Adopting Cleanroom software engineering with a phased approach

by P. A. Hausler
R. C. Linger
C. J. Trammell

Cleanroom software engineering is a theory-based, team-oriented engineering process for developing very high quality software under statistical quality control. The Cleanroom process combines formal methods of object-based box structure specification and design, function-theoretic correctness verification, and statistical usage testing for reliability certification to produce software approaching zero defects. Management of the Cleanroom process is based on a life cycle of development and certification of a pipeline of user-function increments that accumulate into the final product. Teams in IBM and other organizations that use the process are achieving remarkable quality results with high productivity. A phased implementation of the Cleanroom process enables quality and productivity improvements with an increased control of change. An introductory implementation involves the application of Cleanroom principles without the full formality of the process; full implementation involves the comprehensive use of formal Cleanroom methods; and advanced implementation optimizes the process through additional formal methods, reuse, and continual improvement. The AOEXPERT/MVS™ project, the largest IBM Cleanroom effort to date, successfully applied an introductory level of implementation. This paper presents both the implementation strategy and the project results.

Zero or near-zero defect software may seem like an impossible goal. After all, the experience in the first generation of software development has reinforced the seeming inevitability of errors and persistence of human fallibility. Today, however, a new reality in software development belies the first-generation experience. Cleanroom software engineering teams are able to develop software at a level of quality and reliability that would have seemed impossible a few years ago, and are doing so with high productivity.

Cleanroom software engineering is a managerial and technical process for the development of software approaching zero defects with certified reliability.[1,2] The Cleanroom process spans the entire software life cycle; it provides a complete discipline within which software teams can plan, specify, design, verify, code, test, and certify software. The Cleanroom approach treats software development as an engineering process based on mathematical foundations, rather than as a trial-and-error programming process,[3-7] and is intended to produce software with error-free designs and failure-free executions.

In traditional, craft-based software development, errors were accepted as inevitable, and programmers were encouraged to get software into testing quickly in order to begin debugging. Programs were subjected to unit testing and debugging by their authors, then integrated into components, subsystems, and systems for more debugging.

Product use by customers resulted in still more debugging to correct errors discovered in operational use. The most virulent errors were often the result of fixes to other errors,[8] and it was not unusual for software products to reach a steady-state error population, with new errors introduced as fast as old ones were fixed. Today, however, craft-based processes that depend on testing and debugging to improve reliability are understood to be inefficient and ineffective. Experience has shown that craft-based processes often fail to achieve the level of reliability essential to a society dependent on software for the conduct of human affairs.

In the Cleanroom process, correctness is built into the software by development teams through a rigorous engineering process of specification, design, and verification. The more powerful process of team correctness verification replaces unit testing and debugging, and software enters system testing directly, with no execution by development teams. All errors are accounted for from first execution on, with no private unit testing necessary or permitted. Experience shows that Cleanroom software typically enters system testing approaching zero defects and occasionally no defects are found in all testing.

Certification (test) teams are not responsible for "testing in" quality, which is an impossible task, but rather for certifying the quality of software with respect to its specification. Certification is performed by statistical usage testing that produces objective assessments of product quality. Errors, if any, found in testing are returned to the development team for correction. If the quality is not acceptable, the software is removed from testing and returned to the development team for reverification.

The process of Cleanroom development and certification is carried out in an incremental manner. System functionality grows with the addition of successive code increments in a stepwise integration process. When the final increment is added, the system is complete. Because successive increments are elaborating the top-down design of increments already in execution, interface and design errors are rare.

This paper describes key Cleanroom technologies and summarizes quality results achieved by Cleanroom teams. It presents a phased approach to Cleanroom implementation based on the software maturity level of an organization, and summarizes the results of a substantial IBM Cleanroom project (AOEXPERT/MVS*) that successfully applied a phased approach.

Cleanroom perspectives

The Cleanroom software engineering process evolved from concepts developed and demonstrated over the past 15 years by Harlan Mills and colleagues.[3-5,9] Cleanroom practices such as stepwise refinement of procedure and object hierarchies, team verification of correctness, and statistical usage testing, have been successfully applied in commercial and governmental software projects over the past decade. Such practices may not be the rule in software development today, but their use is growing as evidence of their value continues to accumulate. In many cases, software organizations considering a transition to the Cleanroom process have operational practices in place, such as incremental development, structured programming, and team reviews, that support Cleanroom concepts. There are only a few key concepts that must be understood and accepted in a transition to the Cleanroom approach.[10]

Practice based on theory. To be effective, any engineering discipline must be based on sound theoretical foundations. Cleanroom specification, design, and correctness verification practices are based on function theory, whereby programs are treated as rules for mathematical functions subject to stepwise refinement and verification.[4,5] Cleanroom testing and quality certification practices are based on statistical theory, whereby program executions are treated as populations subject to usage-based, stochastic sampling in formal statistical designs.[3,6,11] These theoretical foundations form the basis of a comprehensive engineering process that has been reduced to practice for commercial software development. A growing number of successful, real-world Cleanroom projects have demonstrated the practicality of these methods.

Experienced Cleanroom practitioners and educators have developed comprehensive technology transfer programs based on readily teachable, time-efficient approaches to such Cleanroom technologies as correctness verification and statistical testing. New practitioners will find that processes and tools exist that make the use of these Cleanroom methods highly practical.[12]

Right the first time. A primary objective of the Cleanroom process is to prevent errors, rather than accepting and accommodating errors through institutionalized debugging and rework. For this reason, Cleanroom development teams do not unit test and debug their code. Instead, they rely on rigorous methods of specification and design combined with team correctness verification. These Cleanroom development practices, based on mathematical foundations, yield quality approaching zero defects prior to first execution by certification teams. The purpose of testing in Cleanroom is the certification of software quality with respect to specifications, not the attempt to "debug in" quality.

Management understanding and acceptance of this essential point—that quality will be achieved by design and verification rather than by testing—must be reflected in the development schedule. Time spent in specification and design phases of a Cleanroom development is greater than in traditional projects. Time spent in testing, however, is likely to be less than traditionally required. The manager who wanted to start coding immediately because of the large amount of debugging expected was usually right, but would have difficulty becoming part of a Cleanroom team.

Quality costs less. A principal justification for the Cleanroom process is that built-in quality lowers the overall cost to produce and maintain a product. The exponential growth in the cost of error correction in successive life-cycle phases is well known. Errors found in operational use by customers are typically several orders of magnitude more costly to correct than errors found in the specification phase.[13] The Cleanroom name, taken from the semiconductor industry where a literal cleanroom exists to prevent introduction of defects during hardware fabrication, is a metaphor that reflects this understanding of the cost-effectiveness of error prevention. In the Cleanroom process, incremental development and extensive team review and verification permit errors to be detected as early as possible in the life cycle. By reducing the cost of errors during development and the incidence of failures during operation, the overall life-cycle cost of Cleanroom software can be expected to be far lower than industry averages. For example, the IBM COBOL Structuring Facility product, developed using Cleanroom techniques, has required only a small fraction of its maintenance budget to be consumed during years of field use.

Cleanroom project schedules have equaled or improved upon traditional development schedules.[14–16] In fact, productivity improvements of factors ranging from one and one-half to five over traditional practices have been observed.[15–18] Experienced Cleanroom teams become remarkably efficient at writing clear specifications, simplifying and restricting designs to easily verifiable patterns, and performing correctness verification. Cleanroom is not a more time-consuming development process, but it does place greater emphasis on design and verification to avoid waste of resources in debugging and rework.

A primary objective of the Cleanroom process is to prevent errors.

Cleanroom quality results

As summarized in Table 1, first-time Cleanroom teams in IBM and other industrial and governmental organizations have reported data on close to a million lines of Cleanroom-developed software. The code exhibits a weighted average of 2.3 errors per thousand lines of code (errors/KLOC) in testing.[2,15–19] This error rate represents all errors found in all testing, measured from first-ever execution through test completion. That is, it is a measure of residual errors remaining following correctness verification by development teams, who do not execute the software. The projects represent a variety of environments, including batch, distributed, cooperative, and real-time systems and system parts, and a variety of languages, including microcode, C, C++, JOVIAL, FORTRAN, and PL/I.

Traditionally developed software does not undergo correctness verification, but rather enters unit testing and debugging directly, followed by more debugging in function and system testing

Table 1 Cleanroom project results

Year	Project	Quality and Productivity
1987	IBM Flight Control: Helicopter Avionics System Component 33 KLOC (JOVIAL)	• Certification testing failure rate: 2.3 errors/KLOC • Error-fix reduced 5X • Completed ahead of schedule
1988	IBM Cobol Structuring Facility: Product for automatically restructuring COBOL programs 85 KLOC (PL/I)	• IBM's first Cleanroom product • Certification testing failure rate: 3.4 errors/KLOC • Productivity 740 LOC/PM, 5X improvement • 7 errors in first 3 years of use; all simple fixes
1989	NASA Satellite Control Project 1 40 KLOC (FORTRAN)	• Certification testing failure rate: 4.5 errors/KLOC • 50% improvement in quality • Productivity 780 LOC/PM • 80% improvement in productivity
1990	Martin Marietta: Automated documentation system 1.8 KLOC (FOXBASE)	• First compilation: no errors found • Certification testing failure rate: 0.0 errors/KLOC (no errors found)
1991	IBM System Software First increment 0.6 KLOC (C)	• First compilation: no errors found • Certification testing failure rate: 0.0 errors/KLOC (no errors found)
1991	IBM AOEXPERT/MVS™ Product 107 KLOC (mixed languages)	• Testing failure rate: 2.6 errors/KLOC • Productivity 486 LOC/PM • No operational errors from Beta test sites
1991	IBM Language Product First increment 21.9 KLOC (PL/X)	• Testing failure rate: 2.1 errors/KLOC
1991	IBM Image Product Component 3.5 KLOC (C)	• First compilation: 5 syntax errors • Certification testing failure rate: 0.9 errors/KLOC
1992	IBM Printer Application First increment 6.7 KLOC (C)	• Certification testing failure rate: 5.1 errors/KLOC
1992	IBM Knowledge Based System Application 17.8 KLOC (TIRS™)	• Testing failure rate: 3.5 errors/KLOC
1992	NASA Satellite Control Projects 2 and 3 170 KLOC (FORTRAN)	• Testing failure rate: 4.2 errors/KLOC
1993	University of Tennessee: Cleanroom tool 20 KLOC (C)	• Certification testing failure rate: 6.1 errors/KLOC
1993	IBM 3490E Tape Drive 86 KLOC (C)	• Certification testing failure rate: 1.2 errors/KLOC
1993	IBM Database Transaction Processor First increment 21.5 KLOC (JOVIAL)	• Testing failure rate: 2.4 errors/KLOC • No design errors, all simple fixes
1993	IBM LAN Software First increment 4.8 KLOC (C)	• Testing failure rate: 0.8 errors/KLOC
1993	IBM Workstation Application Component 3.0 KLOC (JOVIAL)	• Testing failure rate: 4.1 errors/KLOC
1993	Ericsson Telecom AB Switching Computer OS32 Operating System 350 KLOC (PLEX, C)	• Testing failure rate: 1 error/KLOC • 70% improvement in development productivity • 100% improvement in testing productivity

	NOTE: All testing failure rates are measured from first-ever execution.	KEY: KLOC = thousand lines of code
		PM = person month
		X = (mathematical) times

following. Measured from first execution, traditional software typically exhibits 25 to 35 or more errors per thousand lines of code.[20] First-time Cleanroom development teams can produce software with quality levels at test entry at least an order of magnitude better than traditionally developed software. The following summaries of three selected projects from Table 1 illustrate the results achieved.

IBM COBOL Structuring Facility. The COBOL Structuring Facility, which consisted of 85 KLOC of PL/I code, was the first Cleanroom product in IBM. It employs proprietary, graph-theoretic algorithms to automatically transform unstructured COBOL programs into a functionally equivalent, structured form for improved maintainability. Relentless design simplification in the Cleanroom process often results in systems that are small for their functionality. For example, the Cleanroom-developed prototype of the COBOL Structuring Facility, independently estimated at 100 KLOC, was developed using just 20 KLOC.

Comparable to a COBOL compiler in complexity, the product experienced 3.4 errors/KLOC in all statistical testing, measured from the first execution. Six months of intensive beta testing at a major aerospace corporation resulted in no functional equivalence errors ever found.[21] Just seven minor errors were reported in the first three years of field use, requiring only a small fraction of the maintenance budget associated with traditionally developed products of similar size and complexity. The product was developed and certified by a team averaging six members, with productivity five times the IBM averages.[16]

IBM 3490E tape drive. The 3490E tape drive is a real-time, embedded software system developed by a five-person team in three increments of C design with a code total of 86 KLOC. It provides high-performance tape cartridge support through a multiple processor bus architecture that processes multiple real-time input and output data streams. The product experienced 1.2 errors/KLOC in all statistical testing. To meet an urgent business need, the third increment was shipped straight from development to the hardware and software integration team with no testing whatsoever. Customer evaluation testing with live data by the integration team resulted in no errors being found.

In a comparison experiment, the project team subjected a selected module to both unit testing and correctness verification. Development of execution scaffolding, definition and execution of test cases, and checking of results required one- and one-half person-weeks of effort and resulted in the detection of seven errors. Correctness verification of the same program by the development team required one and one-half hours, and resulted in the detection of the same seven errors, plus three additional errors.[1]

Ericsson OS32 operating system. Ellemtel Telecommunications Systems Laboratories is completing a 350 KLOC operating system for a new family of switching computers for Ericsson Telecom AB. The code is written in PLEX and C. The 73-person, 33-month Cleanroom project experienced productivity improvements of 70 percent and 100 percent in development and testing, respectively, and the product averaged under one error/KLOC in all testing. Project management reported that an average of less than one person-hour was required to detect an error in team reviews, compared to an average of 17.5 person-hours to detect an error in testing. The project allocated two days per week to prepare and conduct team reviews. The product team was honored by Ericsson as the single project that had contributed the most to the company in 1993.[18]

Cleanroom technologies

In the Cleanroom process, the objective of the development team is to deliver software to the test team that approaches zero defects; the objective of the test team is to scientifically certify the quality of software, not to attempt to "test in" quality. These objectives are achieved through management and technical practices based on the technologies of incremental development, box structure specification and design, correctness verification, and statistical quality certification.

Incremental development. Management planning and control in Cleanroom is based on development and certification of a *pipeline of increments* that represent operational user function, accumulate top-down into the final product, and execute in the system environment.[22] Following specification of required external system behavior, an incremental development plan is created to define schedules, resources, and functional content of a series of code increments to be developed and certified. The initial increment contains stubs (small placeholder programs) that stand in for later increments and permit early execution of the code. The ultimate functionality of the code that will replace the stubs is fully defined in subspecifications for team verification of each increment prior to testing. As incremental development progresses, stubs are replaced by corresponding code increments, possibly containing stubs of their own, in a stepwise system integration process. When the final increment is integrated, the system is complete and no stubs remain.

When the final increment is integrated, the system is complete.

As each increment is integrated, the evolving system of increments undergoes a new step in statistical usage testing for quality certification. Statistical measures of quality provide feedback for reinforcement or improvement of the development process as necessary. Early increments can serve as system prototypes, providing an opportunity to elicit feedback from customers to validate requirements and functionality. As inevitable changes occur, incremental development provides a framework for revising schedules, resources, and function, and permits changes to be incorporated in a systematic manner.

Box structure specification and design. Box structures provide a stepwise refinement and verification process based on *black box*, *state box*, and *clear box* forms for defining system behavior and deriving and connecting objects comprising a system architecture.[5,23] Boxes are object-based, and the box structure process provides a systematic means for developing object-based systems.[24] Specifically, the black box form is a specification of required behavior of a system or system part in all circumstances of use, defined in terms of stimuli, responses, and transition rules that map stimulus histories to responses. The state box form is refined from and verified against the black box, and defines encapsulated state data required to satisfy black box behavior. The clear box form is refined from and verified against the state box, and defines procedural design of services on state data to satisfy black box behavior, often introducing new black boxes at the next level of refinement. New black boxes (specifications) are similarly refined into state boxes (state designs) and clear boxes (procedure designs), continuing in this manner until no new black boxes are required. Specification and design steps are interleaved in a seamless, integrated hierarchy affording complete verifiability and traceability.

Box structures isolate and separate the creative definition of behavior, data, and procedures at each level of refinement. They incorporate the essential property of *referential transparency*, such that the information content of an abstraction, for example, a black box, is sufficient to define and verify its refinement into state and clear box forms without reference to other specification parts. Referential transparency is crucial to maintaining intellectual control in complex system developments. Box-structured systems are developed as *usage hierarchies* of boxes, where each box provides services on encapsulated state data, and where its services may be used and reused in many places in the hierarchy as required. Box-structured systems are developed according to the following principles:[25] (1) all data to be defined and retained in a design are encapsulated in boxes, (2) all processing is defined by sequential and concurrent use of boxes, and (3) each use of a box occupies a distinct place in the usage hierarchy of the system. Clear boxes play an important role in the hierarchy by defining and controlling the correct operation of box services at the next level of refinement.

Correctness verification. As noted, in the Cleanroom process, verification of program correctness in team reviews replaces private unit testing and debugging by individuals. Debugging is an inefficient and error-prone process that undermines the mental discipline and concentration that can achieve zero defects. The intellectual control of software development afforded by team verification is a strong incentive for the prohibition against unit testing. "No unit testing" does not, however, mean "no use of the machine." It is essential to use the machine for experimentation, to evaluate algorithms, to benchmark performance, and to understand and document the semantics of interfacing software.

These exploratory activities are entirely consistent with the Cleanroom objective of software that is correct by design.

Elimination of unit testing motivates tremendous determination in developers to ensure that the code they deliver for independent testing is error-free on first execution. But there is a deeper reason to adopt correctness verification—it is more efficient and effective than unit testing. Programs of any size can contain an essentially infinite number of possible execution paths and states, but only a minute fraction of those can be exercised in unit testing. Correctness verification, however, reduces the verification of programs to a finite and complete process.

In more detail, all clear box programs are composed of nested and sequenced control structures, such as sequence, IF-THEN-ELSE, WHILE-DO, and their variants. Each such control structure is a rule for a mathematical function,[9] that is, a mapping from a domain or initial state to a range or final state. The function mapping carried out by each control structure can be documented in the design as an *intended function*. For correctness, each control structure must implement the precise mapping defined by its intended function. The Correctness Theorem[4] shows that verification of sequence, IF-THEN-ELSE, and WHILE-DO structures requires checking exactly one, two, and three *correctness conditions*, respectively. While programs can exhibit an essentially infinite number of execution paths and states, they are composed of a finite number of control structures, and their verification can be carried out in a finite number of steps by checking each correctness condition in team reviews. Furthermore, verification is complete, that is, it deals with all possible program behavior at each level of refinement. The verification process defined by the Correctness Theorem accounts for all possible mappings from the domain to the range of each control structure, not just a handful of mappings exercised by particular unit tests. For these reasons, verification far surpasses unit testing in effectiveness.

Statistical quality certification. In the Cleanroom process, statistical usage testing for certification replaces coverage testing for debugging. Testing is carried out by the certification team based on anticipated usage by customers. *Usage probability distributions* are developed to define system inputs for all aspects of usage, including nominal scenarios as well as error and stress situations. The distributions can be organized into probabilistic state transition matrices or formal grammars. Test cases are generated based on random sampling of usage distributions. The correct output for each test input is specified with reference to an oracle, that is, an independent authority on correctness, typically the software specification. System reliability is predicted based on analysis of test results by a *formal reliability model*, and the development process for each increment is evaluated based on the extent to which the reliability results attained objectives. In effect, statistical usage testing is based on a *formal statistical design*, from which statistical inferences of software quality and reliability can be derived.[3,11,26]

Debugging is an inefficient and error-prone process.

Coverage testing can provide no more than anecdotal evidence of reliability. Thus, if many errors are found, does that that mean that the code is of poor quality and many errors remain, or that most of the errors have been discovered? Conversely, if few errors are found, does that mean that the code is of good quality, or that the testing process is ineffective? Statistical testing provides scientifically valid measures of reliability, such as mean-time-to-failure (MTTF), as a basis for objective management decision-making regarding software and development process quality.

Empirical studies have demonstrated enormous variation in the failure rates of errors in operational use.[8] Correcting high-failure-rate errors has a substantial effect on MTTF, while correcting low-failure-rate errors hardly influences MTTF at all. Because usage-based testing exercises software the way users intend to use it, high-frequency, virulent errors tend to be found early in testing. For this reason, statistical usage testing is more effective at improving software reliability than is coverage testing. Statistical testing also provides new management flexibility to certify software quality for varying conditions of use and stress, by developing special usage probability distributions for such situations. For example, the reliability of infrequently used functions with severe consequences of failure can be independently measured and certified.

Adopting the Cleanroom process

Rigorous and complete Cleanroom implementation permits development of very high quality software with scientific certification of reliability. However, substantial gains in quality and productivity have also occurred in partial Cleanroom implementations.[15,18] Evidence suggests that a phased approach to implementation can produce concrete benefits and afford increased management control. The phased approach, combined with initial Cleanroom use on selected demonstration projects, provides a systematic management process for reducing risk in technology transfer. Three implementation phases can be defined and sequenced in a systematic technology transfer process. The idea is to first introduce fundamental Cleanroom principles and several key technologies in an *introductory implementation*. As team experience and confidence grows, increased precision and rigor can be achieved in a *full implementation* of Cleanroom technology. Finally, an *advanced implementation* can be introduced to optimize the Cleanroom process. Of course, a particular Cleanroom implementation can combine elements from various phases as necessary and appropriate for the project environment.

Introductory implementation. Key aspects of an introductory implementation are summarized in the first row of Table 2. The fundamental idea is to shift from craft-based to engineering-based processes. The development objective shifts from defect correction in unit testing to defect prevention in specification, design, and verification. As experience grows, developers learn they can write software that is right the first time, and a psychological change occurs, from expecting errors to expecting correctness. At the same time, the testing objective shifts from debugging in coverage testing to reliability certification in usage testing. Because Cleanroom code is of high quality at first execution, testers learn that little debugging is required, and they can concentrate on evaluating quality. A management opportunity exists to leverage these technology shifts to develop systems on schedule with substantial improvement in quality and reduction in life-cycle costs.

All development and testing is accomplished by small teams. Team operations provide opportunities for cross-training and a ready forum for discussion, review, and improvement. All work products undergo a team-based peer review to ensure the highest level of quality. The size and number of teams varies according to resource availability, skill levels, and project size and complexity. Teams are organized during project planning and their membership should remain stable throughout development. Cooperative team behavior that leverages individual expertise is a key factor in successful Cleanroom operations.

In any Cleanroom implementation, zero-defect software is an explicit design goal, and measured performance at a target level is an explicit reliability goal. The Cleanroom practices necessary to achieve these objectives require substantial management commitment. Because compromises in process inevitably lead to compromises in quality, it is crucial for managers to understand Cleanroom fundamentals—the philosophy, process, and milestones— and demonstrate unequivocal support. Management commitment is essential to successful introduction of the Cleanroom process.

A key aspect of customer interaction is to shift from a technology-driven to a customer-driven approach, whereby system functional and usage requirements are subject to extensive analysis and review with customers to clearly understand their needs. Maintaining customer involvement in specification and certification helps avoid developing a system that approaches zero defects but provides the wrong functionality for the user.

Unlike the traditional life cycle of sequential phases, the Cleanroom life cycle is based on incremental development. In an introductory implementation, a project is scheduled and managed as a pipeline of increments for development and testing. Functional content and sequencing of increments is typically based on a natural subdivision of system functions and their expected usage. Successive increments should implement user function, execute in the system environment, and accumulate top down into the final product. This incremental strategy supports testing throughout development rather than at completion. It also integrates system increments in multiple steps across the life cycle, to avoid risks of single-step integration of all system components late in a project when little time or resources remain to deal with unforeseen problems.

Management commitment is essential to successful introduction.

In an introductory implementation, a black box specification is written that precisely defines required system functionality in terms of inputs, outputs, and behavior in all possible circumstances of use, including correct and incorrect use. The specification focuses on required system behavior from the user's viewpoint and does not describe implementation details. At this level, specifications are generally expressed in an outer syntax of specification structures, such as tabular formats or variants of Box Description Language (BDL),[5] and an inner syntax of natural language. Cleanroom specifications are important working documents that drive design and certification activities, and they must be kept current for effective team operations. Definition of system user's guides is initiated in parallel with specifications, for elaboration and refinement throughout the development.

In the design process of an introductory implementation, state and clear box concepts are implemented using sound software engineering practices, including stepwise refinement, structured programming, modular design, information hiding, and data abstraction. Successive increments are specified and designed top-down through stepwise refinement, with frequent team review and discussion of design strategies.[8] Stepwise refinement requires substantial look-ahead and analysis, as successive design versions are developed and revised. In this process, a relentless team drive for *design simplification* can result in substantial reductions in the size and complexity of systems, for more efficient correctness verification and subsequent maintenance.

Design with intended functions is a fundamental practice at the introductory level. High-level intended functions originate in system specifications, and are refined into control structures and new intended functions. Expressed primarily in natural language, intended functions are recorded as comments attached to key control structures in designs. Intended functions precisely define required behavior of their control structure refinements. Behavior is defined in functional, nonprocedural descriptions of the derivation of output data from input data. Intended function refinements are expressed in a restricted set of single-entry, single-exit control structures with no side effects, such as sequence, IF-THEN-ELSE, WHILE-DO, and their variants. Each control structure may contain additional intended functions for further refinement. This stepwise specification and design process continues until no further intended functions remain to be elaborated. Intended functions provide a precise road map for designers in refining design structures, and are essential to team verification reviews.

The *last intellectual pass* through a design occurs in team-based correctness verification, another fundamental practice in an introductory implementation. At the design level, verification reviews prove correctness of program control structures, unlike traditional code inspections that trace program flow paths to look for errors. The verification process is based on reading and abstracting the functionality of control structures in designs and comparing the abstractions with specified intended functions to assess correctness. Team members read, discuss, evaluate, and indicate agreement (or not) that designs are correct with respect to their intended behavior. If changes are required, the team must review and verify the modifications before the designs can be considered finished. Verification reviews provide team members with deep understandings of designs and their correctness arguments. Reviews are conducted with the understanding that the entire team is responsible for correctness. Ultimate successes are team successes, and failures are team failures. All specifications and designs are subject to team review, without exception. Fol-

Table 2 A phased implementation for Cleanroom practice

Cleanroom Practice / Implementation	Management and Team Operations	Customer Interaction	Incremental Development	System Specification
Introductory Implementation	• Document an introductory Cleanroom process. • Shift from craft-based to engineering-based processes. • Shift from defect correction in unit testing to defect prevention in specification, design, and verification. • Shift from debugging in coverage testing to quality certification in usage testing. • Shift from individual to small team operations with team review of all work products. • Establish Cleanroom projects and provide commitment, education, and recognition to teams. • Develop to schedule with substantial quality improvement and life cycle cost reduction.	• Shift from technology-driven to customer-driven development. • Analyze and clarify functional requirements with customers to develop functional specifications. • Analyze and clarify usage requirements with customers to develop usage specifications. • Review and validate functional and usage specifications with customers. • Revise functional and usage specifications as necessary for changing requirements.	• Shift from a sequential (waterfall) to an incremental process. • Define increments that implement user function, execute in the system environment, and accumulate top down into the final product. • Define and evolve an incremental development plan for schedules, resources, and increment content. • Carry out scheduled incremental development and testing with stepwise integration of increments.	• Shift from informal, throwaway specifications to precise, working specifications kept current through the project life cycle. • Define specifications of system boundaries, interfaces, and required external behavior in all possible circumstances of use, including correct and incorrect use. • Express specifications in systematic forms such as tables that define required behavior in natural language. • Develop and evolve system user's guides in parallel with specifications.
Full Implementation	• Document a full Cleanroom process. • Increase development rigor with box structure specification, design, and correctness verification. • Increase testing rigor with scientific measures of reliability. • Establish larger Cleanroom projects as teams of small teams with experienced leaders from previous projects. • Develop to schedule with substantial quality and productivity improvement and life cycle cost reduction.	• Educate customers in Cleanroom to increase value, cooperation, and responsiveness to customer needs. • Review black box functional specifications with customers to support increased rigor in specification. • Review usage specifications with customers to support increased rigor in statistical usage testing. • Provide customers with prototypes and accumulating increments for evaluation and feedback.	• Define increments to incorporate early availability of important functions for customer feedback and use. • Rapidly revise incremental plans for new requirements and actual team performance, and respond to schedule and budget changes.	• Develop prototypes as necessary to validate customer requirements and operating environment characteristics. • Define black box specifications in systematic structures such as transition tables expressed in conditional rules and precise natural language.
Advanced Implementation	• Document an advanced Cleanroom process. • Establish a Cleanroom Center of Competency to monitor Cleanroom technology and train and consult with teams. • Establish Cleanroom projects across the organization led by experienced Cleanroom practitioners. • Develop to schedule with substantial quality and productivity improvements and life cycle cost reduction, even in emergency and adverse circumstances.	• Assist customers in leveraging the quality of Cleanroom-developed software for competitive advantage. • Contract with customer for reliability warranties based on certification with agreed usage distributions and reliability models. • Establish cooperative processes with customers for recording operational system usage to calibrate and improve reliability certification.	• Incorporate comprehensive reuse analysis and reliability planning in incremental development plans. • Plan increment content to manage project risk by early development of interface dependencies, critical functions, and performance-sensitive processes.	• Incorporate advances in formal specification methods into local practices. • Develop guidelines for specification formats and conventions based on team experience. • Apply mathematical techniques in black box specifications to define complex behavior with precision. • Express black box specifications where appropriate with specification functions and abstract models. • Develop a specification review protocol for team reviews based on team experience.

	System Design and Implementation	Correctness Verification	Statistical Testing and Reliability Certification	Process Improvement
	• Shift from programming by aggregation of statements to design by stepwise refinement of specifications. • Refine specifications into structured, modular designs using good software engineering practices with substantial look ahead and analysis. • Express designs in control structures and case-structured intended functions expressed in natural language. • Conduct frequent team development reviews to communicate, simplify, and improve evolving designs. • Conduct execution experiments to document the system environment and semantics of interfacing software.	• Shift from unit testing by individuals to correctness verification by teams. • Shift from path tracing in code inspections to functional analysis in verification reviews. • Conduct demonstration verification reviews to set expectations and train teams. • Verify all control structures in team reviews by reading, function abstraction, and comparison to intended functions. • Verify all design changes in team reviews and deliver verified increments to testing for first execution.	• Shift from coverage testing to usage testing. • Define high-level usage distributions in systematic structures such as hierarchical decision trees. • Develop/acquire test cases from a user perspective based on system specifications and usage distributions. • Evaluate quality of each increment through analysis of measures such as failure rates and severity levels. • Return low-quality increments to development for additional design and reverification.	• Shift from informal review of lessons learned to a systematic, documented improvement process. • Measure team productivity, quality, and cost, and analyze for process improvements. • Document improvements to the introductory implementation based on lessons learned from each increment. • Improve or sustain the development process based on quality results of increment testing. • Assess customer satisfaction with Cleanroom-developed systems for process improvements.
	• Refine black boxes (specifications) into state boxes (data designs) and state boxes into clear boxes (procedure designs) and new black boxes. • Define state boxes in data designs and systematic structures such as transition tables expressed in conditional rules and precise natural language. • Define clear boxes in control structures and intended functions expressed in conditional rules and precise natural language. • Encapsulate system data in boxes and define processing by use of box services. • Identify opportunities for reuse of system components.	• Improve introductory practices through increased precision and formality in verification reviews. • Improve verification by introducing mental proofs of correctness based on box structure theory and Correctness Theorem correctness conditions. • Document and reuse proof arguments for recurring design patterns. • Simplify and standardize designs where possible to reduce proof reasoning.	• Establish reliability targets and conduct statistical usage testing for reliability certification. • Define usage probability distributions for all circumstances of use in formal grammers or state transition matrices. • Define alternative distributions for special environments and critical and unusual usage. • Use automated generators to create test cases randomized against usage probability distributions. • Use reliability models to produce statistical reliability measures based on analysis of test results.	• Document improvements to the full implementation based on team decisions in process reviews after each increment. • Use baseline measurements from introductory projects to set quality and productivity objectives. • Improve or sustain the development process based on reliability measurements of each increment. • Conduct causal analysis of failures found in testing and use to identify process areas for improvement. • Conduct surveys of customer satisfaction with Cleanroom-developed systems for process improvement.
	• Incorporate advances in formal design methods into local practices. • Use box structures to document the precise semantics of interfacing software. • Develop guidelines for design formats and conventions based on team experience. • Apply mathematical techniques in state and clear box designs to define complex behavior with precision. • Develop a design review protocol for team development reviews based on team experience. • Establish libraries of reusable, certified designs.	• Incorporate advances in formal verification methods into local practices. • Use trace tables as necessary to support mental proofs of correctness. • Document written proofs of correctness as required for critical system functions. • Develop verification protocols and extended proof rules for application-, language-, and environment-specific semantics.	• Incorporate advances in scientific software certification methods into local practices. • Apply experience of prior Cleanroom projects and customers in setting reliability targets. • Employ usage analysis to validate functional specifications and plan increment content. • Use automated tools to generate self-checking test cases. • Collect customer usage data to track conformance of usage distributions to actual field use. • Apply and evaluate multiple reliability models for best prediction of system reliability in the development environment.	• Use the full rigor of statistical process control to analyze team performance. • Compare team performance with locally-defined process control standards for performance. • Use error classification schemes to improve specific Cleanroom practices in specification, design, verification, and testing.

lowing verification, increments are delivered to the test team for first execution.

In an introductory implementation, usage testing based on external system behavior replaces coverage testing based on design internals. Usage information is collected by analyzing functional specifications and surveying prospective users (where users may be people or other programs). Based on this information, a high-level usage profile is developed, including nominal scenarios of use, as well as error and stress situations. A usage profile can be recorded in systematic structures such as hierarchical decision trees that embody possible usage patterns in compact form. Next, test scenarios are defined based on the usage profile. The idea is that the test cases represent realistic scenarios of user interaction, including both correct and incorrect usage. For example, if particular system functions are used frequently in particular patterns with occasional user mistakes, this usage should be reflected in the test suite. At this stage, the usage profile may not be extremely precise or detailed, but it does contain sufficient information for the test team to generate realistic test cases.

The effectiveness of the development process is measured by system performance in testing with respect to predetermined quality standards, such as failure rates and severity levels. (More precise statistical measures, such as MTTF and improvement ratio, are introduced in the full implementation.) If test results show that the development process is not meeting quality objectives, testing ceases and the code is removed from the machine for redevelopment and reverification by the development team.

Process improvement is a fundamental activity in an introductory implementation. The idea is to shift from informal discussions of lessons learned to a systematic, documented improvement process. Baseline measurements of fundamental project characteristics, such as quality, productivity, and cost, provide a basis for assessing progress and making improvements. The quality results of usage testing can guide changes to the development process. In addition, customer satisfaction with Cleanroom-developed systems can highlight process areas requiring improvements.

Full implementation. Introductory Cleanroom implementation establishes a framework for maturing the process to a full implementation. As summarized in the second row of Table 2, full implementation adds rigor to practices established in the introductory phase through formal methods of box structure specification and design, correctness verification, statistical testing, and reliability certification. For a Cleanroom project of substantial size and complexity, a *team-of-teams* approach can be applied, whereby the hierarchical structure of the system under development forms the basis for organizing, partitioning, and allocating work among a corresponding hierarchy of small teams.

An opportunity exists for more extensive customer interaction in a full Cleanroom implementation. Customers can be provided with education on Cleanroom practices to improve the effectiveness of functional and usage specification analysis and review. In addition, prototypes and accumulating increments can be provided to customers for evaluation and feedback.

Managers and team leaders can leverage Cleanroom experience into additional flexibility in incremental development to accommodate changing requirements, and shortfalls and windfalls in team performance within remaining schedule and budget. Increment planning can emphasize early development of useful system functionality for customer feedback and operational use.

In specification and design, prototyping and experimentation are encouraged to clarify and validate requirements, and to understand and document semantics of interfacing software. The formal syntax and semantics of box structures are used for black, state, and clear box refinements. Black boxes and state boxes are recorded in an outer syntax of formal structures, such as transition tables, with inner syntax expressed in precise *conditional rules*, often given as *conditional concurrent assignments* combined with precise natural language. In clear box design, intended functions are recorded at every level of refinement, expressed in conditional concurrent assignments and precise natural language.

A box-structured system is specified and designed as a hierarchy of boxes, such that appropriate system data are encapsulated in boxes, processing is defined by using box services, and every use of a box service occupies a distinct place in the hierarchy. Box structures promote early identification of common services, that is, reusable objects, that can simplify development and improve productivity. Duplication of effort is avoided when team members have an early awareness of opportunities for use and reuse of common services. Rigorous team verification reviews are conducted for all program structures, using *mental proofs of correctness* based on box structure theory and the correctness conditions of the Correctness Theorem.

Statistical testing involves a more complete and experimentally valid approach than in an introductory implementation. Reliability objectives are established and extensive analysis of anticipated system usage is carried out. Comprehensive specifications of the population of possible system inputs are defined in usage probability distributions recorded in formal grammars or state transition matrices. Automated tools are used to randomly generate test cases from the distributions, and the correct output for each test input is defined based on the system specification. For example, the IBM Cleanroom Certification Assistant (CCA)[27] automates elements of the statistical testing process based on a formal grammar model for usage probability distributions. It contains a Statistical Testcase Generation Facility for compiling distributions (expressed in a Usage Distribution Language) and creating randomized test cases. Reliability models are employed to measure system reliability based on test results, and the development process for each increment is evaluated based on the extent to which reliability results meet objectives. The CCA provides an automated reliability model, the Cleanroom Certification Model, that analyzes test results to compute MTTF, improvement ratio, and other statistical measures. Alternative distributions are often employed to certify the reliability of special aspects of system behavior, for example, infrequently used functions that exhibit high consequences of failure.

Process improvement is established through reviews, following completion of each increment, to incorporate team recommendations into the documented Cleanroom process. Causal analysis of failures and comprehensive customer surveys can provide additional insight into process areas requiring improvement.

Advanced implementation. Key elements of an advanced implementation are summarized in the third row of Table 2. At this level of experience, the Cleanroom process is optimized for the local environment and continually improved through advances in the software engineering technology. A Cleanroom center of competency can be established, staffed by expert practitioners to monitor advances in Cleanroom technology and provide training and consultation to project teams. The Cleanroom process can be scaled up to ever larger projects and applied across an organization. An opportunity exists to achieve Cleanroom quality, productivity, and cost improvements even in emergency and adverse system developments.

Product warranties may be possible in customer contracts, based on certification with usage distributions and reliability models agreed to by both parties. In the future, a capability for developing software with warranted reliability could become a major differentiating characteristic of software development organizations. Customers can benefit by capturing actual usage from specially instrumented versions of Cleanroom-developed systems, to permit test teams to improve the accuracy of usage distributions employed in certification.

Incremental development can be used to manage project risk through early development of key interfaces with pre-existing software, important user functions, and performance-sensitive components. Increments can also be defined to isolate and reduce dependence on areas of incomplete or volatile requirements, and to focus on early initiation of complex, long-lead-time components. Advanced incremental development also includes systematic reuse and reliability planning,[28] facilitated by such tools as the Cleanroom Reliability Manager.[29] In this approach, libraries of reusable components are searched for functions identified in specification and top-level design. If the reliability of candidate components is not known, statistically valid experiments are conducted to estimate reliability. If reliability of a candidate component has previously been certified, the usage profile used in that certification is compared with the new usage profile to determine if the previous certification is valid for the new use. Once reliability estimates exist for new and reused components, an estimate of total system reliability is generated through calculations based on top-level transition probabilities between subsystems. The results of this analysis are used to set reliability requirements for components, evaluate the viability of component reuse, and factor reliability risks into increment planning.

An advanced use of box structure specification involves formal mathematical and computer science models appropriate to the application. Formal black box and state box outer syntax used in full Cleanroom implementation is combined with formal inner syntax expressed as propositional logic, predicate calculus, algebraic function composition, BNF (Backus Naur form) grammars, or other formal notation that affords a clear and concise representation of function. Clear box designs are expressed in design languages for which target language code generators exist, or in restricted subsets of implementation languages, thereby eliminating opportunities for new errors in translation.

In verification reviews, trace tables are employed where appropriate for analysis of correctness, and written proofs are recorded for critical functions, particularly in life-, mission-, and enterprise-critical systems. Application-, language-, and environment-specific proof rules and standards provide a more complete framework for team verification. Locally-defined standards have been shown to be more effective than generic standards in producing consistent practitioner judgment about software quality.[30] In an advanced implementation, the documented process includes environment-specific protocols for specification, design, and verification based on team experience.

In an advanced approach to statistical testing, Markov- or grammar-based automated tools can be used to improve efficiency and effectiveness. For example, the IBM Cleanroom Certification Assistant permits generation of any required number of unique, self-checking test cases. In addition, the rich body of theory, analytical results, and computational algorithms associated with Markov processes have important applications in software development.[31] Both formal grammar and Markov usage models can reveal errors, inconsistencies, ambiguities, and data dependencies in specifications early in development, and serve as test case generators for statistical testing. Initial versions of systems can be instrumented to record their own usage on command, as a baseline for analysis and calibration of usage distributions in certification of subsequent system versions.

An advanced implementation can benefit from a locally-validated reliability model for software certification. Just as locally-validated standards enable more consistent practitioner judgment about software quality, a locally-validated reliability model will enable more accurate prediction of operational reliability from testing results.

In an advanced implementation, the full rigor of statistical process control can be applied to process improvement. Team accomplishments can be compared to locally-defined process control standards for performance. Errors can be categorized according to an error classification scheme to target specific Cleanroom practices for improvement.

Choosing an implementation approach

Cleanroom software engineering represents a shift from a paradigm of traditional, craft-based practices to rigorous, engineering-based practices, specifically as follow.

From:		To:
Individual operations	→	Team operations
Waterfall development	→	Incremental development
Informal specification	→	Black box specification
Informal design	→	Box structure refinement
Defect correction	→	Defect prevention
Individual unit testing	→	Team correctness verification
Path-based inspection	→	Function-based verification
Coverage testing	→	Statistical usage testing
Indeterminate reliability	→	Certified reliability

A phased approach to Cleanroom implementation enables an organization to build confidence and capability through gradual introduction of new practices with corresponding growth in process control. If organizational support and capability is sufficient for full implementation, the highest software quality and reliability afforded by Cleanroom practices can be achieved. Otherwise, a phased implementation is recommended. In general, a software organization that employs informal methods of specification and design, relies on coverage testing and defect correction to achieve quality, and has little experience with team-based operations, can gain the most benefit through an introductory implementation. This first phase introduces a comprehensive set of practices spanning project management, development, and testing, but without the full formality of Cleanroom technology. Once an organization successfully completes a project using the introductory practices, it has prepared itself for a full implementation. Likewise, maturation from full to advanced implementation can occur when the practices of the second stage have been successfully demonstrated.

Note that very few teams in reality will implement the precise set of practices defined within each implementation. Each team embodies unique skills, processes, and experiences that must be assessed when choosing an appropriate implementation. It is often the case that a team can best utilize practices from more than one implementation level. For example, a team using an introductory implementation may have had prior experience with inspections and code reviews. Consequently, it may shift to a full or advanced implementation of the system design and verification practices. Perhaps another mature Cleanroom team, using primarily advanced practices, will find the rigor of the second phase of system specification to be sufficient.

The well-known Software Engineering Institute Capability Maturity Model provides a useful assessment technique to help define the best Cleanroom approach.[32,33] In general, higher assessment levels indicate that an organization can successfully adopt a more complete Cleanroom implementation. Organizations assessed at levels 1 and 2 will likely benefit from an introductory implementation, at levels 2 and 3, a full implementation, and at levels 4 and 5, an advanced implementation.

Phased implementation on the AOEXPERT/MVS project

AOEXPERT/MVS is the largest completed Cleanroom project in IBM, both in terms of lines of code and project staffing. The project adopted an introductory implementation of the Cleanroom process for development, and realized a defect rate of 2.6 errors/KLOC, measured from the first execution of the code. This represents all errors ever found in testing and installation at three field test sites. Development productivity averaged 486 lines of code per person-month, including all development labor expended in specification, design, and testing. In short, the AOEXPERT/MVS team produced a complex systems software product with an extraordinarily low error rate, while maintaining high productivity. The following summary of the project is elaborated in Reference 15.

The AOEXPERT/MVS product. AOEXPERT/MVS is a decision-support facility that uses artificial intelligence (AI) for predicting and preventing complex operating problems in an MVS environment. Primarily a host-based product, it runs in a NetView* environment on MVS with interfaces to several other IBM program products. A workstation component running under Operating System/2* (OS/2*) in the Personal System/2* (PS/2*) environment provides the user interface for the definition and management of the business policies for system operation to be applied by AOEXPERT/MVS to avoid and correct system problems.

Few teams will implement the precise set of practices defined within each implementation.

The complex development environment required expertise in MVS and its subsystems, expert systems technology, real-time tasking, message passing, and windows-based programming for the workstation component. The product was implemented using PL/I, TIRS* (an AI shell), PL/X (an internal IBM system language), assembler, JCL, and REXX for host software, and C and Presentation Manager* for workstation software. The environment was further complicated by two major dependencies on IBM system management products that were developed by other IBM laboratories.

The project began in July 1989, with the first eighteen months spent in the requirements phase. Development team staffing took place during this initial stage. Four departments were ultimately established: one for requirements, two for development, and one for testing. Various support organizations provided market development, quality assurance, information development, usability analysis, and business and legal services.

Table 3 The AOEXPERT/MVS implementation of the Cleanroom process

Cleanroom Practice	Introductory	Full	Advanced
Team operations	X		
Customer interaction	X		
Incremental development		X	
System specification	X		
System design	X		
Correctness verification		X	
Statistical usage testing	X		
Reliability certification	X		
Process improvement	X		

The project team was newly formed, with members ranging from programmer retrainees to senior programmers with 25 years of development experience. The project team averaged 50 people throughout development. Experience in the product domain was mixed, with considerable experience in application development and AI, but very little in MVS and system programming. As it turned out, AI skills were utilized about 10 percent of the time during development, while MVS and system programming skills were needed 90 percent of the time.

This was the first Cleanroom development experience for all participants, with the exception of one development manager and two developers. Consequently, extensive education and training were required to implement Cleanroom practices. The overall project schedule had been established in late 1989, prior to the decision to use the Cleanroom process. Given the schedule and mix of skills and experience levels, the Cleanroom process was first met with healthy skepticism. The team had to grapple with three important factors at once: a new team, little experience in the subject domain, and the new Cleanroom development process.

Defining an introductory implementation. The decision to use the Cleanroom process was made in the second quarter of 1990, a year after the project started and six months prior to the beginning of development. Due to the aggressive project schedule, the large size of the organization, the lack of prior Cleanroom experience, and the limited amount of training time available, the management and technical team decided on a phased implementation of the Cleanroom approach. As summarized in Table 3, the team defined an introductory approach that included team-based operations, external specification of behavior using intended functions, design expressed in a Process Design Language (PDL) with automatic target translation (for PL/I), and staged delivery of each increment to independent testers for first execution. In addition to the introductory practices, two full practices were used: incremental development and team-based correctness verification of every line of code. While it was agreed that statistical testing would be very effective, the test team did not believe it could learn and apply the methodology in time for the first increment. The greatest concern was the late start on defining a usage probability distribution, a task normally initiated as soon as the functional specification is available. The test team initially followed the spirit if not the form of usage testing, with a testing approach based on expected customer usage. Later, statistical usage testing was employed for a significant subset of the product, the workstation component, which accounted for approximately 40 percent of total product code.

Getting started. Cleanroom education was provided to the entire project, with mandatory management participation. To further define the use of Cleanroom process in the project environment, a process working group was formed to document the AOEXPERT/MVS Cleanroom development process, to establish and maintain project procedures, standards, and conventions, to establish and maintain a measurement and improvement subprocess, and to provide a formal mechanism to resolve process issues and make improvements. Each major project functional area, including architecture, host development, workstation development, test, configuration management, and quality assurance, was represented on the process working group. The group documented a comprehensive set of procedures and standards for an integrated, Cleanroom-based software development process. This document and its subsequent use by the team was critical in achieving acceptance and ownership of the process by the team. Changes to the process required approval by the process working group and management. During the development of AOEXPERT/MVS, a number of useful process revisions resulted from suggestions by team members in periodic meetings held to improve the development process.

Applying the introductory implementation. The decision to use the Cleanroom process was made rather late in the project after the product functional specification (PFS) document was almost completed. The PFS is required for IBM program product development, but it is not an adequate replacement for a Cleanroom specification, as it contains only a subset of the information required. The AOEXPERT/MVS team decided to complete the PFS, and then produce a more formal black box, incremental specification. The formal specification used precise English descriptions in conjunction with intended functions to specify the external behavior of the increments.

Following specification, project technical leaders created an incremental development plan that defined the functional content, development schedule, and resource requirements for three software increments. Although the project completion date had been established earlier, substantial flexibility remained for scheduling increment development and testing within the overall schedule of 12 months. Historical productivity and defect rates from comparable traditionally-developed applications were reviewed and the schedules were adjusted based on historical Cleanroom data, personal experience, and confidence. The first increment was planned to contain the least function of the three, in order to quickly familiarize the project team with the new Cleanroom process and development environment. Development of the first increment required two and one-half months, with the second and third increments requiring three and one-half months each.

Eight principal functional components were defined for AOEXPERT/MVS and organized into functional content comprising the three increments. Each component was assigned to a team composed of from one to five developers, with each team augmented by an architect and a tester. Team membership remained stable throughout development of all three increments, helping to ensure continuity and growth of expertise and capability. A functional management approach was adopted because each team consisted of people from different departments. Since each team had a designated team leader, management ownership was assigned based on the team leader. Thus, a manager was responsible for all teams led by members of the manager's department. This process worked well, but required daily communication between managers, usually in the form of morning status meetings where schedules, plans, resources, and performance were addressed.

Following increment planning, development began for the first increment. It immediately became obvious that the developers lacked a good understanding of the entry criteria for team correctness verification reviews. Most understood how to perform verification, but underestimated the level of rigor and precision required in the design material. For example, intended functions documented in many of the early first increment designs precisely specified intended behavior for normal or steady-state operation, but failed to specify intended behavior for error conditions, exception processing, and unexpected input. As a result, the designs could not be verified for correctness.

To address this problem, project management decided that a demonstration verification review of an actual first increment design should be held as early as possible. A senior-level programmer was asked to prepare a design for the review. When the design was ready, his five-member team conducted a formal correctness verification review, with the remainder of the AOEXPERT/MVS organization, numbering about 45 people, in attendance as observers. Everyone in attendance had a copy of the material and followed along with the review team. The review lasted about three hours, with the design failing to pass the verification process. This outcome proved to be an invaluable teaching tool for the project team. Most were surprised that the design did not pass, and even more surprised at the number of changes required to make it verifiable. The demonstration clearly showed the team what was actually expected in a Cleanroom review, and definitely saved a substantial amount of time and frustration in the remainder of the project. Since the first increment was relatively small and straightforward, the team was able to learn how to correctly apply the Cleanroom approach and still make the first delivery date.

Cleanroom facilitators. The AOEXPERT/MVS project benefited from people with prior Cleanroom experience, who played dual roles as team members and Cleanroom methodology consultants. These people served as teachers and advisors, providing guidance on how to write verifiable designs and conduct effective verification reviews. Equally important was the encouragement they gave and confidence they instilled in their peers through their example and coaching. During the first increment of development, one of these ex-

Table 4 AOEXPERT/MVS error rates measured from first execution

AOEXPERT/MVS Project		Industry Expectation		AOEXPERT/MVS Project Results	
Increment	KLOC	Errors at 30/KLOC	Projected Errors	Actual Software Errors	Errors/KLOC
1	16	480	64	43	2.7
2	50	1500	200	41	0.8
3	41	1230	164	97	2.4
Subtotal	107	3210	428	181	1.7
System testing			107	93	0.9
Total	107	3210	535	274	2.6

Where

- Projected errors included increment testing projected at 4 errors/KLOC, and system testing at 1 error/KLOC
- Actual software errors were measured from the first execution
- System testing included system, performance, and field testing

perts was present at every verification review to ensure the methodology was followed, especially with respect to application of the correctness verification conditions. During development of the second and third increments, other team members, now with experience in the Cleanroom process, joined with the original experts to form a core group of five to six facilitators who served a key role in acceptance, application, and improvement of the Cleanroom process.

Team verification reviews. The Cleanroom correctness verification process was closely followed. A check was made prior to every review to ensure that the entry criteria were satisfied, and a disciplined process of correctness condition verification for every control structure was followed during the review process. A moderator was assigned, usually one of the Cleanroom facilitators, to ensure that the reviews were conducted properly, and that all issues were recorded and all changes reverified. The author of the design under verification typically led the team through the review. Also present were a key reviewer, usually the component team leader who had a broad understanding of the component function, and other reviewers, typically members of other teams whose components interfaced with the designs under review. Review materials were required to be distributed to all reviewers at least 48 hours prior to the review, and all reviewers were expected to have read the materials before attending the review.

Quality results. The AOEXPERT/MVS testing process was composed of two phases, increment testing and system testing. (In a full implementation of the Cleanroom process, all testing would be regarded as system testing.) After examining data from prior Cleanroom projects, the test team estimated expected defect rates in testing and customer use of the product. Four errors/KLOC were estimated for increment testing, an additional 1 error/KLOC for system testing, and an additional 0.5 error/KLOC for customer use after the product was shipped. These estimates were significantly lower than those customarily found for comparable products, but the team believed that such aggressive goals should be set, even for a first-time Cleanroom effort.

Table 4 summarizes error rates for the three product increments, measured from the first execution of the code. For comparison, projected errors are shown based on an average industrial rate of 30 errors/KLOC[20] for traditional development projects measured from the first execution of the code, with a total of 3210 errors expected at this rate. The test team estimate of 5 errors/KLOC (4 in increment testing plus 1 in system testing) totaled to 535 errors expected.

The AOEXPERT/MVS team produced the complex systems software product with only 274 errors found in all testing. This error rate of 2.6 errors/KLOC was over an order of magnitude better than the industry average of 30 errors/KLOC, and nearly halved the projected Cleanroom rate of 5 errors/KLOC. A number of system components completed testing with no errors found. For example, five of the eight components in the first 16 KLOC increment proved to be error-free in all testing. In addition, no operational errors whatsoever were found following product installation at three customer test sites, and no post-ship customer errors have been reported to date.

Productivity results. Productivity estimates for AOEXPERT/MVS were based on rates for comparable, traditionally-developed products, modified by expected gains from the Cleanroom process and the belief that productivity would improve with each successive increment. Productivity was estimated at 300 lines of code per person-month (LOC/PM) for the first increment, 350 for the second increment, and 400 for the third increment. Table 5 shows actual productivity rates achieved, based on total lines of code divided by the person-months accumulated for formal specification through testing of the final increment. The person-months include development staff only. The project achieved very competitive productivity rates, exceeding the projected rates by 36 percent overall. This substantial improvement in productivity was a significant factor in enabling the project to meet its schedule. The original code size estimate was 72 KLOC, but the actual code size was significantly larger (107 KLOC) due primarily to unexpected growth in the workstation software (from 10 to 42 KLOC). The growth resulted from the lack of familiarity with OS/2 Presentation Manager and unanticipated requirements. Thus, while actual productivity was a 36 percent improvement over the projected rate, actual code size was 49 percent larger than planned. The increased productivity enabled the team to stay on schedule during the development.

Observations. From the beginning of the project through delivery and testing of the first increment, many developers and testers were somewhat skeptical about the Cleanroom approach. The real turnaround in acceptance occurred after the first increment was delivered and tested and so few errors were found. In fact, several testers were upset and worried when they failed to find any errors; ironically, so were the developers. But this soon changed for everyone—defects quickly became the exception, not the rule, and a "right the first time" psychology took hold.

The challenges facing a new team in an unfamiliar environment were great, and schedules and resources were extremely tight. Nevertheless, a new methodology was introduced, taught, and implemented with substantial success. The primary success factors in this implementation of Cleanroom process were the use of an introductory implementation, early and ongoing management commitment, incremental development of system function, demonstration reviews for team education, team-based peer review of all work products, full application of correctness verification, adherence to defect prevention practices, and the use of Cleanroom consultants and facilitators.

Table 5 AOEXPERT/MVS productivity rates

Increment	KLOC	Projected Productivity LOC/PM	Actual Productivity LOC/PM	% Actual Exceeds Projected
1	16	300	400	+33
2	50	350	500	+43
3	41	400	513	+28
Average		358	486	+36

Where the actual productivity was the LOC/PM measured from formal specification through testing

The AOEXPERT/MVS experience is representative of the new level of quality that is possible in software development today. Cleanroom is a practical and proven alternative to the high cost and poor quality frequently seen in traditional development processes. As evidence of its effectiveness continues to accumulate, the Cleanroom process will be increasingly adopted by organizations seeking competitive business advantage.

Acknowledgments

The growing worldwide community of Cleanroom practitioners and researchers, as well as Cleanroom-developed software owners and users, owes an enormous debt of gratitude to Harlan Mills, a retired IBM Fellow, for his groundbreaking development of theoretical foundations for the Cleanroom process. Mills's insight that programs are rules for mathematical functions became the foundation for a disciplined engineering process of program development that replaced a sea of ad hoc methods. His insight that program reliability can be scientifically certified in formal statistical designs brought the power of statistical quality control to software engineering and management.

Those of us who worked with Mills during the formative years of the Cleanroom process were indeed fortunate to have shared in the exciting intellectual climate he created, and to have contributed to the development of the Cleanroom process.

Acknowledgment is also due to members of the Cleanroom teams whose accomplishments are reported in this paper. These practitioners are setting whole new standards of professional excellence and achievement in software engineering.

The authors would like to thank Mark Pleszkoch of the IBM Cleanroom Software Technology Center for his suggestions.

*Trademark or registered trademark of International Business Machines Corporation.

Cited references

1. H. D. Mills, M. Dyer, and R. C. Linger, "Cleanroom Software Engineering," *IEEE Software* **4**, No. 5, 19–24 (September 1987).
2. R. C. Linger, "Cleanroom Software Engineering for Zero-Defect Software," *Proceedings of 15th International Conference on Software Engineering*, IEEE Computer Society Press, Los Alamitos, CA (1993), pp. 2–13.
3. P. A. Curritt, M. Dyer, and H. D. Mills, "Certifying the Reliability of Software," *IEEE Transactions on Software Engineering* **SE-12**, No. 1, 3–11 (January 1986).
4. R. C. Linger, H. D. Mills, and B. J. Witt, "Structured Programming: Theory and Practice," Addison-Wesley Publishing Co., Reading, MA (1979).
5. H. D. Mills, R. C. Linger, and A. R. Hevner, "Principles of Information Systems Analysis and Design," Academic Press, Inc., New York (1986).
6. H. D. Mills, "Certifying the Correctness of Software," *Proceedings of 25th Hawaii International Conference on System Sciences*, IEEE Computer Society Press, Los Alamitos, CA (January 1992), pp. 373–381.
7. M. D. Deck and P. A. Hausler, "Cleanroom Software Engineering: Theory and Practice," *Proceedings of Software Engineering and Knowledge Engineering*: Second International Conference, Skokie, IL, June 21–23, 1990. Knowledge Systems Institute, 3420 Main St., Skokie, IL 60076.
8. E. N. Adams, "Optimizing Preventive Service of Software Products," *IBM Journal of Research and Development* **28**, No. 1, 2–14 (1984).
9. H. D. Mills, "Mathematical Foundations for Structured Programming," *Software Productivity*, Little, Brown and Company, Boston, MA (1983), pp. 115–178.
10. P. A. Hausler, "Software Quality Through IBM's Cleanroom Software Engineering," *Creativity!* (ASD-WMA Edition), IBM, Austin, TX, March 1991.
11. H. D. Mills and J. H. Poore, "Bringing Software Under Statistical Quality Control," *Quality Progress* (November 1988), pp. 52–56.
12. R. C. Linger and R. A. Spangler, "The IBM Cleanroom Software Engineering Technology Transfer Program," *Proceedings of SEI Software Engineering Education Conference*, C. Sledge, Editor, Springer-Verlag, Inc., New York (1992).
13. B. W. Boehm, *Software Engineering Economics*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1981).
14. S. E. Green, A. Kouchakdjian, and V. R. Basili, "Evaluation of the Cleanroom Methodology in the Software Engineering Laboratory," *Proceedings of Fourteenth Annual Software Engineering Workshop*, NASA, Goddard Space Flight Center, Greenbelt, MD 20771 (November 1989), pp. 1–22.
15. P. A. Hausler, "A Recent Cleanroom Success Story: The Redwing Project," *Proceedings of Seventeenth Annual Software Engineering Workshop*, NASA, Goddard Space Flight Center, Greenbelt, MD 20771 (December 1992), pp. 256–285.
16. R. C. Linger and H. D. Mills, "A Case Study in Cleanroom Software Engineering: The IBM COBOL Structuring Facility," *Proceedings of 12th Annual International Computer Software and Applications Conference (COMPSAC '88)*, IEEE Computer Society Press, Los Alamitos, CA (1988), pp. 10–17.
17. S. E. Green and R. Pajersky, "Cleanroom Process Evolution in the SEL," *Proceedings of 16th Annual Software Engineering Workshop*, NASA, Goddard Space Flight Center, Greenbelt, MD 20771 (December 1991), pp. 47–63.
18. L-G. Tann, "OS32 and Cleanroom," *Proceedings of 1st Annual European Industrial Symposium on Cleanroom Software Engineering* (Copenhagen, Denmark), Q-Labs AB, IDEON Research Park, S-223 70 Lund, Sweden (1993), Section 5, pp. 1–40.
19. C. J. Trammell, L. H. Binder, and C. E. Snyder, "The Automated Production Control Documentation System: A Case Study in Cleanroom Software Engineering," *ACM Transactions on Software Engineering and Methodology* **1**, No. 1, 81–84 (January 1992).
20. M. Dyer, *The Cleanroom Approach to Quality Software Development*, John Wiley & Sons, Inc., New York (1992).
21. *A Success Story at Pratt and Whitney: On Track for the Future with IBM's VS COBOL II and COBOL Structuring Facility*, GK20-2326, IBM Corporation (1989); no longer available though IBM branch offices.
22. R. C. Linger and A. R. Hevner, "The Incremental Development Process in Cleanroom Software Engineering," *Proceedings of Workshop on Information Technologies and Systems (WITS-93)*, A. R. Hevner, Editor, College of Business and Management, University of Maryland, College Park, MD (December 4–5, 1993), pp. 162–171.
23. H. D. Mills, R. C. Linger, and A. R. Hevner, "Box Structured Information Systems," *IBM Systems Journal* **26**, No. 4, 395–413 (1987).
24. A. R. Hevner and H. D. Mills, "Box-Structured Methods for Systems Development with Objects," *IBM Systems Journal* **32**, No. 2, 232–251 (1993).
25. H. D. Mills, "Stepwise Refinement and Verification in Box Structured Systems," *IEEE Computer* **21**, No. 6, 23–35 (June 1988).
26. R. H. Cobb and H. D. Mills, "Engineering Software Under Statistical Quality Control," *IEEE Software* **7**, No. 6, 44–54 (November 1990).
27. R. C. Linger, "An Overview of Cleanroom Software Engineering," *Proceedings of 1st Annual European Industrial Symposium on Cleanroom Software Engineering* (Copenhagen, Denmark), Q-Labs AB, IDEON Research Park, S-223 70 Lund, Sweden (1993), Section 7, pp. 1–19.
28. J. H. Poore, H. D. Mills, and D. Mutchler, "Planning and Certifying Software System Reliability," *IEEE Software* **10**, No. 1, 88–99 (January 1993).
29. J. H. Poore, H. D. Mills, S. L. Hopkins, and J. A. Whittaker, *Cleanroom Reliability Manager: A Case Study Using Cleanroom with Box Structures ADL*, Software Engineering Technology Report, IBM STARS CDRL 1940 (May 1990). STARS Asset Reuse Repository, 2611 Cranberry Square, Morgantown, West Virginia 26505.
30. C. J. Trammell and J. H. Poore, "A Group Process for Defining Local Software Quality: Field Applications and Validation Experiments," *Software Practice and Experience* **22**, No. 8, 603–636 (August 1992).
31. J. A. Whittaker and J. H. Poore, "Markov Analysis of Software Specifications," *ACM Transactions on Software Engineering and Methodology* **2**, No. 1, 93–106 (January 1993).

32. M. C. Paulk, W. Curtis, M. B. Chrissis, C. V. Weber, *Capability Maturity Model for Software, Version 1.1*, CMU/SEI-93-TR-24, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA 15213 (February 1993).
33. M. C. Paulk, C. V. Weber, S. M. Garcia, and M. B. Chrissis, *Key Practices of the Capability Maturity Model, Version 1.1*, CMU/SEI-93-TR-25, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA 15213 (February 1993).

Accepted for publication October 6, 1993.

Note: At the time of publication, Federal Systems Company, now a unit of Loral Corporation, was an IBM-owned company. Addresses for authors may still be considered valid.

Philip A. Hausler *IBM Corporation, 6710 Rockledge Drive, Bethesda, Maryland 20817 (electronic mail: hausler@vnet.ibm.com).* Mr. Hausler is a senior programmer manager in the Cleanroom Software Technology Center of IBM. His department provides education and consultation for technology transfer of the Cleanroom process. He was a principal developer of IBM's first Cleanroom product, the COBOL Structuring Facility, and has held various development and management positions in IBM. Since 1985, Mr. Hausler has served on the faculty of the Computer Science department at the University of Maryland, Baltimore County, teaching software engineering, programming languages, and compiler theory courses. He received a B.S., *summa cum laude*, in computer science in 1983 from the University of Maryland, Baltimore County, and an M.S. in computer science in 1985 from the University of Maryland, College Park. Mr. Hausler has authored or coauthored numerous refereed papers in technical journals. He is a member of the IEEE.

Richard C. Linger *IBM Corporation, 6710 Rockledge Drive, Bethesda, Maryland 20817 (electronic mail: lingerr@betasvm2.vnet.ibm.com).* Mr. Linger is a member of the Senior Technical Staff of IBM. He is the founder and manager of the IBM Cleanroom Software Technology Center, which is chartered to provide Cleanroom technology transfer services to IBM product laboratories and customers. He worked with Harlan D. Mills in developing the Cleanroom software engineering process, and managed development of the COBOL Structuring Facility product, the first commercial Cleanroom project in IBM. He has written or coauthored two textbooks used in Cleanroom education and over 50 refereed papers on the Cleanroom process, software re-engineering and reverse engineering, and other software engineering topics. Mr. Linger is a member of the ACM and the IEEE.

Carmen J. Trammell *Department of Computer Science, 107 Ayres Hall, University of Tennessee, Knoxville, Tennessee 37996 (electronic mail: trammell@cs.utk.edu).* Dr. Trammell is a research assistant professor and manager of the Software Quality Research Laboratory in the Department of Computer Science at the University of Tennessee. She has held software engineering and management positions in military and commercial product development at Oak Ridge National Laboratory, Martin Marietta Energy Systems, and Software Engineering Technology, Inc. She holds a Ph.D. in psychology and an M.S. in computer science from the University of Tennessee. Dr. Trammell is a member of the ACM and the IEEE.

Reprint Order No. G321-5534.