Cleanroom Software Engineering for Zero-Defect Software

Richard C. Linger

IBM Cleanroom Software Technology Center
100 Lakeforest Blvd.
Gaithersburg, MD 20877

Abstract

Cleanroom software engineering is a theory-based, team-oriented process for developing very high quality software under statistical quality control. Cleanroom combines formal methods of object-based box structure specification and design, function-theoretic correctness verification, and statistical usage testing for quality certification, to produce software that is zero defects with high probability. Cleanroom management is based on a life cycle of incremental development of user-function software increments that accumulate into the final product. Cleanroom teams in IBM and other organizations are achieving remarkable quality results in both new system development and modifications and extensions to existing systems.

Keywords. Cleanroom software engineering, formal specification, box structures, correctness verification, statistical usage testing, software quality certification, incremental development.

Zero-defect software

On first thought, zero-defect software may seem an impossible goal. After all, the experience of the first human generation in software development has reinforced the seeming inevitability of errors and persistence of human fallibility. Today, however, a new reality in software development belies this first generation experience [1]. Although it is, theoretically impossible to ever know for certain that a software product has zero defects, it is possible to know that it has zero defects with high probability. Cleanroom software engineering teams are developing software that is zero defects with high probability, and doing so with high productivity. Such performance depends on mathematical foundations in program specification, design, correctness verification, and statistical quality control, as well as on engineering discipline in their application.

In traditional software development, errors were regarded as inevitable. Programmers were urged to get software into execution quickly, and techniques for error removal were widely encouraged. The sooner the software could be written, the sooner debugging could begin. Programs were subjected to private unit testing and debugging, then integrated into components with more debugging, and finally into subsystems and systems with still more debugging. At each step, new interface and design errors were found, many the result of debugging in earlier steps. Product use by customers was simply another step in debugging, to correct errors discovered in the field. The most virulent errors were usually the result of fixes to other errors in development and maintenance [2]. It was not unusual for software products to reach a steady-state error population, with new errors introduced as fast as old ones were fixed. Today, debugging is understood to be the most error-prone process in software development, leading to "right in the small, wrong in the large" programs, and nightmares of integration where all parts are complete but do not work together because of deep interface and design errors.

In the Cleanroom process, correctness is built in by the development team through formal specification, design, and verification [3]. Team correctness verification takes the place of unit testing and debugging, and software enters system testing directly, with no execution by the development team. All errors are accounted for from first execution on, with no private debugging permitted. Experience shows that Cleanroom software typically enters system testing near zero defects and occasionally at zero defects.

The certification (test) team is not responsible for testing in quality, an impossible task, but rather for certifying the quality of software with respect to its specification. Certification is carried out by statistical usage testing that produces objective assessments of product quality. Errors, if any, found in

testing are returned to the development team for correction. If quality is not acceptable, the software is removed from testing and returned to the development team for rework and reverification.

The process of Cleanroom development and certification is carried out incrementally. Integration is continuous, and system functionality grows with the addition of successive increments. When the final increment is complete, the system is complete. Because at each stage the harmonious operation of future increments at the next level of refinement is predefined by increments already in execution, interface and design errors are rare.

The Cleanroom process is being successfully applied in IBM and other organizations. The technology requires some training and practice, but builds on existing skills and software engineering practices. It is readily applied to both new system development and re-engineering and extension of existing systems. The IBM Cleanroom Software Technology Center (CSTC) [4] provides technology transfer support to Cleanroom teams through education and consultation.

Cleanroom quality results

Table 1 summarizes quality results from Cleanroom projects. Earlier results are reported in [5]. The projects report a "certification testing failure rate;" for example, the rate for the IBM Flight Control project was 2.3 errors per KLOC, and for the IBM COBOL Structuring Facility project, 3.4 errors per KLOC. These numbers represent all errors found in all testing, measured from first-ever execution through test completion. That is, the rates represent residual errors present in the software following correctness verification by development teams.

The projects in Table 1 produced over a half a million lines of Cleanroom code with a range of 0 to 5.1 errors per KLOC for an average of 3.3 errors per KLOC found in all testing, a remarkable quality achievement indeed.

Traditionally developed software does not undergo correctness verification. It goes from development to unit testing and debugging, then more debugging in function and system testing. At entry to unit testing, traditional software typically exhibits 30-50 errors/KLOC. Traditional projects often report errors beginning with function testing (or later), omitting errors found in private unit testing. A traditional project experiencing, say, five errors/KLOC in function testing may have encountered 25 or more errors per KLOC when measured from first execution in unit testing. Quality comparisons between traditional and Cleanroom software are meaningful when measured from first execution.

Experience has shown that there is a qualitative difference in the complexity of errors found in Cleanroom and traditional code. Errors left behind by Cleanroom correctness verification, if any, tend to be simple mistakes easily found and fixed by statistical testing, not deep design or interface errors. Cleanroom errors are not only infrequent, but usually simple as well.

Highlights of Cleanroom projects reported in Table 1 are described below:

IBM Flight Control. A HH60 helicopter avionics component was developed on schedule in three increments comprising 33 KLOC of JOVIAL [6]. A total of 79 corrections were required during statistical certification for an error rate of 2.3 errors per KLOC for verified software with no prior execution or debugging.

IBM COBOL Structuring Facility (COBOL/SF). COBOL/SF, IBM's first commercial Cleanroom product, was developed by a six-person team. The product automatically transforms unstructured COBOL programs into functionally equivalent structured form for improved understandability and maintenance. It makes use of proprietary graph-theoretic algorithms, and exhibits a level of complexity on the order of a COBOL compiler.

The current version of the 85 KLOC PL/I product required 52 KLOC of new code and 179 corrections during statistical certification of five increments, for a rate of 3.4 errors per KLOC [7]. Several major components completed certification with no errors found. In an early support program at a major aerospace corporation, six months of intensive use resulted in no functional equivalence errors ever found [8]. Productivity, including all specification, design, verification, certification, user publications, and management, averaged 740 LOC per person-month. Challenging schedules defined for competitive reasons were all met. A major benefit of Cleanroom products is dramatically reduced maintenance costs. COBOL/SF has required less than one person-year per year for all maintenance and customer support.

Table 1. Cleanroom Quality Results

Year	Technology	Project	Quality/Productivity
1987	Cleanroom Software Engineering	IBM Flight Control: Helicopter Avionics System Component 33 KLOC (Jovial)	• Certification testing failure rate: 2.3 errors/KLOC • Error-fix reduced 5x • Completed ahead of schedule
1988	Cleanroom Software Engineering	IBM COBOL Structuring Facility: Product for automatically restructuring COBOL programs 85 KLOC (PL/I)	• IBM's first Cleanroom product • Certification testing failure rate: 3.4 errors/KLOC • Productivity 740 LOC/PM • Deployment failures 0.2 errors/KLOC, all simple fixes
1989	Partial Cleanroom Software Engineering	NASA Satellite Control Project 1 40 KLOC (FORTRAN)	• Certification testing failure rate: 4.5 errors/KLOC • 50-percent improvement in quality • Productivity 780 LOC/PM • 80-percent improvement in productivity
1990	Cleanroom Software Engineering	University of Tennessee: Cleanroom tool 12 KLOC (Ada)	• Certification testing failure rate: 3.0 errors/KLOC
1990	Cleanroom Software Engineering	Martin Marietta: Automated documentation system 1.8 KLOC (FOXBASE)	• First compilation: no errors found • Certification testing failure rate: 0.0 errors/KLOC (no errors found)
1991	Cleanroom Software Engineering	IBM System Software First increment 0.6 KLOC (C)	• First compilation: no errors found • Certification testing failure rate: 0.0 errors/KLOC (no errors found)
1991	Partial Cleanroom Software Engineering	IBM System Product Three increments, total 107 KLOC (mixed languages)	• Testing failure rate: 2.6 errors/KLOC • Productivity 486 LOC/PM
1991	Cleanroom Software Engineering	IBM Language Product First increment 21.9 KLOC (PL/X)	• Testing failure rate: 2.1 errors/KLOC
1991	Partial Cleanroom Software Engineering	IBM Image Product Component 3.5 KLOC (C)	• First compilation: 5 syntax errors • Certification testing failure rate: 0.9 errors/KLOC
1992	Cleanroom Software Engineering	IBM Printer Application First increment 6.7 KLOC (C)	• Certification testing failure rate: 5.1 errors/KLOC
1992	Partial Cleanroom Software Engineering	IBM Knowledge Based System Application 17.8 KLOC (TIRS)	• Testing Failure Rate: 3.5 errors/KLOC
1992	Cleanroom Software Engineering	NASA Satellite Control Projects 2 and 3 170 KLOC (FORTRAN)	• Testing Failure Rate: 4.2 errors/KLOC
1993	Cleanroom Software Engineering	IBM Device Controller First increment 39.9 KLOC (C)	• Certification testing Failure Rate: 1.8 errors/KLOC
1993	Partial Cleanroom Software Engineering	IBM Database Transaction Processor First increment 8.5 KLOC (JOVIAL)	• Testing Failure Rate: 1.8 errors/KLOC • No design errors, all simple fixes
1993	Partial Cleanroom Software Engineering	IBM LAN Software First increment 4.8 KLOC (C)	• Testing Failure Rate: 0.8 errors/KLOC

Note: All testing failure rates are measured from first-ever execution where correctness verification has taken the place of unit testing and debugging.

KEY: KLOC = thousand lines of code
PM = person month

NASA Satellite Control Project 1. The Coarse/Fine Attitude Determination System (CFADS) of the NASA Attitude Ground Support System (AGSS) was the first Cleanroom project carried out by the Software Engineering Laboratory (SEL) of the NASA Goddard Space Flight Center [9]. The system, comprised of 40 KLOC of FORTRAN, exhibited a certification failure rate of 4.5 errors per KLOC. Productivity was 780 LOC per person-month, an 80% improvement over previous SEL averages. Some 60% of the programs compiled correctly on the first attempt.

Martin Marietta Automated Documentation System. A four-person Cleanroom team developed the prototype of the Automated Production Control Documentation System, a relational data base application of 1820 lines programmed in FOXBASE. No compilation errors were found, and no failures were encountered in statistical testing and quality certification. The software was certified at target

levels of reliability and confidence. Team members attributed error-free compilation and failure-free testing to the rigor of the Cleanroom methodology [10].

IBM System Software. A four-person Cleanroom team developed the first increment of a system software product in C. The increment of 0.6 KLOC compiled with no errors, and underwent certification through 130 statistical tests with no errors found. Subsequent use in another environment resulted in one specification change.

IBM System Product. A Cleanroom organization of 50 people developed a complex system software product. The system, written in PL/I, C, REXX, and TIRS, was developed in three increments totaling 107 KLOC, with an average of 2.6 errors/KLOC found in testing [11]. Causal analysis of errors in the first increment revealed that five of its eight components experienced no errors whatsoever in testing. The project reported development team productivity of 486 LOC per person-month.

IBM Language Product. A seven-person Cleanroom team developed an extension to a language product. The first increment of 21.9 KLOC was up and cycling in less than half the time normally required, and exhibited a certification error rate of 2.1 errors/KLOC in testing.

IBM Image Product Component. A 3.5 KLOC image product component was developed to compress and decompress data from a Joint Photographic Expert Group (JPEG) data stream. The component exhibited three errors in testing, all simple mistakes. No additional errors have been found in subsequent use.

IBM Printer Application. An eleven-member team developed the first increment of a graphics layout editor in C under OS/2 Presentation Manager. The editor operates in a complex environment of vendor-developed code that exports more than 1000 functions, and uses many of the 800 functions of OS/2 PM. The first increment of 6.7 KLOC exhibited a rate of 5.1 errors/KLOC in testing [12]. All but 1.9 errors/KLOC were attributed to the vendor code interface and PM and C misunderstandings.

IBM Knowledge Based System Application. A five-person team developed a prototype knowledge-based system for the FAA Air Traffic Control System. The team reported a total of 63 errors for the 17.8 KLOC application, for a rate of 3.5 errors/KLOC. The fact that Cleanroom errors tend to be simple mistakes was borne out by project experience; only two of the 63 errors were classified as severe, and only five required design changes. The team developed a special design language for knowledge-based applications, together with proof rules for correctness verification.

NASA Satellite Control Projects 2 and 3. A 20 KLOC attitude determination subsystem of the Solar, Anomalous, and Magnetospheric Particle Explorer satellite flight dynamics system was the second Cleanroom project carried out by the Software Engineering Laboratory of the NASA Goddard Space Flight Center. The third project was a 150 KLOC flight dynamics system for the ISTP Wind/Polar satellite. These projects reported a combined error rate of 4.2 errors/KLOC in testing [13].

IBM Device Controller. A five-person team developed two increments of device controller design and microcode in 40 KLOC of C, including 30.5 KLOC of function definitions. Box structure specification of chip set semantics revealed a number of hardware errors prior to any execution. The multiple processor, bus architecture device processes multiple real-time input and output data streams. The project reported a failure rate of 1.8 errors/KLOC in testing.

IBM Database Transaction Processor. A five-person team developed the first increment of a host-based database transaction processor in 8.5 KLOC of JOVIAL. Rigorous use of correctness verification resulted in a failure rate of 1.8 errors/KLOC in testing, with no design errors encountered. The team reported that correctness verification reviews were far more effective in detecting errors than were traditional inspections.

IBM LAN Software. A four-person team developed the first increment of a LAN-based object server in 4.8 KLOC of C, resulting in a failure rate of 0.8 errors/KLOC in testing. The team utilized a popular case tool for recording specifications and designs.

Cleanroom management by incremental development

Management planning and control in Cleanroom is based on developing and certifying a pipeline of software increments that accumulate to the final product. The increments are developed and certified by small, independent teams, with teams of teams for large projects. Determining the number and functional content of increments is an important task driven by requirements, schedule, and resources. Functional content should be defined such that increments accumulate to the final product for continual integration, execute in the system environment for statistical usage testing, and represent end-to-end user function for quality certification.

An incremental development of a miniature interactive application shown in Figure 1, together with corresponding development and certification pipelines. Each increment is handed off from development to certification pipelines in turn, and results in a new quality measurement in MTTF. Early increments that implement system architecture receive more cumulative testing than later increments that implement localized functions. In this way, major architectural and design decisions are validated prior to their elaboration at lower levels.

The time required for design and verification of increments varies with their size and complexity, and careful planning and allocation of resources is required to deliver successive increments to certification on schedule. Long-lead-time increments may require parallel development.

Figure 2 illustrates the Cleanroom life cycle of incremental development and certification. The functional specification is created by the development team, or by a separate specification team for large projects, and the usage specification is created by the certification team. Based on these specifications, a joint planning process defines the initial incremental development and certification plan for the product. The development team then carries out a design and verification cycle for each increment, based on the functional specification, with corresponding statistical test case preparation by the certification team based on the usage specification. Completed increments are periodically delivered to certification for statistical testing and computation of MTTF estimates and other statistical measures. Errors are returned to the development team for correction. If the quality is low, improvements in the development process are initiated. As with any process, a good deal of iteration and feedback is always present to accommodate problems and solutions.

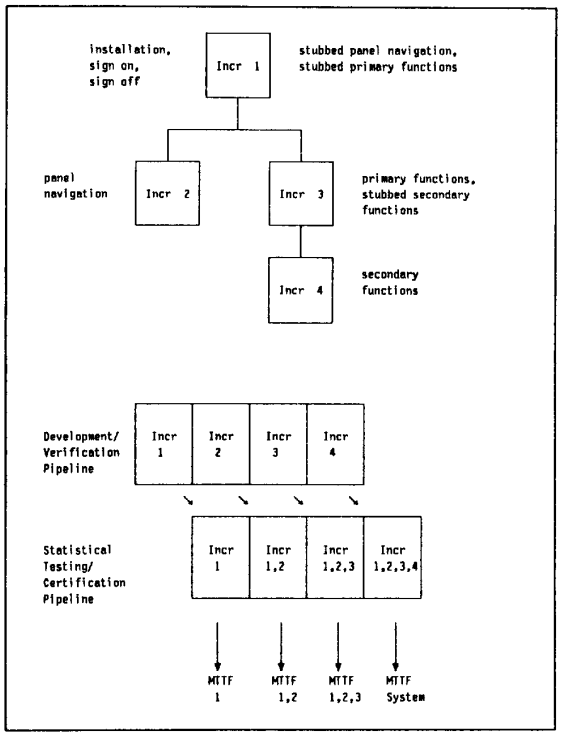

Figure 1. A Miniature Incremental Development

The Cleanroom incremental development life cycle is intended to be "quick and clean," not "quick and dirty" [14]. The idea is to quickly develop the right product with high quality for the user, then go on to the next version to incorporate new requirements arising from user experience.

Experienced Cleanroom teams with sufficient knowledge of subject matter and processing environment can achieve substantially reduced product development cycles. The precision of Cleanroom development eliminates rework and results in dramatically reduced time for certification testing compared to traditional methods. And Cleanroom teams are not hostage to error correction following product release.

Cleanroom affords a new level of manageability and control in adapting to changing requirements. Because formally engineered software is well-documented and under good intellectual control throughout development, the impact of new requirements can be accurately assessed, and changes can be planned and accommodated in a systematic manner.

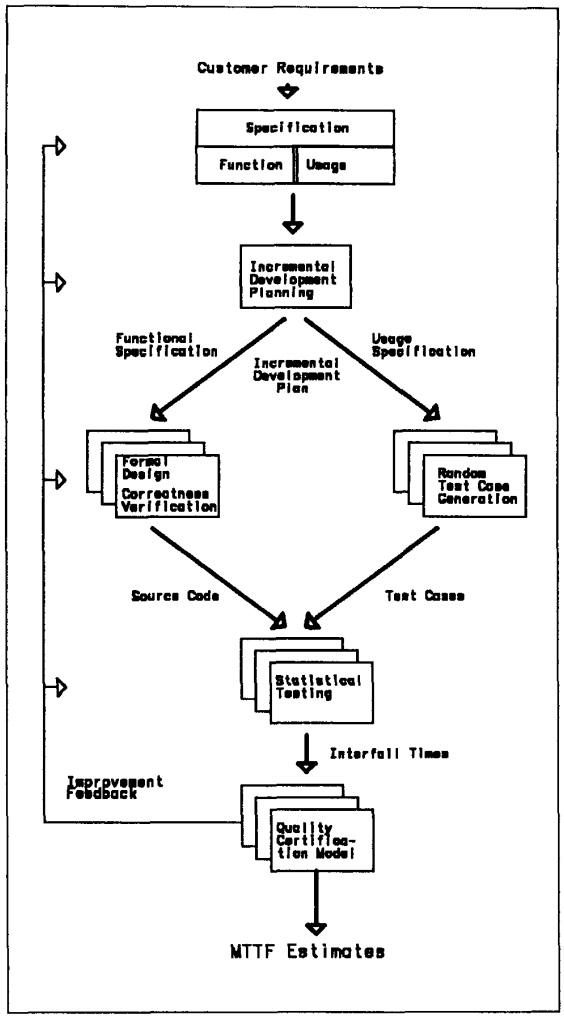

Figure 2. The Cleanroom Life Cycle

Incremental development provides a framework for replanning schedules, resources, and functional content, and permits changes to be incorporated and packaged in a stepwise fashion.

Cleanroom software specification

Cleanroom development begins with a specification of required system behavior and architecture. The object-based technology of box structures is an effective specification technique for Cleanroom development [15, 16]. Box structures provide a stepwise refinement and verification process in black box, state box, and clear box forms for defining required system behavior and deriving and connecting objects comprising a system architecture [17, 18].

Without a rigorous specification technology, there was little incentive in the past to devote much effort to the specification process. Specifications were frequently written in natural language, with inevitable ambiguities and omissions, and often regarded as throwaway stepping stones to the code. Box structures, however, provide an economic incentive for precise specification. Initial box structure specifications often reveal gaps and misunderstandings in user requirements that would ordinarily be discovered later in development at high cost and risk to the project.

There are two engineering problems associated with system specification, namely, defining the right function for users, and defining the right structure for the specification itself. Box structures address the first problem by precisely defining current understandings of required function at each stage of development for informed review and modification.

The second problem deals with scale-up in complex specifications, namely, how to organize myriad details of behavior and processing into coherent abstractions for human understanding. Box structures incorporate the crucial mathematical property of referential transparency, such that the information content of an abstraction, say a black box, is sufficient to define its refinement to state box and clear box forms without reference to other specification parts. This property permits specifications of large systems to be hierarchically organized, with no loss of precision at high levels or of details at low levels.

Three fundamental principles underlie the box structure design process [17]:

1. All data to be defined and retained in a design are encapsulated in boxes (objects, data abstractions).

2. All processing is defined by sequential and concurrent uses of boxes.

3. Each use of a box in a system occupies a distinct place in the usage hierarchy of the system.

Each box can be defined in the three forms of black, state, and clear box, with identical external behavior but increasing internal detail. These forms isolate and focus on successive creative definitions of external behavior, retained data, and processing, respectively, as follows.

The black box of an object is a precise specification of external, user-visible behavior in all possible circumstances of use. The object may be an entire system or system part of any size. The user may be a person or another object. A black box accepts a stimulus (S) from a user and produces a response (R) before the next stimulus is processed. Each response of a black box is determined by its current stimulus history (SH), with black box transition function

$$(S, SH \rightarrow R).$$

Any software system or system part exhibits black box behavior in that its next response is determined by the history of stimuli it has received. In simple illustration, imagine a hand calculator and two stimulus histories

Clear 7 1 3 and Clear 7 1 3 +

Given a next stimulus of 6, the two histories produce a responses of

7136 and 6

respectively. That is, a given stimulus will produce different responses based on history of use, not just on current stimulus.

The objective of a black box specification is to define required behavior in all possible circumstances of use, that is, the responses produced for any possible stimulus and stimulus history. Such specifications include erroneous and unexpected stimuli, as well as correct usage scenarios. By defining behavior solely in terms of stimulus histories, black box specifications do not depend on, or prematurely define, design internals.

Black box specifications are often recorded in tabular form; in each row, the stimulus and condition on stimulus history are sufficient to define the required response. Scale up to large specifications is achieved by identifying classes of behavior for nesting tables, and through use of specification functions [19] to encapsulate conditions on stimulus histories.

The state box of an object is derived from its black box by identifying those elements of stimulus history that must be retained as state data between transitions to achieve required black box behavior. The transition function of a state box is

$$(S, OS) \rightarrow (R, NS),$$

where OS and NS represent old state and new state, respectively. While the external behavior of a state box is identical to its corresponding black box, the stimulus history is replaced by reference to old state and generation of new state as required by each transition.

State boxes correspond closely to the traditional view of objects as encapsulations of state data and services, or methods, on that data. In this view, stimuli and responses are inputs and outputs, respectively, of specific service invocations.

The clear box of an object is derived from its state box by defining a procedure to carry out the state box transition function. The transition function of a clear box is thus

$$(S, OS) \rightarrow (R, NS) \text{ by procedure.}$$

A clear box is simply a program that implements the corresponding state box. Clear box forms include sequence, alternation, iteration, and concurrent structures [15]. A clear box may invoke black boxes at the next level for independent refinement. That is, the process is recursive, with each clear box possibly introducing opportunities for definition of new, or extensions to existing, objects in black box, state box, and clear box forms.

Through this stepwise refinement process, box structure specifications evolve as usage hierarchies of objects wherein the services of a given object may be used and reused in many places at many levels as required. Clear boxes play a crucial role in the hierarchy by ensuring the harmonious cooperation of objects at the next level of refinement. Appropriate objects and their clear box connections are derived out of immediate processing needs at each stage of refinement, not invented a priori with connections left to later invention.

Box structures bring correctness verification to object architectures. State boxes can be verified with respect to their black boxes, and clear boxes verified with respect to their state boxes. [15].

Cleanroom software design and verification

Design and verification of clear box procedures is based on functional and algebraic properties of their constituent control structures. The control structures of structured programming used in clear box design, namely, sequence, ifthenelse, whiledo, etc., are single-entry, single-exit structures with no side effects in control flow possible. In execution, a control structure simply transforms data from an input state to an output state. This transformation, known as a **program function**, corresponds to a mathematical function, that is, it defines a mapping from a domain to a range by a particular rule. Program functions can be derived from control structures. For example, for integers x, y, and z, the program function of the sequence,

```
DO
   z := abs(y)
   w := max(x, z)
OD
```

is, in concurrent assignment form,

```
w, z := max(x, abs(y)), abs(y)
```

and for integer x > = 0, the program function of the iteration,

```
WHILE
   x > 1
DO
   x := x - 2
OD
```

is, in English,

```
set odd x to 1, even x to 0
```

In stepwise refinement of clear box procedures, an **intended function** is defined and then refined into a control structure and new intended functions for refinement, as illustrated in the miniature example of Figure 3. Intended functions are recorded in the design, delimited by square brackets and attached to their refinements. **Design Simplification** is an important objective in refinement, to arrive at compact and straightforward designs for verification. The correctness of each refinement is determined by deriving its program function, that is, the function it actually computes, and comparing it to the intended function.

A Correctness Theorem [20] defines how to make the comparison of intended functions and program functions in terms of **correctness conditions** to be verified for each control structure. The correctness

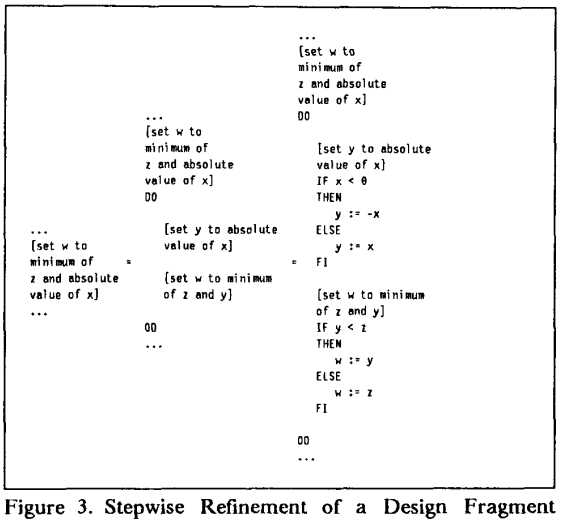

Figure 3. Stepwise Refinement of a Design Fragment with Intended Functions for Verification

conditions make use of function composition for sequence, case analysis for alternation, and function composition and case analysis in a recursive equation

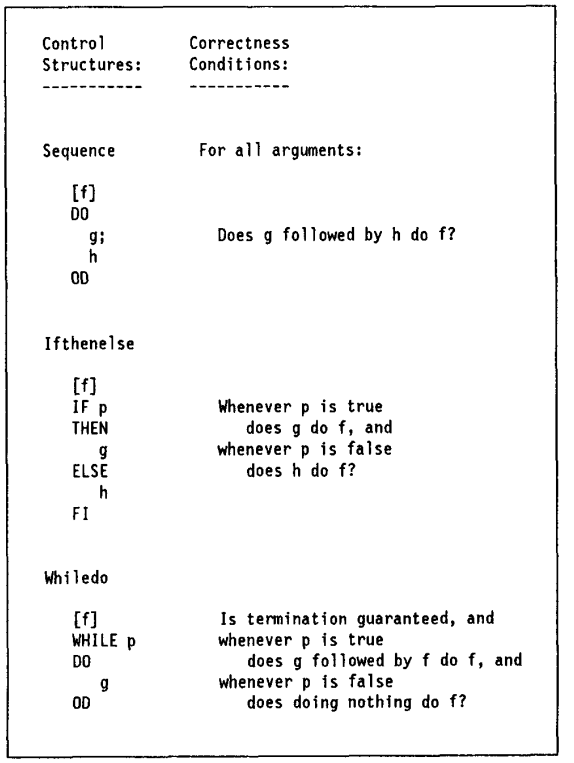

Figure 4. Correctness Theorem Correctness Conditions in Question Form

for iteration. For sequence, one condition must be checked, for alternation, two conditions, and for iteration, three conditions, as shown in Figure 4. The conditions are language and subject matter independent.

The nested and sequenced control structures of a clear box define a natural decomposition hierarchy that enumerates the independent subproofs required, one for each control structure. An **Axiom of Replacement** permits algebraic substitution of intended functions and their control structures in the hierarchy of subproofs. This substitution permits proof arguments to be localized to the control structure at hand, and, in fact, the proofs for each control structure can be carried out in any order. A miniature program and its required subproofs are shown in Figure 5.

In essence, clear boxes are composed of a finite number of control structures, each of which is verified by checking a finite number of correctness conditions. Even though all but the most trivial programs exhibit an essentially infinite number of execution paths, their verification can be carried out in a finite number of steps. For example, the clear box of Figure 6 requires verification of exactly 15 correctness conditions.

The value to software quality of the reduction of verification to a finite process cannot be overemphasized. It permits Cleanroom development teams to verify every line of design and code through **mental proofs of correctness** in team reviews. Written proofs are also possible for extra confidence, for example, in verification of life- or mission-critical software.

In team reviews, every correctness condition of every control structure is verified in turn. Every team member must agree that each condition is correct. An error is possible only if every team member incorrectly verifies a particular correctness condition. The requirement for unanimous agreement based on individual verifications results in software at or near zero defects prior to first execution.

Function-theoretic verification scales up to large systems. Every structured system, no matter how large, has top level programs composed of familiar sequence, alternation, and iteration structures, which typically invoke large-scale subsystems at the next level involving thousands of lines of code (each of which has its own top level programs). The correctness conditions for these structures are scale-free, that is, they are invariant with respect to the size and complexity of the operations involved. Verification

```
Program:                Subproofs:
---------               ----------

 [f1]                   f1 = [DO g1;g2;[f2] OD] ?
 DO
    g1
    g2
    [f2]                f2 = [WHILE p1 DO [f3] OD] ?
    WHILE
       p1
    DO [f3]             f3 = [DO g3;[f4];g8 OD] ?
       g3
       [f4]             f4 = [IF p2 THEN [f5] ELSE [f6] FI] ?
       IF
          p2
       THEN [f5]        f5 = [DO g4;g5 OD] ?
          g4
          g5
       ELSE [f6]        f6 = [DO g6;g7 OD] ?
          g6
          g7
       FI
       g8
    OD
 OD
```

Figure 5. A Program and its Constituent Subproofs

at high levels may take, and well be worth, more time, but it does not take more theory.

Correctness verification produces quality results superior to unit testing and debugging. For each program part, function-theoretic correctness conditions permit verification of all possible effects on data. Unit testing, however, checks only effects of particular test paths selected out of many possible paths. A program or program part may have many paths to test, but only one function to verify.

In addition, verification is more efficient than unit testing. Most verification conditions can be checked in a few seconds in team reviews, but unit tests take substantial time to prepare, execute, and check.

Cleanroom software quality certification

Techniques and benefits of statistical quality control in hardware development are well known. In cases where populations of items are too large to permit exhaustive testing, statistical sampling and analysis methods are employed to obtain scientific assessments of quality.

In simple illustration, the process of statistical quality control in manufacturing is to 1) sample the population of items on a production line, 2) measure the quality of the sample with respect to a design assumed to be perfect, 3) extrapolate the sample quality to the population of items and 4) if the quality is inadequate, identify and correct flaws in production. In applying statistical quality control to hardware products, the statistics lie in the variation

of physical properties of items in the population. But in the case of software products, all copies are identical, bit for bit, so where are the statistics?

It turns out that software has a statistical property of great interest to developers and users, namely, its execution behavior. That is, how long on average will a software product execute before it fails, say by abending, producing incorrect output, etc.? Thus, the process of statistical quality control in software is to 1) sample the essentially infinite population of possible user executions of a product based on the frequency of expected usage, 2) measure the quality of the sample by determining if the executions are correct, 3) extrapolate the quality of the sample to the population of possible executions, and 4) if the quality is inadequate, identify and correct flaws in the development process, for example, improvements to inadequate correctness verification.

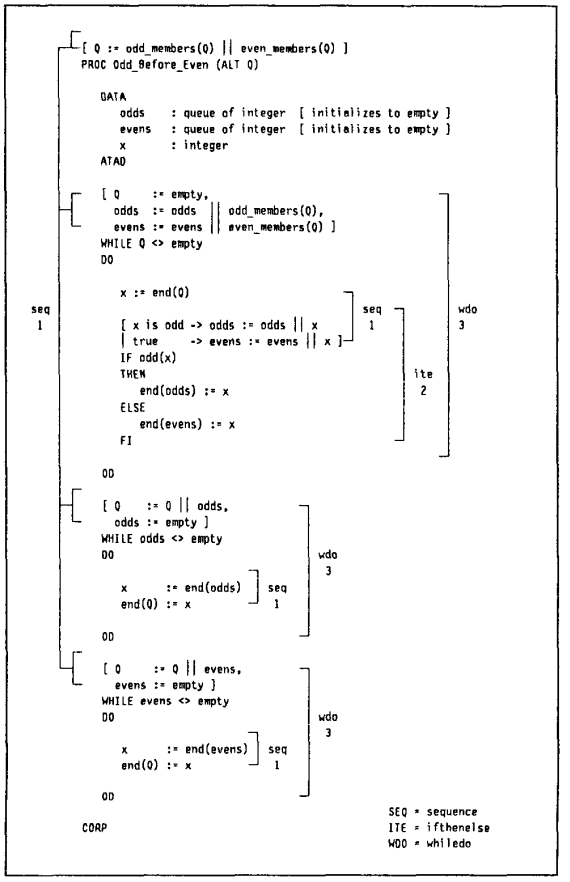

Figure 6. A Procedure with 15 Correctness Conditions

This process, known as **statistical usage testing** [6], amounts to testing software the way users intend to use it. The entire focus of statistical testing is on external system behavior, not internals of design and implementation as in conventional coverage testing. Cleanroom certification teams have deep knowledge of expected usage, but no knowledge of design internals.

As noted, the role of a Cleanroom certification team is not to debug software, but rather to certify its quality through statistical testing techniques. The certification may show adequate quality, but if not, the software will be returned to the development team for rework.

In practice, Cleanroom quality certification is carried out in three steps, as follows:

Step 1: Specify usage probability distributions. Usage probability distributions are models of intended usage of a software product. They define all possible usage patterns and scenarios, including erroneous and unexpected usage, together with their probabilities of occurrence. Usage probability distributions represent the virtually infinite population of possible executions of a software product, together with their expected frequencies of use.

Distributions are defined by the certification team based on box structure specifications of system function, plus information on system usage probabilities obtained from prospective users, actual usage of prior versions, etc. Formal grammars permit compact representations of distributions for analysis and review.

Step 2: Randomize test cases against usage probability distributions. Test cases are derived from the distributions, such that every test represents actual usage and will effectively rehearse user experience with the product. Because the test cases are completely prescribed by the distributions, test case production is a mechanical, and automatable, process.

In miniature illustration, Figure 7 depicts a usage specification and corresponding test case generation for a program with four user stimuli to Update (U), Delete (D), Query (Q), and Print (P). A usage distribution, simplified for illustration, shows projected probabilities of use of 32, 14, 46, and 8 percent for the four stimuli, respectively (omitting scenarios of use, etc., for simplicity). These probabilities are mapped onto an interval of 0 to 99, dividing it into four partitions proportional to the probabilities. Assuming a test case contains six stimuli, each test is

Usage Probabilities:

Program Stimuli	Usage Possibility Distribution	Distribution Interval
U (update)	32%	0 - 31
D (delete)	14%	32 - 45
Q (query)	46%	46 - 91
P (print)	8%	92 - 99

Test Case Generation:

Test Number	Random Numbers	Test Case
1	29 11 47 52 26 94	U U Q Q U P
2	62 88 39 78 82 65	Q U D Q Q Q
3	03 32 58 41 36 17	U D Q D D U
4	36 40 86 02 20 77	D D Q U U Q
...	...	...

Figure 7. Simplified Usage Probability Distribution and Statistical Test Case Generation for a Program with Four User Stimuli

generated by obtaining six two-digit random numbers, determining the partitions within which they reside, and appending the corresponding stimuli (U, D, Q, or P) to the test case. In this way, each test case is faithful to the distribution and represents a possible user execution of the system.

Step 3: Execute random test cases, assess success or failure, and compute quality measures. Each test case is executed and its results are verified against system specifications. Time in execution up to correct completion or failure is recorded in appropriate units, for example, CPU time, wall clock time, number of transactions, etc. In effect, these times, known as **interfail times**, represent the quality of the sample of possible user executions. Interfail times accumulated in testing are processed by a **quality certification model** [6] that computes product Mean Time To Failure (MTTF) and other measures of quality. Figure 8 depicts graphs produced by the certification model. The X and Y axes plot errors fixed and computed MTTF, respectively. The curve for high-quality software shows exponential improvement, such that the MTTF quickly exceeds the total test time, whereas the curve for low-quality software shows little MTTF growth.

Because statistical usage testing embeds the software development process in a **formal statistical design**, MTTF measures provide a scientific basis for management action, unlike the anecdotal evidence of

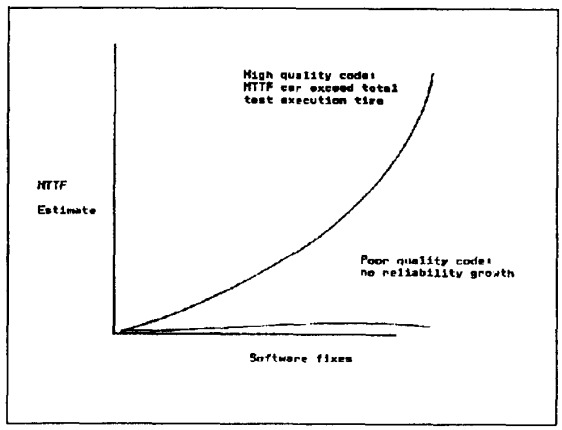

Figure 8. Two Sample MTTF Graphs Produced by the Cleanroom Certification Model

quality characteristic of coverage testing (if many, or few, errors are found, is that good or bad?).

In incremental development, a usage probability distribution can be stratified into subsets that exercise increasing functional content as increments are added, with the full distribution in effect once the final increment is in place. In addition, **alternate distributions** can be defined to permit independent certification of infrequently used system functions (with low probability in primary distributions) that carry high consequences of failure, for example, code for emergency shutdown of a nuclear reactor.

But there is more to the story of statistical usage testing. Extensive analysis of errors in large-scale software systems reveals a spread in the failure rates of errors of some four orders of magnitude [2]. Virulent, high-rate errors can literally occur every few hours for some user, but low-rate errors may show up only after decades of use. High-rate errors have a profound effect on product quality, but they comprise only a small fraction of total errors. In fact, this small fraction (under 3%) is responsible for nearly two-thirds of the software failures reported [5].

Because statistical usage testing amounts to testing software the way users will use it, errors tend to be found in failure-rate order on average, that is, any remaining virulent, high-rate errors tend to be found first. As a result, errors left behind, if any, at completion of testing tend to be low-rate errors that are infrequently encountered by users.

Traditional coverage testing does not find errors in failure-rate order, but rather, in random order. On any given coverage path, an error will either be

found or not. If found, an error may be low rate, high rate or in between. That is, coverage testing is not biased to find errors in any particular rate order. Finding and fixing low-rate errors has little effect on MTTF and the user perception of quality. But finding and fixing errors in failure-rate order has dramatic effect, with each correction resulting in substantial improvement in MTTF. In fact, statistical usage testing is more than 20 times more effective at extending MTTF than is coverage testing [5].

Acknowledgements

The author wishes to thank Kim Hathaway for her contributions and assistance in developing this paper. Suggestions by Michael Deck, Philip Hausler, Harlan Mills, Mark Pleszkoch, and Alan Spangler were appreciated. Special acknowledgement is due to the members of the Cleanroom teams whose quality results are reported in this paper, and who are setting new standards of professional excellence in software development.

References

1. Mills, H. D., "Certifying the Correctness of Software," *Proc. 25th Hawaii International Conference on System Sciences*, IEEE Computer Society Press, January, 1992, pp. 373-381.

2. Adams, E. N., "Optimizing Preventive Service of Software Products," *IBM Journal of Research and Development*, January, 1984.

3. Mills, H. D., M. Dyer, and R. C. Linger, "Cleanroom Software Engineering," *IEEE Software*, September, 1987, pp. 19-25.

4. Linger, R. C. and R. A. Spangler, "The IBM Cleanroom Software Engineering Technology Transfer Program," *Proc. SEI Software Engineering Education Conference*, IEEE Computer Society Press, San Diego, CA, October 5-7, 1992.

5. Cobb, R. H. and H. D. Mills, "Engineering Software Under Statistical Quality Control," *IEEE Software*, November, 1990, pp. 44-54.

6. Curritt, P. A., M. Dyer, and H. D. Mills, "Certifying the Reliability of Software," *IEEE Trans. on Software Engineering*, Vol. SE-12, No. 1, January, 1986, pp. 3-11.

7. Linger, R. C. and H. D. Mills, "A Case Study in Cleanroom Software Engineering: The IBM COBOL Structuring Facility," *Proc. 12th International Computer Science and Applications Conference*, IEEE Computer Society Press, October, 1988.

8. *A Success Story at Pratt & Whitney: On Track for the Future with IBM's VS COBOL II and COBOL Structuring Facility*, publication GK20-2326, IBM Corporation, White Plains, NY.

9. Kouchakdjian, A., S. Green, and V. R. Basili, "Evaluation of the Cleanroom Methodology in the Software Engineering Laboratory." *Proc. Fourteenth Annual Software Engineering Workshop*, NASA Goddard Space Flight Center, Greenbelt, MD, November 1989.

10. Trammell, C. J., L. H. Binder, and C. E. Snyder, "The Automated Production Control System: A Case Study in Cleanroom Software Engineering," *ACM Transactions on Software Engineering and Methodology*, Vol. 1, No. 1, January 1992, pp. 81-94.

11. Hausler, Philip A., "A Recent Cleanroom Success Story: The Redwing Project," *Proc. Seventeenth Annual Software Engineering Workshop*, NASA Goddard Space Flight Center, Greenbelt, MD, December 1992.

12. Deck, M.D., P. Hausler, and R.C. Linger, "Recent Experiences with Cleanroom Software Engineering," *Proc. 1992 IBM Software Development Conference*, Toronto, Canada, 1992

13. Green, S.E. and Rose Pajerski, "Cleanroom Process Evolution in the SEL," *Proc. Sixteenth Annual Software Engineering Workshop*, NASA Goddard Space Flight Center, Greenbelt, MD, December, 1991.

14. Mills, H. D., Private communication.

15. Mills, H. D., R. C. Linger, and A. R. Hevner, *Principles of Information Systems Analysis and Design*, Academic Press, San Diego, CA, 1986.

16. Mills, H. D., R. C. Linger, and A. R. Hevner, "Box Structured Information Systems," *IBM Systems Journal*, Vol. 26, No. 4, 1987, pp. 393-413.

17. Mills, H. D., "Stepwise Refinement and Verification in Box-Structured Systems," *IEEE Computer*, June, 1988.

18. Hevner, A. R. and H. D. Mills, "Box Structured Methods for Systems Development with Objects," *IBM Systems Journal* (to appear).

19. Pleszkoch, M. G., P. A. Hausler, A. R. Hevner, and R. C. Linger, "Function-Theoretic Principles of Program Understanding," *Proc. 23rd Hawaii International Conference on System Sciences*, IEEE Computer Society Press, January, 1990, pp. 74-81.

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

21. Poore, J. H. and H. D. Mills, "Bringing Software Under Statistical Quality Control," *Quality Progress*, November 1988.