Abstract

There are circumstances where quality is paramount, and any kind of error cannot be tolerated. When these circumstances arise, special measures need to be taken to ensure quality from the start. Moreover, quality needs to be verified upon project completion. The Cleanroom method of software development ensures the highest quality when developing software. This document outlines the time, purpose, use and method for employing Cleanroom Software Engineering.
# Table of Contents

Cleanroom Software Engineering ................................................................. 1

Abstract ............................................................................................................. 1

Table of Contents ............................................................................................ 2

Introduction ....................................................................................................... 3

Why Cleanroom? ............................................................................................... 3

Cleanroom Development Cycle ........................................................................ 5

Mathematically Based Design ......................................................................... 6

Correctness Verification .................................................................................... 9

CMM, Formal Methods, and Other Overlap .................................................... 11

Conclusion ....................................................................................................... 14
Introduction

The name Cleanroom is borrowed from hardware cleanrooms, which have been used in technical engineering fields for a very long time. Hardware cleanrooms keep problems out by keeping potential contaminating factors from reaching the product. The same idea can hold true for Software Engineering. The stated goal of Cleanroom is software that exhibit a zero defect rate. This is a very lofty goal, but when mathematical and statistical methods are applied, it is not beyond reach. For example, IBM developed a device controller product using Cleanroom Software Engineering that had zero failures in three years use at 300+ locations.

The material in this document is not intended to teach the reader how to practice Cleanroom software engineering; that requires attending courses on Cleanroom management and technology. The best way to learn Cleanroom is to be on a Cleanroom project with people who have Cleanroom experience, and to have taken certified Cleanroom classes. What you can get from this document is an idea of what Cleanroom is and when it might be useful for your own work.

Why Cleanroom?

The reason to use Cleanroom Software Engineering is simple: quality. Quality improvements of 10 to 20 times have been reported when the Cleanroom process was demonstrated in industry, as well as at NASA and in the Department of Defense STARS program. Quality of this high measure is costly, and Cleanroom does not come without any investment by the organization. Therefore Cleanroom is most suitable for critical applications. For example, if defects can cause loss of life or critical financial loss, then Cleanroom should definitely be considered. In addition, when software cannot be updated at a later time you may need to be certain that your quality level is sufficient, and a software development process like Cleanroom should be employed.

There are other benefits of Cleanroom beyond quality. One overlooked benefit is reported increases in individual productivity. In a quality-centric method, it is often the case that quality comes at the expense of a linear increase in person-hours. However, improvements over
baseline productivity have been reported with Cleanroom. Ericsson Telecom developed an operating system with over 374,000 lines of code and documented gains of 1.7X in productivity. An IBM project using Cleanroom also reported a 2X improvement over their baseline productivity measure. This is not saying that productivity increases will be universal—the goal of Cleanroom is not productivity and therefore there are no specific practices or principles directed at productivity increases. However, these statistics indicate that there need not be more time and cost involved in creating quality software.

Where do these productivity increases come from? Cleanroom is a uniquely defined process that decreases required testing, error correction, and maintenance. These savings offset any additional overhead needed for the quality control! IBM used Cleanroom to reduce overall costs, creating a product that exhibited just seven minor errors in the first three years of field use, a huge drop in typical maintenance costs for them.

The whole concept of Cleanroom is about “Errorless Software Development”. If no errors enter development, no errors need to be tested or debugged. Elimination of testing and debugging means faster product development. Keep reading to find out how Cleanroom can introduce fewer errors when developing a product.
Cleanroom Development Cycle

Cleanroom software engineering uses a fully incremental development cycle, much like many of the agile methods popularized today. This model is in many ways superior to the classic “waterfall” model, and provides a number of benefits to the customer and to the developer. Overall, this model increases quality. The primary goal of the cycle is early and continual quality assessment. At the beginning/end of each cycle, the developers and customer alike can reflect on the past iteration and upcoming iteration in order to maximize quality. A side effect of this is increased user feedback, which can often solve problems related to vague or incorrect requirements.

This type of model also facilitates process improvements. The Cleanroom model can be altered and adapted to fit the organization; no process is applicable as-is to 100% of situations.
After iterations and before the start of the next iteration of the incremental process, it is possible to make changes to repair any process related problems that have arisen.

Cleanroom allows for requirements changes. While true formal methods boast high quality, they do not allow you to change the requirements of the system after they have been laid out. However, in Cleanroom everything is done in small increments, and therefore it is not impossible to add another increment with new functionality or changes at any stage during development. If the user feels that changes need to be made, the process can be adapted and the timeline changed accordingly. Quality will not be lost due to such demands, while in a true formal methods environment we would either have to do all of our formal models again, or accept the loss of quality resulting from the requirements change invalidating the formal model.

Finally, there is always a huge integration risk in the waterfall model. At the end of the development stage, how do we know that everything implemented is really going to work? When we are doing our integration and build only once there is tremendous pressure and therefore a tremendous risk of failure. However, when we can increment through our process, it is not a huge problem if integration fails. We simply take it in stride, and continue to apply Cleanroom methods in the next iteration to enhance our quality until our objectives are met.

**Mathematically Based Design**

The incremental life cycle as described does not yet do what was promised in the introduction of this document. While it excels at detecting problems and allowing flexibility, the incremental life cycle does not keep errors from being introduced into the code.

“The technical basis for incremental development in Cleanroom is the mathematical property of referential transparency. In the context of software development, this property requires that a specification and its design decomposition define the same mathematical function, that is, the same mapping from the domain of inputs to the range of correct outputs” [Linger 1996].
Because of referential transparency, we can show at any time that a system is true to its design. It does not need to be modeled up front as is the case in formal methods. So long as in each increment the mapping of inputs to outputs in the design match the actual mapping of inputs to outputs, we can say that no problems have been introduced. In practice each increment must satisfy the specification of the previous increment. This is done through validating the new functionality it provides combined with the specifications for the next increment. As the increments progress the design will be slowly filled in such that everything previously developed is of guaranteed high quality.

Cleanroom relies on a mathematically sound model of design to ensure that no defects are introduced into the software. Cleanroom works on the assumption that programs carry out transformations from input to output. These transformations can be theoretically envisioned as function mappings, as in mathematics. Cleanroom uses three types of functions, and all code that is developed will follow the basic structure of these functions. Because the functions are sound, the code will likewise be sound. Collectively these three types are called “box structures” and are more specifically the “black box”, “clear box” and “state box”.

Box structures map system inputs to system outputs. In the case of a black box, we do not know what goes on inside the code; we only know what the stimulus is and what the response is. Mathematically we can describe a black box as a function like this:

\[
((\text{current stimulus, stimulus history}) \rightarrow \text{response})
\]

The input for a black box can also include the history of stimuli received. Linger explained this best:

“For example, given a stimulus of 5, a hand calculator will produce a response of 175 if the stimulus history is C 1 7 (C for Clear), but a response of 5 if the history is C 1 7 +. The stimulus is the same in both cases, but the histories of stimuli are different, leading to different responses.” [Linger 1996].

By modeling code as a series of black boxes, we can ensure its quality verses our specification by ensuring that the actual black box performs according to the black box definition.

A state box takes input just as a black box does. However, a state box also knows the current state of the system. The result is more than just a set of output: the system also enters a new
state. This is a more refined view of the black box, and allows us to construct a state model of the system. In essence, the “history” of the black box is replaced by an existing “state”. This is an abstraction of the history that allows us to take a higher-level view of the system, also ensuring that there is no “history” case that is unaccounted for.

\[
((\text{current stimulus}, \text{current state}) \rightarrow (\text{response}, \text{new state}))
\]

Clear boxes show what is actually necessary to go between the old state and the new state. Sometimes there are multiple paths or multiple states that can result from a state box - the clear box lets us examine and design these transitions with flow control. In the clear box, the procedures required to implement the state box transition function are defined explicitly.

In summary, the black box defines the behavior specification of an object. By comparing our specification with the actual black box, we ensure that it behaves properly. The state box defines its data encapsulation, that is to say that it defines how that black box can relate to other black boxes in various states. Again, we can ensure quality versus our requirements and
specifications at this level. Lastly, the clear box defines the methods defined in the object. This is the view of how things are actually done. All three levels of abstraction are exposed both at the design and code level by Cleanroom, so that we can ensure no defects are left unfound.

**Correctness Verification**

Box structures provide a well-defined specification that describes what a particular piece of code is doing. The refinement of the black box, state box and clear box continues until code may be developed from it in an unambiguous fashion. In Cleanroom the actual coding portion of the development process is the least described practice! No constraints are placed on how the code is written. However, once out of development a piece of code must go through a process of correctness verification.

The correctness verification phase takes the developed code and compares it against the specification to see if it really does what is outlined in the specification. The specifications define the conditions that code must meet in order to fulfill the function for which it was developed. Correctness verification uses function-theoretic static code analysis to do just that. The term ‘function theoretic’ implies that there is a one-to-one mapping between the code and the specifications. Static code analysis is a term describing the process of examining source code in a line-by-line way, without ever executing or compiling it.

The process of performing a function-theoretic static code analysis is done in an informal way. First, the Cleanroom team gathers to review a piece of code. Correctness verification is done by examining the control structures of a piece of code and verifying based on the specification that the logic will perform as expected. This review is done mentally and verbally, with the team discussing any problematic code and coming to consensus on its function. Cleanroom does not require written proofs against the logic of each control structure.

In the following example, written in a pseudo C-style language, the code was developed satisfy a specification to test if a character is numeric:

```c
function isNumeric(char c)
{
```
if ((c >= '0') and (c <= '9'))
    return true;
else
    return false;
}

In this case there is one control structure that requires review. If the character passed in is in the set [0,1,2,3,4,5,6,7,8,9] we expect the function to return true. Based on what we know about character sets and the language used to develop the code if the character is in the set then the logic ((c >= '0') and (c <= '9')) will return true. When the character is not in the set then the logic will return false. Both cases result in the expected behavior for the entire method, and so the code passes the correctness verification.

The above case is very trivial but it shows how effective this form of code review can be. Control statements are the most likely area in which a problem can occur and this verification helps to identify any problems introduced by the developer in the code’s logic. Because we are examining the control structures directly, this form of static code analysis can be done in a practical finite amount of time. This process can furthermore be done incrementally as the code changed in later iterations.

It is important to note that this process requires very good specifications in order to be effective. This is where the box structures play an important post development role; without box structure design there is no reference to which we can compare the logic.

In the case of outside software being introduced, anything that has been developed outside of the Cleanroom process must go through this correctness verification. Otherwise the new code would be introducing a potential contaminant into your Cleanroom environment. This can be difficult if the code lacks the proper specifications. Often code needs to be re-engineered from the beginning using the Cleanroom process in order to ensure that it will not introduce errors into the other Cleanroom developed software. This makes code reuse difficult but not impossible for the dedicated Cleanroom development team! If a team is willing to accept that their code cannot henceforth be verified, they may choose to use libraries or other modules.
developed outside the Cleanroom environment. This is contrary to the spirit of Cleanroom, however the Cleanroom process is adaptable and flexible enough to fit the real world.

Statistical Usage Testing

In conjunction with the box structure specifications in the pre-development phase, usage specifications are created for the statistical testing phase. Usage specifications are simply descriptions of how the system will be eventually used. Usage models need to be defined for all possible scenarios for a given piece of code along with the probabilities of each scenario occurring. This information can provide invaluable insight into the complexity and quality of the code at a statistical level.

A usage specification identifies and classifies environments, users and scenarios based on a model. In general, the usage of the system is broken down gradually, starting with the broadest categories of use and ending with specific scenarios. These are then used to establish a hierarchical usage breakdown and a probability distribution for a given piece of software. For example, perhaps the software is offered in English and French, and so language would be the broadest category. For each language, perhaps the software is available for three operating systems, Windows, Mac and Linux. Then for each operating system, it might be that the software supports a number of hardware platforms. The software may have different user types for each platform, such as Administrator and Regular User and so forth. Lastly, for each user type we need to define how often each function is used. Usually this process of breaking down the usage is done in conjunction with the client in order to obtain their agreement as to how certain parts of the software will be used. If desired, multiple usage specifications can be constructed, with more detailed specifications for critical code and less detailed specifications for remaining code.

Cleanroom does not define the format of the usage specification, however often Markov chains are used. Markov chains are essentially directed graphs with nodes as states of use and arcs as the stimuli that cause state transitions. It is possible for these Markov chains to become very complicated very quickly. This is why we record the frequency for every path. If a
frequency is rare enough to be below the statistical measure of quality defined by our customer, then we need not model it any further. If French Linux users on Apple hardware with Administrator privileges only use the Thesaurus 1/1,000,000,000 of the time, then it may not be statistically necessary to perform quality control on that scenario. Statistical Usage Testing concentrates on finding the defects that are statistically most significant. This is the most efficient way to test software, since the most destructive problems will be eliminated first, and money will not be spent on potentially harmless problems if it is not available.

The process of statistical testing involves randomly selecting paths from the hierarchical usage models. Enough paths need to be selected to represent a statistically significant sample of the given occurrence frequencies. Since we are selecting based on the occurrence frequencies, we could also give critical paths in the system higher simulated frequencies if they are rare but very important. Controlled tests are executed on the randomly selected paths and the results are
validated against the requirements and specifications. It should be noted that controlled automated tests can be generated from usage specifications automatically with tool support.

The goal of statistical testing is to verify that the system does in fact behave as specified. Software contains infinite paths of execution, which makes it impossible to test every path. However, by selecting paths randomly and choosing a large enough sample it is possible to statistically verify that the system will work to 99.9999% or even higher if so desired.

Formal Methods, CMM, and Other Overlap

Cleanroom has some significant overlap with other fields in software engineering. Most obvious is the overlap with Formal Methods. Cleanroom, like Formal Methods, is based on mathematical principles. Cleanroom, like Formal Methods, is about 100% quality. However, the approaches are by and large quite different. Formal Methods require that you form a complete view of requirements in advance, and then model the entire system at once in order to prove its quality. Cleanroom allows you to model your system in an incremental way, adapting it to changes as you go along. Formal Methods use logic as their basis, Cleanroom uses function-mapping. But despite these differences, it is actually possible to combine Formal Methods with Cleanroom. Languages such as Z can be used in addition to Cleanroom to ensure even higher quality, either before, during, or after the Cleanroom process. For examples of how Cleanroom and true Formal Methods can be integrated, see “Integrating Z and Cleanroom” by A Stavely, 2000.

There is some overlap between Cleanroom software engineering and the Capability Maturity Model (CMM). Key Process Areas of the CMM (KPAs) describe the features that a process should have in order to be certified at a certain level of maturity. Cleanroom software development covers a large number of KPAs as defined by the CMM, and can be valuable to an organization that is looking to implement or improve a CMM certified process. The SEI has developed a Cleanroom software engineering Reference Model (CRM) that is, much like CMM, a useful tool for project performance assessment and improvement. Some Cleanroom processes do not overlap with CMM KPAs, for example, the Cleanroom processes that are
associated with testing are not addressed by any KPA. For those interested in which Cleanroom practices and processes map to which specific KPAs, the SEI has also produced the document “Cleanroom Software Engineering Implementation of the Capability Maturity Model for Software (1996)” that maps Cleanroom to KPAs in detail. Research has indicated that Cleanroom is consistent with and supportive of an SEI level 4/5 CMM software process.

Cleanroom can also overlap other traditional testing procedures. In fact, the Cleanroom reference model states that after statistical usage testing, additional quality assurance tests should be performed if required or specified by the customer. These can be acceptance tests, alpha or beta tests, or any other kind of test used in software engineering. The important thing to remember if you have concerns about how Cleanroom can fit with this or that practice is that Cleanroom is just a starting point. You need to adapt and expand the process to meet the needs of your project and organization.

Conclusion

Cleanroom software development may be a wonderful advance in the process of software development or may just be a downright weird approach, most likely a little of both. Looking at Cleanroom form a theorists’ point of view Cleanroom provides a theoretical foundation to software development in its use of mathematically based software development and statistical quality control. By not introducing errors into the development phase there should be no testing requirement in the process. Statistical testing provides the benchmark as to performance and failure rate and helps verify the inputs to the development process by checking the its outputs.

On the other end of the spectrum is the pragmatist who sees Cleanroom as too radical a departure from conventional software development. The level of experience and training required to have a functional team of Cleanroom developers may not cost effective. Developers develop code, and Cleanroom is more about specifications and statistical models than coding. Cleanroom is not much like the software development processes of industry today. As well it is too new and still is relatively unproven in the bold claims it makes.
In the end one has to understand the appropriate use of Cleanroom software development. Cleanroom is suitable for very particular types of software where the human and financial risks of having errors are too great to be left to chance. This generally does not fit the mold of mainstream software development, in which the concentration is often on getting the best price in the best time period. From what has been shown, Cleanroom is anything but a mainstream development process.
References


