REPRINTED FROM IBM SYSTEMS JOURNAL, VOL15, NO 3, 1976; © 1976, 1999

Substantial net improvements in programming quality and productivity have been obtained through the use of formal inspections of design and of code. Improvements are made possible by a systematic and efficient design and code verification process, with well-defined roles for inspection participants. The manner in which inspection data is categorized and made suitable for process analysis is an important factor in attaining the improvements. It is shown that by using inspection results, a mechanism for initial error reduction followed by ever-improving error rates can be achieved.

Design and code inspections to reduce errors in program development

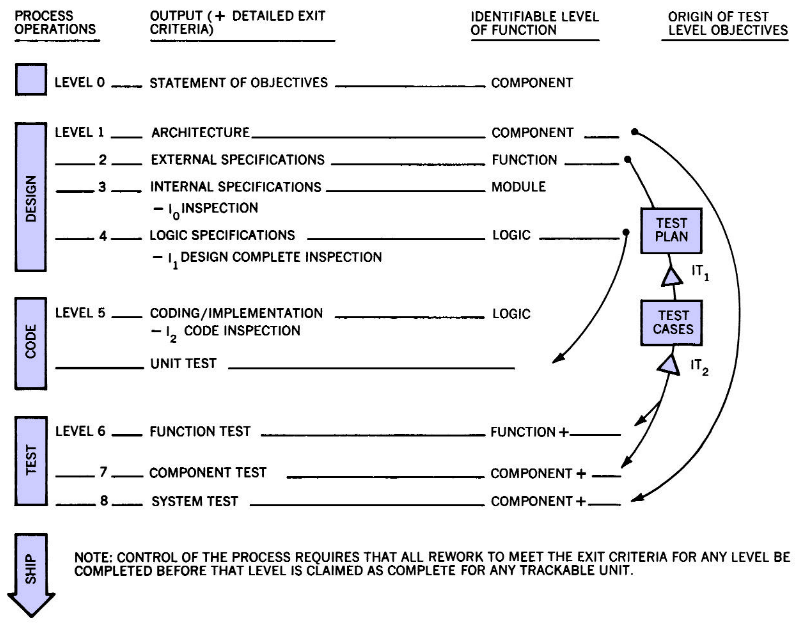
by M. E. Fagan

Successful management of any process requires planning, measurement, and control. In programming development, these requirements translate into defining the programming process in terms of a series of operations, each operation having its own exit criteria. Next there must be some means of measuring completeness of the product at any point of its development by inspections or testing. And finally, the measured data must be used for controlling the process. This approach is not only conceptually interesting, but has been applied successfully in several programming projects embracing systems and applications programming, both large and small. It has not been found to "get in the way" of programming, but has instead enabled higher predictability than other means, and the use of inspections has improved productivity and product quality. The purpose of this paper is to explain the planning, measurement, and control functions as they are affected by inspections in programming terms.

An ingredient that gives maximum play to the planning, measurement, and control elements is consistent and vigorous *discipline*. Variable rules and conventions are the usual indicators of a lack of discipline. An iron-clad discipline on all rules, which can stifle programming work, is not required but instead there should be a clear understanding of the flexibility (or nonflexibility) of each of the rules applied to various aspects of the project. An example of flexibility may be waiving the rule that all main paths will be tested for the case where repeated testing of a given path will logically do no more than add expense. An example of necessary inflexibility would be that *all* code must be

0018-8670/99/$5.00 © 1999 IBM

Figure 1 Programming process



PROCESS OPERATIONS
OUTPUT (+ DETAILED EXIT CRITERIA)
IDENTIFIABLE LEVEL OF FUNCTION
ORIGIN OF TEST LEVEL OBJECTIVES
LEVEL 0 STATEMENT OF OBJECTIVES COMPONENT
DESIGN
LEVEL 1 ARCHITECTURE COMPONENT
2 EXTERNAL SPECIFICATIONS FUNCTION
3 INTERNAL SPECIFICATIONS MODULE
– I_0 INSPECTION
4 LOGIC SPECIFICATIONS LOGIC
– I_1 DESIGN COMPLETE INSPECTION
TEST PLAN
IT_1
CODE
LEVEL 5 CODING/IMPLEMENTATION LOGIC
– I_2 CODE INSPECTION
UNIT TEST
TEST CASES
IT_2
TEST
LEVEL 6 FUNCTION TEST FUNCTION +
7 COMPONENT TEST COMPONENT +
8 SYSTEM TEST COMPONENT +
SHIP
NOTE: CONTROL OF THE PROCESS REQUIRES THAT ALL REWORK TO MEET THE EXIT CRITERIA FOR ANY LEVEL BE COMPLETED BEFORE THAT LEVEL IS CLAIMED AS COMPLETE FOR ANY TRACKABLE UNIT.

inspected. A clear statement of the project rules and changes to these rules along with faithful adherence to the rules go a long way toward practicing the required project discipline.

A prerequisite of process management is a clearly defined series of operations in the process (Figure 1). The miniprocess within each operation must also be clearly described for closer management. A clear statement of the criteria that must be satisfied to exit each operation is mandatory. This statement and accurate data collection, with the data clearly tied to trackable units of known size and collected from specific points in the process, are some essential constituents of the information required for process management.

In order to move the form of process management from qualitative to more quantitative, process terms must be more specific, data collected must be appropriate, and the limits of accuracy of the data must be known. The effect is to provide more precise information in the correct process context for decision making by the process manager.

In this paper, we first describe the programming process and places at which inspections are important. Then we discuss factors that affect productivity and the operations involved with inspections. Finally, we compare inspections and walk-throughs on process control.

The process

a manageable process

A process may be described as a set of operations occurring in a definite sequence that operates on a given input and converts it to some desired output. A general statement of this kind is sufficient to convey the notion of the process. In a practical application, however, it is necessary to describe the input, output, internal processing, and processing times of a process in very specific terms if the process is to be executed and practical output is to be obtained.

In the programming development process, explicit requirement statements are necessary as input. The series of processing operations that act on this input must be placed in the correct sequence with one another, the output of each operation satisfying the input needs of the next operation. The output of the final operation is, of course, the explicitly required output in the form of a verified program. Thus, the objective of each processing operation is to receive a defined input and to produce a definite output that satisfies a specific set of exit criteria. (It goes without saying that each operation can be considered as a miniprocess itself.) A well-formed process can be thought of as a continuum of processing during which sequential sets of exit criteria are satisfied, the last set in the entire series requiring a well-defined end product. Such a process is not amorphous. It can be measured and controlled.

exit criteria

Unambiguous, explicit, and universally accepted exit criteria would be perfect as process control checkpoints. It is frequently argued that universally agreed upon checkpoints are impossible in programming because all projects are different, etc. However, *all* projects do reach the point at which there is a project checkpoint. As it stands, any trackable unit of code achieving a clean compilation can be said to have satisfied a universal exit criterion or checkpoint in the process. Other checkpoints can also be selected, albeit on more arguable premises, but once the premises are agreed upon, the checkpoints become visible in most, if not all, projects. For example, there is a point at which the design of a program is considered complete. This point may be described as the level of detail to which a unit of design is reduced so that one design statement will materialize in an estimated three to 10 source code instructions (or, if desired, five to 20, for that matter). Whichever particular ratio is selected across a project, it provides a checkpoint for the process control of that project. In this way, suitable checkpoints may be selected throughout the development process and used in process management. (For more specific exit criteria see Reference 1.)

The cost of reworking errors in programs becomes higher the later they are reworked in the process, so every attempt should be made to find and fix errors as early in the process as possible. This cost has led to the use of the inspections described later and to the description of exit criteria which include assuring that all errors known at the end of the inspection of the new "clean-compilation" code, for example, have been correctly fixed. So, rework of all known errors up to a particular point must be complete before the associated checkpoint can be claimed to be met for any piece of code.

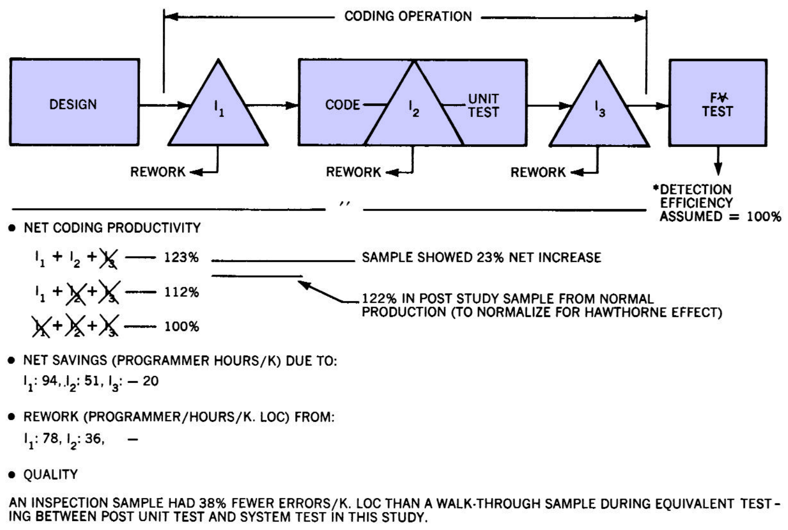
Where inspections are not used and errors are found during development or testing, the cost of rework as a fraction of overall development cost can be suprisingly high. For this reason, errors should be found and fixed as close to their place of origin as possible.

Production studies have validated the expected quality and productivity improvements and have provided estimates of standard productivity rates, percentage improvements due to inspections, and percentage improvements in error rates which are applicable in the context of large-scale operating system program production. (The data related to operating system development contained herein reflect results achieved by IBM in applying the subject processes and methods to representative samples. Since the results depend on many factors, they cannot be considered representative of every situation. They are furnished merely for the purpose of illustrating what has been achieved in sample testing.)

The purpose of the test plan inspection IT_1, shown in Figure 1, is to find voids in the functional variation coverage and other discrepancies in the test plan. IT_2, test case inspection of the test cases, which are based on the test plan, finds errors in the test cases. The total effects of IT_1 and IT_2 are to increase the integrity of testing and, hence, the quality of the completed product. And, because there are less errors in the test cases to be debugged during the testing phase, the overall project schedule is also improved.

A process of the kind depicted in Figure 1 installs all the intrinsic programming properties in the product as required in the statement of objectives (Level 0) by the time the coding operation (Level 5) has been completed—except for packaging and publications requirements. With these exceptions, all later work is of a verification nature. This verification of the product provides no contribution to the product during the essential development (Levels 1 to 5); it only adds error detection and elimination (frequently at one half of the development cost). I_0, I_1, and I_2 inspections were developed to measure and influence intrinsic

Figure 2 A study of coding productivity


CODING OPERATION
DESIGN
I_1
CODE
I_2
UNIT TEST
I_3
F* TEST
REWORK
REWORK
REWORK
*DETECTION EFFICIENCY ASSUMED = 100%
SAMPLE SHOWED 23% NET INCREASE
122% IN POST STUDY SAMPLE FROM NORMAL PRODUCTION (TO NORMALIZE FOR HAWTHORNE EFFECT)

- NET CODING PRODUCTIVITY

  $I_1 + I_2 + \cancel{I_3}$ — 123%

  $I_1 + \cancel{I_2} + \cancel{I_3}$ — 112%

  $\cancel{I_1} + \cancel{I_2} + \cancel{I_3}$ — 100%

- NET SAVINGS (PROGRAMMER HOURS/K) DUE TO:

  I_1: 94, I_2: 51, I_3: − 20

- REWORK (PROGRAMMER/HOURS/K. LOC) FROM:

  I_1: 78, I_2: 36, −

- QUALITY

AN INSPECTION SAMPLE HAD 38% FEWER ERRORS/K. LOC THAN A WALK-THROUGH SAMPLE DURING EQUIVALENT TESTING BETWEEN POST UNIT TEST AND SYSTEM TEST IN THIS STUDY.

quality (error content) in the early levels, where error rework can be most economically accomplished. Naturally, the beneficial effect on quality is also felt in later operations of the development process and at the end user's site.

An improvement in productivity is the most immediate effect of purging errors from the product by the I_0, I_1, and I_2 inspections. This purging allows rework of these errors very near their origin, early in the process. Rework done at these levels is 10 to 100 times less expensive than if it is done in the last half of the process. Since rework detracts from productive effort, it reduces productivity in proportion to the time taken to accomplish the rework. It follows, then, that finding errors by inspection and reworking them earlier in the process reduces the overall rework time and increases productivity even within the early operations and even more over the total process. Since less errors ship with the product, the time taken for the user to install programs is less, and his productivity is also increased.

The quality of documentation that describes the program is of as much importance as the program itself for poor quality can mislead the user, causing him to make errors quite as important as errors in the program. For this reason, the quality of program documentation is verified by publications inspections (PI_0, PI_1, and PI_2). Through a reduction of user-encountered errors, these inspections also have the effect of improving user productivity by reducing his rework time.

A study of coding productivity

A piece of the design of a large operating system component (all done in structured programming) was selected as a study sample (Figure 2). The sample was judged to be of moderate complexity. When the piece of design had been reduced to a level of detail sufficient to meet the Design Level 4 exit criteria[2] (a level of detail of design at which one design statement would ultimately appear as three to 10 code instructions), it was submitted to a design-complete inspection (100 percent), I_1. On conclusion of I_1, all error rework resulting from the inspection was completed, and the design was submitted for coding in PL/S. The coding was then done, and when the code was brought to the level of the first clean compilation,[2] it was subjected to a code inspection (100 percent), I_2. The resultant rework was completed and the code was subjected to unit test. After unit test, a unit test inspection, I_3, was done to see that the unit test plan had been fully executed. Some rework was required and the necessary changes were made. This step completed the coding operation. The study sample was then passed on to later process operations consisting of building and testing.

inspection sample

The inspection sample was considered of sufficient size and nature to be representative for study purposes. Three programmers designed it, and it was coded by 13 programmers. The inspection sample was in modular form, was structured, and was judged to be of moderate complexity on average.

coding operation productivity

Because errors were identified and corrected in groups at I_1 and I_2, rather than found one-by-one during subsequent work and handled at the higher cost incumbent in later rework, the overall amount of error rework was minimized, even within the coding operation. Expressed differently, considering the inclusion of *all* I_1 time, I_2 time, and resulting error rework time (with the usual coding and unit test time in the total time to complete the operation), a *net* saving resulted when this figure was compared to the no-inspection case. This net saving translated into a 23 percent increase in the productivity of the coding operation alone. Productivity in later levels was also increased because there was less error rework in these levels due to the effect of inspections, but the increase was not measured directly.

An important aspect to consider in any production experiment involving human beings is the Hawthorne Effect.[3] If this effect is not adequately handled, it is never clear whether the effect observed is due to the human bias of the Hawthorne Effect or due to the newly implemented change in process. In this case a *control sample* was selected at random from many pieces of work *after the I_1 and I_2 inspections were accepted as commonplace.* (Previous experience without I_1 and I_2 approximated the net coding productivity rate of 100 percent datum in Figure 2.) The difference in coding productivity between the experimental sample (with I_1 and I_2 for the first time) and the control sample was 0.9 percent. This difference is not considered significant. Therefore, the measured increase in coding productivity of 23 percent is considered to validly accrue from the only change in the process: addition of I_1 and I_2 inspections.

control sample

The control sample was also considered to be of representative size and was from the same operating system component as the study sample. It was designed by four programmers and was coded by seven programmers. And it was considered to be of moderate complexity on average.

net savings

Within the coding operation only, the net savings (including inspection and rework time) in programmer hours per 1000 Non-Commentary Source Statements (K.NCSS)[4] were I_1: 94, I_2: 51, and I_3: −20. As a consequence, I_3 is no longer in effect.

If personal fatigue and downtime of 15 percent are allowed in addition to the 145 programmer hours per K.NCSS, the saving approaches one programmer month per K.NCSS (assuming that our sample was truly representative of the rest of the work in the operating system component considered).

error rework

The error rework in programmer hours per K.NCSS found in this study due to I_1 was 78, and 36 for I_2(24 hours for design errors and 12 for code errors). Time for error rework must be specifically scheduled. (For scheduling purposes it is best to develop rework hours per K.NCSS from history depending upon the particular project types and environments, but figures of 20 hours for I_1, and 16 hours for I_2 (*after the learning curve*) may be suitable to start with.)

quality

The only comparative measure of quality obtained was a comparison of the inspection study sample with a fully comparable piece of the operating system component that was produced similarly, except that walk-throughs were used in place of the I_1 and I_2 inspections. (Walk-throughs[5] were the practice before implementation of I_1 and I_2 inspections.) The process span in which the quality comparison was made was seven months of testing beyond unit test after which it was judged that both samples had been equally exercised. The results showed the inspection sample to contain 38 percent less errors than the walk-through sample.

Note that up to inspection I_2, no machine time has been used for debugging, and so machine time savings were not mentioned. Although substantial machine time is saved overall since there are less errors to test for in inspected code in later stages of the process, no actual measures were obtained.

Table 1 Error detection efficiency

Process Operations	Errors Found per K.NCSS	Percent of Total Errors Found
Design		
I_1 inspection		
Coding	38*	82
I_2 inspection		
Unit test		
Preparation for acceptance test	8	18
Acceptance test	0	
Actual usage (6 mo.)	0	
Total	46	100

*51% were logic errors, most of which were missing rather than due to incorrect design.

Inspections in applications development

In the development of applications, inspections also make a significant impact. For example, an application program of eight modules was written in COBOL by Aetna Corporate Data Processing department, Aetna Life and Casualty, Hartford, Connecticut, in June 1975.[6] Two programmers developed the program. The number of inspection participants ranged between three and five. The only change introduced in the development process was the I_1 and I_2 inspections. The program size was 4,439 Non-Commentary Source Statements.

An automated estimating program, which is used to produce the normal program development time estimates for all the Corporate Data Processing department's projects, predicted that designing, coding, and unit testing this project would require 62 programmer days. In fact, the time actually taken was 46.5 programmer days including inspection meeting time. The resulting saving in programmer resources was 25 percent.

The inspections were obviously very thorough when judged by the inspection error detection efficiency of 82 percent and the later results during testing and usage as shown in Table 1.

The results achieved in Non-Commentary Source Statements per Elapsed Hour are shown in Table 2. These inspection rates are four to six times faster than for systems programming. If these rates are generally applicable, they would have the effect of making the inspection of applications programs much less expensive.

Table 2 Inspection rates in NCSS per hour

Operations	I_1	I_2
Preparation	898	709
Inspection	652	539

Inspections

Inspections are a *formal*, *efficient*, and *economical* method of finding errors in design and code. All instructions are addressed

Table 3. Inspection process and rate of progress

Process operations	*Rate of progress* (loc/hr) Design I_1*	*Code I_2*	*Objectives of the operation*
1. Overview	500	not necessary	Communication education
2. Preparation	100	125	Education
3. Inspection	130	150	*Find errors*
4. Rework	20 hrs/K.NCSS	16 hrs/K.NCSS	Rework and resolve errors found by inspection
5. Follow-up	–	–	See that all errors, problems, and concerns have been resolved

*These notes apply to systems programming and are conservative. Comparable rates for applications programming are much higher. Initial schedules may be started with these numbers and as project history that is keyed to unique environments evolves, the historical data may be used for future scheduling algorithms.

at least once in the conduct of inspections. Key aspects of inspections are exposed in the following text through describing the I_1 and I_2 inspection conduct and process. I_0, IT_1, IT_2, PI_0, PI_1, and PI_2 inspections retain the same essential properties as the I_1 and I_2 inspections but differ in materials inspected, number of participants, and some other minor points.

the people involved

The inspection team is best served when its members play their particular roles, assuming the particular vantage point of those roles. These roles are described below:

1. *Moderator*—The *key person* in a successful inspection. He must be a competent programmer but need *not* be a technical expert on the program being inspected. To preserve objectivity and to increase the integrity of the inspection, it is usually advantageous to use a moderator from an unrelated project. The moderator must manage the inspection team and offer leadership. Hence, he must use personal sensitivity, tact, and drive in balanced measure. His use of the strengths of team members should produce a synergistic effect larger than their number; in other words, *he is the coach.* The duties of moderator also include scheduling suitable meeting places, reporting inspection results within one day, and follow-up on rework. *For best results the moderator should be specially trained.* (This training is brief but very advantageous.)
2. *Designer*—The programmer responsible for producing the program design.
3. *Coder/Implementor*—The programmer responsible for translating the design into code.
4. *Tester*—The programmer responsible for writing and/or executing test cases or otherwise testing the product of the designer and coder.

If the coder of a piece of code also designed it, he will function in the designer role for the inspection process; a coder from some related or similar program will perform the role of the coder. If the same person designs, codes, and tests the product code, the coder role should be filled as described above, and another coder—preferably with testing experience—should fill the role of tester.

Four people constitute a good-sized inspection team, although circumstances may dictate otherwise. The team size should not be artificially increased over four, but if the subject code is involved in a number of interfaces, the programmers of code related to these interfaces may profitably be involved in inspection. Table 3 indicates the inspection process and rate of progress.

scheduling inspections and rework

The total time to complete the inspection process from overview through follow-up for I_1 or I_2 inspections with four people involved takes about 90 to 100 people-hours for systems programming. Again, these figures may be considered conservative but they will serve as a starting point. Comparable figures for applications programming tend to be much lower, implying lower cost per K.NCSS.

Because the error detection efficiency of most inspection teams tends to dwindle after two hours of inspection but then picks up after a period of different activity, it is advisable to schedule inspection sessions of no more than two hours at a time. Two two-hour sessions per day are acceptable.

The time to do inspections and resulting rework must be scheduled and managed with the same attention as other important project activities. (After all, as is noted later, for one case at least, it is possible to find approximately two thirds of the errors reported during an inspection.) If this is not done, the immediate work pressure has a tendency to push the inspections and/or rework into the background, postponing them or avoiding them altogether. The result of this short-term respite will obviously have a much more dramatic long-term negative effect since the finding and fixing of errors is delayed until later in the process (and after turnover to the user). Usually, the result of postponing early error detection is a lengthening of the overall schedule and increased product cost.

Scheduling inspection time for modified code may be based on the algorithms in Table 3 *and on judgment*.

I_1 inspection process

Keeping the objective of each operation in the forefront of team activity is of paramount importance. Here is presented an outline of the I_1 inspection process operations.

Figure 3 Summary of design inspections by error type

VP	Individual Name	Inspection file Missing	Wrong	Extra	Errors	Error %
CD	CB Definition	16	2		18	3.5 }10.4
CU	CB Usage	18	17	1	36	6.9
FS	FPFS	1			1	.2
IC	Interconnect Calls	18	9		27	5.2
IR	Interconnect Reqts	4	5	2	11	2.1
LO	Logic	126	57	24	207	39.8 ⬅
L3	Higher Lvl Docu	1		1	2	.4
MA	Mod Attributes	1			1	.2
MD	More Detail	24	6	2	32	6.2
MN	Maintainability	8	5	3	16	3.1
OT	Other	15	10	10	35	6.7
PD	Pass Data Areas		1		1	.2
PE	Performance	1	2	3	6	1.2
PR	Prologue/Prose	44	38	7	89	17.1 ⬅
RM	Return Code/Msg	5	7	2	14	2.7
RU	Register Usage	1	2		3	.6
ST	Standards					
TB	Test & Branch	12	7	2	21	4.0
		295	168	57	520	100.0
		57%	32%	11%		

Figure 4 Summary of code inspections by error type

VP	Individual Name	Inspection file Missing	Wrong	Extra	Errors	Error %
CC	Code Comments	5	17	1	23	6.6
CU	CB Usage	3	21	1	25	7.2
DE	Design Error	31	32	14	77	22.1 ⬅
F1			8		8	2.3
IR	Interconnect Calls	7	9	3	19	5.5
LO	Logic	33	49	10	92	26.4 ⬅
MN	Maintainability	5	7	2	14	4.0
OT	Other					
PE	Performance	3	2	5	10	2.9
PR	Prologue/Prose	25	24	3	52	14.9 ⬅
PU	PL/S or BAL Use	4	9	1	14	4.0
RU	Register Usage	4	2		6	1.7
SU	Storage Usage	1			1	.3
TB	Test & Branch	2	5		7	2.0
		123	185	40	348	100.0

1. *Overview* (whole team)—The designer first describes the overall area being addressed and then the specific area he has designed in detail—logic, paths, dependencies, etc. Documentation of design is distributed to all inspection participants on conclusion of the overview. (For an I_2 inspection, no overview is necessary, but the participants should remain the same. Preparation, inspection, and follow-up proceed as for I_1 but, of course, using code listings *and* design specifications

as inspection materials. Also, at I_2 the moderator should flag for special scrutiny those areas that were reworked since I_1 errors were found *and other design changes* made.)

. *Preparation* (individual) — Participants, using the design documentation, literally do their homework to try to understand the design, its intent and logic. (Sometimes flagrant errors are found during this operation, but in general, the number of errors found is not nearly as high as in the inspection operation.) To increase their error detection in the inspection, the inspection team should first study the ranked distributions of error types found by recent inspections. This study will prompt them to concentrate on the most fruitful areas. (See examples in Figures 3 and 4.) Checklists of clues on finding these errors should also be studied. (See partial examples of these lists in Figures 5 and 6 and complete examples for I_0 in Reference 1 and for I_1 and I_2 in Reference 7.)

. *Inspection* (whole team) — A "reader" chosen by the moderator (usually the coder) describes how he will implement the design. He is expected to paraphrase the design as expressed by the designer. Every piece of logic is covered at least once, and every branch is taken at least once. All higher-level documentation, high-level design specifications, logic specifications, etc., and macro and control block listings at I_2 must be available and present during the inspection.

Now that the design is understood, *the objective is to find errors*. (Note that an error is defined as any condition that causes malfunction or that precludes the attainment of expected or previously specified results. Thus, deviations from specifications are clearly termed errors.) The finding of errors is actually done during the implementor/coder's discourse. Questions raised are pursued only to the point at which an error is recognized. It is noted by the moderator; its type is classified; severity (major or minor) is identified, and the inspection is continued. Often the solution of a problem is obvious. If so, it is noted, but no specific solution hunting is to take place during inspection. (The inspection is *not* intended to redesign, evaluate alternate design solutions, or to find solutions to errors; it is intended just to find errors!) A team is most effective if it operates with only one objective at a time.

Within one day of conclusion of the inspection, the moderator should produce a written report of the inspection and its findings to ensure that all issues raised in the inspection will be addressed in the rework and follow-up operations. Examples of these reports are given as Figures 7A, 7B, and 7C.

Figure 5 Examples of what to examine when looking for errors at I_1

I_1 Logic

Missing

1. Are All Constants Defined?
2. Are All Unique Values Explicitly Tested on Input Parameters?
3. Are Values Stored after They Are Calculated?
4. Are All Defaults Checked Explicitly Tested on Input Parameters?
5. If Character Strings Are Created Are They Complete, Are All Delimiters Shown?
6. If a Keyword Has Many Unique Values, Are They All Checked?
7. If a Queue Is Being Manipulated, Can the Execution Be Interrupted; If So, Is Queue Protected by a Locking Structure; Can Queue Be Destroyed Over an Interrupt?
8. Are Registers Being Restored on Exits?
9. In Queuing/Dequeuing Should Any Value Be Decremented/Incremented?
10. Are All Keywords Tested in Macro?
11. Are All Keyword Related Parameters Tested in Service Routine?
12. Are Queues Being Held in Isolation So That Subsequent Interrupting Requestors Are Receiving Spurious Returns Regarding the Held Queue?
13. Should any Registers Be Saved on Entry?
14. Are All Increment Counts Properly Initialized (0 or 1)?

Wrong

1. Are Absolutes Shown Where There Should Be Symbolics?
2. On Comparison of Two Bytes, Should All Bits Be Compared?
3. On Built Data Strings, Should They Be Character or Hex?
4. Are Internal Variables Unique or Confusing If Concatenated?

Extra

1. Are All Blocks Shown in Design Necessary or Are They Extraneous?

4. *Rework*—All errors or problems noted in the inspection report are resolved by the designer or coder/implementor.

5. *Follow-Up*—It is imperative that every issue, concern, and error be entirely resolved at this level, or errors that result can be 10 to 100 times more expensive to fix if found later in the process (programmer time only, machine time not included). It is the responsibility of the moderator to see that all issues, problems, and concerns discovered in the inspection operation have been resolved by the designer in the case of I_1, or the coder/implementor for I_2 inspections. If more than five percent of the material has been reworked, the team should reconvene and carry out a 100 percent reinspection. Where less than five percent of the material has been reworked, the moderator at his discretion may verify the quality of the rework himself or reconvene the team to reinspect either the complete work or just the rework.

commencing inspections

In Operation 3 above, it is one thing to direct people to find errors in design or code. It is quite another problem for them to find errors. Numerous experiences have shown that people have to be taught or prompted to find errors effectively. Therefore, it

Figure 6 Examples of what to examine when looking for errors at I_2

INSPECTION SPECIFICATION

I_2 *Test Branch*
- Is Correct Condition Tested (If X = ON vs. IF X = OFF)?
- Is (Are) Correct Variable(s) Used for Test (If X = ON vs. If Y = ON)?
- Are Null THENs/ELSEs Included as Appropriate?
- Is Each Branch Target Correct?
- Is the Most Frequently Exercised Test Leg the THEN Clause?

I_2 *Interconnection (or Linkage) Calls*
- For Each Interconnection Call to Either a Macro, SVC or Another Module:
- Are All Required Parameters Passed Set Correctly?
- If Register Parameters Are Used, Is the Correct Register Number Specified?
- If Interconnection Is a Macro,
- Does the Inline Expansion Contain All Required Code?
- No Register or Storage Conflicts between Macro and Calling Module?
- If the Interconnection Returns, Do All Returned Parameters Get Processed Correctly?

is prudent to condition them to seek the high-occurrence, high-cost error types (see example in Figures 3 and 4), and then describe the clues that usually betray the presence of each error type (see examples in Figures 5 and 6).

One approach to getting started may be to make a preliminary inspection of a design or code that is felt to be representative of the program to be inspected. Obtain a suitable quantity of errors, and analyze them by type and origin, cause, and salient indicative clues. With this information, an inspection specification may be constructed. This specification can be amended and improved in light of new experience and serve as an on-going directive to focus the attention and conduct of inspection teams. The objective of an inspection specification is to help maximize and make more consistent the error detection efficiency of inspections where

Error detection efficiency

$$= \frac{\text{Errors found by an inspection}}{\text{Total errors in the product before inspection}} \times 100$$

reporting inspection results

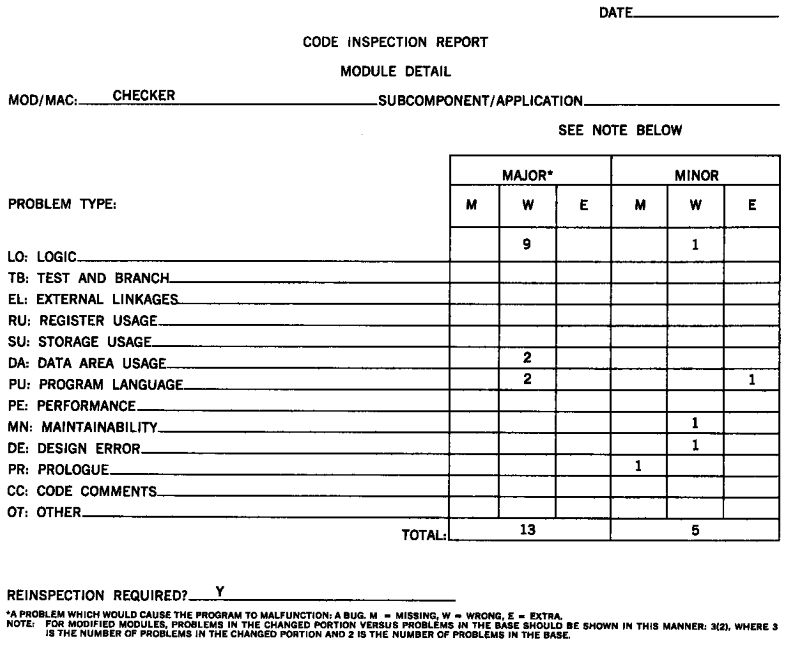
The reporting forms and form completion instructions shown in the Appendix may be used for I_1 and I_2 inspections. Although these forms were constructed for use in systems programming development, they may be used for applications programming development with minor modification to suit particular environments.

The moderator will make hand-written notes recording errors found during inspection meetings. He will categorize the errors

Figure 7A Error list

1. PR/M/MIN	Line 3:	the statement of the prologue in the REMARKS section needs expansion.
2. DA/W/MAJ	Line 123:	ERR–RECORD–TYPE is out of sequence.
3. PU/W/MAJ	Line 147:	the wrong bytes of an 8-byte field (current–data) are moved into the 2-byte field (this year).
4. LO/W/MAJ	Line 169:	while counting the number of leading spaces in NAME, the wrong variable (I) is used to calculate "J".
5. LO/W/MAJ	Line 172:	NAME–CHECK is PERFORMED one time too few.
6. PU/E/MIN	Line 175:	In NAME–CHECK, the check for SPACE is redundant.
7. DE/W/MIN	Line 175:	the design should allow for the occurrence of a period in a last name.

Figure 7B Example of module detail report



DATE__________

CODE INSPECTION REPORT

MODULE DETAIL

MOD/MAC: CHECKER SUBCOMPONENT/APPLICATION__________

SEE NOTE BELOW

PROBLEM TYPE:	MAJOR*			MINOR		
	M	W	E	M	W	E
LO: LOGIC		9			1	
TB: TEST AND BRANCH						
EL: EXTERNAL LINKAGES						
RU: REGISTER USAGE						
SU: STORAGE USAGE						
DA: DATA AREA USAGE		2				
PU: PROGRAM LANGUAGE		2				1
PE: PERFORMANCE						
MN: MAINTAINABILITY					1	
DE: DESIGN ERROR					1	
PR: PROLOGUE				1		
CC: CODE COMMENTS						
OT: OTHER						
TOTAL:		13			5	

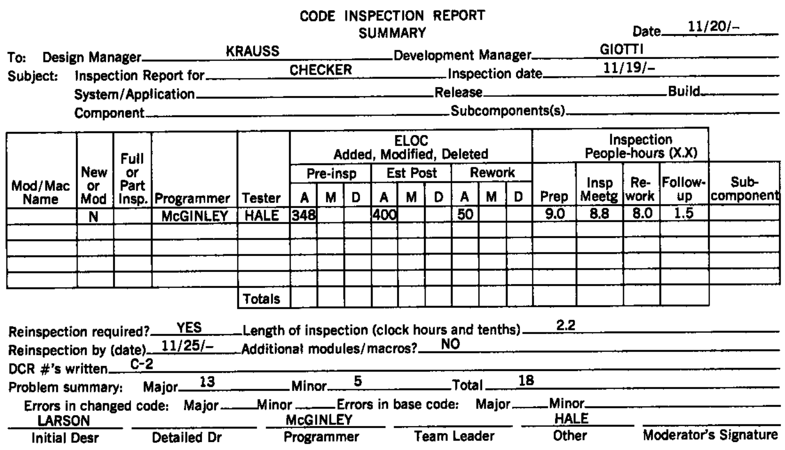
REINSPECTION REQUIRED? Y

*A PROBLEM WHICH WOULD CAUSE THE PROGRAM TO MALFUNCTION: A BUG. M = MISSING, W = WRONG, E = EXTRA.
NOTE: FOR MODIFIED MODULES, PROBLEMS IN THE CHANGED PORTION VERSUS PROBLEMS IN THE BASE SHOULD BE SHOWN IN THIS MANNER: 3(2), WHERE 3 IS THE NUMBER OF PROBLEMS IN THE CHANGED PORTION AND 2 IS THE NUMBER OF PROBLEMS IN THE BASE.

and then transcribe counts of the errors, by type, to the module detail form. By maintaining cumulative totals of the counts by error type, and dividing by the number of projected executable source lines of code inspected to date, he will be able to establish installation averages within a short time.

Figures 7A, 7B, and 7C are an example of a set of code inspection reports. Figure 7A is a partial list of errors found in code inspection. Notice that errors are described in detail and are classified by error type, whether due to something being missing,

Figure 7C Example of code inspection summary report



CODE INSPECTION REPORT
SUMMARY

Date 11/20/-

To: Design Manager KRAUSS Development Manager GIOTTI
Subject: Inspection Report for CHECKER Inspection date 11/19/-
System/Application Release Build
Component Subcomponents(s)

					ELOC Added, Modified, Deleted									Inspection People-hours (X.X)				
Mod/Mac Name	New or Mod	Full or Part Insp.	Programmer	Tester	Pre-insp			Est Post			Rework			Prep	Insp Meetg	Re-work	Follow-up	Sub-component
					A	M	D	A	M	D	A	M	D					
	N		McGINLEY	HALE	348			400			50			9.0	8.8	8.0	1.5	
				Totals														

Reinspection required? YES Length of inspection (clock hours and tenths) 2.2
Reinspection by (date) 11/25/- Additional modules/macros? NO
DCR #'s written C-2
Problem summary: Major 13 Minor 5 Total 18
Errors in changed code: Major Minor Errors in base code: Major Minor

LARSON		McGINLEY		HALE	
Initial Desr	Detailed Dr	Programmer	Team Leader	Other	Moderator's Signature

wrong, or extra as the cause, and according to major or minor severity. Figure 7B is a module level summary of the errors contained in the entire error list represented by Figure 7A. The code inspection summary report in Figure 7C is a summary of inspection results obtained on all modules inspected in a particular inspection session or in a subcomponent or application.

Inspections and languages

Inspections have been successfully applied to designs that are specified in English prose, flowcharts, HIPO, (Hierarchy plus Input-Process-Output) and PIDGEON (an English prose-like meta language).

The first code inspections were conducted on PL/S and Assembler. Now, prompting checklists for inspections of Assembler, COBOL, FORTRAN, and PL/1 code are available.[7]

personnel considerations

One of the most significant benefits of inspections is the detailed feedback of results on a relatively real-time basis. The programmer finds out what error types he is most prone to make and their quantity and how to find them. This feedback takes place within a few days of writing the program. Because he gets early indications from the first few units of his work inspected, he is able to show improvement, and usually does, on later work even during the same project. In this way, feedback of results from inspections must be counted for the programmer's use and benefit: *they should not under any circumstances be used for programmer performance appraisal.*

Skeptics may argue that once inspection results are obtained, they will or even must count in performance appraisals, or at

Figure 8 Example of most error-prone modules based on I_1 and I_2

Module name	*Number of errors*	*Lines of code*	*Error density, Errors/K. Loc*
Echo	4	128	31
Zulu	10	323	31
Foxtrot	3	71	28
Alpha	7	264	27←Average
Lima	2	106	19 Error
Delta	3	195	15 Rate
⋮	⋮	⋮	⋮
	67		

least cause strong bias in the appraisal process. The author can offer in response that inspections have been conducted over the past three years involving diverse projects and locations, hundreds of experienced programmers and tens of managers, and so far he has found no case in which inspection results have been used negatively against programmers. Evidently no manager has tried to "kill the goose that lays the golden eggs."

A preinspection opinion of some programmers is that they do not see the value of inspections because they have managed very well up to now, or because their projects are too small or somehow different. This opinion usually changes after a few inspections to a position of acceptance. The quality of acceptance is related to the success of the inspections they have experienced, the *conduct of the trained moderator*, and the *attitude demonstrated by management.* The acceptance of inspections by programmers and managers as a beneficial step in making programs is well-established amongst those who have tried them.

Process control using inspection and testing results

Obviously, the range of analysis possible using inspection results is enormous. Therefore, only a few aspects will be treated here, and they are elementary expositions.

most error-prone modules

A listing of either I_1, I_2, or combined $I_1 + I_2$ data as in Figure 8 immediately highlights which modules contained the highest error density on inspection. If the error detection efficiency of each of the inspections was fairly constant, the ranking of error-prone modules holds. Thus if the error detection efficiency of inspection is 50 percent, and the inspection found 10 errors in a

Figure 9 Example of distribution of error types

	Number of errors	*%*	*Normal/usual distribution, %*
Logic	23	35	44
Interconnection/Linkage (Internal)	21	31 ?	18
Control Blocks	6	9	13
–	·	8	10
–	·	7	7
–	·	6	6
–	·	4	2
		100%	100%

module, then it can be estimated that there are 10 errors remaining in the module. This information can prompt many actions to control the process. For instance, in Figure 8, it may be decided to reinspect module "Echo" or to redesign and recode it entirely. Or, less drastically, it may be decided to test it "harder" than other modules and look especially for errors of the type found in the inspections.

distribution of error types

If a ranked distribution of error types is obtained for a group of "error-prone modules" (Figure 9), which were produced from the same Process A, for example, it is a short step to comparing this distribution with a "Normal/Usual Percentage Distribution." Large disparities between the sample and "standard" will lead to questions on why Process A, say, yields nearly twice as many internal interconnection errors as the "standard" process. If this analysis is done promptly on the first five percent of production, it may be possible to remedy the problem (if it is a problem) on the remaining 95 percent of modules for a particular shipment. Provision can be made to test the first five percent of the modules to remove the unusually high incidence of internal interconnection problems.

inspecting error-prone code

Analysis of the testing results, commencing as soon as testing errors are evident, is a vital step in controlling the process since future testing can be guided by early results.

Where testing reveals excessively error-prone code, it may be more economical and saving of schedule to select the most error-prone code and inspect it before continuing testing. (The business case will likely differ from project to project and case to case, but in many instances inspection will be indicated). The selection of the most error-prone code may be made with two considerations uppermost:

Table 4. Inspection and walk-through processes and objectives

Inspection		*Walk-through*	
Process Operations	*Objectives*	*Process Operations*	*Objectives*
1. Overview	Education (Group)	–	–
2. Preparation	Education (Individual)	1. Preparation	Education (Individual)
3. Inspection	Find errors! (Group)	2. Walk-through	Education (Group) Discuss design alternatives Find errors
4. Rework	Fix problems	–	
5. Follow-up	Ensure all fixes correctly installed	–	

Note the separation of objectives in the inspection process.

Table 5 Comparison of key properties of inspections and walk-throughs

Properties	*Inspection*	*Walk-Through*
1. Formal moderator training	Yes	No
2. Definite participant roles	Yes	No
3. Who "drives" the inspection or walk-through	Moderator	Owner of material (Designer or coder)
4. Use "How To Find Errors" checklists	Yes	No
5. Use distribution of error types to look for	Yes	No
6. Follow-up to reduce bad fixes	Yes	No
7. Less future errors because of detailed error feedback to individual programmer	Yes	Incidental
8. Improve inspection efficiency from analysis of results	Yes	No
9. Analysis of data → process problems → improvements	Yes	No

1. Which modules head a ranked list when the modules are rated by test errors per K.NCSS?
2. In the parts of the program in which test coverage is low, which modules or parts of modules are most suspect based on $(I_1 + I_2)$ errors per K.NCSS and programmer judgment?

From a condensed table of ranked "most error-prone" modules, a selection of modules to be inspected (or reinspected) may be made. Knowledge of the error types already found in these modules will better prepare an inspection team.

The reinspection itself should conform with the I_2 process, except that an overview may be necessary if the original overview was held too long ago or if new project members are involved.

Inspections and walk-throughs

Walk-throughs (or walk-thrus) are practiced in many different ways in different places, with varying regularity and thoroughness. This inconsistency causes the results of walk-throughs to vary widely and to be nonrepeatable. Inspections, however, having an established process and a formal procedure, tend to vary less and produce more repeatable results. Because of the variation in walk-throughs, a comparison between them and inspections is not simple. However, from Reference 8 and the walk-through procedures witnessed by the author and described to him by walk-through participants, as well as the inspection process described previously and in References 1 and 9, the comparison in Tables 4 and 5 is drawn.

effects on development process

Figure 10A describes the process in which a walk-through is applied. Clearly, the purging of errors from the product as it passes through the walk-through between Operations 1 and 2 is very beneficial to the product. In Figure 10B, the inspection process (and its feedback, feed-forward, and self-improvement) replaces the walk-through. The notes on the figure are self-explanatory.

Inspections are also an excellent means of measuring completeness of work against the exit criteria which must be satisfied to complete project checkpoints. (Each checkpoint should have a clearly defined set of exit criteria. Without exit criteria, a checkpoint is too negotiable to be useful for process control).

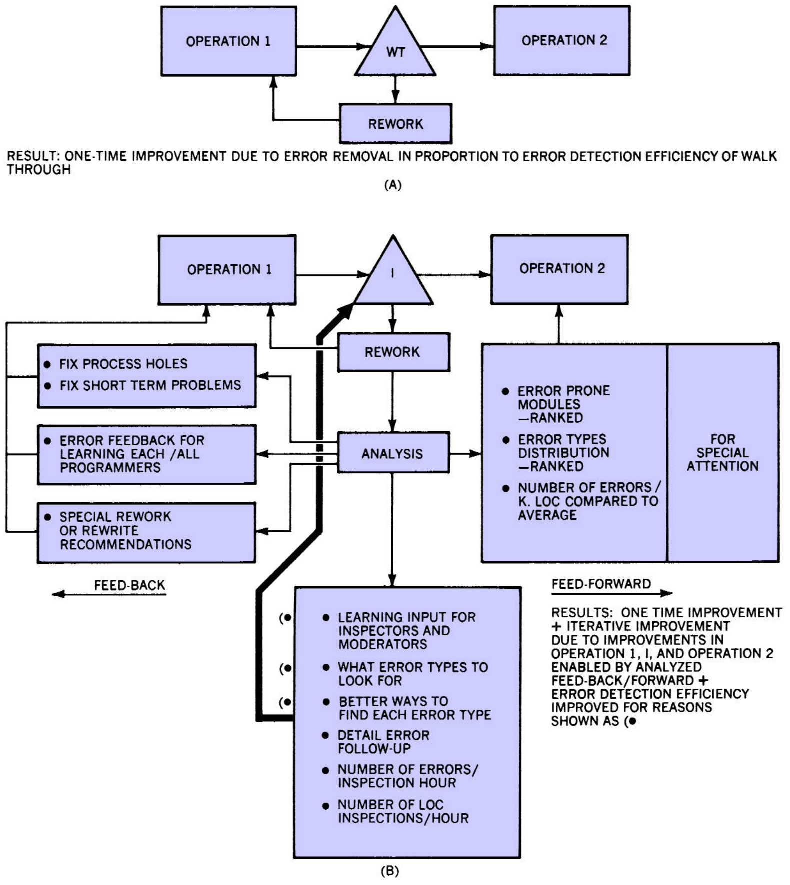
Inspections and process management

The most marked effects of inspections on the development process is to change the old adage that, "design is not complete until testing is completed," to a position where a very great deal must be known about the design before even the coding is begun. Although great discretion is still required in code implementation, more predictability and improvements in schedule, cost, and quality accrue. The old adage still holds true if one regards inspection as much a means of verification as testing.

percent of errors found

Observations in one case in systems programming show that approximately two thirds of all errors reported during development are found by I_1 and I_2 inspections prior to machine testing.

Figure 10 (A) Walk-through process, (B) Inspection process


OPERATION 1
WT
OPERATION 2
REWORK
RESULT: ONE-TIME IMPROVEMENT DUE TO ERROR REMOVAL IN PROPORTION TO ERROR DETECTION EFFICIENCY OF WALK THROUGH
(A)
OPERATION 1
I
OPERATION 2
REWORK
• FIX PROCESS HOLES
• FIX SHORT TERM PROBLEMS
• ERROR FEEDBACK FOR LEARNING EACH /ALL PROGRAMMERS
• SPECIAL REWORK OR REWRITE RECOMMENDATIONS
ANALYSIS
• ERROR PRONE MODULES —RANKED
• ERROR TYPES DISTRIBUTION —RANKED
• NUMBER OF ERRORS / K. LOC COMPARED TO AVERAGE
FOR SPECIAL ATTENTION
FEED-BACK
FEED-FORWARD
(• LEARNING INPUT FOR INSPECTORS AND MODERATORS
(• WHAT ERROR TYPES TO LOOK FOR
(• BETTER WAYS TO FIND EACH ERROR TYPE
• DETAIL ERROR FOLLOW-UP
• NUMBER OF ERRORS/ INSPECTION HOUR
• NUMBER OF LOC INSPECTIONS/HOUR
RESULTS: ONE TIME IMPROVEMENT + ITERATIVE IMPROVEMENT DUE TO IMPROVEMENTS IN OPERATION 1, I, AND OPERATION 2 ENABLED BY ANALYZED FEED-BACK/FORWARD + ERROR DETECTION EFFICIENCY IMPROVED FOR REASONS SHOWN AS (•
(B)

The error detection efficiencies of the I_1 and I_2 inspections separately are, of course, less than 66 percent. A similar observation of an application program development indicated an 82 percent find (Table 1). As more is learned and the error detection efficiency of inspection is increased, the burden of debugging on testing operations will be reduced, and testing will be more able to fulfill its prime objective of verifying quality.

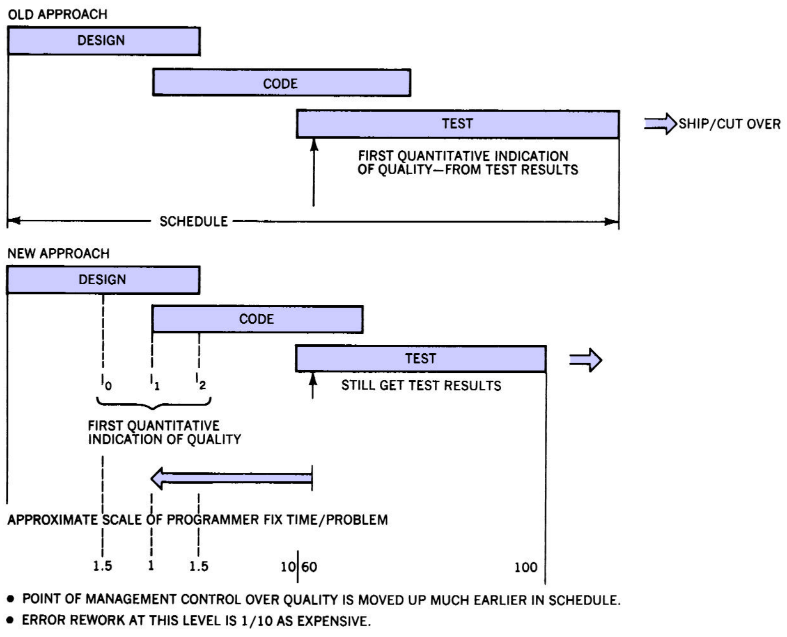
effect on cost and schedule

Comparing the "old" and "new" (with inspections) approaches to process management in Figure 11, we can see clearly that with the use of inspection results, error rework (which is a very significant variable in product cost) tends to be managed more during the first half of the schedule. This results in much lower cost than in the "old" approach, where the cost of error rework was 10 to 100 times higher and was accomplished in large part during the last half of the schedule.

process tracking

Inserting the I_1 and I_2 checkpoints in the development process enables assessment of project completeness and quality to be

Figure 11 Effect of inspection on process management


OLD APPROACH
DESIGN
CODE
TEST
SHIP/CUT OVER
FIRST QUANTITATIVE INDICATION OF QUALITY—FROM TEST RESULTS
SCHEDULE
NEW APPROACH
DESIGN
CODE
TEST
I0 I1 I2
STILL GET TEST RESULTS
FIRST QUANTITATIVE INDICATION OF QUALITY
APPROXIMATE SCALE OF PROGRAMMER FIX TIME/PROBLEM
1.5 1 1.5 10 60 100
• POINT OF MANAGEMENT CONTROL OVER QUALITY IS MOVED UP MUCH EARLIER IN SCHEDULE.
• ERROR REWORK AT THIS LEVEL IS 1/10 AS EXPENSIVE.

made early in the process (during the first half of the project instead of the latter half of the schedule, when recovery may be impossible without adjustments in schedule and cost). Since individually trackable modules of reasonably well-known size can be counted as they pass through each of these checkpoints, the percentage completion of the project against schedule can be continuously and easily tracked.

effect on product knowledge

The overview, preparation, and inspection sequence of the operations of the inspection process give the inspection participants a high degree of product knowledge in a very short time. This important side benefit results in the participants being able to handle later development and testing with more certainty and less false starts. Naturally, this also contributes to productivity improvement.

An interesting sidelight is that because designers are asked at pre-I_1 inspection time for estimates of the number of lines of code (NCSS) that their designs will create, and they are present to count for themselves the actual lines of code at the I_2 inspection, the accuracy of design estimates has shown substantial improvement.

For this reason, an inspection is frequently a required event where responsibility for design or code is being transferred from one programmer to another. The complete inspection team is convened for such an inspection. (One-on-one reviews such as desk debugging are certainly worthwhile but do not approach the effectiveness of formal inspection.) Usually the side benefit of finding errors more than justifies the transfer inspection.

inspecting modified code

Code that is changed in, or inserted in, an existing module either in replacement of deleted code or simply inserted in the module is considered modified code. By this definition, a very large part of programming effort is devoted to modifying code. (The addition of entirely new modules to a system count as new, not modified, code.)

Some observations of errors per K.NCSS of modified code show its error rate to be considerably higher than is found in new code; (i.e., if 10.NCSS are replaced in a 100.NCSS module and errors against the 10.NCSS are counted, the error rate is described as number of errors per 10.NCSS, not number of errors per 100.NCSS). Obviously, if the number of errors in modified code are used to derive an error rate per K.NCSS for the whole module that was modified, this rate would be largely dependent upon the percentage of the module that is modified: this would provide a meaningless ratio. A useful measure is the number of errors per K.NCSS (modified) in which the higher error rates have been observed.

Since most modifications are small (e.g., 1 to 25 instructions), they are often erroneously regarded as trivially simple and are handled accordingly; the error rate goes up, and control is lost. In the author's experience, *all* modifications are well worth inspecting from an economic and a quality standpoint. A convenient method of handling changes is to group them to a module or set of modules and convene the inspection team to inspect as many changes as possible. But all changes must be inspected!

Inspections of modifications can range from inspecting the modified instructions and the surrounding instructions connecting it with its host module, to an inspection of the entire module. The choice of extent of inspection coverage is dependent upon the percentage of modification, pervasiveness of the modification, etc.

bad fixes

A very serious problem is the inclusion in the product of bad fixes. Human tendency is to consider the "fix," or correction, to a problem to be error-free itself. Unfortunately, this is all too frequently untrue in the case of fixes to errors found by inspections and by testing. The inspection process clearly has an operation called Follow-Up to try and minimize the bad-fix problem, but the fix process of testing errors very rarely requires scrutiny of fix quality before the fix is inserted. Then, if the fix is bad, the whole elaborate process of going from source fix to link edit, to test the fix, to regression test must be repeated at needlessly high cost. The number of bad fixes can be economically reduced by some simple inspection after clean compilation of the fix.

Summary

We can summarize the discussion of design and code inspections and process control in developing programs as follows:

1. Describe the program development process in terms of operations, and define exit criteria which must be satisfied for completion of each operation.
2. Separate the objectives of the inspection process operations to keep the inspection team focused on one objective at a time:

Operation	*Objective*
Overview	Communications/education
Preparation	Education
Inspection	Find errors
Rework	Fix errors
Follow-up	Ensure all fixes are applied correctly

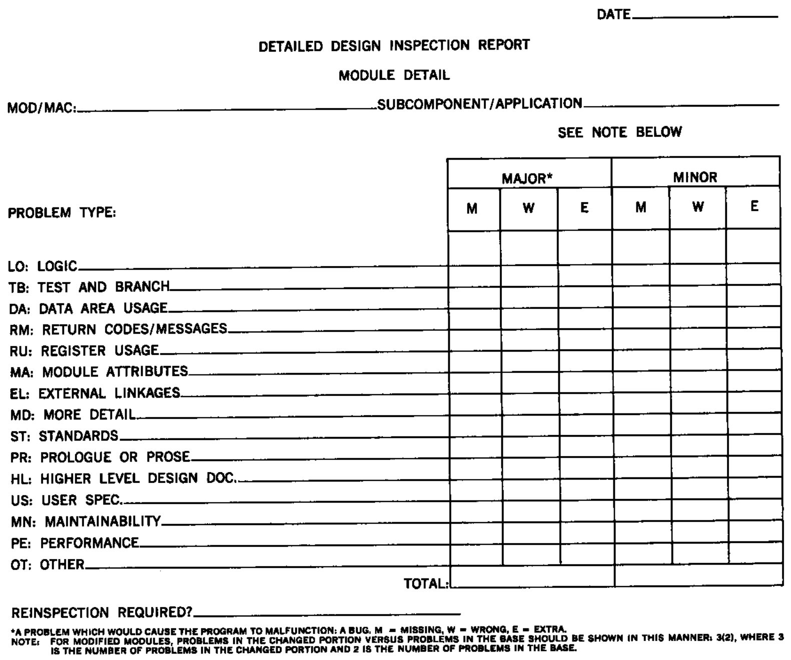
3. Classify errors by type, and rank frequency of occurrence of types. Identify *which types* to spend most time looking for in the inspection.
4. Describe *how* to look for presence of error types.
5. Analyze inspection results and use for constant process improvement (until process averages are reached and then use for process control).

Some applications of inspections include function level inspections I_0, design-complete inspections I_1, code inspections I_2, test plan inspections IT_1, test case inspections IT_2, interconnections inspections IF, inspection of fixes/changes, inspection of publications, etc., and post testing inspection. Inspections can be applied to the development of system control programs, applications programs, and microcode in hardware.

We can conclude from experience that inspections increase productivity and improve final program quality. Furthermore, improvements in process control and project management are enabled by inspections.

ACKNOWLEDGMENTS
The author acknowledges, with thanks, the work of Mr. O. R. Kohli and Mr. R. A. Radice, who made considerable contributions in the development of inspection techniques applied to program design and code, and Mr. R. R. Larson, who adapted inspections to program testing.

Figure 12 Design inspection module detail form



DATE__________

DETAILED DESIGN INSPECTION REPORT

MODULE DETAIL

MOD/MAC:__________ SUBCOMPONENT/APPLICATION__________

SEE NOTE BELOW

	MAJOR*			MINOR		
PROBLEM TYPE:	M	W	E	M	W	E
LO: LOGIC						
TB: TEST AND BRANCH						
DA: DATA AREA USAGE						
RM: RETURN CODES/MESSAGES						
RU: REGISTER USAGE						
MA: MODULE ATTRIBUTES						
EL: EXTERNAL LINKAGES						
MD: MORE DETAIL						
ST: STANDARDS						
PR: PROLOGUE OR PROSE						
HL: HIGHER LEVEL DESIGN DOC.						
US: USER SPEC.						
MN: MAINTAINABILITY						
PE: PERFORMANCE						
OT: OTHER						
TOTAL:						

REINSPECTION REQUIRED?__________

*A PROBLEM WHICH WOULD CAUSE THE PROGRAM TO MALFUNCTION: A BUG. M = MISSING, W = WRONG, E = EXTRA.
NOTE: FOR MODIFIED MODULES, PROBLEMS IN THE CHANGED PORTION VERSUS PROBLEMS IN THE BASE SHOULD BE SHOWN IN THIS MANNER: 3(2), WHERE 3 IS THE NUMBER OF PROBLEMS IN THE CHANGED PORTION AND 2 IS THE NUMBER OF PROBLEMS IN THE BASE.

CITED REFERENCES AND FOOTNOTES

1. O. R. Kohli, *High-Level Design Inspection Specification*, Technical Report TR 21.601, IBM Corporation, Kingston, New York (July 21, 1975).
2. It should be noted that the exit criteria for I_1 (design complete where one design statement is estimated to represent 3 to 10 code instructions) and I_2 (first clean code compilations) are checkpoints in the development process through which every programming project must pass.
3. The Hawthorne Effect is a psychological phenomenon usually experienced in human-involved productivity studies. The effect is manifested by participants producing above normal because they know they are being studied.
4. NCSS (Non-Commentary Source Statements), also referred to as "Lines of Code," are the sum of executable code instructions and declaratives. Instructions that invoke macros are counted once only. Expanded macroinstructions are also counted only once. Comments are not included.
5. Basically in a walk-through, program design or code is reviewed by a group of people gathered together at a structured meeting in which errors/issues pertaining to the material and proposed by the participants may be discussed in an effort to find errors. The group may consist of various participants but always includes the originator of the material being reviewed who usually plans the meeting and is responsible for correcting the errors. How it differs from an inspection is pointed out in Tables 2 and 3.
6. *Marketing Newsletter*, Cross Application Systems Marketing, "Program inspections at Aetna," MS-76-006, S2, IBM Corporation, Data Processing Division, White Plains, New York (March 29, 1976).

7. J. Ascoly, M. J. Cafferty, S. J. Gruen, and O. R. Kohli, *Code Inspection Specification*, Technical Report TR 21.630, IBM Corporation, Kingston, New York (1976).
8. N. S. Waldstein, *The Walk-Thru—A Method of Specification, Design and Review*, Technical Report TR 00.2536, IBM Corporation, Poughkeepsie, New York (June 4, 1974).
9. Independent study programs: *IBM Structured Programming Textbook*, SR20-7149-1, *IBM Structured Programming Workbook*, SR20-7150-0, IBM Corporation, Data Processing Division, White Plains, New York.

GENERAL REFERENCES

1. J. D. Aron, *The Program Development Process: Part 1: The Individual Programmer*, Structured Programs, 137–141, Addison-Wesley Publishing Co., Reading, Massachusetts (1974).
2. M. E. Fagan, *Design and Code Inspections and Process Control in the Development of Programs*, Technical Report TR 00.2763, IBM Corporation, Poughkeepsie, New York (June 10, 1976). This report is a revision of the author's *Design and Code Inspections and Process Control in the Development of Programs*, Technical Report TR 21.572, IBM Corporation, Kingston, New York (December 17, 1974).
3. O. R. Kohli and R. A. Radice, *Low-Level Design Inspection Specification*, Technical Report TR 21.629. IBM Corporation, Kingston, New York (1976).
4. R. R. Larson, *Test Plan and Test Case Inspection Specifications*, Technical Report TR 21.586, IBM Corporation, Kingston, New York (April 4, 1975).

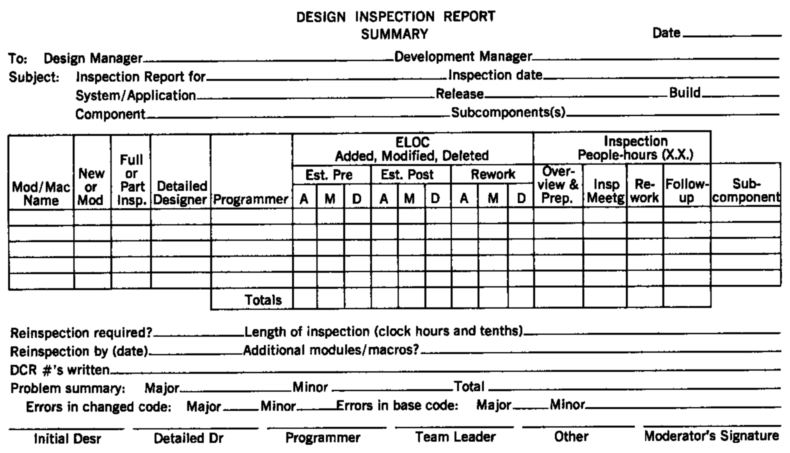
Appendix: Reporting forms and form completion instructions

Instructions for Completing Design Inspection Module Detail Form

This form (Figure 12) should be completed for each module/ macro that has valid problems against it. The problem-type information gathered in this report is important because a history of problem-type experience points out high-occurrence types. This knowledge can then be conveyed to inspectors so that they can concentrate on seeking the higher-occurrence types of problems.

1. MOD/MAC: The module or macro name.
2. SUBCOMPONENT: The associated subcomponent.
3. PROBLEM TYPE: Summarize the number of problems by type (logic, etc.), severity (major/minor), and by category (missing, wrong, or extra). For modified modules, detail the number of problems in the changed design versus the number in the base design. (Problem types were developed in a systems programming environment. Appropriate changes, if desired, could be made for application development.)

Figure 13 Design inspection summary form



DESIGN INSPECTION REPORT
SUMMARY

Date________

To: Design Manager________ Development Manager________
Subject: Inspection Report for________ Inspection date________
System/Application________ Release________ Build________
Component________ Subcomponents(s)________

					ELOC Added, Modified, Deleted									Inspection People-hours (X.X.)				
					Est. Pre			Est. Post			Rework							
Mod/Mac Name	New or Mod	Full or Part Insp.	Detailed Designer	Programmer	A	M	D	A	M	D	A	M	D	Over-view & Prep.	Insp Meetg	Re-work	Follow-up	Sub-component
				Totals														

Reinspection required?________ Length of inspection (clock hours and tenths)________
Reinspection by (date)________ Additional modules/macros?________
DCR #'s written________
Problem summary: Major________ Minor________ Total________
Errors in changed code: Major____ Minor____ Errors in base code: Major____ Minor________

Initial Desr   Detailed Dr   Programmer   Team Leader   Other   Moderator's Signature

4. REINSPECTION REQUIRED?: Indicate whether the module/macro requires a reinspection.

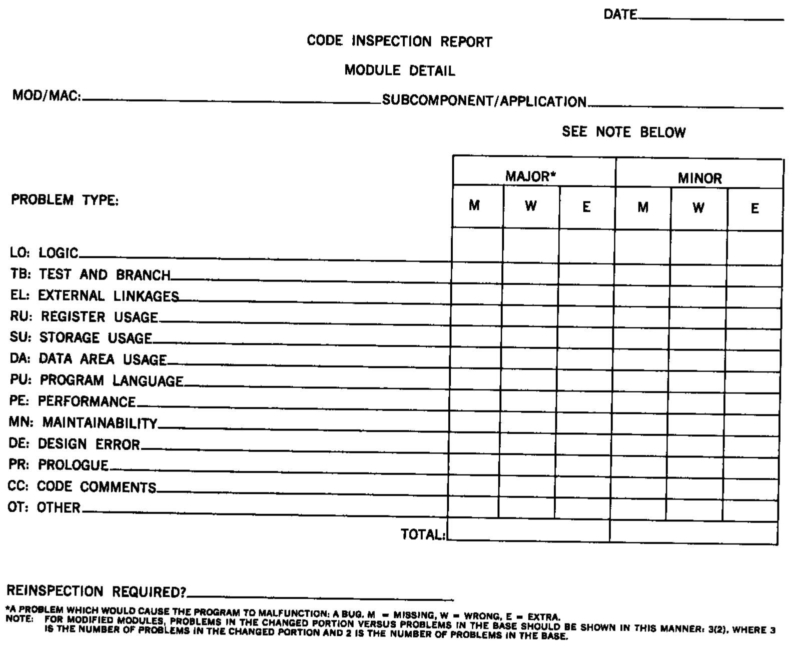
All valid problems found in the inspection should be listed and attached to the report. A brief description of each problem, its error type, and the rework time to fix it should be given (see Figure 7A, which describes errors in similar detail to that required but is at a coding level).

Instructions for Completing Design Inspection Summary Form

Following are detailed' instructions for completing the form in Figure 13.

1. TO: The report is addressed to the respective design and development managers.
2. SUBJECT: The unit being inspected is identified.
3. MOD/MAC NAME: The name of each module and macro as it resides on the source library.
4. NEW OR MOD: "N" if the module is new; "M" if the module is modified.
5. FULL OR PART INSP: If the module/macro is "modified," indicate "F" if the module/macro was fully inspected or "P" if partially inspected.
6. DETAILED DESIGNER: and PROGRAMMER: Identification of originators.
7. PRE-INSP EST ELOC: The estimated executable source lines of code (added, modified, deleted). Estimate made prior to the inspection by the designer.

Figure 14 Code inspection module detail form



DATE______________

CODE INSPECTION REPORT

MODULE DETAIL

MOD/MAC:____________________SUBCOMPONENT/APPLICATION______________

SEE NOTE BELOW

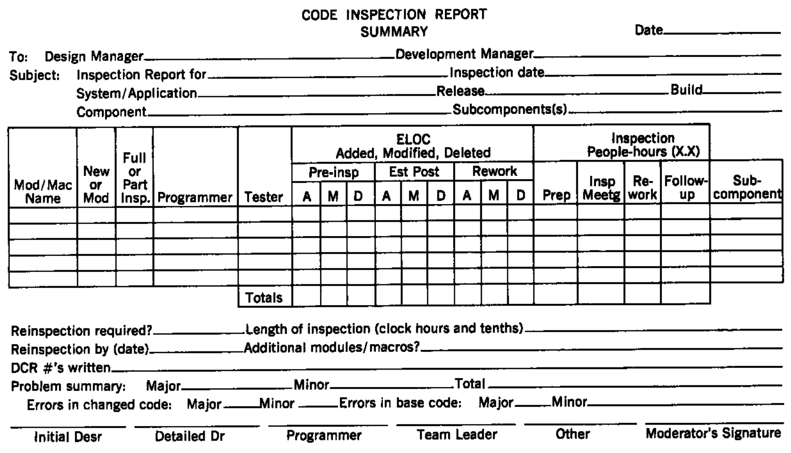
	MAJOR*			MINOR		
PROBLEM TYPE:	M	W	E	M	W	E
LO: LOGIC						
TB: TEST AND BRANCH						
EL: EXTERNAL LINKAGES						
RU: REGISTER USAGE						
SU: STORAGE USAGE						
DA: DATA AREA USAGE						
PU: PROGRAM LANGUAGE						
PE: PERFORMANCE						
MN: MAINTAINABILITY						
DE: DESIGN ERROR						
PR: PROLOGUE						
CC: CODE COMMENTS						
OT: OTHER						
TOTAL:						

REINSPECTION REQUIRED?______________

*A PROBLEM WHICH WOULD CAUSE THE PROGRAM TO MALFUNCTION; A BUG. M = MISSING, W = WRONG, E = EXTRA.
NOTE: FOR MODIFIED MODULES, PROBLEMS IN THE CHANGED PORTION VERSUS PROBLEMS IN THE BASE SHOULD BE SHOWN IN THIS MANNER: 3(2), WHERE 3 IS THE NUMBER OF PROBLEMS IN THE CHANGED PORTION AND 2 IS THE NUMBER OF PROBLEMS IN THE BASE.

8. POST-INSP EST ELOC: The estimated executable source lines of code. Estimate made after the inspection.
9. REWORK ELOC: The estimated executable source lines of code in rework as a result of the inspection.
10. OVERVIEW AND PREP: The number of people-hours (in tenths of hours) spent in preparing for the overview, in the overview meeting itself, and in preparing for the inspection meeting.
11. INSPECTION MEETING: The number of people-hours spent on the inspection meeting.
12. REWORK: The estimated number of people-hours spent to fix the problems found during the inspection.
13. FOLLOW-UP: The estimated number of people-hours spent by the moderator (and others if necessary) in verifying the correctness of changes made by the author as a result of the inspection.
14. SUBCOMPONENT: The subcomponent of which the module/macro is a part.
15. REINSPECTION REQUIRED?: Yes or no.
16. LENGTH OF INSPECTION: Clock hours spent in the inspection meeting.
17. REINSPECTION BY (DATE): Latest acceptable date for reinspection.

Figure 15 Code inspection summary form



CODE INSPECTION REPORT
SUMMARY

Date__________

To: Design Manager__________ Development Manager__________

Subject: Inspection Report for__________ Inspection date__________

System/Application__________ Release__________ Build__________

Component__________ Subcomponents(s)__________

					ELOC Added, Modified, Deleted									Inspection People-hours (X.X)				
					Pre-insp			Est Post			Rework							
Mod/Mac Name	New or Mod	Full or Part Insp.	Programmer	Tester	A	M	D	A	M	D	A	M	D	Prep	Insp Meetg	Re-work	Follow-up	Sub-component
				Totals														

Reinspection required?__________ Length of inspection (clock hours and tenths)__________

Reinspection by (date)__________ Additional modules/macros?__________

DCR #'s written__________

Problem summary: Major__________ Minor__________ Total__________

Errors in changed code: Major_____ Minor_____ Errors in base code: Major_____ Minor_____

Initial Desr | Detailed Dr | Programmer | Team Leader | Other | Moderator's Signature

18. ADDITIONAL MODULES/MACROS?: For these subcomponents, are additional modules/macros yet to be inspected?
19. DCR #'S WRITTEN: The identification of Design Change Requests, DCR(s), written to cover problems in rework.
20. PROBLEM SUMMARY: Totals taken from Module Detail forms(s).
21. INITIAL DESIGNER, DETAILED DESIGNER, etc.: Identification of members of the inspection team.

Instructions for Completing Code Inspection Module Detail Form

This form (Figure 14) should be completed according to the instructions for completing the design inspection module detail form.

Instructions for Completing Code Inspection Summary Form

This form (Figure 15) should be completed according to the instructions for the design inspection summary form except for the following items.

1. PROGRAMMER AND TESTER: Identifications of original participants involved with code.
2. PRE-INSP. ELOC: The noncommentary source lines of code (added, modified, deleted). Count made prior to the inspection by the programmer.
3. POST-INSP EST ELOC: The estimated noncommentary source lines of code. Estimate made after the inspection.

4. REWORK ELOC: The estimated noncommentary source lines of code in rework as a result of the inspection.
5. PREP: The number of people hours (in tenths of hours) spent in preparing for the inspection meeting.

Michael E. Fagan

Data Processing Product Group, Poughkeepsie, New York.

Currently he is manager of the Programming Methodology Department in the Poughkeepsie Laboratory. He originated the use of inspections for program development.