A Brief Examination of the Impact of Late Detection of Defects on Software Projects and the Costs Associated with Source Code Inspections

Richard Chambers, Northern Telecom Inc.

January 13, 1997

Table of Contents

Introduction

Introduction to this Report

There is a fairly constant debate concerning the design of the software development process at the Atlanta lab which has been ongoing for at least the four years the author has been working at the Atlanta lab, ATP. Every person in the lab has a personal opinion as to which particular tool, technique, or methodology will be the deciding factor to enhance productivity, reduce costs, and provide products on schedule and within budget.

Source code inspection is presently a hot topic at ATP due to the large number of problems and quality issues associated with the latest AccessNode release, AN12. This report was written as a part of a personal investigation into the economics of defect reduction processes in general and source code inspections in particular.

Unfortunately, the author of this report, while able to provide quite a bit of circumstantial evidence as to the cost effectiveness of code inspections, is only able to provide mostly qualitative rather than quantitative evidence of their effectiveness. The author has attempted to detail some of the steps of the current ATP software development and verification process so that these details, within which hide a number of accepted costs, can be rethought in a more critical light.

The summation of the literature indicates that source code inspection by trained and experienced personnel embedded within an effective source code inspection process will improve time to market and reduce development costs. But only when an organization's training and software development processes support and encourage an effective source code inspection process will the true benefits be realized.

[Parnas 86] points out that the software development process is a concept, an idea rather than an actuality. Every company has an instantiation of that idealized process. In the end what is of greatest significance to quality and defect reduction efforts is the ability and attitude of the people in an organization. [Brooks 87] as well as [Weinberg 82] echo the point that tools, techniques, or methodologies can't possibly fulfill expectations without organizational focus and management team activity that exploits those tools, techniques, and methodologies so as to realize their potential for quality improvements.

Focus of this report

The focus of this report is to:

- provide a brief summary of some investigations into defect removal steps that have been published;
- look at the costs of defects on the software verification stage of software development;
- look at the impact of source code inspections on those costs.

This report uses data from the Nortel AN12 VTBM project records to investigate the impact of late detection of defects on Verification. It is the opinion of the author that rigorous inspection of source code prior to designer test and verification would have provided higher quality software, reduced the costs of developing AN12, reduced the schedule slippage, and provided management with better data to make the decisions they made.

The author must point out that verification of the AN12 software was started by a deliberate decision by management with the knowledge that development was still ongoing. It is evident from the project records that verification was started before the product development was complete but the action was taken as a recovery mechanism in an attempt to meet customer commitments.

Summary of Conclusions

The summation of the literature indicates that source code inspection by trained and experienced personnel embedded within an effective source code inspection process will improve time to market and reduce development costs.

However source code inspections must be part of a larger, integrated defect detection and removal process. Inspection of requirement and design documents removes potential defects from the final product before those defects can be implemented in the source code. Testing the final product provides a filter for defects that are easier to identify at that stage than at earlier stages of development.

At Nortel, software undergoes a verification/testing phase. But as can be seen in the discussion below which includes data from the AN12 release of AccessNode, testing by itself is expensive when progress must be stopped in order to correct problems in the software being tested. Source code inspections have a positive effect on verification progress as they remove defects before testing commences.

A Review of the Software Development Process at Nortel

Most products developed by Nortel are complex in nature. The problems the products are designed to solve are "wicked" problems with a large amount of technical risk. Usually, the solution that will be implemented is not known when development begins. Part of the development process is involved in exploring the problem to solve while trying to solve it.

The software development process at Nortel follows the waterfall model. The waterfall model is basically a series of phases in the development of a product in which the outputs of one phase are the inputs into the next phase. The standard phases are:

- Requirements Definition, usually with customer involvement
- Requirements analysis and architectural design, usually with customer involvement
- detailed design, coding, and unit test by design
- verification against requirements with maintenance activity to correct identified defects
- deployment and maintenance activity to correct identified defects found after deployment

Each phase marks a greater investment into the product being developed. Each phase also provides an opportunity to identify and correct defects introduced into the product by work within the previous phase.

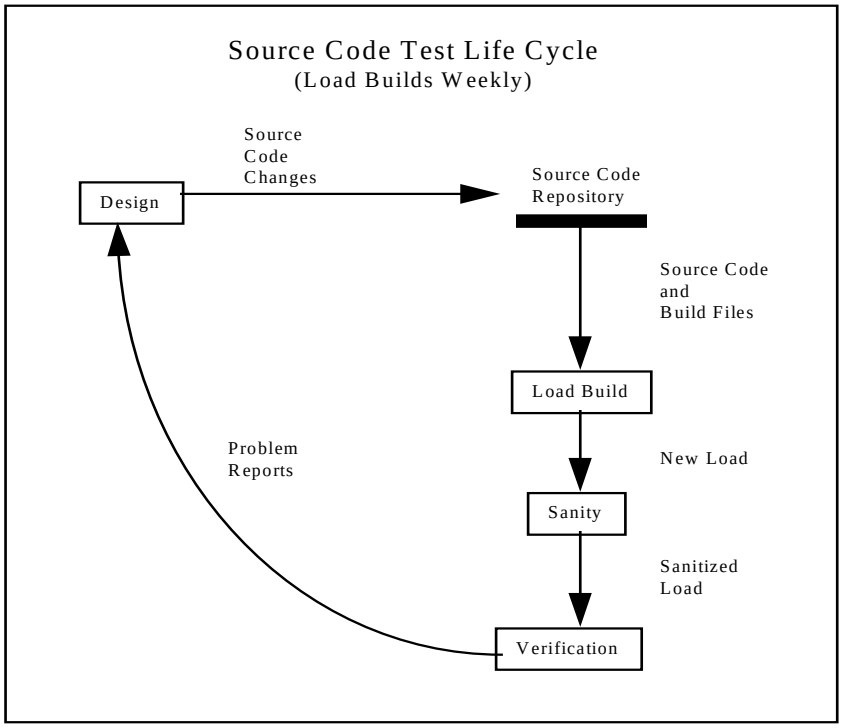
Within the AccessNode development group there are always efforts to reduce time to market through concurrent engineering by rearranging the flow of engineering activities to be as parallel as possible. Unfortunately, attempts such as the parallel development and verification schedule of AN12 have not been successful in their goal of reducing the amount of chronological time from project start to project completion.

The attempts usually depend on low defect density of asynchronously developed products which are then integrated as Verification is beginning the verification phase. The low defect density is never realized at the system level though there are always pockets of excellence primarily due to the individual differences of the developers. (See [Guindon 87] for a discussion of individual cognitive differences and the impact on programming. [Boehm84] specifies a productivity multiplier of 4.18 for the Personnel/Team Capability cost driver in his COCOMO cost model which is almost twice the next highest cost driver, Product Complexity, at 2.36 and is only exceeded by project complexity.)

Verification Phase Workflows

During the verification phase, work flows look something like the figure below. Every week, a new load is generated using the source code in the source code repository (PLS). The load is then checked for sanity, which requires a series of basic test cases to determine if the system is usable. Problem

reports may be generated by the Sanity team but the major focus of sanity is determine if the system is usable.



If the load passes sanity, it is propagated to the Verification test systems. Verification then runs test cases or performs problem report retest using the new load.

Problem reports are tracked using a Nortel proprietary database system known as the Problem Resolution System (PRS). The three letter acronym, PRS, has become a stand-in for the phrase problem report in general. It is used as a verb, “PRS this problem.” as well as a noun “Write a PRS for this.” The PRS cycle is a touch complex and the resolution of a particular PRS can involve negotiation between a number of parties over several weeks. The PRS cycle has the following major steps:

- PRS originated (status is pending) by verification or in response to a field problem
- PRS screened and routed to a group for resolution (status is open)
- PRS is accepted for resolution by the group (status is accepted) or re-routed to a more appropriate group for resolution (status is re-route followed by open)
- PRS is solved by using it as a reason for an update into the source code repository or it is replied with some reason as to why it either isn’t a problem (incorrect test case etc.) or can’t be solved
- PRS is retested by the originator and either closed or failed

Verification is performed on the system level requiring AccessNode hardware, test equipment, and peripherals. The AccessNode Verification standard estimate for Verification productivity is five test cases per five hour lab shift so long as test cases pass. Obviously, if the first test case run on a shift uncovers a major defect, the productivity may be zero test cases if the shift is taken over by Design as they attempt to diagnose and determine the defect cause.

In addition to actually running the tests, verification personnel must also handle the overhead of planning the verification shift, setting up and tearing down test equipment, writing shift reports, and writing PRSs (defect reports which are filed in a database known as the Problem Resolution System which manages the flow of defect reports). As PRSs are solved by Design and the updated software made available to Verification, solved PRSs must be retested and either closed or failed and rerouted back to Design.

Verification personnel attempt to optimize the use of available lab time by selecting test cases to run in a given shift which will require the same test setup and which test functionality without known defects that can stop a shift. In addition, design personnel will usually tell the verification personnel testing software of which areas will have problems and when those areas could be tested.

Basic Defect Detection and Removal Economics

Impact of Late Detection of Defects

In [Boehm 81], Barry Boehm shows that industry cost data supports the assertion that the cost to fix a defect introduced in a phase increases if the defect is not discovered until a later phase. The relative cost to fix a defect varies by the type of defect, the phase in which it was introduced, the phase at which it was discovered, and the size of the project.

For example, let us take the case of a defect introduced at the design stage that is not discovered until after deployment to the field. Let us further assume that this particular defect will require a field retrofit due to its seriousness.

We would have the following types of costs associated with the rework and retrofit of the software:

- the cause of the defective behavior must be tracked down and corrected;
- the software package must be regenerated;
- the fix must be tested and validated as well as the upgrade path from the defective load to the new load;
- the package must undergo regression testing to determine if functionality has been adversely affected by the fix; and
- there is an opportunity cost associated with development and verification performing rework rather than new development.

In addition to the internal, development costs there are also costs to Nortel and Nortel customers:

- there may possibly be changes to the system documentation which would have to be updated and reissued to customers who have already taken delivery;
- there may be customer outages or field problems due to the defect which reduces customer satisfaction and increases the workload of TAS (first line customer support), Field Support, and System Engineering;
- while the defect is being corrected and tested, customer orders must be managed possibly impacting customer commitments and contractual obligations and certainly affecting revenue velocity;
- the ordering system must be updated with the new load and existing orders must be modified to pick the new load rather than the old load;
- existing customers must be notified of the new load and retrofit tapes must be sent out to the existing customer base; and
- either customers or Nortel installation must upgrade existing systems possibly impacting service and certainly costing in terms of extra wages and travel expense.

If this defect were caught at the design review, there would be no customer impact or associated costs nor would there be the source code rework costs incurred by development and verification. The costs would be the rework of a single document, the design document with the possible addition of changes to the product specification if the defect resolution requires changes in the specification.

The standard cost used by AccessNode Customer Service for the AN12.01 to AN12.10 retrofit (approximately 100 sites) and the AN12.10 to AN12.21 retrofit (approximately 350 sites) is US$200 per site to cover tape replication and distribution costs. This means that the high defect density of

AN12 will cost Nortel some US$90,000 in making and distributing software retrofit tapes alone. This figure doesn't include other costs such as the travel of trained upgrade teams which were required for some of the 100 AN12.01 upgrades due to deficiencies in the upgrade software and the additional overhead costs from support and administrative organizations both within Nortel and Nortel's AN12 customers in order to compensate for the deficiencies of the AN12 software.

There is one, multi-million dollar, customer who has refused to deploy AN12 due to problems found in the customer's own lab testing. While AN12 is being re-worked to an acceptable level of quality, the customer is deploying an interim solution. Each remote network element deployed by the customer costs Nortel some US$10,000 in additional hardware according to the marketing representative for the customer in order to provide the ring functionality of AN12.

A Brief Discussion of the Impact of Defects on Project Failure

In [Beizer 84], Boris Beizer describes a study by Abe, Sakamura, and Aiso concerning a software project study performed at Hitachi Software Inc. which Beizer describes as the Japanese project bankruptcy study. The study attempted to provide a project completion predictor based on the test case completion rate of 23 projects. The goal was to be able to determine project status and predict project schedule changes based on the rate of test case completion by Verification.

The reader should remember that verification is a quality assurance step in software development. The purpose of verification is to determine if the number and types of defects in the product are acceptable to the user. If verification has a low test case completion rate, this is indicative of a high defect density in the product being tested. A high defect density requires additional work by the development team to remove defects and improve the quality of the product to a level the customer is willing to accept.

Beizer divides the verification phase into three subphases. The first subphase is the period of time when the test case completion rate is low due to the instability of the system under test. Usually at this point, the system is actually undergoing integration testing and everything isn't working well. The second subphase, started after system shakedown has eliminated the gross defects and system integration is complete, usually has a much faster rate of completion. The third subphase of testing is when subtle, difficult to characterize defects begin to appear slowing completion rate down once more.

These three subphases correspond with three types of test cases, high impact test cases that test basic functionality, medium impact test cases that test the most used and/or visible behavior of the system being tested (which tends to be covered by designer testing as well), and the low impact test cases that test subtle operational characteristics which haven't been touched by previous tests.

If the percentage of test cases completed is graphed against time, the curve will be an S-shaped curve with the first part of the curve having a small slope indicating a slow test case completion rate, the second part of the curve having a high slope indicating a much higher test case completion rate, and finally a third part of the curve having a smaller slope as the test case completion rate again slows.

The report from Abe, et. al made the following conclusions:

- the average duration of the first phase, across the 23 projects, was 22% of the available time;
- if the onset of the second phase was delayed beyond the 55% point, bankruptcy (i.e. project failure) was inevitable;
- for the successful projects, the average duration of the first phase was 15% of the available time and for the bankrupt projects it was 97% of the available time; and
- the duration of the second phase for successful projects was 57% of the time and for bankrupt projects 29% of the time.

These results indicate that what ever can be done to move the curve so that subphase 1, the integration period of verification, takes a small amount of the overall verification schedule will be repaid by the improved possibility of meeting the original schedule and budget. In addition, if the acceleration in the rate of test case completion can be improved the possibility of meeting the original schedule and budget is also improved.

Defect Removal Methods and Efficiencies

In [Jones 86] Capers Jones reports defect removal efficiencies (the percentage of the existing defects discovered by the operation) which are extremely sobering. The modal efficiency for the most effective type of defect removal method, modeling or prototyping, is 65% followed by the second most effective, formal code inspections at 60%. His conclusion are based on observational data on internal IBM programs and from studies of Fagan on defect removal methods. Jones concludes:

> In searching for a balance between defect removal efficiency, schedules, and costs, the weight of evidence indicates that a series which skips or omits design reviews and design inspections will suffer from high tail-end costs, when the undiscovered bugs start showing up during integration and testing.
>
> The most effective combination today appears to be (1) formal design inspections of the critical sections of a system, (2) modeling or prototyping using a high-speed method such as an interpreted language, and (3) normal testing. The extra time and effort at the front for the inspections and prototyping will be compensated for by very short test cycles, with low defect repair and minimal overtime costs.

The reader should be aware that when Capers Jones speaks of inspections he is referring to a procedure that is more rigorous than a walkthrough or review. See [McConnell 93] for a discussion of reviews and inspections. See [Glib 93] for a detailed discussion of inspection methods and processes which are improvements to the Fagan inspection method.

Table 1: Defect removal efficiency for programming defect removal methods [Jones 86]

Removal Step	Lowest Efficiency	Modal Efficiency	Highest Efficiency
Personal checking of design or documents	15%	35%	70%
Informal group design reviews	30%	40%	60%
Formal design inspections	35%	55%	75%
Formal code inspections	30%	60%	70%
Modeling or prototyping	35%	65%	80%
Personal desk checking of code	20%	40%	60%
Unit testing (single modules)	10%	25%	50%
Function testing (related modules)	20%	35%	55%
Integration testing (complete system)	25%	45%	60%
Field testing (live data)	35%	50%	65%
Cumulative efficiency of complete series	93%	99%	99%

The reader should also be aware that different types of defect removal steps tend to have higher efficiencies with certain classes of defects with efficiencies depending on the type of software as indicated from [Glass 92] below.

[Glass 92] provides a summary of three studies on defect removal, two experimental [Meyers 78] and [Basili 87], and one from a "massive real-time software project" [Collofello 89]) with the following summation:

- reviews are more effective than testing in error removal, and they tend to be more cost effective as well. The data here are not unanimous, however.
- in kinds of testing, functional testing (testing behavior against specification) tends to find more errors than structural test (testing pathways through the program), and to find them more cheaply.
- in kinds of reviews, design reviews are far more cost effective than code reviews, but code reviews tend to find more errors.

[Glass 92] also provides the following notes based on the reported findings of the above studies:

- the "number of faults observed, fault detection rate, and total effort in detection depended on the type of software tested." In other words, perhaps the choice of testing methods should be application dependent [Basili 87].
- "Code reading detected more interface faults than did the other methods;" "functional testing detected more control faults than did the other methods." In other words, perhaps the choice of checkout methods should be dependent on the kinds of errors sought [Basili 87].
- "When asked to estimate the percentage of faults detected, code readers gave the most accurate estimates while functional testers gave the least accurate answers." In other words, poking around in the code seems to give people a better perspective on what they've done [Basili 87].
- "There is a tremendous amount of variability in the individual results." In other words, who does the checkout may be more important than how it's done [Meyers 78].
- "The overall results are rather dismal." In other words, none of the error-removal processes, and in fact even combinations of the processes, was very good at identifying all the errors that were present [Meyers 78].
- there was "a negative correlation ... between subject's prior walkthrough/inspection experience and their performance ..." In other words, code reviewers may get tired of the experience over time [Meyers 78].
- "Unfortunately, much of the data was inconsistent and unreliable.... The low number of data points seemed to mirror the disinterest of the developers in recording data.... No one checked the quality of the quality assurance data." In other words, even the use of project data is prone to problems in research analysis [Collofello 89].
- "The high [success] associated with the design review was surprising." In other words, a lot of errors were found very cheaply at design review time [Collofello 89].
- "Interestingly, one reliability assurance technique - testing - was not cost effective." In other words, testing, though necessary is a pretty expensive way of doing business [Collofello 89].

Cleanroom, Additional Evidence Supporting Code Inspection Effectiveness

[Selby 87] describes experimental evidence which indicates a software development process known as the Cleanroom software development methodology is an effective methodology in both reducing costs and increasing software quality. Cleanroom specifies that the programmers can't perform designer test or unit test. Design must focus on inspection as the primary defect removal procedure. The software is actually tested by a separate Verification group which provides the test results back to Design.

The conclusions of the article are:

- most of the developers were able to apply the techniques of Cleanroom effectively (six of the ten Cleanroom teams delivered at least 91% of the required system functions);
- the Cleanroom teams' products met system requirements more completely and had a higher percentage of successful operationally generated test cases (incidence of duplicated failures between verification runs were higher with Cleanroom teams however meaning that some reported failures weren't fixed before the next test run);
- the source code developed using Cleanroom had more comments and less dense control-flow complexity;
- all ten Cleanroom teams made all of their scheduled intermediate product deliveries, while only two of the five non-Cleanroom teams did;
- although 86% of the Cleanroom developers indicated that they missed the satisfaction of program execution to some extent, this had no relation to the product quality measures of implementation completeness and successful operational tests; and
- 81% of the Cleanroom developers said that they would use the approach again.

The author isn't recommending that ATP institute cleanroom software development process since the author believes that such a radical change would be too wrenching at ATP's current level of maturity. The author is instead presenting this experimental result as further evidence that rigorous source code inspection has a definite and positive impact on software quality, defect density, and development schedule.

The Elements of an Effective Source Code Inspection Process

Source code inspection is an official part of the Nortel software development process. Code inspected is a milestone which indicates source code writing is complete and designer test is about to commence. But anecdotal evidence at ATP indicates that source code is reviewed rather than inspected. Many times the source code inspection is performed as a necessary checkmark activity during designer test or after designer test is complete without the full rigour needed to make inspection a cost effective activity.

The major problem with source code inspection is the inspection procedure is tedious to carryout. It is much more fun and satisfying to jump into testing as reported by the participants of the Cleanroom development study above. This major problem is one that requires managerial oversight of well defined processes and a human resources plan that supports such processes.

An effective source code inspection process requires:

- a data collection, storage, and reporting mechanism which provides a way for inspectors to improve inspection checklists and methods as well as to provide information about defect density in subsystems and modules for management decisions;
- a training program for moderators so that the inspection process will be effectively supervised at the working level;
- a training program for participants in structured and object-oriented programming methods along with target domain training so that the participants will have a common vocabulary and vision (see [Guindon 87]);
- an emphasis on source code inspection being as necessary a pre-requisite to designer test as writing the source code itself;
- the creation of a team environment so that the participants will be able to function as a team with the common goal of assessing the quality of the source code being inspected while identifying defects within a comfortable, accepting group; and
- management support of the costs and effort required to implement and manage a source code inspection process despite its visible overhead.

From my personal experience as well as from [Glib 93] and [McConnell 93] a good inspection and review environment would include:

- static analysis tools to catch syntactic errors and to flag possible error conditions such as interface definitions, unusual language constructions, and error prone semantic patterns;
- source code level symbolic debuggers which allow the user to step through modules being inspected so as to observe the actual behavior of a section of source code;
- periodic refresher training for moderators and participants on the checklists and methods using practice source code with known defects injected so as to provide a necessary reality check; and
- good supply of conference rooms with manuals as well as terminals to allow the use of a source code browser to research questions during inspections.

The Costs of Source Code Inspections

Source code inspections take time and cost money.

Assuming the standard 1996 Nortel North American cost per developer of $124,000 per year (including all benefits) it costs roughly $59.50 per hour per employee engaged in an inspection. In addition, employees involved in an inspection are not available for other work.

Working the arithmetic, for three developers (recommended by [McConnell 93] from [Bush 89] based on experience at Jet Propulsion Laboratories) to inspect 5,000 lines of third generation language source code (a typical feature size at ATP) at the rate of 400 lines per hour (Nortel standard rate) would cost about $2,230 (12.5 hours per person at a cost of $740 per person). Because of the difficulty of concentrating for long periods of time (Nortel standard is two hours per inspection session, no more than two sessions per day) it will take anywhere from three days to a week to complete the inspection depending on the workload of the inspection team and scheduling conflicts.

Defects identified must be corrected and the overhead of the inspection process such as data recording must be accommodated as well. Changes may require a second inspection depending on the consensus of the inspection team.

In addition to inspection manpower costs, there are additional facility costs (conference rooms, additional computing facilities for inspection data tracking) and overhead costs of training, checklist preparation, and data recording. Of these additional costs, the largest appear to be additional floorspace for conference rooms and the costs of training (which are not only the costs of the training but the opportunity costs of not having the trainees producing deliverable products).

Source Code Inspections Compared to Testing

There are really two kinds of testing performed in most software development organizations: designer testing in which the programmer subjects the source code to tests at the source code unit level and verification testing in which testing is performed by someone from a different department at the system level. Both of these kinds of tests have similar problems as they are testing though on different scales.

Designer testing however has its own particular problem. Since the person doing the testing is the designer and is not by training (and usually not by aptitude) a tester, designer testing tends to be somewhat patchy and ad hoc. Designer testing is certainly not systemic as it is performed before full integration.

Programming errors tend to cluster within particular types of errors depending on the experience and habits of the programmer who wrote the source code. For instance, a programmer may use a language construct or function/data interface with which they aren't familiar several times in the program introducing the same error in each place. A programmer may also use a language construct or semantic pattern inappropriately due to the experience and habits of the programmer. Source code inspections tend to find these types of errors as, source code inspections being a group activity, participants with different perspectives are examining the same object with the goal of finding problems. Static analysis tools using rule-based logic tend to uncover simple variations of these kinds of defects as well.

One of the benefits of defects found during source code inspections is that the defect is identified at the time it is found. In addition, the overhead involved in the PRS cycle shown above is eliminated when defects are found during a source code inspection.

Source code inspections have less effectiveness on those classes of errors that depend on the inspectors being able to model the process executing through the source code being inspected. This is primarily because people aren't computers and find it difficult to pretend to be one. Errors in program flow is one class of error that testing tends to be more effective at uncovering. However the author's experience as well as other anecdotal evidence (see [McConnell 93] recommendations for designer test procedures) indicates the use of source code level symbolic debuggers to step through routines can provide insight into how the source will run and will uncover existing errors of this type using a test harness for the source code being inspected.

Test case design and execution is every bit as complex as program design, a point that is sometimes forgotten. Testing as the prime defect detection mechanism has the same types of dependency on individual differences as program design and coding which can result in:

- defective test cases that produce defect reports for problems that don't exist,
- defective test case suites that miss functionality that should be tested,
- incorrect execution of test cases that miss functionality or produce incorrect defect reports,
- defect reports which are incorrect or incomplete thereby misdirecting the debugging programmer,

In addition to individual differences, test case execution can also provide less than optimum results due to:

- test cases that fail but then work correctly due to program complexity and state information especially in real time, multi-tasking types of software,
- test cases that pass but then fail in the field due to program complexity, state information, or operating environment especially in real time, multi-tasking types of software
- the inability to artificially replicate faults which can impact the total source code coverage of the test case suite causing fault handling routines to be untested

When a programmer is researching a reported defect, the source code must be examined as a part of the debugging process. [McConnell 93] provides a table of programmer ability to find and fix defects based on test results. The experiments from [Gould 75] indicates a large range of individual differences in the ability to determine the cause of an error and correctly fix the defect in a small program with 12 errors. Similar differences are reported by [Gilb 77] and [Curtis 81].

Table 2: Variation in debug ability from [McConnell 93]

	Fastest Three Programmers	**Slowest Three Programmers**
Average debug time (minutes)	5.0	14.1
Average number of errors not found	0.7	1.7
Average number of errors made correcting errors	3.0	7.7

The major variable affecting the rate of verification test case completion is the efficiency of the verification staff in executing test cases as rapidly and cleanly as possible. Some contributing variables to verification efficiency are:

- defect density in the product being tested (unimplemented yet required functionality is a defect) as each defect found provides an exception to test cases execution reducing efficiency;
- impact of found defects in the product being tested on verification progress (a program that aborts during startup due to a defect impacts the verification progress much more than a misspelled word in a user prompt);
- fix rate of defects along with the rate of introducing new defects as a part of the fixes;
- defect density in the test cases being executed;
- design of the test environment and test tools;
- experience of verification staff with the test environment and test tools; and
- experience of verification staff with the target domain of the product being tested.

The PRS solution rate used in estimation at ATP is 2 PRSs per week per designer. Obviously, the time required for a particular PRS will depend on the difficulty of finding and fixing the problem. The time required may be anywhere from a couple of hours for a simple defect to be traced to its cause, the fix put in and tested, and the corrected software submitted to the source code repository to a couple of weeks for a subtle problem.

Designers usually clump related PRSs together so as to resolve several PRSs with the same set of updates which can increase the solution rate. This clumping behavior is usually enhanced by the Paretto distribution of defects (80% of the defects in 20% of the source code) as well as the tendency to reuse software modules which means that a defective module with a large number of users may originate behavior which appears at the system level to be traceable to different causes. Naturally, these various behaviors will generate several PRSs which are resolved at one time.

PRS resolution by a designer usually involves the following steps:

- setup the test environment;
- duplicate the reported problem;
- debug the problem to its root cause;
- correct the identified defect if it is due to a straightforward cause or design and implement a fix for more complex causes;
- retest the fix to determine the reported problem is indeed corrected; and
- submit the corrected source code to the the source code repository for the next loadbuild

As can be seen from the above sequence, which doesn't include the verification effort of retest, the overhead of PRS resolution is not minimal. If verification is creating a large number of PRSs due to a high defect density, the development staff needed for PRS management and resolution can slow down development progress and impact other programs as development resources are redirected to the PRS reduction effort.

Model Predicting PRS and Schedule Extension Based on Test Case Pass Rate

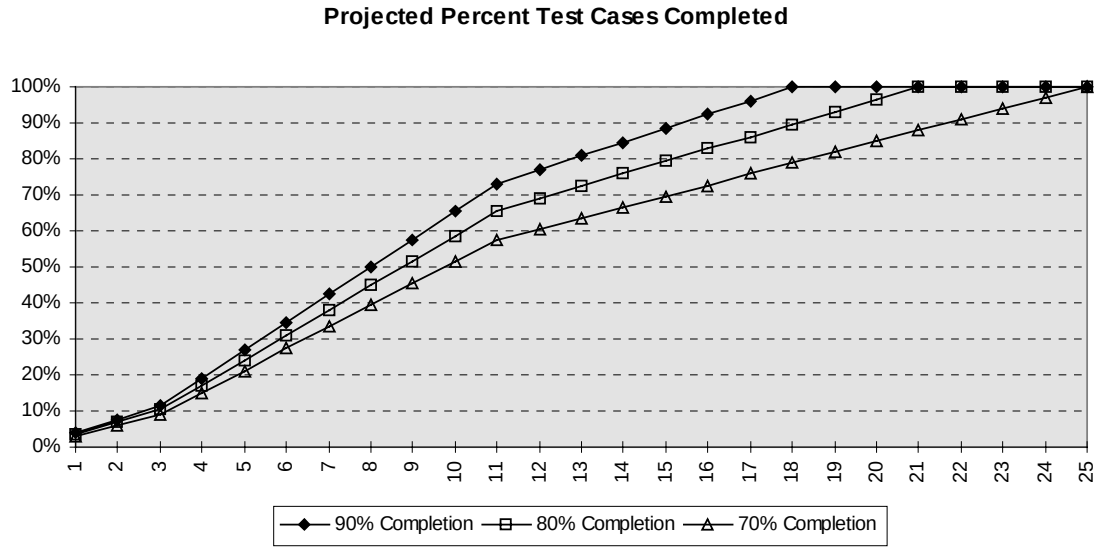
Finally, lets take a look at the effect of test case pass rate on project schedule and PRS rates. These graphs come from a simple model of test case completion and PRS generation from failed test cases.

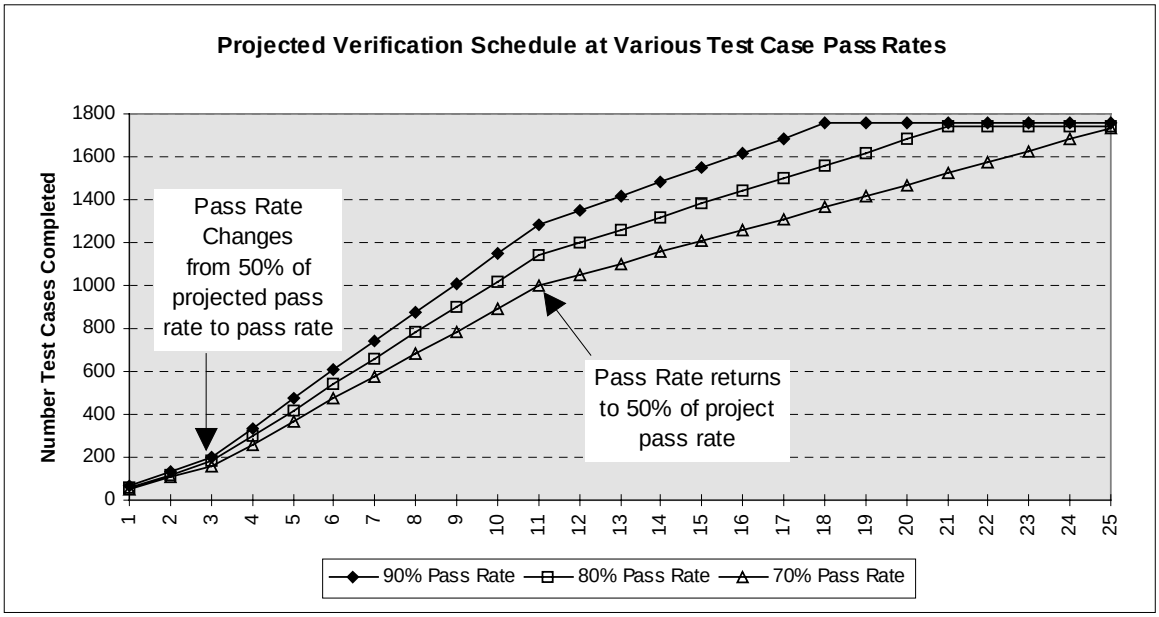
The following simplifying assumptions were made in this model:

- 1,770 test cases have been identified
- every failed test case will generate only a single PRS,
- Verification will be using three, 5 hour shifts per day, five days a week,
- the first three weeks of verification will result in a pass rate of 50% of the projected rate followed by a eight weeks at the projected rate with the remainder of the test cases passing at 50% of the projected rate providing a somewhat S-shaped curve

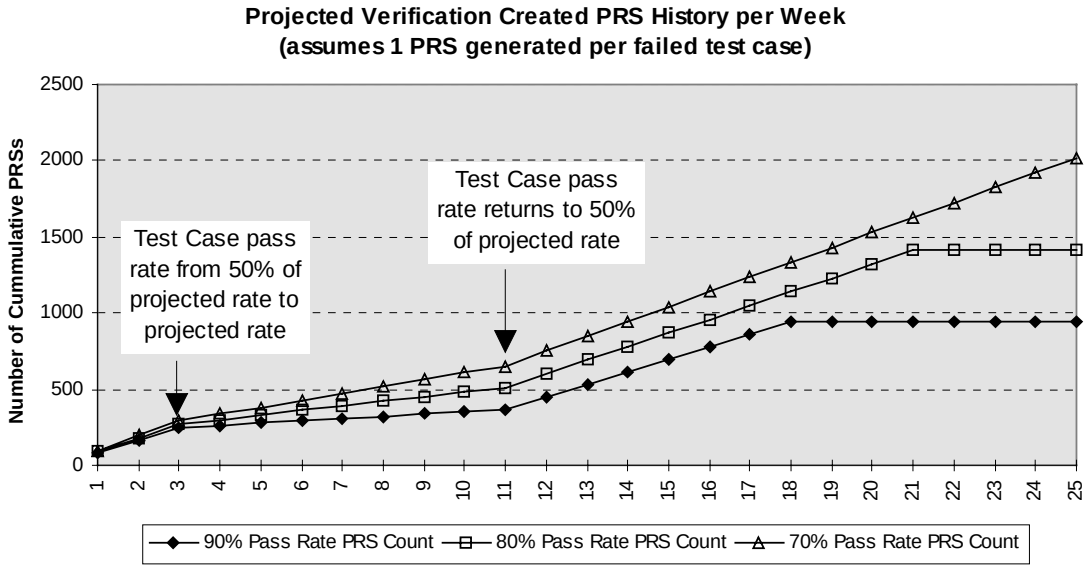
The model shows a large difference in schedules and PRS generation based on the test case completion rate. It indicates that waiting to start verification an additional few weeks with a higher quality load will make no difference in schedule if the pass rate can be increased from 80% to 90% of test cases run. As a bonus, the PRS overhead will be lower and more manageable.

The question that the software manager must ask is, how to predict defect density so as to determine if a higher pass rate can be expected. A good answer from [Glass 92] is the use of source code inspection data.





The chart below attempts to provide some idea of the magnitude of the PRS load, which correlates with the failure rate of the test cases, at different test case completion rates. While studying the chart the reader should remember that the PRS retest failure rate expected by Verification at ATP is around 30% which is not reflected in this chart.



A Brief Case Study Using AccessNode Release AN12

An Overview of AccessNode Release AN12 Project

The AN12 release of AccessNode contains the Virtual Tributary Bandwidth Management software and hardware. VTBM provides the ability to manage self-healing SONET(fiber optic) rings thereby increasing the telephone systems survivability in the face of accidents such as fiber cuts or equipment failures. The ability to manage traffic flow around the ring at the DS1/VT1.5 level rather than the more expensive, less granular DS3/STS-1 level is a definite competitive advantage for Nortel's customers.

The original concept was to piggyback on the development of the VTBM product for TransportNode using the VTBM functionality in AccessNode once development for TransportNode was complete. Due to schedule slips in the TransportNode development along with market demand, VTBM became a joint program with the AccessNode and TransportNode teams working together to bring the VTBM product to their respective markets.

TransportNode is a companion product designed for telephone traffic transport at the SONET level (OC-12 , OC-48, or OC-192) between switching centers where as AccessNode is designed to transport DS3/STS-1, DS1/VT1.5, and DS0 traffic between the subscriber and the switching center. Together, with the DMS switch, TransportNode and AccessNode provide the basic fabric for a telephone network.

The late availability of the fully functional release of the VTBM circuit pack was one of the major issues which impacted the project schedule. The lack of hardware was an especially crucial issue because the design community for AN12 and TN11 depended on the designer test subphase to ensure software had adequate quality (i.e. low defects) before moving into the Verification phase. Because inspection of source code wouldn't have required the availability of the hardware being developed, the question must be asked as to what extent the use of rigorous inspection of all software components would have reduced the impact of the late delivery of hardware.

An Overview of the AN12 Verification Effort

Approximately 1,770 test cases were developed for the verification phase of AN12. Based on the 5 test case per shift measure, running that many test cases would require 354 test shifts. Assuming 15 shifts per week (three, five hour shifts per day with a five day work week) the verification effort would

require 24 manweeks of effort. Using the standard Nortel standard cost of $59.50 per hour (see above for how this cost is derived) the verification effort would cost $105,315 in lab manpower alone assuming all tests pass the first time attempted. This figure is for test shift time only and doesn't include the additional verification overhead of test case development, test case documentation, writing shift reports, attending meetings, writing PRSs, negotiating lab resources, and retesting test cases.

Each additional test shift due to retest would cost an additional $297.50 in personnel costs.

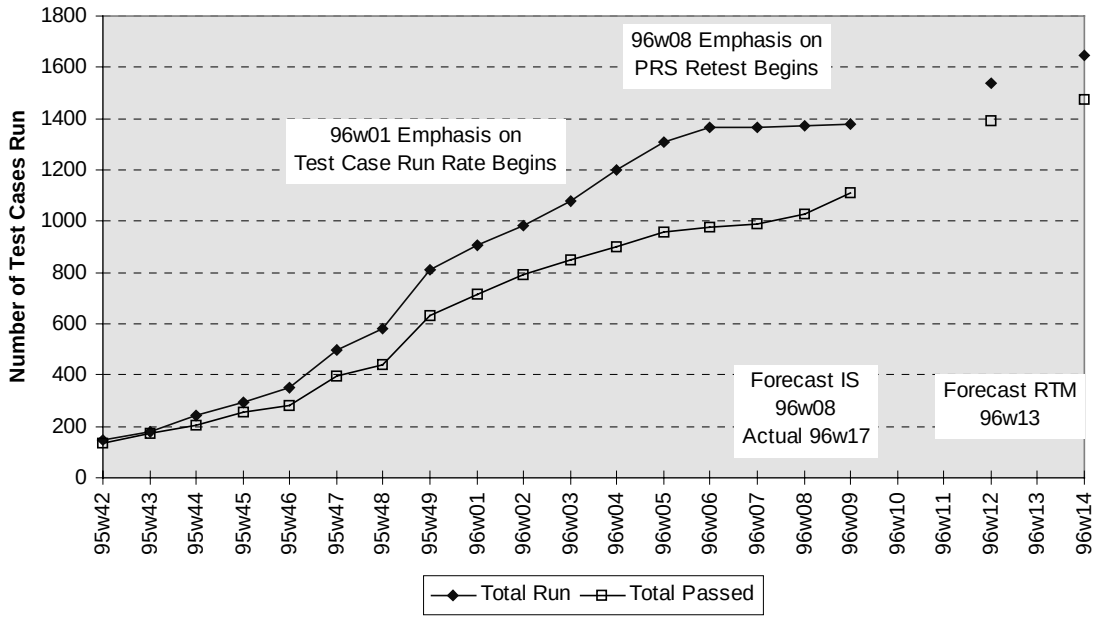
The chart below, compiled from the AN12 project records for Phase 1 of the VTBM project, shows a graph of the actual numbers of test cases run versus the number of test cases that completed. The records are incomplete as Verification started running test cases before 95w42 and Verification continued up to the actual IS milestone which was declared 96w17.

Due to the desire of management to keep the IS milestone at 96w08, a great deal of parallelism was introduced into the project. Verification began verification activities on those areas of the software that were considered sound enough to begin testing while other areas were in the coding and unit test stage of development.

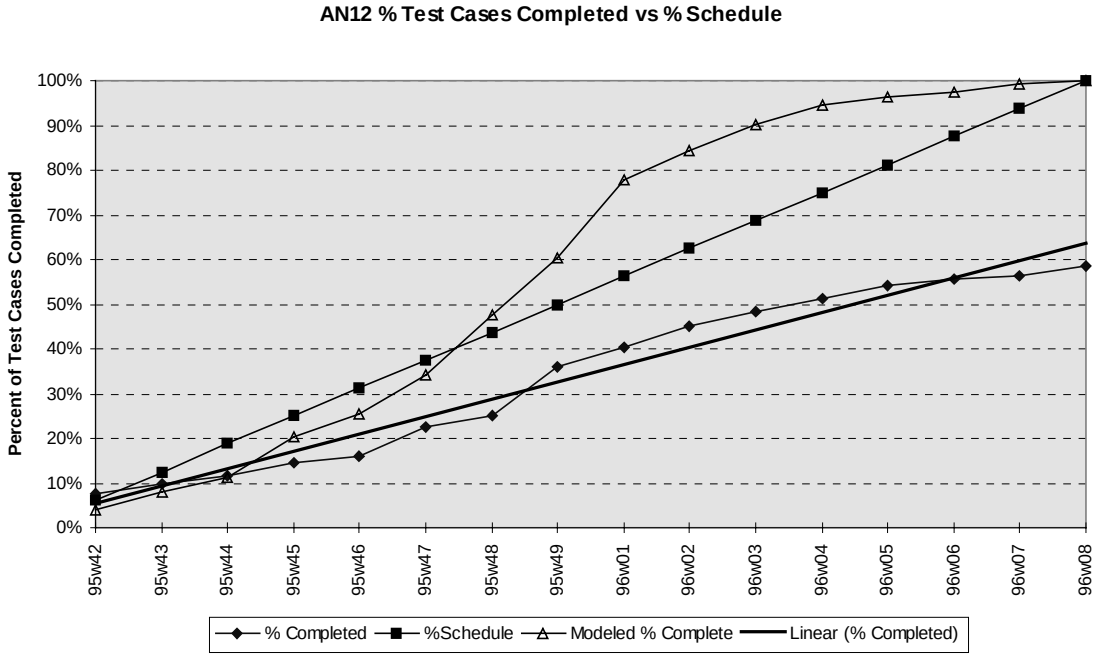
As the end of 1995 approached, Verification began to test functionality that wasn't fully designer tested causing the completion rate to drop as the test case run rate increased.

At this time, due to the large amount of churn in the source code, Design began having difficulty replicating some problems reported by Verification. PRSs that could not be reproduced were marked as such. Verification, which traditionally tries to avoid duplicate PRSs, began to have difficulties determining if problems had already been reported.

AN12 Verification Cumulative Test Case History



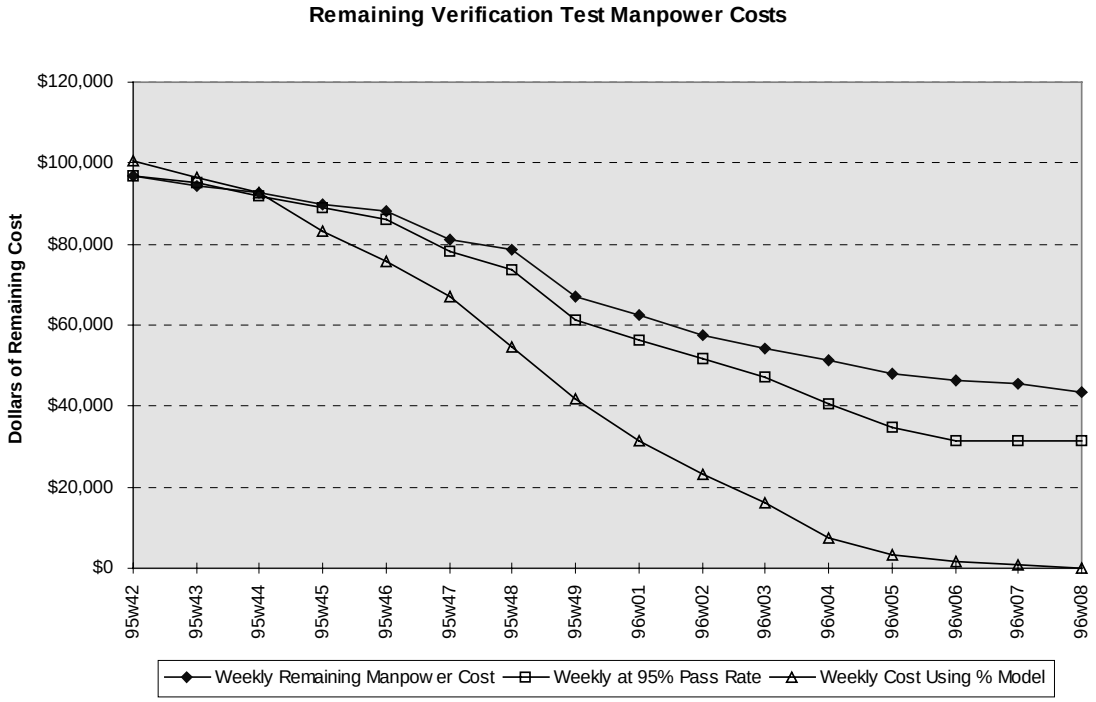
The following chart shows the percentage of test cases completed versus the percentage of the original AN12 schedule. The author has included a modeled percentage test cases completed which shows a possible curve based on the findings reported by Abe, et al from the Hitachi projects.



The following chart provides manpower cost curves for three scenarios, the actual Verification test case completion rate, a projected curve using the same weekly cumulative test cases run with a constant 95% pass rate rather than the actual pass rate, and a projected curve based on the Abel paper which assumes the schedule was held. The dollars are calculated based on the standard Nortel rate of $59.50 per hour.

The reader should be aware that the 95% pass rate is based on the actual number of test cases run. Verification didn't always run at capacity due to the low quality of the AN12 load. This is evident from the graphs which show a larger divergence between the Test Cases Run curve and the Test Cases Passed curve beginning around 96w01 as Verification increased the rate of test case execution.

Based on these curves, the large number of defects in AN12 were quite costly in terms of Verification variable costs (primarily manpower costs). Opportunity costs involving the release of lab facilities and personnel to other projects are not included in these cost curves.



Summation

The summation of the literature indicates that source code inspection by trained and experienced personnel embedded within an effective source code inspection process will improve time to market and reduce development costs.

However source code inspections must be part of a larger, integrated defect detection and removal process. Inspection of requirement and design documents removes potential defects from the final product before those defects can be implemented in the source code. Testing the final product provides a filter for defects that are easier to identify at that stage than at earlier stages of development.

This report is concrete in nature in that it provides data and information concerning defects introduced during design and coding on Verification and the customer. For further reading on managerial influences on quality and productivity, the author suggests the reader look to the following:

- Weinberg, Gerald M. *Quality Software Management*. Published by Dorset House Publishing. A four volume work (three volumes have presently been published; *Volume 1: Systems Thinking*, *Volume 2: First-Order Measurement*, and *Volume 3: Congruent Action*) on the management of software projects written by an excellent author. Weinberg is a software management consultant. During the last two decades, he has turned his attention from tools, techniques, and methodologies to project management and its affect on the success/failure of software projects.
- Hummel, Ralph P. *The Bureaucratic Experience: Second Edition*. 1982 by St. Martin's Press an illuminating study of bureaucracies that I have found most helpful in understanding large companies as well as governmental organizations.
- Kets de Vries, Manfred F. R. and Miller, Danny. *The Neurotic Organization: Diagnosing and Revitalizing Unhealthy Companies*. 1984 by Jossey-Bass Inc., Publishers (paperback by Harper Business) in which the authors write of their experience as business consultants with unhealthy organizations. The authors provide five major classifications of neurotic organizations with case studies of each of the five types. The one major point I came away with is the impact of a neurotic manager on the people within his organization.

- Schaef, Anne W. and Fassel, Diane. *The Addictive Organization*. 1988. Published by Harper & Row Publishers is a guide towards identifying dysfunctional organizational behavior. It also provides case studies of various patterns of such behavior along with how the behavior was addressed. *The Addictive Organization* goes beyond *The Neurotic Organization* in that the focus is on dysfunctional behavior and why substantial change within such organizations is impossible until the root cause of the dysfunction is determined and eliminated.

Finally, the author believes this quote from [Bucciarelli 94] concerning design activity should be kept in mind while the fairly constant debate concerning the design of the software development process continues at ATP.

> Designing is not simply a matter of trade-offs, of instrumental, rational weighing of interests against each other, a process of measuring alternatives and options against some given performance conditions. Nothing is sacred, not even performance specifications, for these, too, are negotiated, changed, or even thrown out altogether, while those that matter are embellished and made rigid with time as design proceeds. They themselves are artifacts of design. So, too, with other constraints; event codes have to be given a reading and an interpretation. They are all there to be negotiated if those readings run in conflict. Specifications become artifacts of process, reconstrued in the engaging of different perspectives of different object worlds.

Bibliography

[Basili 87] Basili, Victor and Selby, Richard. 1987. "Comparing the Effectiveness of Software Testing Strategies" first published in *IEEE Transactions on Software Engineering* Dec 1987 as reported by [Glass 92].

[Beizer 84] Beizer, Boris. 1984. *Software System Testing and Quality Assurance*. New York: Van Nostrand Reinhold Company.

[Boehm 81] Boehm, Barry W. 1981. *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice-Hall, Inc.

[Boehm 84] Boehm, Barry W., et al. 1984. "A Software Development Environment for Improving Productivity" as reprinted in [DeMarco 90].

[Brooks 87] Brooks, Frederick P. Jr. 1986. "No Silver Bullet: Essence and Accidents of Software Engineering" as reprinted in [DeMarco 90].

[Bucciarelli 94] Bucciarelli, Louis L. 1994. *Designing Engineers*. Cambridge, Massachusetts: The MIT Press

[Bush 89] Bush, Marilyn, and John Kelly. 1989. "The Jet Propulsion Laboratory's Experience with Formal Inspections." *Proceeding of the Fourteenth Annual Software Engineering Workshop, November 29*, 1989. Greenbelt, MD.: Goddard Space Flight Center. Document SEL-89-007

[Collofello 89] Collofello, Jim and Woodfield, Scott. 1989. "Evaluating the Effectiveness of Reliability Assurance Techniques" first published in the *Journal of Systems and Software* March 1989 as reported by [Glass 92].

[Curtis 81] Curtis, Bill. 1981. "Substantiating Programmer Variability" *Proceedings of the IEEE* 69, No. 7 as reported in [McConnell 93].

[DeMarco 90] DeMarco, Tom, and Lister, Timothy. 1990. *Software State-ofthe-Art: Selected Papers*. New York: Dorset House Publishing

[Gilb 77] Gilb, Tom. 1977. *Software Metrics*. Cambridge, Mass.: Winthrop as reported in [McConnell 93].

[Glass 92] Glass, Robert L. 1992. *Building Quality Software*. Englewood Cliffs NJ: Prentice-Hall Inc.

[Gould 75] Gould, John D. 1975. "Some Psychological Evidence on How People Debug Computer Programs" *International Journal of Man-Machine Studies* 7 as reported in [McConnell 93].

[Guindon 87] Guindon, Raymonde et al. 1987. "Breakdowns and Processes during the Early Activites of Software Design by Professionals" as reprinted in [DeMarco 90].

[Jones 86] Jones, Capers. 1986. *Programming Productivity*. New York: McGraw-Hill Book Company

[McConnell 93] McConnell, Steve. 1993. *Code Complete: A Practical Handbook of Software Construction*. Redmond, WA: Microsoft Press

[Meyers 78] "A Controlled Experiment in Program Testing and Code Walkthroughs/Inspections" first published in *Communications of the Association for Computing Machinery* Sept. 1978 as reported by [Glass 92].

[Parnas 86] Parnas, David Lorge and Clements, Paul C. 1986. "A Rational Design Process: How and Why to Fake It" as reprinted in [DeMarco 90].

[Selby 87] Selby, Richard W., et al. 1987. "Cleanroom Software Development: An Empirical Evaluation" as reprinted in [DeMarco 90].

[Weinberg 82]Weinberg, Gerald M. 1982. "Overstructured Management of Software Engineering" as reprinted in [DeMarco 90].

A Brief Examination of the Impact of Late Detection of Defects on Software Projects and the Costs Associated with Source Code Inspections

Richard Chambers, Northern Telecom Inc.

January 13, 1997