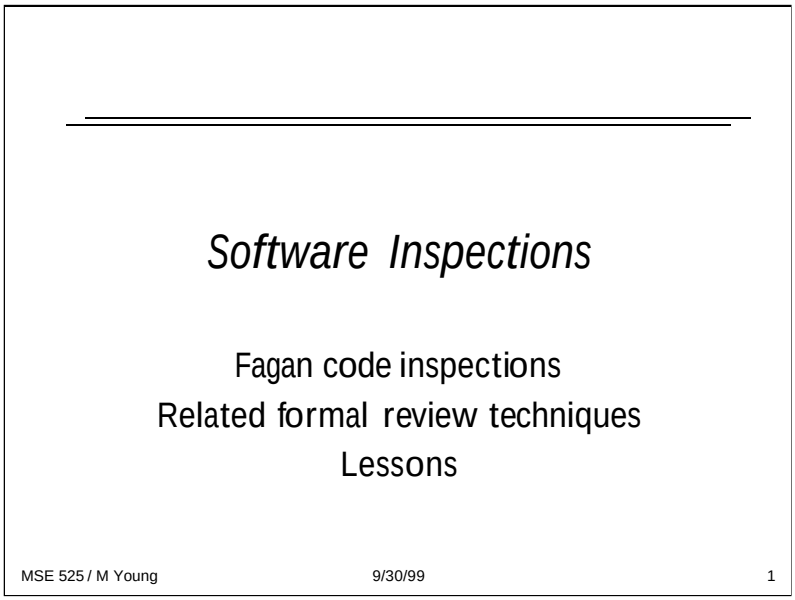


One of the simplest and most general ways to check a work product* is to manually inspect it. Inspections are common practice in many engineering disciplines. For example, a building must be inspected and approved at several stages of design and construction.

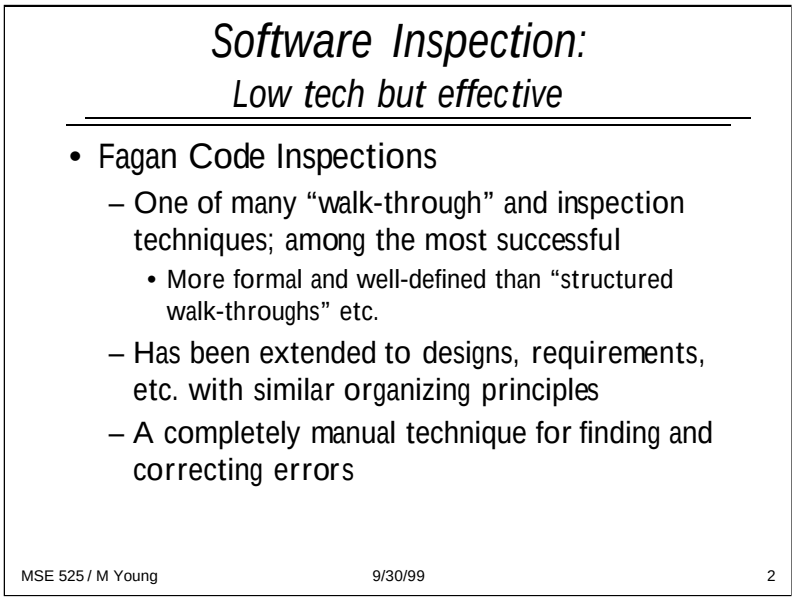
Several inspection techniques have been developed and applied to software development, ranging from relatively unstructured discussions to inspections with detailed, strictly prescribed procedures. Among the most influential and widely used are the formal inspection techniques developed by Michael Fagan at IBM.

Inspections can be applied to any work product, from requirements to test plans to user documentation; this is one of their main strengths. Some organizations inspect almost every work product *except* code, which is tested instead. However, much of the research literature is based on code inspection. There are several reasons for this. For one thing, while other work products (e.g., requirements specifications) vary widely among organizations, coding tends to be constrained to perhaps a half-dozen popular programming languages and a few dominant styles, so code inspections can be described with examples that make sense to most readers. For another thing, it is easier to perform empirical studies of the effectiveness of code inspections (e.g., by by showing their effect on the number of faults found in testing or reported by users) than to devise valid empirical studies of the effectiveness of other kinds of inspection.

We will look first at Fagan code inspections, and then at some related inspection techniques; then we will try to draw some general lessons that are applicable not only to inspection techniques but also to other analysis and testing techniques.

---

*A "work product" is anything produced as the output of some work activity, including intermediate products (like test plans) as well the delivered product.



There are many review and "walk-through" techniques for software development. The inspection process developed and publicized by Fagan is distinguished from earlier approaches in that it is a very detailed, formalized process. (Fagan called them "software inspections;" others call them "Fagan inspections" to distinguish them from other software inspection techniques).

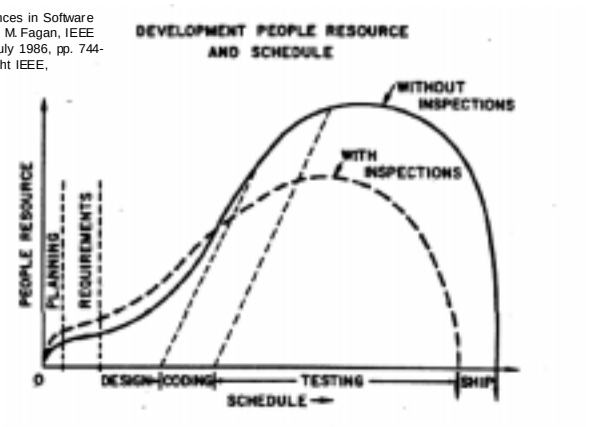
Fagan inspections can be applied to any work product that takes the form of a document, but like many inspection techniques, most of the available research literature describes inspections of source code, and that is what we will review.

As described by Fagan, software inspections are a completely manual technique for detecting and removing defects. A "defect" may be a fault ("bug") that would result in a user-observable behavior that deviates from the software specification, but it can also be a deviation from a specification that is completely invisible to the user. For example, if there is a coding standard that requires every variable declaration to be followed by an explanatory comment, then omission of the comment is a "defect" even though it has no effect on program behavior. Thus inspections can find classes of defects for which dynamic testing is not applicable. If the standard required the comment to be meaningful, a comment that was present but not meaningful would also be a defect. This illustrates that, being a manual process, inspection can rely on human judgment to an extent that more automated testing and analysis techniques cannot.

It is interesting to consider inspection, among other things, as a "base case" for judging the effectiveness of automated analysis and testing techniques. If I propose a new automated analysis for locating a particular kind of fault, then a minimum standard of effectiveness is that the automated analysis alone, or in combination with some inspection, ought to be more cost-effective than inspection alone. This is a surprisingly difficult criterion to satisfy, although many of the comparison studies have used only very primitive testing techniques.

Fagan's Schedule Argument

From: "Advances in Software Inspections," M. Fagan, IEEE TSE 12(7), July 1986, pp. 744-751. Copyright IEEE,



MSE 525 / M Young 9/30/99 3

The argument for inspections often involves an assertion that time invested earlier is repaid later in lower testing and rework costs. This "snail curve" (which has no scale and is not drawn from any particular data) argues that the coding and testing phases of a project will be started later, but finished earlier if inspections are used in each prior stage. Coding is shown as taking the same amount of time; testing is shortened because there are fewer errors to be found and repaired.

Note that this curve is based on a waterfall model (e.g., testing takes after design and coding are finished). Similar argument might be made for other models of development, but iteration (developing several versions of the same component) might cause problems.

Software Inspection Roles

- Moderator:
  - Typically borrowed from another project. Chairs meeting, chooses participants, controls process
- Readers, Testers:
  - Read code to group, look for flaws
- Author:
  - Passive participant; answer questions when asked

MSE 525 / M Young 9/30/99 4

The software inspection process is carried out by a group of people, typically 3 to 5, with well-defined roles. These are the moderator, the readers and testers, the author, and (sometimes) a recorder or scribe. The readers and testers both read the code, but testers are supposed to look at it as if they were testing it.

The moderator is in charge of the process. A moderator is typically a fairly senior technical member of another team (to avoid bias). Fagan and others stress that the moderator role is critical, and moderators must receive training to carry out their responsibilities. Among these responsibilities are:

* Determine whether the entry criteria (see next slide) have been met; otherwise the inspection is canceled.

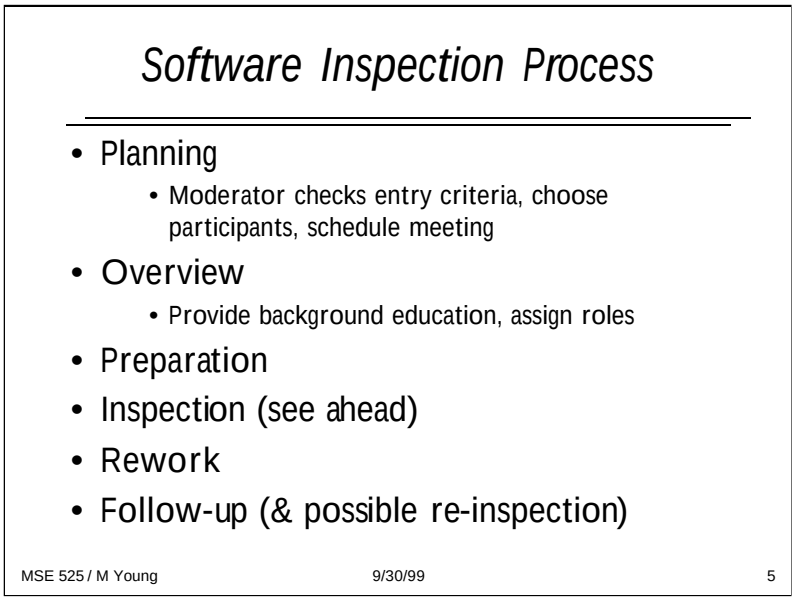
* Choose the other inspection participants.

* Chair and manage the meeting, and keep it on track. Given the potential for personal conflict, this requires some skill.

* Review re-work and determine whether the product should be re-inspected.

The main task of readers and testers is to systematically read the code, looking for defects.

The author is a passive participant in the inspection process. He may answer questions when asked, but he is not permitted to offer explanation or defend the code. Another way of putting this is: It is not enough for the code to be correct, it must be written and documented in a way that makes its correctness obvious to the readers.



The software inspection process is repeated once for each document to be inspected, e.g., for each individual code unit assigned to a particular programmer. An inspection for a particular kind of document has entry criteria that must be satisfied before an inspection can be scheduled. For example, it is typical to require that code be in the "first clean compile" stage, that is, it must already have been compiled without syntactic errors. A particular organization might impose additional entry criteria, such as completion of certain kinds of documentation for maintainers and the inspection team. The moderator is responsible for checking the entry criteria and, if they are satisfied, beginning the remainder of the process by selecting participants (readers and a scribe) and scheduling the meeting.

Participants receive an general background presentation on the material to be inspected in an overview phase; roles for the inspection meeting may also be assigned at that time. They are provided with any reference materials that they may need for the inspection, such as design documents. They are then required to study the material in preparation for the inspection meeting.

The inspection meeting itself is described on the next slide.

After the meeting, the author receives a list of defects to be corrected. After these are (purportedly) corrected, the author asks the moderator to check them individually. The moderator may either conclude that the defects have been satisfactorily repaired, or she may determine that the changes are extensive enough that the inspection process should be repeated for the modified code.

In the Meeting

- Goal: Find as many faults as possible
  - max 2 x 2 hour sessions per day
  - approx. 150 source lines/hour
- Approach: Line-by-line paraphrasing
  - Reconstruct intent of code from source
  - May also "hand test"
- Find and log defects, but don't fix them
  - Moderator responsible for staying on track

MSE 525 / M Young 9/30/99 6

The purpose of the inspection meeting is to find as many defects as possible in a given amount of time. Because an inspection meeting is fatiguing, and a tired inspector is less effective, the time for an inspection meeting is limited to two hours, and a person may participate in at most two such meetings in a given day.

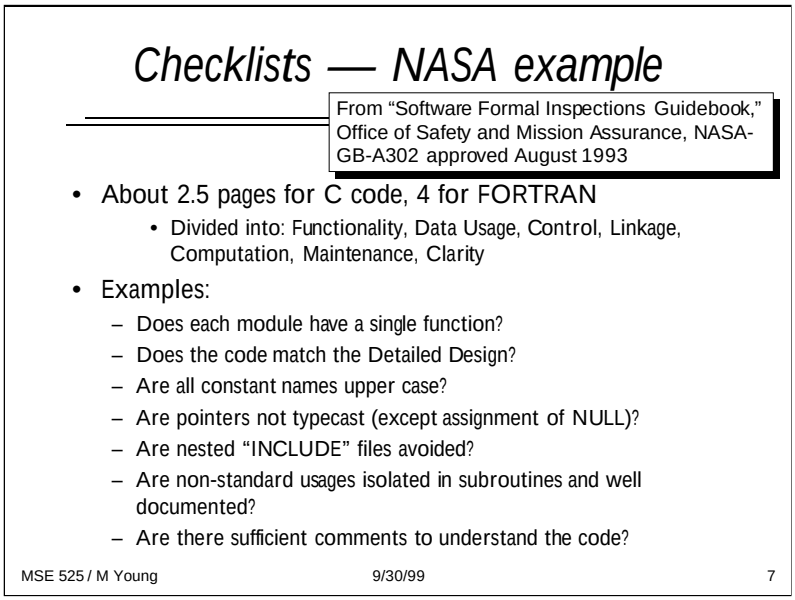
In addition, the rate of inspection is limited. An inspection rate of 150 lines per hour is generally recommended. The defect detection rate reported by Russel is approximately 1 defect per person-hour (4 defects per hour for a 4-person team) *independent of reading rate*.*

The main approach used during the meeting is line-by-line reading and paraphrasing of the code. In other words, the readers should be able to roughly reconstruct detailed pseudocode and detailed design from the actual code. Additionally, inspectors may "hand test" the code by simulated execution, and at the same time they may be checking for several potential problems from a check-list.

A key rule for the inspection meeting is that defects are logged, but not corrected. No participant is permitted to suggest how a defect might be corrected, and a key responsibility of the moderator is to prevent the inspectors to be diverted from finding defects to discussing fixes and improvements. One reason for this is that a correction arrived at in the meeting is likely to inferior to one found later, with more time for consideration of alternatives.

———

* It is interesting to note that similar "fixed rate" phenomena seem to occur in software development. For example, programmer productivity as measured by lines of code per hour or day seems to be independent of whether those are lines of assembly code, conventional programming language code (Java, C++, etc.), or very-high-level code for an application "generator" system.]

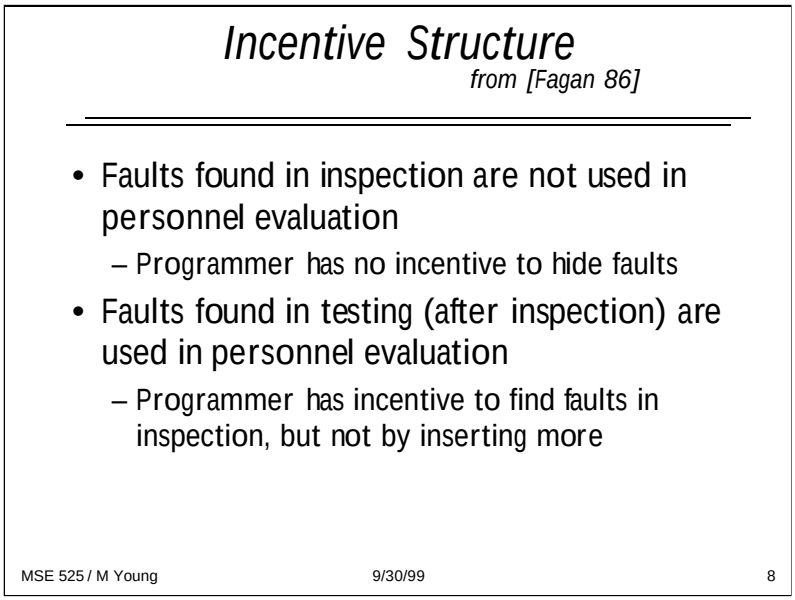


Here are some example items from a checklist published by The National Aeronautics and Space Administration (NASA). The guidebook includes checklists for a variety of artifacts (design documents, requirements, etc.). Checklists for source code are given separately for each source language.

Some of the checklist items are general and require a good deal of human judgment. One could not very well determine whether each module has a single function by testing, and certainly inspection is the only technique that can answer the question about the sufficiency of comments for understanding the code. On the other hand, checks for naming standards, pointer typecasting, and nested #include files require little or no judgment, and could be either augmented or replaced by mechanical checks.

Some of the items are checks for particular faults that tend to occur in a particular language. The NASA checklist for C has mostly general trouble-avoidance standards for C (like not typecasting pointers or over-using the int type), while the FORTRAN checklist contains more particular faults to check for, like use of the letter O in place of the digit 0.

These checklists are a good example of a characteristic of Fagan inspections that Knight and Myers [93] consider a problem: They try to uncover many different kinds of defects at once, and use the same (expensive) set of people for all kinds of defects. Knight and Myers' "Phase Inspection" technique (later in this lecture) is a variation designed to address this problem.



In any defect-detection technique, we must be careful to avoid perverse incentives, and if possible to introduce positive incentives. An example of a perverse incentive would be to reward developers for finding more faults during testing, because one way to find more faults is to make sure there are more faults to find. A more subtle variation of the perverse incentive problem occurs when the main pressure on developers is to meet a deadline, and the developers themselves are responsible for determining whether quality is sufficient to declare part of a project "done."

Fagan describes the following measures to avoid problems in the incentive structure:

* Faults found in the inspection are not used in evaluating the author of the code. Programmers are neither (directly) rewarded nor punished for reviews that find more faults. We have already remarked on the perverse incentive that would occur if programmers were rewarded when inspections found more faults. If, on the other hand, programmers were rewarded when inspections found fewer faults (or punished when they found more), there would be an incentive to be uncooperative in the review.

* Faults found after inspection, in testing, are used in personnel evaluation. This way, the programmer has an incentive to remove as many faults as possible before testing begins, and therefore to make inspections as effective as possible. (Of course, there is a danger of perverse incentive in the testing phase, if the programmer tests his own code or can reduce the effectiveness of an independent tester by being uncooperative.)

Variation: Active Design Reviews

D. Parnas & D. Weiss, "Active design reviews: Principles and practices." Proc. ICSE , Aug. 1985, pp. 132-136.

- Observation:
  - An unprepared reviewer can sit quietly and say nothing.
- Variant process:
  - Choose reviewers with appropriate expertise
    - Several reviewers to look at different aspects
  - Author asks questions of the reviewer
  - Reviewer's job is to answer the questions

MSE 525 / M Young 9/30/99 9

In their article "Active Design Reviews: Principles and Practice," Parnas and Weiss describe several problems with conventional reviews (some of which do not apply to Fagan inspections) and then describe a review process designed to avoid those problems. The characteristics of this process are (quoting from the abstract of the paper):

(1) The efforts of each reviewer should be focused on those aspects of the design that suit his experience and expertise.

(2) The characteristics of the reviewers needed should be explicitly specified before reviewers are selected.

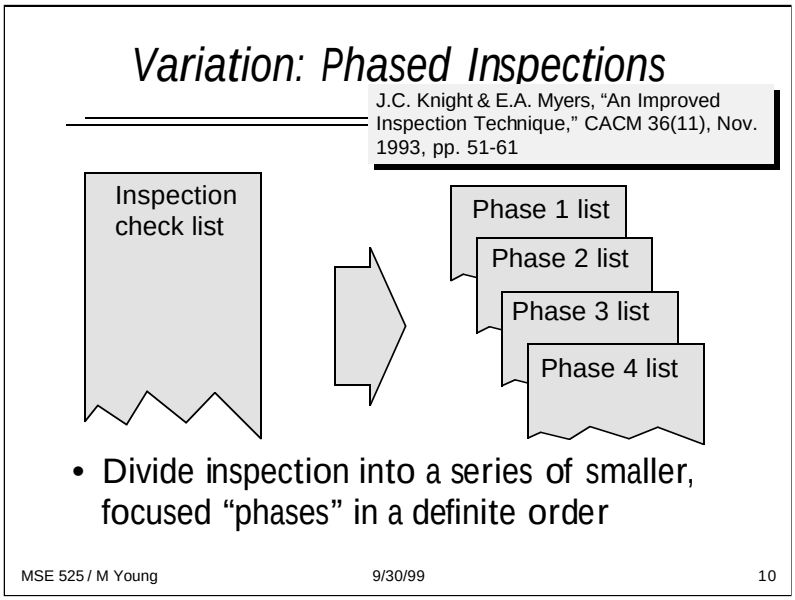
(3) Reviewers should be asked to make positive assertions about the design rather than simply allowed to point out defects.

(4) The designers pose questions to the reviewers, rather than vice versa. These questions are posed on a set of questionnaires that requires careful study of some aspect of the design.

(5) Interaction between designers and reviewers occurs in small meetings involving 2-4 people rather than meetings of large groups.

Note that (5) was already the case for Fagan inspections, but not for some earlier and less structured review techniques; to some extent (3) is also a feature of Fagan inspections, depending on the form of checklist questions. (1) and (2), in addition to (4), are definitely not characteristics of Fagan code inspections.

The most striking characteristic of active design reviews is, obviously, reversing the customary question/answer roles between the author and the reviewers. The inspector cannot get away with skimming the document and then just keeping quiet in the meeting; he is forced to read the document closely enough to answer the questions that have been posed by the author. An example question is: "Under what conditions may this function be applied? Which assumptions describe those conditions?"



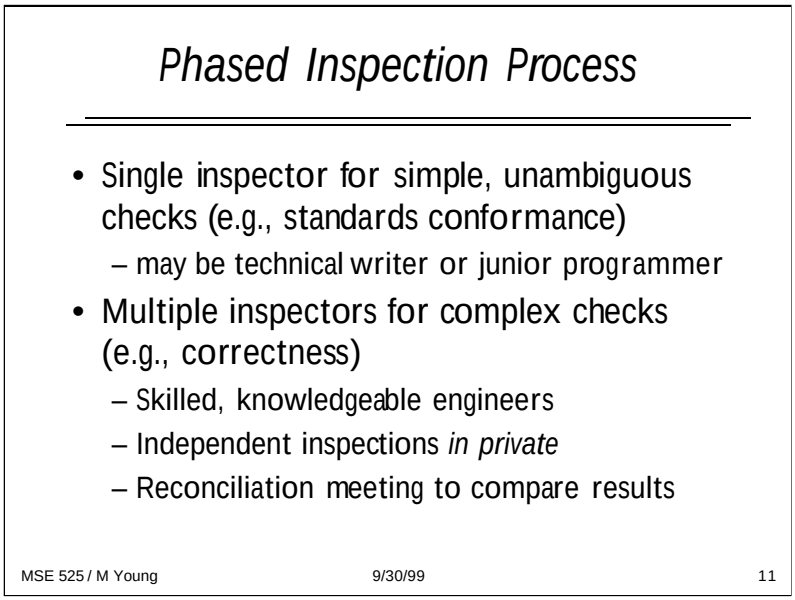
Knight and Myers [93] observed that software inspections try to find many different kinds of defect at once. This has at least two disadvantages:

* It is difficult to look for everything at once. In practice, inspectors tend to concentrate on functional correctness and to pay too little attention to qualities like maintainability, portability, etc.

* The same (expensive) team of people is used for trivial checks like formatting conventions as for the difficult checks that really require their judgment and training, which is not a good use of highly trained people.

The "phased inspection" process that Knight and Myers proposed divides defect detection into a number of smaller phases, each concentrating on just one property or closely related set of properties. The phases are ordered so that each inspection can assume completion of prior phases.

[Reference: J.C. Knight & E.A. Myers, "An Improved Inspection Technique," Communications of the ACM 36(11), Nov 1993, pp 51-61. Knight and Myers also discuss other problems, such as consistency across different people performing reviews, which are not discussed in these slides. Likewise, some features of active design reviews are adapted in phased reviews, but not discussed here.]



In the phased inspection process, some phases are carried out by a single person, and others are carried out by a team.

When a property is simple and can be carried out more-or-less mechanically without much judgment, it is checked in a single inspector phase. One person, who may be a junior programmer or even a non-programmer (e.g., a technical writer) is assigned to perform the check individually. An example from the NASA checklist of a property which could be checked in this way is "are all constant names upper case?." As Knight and Myers also point out, some of these can be automated so that they don't require a manual check at all, but there are likely to be some conceptually simple checks for which no automatic checking tool is available.

Properties that are more complex, including checks for functional correctness as well as the "ilities" (portability, maintainability, etc.) typically require a multiple inspector phase, and these inspections must be carried out by more skilled and knowledgeable developers. Unlike the Fagan inspection technique, the actual inspection is carried out by each inspector individually; then the inspectors meet to compare and "reconcile" their results. In theory, no new problems should appear in the reconciliation meeting. In practice, Knight and Myers report that some problems are noticed only during the reconciliation, for example because a discrepancy between the results of two inspectors leads to further consideration of something.

An example list of phases and inspectors given by Knight and Myers is:
Phase 1: Internal documentation standards, single technical writer
Phase 2: Layout (formatting) standards, single technical writer
Phase 3: Readability, meaningful identifiers, etc; single junior programmer
Phase 4: Good programming practices; single software engineer
Phase 5: Check for common bugs; single software engineer
Phase 6: Check for correctness; multiple senior software engineers

Inspection Automation

- Although a manual technique, many kinds of automated support are possible:
  - Automate trivial checks (e.g., formatting)
  - Reference: Checklists, standards w/ examples
  - Focus (highlight, selection) on relevant parts
  - Annotation & Communication
  - Process guidance and (partial) enforcement
    - e.g., InspeQ will not allow check-off until all relevant parts of a document have been observed

MSE 525 / M Young 9/30/99 12

Although inspection is basically a manual technique, there are many ways in which computer support can make it more efficient.

Some checks really should not be manual if proper tool support is available. For example, the NASA checklists include proper indentation of C code. There are many tools that will perform this indentation automatically, as well as capitalization, etc. Other properties cannot be imposed by a tool, but can be checked. The NASA checklists also include checks for undeclared variables in FORTRAN, which are treated by the compiler as references to external functions (C has a similar problem). Automated tools (e.g., Lint for C) can easily check for these.

Other checks must be manual, but computer support can help. Knight and Myers provide several kinds of support through their InspeQ tool:

* They provide display of checklists and standards, in a form suitable for use during inspection (remember that their inspections are always carried out by individuals, not in a meeting, even when multiple inspectors are involved).

* Source code navigation and display tools can focus on parts of the code relevant to a specific checklist item, e.g., showing all “while” loops if the inspector is checking for loop termination.

* The inspector can log defects and notes in the form of annotations on the document being inspected.

* The inspection process (e.g., phase ordering) is partly enforced; although the tools must believe the inspector, they can at least force some things to be checked off before others are attempted.

Some other work on inspection addresses collaboration aspects. Computer support for collaborative work, also known as groupware, is an active research area now, and inspection is a classic example of the kind of work it is designed to support.

Why does inspection work?

- The evidence says it is cost-effective. Why?
  - Detailed, formal process, with record keeping
  - Check-lists; self-improving process
  - Social aspects of process, esp. for author
  - Consideration of whole input space
  - Applies to incomplete programs
- Limitations
  - Scale: Inherently a unit-level technique
  - Non-incremental; what about evolution?

Of all analysis and testing techniques for software, the empirical evidence for the cost-effectiveness of formal inspections is strongest. In one sense, that is depressing; surely we ought to be able to do better than this labor-intensive, manual review process. A more constructive reaction is to look carefully at why inspection seems to work so well, and see what we might be able to apply to other analysis and testing techniques.

Features of software inspections that seem to contribute to its effectiveness include:

* A detailed, formal process. Detailed instructions for carrying out a process seem to have inherent value because they promote consistency and rigor, almost independent of the particular process.

* Process feedback, in the form of check-lists that are based at least partly on defects that have been detected in the past.

* Social pressure on the author to produce documents that will not be embarrassing in the inspection. There is evidence that programmers produce better code when they know it will be inspected. The peer review process also serves as a kind of mutual education, in which developers learn from both the good and bad features of each other's code.

* Code inspection (like some other static analysis techniques) considers the whole input space of the program, not a set of particular sample inputs.

* Inspections can be applied to code before there is any running system; it can be applied to modules and subsystems without "stubs" and "drivers."

On the other hand, the limited speed of inspection makes it essentially a unit-level technique. Perhaps more important is that it is not incremental. Most code is not written from scratch, but is rather a series of small changes to a large system. The cost of inspection is proportional to the size of the document being inspected, rather than the size of the change.