CLEANROOM PROCESS MODEL

The philosophy behind Cleanroom software engineering is to avoid dependence on costly defect-removal processes by writing code increments right the first time and verifying their correctness before testing. Its process model incorporates the statistical quality certification of code increments as they accumulate into a system.

RICHARD C. LINGER
IBM Cleanroom Software Technology Center

Today's competitive pressures and society's increasing dependence on software have led to a new focus on development processes. The Cleanroom process, which has evolved over the last decade, has demonstrated that it can improve both the productivity of developers who use it and the quality of the software they produce.

Cleanroom software engineering is a team-oriented process that makes development more manageable and predictable because it is done under statistical quality control.

Cleanroom is a modern approach to software development. In traditional, craft-based development, defects are regarded as inevitable and elaborate defect-removal techniques are a part of the development process. In such a process, software proceeds from development to unit testing and debugging, then to function and system testing for more debugging. In the absence of workable alternatives, managers encourage programmers to get code into execution quickly, so debugging can begin. Today, developers recognize that defect removal is an error-prone, inefficient activity that consumes resources better allocated to getting the code right the first time.

This article is based on a paper that appears in *Proc. 15th Int'l Conf. Software Eng.*, IEEE CS Press, Los Alamitos, Calif., 1993, pp. 2-13. The *IEEE Software* Editorial Board has selected it as the best practice paper presented at ICSE-15. The board thanks Richard A. DeMillo, ICSE-15 Program Cochair, for his help in arranging for its publication in *IEEE Software*.

Cleanroom teams at IBM and other organizations are achieving remarkable quality results in both new-system development and modifications and extensions to legacy systems. The quality of software produced by Cleanroom development teams is sufficient (often near zero defects) for the software to enter system testing directly for first-ever execution by test teams.

The theoretical foundations of Cleanroom — formal specification and design, correctness verification, and statistical testing — have been reduced to practice and demonstrated in nearly a million lines of code. Some Cleanroom projects are profiled in the box on p. 56.

QUALITY COMPARISON

Quality comparisons between traditional methods and the Cleanroom process are meaningful when measured from first execution. Most traditional development methods begin to measure errors at function testing (or later), omitting errors found in private unit testing. A traditional project experiencing, say, five errors per thousand lines of code (KLOC) in function testing may have encountered 25 or more errors per KLOC when measured from first execution in unit testing.

At entry to unit testing, traditional software typically exhibits 25 to 35 or more errors per KLOC.[1] In contrast, the weighted average of errors found in 17 Cleanroom projects, involving nearly a million lines of code, is 2.3 errors per KLOC. This number represents all errors found in all testing, measured from first-ever execution through test completion — it is the average number of residual errors present after the development team has performed correctness verification.

In addition to this remarkable difference in the number of errors, experience has shown a qualitative difference in the complexity of errors found in Cleanroom versus traditional software. Errors left behind by Cleanroom correctness verification tend not to be complex design or interface errors, but simple mistakes easily found and fixed by statistical testing.

In this article, I describe the Cleanroom development process, from specification and design through correctness verification and statistical usage testing for quality certification.

INCREMENTAL DEVELOPMENT

The Cleanroom process is based on developing and certifying a pipeline of software increments that accumulate into the final system. The increments are developed and certified by small, independent teams, with teams of teams for large projects.

System integration is continual, and functionality grows with the addition of successive increments. In this approach, the harmonious operation of future increments at the next level of refinement is predefined by increments already in execution, thereby minimizing interface and design errors and helping developers maintain intellectual control.

CLEANROOM DEVELOPMENT IS INTENDED TO BE "QUICK AND CLEAN," NOT "QUICK AND DIRTY."

The Cleanroom development process is intended to be "quick and clean," not "quick and dirty." 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.

In the Cleanroom process, correctness is built in by the development team through formal specification, design, and verification. 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.

Figure 1 illustrates the Cleanroom process of incremental development and quality certification. The Cleanroom team first analyzes and clarifies customer requirements, with substantial user interaction and feedback. If requirements are in doubt, the team can develop Cleanroom prototypes to elicit feedback iteratively.

As the figure shows, Cleanroom development involves two cooperating teams and five major activities:

♦ *Specification.* Cleanroom development begins with specification. Together, the development team and the certification team produce two specifications: functional and usage. Large projects may have a separate specification team.

The functional specification defines the required external system behavior in all circumstances of use; the usage specification defines usage scenarios and their probabilities for all possible system usage, both correct and incorrect. The functional specification is the basis for incremental software development. The usage specification is the basis for generating test cases for incremental statistical testing and quality certification. Usage specifications are explained in the section on certification.

♦ *Increment planning.* On the basis of these specifications, the development and certification teams together define an initial plan for developing increments that will accumulate into the final system. For example, a 100 KLOC system might be developed in five increments averaging 20 KLOC each. The time it takes to design and verify increments varies with their size and complexity. Increments that require long lead times may call for parallel development.

♦ *Design and verification.* The development team then carries out a design and correctness verification cycle for each increment. The certification team proceeds in parallel, using the usage specification to generate test cases that reflect the expected use of the accumulating increments.

♦ *Quality certification.* Periodically, the development team integrates a completed increment with prior increments and delivers them to the test team for execution of statistical test cases. The test cases are run against the accumulated increments and the results checked for correctness

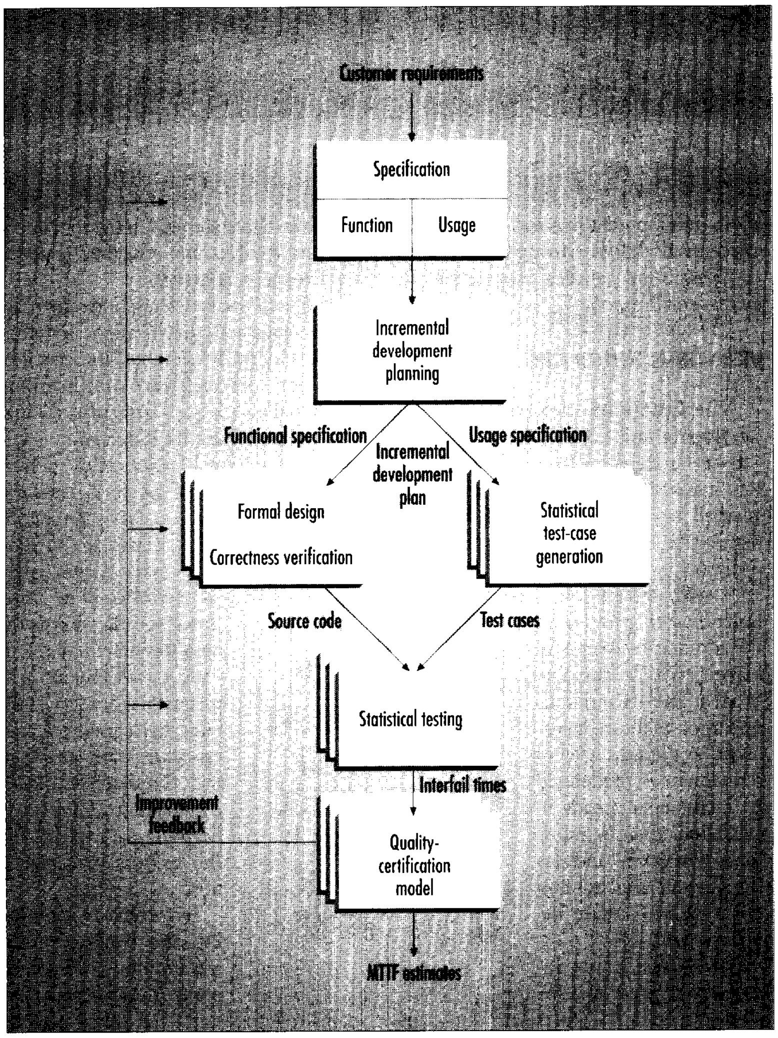


Figure 1. Cleanroom process model. The stacked boxes indicate successive increments.

against the functional specification. Interfail times, that is, the elapsed times between failures, are passed to a quality-certification model[2] that computes objective statistical measures of quality, such as mean time to failure. The quality-certification model employs a reliability growth estimator to derive the statistical measures.

Certification is done continuously, over the life of the project. Higher level increments enter the certification pipeline first. This means major architectural and design decisions are validated in execution before the development team elaborates on them. And because certification is done for all increments as they accumulate, higher level increments are subjected to more testing than lower level increments, which implement localized functions.

♦ *Feedback.* Errors are returned to the development team for correction. If the quality is low, managers and team members initiate process improvement. As with any process, a good deal of iteration and feedback is always present to accommodate problems and solutions.

In the next sections, I describe the specification, design and verification, and quality-certification procedures. A detailed description of increment planning and feedback mechanisms is outside the scope of this article.

FUNCTIONAL SPECIFICATION

The object-based technology of box structures has proved to be an effective technique for functional specification.[3] 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.[4-5]

In the past, without a rigorous specification technology, there was little incentive 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 code.

Box structures provide an economic incentive for precision. 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.

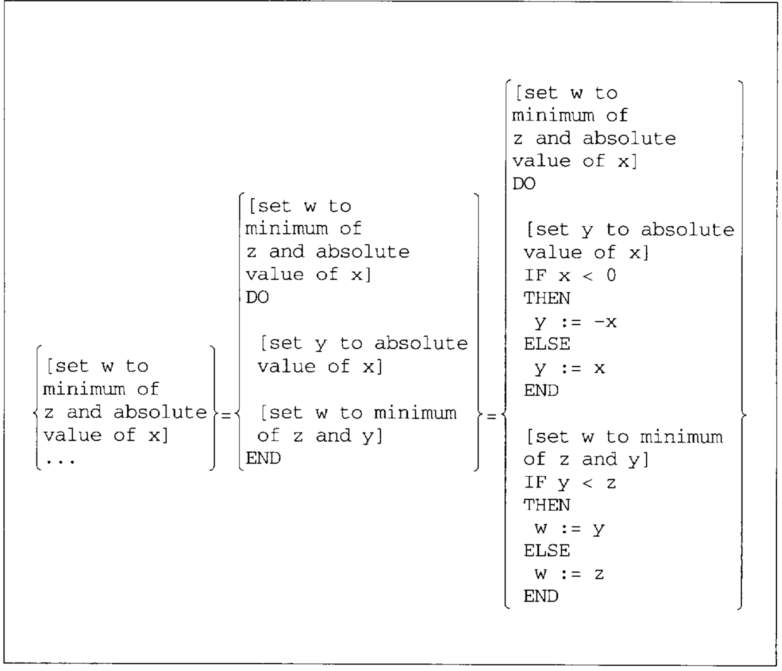
They also address the two engineering problems associated with system specification: defining the right function for users and defining the right structure for the specification itself. Box structures address the first problem by precisely defining the current understanding of required functions at each stage of development, so that the functions can be reviewed and modified if necessary. The second problem is critical, especially for large-system development. How can we organize the myriad details of behavior and processing into coherent abstractions humans can understand?

Box structures incorporate the crucial mathematical property of referential transparency — the information content of each box specification is sufficient to define its refinement, without depending on the implementation of any other box. This property lets us organize large-system specifications hierarchically, without sacrificing precision at high levels or detail at low levels.

Box structures. Three principles govern the use of box structures:[4]

♦ All data defined in a design is encapsulated in boxes.

♦ All processing is defined by using boxes sequentially or concurrently.



```
[set w to
minimum of
z and absolute
value of x]
...
```

=

```
[set w to
minimum of
z and absolute
value of x]
DO

  [set y to absolute
  value of x]

  [set w to minimum
  of z and y]
END
```

=

```
[set w to
minimum of
z and absolute
value of x]
DO

  [set y to absolute
  value of x]
  IF x < 0
  THEN
   y := -x
  ELSE
   y := x
  END

  [set w to minimum
  of z and y]
  IF y < z
  THEN
   w := y
  ELSE
   w := z
  END
```

Figure 2. *Stepwise refinement of a clear-box design fragment that can be verified. Each fragment has identical functional behavior, even though the level of detail increases.*

♦ Each box occupies a distinct place in a system's usage hierarchy.

Each box has three forms — black, state, and clear — which have identical external behavior but whose internals are increasingly detailed.

Black box. An object's black box is a precise specification of external, user-visible behavior in all possible circumstances of its use. The object may be an entire system or any part of a system. Its user may be a person or another object.

A black box accepts a stimulus (S) from a user and produces a response (R). Each response of a black box is determined by its current stimulus history (SH), with a black-box transition function

```
(S, SH) → (R)
```

A given stimulus will produce different responses that are based on history of use, not just on the current stimulus. Imagine a calculator with two stimulus histories

```
Clear 7 1 3
```

and

```
Clear 7 1 3 +
```

If the next stimulus is 6, the first history produces a response of 7136; the second, 6.

The objective of a black-box specification is to define the responses produced for every possible stimulus and stimulus history, including erroneous and unexpected stimuli. By defining behavior solely in terms of stimulus histories, black-box specifications neither depend on nor prematurely define design internals.

Black-box specifications are often recorded as tables. In each row, the stimulus and the condition on stimulus history are sufficient to define the required response. To record large specifications, classes of behavior are grouped in nested tables and compact specification functions are used to encapsulate conditions on stimulus histories.[6]

State box. An object's state box is derived from its black box by identifying the elements of stimulus history that must be retained as state data between transitions to achieve the required black-box behavior.

The transition function of a state box is

```
(S, OS) → (R, NS),
```

where OS and NS represent old state and new state. Although the external behavior of a state box is identical to its corresponding black box, the stimulus histories are replaced with references to an old state and the generation of a new state, as its transitions require.

As in the traditional view of objects, state boxes encapsulate state data and services (methods) on that data. In this view, stimuli and responses are inputs and outputs, respectively, of specific state-box service invocations that operate on state data.

Clear box. An object's clear box 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

```
(S, OS) → (R, NS) by procedure
```

So a clear box is simply a program that implements the corresponding state box. A clear box may invoke black boxes at the next level, so the refinement process is recursive, with each clear box possibly introducing opportunities for defining new objects or extensions to existing ones.

Clear boxes play a crucial role in the usage hierarchy by ensuring the harmonious cooperation of objects at the next level of refinement. Objects and their clear-box connections are derived from immediate processing needs at each stage of refinement, not invented a priori, with uncertain connections left to be defined later. The design and verification of clear-box procedures is the focus of the next section.

Because state boxes can be verified with respect to their black boxes and clear boxes with respect to their state boxes, box structures bring correctness verification to object architectures.[4]

DESIGN AND VERIFICATION

The procedural control structures of structured programming used in clear-box design — sequence, alternation (if-then-else), and iteration (while-do) — are single-entry, single-exit structures that cannot produce side effects in control flow. (Control structures for concurrent execution are dealt with in box structures, but are outside the scope of this article.)

When it executes, a given control structure simply transforms data from an input state to an output state. This transformation, known as its *program function*, corresponds to a mathematical function: It defines a mapping from a domain to a range by a particular rule.

For integers w, x, y, and z, for example, the program function of the sequence,

Sequence

Control structure:	Correctness condition (for all arguments):

`[f]`
`DO`
`  g;`
`  h`
`END`
	Does *g* followed by *h* do *f*?

Alternation

Control structure:	Correctness condition:

`[f]`
`IF p`
`THEN`
`  g`
`ELSE`
`  h`
`END`
	Whenever *p* is true does *g* do *f*, and whenever *p* is false does *h* do *f*?

Iteration

Control structure:	Correctness condition:

`[f]`
`WHILE p`
`DO`
`  g`
`END`
	Is termination guaranteed, and whenever *p* is true does *g* followed by *f* do *f*, and whenever *p* is false does doing nothing do *f*?

***Figure 3.** Correctness conditions in question form for verifying each type of clear-box control structure.*

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

is, in concurrent assignment form,

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

For integer $x \geq 0$, the program function of the iteration

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

is, in English,

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

Design refinement. In designing clear-box procedures, you define an *intended function*, then refine it into a control structure and new intended functions, as Figure 2 illustrates. Intended functions, enclosed in braces, are recorded in the design and attached to their control-structure refinements. In essence, clear boxes are composed of a finite number of control structures, each of which can be checked for correctness.

Design simplification is an important objective in the stepwise refinement of clear boxes. The goal is to generate compact, straightforward, verifiable designs.

Correctness verification. To verify the correctness of each control structure, you derive its program function — the function it actually computes — and compare it to its intended function, as recorded in the design. A correctness theorem[7] defines how to do this comparison in terms of language- and application-independent *correctness conditions*, which you apply to each control structure.

Figure 3 shows the correctness conditions for the sequence, alternation, and iteration control structures. Verifying a sequence involves function composition and requires checking exactly one condition. Verifying an alternation involves case analysis and requires checking exactly two conditions. Verifying an iteration involves function composition and case analysis in a recursive equation and requires checking exactly three conditions.

Correctness verification has several advantages:

♦ *It reduces verification to a finite process.* As Figure 4 illustrates, the nested, sequenced way that control structures are organized in a clear box naturally defines a hierarchy that reveals the correctness conditions that must be verified. An axiom of replacement[7] lets us substitute intended functions for their control structure refinements in the hierarchy of subproofs. For example, the subproof for the intended function `f1` in Figure 4 requires proving that the composition of operations `g1` and `g2` with intended subfunction `f2` has the same effect on data as `f1`. Note that `f2` substitutes for all the details of its refinement in this proof. This substitution localizes the proof argument to the control structure at hand. In fact, it lets you carry out proofs in any order.

It is impossible to overstate the positive effect that reducing verification to a finite process has on quality. Even though all but the most trivial programs exhibit an essentially infinite number of execution paths, they can be verified in a finite number of steps. For example, the clear box in Figure 5 has exactly 15 correctness conditions that must be verified.

♦ *It lets Cleanroom teams verify every line of design and code.* Teams can carry out the verification through group

```
Program:          Subproofs:

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

***Figure 4.** A clear-box procedure and its constituent subproofs. In the figure, each `pi` is a predicate, each `gi` is an operation, and each `fi` is an intended function.*

analysis and discussion on the basis of the correctness theorem, and they can produce written proofs when extra confidence in a life- or mission-critical system is required.

♦ *It results in a near-zero defect level.* During a team review, every correctness condition of every control structure is verified in turn. Every team member must agree that each condition is correct, so an error is possible only if every team member incorrectly verifies a condition. The requirement for unanimous agreement based on individual verifications results in software that has few or no defects before first execution.

♦ *It scales up.* Every software system, no matter how large, has top-level, clear-box procedures composed of sequence, alternation, and iteration structures. Each of these typically invokes a large subsystem with thousands of lines of code — and each of those subsystems has its own top-level intended functions and procedures. So the correctness conditions for these high-level control structures are verified in the same way as are those of low-level structures. Verification at high levels may take, and well be worth, more time, but it does not take more theory.

♦ *It produces better code than unit testing.* Unit testing checks only the effects of executing selected test paths out of many possible paths. By basing verification on function theory, the Cleanroom approach can verify every possible effect on all data, because while a program may have many execution paths, it has only one function. Verification is also more efficient than unit testing. Most verification conditions can be checked in a few minutes, but unit tests take substantial time to prepare, execute, and check.

```
[ Q := odd_numbers(Q) || even_numbers(Q) ]
PROC Odd_Before_Even (ALT Q)

    DATA
       odds   : queue of integer [initializes to empty]
       evens  : queue of integer [initializes to empty]
       x      : integer
    END

    [ Q     := empty,
      odds  := odds  ||odd_numbers(Q),
      evens := evens ||even_numbers(Q) ]
    WHILE Q <> empty
    DO

       x := end(Q)
                                                  seq
       [x is odd -> odds := odds || x              1
       |true     -> evens := evens || x ]                wdo
       If odd(x)                                          3
       THEN                                          ite
          end(odds) := x                              2
       ELSE
          end(evens) := x
       END

    END
    [ Q    := Q || odds,
      odds := empty ]
    WHILE odds <> empty
    DO [end(Q) := end(odds)]                         wdo
                                              seq     3
       x      := end(odds)                     1
       end(Q) := x

    END
    [ Q     := Q || evens,
    evens:= empty]
    WHILE evens <> empty
    DO [end(Q) := end(evens)]                        wdo
                                              seq     3
       X      := end(evens)                    1
       end(Q) := x

    END

END odd-before-even
```
seq 1

Figure 5. *A clear-box procedure with 15 correctness conditions to be verified. The procedural control structures and the number of correctness conditions that must be checked are shown in bold. Seq indicates a sequence, ite indicates an alternation (if-then-else), and wdo indicates an iteration (while-do).*

QUALITY CERTIFICATION

Statistical quality control is used when you have too many items to test all of them exhaustively. Instead, you statistically sample and analyze some items to obtain a scientific assessment of the quality of all items. This technique is widely used in manufacturing, in which items on a production line are sampled, their quality is measured against a presumably perfect design, the sample quality is extrapolated to the entire production line, and flaws in production are corrected if the quality is too low.

In hardware products, the statistics used to establish quality are derived from slight variations in the products' physical properties. But software copies are identical, bit for bit. What statistics can we sample to extrapolate quality?

Usage testing. It turns out that software has a statistical property of great interest to developers and users — its execution behavior. How long, on average, will a software product execute before it fails?

From this notion has evolved the process of *statistical usage testing*,[8] in which you

♦ sample the (essentially infinite) pop-

CLEANROOM QUALITY RESULTS

Cleanroom projects report a *testing error rate per thousand lines of code*, which represents residual errors in the software after correctness verification. The projects briefly described here are among 17 Cleanroom projects, involving nearly a million lines of code, that have reported a *weighted average of 2.3 errors per KLOC found in all testing, measured from first-ever execution of the code* — a remarkable quality achievement.[1]

♦ *IBM Cobol Structuring Facility (Cobol/SF)*. This was IBM's first commercial Cleanroom product, developed by a six-person team. This 85 KLOC PL/I program automatically transforms unstructured Cobol programs into functionally equivalent structured form for improved understandability and maintenance. It had a testing error rate of 3.4 errors per KLOC; several major components completed certification with no errors found. In months of intensive beta testing at a major aerospace corporation, all Cobol programs executed identically before and after structuring.

Productivity, including all specification, design, verification, certification, user publications, and management, averaged 740 LOC per person-month. So far, a small fraction of a person-year per year has been required for all maintenance and customer support. Although the product exhibits a complexity level on the order of a Cobol compiler, just seven minor errors were reported in the first three years of field use, all resulting in simple fixes. — R.C. Linger and H.D. Mills, "A Case Study in Cleanroom Software Engineering: The IBM Cobol Structuring Facility," *Proc. Compsac*, IEEE CS Press, Los Alamitos, Calif., 1988, pp. 10-17.

♦ *NASA satellite-control project*. 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 of the NASA Goddard Space Flight Center. The system, comprising 40 KLOC of Fortran, exhibited a testing error rate of 4.5 errors per KLOC. Productivity was 780 LOC per person-month, an 80 percent improvement over previous SEL averages. Some 60 percent of the programs compiled correctly on the first attempt. — A. Kouchakdjian, S. Green, and V.R. Basili, "Evaluation of the Cleanroom Methodology in the Software Engineering Laboratory," *Proc. 14th Software Eng. Workshop*, NASA Goddard Space Flight Center, Greenbelt, Md., 1989.

♦ *Martin Marietta Automated Documentation System*. A four-person Cleanroom team developed the prototype for this system, a 1,820-line relational database application written in Foxbase. It had a testing error rate of 0.0 errors per KLOC — 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 method. — C.J. Trammell, L.H. Binder, and C.E. Snyder, "The Automated Production Control System: A Case Study in Cleanroom Software Engineering," *ACM Trans. Software Eng. and Methodology*, Jan. 1992, pp. 81-94.

♦ *IBM AOEXPERT/MVS*. A 50-person team developed this complex decision-support facility that uses artificial intelligence to predict and prevent operating problems in an MVS environment. The system, written in PL/I, C, Rexx, and TIRS, totaled 107 KLOC, developed in three increments. It had a testing error rate of 2.6 errors per KLOC. Causal analysis of the first 16-KLOC increment revealed that five of its eight components experienced no errors in testing.

The project reported development team productivity of 486 LOC per person-month. No operational errors have been reported to date from beta test and early user sites. — P.A. Hausler, "A Recent Cleanroom Success Story: The Redwing Project," *Proc. 17th Software Eng. Workshop*, NASA Goddard Space Flight Center, Greenbelt, Md., 1992.

♦ *NASA satellite-control projects*. Two satellite projects, a 20-KLOC attitude-determination subsystem and a 150-KLOC flight-dynamics system, were the second and third Cleanroom projects undertaken at NASA's Software Engineering Laboratory. These systems had a combined testing error rate of 4.2 errors per KLOC. — S.E. Green and Rose Pajerski, "Cleanroom Process Evolution in the SEL," *Proc. 16th Software Eng. Workshop*, NASA Goddard Space Flight Center, Greenbelt, Md., 1991.

♦ *IBM 3090E tape drive*. A five-person team developed the device-controller design and microcode in 86 KLOC of C, including 64 KLOC of function definitions. This embedded software processes multiple real-time I/O data streams to support tape-cartridge operations in a multibus architecture. The box-structure specification for the chip-set semantics revealed several hardware errors. The project had a testing error rate of 1.2 errors per KLOC.

A one-module experiment compared the effectiveness of unit testing and correctness verification. In unit testing, the team took 10 person-days to develop scaffolding code, invent and execute test cases, and check results. They found seven errors. Correctness verification, which required an hour-and-a-half in a team review, found the same seven errors plus three more.

To meet a business need, the third code increment went straight from development, with no testing whatsoever, into customer-evaluation demonstrations using live data. There were no errors of any kind. A total of 490 statistical tests were executed against the final version of the system, with no errors found.

♦ *Ericsson Telecom OS32 operating system*. This 70-person, 18-month project specified, developed, and certified a 350-KLOC operating system for a new family of switching computers. The project had a testing error rate of 1.0 errors per KLOC.

Productivity was reported to have increased by 70 percent for development; 100 percent for testing. The team significantly reduced development time, and the project was honored by Ericsson for its contribution to the company. — L.-G. Tann, "OS32 and Cleanroom," *Proc. 1st European Industrial Symp. Cleanroom Software Eng.*, Q-labs, Lund, Sweden, 1993.

REFERENCES

1. P.A. Hausler, R.C. Linger, and C.J. Trammell, "Adopting Cleanroom Software Engineering with a Phased Approach," *IBM Systems J.*, Mar. 1994, to appear.

Program stimuli	Usage-probability distribution	Distribution interval
U (update)	32%	0-31
D (delete)	14%	32-45
Q (query)	46%	46-91
P (print)	8%	92-99

(A)

Test number	Random numbers:	Test cases:
1	29 11 47 52 26 94	U U Q Q U P
2	62 98 39 78 82 65	Q P D Q Q Q
3	83 32 58 41 36 17	Q D Q D D U
4	36 49 96 82 20 77	D Q P Q U Q

(B)

Figure 6. *(**A**) Simplified usage probability distribution for a program with four user stimuli and (**B**) a sample of associated test cases.*

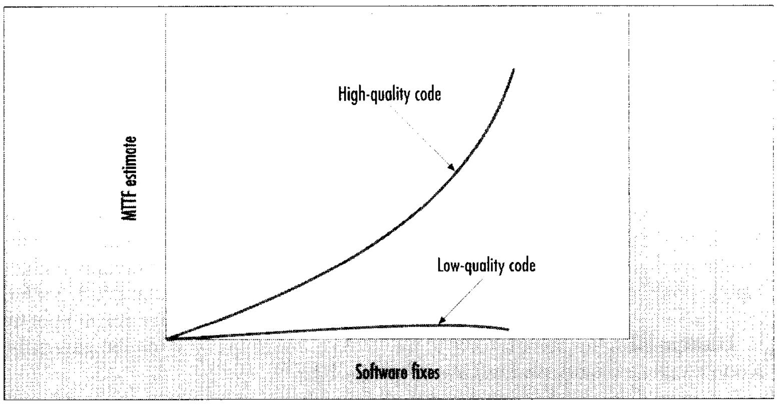


Figure 7. *Two sample graphs. The curve for high-quality software shows exponential improvement, such that the MTTF quickly exceeds the total test time. The curve for low-quality software shows little MTTF growth.*

ulation of all possible executions (correct and incorrect) by users (people or other programs) according to how frequently you expect the executions to happen,

♦ measure their quality by determining if the executions are correct,

♦ extrapolate the quality of the sample to the population of possible executions, and

♦ identify and correct flaws in the development process if the quality is inadequate.

Statistical usage testing amounts to testing software the way users intend to use it. The entire focus is on external system behavior, not the internals of design and implementation. Cleanroom certification teams have deep knowledge of expected usage, but require no knowledge of design internals. Their role is not to debug-in quality, an impossible task, but to scientifically certify software's quality through statistical testing.

In practice, Cleanroom quality certification proceeds in parallel with development, in three steps.

1. Specify usage-probability distributions. Usage-probability distributions define all possible usage patterns and scenarios, including erroneous and unexpected usage, together with their probabilities of occurrence. They are defined on the basis of the functional specification and other sources of information, including interviews with prospective users and the pattern of use in prior versions.

Figure 6a shows a usage specification for a program with four user stimuli: update (U), delete (D), query (Q), and print (P). A simplified distribution that omits scenarios of use and other details shows projected use probabilities of 32, 14, 46, and 8 percent, respectively. These probabilities are mapped onto an interval of 0 to 99, dividing it into four partitions proportional to the probabilities. Usage-probability distributions for large-scale systems are often recorded in formal grammars or Markov chains for analysis and automatic processing.

In incremental development, you can stratify a usage-probability distribution 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, you can define alternate distributions to certify infrequently used system functions whose failure has important consequences, such as the code for a nuclear-reactor shutdown system.

2. Derive test cases that are randomly generated from usage-probability distributions. Test cases are derived from the distributions, such that every test represents actual use and will effectively rehearse user experience with the product. Because test cases are completely prescribed by the distributions, producing them is a mechanical, automatable process.

Figure 6b shows test cases for the probability distribution in Figure 6a. If you assume a test case contains six stimuli, then you generate each test by obtaining six two-digit random numbers. These numbers represent the partition in which the corresponding stimuli (U, D, Q, or P) resides. In this way, each test case is faithful to the distribution and represents a possible user execution. For testing large-scale systems, usage grammars or Markov chains can be processed to generate test cases automatically.

3. Execute test cases, assess results, and compute quality measures. At this point, the development team has released verified code to the certification team for first-ever execution. The certification team executes each test case and checks the results against system specifications. The team records execution time up to the point of any failure in appropriate units, for example, CPU time, wall-clock time, or number of transactions.

These *interfail times* represent the quality of the sample of possible user executions. They are passed to a quality certification model that computes the system's quality, including its mean time to failure. The quality-certification model produces graphs like the one in Figure 7.

Because the Cleanroom development process rests on a formal, statistical design, these MTTF measures provide a scientific basis for management action, unlike the anecdotal evidence from coverage testing (If few errors are found, is that good or bad? If many errors are found, is that good or bad?). In theory, there is no way to ever know that a software system has zero defects. However, as failure-free executions accumulate, it becomes possible to conclude that the software is at or near zero defects with high probability.

WE BELIEVE USE OF THE CLEANROOM PROCESS WILL GROW.

Extending MTTF. 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.[9] Virulent, high-rate errors can literally occur every few hours for some users, but low-rate errors may show up only after accumulated decades of use by many users.

High-rate errors are responsible for nearly two-thirds of software failures reported,[10] even though they comprise less than three percent of total errors. Because statistical usage testing amounts to testing software the way users will use it, high-rate errors tend to be found first. Any errors left behind after testing tend to be infrequently encountered by users.

Traditional coverage testing finds errors in random order. Yet finding and fixing low-rate errors has little effect on MTTF and user perception of quality, while finding and fixing errors in failure-rate order has a dramatic effect. Statistical usage testing is far more effective than coverage testing at extending MTTF.[10]

Software that is formally engineered in increments is well-documented and under intellectual control throughout development. The Cleanroom approach provides a framework for managers to plan (and replan) schedules, allocate resources, and systematically accommodate changes in functional content.

Experienced Cleanroom teams can substantially reduce time to market. This is due largely to the precision imposed on development, which helps eliminate rework and dramatically reduces testing time, compared with traditional methods. Furthermore, Cleanroom teams are not held hostage by error correction after release, so they can initiate new development immediately.

The cost of quality is remarkably low in Cleanroom operations, because it minimizes expensive debugging rework and retesting.

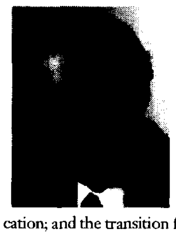
Cleanroom technology builds on existing skills and software-engineering practices. It is readily applied to both new system development and reengineering and extending legacy systems. As the need for higher quality and productivity in software development increases, we believe that use of the Cleanroom process will continue to grow. ♦

ACKNOWLEDGMENTS

I thank Kim Hathaway for her assistance in developing this article. Suggestions by Michael Deck, Philip Hausler, Harlan Mills, Mark Pleszkoch, and Alan Spangler are appreciated. I also thank the members of the Cleanroom teams, whose quality results are reported in this article, and who continue to achieve new levels of quality and productivity.

REFERENCES

1. M. Dyer, *The Cleanroom Approach to Software Quality*, John Wiley & Sons, New York, 1992.
2. P.A. Curritt, M. Dyer, and H.D. Mills, "Certifying the Reliability of Software," *IEEE Trans. on Software Eng.*, Jan. 1986, pp. 3-11.
3. H.D. Mills, R.C. Linger, and A.R. Hevner, *Principles of Information Systems Analysis and Design*, Academic Press, San Diego, 1986.
4. H.D. Mills, "Stepwise Refinement and Verification in Box-Structured Systems," *Computer*, June 1988, pp. 23-35.
5. A.R. Hevner and H. D. Mills, "Box Structure Methods for System Development with Objects," *IBM Systems J.*, No. 2, 1993, pp. 232-251.
6. M.G. Pleszkoch et al., "Function-Theoretic Principles of Program Understanding," *Proc. 23rd Hawaii Int'l Conf. System Sciences*, IEEE CS Press, Los Alamitos, Calif., 1990, pp. 74-81.
7. R.C. Linger, H.D. Mills, and B.I. Witt, *Structured Programming: Theory and Practice*, Addison-Wesley, Reading, Mass., 1979.
8. J.H. Poore and H. D. Mills, "Bringing Software Under Statistical Quality Control," *Quality Progress*, Nov. 1988, pp. 52-55.
9. E.N. Adams, "Optimizing Preventive Service of Software Products," *IBM J. Research and Development*, Jan. 1984, pp. 2-14.
10. R.H. Cobb and H.D. Mills, "Engineering Software Under Statistical Quality Control," *IEEE Software*, Nov. 1990, pp. 44-54.



Richard C. Linger is a member of the senior technical staff at IBM and the founder and manager of the IBM Cleanroom Software Technology Center. His interests are software specification, design, and correctness verification; statistical testing and reliability certification; and the transition from craft-based to engineering-based software development

Linger received a BS in electrical engineering from Duke University. He is a member of the IEEE Computer Society and ACM.

Address questions about this article to Linger at 20221 Darlington Dr., Gaithersburg, MD 20879.