An Industrial Application of Cleanroom Software Engineering - Benefits Through Tailoring

Robert Oshana
Raytheon TI Systems
oshana@ti.com

Abstract

Cleanroom is a set of software engineering principles that support the development of reliable software. The Systems group at Raytheon TI Systems, a SEI level 3 organization, successfully adopted Cleanroom into a pilot CMM level 5 project. The successful introduction of this technology was a result of the principles of Cleanroom being based on fundamental computer science foundations. As with any other methodology, a certain amount of tailoring was required for the technology to be most effective. Our tailoring approach was based on our project needs, our schedule constraints, and how comfortable we felt with the various components of the technology. The paper will describe the tailoring approaches used for the insertion of Cleanroom and the pros and cons of the Cleanroom approach as described by practitioners using the technology day to day. The result of our approach is a model for software development that we feel is very effective at producing quality software.

Introduction.

Cleanroom software development methodology is a method of developing software under statistical quality control. It is a theory based, team oriented engineering process. Cleanroom attempts to develop very high quality software by preventing errors through the application of a sound engineering discipline rather than focus on error detection and removal. Cleanroom Software Engineering is based upon the structured programming ideas of focused thinking and the application of mathematical reasoning to software in a methodical way [1,2].

Cleanroom Software Engineering attempts to replace crafting of software with mathematically based software engineering practices. Crafting applies human creativity in designing a solution to a technical problem and often results in ad hoc solutions to technical problems. The processes and practices often come from personal experience. In engineering, the problem is solved using practices derived from an appropriate science base. Cleanroom provides uniform, scalable processes and artifacts drawn from a science base for software engineering. These processes and artifacts exhibit properties important for effective software engineering, and provide a framework for applying human creativity to the problem at hand.

The Cleanroom development process is a set of stepwise refinements or transformations from the requirements to the code where each transformation is verified to the previous level of refinement in order to minimize errors. Errors are found earlier in the software development cycle which minimizes rework and speeds time to market.

Historical data shows that the application of Software Cleanroom practices improves delivered software in the following ways:

1. **Improved quality**; The Cleanroom process builds in quality which reduces the overall cost. The goal is to prevent errors rather than accommodating errors.
2. **Increased productivity**; The improved quality of the software reduces the time spent in debugging and rework which increase productivity and reduces cycle time.

1060-3425/98 $10.00 (c) 1998 IEEE

3. **Improved software maintainability**. Software developed using Cleanroom techniques has clear, well-defined specifications and a less complicated design which allows maintainers to learn and modify the software faster with fewer errors.

Cleanroom Technology Components.

The main Cleanroom technology components are (Figure 1) [3];

1. **Software specification**; During the specification phase, the software engineers use a strict stepwise refinement and verification process using a box structured approach that allows precise definition of required user function and system object architecture. This approach scales to support large systems development.
2. **Development**; The main goal of the development team is to take a set of software specifications including hardware and human behavior components, and design and verify those behaviors using data and processes that implement the given specifications [4]. Box structures, representing defined system behavior, are used in development. The goal is to develop software that can be verified for correctness against its specification using structured programming correctness proofs.
3. **Correctness verification**; During development the software is verified using strict correctness verification methods that prove that the software meets the specification. Verification reviews are held by the team to formally or informally verify the software using a set of correctness proofs that are standard in a structured programming environment. This method allows the verification of programs of any size by reducing the verification process to a manageable number of independent verification checks. Proof of software correctness is mostly done by direct assertion. Correctness verification is done before the software is ever executed, so the developers avoid a "debugging" mode of operation.
4. **Certification**; The Cleanroom software engineering methodology focuses on certifying the product meets requirements and determining the reliability of the product rather than attempting to test quality into the product. A separate certification team tests the product using usage model based statistical testing. Usage data is used to develop Markov models of the system that allow significant analysis capability both pre and post-test, and straightforward test case generation. This testing approach provides statistically valid quantitative measures of testing progress and software reliability. The quantitative measures provided by the usage-based statistical testing approach provide mechanisms to aid management in determining when testing can and should be stopped [5,6]
5. **Incremental development**; The software product is developed in a series of functional increments that sum to the final deliverable product. These increments represent operational user functions. The integration of these increments is done top down. Incremental development allows early assessment of product quality and gives continuous feedback to management as to the progress of development. When the final increment is integrated and tested, the software product is complete.

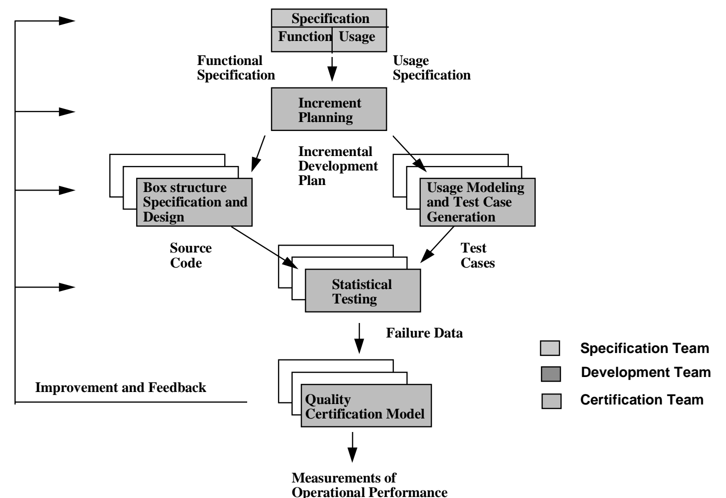

Figure 1. Cleanroom Software Engineering process

There were several motivating factors that led us to consider incorporating Cleanroom methodology into our process:

1. Our schedule required a lower defect rate going into system integration and test. In the past, we used a strict unit testing and branch coverage approach to testing. The program spent too much time and money developing the unit test environment, trying to get every instruction to execute, and not understanding how each module fit into the big picture.
2. Our paradigm for software development was that defects should be avoided instead of detected and removed. Cleanroom focuses on the same goal.
3. A more formal approach to software specification was desired. Instead of the standard English language requirements document, more formal and rigorously defined requirements were desired. In the past changing and unstable requirements caused a lot of rework and uncertainty. Cleanroom specification approaches allow stepwise decomposition and enumeration to more fully and formally specify software requirements. In the past, our software specification was, effectively, a single unit (document). Customer changes (which seem to be ubiquitous) could easily wreak havoc on the specification and development processes. In the past, we found ourselves in a mode of constantly updating documents and sending new versions to the customer. In our current approach using incremental development and design, increments are specified and built with a stable set of requirements. Requirements changes are folded into future increments. We are finding this approach easier on ourselves and the customer.
4. Any technology improvement must not require the software development team to go through a lengthy learning phase. We felt that Cleanroom was an extension to the structured analysis and structured design techniques we had used in the past. Cleanroom is not an orthogonal approach to SA/SD, OO or any other methodology. It's simply an approach that uses fundamental computer science principles in the development of software.
5. Cleanroom permits replacement of unit testing with static verification. Although we felt that some unit testing was necessary, we felt that static verification in the form of code reading and formal peer reviews and code inspections were cost effective. We did not feel comfortable enough using the formal techniques of mathematical verification to completely replace other testing techniques.

Project Organization.

Cleanroom has been incorporated into our mature SEI level program using technology insertion guidelines put in place as part of our SEI level 5 piloting effort [7].

Our program consists of multiple software teams, each developing software at the CSCI (computer software configuration item) level in a teaming environment. Each team is developing their software in a paradigm of "mini" Cleanroom projects. In order to maintain consistency among the teams with respect to Cleanroom artifacts, lessons learned, and best practices, a CSCI convergence team was organized. This team meets periodically to listen to and act on concerns and issues that arise from the teams. This convergence team has proved helpful in keeping all of the teams consistent in the deployment of the methodology (Figure 2).

The software being developed is real-time embedded and distributed software written mainly in C and ADA. The software itself consists of control software running on single board computers, and algorithmic and digital signal processing software running on digital signal processors.

Mid way through the implementation phase and the initial certification phase, we assessed the current status of the program using Cleanroom methods. The results of our in house survey are described here.

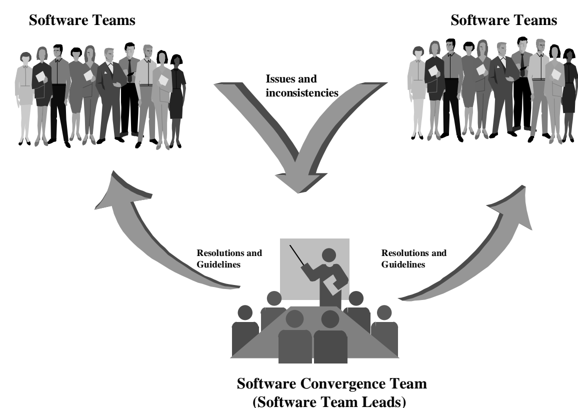

Figure 2. Software convergence team

Survey results.

The survey focused on those Cleanroom principles software engineers liked most and those that they did not see as being useful. The participants were also asked what they thought were the barriers to more ubiquitous Cleanroom use and what best practices they could take from the Cleanroom process. Of the many Cleanroom concepts we were introduced to, some were more useful and applicable in our industrial setting than others. The usefulness of various Cleanroom concepts for our program are shown in Figure 3 (7 being the most useful).

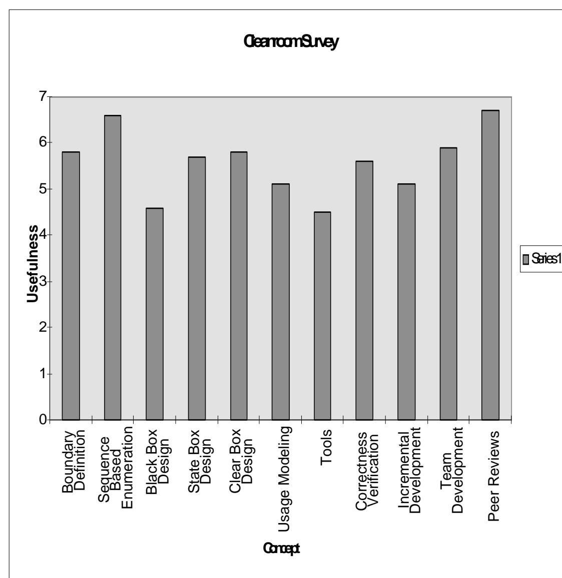

Figure 3. Usefulness of various Cleanroom principles

The respondents, regardless of whether they were working in control related or algorithmic software, thought that rigorous specification using sequence based specification and box structures was well worth the effort. Just about every engineer commented that they found problems in their designs during this process that they otherwise would not have found until later in the development process. Our observations led us to believe that the rigorous specification process effectively elevated the design skills of all of the engineers to a common level. Consistent use of box structures across the various teams led to easier to understand designs at the same level of detail for each of the teams. Formal specification tools like Z and VDM were not required or used.

Team peer reviews also ranked high. The whole teaming approach to developing software was not new to us. We had been using a teaming approach to software design for quite some time. Teaming concepts were not new to the engineers and no training was required. The peer reviews were a modified Gilb [8] approach to reviews. Aside from assigned roles and a moderator, we allowed open and useful discussion to prove designs correct. This allowed us to gain more intellectual control of the product. As a requirement for the review, a software certification team member had to be present. We applied the informal structured programming correctness verification questions loosely [9]. In many cases, they were not used during the peer reviews. In several special cases, a formal proof of correctness was performed to verify algorithmic correctness. This was done only when required.

Many of the concerns about the development of box structures were focused on the available tools. In particular, the box structures approach to design requires a lot of time to develop the sequence based enumeration tables, the state boxes and the clear boxes. We developed an in-house tool using excel macros to automate the process of generating enumerations that was very helpful. Still, creating the other box structures was time consuming to go through the first time and almost impossible to re-do later on if the design changed and still hold schedule. The only commercial tool available at the time of our design was for certification (a specification tool will be available shortly). We found the box structures to be very useful, but if was difficult to keep them updated.

Usage modeling received mixed reviews. Some respondents thought the concept of testing with usage models to be very useful. Others had a hard time buying into this type of testing as the sole source of testing. Indeed, the whole testing approach was tailored in this project. Usage models were created for all of the operational software and used as a base of software testing. Additional “hand crafted” tests and unit level tests were also used to test some of the algorithmic software. There was some difficulty in developing the models at the right level of abstraction. This was mainly due to the fact that this was the first attempt at developing this sort of model and the learning curve affect was obvious.

Most liked and least liked principles.

The survey group was asked the question of what they liked most and least about using Cleanroom (Tables 1 and 2).

Table 1. Cleanroom principles most liked

Most liked Cleanroom aspects
Sequence based enumerations
Peer reviews of every step in the process
Systematic approach to development
Rigorous specification
Forcing thought about "impossible" events
Clarity of the process
Ability to determine exact behavior
Forethought before coding begins
Simplicity

Table 2. Cleanroom principles least liked

Least liked Cleanroom aspects
More automation and cleaner tools
Background required for full understanding
Usage models
Process poor for data based software
Lots of re-work required before getting it right (re-enumeration time consuming)
Maintenance of the Cleanroom artifacts
Cleanroom artifacts not useful
Process not clearly defined
Clear box code not as efficient as expected.

Sequence based enumerations and peer reviews ranked near the top of the list in Cleanroom principles most liked. Others responses such as systematic approach to development, rigorous specification, forcing thought about "impossible" events, and ability to determine exact behavior are closely tied to sequence based enumeration. All of these responses indicate the acceptance of the teams of spending more time up front to get the design right. Follow up interviews indicated that regardless of whether the engineer had a background in SA/SD, OO, of some other development language, the process of rigorous specification was new to them and they thought the process had a big benefit to getting designs right the first time.

There were areas of concern as well. More automation and tools are necessary to allow design artifacts to be changed or modified quickly. There were many cases where a simple change would cause a lot of re-enumeration and box structure changes that frustrated the engineer. Lower on the list is "Lots of rework required..." and "maintenance of Cleanroom artifacts" which are closely tied to this same issue. Usage models were not particularly liked by some of the teams. Much of this, we think, is due to the fact that at the time of this survey, many of the teams were struggling with developing useful usage models and a lot of rework was being done. The other factor is that we were attempting to automate the testing process. This required that we map each state and arc in the usage model into the respective system stimuli and oracle (expected output) and generate these in an automated environment. We were limited to some extent by the design of the special test equipment (STE). Mapping a usage model into a STE environment and automating the whole test forced us, in some cases, to develop usage models at an acceptable level of abstraction for the STE, but not necessarily for the software. This limitation in level of detail in the usage models forced us to craft some additional tests to get acceptable coverage.

Cleanroom worked well for both types of software being developed, control and algorithmic. There were fewer issues raised for those working on the control based software. Control based software inherently allows for easily defined stimuli and responses. Usage modeling was also easier, given the well defined paths in the software architecture. The algorithmic software, by the nature of it computations, had very few stimuli and responses and was mainly a data driven application. The box structures and state data enumeration were simplified. This allowed us to spend more time understanding the complexity of the algorithms. Unit tests developed for prototype algorithm software were re-used which made the effort of producing and running unit tests easier.

Several respondents commented on the process not being clearly defined. This is due to the fact that this was the first attempt at doing Cleanroom in a project of significant size with no previous project experience to draw from. There were times when the teams would diverge in areas such as artifact templates, level of detail, etc. The convergence team was developed to address these problems and force some consistency between the groups. Some of the rework of Cleanroom artifacts was to align then

with the appearance of all of the other teams artifacts. Much of this is typical in projects using new technology for the first time.

Barriers to ubiquitous Cleanroom use.

The survey group was asked to comment on the barriers they perceived to more ubiquitous Cleanroom use. The responses are intriguing (Table 3). Some members do not yet feel that software can be treated as an engineering discipline and therefore, the engineering approach to software is not worth the effort. The majority of respondents, however, did not feel this way. The approach of developing software without a large unit testing phase appeared to be a leap of faith to some team members. Cleanroom permits unit testing, although it provides an opportunity to eliminate or reduce it, because experience shows that verification is more effective in terms of eliminating errors and reducing costs. We are allowing all levels and types of testing to be performed on the software for this project, with the statistical approach being the common underlying model for testing. One of the inherent limitations of the usage modeling approach is developing accurate probabilities to assign to the arcs in the usage model. We were able to obtain accurate data for this and the issue was not considered a major one. Compiling software and desk checking via code reading or simple unit test development is being allowed. There is not a formal unit test development phase and this does not appear to be causing problems. Some simple experiments are showing that we are getting high levels of coverage using statistically generated test cases, but this study is not complete.

Table 3. Barriers to Cleanroom Use

Barriers to Ubiquitous Cleanroom Use
Cleanroom assumes that software development can be treated as an engineering discipline but it is not (yet).
Developing software without testing it requires leap of faith many people are not ready for.
Coding using the Cleanroom method involves just turning the crank. No interesting aspects, clever inventions, elegant solutions, etc.
More up front design and detail frustrates some engineers
Does not address real time issues.
Algorithm intensive applications (little control s/w) harder to do.
Overcoming the notion that Cleanroom is a new technology completely different from SA/SD
Process seems very tedious and “front-loaded”
Lack of tools
Formal correctness verification exceeds the capabilities of much of the staff
Cleanroom is overly process oriented.

Coding too easy?

One interesting response was the concern over the coding phase requiring just “turning the crank”. No interesting aspects, clever or elegant solutions, or crafting is necessary when the software if fully specified at each level. This is exactly what we are trying to do with the Cleanroom process model; eliminate the clever elegant solutions and replace them with simple, easy to understand solutions. Coding in the Cleanroom model is meant to be just turning the crank! A related response that “more up front design frustrates some engineers” and the process being too “front loaded” are, we believe, related to the same concern over not getting on with coding and worrying about the design later. This is exactly what Cleanroom is supposed to prevent but, based on these responses, it is going to take a while for engineers that are used to hacking and crafting to start designing first.

Real-time issues.

Cleanroom does not fully address real time issues and this is a concern for us. We are using other approaches such as rate monotonic analysis, VHDL simulation, and discrete event simulation to supplement the Cleanroom process (Figure 4). There are remaining issues. For example, the Cleanroom process leads to the production of well structured easy to understand software. In our real time environment, we are often forced to optimize the software (by re-structuring the C code to take advantage of the optimizing C compiler and by writing assembly language for certain algorithms). This leads to software that now does not match the structure of the clear box. This leads to a mapping between specification and code that becomes harder to understand. When assembly language is written,

there are two mappings required; one between the clear box and the C code, and another between the C code and hand optimized assembly.

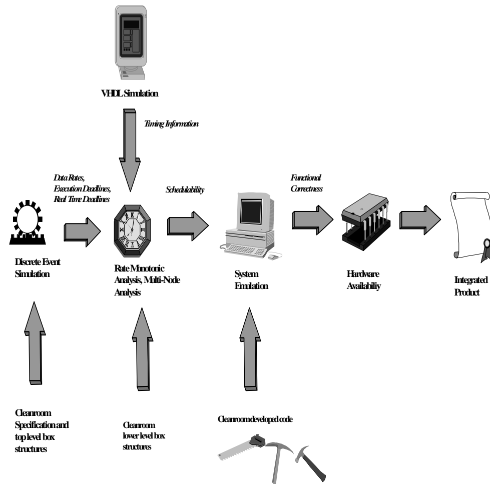

Figure 4. Cleanroom process supplemented with real time analysis and simulation

More tools required.

Cleanroom lacks the tool base that other methodologies enjoy. The tools that do exist are extremely useful. Other tools are coming on line in the near future (some of these are being beta tested by us now) that should improve this situation. The in house tools that we developed did not require a lot of effort and have proved very useful We have also gained leverage off of existing tools. For example, we used Cadre's Teamwork to develop our boundary diagrams and finite state machines. We use Excel to generate many of our box structure artifacts.

Lack of mathematical backgrounds in industry.

Cleanroom teaches us to gain intellectual control over our software using informal or formal proofs of correctness. The informal proofs of correctness are straightforward and easy to apply by most practicing engineers. However, the formal proofs of correctness scare even the well seasoned engineers. Much of the knowledge of formal predicate logic required to perform a formal proof of correctness does not exist in many software engineering teams. We are not doing any formal proofs of correctness, unless absolutely necessary. We understand the need to perform such proofs when warranted. In a sample test, we gave a Cleanroom consultant a version of a special fast fourier transform (FFT) that we thought was correct, to verify using formal techniques. Problems were found in the implementation using formal techniques. The problem was, they were found by a consultant with a strong mathematical background. Many software development teams lack the background to perform such formal proofs of correctness. Possible solutions to this are to contract out the small sections of the design that require formal proofs of correctness, or have a person in house that is able and willing to do this. We took the later approach and, to date, have encountered few situations where formal proofs of correctness were required.

Fundamental principles guide the way.

The last barrier mentioned summarizes one of the underlying problems with Cleanroom acceptance; the perception that it is a concept completely different from SA/SD, OO, or other development methodologies. It is not. The most refreshing aspect of Cleanroom is that the model is based, in large part, on fundamental computer science principles. For example, to perform sequence based enumerations using stimulus history requires a knowledge of regular expressions and finite automata. State box design is nothing more than a tabular representation of a finite state machine. Indeed, often we used finite state machine diagrams to verify the state box design. The entire box structured approach supports the fundamental computer science concepts of referential transparency, transaction closure, and reuse. Formal and informal proofs of correctness requires varying levels of knowledge of predicate logic. Usage modeling requires knowledge of Markov processes, grammars, Monte Carlo simulation techniques, hypothesis testing, and statistics. Nothing here is new and these concepts can be applied to any development methodology ones chooses. One of the main reasons we chose Cleanroom for our project is because many of us more experienced engineers

that had gone through at least one software development effort before saw Cleanroom not as a new approach to software development, but rather a more disciplined, complete way of doing the same job! At this point in our project, that continues to be the case.

Static analysis and dynamic analysis.

Cleanroom places a heavy emphasis on static analysis techniques. Static analysis is the systematic inspection and investigation of the textual source of the software product without having to execute the code. The goal is to find problems and gather metrics without having to compile and execute the code. The entry criteria to verification or testing is often a clean compile in many Cleanroom projects. Dynamic analysis investigates the behavior of the software by executing it. Software execution provides information that cannot be obtained by static analysis such as timing behavior and profiles and execution traces. Because of the real time issues we were facing in our project, we have chosen to use static and dynamic analysis. Our reasons for using dynamic analysis techniques were to determine the behavior of COTS software, to determine the real time behavior of the software, to cover the areas that were not covered by usage modeling, and to do additional path and branch coverage. The control software was modeled very effectively with usage modeling. The algorithmic software supplemented usage modeling with unit tests to verify some of the algorithmic properties of the software. The testing paradigm consisted of a layered approach of static analysis, automatic code analysis, unit testing, and statistical testing with usage models (Figure 5). This flexibility in testing illustrates customization and tailoring of the Cleanroom process to meet specific project needs, while maintaining adherence to overall Cleanroom principles.

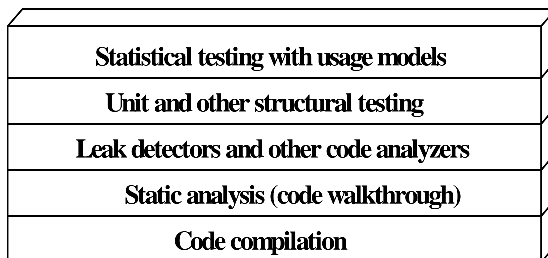

Figure 5. Testing Paradigm for Operational Software

Good specification leads to good testing.

Having a good specification allows testers to develop good test cases. Developing test cases by looking at the code does not insure that the code is doing what it is supposed to be doing. Test cases developed this way only prove that the software does what we think it should do. These types of tests prove nothing (other than the compiler is working correctly). Testing code against a good specification is the correct way to verify software. Not only can errors be discovered in the code, but errors are occasionally, as we discovered, found in the specification as well. In general, better specification allow better testing. With Cleanroom, we felt we were able to develop very rigorous, precise specifications. Our testing was based on these specifications.

Usage models for positive and negative testing.

Usage models enable testing of specified transitions defined in the model. This type of testing is often referred to as “positive” testing because they are designed to verify that the software does what it is supposed to do. On the other hand, “negative” testing is focused on verifying that the software does not do things that it is not supposed to do. We created usage models that allowed us to perform both kinds of testing. Our positive testing was based on usage models of “typical” or “normal” operation. The negative testing was based on usage models of “perverse” or “abnormal” software use.

Testing as a full time job.

One lingering concern with adopting the Cleanroom approach was whether we could convince software engineers who, by education and habit, are used to writing software as their main job, to do testing and certification. Testing as a career has not fared as well in this country as it has in other countries, and people are rewarded mainly based on how well their software writing

skills are, not their testing skills. One solution to this barrier was to institute a round robin type of role playing within some of the teams. For those software programs that had multiple increments, engineers were given the opportunity to be developer on some increments, and certifier on other increments. So far, this has seemed to be a good middle of the road approach. Automation of the testing process will make the job of statistical testing easier (Figure 6).

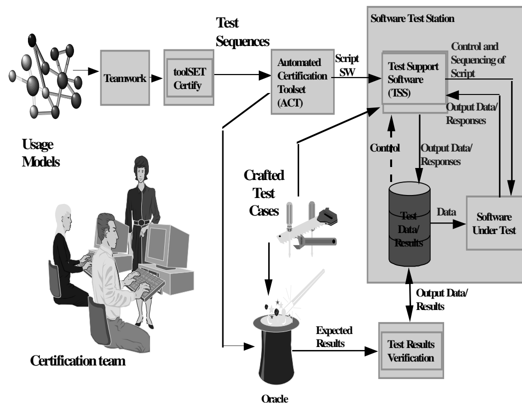

Figure 6. Automation of the Testing Process

Cleanroom seems to address some of the fundamental testing questions that have concerned us in the past:

- Do not let the programmers test their own work. We are using separate certification teams to test software.
- Do not let the testers decide when enough is enough. If statistical tests using Markov modeling techniques are run after all other types of testing, statistics can be used to determine stopping criteria for testing (this is very dependent on the level of abstraction in the usage models and cannot be done blindly).

Still, not everybody likes Cleanroom. That was evident by some of the survey respondents. And there are issues remaining to be worked in some areas.

The formalism and up front design time were the main issues. Most of the more experienced engineers, especially the ones that had at least one tour of system integration and test, however, clearly saw the benefit and need to do a complete, consistent, concise, correct design up front.

We successfully tailored the Cleanroom process to fit within the guidelines of our project in regards to required deliverable documents, schedule, formal reviews, etc. We did not accept the process dogmatically, nor are we being orthodox about the methodology. We have tailored the process based on our judgment and understanding of what we needed to get the job done. Our customer has been totally enamored with our efforts to date using Cleanroom, and views much of what we are doing as a "breadth of fresh air" compared to some of the other work they are reviewing. Reviews by outside, customer hired, consultants verify the same conclusion. We are farther ahead at this point than the customer expected. The customer has continued concerns with testing and we are addressing these issues mainly by adding unit testing in some areas.

One team had to abandon much of the Cleanroom process because of schedule alignment issues. This team had to interface with commercial off the shelf (COTS) software and had a difficult time understanding what the COTS was supposed to do (a case for good documentation). Since this group had to have their product out before any other team (the product was an operating system that was going to be used by several other teams) they were required to have the design completed first. Prototyping was required to determine how the interfaces worked and this prototype code was not developed using Cleanroom. The prototype code effectively became the operational code due to the fact that they spent a great deal of schedule getting the interfaces to work correctly with the COTS software. The team did, however, develop a statistical test approach to verify the software. The statistically generated test cases found many problems in the software. The team lead was very excited about the use of statistical tests to verify his software. The team lead in this area confessed that many of the problems found could have been prevented by doing a complete sequence based specification.

Summary.

Cleanroom has been inserted and used successfully in a mature process driven environment. Many of the principles of the technology are based on fundamental computer

science foundations. As with any other methodology, a certain amount of tailoring is required for the technology to be most effective. We have tailored the use of Cleanroom based on our project needs, our time and schedule constraints, and on how comfortable we felt with the various components of the technology. What we ended up with is a model for software development that we feel is very effective at producing quality software. We particularly enjoyed the concepts of complete, correct, concise, and consistent specifications using a step wise refinement of sequence and state based enumerations. We are having success with our incremental approach to development of software. We see the statistical approach to testing as an excellent addition to some of the existing techniques of software testing and are enthusiastically using these methods.

References

[1] Dyer, Michael; *The Cleanroom Approach to Quality Software Development*, John Wiley and Sons, 1992.

[2] Mills, H.D., M.Dyer, and R.C. Linger, "Cleanroom Software Engineering", *IEEE Software*, September, 1987, pp19-25.

[3] Linger, R.C., "Cleanroom Process Model", *IEEE Software*, March, 1994, pp. 50-58.

[4] Hausler, P.A., Linger, R.C., Trammel, C.J.; "Adopting Cleanroom Software Engineering With a Phased Approach", *IBM Systems Journal*, Volume 33, Number 1, 1994.

[5] Whittaker, J.A. and M.G. Thomason, "A Markov Chain Model for Statistical Software Testing", *IEEE Transactions on Software Engineering*", October 1994, pp. 812-824.

[6] Poore, J.H., H.D. Mills, and D. Mutchler, "Planning and Certifying Software System Reliability", *IEEE Software*, January, 1993, pp 88-99

[7] Kelly, David P. and Robert S. Oshana; "Integrating Cleanroom Software Methods Into an SEI Level 4-5 Program", *Crosstalk*, November 1996.

[8] Gilb, Tom, *Principles of Software Engineering Management*, Reading, MA, Addison-Wesley, 1988.

[9] Linger, R.C., Mills, H.D., Witt, B.I.; *Structured Programming Theory and Practice*, Addison-Wesley, 1979.

1060-3425/98 $10.00 (c) 1998 IEEE