11-1990

Engineering Software Under Statistical Quality-Control

R. H. Cobb

Harlan D. Mills

Engineering Software under Statistical Quality Control

Richard H. Cobb and **Harlan D. Mills**, *Software Engineering Technology*

The costs of continuing to develop failure-laden software with its associated low productivity are unacceptable. Cleanroom engineering promises lower costs and improved quality.

Society has been developing software for less than one human generation. We have accomplished a great deal in this first generation when compared to the accomplishments of other disciplines: During the first generation of civil engineering, the right triangle hadn't been invented; accountants did not discover double-entry concepts in the early generations of their field.

Yet despite such significant progress, software-development practices need improvement. We must solve such problems as

- execution failures, which exist to the extent that software failures are accepted as normal by most people,
- projects that are late and/or over budget, and
- the labor-intensive nature of software development — productivity increases have been modest since the introduction of Cobol.

And at the same time we are having difficulty producing reliable software there is a demand for even more complex, larger software systems.

These problems are symptoms of a process that is not yet under intellectual control. An activity is under intellectual control when the people performing it use a theoretically sound process that gives each of them a high probability of obtaining a commonly accepted correct answer.

When most endeavors begin, they are out of intellectual control. Intellectual control is achieved when theories are developed, implementation practices are refined, and people are taught the process.

A good example is long division. For many generations, performing division with Roman numerals was error-prone. Today, children who learn how to do long division with Arabic numerals obtain the correct answer most of the time. The long division algorithm is:

1. If the division is not complete, invent (estimate) the next digit.

0740-7459/90/1100/0044/$01.00 © 1990 IEEE

2. Verify the invention (estimate) made in step 1.

3. If the verification is correct and the division is not complete, repeat step 1 for the next digit; if the verification is not correct, repeat step 1 for the same digit by adjusting the invention.

This is a powerful algorithm for establishing intellectual control. A difficult problem, which on the surface seems to require a large invention, has been divided into a series of smaller problems, each requiring a smaller invention. Most important, each inventive step is followed immediately by a verification step to appraise the invention's correctness, so subsequent inventions don't build on incorrect results.

This algorithm also applies to software design and development. As software technologists strive to find better ways to develop software, we believe that they are hindered by some widely accepted beliefs about how to develop software. We believe that if we adopt new perspectives about these development myths, we will open the way to development practices that will permit the construction of software that contains few, if any, latent failures.

We have used new perspectives to derive Cleanroom engineering practices. Cleanroom engineering develops software under statistical quality control by

- specifying statistical usage,
- defining an incremental pipeline for software construction that permits statistical testing, and
- separating development and testing (only testers compile and execute the software being developed).

These practices have been demonstrated to provide higher quality software — software with fewer latent execution failures. These same engineering practices also have been observed to improve productivity. Table 1 summarizes some quality and productivity metrics for projects using some or all of these new software-development practices.

Software myths

Myth: Software failures are unavoidable.

This myth holds that software always contains latent execution failures that will be found by users. Therefore, we must learn to live with and manage around software failures.

Fact: Like other engineering activities, engineering software is a human activity subject to human fallibilities. Yet other engineering disciplines have learned how to design large and complex products with a low probability that the designs contain faults that will cause latent execution failures. When structural engineers design a bridge there is a high expectation that the bridge, when built, will not fall down.

Experience to date supports our belief that as software developers move from today's heuristic programming to rigorous software engineering, quality will increase and costs will decrease.

In other engineering disciplines, design failures are neither anticipated nor accepted as normal. When a failure happens, major investigations are undertaken to determine why it occurred. Other engineering professions have minimized error by developing a sound, theoretical base on which to build design practices.

But because software developers expect and accept design failures, software users cannot have the same high expectations as users of other products. We believe this is because software developers rely on an incomplete theory, so their engineering practices don't work.

Software engineers should be required to use engineering practices that produce software that does not contain faults that cause latent execution failures. Users want the same high assurance that software will work according to its specification that they have for products designed by other engineers.

The software profession is young, so we might want to start with modest goals, such as: Design and implement a 100,000 line system so, more often than not, no execution failure will be detected during the software's entire field life.

Even this modest goal is beyond our expectations using the development practices we now rely on. We believe such a goal is well within our capabilities if we use the ideas summarized in this article. For example, the software for the IBM Wheelwriter typewriter, developed using some of these ideas, has been in use for more than six years with millions of users and no software failure has ever been reported.[1]

Myth: Quality costs money.

Many people believe that software designed to execute with no or few failures costs more per line of code to produce.

Fact: Failures and cost are positively correlated. It is more expensive to remove latent execution failures designed into the software than to rigorously design the software to prevent execution failures. For example, touch-typing is both more reliable and productive than hunt-and-peck typing.

We believe — and experience to date supports our belief — that as software developers move from today's heuristic programming to rigorous software engineering, quality will increase and design and development costs will decrease.

Table 1.
Selected sample of Cleanroom projects.
(All other projects known to authors report substantial improvements in quality and productivity.)

Year	Applied technologies	Implementation	Results
1980	Stepwise refinement Functional verification	Census, 25 KLOC (Pascal)	• No failure ever found • Programmer received gold medal from Baldridge
1983	Functional verification Inspections	Wheelwriter, 63 KLOC, three processors	• Millions of users • No failure ever found
1980s	Functional verification Inspections	Space shuttle, 500 KLOC	• Low defect over entire function • No defect in any flight • Work received NASA's Quality Award
1987	Cleanroom engineering	Flight control, 33 KLOC (Jovial),, three increments	• Completed ahead of schedule • 2.5 errors/KLOC before any execution • Error-fix effort reduced by a factor of five
1988	Cleanroom engineering	Commercial product, 80 KLOC (PL/I)	• Certification testing failure rate of 3.4 failures/KLOC • Deployment failures of 0.1/KLOC • Productivity of 740 lines/man-month
1989	Partial Cleanroom engineering	Satellite control, 30 KLOC (Fortran)	• Certification testing error rate of 3.3 failures/KLOC • 50-percent improvement in quality • Productivity of 780 lines/man-month • 80-percent improvement in productivity
1990	Cleanroom engineering with reuse and new Ada design language	Research project, 12 KLOC (Ada and ADL)	• Certified to 0.9978 with 989 test cases; 36 failures found during certification (20 logic errors, or 1.7 errors/KLOC

Myth: Unit verification by debugging works on systems of any size.

Unit verification — debugging — is best done by a single programmer who exercises the program with specially constructed test cases. During debugging, the programmer constructs test cases, develops programs to run isolated units of the system, runs the tests, and fixes discrepancies as they are observed. This process continues until the programmer is satisfied the program performs its intended mission.

Fact: Although it is satisfactory when the software product is small, unit verification by debugging does not scale up. When the product is large and unit verification exercises only a small portion of the total system, the results are not satisfactory.

Debugging doesn't scale up because it often compromises the design's integrity. Typically, software units are built according to a sound design and fit together according to the design when unit debugging begins. But the fixes introduced during debugging, while they may seem to make individual modules perform their intended mission fully, cause design faults when the fixed modules are combined. These failures are then either found during integration testing or left in the product as latent failures. Debugging seems to produce local correctness and global incorrectness.

Ed Adams examined every failure report for nine of IBM's most widely used software products for several years and traced each to its origin. He found that in most cases the cause of the failure was introduced during an attempt to fix another failure.[2]

Fact: Unit verification by logical argument *does* scale up. This method of unit verification is based on the time-tested method of proving the correctness of an assertion by developing a proof. A program specification is a function or relation; a program of any size or complexity is a rule for a function. So all you have to do to show the correctness of a program is to show that it is a complete rule for a subset of the specification.

Experience indicates that using proof arguments to show program correctness is not an academic curiosity that works on small problems — it is a robust technique that works well on large, complex systems.

Table 1 summarizes data for a few projects that used unit verification by logical argument. All our experience with this method indicates that the scale-up problem associated with debugging is very tractable. Unit verification by logical argument seems to work because when a defect is found in a proof argument the focus can't shift to local concerns to make something work — the argument focuses entirely on global issues.

Fact: Unit verification via logical argument is more cost-effective than unit verification via debugging, for four reasons:

- Design errors are caught sooner and as a result are less costly to fix.
- It eliminates the expense of finding the subtle, hard-to-fix failures introduced by debugging.
- It eliminates the expense of building programs to permit unit testing and preparing unit test cases.
- Surprisingly, it takes less time.

Do we really mean that unit tests should not be conducted? Yes. Unit testing is done to demonstrate that the unit satisfies its specification. We believe you can better demonstrate this with logical arguments. So if we don't test units, then what do we test and when? The answer to that ques-

tion involves another myth.

Myth: The only way to perform unit verification via logical argument is to use a computer program.

Researchers have invested significant effort into building programs that use axiomatic arguments to verify programs. These programs, as of now and for the foreseeable future, can verify only small programs using a limited number of language constructs. Developers have not been able to scale up axiomatic verification programs even with today's very fast computers.

Fact: Engineers can verify large programs made up of many language constructs with functional verification. Functional verification, introduced by Richard Linger, Harlan Mills, and Bernard Witt,[3] is quite different from axiomatic verification.

With functional verification, you structure a proof that a program implements its specification correctly. Again, if a program specification is a function then a program is a rule for a function. The proof must show that the rule (the program) correctly implements the function (the specification) for the full range of the function and no more.

Linger, Mills, and Witt have developed a *correctness theorem* that defines what must be shown for each of the structured programming language constructs. The proof strategy is divided into small parts, which are easily accumulated into a proof for a large program. Our experience indicates that people can master these ideas and construct proof arguments for very large programs.

The first reaction of many people is that it must be hard to construct a proof that a program is correct. Our experience indicates that, with a modest amount of training and the opportunity to use the ideas on the job, people can learn to develop proof arguments and talk to other engineers in terms of proofs.

Linger,[4] Mills,[1] Richard Selby,[5] and others have analyzed the performance of software engineers using functional verification to perform unit verification via logical argument. Among their observations:

- Engineers find logic errors with functional verification, leaving only simple errors like syntax oversights to be found during execution testing.
- Many engineers find the mental challenge of functional verification more stimulating and satisfying than debugging.
- Many engineers find the team style associated with functional verification more satisfying than the solo style associated with debugging.
- Engineers can learn how to perform unit verification via functional verification.
- Engineers performing functional verification leave significantly fewer failures to be found during later life-cycle phases than debuggers. Data indicates that functional verification leaves only two to five fixes per thousands lines of code to be made in later phases,[4] compared to 10 to 30 fixes left by unit testing by debugging.[6]
- Engineers practicing functional verification complete the total development process with significantly less effort than those practicing unit verification via debugging. Measurements indicate that the improvement in productivity may be three to five times.[4]

Coverage testing is as likely to find a rare execution failure as it is a frequent one. Usage testing that matches the actual usage profile has a better chance of finding the execution failures that occur frequently.

Myth: Software is best tested by designing tests that cover every path through the program.

This testing method, called coverage testing, requires that the test developer be completely familiar with the software's internal design.

Fact: Statistical usage testing is 20 times more cost-effective in finding execution failures than coverage testing (a claim we will prove later).

In statistical usage testing, the test developer draws tests at random from the population of all possible uses of the software, in accordance with the distribution of expected usage. The test developer must understand what the software is intended to do and how it is expected to be used. The test developer then constructs tests that are representative of expected usage. No knowledge of how the software is designed and constructed is required.

Fact: Users observe failures in execution. While developers talk of finding and fixing errors or faults, users don't observe errors or faults. They observe execution failures, which occur when the software doesn't do something it's required to do.

When a tester observes an execution failure, the software is searched for a way to prevent it. As a result of the search, changes are made to the code that may or may not fix the failure and may or may not introduce new latent failures. The modifications are counted to obtain a count of software errors or faults.

For example, if you change five areas of the program because they were apparently doing something they shouldn't be doing, we say that five errors have been found and fixed. Software failures are precise while software errors are imprecise. It is execution failures that must be found and eliminated from software.

Some execution failures will occur frequently, others infrequently. Coverage testing is as likely to find a rare execution failure as it is a frequent execution failure. Usage testing that matches the actual usage profile has a better chance of finding the execution failures that occur frequently.

Therefore, since the goal of a testing program should be to maximize expected mean time to failure, a strategy that concentrates on the failures that occur frequently is more effective than one that has an equal probability of finding high- and low-frequency failures.

Myth: It doesn't matter how errors or failures are found, as long as they are fixed.

Fact: The failure rates of different errors can vary by four orders of magnitude in complex systems. To measure the increased effectiveness of usage testing over coverage testing, you need to know the frequency of rare failures versus frequent failures in a population of programs under test. The Adams study contains one

Table 2.
Software failures for nine major IBM products, classified from rare to frequent.

		Rare →							Frequent
Group		1	2	3	4	5	6	7	8
MTTF (years)		5,000	1,580	500	158	50	15.8	5	1.58
Percent failures in class for product	1	34.2	28.8	17.8	10.3	5.0	2.1	1.2	0.7
	2	34.3	28.0	18.2	9.7	4.5	3.2	1.5	0.7
	3	33.7	28.5	18.0	8.7	6.5	2.8	1.4	0.4
	4	34.2	28.5	18.7	11.9	4.4	2.0	0.3	0.1
	5	34.2	28.5	18.4	9.4	4.4	2.9	1.4	0.7
	6	32.0	28.2	20.1	11.5	5.0	2.1	0.8	0.3
	7	34.0	28.5	18.5	9.9	4.5	2.7	1.4	0.6
	8	31.9	27.1	18.4	11.1	6.5	2.7	1.4	1.1
	9	31.2	27.6	20.4	12.8	5.6	1.9	0.5	0.0
Average percentage failures		33.4	28.2	18.7	10.6	5.2	2.5	1.0	0.4
Probability of a failure for this frequency		0.008	0.021	.044	0.079	0.123	0.187	0.237	0.300

large database we can use to estimate increased effectiveness.

Table 2 summarizes Adams's data, which has been classified across columns by the frequency that a some user found a failure.[2] Each row represents a major IBM system like MVS, Cobol, and IMS. The columns represent a subdivision of the frequency in which users observed a failure.

For example, the first column represents failures observed by users on the average of once every 5,000 years of usage; the last column represents failures observed by users on the average of once every 1.58 years of usage. The data in each cell defines the percentage of all failures observed for the software system represented by that row with the expected frequency represented by that column. The values in each row sum to 100.

The remarkable fact is that, over this very divergent range of products, the distribution of failures occurring with different frequencies is uniform. This lets us use the data for analysis.

The bottom two rows of Table 2 contain two numbers for each failure frequency, the average percent failures for the group and the probability of a failure of the frequency represented by that group. An examination of these last two rows provides some critical insights. Groups 1 and 2, which represent failures that will be observed less than once in 1,580 years of expected use, account for 61.6 percent of fixes made but only for 2.9 percent of the failures that will be observed by typical users. On the other hand, groups 7 and 8 represent only 1.4 percent of the fixes made to the software but eliminate 53.7 percent of the failures that would be observed by a typical user.

If you use coverage testing, you would spend 61.6 percent of the testing and correction budget on finding and fixing errors that will eliminate only 2.9 percent of the failures, and only 1.4 percent on making fixes that would eliminate 53.7 percent of failures. Coverage testing doesn't appear to very effective at allocating the testing and correction budget to increase MTTF.

On the other hand, a usage testing strategy allocates the budget in accordance with the probability that a failure is observable by the average user: It allocates 53.7 percent to fixes that will occur 53.7 percent of the time in the experience of an average user.

Using the data in Table 2, we can show that usage testing is 21 times more effective at increasing MTTF than coverage testing. Let P be the increase in MTTF obtained by the next fix determined by coverage testing. Then the increase in MTTF obtained by the next fix determined by usage testing will be:

$$((0.008/60) + (0.021/19) + (0.044/6) + (0.079/1.9) + (0.123/0.6) + (0.187/0.19) + (0.237/0.06) + (0.30/0.019))\ P = 20.98\ P$$

This surprising result suggests the prevailing strategy for testing and correcting software is very inefficient.

Myth: Software behavior is deterministic. Therefore, statistics cannot be used to make inferences about software quality.

Fact: Software use is stochastic. A software system has many different uses to perform different missions starting from different initial conditions and given different input data. Each different use is a different event. Given a system that contains some latent failures, some usages will result in a failure; others in a correct execution. If you sample the entire population of all possible usages in accordance with an expected usage profile and maintain a record of failures and success, you can use statistics to estimate reliability.

Fact: You can estimate the expected MTTF for a system from a series of tests drawn at random in accordance with an expected usage profile from the population of all possible uses. The major assumption you must make to make the statistical estimation valid is that the development process is in a state of control. This is not an unreasonable assumption — it is the same one made when statistical quality-control practices are

applied to a production process.

While our experience in applying statistical quality-control techniques to software development is limited, initial experience indicates that five fixes per thousand lines of code can be tolerated without invalidating the application of statistics to estimate MTTF. This failure rate is low compared to normal development practices, where 20 to 60 fixes per thousand lines of code is not atypical.

Fact: Experience indicates that it is possible to design and develop software that requires less than five fixes per thousand lines of code from its first compilation throughout its useful life. The engineering practices that let such quality be achieved before any execution testing are grouped under the heading "Cleanroom engineering."

Myth: The solution to the development problem is to create tools that will do for people what they can't do for themselves.

The general idea behind this myth is that people can't be trusted to make the difficult inventions that software development requires.

Fact: Automation is very effective in helping us do the things we already know how to do. We know how to write. A word processor helps us write faster, but it doesn't help us write better (except that it gives us more time to think).

A translator — a compiler — can translate a high-level language definition into machine-level instructions. For example, compilers translate a Fortran or Cobol program into machine language. While this translation algorithm can be performed by people or computers, computers have an advantage because, once they have been programmed to do it, they are fast and reliable and can free people to do something else.

Fact: Automation is not effective in helping us do things we don't know how to do algorithmically. When we computerize incomplete algorithms, the results are incomplete and unsatisfactory. When database management systems were first introduced, hierarchical and network databases were common. Database management programs encountered failures that were eventually traced to a common set of problems which E.F. Codd named data-maintenance abnormalities.

These abnormalities, which cost business a great deal in terms of wrong decisions and software fixes, were caused by a basic failure in the hierarchical and network database models. These models could not maintain the referential transparency between the actual data and the state data used to represent it in computations: In certain situations, the value of the state data did not accurately represent the actual data as stored in the database. Codd's relational algorithm does maintain referential transparency, and if it is used to maintain keys in a relational database, it eliminates these failures.

This should have been an important lesson learned, but apparently the lesson was lost, because loss of referential transparency is still a common design flaw. The current generation of computer-aided software-engineering tools does not help maintain referential transparency and in some cases even allows designs that do not exhibit referential transparency.

Tools are only as good as the ideas that serve as their foundation. The important factor in selecting design and development tools is to select the ideas you want to use to help guide the inventive process.

For example, some CASE tools help you invent program structures by converting dataflow diagrams into program structures. Due to the one-to-many relationship between a dataflow diagram and a program-structure chart, it is easy to lose referential transparency between the history of stimuli to the software and the state data used to represent the stimuli histories.

Fact: Ideas must precede tools. Tools are only as good as the ideas that serve as their foundation. The important factor in selecting tools to assist in software design and development is to select the ideas that you want to use to help guide the inventive process. Once that is done then the ideas can be organized into an engineering process that helps people exploit the chosen ideas. Then it is possible to select or build tools that enhance peoples' productivity in performing these ideas.

Cleanroom engineering

These ideas are the foundation for the set of software-engineering practices we call Cleanroom engineering.[7]

Cleanroom engineering can help software engineers implement reliable software — software that won't fail during use. Cleanroom engineering

- achieves intellectual control by applying rigorous, mathematics-based engineering practices,
- establishes an "errors-are-unacceptable" attitude and a team responsibility for quality,
- delegates development and testing responsibilities to separate teams, and
- certifies the software's MTTF through the application of statistical quality-control methods.

Process. Cleanroom engineering involves a specification team, a development team, and a certification team. The specification team prepares and maintains the specification and specializes it for each development increment. The development team designs and implements the software. The certification team compiles, tests, and certifies the software's correctness.

In the Cleanroom engineering, the team members

- complete a rigorous, formal specification, even if it is preliminary, before they begin design and development,
- develop a construction plan by decomposing the specification into small (seldom more than 10,000 lines of third-generation code) user-executable increments,
- design, implement, and verify each user-executable increment, and
- assess the software's quality.

Typical project. Figure 1 shows a profile of a typical Cleanroom engineering project, divided into phases.

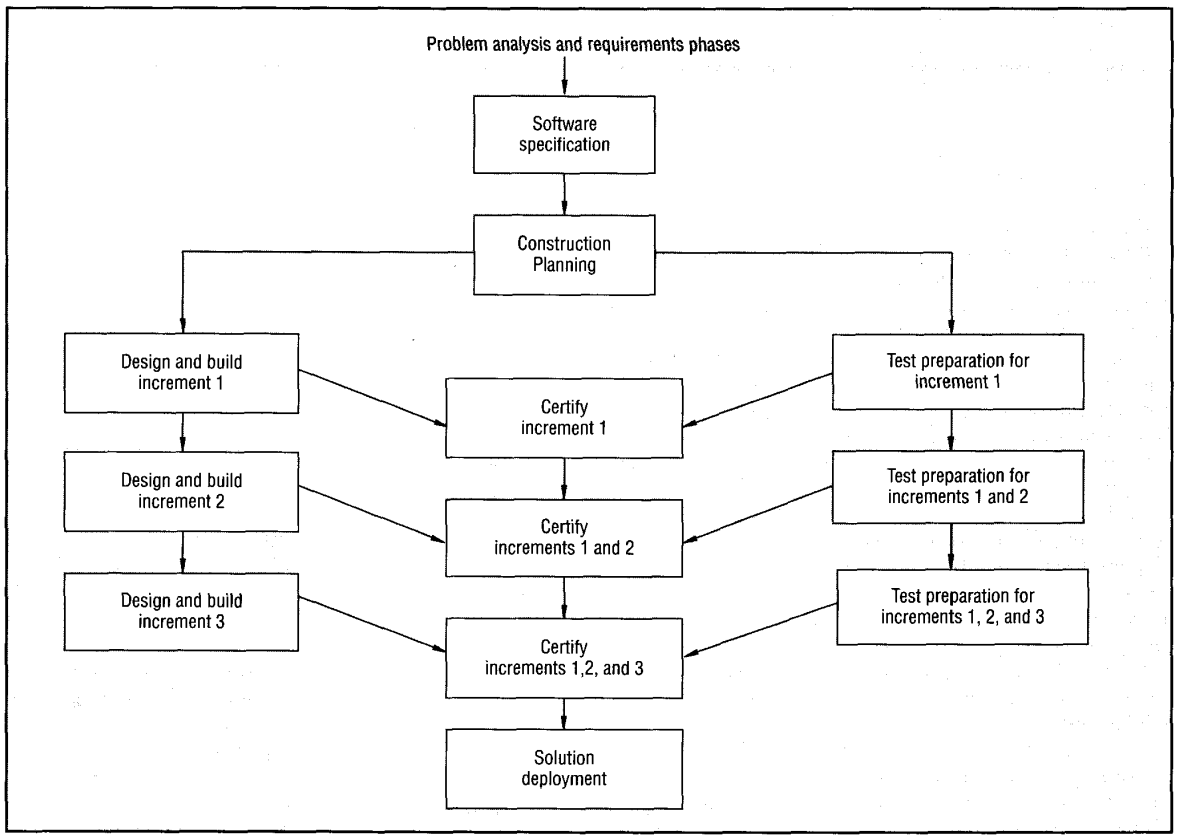

Figure 1. Profile of a three-increment Cleanroom-engineering project.

Specification. The first task is to assemble what is known into a specification document, complete the remaining details, then prepare and publish a formal specification. The first version may be preliminary due to lack of information, but it should still be formal. The specification must be as complete as possible and approved before development begins.

The effort required to prepare the specification depends on how much is known when the decision to develop the software is made. It should be in three parts, which should agree: external specification, internal specification, and expected-usage profile.

The external specification is a user's reference manual. It defines how the software will look and feel from the user's perspective and all the interfaces with the software. The specification should include details on

- the system environment (hardware, peripherals, operating system, related software, and people),
- the application environment (data and use structures),
- initialization and shutdown,
- system use (commands, menus, events, and modes), which must define all stimuli the system can receive from people, computers, and other devices and all responses it will produce,
- performance guidelines (timing and precision), and
- responses to undesired events.

The external specification is written in a language understood by users, but it is not a tutorial. It is not designed to instruct how to use the software; it is intended to define precisely how the software will work. Using only the external specification, someone with appropriate application expertise should be able to use the software with no surprises.

The internal specification is more mathematical. It completely states the mathematical function or, more generally, mathematical relation for which the program implements a rule. This definition is required to implement the program and verify its correctness. It must be implementation-independent so the program architecture can be designed free of preconceptions.

The internal specification augments information in the external specification. For example, while the external specification defines the stimuli the software will act upon and responses it will produce, the internal specification defines the responses in terms of stimuli histories. Specifying the functional relationship between responses and stimuli completely in terms of stimuli histories avoids commitment to implementation details.

Specifying responses this way is usually hard to learn at first because it is natural to use invented abstractions — state data — to represent some portion of the prior stimuli. But as soon as you use state data to define software responses, you begin making implementation commitments.

At the specification stage, you must define *what* is to be done, not *how* to do it. Experience indicates that as soon as you learn to define what is to be done free of implementation details, you can design and implement much better software. (David Parnas recommends traces.[8]) We find using stimuli histories more conve-

nient and natural and therefore easier to teach.

The expected usage profile defines the software's anticipated use. This document primarily guides the preparation of usage tests. To make a valid inference about the software's expected MTTF, you must develop and run tests with stimuli taken from the population of all possible stimuli and in the same proportion as they will be generated when the system is in use.

Statistical testing is a stochastic process. The simplest and best understood stochastic process is the Markov process, which can model the usage of most if not all software systems. In developing a Markov model for expected use, you must define all usage states and estimate the transition probabilities between usage states. This sounds harder than it seems to be in practice. For example, see Jesse Poore's work.[9]

There is no magic in preparing the written specification. The magic is inventing what the software should do to accomplish its mission — a much deeper and harder problem than developing the software. That is why it is so important to use good engineering practices in developing software so the time and attention now being consumed on the easy part of the problem can be redirected to the harder problem of determining what the software should be doing.

Construction plan. This phase determines the development and certification sequence. To do this, you decompose the specification into executable increments. An executable increment can be tested by invoking user commands or supplying other external stimuli.

The criteria to determine the construction sequence include

- the availability of reusable software,
- how much is known about the reliability of the reused software for the expected usage profile,
- increment size (increments should seldom be larger than 10,000 lines), and
- the number of development teams available, which determines the possibilities for parallel development.

Incremental development is not new. The important new idea is the requirement that each increment in the construction plan, including the first, be executable by user commands. This means both that the system must be constructed top-down and that you need write no special testing routines.

It also means that incremental integration testing is done as each new increment is written. And it lets you use all test runs, including the tests on the very first increment, to help estimate the final MTTF. Figure 2 shows a sample construction plan.

When you have decomposed the specification into increments, design, implementation, and testing can begin. These two phases can proceed in parallel.

Your specification must define what is to be done, not how. Experience indicates that as soon as you learn to define what is to be done free of implementation details, you can create much better software.

Design and build. The development team, not an individual engineer, is responsible for the quality of the increments developed. The team uses technologies to construct increments, box structures and stepwise refinement, and functional verification. Development proceeds in three steps:

1. Design each increment top-down, to create a usage hierarchy in three views: black-box, state-box, and clear-box. Verify the correctness of each view.
2. Implement each increment by rigorous stepwise refinement of clear boxes into executable code.
3. Verify that the code performs according to its specification using functional verification arguments.

Box structures. The team uses box structures to create the software's internal design. Box structures view the software from three perspectives:

- The implementation-independent black-box view defines the responses in terms of stimuli histories.
- The data-driven state-box view begins to define implementation details by modifying the black box to represent responses in terms of the current stimuli and state data that represents the stimuli histories.
- The process-driven clear-box view completes the implementation details by modifying the state box view to represent responses in terms of the current stimuli, state data, and invocations of lower level black boxes.

Some advocate the data view, others the process view for designing software. These different points of view cannot be resolved because, in reality, both views are required. Box structures let you define this dualism.

Figure 3 summarizes a design algorithm defined by Mills that uses box structures.[10] The first black box restates the specification that defines all the responses produced by the increment in terms of stimuli histories.

Steps 3 through 5 invent the state data that represents stimuli histories, to preserve referential transparency. The algorithm then determines which of the state data to maintain at this level of the usage hierarchy and which to migrate to a lower level. It is important to migrate state data to the lowest practical level in the usage hierarchy to keep the software's structure under control.

The state-box description is complete when functional relationships exist that define the responses in terms of the current stimuli, the state data being maintained at this level, and stimuli histories for the state data being maintained at lower levels.

Before the team proceeds to define the clear box, it verifies the state-box description by eliminating references to state data in the state-box functions. The result is a derived black-box function that should be the same as the original black-box function.

If the two functions are the same, the team defines the clear box that follows the state box, otherwise it redefines the state-box function to correct the design errors. So the design process is suspended as long as the design contains logic errors. Just as in long division, it is best to fix any error as

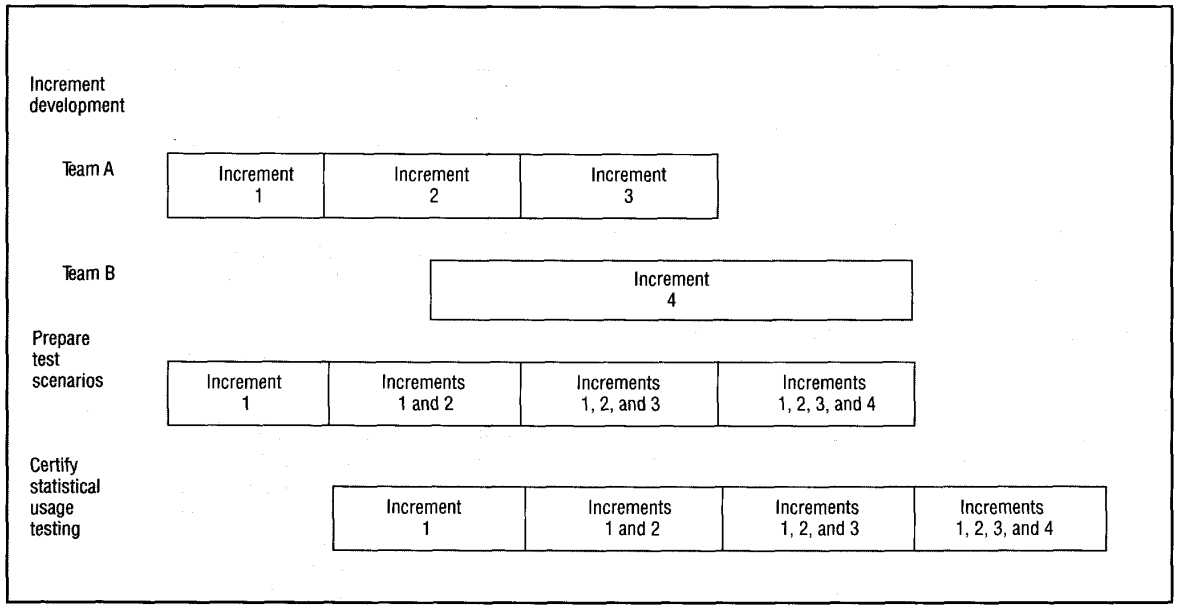

Figure 2. A typical Cleanroom project construction plan.

Define Black Box	1. Define stimuli. 2. Define responses in terms of stimuli histories.
Define State Box	3. Define state data to represent stimuli histories. 4. Select state data to be maintained at this level. 5. Modify black box to represent responses in terms of stimuli and the state data being maintained at this level. 6. Verify state box.
Define Clear Box	7. Record type of reference to state data by stimuli. 8. Define data abstraction for each state data. 9. Modify state box to represent responses in terms of stimuli, this level's state data, and invocations of lower level black boxes. 10. Verify the clear box.

Figure 3. The box-structures algorithm.

soon as possible. It takes less time in the long run.

To design the clear box, the team invents (selects) the data abstractions (like set, stack, and queue) it will use to represent the state data maintained at this level. Then it modifies the state-box function to define responses in terms of current stimuli, state data being maintained at this level, and invocations of lower level black boxes.

When the clear box description is complete, it is verified by eliminating references to lower level black boxes to obtain a derived state-box function that is compared to the original state-box function.

If the original and the derived functions are the same, the design process continues. Otherwise, clear-box design continues until the verification indicates no logic errors.

The design process continues as the team expands each black box until there are no more to be expanded. At that point the design is complete.

Stepwise refinement. Box structures provide a rigorous stepwise refinement algorithm that guides system design in an orderly, logical manner, with natural checkpoints along the way. For example, after step 6 in Figure 3 it is time to evaluate which state data you want to be maintained at this level. This gives you a chance to evaluate the trade-offs in maintaining the state at this level versus migrating it to a lower level. After step 10 you have a chance to evaluate which lower level black boxes to invoke.

The algorithm doesn't make the invention in these two crucial areas for the development team, but it does ensure that all the details following these two inventions are performed correctly with the verification steps. The algorithm also forces the designers and evaluators to focus on the critical software inventions that affect software performance and quality.

The box-structures algorithm is a process that engineers and managers can rely on to invent a high quality, accurate design.

Functional verification. Once the design is complete, the team expands the clear box at each level into code that fully implements the defined rule for the black-box function at that level. Following each expansion, the team uses functional verification to help structure a proof that the expansion correctly implements the specification.

The proof must show that the rule (the program) correctly implements the function (the specification) for the full range of the function and no more. The Linger, Mills, and Witt correctness theorem[3] defines what you must show to prove that a program is equivalent to its specification for each of the structured-programming-language constructs.

The proof strategy is divided into small parts that easily accumulate into a proof for a large program. Experience indicates

that people are able to master these ideas and construct proof arguments for very large software systems.

The Cleanroom development team does not test or even compile the code. It uses a mathematical proof — functional verification — to demonstrate the correctness of the units. Testing and measuring failures by program execution is the responsibility of the certification team.

Certification. In parallel with the development team, the certification team uses the expected-usage profile and the applicable portion of the external specification to prepare test cases and solutions that exercise the increment just developed and the increments developed previously. The team can perform this step in parallel with development because it uses the specification, not the code, to develop tests.

When the development team completes an increment, the certification compiles it, adds it to previous increments, and certifies the software in three steps:

1. It measures T_k, the MTTF of the current version of the software by executing random test cases. T_k is a sample of MTTF for a version of the accumulated increments and $T0$, ..., T_{k-1} (measured previously).

The team compares each test result to a standard; either the result is correct or there was a failure. The cumulative time to failure is an estimate of the MTTF. The team may decide to continue testing by constructing more tests. The new time to failure is another estimate of the MTTF. It uses all estimates of MTTF to predict the MTTF of the next version.

The certification team reports failures to the development team, which makes the fixes. When the development team returns new modules, the certification team compiles a new version, and the measuring process is repeated for the new version of the software.

2. Estimate the reliability for the next version of the software using a certification model and the measured MTTF for each version of the software. The team predicts the MTTF for the next version of the software using the model

$$\mathrm{MTTF}_{k+1} = AB^{k+1}$$

by fitting the data points T_0, ... , T_k to an exponential relationship. The reliability can be calculated from the MTTF. The results of the MTTF estimation can be summarized in a table, as in Table 3, which summarizes data from an actual project.

Table 3.
Results of MTTF estimation.

Version number	Observed MTTF	Predicted reliability	Predicted MTTF	Factor B
0	1.00	—	—	—
1	6.00	—	—	—
2	1.00	.23	.81	0.59
3	16.00	.77	4.38	1.36
4	560.00	.9957	232.62	3.60

During certification, the team should observe the dynamics of change to determine how many more tests are required to certify the software to the required MTTF. B is the factor by which each change increases the MTTF. If B goes below 1, the new version is worse than the previous version. It is desirable that the value of B increase monotonically.

The value of B turns down when failures are encountered late. Failures found early are not expensive in terms of eventually obtaining a high value for MTTF with a reasonable testing budget, but if B drops late in the certification process it will take a large number of tests to achieve the desired MTTF. MTTF, reliability, and testing time (number of test cases) are mathematically related to each other. The team can also calculate confidence bounds on MTTF estimates.

3. Once it has estimated the MTTF for the next version, the team must decide if it wants to

- correct the observed failures and continue to certify the software,
- stop certification because the software has reached the desired reliability for this stage of testing, or
- stop certification and redesign the software because the failure rate is too high or the failures are too serious.

When all the increments are complete and tested, you have a reliable estimate of the software's quality and it can be deployed.

While we don't address the operations and maintenance phases here, we want to point out that

- operations provide actual testing for continued estimates of the MTTF to check against what has been certified during development and
- the maintenance phase will be much simpler for Cleanroom-developed software than for heuristically developed software because of higher quality and the existence of a design and development trail.

Goals. We believe the following are realistic goals for Cleanroom engineering. Our belief comes from observing demonstrations of component practices, including a few demonstrations of the full set of practices. Table 1 summarizes the results of some of these projects.

- Long-term goals (after a team has completed three or four increments): two orders of magnitude (factor of 100) improvement in reliability and one order of magnitude (factor of 10) improvement in productivity.
- Short-term goals (first two or three increments developed by a new Cleanroom team): statistical quality control of development in a pipeline of user-executable increments that accumulate into a system; elimination of debugging by software engineers before independent statistical testing of usage requirements; certification of reliability at delivery; one order of magnitude improvement in reliability; and factor of three improvement in productivity.

Responsible software-development organizations should begin to adopt Cleanroom engineering or some equivalent discipline. An organization always faces risks when it decides to change the way it does business. The best way to manage risk is to identify the risk and determine what actions to take to avoid it or at least minimize its effect.

The potential gains from Cleanroom engineering are enormous compared to the identified risks. The largest risk an organization can take is to decide not to adopt Cleanroom engineering or an equivalent discipline. At the very least, organizations should conduct a trial on at least one or two significant projects.

The cost of continuing to develop failure-laden software with its associated low productivity can at best increase cost and at worst so affect an organization's competitive position that is difficult to remain in business.

Organizations that purchase software should also understand the ramifications of Cleanroom engineering so they can work with their vendors and integrators to ensure that they build high-quality software at an attractive price. Intelligent buyers can have a significant effect on the speed with which developers adopt these superior software-development practices. ❖

References

1. H.D. Mills, "Structured Programming: Retrospect and Prospect," *IEEE Software*, Nov. 1986, pp. 58-66.
2. E.N. Adams, "Optimizing Preventive Service of Software Products," *IBM J. Research and Development*, Jan. 1984.
3. R.C. Linger, H.D. Mills, and B.I. Witt, *Structured Programming: Theory and Practice*, Addison-Wesley, Reading, Mass., 1979.
4. R.C. Linger and H.D. Mills, "A Case Study in Cleanroom Software Engineering: The IBM Cobol Structuring Facility," *Proc. Computer Software and Applications Conf.*, CS Press, Los Alamitos, Calif., 1988.
5. R.W. Selby, V.R. Basili, and F.T. Baker, "Cleanroom Software Development: An Empirical Evaluation," *IEEE Trans. Software Eng.*, Sept. 1987, pp.1.027-1.037.
6. M. Dyer and A. Kouchakdjian, "Correctness Verification: Alternative to Structural Software Testing," *Information and Software Technology*, Jan./Feb. 1990, pp. 53-59.
7. H.D. Mills, M. Dyer, and R.C. Linger, "Cleanroom Software Engineering," *IEEE Software*, Nov. 1986, pp. 19-24.
8. D.L. Parnas and Y. Wang, "The Trace Assertion Method of Module-Interface Specification," *Tech. Rep. 89-261*, Telecommunications Research Inst. of Ontario, Queens Univ., Kingston, Ontario, Canada, 1989.
9. J.H. Poore et al., "A Case Study Using Cleanroom with Box Structures ADL," Tech. Report CDRL 1880, Software Engineering Technology, Vero Beach, Fla., 1990.
10. H.D. Mills, "Stepwise Refinement and Verification in Box-Structured Systems," *Computer*, June 1988, pp. 23-36.

Richard H. Cobb is vice president of Software Engineering Technology. His research interests are software design and development practices and methodologies for improving software quality and developer productivity.

Cobb received a BS in industrial engineering from the University of Cincinnati and an MS in operations research from Rensselaer Polytechnic Institute.

Harlan D. Mills is president of Software Engineering Technology and a professor of computer science at Florida Institute of Technology. His research interests are systems engineering and the mathematical foundations of software engineerring.

Mills received a PhD in mathematics from Iowa State University. He is the recipient of the DPMA Distinguished Information Science Award and the Warnier Prize and is an IEEE fellow.

Address questions about this article to Cobb at SET, 1918 Hidden Point Rd., Annapolis, MD 21401 or Mills at SET, 2770 Indian River Blvd., Vero Beach, FL 32960; CSnet hmills@cs.fit.edu.