Cleanroom Software Development

Harish Ananthpadmanabhan, Chetan Kale, Mujtaba Khambatti, Ying Jin, Shaun Taufiq Usman, Shu Zhang

Arizona State University

ABSTRACT

The Cleanroom process is a theory-based, team-oriented process for the development and certification of high-reliability software systems under statistical quality control. Its principal objective is to develop software that exhibits zero failures in use. For this purpose the life cycle is different from conventional software development techniques. The approach combines mathematical-based methods of software specification, design and correctness verification with statistical, usage-based testing to certify software fitness for use. Therefore the goals in this method is to reduce the failures found during testing by enabling good and correct designs that avoid rework. Most designs pass through detailed specifications and modeling which are evaluated and proved for correctness using formal methods. In this paper we take a look at the Cleanroom software process and describe a few of its methods.

1 Introduction[4,7]

1.1 Specification

Cleanroom software engineering is based on a set of formal specifications describing the external behavior of the system. Instead of developing software quickly so we can rush to debug it, we spend more time up front preventing errors from being put in.

A strict stepwise refinement and verification process is done using a box-structured approach that allows accurate definition of the required user functions and system object architecture. The approach attempts to increase productivity by allowing the product to be incrementally developed. We can say, for example, that 50% of the product is 100% complete instead of 100% being 50% complete. Cleanroom implements this approach using an object-based technology of box structures called black, state, and clear boxes.

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

1.2 Correctness Verification

Verification reviews are held by the team to formally or informally verify the software using a set of correctness proofs that are standard in a structured programming environment. Correctness verification is done before the software is ever executed, so that the developers are not permitted to get into a "debugging" mode of operation.

1.3 Statistical Usage Based Testing and Certification

Cleanroom uses a formal statistical approach to testing that can indicate the quality of the software and stopping criteria as well. This approach differs from the traditional approach, in which testers assume there are errors in the software and set out to find as many as possible, with the faulty assumption that if enough errors are found and fixed, the software quality would improve. Certification (the Cleanroom term for testing) is performed to certify the software reliability, not test the software in the classical sense. The methods used are statistical quality control (SQC) and statistical usage testing.

Cleanroom certification and, specifically, statistical usage based testing, does not measure quality in defects per line of code. Instead, quality is measured in sigma units.

Cleanroom certification can reduce time to market because you are not spending time testing and fixing when you need not be. When you reach the quality level desired, you can ship, even if complete path coverage is not achieved. Cleanroom certification can tell you when testing is complete and the product can be released.

1.4 Incremental Development

Using Cleanroom methodology, software products are developed in a series of functional increments that sum up to the final deliverable product. These increments represent operational user functions. The most stable requirements are implemented first, with further steps of requirements resolution performed with each successive increment. The integration of these increments is done top down.

1.5 Benefits

The goal of Cleanroom software development is to be able to develop software with zero failures in the field. The process, described in the next section, leads to some benefits that can themselves justify the use of Cleanroom software development over other methodologies.

The biggest benefit of the Cleanroom process is that it reduces the amount of errors and failures found during testing. This results in reducing the development cycle time by avoiding the need for rework so prevalent in most software development.

Another benefit that results from the Cleanroom process is that due to the detailed specifications and models created for the product, the product itself can lead a longer life.

2 Cleanroom Software Engineering Reference Model[1,5,6,7]

The Cleanroom Software Engineering Reference Model, or CRM, is expressed in terms of a set of 14 Cleanroom processes, and resultants of those processes are 20 work products. It is intended as a guide for Cleanroom project management and performance, process assessment and improvement, and technology transfer and adoption. The scope of the CRM is software management, specification, development, and testing (certification). As noted, the CRM is a high-level Cleanroom process template that must be tailored for use by a specific organization or project. Each process and work product in the Cleanroom Reference Model is intended to be elaborated through implementation procedures that address specific organizational or project environments and unique development requirements. These 14 processes embody the Cleanroom technologies described above, and are listed below with brief descriptions.

2.1 Cleanroom Management Processes

Project Planning: To tailor the Cleanroom Software Engineering processes for the project, and define and document plans for the Cleanroom project, and review the plans with customer, the project team, and peer groups to get the software project plans. The entry for this process is when there exists a new or revised statement or requirements or software development plan needs to be revised or a new increment begins.

Project Management: This process is guided by the software development plan to manage interaction with the customer and peer organizations, establish and train Cleanroom teams, initiate tracking, control planned Cleanroom processes, eliminate or reduce risks, revise plans as necessary to accommodate changes and actual results, and continually improve Cleanroom team performance.

Performance Improvement: To continually evaluate and improve team performance by causal analysis of deviations from plans and faults found through the Correctness Verification and the Statistical Testing and Certification processes and evaluation and introduction of appropriate new technologies and processes.

Engineering Change: To plan and perform additions, changes, and corrections to work products in a manner that reserves correctness and is consistent with the Configuration Management Plan. The highest level of specification or design affected by a change is identified as the starting point for any re-specification, redesign, re-verification, or re-certification, as well as any other revision activity.

2.2 Cleanroom Specification Processes

Requirement Analysis: To define requirements for the software product (including function, usage, environment, and performance) as well as to obtain agreement with the customer on the requirements as the basis for function and usage specification. Requirements analysis may identify opportunities to simplify the customer's initial product concept and to reveal requirements that the customer has not addressed.

Function Specification: To make sure the requirement behavior of the software in all possible circumstances of use is defined and documented. The function specification is complete, consistent, correct, and traceable to the software requirements. The customer agrees with the function specification as the basis for software development and certification. This process is to express the requirements in a mathematically precise, complete, and consistent form.

Usage Specification: To identify and classify software users, usage scenarios, and environments of use, to establish and analyze the highest level structure and probability distribution for software usage models, and to obtain agreement with the customer on the specified usage as the basis for software certification.

Architecture Specification: The purpose is to define the 3 key dimensions of architecture: Conceptual architecture, module architecture and execution architecture. The Cleanroom aspect of architecture specification is in decomposition of the history-based black box Function Specification into state-based state box and procedure-based clear box descriptions. It is the beginning of a referentially transparent decomposition of the function specification into a box structure hierarchy, and will be used during increment development.

Increment Planning: To allocate customer requirements defined in the Function specification to a series of software increments that satisfy the Software Architecture, and to define schedule and resource allocations for increment development and certification. In the incremental process, a software system grows from initial to final form through a series of increments that implement user function, execute in the system environment, and accumulate into the final system.

2.3 Cleanroom Development Processes

Software Reengineering: The purpose is to prepare reused software (whether from Cleanroom environments or not) for incorporation into the software product. The functional semantics and interface syntax of the reused software must be understood and documented, and if incomplete, can be recovered through function abstraction and correctness verification. Also, the certification goals for the project must be achieved by determining the fitness for use of the reused software through usage models and statistical testing.

Increment Design: The purpose is to design and code a software increment that conforms to Cleanroom design principles. Increments are designed and implemented as usage hierarchies through box structure decomposition, and are expressed in procedure-based clear box forms that can introduce new black boxes for further decomposition. The design is performed in such a way that it is provably correct using mathematical models. Treating a program as a mathematical function can do this. Note that specification and design are developed in parallel, resulting in a box structure hierarchy affording complete traceability.

Correctness Verification: The purpose is to verify the correctness of a software increment using mathematically based techniques. Black box specifications are verified to be complete, consistent, and correct. State box specifications are verified with respect to black box specifications, and clear box procedures are verified with respect to state box specifications. A set of correctness questions is asked during functional verification. Correctness is established by group consensus and/or by formal proof techniques. Any part of the work changed after verification, must be re-verified.

2.4 Cleanroom Certification Processes

Usage Modeling and Test Planning: This is where the usage models are created for the purposes of software testing and defining test plans. Customer agreement is obtained based on the usage models and test plans as the basis for software certification. The test environment is prepared, and statistical test cases are generated.

Statistical Testing and Certification: This is where the software's fitness for use (defined with respect to the usage models and certification goals) is demonstrated in a format statistical experiment. Software increments undergo first execution in this process. The success or failure of test cases is determined by comparison of actual software behavior with the required behavior. The results of the comparisons drive decisions on continuing testing, stopping testing for engineering changes, stopping testing for reengineering and re-verification, and final software certification.

In the next section we will go into some more detail about how the above processes affect the software development process; in particular we will focus on the Development and Certification class of processes.

3 Development & Certification Process[2,3,5,6,7]

In the last section we described the different processes involved in a typical Cleanroom software development project. Here we will describe in a bit more detail what is involved in the development and certification class of processes.

3.1 Specification and Design

There are three system structures for specification and design: black box, state box, and clear box. These structures embody important concepts of data encapsulation and information hiding. These structures exhibit identical external behavior but increasing internal visibility. A black box specifies the external behavior of a system or system component. A state box refinement of a black box specifies state data required to achieve the black box behavior. A clear box refinement of a state box specifies procedure designs required to achieve the state box behavior, and may reuse existing black boxes or introduce new black boxes for subsequent refinement.

Cleanroom Software Specification and Design begins with an external view (black box), and is transformed into a state machine view (state box), and is fully developed into a procedure (clear box). The process of box structure development is as following:

1) Define the system requirements.
2) Specify and validate the black box.
    - Define the system boundary and specify all stimuli and responses
    - Specify the black box mapping rules
    - Validate the black box with owners and users
3) Specify and verify the state box
    - Specify the state data and initial state values
    - Specify the state box transition function
    - Derive the black box behavior of the state box and compare the derived black box for equivalence
4) Design and verify the clear box
    - Design the clear box control structures and operations
    - Embed uses of new and reused black boxes as necessary

- Derive the state box behavior of the clear box and compare the derived state box to the original state box for equivalence

5) Repeat the process for new black boxes

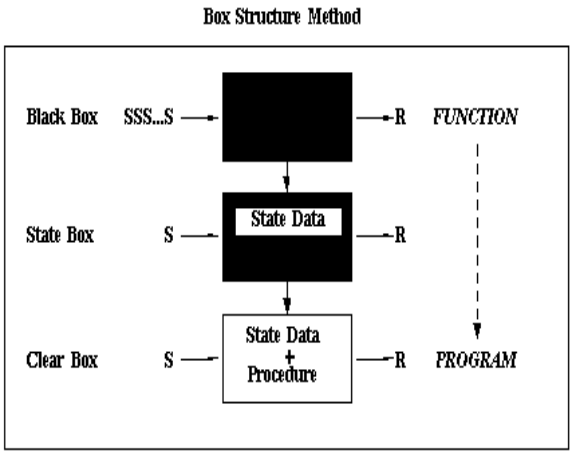

Figure 3.1 shows the 3-tier hierarchy of box structures namely black box, state box, and clear box forms. Referential transparency is ensured which means traceability to requirements.

The theory of *sequence-based specification* is used to develop the specifications. In the sequence-based specification process, all possible sequences of stimuli are enumerated systematically in a strict order, as stimulus sequences of length zero, length one, length two, and so on. As each sequence is mapped to its correct response, equivalent sequences are identified by applying a reduction rule, and the enumeration process terminates when the system has been defined completely and consistently.

3.2 Correctness Verification

There are one-to-many relationship between the black box and the possible state boxes that will mirror the required behavior in software development process. Clear boxes are composed of sequence, alternation, and iteration control structures, such as Sequence, Ifthen, Ifthenelse, Whiledo, dountil, dowhile. All Cleanroom-developed software is subject to function-theoretic correctness verification by the development team prior to release to the certification test team. The correctness theorem is based on verifying individual control structures, rather than tracing paths, so those control structures are the objects to be verified. The correctness theorem reduces verification to a finite number of checks, and permits all software logic to be verified in possible circumstances of use. The verification step is remarkably effective in eliminating defects, and it is a major factor in the quality improvements achieved by Cleanroom teams.

3.3 Usage Modeling & Test Planning

A usage model is represented as a graph, where the nodes represent events and the arcs represent transitions between events. Events may be concrete stimuli, such as inputs from the user or environment, or abstract states-of-use.

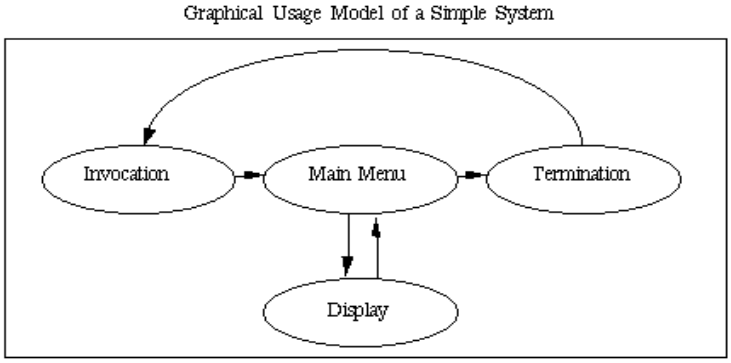

Figure 3.2 Shows a usage model of a simple system

A probability distribution across the usage model structure represents expected usage of the software. Three approaches to defining the usage probabilities are:

- Assignments based on field data,
- Informed assumptions about expected use, and
- If no information is available about expected use, uniform probabilities.

	Invocatio	Main	Displav	Termination
Invocation	0	1	0	0
Main Menu	0	0	0.8	0.2
Displav	0	1	0	0
Terminatio	1	0	0	0

Transition Matrix of the simple system above

The usage model represents possible usage of the software. Test cases are generated from the usage models in statistical testing. Usage models are created incrementally and accumulate into a final form in parallel with increment designs. Usage model analysis provides a basis for test planning, as well as an effective project tracking and management tool.

The customer reviews all the usage models and agrees that they generate all scenarios of use.

3.4 Statistical Testing and Certification Process

Software certification provides scientific way to verify software fitness for usage. Since the set of possible executions of a software system is an infinite population, no testing process can exhaust all execution. Populations and Samples theory are used for verification. If the sample embodied in a set of test cases is a random sample based on projected usage, valid statistical estimates of software quality and reliability for that usage can be obtained. Statistical usage testing produces scientific measures of product and process quality for management decision-making, just as has been done in hardware engineering for decades.

The usage of software can be viewed as a stochastic process, for example, a series of events that unfold over time in a probabilistic way. Next event in a serial can be determined based on present. Multiple models can be defined to address different usage environments, or to provide independent certification of stress situations or infrequently used functions with high consequences of failure. For example, software can be modeled as a finite state, discrete-parameter Markov Chain. The standard analytical results for Markov Chains can be interpreted to yield insights about long-term operation use.

Following correctness verification, software increments are delivered to the certification team for first execution. If required, other forms of testing can be applied prior to statistical testing. Test cases are randomly generated from the usage models, so that every test case represents a possible use of the software as defined by the models. Because statistical usage testing tends to detect errors with high failure rates, it is an efficient approach to improving software reliability.

3.4.1 Exit criteria for testing

A comparison of the usage and testing chains is made on an ongoing basis during testing to gauge the difference between expected usage and actual usage. The comparison is given as the value of a measure called "discrimination," which reflects the degree to which the testing experience has become representative of expected usage.

The value for discrimination will initially be large, will become smaller as testing progresses, and will ultimately converge. When the value converges, the expected and actual usage statistics are indistinguishable, and testing may stop. At this point, everything that is expected to be seen in the field has been tested.

4 Comparisons With Other Approaches[7]

4.1 Cleanroom and the Capability Maturity Model for Software (CMM)

The CMM is fundamentally about management, and Cleanroom is fundamentally about methodology. There is considerable overlap between the scopes of the two, and there are areas of each that are not addressed by the other.

The CMM, for example, has Key Process Areas (KPAs) at Level 2 (Repeatable) that are outside the scope of Cleanroom. Configuration Management and Subcontractor Management, for example, are important management issues not addressed by Cleanroom ideas. Cleanroom, on the other hand, enforces the mathematical basis of software development and the statistical basis of software testing, while the CMM is silent on the merits of various methods.

In general, the CMM and Cleanroom are compatible and complementary. The combination of CMM management and organizational capabilities and Cleanroom technical practices represents a powerful process improvement paradigm.

4.2 Cleanroom and the Object Oriented Approach

Most people who have studied the relationship between Cleanroom and object-orientation regard the two as complementary, each with strengths that can enhance the practice of the other.

4.2.1 Common Characteristics

With respect to the life cycle, Cleanroom follows the incremental development while the object-oriented approach follows the iterative development of the project. Pitfalls of the waterfall approach are recognized in both practices. Cleanroom incremental development and OO iterative development are both intended to provide opportunity for user feedback and to accommodate changing requirements.

The usage scenario involves the application of use case in OO and the usage model in Cleanroom. The OO use case and Cleanroom usage model are both techniques for characterizing the user's view early in development. Artifacts from these activities are used in both design and testing. The State machine representation is common for both the methods for describing the behavior of a design entity.

Both the methods share a common methodology concerning reuse. Reuse is an explicit objective in both practices. The OO class and the Cleanroom common service are the units of reuse.

4.2.2 Key Differences

A key difference lies in the decomposition and the composition done in the two methods. In Cleanroom box structure decomposition, data objects are created and reused as they are needed in the system architecture. In OO, objects are first identified, and then used in system design.

The other notable difference lies in the character usage scenario where in Cleanroom, function theory and Markov theory are used to define "completeness" in characterizing usage. The black box function describes all possible scenarios of use and Markov usage chains can potentially generate all of them. OO use case representation is typically informal.

The hierarchies of both the methods vary. An inheritance hierarchy (i.e., class and its children) is a resource in system design. A usage hierarchy (i.e., a box structure hierarchy) is the system (or a subsystem) itself.

Modern OO practitioners are further from their mathematical roots than are modern Cleanroom practitioners. Consequently, representation formats differ, with OO practitioners using graphics and Cleanroom practitioners using tables or symbolic formalisms.

There is substantial tool support for OO. OO ideas are implemented differently, however, in various programming languages and in various compilers for a given programming language. These differences can have large effects on behavior and performance. There is good automated support for Cleanroom testing, but little for Cleanroom development.

These differences are not necessarily conflicts in the two approaches. They may be complementary, or may serve to check and balance each other. An integrated Cleanroom/OO development process may be stronger than either individual process.

A study/analysis of Cleanroom and three major object-oriented methods: Booch, Objectory, and Shlaer-Mellor, found that combining object-oriented methods (known for their focus on reusability) with Cleanroom (with its emphasis on rigor, formalisms, and reliability) can define a process capable of producing results that are not only reusable, but also predictable and of high quality. Thus object-oriented methods can be used for front-end domain analysis and Cleanroom can be used for life-cycle application engineering.

4.2.3 Shared Fundamentals

There is really no conflict between object-oriented methods and Cleanroom software engineering on the fundamentals of software parts and wholes. There is broad agreement, for example, that objects are defined by (1) their external behavior and (2) their internal data and access programs; systems are defined by (1) their external usage scenarios and (2) their internal

organization of object accesses; and abstraction, decomposition, hierarchy, and other strategies are all important in identifying and relating the parts of a problem.

Furthermore, there is really no more difference between a particular OO method and Cleanroom than there is between particular OO methods—in some cases, perhaps less. There may be more difference between the Objectory and Shlaer-Mellor approaches, for example, than between Objectory and Cleanroom.

5 Conclusion

The typical software lifecycle is about 40% design, 20% code, and 40% unit testing. The Cleanroom lifecycle is 80% design and 20% code and no unit test. Cleanroom spans the entire software development life cycle. The traditional approach of software design tends to culminate in a steady state error population. This type of crafting often fails to provide the software quality needed in today's society.

Cleanroom software engineering allows errors to be found earlier in the lifecycle, which minimizes expensive rework later on and speeds time to market. Designs are simplified, straightforward, and verifiable, resulting in less "spaghetti" code. Quality is achieved by design and verification, not testing. This built in quality lowers the overall cost of the product, and the designs also tend to be more concise and compact than average; always a good thing for embedded developers. Cleanroom supports prototyping, object orientation and reuse. The technology is platform and language independent. And productivity is high. The methodology is based on structured programming and is compatible with reuse. It is a formal, disciplined process, composed of a set of stepwise refinements or transformations from the requirements to the code, where each transformation is verified to the previous level of refinement in order to minimize errors.

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

References

[1] Harlan D. Mills, Michael Dyer and Richard C. Linger, "*Cleanroom Software Engineering*", IEEE software, September 1987

[2] Robert Oshana and Frank P. Clyle "*Implementing cleanroom Software engineering into a mature CMM-based software organization*" Proceedings of the 1997 International Conference on Software Engineering, Boston United States, pp: 572-573, May 1997

[3] Richard C. Linger "*Cleanroom Software engineering for zero-defect software*", Proceedings of the 15th international conference on software engineering, Baltimore, MD USA, pp: 1-13, May 1993

[4] Robert Oshana "*Quality Software Via a Cleanroom Methodology*", http://www.embedded.com/97/feat9609.htm

[5] Cleanroom S/W Engg. - Technology and Process by Stacy J. Prowell, Carmen J. Trammell, Richard C. Linger, Jesse H. Poore

[6] Cleanroom Software Engineering - Reference Model Version 1.0 by Richard C. Linger, Carmen J. Trammell November 1996, http://www.sei.cmu.edu/pub/documents/96.reports/pdf/tr022.96.pdf

[7] A DoD STARS tutorial by Software Engineering Technology, Inc., http://www.asset.com/stars/loral/cleanroom/tutorial/index.html