Cleanroom Review Techniques for Application Development

Michael Deck
Cleanroom Software Engineering, Inc.

voice: (303) 494-3152
fax: (303) 494-3026
internet: deckm@cleansoft.com
FTP: ftp.csn.net/cleansoft
WWW: http://www.csn.net/~deckm

Keywords: Cleanroom, Quality Improvement, Process Improvement, Verification and Validation, Software Inspections

Abstract: Correctness verification in team review is a key factor in the success of Cleanroom approach in improving quality and reducing costs. This paper describes specific, practical techniques that have been used in applying Cleanroom verification to application development (AD) projects. Two areas are given special focus: the mechanics of a team-based iterative review process, including techniques for specification improvement; and techniques for combining verification with inspection, especially assessing potential risks involved when less than complete formal verification is done. This paper can serve as a guide for application development projects that exhibit characteristics such as: volatile specifications, short delivery schedules, limited resources for methodology training, and reliability requirements that are less than safety-critical.
1. Introduction

Cleanroom software engineering combines defect prevention with statistical quality control to achieve high quality software and reduce software development costs. In Cleanroom, software designers produce implementations that are nearly defect free even before testing begins, through team reviews that combine inspection with formal correctness verification. Software testing emphasizes the use of reliability modeling based on operational profiles to measure the quality and reliability of the implementations.

These principles—high quality designs without testing; testing to measure quality—are embedded in an incremental development framework that encourages continuous process improvement. The resulting process is very flexible and can be tailored to meet the needs of many kinds of development teams.

Although the specifics vary from team to team, the Cleanroom process has several key practices that could be considered “typical” usage. We will look at those very briefly after presenting a survey of the Cleanroom literature.

This paper will focus on the techniques that Cleanroom design teams apply during team reviews. While these techniques apply in similar ways to all kinds of systems, we will closely examine a “level of practice” that is used in application development. For reasons of space, details about the mathematical foundations for these review techniques are omitted. These details can be found in a variety of sources, including Linger, et al. [19], Mills [28], Shankar [29], and Deck [1].

1.1 The Cleanroom Literature

There are a number of general works that describe the Cleanroom approach. The Cleanroom idea originated with the late Dr. Harlan Mills and many of the published papers on the topic are due to him and his colleagues at IBM and at Software Engineering Technology [2,3,4,5,6,7]. For the most part, these papers show a maturation and evolution in the way Cleanroom is defined, as new and better techniques were discovered. In particular, that evolution takes Cleanroom from being a narrow, strict approach based heavily on mathematical foundations to a broader, more general approach that can be applied in a variety of ways.

Many development projects have applied all or parts of Cleanroom and some of these have published their results. Because Cleanroom is mostly found in industry, rather than in academia, these results have more of a character of “case studies” rather than double- or triple-blind experiments [8,9,10,11,12].

1.2 Cleanroom Practices

The Cleanroom practices are usually grouped into three areas: management, development, and testing. A team that is new to Cleanroom will frequently choose selected practices to start with, then add practices as they gain experience.

Cleanroom is neither “all new”, nor is it a replacement for the entire body of software engineering practice. Rather, its philosophy and practices are integrated with other techniques appropriate to a particular project. In particular, Cleanroom mandates no specific tools or languages, so it can easily be integrated with existing project knowledge. Thus, while there is a close synergy between Cleanroom and Object-Orientation, for example [13,14,15], the decision to use OO techniques can be made independently of the choice of Cleanroom practices.

Cleanroom management has two key practices. First, an incremental life cycle model [16,17] is preferred. There are many reasons for this preference. It provides an opportunity for continuous process improvement. It also replaces most “clear box” testing with “black box” testing. The incremental model is similar to evolutionary prototyping [18].
The second management practice is team ownership of work products. All work products are reviewed by an owning team (usually 5±2 people, but sometimes larger) with the goal of consensus. Team ownership has many benefits, including having more contributors of good ideas and having backup knowledge.

Cleanroom development is based on a hierarchy of documented abstractions [19]. Over time, a framework for this hierarchy has evolved, called box structures, which will be treated in more detail later. Designs are reviewed by the team for correctness—whether they meet specifications. The reviews also look at design qualities such as simplicity, reusability, maintainability, and performance. An examination of team review techniques will be the focus of this paper.

Cleanroom testing is based on statistical quality control. The input domain is sampled using a usage distribution [20] (also known as an operational profile [21]) and test cases are constructed to meet that sample. Then, the results of testing are entered into a reliability model and an estimate of reliability is produced. The “Certification” model of Currit et al. [20] is frequently used, as is a model based on Markov chains[22,23].

All of these practices are phased-in and tailored to meet the needs of a specific project or team [24].

2. The Application Development Environment

The early use of Cleanroom practices was found primarily in three industry sectors: space and defense [10, 25], telecommunications [11,26], system applications [e.g., 9] and system software [27]. These sectors have traditionally had characteristics that made them natural for early adopters of Cleanroom, including one or more of the following:

- relatively complete and stable requirements and specifications; customers who are closely involved in the requirements process and are aware of the costs of requirements change
- relatively long cycle times, usually measured in years rather than months or weeks
- relatively high reliability requirements, for which the customers were willing to pay extra
- relatively stable development platforms, languages, and environments; and developers who have experience with them
- relatively little reuse of prior higher-level source routines

These characteristics suited the early Cleanroom style perfectly. The stability of requirements essentially made “quality” and “correctness” equivalent, long cycle times permitted greater front-loading of the process (and also gave teams the time to learn a new approach), and Cleanroom offered both high reliability and reliability measurement.

However, many in the application development (AD) community read the early papers about Cleanroom (and looked at the list of projects reporting results) and decided that Cleanroom was not well suited to the AD environment. In addition to having unstable or unknown requirements and short cycle times, application developers are rarely faced with mission- or life-critical reliability requirements and ca not afford to reach for the associated levels of reliability. Application developers are frequently required to use languages, environments, and platforms that are new, unstable, or poorly documented, making experimentation a necessity. They often build on top of existing systems or architectures and make increasing use of framework libraries (both source and object). Furthermore, the mathematics needed in order to work out complex correctness proofs may be beyond the training of most application developers.

The solution has been to separate the vital, defining characteristics of Cleanroom from those parts that can be left to the project’s discretion. Cleanroom is now defined primarily in terms of its principles and basic practices and projects choose techniques that fit their needs. Thus, the “Cleanroom profile” of a telecommunications switch development effort might include:
• team-based, incremental development
• history-based box structures notations,
• rigorous proofs of correctness,
• design languages (e.g., PDL), and
• strict statistical usage-based testing.

The profile of an internal MIS application development project is more likely to include:

• team-based, incremental development
• natural language specifications with occasional formalism,
• implementation languages (e.g., Ada, C, C++; Visual Basic),
• verification-based inspections combined with informal inspections, and
• judicious combination of usage-based testing with other formal testing methods.

The focus of this paper will be on showing how AD projects can use Cleanroom practices that make the most sense in their environment, how they can adapt and tailor the Cleanroom techniques, and how they can access other parts of Cleanroom practice when necessary.

3. Box Structures: A Formal Model for Cleanroom

As Cleanroom evolved, so did the formal models used in Cleanroom development. The earliest Cleanroom projects used the functional model [28,19]. However, the functional model deals with the correctness of procedures; it does not have specific techniques for dealing with data. Cleanroom teams used the method of state machines [29,30] where data verification was important. An additional element was added, and an overall structure proposed, by Mills, et al. [31,32,33], with the result called “box structures.” Since the resulting framework (if liberally interpreted) incorporates both functional verification and state machines as special cases, it is not incorrect to say that box structures has always been the foundation for Cleanroom.

The box structures method provides the software developer with three different views of a software system, object, or part. (We will used the phrase “object” generically to include software entities of any size.) The black box view hides all of the object’s implementation details, including any data implementation or processing. The state box view partially exposes the data implementation while continuing to hide procedurality. The clear box view partially exposes procedurality. Each of these last two views may include references to new black boxes, defining a usage hierarchy (sometimes called a “uses” hierarchy [34] or containment hierarchy). The usage hierarchy is orthogonal to the inheritance hierarchy of object-orientation.

3.1 Specification: The Black Box

We will separately consider two kinds of black box specifications: process specifications and data specifications. A process specification is the black box of a software entity that does not retain data – it does not behave differently at different times depending on prior usage. Process specifications would be used to specify procedures, fragments of inline code, or batch applications. A data specification, on the other hand, is used to represent an object that does encapsulate data, such as a file, database, or object (in the object-oriented sense).

3.1.1 Process Specifications

A process is a definition for how states are transformed by the computer. In the functional model, a process is defined by a mapping from “before” states of the system into “after” states. We will call this mapping a process abstraction or process specification.
There are several aspects of these specifications that are important to Cleanroom, independent of notation. Conceptually, at least:

- the evaluation of all the “before” values is done before the process begins execution
- assignment of “after” values is done all-at-once with no intermediate steps
- only the “before” values can be used to compute the output and there are no undocumented “global” data objects
- there are no undocumented side effects – any data object whose value is changed must be described
- a response must be produced for one input before the next input can be accepted

If these rules are followed, the process specification hides the order of operations and all procedurality, as well as local data of the algorithm and is completely independent of its usage context.

The specification of a complex process such as a batch compiler is necessarily long and detailed, but can be done using this same idea. The inputs would be the contents of source files, options, and command-line parameters. Outputs would include the contents of object files, listings, and screen messages. Environmental structures such as memory and other resources can be treated as both input and output.

Graphics are often used in teaching box structures and in informal discussions among developers. Their usage is somewhat more rare in large-scale development efforts. An example of a named process specification is shown in Figure 1. Throughout this paper, a box with a “clear” top represents a process specification.

![Figure 1. Graphic of Process Specification](image)

### 3.1.2 Data Specifications

In the early work on specifications [19] all of the data objects were “pure mathematical” integer data objects. However, we know that most programs use more complicated data objects. One of the essential characteristics of any data object is the mental picture we have of what is “inside” it. We can use this mental picture, or abstract model, to understand and manipulate the object. This abstract model permits us to discuss the value of the object without knowing its internals. An abstract model defines a new data object in terms of more-primitive or better-understood concepts.

The black box of a data-retaining object, to which we will refer as a data specification, consists of an abstract data model and a collection of process specifications that can be used to manipulate it. These process abstractions are sometimes called services, transitions, or methods. The definition of an abstract data model and all of the operations on it constitutes a data type definition, from which we can instantiate data objects.

The standard box structures literature takes this a step farther and suggests a view of the abstract model that consists of only one entity -- the history of all inputs received by the object. This entity, called stimulus history, has its practical advantages and disadvantages [15].

When writing simple data specifications, we will often use a tabular form that describes each method’s use of, and effect on, the abstract state. Such a table is shown in Figure 2.
<table>
<thead>
<tr>
<th>Stimulus</th>
<th>Condition</th>
<th>Response</th>
<th>Model Update</th>
</tr>
</thead>
<tbody>
<tr>
<td>check out a car to renter</td>
<td>cars available and renter is qualified</td>
<td>“OK”, car number</td>
<td>pool := pool less car selected</td>
</tr>
<tr>
<td></td>
<td>no cars available and renter is qualified</td>
<td>“Sorry, no cars”</td>
<td>no change</td>
</tr>
<tr>
<td></td>
<td>cars available and renter not qualified</td>
<td>“Sorry, not qualified”</td>
<td>no change</td>
</tr>
</tbody>
</table>

Figure 2. Tabular Form of Data Specification

As with the process specification, the notation (English or mathematical, tabular, textual, or graphical.) matters less than the key abstraction directions, which are:

- the condition is evaluated all at once and before any processing of the stimulus begins
- only the stimulus and the prior value of the model can be used to compute the response and the new model value
- the response and model update are produced all-at-once without intermediate steps or data
- no undocumented global variables or side effects
- a response must be produced for one stimulus before the next stimulus can be accepted

The last of these conditions causes the most difficulty for specifying multi-tasking and real-time systems using this approach. For example, a re-entrant code unit may be conceptually in the process of handling several different messages at once. Or, a code unit might send itself a message for recursive processing of a particular stimulus. Different Cleanroom teams have adopted different model changes to accommodate these needs. One possible change is to consider each executing instance to be a separate unit that happens to share data with all other executing units. Another is to expose some of the processing inside the box, in order to have more granularity for discussion.

As with process specifications, we have a graphic representation for data specifications (shown in this paper with a black top and with the model, e.g. “pool”, identified). An example is shown in Figure 3.

Figure 3. Graphic of Data Specification

3.1.3 Data vs. Process -- A Decision

Looking at the similarities between process and data specifications, we can see that it is a small step to convert a data specification into a process specification -- merely move the abstract model “outside the box” and treat it as an input and an output. Although in most cases it will be obvious whether an internal software entity is data-encapsulating or not, at higher conceptual levels the specifier will have to choose whether to consider the entity to be data or process. Very often the entity is not inherently one or the other and the choice will have to be made based on understandability of the specification.
3.1.4 Black Box Validation

A treatment of black box validation—evaluating whether, if implemented, the specified system would meet user requirements—is beyond the scope of this paper. However, it is a significant task, since the techniques of verification (to be discussed next) do not apply. Verification is the activity that evaluates the correctness of one mathematical object (a design) in terms of another (a specification). However, since the user’s requirements are rarely stated in mathematical terms, validation must use other techniques. These techniques commonly include prototyping, storyboards and mockups, focus groups, joint application development, and similar solutions.

3.2 Design: A Hierarchy of Specifications

As the specification nears completion, the design activity begins. An object being specified as a process abstraction is designed first as a clear box; a data abstraction is designed first as a state box. Each design reveals new black boxes that may require further design. Because the black box specification ideas can be applied to both high-level (large) objects and low-level (small) objects, this hierarchy of specifications is self-similar throughout the development process.

3.2.1 Data Design: The State Box

The state box expresses the abstract model of a data-encapsulating system in a more concrete form, in terms of objects that are either simpler to implement than the entire system or that may already exist. The state box has two conceptual parts. The state data is a collection of data objects and the machine is a process specification that has access to the black box inputs as well as to the state data.

To illustrate this refinement, we begin with the black box of system1 shown in Figure 4. That system draws its inputs (probably through a variety of transactions) from several data sources and produces updates to other data sources as outputs. File2 is both an input and an output. The conceptual, abstract model of the data “inside” system1 is called DB.

![Figure 4. Graphic of Black Box “System1”](image)

Figure 5 shows a state box refinement of system1. There are two important notes to be made. First, the black box and state box are different views of the same system. The black box and state box have the same inputs and outputs. The state box view exposes some concrete data aspects, but not all (the implementation of table1 as pointers and files, for example, is still hidden behind an abstraction, “table”) and exposes no operations-ordering or other concrete process artifacts. The second note of importance is that the step from black box to state box is a one-to-many design step -- the decision here to use a single table plus a configuration dataset is that of the designer.
Every design decision in Cleanroom must be checked along two quality axes. Is the design correct—does it express the same behavior as intended by the specification? Is it a good design—does it have the appropriate reusability, maintainability, and performance characteristics? This paper will concentrate on correctness and will leave discussion of design quality characteristics aside.

The state box, like the black box, is usually expressed as a set of process abstractions organized around data objects. However, instead of being organized around abstract data, they are instead grouped by their shared state data objects.

### 3.2.2 State Box Correctness Verification

A state box is correct with respect to a black box if it has the same behavior as the black box, whenever the black box’s behavior is defined. While there is a great deal of mathematics behind the rigorous definition of correctness, its presentation is beyond the scope of this paper.

### 3.2.3 Process Design: The Clear Box

The third view of a system, object, or part is the clear box view. In this view, procedural details of ordering and local data are exposed. As with the state box, this is a partial exposure, as new specifications can be introduced. The common units of structured programming -- sequences, alternations (ifthen and ifthenelse), and iterations (whiledo, dountil) -- are the building blocks of the clear box. Figure 6 shows a clear box refinement for the “system1 machine” process abstraction. We did not have to begin with a data-abstracting black box in order to see a clear box -- any process abstraction will first be designed as a clear box.
3.2.4 Clear Box Correctness Verification

Because the step from process specification (black box) to clear box is a one-to-many design step, it too requires verification of correctness as well as design quality assessment.

Figure 7 shows the general theory of clear box verification. Given an intended process specification and a proposed clear box design, an abstraction operation is done to yield the actual behavior specification of the design. The two specifications are compared and, if the actual specification sufficiently meets the intent, the design is correct. Again, the mathematical reasoning to support this claim can be found in several sources.

Figure 7. Schematic of Clear Box Verification

The fundamental issues of correctness are:

- Is the domain of the actual specification at least the domain of the intended specification? That is, is the actual function defined whenever the intended function is defined?
- For every element of the domain of the intended specification, does the actual specification produce the same result as the intended specification.
This form of correctness is called “sufficient correctness” since we have not required that the domain of the actual specification be exactly the same as that of the intended specification.

3.2.5 Design Summary

Although the emphasis of verification appears to be on design, a design is nothing more than a structure linking carefully chosen and documented specifications. Thus, the central skill of Cleanroom development is the ability to write a good specification for an object of any size.

One of the most challenging parts of Cleanroom development is maintaining these specifications. Almost any code change will necessitate at least a small amount of specification change; a code change that affects the user can require specification changes at many levels, as higher and higher levels of specification are changed in order to maintain the chain of verification. However, maintaining these specifications has many benefits, including visibility of possible ripple effects, re-examination and evaluation of high-level design decisions, training members who have just joined the team, and improving everyone’s proficiency at review. Indeed, verification is easy when specifications are good. A learned Cleanroom skill is writing specifications that give only enough detail to support verification, yet hide enough that changes do not propagate unnecessarily upwards through the system.

Each step of box structures design introduces new black boxes: “table1,” “config,” and “system1 machine” in the state box design of Figure 5; “system1 part1,” “system1 part2,” and “local” in 8. Some of these encapsulate data and will need further refinement through state boxes; the rest will proceed next to refinement as clear boxes. The process is complete when all data and processes are expressed in terms of black boxes that are already implemented, either by the programming language, by reused code, or by environment objects.

Many languages encourage re-use by permitting processes and data to be made generic for instantiation in different parts of the system. In box structures these are called “common services.” Procedures, with or without parameters, are an example of common services, as are shared data objects. Classes (in object-orientation) are a higher-order form of common services, where a number of objects have the same specification.

4. The Cleanroom Review Process

Now that we have surveyed the fundamental theory that supports Cleanroom development, we can consider its practical application to application development. The theory is very important, however, and we must be careful when departing from it. The theory permits us to develop code that is as near to zero defect as is humanly possible, without any testing (notice: no mention has been made of testing; correctness is evaluated through the verification process). However, there are still many opportunities to produce systems that are of less than perfect quality:

• correctness is judged with respect to a specification and the specification may be invalid

• our evaluation of correctness is based on assumptions about the design and these assumptions may be wrong, particularly when they deal with environments or units developed outside of our control

• we may choose a standard of proof that is less than fully rigorous

In the next sections we will look at the review process and discuss how it can be tailored for application development.
### 4.1 The Elevator Model

The development team produces a hierarchy of specification objects linked together by box structures. However, it does not produce this hierarchy all at once. Nor do most Cleanroom teams develop the highest-level black box specification completely before moving on to the first level of design.

Instead, the development team engages in an “elevator” model of specification, design, and verification for any part (large or small). Figure 8 shows an overview of this model. In this model, we make a slight distinction between “development” reviews and “verification” reviews.

Development reviews focus on understanding the object in its context and laying the foundation for eventual verification or validation. The most critical part of that foundation is the adequacy of the specifications that support the design. Design quality issues are also scrutinized in development reviews.

Verification reviews focus on verifying the correctness of a design using the correctness questions described in earlier sections. However, there is still opportunity to re-open design-quality issues or even to question the overall design strategy (though this is hoped to be rare). A typical verification review outcome is to agree that the design is correct, or will be correct if a few small changes are made.

It is not unusual for a review to start out as a verification review, find significant correctness or design-quality problems, and to proceed in the form of a development review. This is especially frequent when a team tries to rush the process, skipping earlier review phases in order to “save time.”

The elevator model applies to any object, large or small. All “sufficiency” or “completeness” judgments are subjective and team-, project-, and object-specific. “Team” refers to the team that owns the object.

The model does not progress monotonically. At any point a review could send the project back to an “earlier” step by agreeing that additional work at that level must be done. In particular, and most important, designs are reviewed in the context of their specifications -- this means that the team could decide to revise a previously “completed” specification, hence the term “conditional completion.” At that point the specification must be re-reviewed in the context of its requirements. If that specification is a part of a higher-level design, then that design must, in turn, be re-reviewed. Only after the team is convinced that the entire structure, from top to bottom, is consistent can it return to the bottom level and continue designing. This “up and down, ending up at the ground floor” pattern gives rise to the name, “elevator model.”

One additional complication, for large projects, is that the project is usually organized as “teams of teams.” Teams that own lower design levels send representatives to form small teams representing higher levels. Coordinating change and re-review in such an environment is a project management challenge.
4.2 Management of Team Review

The goal of team reviews is, ideally, consensus. Although voting, management or team-leader tie-breaking, and the like may be necessary, these should be treated as a last resort. Instead, it should be the role of managers and other leaders to propose consensus points of agreement and to assist the team toward consensus. Consensus also requires the entire team to participate, whereas voting potentially permits minority viewpoints to lead to disaffection and faction.

Cleanroom teams work best when they have many short, focused reviews rather than waiting for a lot of material to build up and then marathon reviewing. Short, frequent design reviews

- reduce the potential re-work involved if the design requires substantial changes;
- limit the amount of new work to be reviewed each time;
- permit the team to see and comment on the evolution; and
- expose design decisions to other team members whose concurrent work is potentially affected.

Unfortunately, many organizations are not set up to accommodate large numbers of short team reviews. There may be difficulty scheduling meeting rooms, individualized work schedules may interfere, and there may be a management perception that “group meetings” are not “producing code.” All these barriers must be overcome, some with the help of technology (videoconferencing and groupware are examples) before the team can really be productive.

It is also a characteristic of software engineers to be perfectionists: they will frequently hold back an object from review until they feel it is “perfect.” While this may, in some cases, speed the process, it may also slow it down considerably because the team may reject its fundamental premise (in which case it
must start all over again), or other team members may be simultaneously developing similar work and could have benefited from seeing the direction of that object’s design or specification. Thus, one of the most difficult obstacles to overcome in introducing Cleanroom is individual and structural resistance to short, frequent team reviews.

Some teams have experimented with the use of specialized rooms where materials are viewed and updated on-line[35]. The cost-effectiveness of these approaches depends on the ease-of-use of the facility. One unsuspected barrier to effective Cleanroom reviews may be the physical unavailability of review rooms on short notice. The more successful Cleanroom projects have acquired one or more special rooms, sometimes even a vacant cubicle, dedicated to their reviews. Such a room also permits team-owned documents to be stored in a central place.

The individual preparation period for development reviews will vary depending on the level of scrutiny required, the amount of material, and its technical complexity. Some teams, believing that the goal of “obviousness” makes advance preparation unnecessary, have done much less of it. Others, faced with inherently complex systems and algorithms, have mandated it.

5. Iterative Specification Improvement

The iterative nature of the review process means that most of a Cleanroom team member’s time will be spent writing and improving specifications. Why specifications? Because a design is nothing more than a collection of sub-specifications linked together by box structures, so if improvement of a design is called for by team review, most likely the improvement will affect the sub-specifications. In this section we describe two techniques regularly used by Cleanroom teams to improve specifications during the development process. The first technique calls for replacing terms and concepts that are either vague, imprecise, or unclear with specification functions. The second technique suggests a chain of specifications that can be used to encapsulate information appropriately.

5.1 Replacing Terms With Functions

A specification function (also called a single-valued functional abstraction [36]) is a mathematical function that is used in specification to describe a particular concept or value. A typical format for the specification function describes its name, its arguments, the type of its value, and the function that derives the value from the arguments. Since this is a function, the arguments are inputs only. Since it is a specification function, it may or may not appear as a design artifact. Its purpose is to improve on a concept or idea expressed in a specification.

The examples shown in this section show how a team that is developing a compiler could improve on its initial draft specification. Figure 9 shows a very preliminary draft. This may be good enough to review, but the review is likely to suggest that certain phrases or concepts be improved. These phrases are highlighted. Knowing which concepts require improvement is an act that requires judgment and experience. Instead of the conditional assignment notation, “if” and “else” are used for familiarity.

Even at this initial draft level of specification, there is still an emphasis on having the black box be a complete definition of the behavior, including exceptional as well as nominal cases.
Figure 10 suggests a few introductory specification functions.

```plaintext
define exists(filename,filesystem) : boolean is
    true, if file system services return “true” when asked if filename already exists
    false, otherwise
end define
define message (type, fill-in) : string is
    case type of
    no_input_file: “Input file “ + fill-in + “ does not exist”
    no_output_space: “Insufficient space for the output file”
    ...
end define
define is_correct_input (filecontents) : boolean is
    true, if file conforms to the language reference guide and meets other semantic constraints
    false, otherwise
end define
```

Figure 10. Initial Specification Functions for Compiler

These specification functions can themselves include concepts (highlighted) that will eventually require amplification and improvement. Figure 11 shows the result of replacing natural-language phrases of Figure 9 with references to specification functions.

```plaintext
[if not exists(input file name), display message(no_input_file,input file name)]
else, if there is insufficient space for the output files, display message(no_output_space)
else, if not is_correct_input(contents of input file name), create listings that are appropriate for the input file
else, display appropriate messages, create the output files to contain object code that is appropriate for the input file and create listings that are appropriate for the input file]
```

Figure 11. Improved Specification of Compiler

At this point we can see the beginnings of a hierarchy evolving in the specification itself. The definition of is_correct_syntax, for example, may point to other specification functions or documents. As with all hierarchical constructions, its shape must be managed: too deep a hierarchy requires too many steps for a reader to find out all the necessary details; too broad a hierarchy and each level may be more than a person can understand. We can also see how a large document could be organized. The compiler example
could have entire chapters devoted to syntax (the definition of is_correct_input), messages (especially if there are to be different messages for different languages), and of course semantics. As with any large document, readability aids like tables of contents, cross-references, and glossaries are vital to swift comprehension and evaluation. Hypertext offers especially interesting opportunities for developing and viewing hierarchical specifications and designs.

Although the structure of the specification need not match that of the design, there is no reason why it cannot do so. It may make sense for the design to have modules corresponding to messages and is_correct_input. Or, it may be important to highlight other specification features (features that are important to the user but not to the designer) and thus have the specification and design structures diverge.

Specification functions can also hide notational complexity as well as areas of remaining vagueness or imprecision. Each individual function can evolve as necessary from good natural language toward mathematical precision, as dictated by the needs of the team.

The hierarchical nature of these specifications is a technique for dividing up the problem space (namely how to precisely specify a large system) without losing sight of the recombination of the individual solutions. And, as teams become more sophisticated and rigorous, they may use type-checking tools or even compilers to assist with specification validation. Indeed, on one project with which the author worked, the specifications for a small component were formalized in specification functions to the point that the specification could be compiled and executed using LISP and could serve as an oracle for testing.

6. Verification and Inspection in Cleanroom

Papers and articles about Cleanroom usually emphasize the mathematical soundness of Cleanroom verification and often this causes readers to believe that Cleanroom is an all-or-nothing approach: if you are not doing full mathematical verification, then you are not doing Cleanroom. However, the actual experience of projects shows this not to be the case [7]. Instead, Cleanroom projects combine all forms of peer review from walk-throughs and informal “buddy” inspections to formal reviews that include correctness proof. The general term used in Cleanroom is “review”, but Dyer’s term [5] “verification-based inspection” is also used. However, all of these terms encompass a wide range of practice. In the next few sections we will look at some of the different kinds of Cleanroom reviews and how they would fit into the elevator process described in Figure 8. The general guide to selecting a point on the spectrum of rigor depends on the risk of a problem escaping detection versus the cost of additional rigor. In the next sections we will look at less-formal review approaches.

6.1 Informal Inspection and Walk-Through

Informal inspections are often used at the very end of a review cycle for an object, when the final review identifies a small number of insignificant changes. Here, the purpose of review is just to ensure that no typographic or other small mistakes are made. A “buddy” system will often be used in such a case, where two developers agree to undertake such a final inspection of each other’s work. The buddy system is less resource-consuming than convening a full review by the entire team, but admits the possibility that a minor change is not actually minor, or that the prior review’s suggestions were not correctly noted.

Walk-throughs often form the basis for the earliest development reviews of an evolving design. Often a quick reaction is needed on a particular idea or direction, or immediate feedback is required to continue making progress. An informal walk-through can be done without extensive preparation by the team members beforehand and materials can be handed out at the review or can even be sketched at the meeting on a whiteboard. Because these are informal meetings, the results and decisions must always be captured and committed to the evolving documentation (specification or design) so that they can be formally reviewed later.
6.2 Verification-Based Inspections

In an effort to formalize the review process, some teams [27] have agreed to adopt elements of the inspection protocol recommended by Fagan [37]. These elements include a separation between reader and designer, the use of a trained facilitator or moderator, and the appointment of an individual to record decisions. Most Cleanroom teams that adopt this approach will augment Fagan’s checklist-based inspection approach (where teams look for bugs or “issues”) with verification-based inspection (where teams apply the verification using, e.g., correctness questions). Bug-hunting alone may be susceptible to a form of Beizer’s “pesticide paradox” – the developers are trained over time to not make those particular mistakes, but will make (and overlook) different ones. Design-quality questions are not off-limits in verification, as they may be in code inspection.

Many Cleanroom teams have combined or distributed the inspection roles. For example, some have adopted the practice of having each team member record design decisions and use the review-preparation period to compare the actual changes with the record. Again, the decision on specific review techniques must be made by the individual project.

Groupware tools, including electronic mail and shared electronic documents, can reduce the amount of face-to-face review time that is needed and can be very helpful when meeting space is tight or when individual work schedules are not conducive to frequent team reviews [38,39]. Care should be taken when using these tools to ensure that the primary goals of review are not compromised.

The ideal approach is to add “correctness shown” to the checklists for design and code inspections. In this way, the organization learns from past mistakes and can apply specific design-quality guidelines. Further, in an organization that is not yet ready for correctness verification, checklist-based inspections are still better than the alternative of unrestrained, ad-hoc unit debugging. A process that employs the kind of hierarchical designs described in the preceding section, but which uses traditional inspections as the Cleanroom “review” techniques, would be a Cleanroom process though not necessarily an optimal one.

6.3 Review: A Summary

There are many techniques for demonstrating correctness and considerable variation in the “standard of proof” required. To use a legal analogy, a team designing life-critical systems might ask for correctness to be demonstrated “beyond a reasonable doubt” whereas a team designing a prototype word processor might only demand that the weight of the evidence points toward correctness. It is the author’s opinion that, for some projects and some objects, inspection techniques sufficiently meet the requirements of a “Cleanroom” review.

6.4 Supplementary Testing

One of the major concerns with sliding the standard of proof away from full verification is that Cleanroom testing is pressed into greater service as a way to find errors instead of focusing on measuring quality. It may be prudent, in some project environments, to back up the Cleanroom reviews with formal testing techniques [40] if less rigorous review methods are used.

It is also the case that testing serves several purposes other than purely reliability measurement. For example, it may serve to engender confidence that all branches have been tested. Or, it may be used as a way for developers to experiment with and learn about language and environment features. These benefits, and other benefits of non-Cleanroom testing, must be weighed against their cost both in terms of resources and the problem that they may detract from the review process.

Before moving toward non-Cleanroom testing, however, it is important to recognize the synergy between avoiding unit test and forcing developers to thoroughly review every unit. As the use of supplemental testing increases, developers may begin to rely on it as a backstop and may eventually return to old habits.
of cut-and-try, ad hoc development. It is therefore recommended that unit testing by developers for the purpose of finding defects be avoided if at all possible.

7. Conclusions

Verification and review are only one part of the big picture of Cleanroom development. This paper has touched on some of the practical issues associated with conducting Cleanroom reviews in the application development. By scaling back the mathematical rigor of those reviews, using an iterative “elevator” model, and gradually combining natural language specifications with formal notations, application developers can achieve many of the benefits of Cleanroom without incurring unnecessary costs.

And it is quality improvement and cost effectiveness that are key to application developers. Quality improvement means fewer defects to find and fix, both during development and after delivery, so better quality translates into lower cost. It also means higher customer satisfaction and lower support costs which again affect the bottom line. But the pursuit of quality is rarely done without an eye to cost, hence pragmatic Cleanroom development teams have adopted the practices that are described in this paper.

8. Acknowledgments

The author would like to express his appreciation to Alan Spangler of IBM’s Cleanroom Software Technology Center for his contributions to the substance of this paper, as well as for his careful review of early drafts.

9. References


