Six-Sigma Software Using Cleanroom Software Engineering Techniques

Virtually defect-free software can be generated at high productivity levels by applying to software development the same process discipline used in integrated circuit manufacturing.

by Grant E. Head

In the late 1980s, Motorola Inc. instituted its well-known six-sigma program.[1] This program replaced the "Zero Defects" slogan of the early '80s and allowed Motorola to win the first Malcolm Baldridge award for quality in 1988. Since then, many other companies have initiated six-sigma programs.[2]

The six-sigma program is based on the principle that long-term reliability requires a greater design margin (a more robust design) so that the product can endure the stress of use without failing. The measure for determining the robustness of a design is based on the standard deviation, or sigma, found in a standard normal distribution. This measure is called a capability index (C_p), which is defined as the ratio of the maximum allowable design tolerance limits to the traditional ±3-sigma tolerance limits. Thus, for a six-sigma design limit $C_p = 2$.

To illustrate six-sigma capability, consider a manufacturing process in which a thin film of gold must be vapor-deposited on a silicon substrate. Suppose that the target thickness of this film is 250 angstroms and that as little as 220 angstroms or as much as 280 angstroms is satisfactory. If as shown in Fig. 1 the ±30-angstrom design limits correspond to the six-sigma points of the normal distribution, only one chip in a billion will be produced with a film that is either too thin or too thick.

In any practical process, the position of the mean will vary. It is generally assumed that this variation is about ±1.5 sigma. With this shift in the mean a six-sigma design would produce 3.4 parts per million defective. This is considered to be satisfactory and is becoming accepted as a quality standard. Table I lists the defective parts per million (ppm) possible for different sigma values.

At first the six-sigma measure was applied only to hardware reliability and manufacturing processes. It was subsequently recognized that it could also be applied to software quality. A number of software development methodologies have been shown to produce six-sigma quality software. Possibly the methodology that is the easiest to implement and is the most repeatable is a technique called cleanroom software engineering, which was developed at IBM Corporation's Federal Systems Division during the early 1980s.[3] We applied this methodology in a limited way in a typical HP environment and achieved remarkable results.

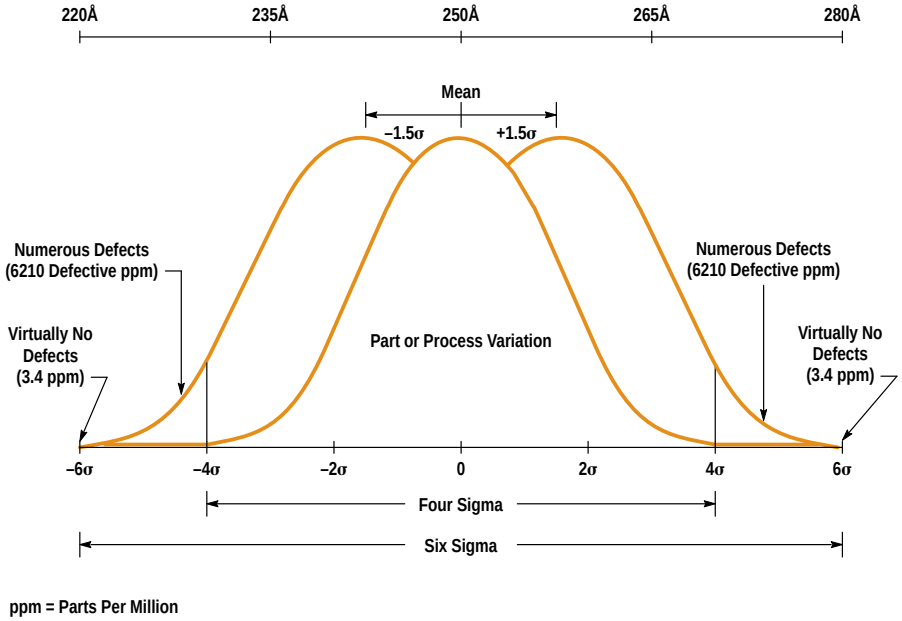

Fig. 1. An illustration of a six-sigma design specification. A design specification of ±30 angstroms corresponds to a ±6-sigma design. Also shown is the ±1.5σ variation from the mean as a result of variations in the process.

Table I
Defective ppm for Different Sigma Values

Sigma	ppm
1	697,700
2	308,733
3	66,803
4	6,210
5	233
6	3.4
7	0.019

When applied to software, the standard unit of measure is called *use,* and six-sigma in this context means fewer than 3.4 failures (deviations from specifications) per million uses. A use is generally defined to be something small such as the single transaction of entering an order or command line. This is admittedly a rather murky definition, but murkiness is not considered to be significant. Six-sigma is a very stringent reliability standard and is difficult to measure. If it is achieved, the user sees virtually no defects at all, and the actual definition of a use then becomes academic.

Cleanroom software engineering has demonstrated the ability to produce software in which the user finds no defects. We have confirmed these results at HP. This paper reports our results and provides a description of the cleanroom process, especially those portions of the process that we used.

Cleanroom Software Engineering

Cleanroom software engineering ("cleanroom") is a metaphor that comes from integrated circuit manufacturing. Large-scale integrated circuits must be manufactured in an environment that is free from dust, flecks of skin, and amoebas, among other things. The processes and environment are carefully controlled and the results are constantly monitored. When defects occur, they are considered to be defects in the processes and not in the product. These defects are characterized to determine the process failure that produced them. The processes are then corrected and rerun. The product is regenerated. The original defective product is not fixed, but discarded.

The cleanroom software engineering philosophy is analogous to the integrated circuit manufacturing cleanroom process. Processes and environments are carefully controlled and are monitored for defects. Any defects found are considered to be defects in one or more of the processes. For example, defects could be in the specification process, the design methodology, or the inspection techniques used. Defects are not considered to be in the source file or the code module that was generated. Each defect is characterized to determine which process failed and how the failure can be prevented. The failing process is corrected and rerun. The original product is discarded. This is why one of the main proponents of cleanroom, Dr. Harlan Mills, suggests that the most important tool for cleanroom is the wastebasket.[4]

Life Cycle

The life cycle of a cleanroom project differs from the traditional life cycle. The traditional 40-20-40 postinvestigation life cycle consists of 40% design, 20% code, and 40% unit testing. The product then goes to integration testing.

Cleanroom uses an 80-20 life cycle (80% for design and 20% for coding). The unexecuted and untested product is then presented to integration testing and is expected to work. If it doesn't work, the defects are examined to determine how the process should be improved. The defective product is then discarded and regenerated using the improved process.

No Unit Testing

Unit testing does not exist in cleanroom. Unit testing is private testing performed by the programmer with the results remaining private to the programmer.

The lack of unit testing in cleanroom is usually met with skepticism or with the notion that something wasn't stated correctly or it was misunderstood. It seems inconceivable that unit testing should not occur. However it is a reality. Cleanroom not only claims that there is no need for unit testing, it also states that unit testing is dangerous. Unit testing tends to uncover superficial defects that are easily found during testing, and it injects "deep" defects that are difficult to expose with any other kind of testing.

A better process is to discover all defects in a public arena such as in integration testing. (Preferably, the original programmer should not be involved in performing the testing.) The same rigorous, disciplined processes would then be applied to the correction of the defects as were applied to the original design and coding of the product.

In practice, defects are almost always encountered in integration testing. That seems to surprise no one. With cleanroom, however, these defects are usually minor and can be fixed with nothing more than an examination of the symptoms and a quick informal check of the code. It is very seldom that sophisticated debuggers are required.

When to Discard the Product

When IBM was asked about the criteria for judging a module worthy of being discarded, they stated that the basic criterion is that if testing reveals more than five defects per thousand lines of code, the module is discarded. This is a low defect density by industry standards,[5] particularly when it is considered that the code in question has never been executed even by an individual programmer. Our experience is that any half-serious attempt to implement cleanroom will easily achieve this. We achieved a defect density of one defect per thousand lines of code the first time we did a cleanroom project. It would appear that this "discard the offending module" policy is primarily intended to be a strong attention getter to achieve commitment to the process. It is seldom necessary to invoke it.

Productivity Is Not Degraded

Productivity is high with cleanroom. A trained cleanroom team can achieve a productivity rate approaching 800 noncomment source statements (NCSS) per engineer month. Industry average is 120 NCSS per engineer month. Most HP entities quote figures higher than this, but seldom do these quotes exceed 800 NCSS.

There is also evidence that the resulting product is significantly more concise and compact than the industry average.[6]

This further enhances productivity. Not only is the product produced at a high statement-per-month rate, but the total number of statements is also smaller.

Needed Best Practices

Cleanroom is compatible with most industry-accepted best practices for software generation. It is not necessary to unlearn anything. Some of these best practices are required (such as a structured design methodology). Others such as software reuse are optional but compatible.

As mentioned above, cleanroom requires some sort of structured design methodology. It has been successfully employed using a number of different design approaches. Most recently however, the cleanroom originators are recommending a form of object-oriented design.[7]

All cleanroom deliverables must be subject to inspections, code walkthroughs, or some other form of rigorous peer review. It is not critical what form is applied. What is critical is that 100% of all deliverables be subjected to this peer-review process and that it be done in small quantities. For instance, it is recommended that no more than three to five pages of a code module be inspected at a single inspection.

Required New Features

In addition to the standard software engineering practices mentioned above, there are a number of cleanroom-specific processes that are required or are recommended. These practices include structured specifications, functional verification, structured data, and statistical testing. Structured specifications are applied to the project before design begins. This strongly affects the delivery schedule and the project management process. Functional verification is applied during design, coding, and inspection processes. Structured data is applied during the design process. Finally, statistical testing is the integration testing methodology of choice. Fig. 2 summarizes the cleanroom processes.

Structured Specifications

Structured specifications[8] is a term applied to the process used to divide a product specification into small pieces for implementation. It is not critical exactly how this division is accomplished as long as the results have the following characteristics:

- Each specification segment must be small enough so that it can be fully implemented by the development team within days or weeks rather than months or years.
- The result of implementing each segment must be a module that can be completely executed and tested on its own. This means that no segment can contain partially implemented features that must be avoided during testing to prevent program failure.
- The segments may not have mutual dependencies. For example, it is satisfactory for segment 4 to require the implementation of segment 3 to execute correctly. It is assumed that segment 3 will be implemented first and will exist to support the testing of segment 4. However, it is not satisfactory for segment 3 to require segment 4 to execute properly at the same time.

The structured specifications process is used by cleanroom to facilitate control of the process by allowing the development team to focus on small, easily conceptualized pieces. A secondary but very important effect is that productivity is increased. Increased productivity is a natural effect of the team's being focused. Each deliverable is small and the time to produce it is psychologically short. The delivery date is therefore always imminent and always seems to be within reach. Morale is generally high because real progress is visible and is achievable.

Structured specifications also offer a very definite project management advantage. They serve to achieve the frequently quoted maxim that when a project is 50% complete, 50% of its features should be 100% complete instead of 100% of its features being 50% complete. Proper management visibility and the ability to control delivery schedules depend upon this maxim's being true.

Structured specifications are very similar to incremental processes described in other methodologies but often the purposes and benefits sound quite different. For instance, in one case the structured specifications process is called *evolutionary delivery.*[9] The primary benefit claimed for evolutionary delivery is that it allows "real" customers to examine early releases and provide feedback so that the product will evolve into something that really satisfies customer needs. HP supports this approach and has classes to teach the evolutionary delivery process to software developers.

From the description just given it would appear that each evolutionary release is placed into the hands of real customers. This implies to many people that the entire release process is repeated on a frequent (monthly) basis. Since multiple releases and the support of multiple versions are considered headaches for product support, this scenario is frowned

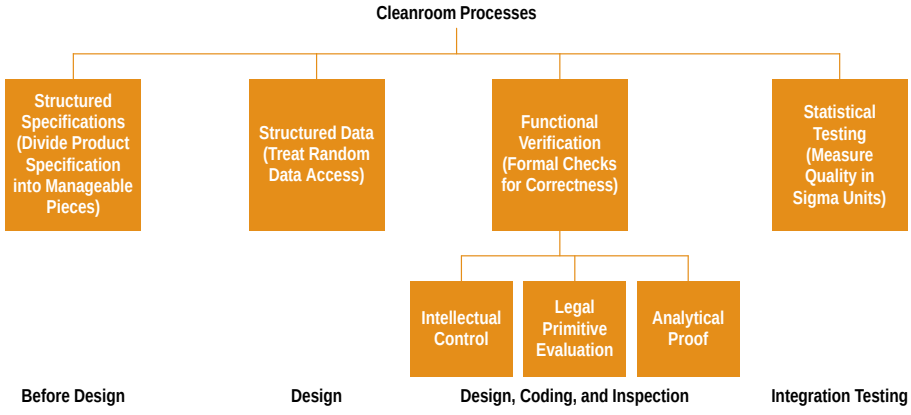

Fig. 2. The processes recommended for software cleanroom engineering.

upon. Cleanroom does not make it a priority to place each stage of the product into the hands of real customers.

Looking at the definition of a real customer in the evolutionary delivery process, you realize that a real customer could be the engineer at the next desk. In practice, the product cannot be delivered to more than a handful of alpha or beta testers until the product is released to the full market. This type of release should not occur any more often than normal. In fact, since cleanroom produces high-quality products, the number of releases required for product repair is significantly reduced.

Another type of structured specifications technique, which is applied to information technology development, uses information engineering time boxes.[10] Time boxes are used as a means of preventing endless feature creep while ensuring that the product (in this case an information product) still has flexibility and adaptability to changing business requirements.

HP has adopted a technique called *short interval scheduling*[11] as a project management approach. Short interval scheduling breaks the entire project into 4-to-6-week chunks, each with its own set of deliverables. Short interval scheduling can be applied to other projects besides those involved in software development. This is an insight that is not obvious in other techniques.

All of these methods are very similar to the structured specifications technique. As different as they sound, they all serve to break the task into bite-sized pieces, which is the goal of the structured specifications portion of cleanroom.

Functional Verification

Functional verification is the heart of cleanroom and is primarily responsible for achieving the dramatic improvement in quality possible with cleanroom. It is based on the tenet that, given the proper circumstances, the human intellect can determine whether or not a piece of logic is correct, and if it is not correct, devise a modification to fix it.* Functional verification has three levels: intellectual control, legal primitive evaluation, and analytical proof.

Intellectual control requires that the progression from specifications to code be done in steps that are small enough and documented well enough so that the correctness of each step is obvious. The working term here is "obvious." The reviewer should be tempted to say, "Of course this refinement level follows correctly from its predecessor! Why belabor the point?" If the reviewer is not tempted to say this, it may be advisable to redesign the refinement level or to document it more completely.

Legal primitive evaluation enhances intellectual control by providing a mathematically derived set of questions for proving and testing the assumptions made in the design specifications. Analytical proof[12] enhances legal primitive evaluation by answering the question sets mathematically. Analytical proof is a very rigorous and tedious correctness proof and is very rarely used.

We have demonstrated here at HP that intellectual control alone is capable of producing code with significantly improved defect densities compared to software developed with other traditional development processes. Application of the complete cleanroom process will provide another two to three orders of magnitude improvement in defect densities and will produce six-sigma code.

Intellectual Control. The human intellect, fallible though it may be, is able to assess correctness when presented with reasonable data in a reasonable format. *Testing is far inferior to the power of the human intellect.* This is the key point. All six-sigma software processes revolve around this point. It is a myth that software must contain defects. This myth is a self-fulfilling prophecy and prevents defect-free software from being routinely presented to the marketplace. The prevalence of the defect myth is the result of another myth, which is that the computer is superior to the human and that computer testing is the best way to ensure reliable software.

We are told that the human intellect can only understand complexities when they are linked together in close, simple relationships. This limitation can be made to work for us. If it is ignored, it works against us and handicaps our creative ability. Making this limitation work for us is the basis of functional verification.

The basis for intellectual control and functional verification is a structured development hierarchy. Most of us are familiar with a representation of a hierarchy like the one modeled in Fig. 3. This could be an illustration of how to progress from design specifications to actual code using any one of

* This tenet is also the definition of intellectual control.

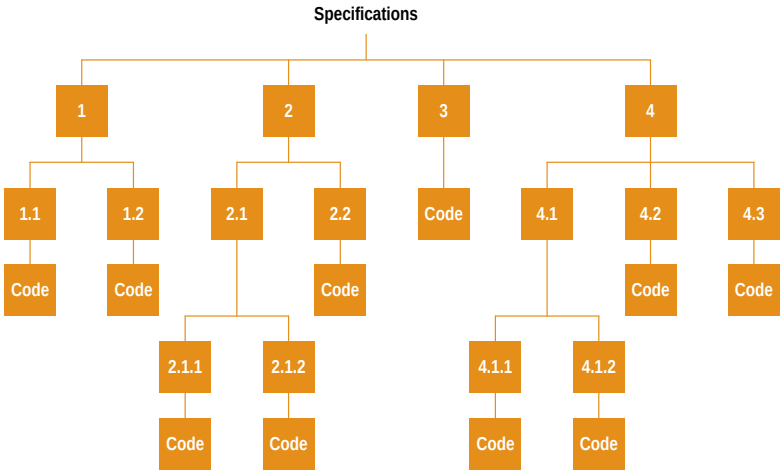

Fig. 3. A typical representation of a hierarchical diagram. In this case the representation is for a software design.

the currently popular, industry-accepted best practices for design. Each of these practices has some form of stepwise refinement. Each breaks down the specifications into ever greater detail. The result is a program containing a set of commands in some programming language.

The difference between the different software design methods is reflected in the interpretation of what the squares and the connecting lines in Fig. 3 represent. If the developer is using structured design techniques, they would mean data and control connections, and if the developer is using object-oriented design, they would represent objects in an object-oriented hierarchy.

Functional verification does not care what these symbols represent. In any of these methods, the squares 1, 2, 3, and 4 are supposed to describe fully the functionality of the specifications at that level. Similarly, 4.1 through 4.3 fully describe the functionality of square 4, and the code of square 3 fully implements the functionality of square 3. Intellectual control can be achieved with any of them by adhering to the following five principles.

Principle 1. Documentation must be complete. The first key principle is that the documentation of the refinement levels must be complete. It must fully reflect the requirements of the abstraction level immediately above it. For instance, it must be possible to locate within the documentation of squares 1 through 4 in our example every feature described in the specifications.

If documentation is complete, intellectual control is nearly automatic. In the case described above, the designer intuitively works to make the documentation and the specifications consistent with each other. The inspectors intuitively study to confirm the correctness.

Note that it is not always necessary to reproduce the specifications word for word. It will often be possible to simply state, "This module fully implements the provisions of specification section 7-4b." The inspectors need only confirm that it makes sense for section 7-4b to be treated in a single module.

Other times it may be necessary to define considerably more than what is in the specifications. A feature that is spread over several modules requires a specific description of which portion is treated in each module and exactly how the modules interact with each other. It must be possible for the inspectors to look at all the modules as a whole and determine that the feature is properly implemented in the full module orchestration.

This principle is commonly violated. All industry-accepted best design processes encourage full documentation, but it is still not done because these design processes often lack the perspective and the respect for intellectual control that is provided by the principles of functional verification, or they are insufficiently compelling to convey this respect. The concept of intellectual control is often lost by many design processes because the main emphasis is on the mechanics of the specific methodology.

The result is that frequently the documentation for the first level of the system specifications is nothing more than the names of the modules (e.g., 1. Data Base Access Module, 2. In-Line Update Module, 3. Initialization Module, 4. User Interface Module). It is left to the inspectors to guess, based on the names, what portions of the specifications were intended to be in which module.

Even when there is an attempt to conform fully to the methodology and provide full documentation, neither the designer nor the inspectors seem to worry about the continuity that is required by functional verification. For example, a feature required in the design specification might show up first in level 4.1.2 or in the code associated with level 4.1.2 without ever having been referenced in levels 4 or 4.1. Sometimes the chosen design methodology does not sufficiently indicate that this is dangerous. Once again, this is the result of a failure to appreciate and respect the concept of intellectual control.

If proper documentation practices are followed, the result of each inspection is confidence that each level fully satisfies the requirements. For example, squares 1 through 4 in Fig. 3 fully implement the top-level specifications. Nothing is left out, deferred, undefined, or added, and no requirements are violated. Similarly, 4.1 through 4.3 fully satisfy the provisions of 4, and 4.1.1 and 4.1.2 fully satisfy 4.1.

With these conditions met, inspections of 4.1 through 4.3 should only require reference to the definition for square 4 to confirm that 4.1 through 4.3 satisfy 4. If 4.2 attempts to implement a feature of the specifications that is not explicitly or implicitly referenced in 4, it is a defect and should be logged as such in the inspection meeting.

Principle 2. A given definition and all of its next-level refinements must be covered in a single inspection session.

This means that a single inspection session must cover square 4 and all of its next-level refinements, 4.1 through 4.3. Altogether, 4.1 through 4.3 should not be more than about five pages of material. More than five pages would indicate that too much refinement was attempted at one time and intellectual control probably cannot be maintained. The offending level should be redone with some of the intended refinement deferred to a lower level.

Principle 3. The full life cycle of any data item must be totally defined at a single refinement level and must be covered in a single inspection.

This is the key principle that allows us to be able to inspect 2.1.1 and 2.1.2 and only be concerned about their reaction with each other and the way they implement 2.1. There is no need to determine, for example, if they interact correctly with 1.1 or 4.3.

This principle is a breakthrough concept and obliterates one of the most troublesome aspects of large-system modification. One seems never to be totally secure making a code modification. There's always the concern that something may be getting broken somewhere else. This fear is an intuitive acknowledgment that intellectual control is not being maintained.

Such "remote breaking" can only occur because of inconsistent data management. Even troublesome problems associated with inappropriate interruptability or bogus recursion are caused by inconsistent data management. Intellectual control requires extreme respect for data management visibility.

This visibility can be maintained by ensuring that each data item is fully defined on a single abstraction level and totally studied in a single inspection session. It should be clear:

- Where and why the data item comes into existence
- What each data item is initialized to and why
- How and where each data item is used and what effects occur as a result of its use
- How and where each data item is updated and to what value
- Where, why, and how each data item is deleted.

Note that careful adherence to this principle contributes significantly to creating an object-oriented result even if that is not the intent of the designer. This principle is also one of the reasons why cleanroom lends itself so well to object-oriented design methodologies.

Once the inspection team is fully satisfied that the data management is consistent and correct, there is no need to be concerned about interactions. For instance, the life cycle for data that is global to the entire module would be fully described and inspected when squares 1, 2, 3, and 4 were inspected. Square 2 then totally defines its own portion of this management and 2.1, 2.2, 2.1.1, and 2.1.2 need only be concerned that they are properly implementing square 2's part of this definition. Squares 1, 3, and 4 can take care of their own portion with no worry about the effects on 2.

Adherence to principle 3 means that it is not necessary to inspect any logic other than that which is presented in the inspection packet. There is no intellectually uncontrolled requirement to execute the entire program mentally to determine whether or not it works.

Principle 4. Updates must conform to the same mechanisms.

Since even the best possible design processes are fallible, it is likely that unanticipated requirements will later be discovered. Functional verification does not preclude this. For instance, it may be discovered that it is necessary to test a global flag in the code for 2.1.1 which in turn must be set in the code for 4.2. This is a common occurrence and the typical response is simply to create the global flag for 2.1.1 and then update 4.2 to set it properly. Bug found. Bug fixed. Everything works fine.

However, we have just destroyed the ability to make subsequent modifications to this mechanism in an intellectually controlled way. Future intellectual control requires that this new interface be retrofitted into the higher abstraction levels. The life cycle of this flag must be fully described at the square 1 through 4 level. In that one document, the square 2 test and the square 4 update must be described, and then the appropriate portions of this definition must be repeated and refined in 2.1, 2.1.1, and 4.2, and of course, all of this should be subject to a full inspection.

Principle 5. Intellectual control must be accompanied by bottom-up thinking.

These principles can lull people into believing that they have intellectual control when, in fact, intellectual control is not possible. Intellectual control is, by its nature, a top-down process and is endangered by a pitfall that threatens all top-down design processes: the tendency to postpone real decisions indefinitely. To avoid this pitfall, the designers must be alert to potential "and-then-a-miracle-happens" situations. Anything that looks suspiciously tricky should be prototyped as soon as possible. All the top-down design discipline in the world will not save a project that depends upon a feature that is beyond the current state of the art. Such a feature may not be recognized until very late in the development cycle if top-down design is allowed to blind the developers to its existence.

The Key Word Is "Obvious." It must be remembered that these five principles are followed for the single purpose of making it obvious to the moderately thorough observer that the design is correct. Practicality must be sufficiently demonstrated, documentation must be sufficiently complete, the design must be tackled in sufficiently small chunks, and data management must be sufficiently clarified. All of these must be so obviously sufficient that the reviewer is tempted to say, "Of course! It's only obvious! Why belabor it?" If this is not the case, a redesign is indicated.

Our experience suggests that the achievement of such a state of obviousness is not a particularly challenging task. It requires care, but, if these principles are well understood, this care is almost automatic.

Legal Primitive Evaluation

Legal primitive evaluation enhances intellectual control by providing a mathematically derived set of questions for each legal Dykstra primitive (e.g., If-Then-Else, While-Do, etc.). For each primitive, the designers and the reviewers ask the set of questions that apply to that primitive and confirm that each question can be answered affirmatively. If this is the case, the correctness of the primitive is ensured.

A rigorous derivation of these questions can be found elsewhere.[13] There is insufficient space here to go through these derivations in detail, but we can illustrate the process and its mathematical basis by using a short, nonrigorous analysis of one of these sets, the While-Do primitive. Questions associated with the other primitives are given on page 47.

The While-Do construct is defined as follows:

S = [While A Do B;]

which means:

S is fully achieved by [While A Do B;].

The symbol S denotes the specification that the primitive is attempting to satisfy, or the function it is attempting to perform. The symbol A is the while test, and B is the while body.

As an example, S could be the specification: "The entry is added to the table." The predicate represented by A would then be an appropriate process to enable the program to perform an iteration and to determine if the operation is complete. B would be the processing required to accomplish the addition to the table. We have chosen to use a While-Do because, presumably, we think it makes sense. We may be intending to accomplish the entry addition by scanning the table sequentially until an appropriate insertion point is found and then splicing the entry into the table at that point. Whether or not this makes sense depends upon the known characteristics of the entry and the table. It also may depend upon the explicit (or implicit) existence of a further part of the specification such as "…within 5 ms."

To investigate whether it has been coded correctly, the following three questions are asked:

1. Is loop termination guaranteed for any argument of S?
2. When A is true, does S equal B followed by S?
3. When A is false, does S equal S?

When the answer to these three questions is yes, the correctness of the While-Do is guaranteed. The people asking these questions should be the designer and the inspectors.

These questions require some explanation.

1. Is loop termination guaranteed for any argument of S?

This means that for any data presented to the function defined by S, will the While-Do always terminate? For instance, in our example, are there any possible instances of the entry or the table for which the While-Do will go into an endless loop because A can never acquire a value of FALSE?

This would appear to be an obvious question. So obvious, that the reader may be tempted to ask why it is even mentioned. However, there is a lack of respect for While-Do termination conditions and many defects occur because of failure to terminate for certain inputs. A proper respect for this question will cause a programmer to take care when using it and will significantly help to avoid nontermination failures.

Respect for this question is justified because it is difficult to prove While-Do termination. In fact, it can be mathematically proven that, for the general case, it is impossible to prove termination.[14] To guarantee the correctness of a While-Do, it is therefore necessary to design simple termination conditions that can be easily verified by inspection. Complicated While-Do tests must be avoided.

2. When A is true, does S equal B followed by S?

This means that, when A is true, can S be achieved by executing B and then presenting the results to S again? This question is not quite so obvious.

Iterative statements are very difficult to prove. To prove the correctness of the while statement, it is desirable to change it to a noniterative form. We change it by invoking S recursively. Thus, the expression:

$$S = [\text{While A Do B};] \quad (1)$$

becomes:

$$S = [\text{If A Then (B; S)};] \quad (2)$$

Expression 2 is no longer an iterative construct and can be more readily proven. Fig. 4 shows the diagrams for these two expressions.

The equivalence of these two statements can be rigorously demonstrated.[15] A nonrigorous feeling for it can be obtained by observing that when A is true in [While A Do B], the B expression is executed once and then you start at the beginning by making the [While A] test again. If [While A Do B] is truly equal to S, then one could imagine that, rather than starting again at the beginning with the [While A] test, you simply start at the beginning of S. That changes the While-Do to a simple If-Then, and the predicate A is tested only once. If it is true, you execute B one time and then execute S to finish the processing.

The typical first reaction to this concept is that we haven't helped at all. The S expression is still iterative and now

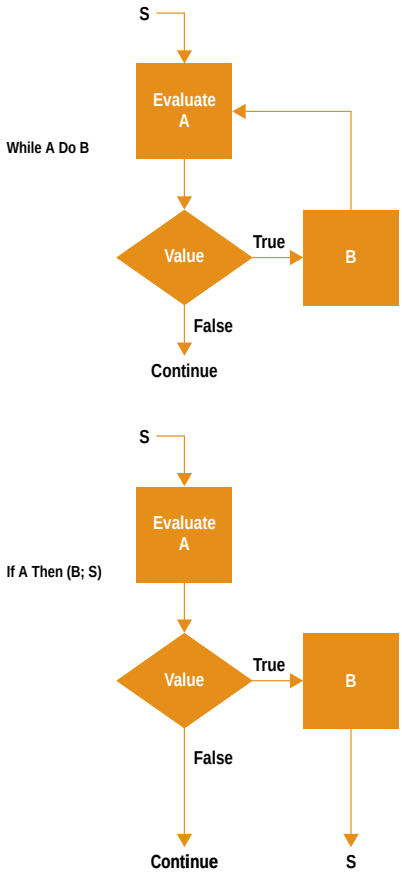

Fig. 4. Diagrams of the primitives While A Do B and If A Then (B; S).

we've made it recursive making it seem that we have more to prove. The response to this complaint is that we don't have to prove anything about S at all. The specification (the entry is added to the table) is neither iterative nor recursive. We have simply chosen to implement it using a While-Do construct. We could, presumably, have implemented it some other way.

S is nothing more than the specification. In the general case, it may be a completely arbitrary statement from any source. Whether the specification is correct or not is not our responsibility. Our responsibility is to implement it as defined.

Question 2 can therefore be restated as follows: If A is true, when we execute B one time and then turn the result over to whatever we've defined S to be, does the result still achieve S? An affirmative answer satisfies question 2.

In terms of our example, B will have examined part of the table. It will either already have inserted the new entry into the table or it will have decided that the portion of the table it examined is not a candidate for inserting of the entry. The unexamined portion of the table is now the new table upon which the construct must execute. This new instance of the table must be comparable to a standalone instance of the table so that the concept of adding an entry to the table still makes sense. If the resulting table fragment no longer looks like any form of the table for which the specification S was generated, question 2 may not be answerable affirmatively and the proposed code would then be incorrect.

3. When A is false, does S equal S?

This question seems fairly obvious but it is frequently overlooked. If A is found to be false the first time the While-Do is executed and therefore no processing of B occurs, is this satisfactory? Does the specification S allow for nothing to happen and therefore for no change to occur as a result of its execution?

In our example, the test posed by this question would likely fail. S requires something to happen (i.e., an entry to be added to the table). This would suggest that the While-Do may not be the appropriate construct for this S. We may never have noticed this fact if we hadn't been forced to examine question 3 carefully.

Structured Data

The principle of structured data[16] recognizes that undisciplined accesses to randomly accessed arrays or accesses that use generalized pointers cause the same kind of "reasoning explosion" produced by the undisciplined use of GOTOs. For instance, take the instruction:

```
A[i] = B[j+k];
```

This statement looks innocent enough. It would appear to be appropriate in any well-structured program. Note, however, that it involves five variables, all of which must be accounted for in any correctness analysis. If the program in which this statement occurs is such that this statement is executed several times, some of these variables may be set in instructions that occur later in the program. Thus, this instruction all by itself creates a reasoning explosion.

Just as Dykstra suggested that GOTOs should not be used at all,[17] the originators of cleanroom suggest that randomly accessed arrays and pointers should not be used. Dykstra recommended a set of primitives to use in place of GOTOs.

In the same way, cleanroom recommends that randomly accessed arrays be replaced with structures such as queues, stacks, and sets. These structures are safer because their access methods are more constrained and disciplined. Many current object-oriented class libraries support these structures directly and take much of the mystery and the complexity out of mentally converting from random-array thinking.

Statistical Testing

Statistical testing[8] is not really required for cleanroom, but it is highly recommended because it allows an assessment of quality in sigma units. It does not measure quality in defects per lines of code. Measuring quality in sigma units gives users visibility of how often a defect is expected to be encountered. For instance, it makes no difference if there are 100 defects per thousand lines of code if the user never actually encounters any of them. The product would be perceived as very reliable. On the other hand, the product may have only one defect in 100,000 lines of code, but if the user encounters this defect every other day, the product is perceived to be very unreliable.

Statistical testing also clearly shows when testing is complete and when the product can safely be released. If the model is predicting that the user will encounter a defect no more often than once every 5000 years with an uncertainty of ±1000 years, it could be decided that it is safe to release the product. This is usually better than some industry-standard methods (e.g., when the attrition rate from boredom among the test team exceeds a certain threshold, it must be time to release, or "When is this product supposed to be released? May 17th. What's today's date? May 17th. Oh. Then we must be finished testing.").

Statistical testing specifies the way test scenarios are developed and executed. Testing is done using scenarios that conform to the expected user profile. A user profile is generated by identifying all states the system can be in (e.g., all screens that could be displayed by the system) and, on each one, identifying all the different actions the user could take and the relative percentage of instances in which each would be taken. As the scenario generator progresses through these states, actions are selected randomly with a weighting that corresponds to the predicted user profile.

For instance, if a given screen has a menu item that is anticipated to be invoked 75% of the time when the user is in that screen, the invocation of this menu item is stipulated in 75% of the generated scenarios involving the screen. If another

Legal Primitive Evaluation

As described in the accompanying article, the process of doing legal primitive evaluation involves asking a set of mathematically derived questions about the of basic programming primitives (e.g., If-Then-Else, For-Do, etc.) used in a program. The following is a list of the questions that must be investigated for each primitive.

In the following list S refers to the specification that must be satisfied by the questions asked about the referenced primitive.

- Sequence S = [A; B;]
  Does S equal A followed by B?
- For-Do S = [For A Do B;]
  - Does S equal first B followed by second B … followed by last B?
- If-Then S = [If A Then B;]
  - If A is true, does S = B?
  - If A is false, does S = S?
- If-Then-Else S = [If A Then B Else C;]
  - If A is true, does S equal B?
  - If A is false, does S equal C?
- Case S = [Case P part (C1) B1 … part (Cn) Bn Else E;]
  - When p is C1, does S equal B1?

    .
    .
    .

    When p is Cn, does S equal Bn?
    When p is not a member of set (C1, …, Cn), does S equal E?
- While-Do S = [While A Do B;]
  - Is loop termination guaranteed for any argument of S?
  - When A is true, does S equal B followed by S?
  - When A is false, does S equal S?
- Do-Until S = [Do A Until B;]
  - Is loop termination guaranteed for any argument of S?
  - When B is false, does S equal A followed by S?
  - When B is true, does S equal A?
- Do-While-Do S = [Do_1 A While B Do_2 C;]
  - Is loop termination guaranteed for any argument of S?
  - When B is true, does S equal A followed by C followed by S?
  - When B is false, does S equal A?

menu item will only be invoked 1% of the time, it would be called in only 1% of the scenarios.

These scenarios are then executed and the error history is evaluated according to a mathematical model designed to predict how many more defects the user would encounter in a given period of time or in a given number of uses. There are several different models described in the literature.[18]

In general, statistical testing takes less time than traditional testing. As soon as the model predicts a quality level corresponding to a predefined goal (e.g., six sigma) with a sufficiently small range of uncertainty (also predefined), the product can be safely released. This is the case even when 100% testing coverage is not done, or when 100% of the pathways are not executed.

Statistical testing requires that the software to which it is applied be minimally reliable. If an attempt is made to apply it to software that has an industry-typical defect density, any of the statistical models will demonstrate instabilities and usually blow up. When they don't blow up, their predictions are so unfavorable that a decision is usually made to ignore them. This is an analytical reflection of the fact that you can't test quality into a program.

Quality Cannot Be Tested into a Product

Although it is the quality strategy chosen for many products, it is not possible to test quality into a product. DeMarco[19] has an excellent analysis that demonstrates the validity of this premise. This analysis is based on the apparent fact that only about half of all defects can be eliminated by testing, but that this factor of two is swamped by the variability of the software packages on the market. The difference in defect density between the best and worst products is a factor of almost 4000.* Of course, these are the extremes. The factor difference between the 25th percentile and the 75th percentile is about 30 according to DeMarco. No one suggests that testing should not be done—it eliminates extremely noxious defects which are easy to test for—but compared to the variability of software packages, the factor of two is almost irrelevant. What then are the factors that produce quality software?

Capers Jones[20] suggests that inspections alone can produce a 60% elimination of defects, and when testing is added, 85% of defects are eliminated. There is no reported study, but the literature would suggest that inspections coupled with functional verification would eliminate more than 90% of defects.[21] Remarkably enough, testing seems to eliminate most (virtually all) of the remaining defects. The literature typically reports that no further defects are found after the original test cycle is complete and that none are found in the field.[21] This was also our experience.

There is apparently a synergism between functional verification and testing. Functional verification eliminates defects that are difficult to detect with testing. The defects that are left after application of inspections and functional verification are generally those that are easy to test for. The result is that >99% of all defects are eliminated via the combination of inspections, functional verification, and testing. Table II summarizes the percentage of defect removal with the application of individual or combinations of different defect detection strategies.

Table II
Defect Removal Percentages
Based on Defect Detection Strategies

Detection Strategy	% Defect Removal
Testing	50%
Inspections	60%
Inspections + Testing	85%
Inspections + Functional Verification	90%
Inspections + Functional Verification + Testing	>99%

Our Experience

We applied cleanroom to three projects, although only one of them actually made it to the marketplace. The project that made it to market had cleanroom applied all the way through its life cycle. The other projects were canceled for nontechnical reasons, but cleanroom was applied as long as they existed. The completed project, which consisted of a relatively small amount of code (3.5 KNCSS), was released as part of a large Microsoft® Windows system. The project team for this effort consisted of five software engineers.

All the techniques described in this paper except structured data and statistical testing were applied to the projects. All the products were Microsoft Windows applications written in C or C++. Structured data was not addressed because we never came across a serious need for random arrays or pointers. Although statistical testing was not applied, it was our intent eventually to do so, but the total lack of defects demotivated us from pursuing a complicated, analytical testing mode particularly when our testing resources were in high demand from the organization to help other portions of the system prepare for product release.

Design Methodology. We applied the rigorous object-oriented methodology known as box notation.[7] This is the methodology recommended by the cleanroom originators. We found it to be satisfyingly rigorous and disciplined.

Box notation is a methodology that progresses from functional specification to detailed design through a series of steps represented as boxes with varying transparency. The first box is a black box signifying that all external aspects of the system or module are known but none of the internal implementation is known. This is the ultimate object. It is defined by noting all the stimuli applied to the box by the user and the observable responses produced by these stimuli.

Inevitably, these responses are a function not only of the stimulus, but also of the stimulus history. For example, a mouse click at location 100,200 on the screen will produce a response that depends upon the behavior of the window that currently includes the location 100,200. The window at that location is, in turn, a function of all the previous mouse clicks and other inputs to the system.

* This factor is based on a defect density of 60 defects per KNCSS for the worst products and 0.016 defects per KNCSS for the best products. The factor difference between these two extremes is 60/0.016 = 3750 or ~4000.

The black box is then converted to a state box in which the stimulus history producing the responses of the black box is captured in the form of states that the box passes through. The response produced by a given stimulus can be determined not necessarily from the analysis of a potentially infinite stimulus history, but more simply by noting the state the system is in and the response produced by that stimulus within that state. States are captured as values within a set of state data. The state box fully reveals this data. It contains an internal black box that takes as its input the stimulus and the current set of state data and produces the desired response and a new set of state data. The state data is fully revealed but the internal black box still hides its own internal processing.

The state box is then converted to a clear box in which all processing is visible. However, this processing is represented as a series of interacting black boxes in which the interactions and the relations are clearly visible but, once again, the black boxes hide their own internal processing. This clear box is the final implementation of the object. In this object, the encapsulated data and the methods to process it are clearly visible.

Each of these internal black boxes is then treated similarly in a stepwise refinement process that ends only when all the internal black boxes can be expressed as single commands of the destination language.

This process allows many of the pitfalls of object-oriented design and programming to be avoided by carefully illuminating them at the proper time. For instance, the optimum data encapsulation level is more easily determined because the designer is forced to consider it at a level where perspective is the clearest. Data encapsulation at too high a level degrades modularity and defeats "object orientedness," but data encapsulation at too low a level produces redundancies and multiple copies of the same data with the associated possibility of update error and loss of integrity. These pitfalls are more easily avoided because the designer is forced to think about the question at exactly that point in the design when the view of the system is optimum for such a consideration.

Inspections. We employed a slightly adapted version of the HP-recommended inspection method taught by Tom Gilb.[9] We found this method very satisfactory. Our minimal adaptation was to allow slightly more discussion during the logging meeting than Gilb recommends. We felt that this was needed to accommodate functional verification.

Functional Verification. No attempt was made to implement anything but the first level of functional verification—intellectual control. This was found to be easily implemented, and when the principles were adequately adhered to, was almost automatic. Inspectors who knew nothing about functional verification or intellectual control automatically accomplished it when given material that conformed to its principles and, amusingly, they also automatically complained when slight deviations from these principles occurred.

Structured Specifications. The project team called cleanroom's structured specifications process evolutionary delivery because of its similarity to the evolutionary delivery methodology mentioned earlier and because evolutionary delivery is more like our HP environment. Structured specifications were developed in a defense-industry environment where dynamic specifications are frowned upon and where adaptability is not a virtue. However, evolutionary delivery assumes a dynamic environment and encourages adaptability. Regardless of the differences, both philosophies are similar.

At first, both marketing and management were skeptical. They were not reassured by the idea that a large amount of time would elapse before the product would take shape because of the large up-front design investment and because some features would not be addressed at all until very late in the development cycle. They were told not to expect an early prototype within the first few days that would demonstrate the major features.

Very quickly, these doubts were dispelled. Marketing was brought into the effort during the early rigorous design stages to provide guidance and direction. They participated in the specification structuring and set priorities and desired schedules for the releases. They caught on to the idea of getting the "juiciest parts" first and found that they were getting real code very quickly and could have this real code reviewed by real users while there was still time to allow the users' feedback to influence design decisions. They also became enthusiastic about participating in the inspections during the top-level definitions.

Management realized that the evolutionary staged releases were coming regularly enough and quickly enough that they could predict very early in the development cycle which stage had a high possibility of being finished in time to hit the optimum release window. They could then adjust scope and priority to ensure that the release date could be reliably achieved.

Morale. The cleanroom literature claims that cleanroom teams have a very high morale and satisfaction level. This is attributed to the fact that they have finally been given the tools necessary to achieve the kind of quality job that everyone wants to do. Our own experience was that this occurred surprisingly quickly. People with remarkably disparate, scarcely compatible personalities not only worked well together, they became enthusiastic about the process.

It appears that the following factors were influential in producing high morale:

- Almost daily inspections created an environment in which each person on the team took turns being in the "hot seat." People quickly developed an understanding that reasonable criticism was both acceptable and beneficial. The resulting frankness and openness were perceived by all to be remarkably refreshing and exhilarating.
- Team members were surprised that they were being allowed to do what they were doing. They were allowed to take the time necessary to do the kind of job they felt was proper.

Productivity. Productivity was difficult to measure. Only one project actually made it to the market place, and it is difficult to divide the instruction count accurately among the engineers that contributed to it. However, the subjective impression was that it certainly didn't take any longer. When no defects are found one suddenly discovers that the job is finished. At first this is disconcerting and anticlimactic, but it also emphasizes the savings that can be realized at the end of the project. This compensates for the extra effort at the beginning of the project.

Conclusion

The cleanroom team mentioned in this paper no longer exists as a single organization. However, portions of cleanroom are still being practiced in certain organizations within Hewlett-Packard. These portions especially include structured specifications and intellectual control.

We believe our efforts can be duplicated in any software organization. There was nothing unique about our situation. We achieved remarkable results with less than total dogmatic dedication to the methodology.

The product that made it to market was designed using functional decomposition. Even though functional decomposition is minimally rigorous and disciplined, we found the results completely satisfactory. The project consisted of enhancing a 2-KNCSS module to 3.5 KNCSS.

The original module was reverse engineered to generate the functional decomposition document that became the basis for the design. The completed module was subjected to the intellectual control processes and the reviewers were never told which code was the original and which was modified or new code. A total of 36 defects were found during the inspection process for a total of 10 defects per KNCSS. An additional five defects were found the first week of testing (1.4 defects per KNCSS). No defects were encountered in the subsequent 10 months of full system integration testing and none have been found since the system was released.

It was interesting to note that the defects found during inspections included items such as a design problem which would have, under rare conditions, mixed incompatible file versions in the same object, a piece of data that if it had been accessed would have produced a rare, nonrepeatable crash, and a number of cases in which resources were not being released which would, after a long period of time, have caused the Windows system to halt. Most of these defects would have been very difficult to find by testing.

Defects found during testing were primarily simple screen appearance problems which were readily visible and easily characterized and eliminated. These results conform well to expected cleanroom results. About 90% of the defects were eliminated by inspections with functional verification. About 10% more were eliminated via testing. No other defects were ever encountered in subsequent full-system integration testing or by customers in the field. It can be expected on the basis of other cleanroom results reported in the literature that at least 99% of all defects in this module were eliminated in this way and that the final product probably contains no more than 0.1 defect per KNCSS.

References

1. M.J. Harry, *The Nature of Six Sigma Quality,* Motorola Government Electronics Group, 1987.
2. P.A. Tobias, "A Six Sigma Program Implementation," *Proceedings of the IEEE 1991 Custom Integrated Circuits Conference,* p. 29.1.1.
3. H.D. Mills, M. Dyer, and R.C. Linger, "Cleanroom Software Engineering," *IEEE Software,* Vol. 4, no. 5, September 1987, pp. 19-25.
4. H.D. Mills and J.H. Poore, "Bringing Software Under Statistical Quality Control," *Quality Progress,* Vol. 21, no. 11, November 1988, pp. 52-55.
5. T. DeMarco, *Controlling Software Projects,* Yourdon Press, 1982, pp. 195-267.
6. R.C. Linger and H.D. Mills, "A Case Study in Software Engineering," *Proceedings COMPSAC 88,* p. 14.
7. H.D. Mills, R.C. Linger, and A.R. Hevner, *Principles of Information Systems Analysis and Design,* Academic Press, 1986.
8. P.A. Currit, M. Dyer, and H.D. Mills, "Certifying the Reliability of Software," *IEEE Transactions on Software Engineering,* Vol. SE-12, no. 1, January 1986, pp 3-11.
9. T. Gilb, *The Principles of Software Engineering Management,* Addison-Wesley, 1988, pp. 83-114.
10. J. Martin, *Information Engineering Book III,* Prentice Hall, 1990, pp. 169-196.
11. *Navigator Systems Series Overview Monograph,* Ernst & Young International, Ltd., 1991, pp. 55-56.
12. R.C. Linger, H.D. Mills, and B.I. Witt, *Structured Programming: Theory and Practice,* Addison-Wesley, 1979, pp. 227-229.
13. *Ibid,* pp. 213-300.
14. H.D. Mills, et al, *Principles of Computer Programming,* Allyn and Bacon, Inc., 1987, pp. 236-238.
15. R.C. Linger, H.D. Mills, and B.I. Witt, *op cit,* pp. 219-221.
16. H.D. Mills and R.C. Linger, "Data Structured Programming: Program Design without Arrays and Pointers," *IEEE Transactions on Software Engineering,* Vol. SE-12, no. 2, Feb. 1986, pp. 192-197.
17. E.W. Dijkstra, "Structured Programming," *Software Engineering Techniques,* NATO Science Committee, 1969, pp. 88-93.
18. P.A. Currit, M. Dyer, and H.D. Mills, *op cit,* pp. 3-11.
19. T. DeMarco, *op cit,* p. 216.
20. Unpublished presentation given at the 1988 HP Software Engineering Productivity Conference.
21. R.C. Linger and H.D. Mills, *op cit,* pp. 8-16.