Improving Quality Through Software Inspections[1]

Karl E. Wiegers

Process Impact
www.processimpact.com

First the earth cooled. Then the dinosaurs came, but they didn't make it. Next, humans began programming computers using coding forms and punched cards. With just a few compilation cycles per day, there was plenty of time to study the listings and look for those elusive bugs.

Now, in modern times, we all write programs at the keyboard and compile them as frequently as we like. Complete listings are rarely printed out for entire software applications, let alone examined line by line by a group of people on a bug hunt. Ironically, this very process, officially termed "software inspection," is one of the most effective methods we have for identifying errors in a program. So much for progress.

Every time I actually sit down and read through an entire listing of a program, I find ways it can be improved: redundant logic, unused variables, incorrect boolean tests, comments that don't match the code, and a hundred other errors that make me glad I took the time to look. Software inspections and their cousins, reviews and walkthroughs, are proven techniques for reducing the number of defects in a program before it goes out the door. If you are in an organization of two or more people, some kind of inspection activity should be a part of your standard software development process. If you work alone, well, at least read the code.

Benefits of Inspections

Any software engineering textbook will tell you that the cost of fixing a defect rises dramatically the later it is found in the development lifecycle of the program. A large German company found that a defect caught by testing cost 14.5 times as much to correct as did one found by formal inspection, while a defect discovered by a customer cost 68 times as much to fix. One IBM report indicated that an error found after product release cost 45 times as much to correct as one uncovered during design. While exact numbers from different sources may vary, it is clear that if we can detect and correct errors earlier, we will save money and time in the long run.

One implication of this wisdom is that we should not be inspecting only source code. Indeed, any human-readable artifact produced during software development can be inspected: requirements specifications, design documents and models, test plans, system documentation, and user aids all are candidates. Inspections are one of the few "testing" techniques available for software work products other than code.

In fact, the greatest leverage from the time spent on software inspections comes from examining requirements documents, since correcting a bug this early in the game will save you the most money. Of course, this assumes that you HAVE written software requirements specifications. If you do not, ask yourself how you and your customer will both know when the project is completed. If you don't come up with a good answer, you might want to make written requirements documents a part of your software development process.

Incorporating inspections into your software engineering process is not free. They can consume between 5 and 15% of the total project budget. However, many companies have learned

[1] This paper was originally published in *Software Development*, April 1995. It is reprinted (with modifications) with permission from *Software Development* magazine.

that the results yielded by a good inspection process far outweigh the costs of performing them. Savings can be estimated by multiplying the number of bugs found by inspection before the product is shipped by the approximate cost of fixing a bug that is found by a customer. As an example, the Jet Propulsion Laboratory estimated a net savings of $7.5 million from 300 inspections performed on software they produced for NASA. Another large company estimated an annual savings of $2.5 million due to their inspection activities, based on costs of $146 to fix a major defect found by inspection and $2,900 to fix one found by the customer. As with all software practices, your mileage may vary.

Inspections shorten delivery time by reducing the time spent in the integration and system test/debug phases, since a cleaner product is passed into those late-stage quality filters. Better quality in the completed product saves on maintenance time, too. Reducing the effort that you have to spend fixing bugs after delivery frees up time that can be used for new development work.

Cutting down on rework always improves productivity. While it is difficult to compute an anticipated return on investment for implementing inspections in a particular organization, the software literature has many examples that justify the investment made in this error detection technique. The excuse of “I don’t have time to do inspections” simply doesn’t hold water.

In an organization functioning at a high level of software process maturity, the data collected from inspections can be used to improve the process further. Data from inspection summary reports can be used to identify the most common or most costly kinds of bugs, determine their root causes, and change how work is performed so as to prevent those types of errors. At Bull HN Information Systems, such inspection and test defect data is used to plan test activities, estimate the number of bugs to be found at various test stages, and track and analyze project progress.

There are some less obvious side benefits of performing inspections. I have found that nearly all participants can learn something from every inspection. Group inspections provide an opportunity to see how other team members do things. Knowledge is automatically and effortlessly exchanged about programming language features, coding and commenting style, program architecture, design notations, ways to document requirements, and all the other aspects of the software development process.

Every inspection I have been a part of has generated new ideas about how to do my own work better, thereby supporting both individual and team efforts for continual process improvement. While the purpose of an inspection is not education, they do provide an effective forum for less experienced team members to learn a lot while still making useful contributions.

Inspections tend to reveal different kinds of errors than do testing activities. Table 1 shows some common types of programming problems and which techniques are effective for detecting them. The combination of formal inspections and structured, systematic testing provides a powerful tool for creating defect-free programs.

Inspections, Walkthroughs, and Reviews

A variety of related manual defect detection activities go by names such as inspection, formal inspection, Fagan inspection, walkthrough, peer review, formal technical review, and so on. In the most general sense, these are all ways in which someone other than the creator of a software work product examines the product with the specific intent of finding errors in it.

I will use the terms “inspection” and “review” interchangeably, although strictly speaking they are not the same beast. Daniel Freedman and Gerald Weinberg explore a variety of review disciplines in their book *Handbook of Walkthroughs, Inspections, and Technical Reviews, Third Ed.* (Dorset House, 1990). Let’s look at some of the different ways these activities can be performed.

Michael Fagan developed the formal software inspection process at IBM in the mid 1970s, hence the term "Fagan inspection." Fagan inspections are thoroughly discussed in *Software Inspection* by Tom Gilb and Dorothy Graham (Addison-Wesley, 1993). To qualify as a true inspection, the activity follows a specified process and the participants play well-defined roles. An inspection team consists of 3-8 members and includes these roles:

Moderator –	leads the inspection, schedules meetings, controls the meetings, reports inspection results, and follows up on rework issues. Moderators should be trained in how to conduct inspections, including how to keep participants with strong technical skills but low social skills from killing each other.
Author –	created or maintains the work product being inspected. The author may answer questions asked about the product during the inspection, and he also looks for defects. The author cannot serve as moderator, reader, or recorder.
Reader –	describes the sections of the work product to the team as they proceed through the inspection. The reader may paraphrase what is happening in the product, such as describing what a section of code is supposed to do, but he does not usually read the product verbatim.
Recorder –	classifies and records defects and issues raised during the inspection. The moderator might perform this role in a small inspection team.
Inspector –	attempts to find errors in the product. All participants actually are acting as inspectors, in addition to any other responsibilities. Good people to invite as inspectors include: the person who created the predecessor specification for the work product being inspected (e.g., the designer for a code inspection); those responsible for implementing, testing, or maintaining the product; a quality assurance representative to act as standards enforcer; other project members; and someone who is not involved in the project at all but who has the skill set and defect-detection abilities to be able to contribute usefully to inspecting any work product of this type. We also require that our key customer representatives participate in requirements specification inspections.

Did you notice the word "manager" did not appear in the previous section? The conventional wisdom is that managers do not belong at inspections, as their presence may inhibit the process of finding defects. However, in many small software groups, the first-line supervisor is also a developer. Such a person probably is creating products that ought to be inspected, as well as being technically qualified to contribute to inspections. If the culture permits, I think it is appropriate for this sort of manager to participate in review activities. In fact, if his own work products are reviewed to the same standards as those created by anyone else, this can reinforce the cultural foundation for the inspection process in the group.

A formal inspection consists of several activities:

1. Planning – The moderator selects the inspection team, obtains materials to be inspected from the author, and distributes them and any other relevant documents to the inspection team in advance. Materials should be distributed at least two or three days prior to the inspection. We leave the responsibility for requesting an inspection to the author, but you should establish some entry criteria for assessing readiness of a product for inspection. For code, you might require that it compiles cleanly and that the listing provided to inspectors includes line numbers.

2. Overview meeting – This meeting gives the author an opportunity to describe the important features of the product to the inspection team. It can be omitted if this information is already known to the other participants.

3. Preparation – Each participant is responsible for examining the work artifacts prior to the actual inspection meeting, noting any defects found or issues to be raised. Perhaps 75% of the errors found during inspections are identified during the preparation step. The product should be

compared against any predecessor (specification) documents to assess completeness and correctness.

Checklists of defects commonly found in this type of work product should be used during preparation to hunt for anticipated types of errors. A sample checklist for code inspections is found in Figure 1. Checklists for inspecting nearly a dozen types of software work products can be found (along with tons of other useful information) in *Software Inspection Process* by Robert G. Ebenau and Susan H. Strauss (McGraw-Hill, 1994). If any standards are being used that pertain to this class of work product, each preparer should look for deviations from the standard.

4. Inspection meeting – During this session, the team convenes and is led through the work product by the moderator and reader. If the moderator determines at the beginning of the meeting that insufficient time has been devoted to preparation by the participants, the meeting should be rescheduled. During the discussion, all inspectors can report defects or raise other issues, which are documented on a form by the recorder.

   The meeting should last no more than two hours. At its conclusion, the group agrees on an assessment of the product: accepted as is (I have never seen this happen); accepted with minor revisions; major revisions needed and a second inspection required; or rebuild the product.

5. Causal analysis – An important long-term benefit of an inspection program is the insight it can provide into the kinds of defects being created and process changes you can make to prevent them. This “causal analysis” step provides that understanding. While not essential to the current inspection, it can lead to improved quality on future work by helping to avoid making the same mistakes in the future.

6. Rework – The author is responsible for resolving all issues raised during the inspection. This does not necessarily mean making every change that was suggested, but an explicit decision must be made about how each issue or defect will be dealt with.

7. Follow-up – To verify that the necessary rework has been performed properly, the moderator is responsible for following up with the author. If a significant fraction (say, 10 percent) of the work product was modified, an additional inspection may be required. This is the final gate through which the product must pass in order for the inspection to be completed. You may wish to define explicit exit criteria for completing an inspection. These criteria might require that all defects are corrected and issues resolved, or that uncorrected defects are properly documented in a defect tracking system.

Whew! This seems like a lot of structure and a lot of work. Is there a simpler way to review software products? The answer is yes, but experience has shown this formal inspection process to be the most effective way for a group of people to find defects in a software work product.

Clearly, there is a whole spectrum of methods that can be used between the one extreme of a formal Fagan inspection and the other extreme of no review process at all. Most organizations probably do practice one of these less rigid technical review processes. Even if they don’t follow all of the official rules for inspections, any level of software examination by peers of the author will find enough bugs to make it worthwhile.

The term “walkthrough” generally refers to a group activity in which the producer of the artifacts being discussed guides the progression of the review. Walkthroughs are less rigorous than either formal inspections or peer reviews in which the author plays a more passive role. They can easily turn into a presentation by the author, which misses the point of having others less familiar with the product attempt to understand it and find errors in it. After all, if the author was aware of defects, he probably would have corrected them already. Walkthroughs are usually less successful at detecting bugs than are more formal review methods.

Two review approaches that our group has found to be useful are group reviews with individual preparation, and individual peer desk-checks. Our group reviews are not as rigorous as Fagan inspections, but they involve many of the same activities and participant roles, such as individual preparation and the use of a moderator and recorder. We generally bypass the overview meeting and follow-up steps, and we began using checklists only recently. We also have experimented with the use of readers to paraphrase their interpretation of what a program is doing, with encouraging results.

Individual peer desk-checks are inexpensive because only one person besides the author examines the material. This approach can be effective if you have available individuals who are extremely good at finding defects on their own. If someone consistently finds most of the group-identified defects during his own individual preparation step, he may be a candidate for such a role. Table 2 describes the pluses and minuses we have experienced for each of the reviewing techniques we have tried.

Guiding Principles

Every inspection or review process should follow some basic guiding principles.

1. Check your egos at the door. It's not easy to bare your soul and expose your carefully crafted products to a bloodthirsty mob of critical coworkers, and it is easy to become defensive about every prospective bug that is brought up in the meeting. A climate of mutual respect helps ensure that this polarization does not occur. The group culture must encourage an attitude of, "We prefer to have a peer find a defect, rather than a customer." Inspections are then viewed as a non-threatening way to achieve this goal, rather than as an ego-destroying experience. The inspectors should also look for positive things to say about the product.

2. Critique the products, not the producers. The purpose of the inspection is not to point out how much smarter the inspector is compared to the author. The purpose is to make a work product as error-free as possible, for the benefit of both the development team and its customers. The moderator should promptly deal with any behaviors that violate this principle.

3. Find problems during the review; don't try to fix them. It is easy to fall into the trap of amusing and interminable technical arguments about the best way to handle a particular problem. The moderator should squelch these tangents within a few seconds. The author is responsible for fixing defects after the inspection, so just concentrate on identifying them during the meeting.

4. Limit inspection meetings to a maximum of two hours. Our attention spans are not conducive to longer meetings, and their effectiveness decays quickly after this duration. If the material was not completely covered in two hours, schedule a second inspection meeting to complete the task.

   Data in the software literature indicates that slowing the preparation and inspection rates increases the number of bugs found, with the optimum balance around 150-200 lines of code per hour. This rule limits the quantity of material that can be covered in a single inspection to about 8-12 pages of design or text documents, or 300-400 lines of source code.

5. Avoid style issues unless they impact performance or understandability. Everyone writes, designs, and programs a little differently. When a group is beginning to use reviews, there is a tendency to raise a lot of style issues that are probably not defects but rather preferences. Style issues that impact clarity, readability, or maintainability (such as nesting functions five levels deep) should certainly be raised, but discussing whether code indentation should be done in 2-character or 3-character increments should not. The use of coding standards or code beautifiers can eliminate many style issues.

Whenever I see an inspection record form with mostly style issues on it, I am suspicious that actual errors may have been missed. Inspectors should be checking for completeness, correctness, clarity, and traceability to predecessor documents (designs back to requirements, code back to design, and so on). Focus on uncovering errors in logic, function or implementation, not just physical appearance.

6. Inspect early and often, formally and informally. There is a tendency to not want to share incomplete products with your peers for review. This is a mistake. If a product has a systematic weakness, such as a C program that is using literal constants where #defined macros should be used, you want to point this out when a small amount of code has been written, not in a completed 5,000 line program. So long as reviewers know what state the product is in, most people can examine it from an appropriate perspective.

You will probably establish your own conventions for how best to examine various types of material. For example, during a code review, is it better to go through the source listing page by page, or to trace down the hierarchical call tree as you encounter calls to functions within the program being inspected? We have tried both, and opinions vary. I find that I understand the program better, and I also find more interface errors, if I trace through the call tree. You may have to experiment to settle on the best approach for your team, or simply agree to disagree.

Inspecting for Usability

At the Software Development '94 East conference, consultant Lucy Lockwood described a technique developed by her and her partner, *Software Development* columnist Larry Constantine, for "collaborative usability inspections" of software. This method extends the practice of systematic inspection to designed, prototyped, or completed user interfaces. The inspection team includes a lead reviewer (like the moderator), a recorder, the user interface developers, actual users and application domain experts, perhaps a usability expert, and a "continuity clerk." This last participant looks for inconsistencies among different parts of the user interface, as well as for standards violations.

The usability inspection process differs somewhat from that of other software artifacts. Participants are not expected to prepare for the inspection by examining the materials in advance. The best inspection results are obtained by making two passes through the user interface. In the first pass, the interface is "executed" by walking through various usage scenarios. This can be simulated with paper prototypes if an operational interface is not yet available. In the second pass, each of the interface components is examined individually and in detail, independently of how that component fits into a specific usage scenario. Extensive interfaces likely will require several inspection sessions, particularly if many usage scenarios are evaluated.

Lockwood defines a usability defect as a "clear, evident violation or departure from accepted usability principles," which can result in user confusion, delay, or errors. Collaborative usability inspections provide an effective alternative to an expensive usability testing laboratory. They complement the more traditional inspection of the underlying code, which rarely reveals potential usability problems. If you want to explore a variety of other techniques for assessing program usability, check out *Usability Inspection Methods*, edited by Jakob Nielsen and Robert L. Mack (Wiley, 1994).

Keeping Records

Recordkeeping is a major distinction between informal and formal review activities. There are three aspects to this task: recording defects during the inspection meeting; collecting data from multiple inspections; and analyzing the defect trends to assess inspection effectiveness and identify ways to improve your software development process to prevent common types of defects.

Many books contain sample inspection recording forms. *Managing the Software Process* by Watts Humphrey (Addison-Wesley, 1989) has a variety to choose from. Figure 2 shows the technical review summary report form that our group uses; our issues recording form is shown in Figure 3. These are typical of the forms depicted in many books on software inspections.

As inspectors raise issues during the review meeting, the recorder enters them on the issues list from Figure 3. We try to distinguish “defects” from “issues.” Defects are subdivided into the categories missing, wrong, extra, performance, and usability, while issues can be questions, points of style, or requests for clarification. We also try to note the development phase in which the underlying error was introduced (requirements, design, implementation). If you wish, you can classify errors with a very high degree of precision, using tools such as Boris Beizer’s “taxonomy of bugs” from his book *Software Testing Techniques, 2nd ed.* (Van Nostrand Reinhold, 1990), but this simple scheme serves our purposes for now.

In addition, we classify errors found according to a severity scale. Our scale (also used for errors found in testing) includes the categories cosmetic (such as misspelled screen text), minor (nuisance or a workaround exists), severe (some functionality is not available), or fatal (program will probably crash).A simpler method is to just classify defects as major (some product failure is expected) or minor (product may be misunderstood, but the program will work).

Each error found is described in enough detail that the author can refer to this list during the rework step and understand the point that was raised. References to the line numbers where errors were found are important for specific defects, although more general observations about standards violations or style suggestions may apply to the entire product.

The defect list from a single inspection should be distilled down to a summary report with a count of the defects in each category you are using for classification. This summary can be entered into an inspection database, if you are maintaining one. An inspection management report also should be prepared for (guess who) project management. The management report contains information about the material that was inspected and the disposition of the product (accepted with minor changes, etc.), but no actual information about the defects found is included. The purpose is to allow managers to know how the project is progressing and to look for areas where improvements should be made. The moderator usually is responsible for preparing these post-inspection reports.

An effective, ongoing inspection process permits an organization to combine data from multiple inspections to gain insight into the quality of both the review process and the products being reviewed. The ultimate objective is to have a database of inspection data so that quantitative conclusions can be drawn from defect trends and inspection process information. Most organizations are not ready for that level of sophistication at the time they launch an inspection effort.

We use a simple spreadsheet to pool data from inspections of particular types of work products (code in this case). In this spreadsheet, all issues raised in an inspection are tallied, but only items other than style/clarity/question issues are termed “defects.” Only our more recent inspections have the complete set of defect classification data, rather than just the total number of errors detected. The data shown in Figure 4 are representative of our recent experience with both peer desk-checks and team reviews.

This spreadsheet allows us to compute the number of defects found per thousand lines of source code (KLOC), the lines of code inspected per hour, and the defects found per hour of inspection time for each review activity. Our data for 20,000 lines of code in several languages shows an average of 18 defects found per KLOC, an average inspection rate of about 200 LOC/hour, and an average of 3.6 defects found per person hour. In our environment, we found that individual desk-checks by skilled bug hunters are more efficient than many team inspections, in terms of defects found per total number of review hours. One reason for this is that individual desk-checkers do not get sidetracked onto discussions of solutions or other fascinating issues, which so often slow down group reviews. However, not many people are really outstanding at finding bugs on their own, so team reviews are probably superior in most groups.

While results will vary from one organization to the next for many reasons, if you begin recording and analyzing your inspection data, you will be able to determine which methods work best for you and you can begin to assess the quality of your work products. Our data suggests that we still have plenty of opportunities to improve our programming work, but I think the participants in the 23 reviews we held were happy to have 360 defects found by peers, rather than lingering into the final product.

Making Inspections Work for You

An organization's culture of shared beliefs and behaviors has a big impact on whether any inspection process can be implemented successfully. They will be most effective when certain values are shared by most team members:

- A desire to make every work product generated by any team member as defect-free as possible before it passes to the next development stage or to the customer;
- A level of mutual respect, which ensures that problems found are with the product and not the author, and which makes each participant receptive to suggestions for improvement;
- A sense of unease when the author is the only person who has viewed a completed product;
- A recognition that spending the time on quality activities up front will save time for the whole organization in the long run, as well as increasing customer satisfaction by releasing cleaner products in version 1.0.

You can't change your culture overnight, but there are some specific things you can do to increase the chance of successfully initiating inspections in any organization. Keep these tips in mind:

- Provide inspection training for all moderators, and for as many of the other participants as you can afford. Software managers should receive inspection training also, so they can appreciate the value of the process and understand the importance of allocating time in the schedules for people to participate in inspections of other developers' work products.
- Build inspections into the project schedule. Don't forget to factor in time for the inevitable rework that follows a successful inspection.
- Inform the participants and, if appropriate, your customers of the benefits of inspection. Those members of our team who regularly have their work products reviewed by others regard the activity as highly valuable.
- Recognize that the time you devote to inspecting another person's work product is for the benefit of the whole organization, and also that others will assist your project efforts in the same way. This quid pro quo balances the resistance team members may express to taking time away from their own work to review someone else's products.
- Have a local champion who preaches the merits of inspection from his own experience, coaches others as they get started, and strives to improve your inspection processes. However, a champion probably cannot overcome management resistance or disinterest.

How should code reviews and testing interact for maximum effectiveness? There are three possible sequences, depending on how experienced you are in performing inspections and how effective your approach is in finding defects:

1. code/inspect/compile/link/test
2. code/compile/inspect/link/test

3. code/compile/link/test/inspect/retest

Sequence 1 can work if you have a rigorous inspection process in place. This is similar to the Cleanroom software development approach, in which inspections are a dominant verification and validation activity. They are conducted before even attempting to execute the program, in an effort to get the code right on the first try. If you have a pretty good inspection process, sequence 2 might be more appropriate. Let the compiler find the syntax errors it is so good at, but have people inspect the code for logic and problem domain errors before doing structured testing.

If you are just getting started with code reviews, a common approach is sequence 3. Test enough to see that the program mostly works, then review the code, and follow up with regression testing after making any required changes. Adjust your practices as you learn what works best in your environment.

As with any other technology, organizations are not always successful in their attempts to begin software inspections. There are many reasons why they may fail. Try to assess your group's readiness for inspection by considering these inhibiting factors:

- Developers are concerned that inspection results may be used against them at performance appraisal time.
- Authors are unable to separate their egos from their work products, claim that they do not need reviews, or become defensive during the inspection meeting.
- Participants do not prepare adequately prior to the inspection meeting.
- The items being inspected are not documented adequately for a reviewer to be able to understand and critique them (this is actually an inspection result in itself).
- Lack of training leads participants to dogmatically follow the letter of some inspection rule book, rather than seeking the spirit of the activity.
- The author does not take the results of the inspection seriously and make the suggested changes (indicates a need for the follow-up step).
- Inspection meetings lose their focus, wander into interesting discussions about technical solutions, and fail to cover the scheduled material. This can actually be constructive on large, complex projects, or when using a multidisciplinary review team that doesn't ordinarily have a chance to get together, but the emphasis must be on covering the material as planned.
- It is difficult to schedule a time for a review with the desired participants in an appropriate time frame for the project schedule.
- The project schedule doesn't leave room for inspections, so they are cut when time pressures mount.
- If the current development environment does not contain enough pain due to quality problems that inspections might improve, there is little incentive to do them.

I believe almost any group of two or more software developers can improve quality through some form of peer review process. If you don't feel the time is right for doing full-blown formal inspections, begin with individual peer desk-checks through a buddy system. Continue to learn about inspections, and evolve your review activities toward the more formal approaches: record-keeping, defined roles, pre-meeting preparation, written inspection reports, and so on.

Implementing software inspections is an important step along the path to a more mature software development process. In the cycle of continual process improvement that leads to concurrent improvements in both quality and productivity, inspections can play a major role. Peer reviews have become a valuable part of our group's software engineering practice, and they can help you, too.

Table 1. Finding Different Kinds Of Bugs By Code Inspection Or Testing.

Error Type	**Inspection**	**Testing**
Module interface errors	x	
Excessive code complexity	x	
Unrequired functionality present	x	
Usability problems		x
Performance problems	x	x
Badly structured code	x	
Failure to meet requirements	x	x
Boundary value errors	x	x

Table 2. Comparison of Three Reviewing Techniques.

Group Review With Individual Preparation

Pluses	Minuses
1. multiple participants find more defects	1. many defects are found redundantly
2. interactions at meeting time can reveal new defects	2. most defects are found during preparation, not in the meeting
3. more learning opportunities for more participants	3. can get sidetracked on discussions of solutions
	4. cost per defect is high because of multiple reviewers

Formal Inspection, With Individual Preparation

Pluses	Minuses
1. more thorough coverage of documents	1. slower than page-by-page coverage
2. best for code or designs, not necessary for other documentation	2. participants should be trained in roles: reader, moderator, etc.
3. good test of understandability/ maintainability	3. still need to do preparation prior to the inspection meeting
4. find the most defects this way	4. cost per defect is high because of multiple inspectors and slower coverage

Individual Peer Desk-check

Pluses	Minuses
1. cheapest approach: only one reviewer, plus followup time	1. a single person plays all roles: reader, reviewer, recorder.
2. could be more comfortable for the author	2. author is not there to answer questions and hear the discussion; must have a follow-up session with author to explain findings
3. works well if you have someone who is very good at finding defects; otherwise, use a group	3. no group synergy; errors found will be those that the brain of the one reviewer is best at spotting

Figure 1. Sample checklist for code inspections.

Structure

- Is the compilation listing free of warning messages?
- Does the code properly implement all elements of the design as specified?
- Are there any uncalled or unneeded procedures?
- Can any code be replaced by calls to external reusable components or library functions?
- Do processes occur in the correct sequence?
- Will requirements on execution time be met?
- Is the code well-structured, consistent in style, and consistently indented?
- Are there any blocks of repeated code that could be condensed into a single procedure?
- Is storage use efficient?
- Are symbolics used rather than “magic number” constants or string constants?
- Are there good reasons why any procedures having extended cyclomatic complexity greater than 15 should not be restructured or split into multiple routines?

Documentation

- Is the code clearly and adequately documented (both file prologues and module headers)?
- Are there any inconsistencies between code and comments?
- Is the commenting style easy to maintain?

Variables

- Are all variables properly defined and given meaningful, consistent, and clear names?
- Do all assigned variables have proper type consistency or casting?
- Are loop index variables more meaningful than *i* and *ii*?
- Are there any redundant or unused variables?
- Are all array references in bounds?

Arithmetic Operations

- Does the code avoid comparing floating-point numbers for equality?
- Does the code systematically prevent rounding errors?
- Does the code avoid additions and subtractions on numbers with greatly different magnitudes?
- Will all mixed-mode arithmetic operations give the correct results? Should type conversions be done in such cases?

Loops and Branches

- Are all loops, branches, and logic constructs complete, correct and properly nested?
- Are the most common cases tested first in IF-THEN-ELSEIF chains?
- Are all cases covered in an IF-THEN-ELSEIF or CASE block, including ELSE or DEFAULT clauses?
- Does the normal case follow the IF, not the ELSE?
- In C, does the end of each case in a CASE statement have a *break*?
- Does every switch statement have a default?
- Are loop termination conditions obvious and invariably achievable?
- Are indexes or subscripts properly initialized, just prior to the loop?

- Can any statements that are enclosed within loops be placed outside the loops without computational effect?
- Are loops nested to three levels or less?
- Does the code in the loop avoid manipulating the index variable or using it upon termination of the loop?

Defensive Programming

- Are indexes, pointers, and subscripts tested against array, record or file bounds?
- Are imported data and input arguments tested for validity and completeness?
- Are divisors tested for zero or noise?
- Are all output variables assigned?
- Are the correct data being operated on in each statement?
- Is every memory allocation being deallocated?
- Are timeouts or error traps used for external device accesses?
- Are files checked for existence before attempting to access them?
- Are cleanup routines used to make sure all files and devices are left in the correct state upon program termination?
- Where multiple exit points exist in a routine, does each exit execute the required cleanup routines?

Figure 2. Inspection Summary Report

Inspection Identification:

Project:	CARAT
Inspection Number:	CODE-3
Date:	11/4/94
Time:	10:00 AM
Location:	Room 352, Building 59

Product Identification:

Material Inspected:	Source code for the manifold housing coupler linkage controller and datalogging system
Developer:	Fallacious Wiscenowskowicz
Customers:	Gem exploration team

Inspection Team

		SIGNATURE
1. Developer:	Fallacious W.__________	______________________________
2. SQA Coordinator:	Karl Wiegers__________	______________________________
3. Extra inspector:	Sir Findsalot__________	______________________________
4. Customer:	______________________	______________________________
5. Other:	Diamond Gym_________	______________________________
	Ruby Lipps____________	______________________________

Product Appraisal

ACCEPTED (*no further review)*
___ as is
___ with minor revision

NOT ACCEPTED *(new review required)*
___ major revision necessary
___ rebuild
___ Inspection Not Completed

Pages Scheduled to be Inspected: _____ Pages Actually Inspected: _____
Lines of Code: ___________
Preparation Time: ___________ hours

Figure 3. Issues List

Project:
Inspection Number:
Inspection Date:
Recorder:

Type:	**R**equirements, **D**esign, **I**mplementation, etc.
Class:	**M**issing, **W**rong, **E**xtra, **S**tyle, **C**larity, **Q**uestion, **U**sability, **P**erformance
Severity:	**C**osmetic, **M**inor, **S**evere, **F**atal

	Type	Class	Severity	Description
1.	______	_______	______	______________________________
				______________________________
2.	______	_______	______	______________________________
				______________________________
3.	______	_______	______	______________________________
				______________________________
4.	______	_______	______	______________________________
				______________________________
5.	______	_______	______	______________________________
				______________________________