Phased Inspections and Their Implementation

John C. Knight
Ethella Ann Myers

Computer Science Report No. TR-91-10
May 13, 1991

ABSTRACT

Since the 1970s, non-mechanical review methods have become very popular as verification tools for software products. Examples of existing review methods are formal reviews, walkthroughs, and inspections. Another example is Fagan Inspections, developed in 1976 by Michael Fagan in an effort to improve software quality and increase programmer productivity. Fagan Inspections and other existing methods have been empirically shown to benefit the software development process, mainly by lowering the number of defects in software early in the development process. Despite this success, existing methods are limited. They are not rigorous, therefore, they are not dependable. A product that has been reviewed with an existing method has no quantitative qualities that are ensured by the method used.

This thesis presents a new review method, Phased Inspection, that was developed to be rigorous, reliable, tailorable, heavily computer supported, and cost effective. Phased Inspection consists of a series of partial inspections termed phases. Each phase is intended to ensure a single or small set of related properties. 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.

A detailed description of Phased Inspection, an evaluation framework and preliminary evaluation, and a prototype toolset for support of Phased Inspection is presented.

TABLE OF CONTENTS

TABLE OF CONTENTS

LIST OF FIGURES

inspection \in-'spek-shən\ *n* (14c) **1** : the act of inspecting **2** : a checking or testing of an individual against established standards

From *Webster's Ninth New Collegiate Dictionary*

CHAPTER 1. INTRODUCTION

1.1 Software Inspections

Software inspections 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 Neuman both regularly asked colleagues to examine their programs [8]. In the 1950s and 1960s, large software projects usually included some sort of software reviews, and in the 1970s these review mechanisms and forms began appearing in publications. By the mid to late 1970s, various inspection methods had emerged with different names: software reviews, technical reviews, formal reviews, walkthroughs, structured walkthroughs, and code inspections. Each inspection method had different forms to fill out, different inspection team sizes and makeup, etc., but none of these much touted inspection methods provided any procedures for inspecting the code other than just looking at it. We don't write programs by flipping switches and punching cards anymore, we shouldn't be checking software correctness by just *looking* at it.

Why do software inspections at all? Why not depend on testing to find errors? Because testing is expensive, and is not sufficient. Linger, Mills, and Witt [12] state that:

> It is well known that a software system cannot be made reliable by testing. (p. 13)

A program cannot be tested completely, but all source and design code can be inspected [6]. The goal of traditional software inspection is to find as many faults in a software product as possible as early as possible. The earlier a fault is found, the lower the cost of repairing the fault. Besides, errors that cause software to execute incorrectly is a subset of the possible faults a software product might have.

In order to make inspections dependable, it must be possible to assert that the product has certain properties either with certainty or with high probability after an inspection of the product. This means that the inspection process must be *rigorous*. Rigor allows conclusions to be drawn about a property of a product, and allows these same conclusions to be drawn about every product that is inspected.

People make mistakes when they write software. If they did not, we would not need to inspect or test the software in the first place. From *To Engineer Is Human* by Henry Petroski [17]:

> 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. (p. 52)

People also make mistakes when they inspect software, and software review methods should account for the possibility of inspectors making mistakes.

1.2 Goal of Research

The goal of this research is to develop an inspection process that is rigorous and repeatable. The inspection process is broken up into manageable sections. Each section or *phase* is then broken into a series of specific checks designed to support the verification of a single or small set of related properties of the product. By providing unambiguous checks and the order of execution of those checks, the resulting inspection process is certainly made repeatable. The checks determine the rigor of the inspection. The more explicit and exact the checks, the more rigorous the inspection.

1.3 Summary of Chapters

Chapter 2 describes existing software inspection methods. Chapter 3 discusses the limitations of the existing software inspection methods. A software inspection method that rectifies the shortcomings of existing inspection methods is introduced in Chapter 4. Computer support is essential for the management and enforcement of phased inspections. A toolset for supporting phased software inspections was prototyped to address the question of computer support for inspections. The goals and features of this prototype is presented in Chapter 5. An evaluation of the phased inspection concept and the supporting toolset is discussed in Chapter 6. Conclusions and some suggestions for further work are provided in Chapter 7.

CHAPTER 2. EXISTING REVIEW METHODS

Informal reviews have been part of the development of software from the beginning. Expensive computing time and the scarcity of computer hardware demanded it. Reviews were mainly used to uncover syntax errors since compiling a program to find syntax errors was out of the question [24]. As time went on and the turnaround time for compiling programs got shorter, reviews focused less on syntax errors and more on the logic and functionality correctness of software. While compilers can be very good at syntactic analysis of programs, they cannot check for design flaws, style errors, or correctness.

In the 1950s and 1960s, most large software projects included a software review in the development process, but it was not until 1971 when *The Psychology of Computer Programming* by Gerald Weinberg [23] was published that a book advocated the review of programs in all stages of the development process [7]. Since that time, review methods have frequently appeared in computing literature. These review methods can be placed into one of three 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 presenter's presentation and issues raised by the reviewers.

(2) Walkthroughs
Walkthroughs are usually used to review source code as opposed to design and requirements documents. The reviewers 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 reviewers might have.

(3) Inspections
In inspections, a list of criteria the software must satisfy determines the flow of the review. While walkthroughs and formal reviews are generally biased towards error detection and ensuring correctness, inspections are often used to establish other properties such as portability and adherence to standards [7]. 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 Michael Fagan at IBM Kingston, NY, in 1976 [1][4]. This method, called *Fagan Inspections*, is a combination of a formal review, an inspection, and a walkthrough. This combination of review methods has made Fagan Inspections seem more formal and therefore more effective than previous methods.

The Cleanroom software development method [20] is described in this section because of its dependence on review methods. In a cleanroom, the software developer is not allowed to compile his code at all, but must depend on review and formal verification methods to detect errors in the software.

2.1 Formal Reviews

A formal review usually takes the form of presentation of the product [7][24]. The presentation may be in the form of a lecture to a large group of people intended to quickly familiarize the reviewers with the product. At the other end of the spectrum, the review may be more of a discussion about the product by the reviewers with the presenter providing information as needed. The presenter can be the author or a reviewer familiar with the software. A formal review has an advantage in that it can be used to present the software to a large number of people. A formal review involving people of different backgrounds takes advantage of different perspectives. But too often, one reviewer dominates the formal review, losing the intended effect of having several different people examine the code. To counter this problem, several tactics have been employed to ensure the participation of all the reviewers.

(1) Round-Robin Reviews
Round-Robin reviews [7] force all of the reviewers to contribute something to the formal review. The reviewers take turns raising issues about the code or may be assigned units of the product to review and present to the rest of the reviewers. Either way, the each reviewer is forced to actively participate in the reviewing process.

(2) Devil's Advocate
When the devil's advocate [15] method is used in a formal review, one reviewer is assigned the task of finding as many things wrong with the software as possible. The other reviewers' job is to defend the software as much as possible, only conceding a fault when all else fails.

(3) Error Insertion
Error insertion (sometimes called error seeding) is an incentive tactic [7][15]. Errors are inserted into the software before it is reviewed. After the software is reviewed, the list prepared by the reviewers is compared with the list of known bugs. If not all of the known bugs were found, the review was not thorough enough. The problem with this tactic is that naturally occurring errors that are not trivial are hard to fake.

The results of a formal review depend upon the experience and skill of the reviewers, the quality of the presentation, and even the amount of camaraderie among the reviewers.

2.2 Walkthroughs

A walkthrough is a step by step review of the software where the steps are determined by the organization of the reviewed material [7][8][24]. Walkthroughs are usually used to review source code as opposed to design and requirements documents. The reviewers do a step by step, line by line simulation of the code. A walkthrough is usually performed after the first *clean* compilation of the code. In other words, the code is free of syntax errors.

Walkthrough methods have certain roles that reviewers must play. The roles of presenter, coordinator, and recorder are required roles. The roles of maintenance oracle, user representative, and standards bearer are optional, but can make useful contributions to the review. The presenter makes a quick introduction to the code, and performs the reading out loud of the source code lines. The coordinator controls the flow of the review, making sure the reviewers do not waste time on trivial details or irrelevant issues. The recorder takes notes of the review, recording errors for the author to review before reworking the code. The optional roles - maintenance oracle, user representative, and standards bearer perform as the names imply. The maintenance oracle reviews the code with an eye to future maintenance. The user representative reviews the user interface. And the standards bearer makes sure the code adheres to any kind of standards the organization requires.

Structured walkthroughs [24] sound as if they are more formal than other walkthrough methods. But Weinberg and Freedman [7] assert that the only difference between a structured walkthrough and any other walkthrough is: "The addition of the word "structured" to the name."

2.3 Inspection

An inspection is a walkthrough driven by a list of criteria the product must meet. The reviewers run through a list of faults over the entire product. Inspections are usually used to cover a large amount of material in a short amount of time by allowing the reviewers to concentrate on one aspect of the product [7][8].

2.4 Fagan's Inspection Method

One of the most popular review methods was developed by Michael Fagan at IBM Kingston, NY, in 1976 [4][5]. Fagan wanted to create a new review process that would improve software quality and increase programmer productivity. This method is informally known as *Fagan Inspections*.

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 talks the inspectors through the code, paraphrasing what the author said in the overview, but in more detail. Every line of code is reviewed. A report is prepared and given to the author so he can rework the software, fixing any errors found during inspection. The follow-up step checks that the errors in the inspection report are fixed correctly.

2.5 CleanRoom Development Method

The Cleanroom software development was developed at IBM Federal Systems Division (FSD) [20] in an attempt to apply methods of other engineering disciplines to software development. The term "Cleanroom" refers to the dust free environments maintained for assembly of delicate hardware components. Applied to software, the Cleanroom method tries to create an environment that prevents errors from entering the product during development.

The main idea behind the Cleanroom method is nonexecution-based program development. Developers cannot depend on compilation or execution for verification of the software, but must use code inspections, walkthroughs, and formal verification methods to detect errors.

CHAPTER 3. LIMITATIONS OF EXISTING REVIEW METHODS

Empirical evidence has emerged showing that review methods based on human examination of a paper version of a product can have considerable benefit to a software development process [3], usually in the form of lowering the number of faults in software. Freedman and Weinberg [7] report that in large systems, reviews have reduced the number of errors reaching the testing stages by a factor of 10. This reduction cut testing costs by 50 - 80% - including review costs. Fagan, referring to results compiled by Russell [19], states that "65 - 90% of operational defects are detected by inspection at 1/4 to 2/3 the cost of test defects and removed at 1/7 - 1/2 the cost" [6].

Despite this success, major limitations remain:

(1) Existing review methods are not rigorous.

Although existing review methods are cost effective statistically and generally beneficial to software development, they do not ensure that a *given* reviewed product has any *specific* or *quantitative* quality.

(2) Human resources involved are not used effectively.

For example, in the overview step of Fagan inspections, design and implementation information about the product is conveyed to the reviewers verbally. Such information should be documented so as to not penalize anyone not present at the overview. This approach also assumes that the author will present everything relevant and the reviewers will not forget anything presented. As a second example, anecdotal evidence suggests that reviewers often use review time ineffectively by discussing trivial difficulties with the product such as spelling errors in comments and deviations from standards. In one industrial study of inspections of source code [2], a large percentage of the defects detected were commentary defects. In one stage, 16591 lines of source code were inspected, and there was an average of 148 defects found per 1000 lines of code. Of that 148, 61 were commentary defects.

(3) Inappropriate Focus.

Existing methods usually focus on defect detection. Since maintenance costs make up approximately 50 - 75% of the overall budget in most organizations [24], it is reasonable to look for coding and design practices that reduce software maintainability.

It should also be possible to review products for qualities specific to their domain. For example, an operating system would be reviewed with attention paid to concurrency issues, but a compiler would not. Software developed to be reusable would have stricter parameterization and documentation standards than software used only once.

(4) Existing review methods depend on human effort with essentially no computer support.

It is possible to supplement the review process considerably with computer resources. This permits far more efficient use of human time and more complete coverage of items that have to be reviewed. Not only is there no existing computer support for reviews, the existing methods are such that it would be difficult to provide computer support.

The rest of this chapter further explores these limitations that formal reviews, walkthroughs, and inspections have in common. Fagan Inspections deserve special attention; a separate section is dedicated to this method's limitations. The Cleanroom Development method is similarly discussed in a separate section.

3.1 Lack of Rigor

A rigorous or precise review method allows conclusions to be drawn with confidence about a property that holds for a product that has been reviewed. A lack of rigor makes a review method undependable, and if a review method is undependable, then:

(1) The completeness of a particular inspection is questionable. The manager responsible for the reviewed product cannot be sure whether a property holds for the product or not.

(2) The effectiveness of a particular inspection cannot be determined. Even if the reviewers claim that a property holds after an inspection, that manager cannot be sure that this is the case.

Lack of rigor in Formal Reviews

Weinberg and Freedman [8] describe the order of formal reviews as "determined by the flow of the meeting as it unfolds." This is *not* a description of a rigorous process. The reviewers in a formal review are essentially told: "Go forth and find errors. And you had better find some, because tactics such as Round-Robin and Error Insertion will be used to see just how good you are at finding errors."

A formal review involving people of different backgrounds takes advantage of different perspectives. But, as was mentioned in the previous chapter, often one inspector performs a more thorough job of reviewing the product and dominates the review. This problem has not gone unnoticed and tactics have been suggested to counter this problem. Examples such as Round-Robin, Devil's Advocate, and Error Insertion are described in the previous chapter. The very existence of these tactics demonstrates the unpredictability of formal reviews. These tactics also tempt reviewers to raise trivial issues. What is the reviewer supposed to do when it is his turn in a Round-Robin formal review and he does not have a relevant issue to raise? The results of formal reviews depend upon the experience and skill of the reviewers, the quality of the presentation, and even the amount of camaraderie among the reviewers. Formal reviews have too many outside influences to be an dependable verification tool.

Lack of rigor in Walkthroughs

In a walkthrough, the organization of the product being reviewed guides the flow of the review. This is a step up from formal reviews as far as rigor is concerned. At least the order is determined by something concrete. But this method makes some *faulty* assumptions.

(1) Reviewers always remember everything that can go wrong.
People just cannot be depended upon to remember everything that can be wrong with a product.

(2) Everyone is paying attention.
This is a problem when reviewers other than the presenter are too familiar with the code. Reviewers who are bored may not pay enough attention without even realizing what they are *not* doing [7].

(3) Every reviewer understands the material being reviewed.
Reviewers will grasp the material being presented at different rates. A reviewer may not understand the material completely and either does not know it because of the skill of the presenter or is too shy to speak up [7].

In the best case, all the reviewers in a walkthrough understand the code and are paying attention. In the worst case, a walkthrough is performed by a group of reviewers - half know the code very well and are bored so spend the time looking out the window. The rest are spending so much time trying to figure out what is going on they do not have any time to look for errors.

According to Weinberg and Freedman [8], well done walkthroughs produce products that are complete, correct, dependable as a base for related work, and measurable for purposes of tracking progress. Unfortunately, they do not present a way to determine whether a walkthrough is well-done or not.

Lack of rigor in Inspections

As far as rigor is concerned, inspections hold the most promise. The flow of an inspection is determined by a list of criteria the product must meet. The key to the rigor of an inspection is the list of criteria. The more precise and explicit the list, the more rigorous the inspection. But, as can be seen by the following examples from checklists of existing review methods, the list items are often too general or ambiguous to draw any conclusions about the results of an inspection.

These are checklist items excerpted from an *IBM COBOL Program Checklist* presented by Freedman and Weinberg [7] on pages 339-344:

> Has all the design been implemented?
> Does code do what the design called for (i.e., is the design translated correctly)?
> Is the design correct and complete?
> Is logic coded optimally (i.e., in the fewest and most efficient statements)?

Some of the problems with these checklist items are:

(1) The inspector is being to asked to check whether the design is correct and complete at the same time he is checking whether all the design has been implemented *and* implemented correctly.

(2) They are too general. How can the inspector be sure that all the design has been implemented and implemented correctly?

(3) Whether logic is coded optimally is a judgement call on the part of the inspector and in no way provides any conclusions that can be drawn about the reviewed product.

Not only are these checklist items difficult to verify, they are included in the same checklist with the following two items.

> Are THEN/ELSE groups aligned?
> Are nested IFs indented properly?

A reviewer who is checking logic and design information should not also be checking the source code format.

3.2 Inefficient Use of Resources

Existing review methods make no distinctions among the types of errors the inspectors should look for. And existing methods assume all personnel are experienced software engineers. This means that expensive, skilled personnel are checking for comment correctness, uninitialized variables, and standards deviations as well as the much more difficult to locate logic errors. Not only are costly personnel inspecting for trivial errors, but anecdotal evidence indicates that they often waste time debating trivial issues related to these types of errors.

Walkthroughs, formal reviews, and Fagan Inspections all include some form of verbal presentation that does not have to be preserved in documentation. This is a costly and inappropriate activity. Maintenance issues alone demand that anything verbally presented should be documented.

Inefficient use of resources in Formal Reviews

A formal review has an advantage in that it can be used to present the software to a large number of people. But, the formal review is overly dependent on particular personnel. Unless the author documents everything discussed at the formal review, information is lost to anyone not attending the review. This is especially important to future maintenance costs. It is estimated that 50-75% of the budget costs are maintenance activities, not development [24]. Information not documented is quickly lost or forgotten, especially if the author of the product leaves the organization.

Inefficient use of resources in Walkthroughs

A major difficulty with walkthroughs is a dependence on the role of the presenter. The presenter is the reviewer who presents the product. If the product is source code, the presenter is the one who reads the source code out loud line by line and answers any questions the other reviewers might have about the code. If the presenter is poorly prepared (does not know the code well enough), he will not be able to clear up difficult issues and the review will be unable to determine the quality of the code. On the other hand, an accomplished presenter can mask the lack of documentation or clarity of the source code. In other words, a skilled presenter may be able to answer questions about code that should be available from the code. Either way, walkthroughs are too dependent on the presenter.

Inefficient use of resources in Inspections

Inspections do not make any distinction between the types of errors to check for and the skill and experience of the reviewer. It is common to see a checklist item describing correct indentation in the same list as a checklist item detailing the correct use of recursive procedures [7]. Expensive, skilled personnel are checking for trivial errors as well as the much more difficult to locate logic errors.

Walkthroughs and formal reviews share this limitation with inspections. In addition, as noted previously, in walkthrough and formal reviews the reviewers interact, and anecdotal evidence suggests that not only do these skilled, experienced, *costly* reviewers check for trivial errors, they also spend time debating trivial issues.

3.3 Inappropriate Focus

While all kinds of faults can be found by existing review methods, they tend to focus on defect detection. Defect detection is important, but correctness is not the only desirable property of software products. Maintainability, portability, reusability, and concurrency issues are just a few of the areas that a review method might be concerned with other than defect detection. For example, a software product might have no errors, but its value is drastically reduced if it is not maintainable.

Inappropriate focus in Formal Reviews

The following appears in *Handbook of Walkthroughs, Inspections, and Technical Reviews* by Freedman and Weinberg [7]:

> **...Why All This Emphasis on Error?**
>
> Primarily because "error" is the least controversial thing to measure. Not that it's always simple to tell when there is or is not an error, but at least it's simpler than telling whether or not a particular piece of code is "modifiable" or "portable." To be sure, these attributes *should* be measurable, too, but at the present state of the art, few installations are prepared to measure them, let alone provide a past history of measurement.....
>
> Besides, if the software doesn't work properly, what's the use of measuring other aspects of its performance? (p. 334-335)

The following are arguments countering this statement.

(1) Just because it is more difficult to tell if a particular piece of code is "modifiable" or "portable", that does not mean these properties should not be included in a review.

(2) The past history of measurement should have no bearing on what is included in a review method. If there is no past history, one is initiated.

(3) If other aspects of the software's performance are not measured during the review, when are they going to be measured? Maintainability and portability cannot be checked in the testing phase of the development process.

"Error" or defect detection should be included as part of any review method, but not at the expense of other desirable attributes of the product.

Inappropriate focus in Walkthroughs

Like formal reviews, walkthroughs often focus on defect detection, but they are also often used to validate the overall approach to the problem. The reviewers are asked to judge the solution at the same they are looking for errors in it. This combination of functionality can be a waste of time and money. What if the reviewers decide the solution is all wrong? What is the point of looking for specific errors in something that is going to change? How can any concrete conclusions be drawn from a walkthrough about a reviewed product when that product is changing? Validation is not verification [11], and the two should not be confused nor combined. Walkthroughs can be a valuable tool during development for educating and getting feedback from a large and diverse number of people, but they are *not* a dependable verification tool.

Inappropriate focus in Inspections

Instead of focusing on defect detection, inspections tend to focus on some other property, neglecting defect detection. This is just as inappropriate as focusing on defect detection. As Weinberg and Freedman [7] point out, what is the point of a maintainable, portable, efficient piece of software that does not work?

A review method should not focus on one or a small set of properties. One of the arguments for reviews over testing was that testing cannot achieve complete coverage but reviews can [6]. So why limit reviews to the types of faults that testing can check? On the other hand, reviews should not go to the other extreme and ignore defect detection. Another advantage of reviews over testing is the timing of reviews in the development process; errors found earlier are easier and less expensive to remove [18]. Reviews should make the most of their unique advantages and attempt to verify as many desirable properties as is feasible.

3.4 Insufficient Computer Support

Existing review methods tend to assume that a process involving the human examination of product must be done from a hardcopy or paper version of a product. Not only do the existing methods assume that paper is necessary, but their mode of operations makes it difficult to provide computer support. Computer support usually implies a one-to-one interface with a single person. Existing methods usually require a one-to-many interface to the product being reviewed.

Insufficient computer support in Formal Reviews

A formal review is a presentation to a group of reviewers with varying degrees of discussion. This form of review has little chance of computer support that would make more efficient use of human time.

Insufficient computer support in Walkthroughs

Like formal reviews, walkthroughs involve a group of humans interacting. This limits the kinds of computer support that can be developed for the method. Display mechanisms might be helpful, but compliance checking and interactive recording of issues would be difficult to extend over the whole group of reviewers.

Insufficient computer support in Inspections

Since inspections do not involve a group of people interacting, but a group of people acting independently, inspections have a better chance than walkthroughs or formal reviews of including effective computer support as part of the method. Not only could computer support provide display mechanisms of the product, but the computer could display the criteria list and keep track of the reviewer's progress. The computer could also provide ways for the reviewer to record issues electronically, limiting and maybe even eliminating the need for documents in a review.

3.5 Limitations of Fagan's Inspection Process

Fagan Inspections [4][5] is one of the best known and most popular of the existing review methods. Because this method is so popular, limitations of Fagan Inspections are discussed in this section. As described in Chapter 4, a Fagan Inspection consists of five steps: overview, preparation, inspection, rework, and follow-up. The first three steps are actually a combination of a formal review, an inspection, and a walkthrough. This combination of review methods has made Fagan Inspections seem more formal and therefore more effective than other methods. But each step of a Fagan Inspection has major limitations, and a combination of faulty parts does not result in a healthy whole.

Overview

During the overview of a Fagan Inspection, the author explains the function and design of the software to the inspectors. This step is similar to a formal review with the same limitations. Material presented at the overview is lost to anyone not attending the overview. Moreover, without documentation, the overview cannot be duplicated. This information should be documented and provided with the source code with the intent that individuals not present at the overview are not at a disadvantage.

Preparation

In the preparation operation, inspectors involved in a Fagan Inspection try to comprehend the design and the function of the software. Inspectors also study a list of the types of errors found in recent inspections and a checklist of things to look for in the source code.

There is no step-by-step process for the preparation operation. Should the inspector go over the source code line-by-line, or should he overview the source code, only making sure he understands the functionality? Different inspectors will prepare differently, skewing the results of the final inspection. Fagan himself says that sometimes flagrant errors are found during preparation, but usually not even close to the number of errors found during inspection. If a rigorous process was introduced, the preparation operation could become more productive, saving time during the final inspection.

Inspection

During the inspection step of a Fagan Inspection, the reader paraphrases what the author said in the overview while the inspectors cover every line of the source code. Like walkthroughs, experienced personnel are checking for correct comments, correct spelling, compliance with coding standards, and low-level errors such as initialization of variables. These types of errors could be checked before the inspection by less experienced personnel and software tools. The inspection by experienced personnel should concentrate on errors of logic and functionality.

The only process provided for inspecting the code is the layout of the source code. This may be one reason one experiment in software inspections [16] found a lot of variability in the types of defects found by reviewers using Fagan Inspections.

When a Fagan Inspection is completed, there is not a single thing that can be said for certain about the source code. Possibly some errors are found that would have been very costly if they were not found until later in the development process, but there is positively a lot of room for improvement.

3.6 Limitations of the Cleanroom Development Method

The Cleanroom development method is not a review method, but a software development method. It is included here because of its dependence on code inspections and reviews.

The Cleanroom development method forces developers to pay intellectual attention to the code and its preparation by not allowing them to execute software during development. Developers are expected to use code inspections [4], group walkthroughs [15], and formal verification methods to detect errors. But no specific process or development method is provided for the developers. Cleanroom assumes developers "know" how to program correctly and only need to be forced to do what they know. Here are some other limitations of the Cleanroom development method:

(1) Ensuring code reading does not ensure quality

Cleanroom implicitly suggests that forcing the developers to *read* their products will ensure the quality of the products. It is certainly desirable for developers to scrutinize their work, but it does not ensure quality.

(2) Cleanroom uses the "sledgehammer" approach

Selby, Basili, and Baker [20] assert that:

> The intention in Cleanroom is to impose discipline on software development so that system correctness results from a coherent, readable design rather that from a reliance on execution-based testing. (p. 1028)

Cleanroom treats developers like children. It should be possible to develop coherent, readable code without the use of kindergarten tactics. An inspector who knows that his work will be open to scrutiny by others will produce work products that are more readable and understandable [13][21] without being "forced". Cleanroom also reduces a developer's opportunities to learn. If a developer is never allowed to execute code, an educational opportunity for developing quality code is lost.

(3) Cleanroom uses existing review methods

Developers use the methods described previously to verify products. The limitations of these methods prevent any concrete conclusions from being drawn about a product developed in a Cleanroom environment.

(4) Cleanroom is based on an unsubstantiated claim

Cleanroom is based on the claim that if software engineers are allowed to execute their code, they will lean on the results of the execution for verification purposes. But no evidence to support this claim is presented. Commonly accepted notions are not an acceptable basis for a development method.

What do you **know** about a product that has been developed in Cleanroom? Well, you know that the developer was not allowed to execute the code and the code was independently tested. That is all. Just like the other existing methods, Cleanroom has improved the overall quality of software products. But, also like the other methods, Cleanroom does not ensure the quality of any particular product.

CHAPTER 4. PHASED INSPECTIONS

4.1 Concept Overview

Phased Inspection is a new software review method that addresses the limitations identified in the existing review methods. The concept of Phased Inspections is simple. A Phased Inspection consists of a series of partial inspections termed phases. Each phase addresses one or a small set of related properties that it is deemed desirable for the product to have. Phases are conducted in series with each depending on the properties established in preceding phases. Each phase of is carried out by staff whose training and experience is appropriate for the phase. Some phases, for example, require only elementary knowledge of the implementation language whereas other phases require detailed knowledge of the application domain. A complete Phased Inspection will involve a variety of different software professionals. Each inspector associated with each phase is required to sign a statement after the phase that the product possesses the prescribed property to the best of his knowledge.

A Phased Inspection has two major goals:

(1) To inspect the software in such a way that many important yet different properties are checked explicitly.

(2) To inspect the software in such a way that compliance with each of the required properties is assured, at least informally.

As was noted above, a Phased Inspection consists of a series of complimentary partial inspections. Each phase is designed to support the verification of a single or small set of related properties of the product inspected. There are two types of phases: single-inspector phases and multiple-inspector phases.

4.2 Single-Inspector Phases

A single-inspector phase is a rigidly formatted process driven by a list of unambiguous checks. For each check, the software either complies or it does not. The software cannot complete this type of phase until it complies with all of the checks in the list. This type of phase is performed by a single person whose training and experience is appropriate for the phase. For example, phases that inspect for documentation or source layout properties can be performed by technical editors or junior level engineers. Phases that inspect for software semantics or programming practices properties should be performed by more experienced software engineers. A software engineer may also be assigned to conduct a phase based on his particular areas of expertise that can provide specialized insight relating to the focus of that phase.

Phases and accompanying checklists can and will be different for each implementation of the Phased Inspection process. The phases will depend on the properties that the developer feels are most appropriate for the software being inspected. Checklists not only change from implementation to implementation, but are constantly evolving entities that change as the developer discovers new checks appropriate for verifying the desired properties.

Many checks can be made automatically with little or no human assistance. Automating checks promotes more efficient use of the inspectors' time and provides more complete coverage of the checklist items. When compliance with a checklist item is automatically checked, the inspector assigned to conduct that phase ensures the use of the requisite software tools, monitoring the tools' progress if necessary.

Checklist Development

Checklist development is difficult. It is hard to identify the explicit checks that together ensure a particular quality of the product. Starting points for checklist may be style guides, organization standards criteria, or lists of errors found in previously developed products. For example, previous software written in C may have had elusive errors associated with side effects. But the checklist item should not just say: "Are there no side effects?." That item is *not* explicit enough. Instead, a series of checklist items like: "Are there no assignments in conditional expressions?" should inspect for potential side effects in a product.

Another starting point for checklists may be a desirable property. For example, maintainability is a desirable property. One aspect of maintainability is the difficulty of inserting new code without harmfully effecting other parts of the product. In the language C, *switch* statements that have cases with no *breaks* make it very difficult to insert new cases with the assurance that the *switch* will still function on the previous cases as intended. As a result, a maintainability phase checklist for reviewing C would include an item that stated that all cases for all *switch* statements must have a *break* statement. This is a very simple example, but it demonstrates the process by which properties are turned into checklist items.

Phase Completion

A product may not pass a single-inspector phase of a Phased Inspection until it complies with all of the checklists items associated with that phase. Not all products will pass each phase the first time through. When a product fails to comply with a checklist item, the inspector has to decide whether to continue or suspend his review. If the product fails several checks, or fails a check that requires significant changes, it is clear that the inspector should discontinue reviewing the product. The inspector should also fail the product if it is poorly organized or difficult to inspect. The inspectors are held responsible for the quality of their reviews. It is their prerogative to fail a product they feel does not comply with the spirit as well as the letter of the phase requirements.

4.3 Multiple-Inspector Phases

The second type of phase is designed to inspect for those properties of the software that cannot be captured in an precise yes/no statement. Multiple personnel inspect the product. This type of phase is made up of three parts:

(1) Documentation Inspection
First, the personnel inspect the documentation for completeness related to the property associated with that phase. For example, if the personnel are inspecting for portability properties, they check to make sure that all the assumptions made by the author about the underlying machine are included in the documentation of the product. Anything missing or not clear will have to be rewritten by the author before the phase continues.

(2) Individual Inspection
Each inspector examines the product thoroughly in isolation and prepares an error list. There is not a checklist of precise yes/no items associated with this type of phase, but there is a list of general guidelines to assist the inspectors in reviewing the code.

(3) Reconciliation.
In the reconciliation, the inspectors compare error lists. If there is a discrepancy between the lists, the inspectors will decide if the error was overlooked by an inspector, or is not actually an error. Records are kept of who is incorrect and inspectors that habitually miss errors the other inspectors find should not participate in further inspections. The goal is always that the individual error lists will be identical.

A multiple-inspector phase will usually occur after several single inspector phases that include checking for trivial errors. The inspectors in multiple inspector phases might concentrate on logic and functionality issues, for example, and not have to deal with issues that can be checked by less skilled, less experienced personnel. Previous phases will check for a lot more than trivial errors, and in general, multiple-inspector phases will have less to inspect for than normal Fagan Inspections.

Documentation Inspection

At first glance, multiple-inspector phases look very much like Fagan Inspections, but they are fundamentally different. The documentation inspection replaces the overview of Fagan Inspections. Unlike a Fagan Inspection overview, nothing is presented in a documentation inspection that is not documented in some way. This is *not* an inspection of the design of the software. This step in a multiple-inspector phase *is* an information availability inspection. The question answered is: "Can the inspectors get everything they need to know in order to inspect the product from the product or accompanying documentation?" The product cannot continue in a multiple-inspector phase until the answer to this question is "yes."

The documentation inspection does more than ensure that a product is ready for individual inspection. As was pointed out previously, maintenance costs usually make up the bulk of the cost of a product over its entire lifetime [24]. Because of this, complete documentation is essential for any software product. If the product passes the documentation inspection and the inspectors can obtain enough information from the product and its accompanying documentation to inspect it, any information required for future maintenance of the product is presumably available.

Individual Inspection

When an inspector is isolated in an individual inspection, he is inspecting the product in parallel and compiling error lists for the reconciliation step rather than merely preparing for an inspection as he would in a Fagan Inspection. This step of the multiple-inspector phase ensures the participation of all the inspectors and avoids the possibility of one inspector dominating the review as can happen in formal reviews, walkthroughs, and Fagan Inspections. Since inspectors work at their own pace, individual inspections also prevent inspectors from becoming bored or lost.

Reconciliation

The inspection part of Fagan Inspections is replaced with a reconciliation. Instead of stepping through the product as a group, the inspectors compare the error lists from the individual inspection step. The reconciliation provides a challenge to the inspectors as an incentive for inspecting the code as thoroughly as possible.

4.4 Multiple-Inspector Phase Development

The goal of Phased Inspection is to be as rigorous as possible. Single-inspector checklist driven phases are defined by a list of rules that must be completely satisfied before a product passes a phase. The more explicit and exact the checklist items, the more rigorous the review. But there are desirable properties that are not easily captured in explicit exact statements, and the multiple-inspector phase type was developed with this category of properties in mind. The multiple-inspector type phase is our first approach to managing the psychological component of reviews. It is not as rigorous as we would like and has limitations, but does attempt to address some of the problems associated with reviewing for indeterminate attributes.

The most significant limitation of multiple-inspector phases is a lack of rigor. But, as noted in chapter 3, there is statistical evidence that existing inspections find defects. How do we reconcile these results with the limitations also listed in chapter 3? The most obvious answer is that reviewers are doing something more than the existing review methods dictate; that experienced people "know" how to review a product effectively. If this is the case, the first job in improving the multiple-inspector type phase is to somehow document and systemize exactly what reviewers are doing now when they are using existing review methods. These results could then be used to develop a more rigorous process that will replace the multiple-inspector type phase.

Experienced inspectors may also "know" what to look for, and they can be monitored in order to develop a checklist of things they look for. Examples might be: end cases, special cases, omission of processing, and special properties of certain system, e.g. checking writes/reads to shared variables in tasking systems. As these types of checks evolve to be more specific, they will become phases in their own right or checklist items in already existing single-inspector phases.

Multiple inspector phases might also be improved by including a list of questions about the product that the inspectors must answer correctly before concluding the phase. The question lists may be generated by the author from a stylized question set, or generated by a computer system [3].

Neither the Phased Inspection process nor specific implementations of the process are static. The process should continue to evolve, becoming more rigorous, more dependable, and more flexible. Specific implementations should continually update checklists by including appropriate items discovered in multiple-inspection phases and by removing checklist items that are moved to preprocessing tools.

4.5 Enforcement

Enforcement is an integral part of the Phased Inspection process because it is essential to achieving the goal of rigor, and there are two distinct components. First, there is the enforcement of the checklists. If a computer support toolset is used, it is fairly easy to ensure that inspectors check every product feature relevant to each checklist item, and that the product is checked for compliance with every checklist item. This is a very important part of the Phased Inspection process because it ensures that the inspector is achieving the maximum coverage possible. But, it is not the complete picture of enforcement. The second component of enforcement is psychological. The computer can check if the inspector looks at a *while* statement, but the computer cannot make sure the inspector checks the *while* statement correctly. If it could, that checklist item would be in a preprocessing tool and not in a Phased Inspection checklist. So, the inspector must be given some incentive to do the best possible job he can. One possible incentive is to hold inspectors accountable for products they review. Not only does the inspector pass a product through a phase with computer support facilities, but the inspector must **sign** a document verifying that the product complies with all of the checklist items of the phase. Inspectors could also be held responsible for subsequent failures of the product.

Enforcement is particularly important in the phases that are not driven by checklists - the multiple-inspector type phases. It is harder to verify that the inspector has indeed reviewed the product when there is not list of checks he should have made. The reconciliation part of the multiple inspector phases is intended to use challenge, competition, and maybe a little fear as incentives for doing the best job possible. If an inspector does not review the product properly, and the other inspectors do, this will be very apparent in the reconciliation part of the phase.

4.6 Addressing The Limitations of Existing Methods

Several limitations of existing review methods are discussed in Chapter 3. This section shows how Phased Inspections address these problems.

Rigor in Phased Inspections

Rigor in a review method allows conclusions to be drawn with confidence about a property that holds for the product that has been reviewed. Not only does rigor in a review process allow conclusions to be drawn about the product, but it allows the same conclusions to be drawn each time the product is reviewed irrespective of who is performing the review. As was pointed out in Chapter 3, existing review methods are *not* rigorous, and the lack of rigor makes these review methods undependable. Phased Inspections introduce rigor into the review process, most especially in single-inspector phases. The following are components of single-inspector phases that support rigor:

(1) Elaborate checklists
Single-inspector phases are made rigorous by the elaborate checklists made up of straightforward, clearly-defined questions. A product being reviewed either complies with all of the checklist items or it does not. The results of the review are not dependent on the judgement of the reviewer.

(2) Self-contained phases with specific goals
Each phase is designed to establish one property. The phase may assume that the product has properties established by previous phases in the Phased Inspection process, but no part of the phase will check for anything that does not support the verification that phase's property.

Multiple-inspector phases are not as rigorous as single-inspector phases because they check for desirable properties that cannot be captured in unambiguous statements. But rigor was not ignored in the development of this type of phase. The following are components in the multiple-inspector type phase that promote rigor:

(1) *Does not include an overview or any type of verbal presentation*

(2) *Ensures the participation of all the reviewers*
By having the reviewers inspect the product in isolation and then holding a reconciliation, multiple-inspector phases secure the participation of all the reviewers without tactics that may tempt reviewers to raise trivial issues.

(3) *Reconciliation ensures coverage*
The challenge, threat, and competition created by the reconciliation step assure coverage. If an issue is raised in reconciliation about a section of the product that a particular reviewer did not cover, it will become apparent either when the same issue does not appear on his error list, or when he does not have some justification of why the issue did not appear on his error list.

Unlike existing methods, at the end of a Phased Inspection, specific qualities are known to be present in a reviewed product with a very high degree of assurance. These qualities can be itemized for management, included in maintenance documentation, and used during testing to locate errors. If it is known that there are no uninitialized variables in a product, the developers will not have to check for uninitialized variables when searching for an error discovered in the testing stage.

Efficient use of resources in Phased Inspections

Unlike existing methods, Phased Inspections do not depend on any type of verbal presentation. Any information that is required to understand a product *must* be documented. A product must stand alone, with no dependence on any particular person.

Existing methods also have the problem of not matching types of errors with the skills of the reviewers. Phased Inspections solve this problem by grouping similar types of errors in a phase that is conducted by personnel with the appropriate skills and background. Technical editors conduct phases that deal with spelling, grammar, and other simple comment requirements. Low-level software engineers conduct phases that ensure correct coding practices. Senior software engineers do not even see the product until trivial errors are removed.

Appropriate focus in Phased Inspections

Existing methods tend to focus on defect detection or some other aspect of the product. Phased Inspection phases focus on a single or small set of properties, but the entire process can verify a number of diverse, desirable properties.

Computer support for Phased Inspections

Not only is there no computer support for existing methods, but the structure of existing methods makes it difficult to provide computer support. This is not true for Phased Inspections. The bulk of the phases in a Phased Inspection are single-inspector phases. Single-inspector phases lend themselves quite well to computer support. Product and checklist displays, static analyzers, and documentation displays are just a few of the facilities that can be made available to reviewers. The entire process could be managed by software that controls who is doing what phase and in what order. Chapter 5 presents a prototype of a toolset that is an example of software that can be used to support Phased Inspections.

CHAPTER 5. A PHASED INSPECTION TOOLSET

As was pointed out in Chapter 4, Phased Inspections are well suited for computer support. A prototype of a toolset for supporting Phased Inspections has been developed. This toolset is called **InspeQ**[1] for **In**specting software in **p**hases to **e**nsure **Q**uality. Computer support features that have been implemented in **InspeQ**, along with planned additions, are discussed in this chapter.

5.1 Goals of a Computer Support Toolset

A Phased Inspection toolset should provide services in at least three areas:

(1) Provide computer support for inspectors

A toolset should provide many different displays for an inspector. An inspector should be able to view the work product and checklists with their associated background and rationale information. There should be a facility for the inspectors to communicate with the author electronically, especially in cases where the product must be reworked in order to successfully pass the phase. There should also be tools that allow the inspectors to quickly find and review specific sections of the product. At the very least, a string search mechanism should be present in all text displays.

In the checklist display, there should be some mechanism for marking checklist items with an appropriate tag. For example, the inspector should be able to indicate that the prod-

1. InspeQ is implemented on a Sun 3 Workstation using Hewlett Packard's X Widget set. InspeQ is also implemented on an IBM RS/6000 and a Sun 4 Workstation with the OSF/Motif Widget set.

uct either complies or does not comply with a particular checklist item.

(2) Management of Phased Inspection process

This service of the toolset should be designed to deal with configuration management of the products to be reviewed or in the review process, allocation of staff to the various phases, and management information concerning the state of various inspections.

(3) Enforcement of Phased Inspection process

A toolset should provide enforcement of the Phased Inspection process in two ways. First, it should make sure that the phases are performed in the designated order and by the assigned staff. Second, the toolset should keep track of the inspector's actions and make sure he checks the product for compliance with all of the items of the checklist associated with the assigned phase.

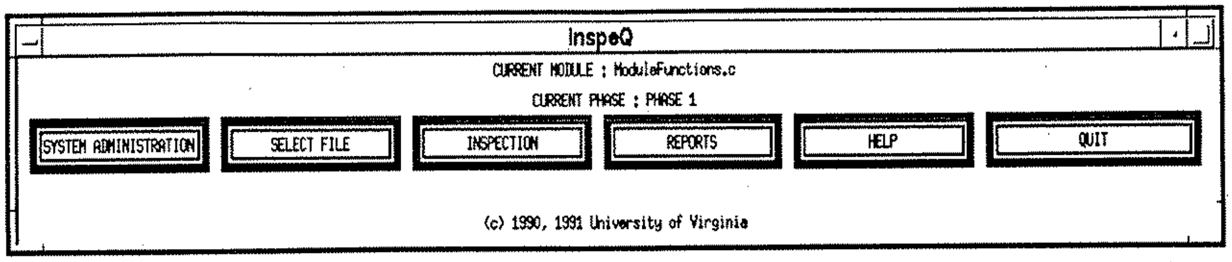
These areas are a minimum set of features required for an effective Phased Inspection toolset.The prototype in development has many of these features, and more will be implemented in the near future. Facilities that have been implemented and those marked for future implementation are presented in this chapter.

5.2 Overview of Toolset

The main window of **InspeQ** is shown in Figure 5.1 . There are six options in the main window. This section contains an overview of each of these options, except for **HELP** and **QUIT**.

All of the display windows, including the main window, display the name of the product[2] being inspected and the phase of the inspection the product is in. The main window, again like all display windows, can be moved, resized, minimized, maximized, and

2. The product displayed in the example figures is C source code from the prototype toolset InspeQ.



CURRENT MODULE : ModuleFunctions.c

CURRENT PHASE : PHASE 1

(c) 1990, 1991 University of Virginia

Figure 5.1 Main Window Display

iconified.

System Management

The management configuration facilities for the option **SYSTEM ADMINISTRATION** have not yet been implemented. When they are, there will be facilities for entering a product into the inspection process, assigning personnel for each phase of the inspection process, and checking the progress of a product through the Phased Inspection process.

Selecting a File

The second option in the main window is **SELECT FILE**. For a user other than the system administrator, this is the first option selected. There are two options for selecting a file:

(1) NEW FILE
This option allows the user to apply the toolset to products that have not entered the Phased Inspection configuration management - products that are not in the process of being inspected.

(2) EXISTING FILE
This option allows the user to apply the toolset to products that have entered the Phased Inspection configuration management.

Inspection

The third option, **INSPECTION**, provides various tools to support the actual process of examining the product. Examples include a general file display, scrolling, and searching facilities that allow the source file to be reviewed rapidly, a facility to permit inspectors to note their conclusions electronically, a syntax-based highlight mechanism that permits various important syntactic structures to be made readily visible, and a display of the appropriate checklist and standards.

InspeQ supports compliance checking of inspectors. When items are checked by inspectors, it is essential that the checks be complete. The compliance support facilities monitor the inspector's use of the tool and the checklists and ensures, to the extent possible, that the inspector is achieving complete coverage. An inspector may not pass a product through a phase until he has marked, in some way, all the items of the checklist of the phase. In the future, it will be possible to associate language features with checklist items, and the inspector will not be able to mark a checklist item until the toolset has verified that he has looked at every feature of the types associated with the checklist item.

Generating Reports

Getting away from paper as much as possible is a continuing goal in development of **InspeQ**, but paper documents have their place. The **REPORTS** option will allow users to print copies of documents associated with the toolset.

5.3 Management Features

As was discussed in the overview section, the management facilities of **InspeQ** have not been implemented. When they are, facilities will be available for entering a product into the inspection process for configuration management. After the product has entered the inspection process, all access to the product must be controlled by **InspeQ** to prevent changes that might invalidate assumptions made about the product after is it inspected.

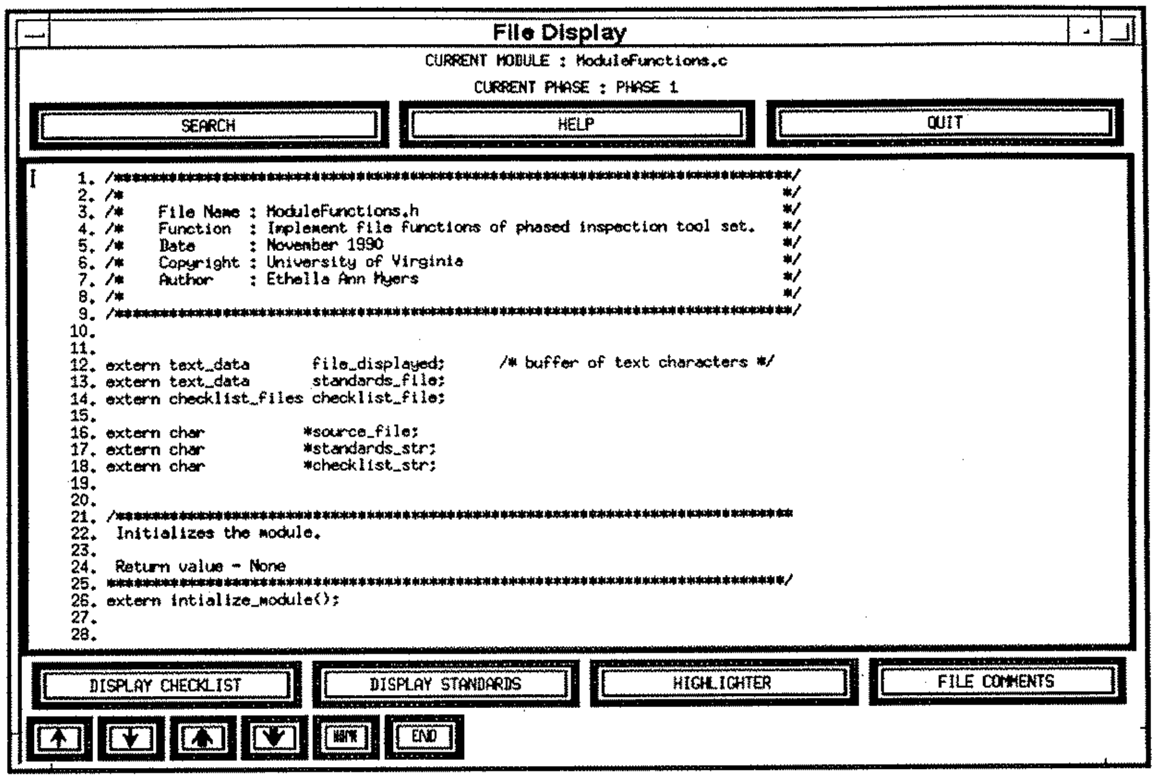
When a product enters the inspection process, the system manager must assign personnel for each phase of the process of inspecting the product. **InspeQ** will make the product available to those personnel, and in some way verify who is performing a particular phase. The tool will also be responsible for electronic mail messages informing the inspector that he has been assigned to perform a particular phase of the inspection process. System managers will also be able to check the progress of a product through the inspection process, making sure inspectors are not too leisurely about getting their part of the inspection process completed.

5.4 Inspection Features

The **INSPECTION** features of **InspeQ** provide tools for the user to support the process of examining the product. All text displays for these tools provide searching and navigation facilities for text, and text that can be edited by the user can be saved so that the user may stop and continue inspecting the product as is convenient. Text navigation is provided in all text display windows by a menu located at the bottom of the window. The first two buttons of this menu are arrows that point up and down. These arrows represent *scroll up one line* and *scroll down one line*. The next two buttons of the menu are larger arrows that represent *page up* and *page down*. The last two button are labeled **HOME** and **END**. **HOME** moves the display to the beginning of the file in the text display, **END** to the end of the file.

Source display

Figure 5.2 is an example of a source display. Like all of the text display windows in



CURRENT MODULE : ModuleFunctions.c

CURRENT PHASE : PHASE 1

```
 1. /************************************************************************/
 2. /*                                                                      */
 3. /*    File Name : ModuleFunctions.h                                     */
 4. /*    Function  : Implement file functions of phased inspection tool set. */
 5. /*    Date      : November 1990                                         */
 6. /*    Copyright : University of Virginia                                */
 7. /*    Author    : Ethella Ann Myers                                     */
 8. /*                                                                      */
 9. /************************************************************************/
10.
11.
12. extern text_data       file_displayed;      /* buffer of text characters */
13. extern text_data       standards_file;
14. extern checklist_files checklist_file;
15.
16. extern char           *source_file;
17. extern char           *standards_str;
18. extern char           *checklist_str;
19.
20.
21. /************************************************************************
22.  Initializes the module.
23.
24.  Return value - None
25. ************************************************************************/
26. extern intialize_module();
27.
28.
```

Figure 5.2 File Display

InspeQ, the source display indicates the name of the product being inspected and the phase of the inspection the product is in at the top of its window. The product is displayed exactly as it appears in the source file except that line numbers are added by **InspeQ** to help the user refer to specific sections of the product.

The source display window is at the top of a hierarchy of window interfaces to tools that facilitate the inspection process. Near the bottom of the source display window is a menu with four options:

(1) *Display Checklist*
Displays the checklist associated with the phase of the inspection process that the work product is in.

(2) *Display Standards*
Checklist items have a 1-1 correspondence to standards items. Standards items provide a guideline for the corresponding checklist item, a rationale for the guideline, and possibly examples to further define and explain the guideline.

(3) *Highlighter*
The highlighter is a tool that allows the user to isolate specified features of the product.

(4) *File Comments*
The file comments window allows the inspector to electronically record comments about the work product.

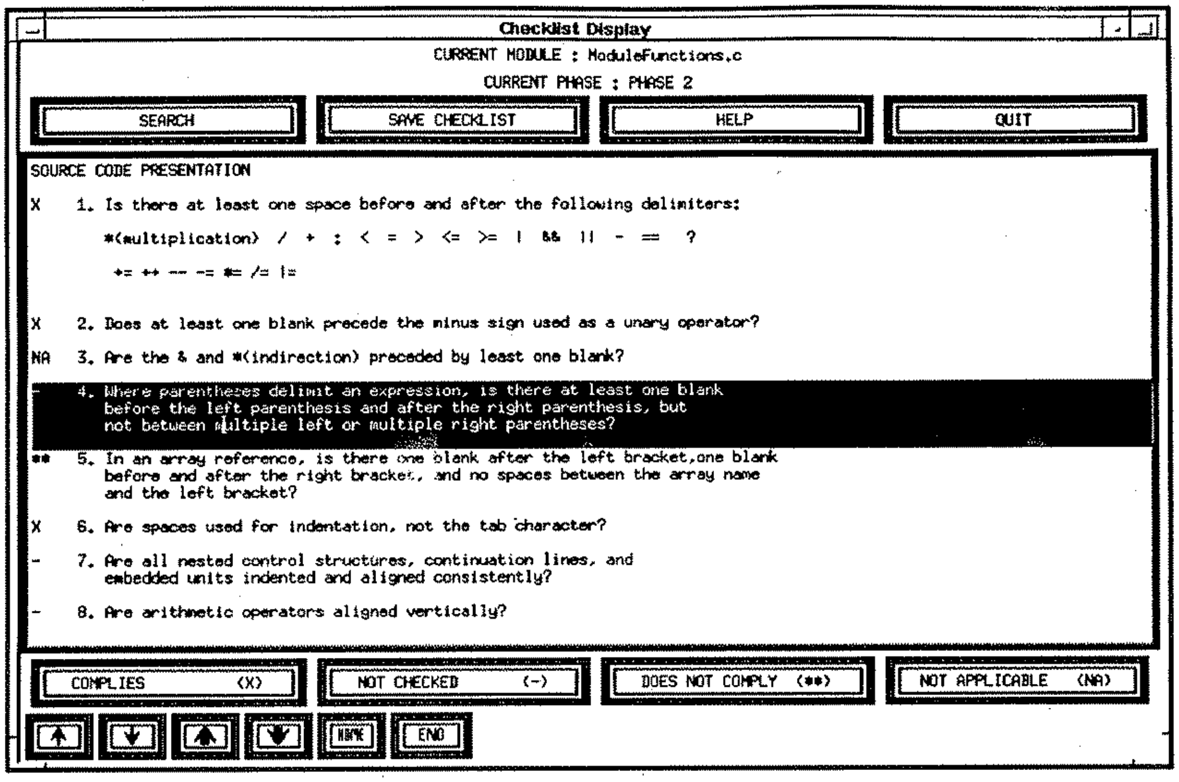
Checklist Display

The checklist display window not only displays the checklist associated with the current phase of the product, but provides the user with four options for marking each of the checklist items. The user selects an item with the mouse. The selected item is displayed in reverse video, and any options selected will be applied to the selected checklist item. In Figure 5.3, checklist item 4 is selected. The use may choose one of four options:

(1) *COMPLIES*
This option indicates that the product complies with the checklist item.

(2) *NOT CHECKED*
This option indicates that the user has not checked whether the product complies with the checklist item or not.

(3) *DOES NOT COMPLY*
This option indicates that the product does not comply with the checklist item.



```
CURRENT MODULE : ModuleFunctions.c
CURRENT PHASE : PHASE 2

SOURCE CODE PRESENTATION
X     1. Is there at least one space before and after the following delimiters:
         *(multiplication)  /  +  :  <  =  >  <=  >=  !  &&  ||  -  ==   ?
          += ++ -- -= *= /= !=

X     2. Does at least one blank precede the minus sign used as a unary operator?
NA    3. Are the & and *(indirection) preceded by least one blank?
-     4. Where parentheses delimit an expression, is there at least one blank
         before the left parenthesis and after the right parenthesis, but
         not between multiple left or multiple right parentheses?
**    5. In an array reference, is there one blank after the left bracket,one blank
         before and after the right bracket, and no spaces between the array name
         and the left bracket?
X     6. Are spaces used for indentation, not the tab character?
-     7. Are all nested control structures, continuation lines, and
         embedded units indented and aligned consistently?
-     8. Are arithmetic operators aligned vertically?
```

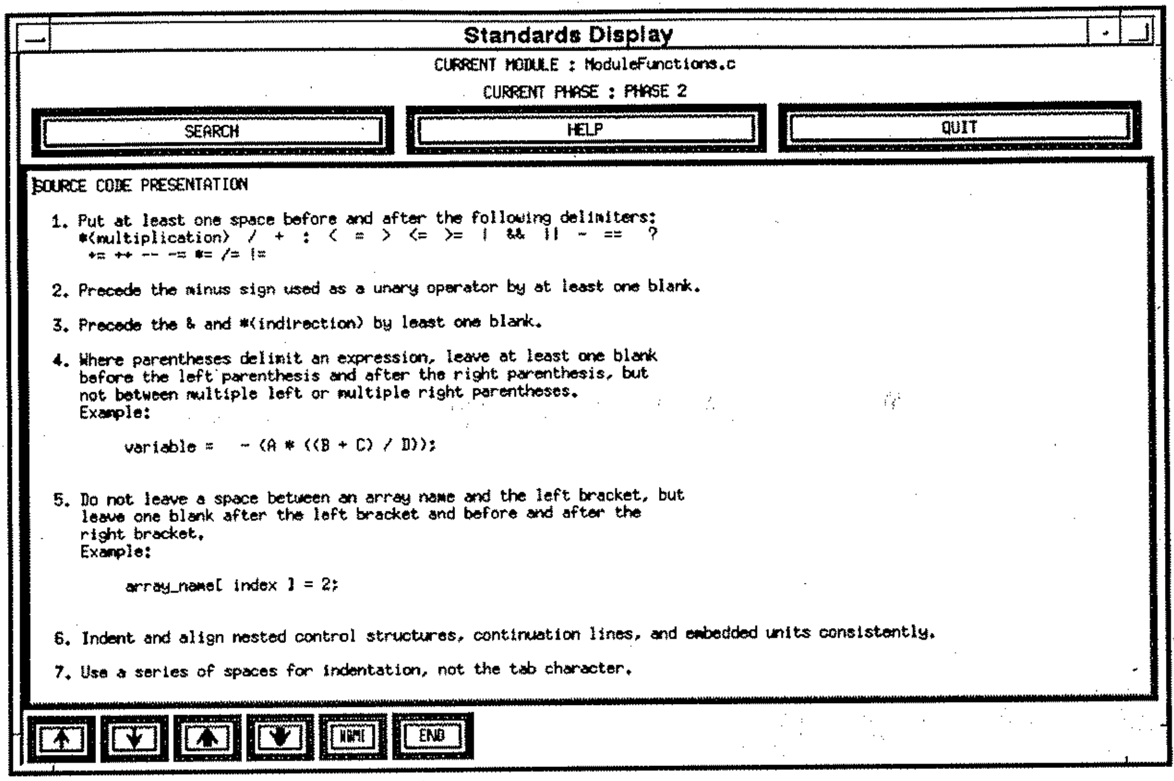
Figure 5.3 Checklist Display

(4) NOT APPLICABLE

This option indicates that selected checklist item is not applicable to the product being inspected.

Standards Display

The standards display window displays the standards file associated with the checklist of the current phase. Standards items have a 1-1 correspondence to checklist items and this is reflected in numbering in the text displays. Standards item 4 in Figure 5.4 gives an example of a correct instantiation of checklist item 4 in Figure 5.3



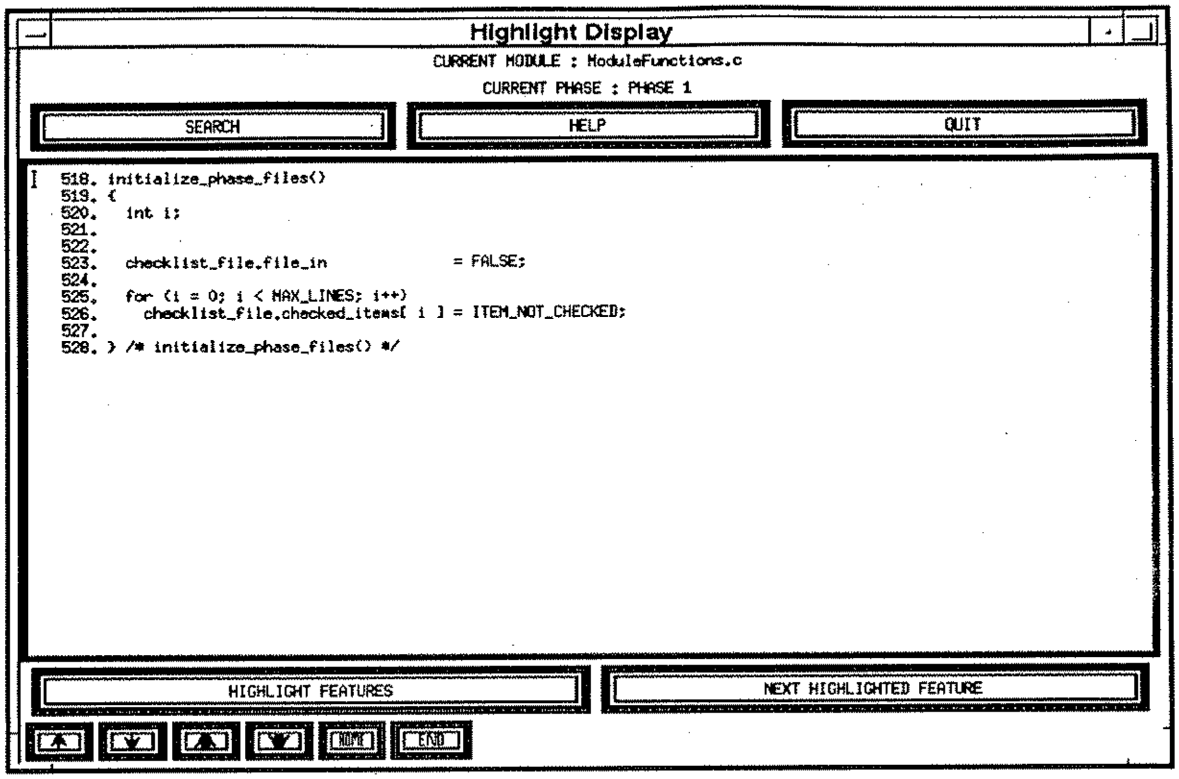
```
CURRENT MODULE : ModuleFunctions.c
CURRENT PHASE : PHASE 2

SOURCE CODE PRESENTATION

 1. Put at least one space before and after the following delimiters:
    *(multiplication)  /  +  :  <  =  >  <=  >=  |  &&  ||  ~  ==   ?
     += ++ -- -= *= /= |=

 2. Precede the minus sign used as a unary operator by at least one blank.

 3. Precede the & and *(indirection) by least one blank.

 4. Where parentheses delimit an expression, leave at least one blank
    before the left parenthesis and after the right parenthesis, but
    not between multiple left or multiple right parentheses.
    Example:

         variable =   - (A * ((B + C) / D));

 5. Do not leave a space between an array name and the left bracket, but
    leave one blank after the left bracket and before and after the
    right bracket.
    Example:

         array_name[ index ] = 2;

 6. Indent and align nested control structures, continuation lines, and embedded units consistently.

 7. Use a series of spaces for indentation, not the tab character.
```

Figure 5.4 Standards Display

Highlight Display

The highlight display window helps the user find and isolate specific product features quickly. The highlighter extracts features from the file and displays them one at a time in a separate window. To begin, the user selects the **HIGHLIGHT FEATURES** command which offers a selection of features for the user to choose from. After the user selects a feature or combination of features, the command **NEXT HIGHLIGHTED FEATURE** will cause the highlight window to display the first feature.

Examples of features of source code that could be highlighted are: *for* statements, *while* loops, *if* statements, and functions. In Figure 5.5, functions in the product



```
518. initialize_phase_files()
519. {
520.   int i;
521.
522.
523.   checklist_file.file_in              = FALSE;
524.
525.   for (i = 0; i < MAX_LINES; i++)
526.     checklist_file.checked_items[ i ] = ITEM_NOT_CHECKED;
527.
528. } /* initialize_phase_files() */
```

Figure 5.5 Highlight Display

ModuleFunctions.c have been highlighted. The function *initialize_phase_files* is currently highlighted and the line numbers indicate this function's position in the original product.

The highlight facility is very useful for a user attempting to achieve the maximum coverage possible. For example, a checklist item might require the user to verify that all *while* statements terminate. The highlight facility would allow the user to highlight all the *while* statements in the product, and sequentially check each one until he has checked them all. Without this facility, the user would have a more difficult time verifying the completeness of his inspection.

Isolating features in a separate window allows the user to concentrate in narrow sections of the product, avoiding distraction by the feature's surroundings. If the user is checking a *switch* statement, he does not need to check how the control variable for the *switch* statement is used before or after the *switch* statement.

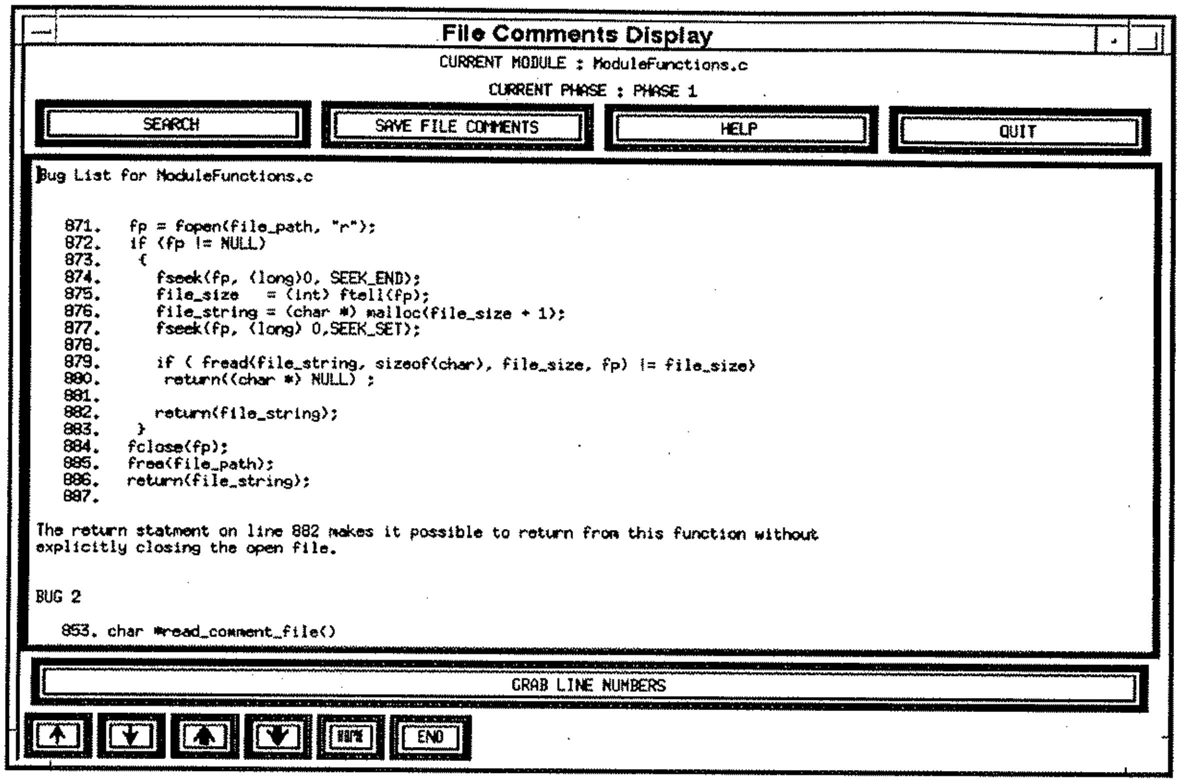
The highlight facility is not implemented completely. It requires syntactic information about the product produced by a parser. Parsers for **InspeQ** have yet to be implemented. Currently, feature locations in inspected products are hardcoded into the implementation of **InspeQ**.

File Comments

An inspector needs to record anything in the product that does not comply with the checklist. The file comments window provides a fully-editable text display for the inspector to do this. The commands controlling the display of the file comments window are roughly equivalent to Emacs text editor commands. The user can use the Emacs commands, the mouse, or the buttons at the bottom of the window to navigate about the file.

Not only can the user type messages, but he can cut and paste text from any text display on the screen or grab selected line numbers from the text display of the product. The user must select the desired lines or text with the mouse by depressing the left mouse button at the beginning of the text and releasing the left button at the end. The selected text appears in reverse video. The **GRAB LINE NUMBERS** command prints the line numbers in a formatted string in the text display of the **FILE COMMENTS** window at the current sprite position. To grab text, the user moves the sprite to the **FILE COMMENTS** window and presses the middle mouse button, placing the text in the desired position.

Grabbing text can be very useful when it is hard to explain but easy to show by example what is wrong with the product. The user would grab the faulty section of code once and



```
CURRENT MODULE : ModuleFunctions.c
CURRENT PHASE : PHASE 1

Bug List for ModuleFunctions.c


   871.   fp = fopen(file_path, "r");
   872.   if (fp != NULL)
   873.    {
   874.      fseek(fp, (long)0, SEEK_END);
   875.      file_size   = (int) ftell(fp);
   876.      file_string = (char *) malloc(file_size + 1);
   877.      fseek(fp, (long) 0,SEEK_SET);
   878.
   879.      if ( fread(file_string, sizeof(char), file_size, fp) != file_size)
   880.       return((char *) NULL) ;
   881.
   882.      return(file_string);
   883.    }
   884.   fclose(fp);
   885.   free(file_path);
   886.   return(file_string);
   887.

The return statment on line 882 makes it possible to return from this function without
explicitly closing the open file.


BUG 2

   853. char *read_comment_file()
```

Figure 5.6 File Comments Display

indicate what the problem is, then grab it again and alter the second selection to show what the text should be.

5.5 Compliance Features

To indicate that a product has passed or failed a phase, the user must select an option under the **INSPECTION** option labeled **PASS CURRENT PHASE.** The user may fail the product, or he may attempt to pass the product. If the user selects **PASS, InspeQ** confirms that the user has marked all of the checklist items with **COMPLIES** or **NOT APPLICABLE**. If not all of the checklist items are marked with one of these two options, **InspeQ** does not allow the user to pass the product through the current phase.

Future versions of **InspeQ** will make it possible to associate product features with checklist items, and the user will not be able to mark a checklist item until **InspeQ** has verified that he has examined every feature of the types associated with the checklist item.

CHAPTER 6. EVALUATION FRAMEWORK

Phased Inspections were developed to create a rigorous, reliable, cost effective review method for software products. Phased Inspections should reduce the testing cost and effort, and make maintenance easier thereby reducing maintenance costs. An evaluation should determine whether Phased Inspections fulfill these expectations.

This chapter presents a framework for evaluation of Phased Inspections. The evaluation is presented in three parts:

(1) The Phased Inspection Concept
(2) Particular Implementations of Phased Inspections
(3) Computer Support Toolsets for Phased Inspections

The results of an evaluation would be used to answer the most important question: "Are Phased Inspections cost effective?" No matter how reliable, dependable, or rigorous Phased Inspections are, if they are not cost effective, they will not be used.

The evaluation framework proposed here is slanted towards inspecting source code. Some of the evaluation framework would be valid for any software work product, but there are other questions that might be raised should the product under review be, for example, design or requirements specifications.

6.1 Evaluation Framework for the Phased Inspection Concept

Evaluating the Phased Inspection concept raises questions that involve both independent and dependent variables. Independent variables are the inputs that must be controlled and/or measured during experimentation. Examples of independent variables that must be considered when evaluating Phased Inspections are:

(1) The application domain
(2) The programming language or other notation
(3) The inspectors' skills and backgrounds
(4) The programmers' skills and backgrounds
(5) Specific implementations of Phased Inspections

The evaluation of the Phased Inspection concept would be made up of a series of evaluations of specific implementations of Phased Inspections. For example, evaluation of the Phased Inspection process would attempt to answer the question of whether effective checklists can be developed overall; evaluation of specific implementations would determine if appropriate checklists could be developed that would ensure the specific qualities associated with the implementation with a high degree of confidence. Each of the questions outlined in the evaluation of the Phased Inspection concept would be used in the evaluation of a particular implementation of Phased Inspections. In addition, there would be concerns related to the attributes ensured by the implementation. For example, if the Phased Inspection is reviewing for concurrency characteristics, an evaluation would consider whether a checklist approach is effective when software is not sequentially executed.

Dependent variables are the results of implementing Phased Inspections. Questions about feasibility of, resources required by, and the resulting quality of products inspected with Phased Inspections are examples of dependent variables. All of these variables are considered when assessing the cost effectiveness of Phased Inspections.

6.1.1 The Independent Variables of Phased Inspection

In every experiment, there are independent variables that must be accounted for. The experimenter must either control the effects these variables have on the results of the experiment, or take them into account when assessing results, or both. In this respect, software experiments are no different from any other type of experiment.

Application domain

A strength of Phased Inspections is the ability to develop phases focusing on qualities specific to an application domain. But are Phased Inspections equally effective over different application domains? The following are concerns that should be raised when evaluating Phased Inspections over different application domains.

- *How much does the domain affect the effort and cost of developing checklists?*
  Some domains are probably more clearly understood than other domains. For example, general correctness Phased Inspections will probably be easier to implement than concurrency inspections. But this is not certain and should be determined by experimentation.

- *If it is found that in certain domains is it costly and difficult to develop effective checklists, then why?*
  Are the desirable qualities of that domain poorly understood, or is it a lacking in the development of Phased Inspection checklists?

- *Do phases that inspect for properties not related to the domain, such as local programming practices, give the same results regardless of domain?*
  The application domain should be considered not only with respect to qualities specific to the domain, but to other desirable characteristics. A Phased Inspection can and probably will include phases that have nothing to do with the domain of the product. Phases that are not related to the application domain should give the same results regardless of the domain of the work product. If they do not, why they do not should be ascertained.

Different programming languages

This evaluation framework emphasizes reviewing source code. The most significant characteristic of source code is the computer programming language it is written in. Therefore, is it appropriate to evaluate how much particular programming languages affect Phased Inspections of source code.

- *Does the language affect the feasibility of and resources expended in the development of checklists and to what extent?*
  The types of errors typically found in certain programing languages may be better understood. An evaluation should experiment over a range of languages to determine the effectiveness of Phased Inspections. For example, is it harder to develop checklists for C than for Ada?

- *Does the programming language used affect the results of Phased Inspections and to what extent?*
  Some languages may lend themselves better to checklist examination. Perhaps one language is easier to inspect than another and the checklists should be developed with that in mind.

Inspectors' skills and backgrounds

People are always the hardest to control in experiments. Their unpredictability makes it difficult to repeat experiments and to draw statistically valid conclusions. Questions that arise include:

- *Does an inspector's familiarity with the language of a source code product affect the results of a Phased Inspection?*
  An inspector's familiarity with the language of a source code product may mask a lack in a checklist. It is acceptable that an inspector's knowledge of the programming language affect the amount of time it takes to complete a phase of the inspection. It is not acceptable that prior experience affect the results of a phase.

- *Does past experience with existing methods affect the inspector's ability to*

perform phases in a Phased Inspection?
As was pointed out in Chapters 3 and 4, inspectors using existing methods may "know" what to look for. An evaluation should determine whether the results of a Phased Inspection are due to the method used and not the prior experience of the inspectors.

Programmers' skills and backgrounds

The programmer's skill and background are reflected in the products produced. This raises the following questions:

- *Do products produced by experienced programmers versus inexperienced programmers have different results?*
  Phased Inspections should produce products with same properties regardless of the state of the product at the beginning of the review. The evaluation should make sure that the quality of a product is ensured because of the Phased Inspection, and not because the product had the quality to begin with.

- *Do products produced by programmers with experience in software review methods versus programmers with no review experience affect the results of a Phased Inspection?*
  The experience programmers have had with software review methods may affect the work product. As noted above, Phased Inspections should produce products with same properties regardless of the state of the product at the beginning of the review.

Specific Implementations

Specific implementations have concerns associated with the overall focus of the implementation. Phased Inspections that inspect for concurrency attributes should be distinguished in an evaluation from general correctness implementations. Some examples of questions that would be specific to the basis of the particular implementation of Phased Inspections are:

- *What type of checks are needed to ensure "correctness" in software products?*
  Do checks that ensure "correctness" in software work products merely reflect typical errors that have been found in the past, or should checks be designed to avoid conditions that are error prone?

- *Can complete coverage be assured in Phased Inspections designed to inspect concurrent software?*
  An evaluation should study whether methods that assure coverage in sequential executable work products apply to concurrent software.

- *Can checklists that support reusability properties be developed?*
  Are there explicit checks that can be made on software work products that promote reusability?

6.1.2 The Dependent Variables of Phased Inspection

The utility of Phased Inspection should be evaluated in three areas: feasibility, resources required, and quality of the resulting products. Feasibility evaluation will not only tell whether the process can be implemented, but will also provide information on ways to improve the process. The resources required will be part of the information used to determine whether Phased Inspections are cost effective. Cost effectiveness is also a function of performance. For Phased Inspections to be worthy of routine adoption in the software development process, they need to establish properties in work products that result in lower costs elsewhere during development or maintenance.

Feasibility

- *Can phases appropriate to particular implementations be developed?*
  Each implementation of Phased Inspections has a focus. That focus might be concurrency, reusability, portability, real-time, etc. In order for the im-

plementation to be split into phases, properties that support the focus have to be identified. Are there properties that support the verification of concurrency, reusability, portability, real-time, etc.?

- *Can appropriate checklists be developed?*
  The success of Phased Inspections depends on this question. If appropriate checklists cannot be developed, the Phased Inspection process will not be rigorous, and will not reliably produce quality products.

- *Does reconciliation work?*
  The goal of the reconciliation of multiple-inspector phases is that the inspectors' error lists be identical. It should be determined just how realistic this goal is. If error lists are seldom if ever similar, then much of the incentive for the inspectors will be lost.

Resources Required

- *What resources are required to develop useful checklists?*
  It may be possible to develop useful checklists, but so hard that it is not worthwhile.

- *Can inspectors complete a phase in a reasonable amount of time?*
  The cost effectiveness of Phased Inspections is directly related to the average amount of time it takes to inspect a product. Phased Inspections will probably not be very useful if it takes 10 hours per phase to inspect 10 lines of code.

- *Is the Phased Inspection process manageable from the system administration point of view?*
  The Phased Inspection process requires quite a bit of management. Personnel have to be assigned to perform each single-inspector phase for each product. The progress of a product through a Phased Inspection must be tracked to make sure inspectors are completing the phases in a timely fashion. Multiple-inspector phases require not only assignment of personnel, but arrangement of meeting times for the documentation inspection and reconciliation. The cost of managing a Phased Inspection must be included in the

overall cost of Phased Inspections.

- *Is the cost of developing checklists reasonable?*
  Actually, the cost of developing checklists can be relatively high since it is a one time cost and the benefits of rigorous checklists are numerous. The more care taken in developing checklists, the more dependable and complete the results of Phased Inspections.

- *What are the life cycle savings in development cost, testing costs, pre-delivery delays, post-delivery service, and maintenance costs?*
  Possible savings from Phased Inspections can include more than reduction in testing costs. If these other savings occur, they should be measured and included in the overall cost effectiveness value of Phased Inspections.

Quality of Resulting Software

- *Do inspectors using the same checklists deliver identical results?*
  In order for Phased Inspections to be dependable, the results of a review must be the same each and every time, regardless of who conducts the review.

- *Can phases depend on previous phases?*
  This question is really asking if single-inspector phases ensure the properties they are supposed to. Can inspectors in subsequent phases rely on the results of all the previous phases? Is this a useful thing to do?

- *Are signatures a useful way to ensure compliance?*
  Computer support can be used to ensure that an inspector thoroughly displayed a product, but cannot ensure that the inspector thoroughly inspected it. Are signatures a useful way to hold the inspector responsible for the quality of the inspection?

- *Are Phased Inspections dependable?*
  Can Phased Inspections reliably ensure high quality work products?

6.2 Evaluation Framework for Computer Support

The evaluation of computer support of Phased Inspection should focus on the word *support*. Once it is determined if computer support is feasible, evaluation should focus on how well a toolset supports the Phased Inspection process. Estimations of the resources required should not come from the development of the a toolset, but how the toolset affects the resources required in Phased Inspections.

Feasibility

- *Is computer support feasible?*
  Whether or not computer support for Phased Inspections is feasible has already been determined by the existence of the prototype toolset presented in Chapter 5.

Resources Required

- *Does computer support reduce the time required for Phased Inspection?*
  It seems intuitive that computer support would reduce the amount of time required for a Phased Inspection. Displays of the product and checklists, management of the process, and enforcement features are all facilities that might reduce time requirements.That might not be the case. It is possible that computer support actually increases the amount of time it takes to complete a phase of the Phased Inspection process. If computer support does reduce effort, the reduction should be measured to help compute cost effectiveness.

- *What is the most effective user interface of computer support?*
  A clear, efficient, easy to use user interface is essential for any Phased Inspection computer support. Inspectors will be loathe to use any tool that is cumbersome or inefficient.

- *What are the basic facilities required for effective computer support of Phased Inspections?*
  There are a lot of potential tools that would support Phased Inspections.

Some of these tools can be fairly expensive to develop and implement. An evaluation should compare the cost of development with resulting increase in the cost effectiveness of Phased Inspections when determining the desirability of support tools.

Quality of Resulting Software

- *Is compliance checking useful for ensuring coverage of the work product?*
  The prototype toolset presented in Chapter 5 forces the inspector to somehow mark all of the checklists items of the checklist associated with the current phase of the Phased Inspection before passing to the next phase. Future versions of the toolset will add compliance checking facilities that will track the inspector's display of the reviewed product. Is this type of compliance checking necessary?

- *Does computer support help ensure high levels of assurance that the desirable properties exist in the work product?*
  Do products that were reviewed by computer supported Phased Inspections have a higher quality than products that were reviewed by unsupported Phased Inspections.

Is computer support of Phased Inspections cost effective?

The answer to this question seems intuitive. Personnel are much more expensive than computer time. Cost reductions because of computer support should be used to assess the cost effectiveness of Phased Inspections.

6.3 Evaluation Analysis

Performing this evaluation requires experiments that can be rigidly controlled and replicated many times to produce statistically significant results. To produce dependable results, each experiment must be designed so that only one of all the independent variables varies over all the replications of the experiment. The results would be related to the variations in the changing variable. A complete evaluation would involve a series of experiments, each designed to assess the effects of one independent variable, each replicated enough times to draw conclusions with a high degree of confidence. For example, an implementation of Phased Inspections designed to ensure general correctness properties in Ada source code written by experienced programmers for flight simulators could be used to experiment with the skills and backgrounds of the inspectors. (See Figure 6.1). But, re-

Controlled Variables

Application Domain:	Flight Simulators
Programming Language:	Ada
Programmers' skill level:	Experienced

Permuted Variable

Inspectors' Skill and Background

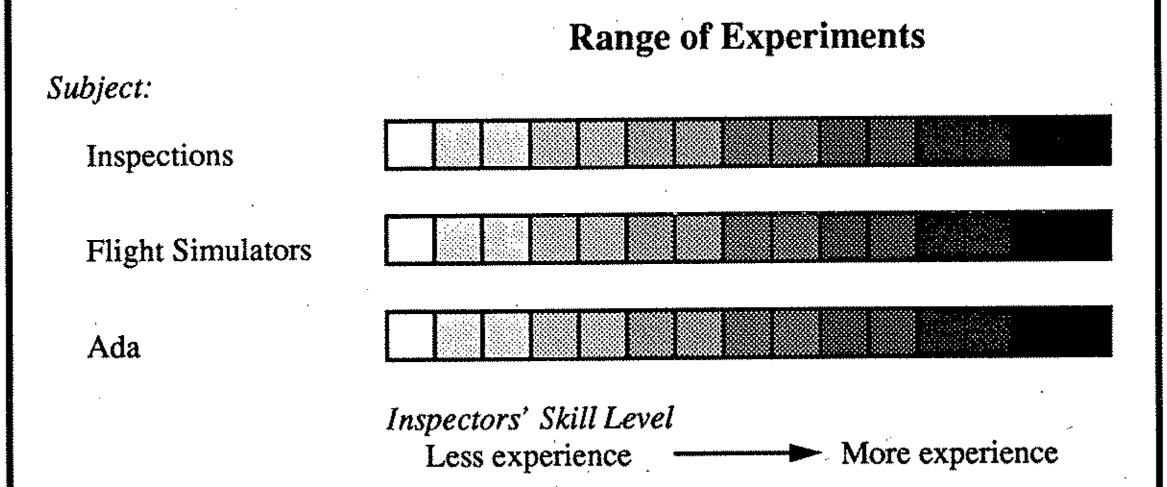


Figure 6.1 Experiment Design

sults from this experiment should not be used to make specific conclusions about Phased Inspections of C source code written for database applications.

Unfortunately, the full-scale experimentation required to perform the evaluation presented here is usually not feasible in a software development environment. The people, effort, and cost required is more than most organizations can afford. That does not mean that experiments with Phased Inspections should not be conducted. Quite the contrary, constrained experiments may not produce conclusive results, but they may certainly provide good indications of the usefulness of Phased Inspections.

CHAPTER 7. PRELIMINARY EVALUATION

An evaluation framework for evaluating Phased Inspections was presented in Chapter 6.Though a complete evaluation is desirable, the full-scale experimentation required to perform that evaluation is not feasible in an academic environment. The resources required are simply not available. When conclusive experimentation is not possible, constrained experimentation can be valuable. Limited, preliminary evaluation can provide some initial indications about feasibility and effort, guiding future experimentation and further development of the Phased Inspections process and supporting toolset **InspeQ**.

This chapter presents a preliminary evaluation of Phased Inspections. This preliminary evaluation cannot be used to draw any definitive conclusions about the resources required and the quality ensured by Phased Inspections. The most that can inferred are some preliminary indications of feasibility. Keeping that in mind, some of the feasibility concerns presented in Chapter 6 are discussed in this chapter. In addition, changes in **InspeQ** that were implemented because of the evaluation are included.

This evaluation is limited in its conclusions. Substantially more work is required before general conclusions can be drawn about how effective and reliable Phased Inspection is in producing high-quality software work products.

7.1 Experiment Description

An evaluation of the Phased Inspection process was performed using graduate students in the Computer Science Department of the University of Virginia. The instantiation of Phased Inspections used in the evaluation is presented in Appendix A - a general instantiation of Phased Inspections for C source code. The target of the inspection was source code of the prototype toolset **InspeQ**. Two experiments were conducted. The first evaluated single-inspector phases; the second multiple-inspector phases.

7.1.1 Single-Inspector Phase Experiment

Four inspectors performed phases 1-4 of the General C Source Code implementation of Phased Inspections. Three of the inspectors had industrial experience and experience with the programming language C. One inspector did not know C and had no industrial experience. One of the inspectors had previous experience with software review methods (See Figure 7.1).

The products inspected in this part of the evaluation were source code files from the toolset, **InspeQ**, that displayed the main window of the toolset. These products were chosen because of their relative simplicity. There are no complicated algorithms in these pieces of software. There are 643 lines of commented source code. Comments are included in the number of lines inspected since comments are not excluded from inspection.

7.1.2 Multiple-Inspector Phase Experiment

Two sets of 2 inspectors participated in the evaluation of the multiple-inspector type phase. In Group 1, both of the inspectors had extensive experience with C, and one of the inspectors had prior experience in industry and with inspections. In Group 2, neither of the inspectors much experience with C. One of the inspectors had industrial experience and previous experience with inspections (See Figure 7.2).

Product Inspected:

Files Inspected:	MainWindow.c MainWindow.h
Lines Inspected:	643

Experiment Description:	*Inspectors:* A	B	C	D
C Experience	No Experience	Experienced	Experienced	Experienced
Industrial Experience	No Experience	Experienced	Experienced	Experienced
Software Review Methods Experience	No Experience	No Experience	No Experience	Experienced
Phase Performed	Phase 1	Phase 2	Phase 3	Phase 4
Time Required	1.5 hrs	1.45 hrs	1 hr	1.5 hrs

Figure 7.1 Single-Inspector Phase Experiment

The products inspected were source code files from the toolset, **InspeQ**, that perform all the file operations. These products have 1015 lines of commented source code. The documentation inspection for each group took about 1 hour to complete. The average time for individual inspection was 2 hours, and the reconciliations for both groups took 1 hour.

7.2 Evaluation Results

This preliminary evaluation in no way resembles a thorough evaluation of Phased In-

Product Inspected:

Files Inspected:	ModuleFunctions.c ModuleFunctions.h
Lines Inspected:	1015

Inspectors:

Experiment Description:	**Group 1**		**Group 2**	
	E	**F**	**G**	**H**
C Experience	Experienced	Experienced	Some Experience	No Experience
Industrial Experience	Experienced	No Experience	No Experience	Experienced
Software Review Methods Experience	Experienced	No Experience	No Experience	Experienced
Individual Inspection Time	1.5 hrs	1.25 hrs	1 hr	2 hrs

Figure 7.2 Multiple-Inspector Phase Experiment

spections. Therefore, it would be inappropriate to present results in the format suggested by the evaluation framework presented in Chapter 6. Instead, some preliminary indications of feasibility are presented.

7.2.1 Implementation Development

The implementation of Phased Inspections used in this preliminary evaluation was designed to inspect C source code for general correctness properties. This instantiation is presented in Appendix A. An example standards and checklist for phase 2 of this instantiation is presented in Appendix B. The basis for the phases and the checklists was the *Ada Quality and Style: Guidelines for Professional Programmers* produced by the Software Productivity Consortium [22], published texts on C and programming style [9][10][12], and the personal experience of the author with the types of errors typically found in C source code.

It was not difficult to develop the phases and checklists of the C source code instantiation of Phased Inspections. Publications listing specific programming practices that promote correctness in C source code were numerous, and the Ada style guide [22] provided the readability and commentary properties (See Appendix A).

On the other hand, an instantiation of Phased Inspections developed for Science Applications International Corporation to inspect for reusability in Ada was more difficult to develop. The author had no experience with developing Ada software or developing reusable software products. An extensive literature search failed to produce more than a few specific checks and programming practices that would promote reusability in Ada source code. Indicating that information hiding and abstract data types should be used in reusable software is not a sufficient basis for checklist items in a Phased Inspection.

It is reasonable to conclude that the basis of an instantiation of Phased Inspections has an effect on the resources required to develop its phases and checklists. A well-defined, reasonably understood property will translate to explicit checks more easily than a less clearly defined property.

7.2.2 Single-Inspector Phase

The results of the single-inspector phase experiment suggest that it is feasible to conduct a series of single-inspector phases. For the most part, inspectors were able to decide whether the product complied with the checklist items of a phase or not. The inspectors generally felt that **InspeQ** was understandable and easy to use, and that the computer support provided by **InspeQ** was useful in conducting single-inspector phases.

Time required

None of the inspectors required more than 2 hours to complete their assigned phases. Inspector B also noted that as he became more familiar with the checklist items associated to the phase he was conducting, his rate of progress improved rapidly.

Display facilities

All of the inspectors who participated in the single-inspector phase experiment indicated the highlight and file comments features were convenient and useful. Inspectors A and B were of the opinion that the checklist display did not provide adequate options for marking checklist items.

Effect of past experience

In the single-inspector phase experiment, inspector A, who had very little knowledge of C, had a more difficult time inspecting the product than the inspectors who had experience with C. Inspector A felt that his inspection could have been more effective if the checklist items had been more explicit. An inspector with previous experience with the programming language or inspections can possibly augment his understanding of ambiguous checklist items with that experience. The more explicit, understandable, and unambiguous the checklist items, the more likely that the results of a phase are the same regardless of the skills and background of the inspector conducting the inspection.

7.2.3 Multiple-Inspector Phase

Lack of computer support

There was a lack of computer support for the multiple-inspector phase experiment. The only facilities available in the toolset for use with a multiple-inspector phase is the display of the source file and the file comments feature. The design documentation had to be distributed to the inspectors in printed form because there are no facilities for displaying it in the toolset. The inspectors felt that it was difficult and time consuming to not have access to a display of the design documentation. Future **InspeQ** enhancements should include facilities that will better support multiple-inspector phases.

Individual Inspection

The process of individual inspection has to be better defined. The inspectors were not sure of what they were supposed to do in the individual inspection. Inspectors E and G inspected both the design and source code. Inspector F concentrated on detecting defects in the source code. And inspector H reviewed the design documentation. Inspectors focusing on different aspects of the work product are unlikely to produce similar error lists.

Reconciliation

A knowledge of the programming language affected results in both experiments, but more so in the multiple-inspector phase experiment. The inspectors in Group 1 with similar and extensive backgrounds in C detected similar errors. Where they did not detect similar errors, the inspectors quickly came to an agreement over whether there was actually an error and why one of the inspectors had overlooked it.

In Group 2, the inspectors with little experience with C produced greatly dissimilar error lists. They were less successful than Group 1 at detecting defects in the source code, and tended to focus on detected defects in the design of the product rather than the implementation.

Past experience, either with inspections or with the programming language affected performance. This indicates a need for more direction in the multiple-inspector phases, but it is also possible that inspectors who do not have sufficient experience should not be included in multiple-inspector phases.

7.3 Evaluation of Computer support

This evaluation was too limited to assess conclusively the contribution of **InspeQ** to the Phased Inspections process. It did however uncover shortcomings of the first version of **InspeQ** that were corrected in a revised version.The changes that were made are:

(1) Defects in the toolset source code found during the single-inspector phase experiment by abnormal termination.
These defects have been corrected in the current version of **InspeQ**.

- The C library function *strcpy* was used improperly. S*trcpy* will sometimes cause a segmentation error if a null string is an argument in the function call.
- In the file comments display, if the C library function*fopen* failed, the user was not notified. The contents of the file comments window was not saved, even though the toolset indicated that it was. Occasionally, this error caused the toolset to terminate abnormally.

(2) Defects in the toolset source code found in the multiple-inspector phase experiment.
These defects have been corrected in the current version of **InspeQ**.

- One viable path of execution allowed a function to return without explicitly closing a file that had been successfully opened.
- Memory allocated for use in a function was not deallocated.

- The null character was not explicitly set in a string variable input by the C library function *fread*. This could cause a segmentation error and abnormal termination.
- A viable path of execution in a function did not return the correct value.
- Two functions specified to return a true/false value for successful file saves had no explicit return statements.
- There were several sections of the product where the code was unclear, or there was insufficient commentary.

(3) **FILE COMMENTS** *Window*

In the beginning of the development of Phased Inspection, we thought inspectors would prefer to write comments, corrections, etc. directly on a hardcopy of the product being inspected. An inspector pointed out that:

- Hardcopy notations make communication with the author more difficult.
- Not having a history of defects found in electronic form would make past history defect compilation more difficult.

In response to these valid points, the **FILE COMMENTS** window was implemented in **InspeQ**.

(4) Changes in string search mechanism

In the first version of **InspeQ**, the search mechanism was inconvenient for the user. There was no repeated search, and users could not press the return key to initiate a string search. Both of these problems have been fixed in the current version of **InspeQ**.

(5) Addition of **NOT APPLICABLE** *and* **DOES NOT COMPLY** *to choices for marking checklist items*

The initial implementation of **InspeQ** had two choices for marking checklist items -

MARK or **CLEAR**. This selection scheme did not provide for indicating that the product had been checked but did not comply with a checklist item, nor did it provide a way to indicate that the checklist item did not apply to the product being inspected. The current version of **InspeQ** has 4 choices for marking checklist items.

(6) Ability to "grab" text from any display for the File Comments window

When the grab text facility was initially implemented, it was only possible to cut and paste text from the source file display window. It was pointed out that it would be extremely useful to be able to include checklist items that the product does not comply with in the file comments. The user can now grab text from any display (including displays not part of the toolset) and paste it in the file comments window.

Future Extensions

These are some of the facilities that are being considered for future versions of **InspeQ**. These are only a few of the possible features that could be added to the toolset to increase its value in supporting Phased Inspections.

(1) **PREVIOUS HIGHLIGHTED FEATURE** *selection in highlight window*

In the highlight display window, it is possible to sequentially display highlighted features by selecting **NEXT HIGHLIGHTED FEATURE**. Inspectors found it inconvenient that they could not display previously highlighted features. Future versions of **InspeQ** will provide this capability.

(2) Regular expression searches

The string search mechanism and the highlight facility are not sufficient for providing quick and selective navigation of the source file. Regular expression searches are being considered as way to extend navigation capabilities of **InspeQ**.

(3) Display of multiple parts of the work product

In the current version of **InspeQ**, the user can adjust the size of the display of the source file, but he cannot view more than one section of the source file at a time. Displaying multiple sections of the source file may be useful and this feature will be considered.

(4) Integration of design documentation presentation

The design documentation for the products reviewed in this evaluation was provided in paper form. The inspectors all found this extremely inconvenient, and it was suggested that it would be very useful of a toolset not only displayed documentation, but integrated the display of the design documentation with the display of the source code being reviewed.

CHAPTER 8. CONCLUSIONS AND FURTHER RESEARCH

8.1 Extensions of Phased Inspection

Phased Inspections were built to be tailorable to an organization's specific needs. Phases are added and deleted, checklists modified, and the method itself should grow and adapt to the changing requirements of a software development environment. Future extensions can ease the inspector's job while promoting the rigor and reliability goals of Phased Inspection.

8.1.1 Execution

There is no place in a rigorous review process for hand simulation of source code. Existing review methods tend to ignore execution, relegating anything but static analysis to the testing phase of software development. It is not inappropriate to allow inspectors limited execution of the source code product. Extensive research would be needed to set the limits of execution, and developers of debugging tools can testify to the difficulty of implementing a tool that would adequately display execution information, but the value of inspection execution simulation could far out weigh the cost.

8.1.2 Graphical Representations

Illustrations of concepts, graphs and symbols are often more effective than words at describing relationships and ideas. Graphical representations of components of software products are often easier to understand and easier to verify than the original components. This section presents possible graphical extensions to Phased Inspections.

Expressions

Expressions in software products can be very complex. Figure 8.1 is an example of a graphical representation of an expression.

Expression:

```
(((A + B) * (C - D)) ** 3)
```

Graphical Representation:

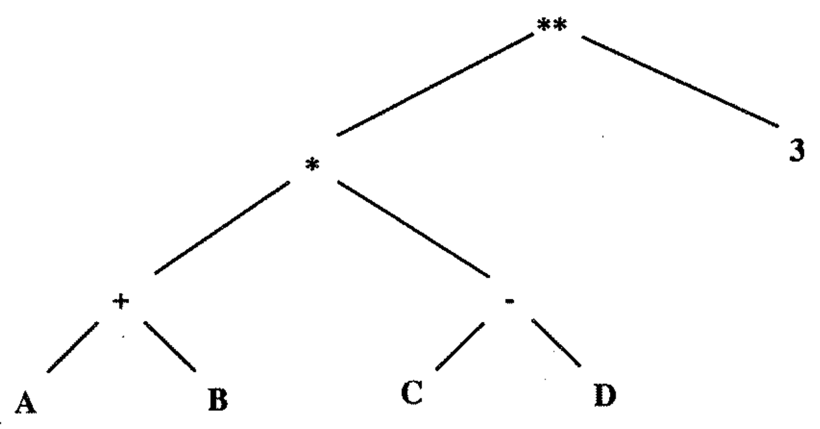


Figure 8.1 Graphical Representation of an Expression

Subprogram dependency hierarchy

A facility for displaying the subprogram dependency hierarchy of software programs can assist inspectors in understanding the often complex interrelationships in large-scale programming projects. This type of display could be very useful in determining maintainability, concurrency, and reusability properties.

Product block reductions

In product block reduction displays, lines of the product being reviewed are replaced in the display by icons or mnemonic strings. Sections of software products can be verified and then replaced, gradually reducing the complexity of the product being reviewed. In figure 8.2, the main function of a C program that performs binary multiplication is displayed. In the reduction, the declarations and the input of the multiplier and multiplicand have been verified and replaced. Verification and replacement would continue until only 1 line remained in the main function of the program.

8.2 Conclusions

Phased Inspections as a verification tool is not meant to replace but to complement testing in software development; checking for defects more expensive to detect in testing stages of development, ensuring properties not detectable by automated analysis.

Phased Inspections were developed to be a rigorous, reliable, tailorable, heavily computer supported, and cost effective software products review method. Desirable qualities of a software product reviewed within the framework of Phased Inspections will be ensured with a high degree of confidence.

Harlan Mills said [14]: "The best debugging tool . . . is the human mind." Phased Inspections provide a way to combine the flashes of inspiration associated with the human mind with the predictability of the computer - the best of both worlds.

Description of Product:
Main Function in C Program that Performs Binary Multiplication

Before Reduction

```
main()
{
  int n;
  unsigned int md, mq; /* multiplicand and multiplier */
  char instring[33];
  int numbits;
  char eol;

  printf("\n Number of bits in an integer? ");
  scanf("%[^\n]%c",instring,&eol);
  numbits = atoi(instring);
  n = 1;
  if (numbits >= 32)
     printf("\n\n Sorry, can only multiply 31 or less bits\n\n");
  else
   while (n != 0)
   {
    printf(" Multiplicand? ");
    n = scanf("%[^\n]%c",instring,&eol);
    if (n == 0) break;
    if (strlen (instring) > numbits)
     {
      printf("\n\n Error, Too many bits in multiplicand\n");
      break;
     }
    md = atob(instring);
    printf(" Multiplier? ");
    n = scanf("%[^\n]%c",instring,&eol);
    if (n == 0) break;
    if (strlen (instring) > numbits)
     {
      printf("\n\n Error, Too many bits in multiplier\n");
      break;
     }
    mq = atob(instring);


    carry_save(numbits,md,mq);
   }  /* end of while */
} /* end of main */
```

After Reduction

```
main()
{

  DECLARATIONS

  printf("\n Number of bits in an integer? ");
  scanf("%[^\n]%c",instring,&eol);
  numbits = atoi(instring);
  n = 1;
  if (numbits >= 32)
     printf("\n\n Sorry, can only multiply 31 or less bits\n\n");
  else
   while (n != 0)
   {

    INPUT MULTIPLIER AND MULTIPLICAND

    carry_save(numbits,md,mq);
   }  /* end of while */
} /* end of main */
```

Figure 8.2 Product Block Reduction

Appendix A INSTANTIATION OF PHASED INSPECTIONS

General C Source Code Implementation of Phased Inspections

The following is an example instantiation that can serve as a starting point for Phased Inspection implementations. There are 4 parts of each phase overview:

(1) Type of Phase
There are two types of phases:
- Single Inspector
- Multiple Inspector

(2) Personnel
There are 4 types of inspectors:
- Technical Editor
- Junior Software Engineer
- Software Engineer
- Senior Software Engineer

(3) Overview
The overview part of the phase description briefly describes the property or set of properties the phase is designed to verify.

(4) Types of Requirements
This section provides a list of the types of requirements that would be included in the phase. This list not only describes existing requirements, but also serves as a guideline for placing future requirements in the Phased Inspection Process.

Phased Inspections for C

This groups of phases is a general source code inspection implementation. There is no emphasis on a particular property such as portability or maintainability. This particular phase inspection would be used to inspect source code written in the language C.

Phase 1 - Software Comments Inspection

(1) Type of Phase

- Single Inspector

(2) Personnel

- Technical Editor

(3) Overview

- The source text is inspected for compliance with comment requirements. Spelling and grammatical errors, layout of headers, and copyright notices are the type of items checked in this phase. These checks are not essential for the software to execute correctly, but they are important for maintenance, reuse, and legal requirement properties.

(4) Types of Requirements

- Ensure that comments comply with the standards for placement and required content. For example, standards may require that each module is headed by a comment block that includes the author's name, organization, date, and copyright notices.
- Ensure that comments are free from spelling errors.

Phase 2 - Source Code Layout Inspection

(1) Type of Phase

- Single Inspector

(2) Personnel

- Technical Editor

(3) Overview

- The source text is inspected for compliance with the source code layout requirements.

(4) Types of Requirements

- Put at least one space before and after the following delimiters: & * / +: < = > <= >= | && || - == ?.
- Where parentheses delimit an expression, leave at least one blank before the left parenthesis and after the right parenthesis, but not between multiple left or multiple right parentheses.

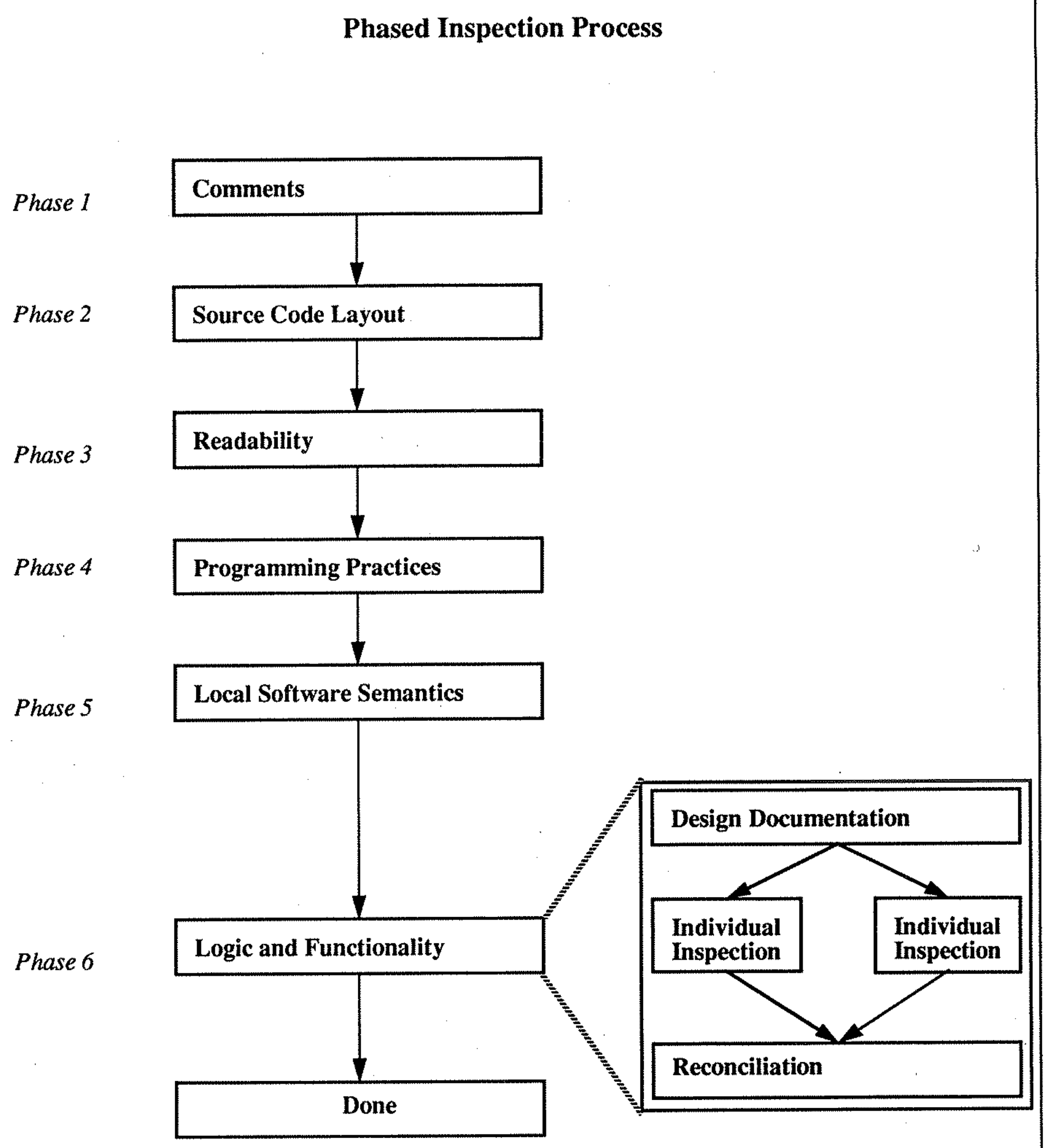


Figure A.1 General Instantiation of Phased Inspections

Phase 3 - Readability Inspection

(1) Type of Phase

- Single Inspector

(2) Personnel

- Junior Software Engineer

(3) Overview

- The source text is inspected for compliance with the readability requirements.

(4) Types of Requirements

- Use underscores to separate words in a compound name - miles_per_hour, entry_value.
- If a project has accepted abbreviations, maintain a list and use only abbreviations on that list.

Phase 4 - Programming Practices Inspection

(1) Type of Phase

- Single Inspector

(2) Personnel

- Software Engineer

(3) Overview

- The source text is inspected for compliance with the programming practices requirements.

(4) Types of Requirements

- Use parentheses to specify the order of subexpression evaluation where operators from different precedence levels are involved, and to clarify expressions.
- Do not use the assignment operator in boolean expressions.

Phase 5 - Local Software Semantics Inspection

(1) Type of Phase

- Single Inspector

(2) Personnel

- Software Engineer

(3) Overview

- The source text is inspected for occurrences of know specific problems in program logic.

(4) Types of Requirements

- Make sure the control variable of a while loop is updated within the loop body.
- Verify that all files successfully opened are explicitly closed.

Phase 6 - Logic and Functionality Inspection

(1) *Type of Phase*

- Multiple Inspector

(2) *Personnel*

- Group of Senior Software Engineers

(3) *Overview*

- The items in this phase are used as a guide for inspectors who are checking the logic and functionality of the product. These items require judgment calls on the part of the inspector.

(4) *Types of Requirements*

- Validation - does the source code execute according to the rules of requirements and design documents.

Appendix B EXAMPLE STANDARDS AND CHECKLIST

The following are the standards and checklist associated with the second phase of the general C source code instantiation presented in Appendix A.

Standards for Phase 2 - Source Code Presentation

1. Put at least one space before and after the following delimiters: *(multiplication) / + : < = > <= >= | && || - == ? += ++ -- -= *= /= !=
2. Precede the minus sign used as a unary operator by at least one blank.
3. Precede the & and *(indirection) by least one blank.
4. Where parentheses delimit an expression, leave at least one blank before the left parenthesis and after the right parenthesis, but not between multiple left or multiple right parentheses.
   Example:
   ```
   variable =   - (A * ((B + C) / D));
   ```
5. Do not leave a space between an array name and the left bracket, but leave one blank after the left bracket and before and after the right bracket.
   Example:
   ```
   array_name[ index ] = 2;
   ```
6. Use a series of spaces for indentation, not the tab character.
7. Indent and align nested control structures, continuation lines, and embedded units consistently.

Example :

```
if (mice_counter == 2)
   {
     while (mice_dead >= 4)
       {
         number_of_rats = number_of_cats * 3;
         mice_dead--;
         switch (number_of_cats)
           {
             case 1 :  kill_a_mouse();
                       break;

             case 2 :  birth_a_mouse();
                       break;

             case 3 :  birth_a_cat();
                       break;

             case 4 :  kill_a_cat();
                       break;

             default:  kill_all_rats();
                       break;
           } /* end switch */
       } /* end while */
   } /* end if */
```

8. Align operators vertically to emphasize local program structure.
   Examples:

```
if (slot_a >= slot_b) then
  {
    temporary    =  slot_a;
    slot_a       =  slot_b;
    slot_b       =  temporary;
 }


x   =     a * b
      +   c * d
      +   e * f;


y   =   (a * b) + c
                - 3.5
     +  (2 * d) - e;
```

9. Organize declarations as a table.

10. Provide at most one declaration per line.
    Examples:

```
    int      c = 0;
    float    d = 0.0;
    long     condition;
    char     str;
```

11. Use blank lines to group logically related lines of text.

12. Mark the top and bottom of a function definition.
    Example :

```
/***************************************************/
main()
{
   .
   .
} /* end main() */
/***************************************************/
int shuffle()
{
  .
  .
 } /* end shuffle() */
/***************************************************/
```

13. Write no more than one simple statement per line.

14. Break compound statements over multiple lines.

15. Adhere to a maximum line length limit for source code.

Checklist for Phase 2 - Source Code Layout

1. Is there at least one space before and after the following delimiters: *(multiplication) / + : < = > <= >= | && || - == ? +=++---=*=/=!=
2. Does at least one blank precede the minus sign used as a unary operator?
3. Are the & and *(indirection) preceded by least one blank?
4. Where parentheses delimit an expression, is there at least one blank before the left parenthesis and after the right parenthesis, but not between multiple left or multiple right parentheses?
5. In an array reference, is there one blank after the left bracket, one blank before and after the right bracket, and no spaces between the array name and the left bracket?
6. Are all nested control structures, continuation lines, and embedded units indented and aligned consistently?
7. Are spaces used for indentation, not the tab character?
8. Are arithmetic operators aligned vertically?
9. Are declarations organized as a table?
10. Is there at most one declaration per line?
11. Are blank lines used to group logically related lines of text?
12. Are the tops and bottoms of each program or function marked?
13. Is there no more than one simple statement per line?
14. Are compound statements broken over multiple lines, not continuing past the maximum line length?
15. Do all the statements adhere to the maximum line length limit?

BIBLIOGRAPHY

1 Ackerman, A. Frank, Priscilla J. Fowler, and Robert B. Ebenau,"Software Inspections and the Industrial Production of Software," in *Software Validation*, ed. H.L. Hausen, pp. 13-40, Elsevier Science Publishers B.V., North-Holland, 1984.

2 Buck, Robert D. and James H. Dobbins, "Application of Software Inspection Methodology in Design and Code," in *Software Validation*, ed. H.L. Hausen, pp. 41-56, Elsevier Science Publishers B.V., North-Holland, 1984.

3 Dyer, M. and A. Kouchakdjian, "Correctness Verification: Alternative to Structural Software Testing," *IBM Systems Journal*,vol. 32, no. 1, pp. 53-59, 1990.

4 Fagan, Michael E., "Design and Code Inspection to Reduce Errors in Program Development," *IBM Systems Journal*, vol. 15, no. 3, pp. 182-211, 1976.

5 Fagan, Michael E., "Advances in Software Inspections," *IEEE Transactions on Software Engineering*, vol. SE-12, no. 7, pp. 744-751, July, 1986.

6 Fagan, Michael E. and John 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.

7 Freedman, Daniel P. and Gerald M. Weinberg, *Handbook of Walkthroughs, Inspections, and Technical Reviews*, Little, Brown Computer Systems Series, Little, Brown and Company, Boston, Toronto, 1982.

8 Freedman, Daniel P. and Gerald M. Weinberg, "Reviews, Walkthroughs, and Inspections," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 1, January, 1984.

9 Kernighan, Brian W. and P. J. Plauger, *The Elements of Programming Style*, Second Edition, McGraw-Hill Book Company, New York, 1978.

10 Kernighan, Brian W. and Dennis M. Ritchie, *The C Programming Language*, Prentice Hall Software Series, Second Edition, Prentice Hall, Englewood Cliffs, New Jersey, 1988.

11 Lamb, David Alex , *Software Engineering: Planning for Change*, pp. 193-199, Prentice Hall, Englewood Cliffs, New Jersey, 1988 .

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

13 McConnell, Peter R. H. and Wolfgang B. Strigel, "Results of Modern Software Engineering Principles Applied to Small and Large Projects," in *National Computer Conference*, pp. 273-281, MacDonald, Dettwiler, and Associates, Richmond, British Columbia, Canada, 1984.

14 Mills, Harlan D., “Software Development,” *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 265-273, December, 1976.

15 Myers, Glenford J., *Software Reliability: Principles and Practices*, John Wiley & Sons, New York, 1976.

16 Myers, Glenford, “A Controlled Experiment in Program Testing and Code Walkthroughs-Inspections,” *Communications of the ACM*, pp. 760-768, September 1978.

17 Petroski, Henry, *To Engineer Is Human: The Role of Failure in Successful Design*, St. Martin’s Press, New York, 1985.

18 Remus, Horst, “Integrated Software Validation in the View of Inspections/Reviews,” in *Software Validation*, ed. H.L Hausen, pp. 57-64, Elsevier Science Publishers B.V., North-Holland, 1984.

19 Russell, Glen W., “Experience with Inspection in Ultralarge-Scale Developments,” *IEEE Software*, pp. 25-31, January 1991.

20 Selby, Richard W., Victor R. Basili, and F. Terry Baker, “Cleanroom Software Development: An Empirical Evaluation,” *IEEE Transactions On Software Engineering*, vol. SE-13, no. 9, pp. 1027-1037, Sept. 1987.

21 Shelly, Gary B. and Thomas J. Cashman, “Implementation of Structured Walkthroughs in the Classroom,” in *Handbook of Walkthroughs, Inspections, and Technical Reviews*, Little, Brown Computer Systems Series, pp. 425-434, Little, Brown Computer and Company, Boston, Toronto, 1982.

22 Software Productivity Consortium, *Ada Quality and Style:Guidelines for Professional Programmers*, Van Nostrand Reinhold, New York, 1989.

23 Weinberg, Gerald M., *The Psychology of Computer Programming*, Van Nostrand Reinhold, New York, 1971.

24 Yourdon, Edward, *Structured Walkthroughs*, Yourdon Press Computing Series, Prentice-Hall, Inc., Englewood Cliffs, New Jersey,1989.