CHAPTER 4

IMPROVING SOFTWARE RELIABILITY THROUGH INTEGRATED CLEANROOM AND OBJECT ORIENTATION DEVELOPMENT

Dr. Alaa Mohamed Fahmy

ABSTRACT

Cleanroom software engineering is an approach to software development that improves quality and reduces cost. The approach takes its name from the clean rooms used in chip manufacturing, where statictical quality control techniques emphasize defect prevention over defect removal. Cleanroom focuses on software reliability, but does not prescribe design techiques. Object-Oriented (OO) development approaches offer techniques for developing designs and architectures that are robust, reusable, and maintainable. The combination of these two approaches leading to an integrated Cleanroom/OO development process stronger than either individual process. This combination offers reliability through correctness verification based on the functional mode of Cleanroom ads well as design guidance characterizing OO development. This paper shows how to integrate object-oriented analysis with Cleanroom black box specification, suggests documentation needed to support verification of OO designs. A guidance in making tradeoffs during the integration process is also presented.

1. INTRODUCTION

Cleanroom software engineering is a set of principles and practices for software management, specification, design and testing that have been proven effective in improving software development quality while at the same time improving productivity and reducing cost [1, 2, 3,]. The name Caleanroom comes from the clean rooms used in high-precision manufacturing [4]. There many similarities between Cleanroom software engineering and the hardware clean rooms after which it is named.

The demonstrated success of the large-scale users of clean rooms, especially the makers of integrated circuits, prompted Dr. Harlan Mills of IBM's Fedral Systems Division (FSD) to suggest that the same principles' defect prevention during design, separation of design and test, and objective testing criteria, be applied to software development. Although the Cleanroom name, and some of its practices, were first published in 1981 [5], it was not described in major journals until 1986 [6] and 1987 [7]. Together, these papers describe Cleanroom practices at that time.

Beginning in 1987, a number of other organizations, in addition to IBM, began to apply Cleanroom techniques [8, 9]. Cleanroom have been evolved to keep up with the changing world of software. Users of Cleanroom have adapted it to coexist with a variety of tools and techniques. Several consulting organizations have been formed to help teams use Cleanrooms, and Ph.D. Dissertations have addressed special issues with the methodology. The keynote phrase for the large-scale adoption of Cleanroom hinges on the idea of an "expectation of quality". Cleanroom represents a fundamental shift away from the notion that errors are likely and frequent.

Object-oriented development is about analyzing and implementing systems that comprise collaborating objects, where each object encapsulate the data methods necessary to satisfy its processing requests. Object-orientation emphasizes the specification of the external interfaces of objects, and requires the practice of information hiding and the encapsulation of functions and data that perform the work of the object [10]. Objects provide convenient concept in which to think about the composition of systems. System composition through object requires thinking about the architecture for a system in terms of assembling systems through the use of both large and fine graines components. Viewing systems as a composition of collaboration objects, also supports the idea of developing not just reusable assets, but domain- specific solution architectures for classes of problems, where architectures, as well as objects, become units of system development and integration.

In pracitce, even though an enormous amount of software has been developed, only a portion of it is recoverable from "mining and defining" efforts. As software is developed in the future, and good software engineering techniques are employed to define and develop robust software objects, we may one day build up a sufficient quantity of software objects such that systems can be composed from reusable components, and software development may become more of an integration acitvity than one of development. One of the most important contributions of OO methods is the concept of developing reusable classes, which through inheritance, may be specialized [11, 12, 13]. This speciatizing of generalized classes and their methods permits some methods to be inherited without modification, and allows others to be specialized as necessary to satisfy unique processing requirements not ad-

dressed by a parent class. It is the class concept that is the driving force behind the composition of systems from reusable components.

Booch describes the underlying modèls (meta-model) upon which all object-oriented methods are based [I4]. He identifies the major elements in this meta model as the software techniques of abstraction, encapsulation, modularity, and hierarchy. The minor elements of the meta-model are typing, concurrency and persistence [14]. These elements form the conceptual framework for the development of classes that represent the behaviors and properties of an object type and their integration into a system. Also identifies three categories of methods that are used to support system analysis and design: top-down structured design, data-driven design, and object-oriented design [14]. Top-down structured design is algorithmic decomposition. Data-driven design derives the structure of software systems by mapping system inputs to system outputs.

One of the most important characteristics of object-orientation is its focus on the behaviors of the system we are to develop, and the behavior of its objects through this descrption of object behaviors, that we define the stimulus sequences, and responses involved in a system communicating with external objects and among communicating objects within the system.

The discipline called "object-oriented analysis" has recognized the importance of understanding the behavior of the software systems and objects of which they are to be composed, Booch, Shlaer, and Jacobson [14, 15, I6] all identify the need for performing analysis of and developing models to describe the behavior of a system. It is interesting to note that one of the most important aspects of Cleanroom is the development of an implementation-and state-free behavioral specification for a proposed system and each of its objects (black boxes).

2. THE CLEANROOM SOFTWARE ENGINEERING

2.1 Fundamental Principles

Although some of the specific practices have changed over time, the fundamental principles of Cleanroom have remained very much the same [17]. First; programming teams can and should strive to produce systems that are nearly error-free upon entry to testing. Second; the purpose of testing is to certify the reliability of the developed software product, and not to "test quality in". Third; incremental team-based management practices allow in-process feedback for continuous improvement, and limit the scope of human fallibility.

2.2 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 [17, 18]. 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 [19, 20] where data verification was important. An additional element was added, and an overall stucture proposed, by Mills [21, 22], with the result called "box structures". Since the resulting framework 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 as shown in figure 1, provides the software developer with three different views of a software system, object, or part. The black box view hides all of the object's implementation details, including any 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 as shown in Figure 2. The box structure usage hierarchy results from stepwise decomposition of black boxes until no new black boxes remain. The usage hierarchy is orthogonal to the inheritance hierarchy of object-orientation.

2.2.1 Specification : The Black Box

Two kinds of black box specifications are separately considered: process specifications and data specifcations. 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).

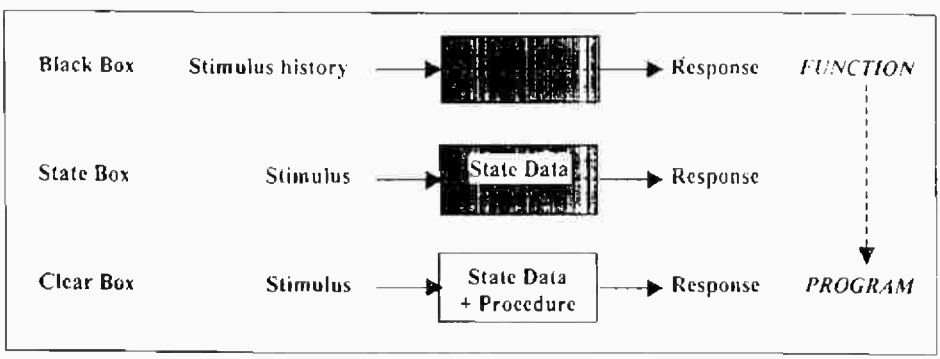

Figure 1 : Box Stucture Method

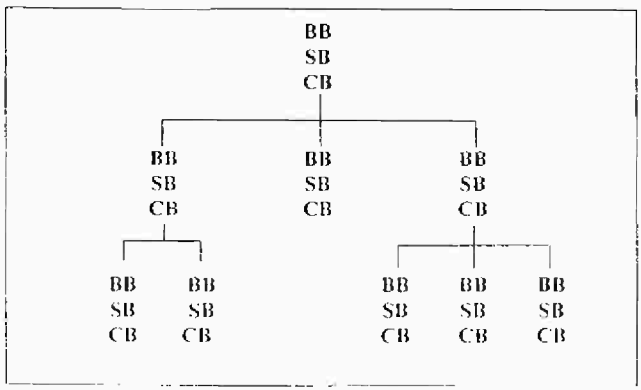

Figure 2 : Box Structure Hierarchy

2.2.2 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. This mapping is called 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 undocumentes "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 such as a batch compiler is necessarily long and detailed, but can 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.

2.2.3 Data Specifications

In the early work on specifications [8], 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 defintion, 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 [23].

2.3 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 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 as shown in figure 2.

2.3.1 Data Disign : 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. 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, 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.

2.3.2 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 defintion of correctness, its presentation is beyond the scope of this paper.

2.3.3. 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 (if then and if thenelse), and iterrations (whiledo, dountil) are the building blocks of the clear box. 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.

2.3.4. 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 [24]. Figure 3 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 several sources. 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.

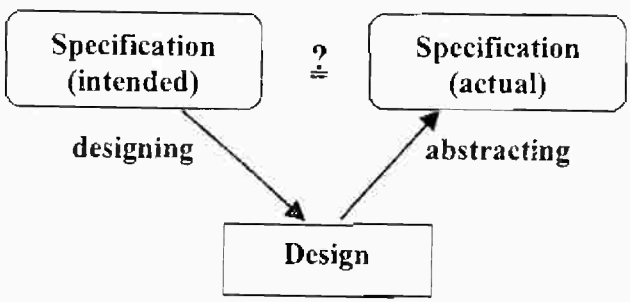

Figure 3 : Schematic of Clear Box Verifcation

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.

2.3.5. 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 effeccts, 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. The process is complete when all data and processes are expressed in terms of black boxes as shown in figure 2, that are already implemented, either by the programming language, by reused code, or by environment objects. Many languages encourage reuse 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 examples 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.

3. OBJECT ORIENTED SOFTWARE ENGINEERING

Object oriented software engineering is defined as a software engineering process that incorporates several concepts [10].

- Objects are the basic units of construction, whether in concept or in actual programming. Objects are organized into classes that have common attributes and/or operations.
- Abstraction hides implementation details of an object (or class) from other objects in the system. The related, though not identical, notions of encapsulation

and information hiding are included in the broader idea of abstraction.

- Messages are the primary, if not sole, information-transport mechanism among objects. Within the broader category of messages, we include their various implementations, e.g., call/return and messages queuing.
- Inheritance defines a particular relationship among objects or classes, namely the ISA relationship. Distinction among the different forms of inheritance is not essential to the understanding of the ideas in this paper.
- Polymorphism means the ability to use the same expression to denote different operations. It includes both operator overloading and genericity.

Any number of OO methodologies [11, 12, 13] have evolved to turn these abstract concepts into workable software design paradigms. This paper favors no particular methodology or tool.

4. INTEGRATING OBJECT-ORIENTED ANALYSIS AND BLACK BOX SPECIFICATIONS

Object-oriented analysis can be alternated with black box techniques to arrive at a specification that combines the best of both approaches. The black box techniques focus on describing the externally-visible behavior, while OOA focuses on organizing abstract attributes for ease of understanding that description. Where to begin depends primarily on how well-formed a set of conceptual objects already exists in the initial requirements or in the physical system being modeled. How specification evolves will be considered, starting from both history and objects.

4.1 OO/Black Box Specification Beginning with History

Most box-structures examples begin with very general understanding of the conceptual state and a list of stimuli that must be accepted and processed. This situation might present itself in a transaction-processing enviroment, such as the car-rental service example described by Hevner [25]. That system was expected to respond to requests such as "check out a car," "return a car," and so forth. At such an early level of discourse, not much is known about conceptual objects except for the basic object "car." As we progress with a pure stimulus-history analysis, we see that the responses to "check out a car" depend on system attributes like "cars available." An initial specification sketch might have a fragment such as that shown in Figure 4.

Stimulus	Condition	Response
Check out a car to renter	cars are available and renter is qualified	OK, car number
Check out a car to renter	cars not available and renter is qualified	Sorry, no cars
Check out a car to renter	cars available and renter not qualified	Sorry, not qualified

Figure 4 : Car- Rental System Specification (Fragment 1)

Our primitive statements about "cars available" are understood to be predicates on stimulus history that we will want to understand and documents as such. At some point, we will have accumulated enough knowledge to decide that the concept of a "pool" of cars is important, and we can write a function on stimulus history that returns a set of cars - the current pool. We could then rewrite the specification in Figure 4 using "is - in pool (history)" rather than "cars are available." As we start asking more questions about the car-rental system, we would soon discover that the cars themselves abstract attributes. For example, a renter might ask for a particular kind of car, and we would need to extract a pool subset of cars having that attribute. This kind of analysis might lead us to a hierarchy of car classes, where the base class has certain attributes (e.g. Current Fuel Level) and the derived classes (e.g. mid-size) have additional attributes. We can continue to describe car classes and their attributes, all within the stimulus history framework.

One thing that separates this model from a true OO viewpoint is there are no "write" methods or operations on the abstract attributes. There's a very good reason for this - the only "real" entity in the conceptual model is stimulus history. So, there is no way to change the "pool of cars" directly - the history is changed by a stimulus from outside the system (e.g. check out) and then specification functions on history enable a view of the pool of cars. As we notice, this feature of the history model has its advantages and disadvatages.

4.2. OO/Black Box Specitication Beginning with Objects and Classes

When the specification activity starts with a richer conceptual model already in place, it may make more sense to begin with objects and add the black box framework to it. We may also decide, after some initial effort, that the history-based specification is growing to be too cumbersome, especially if a conceptual model has already been established. This is an occurrence that, in the author's experience, happens frequently with Cleanroom teams that are unfamiliar with black box analysis, pointing us toward recommending the more generous abstract models to new

teams, and gradually phasing in history (if necessary) after the team has some experience. Whatever has motivated us to do so, we may choose to organize specification around a set of abstract attributes other than stimulus history. The form of the black box description reflects this choice, as in Figure 5.

Stimulus	Condition	Response	Model Update
check out a car to renter	pool not empty and renter is qualified	OK, car number	pool : = pool less car selected
check out a car to renter	pool empty and renter is qualified	Sorry, no cars	no change
check out a car to renter	pool not empty and renter not qualified	Sorry, not qualified	no change

Figure 5 : Car- Rental System Specification (Fragment 2)

Now that we have apportioned the abstract conceptual model of the entire system among conceptual objects (and, potentially, classes, though space does not permit us to expose these details) at the specification level, we can consider the methods we are applying to those objects (e.g., pool. remove - car; pool. is - empty) and the effect of those methods on the abstract attributes of the object. Again, this is not the same as state box design, since the object "pool" is not necessarily identified with any state data - we could choose to implement the concept of a "pool" object as several discrete objects, or further apportion its functions throughout a more comprehensive state data object.

Applying transction closure to the richer models is the same as applying it to the history model. We must ensure that every attribute is equipped to process any method we apply to it. So, when we wrote pool. is - empty, we had to ask whether the attribute pool has been initialized properly to give an appropriate response.

4.3. Finishing the OO/Black Box Specification

The process of completing the OO/black box specification alternates object analysis with black box defintion. The purpose of both is to arrive at a complete description of the externally-visible of the system that can be easily validated against the requirements and that gives appropriate guidance to designers. Space does not permit a lengthy discussion of techniques to be used in validating the specification-

evaluating whether the specification will meet customer needs. Clearly this is an important aspect of system quality, yet there is nothing unique either Cleanroom or OOA that addresses it. Techniques such as prototyping (either rapid prototypes or early increments), facilitated application specification technique (FAST) [28], Quality Function Deployment [29], and through team review can all make important contributions. Many objects in the system will require a black box description, and some of them will be complex enough to warrant their own object-oriented analysis. In a pure world, such objects would not technically appear until the state box or clear box designs, and so their actual specification would begin at that point. However, if one chooses to have a close mapping between specification objects and design objects, then this division becomes much less clear, as does the entire distinction between the "specification phase" and "design phase."

4.4. Correctness Verification in OO Development

The heart of Cleanroom review is correctness verification based on the functional model. In that model, the overall function of a procedure or process can be derived from the functions of its parts. Then, the derived (or abstracted) function is compared with the intended function (specification) to determine if the procedure is correct with respect to its specification. The goal of this section is to introduce techniques that support the use of the functional model in verifying Cleanroom designs that use objects. Warning: some of these techniques are not pretty! However, we will show that they are necessary in order to support true program verification. The most important part of correctness evaluation, and the hardest to get right, is to choose which subspecifications to document. If too many are documented, then a productivity price is paid in maintaining subspecifications that restate the obvious effect of the code. If there are too few, then the review team must derive intermediate subspecifications in real time in order to do an adequate job of review. If those subspecifications are not recorded, then subsequent re-reviews must again re-derive the functions. One common standard among Cleanroom teams has been to recommend that the function of every procedure-call be documented. We might call this activity "flattening" since it has the effect of producing a module that has on off-page references to other functions. Such a module canideally be verified in a vacuum, without knowing either the context in which it is invoked, or the subprocedures that it relies on. Flattening presents more of a challenge in the object-oriented world but it is even more important there. We will look at techniques for flattening, beginning with simple procedures and ending up with subclasses and polymorphism.

4.5. Application Frameworks : Special Issues

Frameworks are class libraries that simplify the construction of specific kinds of

applications. For example, the Microsoft Foundation Classes (MFC) and Borland's Object Windows Library (OWL) are frameworks for creating programs to run under Windows. In the pre-frameworks days, programming for Windows required manipulating a large number of application programming interface (API) functions in order to even display a rudimentary window on the screen. Now, it is as simple as creating an instance of the TWindow class (in OWL) and sending it the "Show" message. Of course, the increase in power is somewhat offset by a decrease in flexibility, and in most framework there are ways to directly access the underlying Windows API. It is for the latter reason that most framework classes present an abstract model (e. g. window) that is nearly identical to its implementation.

There are two special considerations when using application frameworks with Cleanroom. First, the class inheritance hierarchy can grow to be quite deep (as much as 6 deep) within the framework itself, even before the application devlopment team has begun its work. There are dozens of classes and hundreds of methods, combining all types of polymorphism and virtual binding. Understanding the abstract model of each class and the actual function of each method - the black box behavior of each object - is a significant task.

There are many other challenges in framwork-based application development. One is that programs are typically "dense" in method invocations: perhaps half or more of the lines of code are method invocations. Another is that the framework are often packaged with application and class generators that alleviate much of the tedium of program construction. All of these considerations motivate careful and judicious use of intermediate specifications, with it being extremely unlikely that every method invocation will possess one. It also places a premium on development teams that are very familiar with the framework in which they are developing, so as to be able to treat many kinds of objects as if they were language primitives like integer and string.

Finally, it is worth noting that most frameworks require applications to use pointers and arrays, and other "less Cleanroom-desirable" constructs. Although techniques for full formal verification of pointer operations have been proposed [30], a more common approach is to always use care to make pointer-accessed data objects as similar to full-fledged objects as possible, and to verify them as you normally would.

4.6. Verifcation and Inspection: Choices and Guidance

Although most Cleanroom literature has emphasized the mathematical purity of verification, there are many projects that do not require life-critical reliability levels and are willing to accept lower reliability levels, if doing so will improve their

time-to-market. The decision to step back from formal verification is a complex one involving management and technical staff. If done, cycle time may be only marginally improved since the cost of fixing defects found in testing will likely increase, as will the cost of technical support. Since the Cleanroom process is an incremental one, some amount of adjustment can be made throughout the process, but a clear understanding of the goals and priorities is important at the outset. Although this opinion is not universal in the Cleanroom community, it is this author's belief that acceptable Cleanroom team review techniques span a continum between informal inspection and full formal correctness proofs [31]. It is for each team to decide, on an object-by-object basis, what point on this continuum to choose based on the priorities and conditions at that time. An important part of that decision is an assessment of the risks engendered by choosing to de less.

To illustrate, let's again take the example in Figure 4 (the conditional integer exchange) and think about what would be required for a "full" verification. Before we de so, we must point out that the code itself is a model for the object, and thus our precise understanding of the language semantics are critical in program verification. Since achiving that level of understanding for the entire C ++ language can be very difficult, it may be useful to agree on a subset of C ++ or even to choose design language [32]. With that said, it is very much up to us what level of rigor we want to present in our abstractions, and what parts of the semantics we feel comfortable leaving for careful inspections or even for testing.

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 suffciently meet the requirements of a "Cleanroom" review. 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 [33] if less rigorous.

5. CONCLUSION

There is a relationship between box structures and object evientation. In an object-based system, each object (instance) is specified by a data abstraction. Each instance has its own "actual" set of abstract attributes to which a process abstraction would refer. In a class-based system, each class has an associated data specification

whose abstract attributes are part of a "formal" attribute set. Parameter substitution of the instance name for the formal attributes set would occur when the object is instantiated or used, just as one would do with a procedure call. The box structures approach provides for either a strong separation (e.g., stimulus history) between abstract and concrete attributes or a weak separation (e. g., integer ID for both). It also provides for a separation between abstract processes and their implementations. Every box has stimulus-response behavior, and a box always responds to one message before accepting the next. However, there are certain difficulties at the implementation level where, for example, an object may receive one message while still processing a prior one. Concurrency is also a complicating effect. In practice, these problems can be resolved by introducing a more robust or complex mathematical model, or by paying careful attention during review to issues like reentrancy and shared-data conflict resolution.

In an object-oriented system, objects are also related to one another by inheritance. From the point of view of box structures, inheritance is a way of contructing a new data abstraction without having to copy an existing one. Depending on how we define these specifications, a user of the derived class may or may not know it is a derived class. Different languages have different facilities for defining objects; we believe this specification approach can be used, with slight variations, in all cases. Polymorphism and genericity present a signifcant challenge to understanding object-oriented systems, because it may be diffcult to determine which method is being invoked. We have described two approaches to documenting these methods. One way is to completely document such a statement, using one conditional, concurrent assignment for each possible virtual method. Another technique would be to have each method be so similar at the abstract views were identical, even though their concrete implementations are differed. The latter is more in keeping with the *Intent* of polymorphism, but the writing of good, clear, reusable abstractions is diffcult. In some of the preceding sections it may have appeared as if the Cleanroom development is strictly top-down: the box specification is developed to ultimate completion and design begins only at that point. This may reprresent an ideal circumstance in some project situations, but it is far from normal. In practice, some amount of specification is done, then some design or architecture work, and then additional specification. Reviews are used to make sure the evolving design meets the evolving specification. and to make modifications to either or both when necessary to keep them synchronized. Much designing takes place before and during the specification activity for any object. This is an "elevator" model of development: One moves up and down in the hierarchy of objects, but the last ride is top-down, to force the developers to think primarily in terms of what the system does, instead of how the system does it. The test for when any phase is "done" is usually, "is it

sufficiently compete that the risk of proceeding is minimized"?.

However, this model is to be distingushed from "iterative hacking", where there is no clear documented goal of development, and the code- test- debug cycle proceeds as rapidly as possible until either the delivery date arrives or someone declares it to be "complete". Throughout the paper we have noted that there are some places where additional tools and research would be useful. These include:

- Tools to identify ripple effects of changes, especially those involving the class hierarchy.
- Documentation black box specifications for widely used application frameworks.
- Tools to manage the hierarchy of abstractions and perform automatic flattening.

While the absence of such tools is not a barrier to effective Cleanroom use with object-orientation the tools would enhance the productivity of teams that combine these approaches and would make the unique synergies between them even clearer.

Where Cleanroom techniques may be easily added to the stated OO methods, technique integration may depend on successful demonstration, cost benefit analysis, and the desire and will to improve software quality.

6. REFERENCES

1. H. D. Mills, M. Dyer, and R. C. Linger, "Cleanroom Software Engineering", IEEE Software, September, 1987, pp. 19-25.
2. R. H. Cobb, and H.D. Mills; "Engineering under Statistical Quality Control", IEEE Software, November, 1990, pp. 44-45.
3. M. Dyer, "The Cleanroom Approach to Quality Software Development", Wiley, 1992.
4. Kouchalcdjian; "Iessons Learned Using Cleanroom Software in the Software Engineering Laboratory", MSc Thesis, University of Maryland, College Park, 1990.
5. M. Dyer, and H. D. Mills' "The Cleanroom Approach to Reliable Software Development", in Proc. Validation Methods Research for Fault- Tolerant Avionics and Systems Sub- Working- Group Meeting: Production of Reliable Fligh-Crucial Software, Research Triangle Institute, NC, Nov. 2-4, 1981.
6. P. A. Currit, M. Dyer, and H.D. Mills; "Certifying the Reliability of Software" IEEE Transactions on Software Engineering, Vol. SE, No. 1. January, 1986, pp. 3-11.

7. R. W. Selby, V. R. Basili, and F. T. Baker; "Cleanroom Software Development: An Empirical Evaluation", IEEE Transactions on Software Engineering. Vol. SE-13, No. 9, September, 1987, pp. 1027-1037.

8. J. Trammel, L.H. Binder, and C. E. Snyder; "The Automated Production Documentation System: A Case Study in Cleanroom Software Engineering", ACM Tranc. On Software Engineering and Methodology, Vol. 1, No. 1, January, 1992, pp. 81-94.

9. P. A. Hausler; "A Recent Cleanroom Success Story; The Redwing Project", Pro. 17th Annual Software Engineering Workshop, NASA Goddard Space Flight Center, December, 1992.

10. Jan Graham; "Object-Oriented Methods", Second Edition, Addison-Wesley, 1994, pp. 8-9.

11. G. Booch; "Object-Oriented Design with Applications", Redwood City, CA: Benjamin Cummings, 1991.

12. J. Rumbaugh, M. Blaha, and W. Premerlani; "Object-Oriented Modeling and Design, Englewood Cliffs, NJ: Prentice-Hall, 1991.

13. S. Shlaer, and S. J. Mellor; "Object-Lifecycles: Modeling the World in States", Englewood Cliffs, NJ: Yourdon Press, 1991.

14. Booch & Grady; "Object-oriented Analysis and Design with applications", Redwood City, CA: Benjamin- Cummings, 1991.

15. Shlaer, Sally and Steve Mellor; "Object-oriented Lifecycles, Modelling the World in States", Prentice - Hall, 1992.

16. Jacobson, Lvar, Magnus Christerson, M. Jonsson, and G. Overganrd; "Object-oriented Software Engineering", Addison- Wesley, 1992.

17. Richard C. Linger; "Cleanroom Software Engineering for Zero-Defect Software", Proc. 15th International Conference on Software Engineering, May, 1993.

18. Linger R. & Mills H.; "A Case Study in Cleanroom Software Engineering: The IBM COBOL Stucturing Facility, Proc. 12th International Computer Science and Applications Conference, October 1988.

19. H. D. Mills' "The New Math of Computer Programming", Communications of the ACM, Vol. 18, No. 1, January, 1979, pp. 43-48.

20. R. C. Linger, H. D. Mills, and B. I. Witt' "Stuctured Programming: Theory and Pracitce", Addison-Wesley, 1979.

21. K. S. Shankar; "A Functional Approach to Module Verification", IEEE Transactions on Software Engineering, Vol. Se-8, No. 2, February, 1982, pp.147-160.

22. K. S. Shankar; "Data Structures, Types, and Abstractions", IEEE Computer, April, 1980, pp. 67-77.

23. H. D. Mills, R.C. Linger, and A.R. Hevner' "Box information Systems", IBM Systems Jaurnal, Vol. 62, No. 4, 1987, pp. 395-413.

24. H. D. Mills' "A Stepwise Refinement and Verification in Box Structured Systems", IEEE Computer, June, 1988, pp. 23-36.

25. M. S. Deck' "Cleanroom and Object- Oriented Software Engineering: A Unique Synergy", Proc, 8th Annual Software Technology Conference, Salt Lake City, UT, USA, April, 1996.

26. M. D. Deck; "Cleanroom Review Techiques for Application Development", Proc. 6th International Conference on software quality Ottawa, Canada, October 1996.

27. A. R. Hevner, "Object-Oriented System Development Methods", Advances in Computer, Vol. 35, Academic Press, 1992, pp. 135-148.

28. R. S. Pressman; "Software Engineeing: A Parctitoner's Approach, 3rd Edition, MCGraw-Hill, 1992, pp. 180-183.

29. W. E. Eureka and N. E. Ryan, eds.; "Quality Up, Costs Down: A Manager's Guide to Taguchi Methods and QFD", Irwin Professional Publishing, Burr Ridge, IL, USA,, 1995.

30. Y. C. Wong: "A Functional Semantics of Pointer Operations and its Application", IBM Software Engineering ITL, March, 1992.

31. M. D. Deck; "Cleanroom Software Engineering: Quality Imporvement and Cost Reduction", Proc. Pacific Northwest Software Quality Conference, October. 1994, pp. 243-258.

32. J. Rosen; "Design Languages for Cleanroom Software Engineering", Proc. 25th Hawaii International Conference on System Sciences, 1992.

33. B. Beizer; "Software Testing Techniques, 2nd ed., Van Nostrand Reinhold, 1990.