AFIT/GSS/LAS/97D-1

AN ANALYSIS OF EARLY SOFTWARE RELIABILITY IMPROVEMENT TECHNIQUES

THESIS

John G. Leonard, Captain, USAF
Ric K. Nordgren, 1Lt, USAF

AFIT/GSS/LAS/97D-1

Approved for public release; distribution unlimited

The views expressed in this thesis are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

AFIT/GSS/LAS/97D-1

AN ANALYSIS OF EARLY SOFTWARE RELIABILITY IMPROVEMENT TECHNIQUES

THESIS

Presented to the Faculty of the Graduate School of Logistics and Acquisition Management of the Air Force Institute of Technology

Air University

Air Education and Training Command

in Partial Fulfillment of the Requirements for the

Degree of Master of Science in Software Systems Management

John G. Leonard, B.S.
Captain, USAF

Ric K. Nordgren, B.S.
First Lieutenant, USAF

December 1996

Acknowledgments

I would like to dedicate this thesis to the women in my life, my wife Penny and my daughters; Stephanie, Danielle, and Gabrielle. For without their love and understanding this effort could not have been accomplished. Each of you have inspired me at different times and in different ways. Penny's constant support and motivation; Stephanie's prayers for me, Danielle's good morning hugs; and Gabrielle's smiles have all helped me to persevere. I would also like to thank Professor Dan Ferens, Major Jim Skinner and Major Terry Adler for their leadership and tenacity. They both genuinely cared about our thesis and our education.

Capt John G. Leonard

I would like to dedicate this thesis to my loving wife, Robin, for all her support and confidence. I would also like to dedicate this thesis to my son, Tycho, and to thank him for waiting until I was nearly complete to be born. You have both provided motivation to produce the best thesis possible. I would like to thank Prof. Daniel Ferens for his ability to point out the style guide errors on our rough drafts, because it is the obvious that sometimes slips by. I would also like to thank Majors James Skinner and Terry Adler for all their advice and guidance in creating a worthwhile document that has real-world application to the Air Force, and more importantly, software development in general.

1Lt Ric K. Nordgren

Table of Contents

List of Tables

Abstract

The research explores current early life cycle software reliability prediction models or techniques that can predict the reliability of software prior to writing code, and a method for increasing or improving the reliability of software products early in the development life cycle. Five prediction models and two development techniques are examined. Each model is statically analyzed in terms of availability of data early in the life cycle, ease of data collection, and whether the data is currently collected. One model and the two techniques satisfied requirements and were further analyzed for their ability to predict or improve software reliability. While the researchers offer no significant statistical results of the ability of the model to predict software reliability, important conclusions are drawn about the cost and time savings of using inspections as a means of improving the reliability of software. The results indicate that the current software development paradigm needs to be changed to use the Cleanroom Software Development Process for future software development. The major conclusion of this research is that a proactive approach to developing reliable software saves development and testing costs. One obvious benefit of this research is that cost savings realized earlier in the software development cycle have a dramatic effect on making software development practices better and more efficient.

AN ANALYSIS OF EARLY SOFTWARE RELIABILITY IMPROVEMENT TECHNIQUES

1. Executive Summary

1.1 Introduction

The current software development process is reactive and expensive. It involves the iterative process of coding, testing, (to determine the number of faults in the software), and rework of the code. During rework, the faults are removed and then the software is re-tested to determine if the faults were actually removed and to see if new faults were introduced.

1.2 Research Question

Software reliability models determine the amount of testing required to say with confidence that the software is fault-free. These models are used late in the development life cycle when the costs to make those corrections are 100 times more expensive (Fagan, 1976:37). Therefore, what is needed are software reliability models that can be used early in the software development life cycle to determine the reliability of the software. This need led to the question that the Air Force Operational Test and Evaluation Center (AFOTEC) wanted answered:

"What are the current early life cycle software reliability prediction models and which should be recommended for AFOTEC to use in support of operational assessment?"

1.3 Findings

Five early life cycle software reliability prediction models were identified and evaluated along with two early life cycle software reliability improvement techniques. Each model and technique needed to satisfy the "early" life cycle definition of "at or prior to Software Specification Review (SSR)." Each also required that data be available, easy to collect, and collected by AFOTEC. The result was that only one model appeared capable of being used by AFOTEC early in the software development life cycle and could have data easily collected. Of the two techniques, neither of them can be mandated by AFOTEC for software developers to use, but developers could be given greater consideration during source selection if AFOTEC shows they are advantageous.

1.4 Revised Research Question

The five models and their reactive predictive nature convince the researchers that merely predicting the reliability of software early in the development life cycle is a less effective allocation of management resources. The researchers concluded that the question AFOTEC should really want answered is:

"What are the means of improving the software development process, which in turn, will improve software reliability through fault avoidance?"

1.5 Revised Findings

The two techniques identified by the researchers are proven methods of improving software quality and hence, reliability. AFOTEC's mission is to test and

evaluate Air Force systems. AFOTEC can reduce their software testing costs if the products they test are known to have fewer faults in them. Inspections and the Cleanroom Software Development Process are two proactive approaches of improving software reliability through fault elimination and avoidance. These approaches are proactive in that by performing them, the reliability of the software will consequently improve because the processes do not allow for faults to remain in, or propagate through, the development. None of the models studied can improve the reliability of the software developed, but instead can only identify software that is likely to contain faults, or have a high fault density.

1.6 Conclusion

Of the five models, only the Rome Lab Model appeared applicable to the scope of this thesis, and its usefulness is dependent upon what the manager does with its results. Inspections, by themselves, cannot improve software reliability. They are only effective when feedback about faults is given to the developers early enough to implement changes in the process. The Cleanroom process is the only method identified capable, in itself, to improve the reliability of software.

1.7 Recommendation

The researchers recommend the Cleanroom Software Development Process as the most effective means of improving software reliability based on the criteria that changes can be made early in the software development cycle, and the cost savings associated with early detection and removal of faults are the greatest. The Cleanroom

process could be a radical change for organizations that are not developing software in a formal process, and therefore implementing a Cleanroom process is economically and physically impractical. The researchers then recommend inspections, coupled with feedback to the developers, as the next best, cost effective, alternative for ensuring software reliability. If neither of these methods are feasible, then the Rome Lab Model 1 is the recommended model for predicting a fault density. Utilizing the Rome Lab Model 1 to identify areas that are possibly linked to increases in fault density is one means of improving software reliability with the model.

2. Literature Review

2.1 Software

Software is an increasingly important factor in the safety of systems being developed and delivered to the Air Force. In many cases, software components are replacing existing hardware components. The introduction of software into safety-critical systems introduces new modes of failure for the systems which cannot be analyzed by traditional engineering techniques. This is because software fails differently from hardware and more importantly, software failure is less predictable than hardware failure (AFOTEC SREG, 1992:14).

In addition to software failing differently than hardware, software is developed in a structured sequential process, called the software development life cycle. The software life cycle consists of the five primary phases: requirements analysis, design, implementation, test, and maintenance. Requirements analysis is the phase where the problem is analyzed and defined; the result of this phase is a formal specification of a system that will meet customer needs. In design, the software specification is transformed into a conceptual solution. Decisions are made about issues such as programming language, platform the software will operate on, type of data structures required, and other performance-related issues. The output of the design phase is a design representation in the form of text, flowcharts, decision tables, or program design language. During implementation, the conceptual design representation is transformed into an actual program language through coding or reuse. Testing ensures

the software performs according to the specification, identifies faults, and collects failure data. An analysis of the data is performed to determine reliability predictions of the time of the next failure. This phase also addresses the correction of those faults and the testing of those changes. Finally, the maintenance phase is the process of responding to user needs by fixing errors, making user-specified modifications, and honing the program for usability.

2.2 Reliability

Reliability is one measure of quality. "Software quality refers to the degree to which the attributes of software enable it to perform its specified end-item use" (DoD-STD-2168). The end-item use of software is the relationship of the software to the accomplishment of the system's mission task or mission task element (GSAMSIS, 1996:O-22). Reliability refers to the degree to which a system satisfies its requirements and delivers usable services (i.e., those services the system is designed to perform). Predicting hardware reliability has almost developed into a science; mechanical and electronic components have known failure rates that have been tested and verified over and over. Predicting software reliability, on the other hand, is not nearly as precise as hardware reliability. An explanation of the differences between hardware and software is often over stated, yet it is a simple concept.

Hardware is the physical aspect of a system. Hardware reliability is the probability a system will perform satisfactorily, for a specified amount of time, in its normal operating conditions. When considering hardware reliability predictions,

components are subjected to extensive life testing under realistic conditions. As components fail, the nature of the failure is thoroughly examined, and if possible, the manufacturing or assembly process is modified to prevent or reduce that failure occurrence. These hardware components are sampled and tested regularly to determine the mean-time-between-failure (MTBF) or reliability of the system. The foundation of hardware reliability prediction assumes independence of failure and that the reliability measuring process does not affect the failure rate (Beizer, 1984).

Software, the tractable medium (Brooks, 1995:7) of the system, provides the hardware's operating instructions. Software reliability is defined as the probability of failure-free software operation for a specified period of time in a specified environment (Lyu, 1996:5). Unlike hardware, software is not easily tested to a MTBF standard. This inability to test for MTBF on software is attributed to the nature and size of the code (Beizer, 1984). The nature of the code is its tractability and the size is growing larger as the users demand more capabilities and performance.

The characteristics of software reliability as compared to hardware reliability is environmentally dependent. The sources of failure in software are design faults, misunderstood requirements, coding errors, and faulty testing, while the principle source in hardware has generally been physical deterioration (Musa, 1990:7).

Lyu (1996:18) supports the similarities of hardware and software reliability

> ...in that both are stochastic processes and can be described by probability distributions. Software can be evaluated via usage distributions to predict the most common failure and hardware reliability is based on historical failure data. However, software reliability is different from hardware reliability in the sense that

software does not wear out, burn out, or deteriorate (i.e., its reliability does not decrease with time). Moreover, software generally enjoys reliability growth during testing and operation since software faults can be detected and removed when software failures occur. On the other hand, software may experience reliability decrease due to abrupt changes of its operational usage or incorrect modifications to the software.

Software is also continuously modified throughout its life cycle. The malleability of software makes it inevitable for us to consider variable failure rates. As a result, software reliability is a much more difficult measure to obtain and analyze than hardware reliability. Usually, hardware reliability theory relies on the analysis of stationary processes, because only physical faults are considered. However, with the increase of systems' complexity and the introduction of design faults in software, reliability theory based on stationary process becomes unsuitable to address non-stationary phenomena, such as reliability growth or decrease.

2.3 Software Reliability

The term "software failure" can be interpreted to mean many different things. Musa *et al.* (1990:7) defines software failure as "the departure of the external results of program operation from the requirements." They further explain failures as dynamic; the program has to be executing for a failure to occur. They also make a distinction between failures and faults, suggesting that a failure is not the same thing as a bug or fault. Musa *et al.* (1990:7) define a fault as the defect in the program that, when executed under specific conditions, causes a failure. In other words, a fault is a manifestation of a code generation error or errors in logic, computer instructions or data. They are introduced during the coding and maintenance phases of the life cycle and exist in code whether the code is exercised or not. Faults exist in the program's static operating state and do not become a failure until the code that contains the fault is executed.

Mathematical forms can be used to model the failure process. A specific model can be determined from the general mathematical form by establishing the values of the parameters of the model through either estimation or prediction. In estimation, the statistical inference procedures are applied to failure data taken from the program (i.e., historical data). Prediction, on the other hand, is a determination of the parameters derived from the properties of the software product and the development process (i.e., inherent properties). Accordingly, the values of the parameters of the specific model can be determined either through estimation (i.e., after execution) or prediction (i.e., prior to any execution of the program) (Musa *et al.*,1990).

Ideally, software reliability should represent a user-oriented view of software quality. Early, and many present, approaches to measuring software reliability were developer-oriented and focused on counting the faults or defects found in a program (Musa *et al.*, 1990:5). "Also, what was usually counted were either failures or repairs, neither of which are equivalent to faults. Even if faults found are correctly counted, they are not a good status indicator" (Musa *et al.*, 1990:5). What would be a good status indicator of reliability is faults remaining (Musa *et al.*, 1990:5). However, it should be understood, the only means to obtain an accurate measure of faults remaining is through exhaustive testing. Since exhaustive testing is already considered unrealistic, faults found can be used to make rough comparisons with other software development projects. Such comparisons are commonly made in terms of faults per 1000 developed source lines of code (Musa *et al.*, 1990:5, Dyer, 1992:7).

Software reliability, a much richer measure than fault measures, is user-oriented rather than developer-oriented. It relates to operation rather than design of the program; hence, it is dynamic rather than static, and takes into account the frequency with which problems occur. Software reliability relates directly to the operational experience and the influence of faults on that experience; thus, it is easily associated with costs (Musa *et al.,* 1990:5). It is more suitable for examining the significance of trends, for setting objectives, and for predicting when those objectives will be met. It permits analysis, in common terms, of the effect on system quality of both software and hardware, both of which are present in any real system. Thus, reliability measures are much more useful than fault measures (Musa *et al.*,1990:6).

This does not mean that researchers should completely disregard faults; analysis should focus on faults as predictors of reliability and on the nature of those faults. "A better understanding of faults and the causative human error processes should lead to strategies to avoid, detect and remove faults, or compensate for them" (Musa *et al.*,1990:6). These strategies should be employed early in the life cycle to improve the overall design, process and the efficiency of the program, as well as reduce the software life cycle cost.

"Experience with the application of software reliability measurement and prediction on projects indicates a cost of implementation of around 0.1 to 0.2 percent of project development costs" (Musa *et al.,* 1990:6). This percentage may seem small, but when the overall cost of a single weapon system acquisition program is considered, the software measurement and prediction costs could easily reach a sizable sum.

Consider the USAF's F-22 program, reportedly an $18 billion development effort; the measurement and prediction of the software reliability could exceed $36 million (Gordon, 1996). Thus, developing reliable software is an important topic because of the high costs involved with measurement and prediction. In addition, if the software does not pass reliability acceptance standards, then the cost to rework and reevaluate the software increases the software development costs.

2.4 Early Life Cycle Activities That Affect Reliability

The reliability of software can be improved by reducing the number of faults introduced through human-error, increasing the rate of discovery of these faults, or both. This can be accomplished through four activities linked with the software life cycle: fault avoidance, fault elimination, fault tolerance, and structured maintenance (Lyu, 1996:19).

Fault Avoidance

Fault avoidance consists of applying sound software engineering practices, including comprehensive standards such as documentation, design, and programming; rigorous quality assurance techniques like formal reviews, inspections, and audits; and independent verification and validation (Lyu, 1996:20). Inspections, reviews, audits and independent verification and validation can each be applied to any well-defined work product such as requirements and design documents, test plans, hardware logic, and code (Russell, 1991:26). Another engineering technique, the Cleanroom Software

Development Process, also known simply as Cleanroom, also provides a mathematical correctness verification approach to software fault avoidance.

Detection of faults in the transformation of requirements to specification is an early step in fault avoidance. By inspecting random samples of the formal specification as it is being written, ambiguities and misunderstandings in the transformation process can be identified. These ambiguities are brought to the attention of the developer and the defects are pointed out. According to Gilb, (1996:26) 62% of defects occur during the design process, and 38% are created during coding. Inspections and Cleanroom processes are proactive approaches to ensuring that defects are not allowed to reside in a software program and that they are removed prior to coding or testing. The use of inspections and Cleanroom processes result in a software product that is less prone to defects, thus increasing software quality and reliability.

Fault Elimination

Fault elimination is accomplished through design and code inspections, mathematical correctness verification, and effective testing. While Lyu (1996:20) claims, "it is possible through exhaustive testing or mathematical-proof-of-correctness to remove all faults in the code, it is usually impractical to do so in systems of significance size and complexity." Inspections are a more practical approach than exhaustive testing to eliminating software faults and they are conducted earlier in the development life cycle when the cost savings are the greatest. Test coverage models are available to assess the effectiveness of the test strategies employed by the software

developer later in the life cycle. Common testing methods will identify many, but certainly not all, faults (Lyu, 1996:20).

Fault Tolerance

Fault tolerance is achieved through special programming techniques that enable the software to detect and adequately recover from error conditions. One method of programming fault tolerant software is the development of redundant software elements that provide alternative means of fulfilling the same function. The different versions must be programmed such that they will not both fail in response to the same input state. A more common, but less effective, example of fault tolerance is the use of exception handling in Ada (Lyu, 1996:21).

Structured Maintenance

Each software maintenance action should be performed as a microcosm of the full development life cycle. As such, the techniques of fault avoidance, elimination, and tolerance can be applied to modifications made during the maintenance of software as well. This is necessary to avoid introducing new faults as a result of code modifications made to correct known faults, add enhancements, or adapt the software to changes in the computing environment (Lyu, 1996:21).

2.5 Methods of Evaluating Software Reliability

2.5.1 Software Reliability Estimation

(Lyu, 1996:19) identifies software reliability estimation as the determination of

> ...current software reliability by applying statistical inference techniques to failure data obtained during system test or during system operation. This is a measure regarding the achieved reliability from the past until the current point. Its main purpose is to assess the current reliability and determine whether a reliability model is a good fit in retrospect. When failure data is available (e.g., software is in system test or operation stage), the estimation techniques can be used to verify software reliability models.

Although estimation can provide a level of knowledge required for management, it does nothing to improve the quality of the software. Estimation takes place late in the life cycle, after system testing has begun and thus, the faults have already been designed and coded into the software.

2.5.2 Software Reliability Prediction

Software reliability prediction attempts to determine future software reliability based upon available software metrics and measures. The process can begin as early as Concept Exploration (Milestone 0) with a prediction based upon the historical performance of similar applications. This prediction is refined and converted to a failure rate as additional program information becomes available. "The main purpose in computing a reliability prediction is to identify software development or maintenance processes or product characteristics which could be changed to improve the operational reliability of the software product" (AFOTEC SREG, 1992:17).

"When failure data is not available (e.g., software is in the design or coding stage), the metrics obtained from the software development process and the characteristics of the resulting product, to that point, can be used to determine the early prediction of the software upon testing" or delivery (Lyu, 1996:17).

2.6 Existing Early Software Reliability Prediction Techniques

Software reliability prediction translates software measurements taken during early life cycle phases into a predicted reliability. Five approaches that attempt to predict software reliability based on static analysis of product and process metrics were identified: 1) Rome Lab Model 1, 2) Munson's complexity measures research, 3) Smidts et. al's, Software process failure mode model, 4) Agresti and Evanco's Projecting Software Defects from Analyzing Ada Designs, and 5) Nikora's software life cycle research (Error Introduction and Removal Method). In addition, two other approaches to improve software reliability were examined; Fagan-style inspections and the Cleanroom process. A summary of each of these methods follows.

2.6.1 Rome Lab Model 1

The Rome Lab Model 1 provides a prediction of software reliability for each block in the system/hardware block diagram. When using this model, the predicted software reliability figure of merit is a fault density. For each software component or component grouping on the block diagram, the predictive Reliability Prediction Figure of Merit (RPFOM) is calculated according to the formula:

$$RPFOM = A * D * S \qquad (2\text{-}1)$$

where RPFOM is the predicted fault density, A the application type metric, D the software development metric, and S the software characteristic metric (RL-TR-92-52, 1992:53).

The application type metric (A), is expressed as a baseline fault density. The Application-type RPFOM is calculated for each test project. This baseline RPFOM, determined prior to initiation of software development, is an average fault density based on the principle application type of the test project. The application types of the test project include airborne, strategic, tactical, process control, production center, or developmental. Each of these are assigned a numerical value from the look up table found in (RL-TR-92-52, 1992:58) and are used with the Worksheet 0.

The Development Environment RPFOM, which is a refinement of the baseline RPFOM, incorporates information, summarized in the (D) metric, which should be available during a software pre-development phase of the life cycle. There are three development environment categories: embedded, semi-detached, and organic. The embedded mode is more prone to introducing faults because the personnel operate within tight constraints. The software team has computer expertise, but may be unfamiliar with the application served by the program. The semi-detached mode corresponds with a software team that is experienced in the application but not affiliated with the user, and does not affect the introduction of faults. The organic mode is for software teams that are part of the organization that is served by the program and assumes the least likelihood of introducing faults. These also are provided in a look up table in (RL-TR-92-52, 1992:58) and are used with the Worksheets 1 or 1A.

The Software Characteristics metric (S) has two separate metrics, (S1) and (S2), that can be used depending upon where in the software development life cycle the

prediction is being made. The Requirements and Design Representation Metric, (S1) comprises Anomaly Management (SA), Traceability (ST), Quality Review Results (SQ) and Discrepancies (DR). These metrics can be derived by Rome Lab Model Worksheets 2, 3, 10, and 5, respectively, and are a measure of the development processes and practices (RL-TR-92-52, 1992:59).

> The fault density, predicted by Rome Lab Model 1 is used as an early indicator of software reliability based on: 1) the number of problems being identified and an estimate of size are relatively easy to determine during the early phases of a development and 2) most historical data available for software systems support the calculation of a fault density, but not a failure rate. (RL-TR-92-52, 1992:61)

2.6.2 Relative Complexity Metric

The relative complexity metric "is the relationship of software faults and measurable software attributes, such as software complexity measures" (AFOTEC SREG, 1992: 19). The simplest level at which relative complexity can be measured is the module, and these relative complexities can be combined to indicate reliability at higher levels. Munson and Khoshgoftaar (1992) analyzed 16 complexity metrics grouping the 16 variables so that all within the group are highly correlated but have relatively small correlation with variables in other groups. From this analysis, they identified five independent factor domains of software complexity. These clusters each have an underlying factor that is responsible for the correlation, and that factor is the independent domain. The five independent domains are: 1) control, 2) size, 3) modularity, 4) effort or information content, and 5) data structure. Munson and

Khoshgoftaar (1992:50) mapped these domains into a single factor-domain called relative complexity.

Munson and Khoshgoftaar (1990) have shown that relative complexity is directly related to software faults. They assert that "relative complexity is a substitute for aspects of software that cannot be directly measured such as software faults. Relative complexity is related to software reliability in that the likelihood that a program will fail is directly related to its complexity" (AFOTEC SREG, 1992:19).

The relative complexity metric is the weighted sum of the five domains. The weights for each domain are determined by summing its eigenvalues, which represent the domain's variance. After establishing the factor domains' weighted sums, the relative complexity of the factored module is calculated. The relative complexity characterizes the complexity of each module as a single value. The relative complexity measure, ρ_i, of module i is:

$$\rho_i = \lambda_1 f_{1i} + \lambda_2 f_{2i} + \ldots + \lambda_m f_{mi} \qquad (2\text{-}2)$$

"where λ_j is the eigenvalue associated with the jth factor domain, f_{ji} is the factor score of the ith module on the jth factor" and m is the number of factors in the factor pattern (Munson and Khoshgoftaar, 1992:50).

The distribution represents observations from a normally distributed population with an expected value of zero, and a variance of

$$V(\rho) = \sum_{i=1}^{m} \lambda_i^2 \qquad (2\text{-}3)$$

Because the numbers range from negative to positive, Munson and Khoshgoftaar scaled the metric so that the mean is 50 and the standard deviation is 10. The scaled relative complexity, ρ_i' of module i is:

$$\rho_i' = 10\frac{\rho_i}{\sqrt{V(\rho)}} + 50 \qquad (2\text{-}4)$$

"The relative complexity metric classifies modules by complexity and provides a method for ordering them." (Munson and Khoshgoftaar, 1992:51) examined the metric's sensitivity for modules of sample programs from a larger population and found the metric insensitive to subtle variations in complexity values associated with the programs, indicating that it is robust.

Munson and Khoshgoftaar (1992:51) identify two reasons the relationship between complexity metrics and reliability and performance are significant:

> 1. These metrics can be collected early in the life cycle. Relative complexity can be determined from design documents, therefore, its earliest use is at the critical design review, prior to coding.
>
> 2. To the extent that a definite relationship between metrics and reliability and performance can be established, these metrics can be used as adjuncts to reliability and performance modeling.

2.6.3 Software Process Failure Mode Model

This model suggests an approach to the "prediction of software reliability based on systematic identification of software process failure modes and their likelihoods. A direct consequence of the approach and its supporting data collection effort is the identification of weak areas in the software development process" (Smidts *et al.*,

1996:132). This model permits an iterative approach to prediction by allowing updates throughout the different phases of the life cycle.

Failures for this model are interpreted two ways. First, from the developers point of view, failure to implement any requirement may lead to lawsuits and other adverse outcomes; hence, for such a company, software reliability is the probability that all requirements will be met (Smidts *et al.,* 1996:133). Secondly, "the user's definition is clearly focused on the mission to be achieved, and software reliability is the probability that a software achieves its mission" (Smidts *et al.,* 1996:133).

Smidts *et al.* (1996:134) model consists of the following four steps:

> 1) identification of software process failure modes; 2) development of a requirements-based fault tree that relates software failures to applicable software process failure modes; 3) Estimation of software process failure mode probabilities; and 4) Determination of software reliability through fault tree solution.

A brief discussion of each follows.

Identification of software process failure modes. Attempting to identify all of the software process failure modes requires close coordination between the user and the developer. "The failure modes are obtained by applying the software process origination failure modes to each requirement category." Then, "[o]nce the software process failure modes are identified, factors that influence their generation (the actual human error activity) should be determine." Finally, "[o]nce all failure modes and influencing factors have been established, data collection can start" (Smidts *et al.,* 1996:135).

Requirements-based fault tree development. The fault tree is built by determining how each of the requirements relate to each other and how they can be failed. "The next step is to find the combination of software process failure modes that will fail a software requirement. These failure modes are dependent on the process and specific parts of the software product (architecture)" (Smidts *et al.*, 1996:136). Next the probability that the software will fail due to process failures occurring in the current phase is determined (Smidts *et al.*, 1996:136). The actual fault tree will be quite complex since software process failure modes generated by all phases of the development process may contribute to the failures. The operations the software is used to achieve should be contained in the operational profile drawn for the software, which can then be used to derive the user reliability (Smidts *et al.*, 1996:136).

Estimation of failure mode probabilities and determination of software reliability through fault tree solution. This model uses Bayesian statistics to quantify software process failure modes. The requirements failure mode probability density function is bimodal, and the values of the two modes are correlated. Multiple statistical formulas for application based on the assumption of time-dependent requirement failure behavior can be found in Smidts *et al.* (1996:138). "As an initial step in estimating the probabilities of the requirements-based fault tree events, all possible failure modes of a single requirement occurring during one life cycle phase are grouped" (Smidts *et al.*, 1996:138). They map these groups of requirements failure intensities as a function of life cycle effort, suggesting a growth in reliability or a reduction in failure intensity with each phase from preliminary design to the beginning

of testing. Once into the unit test, the failure intensity begins to climb. This is as expected, since the purpose of testing is to find faults and failures. This model also suggests a use of historic software process failure mode data for creating a baseline for each phase of the life cycle and includes two Bayesian approaches to allow for the use of historic information with varying degrees of relevance.

2.6.4 Projecting Software Defects From Analyzing Ada Designs

Agresti and Evanco (1992:988) suggest a model to predict defect density based on the product and process characteristics. "Product characteristics are extracted from a static analysis of Ada subsystems, focusing on context coupling, visibility, and the import-export of declarations. Process characteristics provide for effects of reuse level and extent of changes" (Agresti and Evanco, 1992:988). Their research emphasizes the software quality factors of reliability, maintainability, and flexibility. These quality factors are associated with events (i.e., system failures) and activities (i.e., software correction and enhancements) that occur after implementation (Agresti and Evanco, 1992:988). Their research based the quality estimate on the analysis of the design structure. Agresti and Evanco differ from others by using "multivariate models that incorporate more than a single design characteristics in an attempt to explain software defect density. They also considered the way organizations develop software and how the organization's inherent processes (e.g., process characteristics) affect the quality of the end product."

The general form of the multivariate model to explain variation in a software quality factor is as follows:

$$Y = f(X_1, X_2,...X_n, Z_1, Z_2,..., Z_m \mid a_1, a_2,...,a_p) + e \qquad (2\text{-}5)$$

"where Y = software quality factor (e.g., reliability), X_i = the ith product characteristic variable ($i=1,...,n$), Z_i = the ith process characteristic variable ($i=1,...,m$), a_i = the ith parameter ($i=1,...,p$), and e = the disturbance term for the unexplained variation" (Agresti and Evanco, 1992:989).

The Agresti and Evanco (1992:989) approach to the design process model considered software as a set of design units and relations on the set. Relations included control coupling and data coupling. The dependent variable in this model is defect density, the number of defects per thousand source lines of code. "The model demonstrated that the source lines of code can be factored out of the explanatory variables by using defect density as the dependent variable" (Agresti and Evanco, 1992:991).

To define the explanatory variables Agresti and Evanco (1992:991) identified measures for product and process characteristics. "The measures were introduced as independent (explanatory) variables in the model of defect density" (Agresti and Evanco, 1992:991). These included four product characteristics (context coupling, visibility, import origin, and internal complexity) and two process characteristics (volatility and reuse) (Agresti and Evanco, 1992:991).

2.6.5 Error Introduction and Removal Model

Nikora's (1993) research, although not complete, is considered for examination. He proposes to build "a model for error discovery and removal that could be used to predict the reliability of software during early life cycle phases" (AFOTEC SREG, 1992:19). The model is loosely based on the Rome Lab Model 1, although it does not consider anomaly management, traceability, language or extent of reuse, because the data to support those factors are not readily available until late in the development life cycle. The desired characteristics of the model are: 1) that it relate product and process metrics to reliability, 2) that it use information available early in a software development effort, 3) that its predictions be easily updated, and 4) that its results be expressed in useful terms.

Nikora (1993) expects to prove that the error discovery rate and the error removal rate can be determined from various process and product metrics. Product and process metrics include the type of application being developed, the specification and design methods used, and the type of technical reviews used. Information available early in the software life cycle includes the development schedule, staffing profile, and a plan for reviews (Nikora, 1993:48, AFOTEC SREG, 1992:19).

Nikora identifies factors that influence the introduction of errors. Several of the factors are included from the Rome Lab Model 1 RPFOM calculation such as type of application and development environment. Other factors considered as important indicators of probability of error include program size, modularity, and complexity.

In addition to those factors identified in (RL-TR-92-52, 1992), Nikora (1993:46) added current development phase and average workload per developer.

> These two factors appear to be more human issues than software reliability issues. The current development phase factor attempts to capture the notion that developers work at a different level of detail during the requirements phase than they do during subsequent design and implementation phases and that error introduction rates may differ from phase to phase. The average workload per developer factor assumes that the rate of error introduction depends on the amount of time that developers have to accomplish their task. (AFOTEC SREG, 1992:20)

Nikora (1993) identified eight factors that influece error discovery. Those factors associated with error discovery were shown to be directly related with product inspection. "These include the amount of time spent in inspections, the preparation time for inspections, the fraction of the product inspected during each phase, the phases during which inspections occur, and the number of inspections" (AFOTEC SREG, 1992:20). In addition to error discovery, application type, development environment, and product characteristics also influence error introduction.

> Based on the factors that influence error introduction and error discovery, Nikora is developing a dynamic model for describing the identification and correction of errors during early phases of the software development life cycle. The goal of this model is to determine a relationship between the number of faults discovered during a particular life cycle phase, the number of development personnel available during the phase, and the scheduled development time. Assuming that error introduction and recovery rates could be determined for each phase of the software life cycle, the model could be applied to determine the total number of errors that will have been introduced into the product during the testing phase by the fault removal process. From that, an estimate of the reliability of the software could be made. (AFOTEC SREG, 1992:20)

2.6.6 Fagan-Style Inspections

The basic principle of inspection is the "systematic examination of interim work products to detect errors as early as possible" (Russell, 1991:25). The inspection procedure has ten rules, which if followed, can be up to 20 times more efficient than testing (Russell, 1991: 29). The rules originally developed by Fagan (1976) are:

> Inspections are carried out at a number of points in the process of project planning and system development.
>
> All classes of defects in documentation are inspected, not merely logic or function errors.
>
> Inspections are carried out by colleagues at all levels of seniority, except the big boss.
>
> Inspections are carried out in a prescribed series of steps, such as individual preparation, public meeting, error rework, etc.
>
> Inspection meetings are limited to two hours.
>
> Inspections are led by a trained moderator.
>
> Inspectors are assigned specific roles to increase effectiveness.
>
> Checklists of questions to be asked by the inspectors are used to define the task and to stimulate increased defect finding.
>
> Material is inspected at a particular rate which has been found to give maximum error-finding ability.
>
> Statistics on types of defects are kept, and used for reports which are analyzed in a manner similar to financial analysis.

Inspections are a formal, efficient, and economical method of finding errors in design and code. Fagan identifies three distinct inspections during the programming process: internal specifications inspection, logic specifications or design complete inspection, and coding/implementation inspection. The internal specifications

inspection yields an improvement in productivity by purging errors from the product which allows rework of these errors near their origin, early in the process (Fagan, 1976:29). Rework performed in the last half of the development process is 10 to 100 times more expensive than if it is performed during the level the defect was created. Finding errors by inspection and reworking them earlier in the process reduces the overall rework time and increases productivity (Fagan, 1976:29).

The objective of inspections is to find defects. A defect or fault is any condition that causes malfunction or that precludes the attainment of expected or previously specified results (Fagan, 1976:36). In early software life cycle terms, deviations from specifications are termed defects. During an inspection, deviations are pointed out, and noted by the moderator. The defect is classified and given a severity classification (e.g., major, minor, or commentary), and then the inspection continues. An obvious solution to a defect can be noted, but no specific solution hunting is to take place during inspections, because it slows the inspection and also leads to inferior quick fixes and limits the designer's responsibilities (Russell, 1991:26).

The quality of the software increases with the use of inspections because when defects are identified for removal during inspections, the result is fewer defects in the final product. Thus, when the software is tested there are fewer failures, which increases the software reliability. The initial specifications inspection, like the early life cycle software reliability prediction models, takes place prior to coding.

2.6.7 The Cleanroom Software Development Process

Traditional software engineering accepts the inevitability of software defects as part of the life cycle and implements testing and defect detection mechanisms or processes in an effort to identify and remove the defects. Even after the execution of the often elaborate and expensive test and removal process, a certain number of defects are considered acceptable. In the Cleanroom process this is unacceptable. The Cleanroom process is a team-oriented process that makes software development more manageable and predictable through the use of statistical quality control. "Cleanroom delivers software with a known and certified mean-time-to-failure (MTTF)" (GSAMSIS, 1996:15-49). As the name implies, the approach is to prevent the inclusion of defects rather than their detection and removal. Like the development of computer hardware components, the environment is to be as free of contaminants as possible. For hardware this is generally controlled by clothing restrictions, cleanliness standards, and strict policies and procedures for their development. Software does not lend itself to clothing and cleanliness constraints, however, the mentality can be applied. The use of practices and procedures to eliminate errors or defects from software requirements, specifications, design, and development should provide a zero defect mentality resulting in lower costs and higher quality.

"In the Cleanroom Software Development Process, correctness is built in by the team through formal specification, design, and verification" (Linger, 1994:51). "Cleanroom achieves statistical quality control over software development by strictly separating the design process from the testing process" (GSAMSIS, 1996:15-49). The

correctness verification replaces unit testing and debugging, thus the team never executes the code prior to entering system testing (Linger, 1994:51). The quality of the software is built in, versus tested in. "Cleanroom development teams produce software approaching zero-defects through the use of a rigorous stepwise refinement and verification process for the specification and design using object-based "Box Structure" technology" (GSAMSIS, 1996:15-49). "Box Structures define required system behavior, derive and connect objects comprising a system architecture" (Linger, 1994:52).

> Cleanroom-developed software is subject to rigorous verification by development teams prior to release to certification test teams. A practical and powerful process, correctness verification permits development teams to completely verify the software with respect to specifications. A "Correctness Theorem" defines the conditions to be met for achieving zero-defect software. These conditions are confirmed in special team reviews - through group reasoning and analysis that result in 'mental-proofs-of-correctness'. (GSAMSIS, 1996:15-51)

It is well known that exhaustive testing for large (sometimes even small) programs could take more than a lifetime. "Even though programs of any size contain virtually an infinite number of paths, the Correctness Theorem reduces verification to a finite number of steps and ensures that software logic is completely validated in all possible circumstances" (GSAMSIS, 1996:15-51).

> Cleanroom certification teams mathematically certify software reliability. Following correctness verification, software increments are placed under engineering change control and undergo first execution. Statistical usage testing is performed to produce scientifically valid measures of software quality and reliability by testing software the way it is intended to be used. Test cases are built based on usage probability distributions that model anticipated use in all possible circumstances, including unusual and stressed situations. Objective statistical measures of software reliability, such as MTTF, are computed based on test results. Because Cleanroom statistical testing is based on random

> sampling driven by input probability distributions (i.e., independent of human selection), it is uniquely effective at finding first those more serious high-frequency defects with high failures rates first. (GSAMSIS, 1996:15-51)

It is, thus, far more effective at improving software reliability in less time than traditional testing methods.

Cleanroom testing is language, environment, and subject-matter independent. "It is responsible for validating software quality with respect to the software specification. If the software's quality is not acceptable, it is removed from testing and returned to the development team for rework and re-verification" (GSAMSIS, 1996:15-53).

2.7 Summary

With the ever increasing role that software plays in modern-day systems, concern has steadily grown over the quality of software. Since the most important facet of quality is reliability, software reliability engineering has generated quite a bit of interest and research in the software community. Most approaches that attempt to predict software reliability do so in the later stages of the life cycle (i.e., integrated test and beyond). What is lacking are models and techniques that are applicable in the earlier phases, when less costly changes can be made (Lyu, 1996:111).

3. Methodology

3.1 Introduction

The primary goal in this research effort was to provide the USAF software testers with an analysis of existing early reliability prediction models, recommendations as to which would provide the most utility at a reasonable cost (in terms of effort), and suggestions for improving early reliability prediction capabilities. Initial plans were to perform:

1. Identification: Identify existing approaches to early reliability prediction. The operational definition of "early life cycle" is defined as, "prior to and including the software specification review."

2. Static Analysis: Analyze these approaches statically in terms of utility, including availability of required data early in the project, difficulty in collecting needed data, and whether required data is currently collected.

3. Dynamic Analysis: Analyze promising approaches from the Static Analysis in a dynamic manner, testing their ability to measure software reliability on Computer Software Configuration Items (CSCIs).

3.2 Identification

Seven techniques to improve software reliability early in the life cycle were identified in the Literature Review. Each are briefly reviewed for their application to this research effort.

3.2.1 Rome Lab Model 1

The Reliability Prediction Figure of Merit (RPFOM) is calculated by taking the product of the values associated with three metrics (system application type, system development environment and software characteristics) and producing the predictive fault density. The system application type (A) and the system development environment (D) metrics are determined during software pre-development, or by the Software Requirements Review. The software characteristics (S) metric is subdivided into two separate metrics: the Software Characteristics During Software Requirements and Design Specification(S1), and Software Characteristics During Software Implementation (S2). The S1 metric is determined beginning in the software requirements analysis and completed by the critical design review. The S2 metric is a refinement of the RPFOM and is calculated by the completion of coding and unit testing.

3.2.2 Relative Complexity Metric

The relative complexity metric is the relationship of software faults and measurable software attributes, such as software complexity measures. These software complexity measures are determined by performing a factor analysis on 16 complexity metrics collected with an Ada Analyzer. For example, those 16 metrics, to name a few, include: the number of executable statements in the unit, the number of unique operators referenced in the unit, the count of references to the operators in the unit,

the count of the number of unique operands, total count of all references to the operands, McCabe's cyclomatic complexity, and a count of all the nodes.

3.2.3 Software Process Failure Mode Model

This Software Process Failure Mode Model suggests an approach to predict software reliability based on systematic identification of software process failure modes and their likelihood. It permits an iterative approach to prediction, requiring updates throughout the different phases of the life cycle. A requirement-based fault tree is constructed that relates software failures to applicable software process failure modes. An estimation of software process failure mode probabilities is established along with a determination of software reliability through the fault tree solution.

3.2.4 Projecting Software Defects from Analyzing Ada Designs

This model, developed by (Agresti and Evanco 1992), predicts defect density based on the product and process characteristics. The research associated with the model is concerned with software quality factors of reliability, maintainability, and flexibility. The approach of this model is the use of multivariate models that incorporate design characteristics in an attempt to explain software defect density. This model projects the quality of the complete system from the analysis of the design structure of that system.

3.2.5 Error Introduction and Removal Model

Nikora proposes a model for error discovery and removal to predict the reliability of software during early life cycle phases. The error discovery rate and the

error removal rate can be determined from various process and product metrics, such as the type of application being developed, the specification and design methods used, and the type of technical reviews (Nikora, 1993:41).

Based on these factors' influence on error introduction and error discovery, Nikora is developing a dynamic model for describing the identification and correction of errors during early phases of the software development life cycle. The goal of this model is to determine a relationship between the number of faults discovered during a particular life cycle phase, the number of development personnel available during the phase, and the scheduled development time. Assuming that error introduction and discovery rates could be determined for each phase of the software life cycle, the model could be applied to determine the total number of errors that will have been introduced into the product during the testing phase by the fault removal process. From that, an estimate of the reliability of the software could be made (AFOTEC SREG, 1992:20).

3.2.6 Fagan-Style Inspections

The inspection process suggested by Fagan can be employed as early as requirements analysis and does not require historical data to compute metrics. Inspections utilize the developer-produced requirements documents as they are completed and before the next phase of development begins. Inspectors provide feedback to the developer so that corrections can be made before the specification is

finished, resulting in a better specification. Inspections require strict adherence to the ten rules for successful operations, as outlined in Section 2.6.6.

3.2.7 The Cleanroom Software Development Process

The Cleanroom process is a team-oriented process that makes software development more manageable and predictable through the use of statistical quality control. As the name implies, the approach is to prevent the inclusion of defects rather than their detection and removal. The use of practices and procedures to eliminate errors or defects from software requirements, specifications, design, and development should provide a zero-defect mentality, resulting in lower costs and higher quality. Therefore, correctness is built in by the team through formal specification, design, and verification. "Cleanroom testing is language, environment, and subject-matter independent. It is responsible for validating software quality with respect to the software specification" (GSAMSIS, 1996:15-52).

3.3 Static Analysis

The static analysis of the seven early life cycle software reliability improvement techniques consists of three questions and answers. If the answer to either the first or second question is "No" the technique is removed from further consideration. If the answer to the third question is "No" the technique is considered, and suggestions are made as to how the answer might be changed to "Yes."

Question 1: **Is the data available early in the development life cycle, thereby meeting the operational definition of "early" as specified in Section 3.1?**

The scope of this study has focused on early life cycle because that is when the largest and most effective savings can be made in cost and schedule. Faults remaining in the design documents and implemented into code are 100 times more expensive to remove during testing than if the faults were removed when they were introduced during the transformation of specifications to design documents (Fagan, 1976:37, Fagan, 1986:746, DeMillo *et al.*, 1986:23).

Question 2: **Is there ease in collecting the data?"**

The ease of data collection is stipulated because for metrics to be worthwhile they must be regularly collected. Therefore, the data collection must be sufficiently simple so as to minimize disruption to normal working patterns (Fenton and Pfleeger, 1997:477).

Question 3: **Is the required data currently collected?"**

The models cannot be executed or statistically verified without data.

3.3.1 Rome Lab Model 1

Answer to Question 1: Yes.

The Rome Lab Model 1 requires the completion of specific worksheets depending upon the phase of the development life cycle, and has worksheets specifically for Software Specification Review (SSR). A fault density can be determined as early as SSR by completing the worksheets designed for anomaly management,

traceability, and quality review for this phase of the development. Only the S1 metric of the software characteristics (S) metric can be considered since data for S2 is not available during the early life cycle phases.

Answer to Question 2: Yes.

Look-up tables are used to obtain the application type and the system development environment metrics. The software characteristics metric, and the metrics that comprise it, are derived by comparing the ratio of yes and no answers from the worksheets to predetermined values found in (RL-TR-92-52, 1992).

Once the data is converted to the specific metric, the RPFOM can be calculated resulting in a fault density. Then, based on expert judgment, an approximate estimate of software size can be obtained, and the subsequent number of defects, by multiplying the fault density with the CSCI size.

Answer to Question 3: No.

However, because the data is not currently being collected, the model cannot be exercised, nor can the metrics for predicting early software reliability be calculated. What Air Force software testers need to do to use this model, is provide the worksheets and require software developers complete them. Once the worksheets are collected, the data can be converted automatically to the metrics, and the fault density determined. Once the fault density is determined, the program can then be tested and the actual fault density can be validated. Additionally, after the fault density is determined, managers can use this information to improve the software reliability through resource and risk management.

3.3.2 Relative Complexity Metric

Answer to Question 1: No.

This model requires actual completed designs to determine a ranked ordering of modules' complexity. This completed design implies a phase of development beyond what is defined as early. Although, it is reasonable to assume that this method could be applied earlier in the development phase if there were methods to measure the complexity of design and specification documents (Nikora, 1993:36). However, even if this were possible, this model does not meet the "early life cycle" requirement because it relies on data that requires a completed design and some code. Therefore, this model will not be considered for further analysis.

3.3.3 Software Process Failure Mode Model

Answer to Question 1: Yes.

The data for the Software Process Failure Mode Model are the failure modes and their influencing factors, which originate early in the life cycle from the software design requirements. The fault tree is then built by determining how each of the requirements relate to each other and how they can fail.

Answer to Question 2: No.

"The actual fault tree will be quite complex since software process failure modes generated by all phases of the development process may contribute to the failures" (Smidts *et al.*, 1996:136).

3.3.4 Projecting Software Defects from Analyzing Ada Designs

Answer to Question 1: No.

This model focuses on designs that will eventually be implemented in the Ada programming language. The model concentrates on a higher level of abstraction than the control flow and data flow within individual procedures. The sole assumption is that design representations be compilable, indicating some form of written code. Additionally, as the name implies, the model requires design data that will not be available until preliminary design review, well after the Software Specification Review. As such, this model is not considered further.

3.3.5 Error Introduction and Removal Model

Answer to Question 1: No.

The Error Introduction and Removal Model requires the collection of data that can be used to determine the rate at which errors are introduced and removed from a product. Moreover, the model could be applied to determine the total number of defects introduced into the product during the testing phase by the fault removal process. Determining the total number of defects during the testing phase does not meet the "early" requirement, as specified in the Section 3.1.

3.3.6 Fagan-Style Inspections

Answer to Question 1: Yes.

The Fagan-Style Inspections can be utilized throughout the software development life cycle making them attractive for improving software reliability. The

required data for inspections prior to the software specification review is the specifications itself. The inspection can occur prior to the entire specification being written, indicating that the inspection process can begin as early as the first day, making data collection easy.

Answer to Question 2: Yes (sort of).

Two approaches to inspections were identified and evaluated: the more exhaustive Fagan style inspections and Gilb's random sample inspections. Fagan suggests completely inspecting each product developed by the process. However, Gilb suggests using a random sample of representative documents and products from the process to determine the quality of the products being developed. Each of these requires a commitment to the software development process that many developers are capable of introducing to their process.

Answer to Question 3: No.

The Air Force should require inspections be used in software development, but they can recommend the Air Force require it on the basis that it reduces the testing cost (Humphrey, 1990:189).

3.3.7 The Cleanroom Software Development Process

Answer to Question 1: Yes.

As discussed in the Literature Review, the Cleanroom process builds in correctness through formal specifications, design, and verification. Therefore, the quality of the software is built in, rather than determined through testing. The

technique for early software reliability prediction is the process, and it can be implemented as early as the formal specification.

Answer to Question 2: Yes (maybe).

The Cleanroom process translates to a CMM Level 5 process (Dyer, 1992:7). Software developers would need to invest a significant amount of time and capital to adopt this process. However, methods exist for a phased-in approach which can realize results early on (Dyer, 1992:5).

Answer to Question 3: No.

The Air Force should strive for developers to use the Cleanroom development process for future software products, although they cannot require it.

3.3.8 Summary

Of the seven techniques that were statically analyzed, only three received "Yes" answers to the first two questions, and none were able to answer "Yes" to the third question. Suggestions were made for those techniques that did not answer "Yes" to the third question, when applicable.

3.4 Dynamic Analysis

The static analysis identifies three promising approaches, and they are analyzed according to their ability to enhance software reliability on Computer Software Configuration Items (CSCIs). Unfortunately, data was not available for the Rome Lab Model 1, but had it been, the dynamic analysis describes how the tests would have been conducted.

3.4.1 Rome Lab Model 1

Thirty CSCIs that have completed operational test and evaluation and have a known fault density would be randomly selected. The model would then be applied to the CSCIs to obtain the fault density prediction. After selecting a confidence interval of 90-95%, the fault density predictions would be plotted against the true fault density to validate the model.

3.4.2 Fagan-Style Inspections

Inspections do not lend themselves directly to a dynamic analysis because there is no execution of a model, but the analysis of how to execute an inspection and the cost and quality benefits are presented instead.

Fagan-Style Inspections require the specification be collected early, as it is produced, and reviewed in a precise manner allowing for feedback to the developer. The developer then has the ability to modify the creation of formal specifications so that they are correct and less ambiguous. It has been shown that inspections yield less defective products than those created without inspections, which rely on testing alone to remove software defects.

Benefits of Inspections:

Relying on testing alone to remove software defects leaves as much as 25% of all defects in the software to be passed on to the users. Also of consideration is that testing does nothing for the front-end defects found in requirements and designs (GSAMSIS, 1996:15-40). Testing concentrates on the dynamic execution of the code,

while inspections can be applied to any human readable product developed in the life cycle process. These products might include requirements, specifications, designs, test plans, and even test cases. As discussed, inspections are defect reduction techniques at the point of the defects' origin. Also, inspections return the problem to the creator, thus forcing the individual developer to recognize personal defects produced and hopefully allowing an opportunity to prevent like errors from occurring.

Cost and Quality Benefits of Inspections:

Formal peer inspections are an industry-proven, verified and documented, successful method for removing defects and reducing costs. They can eliminate approximately 80% of all software defects, and when combined with normal testing, can reduce the number of latent defects in fielded software by a factor of 10 (Brykczynski, 1993).

The Institute for Defense Analysis estimates inpsections require an upfront investment of up to 15% of total software development costs. They indicate the peer inspection investment reaps 25% to 35% increases in total productivity, which translates in to 25% to 35% schedule savings by reducing costly rework in later phases (GSAMSIS, 1996:15-41).

The really significant news about inspections is that the statistical feedback it gives on defects and costs provides the manager with a software engineering management accounting system. This can be used to identify a wide range of productivity problems in a software development process, and then to measure and see if the suggested solutions are working as expected (Gilb, 1988:259).

The Economics of Inspection:

Even though inspections utilize limited human resources and often involve more development time, their effectiveness is due to their application early in the life cycle. Gilb (1988:218) presents the following statistics supporting inspections:

- Over 60% of the bugs which will occur in the operational software will be there before the code is ever written. They exist when code is written because the developers are so human at creating them and so poor at removing them.
- Code checking, even inspection of code, and thorough testing, will only serve to confirm the existence of these bugs in the real software.
- Testing is a maximum of 50% to 55% effective at defect identification and removal for a single test process.
- Inspection is about 80% (± 20%) effective in removing existing defects.
- Inspections can operate on specifications which computers cannot examine. It can operate in the area where the defects occur. It can operate at much earlier stages where the defects are one or two orders of magnitude cheaper to correct, when found.
- Although correction and finding of defects at inspection phases uses scarce qualified technologists, it is these same level of people who must carry out the defect finding and correction at later stages. The average is five hours saved for every hour invested in inspections."

3.4.3 The Cleanroom Software Development Process

It is possible to test the ability of the Cleanroom process to improve software reliability by producing two identical CSCIs, one using the Cleanroom process and one without, and then comparing their fault density. The Cleanroom process has shown incredible defect prevention for numerous projects, with some having 0.0 defects per thousand lines of code (Linger, 1994:56).

The Cleanroom process, unlike the current development process is effective in preventing faults from being propagated along the development process. "Defect detection (through testing) and their removal (through rework) leads to more defects through bad fixes. The Cleanroom process is a proven method for eliminating this source of defects" (GSAMSIS, 1996:15-42).

3.5 Summary

The five models and two techniques were identified and statically analyzed for their ability to improve software reliability. Only one model (Rome Lab Model 1) and the two techniques (Inspections and Cleanroom) have data available early in the software development life cycle, and have relative ease in collecting the requested data. But because data is not available to validate the Rome Lab Model 1, the model could not be dynamically analyzed to verify its accuracy. Consequently, because inspections and the Cleanroom process are processes, they cannot be dynamically analyzed other than examining their cost and quality benefits.

4. Results

4.1 Conventional Numerical Models

The five conventional or numeric models identified have shown to be not only passive, but reactive in nature. Each of the models uses subjective information with the intent of predicting the final reliability of a software product, without any effort to remove the faults or improve the design. They still rely on the traditional detection and removal of faults, which is expensive in both time and capital. With the constant quest for a smaller force and smaller budgets, the USAF must find ways to reduce the costs of its software programs.

4.2 Data Availability for the Conventional Numerical Models

AFOTEC/SAS, the Air Force's primary software test agency, uses ASETS (Automated Software Evaluation Tool System), a UNIX based system, that preprocesses 10% of the source code and provides a representative sample and collects 10 metrics. The ten metrics collected by ASETS are provided in Table 4.1. It must also be understood that ASETS requires code to collect metrics and thus cannot actually be used for early life cycle predictions.

Since the data is not available to implement the conventional numerical models, and even if they could be implemented, their value is questionable at improving the software development process. Therefore, more emphasis should be placed on the proactive methods for improving software reliability.

Table 4.1: List of ASETS Metrics

Metric	Definition
CMTS_LOC	comments lines of code
CONTROL	control density
CY_CMPLX	cyclomatic complexity
EXEC_LOC	executable lines of code
OPRNDS	total operands
OPRTRS	total operators
U_OPRNDS	unique operands
U_OPRTRS	unique operators
SIZE	size
VOLUME	volume

4.3 Improvement Techniques Assessment

An analysis of the three techniques reveals each provides benefits early in the software life cycle. However these benefits vary based on the technique chosen. When evaluated for proactive defect prevention, each technique reflects a different level of proactive-attempts to improve reliability.

The Rome Lab Model 1 identifies CSCIs' fault density and makes no attempt to improve reliability. The two methods identified as proactive in nature and reportedly

provide significant improvements in software reliability are inspections and the Cleanroom process.

4.3.1 Rome Lab Model 1

No empirical evidence of the validity of the Rome Lab Model 1 was found, nor was there any data for this research effort to validate the model. The Rome Lab Model 1 provides an early RPFOM based on a review (an informal inspection) of the specification documentation and the product's characteristics (application and environment) through the use of certain checklists or questionnaires as found in the appendices of (RL-TR-92-52, 1992). However, this review is strictly a one-way static analysis without any feedback to the developers to correct deficiencies found, whether they are problems with the process or defects in the products. This is the true weakness of this model.

This model provides a prediction of faults, or of the failures of the process, and the defects of the products. However, it is merely for forecasting a problem without any indication of a cure. The most a user of the Rome Lab Model 1 can hope for is an indication of which modules or CSCIs might have high defect densities. It fails to make use of techniques to capitalize on the review efforts beyond that of predicting a RPFOM.

The Rome Lab Model 1 is not without merit. It should provide management an early indication of fault prone modules. This gives the development team the opportunity to invest resources early to correct deficiencies and defects when the costs

are at their lowest. However, the identification of the problems will require additional resources, as the implementation of this model is merely to identify yes or no answers to questionnaires and use the totals of those answers to develop an RPFOM. The model alone cannot improve the reliability of a software product, as some other management action is required to increase the software reliability.

4.3.2 Inspections

Two approaches to inspections were identified and evaluated: Fagan style inspections and Gilb's random sample inspections. Each of these requires a commitment to the software development process that many developers are capable of performing.

Formal inspections do not provide a fault density as it has been introduced in this research. However, it can be used to derive a defect density, much the same way as fault density is currently measured. They also provide immediate feedback for both process and product improvement. (Jones, 1991) indicates formal peer inspections: of requirements identify 40% of requirements errors; of designs identify 55% of design errors; of code identify 65% of code defects, 20% of requirements errors, and 40% of design errors escaping design reviews; and result in cumulative defect removals of 59% of requirement errors, 73% of design errors, and 65% of code defects. All of these reductions in errors are without executing a single line of code. Each of these reviews are static and do not require formal test plans or test cases.

Boehm (1989) shows the level of effort for rework to correct defects is typically 40% to 50% of the total software development. The Institute for Defense Analysis (IDA) states one reason for DoD's reluctance to jump on the peer inspection bandwagon is the increased upfront spending required - even though downstream benefits are well documented. Formal peer inspections can eliminate approxiamtely 80% of all software defects, and when combined with normal testing, can reduce the number of latent defects in fielded software by a factor of 10 (Brykczynski,1993). Furthermore, software testing cannot remove all defects. Unit testing only has a 70% to 75% cumulative defect removal efficiency (Jones, 1991).

However, just as in the Rome Lab Model 1, the use of inspections alone will not decrease the number of errors in the software product, nor will they alone increase the software's reliability. For the role of the inspection team is to identify problems, not to resolve them. Follow-up and defect removal are the responsibility of the inspected product's creator. Thus, management can use inspections to provide an indicator of where the problems lie, but the inspection process alone does nothing to increase the product's reliability.

4.3.3 The Cleanroom Software Development Process

Cleanroom appears to be the best solution, however, it is a process for the entire development. The zero-defect mentality has been shown to reduce costs and schedule as well as produce superior quality code. This process requires a major

investment into the process and can take years to implement. However, once implemented, it should reduce the overall costs of software development.

"Defect detection (through testing) and their removal (through rework) leads to more defects through 'bad fixes.' The Cleanroom process is a proven method for eliminating this source of defects" (GSAMSIS, 1996:15-42). Cleanroom, like formal inspections, do not provide a fault density. Instead, it provides a process for the complete life cycle. In the early life cycle, emphasis is placed on the use of box structures to define the functional specification. "Through stepwise refinement, objects are defined and refined as different box structures, resulting in a usage hierarchy of objects in which the services of an object may be used and reused in many places and at many levels. Box structures, then, define required system behavior and derive and connect objects comprising a system architecture" (Linger, 1994:52). The box structures are comprised of three boxes, the black box, the state box, and the clear box. Through these boxes, "box structures bring correctness verification to object architectures as the state boxes can be verified with respect to their black boxes and clear boxes with respect to their state boxes" (Linger, 1994:53).

"Initial box-structure specifications often reveal gaps and misunderstandings in customer requirements that would ordinarily be discovered later in development at high cost and risk to the project" (Linger, 1994:52). They also address two of the engineering problems associated with system specification. Those are defining the right function for users and defining the right structure for the specification itself. Box structures address the first problem by precisely defining the current understanding of

required functions at each stage of development, so that the functions can be reviewed and modified if necessary. For the second problem, box structures incorporate the crucial mathematical property of referential transparency. That is, the information content of each box specification is sufficient to define its refinement, without depending on the implementation of any other box. This property permits large-system specifications to be organized hierarchically, without sacrificing precision at high levels or detail at low levels (Linger, 1994:52).

Experience with over one million lines of Cleanroom-developed software from a variety of programs have shown extraordinary quality compared to traditional results, as depicted in Table 4.2.

Table 4.2 Cleanroom Performance Measures (GSAMSIS, 1996:15-54)

Software Development Practices	Development Defects/KLOC	Operational Failures/KLOC	Productivity LOC/Staff Month
Traditional Software-as-art	50-60	15-18	Unknown
Software Engineering	20-40	2-4	75-475
Cleanroom Engineering	0-5	<1	>750

Unlike the Rome Lab Model 1 and Fagan Inspections, the Cleanroom implementation can improve the reliability of the software for it is a complete process that includes feedback and rework prior to testing. Cleanroom has been adopted by over 30 development organizations who have never experienced a Cleanroom program failure (GSAMSIS, 1996:15-53). Obviously, this must consider the fact that very few

organizations will publish their failures for the world's reveiw, but in itself, it is quite a remarkable statement.

4.4 Summary

Coupling the Rome Lab Model 1 with Fagan-style inspections would greatly enhance the reliability of the products by identifying the defects in the products and the shortcomings in the processes early in the life cycle when the cost to repair them is at its lowest. Ideally, software development should move towards more precise and rigorous standards, which the Cleanroom process embodies, for improving software reliability.

5. Conclusions and Recommendations

5.1 Conclusions

Although each of the models evaluated have their own merits, none of the quantitative models can be recommended because the data required to perform a quantitative analysis is not available. The Air Force is not currently collecting the metrics data required to fully exercise any of the early life cycle software reliability prediction models. If the Air Force chooses to pursue this avenue of determining fault densities from early metrics data, then they must start collecting the data for future research and analysis.

It should be considered possible that early software prediction models may have become obsolete prior to ever being fully implemented. The alternatives (e.g., Cleanroom and Inspections) to using fault density predictors are founded in the concept of designing-in quality instead of trying to test-in quality.

5.2 Recommendations

It is highly possible the time has come for the Air Force to migrate to a new paradigm. The era of spending resources to determine the number of faults in a software program has reached its obsolescence. No longer should the Air Force spend money to determine how poorly the efforts of the development teams have been. No longer should the they spend time to fix problems that should have been fixed early in the life of the project. The Air Force should not be spending its resources to determine the fault densities of code. This approach tries to determine an acceptable

level of faults and probabilities of failure, thus accepting the latent faults of the developers without holding them accountable. What is recommended is the program managers should be trying to determine the best way to get as close to zero defects in the programs as possible. The efforts spent on determining the number of faults can be reduced, the number of faults can be reduced, and the quality of the software can be increased without any effort to determine the fault densities of modules of code.

The software development process has reached a point of maturity that should permit the inauguration of a new paradigm, one that works to reduce faults in software by actually designing for near zero defects. Two techniques exist for designing for near zero defects: Inspections (both Fagan and Gilb style) and the Cleanroom process. Each of these have proven to be effective and efficient means to reduce the errors or defects in software projects (Russell, 91:25 and Linger, 94:50).

The Cleanroom process, more so than inspections, has shown to be a process that makes development more manageable and cost-reducing. The Cleanroom process is a complete process that requires a major investment to shift to this process. As a government agency, the Air Force cannot dictate the development process to this level of detail for contractors. However, this should be considered during source selection. It should be noted that the Cleanroom process requires inspections and structured programming as a baseline for meeting software development goals. This requirement is expounded upon in Section 5.3.

The less obtrusive approach is inspections. This approach has been around for over 21 years, first introduced by Michael Fagan at IBM in 1976, and has shown its worth in not only reducing the number of defects in the software, but also in reducing

overall project costs (Russell, 91:26). Originally developed for general-purpose verification, inspections can be applied to many work products including System Requirements Documents (SRDs), System Specifications Documents (SSDs), System Design Documents (SDDs), test plans, and even the implementation code. The inspections are formal processes with entrance and exit criteria, and should be introduced very early in the system's life cycle. This suggests the Air Force would be actively involved in the product development at the Requirements Analysis Phase and work with the development teams to ensure the SRD is as close to defect free as possible. Although this requires an investment of resources up front, Fagan (1976:45), (1986:54); Gilb (1996:7), and Russell (1991:26) have shown the potential for a high return on investment and an overall reduction in life cycle costs. Fagan (1986) suggests all design and code inspection costs are 15 percent of development costs and Ferens (1997) suggests 22 percent of development costs are for testing without inspections. This would suggest an overall seven percent reduction of total development costs.

5.3 A Cleanroom Software Development Process Vignette

All software development programs have multiple choices for the development methodologies and tools. Cleanroom and Inspections are two of the approaches available to software development managers and engineering teams. Since Inspections are a required baseline (along with structured programming) for the Cleanroom process, this vignette will focus on the Cleanroom process. It should also be noted that both of these techniques can be applied to all different types of system development, but have the greatest potential for larger, more complex programs.

Cleanroom has been used successfully in all corners of the Department of Defense. The Air Force STARS Demonstration Project, managed by the USAF Space and Warning Support Center at Peterson AFB, Colorado, is integrating a Cleanroom engineering process and an Ada Process Model to develop a Space Command and Control Application. The Army STARS Demonstration Project, managed by the Army Life Cycle Software Engineering Center at Picatinny Arsenal, is using the Cleanroom process on two programs, the Conduct of Fire Trainer (COFT) and the Mortar Ballistics Computer (MBC). The Department of the Navy's Naval Coastal Systems Station team is applying the Cleanroom process to develop an embedded software system that provides a bridge between workstations and the Position Location Reporting System (GSAMSIS, 1996:15-53). From space to ballistic munitions, the Cleanroom process can be applied to reduce the program's software errors.

Industry also boasts examples of successful Cleanroom and Inspection applications. IBM's COBOL Structuring Facility used Cleanroom to develop a 85 thousand lines of code (KSLOC) which automatically structures unstructured COBOL programs. In this effort the developers were able to reduce the average number of errors per to 3.4 errors/KSLOC in all testing. Furthermore, the product experienced only seven minor defects in three years of use. This is a testimony to the claims of reduced maintenance costs associated with products developed with the Cleanroom process. Another example was the Software Engineering Laboratory (SEL) at NASA Goddard. They have completed their second and third Cleanroom programs, a 20 KSLOC attitude determination subsystem and a 150 KSLOC flight dynamics system

that experienced a combined defect rate of 4.2 defects/KSLOC in all testing (GSAMSIS, 1996:15-55). Dyer (1992: 23) reports the SEL realized a reduction of average error rates from six to 3.3 errors/KSLOC for one of the projects. Ericsson AB Telecommunications Operating systems, a 350 KSLOC system for switching computers resulted in 1.0 defects/KSLOC in all testing. Productivity improvements were reported as 70% for development and 100% for testing (GSAMSIS, 1996:15-55).

Cleanroom techniques can be applied to any software development project, but it must be done with care and commitment if the improvements are to be realized. As with any radical change within an organization, lack of top level support is certain to lead to implementation failure. However, once this support is obtained and the organization embraces the Cleanroom concepts, improvements can be made in almost every aspect of the software product's life cycle. Furthermore, the techniques can be applied to every level of effort in every phase of the software development life cycle.

Dyer (1992:5) states "The Cleanroom components are organized along the six technical lines of software specification, software development, software correctness verification, independent software product testing, software reliability measurement, and statistical process control." This represents a component for each of the software development phases after initial need is identified. As discussed in the Literature Review, the software life cycle consists of the five primary phases: requirements analysis, design, implementation, test, and maintenance. Following is a brief explanation of how Cleanroom is used at each stage.

Requirements analysis is the phase where the problem is analyzed and defined; the result of this phase is a formal specification of a system that will meet customer

needs. This is generally where the problems begin. An undetected error in the formal specification that has an impact may actually manifest itself as a fault in the design or the implementation. The longer an error is allowed to remain in the system, the more it will cost in time and money to fix it. In the requirements phase, Cleanroom forces requirement deficiencies into the open and establishes early control of the requirements process (Dyer, 1992:5). It also requires a formal notation for the specification (Dyer, 1992:17), no longer are natural language specifications accepted.

> These formal methods force a more exacting analysis of the requirements and tend to minimize ambiguity, inconsistency, and incompleteness in the resultant software specification. The methods support the verification of developed specifications so that specification correctness can be demonstrated at the start of software design. (Dyer, 1992:18)

In design, the software specification is transformed into a conceptual solution. Decisions are made about issues such as programming language, platform the software will operate on, type of data structures required, and other performance-related issues. The output of the design phase is a design representation in the form of text, flowcharts, decision tables, or program design language. With Cleanroom, the most significant impact to the design process is the introduction of functional correctness verification (Dyer, 1992:18).

> The Cleanroom design ethic is one of requirements specification, followed by design of a solution to the specification, followed by verification of the equivalence between the design and requirements. This ethic is applied in small design steps that give a decomposition of a specification into appropriate logic tying together a breakdown of the original specification into more detailed level specifications. These in turn need subsequent design and verification. Verification is integral to

> the design construction and imposes a control on the designer which gates the refinement of the software specification. (Dyer, 1992:19)

During implementation, the conceptual design representation is transformed into an actual program language through coding or reuse. With Cleanroom, the implementation can be a simple process. "If design and verification are performed to full detail in the design step, then implementation becomes a transliteration of design notation into programming language notation" (Dyer, 1992:19).

Testing ensures the software performs according to the specification, identifies faults, and collects failure data. An analysis of the data is performed to determine reliability predictions of the time of the next failure. This phase also addresses the correction of those faults and the testing of those changes. Testing generally takes two forms: developer and independent. The developer tests the software to ensure it functions in accordance with the design; while the independent test is to ensure it meets the requirements. With Cleanroom, developer testing is eliminated. "Close adherence to functional correctness verification ensures that all of the error detection situations addressed by developer testing are addressed in verification" (Dyer, 1992:19). Independent testing is still a valuable tool in software development. Cleanroom relies on independent testing to validate that the software requirements were correctly specified, designed, and ultimately implemented. "To ensure that the testing focuses on software usage, the Cleanroom method impacts traditional testing approaches by introducing statistical techniques" (Dyer, 1992:20).

The maintenance phase is the process of responding to user needs: fixing errors, making user-specified modifications, and honing the program for usability. Cleanroom technique were developed in an effort to improve the overall quality of software products. Results indicate it is achieving this goal. "In both the COBOL and SEL cases, the reported postdelivery errors were extremely small and measured in fractions of one error per KSLOC" (Dyer, 1992:23).

With the introduction of all of the new components and techniques required to implement Cleanroom fully, it might be expected to negatively impact productivity and schedules. However this has not been the case.

> A surprising result of Cleanroom work is that software productivity did not go down and, in fact, increased in several cases. From the development side, design simplicity and the complete elimination of developer testing resulted in reduced effort that more than compensated for the work to integrate correctness into software designs. In the case of COBOL S/F and NASA SEL projects the reported productivities were in the range of 750 lines of source code per labor month, which is three to four time higher than the average productivities reported in the software literature. From the testing perspective, the structure that statistical techniques provided to the sampling process and the exceptional quality of the software to be tested resulted in either the same or better productivity for performing independent software validation. (Dyer, 1992:25)

The Cleanroom process is not a procedure that can be implemented in a single phase of the software life cycle, rather one that should be implemented and enforced throughout each phase. However, only the developing organization can make the decision to fully implement Cleanroom. Its implementation will require commitment from the top, the middle and the bottom to succeed. This is not a case where the customer wishes it to happen and the supplier quickly makes it so. Cleanroom

implementation will require the long-term commitment of resources and a radical change in the current business practices of the software developing organization.

5.4 Suggestions for Further Study

As a means of verifying that inspections are a timely and cost effective way of preventing faults, (i.e., increasing the reliability of software programs), the researchers suggest an experiment for managing three similar CSCIs. The researchers note the requirements for this experiment are unrealistic, so close adherence to them is the best alternative.

First, the three CSCIs must be identical in size, application type, number of requirements, etc. Since no three CSCIs can be exactly alike, unless they are all the same (which would be expensive and wasteful), they must be as close to the same as possible. Second, the developers of each of the CSCIs must be as experienced and capable as each other. Just as no three CSCIs are alike, no three developers are alike either. Instead, the most similar developers must be selected to each develop one of the similar CSCIs.

The experiment requires each of the developers translate the software requirements of a CSCI to formal specifications, but each of the developers will use a different process. The first developer will use inspections after several pages of the specification are translated to determine the number of faults or ambiguities that exist in the specification. Those faults will be noted by inspectors but will not be brought to the attention of the developers. The second developer will also use inspections like the first, but the faults will be pointed out to the developers and that feedback will be

used to make the following pages of the specification better. The last developer will not use inspections at all and will be the control person.

Once the three CSCIs are developed, system testing will determine which of the CSCIs has the lowest number of faults. The experiment can simultaneously keep track of costs and when each of the projects have similar fault densities, through rework, the cost to achieve that fault density can be compared.

5.5 Summary

The results and conclusions of this research effort suggest the Cleanroom Software Development Process is the most effective means of improving the reliability of software. Developing software with a Cleanroom process is a radical change in software development, and like all changes in organizations, is difficult to accomplish. What can be done, though, is to implement the baseline activities as discussed in the vignette. On the other hand, if the Cleanroom Software Development Process cannot be implemented, then Fagan-style inspections would provide an efficient and cost effective means to improving the quality of software for USAF development projects. These techniques must be incorporated into the program management tasks early in the development life cycle and maintained throughout the development to realize the greatest return on investment and the highest quality software.

Bibliography

AFOTEC SREG, "Software Reliability Evaluation Guide, Methodology Investigation Report," (SREG) Air Force Operational Test and Evaluation Center/SAS, Feb 1995, pp. 17-20.

Agresti, William W. and William M. Evanco, "Projecting Software Defects From Analyzing Ada Designs," IEEE Transactions on Software Engineering, 18, No. 11, Nov. 1992, pp. 988- 997.

Beizer, Boris, Software System Testing and Quality Assurance, Van Nostrand Reinhold, New York NY, 1984.

Brykczynski, Bill and David A. Wheeler, "An Annotated Bibliography on Software Inspections," Institute for Defense Analysis, February 5, 1993.

DeMillo, Richard A., et. al., Software Testing and Evaluation, Menlo Park CA: Benjamin/Cummings Pub. Co., 1987.

Department of the Air Force, Software Technology Support Center's "Guidelines for Successful Acquisition and Management of Software-Intensive Systems: Weapon Systems, Command and Control Systems, Management Information Systems," (GSAMSIS) June 1996, Ver 2.0, Vol 1.

Dyer, Michael, The Cleanroom Approach to Quality Software Development, John Wiley & Sons, Inc., New York NY, pp. 1-25.

Fagan, Michael E., "Design and Code Inspection to Reduce Errors in Program Development," IBM Systems J., 15, No. 3, 1976, pp. 25-53.

Fagan, Michael E., "Advances in Software Inspections," IEEE Transactions on Software Engineering, 12, No. 7, July 1986, pp. 744-751.

Fenton, Norman E. and Shari Lawrence Pfleeger, Software Metrics: A Rigorous & Practical Approach, Second Edition, PWS Publishing Company, Boston MA, 1997.

Ferens, Daniel V., Software Cost Estimation Lecture Notes, Air Force Institute of Technology, Spring Quarter, 1997.

Gilb, Tom, "Inspection Economics," Addison-Wesley Longman, 1993.

Gilb, Tom, Principles of Software Engineering Management, The Bath Press, Avon, 1988.

Gordon, Daniel, Personal Interview with F-22 Systems Program Office, Computer Support Officer, Wright-Patterson AFB OH, 1996.

Humphrey, Watts, Managing the Software Process, Software Engineering Institute, Addison-Wesley Publishing Company, Reading MA, 1990.

Jones, Capers, Programming Productivity, McGraw-Hill Book Co., New York NY 1986.

Linger, Richard C., "Cleanroom Process Model," IEEE, March 1994, pp. 50-58.

Lyu, Michael R., Handbook of Software Reliability Engineering, McGraw Hill, New York NY, 1996.

Munson, J. C. and T. M. Khoshgoftaar, "Predicting Software Development Errors Using Complexity Metrics," IEEE Selected Areas in Communications, 12, No. 3, 1990, pp. 283-291.

Munson, J. C. and T. M. Khoshgoftaar, "Measuring Dynamic Program Complexity," IEEE Software, November 1992, pp. 48-55.

Musa, John D., Anthony Iannino, and Okumoto Kazuhira, Software Reliability: Measurement, Prediction, Application, McGraw Hill, New York NY, 1990.

Nikora, Allen P. "Early Prediction of Software Reliability From Product and Process Characteristics," Dissertation, University of Southern California, 1993.

Russell, Glen W., "Experience With Inspection in Ultra-large-Scale Developments," IEEE Software, January 1991, pp. 25-31.

Science Applications International Corp (SAIC) and Research Triangle Institute (RTI), "Software Reliability, Measurement, and Testing Guidebook for Software Reliability Measurement and Testing" 1 and 2, RL-TR-92-52, April 1992.

Smidts, C. R., W. M. Stoddard, and M. Stutzke "Software Reliability Models: An Approach to Early Reliability Prediction," IEEE, 1996, pp. 132-141.

Vita

Capt John G. Leonard [redacted] attended Ashtabula Senior High School, and enlisted in the US Air Force in May 1981. Over the course of the next 12 years he attended colleges at each of his assignments, and in May 1988 he entered the University of Maryland, European Division. He graduated with a Bachelor of Science degree in Computer Studies, Information Systems Management in January 1992.

His first assignment was at Grissom AFB as an Inventory Management Specialist, after which he was assigned to Carswell AFB as a Operations Intelligence Technician. While in the Intelligence career field, he was assigned to RAF Alconbury, UK. It was while at RAF Alconbury he was accepted to attend OTS, where he graduated on 23 September 1992. After OTS, he was assigned to the Aeronautical Systems Center as a Communications-Computer Systems Officer. In May 1996, he entered the School of Logistics and Acquisition Management, Air Force Institute of Technology, Software Systems Management program.

[redacted]

Vita

First Lieutenant Ric K. Nordgren [redacted] graduated from Chugiak High School in Eagle River, Alaska in 1989 and entered undergraduate studies at Embry-Riddle Aeronautical University in Prescott, Arizona. He graduated with a Bachelor of Science degree in Electrical Engineering with a minor in Mathematics in December 1993. He received his commission on 17 December 1993 upon graduation from college and completion of the Reserve Officer Training Corps program.

His first assignment was at Wright-Patterson AFB as a Space Systems Analyst at the National Air Intelligence Center. He was later personally selected as one of the original members of the newly-created Integrated Air Defense Systems division. In May 1996, he entered the School of Logistics and Acquisition Management, Air Force Institute of Technology for a Masters of Science in Software Systems Management.

[redacted]

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1997	Master's Thesis

4. TITLE AND SUBTITLE
AN ANALYSIS OF EARLY SOFTWARE RELIABILITY IMPROVEMENT TECHNIQUES

5. FUNDING NUMBERS

6. AUTHOR(S)
John G. Leonard, Capt, USAF
Ric K. Nordgren, 1Lt, USAF

7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(S)
Air Force Institute of Technology
2950 P Street
WPAFB OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GSS/LAS/97D-1

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
HQ AFOTEC/SAS
8500 Gibson Blvd SE
Kirtland AFB, NM 87117-5558

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT (*Maximum 200 Words*)
This research explores early life cycle software reliability prediction models or techniques to predict the reliability of software prior to writing code, and a method for increasing or improving the reliability of software products early in the development life cycle. Five prediction models and two development techniques are examined. Each model is statically analyzed in terms of availability of data early in the life cycle, ease of data collection, and whether data is currently collected. One model and the two techniques satisfied those requirements and are further analyzed for their ability to predict or improve software reliability. While the researchers offer no significant statistical results of the model's ability to predict software reliability, important conclusions are drawn about the cost and time savings of using inspections as a means of improving software reliability. The results indicate that the current software development paradigm needs to be changed to use the Cleanroom Software Development Process for future software development. This proactive approach to developing reliable software saves development and testing costs. One obvious benefit of this research is that cost savings realized earlier in the software development cycle have a dramatic effect on making software development practices better and more efficient.

14. Subject Terms	15. NUMBER OF PAGES
Software, Software Engineering, Reliability, Inspection, Prediction, Quality Control	78
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UNLIMITED

AFIT Control Number AFIT/GSS/LAS/97D-1

AFIT RESEARCH ASSESSMENT

The purpose of this questionnaire is to determine the potential for current and future applications of AFIT thesis research. **Please return completed questionnaire** to: AIR FORCE INSTITUTE OF TECHNOLOGY/LAC, 2950 P STREET, WRIGHT-PATTERSON AFB OH 45433-7765. Your response is **important.** Thank you.

1. Did this research contribute to a current research project? a. Yes b. No

2. Do you believe this research topic is significant enough that it would have been researched (or contracted) by your organization or another agency if AFIT had not researched it?
a. Yes b. No

3. **Please estimate** what this research would have cost in terms of manpower and dollars if it had been accomplished under contract or if it had been done in-house.

Man Years_____ $_____

4. Whether or not you were able to establish an equivalent value for this research (in Question 3), what is your estimate of its significance?

a. Highly Significant b. Significant c. Slightly Significant d. Of No Significance

5. Comments (Please feel free to use a separate sheet for more detailed answers and include it with this form):

_____ _____
Name and Grade Organization

_____ _____
Position or Title Address