



Quality Software Via a Cleanroom Methodology

by Robert Oshana

Using a cleanroom development process can help improve the quality of your software. And best of all, you don't even need to wear a bunny suit.

What would you do if your boss came to you and said, "I'm sorry, but you won't be able to unit test your software before it's shipped to integration and test. Your testing will be done by another team and you won't be able to compile your program before giving it to the test team. Sorry." Be honest&emdash;would you have confidence in your software enough not to be totally embarrassed?

Believe it or not, this post-partum separation is exactly what is done using a software development process called cleanroom software engineering. Cleanroom is a software development process based on formal mathematical and engineering principles that promises to produce high quality software under statistical quality control. Originally started at IBM Federal Systems Division in the early '80s, cleanroom engineering has been used in projects at IBM, NASA Goddard Space Flight Center, Ericsson, and Texas Instruments.

Cleanroom is the name used in the semiconductor

industry to refer to those rooms where integrated circuits are manufactured and the workers wear those scientific-looking white robes and face masks. This high level of hygiene attempts to prevent any contamination such as dust and other small particles from damaging the product. The concept is that any failure is considered a failure in the process of producing the product, not in the actual product itself. Errors are tracked to process failures. The process is fixed and the failed product is thrown away. Cleanroom in the software world takes a similar approach. A failure in the software is considered a process failure which can be in software specification, design, or verification. The process is fixed and the failed product is discarded, which for software means sending the offending software unit back to the point in the process that caused the failure.

Cleanroom is also based on the concept of six sigma quality. The quality of a software design can be measured in standard distributions or sigmas based on a standard normal distribution like that shown in Figure 1. A six sigma quality level, which allows 3.4 defects per million units, is becoming standard in many companies. In cleanroom, a unit is a software use which can be defined in many ways, depending upon the application. Regardless of how you define a use, six sigma quality is hard to achieve. Cleanroom software engineering uses a mathematically-based, scientific process to approach this level of quality.

Software development today is still, in many ways, a

form of crafting: applying human creativity to design a solution to a technical problem. The processes and practices that we use often come from personal experience. In true engineering, problems are solved using practices derived from an appropriate science base. Cleanroom attempts to replace crafting with mathematically-based, scientific software engineering.

The cleanroom software development process is shown in Figure 2. The rest of this article will explain the basics behind the main components of the cleanroom development process:

- Formal specification and design
- Correctness verification
- Statistical usage-based testing
- Incremental development

Formal Specification and Design

Cleanroom software engineering is based on a set of formal specifications describing the external behavior of the system. A

A strict stepwise refinement and verification process is done using a box-structured approach that allows accurate definition of the required user functions and system object architecture. Stepwise refinement allows a product specification to be divided up into small pieces for implementation. This stepwise approach allows the developer to maintain intellectual control over the product being developed and more easily focus on smaller problems. The approach attempts to increase

productivity by allowing the product to be incrementally developed. We can say, for example, that 50% of the product is 100% complete instead of 100% being 50% complete. Cleanroom implements this approach using an object-based technology of box structures called black, state, and clear boxes.

A black box is a specification of external behavior of the function for all possible circumstances. Inputs to the black box are called stimuli and outputs are responses. The responses from the black box are driven by the stimulus history and the current stimulus. We consider all cases, including unexpected and erroneous stimuli. This inclusion forces the developer to consider all possible cases of external behavior instead of a simple English language description of what is required that can be easily misinterpreted and incomplete. If you have used other development methodologies, you will find that this is not too different. Structured analysis and design uses data flows instead of stimuli and responses and the object-oriented approach (OO) uses messages as shown in Figure 3. A small mindset change is required to start thinking in terms of stimuli/response, but a small one.

The state box is derived from the black box and represents the external behavior plus the data needed to implement that behavior. The final structure is the clear box which shows the external behavior, the data needed to implement that behavior, the procedures required to implement that behavior and, finally, any new lower level black boxes that are required to implement the function.

This can be seen graphically in Figure 4.

Coming from a hardware background, I view the state box in the cleanroom sense, as a finite state machine ROM controller (as shown in Figure 5). When I designed simple embedded controllers, I had to consider the current input plus what state I was currently in, then determine what outputs I wanted to generate and what state to transition to. I had to consider all combinations of input/state in order to be complete and correct in my design. I used an engineering approach to solve the problem. Cleanroom is attempting to use the same engineering approach in the design of software: consider all possible combinations of input and state and ensure mapping to output state and response for every possible condition. This makes the software design complete, correct, and concise.

Box structures of the kind used in cleanroom fixes some of the problems found in object-oriented (OO) approaches. One example is the attempt to encapsulate data in OO. Encapsulating data at the right level of abstraction can be difficult: too high a level degrades modularity and too low a level can lead to redundancies in the data. The cleanroom approach forces the designer to invent only the data needed at that time in the design. This box-structured approach also allows for intellectual control in an object-based development environment and forces the concept of top-down refinement and verification. The aim in doing all this is to achieve the transition from specification to code in small enough

increments so that each step is obviously correct. The key word is obvious.

The logic behind proceeding in small increments (stepwise refinement) is based on the fact that the human intellect can only understand complexities when they are linked in close, simple relationships. Anyone who has gone into a legacy system to make a "simple" change and broke something else knows what I am talking about. Cleanroom, if done correctly, can prevent this by allowing incremental development and verification where each black box can be considered totally independent and fixes or updates can be more isolated.

Correctness Verification

During software design and development, the software development team conducts a rigorous verification activity on the design and code 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 verification checks. For well-specified, well-structured software, proof of software correctness can be done by direct assertion. Intellectual control of the product is thereby assured, allowing developers to completely understand the problem at the right level of abstraction. Correctness verification is done before the

software is ever executed, so that the developers are not permitted to get into a "debugging" mode of operation.

The main point in this activity is that the program functions are looked at as mathematical functions defining a mapping from a range to a domain using a rule. If you look at program control structures such as DO-WHILE and IF-THEN-ELSE, they are actually mathematical rules. Correctness theorems, found in most structured programming text books, define correctness conditions for each control structure. As an example, consider the control structure for a WHILE loop:

```
f
WHILE p
DO
g;
END
```

The correctness condition used for all arguments is the following: "Is termination guaranteed, and whenever p is true, does g followed by f do f, and whenever p is false, does doing nothing do f?"

There is a set of correctness conditions for each of these Dykstra primitives. If each question can be answered positively then correctness is ensured. Some of these correctness questions seem obvious at first, but having

the due respect for each legal primitive can prevent errors such as nontermination failures, possible recursion errors, and so on.

Cleanroom development teams derive program functions, compare them to intended functions, and use these correctness conditions to prove, formally or informally, correctness of the structure. The beauty of all this is that verification is reduced to finite process which has a large impact on quality. Even modest programs can have an infinite number of execution paths. Testing all these paths is impossible. However, the problem can be reduced to a finite number of steps by verifying the finite number of control constructs. In the case where one is trying to verify life critical systems and components, formal written proofs of correctness can be used.

This type of correctness verification can result in a near zero defect level. This process also scales to programs of any size, because all programs have a finite number of control constructs. This method of correctness verification allows developers to deliver software without unit testing and produces better code than the old unit testing way. No unit testing does not mean "do not use the machine." The computer is still a must for experimenting, evaluating complex algorithms, and benchmarking real-time performance.

Statistical Usage Based Testing and Certification

I have never really liked coverage testing. If I find only a

few errors, is that good or bad? If I find a lot of errors, is that good or bad? When do I stop testing? Usually when it's time to ship the product or go to system integration. I always seemed to spend a lot of time finding and fixing errors that seemed pretty trivial.

The cleanroom concept asserts that this type of unit testing should be private to the programmer. Unit testing is actually dangerous because it tends to find many of the superficial defects and lets some of the other more important errors slip through. Cleanroom uses a formal statistical approach to testing that can indicate the quality of the software and stopping criteria as well. This approach differs from the traditional approach, in which testers assume there are errors in the software and set out to find as many as possible, with the faulty assumption that if enough errors are found and fixed, the software quality would improve. Cleanroom operates under the realistic assumption that it is not possible to test quality into a product. Certification (the cleanroom term for testing) is performed to certify the software reliability, not test the software in the classical sense.

The concept of statistical quality control (SQC) has been around for a while in the manufacturing industry. SQC is used when a manufacturing organization has too many items to test exhaustively. A sample is measured against an assumed perfect design. This measurement is then extrapolated to the entire production line to determine a quality number. Some of these manufacturers also test their products in a way that the most abusive customer

would use the product (similar to all those truck commercials). The same thing can be done with software. On the hardware manufacturing side, physical properties of the product are used to drive the quality numbers. In software, we focus on software "uses."

Statistical usage testing involves testing software the way the users use it. This type of testing focuses on external system behavior, not the internals of the software. A set of usage probabilities are defined and test cases are derived randomly, executed, and recorded to compute quality measures.

Statistical usage testing all starts with test planning and the development of a usage model. Planning involves developing a usage specification which defines usage scenarios of the product. These usage scenarios can be both correct and incorrect uses of the product (you can't always expect your customers to use your product the way you do). These usage scenarios are used for test case generation, and testing is done using user profiles. Once the types of users have been identified, data needs to be collected to determine, accurately, how that class of user will use the product. This information can come from legacy systems, prototypes, domain experts, and conversations with the user. The software is represented in a sequence of states and arcs which represent the probability distribution of the model. Figure 6 shows a simple usage model with probabilities assigned to each of the arcs, showing the various states that the software can be in and the probabilities of

transition between each of the states. This model is formally called a Markov model, but I won't get into the specific details of that model here. Suffice it to say, once this type of model has been created, many statistical measures can be computed. Test scripts can also be generated from this model that can be used to directly drive the statistical tests. These scripts and statistical computations can be done manually, or software can be written to do the computations for you.

Cleanroom certification and, specifically, statistical usage based testing, does not measure quality in defects per line of code. Instead, quality is measured in sigma units. This value gives managers, testers, and users an indication of how often a failure is likely to be encountered. For example, if there are 100 defects per thousand lines of code, it makes no difference to the user if she never sees any of them—she still believes she has a quality product. On the other hand, if you field a product with 0.5 defects per thousand lines of code that the user sees all the time, the product is perceived, in her mind, as unreliable.

High rate errors are responsible for over 60% of software failures. Using the cleanroom approach of usage based testing, these errors are found first. The errors that tend to be left behind using the cleanroom method are infrequently encountered by users. Correcting the high failure rate errors has a big impact on mean time to failure (MTTF). Low rate errors have a very small impact on MTTF. The perceived quality of the product from the

user's point of view (is there another point of view that's important?) is higher. Because you are developing a formal model based on usage data, testing metrics can be run before a single line of code is written. For example, which tests to execute can be answered by the usage model. The answer to the question of when to stop testing can be answered by the statistical measures run on the model. If you have faith in the usage data you developed, this type of stopping criteria based on desired reliability and quality can be estimated before testing ever begins. Management loves this. They now have a formal scientific basis for managerial action.

Cleanroom certification can reduce time to market because you are not spending time testing and fixing when you need not be. When you reach the quality level desired, you can ship, even if complete path coverage is not achieved. Keep in mind that this is totally dependent on the usage model. An inaccurate usage model and probabilities result in inaccurate statistics. Statistical usage testing is also more efficient than coverage testing. Re-work also tends to be reduced. More importantly, maintenance teams are not held hostage supporting post release error correction. Keep in mind that if the software was developed poorly to begin with, certification will ultimately tell you that. Do it right the first time.

Statistical testing using usage models can be used on existing systems as well as new systems. The probabilities and usage models can be developed based

on management objectives (nominal usage, user type, risk, and so on).

Finally, how would you like to put a warranty on your software product similar to the one you get when you buy a hardware product? Warranties are actually possible using statistical testing approaches. Cleanroom certification can tell you when testing is complete and the product can be released. Based on the errors found during testing, a statistical prediction can be made that the user will see a defect no more often than once every N uses. If this number turns out to be something like one failure every 500 years, +/- 100 years, the product can be released knowing the product is reliable (instead of when you run out of money or time). Even if you do not release a product with a warranty anytime soon, having certifiable software components makes for a very profound impact on the whole software reuse area. Software components that are candidates for reuse can be archived along with the related usage scenarios and statistical probabilities. These software components can all have formally certified reliability information that can be of great use to other groups looking to reuse these components.

Incremental Development

Using cleanroom methodology, software products are developed in a series of functional increments that sum up to the final deliverable product. These increments represent operational user functions. The most stable requirements are implemented first, with further steps of

requirements resolution performed with each successive increment. The integration of these increments is done top down. This approach allows early assessment of the product quality and gives continuous feedback to management as to the progress of development. When the final increment is integrated, the system is complete.

Incremental development allows developers to implement core parts of the product early, allowing the ability to get feedback from customers while there is still time to make adjustments based on the feedback received. It's not a bad approach for the marketing group either, who can demonstrate core parts of the product early.

Final Comments

The typical software lifecycle is about 40% design, 20% code, and 40% unit testing. The cleanroom lifecycle is 80% design and 20% code—no unit test.

Cleanroom spans the entire software development life cycle. The traditional approach of software design tends to culminate in a steady state error population. This type of crafting often fails to provide the software quality needed in today's society.

Cleanroom software engineering allows errors to be found earlier in the lifecycle which minimizes expensive rework later on and speeds time to market. Designs are simplified, straightforward, and verifiable, resulting in less "spaghetti" code. Quality is achieved by design and verification, not testing. This built in quality lowers the

overall cost of the product, and the designs also tend to be more concise and compact than average—always a good thing for embedded developers. Cleanroom supports prototyping, object orientation and reuse. The technology is platform and language independent. And productivity is high, as measured by companies that have been using the technology—anywhere from 800 lines of code per engineer month and up. The industry average is much lower than this (120-200).

There is nothing inherently new about cleanroom. The methodology has a lot in common with other practices. You do not need to relearn anything. The methodology is based on structured programming and is compatible with reuse. It is a formal, disciplined process, composed of 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. Admittedly, in companies where there is little or no formal process in place, this methodology will require a fundamental change in the way software is developed—from a craft or an art to an engineering discipline with a sound foundation.

Cleanroom can be phased into a corporate process in increments. Organizations have had success by using selected cleanroom capabilities and have added additional capabilities as confidence grew.

Cleanroom can be applied to new systems as well as

existing systems. For example, poor quality sections of software in existing systems can be re-engineered using certain cleanroom techniques such as formal correctness verification.

Software development is the only profession where errors are not only taken for granted, but planned for. In all other engineering disciplines, errors in the final product are not expected or treated normally. Cleanroom take steps towards this same end. It requires a commitment from management and a lot of discipline from the technical development staff. But the benefits of high quality software, developed within budget and schedule, are the rewards of cleanroom methodology.

Robert Oshana is a software team lead in the Systems Group at Texas Instruments. He is working on real-time software development using cleanroom software engineering methodology. Rob can be reached at oshana@ti.com.

References

1. *Dyer, Michael, The Cleanroom Approach to Quality Software Development, John Wiley & Sons, New York, NY, 1992.*
2. *Linger, R.C., Mills, H.D., Witt, B.I., Structured Programming Theory and Practice, Addison-Wesley Publishing Co., Reading, MA, 1979.*
3. *Hausler, P.A., Linger, R.C., Trammel, C.J., "Adopting*

Cleanroom Software Engineering With a Phased Approach," IBM Systems Journal, Vol. 33, No.1, 1994.

4. Linger, Richard C., "Cleanroom Software Engineering: Management Overview," *Software Engineering Institute, April 1995.*

5. *Cleanroom System and Software Engineering Abstract, Software Engineering Technology, April 1993.*

6. Henderson, Johnie, "Why Isn't Cleanroom the Universal Software Development Methodology?," *CrossTalk, May 1995, p. 11.*

7. Linger, Richard C., "Cleanroom Process Model," *IEEE Software, March 1994, pp. 50-58.*

8. Head, Grant E., "Six-Sigma Software Using Cleanroom Software Engineering Techniques," *Hewlett Packard Journal, June 1994, pp. 40-50.*

9. Linger, Richard C. , "Developing Ultra-High-Quality Software with Cleanroom Software Engineering," *Software Engineering Institute, April 1995.*