that will solve all problems.
- Process improvement must be an integral part of the process.
- Process improvement tales time to learn.

The steps of software defect prevention:

- Defect reporting
- Cause analysis
- Action plan development
- Performance tracking
- Staring over - do it all again for the most prevalent remaining defects.

Error cause categories:

- Technological - Definability of the problem, feasibility of solving it, availability of tools and procedures.
- Organizational - Division of workload, available information, communication, resources.
- Historic - History of project, of the program, special situations, and external influences.
- Group dynamic - Willingness to cooperate, distribution of roles inside the project group.
- Individual - Experience, talent, and constitution of the individual programmer.