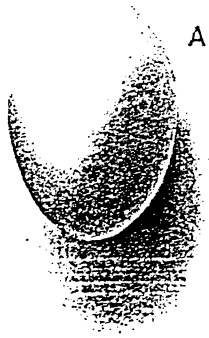

Cleanroom Software Engineering

Harlan D. Mills, *Information Systems Institute*
Michael Dyer *and* **Richard C. Linger**, *IBM Federal Systems Division*

Software quality can be engineered under statistical quality control and delivered with better quality. The Cleanroom process gives management an engineering approach to release reliable products.

Recent experience demonstrates that software can be engineered under statistical quality control and that certified reliability statistics can be provided with delivered software. IBM's Cleanroom process[1] has uncovered a surprising synergy between mathematical verification and statistical testing of software, as well as a major difference between mathematical fallibility and debugging fallibility in people.

With the Cleanroom process, you can engineer software under statistical quality control. As with cleanroom hardware development, the process's first priority is defect prevention rather than defect removal (of course, any defects not prevented should be removed). This first priority is achieved by using human mathematical verification in place of program debugging to prepare software for system test.

Its next priority is to provide valid, statistical certification of the software's quality through representative-user testing at the system level. The measure of quality is the mean time to failure in appropriate units of time (real or processor time) of the delivered product. The certification takes into account the growth of reliability achieved during system testing before delivery.

To gain the benefits of quality control during development, Cleanroom software engineering requires a development cycle of concurrent fabrication and certification of product increments that accumulate into the system to be delivered. This lets the fabrication process be altered on the basis of early certification results to achieve the quality desired.

Cleanroom experience

Typical of our experience with the Cleanroom process were three projects: an IBM language product (40,000 lines of code), an Air Force contract helicopter flight program (35,000 lines), and a NASA contract space-transportation planning system (45,000 lines). A major finding in these cases was that human verification, even though fallible, could replace debugging in software development — even informal human verification can produce

0740-7459/87/0900/0019/$01.00 ©1987 IEEE

software sufficiently robust to go to system test without debugging.

Typical program increments were 5000 to 15,000 lines of code. With experience and confidence, such increments can be expected to increase in size significantly. All three projects showed productivity equal to or better than expected for ordinary software development: Human verification need take *no more time* than debugging (although it takes place earlier in the cycle).

The combination of formal design methods and mathematics-based verification had a positive development effect: More than 90 percent of total product defects were found before first execution. This is in marked contrast to the more customary experience of finding 60 percent of product defects before first execution. This effect is probably directly related to the added care and attention given to design in lieu of rushing into code and relying on testing to achieve product quality.

A second encouraging trend is the drop in total defect count (by as much as half), which highlights the Cleanroom focus on error prevention as opposed to error detection. With industry averages at 50 to 60 errors per 1000 lines of code, halving these numbers is significant.

The IBM language product (Cobol/SF[2]) experience is especially instructive. This advanced technology product, comparable in complexity to a compiler, was formally specified and then designed in a process-design language. Specification text exceeded design text by about four to one. Every control structure in the design text was verified in formal, mathematics-based group inspection, so the product proved very robust. A first phase of development (20,000 lines) had just 53 errors found during testing.

Correctness verification was the cornerstone of the project; many programs were redesigned to permit simpler verification arguments. Productivity averaged more than 400 lines of code per man-month, largely as a result of sharply reduced testing time and effort compared to conventional developments.

A controlled experiment at the University of Maryland, with student teams developing a common project in message processing (1000 to 2000 lines), indicates better productivity and quality with the Cleanroom process than with interactive debugging and integration — even the first time you use it.[3]

Management perspective

At first glance, statistical quality control and software development seem incompatible. Statistical quality control seems to apply to manufacturing, especially manufacturing of multiple copies of a previously specified and designed part or product. Software development seems to be a one-of-a-kind logical process with no statistical properties at all. After all, if the software ever fails under certain conditions, it will always fail under those conditions.

Developing stable specifications early establishes clear accountability.

However, by rethinking the process of statistical quality control itself from a management perspective, we can find a way to put software development under statistical quality control with significant management benefits.

But where do the statistics come from, when neither software nor its development have any statistical properties at all? The statistics come from the usage of the software, not from its intrinsic properties. Engineering software under statistical quality control requires that we not only specify the functional behavior of the software but also its statistical usage.

Cleanroom software engineering is a practical process to place software development under statistical quality control. The significance of a process under statistical quality control is well-illustrated by modern manufacturing techniques where the sampling of output is directly fed back into the process to control quality. Once the discipline of statistical quality control is in place, management can see the development process and can control process changes to control product quality.

The Cleanroom process permits a sharper structuring of development work between specification, design, and testing, with clearer accountabilities for each part of the process. This structuring increases management's ability to monitor work in progress. Inexperienced software managers often fail to recognize and expose early software problems (like hardware or specification instability, inexperienced personnel, and incomplete design solutions) and mistakenly think they can resolve and manage these problems over time. The Cleanroom process forces these early problems into the open, giving all levels of management an opportunity to resolve them.

The Cleanroom process requires stable specifications as its basis. Because specifications are often not fully known or verified during initial development, it might appear at first glance that the Cleanroom process does not apply. But, in fact, the discipline of the Cleanroom process is most useful in forcing specification deficiencies into the open and giving management control of the specification process.

As long as development is treated as a trial-and-error process, the incompleteness of specification can be accommodated as just one more source of trial and error. The result is diluted accountability between specifiers and developers. A better way is to develop software to early, stable specifications that remain stable in each iteration. This establishes a clear accountability between specification and development, keeping management in control of specification changes.

Statistical quality control

Statistical quality control begins with an agreement between a producer and receiver. A critical part of this agreement, explicit or implicit, is how to measure quality, particularly *statistical* quality. For simple products with straightforward physical, electrical, or other measurements, the agreement may be simply stated — for example, 99 percent of certain filaments are to exhibit an electrical resistance within 10 percent of a fixed value. However, software is complex enough to require a new understanding on how

statistical quality can be measured.

For even the simplest of products, there is no absolute best statistical measure of quality. For example, a statistical average can be computed many ways — an arithmetic average, a weighted average, a geometric average, and a reciprocal average can each be better than the others in various circumstances.

It finally comes down to a judgment of business and management — in every case. In most cases, the judgment is practically automatic from experience and precedent, but it *is* a judgment. In the case of software, that judgment has no precedent because the concept of producing software under statistical quality control is just at its inception.

A new basis for the certification of software quality, given in Currit, Dyer, and Mills,[1] is based on a new software-engineering process.[4] This basis requires a software specification and a probability distribution on scenarios of the software's use; it then defines a testing procedure and a prescribed computation from test data results to provide a certified statistical quality of delivered software.

This new basis represents scientific and engineering judgment of a fair and reasonable way to measure statistical quality of software. As for simpler products, there is no absolute best and no logical arguments for it beyond business and management judgment. But it can provide a basis for software statistical quality as a contractual item where no such reasonable item existed before.

The certification of software quality is given in terms of its measured reliability over a probability distribution of usage scenarios in statistical testing. Certification is an ordinary process in business — even in the certification of the net worth of a bank. As in software certification, there is a fact-finding process, followed by a prescribed computation.

In the case of a bank, the fact-finding produces assets and liabilities, and the computation subtracts the sum of the liabilities from the sum of the assets. For the bank, there are other measures of importance besides net worth — such as goodwill, growth, and security of assets — just as there are other measures for software than reliability — such as maintainability and performance. So a certification of software quality is a business measure, part of the overall consideration in producing and receiving software.

Once a basis for measuring statistical quality of delivered software is available, creating a management process for statistical quality control is relatively straightforward. In principle, the goal is to find ways to repeatedly rehearse the final measurement during software development and to modify the development process, where necessary, to achieve a desired level of statistical quality.

The Cleanroom process has been designed to carry out this principle. It calls for the development of software in increments that permit realistic measurements of statistical quality during development, with provision for improving the measured quality by additional testing, by process changes (such as increased inspections and configuration control), or by both methods.

Statistical quality measurements ultimately come down to management and business judgments.

Mathematical verification

Software engineering without mathematical verification is no more than a buzzword. When Dijkstra introduced the idea of structured programming at an early software-engineering conference,[5] his principal motivation was to reduce the length of mathematical verifications of programs by using a few basic control structures and eliminating gotos.

Many popularizers of structured programming have cut out the rigorous part about mathematical verification in favor of the easy part about no gotos. But by cutting out the rigorous part, they have also cut out much of the real benefit of structured programming. As a result, a lot of people have become three-day wonders in having no gotos without acquiring the fundamental discipline of mathematical verification in engineering software — of even discovering that such a discipline exists.

In contrast, learning the rigor of mathematical verification leads to behavioral modification in both individuals and teams of programmers, whether programs are verified formally or not. Mathematical verification requires precise specifications and formal arguments about the correctness with respect to those specifications.

Two main behavioral effects are readily observable. First, communication among programmers (and managers) becomes much more precise, especially about program specifications. Second, a premium is placed on the simplest programs possible to achieve specified function and performance.

If a program looks hard to verify, it is the program that should be revised, *not* the verification. The result is high productivity in producing software that requires little or no debugging.

Cleanroom software engineering uses mathematical verification to replace program debugging before release to statistical testing. This mathematical verification is done by people, based on standard software-engineering practices[4] such as those taught at the IBM Software Engineering Institute. We find that human verification is surprisingly synergistic with statistical testing — that mathematical fallibility is very different from debugging fallibility and that errors of mathematical fallibility are much easier to discover in statistical testing than are errors of debugging fallibility.

Perhaps one day automatic verification of software will be practical. But there is no need to wait for the engineering value and discipline of mathematical verification until that day.

Experimental data from projects where both Cleanroom verification and more traditional debugging techniques were used offers evidence that the Cleanroom-verified software exhibited higher quality. For the verified software, fewer errors were injected, and these errors were less severe and required less time to find and fix. The verified product also experienced

better field quality, all of which was due to the added care and attention paid during design.

Findings from an early Cleanroom project (where verified software accounted for approximately half the product's function) indicate that verified software accounted for only one fourth the error count. Moreover, the verified software was responsible for less than 10 percent of the severe failures. These findings substantiate that verified software contains fewer defects and that those defects that are present are simpler and have less effect on product execution.

The method of human mathematical verification used in Cleanroom development, called functional verification, is quite different than the method of axiomatic verification usually taught in universities. It is based on functional semantics and on the reduction of software verification to ordinary mathematical reasoning about sets and functions as directly as possible.

The motivation for functional verification and for the earliest possible reduction of verification reasoning to sets and functions is the problem of scaling up. A set or function can be described in three lines of ordinary mathematics notation or in 300 lines of English text. There is more human fallibility in 300 lines of English than in three lines of mathematical notation, but the verification paradigm is the same.

By introducing verification in terms of sets and functions, you establish a basis for reasoning that scales up. Large programs have many variables, but only one function. Mills and Linger[6] gave an additional basis for verifying large programs by designing with sets, stacks, and queues rather than with arrays and pointers.

While initially harder to teach than axiomatic verification, functional verification scales up to reasoning for million-line systems in top-level design as well as for hundred-line programs at the bottom level. The evidence that such reasoning is effective is in the small amount of backtracking required in very large systems designed top-down with functional verification.[7]

Cleanroom software engineering

While it may sound revolutionary at first glance, the Cleanroom software engineering process is an *evolutionary* step in software development. It is evolutionary in eliminating debugging because, over the past 20 years, more and more program design has been developed in design languages that must be verified rather than executed. So the relative effort for advanced teams in debugging, compared to verifying, is now quite small, even in non-Cleanroom development.

It is evolutionary in statistical testing because with higher quality programs at the outset, representative-user testing is correspondingly a greater and greater fraction of the total testing effort. And, as already noted, we have found a surprising synergism between human verification and statistical testing: People are fallible with human verification, but the errors they leave behind for system testing are much easier to find and fix than those left behind from debugging.

In verified software, developers essentially never resorted to debugging.

Results from an early Cleanroom project where verification and debugging were used to develop different parts of the software indicate that corrections to the verified software were accomplished in about one fifth the average time of corrections to the debugged software. In the verified software case, the developers essentially never resorted to debugging (less than 0.1 percent of the cases) to isolate and repair reported defects.

The feasibility of combining human verification with statistical testing makes it possible to define a new software-engineering process under statistical quality control.[1] For that purpose, we define a new development life cycle of successive incremental releases to achieve a structured specification of function and statistical usage. A structured specification is a formal specification (a relation or set of ordered pairs) for a decomposition into a nested set of subspecifications for successive product releases. A structured specification defines not only the final software but also a release plan for its incremental implementation and statistical testing.

A stepwise refinement or decomposition of requirements creates successive levels of software design. At each level of decomposition, mathematics-based correctness arguments ensure the accuracy of the evolving design and the continued integrity of the product requirements. The work strategy is to create specifications and the design to those specifications, as well as to check the correctness of that design before proceeding to the next decomposition.

The Cleanroom design methods use a limited set of design primitives to capture software logic (sequence, selection, and iteration). They use module and procedure primitives to package software designs into products. Decomposition of software data requirements is handled by a companion set of data-structuring primitives (sets, stacks, and queues) that ensure product designs with strongly typed data operations. Specially defined design languages document designs and provide a straightforward translation to standard programming forms.

In the Cleanroom model, structural testing that requires knowledge of the design is replaced by formal verification, but functional testing is retained. In fact, this testing can be performed with the two goals of demonstrating that the product requirements are correctly implemented in the software and of providing a basis for product-reliability prediction. The latter is a unique Cleanroom capability that results from its statistical testing method, which supports statistical inference from the test to operating environments.

The Cleanroom life cycle of incremental product releases supports software testing throughout the product development rather than only when it is completed. This allows the continuous assessment of product quality from an execution perspective and permits any necessary adjustments in the process to improve observed product quality.

As each release becomes available,

statistical testing provides statistical estimates of its reliability. Software process analysis and feedback can be used to meet reliability goals (for example, by increased verification inspections and by more intermediate specification formality) for subsequent releases. As errors are found and fixed during system testing, the growth in reliability of the maturing system can also be estimated so a certified reliability estimate of the system-tested software can be provided at final release.

Cho[8] has also proposed the development of software under statistical quality control, using as a measure the ratio of correct outputs to total outputs. He regards software as a factory for producing output, rather than for producing a product itself. The ratio of correct outputs to total outputs is directly related to the mean time between failures, where time is normalized to output production. Such a normalization is one possibility in the Cleanroom process, but other normalizations may be more meaningful in most system applications.

A principal difference between the Cleanroom and Cho's ideas is the use of a certification model to account for the growth in reliability during development. Another major difference is an insistence on human mathematical verification with no program debugging before representative-user testing at the system level. As Mills discussed,[9] human mathematical verification is possible and practical at high production rates. The time spent on verification can be less than the time spent on debugging.

Statistical basis

Software people customarily talk about errors in the software, typically measured in errors per thousand lines of code. Current postdelivery levels in ordinary software are one to 10 errors per thousand lines. Good methodology produces postdelivery levels under one error per thousand lines. But such numbers are irrelevant and misleading when you consider software reliability. Users do not see errors in the software, they see failures in execution, so the measurement of times between failures is more relevant.

If each error had the same or similar failure rate, there would be a direct relationship between the number of errors in software and the time between failures in its execution. Half as many errors would mean half the failure rate and twice the mean time between failures. In this case, efforts to reduce errors would automatically increase reliability.

It turns out that every major IBM software product — without exception — has an extremely high error-failure rate variation. In stable released products, these failure rates run from 18 months between failures to more than 5000 years. More than half the errors have failure rates of more than 1500 years between failures. Fixing these errors will reduce the number of errors by more than half, but the decrease in the product failure rate will be imperceptible. More precisely, you could remove more than 60 percent of the errors but only decrease the failure rate by less than 3 percent.

Users do not see errors in software, they see failures in execution.

These surprising refutations of conventional wisdom in software reliability are due to data painstakingly developed over many years by Adams.[10]

To be more precise about software errors and failures, assume that a specification and its software exist. Then, when the software is executed, its behavior can be compared with its specification and any discrepancies (failures). Such failures may be catastrophic and prevent further execution (for example, by abnormal termination). Other failures may be so serious that every response from then on is incorrect (for example, if a database is compromised). Less serious failures represent the case in which the software continues to execute with at least partially correct behavior beyond the failure.

These examples illustrate that failures represent different levels of severity, beginning with three major levels:

- terminating failures,
- permanent failures (but not terminating), and
- sporadic failures.

Even terminating or permanent failures may be followed by a restart of the software, so you can imagine a long history of execution and, in this history, the failures marked at each instant of time. Clearly, this history will depend on the software's initial conditions (and data) and on the subsequent inputs (as commands and data) to it. Such a history can be very arbitrary, but suppose for argument's sake that representative histories (scenarios of use) are conceivable.

The behavior of software is deterministic in that repeating an initial condition and history of use will reproduce the same outputs (with the same failures). But, in fact, if software is used in more than one history by more than one user, the histories of use will usually be different. For that reason, we consider as part of a structured specification a probability distribution of usage histories, typically defined as a stochastic process.

This probability distribution of usage histories will, in turn, induce a probability distribution of failure histories in which statistics about times between failures, failure-free intervals, and the like can be defined and estimated. So, even though software behavior is deterministic, its reliability can be defined relative to its statistical usage. Such a probability distribution of usage histories provides a statistical basis for software quality control.

Certifying statistical quality

For software already released, it is simple to estimate its reliability in mean times to failure: Merely take the average of its times between failure in statistical testing. However, for software under Cleanroom development, the problem is more complicated, for two reasons:

1. In each Cleanroom increment, results of system testing may indicate software changes to correct failures found.
2. With each Cleanroom increment release, untested new software will be added to software already under test.

In fact, each change or set of changes to correct failures in a release creates a new

software product very much like its predecessor but with a different reliability (intended to be better, but possibly worse). However, each of these corrected software products, by itself, will be subject to a strictly limited amount of testing before it is superseded by its successor, and statistical estimates of reliability will be correspondingly limited in confidence.

Therefore, to aggregate the testing experience for an increment release, we define a model of reliability change with parameters M and R (as discussed in Currit, Dyer, and Mills[1]) for the mean time to failure after c software changes, of the form $MTTF = MR^c$ where M is the initial mean time to failure of the release and where R is the observed effectiveness ratio for improving mean time to failure with software changes.

Although various technical rationales are given for this model by Currit, Dyer, and Mills,[1] it should be considered a contractual basis for the eventual certification of the finally released software by the developer to the user. Moreover, because there is no way to know that the model parameters M and R are absolutely correct, we define statistical estimators for them in terms of the test data. The choice of these estimators is based on statistical analysis, but the choice should also be a contractual basis for certification.

The net result of these two contractual bases — a reliability change model and statistical estimators for its parameters — gives producer and receiver (seller and purchaser) a common, objective way to certify the reliability of the delivered software. The certification is a scientific, statistical inference obtained by a prescribed computation on test data warranted to be correct by the developer.

In principle, the estimators for software reliability are no more than a sophisticated way to average the times between failure, taking into account the change activity called for during statistical testing. As test data materializes, the reliability can be estimated, even change by change. And with successful corrections, the reliability estimates will improve with further testing, providing objective, quantitative evidence of the achievement of reliability goals.

This objective evidence is itself a basis for management control of the software development to meet reliability goals. For example, process analysis may reveal both unexpected sources of errors (such as poor understanding of the underlying hardware) and appropriate corrections in the process itself for later increments. Intermediate rehearsals of the final certification provide a basis for management feedback to meet final goals.

The treatment of separate increment releases should also be part of the contractual basis between the developer and user. Perhaps the simplest treatment is to treat separate increments independently. However, more statistical confidence in the final certification will result from aggregate testing experience across increments. A simple aggregation could complement separately treated increments with management judgment.

A more sophisticated treatment of separate releases would be to model the failure contribution of each newly released part of the software and to develop stratified estimators release by release. Earlier releases can be expected to mature while later releases come under test. This maturation rate in reliability improvement can be used to estimate the amount of test time required to reach prescribed reliability levels.

Mean time to failure and the rate of change in mean time to failure can be useful decision tools for project management. For software under test, which has both an estimated mean time to failure and a known rate of change in mean time to failure, decisions on releasability can be based on an evaluation of life-cycle costs rather than on just marketing effect.

When the software is delivered, the average cost for each failure must include both the direct costs of repair and the indirect costs to the users (which may be much larger). These postdelivery costs can be estimated from the number of expected failures and compared with the costs for additional predelivery testing. Judgments could then be made about the profitability of continuing tests to minimize lifetime costs. ⊹

References

1. A. Currit, M. Dyer, and H.D. Mills, "Certifying the Reliability of Software," *IEEE Trans. Software Eng.*, Jan. 1986, pp. 3-11.
2. *Cobol Structuring Facility Users Guide*, IBM Corp., Armonk, N.Y., 1986.
3. R.W. Selby, V.R. Basili, and F.T. Baker, "Cleanroom Software Development: An Empirical Evaluation," Tech. Report TR-1415, Computer Science Dept., Univ. of Maryland, College Park, Md., Feb. 1985.
4. R.C. Linger, H.D. Mills, and B.I. Witt, *Structured Programming: Theory and Practice*, Addison-Wesley, Reading, Mass., 1979.
5. E.W. Dijkstra, "Structured Programming," in *Software Engineering Techniques*, J.N. Burton and B. Randell, eds., NATO Science Committee, New York, 1969, pp. 88-93.
6. H.D. Mills and R.C. Linger, "Data-Structured Programming," *IEEE Trans. Software Eng.*, Feb. 1986, pp. 192-197.
7. A.J. Jordano, "DSM Software Architecture and Development," *IBM Technical Directions*, No. 3, 1984, pp. 17-28.
8. C.K. Cho, *Quality Programming: Developing and Testing Software with Statistical Quality Control*, John Wiley & Sons, New York, 1987.
9. H.D. Mills, "Structured Programming: Retrospect and Prospect," *IEEE Software*, Nov. 1986, pp. 58-66.
10. E.N. Adams, "Optimizing Preventive Service of Software Products," *IBM J. Research and Development*, Jan. 1984.