Seven Basic Principles of Software Engineering

Barry W. Boehm
TRW Defense Systems Group

This paper attempts to distill the large number of individual aphorisms on good software engineering into a small set of basic principles. Seven principles have been determined which form a reasonably independent and complete set. These are: (1) manage using a phased life-cycle plan; (2) perform continuous validation; (3) maintain disciplined product control; (4) use modern programming practices; (5) maintain clear accountability for results: (6) use better and fewer people; and (7) maintain a commitment to improve the process. The overall rationale behind this set of principles is discussed, followed by a more detailed discussion of each of the principles.

INTRODUCTION

What does it take to ensure a successful software development project? If you follow one or two basic principles (e.g., "Use top-down structured programming," "Use a few good guys"), will that be enough to guarantee a responsive, reliable product developed within schedule and budget? Or do you need dozens of checklists with dozens of items in each?

This paper reports on some recent attempts to condense roughly 30,000,000 man-hours of software development experience at TRW into a small number of basic principles which provide the keys to a successful software effort. Through this experience, we have found that one or two such principles are insufficient to guarantee such a successful outcome. It now appears that at least seven basic principles are involved. These are:

1. Manage using a phased life-cycle plan.
2. Perform continuous validation.
3. Maintain disciplined product control.
4. Use modern programming practices.
5. Maintain clear accountability for results.
6. Use better and fewer people.
7. Maintain a commitment to improve the process.

This is one of a series of efforts at TRW to define such a set of principles, beginning with a set of five principles formulated by Royce in 1970 [I 1, and refined into different sets of principles by Mangold in subsequent efforts [2].

Journal of Systems and Software 3.3-24 (1983)
© Elsevier Science Publishing Co., Inc. 1983

CRITERIA FOR A SET OF BASIC PRINCIPLES

Why should one consider the above seven principles as a basic set? The criteria for a set of basic principles should be similar to those for a set of basis vectors for a vector space:

1. They should be independent.
2. The entire space (of useful software principles) should be representable (or implied) by combinations of the basic principles.

The independence of the principles follows from the fact that no combination of six of the principles implies the seventh. Or, to put it another way, software projects can be run using any six of the basic principles, but violating the remaining one.

The completeness of the principles can't be demonstrated absolutely. The best that we've been able to do has been to take over a 100 useful software principles and show that they can be implied, fairly reproducibly, from combinations of the seven basic principles.

A short example is given in Table 1. Here, we took Ward's "Twenty Commandments for Tactical Software Acquisition" [3] and gave them to three different people to determine independently the extent to which they were implied by the above seven basic principles.

Table 1. Seven Basic Principles vs Ward's "20 commandments"

	Commandments	1-manage to plan	2-continuous validation	3-product control	4-MPP	5-accountability	6-people	7-improve process	Other
1.	One prime contractor					***			
2.	No multiprocessing								***
3.	Indep. developed subsystems				·	*·			*·
4.	Computer Program Integration Documentation			**			*		
5.	Common exec for all computers						·		**
6.	Unified HW-SW management					***			
7.	Computer program development plan	***							
8.	Continuous integration	*·	·		**				
9.	Test plans	**·	*·			··			
10.	Customer has computer experts		*·				*·		
11.	Close customer-contractor relation		**			··	*		
12.	Technical office has computer experts		*				·		
13.	Thorough design reviews	·	**·						
14.	Customer operates computers		***						
15.	Use simulators		***						
16.	Realistic test scenarios	·	**·						
17.	Rigid configuration control			***					
18.	HW under configuration control			***					
19.	SW gets topside attention					*		·	·

Note: * indicates principle implies commandment, according to one respondent. · indicates principle is correlated with commandment, according to one respondent.

In Table 1, each asterisk indicates that one of the respondents considered that the corresponding Ward commandment was implied by the principle in the indicated column. Thus, all three individuals considered the first commandment, “One prime contractor,” to be implied by Principle 5, “Maintain clear accountability for results.” All three considered the second commandment, “No multiprocessing,” as impossible to deduce from the seven principles-but they also considered that this was not an acceptable general commandment from their experience, since some projects have successfully used multiprocessor architectures. The dots in Table 1 indicate that a respondent considered that there was a correlation between the commandment and the principle, but less than an implication.

Table 1 shows a strong degree of unanimity among the three respondents, indicating that people can use the principles fairly reproducibly to imply other general advice on good software practice. [Another, more amusing, fact that we discovered in generating Table 1 was that the Twenty Commandments, which were numbered (a) through (s) in Ward’s paper, actually numbered only 19!]

Additional corroboration of the completeness of the Seven Basic Principles is given in Table 2, which shows correlations and implications between the Seven Basic Principles and another selection from our compilation of rules for good software practice. Again, the principles fairly reproducibly imply the rules of good practice, with some exceptions in the area of tactical rules for good coding practice, such as “Initialize Variables Prior to Use,” Thus, the principles don’t give all the answers for all tactical software questions, but they provide good, reasonably complete guidance on the strategic questions, where there is more leverage for big savings in project performance.

Table 2. Seven Basic Principles vs Other Software Principles

Other software principles	l-manage to plan	2-continuous validation	3-product control	4-MPP	5-accountability	6-people	7-improve process	Other
Do a complete preliminary design	*							
Involve the customer and user		*						
Current, complete documentation			*					
Discipline test planning	*	*						
Testable requirements		*						
Prepare for operational errors		*						
Detailed resource control & data collection	*				*		*	
Use automated aids						*		
Requirements traceability			*					
Use measurable milestones					*			
Use structured code				*				
Incremental top-down development				*				
Use a chief programmer					*	*		
Use a program library			*					
Use walk-throughs		*						
Use & analyze problem reports					*		*	
Avoid “human-wave” approach						*		
Enforceable standards					*			
Unit Development Folder					*			
Early data base design				*				
Deliverables plan	*							
Transition and turnover plan	*							
Independent test team		*			*			
Project work authorizations					*			
Initialize variables prior to use								*
Early baselining of requirements			*					
Thorough requirements & design reviews		*						

Note: * indicates principle implies commandment, according to one respondent.
brooke 3-3-83

More detail on each of the principles is now given below, followed by a summary indicating the results of using more and more of the principles over five generations of developing a particular line of large command and control software products.

PRINCIPLE 1: MANAGE USING PHASED LIFECYCLE PLAN

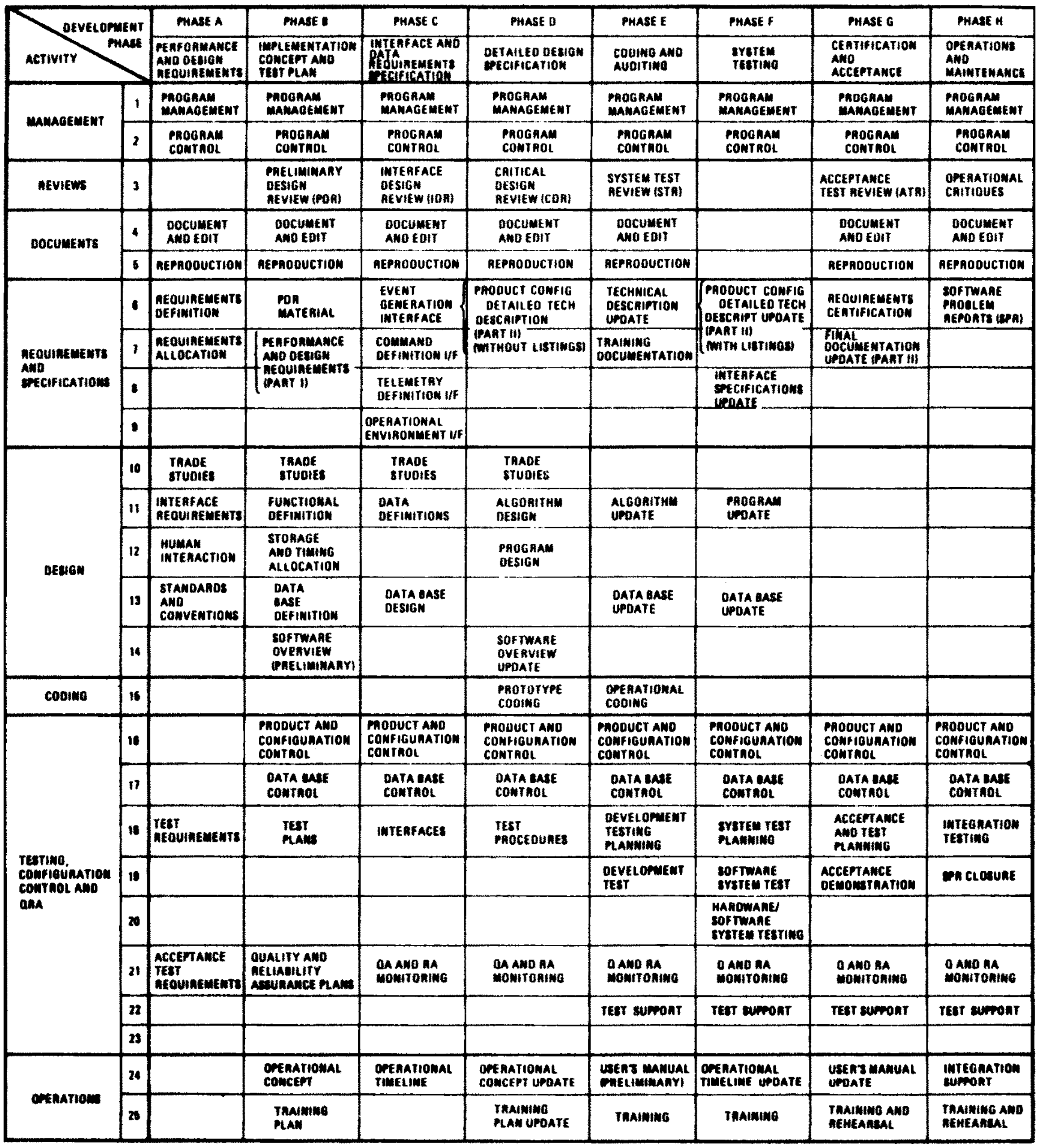
Create and Maintain a Life-Cycle Project Plan

Importance of the project plan. How important is it to have a project plan? Metzger [4] goes as far as to say that of all the unsuccessful projects of which he had knowledge, roughly 50% failed because of poor planning. His list of reasons for the failure of software projects looks like this:

- Poor planning
- Ill-defined contract
- Poor planning
- Unstable problem definition
- Poor planning
- Inexperienced management
- Poor planning
- Political pressures
- Poor planning

Essentials of a software project plan. The essentials of a software project plan are given below, based partly on those given by Metzger [4] and Abramson [5].

1. *Project overview.* A summary of the project that can be read by anyone (say a high-level manager) in a few minutes and that will give him the essentials of the project.
2. *Phased milestone plan.* A description of the products to be developed in each phase, and of the associated development activities and schedules. Major products of each phase should be identified as discrete milestones, in such a way that there can be no ambiguity about whether or not a milestone has been achieved. The component activities in each phase should be identified in some detail. A good example of a checklist of component activities is that given by Wolverton [6] and shown here as Figure 1.



ACTIVITY \ DEVELOPMENT PHASE		PHASE A	PHASE B	PHASE C	PHASE D	PHASE E	PHASE F	PHASE G	PHASE H
		PERFORMANCE AND DESIGN REQUIREMENTS	IMPLEMENTATION CONCEPT AND TEST PLAN	INTERFACE AND DATA REQUIREMENTS SPECIFICATION	DETAILED DESIGN SPECIFICATION	CODING AND AUDITING	SYSTEM TESTING	CERTIFICATION AND ACCEPTANCE	OPERATIONS AND MAINTENANCE
MANAGEMENT	1	PROGRAM MANAGEMENT	PROGRAM MANAGEMENT	PROGRAM MANAGEMENT	PROGRAM MANAGEMENT	PROGRAM MANAGEMENT	PROGRAM MANAGEMENT	PROGRAM MANAGEMENT	PROGRAM MANAGEMENT
	2	PROGRAM CONTROL	PROGRAM CONTROL	PROGRAM CONTROL	PROGRAM CONTROL	PROGRAM CONTROL	PROGRAM CONTROL	PROGRAM CONTROL	PROGRAM CONTROL
REVIEWS	3		PRELIMINARY DESIGN REVIEW (PDR)	INTERFACE DESIGN REVIEW (IDR)	CRITICAL DESIGN REVIEW (CDR)	SYSTEM TEST REVIEW (STR)		ACCEPTANCE TEST REVIEW (ATR)	OPERATIONAL CRITIQUES
DOCUMENTS	4	DOCUMENT AND EDIT	DOCUMENT AND EDIT	DOCUMENT AND EDIT	DOCUMENT AND EDIT	DOCUMENT AND EDIT		DOCUMENT AND EDIT	DOCUMENT AND EDIT
	5	REPRODUCTION	REPRODUCTION	REPRODUCTION	REPRODUCTION	REPRODUCTION		REPRODUCTION	REPRODUCTION
REQUIREMENTS AND SPECIFICATIONS	6	REQUIREMENTS DEFINITION	PDR MATERIAL	EVENT GENERATION INTERFACE	PRODUCT CONFIG DETAILED TECH DESCRIPTION (PART II) (WITHOUT LISTINGS)	TECHNICAL DESCRIPTION UPDATE	PRODUCT CONFIG DETAILED TECH DESCRIPT UPDATE (PART II) (WITH LISTINGS)	REQUIREMENTS CERTIFICATION	SOFTWARE PROBLEM REPORTS (SPR)
	7	REQUIREMENTS ALLOCATION	PERFORMANCE AND DESIGN REQUIREMENTS (PART I)	COMMAND DEFINITION I/F		TRAINING DOCUMENTATION		FINAL DOCUMENTATION UPDATE (PART II)	
	8			TELEMETRY DEFINITION I/F			INTERFACE SPECIFICATIONS UPDATE		
	9			OPERATIONAL ENVIRONMENT I/F					
DESIGN	10	TRADE STUDIES	TRADE STUDIES	TRADE STUDIES	TRADE STUDIES				
	11	INTERFACE REQUIREMENTS	FUNCTIONAL DEFINITION	DATA DEFINITIONS	ALGORITHM DESIGN	ALGORITHM UPDATE	PROGRAM UPDATE		
	12	HUMAN INTERACTION	STORAGE AND TIMING ALLOCATION		PROGRAM DESIGN				
	13	STANDARDS AND CONVENTIONS	DATA BASE DEFINITION	DATA BASE DESIGN		DATA BASE UPDATE	DATA BASE UPDATE		
	14		SOFTWARE OVERVIEW (PRELIMINARY)		SOFTWARE OVERVIEW UPDATE				
CODING	15				PROTOTYPE CODING	OPERATIONAL CODING			
TESTING, CONFIGURATION CONTROL AND QRA	16		PRODUCT AND CONFIGURATION CONTROL	PRODUCT AND CONFIGURATION CONTROL	PRODUCT AND CONFIGURATION CONTROL	PRODUCT AND CONFIGURATION CONTROL	PRODUCT AND CONFIGURATION CONTROL	PRODUCT AND CONFIGURATION CONTROL	PRODUCT AND CONFIGURATION CONTROL
	17		DATA BASE CONTROL	DATA BASE CONTROL	DATA BASE CONTROL	DATA BASE CONTROL	DATA BASE CONTROL	DATA BASE CONTROL	DATA BASE CONTROL
	18	TEST REQUIREMENTS	TEST PLANS	INTERFACES	TEST PROCEDURES	DEVELOPMENT TESTING PLANNING	SYSTEM TEST PLANNING	ACCEPTANCE AND TEST PLANNING	INTEGRATION TESTING
	19					DEVELOPMENT TEST	SOFTWARE SYSTEM TEST	ACCEPTANCE DEMONSTRATION	SPR CLOSURE
	20						HARDWARE/ SOFTWARE SYSTEM TESTING		
	21	ACCEPTANCE TEST REQUIREMENTS	QUALITY AND RELIABILITY ASSURANCE PLANS	QA AND RA MONITORING	QA AND RA MONITORING	Q AND RA MONITORING	Q AND RA MONITORING	Q AND RA MONITORING	Q AND RA MONITORING
	22					TEST SUPPORT	TEST SUPPORT	TEST SUPPORT	TEST SUPPORT
	23								
OPERATIONS	24		OPERATIONAL CONCEPT	OPERATIONAL TIMELINE	OPERATIONAL CONCEPT UPDATE	USER'S MANUAL (PRELIMINARY)	OPERATIONAL TIMELINE UPDATE	USER'S MANUAL UPDATE	INTEGRATION SUPPORT
	25		TRAINING PLAN		TRAINING PLAN UPDATE	TRAINING	TRAINING	TRAINING AND REHEARSAL	TRAINING AND REHEARSAL

Figure 1. Activities as a function of software development phase.

3. *Project control plan*. This plan indicates the *project organization* and associated responsibilities (and how these may evolve throughout the project); a *work breakdown structure* identifying all project tasks to be separately controlled by job numbers; a *resource management plan* indicating how the expenditure of all critical resources (including such things as core memory and execution time) will be scheduled, monitored, and controlled; and a *project and milestone review*, plan identifying the periodic and milestone reviews, and all of the related essentials to performing the review (by whom, for whom, when, where, what, and how to prepare, perform, and follow up on the review).

4. *Product control plan*. This plan identifies the major activities involved in software product or configuration control-configuration identification, configuration control, configuration status accounting, and configuration verification-and how these activities evolve through the software life-cycle. Also included under product control plans are a *traceability plan* ensuring requirements-design-code traceability and a *deliverables plan* indicating when all contract (or internal) deliverables are due, in what form, and how they will be prepared. This subject will be discussed in more detail under Principle 3 below.
5. *Validation plan*. This plan covers much more than the usual "test plan," and is discussed in more detail under Principle 2 below.
6. *Operations and maintenance plan*. This includes an overview of the concept of operation for the system, and plans for training, installation, data entry, facility operations, output dissemination, program and data base maintenance, including associated computer, facility, personnel, and support software requirements.

The books by Hice et al. [7] and Tausworthe [8] contain good examples of the detail to be found in software project plans.

Orient the Plan Around a Phased Development Approach

The phased approach means that the major products of each phase must be thoroughly understood, and preferably documented, before going to the next one, as indicated in the "waterfall" chart in Figure 2. When this isn't done, the result is the costly variant shown in Figure 3. In this case, problems are detected in later phases (usually code and test, but frequently even during operations), which could have been detected easily in early phases and corrected inexpensively at that time. Correcting them in later phases means that a large inventory of design, code, documentation, training material, etc., must be reworked (and retested), involving much greater expenses and delays in project schedules.

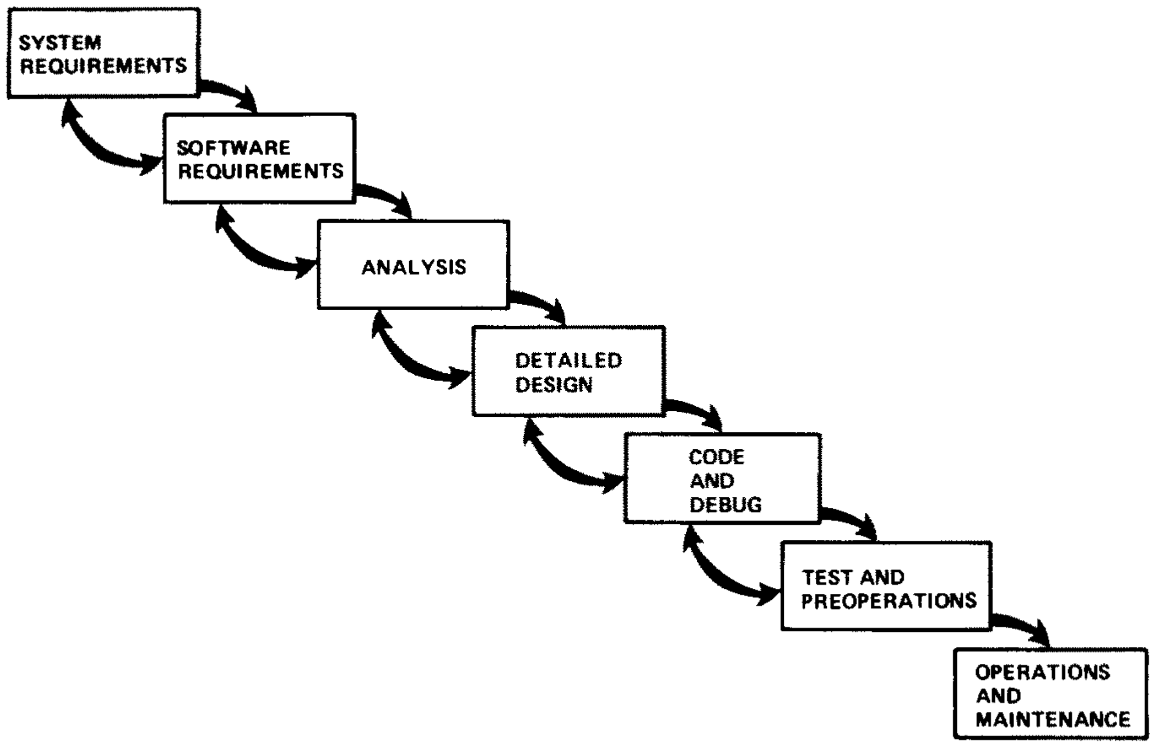


Figure 2. Confine iterations to successive stages.

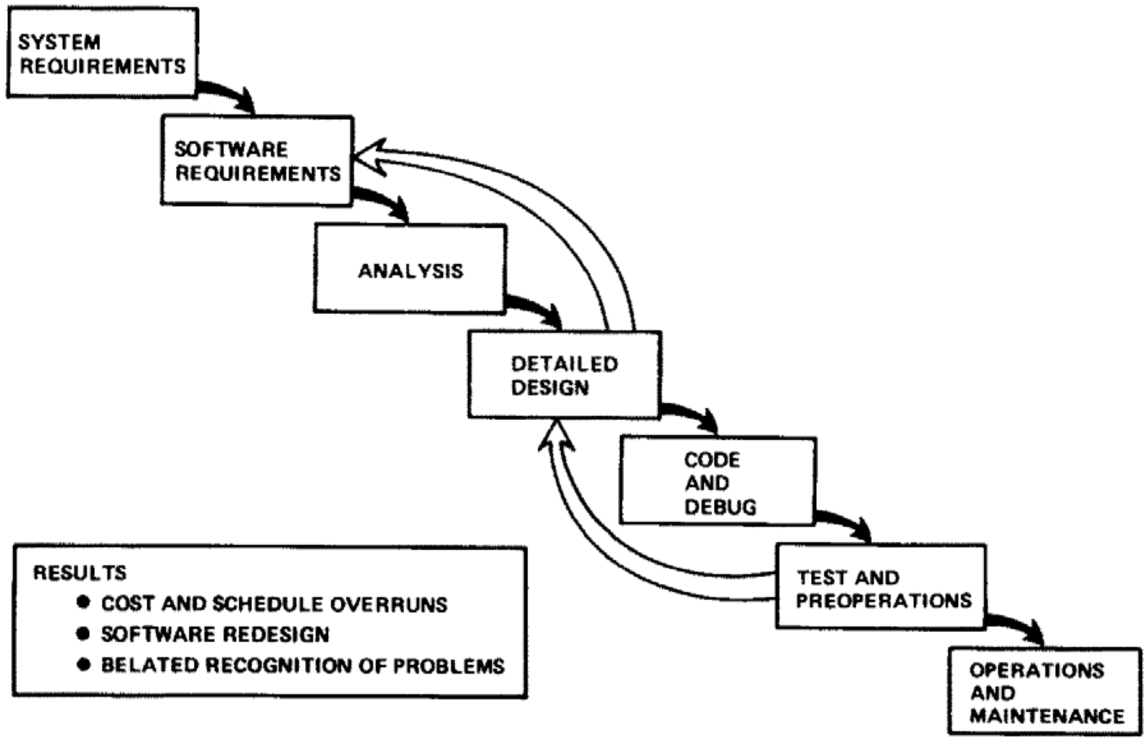


Figure 3. Examples of multiphase iteration loops.

The importance of good requirements specifications. The most important phase products in this regard are the system and software requirements because:

1. They are the hardest to fix up or resolve later.
2. They are the easiest to delay or avoid doing thoroughly.

Besides the cost-to-fix problems, there are other critical problems stemming from a lack of a good requirements specification. These include:

1. Top-down design is impossible, for lack of a well-specified "top".
2. Testing is impossible, because there is nothing to test against.
3. The user is frozen out, because there is no clear statement of what is being produced for him.
4. Management is not in control, as there is no clear statement of what the project team is producing.

Often, the problems with requirements specifications are simple omissions or errors. More often, though, the problems are those of ambiguities which look definitive but which provide a wide latitude for conflicting interpretation, which is only discovered much later. One very effective measure for detecting such insidious ambiguities is to check whether the requirements are *testable*, as exemplified in Table 3.

Table 3. Make Sure Requirements Are Testable

Nontestable	Testable
1. Accuracy shall be sufficient to support mission planning	1. Position shall be: ≤ 50′ along orbit ≤ 20′ off-orbit
2. System shall provide real-time response to status queries	2. System shall respond to: Type A queries in ≤ 2 sec Type B queries in ≤ 10 sec Type C queries in ≤ 2 min
3. System shall provide adequate core capacity for growth options.	3. System shall provide an added 25% contiguous core capacity for growth options

Making sure the requirements are testable is something that can be done early. It does require some additional hard thinking and decision making, which is one of the reasons the process is so often postponed. Another is that by hurrying on to designing and coding, one creates the illusion of rapid progress—an illusion that is virtually always false.

Prototyping, incremental development, and scaffolding. The emphasis above on phased development does not imply that a project should defer all coding until every last detail has been worked out in a set of requirements and design specifications. There are several refinements of the waterfall approach which require code to be developed early. These are:

Prototyping. the rapid, early development of critical portions of the software product as a means of user requirements determination (e.g., of user-interface features such as displays, command language options, required inputs and outputs) or requirements validation (e.g., of the real-time performance capabilities of the system).

Incremental Development. the development of a software product in several expanding increments of functional capability, as a way of hedging against development risks, of smoothing out the project's personnel requirements, and of getting something useful working early.

Scaffolding. the early development of software required to support the development, integration, and test of the operational product (e.g., interface simulators, test drivers, sample files, crossreference generators, standards checkers, diagnostic and debugging aids).

The results of a recent multiproject experiment comparing the pure-specifying and pure-prototyping approaches indicated that a mix of the two approaches was preferable to either approach used by itself [9]. Further discussion of these refinements of the waterfall model of software development is provided in Chapters 4 and 33 of [10].

The importance of Principle 1 is summarized in Figure 4, which shows the results of 151 management audits of problems in the acquisition and use of computer systems in the U.S. Government [ll]. The audits showed that deficiencies in management planning (found in 51% of the audits) and control (34%) were significantly larger sources of problems than were technology factors (15%) or other acquisition and usage factors. Principle 1 itself is well summarized in the following piece of aviators' advice: *plan the flight and fly the plan*.

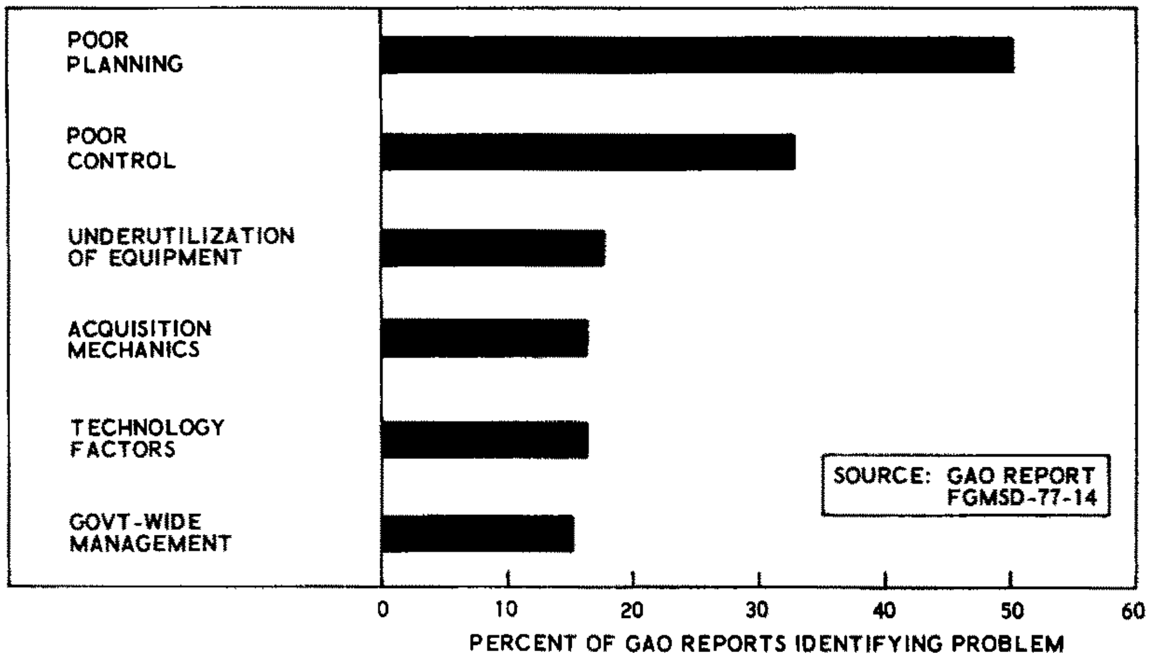


Figure 4. Problems with computer system acquisition and use in U.S. Government, 1965–1976.

Use the Plan to Manage the Project

This is the real crunch point, *where textbook management looks so easy and real-world project management is in fact so hard*. The project manager must contend with a continual stream of temptations to relax his project's discipline-especially when they are initiated by the customer or his project superiors. Some typical examples include:

- A customer request to add "a few small capabilities" to the product—but without changing the budget or schedule.
- An indication that higher management won't feel that progress is being made until they see some code produced.
- A suggestion to reduce the scope of the design review in order to make up some schedule.
- A request to take on some personnel who are between projects and find something useful for them to do.
- A request to try out a powerful new programming aid which hasn't been completely debugged.

There are usually a lot of persuasive reasons to depart from the project plan and accommodate such requests.

The easy thing to do is to be a nice guy and go along with them. It’s a lot harder to take the extra time to determine the impact of the request on the project plan and to negotiate a corresponding change in plans, schedules, and budgets. But above all else, it’s the thing that most often spells the difference between successful and unsuccessful projects.

PRINCIPLE 2: PERFORM CONTINUOUS VALIDATION

There is one single message about developing reliable software which outweighs all the others. It is to *get the errors out early*. This is the major thrust of Principle 2, “Perform Continuous Validation.” The sections below discuss why this is so important and what can be done about it.

Problem Symptoms

One of the most prevalent and costly mistakes made on software projects today is to defer the activity of detecting and correcting software problems until late in the project, i.e., in the “test and validation” phase after the code has been developed. There are two main reasons why this is a mistake: (1) Most of the errors have already been made before coding begins; and (2) The later an error is detected and corrected, the more expensive it becomes.

Figure 5, based on results obtained both at TRW [12] and at IBM [13,14], illustrates the earlier point. On large projects, and often on smaller ones, requirements and design errors outnumber coding errors. Problems such as interface inconsistencies, incomplete problem statements, ambiguous specifications, and inconsistent assumptions are the dominant ones. Coding problems such as computational accuracy, intraroutine control, and correct syntax still exist as error sources, but are relatively less significant. Table 4 [15] shows a more detailed classification of the 24 error types encountered in the command-and-control software development project shown in Figure 5. The predominant design errors tended to involve interface problems between the code and the data base, the peripheral I/O devices, and the system users.

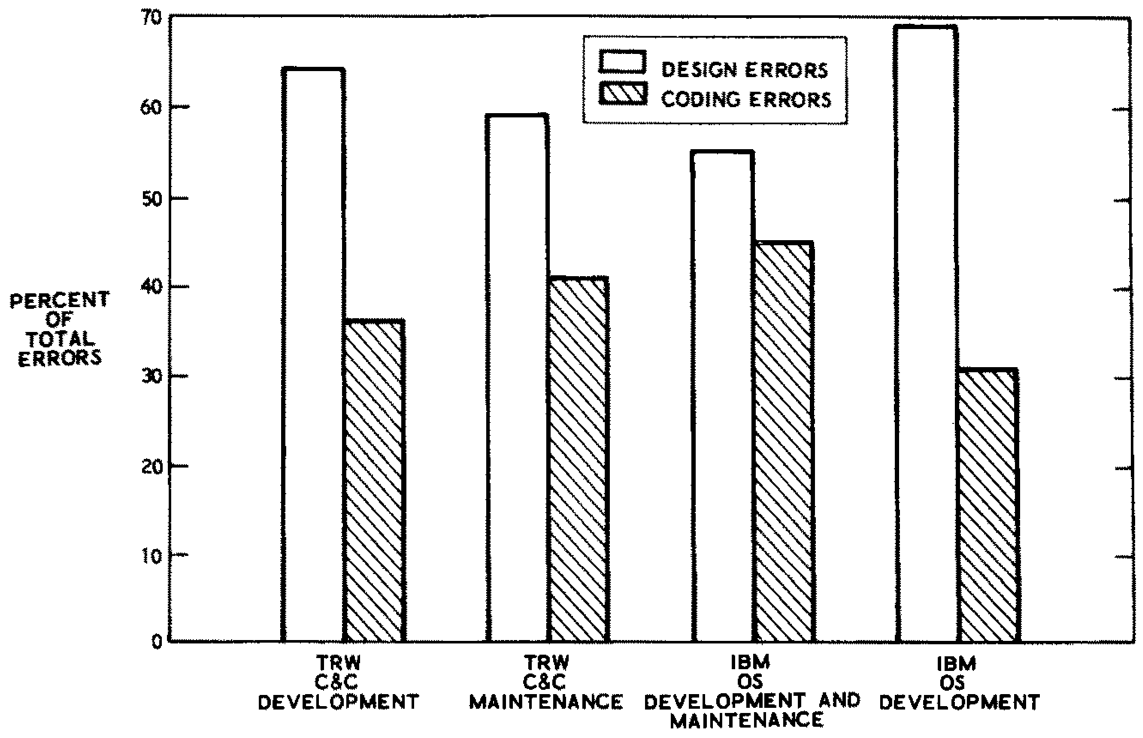


Figure 5. Most errors in large software systems are in the early stages.

Table 4. Design vs Coding Errors by Category

	No. of error types	
Error category	Design	Coding
Mostly design error types		
Tape handling	24	0
Hardware interface	9	0
Card processing	17	1
Disk handling	11	2
User interface	10	2
Error processing message	8	3
Bit manipulation	4	2
Data base interface	19	10
About even		
Listable output processing	12	8
Software interface	9	6
Iterative procedure	7	8
Mostly coding error types		
Computation	8	20
Indexing and subscription	1	19

Figure 6, based on results obtained at TRW [16], IBM [13], GTE [17], and Bell Labs [18], illustrates the second point above: that the longer you wait to detect and correct an error, the more it costs you—by a long shot. Couple that with the facts above that most errors are made early and you can see one of the main reasons why software testing and maintenance costs so much. Couple *that* with the pressures to “complete” software projects within schedule and budget and you can see one of the main reasons why software is delivered with so many errors in it.

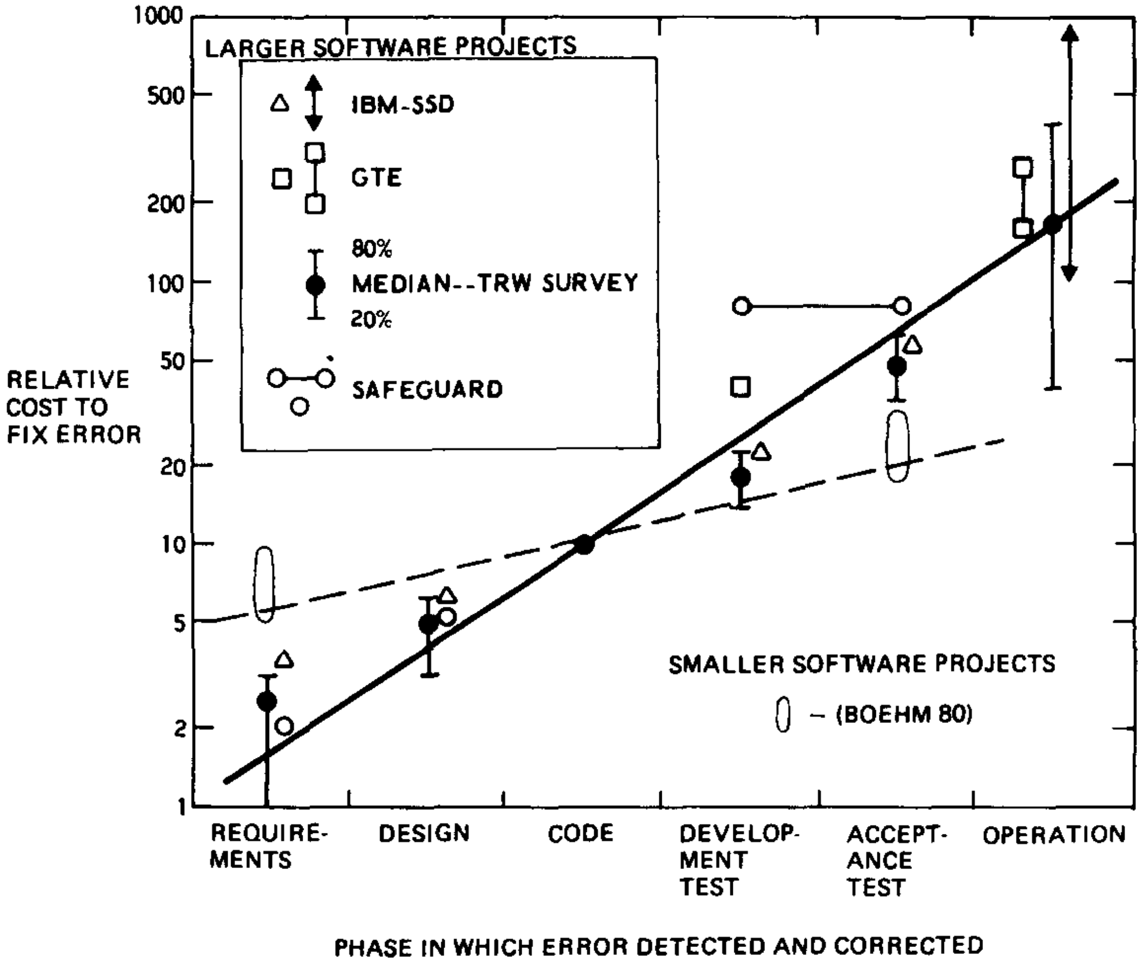


Figure 6. Increase in cost to fix or change software throughout life cycle.

Thus, we can see that it’s important both to “get the errors out early” and to “make testing and validation more efficient.” Ways to do this are discussed next.

Getting Errors Out Early

The first step is to incorporate early validation activities into the life-cycle plan. Principle 2, Perform Continuous Validation, counsels us to expand each phase of the software development process to include an explicit validation activity. The resulting elaboration of the waterfall chart is shown as Figure 7.

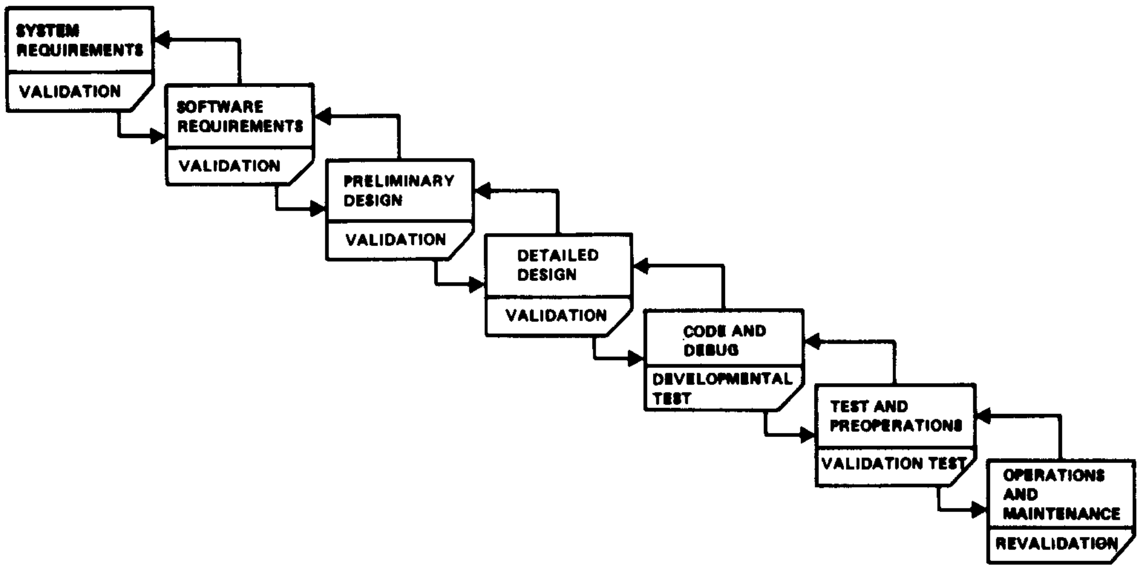


Figure 7. Manage to reliability-oriented life-cycle plan.

Each early-validation subphase implies two things: the validation activity itself and a plan preceding it. Not only should such validation activities exist in the early phases, but also, as with test planning, there should be counterpart efforts to precede the requirements and design validation subphases with explicit requirements and design validation plans.

Specific activities which aid in eliminating errors in the requirements and design phases include the following:

In-depth reviews. All too often, the review of a requirements or design specification is a one-day affair in which the reviewers are presented at 9:00 a.m. with a huge stack of paper and are expected to identify and resolve all problems with the specification by 5:30 p.m. that afternoon. This sort of "review" is bound to leave lots of errors and problems in the specification. An effective review begins with the reviewers being presented with the specification a week to a month before the official review meeting, and being provided in the meantime with briefings, walkthroughs, and other specialized meetings to discuss the intent and content of portions of the specification.

Early user documentation. Draft user's manuals, operator's manuals, and data preparation manuals should be produced and reviewed in the early design stages, not left until just before turnover. Many potential operational problems can be resolved early if the user gets a chance to understand, *in his terms,* what the system is really going to do for him from day to day—and what he will be expected to do to make the system work.

Prototyping. As discussed under Principle 1, prototyping provides an even better way to enable users to understand and determine how they wish the software to work for them. It also provides an opportunity to understand potential high-risk performance issues.

Simulations. Particularly on larger or real-time software systems, simulations are important in validating that the *performance* requirements—on throughput, response time, spare storage capacity, etc.—can be met by the design. In addition, however, simulation is a very valuable *functional* design validation activity, as it involves an independent group of operations-research oriented individuals going through the design and trying to make a valid model of it, and generally finding a number of design inconsistencies-in the process [19].

Automated aids. In analyzing the nature of design errors on TRW projects, we have found that many of them involve simple inconsistencies between module specs, I/O specs, and data base specs, on the names, dimensions, units, coordinate systems, formats, allowable ranges, etc. of input and output variables. We have had some success in building and using automated aids to detect such errors. One such aid, the design assertion consistency checker (DACC), has been used to check interface consistencies on projects with as many as 186 modules and 967 inputs and outputs. On this project, DACC was able to detect over 50 significant interface inconsistencies, and a number of other minor ones, at a cost of less than $30 in computer time [15]. Other automated aids are becoming available to support requirements and design validation, such as Teichroew's ISDOS system [20], Boeing's DECA system [21], CFG's Program Design Language support system [22], and TRW's Requirements Statement Language and Requirements Evaluation and Validation System [23,24] developed for the U.S. Army Ballistic Missile Defense Advanced Technology Center.

Design inspections and walkthroughs. An extremely effective method of eliminating design errors is to have each piece of the design reviewed by one or more individuals other than the originator. The choice of scope, technique, and degree of formality of the independent review is still fairly broad:

1. *Review team*: Generally 1-4 people, not to include managers, but generally to include the eventual programmer and tester of the item designed.
2. *Scope*: Should include checks for consistency, responsiveness to requirements, standards compliance, and "good design practices" (e.g., modularity, simplicity, provisions for handling nonstandard inputs). Detailed accuracy and performance checks are optional.
3. *Technique*: Some approaches highlight a manual walkthrough of the design element; others concentrate on independent desk-checking, generally but not necessarily followed by a review meeting. In any case, meetings are more effective when the reviewers have done homework on documentation received in advance.
4. *Formality*: Some approaches are highly formalized, with agendas, minutes, and action item worklists. Others simply specify that someone in the meeting take notes for the originator to consider in his rework. The most important thing to

formalize is that each and every design element goes through the independent review process.

The above activities may seem time consuming, but they have been shown to pay their way in practice. A fairly well-controlled study at IBM by Fagan [13] showed a net saving of 23% in total programmer time during the coding phase, and a reduction of 38% in operational errors. A study by Thayer et al. [12] of errors on projects without such inspections indicated that design inspections would have caught 58% of the errors, and code inspections 63%.

A summary of the currently known quantitative information on the relative frequency of software errors by phase, and of the relative effort required to detect them, is given in Chapter 24 of Software Engineering Economics [10]. More detailed information is given in the excellent studies of Jones [25] and Thayer et al. [12].

PRINCIPLE 3: MAINTAIN DISCIPLINED PRODUCT CONTROL

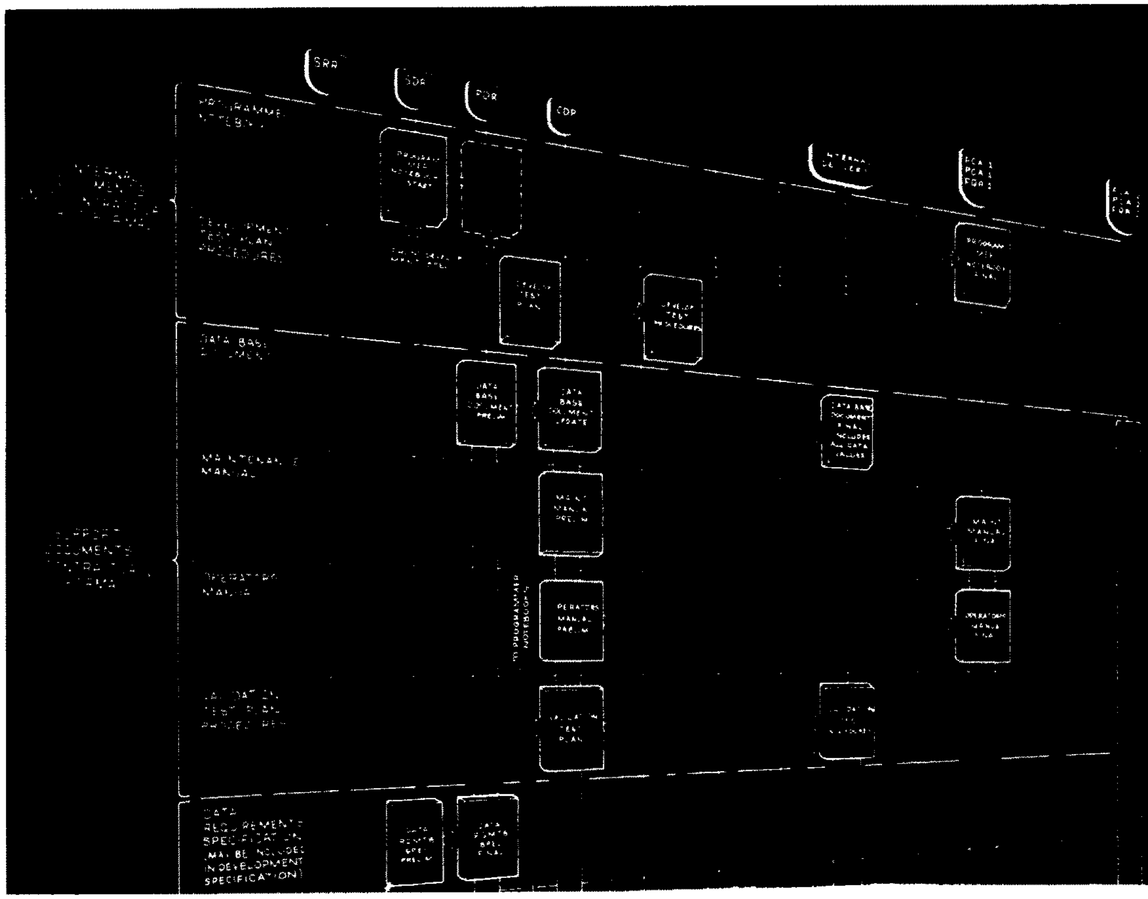
The Need for Product Control

The waterfall chart shown above is actually an oversimplified model of the software development process, even considering the refinements discussed under Principle 1. In fact, any good-sized project must accommodate changes in requirements throughout the development cycle. Some occur as the information processing job be-
Barry W. Boehm comes better understood in the more detailed phases. Many occur because of changes in the external environment: new government reporting regulations, improvements in technology, user organizational changes, or changes in the overall system-aircraft, bank, refinery, retail store-of which the software and information system is a part.

As these changes impact the system development, it is very easy for different versions of the documentation and the code to proliferate. Then, when testers or users find that the actual system is different than the one they have been preparing for, it can often take a good deal of time, money, personal strain, and sometimes legal action to straighten things out. Thus, it is most important to maintain a disciplined product control activity throughout the system life-cycle to avoid such mismatches.

Baseline Configuration Management

The most effective system of software product control that we have found is that of baseline configuration management. A *baseline* is a document or program which undergoes a formal validation or approval process and thereafter may only be modified by formal procedures established by the project. Before being baselined, a document such as a preliminary design specification for a software subsystem is easy to modify. After undergoing a preliminary design review, the document is baselined and enters into formal configuration management, under which any proposed changes must be approved by representatives of all parties concerned, and audit trails kept of all change activities. Figure 8 shows the interrelationship between the system phases and the major system reviews and audits during the software life-cycle [26].



Software Development Under Baseline Configuration Management

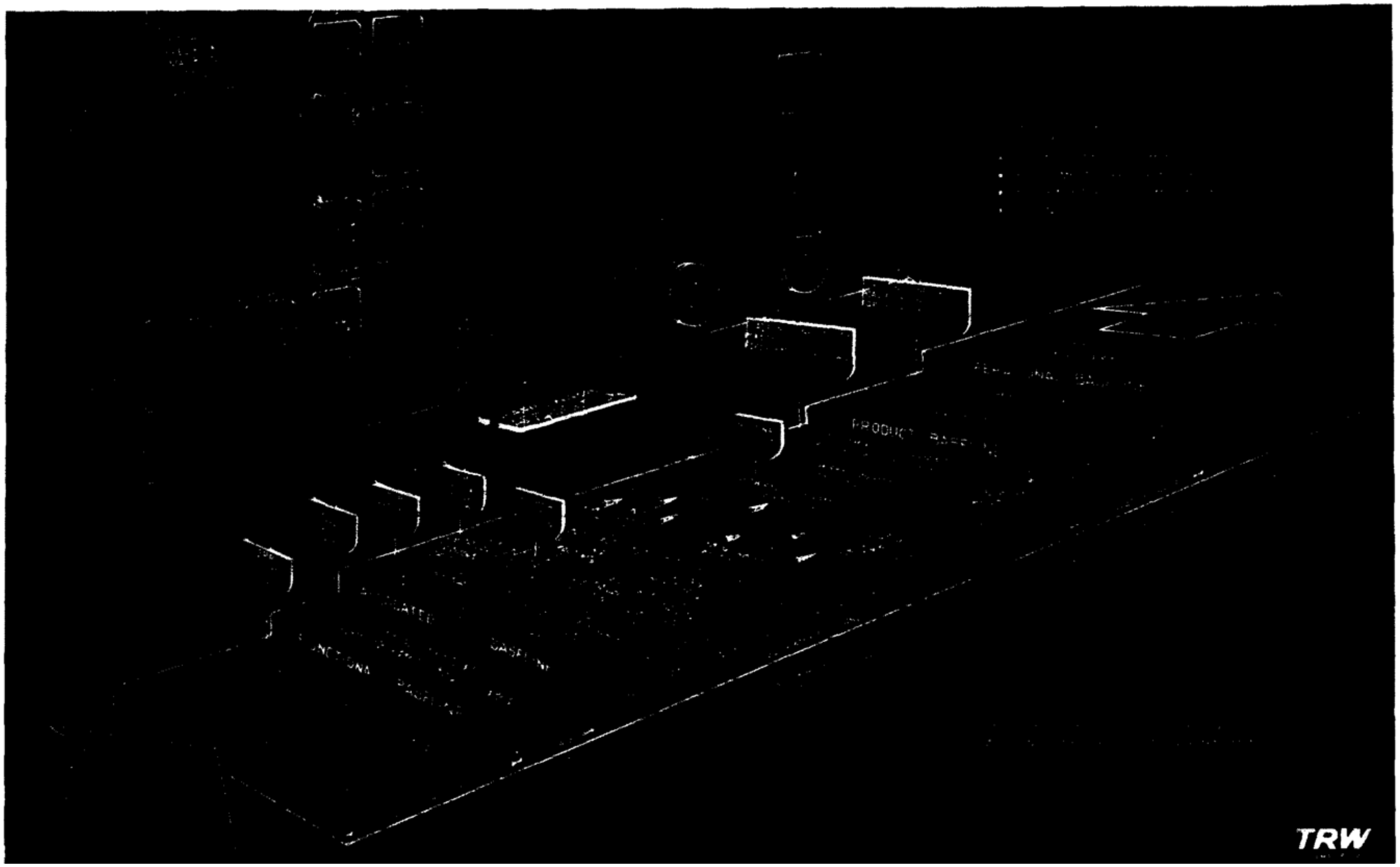


Figure 8. Detailed system life-cycle model for baseline configuration management.

The process of product control with respect to these established baselines is termed configuration management. It consists of four basic functions:

1. *Configuration identification:* The configuration of a computer program is identified by, and documented in, a series of specifications, some of which identify its required configuration and others its achieved configuration.
2. *Configuration control:* In the configuration control process, changes to the established specifications of a computer program and to the program itself are classified, evaluated, approved or disapproved, released, implemented, and verified. The purpose is to assure that the computer program configuration used in critical phases of testing, acceptance, and delivery is known and it compatible with the specifications.
3. *Configuration status accounting:* Configuration status accounting is the recording and reporting of data concerning a computer program's configuration identification, proposed changes to its configuration identification, and the implementation status of approved changes.
4. *Verification:* A series of configuration reviews and audits provide verification that the performance achieved by the product is the performance required by the development specification, and that the configuration of the product is accurately specified in the product specification.

On large projects, there may be quite a few baselined documents, including such items as interface specifications and data requirements specifications, and the configuration management system is quite formal. On small projects, an up-to-date programmer's notebook and informal change approval procedures may suffice—but the objectives and four basic functions of configuration management should still govern the change process. For further detail on software configuration management, an excellent source is the book by Bersoff et al. [27].

PRINCIPLE 4: USE MODERN PROGRAMMING PRACTICES (MPP)

The use of modern programming practices (MPP), including top-down structured programming (TDSP) and other practices such as information hiding, helps to get a good deal more visibility into the software development process, contributes greatly to getting errors out early, produces much more understandable and maintainable code, and makes many other software jobs easier, like integration and testing. However, MPP’s are by no means a substitute for the other basic principles, as some projects have proven. These are projects which started using MPP but got out of control through deficiencies in planning, project control, or validated requirements, and were not even brought to completion. Further, there are a number of ways that MPP themselves can be misused. This section begins with a short discussion of results to date using MPP and ends with a summary of problems and pitfalls in using MPP and some enhancements which help avoid the pitfalls. The two anthologies edited by Yourdon [28,29] contain many of the significant source papers on MPP, and the book edited by Glass [30] summarizes several experiences in using them.

Results to Date

There have been quite a number of papers written which cite impressive productivity gains due to the adoption of modern programming practices. Even more so than with the other factors, it has been difficult to distinguish the gains due to MPP from the effects of possibly correlated factors: use of better people, better software tools, higher management visibility, concurrent improvements in management, etc. Table 5 summarizes the results from a workshop in which a reasonably thorough (but far from exhaustive) attempt was made to relate several sets of collected data within a consistent framework [31].

Table 5. Early Experiences with MPP Average

Company	Application area	No. of projects	Average DSI	Average improvement in productivity
IBM	many	20	2–500 K	1.67
Hughes	real-time	2	10K	2.00
McAuto 73	business	2	3K	0.69
McAuto 74	business	4	6K	1.25

The negative impact of McAuto 73 was due to inadequate preparation and misuse of some of the MPP.

Subsequently, a more extensive analysis of the IBM data base was performed in [32]. Their productivity ranges represent the ranges between projects using an MPP 66%

of the time and projects using an MPP 33% of the time: structured programming: 1.78; design and code inspections: 1.54; top-down development: 1.64; chief programmer teams: 1.86.

Here these figures may include the effects of correlated factors, and certainly include a high degree of correlation between the individual factors above. That is, the ranges above largely represent the joint effect of all four MPP, since they were usually used together. The larger productivity range for use of Chief Programmer teams may also be due to another correlation effect: chief programmer teams tend to be more frequently on smaller projects than on larger projects, and smaller projects were the more productive in the IBM sample.

More recently, an MPP productivity improvement factor of 1.44 has been cited at the Bank of Montreal [33] and a factor of 1.48 at SNETCo [34].

The analysis of 63 software projects leading to the COCOMO software cost model yielded a potential productivity improvement factor due to MPP of 1.51 during development and of up to 2.07 during maintenance [10].

The GUIDE Survey of MPP Usage

Of particular interest are the results of a recent survey by GUIDE (the commercial IBM users group) of about 800 user installations on their use and evaluation of MPP [35]. Table 6 shows the installations' use of various MPP, indicating that Structured Programming and Top-Down Design have the most acceptance (roughly 50% of the installations using them; less than 10% rejecting them), while Chief Programmer Team and HIP0 have the least acceptance (roughly 33% of the installations using them; over 20% rejecting them).

Table 6. GUIDE Survey of MPP Usage: Use of Individual Techniques

What is your current use of new programming technologies?

	Rejected	Considering	Using	Total responding
Chief programming teams	134	307	224	665
Walkthru	51	288	307	646
Topdown design	43	329	332	704
Structured programming	37	351	412	800
HIPO	139	278	188	605
Librarian function	109	286	237	632
Interactive programming	86	320	280	686

Table 7 shows the installations' estimates of the effects of MPP usage on various software product and life-cycle characteristics. It indicates that the areas of greatest improvement have been in code quality and early error detection. The effects on programmer productivity and maintenance cost are strongly positive, with about 50% of the installations reporting "improved some" and about 30% reporting "improved greatly."

Table 7. GUIDE Survey of MPP Usage: Effect on Software Product and Life-Cycle

Consider only the new programming technologies you entered in the "using" column. What has been the effect on each of the following?

	Improved greatly	Improved some	No effect	Negative improvement	Total responding
Project estimating or control	63	294	206	8	571
Communication with users	89	227	252	3	571
Organizational stability	47	193	303	10	553
Accuracy of design	166	297	107	3	573
Code quality	206	287	94	2	589
Early error detection	213	276	87	4	580
Programmer productivity	165	350	80	6	601
Maintenance time or cost	178	272	108	11	569
Programmer or analyst morale	108	292	160	20	580

The productivity improvements in Table 7 represent only part of the potential MPP improvement achievable at the installation, as indicated in Table 8. This table shows the installations' response to the question: "How much further productivity improvement would you expect from using your current MPP as extensively as practical?" The results indicate that about 40% of the installations could realize an additional 10–25% productivity gain, and about 12% could realize an additional 25–50% productivity gain.

Table 8. GUIDE Survey of MPP Usage: Further Productivity Improvement Potential

Consider only the new programming technologies you checked in the "using" column. If they were to be used as extensively as practical in your installation and your current number of development people were to continue doing the same kind of work, the level (amount) of application development would be:

8	Decrease from current level
132	The same as the current level
153	0 to 10% increase over current level
264	10 to 25% increase over current level
82	25 to 50% increase over current level
18	50 to 100% increase over current level
1	More than 100% increase over current level
	Total responding = 658

MPP Implementation Guidelines

The best sets of guidelines on how to implement MPP in an organization are those in a book by Yourdon [36] and in an extensive study by Infotech [37]. Detailed checklists and representative experiences can best be obtained from these sources; the following is a set of general guidelines for MPP implementation [l0]:

1. Ensure that management is committed to the MPP implementation effort.
2. Embed the MPP implementation within an overall strategy for improving software productivity: include other options such as computer capabilities, work environment, staffing, career development; include such features as a productivity agent, an implementation study, a productivity plan, a training program, a pilot project, an incremental implementation approach, and an evaluation and improvement activity.
3. Make sure that both managers and performers agree on an appropriate set of objectives and performance criteria, e.g., clear, adaptable software rather than complex, hyperefficient software; public rather than private software; thorough, validated requirements and design specifications rather than early code.
4. Don't implement all the MPP at once. One effective three-step approach is the following: structured code and walkthroughs; top-down requirements analysis and design, structured design notation, and top-down incremental development; program library, librarian, and other team/organizational changes.
5. Allow enough time for training. Make sure that performers understand the objectives and principles of the new techniques as well as the rules.
6. Make sure that the techniques are consistently applied. Verify compliance and reject noncompliant work.
7. Avoid "structured purism." Occasionally, a GOT0 or an extra-large module is the best way to do the job. Don't confuse means and ends.
8. Don't be afraid of mistakes and false starts. They are part of the learning and assimilation experience.
9. Don't expect instant results. In fact, be prepared for some reduction in productivity during the training and assimilation period.
10. Establish a continuing evaluation activity with feedback into improving the techniques used.

PRINCIPLE 5: MAINTAIN CLEAR ACCOUNTABILITY FOR RESULTS

Even if you follow Principles 1-4, there are still ways your project can run into trouble. One such is illustrated by the data in Fig. 9. These data came from "percent complete" estimates given in weekly progress reports on a project several years ago. A manager receiving such estimates has somewhat the feeling of having sent the programmer into a deep, dark tunnel into which he has no visibility. From time to time the programmer can be heard saying, "I'm 90% through," or "I'm 98% through," but it's often a long time before he finally emerges.

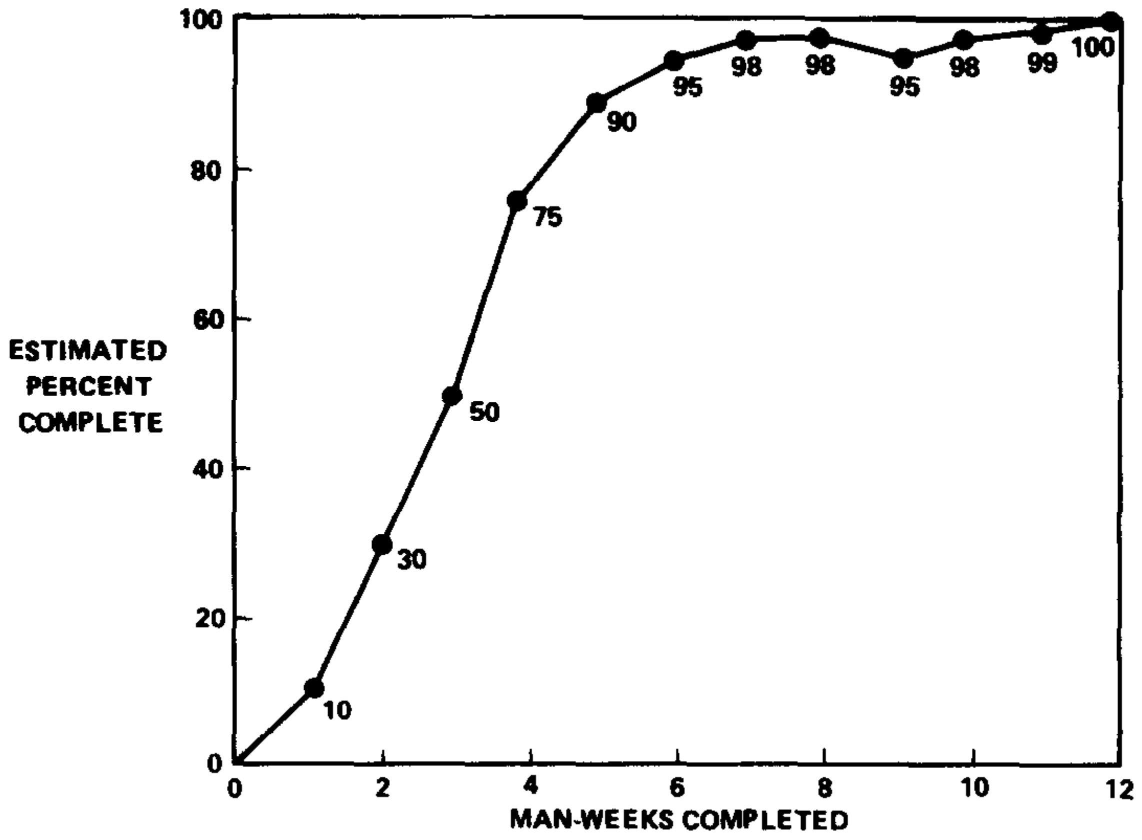


Figure 9. Sample software module development estimate.

The main problem here is insufficient concern with Principle 5: Maintain Clear Accountability for Results. Each individual on the project team should have a clear statement of the results for which he or his group are accountable, and a clear understanding that his future rewards depend on how well he does in producing those results. For a project to be able to do this, one additional thing is needed: adequate visibility into each person's or group's project performance.

This visibility is particularly difficult to achieve on software projects, with their lack of tangible products. To get this visibility, it is necessary to break the software process down into a large number of well-defined steps, and to organize the project around these steps in such a way that each individual can tell whether he’s doing his job well or not.

Macroscale Accountability

On a macroscale within the project, these steps are the major milestones indicated in Fig. 8. Thus, project managers and assistant project managers have clearly established objectives (and associated schedules and budgets). Their project plans also include a statement of their operating assumptions about interfaces, government furnished equipment, key personnel, computer availability, and any other factors which will influence how well they do their job. These statements, of course, imply accountability items for other project personnel.

Intermediate-Scale Accountability

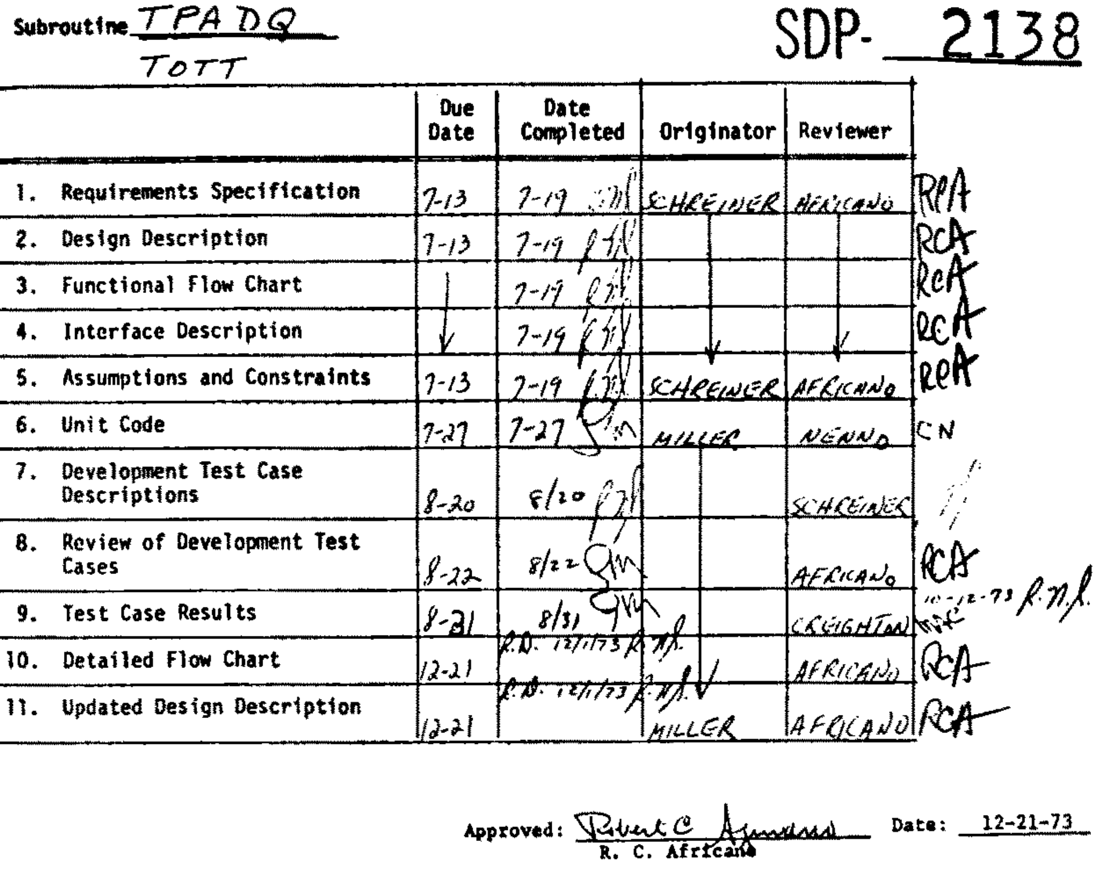
On an intermediate scale, accountability mechanisms include not only milestones but also formal agreements between subgroups. One such is a written Project Work Authorization between the project organization and the performing organization, identifying the specific subproject deliverables to be produced, and their associated schedules and budgets. (These can work well even if the project organization and the performing organization are identical.) Another involves establishing a set of formal handover criteria by which one group (say, the independent test team) determines whether or not to accept the product of another group (say, the program developers). Thus, on TRW projects, the independent test team will not accept a module for test until it has passed the following objective tests [38]:

- Satisfactorily exercised every program branch and every executable statement;
- Satisfactorily passed a “Functional Capabilities List” test, based on a list prepared from the design spec by the independent test team;
- Demonstrated conformance to all project programming standards by passing through a standards-checking program (the Code Auditor, to be described below).

Another is a formal Software Problem Report activity, in which the independent test team documents all problems detected on special forms. A log of all problem reports is kept by the project, including information on the problem originator, the date originated, the routine(s) involved, and the date closed. Closure of a problem report is handled by another special form, which must be reviewed and signed off by the independent test team.

Individual Accountability

The most effective device we have found for ensuring individual accountability, and avoiding the "90% complete" syndrome illustrated in Figure 9, is the Unit Development Folder (UDF) and its associated cover sheet [39,40]. Each routine has its own UDF, which serves as a common collection point for all the requirements, design, code, and test information associated with the routine. Visibility and accountability are ensured via the UDF cover sheet, an example of which is shown as Figure 10. Each line of the UDF cover sheet identifies a specific micromilestone associated with the routine: its requirements specification, design description, functional capabilities list for testing, etc. For each micromilestone, the individual performer schedules the date on which he will get it completed (after some negotiation with his supervisor on the endpoint dates). This obtains his personal commitment to meeting all the milestone dates. In addition, each product is independently reviewed by a co-worker, tester, or supervisor to ensure that it has been satisfactorily completed.



Subroutine TPADQ
TOTT

SDP- 2138

	Due Date	Date Completed	Originator	Reviewer	
1. Requirements Specification	7-13	7-19	SCHREINER	AFRICANO	RCA
2. Design Description	7-13	7-19			RCA
3. Functional Flow Chart		7-19			RCA
4. Interface Description		7-19			RCA
5. Assumptions and Constraints	7-13	7-19	SCHREINER	AFRICANO	RCA
6. Unit Code	7-27	7-27	MILLER	NENNO	CN
7. Development Test Case Descriptions	8-20	8/20		SCHREINER	
8. Review of Development Test Cases	8-22	8/22		AFRICANO	RCA
9. Test Case Results	8-31	8/31		CREIGHTON	10-12-73
10. Detailed Flow Chart	12-21	R.D. 12/1/73		AFRICANO	RCA
11. Updated Design Description	12-21	R.D. 12/1/73	MILLER	AFRICANO	RCA

Approved: [illegible] Date: 12-21-73
R. C. Africano

Figure 10. Unit development folder cover sheet.

The resulting visibility, accountability, and personal commitments have been very effective. Instead of a time-consuming and often inconclusive "how's it going" conversation, supervisors can find out readily how well each performer is doing on his milestones, and can thus initiate corrective action (if needed) earlier, and thus more effectively. Also, the personal commitment made by the performer will often lead to his doing more thorough preparation, or putting in a few extra hours, when necessary, in order to meet his scheduled dates.

Summary

The primary items involved in ensuring accountability are:

- Organizing the project with clearly defined responsibilities, and providing authority commensurate with responsibility;
- Establishing a goal and incentive structure such that goals are mutually reinforcing and incentives are well-aligned with goals.

The general practices of management by objectives [41,42] provide overall guidance on how to ensure accountability. More specific guidance for matching accountability practices to software projects can be found in the goal-setting and project control techniques discussed in Chapters 3 and 32 of [10], and in the people-management guidelines presented in such books as Weinberg [43].

PRINCIPLE 6: USE BETTER AND FEWER PEOPLE

Let's suppose that you conscientiously follow Principles l-5 above. Will this guarantee you a successful project? Not necessarily. You could still fail very easily, for example, if you had to staff the project with EDP school dropouts. Avoiding such a pitfall is the subject of Principle 6: "Use Better and Fewer People."

Variability Among Personnel

One main motivation for Principle 6 is the wide range of variability among individuals with respect to their software productivity. In one controlled experiment, Sackman found a variation of up to 26:1 in the time required to complete a programming assignment [44]. Another experiment at TRW found a variation among individuals of. 10:1 in the number of errors remaining in a "completed" program [45]. IBM's analyses of structured programming productivity has shown a typical variation of 5:1 among individuals [32].

Interpersonal Communications Overhead

Another main motivation for Principle 6 is that of reducing the communications overhead on a project by getting the job done with fewer, more productive performers. As indicated by Brooks [46] and Aron [47], a group of N performers has a potential of N(N-1)/2 communication paths to be concerned about. On larger projects, a hierarchical project organization will reduce this number, but the effect is still considerable. Figure 11 shows how fast the communications overhead grows when each individual spends 5% of his time in "sideways" communication with each member of his group, and 15% of his time in upwards communication with his supervisor, assuming an average group size of 4 at each hierarchical level. Even with a ten-person task, the communications overhead—the difference between the upper curve of potential productivity and the lower curve of actual productivity in Figure 11—has grown to 37%. (On the OS/360 development project, it has been estimated that 700 of the 900 project personnel were involved in communications overhead functions.)

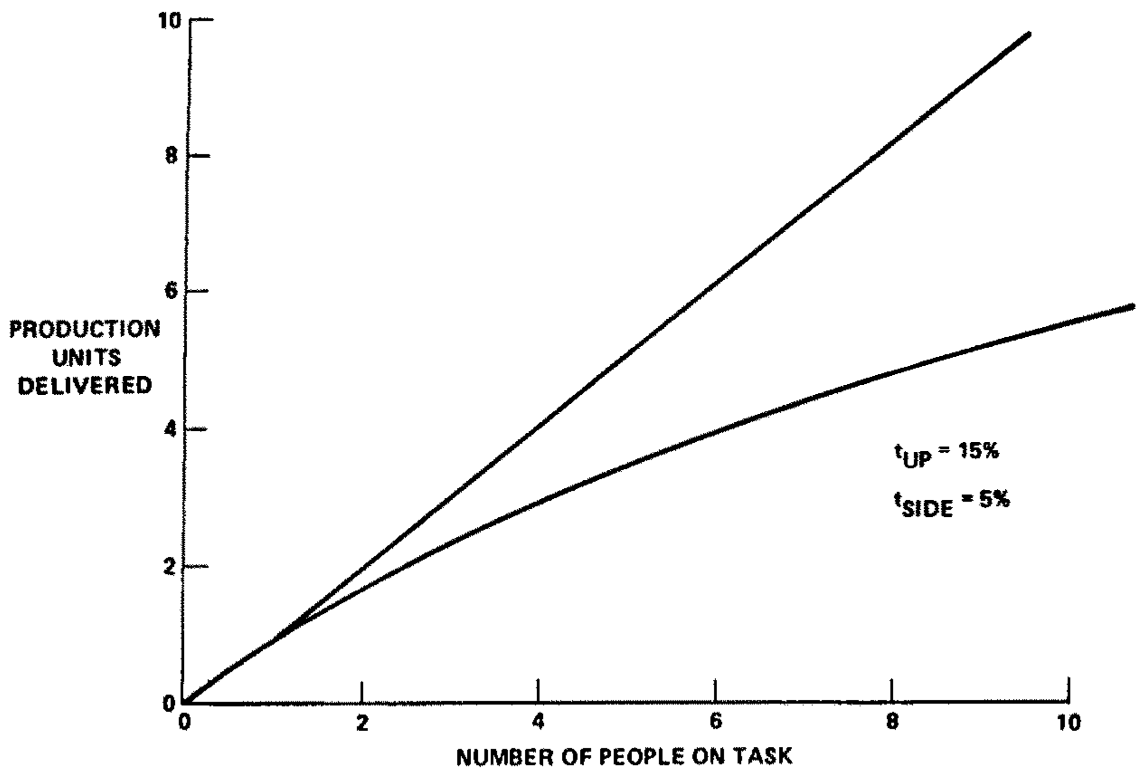


Figure 11. Rationale for automated software development aids.

Applications of Principle 6

Here are some particular high-leverage applications of Principle 6:

Don't try to solve project schedule problems by adding more people. You'll just be adding more communications overhead, particularly as the new people try to learn what's going on. This is the "mythical man-month" trap highlighted by Brooks [46].

Don't load up a project with a lot of people in the early stages. This is the period during which communications are most intense, and the requirements and design are most volatile. Getting a lot of people into the act at this time just means getting a lot of wasted effort.

Set up career paths, salary scales, and other benefits to reward high performers commensurately. As seen above, top performers typically do 5 times as much work as the bottom performers, but they are never paid anywhere near 5 times as much. Going in this direction will increase your average cost per performer, but decrease your average cost per instruction even more.

Phase out the bottom performers. This is never a pleasant thing to do, but with enough planning, time, and sensitivity, it can be done in a humane way—with no embarrassment, a more secure and satisfying alternate job for the employee, and a healthier situation for all concerned.

Automated Aids

Another major application of Principle 6 is the use of automated aids to the software process. Clearly, replacing manual tasks by computer runs will lead to projects with fewer required performers and less communications overhead. However, the automated aids can be used to even more advantage. They can make it so that people find it easier (quicker, less tedious) to do the "right" thing for the project than it is to do the wrong thing (where "right" means less error prone, easier to understand, test, modify, use, etc.). Higher-order languages and well-engineered operating systems are clear examples.

Others include [48,49]:

1. COMMON package or other data base generators;
2. Preprocessors to accommodate special applications, decision tables, COBOL shorthand, etc;
3. Subroutine and data cross-reference generators;
4. Automatic flow-charters;
5. Documentation generators;
6. Program performance evaluators;
7. Software library and module-management systems;
8. Source code consistency and singularity analyzers;
9. Test data generators;
10. Program structure analyzers and associated test data generation and test monitoring aids;
11. Test data management and retest exception reporting capabilities.

An Example of an Automated Aid: Code Auditor

As mentioned earlier in the context of a structured programming, any standard which is promulgated without any means of enforcement is highly likely to become a dead letter in a short time. This has been particularly true in the area of programming standards, where it led to the development of TRW's Code Auditor program.

The Code Auditor program can scan any FORTRAN program and produce an exception report indicating where this FORTRAN program violates a predefined set of programming standards. There are currently about 40 such standards, including:

- A set of rules for writing structured programs in standard FORTRAN;

- Requirements for commentary header blocks and comment cards at appropriate places in the code;
- Module size limits;
- Parameter passing conventions;
- Some simple data type checking;
- Conventions on supervisor calls;
- Formatting and labeling conventions.

Programmer acceptance of the Code Auditor program was somewhat difficult at first, but now it is used enthusiastically by programmers. It is used to check for standards-compliance on every line of code produced (and every routine modified) on some extremely large programs (over 500,000 card images) which would have been impossible to check otherwise. The resulting code is much easier to read and modify, and has fewer actual errors.

PRINCIPLE 7: MAINTAIN A COMMITMENT TO IMPROVE THE PROCESS

Principles l–6 are not quite enough to guarantee a healthy software engineering organization. They are enough to get an organization to do good 1982 vintage software engineering, but not enough to ensure that the organization keeps up with the times. Further, there needs to be a way to verify that the particular form of the principles adopted by an organization is indeed the best match for its particular needs and priorities. This is the motivation for Principle 7: "Maintain a Commitment to improve the Process."

This commitment is not large in terms of dollars, but it is significant in terms of the need for planning and understanding your organization. It implies not only that you commit to trying new software techniques that look promising, but also that you commit to set up some plan and activity for evaluating the effect of using the new techniques. This in turn implies that you have a way of collecting and analyzing data on how your software shop performs with and without the new techniques.

Data Collection and Analysis

Such data collection can be expensive, but it doesn't have to be. In fact, it is most effective when it is done as part of the process of managing software projects. The need for visibility and accountability expressed in Principle 5 requires that projects collect data on how their schedules and resource expenditures match up with their project plans. These data can be used as a basis for determining where the bottlenecks are in your projects, where most of the money goes, where the estimates are poorest, and as a baseline for comparing how well things go next time, when you use more new techniques. They can also be used to help estimate the costs of future software projects. The data base of 63 completed software projects used to develop the COCOMO cost model in [10] is a good example of what can be done.

Another type of project management data which can be analyzed for useful insights is the error data resulting from a formal software problem reporting activity such as that discussed under Principle 5. Table 9 shows the type of information on software errors that can be gleaned from problem reports [12]. We have been able to use such data to determine priorities on developing cost-effective tools and techniques for improving the software process [11].

Table 9. Sample Error Category List

Category ID	Categories	Project 2					Project 3	Project 4
		MODIA	MOD1B	MOD1BR	MOD2	Total		
AA000	**Computational errors**	0	0	0	0	0	0	0
AA010	Total number of entries computed incorrectly	0	0	0	0	0	19	0
AA020	Physical or logical entries number computed incorrectly	8	6	2	21	37	27	0
AA030	Index computation error	2	7	1	17	27	31	4
AA040	Wrong equation or convention used	3	6	4	11	24	57	0
AA041	Mathematical modeling problem	0	0	0	1	1	7	0
AA050	Results of arithmetic calculation inaccurate / not as expected	0	0	2	5	7	74	0
AA060	Mixed mode arithmetic error	0	0	0	0	0	0	2
AA070	Time calculation error	2	1	5	13	21	36	0
AA071	Time conversion error	0	0	0	0	0	7	0
AA072	Time truncation / rounding error	1	0	1	2	4	2	0
AA080	Sign convention error	0	2	0	5	7	16	0
AA090	Units conversion error	1	0	2	15	18	28	1
AA100	Vector calculation error	1	0	0	0	1	13	0
AA110	Calculation fails to converge	0	0	3	2	5	4	0
AA120	Quantization / truncation error	1	4	1	4	10	32	0
	Totals	19	26	21	96	162	353	7
BB000	**Logic errors**	0	0	0	0	0	0	0
BB010	Limit determination error	2	5	4	5	16	37	1
BB020	Wrong logic branch taken	1	4	1	5	11	49	0
BB030	Loop exited on wrong cycle	0	0	0	0	0	0	2
BB040	Incomplete processing	4	2	4	10	20	58	0
BB050	Endless loop during routine operation	1	4	1	0	6	35	0
BB060	Missing logic or condition test	6	9	8	26	49	233	72
BB061	Index not checked	2	0	0	1	3	59	0
BB062	Flag or specific data value not tested	5	4	8	34	51	139	0
BB070	Incorrect logic	0	0	0	0	0	0	57
BB080	Sequence of activities wrong	4	7	2	18	31	57	3
BB090	Filtering error	1	3	0	4	8	7	1
BB100	Status check / propagation error	6	3	1	2	12	103	0
BB110	Iteration step size incorrectly determined	0	0	0	0	0	0	1
BB120	Logical code produced wrong results	3	4	1	19	27	39	0
BB130	Logic on wrong routine	0	0	0	2	2	6	0
BB140	Physical characteristics of problem to be solved, overlooked, or misunderstood	1	1	0	0	2	64	2
BB150	Logic needlessly complex	0	0	0	0	0	5	0
BB160	Inefficient logic	0	2	0	2	4	26	1
BB170	Excessive logic	1	3	1	9	14	18	0
BB180	Storage reference error (software problem)	0	0	0	0	0	2	0
	Totals	37	51	31	137	256	937	140

Maintaining Perspective

Another reason for Principle 7 is to make sure that the principles serve as a *stimulus* to *thinking* about how best to do your project, *not as a substitute for thinking* about it. As long as software engineering involves people, there will be no way of reducing everything to a cookbook of principles and procedures. Another way of putting the above is:

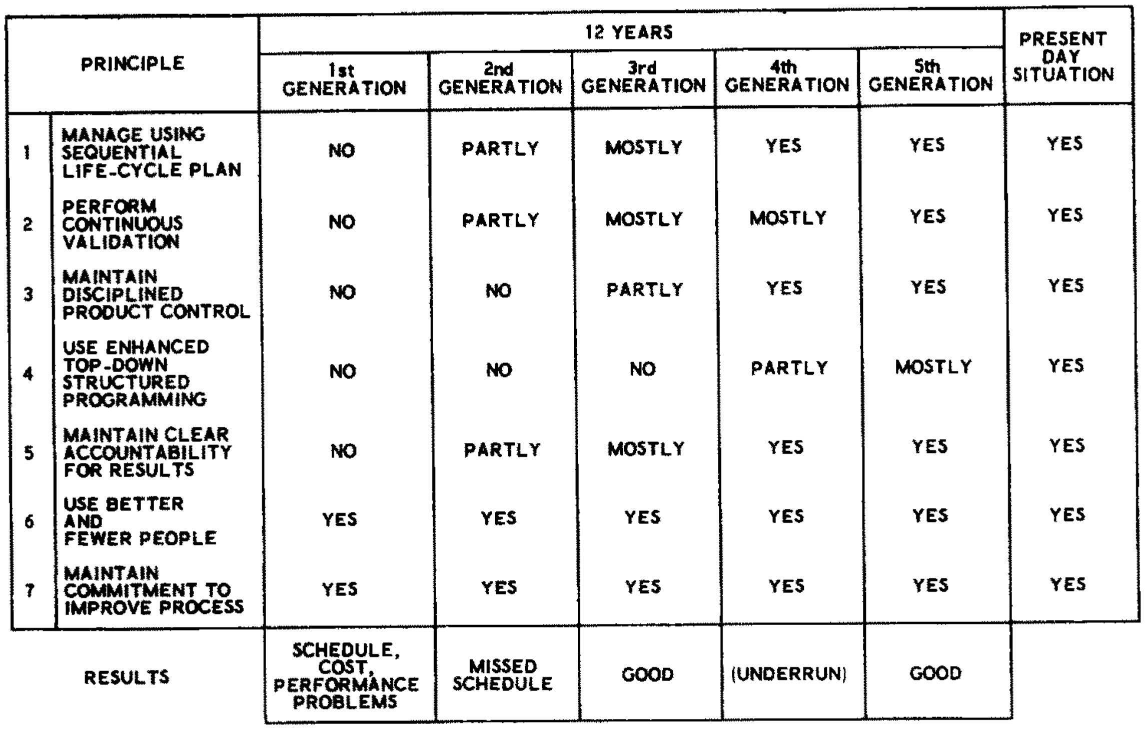
If the principles conflict with common sense, use common sense and iterate the principles.

For example, Principle 6 says “Use Better and Fewer People.” If you take this too literally, you would use a top-flight programmer to serve as your Program Librarian on a chief programmer team. But this has already been shown to lead to problems [31]. Thus, an iteration of Principle 6 would certainly include as an added interpretation or guideline: “Match the right people to the right jobs.”

SUMMARY: FIVE GENERATIONS OF EXPERIENCE

Figure 12 is a “report card” which summarizes TRW’s usage of the Seven Basic Principles over five successive command and control software projects, performed over a period of more than 12 years. It shows the extent to which each principle was followed in each project (or generation), and the resulting outcome of the project.

From the results of the ant-generation project, it is clear that simply betting on good people to pull you through is not a sufficient condition for project success. The project personnel were outstanding, but none of the first five principles were followed; the resulting project outcome displayed serious cost, schedule, and performance problems. However, the main saving grace of outstanding people is that they learn through experience. Thus, during later generations, more and more of the principles were followed more and more completely, and the resulting project outcomes became more satisfactory.



	PRINCIPLE	12 YEARS					PRESENT DAY SITUATION
		1st GENERATION	2nd GENERATION	3rd GENERATION	4th GENERATION	5th GENERATION	
1	MANAGE USING SEQUENTIAL LIFE-CYCLE PLAN	NO	PARTLY	MOSTLY	YES	YES	YES
2	PERFORM CONTINUOUS VALIDATION	NO	PARTLY	MOSTLY	MOSTLY	YES	YES
3	MAINTAIN DISCIPLINED PRODUCT CONTROL	NO	NO	PARTLY	YES	YES	YES
4	USE ENHANCED TOP-DOWN STRUCTURED PROGRAMMING	NO	NO	NO	PARTLY	MOSTLY	YES
5	MAINTAIN CLEAR ACCOUNTABILITY FOR RESULTS	NO	PARTLY	MOSTLY	YES	YES	YES
6	USE BETTER AND FEWER PEOPLE	YES	YES	YES	YES	YES	YES
7	MAINTAIN COMMITMENT TO IMPROVE PROCESS	YES	YES	YES	YES	YES	YES
	RESULTS	SCHEDULE, COST, PERFORMANCE PROBLEMS	MISSED SCHEDULE	GOOD	(UNDERRUN)	GOOD	

Figure 12. A report card.

Does this mean that we can now guarantee success on a project? Not yet. Software is still a highly complex and incompletely-understood field, with a great deal of room for differences in human judgment on how to perform a software function, or on how to tailor the seven principles to fit a particular situation. Thus, there is still a great deal of room to make mistakes—which, of course, is why the software field remains so challenging. But we can indeed say that the use of the Seven Basic Principles helps us to avoid many of these mistakes, and to identify high-risk situations earlier and with more certainty.

REFERENCES

1. W. W. Royce, Managing the Development of Large Software Systems: Concepts and Techniques, *Proceedings of WESCON*, August 1970; also in TRW Software Series, SS-70-01, August 1970.
2. E. R. Mangold, Software Visibility and Management: Technology, *Proceedings of the TRW Symposium on Reliable, Cost-Effective, Secure Software*, TRW-SS-74-14, March 1974.
3. J. A. Ward, Twenty Commandments for Managing the Development of Tactical Computer Programs, *Proceedings, I974 National Computer Conference*, pp. 803–806.
4. P. W. Metzger, *Managing a Programming Project,* Prentice-Hall, Englewood Cliffs, NJ, 1973 (2nd ed. 1981).
5. B. N. Abramson, and R. D. Kennedy, *Managing Small Projects*, TRW-SS-69-02, 1969.
6. R. W. Wolverton, The Cost of Developing Large-Scale Software, *IEEE Transactions on Computers,* 1974.
7. G. F. Hice, W. S. Turner, and L. F. Cashwell, *System Development Methodology,* North-Holland, New York, 1974.
8. R. C. Tausworthe, *Standardized Development of Computer Software,* Prentice-Hall, New York, 1977.
9. B. W. Boehm, T. E. Gray, and T. Seewaldt, Prototyping vs. Specifying: A Multi-Project Experiment, *Proceedings, IFIP 83* New York, (to appear).
10. B. W. Boehm, *Software Engineering Economics*, Prentice- Hall, New York, 1981.
11. General Accounting Office, Problems Found With Government Acquisition and Use of Computers From November 1965 to December 1976, Report FEMSD-77-14, GAO, Washington, DC. March 1977.
12. T. A. Thayer, M. Lipow, and E. C. Nelson, *Software Reliability: A Study of Large-Project Reality,* North- Holland, New York, 1978.
13. M. E. Fagan, *Design and Code Inspections and Process Control in the Development of Programs,* IBM, TR 21-572, 12/74.
14. A. B. Endres, An Analysis of Errors and Their Causes in System Programs, *IEEE Trans. Software Eng.* 140–149 (1975).
15. B. W. Boehm, R. K. McClean, and D. B. Urfrig, Some Experience with Automated Aids to the Design of Large-Scale Reliable Software, *IEEE Trans. Software Eng.* 125-138 (1975); Also TRW-SS-75-01.
16. B. W. Boehm, Software Engineering, *IEEE Trans. Computers* 1226-1241 (1976).
17. E. B. Daly, Management of Software Engineering, *IEEE Trans. Software Eng.* 229–242 (1977).
18. J. D. Musa, A Theory of Software Reliability and its Applications, *IEEE Trans. Software Eng.* 312–327 (1975).

19. T. E. Bell, *Modeling the Video Graphics System: Procedure and Model Description,* The Rand Corporation, R-5l9-PR, 1970.
20. D. Teichroew, and H. Sayari, Automation of System Building, *Datamation* 25–30 (1971).
21. L. C. Carpenter, and L. L. Tripp, Software Design Validation Tool, *Proceedings of the 1975 International Conference on Reliable Software,* IEEE Cat. No. 75 CH0940-7CSR, April 1975, pp. 395–400.
22. S. H. Caine, and E. K. Gordon, PDL-A Tool for Software Design, *AFIPS Conference Proceedings 1975 National Computer Conference* 27 l–276 (1975).
23. T. E. Bell, D. C. Bixler, and M. E. Dyer, *An Extendable Approach to Computer-Aided Software Requirements Engineering, IEEE Trans. Software Eng.* 49–59 (1977).
24. M. W. Alford, A Requirements Engineering Methodology for Real-Time Processing Requirements, *IEEE Trans. Software Eng.* 60–68 (1977).
25. T. C. Jones, Measuring Programming Quality and Productivity, *IBM Systems J.* 17, 39–63 (1978).
26. E. R. Mangold, et al. *TRW Software Development and Configuration Management Manual,* TRW-SS-73-07, 1973.
27. E. H. Bersoff, V. D. Henderson, and S. L. Siegel, *Software Configuration Management,* Prentice-Hall, New York, 1980.
28. *Classics in Software Engineering,* E. N. Yourdon (ed.), Yourdon, New York, 1979.
29. *Writings of the Revolution,* E. N. Yourdon, (ed.), Yourdon, New York, 1982.
30. R. L. Glass, *Modern Programming Practices: A Report from Industry,* Prentice-Hall, New York, 1982.
31. B. W. Boehm, C. E. Holmes, G. R. Katkus, J. P. Romanos, R. C. McHenry, and E. K. Gordon, Structured Programming: A Quantitative Assessment, *IEEE Computer* 38–54 (1975).
32. C. E. Walston, and C. P. Felix, A Method of Programming Measurement and Estimation, *IBM Systems J.* 16, 54–73 (I 977).
33. F. A. Comper, Project Management for System Quality and Development Productivity, *Proceedings, SHAREGUIDE Application Development Symposium,* 1979.
34. R. Pitchell, The GUIDE Productivity Program, *GUIDE Proceedings,* GUIDE, Inc., Chicago, II. 1979, pp. 783–794.
35. GUIDE, Inc., GUIDE Survey of New Programming Technologies, *GUIDE Proceedings,* GUIDE, Inc., Chicago, Ii, 1979, pp. 306–308.
36. E. N. Yourdon, *Managing the Structured Technologies,* Prentice-Hall, New York, 1979.
37. Infotech, Ltd., *Structured Programming: Practice and Experience,* Infotech International Ltd., Maindenhead, England, 1978.
38. F. J. Mullin, Software Test Tools: Project Experience, *Proceedings of the TRW Symposium on Reliable, Cost- Effective, Secure Software,* TRW-SS-74-14, 1914.
39. R. D. Williams, Managing the Development of Reliable Software, *Proceedings of the 1975 International Conference on Reliable Software,* IEEE Cat. No. 75 CH0940-7CSR, 1975, pp. 3–8.

40. F. S. Ingrassia, Combating the 90% Syndrome, *Datamation* 171–76 (1978).
41. P. F. Drucker, *The Practice of Management,* Harper and Row, New York, 1954.
42. H. Koontz, C. F. O'Donnell, *Principles of Management: An Analysis of Managerial Functions,* McGraw-Hill, New York, 1972.
43. G. M. Weinberg, *The Psychology of Computer Programming,* Van Nostrand Reinhold, New York, 1971.
44. H. Sackman, *Man-Computer Problem Solving,* Auerbath, 1970.
45. J. R. Brown, et al., *Quantitative Software Safety Study,* TRW report SDP-1776, 1973.
46. F. P. Brooks, Jr., *The Mythical Man-Month,* Addison- Wesley, New York, 1975.
47. J. D. Aron, *The Program Development Process: The Individual Programmer,* Addison-Wesley, New York, 1974.
48. D. J. Riefer, and S. Trattner, A Glossary of Software Tools and Techniques, *Computer* 52–60 (1977).
49. R. C. Houghton, Jr., Software Development Tools, NBS Special Publication 500-88, NBS, Washington, DC, 1982.