Advances in Software Inspections

MICHAEL E. FAGAN, MEMBER, IEEE

***Abstract*—This paper presents new studies and experiences that enhance the use of the inspection process and improve its contribution to development of defect-free software on time and at lower costs. Examples of benefits are cited followed by descriptions of the process and some methods of obtaining the enhanced results.**

Software inspection is a method of static testing to verify that software meets its requirements. It engages the developers and others in a formal process of investigation that usually detects more defects in the product—and at lower cost—than does machine testing. Users of the method report very significant improvements in quality that are accompanied by lower development costs and greatly reduced maintenance efforts. Excellent results have been obtained by small and large organizations in all aspects of new development as well as in maintenance. There is some evidence that developers who participate in the inspection of their own product actually create fewer defects in future work. Because inspections formalize the development process, productivity and quality enhancing tools can be adopted more easily and rapidly.

***Index Terms*—Defect detection, inspection, project management, quality assurance, software development, software engineering, software quality, testing, walkthru.**

Introduction

THE software inspection process was created in 1972, in IBM Kingston, NY, for the dual purposes of improving software quality and increasing programmer productivity. Its accelerating rate of adoption throughout the software development and maintenance industry is an acknowledgment of its effectiveness in meeting its goals. Outlined in this paper are some enhancements to the inspection process, and the experiences of some of the many companies and organizations that have contributed to its evolution. The author is indebted to and thanks the many people who have given their help so liberally.

Because of the clear structure the inspection process has brought to the development process, it has enabled study of both itself and the conduct of development. The latter has enabled process control to be applied from the point at which the requirements are inspected—a much earlier point in the process than ever before—and throughout development. Inspections provide data on the performance of individual development operations, thus providing a unique opportunity to evaluate new tools and techniques. At the same time, studies of inspections have isolated and fostered improvement of its key characteristics such that very high defect detection efficiency inspections may now be conducted *routinely*. This simultaneous study of development and design and code inspections prompted the adaptation of the principles of the inspection process to inspections of requirements, user information, and documentation, and test plans and test cases. In each instance, the new uses of inspection were found to improve product quality and to be cost effective, i.e., it saved more than it cost. Thus, as the effectiveness of inspections are improving, they are being applied in many new and different ways to improve software quality and reduce costs.

Manuscript received September 30, 1985.
The author is with the IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598.
IEEE Log Number 8608192.

Benefits: Defect Reduction, Defect Prevention, and Cost Improvement

In March 1984, while addressing the IBM SHARE User Group on software service, L. H. Fenton, IBM Director of VM Programming Systems, made an important statement on quality improvement due to inspections [1]:

> "Our goal is to provide defect free products and product information, and we believe the best way to do this is by refining and enhancing our existing software development process.
>
> Since we introduced the inspection process in 1974, we have achieved significant improvements in quality. IBM has nearly doubled the number of lines of code shipped for System/370 software products since 1976, while the number of defects per thousand lines of code has been reduced by *two-thirds*. Feedback from early MVS/XA and VM/SP Release 3 users indicates these products met and, in many cases, exceeded our ever increasing quality expectations."

Observation of a small sample of programmers suggested that early experience gained from inspections caused programmers to reduce the number of defects that were injected in the design and code of programs created later during the same project [3]. Preliminary analysis of a much larger study of data from recent inspections is providing similar results.

It should be noted that the improvements reported by IBM were made while many of the enhancements to inspections that are mentioned here were being developed. As these improvements are incorporated into everyday practice, it is probable that inspections will help bring further reductions in defect injection and detection rates.

Additional reports showing that inspections improve quality *and* reduce costs follow. (In all these cases, the cost of inspections is included in project cost. Typically, all design and code inspection costs amount to 15 percent of project cost.)

0098-5589/86/0700-0744$01.00 © 1986 IEEE

AETNA Life and Casualty. 4439 LOC [2]	—0 Defects in use. —25 percent reduction in development resource.
IBM RESPOND, U.K. 6271 LOC [3]	—0 Defects in use. —9 percent reduction in cost compared to walkthrus.
Standard Bank of South Africa. 143 000 LOC [4]	—0.15 Defects/KLOC in use. —95 percent reduction in corrective maintenance cost.
American Express, System code). 13 000 LOC	—0.3 Defects in use.

In the AETNA and IBM examples, inspections found 82 and 93 percent, respectively, of all defects (that would cause malfunction) detected over the life cycle of the products. The other two cases each found over 50 percent of all defects by inspection. While the Standard Bank of South Africa and American Express were unable to use trained inspection moderators, and the former conducted only code inspections, both obtained outstanding results. The tremendous reduction in corrective maintenance at the Standard Bank of South Africa would also bring impressive savings in life cycle costs.

Naturally, reduction in maintenance allows redirection of programmers to work off the application backlog, which is reputed to contain at least two years of work at most locations. Impressive cost savings and quality improvements have been realized by inspecting test plans and then the test cases that implement those test plans. For a product of about 20 000 LOC, R. Larson [5] reported that test inspections resulted in:

- modification of approximately 30 percent of the functional matrices representing test coverage,
- detection of 176 major defects in the test plans and test cases (i.e., in 176 instances testing would have missed testing critical function or tested it incorrectly), and
- savings of more than 85 percent in programmer time by detecting the major defects by inspection as opposed to finding them during functional variation testing.

There are those who would use inspections whether or not they are cost justified for defect removal because of the nonquantifiable benefits the technique supplies toward improving the service provided to users and toward creating a more professional application development environment [6].

Experience has shown that inspections have the effect of slightly front-end loading the committment of people resources in development, adding to requirements and design, while greatly reducing the effort required during testing and for rework of design and code. The result is an overall *net* reduction in development resource, and usually in schedule too. Fig. 1 is a pictorial description of the familiar "snail" shaped curve of software development resource versus the time schedule including and without inspections.

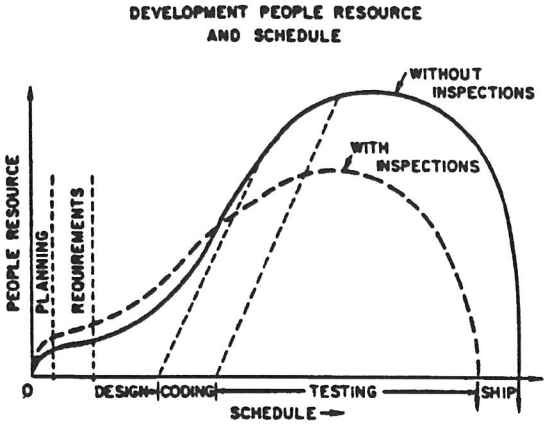

Fig. 1.

The Software Quality Problem

The software quality problem is the result of defects in code and documentation causing failure to satisfy user requirements. It also impedes the growth of the information processing industry. Validity of this statement is attested to by three of the many pieces of supporting evidence:

- The SHARE User Group Software Service Task Force Report, 1983 [1], that recommended an order of magnitude improvement in software quality over the next several years, with a like reduction in service. (Other manufacturers report similar recommendations from their users.)
- In 1979, 12 percent of programmer resource was consumed in post-shipment corrective maintenance alone and this figure was growing [8]. (Note that there is also a significant percentage of development and enhancement maintenance resource devoted to correcting defects. This is probably larger than the 12 percent expended in corrective maintenance, but there is no substantiating research.)
- The formal backlog of data processing tasks most quoted is three years [7].

At this point, a very important definition is in order:

A defect is an instance in which a requirement is not satisfied.

Here, it must be recognized that a requirement is any agreed upon commitment. It is not only the recognizable external product requirement, but can also include internal development requirements (e.g., the exit criteria of an operation) that must be met in order to satisfy the requirements of the end product. Examples of this would be the requirement that a test plan completely verifies that the product meets the agreed upon needs of the user, or that the code of a program must be complete before it is submitted to be tested.

While defects become manifest in the end product documentation or code, most of them are actually injected as the functional aspects of the product and its quality attributes are being created; during development of the requirements, the design and coding, or by insertion of changes. The author's research supports and supplements

that of B. Boehm *et al.* [9] and indicates that there are eight attributes that must be considered when describing quality in a software product:

- intrinsic code quality,
- freedom from problems in operation,
- usability,
- installability,
- documentation for intended users,
- portability,
- maintainability and extendability, and "fitness for use"—that implicit conventional user needs are satisfied.

Inspections and the Software Quality Problem

Previously, each of these attributes of software quality were evaluated by testing and the end user. Now, some of them are being partly, and others entirely, verified against requirements by inspection. In fact, the product requirements themselves are often inspected to ascertain whether they meet user needs. In order to eliminate defects from the product it is necessary to address their prevention, or detection and resolution as soon as possible after their injection during development and maintenance. Prevention is the most desirable course to follow, and it is approached in many ways including the use of state machine representation of design, systematic programming, proof of correctness, process control, development standards, prototyping, and other methods. Defect detection, on the other hand, was once almost totally dependent upon testing during development and by the user. This has changed, and over the past decade walkthrus and inspections have assumed a large part of the defect detection burden; inspections finding from 60 to 90 percent defects. (See [2], [3], and other unpublished product experiences.) They are performed much nearer the point of injection of the defects than is testing, using less resource for rework and, thus, more than paying for themselves. In fact, inspections have been applied to most phases of development to verify that the key software attributes are present immediately after the point at which they should first be introduced into the product. They are also applied to test plans and test cases to improve the defect detection efficiency of testing. Thus, inspections have been instrumental in improving all aspects of software product quality, as well as the quality of logic design and code. In fact, inspections *supplement* defect prevention methods in improving quality.

Essential to the quality of inspection (or its defect detection efficiency) is proper definition of the development process. And, inspection quality is a direct contributor to product quality, as will be shown later.

Definition of the Development Process

The software development process is a series of operations so arranged that its execution will deliver the desired end product. Typically, these operations are: Requirements Definition, System Design, High Level Design, Low Level Design, Coding, Unit Testing, Component or Function Testing, System Testing, and then user support and Maintenance. In practice, some of these operations are repeated as the product is recycled through them to insert functional changes and fixes.

The attributes of software quality are invested along with the functional characteristics of the product during the early operations, when the cost to remedy defects is 10–100 times less than it would be during testing or maintenance [2]. Consequently, it is advantageous to find and correct defects as near to their point of origin as possible. This is accomplished by inspecting the output product of each operation to verify that it satisfies the output requirements or *exit criteria* of the operation. In most cases, these exit criteria are not specified with sufficient precision to allow go/no verification. Specification of exit criteria in unambiguous terms that are objective and preferably quantitative is an essential characteristic of any well defined process. Exit criteria are the standard against which inspections measure completion of the product at the end of an operation, and verify the presence or absence of quality attributes. (A deviation from exit criteria is a defect.)

Shown below are the essence of 4 key criteria taken from the full set of 15 exit criteria items for the Coding operation:

- The source code must be at the "first clean compilation" level. That means it must be properly compiled and be free of syntax errors.
- The code must accurately implement the low level design (which was the verified output of the preceding process operation).
- All design changes to date are included in the code.
- All rework resulting from the code inspection has been included and verified.

The code inspection, I2, must verify that all 15 of these exit criteria have been satisfied before a module or other entity of the product is considered to have completed the Coding operation. Explicit exit criteria for several of the other inspection types in use will be contained in the author's book in software inspections. However, there is no reason why a particular project could not define its own sets of exit criteria. What is important is that exit criteria should be as objective as possible, so as to be repeatable; they should completely describe what is required to exit each operation; and, *they must be observed by all those involved.*

The objective of process control is to measure completion of the product during stages of its development, to compare the measurement against the project plan, and then to remedy any deviations from plan. In this context, the quality of both exit criteria and inspections are of vital importance. And, they must both be properly described in the manageable development process, for such a process must be controllable by definition.

Development is often considered a subset of the maintenance process. Therefore, the maintenance process must be treated in the same manner to make it equally manageable.

Software Inspection Overview

This paper will only give an overview description of the inspection process that is sufficient to enable discussion of updates and enhancements. The author's original paper on the software inspections process [2] gives a brief description of the inspection process and what goes on in an inspection, and is the base to which the enhancements are added. His forthcoming companion books on this subject and on building defect-free software will provide an implementation level description and will include all the points addressed in this paper and more.

To convey the principles of software inspections, it is only really necessary to understand how they apply to design and code. A good grasp on this application allows tailoring of the process to enable inspection of virtually any operation in development or maintenance, and also allows inspection for any desired quality attribute. With this in mind, the main points of inspections will be exposed through discussing how they apply in design and code inspections.

There are three essential requirements for the implementation of inspections:

- definition of the DEVELOPMENT PROCESS in terms of operations and their EXIT CRITERIA,
- proper DESCRIPTION of the INSPECTION PROCESS, and
- CORRECT EXECUTION of the INSPECTION PROCESS. (Yes, correct execution of the process is vital.)

The Inspection Process

The inspection process follows any development operation whose product must be verified. As shown below, it consists of six operations, each with a specific objective:

Operation	Objectives
PLANNING	Materials to be inspected must meet inspection entry criteria. Arrange the availability of the right participants. Arrange suitable meeting place and time.
OVERVIEW	Group education of participants in what is to be inspected. Assign inspection roles to participants.
PREPARATION	Participants learn the material and prepare to fulfill their assigned roles.
INSPECTION	*Find defects.* (Solution hunting and discussion of design alternatives is discouraged.)
REWORK	The author reworks all defects.
FOLLOW-UP	Verification by the inspection moderator or the entire inspection team to assure that all fixes are effective and that no secondary defects have been introduced.

Evaluation of hundreds of inspections involving thousands of programmers in which alternatives to the above steps have been tried has shown that all these operations are really necessary. Omitting or combining operations has led to degraded inspection efficiency that outweighs the apparent short-term benefits. OVERVIEW is the only operation that under certain conditions can be omitted with slight risk. Even FOLLOW-UP is justified as study has shown that approximately one of every six fixes are themselves incorrect, or create other defects.

From observing scores of inspections, it is evident that participation in inspection teams is extremely taxing and should be limited to periods of 2 hours. Continuing beyond 2 hours, the defect detection ability of the team seems to diminish, but is restored after a break of 2 hours or so during which other work may be done. Accordingly, no more than two 2 hour sessions of inspection per day are recommended.

To assist the inspectors in finding defects, for not all inspectors start off being good detectives, a checklist of defect types is created to help them identify defects appropriate to the exit criteria of each operation whose product is to be inspected. It also serves as a guide to classification of defects found by inspection prior to their entry to the inspection and test defect data base of the project. (A database containing these and other data is necessary for quality control of development.)

People and Inspections

Inspection participants are usually programmers who are drawn from the project involved. The roles they play for design and code inspections are those of the *Author* (Designer or Coder), *Reader* (who paraphrases the design or code as if they will implement it), *Tester* (who views the product from the testing standpoint), and *Moderator*. These roles are described more fully in [2], but that level of detail is not required here. Some inspections types, for instance those of system structure, may require more participants, but it is advantageous to keep the number of people to a minimum. Involving the end users in those inspections in which they can truly participate is also very helpful.

The Inspection Moderator is a *key player* and *requires special training* to be able to conduct inspections that are optimally effective. Ideally, to preserve objectivity, the moderator should not be involved in development of the product that is to be inspected, but should come from another similar project. The moderator functions as a "player-coach" and is responsible for conducting the inspection so as to bring a peak of synergy from the group. This is a quickly learned ability by those with some interpersonal skill. In fact, when participants in the moderator training classes are questioned about their case studies, they invariably say that they sensed the presence of the "*Phantom Inspector*," who materialized as a feeling that there had been an additional presence contributed by the way the inspection team worked together. The moderator's task is to invite the Phantom Inspector.

When they are properly approached by management, programmers respond well to inspections. In fact, after they become familiar with them, many programmers have been known to complain when they were not allowed enough time or appropriate help to conduct inspections correctly.

Three separate classes of education have been recognized as a necessity for proper long lasting implementation of inspections. First, *Management* requires a class of one day to familiarize them with inspections and their benefits to management, and *their role* in making them successful. Next, the *Moderators* need three days of education. And, finally, the other *Participants* should receive one half day of training on inspections, the benefits, and their roles. Some organizations have started inspections without proper education and have achieved some success, but less than others who prepared their participants fully. This has caused some amount of start-over, which was frustrating to everyone involved.

Management and Inspections

A definite philosophy and set of attitudes regarding inspections and their results is essential. The management education class on inspections is one of the best ways found to gain the knowledge that must be built into day-to-day management behavior that is required to get the most from inspections on a continuing basis. For example, management must show encouragement for proper inspections. Requiring inspections and then asking for shortcuts will not do. And, people must be motivated to find defects by inspection. *Inspection results must never be used for personnel performance appraisal.* However, the results of testing should be used for performance appraisal. This promotes finding and reworking defects at the lowest cost, and allows testing for verification instead of debugging. In most situations programmers come to depend upon inspections; they prefer defect-free product. And, at those installations where management has taken and maintained a leadership role with inspections, they have been well accepted and very successful.

Inspection Results and Their Uses

The defects found by inspection are immediately recorded and classified by the moderator before being entered into the project data base. Here is an example:

In module: *XXX*, Line: *YYY*, NAME-CHECK is performed one less time than required—LO/W/MAJ

The description of the defect is obvious. The classification on the right means that this is a defect in Logic, that the logic is Wrong (as opposed to Missing or Extra), and that it is a Major defect. A MAJOR defect is one that would cause a malfunction or unexpected result if left uncorrected. Inspections also find MINOR defects. They will not cause malfunction, but are more of the nature of poor workmanship, like misspellings that do not lead to erroneous product performance.

Major defects are of the same type as defects found by testing. (One unpublished study of defects found by system testing showed that more than 87 percent could have been detected by inspection.) Because Major defects are equivalent to test defects, inspection results can be used to identify *defect prone design and code*. This is enabled because empirical data indicates a directly proportional relationship between the inspection detected defect rate in a piece of code and the defect rate found in it by subsequent testing. Using inspection results in this way, it is possible to identify defect prone code and correct it, in effect, performing real-time quality control of the product as it is being developed, *before it is shipped or put into use*.

There are, of course, many Process and Quality Control uses for inspection data including:

- Feedback to improve the development process by identification and correction of the root causes of systematic defects before more code is developed;
- *Feed-forward* to prepare the process ahead to handle problems *or to evaluate corrective action in advance* (e.g., handling defect prone code);
- Continuing improvement and control of inspections.

An outstanding benefit of feedback, as reported in [3] was that designers and coders through involvement in inspections of their own work learned to find defects they had created more easily. This enabled them to *avoid* causing these defects in future work, thus providing much higher quality product.

Various Applications of Inspections

The inspection process was originally applied to hardware logic, and then to software logic design and code. It was in the latter case that it first gained notice. Since then it has been very successfully applied to software test plans and test cases, user documentation, high level design, system structure design, design changes, requirements development, and microcode. It has also been employed for special purposes such as cleaning up defect prone code, and improving the quality of code that has already been tested. And, finally, it has been resurrected to produce defect-free hardware. It appears that virtually anything that is created by a development process and that can be made visible and readable can be inspected. All that is necessary for an inspection is to define the exit criteria of the process operation that will make the product to be inspected, tailor the inspection defect checklists to the particular product and exit criteria, and then to execute the inspection process.

What's in a Name?

In contrast to inspections, walkthrus, which can range anywhere from cursory peer reviews to inspections, do not usually practice a process that is repeatable or collect data (as with inspections), and hence this process cannot be reasonably studied and improved. Consequently, their defect detection efficiencies are usually quite variable and,

when studied, were found to be much lower than those of inspections [2], [3]. However, the name "walkthru" (or "walkthrough") has a place, for in some management and national cultures it is more desirable than the term "inspection" and, in fact, the walkthrus in *some* of these situations are identical to formal inspections. (In almost all instances, however, the author's experience has been that the term walkthru has been accurately applied to the less efficient method—which process is actually in use can be readily determined by examining whether a formally defined development process with exit criteria is in effect, and by applying the criteria in [2, Table 5] to the activity. In addition, initiating walkthrus as a migration path to inspections has led to a lot of frustration in many organizations because once they start with the informal, they seem to have much more difficulty moving to the formal process than do those that introduce inspections from the start. And, programmers involved in inspections are usually more pleased with the results. In fact, their major complaints are generally to do with things that detract from inspection quality.) What is important is that the same results should not be expected of walkthrus as is required of inspections, *unless a close scrutiny proves the process and conduct of the "walkthru" is identical to that required for inspections*. Therefore, although walkthrus do serve very useful though limited functions, they are not discussed further in this paper.

Recognizing many of the abovementioned points, the IBM Information Systems Management Institute course on this subject is named: "Inspections: Formal Application Walkthroughs." They teach about inspection.

Contributors to Software Inspection Quality

Quality of inspection is defined as its ability to detect all instances in which the product does not meet its requirements. Studies, evaluations, and the observations of many people who have been involved in inspections over the past decade provide insights into the contributors to inspection quality. Listing contributors is of little value in trying to manage them as many have relationships with each other. These relationships must be understood in order to isolate and deal with initiating root causes of problems rather than to waste effort dealing with symptoms. The ISHIKAWA or FISHBONE CAUSE/EFFECT DIAGRAM [11], shown in Fig. 2, shows the contributors and their cause/effect relationships.

As depicted in Fig. 2, the main contributors, shown as main branches on the diagram, are: *PRODUCT INSPECTABILITY, INSPECTION PROCESS, MANAGERS,* and *PROGRAMMERS*. Subcontributors, like *INSPECTION MATERIALS* and *CONFORMS WITH STANDARDS*, which contribute to the *PRODUCT INSPECTABILITY*, are shown as twigs on these branches. Contributors to the subcontributors are handled similarly. Several of the relationships have been proven by objective statistical analysis, others are supported by empirical data, and some are evident from project experience. For example, one set of relationships very thoroughly established in a controlled study by F. O. Buck, in "Indicators of Quality Inspections" [10], are:

- excessive SIZE OF MATERIALS to be inspected leads to a PREPARATION RATE that is too high.
- PREPARATION RATE that is too high contributes to an excessive RATE OF INSPECTION, and
- Excessive RATE OF INSPECTION *causes fewer defects to be found*.

This study indicated that the following rates should be used in planning the I2 code inspection:

OVERVIEW:	500 Noncommentary Source Statements per Hour.
PREPARATION:	125 Noncommentary Source Statements per Hour.
INSPECTION:	90 Noncommentary Source Statements per Hour.
Maximum Inspection Rate:	125 Noncommentary Source Statements per Hour.

The rate of inspection seems tied to the thoroughness of the inspection, and there is evidence that defect detection efficiency diminishes at rates above 125 NCSS/h. (Many projects require reinspection if this maximum rate is exceeded, and the reinspection usually finds more defects.) Separate from this study, project data show that inspections conducted by trained moderators are very much more likely to approximate the permissible inspection rates, and yield higher quality product than moderators who have not been trained. Meeting this rate is not a direct conscious purpose of the moderator, but rather is the result of proper conduct of the inspection. In any event, as the study shows, requiring too much material to be inspected will induce insufficient PREPARATION which, in turn, will cause the INSPECTION to be conducted too fast. Therefore, it is the responsibility of management and the moderator to start off with a plan that will lead to successful inspection.

The planning rate for high level design inspection of *systems design* is approximately twice the rate for code inspection, and low level (Logic) design inspection is nearly the same (rates are based upon the designer's estimate of the number of source lines of code that will be needed to implement the design). Both these rates *may* depend upon the complexity of the material to be inspected and the manner in which it is prepared (e.g., unstructured code is more difficult to read and requires the inspection rate to be lowered. Faster inspection rates while retaining high defect detection efficiency *may* be feasible with highly structured, easy to understand material, *but further study is needed*). Inspections of requirements, test plans, and user documentation are governed by the same rules as for code inspection, although inspection rates are not as clear for them and are probably more product and project dependent than is the case of code.

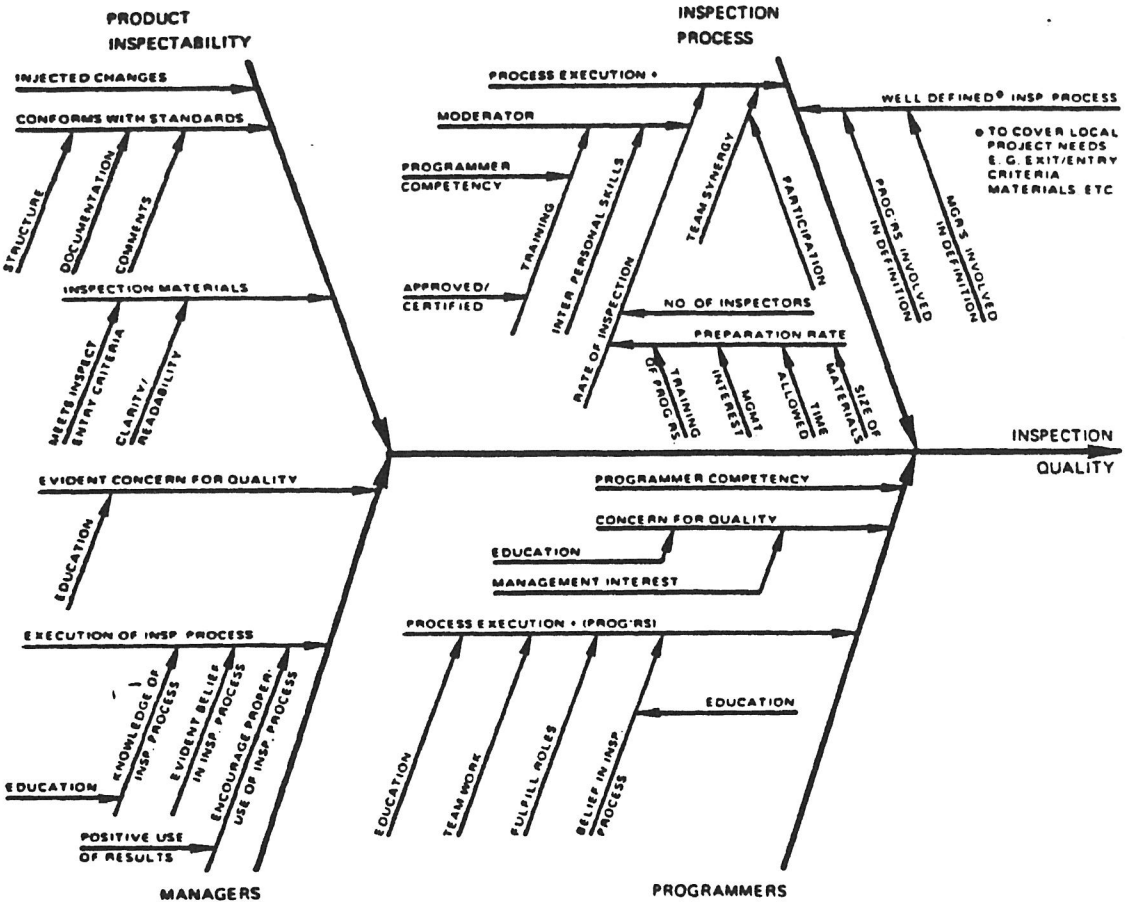

Fig. 2. Fishbone diagram of contributors to inspection quality.

With a good knowledge of and attention to the contributors to inspection quality, management can profoundly influence the quality, and the development and maintenance costs of the products for which they are responsible.

Summary

Experience over the past decade has shown software inspections to be a potent defect detection method, finding 60–90 percent of all defects, as well as providing feedback that enables programmers to avoid injecting defects in future work. As well as providing checkpoints to facilitate process management, inspections enable measurement of the performance of many tools and techniques in individual process operations. Because inspection engages similar skills to those used in creating the product (and it has been applied to virtually every design technique and coding language), it appears that anything that can be created and described can also be inspected.

Study and observation have revealed the following key aspects that must be managed to take full advantage of the many benefits that inspections offer:

Capability	Action Needed to Enhance the Capability
• Defect Detection	— Management understanding and continuing support. This starts with education. — Inspection moderator training (3 days). — Programmer training. — Continuing management of the contributors to inspection quality. — Inspect all changes. — Periodic review of effectiveness by management. — Inspect test plans and test cases. — Apply inspections to main defect generating operations in development *and* maintenance processes.
• Defect Prevention (or avoidance)	— Encourage programmers to understand how they created defects and what must be done to avoid them in future. — Feedback inspection results promptly and removes root causes of systematic defects from the development or maintenance processes. — Provide inspection results to

quality circles or quality improvement teams.

— Creation of requirements for expert system tools (for defect prevention) based upon analysis of inspection data.

- Process Management — Use inspection completions as checkpoints in the development plan and measure accomplishment against them.

References

[1] L. H. Fenton, "Response to the SHARE software service task force report," IBM Corp., Kingston, NY, Mar. 6, 1984.

[2] M. E. Fagan, "Design and code inspections to reduce errors in program development," *IBM Syst. J.*, vol. 15, no. 3, 1979.

[3] *IBM Technical Newsletter GN20-3814*, Base Publication GC20-2000-0, Aug. 15, 1978.

[4] T. D. Crossman, "Inspection teams, are they worth it?" in *Proc. 2nd Nat. Symp. EDP Quality Assurance*, Chicago, IL, Mar. 24-26, 1982.

[5] R. R. Larson, "Test plan and test case inspection specification," IBM Corp., Tech. Rep. TR21.585, Apr. 4, 1975.

[6] T. D. Crossman, "Some experiences in the use of inspection teams in application development," in *Proc. Applicat. Develop. Symp.*, Monterey, CA, 1979.

[7] G. D. Brown and D. H. Sefton, "The micro vs. the applications logjam," *Datamation*, Jan. 1984.

[8] J. H. Morrissey and L. S.-Y. Wu, "Software engineering: An economical perspective," in *Proc. IEEE Conf. Software Eng.*, Munich, West Germany, Sept. 14-19, 1979.

[9] B. Boehm *et al.*, *Characteristics of Software Quality*. New York: American Elsevier, 1978.

[10] F. O. Buck, "Indicators of quality inspections," IBM Corp., Tech. Rep. IBM TR21.802, Sept. 1981.

[11] K. Ishikawa, *Guide to Quality Control*. Tokyo, Japan: Asian Productivity Organization, 1982.

Michael E. Fagan is a Senior Technical Staff Member at the IBM Corporation, Thomas J. Watson Research Center, Yorktown Heights, NY. While at IBM, he has had many interesting management and technical assignments in the fields of engineering, manufacturing, software development, and research. In 1972, he created the software inspection process, and has helped implement it within IBM and also promoted its use in the software industry. For this and other work, he has received IBM Outstanding Contribution and Corporate Achievement Awards. His area of interest is in studying and improving all the processes that comprise the software life cycle. For the past two years, he has been a Visiting Professor at, and is on the graduate council of, the University of Maryland.

Laurie A. Williams and
Robert R. Kessler

All I Really to Know About Programmi I Learned Kindergar

When it comes to programming practices, studies show two heads are almost always better than one.

NEED PAIRING IN TEN

Pair programming is a practice in which two programmers work side-by-side at one computer, continuously collaborating on the same design, algorithm, code, or test. This method has been demonstrated to improve productivity and the quality of software products. Moreover, a recent survey (hereafter referred to as "the pair programming survey") found that programmers were universally more confident in their solutions when programming in pairs as opposed to working alone. Likewise, 96% agreed they enjoy their jobs more when pair programming [12].

However, most programmers are long conditioned to working alone and often resist the transition to pair programming. Ultimately, most make this transition with great success. The goal of this article is to help programmers become effective pair programmers. The transition to and on-going success as a pair programmer often involves practicing everyday civility, as illustrated in an essay by Robert Fulghum (see box). Here, we take each line from the essay (with occasional poetic license) to explore the inherent lessons related to successful pair programming.

Anecdotal and initial statistical evidence indicates pair programming is highly beneficial. In extreme programming (XP)—an emerging software development methodology—all production code is written with a partner. XP was developed initially by Smalltalk code developer and consultant Kent Beck with colleagues Ward Cunningham and Ron Jeffries. XP's requirements gathering, resource allocation, and design practices are a radical departure from most accepted methodologies. Customer requirements are written as fairly informal "User Story" cards where a

rough effort estimate is assigned to the cards. The cards are then designated for a programming pair, and coding begins. With no formal design procedures or discussions on overall system planning or architecture, the pair determines which code in the code base needs to be added or changed. This practice requires the use of *collective code ownership* whereby any programming pair can modify or add to any code in the code base, regardless of the original programmer. Extensive unit testing is continually performed on this ever-enlarging code base.

The evidence of XP's success is highly anecdotal, but so impressive it has aroused the curiosity of many highly respected software-engineering researchers and consultants. The largest example of its accomplishment is the sizable Chrysler Comprehensive Compensation system launched in May 1997. After finding significant, initial development problems, Beck and Jeffries restarted this development using XP principles. The payroll system pays some 10,000 employees each month and has 2,000 classes and 30,000 methods [1]. It went into production almost on schedule, and is still operational today.

XP attributes great success to the use of pair programming by all programmers—experts and novices alike. XP advocates pair programming with such fervor that even prototyping done solo is scrapped and rewritten with a partner. One key element is that a continuous code review is performed while working in pairs. It is amazing to see how many obvious, yet unnoticed, defects are recognized when another person is watching over a shoulder. According to [11], the results demonstrate that two programmers working together are more than twice as fast and think of more than twice as many solutions to a problem as two working alone, while attaining higher defect prevention and defect removal, leading to a higher quality product.

In addition, two other studies support the use of pair programming. Larry Constantine, a noted programmer and consultant, reported on some "dynamic duos" during a visit to P.J. Plaugher's software company, Whitesmiths, Ltd., providing anecdotal support for collaborative programming. He immediately noticed a room full of paired programmers working on the same code at one computer. "Having adopted this approach, they were delivering finished and tested code faster than ever ... The code that came out the back of the two programmer terminals was nearly 100% bug free ... it was better code, tighter and more efficient, having benefited from the thinking of two bright minds and the steady dialogue between two trusted terminal-mates ... Two programmers in tandem is not redundancy; it's a direct route to greater efficiency and better quality, he contends." [3].

An experiment by John Nosek at Temple University studied 15 full-time, experienced programmers working for 45 minutes on a challenging problem, important to their organization, in their own environment, and with their own equipment. Five worked individually, 10 worked collaboratively in five pairs. Conditions and materials used were the same for both the experimental (team) and control (individual) groups. This study provided statistically significant results, using a two-sided *t*-test. "To the surprise of the managers and participants, all the teams outperformed the individual programmers, enjoyed the problem-solving process more, and had greater confidence in their solutions," Nosek explains.

Moreover, the groups completed the task 40% more quickly and effectively by producing better algorithms and code in less time. The majority of the programmers were skeptical of the value of collaboration in working on the same problem and thought it would not be an enjoyable process. However, results show collaboration improved both their performance and their enjoyment of the problem-solving process [8].

The respondents of the pair programming survey gave overwhelming support for the technique. Says one: "I strongly feel pair programming is the primary reason our team has been successful. It has given us a very high level of code quality (almost to the point of zero defects). The only code we have ever had errors in was code that wasn't pair programmed ... utilized."

Examination of why pair programming works with such success reveals that a number of elementary

MOST PROGRAMMERS ARE LONG CONDITIONED TO WORKING ALONE AND OFTEN RESIST THE TRANSITION TO PAIR PROGRAMMING. ULTIMATELY, HOWEVER, MOST MAKE THIS TRANSITION WITH GREAT SUCCESS.

principles come into play. These principles can be discussed in the context of Fulghum's essay:

Share everything.

In pair programming, two programmers are assigned to jointly produce one artifact (design, algorithm, code, among others). The two programmers are like a coherent, intelligent organism working with one mind, responsible for every aspect of this artifact. One person is typing or writing, the other is continually reviewing the work. Both are equal participants in the process. It is not acceptable to say or think things such as, "You made an error in your design," or "That defect was from your part." Instead, "We screwed up the design," or better yet, "We just got through our test with no defects!" Both partners own everything.

Play fair.

With pair programming, one person drives (has control of the keyboard or is recording design ideas) while the other is continuously reviewing the work. Even when one programmer is significantly more experienced than the other, it is important to take turns driving, lest the observer become disjointed, feel out of the loop, or unimportant.

The person not driving should not be a passive observer, but instead should always be active and engaged. "Just watching someone program is about as interesting as watching grass die in a desert" [2]. In the pair programming survey, approximately 90% stated the main role of the person not typing was to perform continuous analysis, design and code reviews. "When one partner is busy typing, the other is thinking at a more strategic level. Where is this line of development going? Will it run into a dead end? Is there a better overall strategy?"

Don't hit people.

Make sure he or she stay focused and on-task (nonviolently, of course). Undoubtedly, a benefit of working in pairs is that each person is far less likely to waste time reading email, Web surfing, or staring out the window because their partner is awaiting continuous contribution and input. "Two people working together in a pair treat their shared time as more valuable. They tend to cut phone calls short; they don't waste each other's time" [10].

Additionally, each is expecting the other to follow the prescribed development practices. "With your partner watching, though, chances are that even if you feel like blowing off one of these practices, your partner won't … the chances of ignoring your commitment to the rest of the team is much smaller in pairs then it is when you are working alone" [2].

As summarized in the pair programming survey, "It takes more effort because the pace is forced by the other person all the time; neither person feels they can slack off." As each keeps his or her partner focused and on-task, tremendous productivity gains and quality improvements are realized.

Put things (especially negative thoughts) *back where they belong.*

The mind is a tricky thing. If you think about something long enough, the brain will consider it a truth. If you tell yourself something negative, such as "I'm a terrible programmer," soon your brain will believe you. However, anyone can control this negative self-talk by putting these thoughts where they belong—out of mind—every time they start to creep in. The surveyed pair programmers indicated it was very difficult to work with someone who had insecurity or anxiety about their programming skills. They tend to have a defensiveness about them. Programmers with such insecurity should view pair programming as a means to improve their skill by constantly watching and obtaining feedback from another.

A survey respondent reflected, "The best thing about pair programming for me is the continuous discussion that gave me training in formulating the thoughts I have about design and programming. It helps me reflect over these thoughts, which has made me a better designer/programmer." Indeed, two researchers surveyed 750 working programmers on coordination techniques in software development [7]. The communication technique with both the highest use and the highest value was discussion with peers. "The standard response when one confronts a problem that cannot be solved alone is to go to a colleague close by." When pair programming, the "colleague close by" is continuously available. Together, the pair can solve problems they couldn't solve alone and can help improve each other's skills.

Also, negative thoughts such as "I'm an awesome programmer, and I'm paired up with a total loser" should also be rejected, lest the collaborative relationship be destroyed. None of us, no matter how skilled, is infallible and above the input of another. John von Neumann, the great mathematician and creator of the von Neumann computer architecture, recognized his own inadequacies and continuously asked others to review his work. "And indeed, there can be no doubt of von Neumann's genius. His very ability to realize his human limitation put him head and shoulders above the average programmer today … Average people can be trained to accept their humanity—their inability to function like a machine—and to value it

and work with others so as to keep it under the kind of control needed if programming is to be successful" [9].

Clean up your mess.
Pair programmers have the advantage of the presence of a partner to help them clean up. Many have related that many obvious, but undetected, defects were noticed by another person watching over their shoulder. Additionally, these defects can be removed without the natural animosity that might develop in a formal inspection meeting. Established software engineering techniques often stress the importance of defect prevention and efficient defect removal. Perhaps this "over the shoulder" technique epitomizes defect prevention and defect removal efficiency.

***Don't take things* too seriously.**
"Ego-less programming," an idea that surfaced 25 years ago by Gerald Weinberg in *The Psychology of Computer Programming*, is essential for effective pair programming. According to the pair programming survey, excess ego can manifest itself in two ways, both damaging the collaborative relationship. First, having a "my way or the highway" attitude can prevent the programmer from considering other ideas. Secondly, excess ego can cause a programmer to be defensive when receiving criticism or to view this criticism as mistrust.

A true scenario about a programmer seeking review of the code he produced is discussed in [9]. On this particular bad programming day, an individual egolessly laughed because his reviewer found 17 bugs in 13 statements. After fixing these defects, however, the code performed flawlessly during testing and in production. How different this outcome might have been had the programmer been too proud to accept the input of others or had viewed this input as an indication of his inadequacies. Having another review design and coding continuously and objectively is an extremely beneficial aspect of pair programming. "The human eye has an almost infinite capacity for not seeing what it does not want to see ... Programmers, if left to their own devices, will ignore the most glaring errors in their output—errors that anyone else can see in an instant" [9].

Conversely, a person who always agrees with their partner lest create tension also minimizes the benefits of collaborative work. For favorable idea exchange, there should be some healthy disagreement/debate. Notably, there is a fine balance between displaying too much and too little ego. Effective pair programmers hone this balance during an initial adjustment period. Ward Cunningham, one of the XP founders and experienced pair programmer, reports this initial adjustment period can take hours or days, depending on the individuals, nature of work, and their past experience with pair programming.

NONE OF US, NO MATTER HOW SKILLED, IS INFALLIBLE AND ABOVE THE INPUT OF ANOTHER.

Say you're sorry when you hurt somebody.
In the pair programming survey, 96% of the programmers agreed that appropriate workspace layout was critical to their success. Pair programmers take aggressive action on improving their physical environment, by taking matters into their own hands (armed with screwdrivers). The programmers must be able to sit side-by-side and program, simultaneously viewing the computer screen and sharing the keyboard and mouse. Extreme programmers have a "slide the keyboard/don't move the chairs" rule.

Effective communication, both within a collaborative pair and with other collaborative pairs, is paramount. Without much effort, programmers need to see each other, ask each other questions, and make decisions on things such as integration issues, lest these questions/issues are not discussed adequately. Programmers also benefit from "accidentally" overhearing other conversations to which they can have vital contributions. Separate offices and cubicles can inhibit this necessary exchange. "If any one thing proves that psychological research has been ignored by working managers, it's the continuing use of half partitions to divide workspace into cubicles. ... Like many kings, some managers use divide-and-conquer tactics to rule their subjects, but programmers need contact with other programmers" [9].

***Wash your hands before you* start.**
Many programmers venture into their first pair programming assignment skeptical of the value of collaboration in programming, not expecting to benefit from or to enjoy the experience. Two skeptical pro-

grammers joined together in a team could certainly carry out this self-fulfilling prophecy. In the pair programming survey, 91% agreed that "partner buy-in" was critical to pair programming success.

Pair programming relationships can be established informally by one programmer asking another to have a seat and give them some help—and carry on from there. Once the relationship has been created, one could say, "That went well. I have some extra time now. Is there anything this afternoon that I can help you with?" Experience has shown that having just one programmer, very positive and/or experienced in pair programming, can lead the pair to become one victoriously jelled collaborative team.

Tom DeMarco shares his inspiring view on this type of union in [4]. "A jelled team is a group of people so strongly knit that the whole is greater than the sum of the parts. The production of such a team is greater than that of the same people working in unjelled form. Just as important, the enjoyment that people derive from their work is greater than what you'd expect given the nature of the work itself. In some cases, jelled teams working on assignments that others would declare downright dull have a simply marvelous time. ... Once a team begins to jell, the probability of success goes up dramatically. The team can become almost unstoppable, a juggernaut for success."

Advice to an up-and-coming pair programmer: Wash your hands of any skepticism, develop an expectation of success, and greet your collaborative partner by saying, "Jell me!" This is an unprecedented opportunity for the two to excel as one.

Flush.

Inevitably, the pair programmers will work on something independently. Of the programmers surveyed, over half said they reviewed work done independently when they rejoined with their partner, and incorporated it into the project. Alternately, extreme programmers flush and rewrite independent work. In their XP experience, the majority of the defects could be traced back to a time when a programmer worked independently. In fact, during the five months prior to first production from the Chrysler Comprehensive Compensation project, the only defects that made it through unit and functional testing were written by someone programming alone. In rewriting, the author must undergo the customary continuous review of the work, which identifies additional defects.

The decision to flush or to review work done independently can be made by a pair of programmers, or the choice may be encouraged, as it is with XP. However, it is important to note none of the programmers surveyed incorporated work done independently without reviewing it.

Warm cookies and cold milk are good for you.

Because pair programmers must keep each other continuously focused and on-task, it can be a very intense and mentally exhausting experience. Taking a break periodically is important for maintaining the stamina for another round of productive pair programming. During the break, it is best to disconnect from the task at hand and approach it refreshed when restarting. Suggested activities: checking email, making phone calls, surfing the Web, eating warm cookies, and drinking cold milk.

Live a balanced life—learn some and think some and draw and paint and sing and dance and play and work every day some.

Communicating with others on a regular basis is key for leading a balanced life. "If asked, most programmers would probably say they preferred to work alone in a place where they wouldn't be disturbed by other people" [9]. But, informal discussions with other programmers—the one you are paired with or any other—allow for effective idea exchange and efficient transfer of information. For example, Weinberg [9] discusses a large university computing center, in this case a common space with a collection of vending machines in the back of the room. Some of the more serious students complained about the noise in this common space, and the machines were moved out. Soon after the removal of the machines, a different complaint echoed the walls: Not enough computer consultants! Suddenly, the lines for the computer consultant wound around the room. The cause of the change was the fact that informal chat around the vending machines offered idea exchanges and information transfers between the mass of programmers. Now, all this discussion had to be done with the relatively few consultants. (Sadly, the vending machines were never moved back in.)

Take a nap* (or a break from working together) *every afternoon.

It's certainly not necessary to work separately every afternoon. But, according to 50% of the surveyed programmers, it is acceptable to work alone 10%–50% of the time. Many prefer to do experimental prototyping, tough, deep-concentration problems, and logical thinking alone. Most agree that simple, well-defined, rote coding is more efficiently done by a solitary programmer and then reviewed with a partner.

When you go out into the world, watch out for traffic, hold hands and stick together.

With pair programming, the two programmers become one. There should be no competition between the two; both must work for a singular purpose, as if the artifact was produced by a singular good mind. Blame for problems or defects should never be placed on either partner. The pair needs to trust each other's judgement and each other's loyalty to the team.

***Be aware of wonder* (and the power of two brains working together).**

Human beings can only remember and learn a bounded amount. Therefore, they must consult with others to increase this bounty. When two are working together, each has their own set of knowledge and skills. A large subset of this knowledge and these skills will be common between the two, allowing them to interact effectively. However, the unique skills of each individual will allow them to engage in interactions that pool their resources to accomplish their tasks. "Collaborative people are those who identify a possibility and recognize that their own view, perspective, or talent is not enough to make it a reality. Collaborative people see others not as creatures who force them to compromise, but as colleagues who can help them amplify their talents and skills" [6].

Experiences show that a pair will come up with more than twice as many possible solutions as two individuals working alone. They will then proceed to more quickly narrow in on the best solution and will implement it more quickly and with better quality. A survey respondent reflects, "It is a powerful technique as there are two brains concentrating on the same problem all the time. It forces one to concentrate fully on the problem at hand."

Final Thoughts

Both anecdotal and initial statistical evidence indicate that pair programming is a powerful technique for generating high-quality software products. The pair works and shares ideas together to tackle the complexities of software development. They continuously perform inspections on each other's artifacts leading to the earliest, most efficient form of defect removal possible. In addition, they keep each other intently focused on the task at hand.

Programmers, however, have generally been conditioned to working alone. Making the transition to pair programming involves breaking down some personal barriers. First, the programmers must understand the benefits of intercommunication outweigh their common (perhaps innate) preferences for working alone and undisturbed. Secondly, they must confidently share their work, accepting instruction and suggestions for improvement in order to improve their own skills and the product at hand. They must display humility in understanding they are not infallible and their partner has the ability to make improvements in what they do. Lastly, a pair programmer must accept ownership of his or her partner's work and, therefore, be willing to constructively express criticism and suggested improvements.

The transition to pair programming takes conditioned solitary programmers out of their comfort zone. However, the potential for achieving results impossible by a single programmer makes this a journey to greatness. C

References

1. Anderson, A., Beattie, R., Beck, K. et al. Chrysler goes to "Extremes." *Distrib. Comput.* (Oct. 1998), 24–28.
2. Beck, K. *Extreme Programming Explained: Embrace Change.* 1999. Addison-Wesley, Reading, PA.
3. Constantine, L. L. *Constantine on Peopleware.* Yourdon Press, Englewood Cliffs, NJ. 1995.
4. DeMarco, T., Lister, T. Peopleware. Dorset House, New York, NY. 1977.
5. Fulghum, R. *All I Really Need to Know I Learned in Kindergarten.* 1988. Villard Books, New York, NY.
6. Hargrove, R. *Mastering the Art of Creative Collaboration.* McGraw-Hill, New York, NY. 1988.
7. Kraut, R. E., Streeter, L.A. Coordination in software development. *Commun. ACM 38,* 3 (Mar. 1995), 69–81.
8. Nosek, J. T. The case for collaborative programming. *Commun. ACM 41,* 3 (Mar. 1998), 105–108.
9. Weinberg, G. M. *The Psychology of Computer Programming Silver Anniversary Edition.* Dorset House, New York, NY. 1998.
10. Wiki. Pair Programming Facilities. Portland Pattern Repository. Mar. 16, 1999; c2.com/cgi/wiki?PairProgrammingFacilities.
11. Wiki. Programming In Pairs. Portland Pattern Repository. June 29, 1999; c2.com/cgi/wiki?ProgrammingInPairs.
12. Williams, L. Pair Programming Questionnaire. 1999; limes.cs.utah.edu/questionnaire/questionnaire.htm.

Laurie A. Williams (lwilliam@cs.utah.edu) is a Spring 2000 computer science Ph.D. graduate and instructor at the University of Utah, Salt Lake City, UT.

Robert R. Kessler (kessler@cs.utah.edu) is a professor and chair of the Department of Computer Science at the University of Utah, Salt Lake City. He is the founder of the Center for Software Science, a state of Utah Center of Excellence.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

© 2000 ACM 0002-0782/00/0500 $5.00