Lessons Learned from Fifteen Years of Cleanroom Testing

Michael Deck
Cleanroom Software Engineering, Inc.
7526 Spring Dr.
Boulder, CO 80303
(303) 494-3152 voice
(303) 494-3026 fax
deckm@cleansoft.com
http://www.cleansoft.com/cleansoft

James A. Whittaker
Assistant Professor of Computer Science
Florida Institute of Technology
150 West University Blvd.
Melbourne, FL 32901-6975
407-768-8000 x7638 voice
407-984-8461 fax
jw@cs.fit.edu
http://cs.fit.edu/~jw

Keywords: Cleanroom, Statistical Testing, Coverage Testing, Operational Profile

Abstract: This paper describes lessons learned through fifteen years of applying the Cleanroom software engineering approach. It focuses on the real-life practice of Cleanroom testing, rather than on a rigid ideal or orthodoxy. Although Cleanroom still recommends that development teams devote most of their testing effort to statistical testing (in concert with verification-based inspections and other development techniques), we also recognize the value of other formal testing techniques. This paper will look at practical techniques for defining the expected usage of the software under development, modeling that usage with software tools, generating test scenarios, and interpreting outcomes. Then, we will describe how statistical testing can work together with other testing strategies to cover the quality assurance spectrum.

Introduction

The Cleanroom approach to software engineering was first proposed just over fifteen years ago.[1] Like many other good ideas, it was initially greeted with skepticism by the software engineering community. However the tenacity of Dr. Harlan Mills and his team, the unique qualities of the Cleanroom approach, and above all its ability to incorporate new ideas, have made Cleanroom a recognized part of today’s software engineering curriculum.

Cleanroom has now seen two generations of users and proponents. Those who started out as junior staff or students of Cleanroom's original developers are now leading projects of their own. The authors represent this second generation. We have been both users of Cleanroom and teachers of its effective use. This paper presents ideas gained through that experience. In it, we will look at certain aspects of that experience and analyze how Cleanroom projects of the future will look. Unfortunately, there is not enough space either to describe Cleanroom thoroughly or to discuss all of the lessons we have learned. For that reason we have decided to concentrate on a few key aspects of Cleanroom testing.

This paper will describe four “lessons learned” from the authors’ personal experience using the Cleanroom practices in a variety of settings. In so doing, we will describe and motivate key Cleanroom testing practices. Although some of the projects from which we draw experience would prefer for business reasons not to be identified, additional details about Cleanroom projects on which we draw can be found in other sources [2,3,4,5]. The four lessons are:

- Testing and development should work closely together, sharing information as appropriate.
- Statistical testing is a significant engineering challenge. We propose ways to address this issue.
- Testing techniques must be appropriate to project goals. This leads us to conclude that statistical testing should be supplemented with other kinds of formal testing.
- More information, including how-to manuals, is needed by Cleanroom users.
We will expand upon each of these in turn after giving a brief summary of Cleanroom trends.

**The Cleanroom Experience**

The published record of Cleanroom project experience is uniformly good where it exists. We attribute the relatively small number of published (and non-published) Cleanroom case studies to three factors:

- Cleanroom has done a poor job of selling itself to potential users. The early papers on Cleanroom emphasized mathematics, statistics, and proof. In return they promised “software that is zero defects with high probability.” [6] Many potential users ran away as soon as they saw the word “mathematics” and in any case weren’t interested in perfection but improvement. These Cleanroom papers projected an all-or-nothing choice that dented market interest. At the other end of the market, Cleanroom proponents didn’t do enough to emphasize what was new to those who were interested in high reliability. Neither correctness verification nor statistical testing was truly new - what was new was their application in real-life situations.

- Cleanroom is independent of any particular tool suite. Although this means lower initial adoption costs, it also means there aren’t well-known vendors pushing Cleanroom in order to sell their tool.

- Cleanroom is not a magic bullet. It requires discipline, engineering capabilities, and qualified and motivated staff. Although these traits are shared by all serious approaches to process improvement, the Cleanroom proponents have been somewhat more willing than most to gloss over these exigencies. Cleanroom also has one unique feature – the strict limitations it places on developer “unit” testing – that engenders an initial culture shock in its users.

Despite these factors, Cleanroom continues to add projects to its experience base. NASA is continuing its long-term study of Cleanroom use at the Software Engineering Laboratory of Goddard Space Flight Center[7]. Cleanroom is finding its way into embedded-systems work [4,5,8] and telecommunications[9]. Perhaps most importantly, the Software Engineering Institute (SEI) has begun to show real support for Cleanroom practices and has published two important documents on Cleanroom that will greatly enhance awareness of Cleanroom and its techniques[10].

The result of the Cleanroom experience has been to demonstrate the flexibility of Cleanroom and its ability to adapt to changing circumstances. Cleanroom today is a set of tailorable guiding principles rather than as a rigid orthodoxy.

**A Summary of Practices**

Cleanroom is an overloaded term: it identifies a set of principles as well as some specific practices. Although some of the specific practices have changed over time, the fundamental principles of Cleanroom have remained very much the same [11]:

- **Design Principle.** Programming teams can and should strive to produce systems that are nearly error-free upon entry to testing.

- **Testing Principle.** The primary purpose of testing is to measure the reliability of the developed software product, and not to “test quality in.”

These fundamental principles lead to a number of specific practices, but project-specific tailoring within the bounds of the above principles is the rule, rather than the exception. The following are typical practices.

- **Incremental development.** An increment is a functional subset of the system that is executable in a user-like environment. This “end-to-end executability” requirement ensures that no effort is wasted in building artificial scaffolding and drivers and that failure-rate results can be assessed accurately.

- **Team organization.** Cleanroom projects have ranged in size from a few people to dozens. Team structures are usually organized along architectural lines. Development work products are reviewed by teams before they go into production.

- **Formal methods of specification and design.** Although the original Cleanroom practice required formal methods for specification and design, practical usage has suggested that the level of formalism can vary widely from project to project and even within a single project.
• **Intensive review.** Intensive team review has become the consensus of Cleanroom usage. The variable in practice is the level of formality, which to some extent is tied to the choice of formal methods for specification and design. A team that does not use functions to specify programs and program parts cannot effectively use functional verification to assess correctness. However, inspections can be applied in any case. The most frequent choice is to use some level of functional specification plus a “verification-based inspection” as documented by Dyer [12].

• **Testing.** Cleanroom testing emphasizes reliability measurement over bug-hunting. Testing is typically conducted from the black box user’s view. In this paper we will describe the continuing evolution of Cleanroom testing practices to include certain techniques from outside of statistical testing, including formal coverage testing and similar practices.

In sum, there is enormous variation in Cleanroom practice, within the boundaries defined by its two fundamental principles.

**Lesson 1: Integrate Development and Testing**

The first lesson that the Cleanroom experience has taught us is to reduce and, if possible, eliminate the adversarial relationship between development and testing. Too many teams evince this pathological relationship which destroys morale and works to the detriment of software quality. And yet, it’s easy to see how the traditional software-testing process encourages this kind of behavior. If the purpose of testing is to find bugs, then the better tester is the one that finds the most bugs. So if a tester dreams up a way to “break” the software that’s about to be sent into testing, what incentive is there to inform the developer of this? Even though the right time and place to fix the bug is before delivery to testing. And, as a management practice, if your testing team finds a lot of bugs are they good testers or did they get lousy software to test? There’s no way to know.

**The Cleanroom Approach**

Cleanroom looks to convert an adversarial relationship into a collegial one. All team members are focused on delivering quality software to the customer. The developers are focused on producing software that has high initial quality, testers are focused on measuring the quality of that software. Of course if the software fails during testing then there will still be bugs to fix, but the quality of the testers isn’t measured by the number of bugs they find but by the accuracy of their quality measurement. Since both parts of the team have an orthogonal (rather than opposing/policeman) contribution to quality, there is little evidence of the adversarial relationship on Cleanroom projects.

**A Little More About Cleanroom Testing**

The separation of development from testing is a defining characteristic of Cleanroom. The early Cleanroom projects (e.g., [13, 14]) implemented this separation with strict rules. No testing was done by developers, and there were even projects where programmers were denied access to the compiler.

As a result of this strict separation, the “sound bite” definition of Cleanroom became “no unit testing.” This is a memorable phrase, but it can be misinterpreted. Generally speaking, team reviews replace testing by developers for the purpose of finding defects. Although it is not as bad as ad-hoc testing and debugging, even formal unit testing by developers is considered off-limits by most Cleanroom projects.

There are three primary reasons for prohibiting developer testing.

• It forces more thorough and complete team reviews.

• Team reviews are a more cost-effective means of improving quality.

• Team reviews have numerous other benefits, including knowledge dissemination.

This is not to say there is never any execution of code by developers. There will often be executions for the purpose of understanding undocumented interfaces or learning new language features.

Switching to Cleanroom is not as simple as cutting off developer testing. Other development and testing practices must be ready to replace that testing in the Cleanroom process.
The development and testing functions do not necessarily have to be separate organizations. On a small project, a
single department could divide the development and testing responsibilities. On a very small project, developers will
often be called upon to wear a “testing hat” from time to time. As long as the separation of development and testing
is achieved functionally, it need not be achieved organizationally.

Most Cleanroom testing is behavioral testing\footnote{We will use the term “behavioral” testing for what is often called “black box” testing.} because that kind of testing emphasizes user-visible quality attributes
such as the likelihood of failures. Cleanroom’s historical emphasis on behavioral testing over structural testing has
several reasons:

- Behavioral testing does not require any knowledge of the software’s internals. So, it can be applied to software
  that contains elements obtained as object-code only.

- Behavioral testing is not biased by knowledge of internals. This, combined with the separation of testing and
  development functions, makes it less likely that the developer and the tester will commit the same errors.

- It does not require construction of environments for the testing of internal interfaces.

- Constructing the test plan and scenarios contributes to a better understanding of requirements.

This is not to set aside or downplay the importance of internal qualities such as reusability or maintainability, but just
to say that these qualities are assessed during review and not through testing.

Most Cleanroom teams will further refine their testing practice to emphasize profile-based testing. This kind of
testing uses a model of the system’s expected usage (called the operational profile \[15\]) to guide the selection or
generation of test cases. This is different from other forms of black-box testing where one partitions the input domain
(either using behavioral or structural aspects) or selects inputs purely at random.

The operational profile assigns a probability to each possible use of the system. A simple example might be that of a
transaction-processing system. We could begin by addressing the likelihood of each kind of input, say “10% of the
inputs will be ADDs, 50% will be DELETES” and so on. We could then add to this by looking at data that
accompanies the ADD or DELETE transaction and determining the relative likelihoods of “bad” versus “good” data.
Next, we could look at transaction streams: we might define a “session” to be between 0 and 20 transactions,
uniformly distributed. But we will quickly discover that some kinds of users are more likely to have long sessions
(e.g. novice users) while others can get their work accomplished with greater efficiency. So, we could construct
individual profiles for each user category, and combine them using an estimate of the size of each user community.

Once a usage model is constructed, test cases can be created, acquired, or generated that reflect the likelihood
estimates in that model. So, taking the preceding example, over all the possible test cases, 50% of the transactions
will be DELETES, but we may have sessions that contain many more or many fewer than that.

Testing based on the overall expected usage model is often supplemented with stratified testing. There, the
likelihood of a subclass of tests is artificially elevated to give insight into particular quality aspects. For example, one
could imagine creating a test stratum that is designed to answer the question, “what is the reliability of the control
software after the machine enters a ‘safety’ state?” Or, “what is the reliability of the software assuming the user is a
novice?”

There is one more refinement of profile-based testing: statistical testing. The usage model defines a probability
distribution for the entire domain of inputs. Statistical testing samples that distribution, conducts tests, and estimates
field reliability (usually stated in terms of mean time to failure, or MTTF) based on the results. There are numerous
reliability models into which the testing data can be entered for that prediction, for example the reliability growth
models of Musa et al. \[16\]. The Certification model \[17\] and Markov models \[18\] are most often used in Cleanroom.

There is considerable debate in the software testing community over the value of statistical testing. Opponents find
fault with it on the following grounds:

- It is difficult to select a reliability model that relies on assumptions which are accurate for the software being
tested.

- It is difficult to obtain a usage profile that is sufficiently accurate to predict safety-critical reliability levels.
• It is difficult to know what the target MTTF ought to be and thus when to stop testing.

• For systems in which the reliability requirements are very high, an enormous number of tests must be executed to predict that reliability with any level of confidence.

• Because these tests are usually generated by a tool, they are difficult to check automatically and the costs (and risks of oversight) associated with human checking are high.

• Statistical testing, while good at measuring reliability, is poor at finding bugs.

All of these issues must be placed in perspective. If we are building safety-critical systems, then ultra-high reliability and a perfectly well-known MTTF are not our goals. Instead, our goal is to apply our project’s resources as cost-effectively as possible. If we are building safety-critical systems, we may be willing to expend the extra effort needed to obtain statistically valid results. In either case, we will see that statistical testing will often be supplemented with other formal testing methods.

Testing and Design Interact

In practice, it has been almost impossible to completely separate the development from the testing function. On most projects the testers and the developers are colleagues and friends. Testers may have been drawn from the development ranks and vice versa, and it is unlikely that the testers are so completely unaware of design issues that they can do “pure” black box statistical testing. In addition, we will often want to use design knowledge to simplify the testing job. For example, suppose we are testing a graphical user interface (GUI). A purist view of the statistical usage model would have to allocate likelihoods to both the mouse-click and the keystroke for every menu-bar pulldown. Is it worth our time to determine what the relative likelihoods are, and then run sufficient test cases on each? Or can we assume that the operating system will take care of both stimuli in the same way? More often than not, we will accept the risk that a significant failure could occur in one but not the other, and focus our testing on the “real” system behavior. Likewise we may be planning to develop a product and then port it to another environment. During design, we become aware that 90% of the system is likely to be unaffected by the port, and we know which 90% that is. Now a ‘pure’ operational profile would assign a likely platform to each test case based on expected market share, so that a certain percentage of the tests would be run on each platform based on that likelihood. To have all of the platforms (and environments) running simultaneously may be expensive, so we may prudently decide to do 90% of our testing on one platform and then conduct the rest after the port of the tested system is complete. Again, any such decision is based on the idea that it is an acceptable risk to deviate from a pure profile.

Whenever we consider such corner-cutting, however, we need to be cautious that the developers and testers don’t know too much about the others’ activities. The less the testers know about the development internals, the less likely they will be to make similar mistakes.

Operational Profile Valuable Early

Another way that testing interacts with design is through the operational profile. In most Cleanroom development, the testing team is developing the model of expected usage at the same time the development team is completing the behavior specification and beginning design. (Cleanroom projects with fewer staff resources or with less-well-defined requirements may serialize these operations, completing the initial behavior specification before beginning with the operational profile.) In either case, information about anticipated system usage modes is available to the designers early in the software development life cycle. Indeed, a set of test cases can even be produced based on the model and can help the designers allocate system resources (e.g., likely events may be given higher priority).

Recent projects have begun to use the use case modeling technique both in requirements elicitation and then in operational profiling. This natural progression has been a part of Cleanroom for a long time - first ask “who are the users and what are their needs,” then devise both a functional specification and an operational profile. However, use cases and associated modeling tools offer a formalism that shows great promise for integration into Cleanroom.

Directions and Recommendations

Although developers and testers should be careful to keep their roles separate, there are many times when they must collaborate to achieve the goal of delivering high quality software. Measurement structures and incentives must be carefully defined so that an adversarial role does not develop between these groups. The strictly orthodox separation
of these groups may be questioned and changed as long as the risks engendered are well understood and documented.

**Lesson 2: Statistical Testing is an Engineering Challenge**

Let’s be frank: the examples in the literature that show operational profiles (sometimes called usage models or usage specifications) are trivial compared to the problem of building a real operational profile that can give meaningful reliability data. Of course, the same is true about implementation, too: programming textbooks show simple examples; designing million-line systems is incredibly difficult. The point is, both the operational profile and the system implementation are significant engineering challenges that require planning, process, resources, and creativity. What we have learned in fifteen years is that the specific techniques chosen for operational profile construction and statistical testing must be carefully tailored to meet specific project risks for which they are well-suited.

Every task in the development process focuses on some aspect of enhancing the likelihood of project success. Any software process improvement effort must continually evaluate the development tasks to ensure that they are the most cost-effective ways of contributing to that eventual success.

The Cleanroom process has traditionally seen testing as contributing to risk-reduction in three ways:

- Reduce the likelihood of user dissatisfaction by providing high reliability. Using an operational profile for testing is valuable in this regard, because the most likely product uses are tested early and often. Of course, the product quality goes in during development, but reliability growth is most rapid when the product is tested the way it will be used.
- Reduce the likelihood of unexpected service costs (e.g. technical support, repair, updates) by having a reliability prediction before release. Statistical reliability testing provides this knowledge, though its accuracy depends on the amount of resources devoted to that kind of testing.
- Provide objective measurement of the impact of upstream process changes on reliability, for the purpose of continuous process improvement.

As we will see later, there are additional process risks that may be cost-effectively addressed through testing that aren’t well-addressed by statistical testing. In this section we will look at the statistical testing component - its processes, techniques, and outcomes.

**A Statistical Testing Process**

The statistical testing process can be broken into four work areas: profile development, testcase acquisition, results evaluation, and reliability modeling.

If we think of a program as a mathematical function for transforming a domain into a range, then the operational profile is neither more nor less than the definition of a probability density function for the domain. Simple, right? Except that the domain in this case is colossal! Take a simple compiler for example: if its inputs are a small set of command-line arguments plus an input file of text, then the domain is the cross-product of all of the possible command-line arguments and all of the possible input files of text! It’s an almost infinite set and this is a small example. And we have ignored the influence of environment (available file handles, memory, etc.) which also should be included in the domain. How can we possibly assign a likelihood to each and every point in this enormous domain? The answer is, we assign them by classes. Just as the functional specification for the compiler breaks down all the possible inputs into classes (like “syntactically correct input file”) so can the probability function. We might, therefore, begin to consider a hierarchical organization:

input file, of which
  - 20% are syntactically correct
  - 80% are syntactically incorrect, of which
    - 70% have 1 or 2 syntax errors
    - 20% have 3 to 10 syntax errors
    - 10% have more than 10 syntax errors
We can also characterize the distribution of syntax errors throughout the input file (we might assert they are uniformly distributed) and their nature (for example we might distinguish between subtle and obvious errors, or key-transposition errors, or the like). The lesson here is that the operational profile can’t just be concerned with the distribution of individual commands or keystrokes, but must also address the semantics of the domain.

Assuming that this sort of profile can be built and modeled using some kind of tool, then the task of generating test cases becomes rather trivial: take a random number generator and pump out test cases using this structure. 80% of them will be somehow syntactically incorrect, and 20% of them will be syntactically correct. Unfortunately, designing an automat that generates a truly representative operational profile is extraordinarily difficult. In the next section we will look at some tactics for addressing this problem.

After usage modeling, the hardest part about statistical testing is judging whether or not the test case executions succeeded or failed. When test cases are hand-crafted, some amount of self-checking may be built into them. When test cases are re-used over and over, the results of later uses can be compared against those from earlier uses. But statistical testing requires a very large number of new test cases in order to produce accurate reliability estimates. Determining whether each and every one of these test cases ran successfully is a big job. There are some ways that the task can be made reasonable. For example, if we are applying statistical testing to an improved version of an existing product, then the same testcase can be supplied to both products and the results compared. Or, we may be able to add knowledge to the generator tool so that it also generates some information about how to evaluate testing progress.

A technique that is being used on at least one Cleanroom project is to rely on the testability of the software. Voas [19] defines a testable program as one in which an incorrect state will quickly manifest itself in a failure that is evident to the tester. Our idea, then, is to use two phases of testing. During the first phase, a relatively small number (a few dozen) test cases are run and closely (manually) checked against specifications. After that phase is complete and any faults it locates are corrected, the second phase can begin. During the second phase, we run a very large number (hundreds or even thousands) of test cases. Instead of manually checking every aspect of the output, we look for signs of failure including error messages, dumps, crashes, and hangs. This can be done by logging output to files and applying standard search/filtering techniques. If the software under evaluation is sufficiently testable, this method should not miss very many serious failures that occur. (Of course, failures like erroneous message text and the like will have to be caught in the first phase.) Now, testability is a design criteria, and this is another example of a place where the testers and designers must work together. If this testing method is to work, designers must specifically work to make the software very testable and testability assessment must be an explicit part of every design review.

The fourth phase of the statistical testing process is the estimation of reliability based on statistical models. Lyu’s excellent reference [20] contains several models, and there are many more in the literature. Discussion of modeling specifics is beyond the scope of this paper; suffice it to say that most Cleanroom teams will select a set of models based on an understanding of their particular strengths and weaknesses. For example, some models are more accurate when they have a lot of failure data, while others are more accurate on a long string of successful executions. It is probably a good idea to start out with several models and evaluate their predictive accuracy through the first few increments before settling on any one.

**Incremental Development of the Operational Profile**

The incremental development process is the primary technique in Cleanroom for managing risk and incorporating risk mitigation into the Cleanroom process. An increment is a functional subset of the eventual software product. It can be tested in an environment similar (if not identical) to that of the final software, so inferences about field quality can be drawn from test results. Each increment makes maximum reasonable re-use of prior increments. By having many increments within a project, the results of one increment can be used to tailor the process for subsequent increments. Incremental development is similar to evolutionary delivery [21] in many respects. Incremental development also incorporates aspects of the spiral model [22], especially when risk assessment is used to plan increment content.

Although the initial papers on incremental development (and most papers on spiral and evolutionary models) think of the increments as primarily delivering implementation, the usage model itself is a significant work product that can be nearly as complex as the implementation. It is therefore useful to consider delivering it in increments as well. Since most of the usage modeling languages (including Markov chains and probability grammars) are hierarchical in
nature, it makes sense for an early increment to leave certain parts unelaborated and then to elaborate them further in each increment. To use the preceding example of mouse clicks versus keystrokes, in the first increment the operational profile might choose one or the other at random, or might choose only keystrokes; in the second increment we could add a single probability distribution for mouse clicks versus keystrokes; in the third increment (if necessary) we could consider how that distribution might be different for expert versus beginning users.

Most teams will take a moderate incremental approach by gradually acquiring test cases in addition to generating them. By the time the compiler-builders get to optimization, they not only want syntactically correct programs but they want a lot of “meaningful” programs that contain patterns similar to those humans would create. The compiler-testers might, for example, use a fairly crude profile for the first few increments of the compiler, improving it modestly over time. Then for a few increments they might rely on buckets of programs (mostly syntactically correct, but they can supplement with generated incorrect ones) acquired from a variety of users. For their final few increments they will want to be measuring the product in “live” use either by their peers or by “beta” customers. Using this approach, the “semantic gap” in generated test cases can be bridged and a meaningful statistical reliability estimate will be available when the product approaches release readiness.

The considerations for incremental development of the usage model are the same as they are for the implementation: good initial planning and an understanding of requirements leads to architectural structures (that organize the usage model) which are resilient to incremental improvement.

The great advantage to any iterative process, whether it be spiral, evolutionary, or incremental, is the ability to use feedback from early iterations to improve the process and thus the chances of success. However, we must be careful not to read too much into the earliest data. The results of “statistical” testing when applied to the first increment of code based on the first increment of the operational profile will be a slim reed on which to judge a process. Subjective metrics, as well as traditional metrics such as lines of code and labor hours, will be needed throughout the process to supplement the statistical measurement. But nowhere is this more true than at the beginning of the process.

**Tools Support Important**

We have learned that statistical testing, perhaps more than other parts of the Cleanroom process, is an area where good support tools are essential. Fortunately, several vendors are beginning to develop products in this area. It is not our goal to catalog all the tools that are available, but to mention the areas in which tools support is most critical.

Profile development and recording tools are making strides. The ToolSET_Certify tool from Software Engineering Technology, Inc. is a full-featured tool for recording usage models using Markov chains, generating test cases, recording results, and predicting reliability. It also has several other features that make it useful for non-statistical testing as well. Cleanroom Software Engineering, Inc. has a prototype tool, “CleanTest,” that provides recording and generation features based on probability grammars. For several years the IBM Cleanroom Software Technology Center was offering a usage modeling and testcase generation tool as part of its service offering; the status of that tool is presently unclear. All of these organizations have offered consulting services to help teams get started using Cleanroom and statistical testing. What appears to be missing is the ability to load these modeling tools with information captured from actual use. It would be nice, for example, to instrument the current version of the Foo product so as to capture patterns of usage, and feed them into a modeling tool to guide statistical testing of the next Foo version.

The testcase generation tools seem, for the most part, to be format-independent. So, for example, they could generate scripts that could be used by GUI playback tools to conduct hands-off testing of GUI interfaces. However, none of the generation tools are known to be specifically linked to any particular playback tool.

There are also tools to fill the last niche, reliability modeling. Lyu’s handbook includes a CD-ROM version of SMERFS and other reliability-modeling tools. The SMERFS toolset allows you to present the same set of raw data to several models and ask it to decide which is the best fit for that data. ToolSET_Certify contains its own statistical modeling capability.

**Directions and Recommendations**

Statistical testing is by no means a perfect solution to every problem. It is hard work, and it gets harder the more you demand of it. That is why it is important to set statistical testing in the context of overall project risk reduction and to use it appropriately. Because of its complexity, statistical testing requires competent staff and advance planning. But
it has many side benefits in addition to its value in predicting field reliability. We expect that statistical testing will become a typical component of the overall software testing process as its methods and techniques become better-supported by tools and training.

Lesson 3: Techniques Must Be Appropriate to Project Goals

Although statistical testing is a useful and important technique, we believe that the early Cleanroom proponents were wrong to suggest it be the only testing technique used in Cleanroom development. Indeed, the most significant lesson we take away from fifteen years of Cleanroom application is that statistical testing should be supplemented by other forms of testing in the vast majority of projects.

Cleanroom teams have been stepping back from the initial strident orthodoxy since the earliest days. Although Harlan Mills’ article in the Encyclopedia of Software Engineering [23] still asserts, “the Cleanroom development team does not test or even compile (emphasis added),” in fact most of today’s Cleanroom development teams are compiling their code prior to submitting it to Cleanroom testing. It’s all a question of balancing costs and benefits. True, if developers could be prevented from compiling then they would effectively be prevented from conducting ad-hoc, off-the-books testing. And the number of syntax defects probably provides a clear pointer to which modules have been thoroughly reviewed (so a large number of syntax errors probably correlates to a large number of other errors). Nevertheless these benefits are probably offset by the benefits of having programmers submit clean code to testing. This is especially true in today’s environment where nearly every project seems to be using a new language, compiler, or environment, and thus the compilation process provides feedback that is necessary for language-learning.

In this section, we will explore some of the ways Cleanroom teams are supplementing statistical testing.

Experimentation

Before proceeding, it is probably important to distinguish between testing for the purpose of finding bugs and experimenting for the purpose of understanding languages and interfaces. The former activity is prohibited to developers on a Cleanroom project: we want to force developers to do good, thorough verification-based inspections, and our technique for doing that is to prohibit ad-hoc so-called “unit debugging.” Although in the modern world it may be impossible to physically prevent such testing, we believe it can be significantly limited through peer pressure and other management techniques. Further, if a programmer has a full work-schedule comprised of his or her own design plus participating in others’ reviews, any unit debugging will have to be done on his or her personal time. Our experience is that, after one increment of Cleanroom development, such “midnight debugging” will die of its own accord.

Experimentation, on the other hand, is sometimes a necessary evil. If we are writing a program that interacts with an existing program or device, the latter may be poorly documented even though we want to use Cleanroom verification for the former. We may need to resort to testing the existing program in order to discover its behavior. Although there are risks that accompany this strategy, it is often more cost-effective than trying to obtain internals information about the device from its maker. Deck [24] describes additional techniques for applying the Cleanroom process to “old code.”

Coverage

Statistical testing is a tool that enables reliability estimation. Hamlet has observed that statistical testing is the only way to achieve confidence in system reliability. As he puts it, “confidence in daily performance can be gained only by testing that mimics the ‘operational distribution’ of typical usage.[25]” However, there are other things in the world than confidence in system reliability, and these may be cost-effectively addressed through various forms of coverage testing. Here are a few of them:

- Safety-critical software. Statistical testing emphasizes the overall reliability of the software. It will therefore spend comparatively more time testing and re-testing frequently-used code paths. In cases where a very unlikely code path could lead to a catastrophic response, it behooves us to know whether those code paths have been tested. And, if they are not, we may want to construct test cases (outside of the statistical framework) to test them.
• Coverage as a customer requirement. Although we may work to change this attitude, the fact remains that some customers demand, as a deliverable, some certification that all statements, branches, or claims have been tested. We cannot recommend that you turn down business opportunities just because they aren’t “Cleanroom.”

• Checking the statistical model. Just as we can make mistakes in the software, we can also make mistakes in the statistical model. By measuring some form of code coverage during the statistical test, we can determine whether the model has left significant gaps in its coverage of the system usage.

• Certification of reusable components. In addition to developing a system for use, we may also have a requirement that certain components of that system be reusable in other environments. To be sure, we could conduct independent statistical testing on the components themselves, but if our reliability needs are not that high we may just want to ensure that the system testing has provided sufficient coverage.

• Finding bugs, when the design team cannot be convinced to use Cleanroom development techniques such as verification-based inspection. Statistical testing, while it is good at measuring reliability, is not as effective as coverage testing for the purpose of finding all (or even most) bugs. It re-covers the same paths many times, and its effectiveness in bug-finding decreases the more testing is done. While the virtues of the Cleanroom development techniques have been extensively argued, sometimes the testers are the pioneers on the road to full Cleanroom adoption.

In all of these cases, we stress that the first principle is using specific testing techniques to cost-effectively address particular project risks. There are significant costs to conducting formal coverage tests, and the benefits of those tests should be weighed whenever their use is considered.

We also point to data collected by Adams [26] and analyzed by Mills [27] in which it appears that coverage testing focuses the majority of the testing effort on finding those bugs that are least likely to actually occur in use, while profile-based testing focuses majority of the testing effort on finding the bugs most likely to occur in use. Worse yet, remember that fixing any bug carries with it some probability of introducing a new bug. Although the Cleanroom experience suggests that this probability is quite low in Cleanroom projects, nevertheless it is nonzero. Now, suppose you spend a lot of resources tracking down a very unlikely bug, only to introduce a much more likely bug in its correction. In this case the testing/debugging process has worsened the reliability of the system rather than improving it.

What some Cleanroom projects are doing is using coverage analysis during the statistical testing to determine how thoroughly that phase of testing has covered statements and branches. Then, if the level of coverage is insufficient and it is deemed to be cost-effective based on the specific coverage gaps, supplemental “crafted” test cases are produced and executed outside of the statistical model. If that supplemental testing uncovers faults, we want to analyze those faults to estimate their impact on reliability and other quality attributes.

**Regression and Re-Test**

There is yet another aspect of “orthodox” Cleanroom testing that astounds many in the “traditional” testing community: for the most part, running the same test case more than once is discouraged! From the point of view of the statistical testing purist, you can understand why this might be so.

• Re-running the same test cannot contribute to the statistical model because it isn’t random.

• Failures or faults found in the re-test are hard to account for in the statistical models.

• The purpose of testing is to measure quality, so trying to “see if we fixed it” or “to make sure we didn’t break anything else” are both better done in design review rather than in testing.

• If the problem didn’t get fixed, it is likely to occur again in another random test case.

While all of these statements may be mathematically true, they deny the economics of the situation: once you have run a test case, re-running and comparing the results it is cheaper than running a new one that will have to be evaluated.

In order to have the best statistical evidence, and still have a cost-effective process, what we recommend is to conduct all re-testing after the bulk of statistical testing is complete. Further, whenever a change is made to the system, to conduct at least some amount of statistical testing afterwards, rather than relying solely on regression.
against buckets of previously-used test cases. That way the system isn’t “trained” to perform only on a specific set of test cases, and is constantly exposed to new random test cases.

**An Improved Cleanroom Testing Process**

Given the preceding suggestions, the following testing phases are proposed for a Cleanroom testing process that includes statistical testing as its major component.

1. **Build and cycle.** Remember, prior to this point the developers aren’t doing any testing at all, just verification-based inspection. It therefore seems prudent to allocate a special testing phase to getting over any initial faults that may prevent “real” test cases from executing. This is especially true when new hardware or interfaces are integrated at testing time. This testing can still be based on the usage model, but the idea is once the first stimulus (usually “start” or “power-on”) produces its response, proceed to the next phase. This phase should take care to be as scientific as the others, recording execution times, failures, and faults so that this data can be incorporated into the final metrics and models.

2. **Initial profile-based testing.** This is testing using the operational profile per the preceding section. Reliability models may be used but their predictive value may be low. However, some reliability models that base their prediction on the number of failures may be useful here. We strongly recommend that profile-based testing precede supplemental testing because profile-based testing focuses on how the user will use the system, so the most-likely failures are dealt with first. This phase of testing ends when, in the subjective judgment of testers and managers, the system is stable.

3. **Supplemental testing.** Now we are free to use coverage, re-test, and other methods on an as-needed basis. This phase ends when coverage targets are reached and all necessary re-tests are complete.

4. **Final statistical testing.** Certain statistical models are sensitive to failures and need a lot of successful execution to predict a valid reliability estimate. These models can be used on a very stable system to produce a final estimate of system reliability. Ideally, this phase ends when a pre-selected reliability target is attained. In practice, the selection of such a reliability target is rare, and testing ends when other subjective and objective measurements reach satisfactory values.

In any phase, it could be decided that progress is too slow and that testing should be sent back to an earlier phase. Or, the initial phases (1 and 2) may decide that the code is of poor enough quality that it should be sent back to development for rework and re-review. This usually happens about once on every project, as developers become complacent through early success and send an increment into testing without adequate review. It is very important that managers recognize when the number of failures indicates defects in the development process, and “resets” that process rather than trying to test-in quality.

**Directions and Recommendations**

We have seen that the amount of effort allocated to the various parts of the process, and the level of automation, are tailor able. What is important is that each project understand its needs and that it choose testing techniques that are appropriate to meeting those risks. At some point in that tailoring, a process probably ceases to be “Cleanroom” any more. However, we are hard-pressed to say exactly at what point that is. Practically speaking, the prohibition of unit testing by developers, and the use of some form of usage-driven testing are probably the hallmarks of a minimal Cleanroom process.

**Lesson 4: More Information is Needed**

Finally, we also learned that much more information is needed by projects that want to use Cleanroom. Currently, most projects will be successful with Cleanroom only if they receive assistance from expert consultants who can guide them through the tailoring process. Such consultants exist, but they will be unable to fill the growing demand for Cleanroom. We believe that four kinds of information are needed:

- Experience reports that describe the actual application of Cleanroom practices to real projects. The best of these will say what projects did and did not do, and will describe specifics of the projects so that other adopters will know how similar their task may be. These experience reports should eschew lengthy discourse on the “ideal” Cleanroom, but instead focus on detailed descriptions of what they really did and didn’t do.
• Practical “how-to” guides for novices. The bulk of the Cleanroom literature describes the process in sufficient detail to motivate its adoption, but not thoroughly enough to guide a project through its first increment. Such a process guide would necessarily be quite large, but it should have easy-to-use checklists as well as discussion of tailoring opportunities.

• Sample Cleanroom work products ranging from requirements to verification checklists to the output of statistical models. These are extraordinarily difficult to obtain, however, since most Cleanroom projects are developing proprietary information. Further, there is some cause for concern that a team may copy a specification form (for example) that is ill-suited to its needs and thus come to grief. The work product samples must therefore be accompanied by the how-to guides that describe background and tailoring.

• Leading-edge research. Although the technologies underlying Cleanroom are not new, there are still interesting questions that require further study. These include questions about specification and verification in the real-time environment, the precise relationship between various object-oriented methods and Cleanroom, and the applicability of the various statistical reliability models.

Conclusions

Cleanroom software engineering is continuing to evolve within the guidance of its fundamental principles. Although members of the first Cleanroom generation may be shocked to see use espousing the use of formal coverage testing, we believe Cleanroom must grow and expand to meet new challenges of the marketplace. The lessons we have described are just a few of the valuable results of fifteen years working with Cleanroom projects. We expect that the next fifteen years will see acceptance of Cleanroom as a widespread best practice and that it will continue to be adapted to meet the needs of its users.

Authors

Michael Deck is an internationally-recognized expert in Cleanroom software engineering practices. His company, Cleanroom Software Engineering, Inc., specializes in training project teams to tailor and use Cleanroom practices to solve real-life software process problems. From 1982 to 1993 he was a member of the IBM Cleanroom Software Technology Center, where he worked closely with the inventors of the Cleanroom approach. He has a BA in Mathematics from Kalamazoo College and an MS in Computer Science from the University of Maryland, College Park. His current research interests include application of Cleanroom to object-oriented development, real-time and embedded software, and highly reliable systems. He has published widely on various Cleanroom topics.

Dr. James Whittaker specializes in formal methods for software testing and has worked with Cleanroom testing for about nine years. From 1989 to 1996 he worked for Software Engineering Technology where he consulted around the world on Cleanroom testing projects. Currently he is on the faculty in the Computer Science department at Florida Tech and is the Director of the Software Reliability and Testing Laboratory which focuses on stochastic methods for software testing and reliability analysis. He received the BA in Computer Science from Bellarmine College in 1987 and the MS and Ph.D. degrees in Computer Science from the University of Tennessee in 1990 and 1992, respectively.

References

Note items marked with * can be obtained in softcopy via the Cleanroom web site, http://www.cleansoft.com/cleansoft.