Case Study

As with any other methodology, a certain degree of tailoring is required for a Cleanroom approach to be most effective in software engineering. We have done so based on project needs, cost and schedule constraints, and comfort level. The result is an effective model for developing quality software, especially in the areas of requirements analysis and specifications development.

Tailoring Cleanroom for Industrial Use

Robert S. Oshana, Raytheon Systems Company

Raytheon Systems Company, a defense electronics firm, typifies many US Defense Department environments in its strong, process-oriented development strategy. The company has been actively engaged in software process improvement at the organizational and project levels for over a decade. Raytheon bases its efforts on the Software Engineering Institute's Capability Maturity Model, and our segment of the company has been formally assessed at level 3 (defined).

The project I'm involved with is a real-time embedded software application using a heterogeneous computer architecture: a single-board PowerPC using Ada for embedded command and control, and digital signal processors running applications written in C. There are significant real-time constraints within the signal processing application.

To meet competitive pressures, the project's management team sought strategies to improve productivity and increase the quality of the software we deliver to our system integration and test lab. The most important consideration was the need for high-quality, low-defect software that would require little or no rework.

Raytheon chose our project to pilot a CMM level 4 (managing) and 5 (optimizing) environment. We are now implementing the respective key process areas—including components of continuous process improvement, process optimization, and process measurement and analysis. The project follows a tailored version of DoD STD-2167A and MIL-STD-498 documentation standards, and is built upon an Integrated Product Team structure. IPTs are management processes that integrate

0740-7459/98/$10.00 © 1998

For me, the term "Cleanroom" always conjures up images of men in white suits working on some elaborate equipment, destined to be launched into outer space. Most things are not built in cleanrooms and most software developers cannot afford cleanroom software engineering. But there are useful lessons to be learned. If, for example, you want to build a house for $300,000 it does not hurt to look at a few houses in the $1 million range. You can glean some neat features and make a list of ideas for your own dream house. Then you start descoping these features until the results fit your budget. You won't get a perfect house, but certainly a better one than if you used only $100,000 houses for inspiration.

I suggest you read Oshana's article with this analogy in mind. Not many companies have reached CMM level 3 or higher. In most cases, particular business imperatives don't allow for all the techniques described here. But there are many practices that can serve as the "million dollar house." After all, the results obtained in this project speak for themselves. I know many companies that could dramatically improve the quality, productivity, and predictability of their projects if they only used one or two of the practices described.

—Wolfgang Strigel, From the Trenches editor

all activities from product concept through production using multifunctional teams to meet cost and performance goals. Further, we benefit from a fully defined software development process as well as system engineering and hardware development processes. The software effort for the project is divided into several *computer software configuration items*, with a software team of two to 16 software engineers per CSCI. A CSCI defines a complete set, or any of the individual items of the set, of computer programs, procedures, and associated documentation and data designated for delivery to a customer or end user.

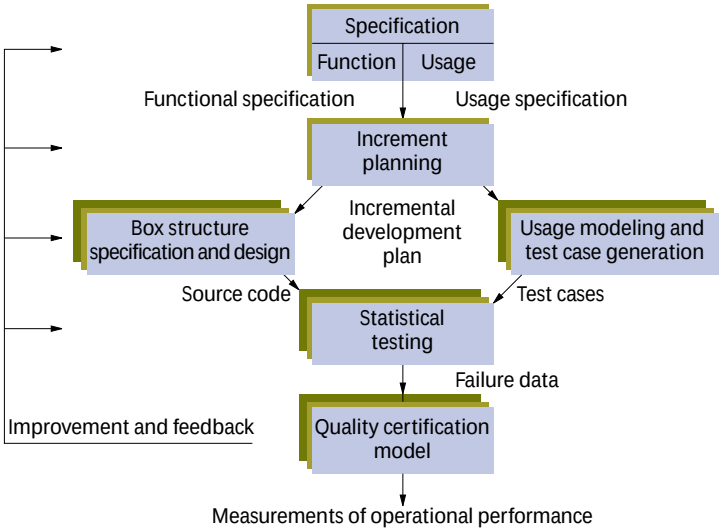


Figure 1. The Cleanroom software engineering process.

Despite this strong, process-oriented structure, management felt that, given the proper tools, our productivity and software's quality could be further enhanced. Research indicated that several strategies exist for process improvement, but that Cleanroom improves the software quality on new projects with a formal defined process, as well as on projects with a relatively immature process.[1-3] The literature also demonstrated how Cleanroom can be inserted into a process in phases.[4] We decided to implement this methodology, as shown in Figure 1. Since then, our project has successfully inserted Cleanroom technology into our mature CMM-based framework.[5]

Cleanroom Software Engineering

Cleanroom consists of a body of theoretically sound yet practical engineering principles applied to the activity of software engineering.[6-8] In a Cleanroom approach, a rigorous specification phase leads to a complete, precise description of the software system. Following this phase a stepwise refinement procedure produces verifiably correct software parts. Software development proceeds in parallel with a usage specification of the software. This usage profile becomes the basis for a statistical test of the software, resulting in a scientific certification of the quality of the software system. Defect prevention rather than removal is the focus.

The Cleanroom process spans the software life cycle. The technology provides engineering methods with which software teams can plan, measure, specify, design, verify, code, test, and certify software. Cleanroom combines formal methods of box structure specification and design-function-theoretic verification of correctness with statistical usage

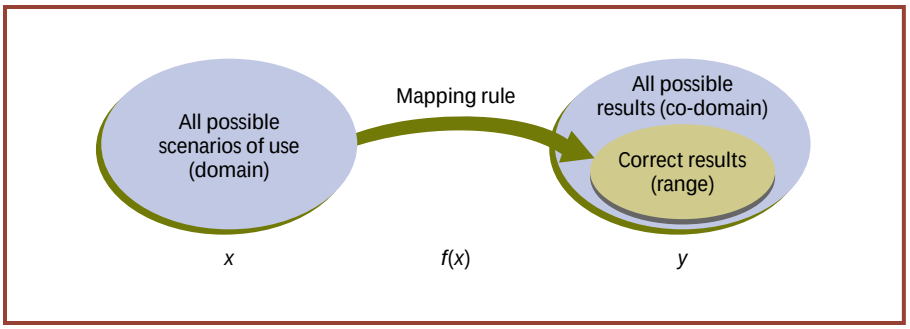


Figure 2. A software program is a rule for a function, ***y = f(x)***.

testing for reliability certification to produce software that contains very few defects. Cleanroom process management is based on an incremental life cycle in which development and certification are conducted in a pipeline of user-function increments.

Cleanroom has three main components:

1. *Incremental development*. This approach provides a basis for statistical quality control of the development process. Each increment is a complete iteration of the software development process; each is measured and compared with pre-established quality standards. If the process adheres to those standards, work on the next increment continues, using the same process. If quality standards are not met, development stops and the process is fixed. Results from each increment are used for management and process improvement. In lessons-learned and causal-analysis meetings held at the completion of each increment, the team examines all results, identifies problem areas, and replans if necessary.

2. *Software development based on mathematical principles*. As shown in Figure 2, the main concept in Cleanroom software engineering is to treat a computer program as a rule for a function. The function's domain is the set of all possible input histories, its co-domain is the set of all possible outputs, and its range is the set of all correct outputs. The program specification maps input histories to their correct outputs. The Box Structure Method, used for specification and design, defines three levels of abstraction for describing programs as rules for functions: a behavioral view, a finite state machine view, and a procedural view. Functional verification is used to confirm that the design is a correct implementation of the specification. Program correctness is verified through team reviews using formal and informal correctness questions.

3. *Software testing based on statistical principles*. The statistical testing approach to software treats software as a statistical experiment.[9,10] First we generate a statistical subset of all possible software uses. We use performance on this subset to form conclusions about operational performance based on the usage model developed. Then we represent the expected operational use in a usage model of the software. We randomly generate test cases from the usage model and then execute them in an operational environment. Failures are interpreted according to mathematical and statistical models.

Cleanroom and the Capability Maturity Model

The Capability Maturity Model for software describes an evolutionary improvement path from an ad hoc, immature software development process to a mature, disciplined one. The CMM covers practices for planning, engineering, and managing software development and maintenance. When followed, these key practices improve the ability of organizations to meet goals for cost, schedule, functionality, and product quality.

The CMM for software and the Cleanroom engineering process share a common concern with aspects of software quality and software development. They are compatible and complementary: the CMM focuses on process management and the principles and practices associated with software process maturity, whereas Cleanroom focuses on theory-based engineering practices for developing and certifying software under statistical quality control. Cleanroom software engineering implements most of the CMM for software development.

The SEI has developed a Cleanroom Software Engineering Reference Model[11,12] that provides a framework, in the form of a high-level template, for developing a project- or organization-level Cleanroom process. The CRM serves as a guide to Cleanroom project management and performance. It is also useful for project performance assessment and improvement, and for technology transfer. Therefore, the Cleanroom processes and work products should be tailored for use by the specific project and organization. The actual implementation details and procedures are left to the individual project, and should be defined in the Cleanroom Engineering Guide or other equivalent document. The SEI has also produced a Cleanroom implementation document that provides a more detailed mapping of the Cleanroom processes to the CMM key process areas.

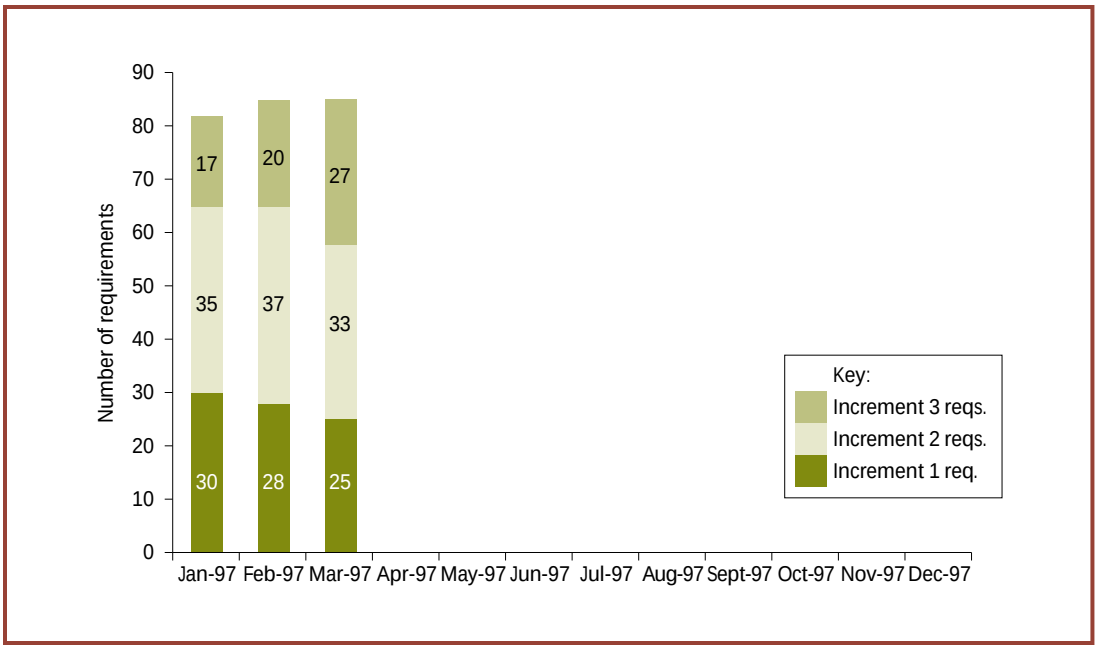


Figure 3. Increment metric used for tracking requirements for each CSCI.

Tailoring Cleanroom for Industry Use

In the spectrum of formal methods, Cleanroom is more formal than a formal inspection-based development effort, but less formal than those approaches that use fully formal specification languages with rigorous semantics such as VDM or Z.[13] Indeed, although we describe our Cleanroom specification as formal, we created it using an in-house tool written with Excel plug-ins and extensions. Commercial Cleanroom specification tools are now available (for instance, see http://www.toolset.com).

Like other formal methods, Cleanroom is well suited for complex problems. The Cleanroom techniques of sequence-based specification are well suited to problems with a high degree of logical interactions (many modes and/or states determined by Boolean conditions). We found that sequence-based specifications are less suited for the higher-level black boxes used in numerical or highly computational applications. Specification of a numerical algorithm using Cleanroom methods requires specification of the domain and the mapping rule for the mathematical functions to be implemented by the algorithm. Cleanroom methods such as incremental development, correctness verification, and statistical testing are very effective for all types of software. Cleanroom appears to offer significant benefit to applications of moderate to large size that are decomposable into subsystems. Our application was decomposable in such a way that we could implement Cleanroom at the CSCI level instead of starting with a top-level system black box. Each CSCI is effectively a "mini" project implementing each of the Cleanroom technology components. The Cleanroom process is defined formally in our *Software Standards and Procedures Manual* as well as in our Software Development Plan. The software leads meet periodically to ensure that each of our teams is using the Cleanroom process consistently and correctly. This assures adherence to the defined process as well as consistency of artifacts.

Project management planned increments at the CSCI level. Each CSCI has one or more software increments that are tracked monthly using an increment metric. All CSCI-specific software requirements are allocated to one of the defined increments based on stability of the requirement, reusability potential, technical risk, functionality, and other factors. This allows management and the customer to control the increment content over the life of the program and foresee any problems with incremental development early. The main concern is the postponement of requirements from increment to increment, resulting in a final increment containing numerous requirements passed along from earlier increments. Figure 3 shows an example increment metric for a three-increment software program.

The requirements analysis phase involves Cleanroom specification techniques at the first level of decomposition, as shown in Figure 4. Requirements are passed down from the system level (A-spec) to the "box" level (B-spec), then allocated to hardware and software. These English language software requirements are then translated into more formal box structures based on the function theory approach of sequence and state enumeration. This effort covers a

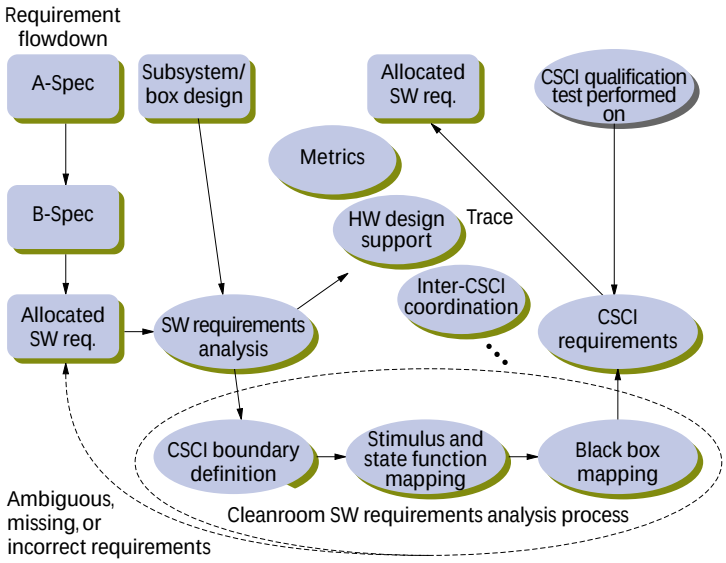


Figure 4. Decomposing requirements during the requirements analysis phase.

significant portion of the functional requirements for the CSCI. Requirements that describe real-time and accuracy constraints, as well as a significant portion of the algorithmic requirements, did not go through the Cleanroom specification process for this first level of decomposition; the particular techniques we were implementing were not useful for these types of requirements at this level.

Cleanroom does not inherently address real-time issues. The strength of the methodology lies in its functional stepwise decomposition of requirements

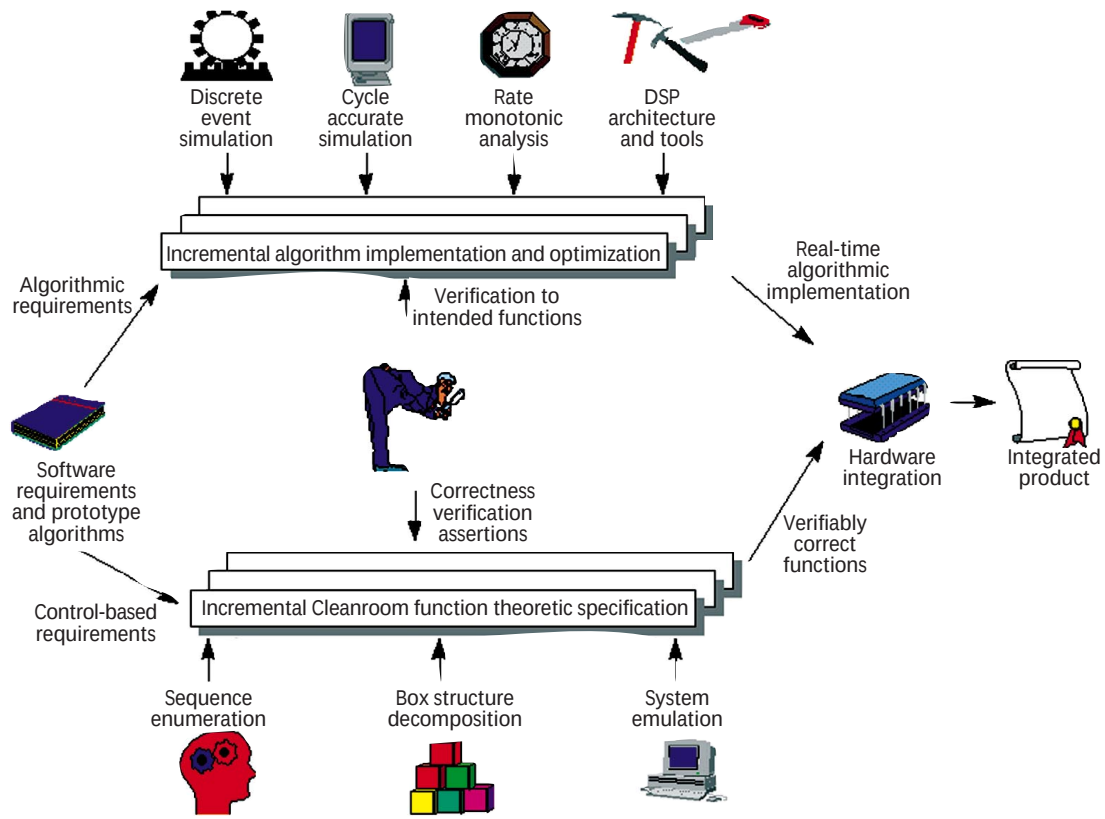


Figure 5. Real-time issues are addressed in parallel with function verification.

and its certification and reliability estimation. As Figure 5 shows, our process consisted of using Cleanroom to verify a functionally accurate software system, supplemented with a parallel process of temporal verification. Because we designed the project as a hardware–software codevelopment effort, hardware was not available until late in the development cycle. Thus we estimated real-time performance using discrete-event simulation, cycle-accurate processor simulation, and rate-monotonic analysis (mathematical conditions for assessing whether a set of tasks can be scheduled) to verify the software tasking architecture, and other real-time performance measures of the system under worst-case scenarios. We optimized "hot spots" in the algorithmic portions of the software using more efficient language implementation, algorithmic efficiencies, and assembly language as a last resort. Once the algorithms were shown to be functionally correct, the optimization process had truth data to verify against.

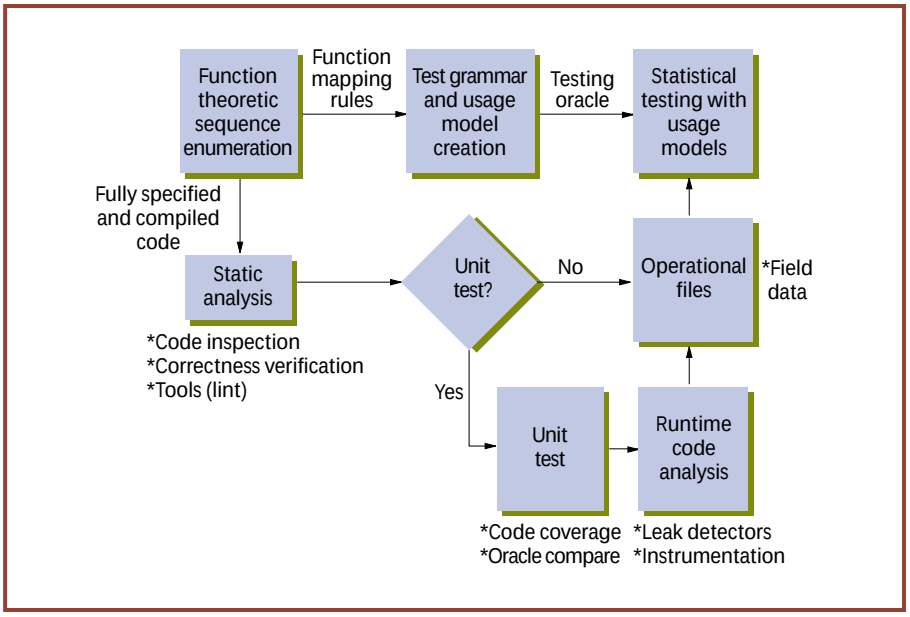


Figure 6. A certification model for algorithmically intensive software.

We tested the software at the CSCI level using various levels of statistical testing, supplemented with unit and functional testing where required as well as with operational-profile testing techniques.[14] We generally did unit and function testing for the algorithmic portions of the software, which involved the more mathematically based functions with strict real-time constraints. We used various implementation and optimization strategies to effectively map the algorithms to the processor for optimal real-time performance. Because of the inherent lack of user stimuli and state data in these algorithms, sequence and state enumeration were not as effective and were performed only when necessary. We developed an effective model for certifying algorithmically intensive software that included

- a formal code inspection and correctness verification phase;
- an optional level of unit and function testing (based on the nature of the algorithms), including both static and runtime analysis;
- an operational profile phase using real data collected from various user environments; and
- a final statistical testing phase using usage models developed for different user and usage stratifications.

Figure 6 shows this certification model.

Figure 7 shows the automated testing environment we developed to execute the statistically generated test scripts during the certification phase. Usage models were created directly from the box structures developed during the specification phase. The usage models were then translated into the appropriate test grammars and executed in a special test equipment environment. As a contingency for the customer, we agreed to craft specific test cases for those CSCI-level requirements that could not be verified using the statistical approach. To date, several of the algorithmic requirements were verified using one or more crafted test cases. But essentially all of the control-based requirements have been verified using statistically generated test scripts based on a set of usage models representing different stratifications of the system (normal use, adverse use, and so on). A minimal covering set of test scripts from the usage model is achieving 80 to 90 percent statement coverage in most CSCIs (not all increments have been completed).

Feedback from practitioners

Tailoring Cleanroom technology provided us with process improvements. Although there were some anti-Cleanroom advocates, in general, most

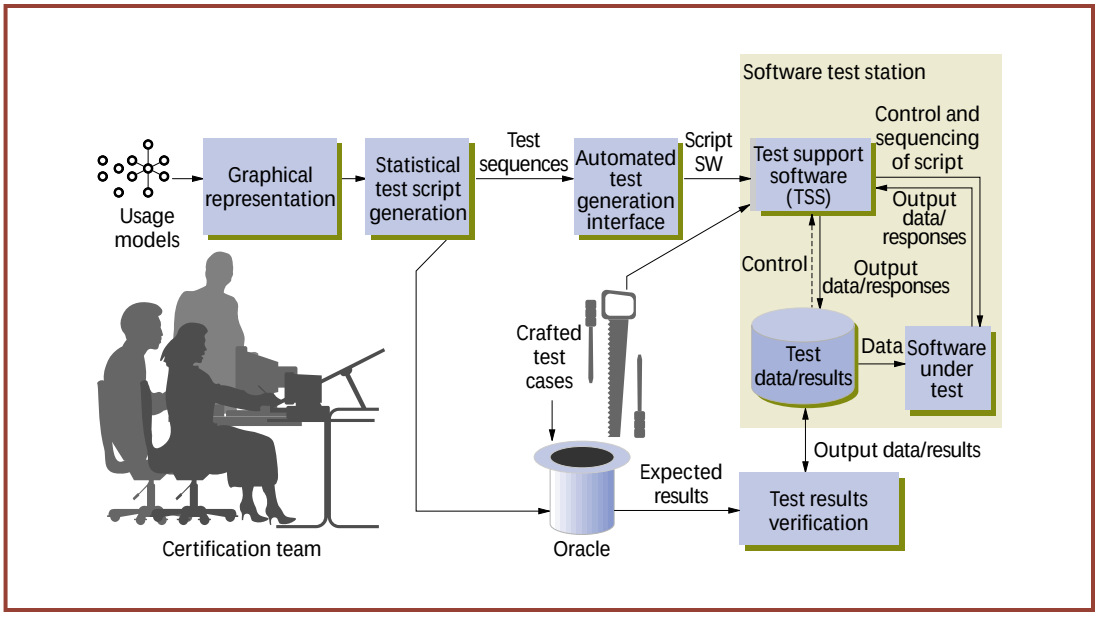


Figure 7. Testing environment automating the execution of statistically generated test scripts.

of the staff saw benefits in one or more of the Cleanroom process components. Based on feedback from the practitioners across several CSCIs, the main advantages of the technology are

- the systematic approach to development,
- using sequence enumeration to force thoughts about "impossible" events,
- ability to determine exact behavior,
- forethought before coding begins,
- simplicity of the process, and
- peer reviews of every step in the process.

Practitioners also cited some concerns. We addressed some of these by tailoring the use of the technology. Practitioners using Cleanroom methodology mentioned these issues most often:

- Rework of the box structure artifacts is a time-consuming process.
- It is difficult to map usage models to the special test equipment environment.
- Statistical testing should not be the only method of testing.
- Cleanroom does not address real-time issues.
- Formal proofs of correctness are difficult for many practitioners.

Using box structures to perform stepwise decomposition of the requirements was very helpful, especially during the specification phase (software requirements analysis). Changing requirements cause a ripple effect in the box structure artifacts. Although redoing these box structures is time consuming, we see no other way of verifying the total impact of a requirements change without this re-enumeration phase.

The initial perception among many of the practitioners was that correctness verification—using formal rules of discrete and predicate logic to verify functionality—was too mathematically intense an approach to software development. Although these techniques should be used for certain applications, a much less formal approach can and should be used for other types of software. Many of today's practitioners do not possess the skills, budget, or schedule necessary to verify software using this level of formality. We discovered that various levels of Cleanroom formality can be tailored to each application, based on cost, budget, and practitioner experience and comfort level.

Barriers to ubiquitous use

An informal survey of Cleanroom practitioners asked what barriers would prevent Cleanroom from being a more ubiquitous methodology. Here are the most common responses (and some of our reactions).

- More up-front design and detail frustrate some engineers. Software practitioners still do not see the benefit of spending more time specifying systems in more detail before starting the implementation phase.
- Cleanroom does not address real-time issues effectively. It is useful for functional verification but

does not guarantee that the software is optimal for real-time purposes. The guideline "Make it work right, then make it work fast" applies.

♦ Cleanroom techniques provide less support for algorithm-intensive applications that contain few Boolean control operations. DSP algorithms, for example, may have a limited number of input stimuli, making the function-mapping approach less useful.

♦ Many developers believe that the Cleanroom approach is orthogonal to structured analysis and design, and to object orientation. Proponents must convey the message that Cleanroom is based on a set of fundamental computer science principles that are effective in any type of development approach.

♦ Cleanroom lacks automation and tools. As requirements change, reworking a detailed stimuli–response mapping without tools is difficult and time-consuming.

♦ Many assume that Cleanroom is an "all or nothing" technology. The message must be spread that developers can use Cleanroom components effectively without having to adopt the entire methodology.

♦ Statistical testing is the only form of "testing" in Cleanroom. Statistical testing is useful as a component in the overall testing strategy, but more must be done in this area before many in industry will adopt statistical testing as the exclusive testing strategy.

Our project addressed some of these concerns by tailoring its use of various components of the technology to fit within project goals and requirements. We discovered that Cleanroom is not an "all or nothing" technology—it should be used to complement, not supplant, any development methodology. Further, even though we used Cleanroom successfully in a structured environment, the technology is just as useful in object-oriented development methodologies.

In fact, the project initially experienced somewhat of a culture shock when we began to use this methodology. The customer was somewhat concerned over the risk and learning curve associated with using a new methodology, so we put a risk mitigation plan in place in case the technology did not demonstrate its intended effectiveness. Since we used Cleanroom as a complement to our structured development approach, reverting back to a pure structured analysis and design approach would have been relatively easy.

Some practitioners never totally accepted the Cleanroom approach. Some thought the methodology is too process-oriented. Others simply did not think software development is mature enough to be considered an engineering discipline and should still be based on craftsman-like approaches and innovation. Many of us are not ready to base all software testing on statistical methods, but see the statistical testing approach as an excellent supplement to other techniques. Indeed, some CSCI teams are currently doing 100 percent statistical testing on their software, and are seeing good statement coverage—over 90 percent.

Tailoring the Cleanroom process within the parameters of cost, schedule, project risk, and level of practitioner experience benefitted the project. We chose this particular level of formalism to optimize the cost and benefits to the customer as well as to the project. Costs included training every software engineer, the customer, and a selected group of other disciplines such as systems engineering, quality assurance, and management in Cleanroom methods—all of whom required approximately two weeks of training each. This was a relatively insignificant factor in the overall project budget. Other costs included the employment of one to two Cleanroom contractors for the initial development increments. These consultants provided additional training sessions and workshops to the teams, helped us develop Cleanroom artifacts (box structures, usage models, expected test results, and so on), and provided timely help in resolving the many questions and problems that arose in the early months of the project. As our confidence with the new technology grew, we were able to reduce the amount of time spent with these consultants.

We found defects during the specification and certification phases that were not found using other verification techniques.

We found defects during the specification and certification phases that were not found using other verification techniques. The specification principles were an excellent approach to defect prevention, and saw the biggest cost-to-benefit ratio during that phase. Spending more time in the specification phase to ensure a consistent, correct design has provided benefits in the later phases of devel-

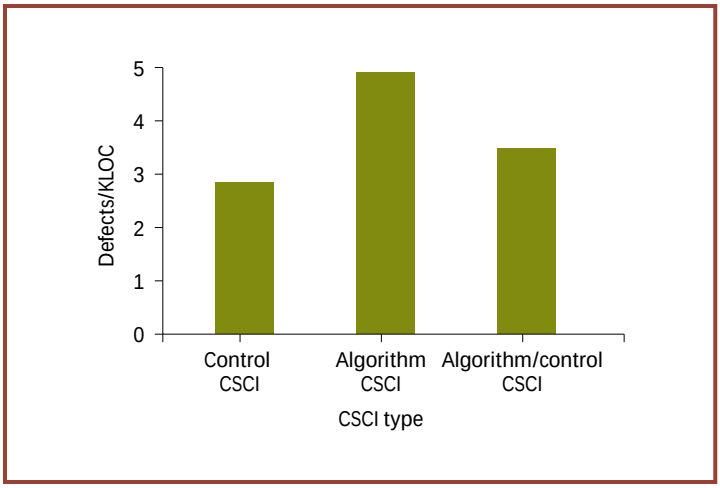


Figure 8. Estimated defects by software type for several computer software configuration items.

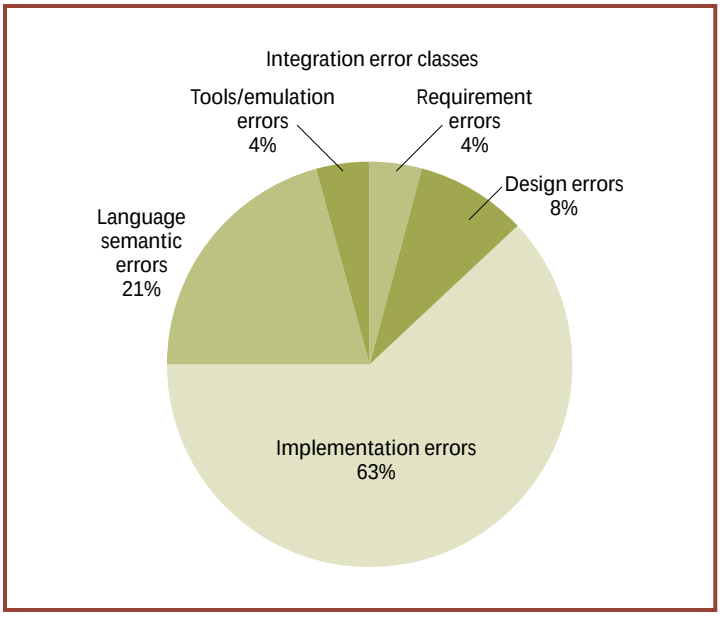


Figure 9. Certification defects by type (two out of five increments completed) for one computer software configuration item.

opment. As Figure 8 shows, very few defects appeared in the software that was fully specified using sequence-based enumeration. We have traced most of the defects to language semantics, algorithm misunderstanding, and other "silly" mistakes. This additional effort in the early specification and design phases is also consistent with Personal Software Process principles.[15] Cleanroom requires commitment and a disciplined approach, and fits well within the mature CMM framework established for the project. Incremental development, for example, provides opportunities for continuous process improvement and feedback between increments, which is a higher-level process maturity KPA in the CMM. We see the statistical approach to testing as an excellent complement to some of the existing testing techniques and are enthusiastically using these methods.

Further, the concept of separate development and certification teams is very effective. In one case, our certification team found defects immediately after the development team had handed over a product they felt was working, based on a significant amount of function testing.

Cleanroom technology, like other formal methods, does not necessarily guarantee a superior product. Erroneous specifications, flawed verification techniques, and mistakes in interpreting certification results can all result in defects in the final product. These issues are common regardless of the development approach.

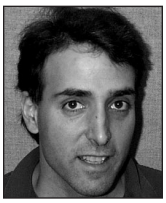
Still, the results of using Cleanroom are encouraging. We are maintaining overall cost and schedule. The customer is pleased with our product. Our statistical testing techniques are finding defects that are not found in unit and functional testing. (In all fairness, our unit and function testing is finding errors that would not be found using statistical testing. Using both approaches is very effective.) As shown in Figure 9, the defect types have mainly been implementation defects based on lack of knowledge of the programming language and the algorithmic function. Lessons learned and causal analysis performed at the completion of each software increment have led to some process improvements that we feel will have a positive impact on remaining increments. ❖

References

1. S.W. Sherer, A. Kouchakdjian, and P.G. Arnold, "Experience Using Cleanroom Software Engineering," *IEEE Software*, May 1996, pp. 69-76.
2. V. Basili et al., "SEL's Software Process Improvement Program," *IEEE Software*, Vol. 12, No. 6, Nov. 1995, pp. 83-87.
3. R.W. Selby, V.R. Basili, and F.T. Baker, "Cleanroom Software Development: An Empirical Evaluation," *IEEE Trans. Software Eng.*, Sept. 1987, pp. 1027-1037.
4. P.A. Hausler, R.C. Linger, and C.J. Trammel, "Adopting Cleanroom Software Engineering With a Phased Approach," *IBM Systems J.*, Vol. 33, No. 1, 1994.
5. D.P Kelly and R.S. Oshana, "Integrating Cleanroom Software Methods into an SEI Level 4-5 Program," *Crosstalk*, Nov.1996.
6. M. Dyer, *The Cleanroom Approach to Quality Software Development*, John Wiley & Sons, New York, 1992.
7. H.D. Mills, M. Dyer, and R.C. Linger, "Cleanroom Software Engineering," *IEEE Software*, Sept. 1987, pp. 19-25.

8. R.C. Linger, "Cleanroom Process Model," *IEEE Software*, Mar. 1994, pp. 50-58.
9. J.A. Whittaker and J.H. Poore, "Markov Analysis of Software Specifications," *ACM Trans. Software Eng. and Methodology*, Vol. 2, No. 1, Jan. 1993, pp. 93-106.
10. G.H. Walton, J.H. Poore, and C.J. Trammel, "Statistical Testing of Software Based on a Usage Model," *Software Practice and Experience*, Vol. 25, No. 1, Jan. 1993, pp. 97-108.
11. R.C. Linger and C.J. Trammell, "Cleanroom Software Engineering Reference Model, Version 1.0," Software Eng. Inst., Pittsburgh, Nov. 1996.
12. R.C. Linger and C.J. Trammell, "The SEI Cleanroom Software Engineering Reference Model and its Implementation for the CMM for Software," *Proc. 3rd Ann. Int'l Conf. Cleanroom Software Eng. Practices*, Oct. 1996, Section VI, pp. 1-15.
13. J.C. Kelly and K. Kemp, "Formal Methods Specification and Verification Guidebook for Software and Computer Systems—Planning and Technology Insertion," Release 1.0, NASA, Office of Safety and Mission Assurance, Washington, D.C., July 1995.
14. J.D. Musa, "Operational Profiles in Software Reliability Engineering," *IEEE Software*, Mar. 1993, pp. 14-32.
15. W.S. Humphrey, "Using a Defined and Measured Personal Software Process," *IEEE Software*, May 1996, pp. 77-88.

About the Author



Robert Oshana is a software team lead and a member of the group technical staff at Raytheon Systems Company. He is leading development of a real-time embedded DSP software development application using Cleanroom techniques. He is also a PhD candidate at Southern Methodist University, where he is an adjunct professor in the graduate software engineering program, teaching Cleanroom software engineering and software systems engineering.

Robert received a BSEE from Worcester Polytechnic Institute, an MSEE from the University of Texas at Arlington, an MS in computer science from Southern Methodist University, and an MBA from the University of Dallas.

Address questions about this article to Oshana at Raytheon Systems Company, PO Box 660246, Mail Station 444, Dallas, TX 75266; oshana@ti.com.