An Improved Software Inspection Technique
and
An Empirical Evaluation of Its Effectiveness

E. Ann Myers
John C. Knight

Computer Science Report No. TR-92-15
May 28, 1992

Submitted to the *Communications of the ACM*.

AN IMPROVED SOFTWARE INSPECTION TECHNIQUE AND AN EMPIRICAL EVALUATION OF ITS EFFECTIVENESS†

E. Ann Myers *John C. Knight*

Department of Computer Science
University of Virginia
Thornton Hall
Charlottesville, VA, 22903

(804) 924-7605

ABSTRACT

Inspection of software work products is common practice and has been shown to be a valuable tool for the software engineer. However, we believe that the technology is not being exploited as fully as possible. We define an enhanced inspection technique called *Phased Inspection* that addresses the deficiencies of existing inspection techniques. This technique is designed to permit the inspection process to be rigorous, tailorable, efficient in its use of resources, and heavily computer supported. The Phased Inspection process is designed to permit the engineer to trust the results of a specific inspection and to ensure that inspection results are repeatable. Phased Inspections inspect the work product in a series of small inspections termed phases each of which is designed to ascertain whether the work product possesses some desirable property. The skills of the staff performing a phase are tailored to the goals of the phase, and the checking that is performed during a given phase is defined precisely and computer supported. An important goal of Phased Inspection is computer support and we present details of a toolset that supports Phased Inspection by providing the inspector with extensive computer assistance and by checking for compliance with the required process. A preliminary evaluation of Phased Inspection is also presented.

Keywords and Phrases: Software inspections, reviews, walkthroughs.

---

† Sponsored in part by NASA grants NAG-1-1073 and NAG-1-1123, in part by SAIC Inc., in part by the MITRE Corporation, and in part by the Virginia Center for Innovative Technology grant number CAE-92-003.

1. INTRODUCTION

Software reviews are not a new idea, they have been around almost as long as software. One of the most natural ways to check if something is correct is to look at it. Babbage and von Neumann both regularly asked colleagues to examine their programs [6]. In the 1950's and 1960's, large software projects often included some sort of software reviews, and in the 1970's these review mechanisms began appearing in publications. By the mid to late 1970's, various review methods had emerged with different names: software reviews, technical reviews, formal reviews, walkthroughs, structured walkthroughs, and code inspections. Each review method had different forms to fill out, different review team sizes and makeup, etc., but none suggested any approach for reviewing the software or other work product other than just looking at it and discussing it.

One might wonder why reviews are used at all since most software is tested anyway. There are several reason for doing something other than testing, including the the expense of testing and its insufficiency. Linger et al [9] state "It is well known that a software system cannot be made reliable by testing." Similarly, in support of inspections in engineering, Petroski states in his text *To Engineer Is Human* [12, p52]:

> "Engineers today, like Galileo three and a half centuries ago, are not superhuman. They make mistakes in their assumptions, in their calculations, in their conclusions. That they make mistakes is forgivable; that they catch them is imperative. Thus it is the essence of modern engineering not only to be able to check one's own work, but also to have one's work checked and to be able to check the work of others."

Since independent inspections are routine in many other disciplines, for example, financial accounting and building construction, it is surprising that inspection is not a significant element of all software development.

Empirical evidence has emerged showing that review methods based on human examination of a paper version of a work product can have considerable benefit, usually by lowering the number of errors in the software. Freedman and Weinberg [5] report that in large systems, reviews have reduced the number of errors reaching the testing stages by a factor of ten. This reduction cut testing costs by 50 - 80% including review costs. Fagan, referring to results compiled by Russell [13], states that "65 - 90% of operational defects are detected by inspection at 1/4 to 2/3 the cost of testing and removed at 1/7 - 1/2 the cost." [4]. Despite their demonstrated performance, existing review methods are far from universally accepted.

Although successful, existing review methods have several limitations. For example, existing review methods are not rigorous and are far too dependent on human effort. The lack of rigor means that, although existing review methods are cost effective statistically and generally beneficial to software development, they do not ensure that a *particular* work product has any clear-cut quality after review. The general dependence of review methods on human effort is unnecessary. Supplementing the review process with computer resources permits far more efficient use of human time and more complete coverage of items that have to be reviewed.

In this paper we describe an enhanced technique for the inspection of software work products called *Phased Inspections*. This technique is designed to permit the inspection process to be rigorous, tailorable, efficient in its use of resources, and heavily computer supported. Phased Inspections inspect the work product in a series of small inspections termed phases each of which is designed to ascertain whether the work product possesses some desirable property. The skills of the staff performing a phase are tailored to the goals of the phase, and the checking that is performed during a given phase is defined precisely and computer supported.

As well as describing an enhanced review process for software engineers to follow, we also present details of a comprehensive toolset to support Phased Inspections. The toolset contains extensive facilities that assist the inspector thereby allowing inspections to proceed rapidly. The toolset also supports enforcement of the process thus ensuring that inspections are carried out as required.

Since it is not sufficient merely to claim benefits for a new process, we also present a framework for evaluation of Phased Inspections and the results of a preliminary experimental evaluation.

2. EXISTING REVIEW METHODS

In the 1950's and 1960's many large software projects included some form of software review in the development process, but it was not until the work of Weinberg appeared in 1971 [16] that the review of software in all stages of development was advocated and a method proposed. Since that time, review methods have appeared frequently in the literature. These review methods can be placed into one of three general categories characterized by the strategy that drives the review process:

(1) *Formal Reviews*
In a formal review, the author of the software or one of the reviewers familiar with the software introduces the software to the rest of the reviewers. The flow of the review is driven by the presentation and issues raised by the reviewers.

(2) *Walkthroughs*
Walkthroughs are usually used to examine source code as opposed to design and requirements documents. The participants do a step-by-step, line-by-line simulation of the code. The author of the code is usually present to answer any questions the other participants might have.

(3) *Inspections*
In an inspection, a list of criteria that the software must satisfy determines the flow of the review. While walkthroughs and formal reviews are generally biased towards error detection, inspections are often used to establish other properties such as portability and adherence to standards [6]. A reviewer may be supplied with a checklist of items, or he may only be informed of the desired property. Inspections are also used to check for particular coding errors that have been prevalent in the past.

One of the most popular review methods was developed by Fagan [2, 3]. Fagan wanted to create a new review process that would improve software quality and increase programmer productivity. His method, informally known as *Fagan Inspections*, is a combination of a formal review, an inspection, and a walkthrough. This combination of review methods has made Fagan Inspections somewhat more formal and therefore more effective than previous methods.

Fagan's inspection method consists of five steps: overview, preparation, inspection, rework, and follow-up. In the overview, the author of the software explains the design and the logic of the software to the inspectors. During preparation, the inspectors study the software and any design documentation to prepare for the inspection. The inspection is controlled by a moderator, who in turn chooses a reader. The reader guides the inspectors through the work product in a detailed examination searching for errors. Every line of code is examined. A report of the inspection is prepared and given to the author who corrects the errors that were identified. The follow-up step checks that the errors were corrected.

Active Design Reviews are an important advance in review methods introduced by Parnas and Weiss [11]. The approach taken is to conduct several brief reviews with each focusing on a part of the work product (usually a design) rather than one large review thereby avoiding many of the difficulties of conventional reviews cited by Parnas and Weiss. In addition, participants in Active Design Reviews are guided by a series of questions posed by the *author(s)* of the design so as to encourage a thorough review. Some of the ideas in Active Design Reviews have been adapted for Phased Inspections.

The Cleanroom approach to software development is far more than a review method although human review of work products is a major component of the technique [1, 14]. The Cleanroom process, however, forces the author(s) to perform what amounts to an unsystematic review of a work product by not permitting them to execute the software artifacts. In some cases, even compilation of software is not permitted. The reason for this approach is the belief that the formalized methods that Cleanroom embodies obviates the need for anything other that functional testing.

Although the formalism of Cleanroom is effective in many ways, we find the restrictions on execution to be unnecessarily restrictive, even counter productive. Execution of software by the author under appropriate circumstances is often extremely supportive of good engineering. We offer rigorous inspection in the form of Phased Inspections within a complete development lifecycle as a much more effective way of achieving some of the goals that Cleanroom seeks to address.

3. DEFICIENCIES OF EXISTING METHODS

Although existing methods are very successful, careful examination of their application in practice reveals various limitations. Clearly, no single method suffers from all of the limitations we identify. We note specifically that Active Design Reviews [11] suffer from relatively few. The following is an accumulation of limitations from various techniques:

- Despite efforts by their developers to make them more general, existing methods tend to focus on error detection [2, 11]. Error detection is important, but correctness is not the only desirable characteristic of software products. Maintainability, portability, and reusability are examples of other characteristics with which a review method might be concerned. These other characteristics are important since, for example, a software product might have no errors but its value might be drastically reduced if it is not maintainable. Such characteristics are sufficiently complex that their determination by inspection cannot be effected by a single, general-purpose inspection as is attempted with existing methods.

- In general, existing review methods are not consistent. As noted above, although they are beneficial in a statistical sense, existing methods do not ensure that a *particular* work product has any specific quality after review. A project manager can usually say only that reviews improve the general quality of his organization's products. This is a serious limitation. Managers should be able to make assumptions about qualities held by a particular product after review. In order to make reviews dependable, it must be possible to assert, either with certainty or with high probability, that a product which has been reviewed has certain properties. This means that the review process must be rigorous. Rigor permits conclusions to be drawn about a property of a product, and allows these same conclusions to be drawn about every product that is inspected. Equally important, rigor also allows the same conclusions to be drawn about a product irrespective of who is performing the review.

- Existing methods make ineffective use of human resources. It is not uncommon for highly paid software engineers participating in a review to debate spelling, comment conventions, and like trivia. Also, reviews are group activities and as such are susceptible to dominance by a single strong-willed individual. Others may have useful comments but are inhibited in such situations. In addition, a group activity in which there is no detailed, required, active participation by each member permits individuals who failed to prepare to sit quietly, not contribute, and for this to go largely unnoticed.

- There are many different types of error that a software product might have. For example, there might be errors in the logic, the computations, or the tasking structure; there might be unacceptable inefficiencies; or there might be errors in the form of omitted functionality. In an inspection that follows traditional practice, the product is examined once, and it is expected that errors of all types will be checked for during this single examination. Although the participants in a traditional inspection might be experts in appropriate different areas, the inspectors are required to be checking for all the different types of error simultaneously. It is unlikely that they will be able to meet this intellectual challenge.

- Existing review methods target paper products for examination and perform examinations typically in a meeting. Little to no computer support is used thereby exposing the process to risks of incompleteness and making poor use of human resources.

- Certain elements of existing methods are inappropriate. The overview step included in many review methods, for example, is quite inappropriate. It suggests strongly that the documentation of the product being reviewed is deficient in some way, perhaps even missing. If the documentation is complete *and properly presented*, an overview should not be necessary.

Active Design Reviews address some of these issues. By addressing these limitations systematically and building on the positive elements of existing methods, we aim to improve inspection technology. As we document in sections 6 and 7, we have been partially successful. We also have clear indications of how to increase the degree of improvement.

4. PHASED INSPECTIONS

We believe the benefits of inspections to be so great that they should be a required component of the creation of every work product in the software lifecycle. Further, we believe that for inspections to achieve their maximum cost effectiveness (and thereby productivity), they must be rigorous. Inspection should be a precisely defined activity that achieves a prescribed set of results. These results, once achieved, should be completely dependable thereby permitting other parts of the software lifecycle, such as testing, other forms of verification, and maintenance to be simplified, reduced, or streamlined.

Phased Inspection is an enhanced review method that is designed to deal with the limitations noted in the previous section and to provide the benefits just outlined. The goals of the method are (1) that it be rigorous so that the results are specific to a particular product and repeatable, (2) tailorable so that it can serve functions other than error detection, (3) heavily computer supported so that human resources are used only where necessary, and (4) efficient so that maximum use is made of available resources.

Of these goals, rigor is the most complex and difficult to achieve. Rigor must be supported in at least two areas, process and enforcement. From the process definition, it must be possible to know exactly what actions will take place during an inspection so that inspectors know exactly what is required of them and when. Compliance checking is necessary in order to show that the rigorous process definition has actually been followed in practice. Just as inspections are required to check the work of others, so the work of the inspector must be checked.

A Phased Inspection consists of a series of coordinated partial inspections termed *phases*. Each phase is intended to ensure that the product being inspected possesses either a single specific property or small set of closely related properties. The property checked during a given phase is chosen to be intellectually manageable so that comprehensive checking is a reasonable expectation. If this is not possible, the property is split so that multiple phases can be used. The properties examined are ordered so that each phase can assume the existence of properties checked in preceding phases. The inspectors performing a given phase are held responsible for assuring that the properties defined for that phases have been fully checked. Taken together, the set of phases constitute a single Phased Inspection.

Phased Inspections are tailorable so that they can be used to check for a wide range of desirable characteristics. They are not intended solely for finding errors. For example, they can be used to ensure that a particular work product has certain important characteristics such as portability, reusability, or maintainability. The present level of understanding of what is required to make software truly portable, for example, requires that the software comply with an extensive set of design rules. Inspection for compliance is a significant undertaking over and above what might be needed to inspect for errors and warrants a separate inspection in its own right. Clearly, multiple Phased Inspections can be undertaken to establish several of these desirable characteristics.

The concept of Phased Inspection has benefited from the work on Active Design Reviews [11]. Active Design Reviews focus on error detection in designs whereas Phased Inspections are intended to be used on any work product including requirements, designs, and source code. In addition, the phases of a Phased Inspection are orthogonal to the reviews of which an Active Review would be composed. A phase examines an entire product for compliance with a specific property whereas a review in an active design review examines part of the product. There are other differences between the two techniques especially in the area of computer support.

Phases are designed to be as rigorous as possible so that compliance with associated properties is ensured, at least informally, with a high degree of confidence. To achieve this, there are two types of phase, *single-inspector* and *multiple-inspector*, with different formats.

4.1. Single-Inspector Phases

A single-inspector phase is a rigidly formatted process driven by a list of unambiguous checks. For each check, the product either complies or it does not. The work product cannot complete this type of phase until it complies with all of the checks in the list. As the name implies, the intent is that the checks will be performed by a single inspector working alone.

Single-inspector phases are used to establish a wide variety of relatively simple yet important properties ranging from compliance with simple documentation standards to compliance with important programming practices. Clearly, many simple qualities of this type can be established with a static analyzer. Our goal is to provide an inspection technology for those situations where static analysis is beyond the state of the art or a suitable analyzer does not exist.

4.2. Multiple-Inspector Phases

A multiple-inspector phase is designed to check for those properties of the product that cannot be captured by a set of application-independent, precise questions with yes/no answers. Typically, such properties include completeness or correctness issues for requirements or functional correctness concerns for implementations. In a multiple-inspector phase, several inspectors examine the product *independently* and in a highly structured way.

The inspectors are provided at the outset of the phase with the necessary reference documentation for the product and begin with an examination of this documentation. The inevitable questions of clarification that they generate serve to improve that documentation. Note that in contrast to the overview of a traditional inspection, the inspectors are not provided with information that is not generally available.

Using the reference documents as necessary, the inspectors proceed with independent inspections of the work product with the goal of establishing that the work product has the property defined for the phase. These individual inspections are driven by checklists that are in part domain-specific and in part application-specific. The goal of the checklists is to ensure that the inspectors focus on the work product in a systematic way and with complete coverage. As presently defined, the checklists do not have yes/no answers for the most part, but take more of the form of asking the inspector to check all of a certain syntactic structure.

The domain-specific checks require the inspector to look for known areas of difficulty in the associated domain. A check used within many domains would be to ensure that the relational operators are used correctly. It is very difficult to test for errors in which a “<” has been used in place of a “<=” but such errors can be located by inspection. Such a check also requires detailed knowledge of the domain on the part of the inspector.

The application-specific check lists are designed to force a thorough examination of the work product by the inspector. Using the concept developed for Active Reviews, the checks take the form of a systematic set of questions about the work product itself developed by the author. The questions take the form “What is this statement for?” or “What is this data type used for?” Such questions are generated at a fixed rate per thousand lines of source code, for example, and are of a predefined form. The ability to answer such questions successfully ensures that the inspector checked the selected item and understood the work product sufficiently well to be able to answer such involved questions. Being unable to answer a question is an important outcome of a multiple-inspector phase since it indicates that the work product was not sufficiently documented or was not clearly written. This is just the kind of information that is essential to be able to ensure that a work product will be amenable to maintenance.

The separate inspections are followed by a *reconciliation* in which the individual inspectors compare their findings. Since the goals of the various checklist items in a multiple-inspector phase are to force coverage and consistency in the individual inspections, the inspectors' findings should be identical but in practice they will only be similar. The intent of the reconciliation is to avoid the personnel difficulties found to occur in typical group inspections.

4.3. Selection Of Inspectors

The staff used in the various phases can be chosen so that their qualifications meet the needs of the phase. This helps address the goal of making efficient use of human resources. A single-inspector phase that is checking compliance with internal documentation standards, for example, might be undertaken by a technical writer whereas a phase checking programming practices might be performed by a junior software engineer.

A major benefit of this flexibility is the possibility of using staff with particular skills as inspectors for phases in highly specialized areas. This would permit them to comment about the work of their colleagues in these specialized areas and thereby rapidly impart their skills on the product either by confirming the quality of the product or suggesting appropriate changes. This is a familiar and valuable concept that is neither systematized nor exploited in existing review methods.

4.4. Example Phased Inspection

As an example, consider the goal of checking source code for elementary desirable characteristics considered important in production software. A simple Phased Inspection could consist of six phases. Phase 1 would ensure compliance with required internal documentation checking format, placement, spelling, and grammar at the same time. Phase 2 would examine the source code layout for compliance with required format. Phase 3 would check the source code for readability in areas such as meaningful identifiers, use of abbreviations, and compliance with local naming standards. Production software has been known to use meaningless single-character identifiers thereby making the maintenance task much harder. Checking for compliance with good programming practices would be done in phase 4. Checks in this phase might include freedom from unnecessary goto statements and appropriate use of global variables. The checks performed in phase 5 would assure the correct use of various programming constructs such as updating the variables controlling while statements and explicitly closing files that are successfully opened. Finally, phase 6 would be a multiple-inspector phase aimed at checking functional correctness.

Clearly, phases 1 and 2 might be obviated by a formatting tool that enforces local standards. Similarly, phases 4 and 5 might be supplemented or obviated by static analyzers. Where these phases were performed by human inspection, phases 1 and 2 could be performed by a technical writer, phase 3 by a junior engineer, phases 4 and 5 by a software engineer, and phase 6 by senior software engineers.

5. COMPUTER SUPPORT

Phased Inspections are well suited to computer support, and a prototype toolset (**InspeQ**†) has been developed. The overriding goal of the toolset is to provide the highest level of support possible for human inspectors. Naturally, this takes the form in many cases of fairly straightforward bookeeping aids. However, elimination of these functions from human concerns changes the character of inspections dramatically and improves the overall performance because details do not ''drop through the cracks.'' The secondary goal of computer support, that of compliance checking, is almost transparent to the conscientious inspector yet provides a great deal of support for project management.

† Inspecting software in phases to ensure Quality.

5.1. Inspection Support

Features provided by the toolset to support inspection are in the general categories of work product navigation and display, documentation display, and comment recording. Specific tools are:

- *Work Product Display*
  The work product display is a general tool for looking at the work product and is the primary facility that the inspector uses during an inspection. The tool permits display, scrolling, repositioning, and searching the text. Multiple instances of the display window can be used to permit inspection of related but separate areas of the work product.

- *Checklist Display*
  The checklist display shows the checklist associated with the current inspection phase. It ensures that the inspector is informed of exactly what checks are involved in a given phase. The display also accepts input from the inspector indicating the status of the various required checks thereby facilitating compliance. The inspector can indicate for each checklist item either that the product *complied*, *did not comply*, was *not checked*, or that the check was *not applicable*.

- *Standards Display*
  The standards display shows the standards that the checklists are designed to check including compliant examples for illustration.

- *Highlight Display*
  The highlight display allows the inspector to identify certain syntactic categories of interest, such as if statements or arithmetic expressions, by menu selection. Instances of the selected syntactic category are extracted and displayed one at a time in a separate window. The intent of this display is to help the inspector quickly find and isolate specific syntactic features that relate to inspection checklist items.
  Isolating features in a separate window allows the inspector to concentrate on narrow sections of the product if desired, avoiding distraction by the feature's surroundings. If the inspector is checking a switch statement in a "C" program, for example, he does not need to check how the control variable for the switch statement is used before or after the switch statement.
  The highlight facility is useful in a number of ways. For example, a checklist item might require the inspector to check that all while statements terminate. The highlight facility allows the inspector to highlight all the while statements in the product, and sequentially check each one until he has checked them all. Without this facility, the inspector would have to locate the statements of interest either manually or using some general-purpose editor, and would have to monitor compliance manually.
  The highlight facility does not support all desired syntactic elements of all possible work products. It requires syntactic information about the product produced by a syntax analyzer. A general syntax analyzer is provided for "C" source code permitting highlighting of statements, functions, expressions, and operators. A limited syntax analyzer for "Ada" has also been developed.

- *Comments Display*
  The comments display provides an editable text display for an inspector to record anything in the work product with which he is not satisfied. The commands controlling this display are roughly equivalent to Emacs text editor commands.
  In order to provide context for the inspector's typed comments, sections of text or just the associated line numbers from any text display can be pasted into the comments. Pasting text from the work product being examined can be useful when it is hard to explain a

problem but easy to show by example. The inspector can paste a copy of the non-compliant text and then edit his comments. Another useful technique is to paste two copies of the non-compliant text and edit one to show how a correction can be made. This is sometimes an easy way of explaining a complex idea to the author. **InspeQ** formats the inspector's comments in a file for submission to the author.

5.2. Enforcement

Enforcement facilities provided by the toolset are in the general categories of explicit process support and compliance checking. Support for the process includes (1) tracking the assignment of personnel to phases, (2) tracking associated files, (3) permitting files to progress through phases only as each phase is completed, and (4) ensuring the correct order of phases.

Compliance checking is limited to maintaining the checklist and phase status of any particular inspection. During inspection, an inspector is believed if he marks a checklist item. Progression between phases is disabled for incomplete checklists. A planned future improvement of the toolset's support for enforcement will associate specific types of product features with checklist items. Thus, an inspector will not be able to mark a checklist item until he has examined every feature of the types associated with the checklist item. For example, if a checklist item requires an inspector to check that every while statement in a program terminates, **InspeQ** will ensure that every while statement was at least examined in isolation in the highlight display.

6. PRELIMINARY EVALUATION

Phased Inspections were developed to create a rigorous and reliable review method for software work products. We expect Phased Inspections to reduce the cost and effort of some other stages of development also. For example, both system testing effort and maintenance effort might be reduced by Phased Inspections of requirements, designs, and code. It is not sufficient, however, to claim these benefits based purely on the insight (or perhaps fantasy) of the developers of the method; a systematic evaluation is required to determine whether Phased Inspections fulfill these expectations. Fundamentally, an evaluation has to answer the most important question: "Are Phased Inspections cost effective?" No matter how reliable or rigorous Phased Inspections are, if they are not cost effective, they will not be used.

Cost effectiveness in this case is almost impossible to model analytically in a convincing way. Its determination can only be achieved by experimentation using industrial work products as targets, operating in an industrial environment, running multiple replicated experiments to permit statistical variance to be estimated, and comparing with full-scale controls using existing methods. Such experimentation is impractical without an investment of substantial industrial resources over many years. No industrial organization is likely to support this level of experimentation unless there is good reason to believe that the outcome will be favorable, and it is not feasible in an academic environment.

This does not mean, however, that experiments with Phased Inspections should not be conducted. Quite the contrary, constrained experiments might not produce conclusive results, but they might provide good indications of the relative utility of Phased Inspections. Thus we have followed the traditional path of acquiring what experimental data we could using volunteer graduate students.

In this section, we outline an evaluation framework and report the results of two evaluation experiments. The experiments were designed to answer as many questions from the framework as possible. They supplied a number of surprises in addition.

6.1. Evaluation Framework

The purpose of developing an evaluation framework was to define the way in which the long-term process of experimentation might proceed so as to evaluate Phased Inspections thoroughly. The framework breaks the problem of evaluation down into five areas; feasibility, performance, resources used, consistency achieved, and utility of computer support. Examples of the concerns in these five areas are as follows:

Feasibility:

(1) Is Phased Inspection a workable process?

(2) Is significant computer support feasible?

Performance:

(1) Does the performance achieved depend on the particular type of work product? For example, are Phased Inspections more effective on source code than test plans or requirements specifications?

(2) Does the notation in which the work product is written affect performance? For example, are Phased Inspections of source programs more useful on programs written in "C" than those written in "Ada"? One might expect so given the difference in philosophy of the two languages.

(3) Does the performance achieved depend on the experience and specific skills of the inspectors?

Resources:

(1) Can inspectors with lessor skills be used in phases involving only simple checks, and does this produce the expected savings?

(2) How long do inspections take and what is the variance in inspection times? Does the time taken depend on inspectors skills and background?

Consistency:

(1) Do different groups of inspectors implementing the same instantiation of Phased Inspection on the same work products consistently achieve the same results?

(2) Does a Phased Inspection permit useful conclusions to be drawn about a specific work product after inspection as desired?

Computer Support:

(1) Does computer support reduce the resources needed to perform an inspection?

(2) Does computer support improve the rigor or quality of the inspection process or the work products being inspected?

6.2. Experiment One

Early in the development of Phased Inspections, a limited experimental evaluation was performed using the phases summarized in the example in section 4 for software source code

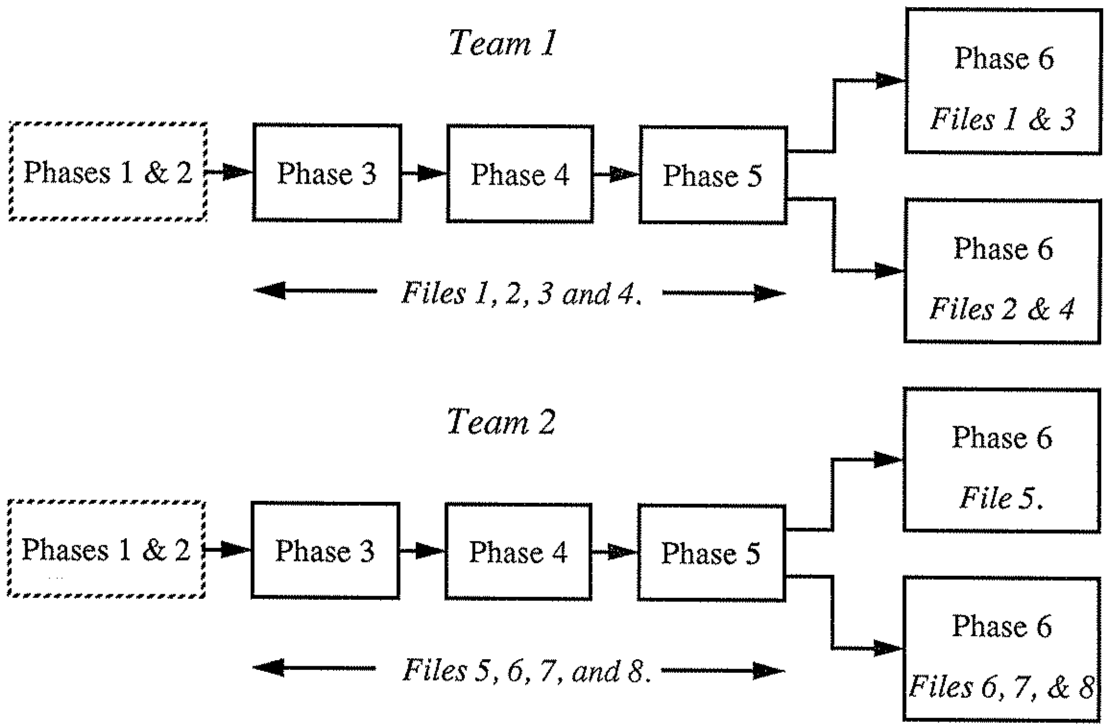

Figure 1 - Organization Of Phases In Experiment 2.

written in ''C'' with parts of the toolset as the target. The goal of this first experiment was to get early feasibility assessments. The single- and multiple-inspector phases were applied to separate files and were treated as separate partial experiments.

The inspectors involved in the first experiment had degrees of experience with ''C'', industrial software development, and software reviews that they individually described as varying from ''none'' to ''extensive''. In the single-inspector phases, the work product was 643 lines long including comments and the rate of inspection was about 470 lines per hour with little variance. The multiple-inspector phase in the first experiment was directed at a work product that was 1015 lines long including comments. Each inspector spent approximately one hour on documentation review, two hours actually inspecting the product, and an hour in the reconciliation meeting, again with little variance.

The primary results of the experiment were an indication of the overall feasibility of the process and a list of suggested improvements to the toolset. Other observations were that rate of inspection climbed as inspectors became familiar with the checklists and that knowledge of ''C'' was, as expected, the major factor affecting inspection rate. Major changes to the process and to the toolset were made as a result of the first experiment.

6.3. Experiment Two

After the changes suggested by the first experiment had been effected, a second more elaborate experiment was undertaken. Detailed feasibility and performance results were sought. The phases used were improved versions of those used in the first experiment and the target was, once again, the support toolset. Because of their simplicity, phases 1 and 2 were omitted in the second experiment. Phases 3, 4, and 5 were single-inspector phases checking source-code readability, local programming practices, and X-windows related qualities, and involving 11, 25, and 10 checklist items respectively. Examples of the checklist items were:

Phase 3 - Are all constants values identified by defined symbolic constants?

Phase 4 - Is there a default choice in all switch statements? If the default choice is not used for error detection, is there a comment explaining why?

Phase 5 - If a dialog widget XmNautoUnmanage resource is FALSE, is it unmapped before popping down its parent?

Phase 6 was a multiple-inspector phase checking functional correctness. For this experiment, there were two inspectors in phase 6. The phase 6 checklist contained 7 domain-specific checklist items and varying numbers of application-specific checklist items depending on the specific file. An example domain-specific checklist items is:

Phase 6 - Do expressions compute the expected value? Are < and <= used properly? Are > and >= used properly? Are parenthesizes used where precedence rules may make the expression hard to understand?

Eight files were inspected containing a total of approximately 4,345 lines with the shortest file containing 219 lines and the longest containing 1,320. Two inspection teams were used in phases 3, 4, and 5 so that two inspections could be undertaken in parallel. Thus each inspector in phases 3, 4, and 5 of each inspection team inspected a total of four files. Phase 6 was further duplicated so that each pair of inspectors involved in phase 6 only needed to examine two files. This also permitted four phase 6 inspections to be carried out concurrently. With these replications, a total of 14 inspectors performed the inspections. The inspection structure used in the second experiment is summarized in Figure 1.

In order to obtain some quantitative information, the work products supplied to the inspectors were deliberately seeded for each phase with deficiencies that should have been found by that phase. The seeding rate was approximately four deficiencies per 1,000 lines of commented source text. The inspectors were not aware that the seeded deficiencies were present.

For phases 3 and 4, the seeded deficiencies were synthetic and merely represented instances of the kind of situation that the inspectors should be able to locate. Phase 5 consisted of specific coding standards directed towards the correct use of the X-window system. Apparently simple mistakes are easily made in programs using X-windows and these mistakes are often very hard to locate. The checklist for phase 5 was developed after having to deal with many of these difficult debugging situations. The seeded deficiencies installed in the inspection target files prior to phase 5 were based on experience and so were very realistic. The deficiencies seeded prior to phase 6, the functional-correctness phase, were also based on experience and were similarly realistic.

Tables 1, 2, and 3 summarize the performance data obtained from phases 3, 4, and 5. In the columns headed "SEEDED FOUND", the first number is the number of seeded deficiencies that

	LENGTH (Lines)	SEEDED FOUND	INDIGENOUS FOUND	TIME
File 1	828	0/2	4	4h33m
File 2	517	1/2	2	
File 3	219	1/1	0	
File 4	475	2/2	2	
File 5	1320	1/1	3	4h14m
File 6	317	1/1	2	
File 7	401	2/2	1	
File 8	268	1/1	1	

Table 1 - Phase 3 Results

	LENGTH (Lines)	SEEDED FOUND	INDIGENOUS FOUND	TIME
File 1	828	1/2	0	2h51m
File 2	517	0/2	0	
File 3	219	0/1	3	
File 4	475	2/2	3	
File 5	1320	3/3	3	10h10m
File 6	317	1/1	3	
File 7	401	2/2	5	
File 8	268	1/1	4	

Table 2 - Phase 4 Results

were found and the second is the number that were present in the file. As the tables show, performance at locating the seeded deficiencies was high.

The columns headed ''INDIGENOUS FOUND'' reports the numbers of indigenous deficiencies that the inspectors found. These deficiencies were unknown to the authors prior to the inspection. Since the software that was inspected has been in use for an extended period and was carefully written, the location of these deficiencies was a pleasant surprise.

The times shown are the total times taken for the various phases as measured by a real-time clock. The times include all idle time accrued during inspection whether or not the idle time occurred because the inspector was actively looking at the work product. The times were obtained from the system clock and did not rely on any form of human recording.

Tables 4 and 5 summarize the results obtained in phase 6. In this case, indigenous deficiencies were detected in three major categories; those which affected correct functionality, those viewed as significant deficiencies in internal documentation, and those considered stylistic deficiencies of sufficient significance that they would affect long-term product maintenance.

The deficiencies documented in Table 4 are further broken down into those that were found during the inspection step and those found during the reconciliation step. The multi-inspector phase format was designed with the goal of all defects being detected during the inspection step. That this did not happen and that many deficiencies were detected during the reconciliation indicates that additional work needs to be done on the checklists used to drive phase 6. In practice, the reconciliation steps turned into highly focused discussions of the functional correctness of the associated work product.

It is important to keep in mind when reviewing these results that they were obtained with essentially untrained volunteers. In a post-experiment questionnaire, the inspectors were asked to rate their own performance without knowledge of the data that had been obtained. The inspectors' assessments of themselves correlated strongly with the number of seeded deficiencies that had been found. In a more traditional work environment in which inspectors were paid and had a degree of loyalty to their employer and the product, the performance might be better that the results we obtained.

	LENGTH (Lines)	SEEDED FOUND*	INDIGENOUS FOUND	TIME
File 1	828	2/2	2	2h48m
File 2	517	0/0	0	
File 3	219	0/0	0	
File 4	475	2/2	0	
File 5	1320	1/3	1	2h27m
File 6	317	0/0	0	
File 7	401	0/0	0	
File 8	268	0/0	0	

Table 3 - Phase 5 Results

* Phase 5 checked rules associated with X-windows programming. Since some files did not contain X-specific code, no faults were seeded in those files for this phase.

Some of the results obtained from the second experiment cannot be tabulated. These results are derived from various kinds of observations and comments by the inspectors. Specifically:

- Certain checks in early phases did not have the degree of completeness that was expected. For example, a required check in phase 3 is to examine every identifier to determine whether it is meaningful. Although work products passed this phase, inspectors in phase 6 found that identifiers thought to be meaningful during the simple phase 3 check were, in fact, not as meaningful as they could be once an understanding of the software was achieved. This effect occurred with comments also. Comments are checked for syntax, grammar, and superficial content in phase 1 but the serious content cannot be checked until phase 6. This problem suggests several revisions to the various checklists.

	LENGTH (Lines)	SEEDED FOUND	INDIGENOUS					
			Functionality		Documentation		Signif. Style	
			Insp.	Recon.	Insp.	Recon.	Insp.	Recon.
File 1	828	1/2	1	0	1	1	1	0
File 3	219	1/1	2	0	1	1	1	5
File 2	517	1/2	0	1	1	0	5	0
File 4	475	0/2	1	0	0	0	1	0
File 5	1320	1/2	0	1	1	0	0	1
File 6	317	1/1	2	0	2	0	1	2
File 7	401	1/1	0	0	1	0	3	0
File 8	268	0/1	2	2	3	0	2	0

Table 4 - Phase 6 Deficiency Detection Results

	Team 1A	Team 1B	Team 2A	Team 2B
Total Lines	1,047	992	1,320	986
Total Inspection Time	6h01m	4h55m	3h00m	7h31m
Reconciliation Time	1h45m	2h00m	3h40m	1h30m

Table 5 - Phase 6 Timing Results

- A small part of the source code was compliant with most of the standards demanded by the Phased Inspection process but was considered to be generally poorly written. We were pleased to discover that this situation was immediately obvious to the inspectors in phase 6 who unanimously rejected the associated files and suggested that they be totally rewritten.

- Some inspectors chose to expend far more effort than was required by the process with both good and bad results. A positive example was an inspector who chose to rewrite a complete function to show how it could be improved. A negative example was an inspector in an early phase who essentially undertook several phases at once thereby supplying a lengthy and very confusing report. The conclusion in that case was that the inspector was essentially over qualified for the relatively simple checks in the phase.

In terms of the questions raised by the evaluation framework, the results are as follows. In the area of feasibility, the conduct of these two experiments has revealed that the process is feasible and that computer support is achievable. More significantly, in post-experiment questionnaires, the inspectors were uniformly enthusiastic about the merit of the process and the toolset. In the area of performance, we have no data on the effect of the type of work product nor on the notation used since all of the targets used in the experiments were source text written in "C". However, we do have strong evidence that substantial checking of relatively large volumes of source code can be achieved in times that we consider reasonable for the benefit gained. In the area of resources, we used inspectors with essentially equivalent backgrounds because that was the pool available to us. However, as noted above, we do have preliminary data on the time required to perform inspections and the associated variance. By matching times with pre-experiment questionnaire data on background and experience, we have confirmed that inspection rate is heavily influenced by language experience. We also confirmed and the inspectors reported again that inspection rates improved with increasing familiarity with the checklists.

The area of consistency is perhaps the most important in the framework. The results in this area are mixed and suggest that the process as presently defined is not achieving the degree of consistency that we desire. This is indicated by the fact that seeded deficiencies were not always caught and by the fact that deficiencies were located during the phase 6 reconciliation steps. However, the fact that many important indigenous faults were discovered in software thought to be fully compliant is a strong indication that the process is achieving considerable thoroughness.

In the area of computer support, we have no data on whether it reduces the resources required or improves inspection quality since we have no statistical controls. Evidence from the post-experiment questionnaires indicates that the inspectors found the toolset, for the most part, either useful or very useful.

7. CONCLUSION

We believe that inspection is one of the most valuable tools that the software engineer has available but that the technology is not being exploited to its full potential. We have defined an enhanced inspection technique called Phased Inspection that addresses the deficiencies of existing inspection techniques. The most important goal of Phased Inspection is rigor so that engineers can trust the results of a specific inspection and so that inspection results are repeatable. We have also presented details of a toolset that supports Phased Inspection by providing the inspector with as much computer assistance as possible and by checking for compliance with the required process of Phased Inspection.

Experimental evaluations of Phased Inspections lead us to conclude that the goals are being partially achieved and that further refinement of the checklists used and process structure will permit further improvements in inspection efficiency.

8. ACKNOWLEDGEMENTS

It is a pleasure to acknowledge the many graduate students in Computer Science at the University of Virginia who volunteered for the evaluation experiments and spent many hours of their own time learning about Phased Inspections, the toolset, and performing the inspections while being monitored. We also thank Keith Miller and Gina Bull. This work was funded in part by NASA under grant numbers NAG-1-1073 and NAG-1-1123, in part by SAIC Inc., in part by the MITRE Corporation, and in part by the Virginia Center for Innovative Technology grant number CAE-92-003.

REFERENCES

[1] Dyer, M., ''A Formal Approach to Software Error Removal'', *The Journal of Systems and Software*, Vol. 7, 1987.

[2] Fagan, M.E., ''Design and Code Inspections to Reduce Errors in Program Development'', *IBM Systems Journal*, Vol. 15, No. 3, 1976.

[3] Fagan, M.E., ''Advances in Software Inspections'', *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 7, July, 1986.

[4] Fagan, M.E. and J.C. Knight, ''Testing is Not the Best Means of Defect Detection and Removal'', Achieving Quality Software - A National Debate, Society for Software Quality, San Diego, CA, January, 1991.

[5] Freedman, D.P. and G.M. Weinberg, *Handbook of Walkthroughs, Inspections, and Technical Reviews*, Little, Brown and Company, Boston, Toronto, 1982.

[6] Freedman, D.P. and G.M. Weinberg, ''Reviews, Walkthroughs, and Inspections'', *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 1, January, 1984.

[7] Kernighan, B.W. and P.J. Plauger, *The Elements of Programming Style*, Second Edition, McGraw Hill, New York, 1978.

[8] Kernighan, B.W. and D. M. Ritchie, *The C Programming Language*, Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[9] Linger, R.C., H.D. Mills, and B.I. Witt, *Structured Programming : Theory and Practice*, Addison-Wesley, Reading, MA., 1979.

[10] Myers, E.A., *Phased Inspections and Their Implementation*, M.S. Thesis of Univ. of VA, May 1991.

[11] Parnas, D.W., D.M. Weiss, ''Active Design Reviews: Principles and Practices'', Proceedings of *Eighth International Conference on Software Engineering*, London, England, August, 1985.

[12] Petroski, H., *To Engineer Is Human: The Role of Failure in Successful Design*, St. Martin's Press, New York, 1985.

[13] Russell, G.W., ''Experience with Inspection in Ultralarge-Scale Developments'', *IEEE Software*, Vol. 8, No. 1, January 1991.

[14] Selby, R.W., V.R. Basili, and F.T. Baker, ''Cleanroom Software Development: An Empirical Evaluation'', *IEEE Transactions on Software Engineering*, Vol. SE-13, No. 9, September, 1987.

[15] Software Productivity Consortium, *Ada Quality and Style: Guidelines For Professional Programmers*, Van Nostrand Reinhold, New York, 1989.

[16] Weinberg, G.M., *The Psychology of Computer Programming*, Van Nostrand Reinhold, New York, 1971.