Cleanroom Software Development Process and Assessment of Application

Emmeline Pearson
Department of Computer Science
Drexel University
Philadelphia, PA

Abstract

The overall goal of the cleanroom development process is to create reliable software by using formal methods to design and develop software without bugs. Cleanroom has been used in many industrial software projects, but despite its success in those projects, it has not been widely adopted in industry. Projects that used the cleanroom process boasted dramatic reductions in errors, higher productivity of developers and predictable development times. However, some developers question the need for more training (and time) to create formal specifications and the separation of testing from development. Despite its drawbacks, the cleanroom process has shown remarkable progress in creating highly dependable, and reliable, software with very few bugs.

Introduction

The development of software is a relatively young engineering discipline compared to other types of engineering such as mechanical or civil engineering. This newness of the software development field means that the processes of quality control and expectations are not as refined as other engineering disciplines. Software is often shipped with known errors whereas that would be unacceptable for other engineering fields. For example, in civil engineering building a faulty bridge would not be tolerated or seen as acceptable. The cleanroom process aims to fix this problem for software engineering. The goal of the cleanroom process is to prevent defects in a piece of software rather than focusing on removing them after a product is created. Cleanroom uses rigorous mathematical proofs of correctness to create specification and statistical usage-based tests to verify functionality of a product [1]. Along with reducing errors in code, cleanroom aims to make software development more predictable, and manageable.

The Cleanroom process was developed by Harlan Mills and his colleagues at IBM in 1988 [2]. The first project to use the cleanroom process was the IBM Cobol Structuring Facility project which was a very large-scale project of 85 thousand lines of code (KLOC) [1]. Despite the scale of the project there were only 3.4 errors identified per KLOC [1]. After the success of the first project, Mills published his description of the process, and it was adopted by other development teams from there. Although the cleanroom process has not been adopted across the industry, it has proven to be very effective and produces reliable software with very few bugs.

Cleanroom Process Definition

The goal of the cleanroom process is to create software that is certified to be reliable and free from critical bugs. The main component of this process is formal mathematical specification. Mathematical functions define the relation between how a given input corresponds to output. This

definition can be applied to programs because programs take in some input and perform operations to generate an output [3]. Every part of software can be decomposed into this type of function structure. To be able to prove that software is free from critical errors, it must be shown that it is correct. After defining the mathematical function that each program implements, the function undergoes correctness verification. Correctness verification involves defining conditions to be met for achieving the correct software which are verified through proofs [3]. To be able to show that software will be reliable once it is released, statistical testing is used. Exhaustively testing software would be very difficult and ultimately not useful in detecting critical errors quickly. Sampling, much like in a manufacturing line, is used to test some highly critical areas of functionality to get a sense of the overall reliability and quality of a process [1]. Cleanroom aims to test the cases that will be most used, and therefore are the most important to be free from errors [1]. To see a complete view of the cleanroom development process refer to figure 1. The cleanroom process is carried out by three distinct teams: the specification team, the development team, and the certification team.

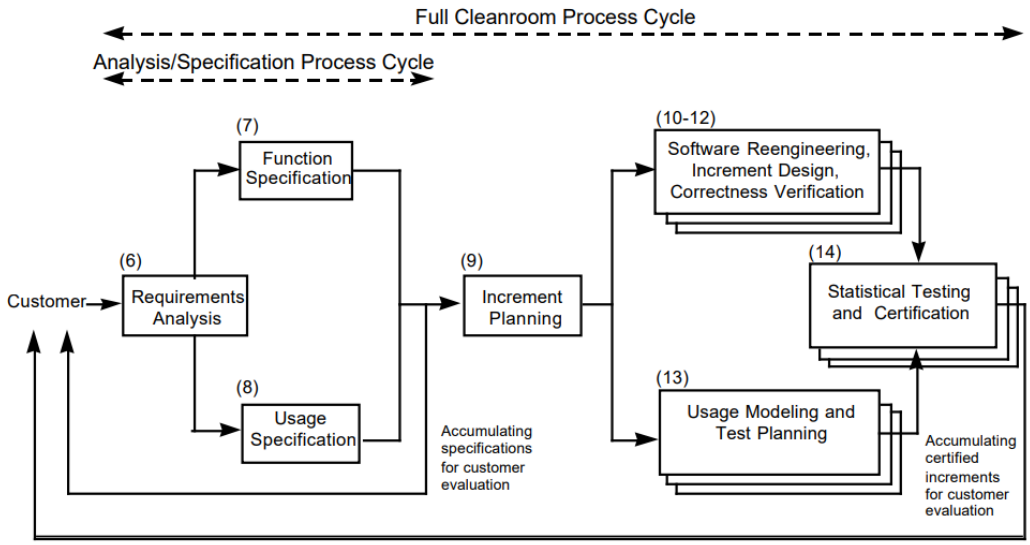

Figure 1: Flow chart of Cleanroom development process steps from The Cleanroom Software Engineering Reference Model [3].

Specification team

The specification team is responsible for setting up specifications and requirements for the development and certification teams. This includes defining software increments, statistics of use, functional and performance requirements. The specification process is split into two main parts. The first part is to identify requirements in coordination with the customer and the second is to define functional specifications. In identifying requirements substantial customer (or user) interaction and feedback is required to ensure the right end product is being produced [1]. When considering user feedback, it may be possible to simplify initial requirements or avoid rework later by making sure that the requirements will meet what the user is looking for [3].

After user feedback is addressed, the specification team then must define functional specifications for the development and certification teams to use. To define functional requirements the team uses the box structure design process (see section below) and mathematical function definitions. A mathematical function is a rule that defines all sets of unique input to output pairs. For the certification team, the specification team can define usage scenarios and the probabilities that those situations occur which are used to generate test cases [3]. Markov models are created for the usage of the product to give estimates of the long term expected use of the software. Once these functional specifications and usage probabilities are determined feedback from the customer is once again obtained to ensure that the outcome product fits what the customer wants.

Box Structure

The box structure design process is based off of Parnas usage hierarchy of modules. Parnas defines modularization as "… a mechanism for improving the flexibility and comprehensibility of a system while allowing the shortening of its development time" [9]. To effectively use modularization the components that a system is divided into must be optimized. Often, programmers design systems in a logical flow chart set of steps, but this is not always the most optimal, and is likely to be hard to change if design decisions change. The hierarchical system Parnas defined starts by first listing out difficult design decisions, and decisions that are likely to change in the future [9]. Then using the list of those decisions, the system should be broken into modules to hide or encapsulate these decisions from other modules. By encapsulating these decisions, it becomes much easier to change a single module, if the design approach changes, without affecting any other functionality of the system.

In the case of cleanroom design, the encapsulation approach is applied to the idea of box structures. Boxes are independent modules or data abstraction objects that define a set of operations that can be used to access internally stored data [4]. Each box has three types of definition: black box, state box and clear box. Black box defines external interactions between boxes with no regard to the internal processing or functionality within the box. This is user visible behavior. Mathematical functions are used to define input and outputs of a black box. State boxes are also defined using mathematical functions, but these define the internal state of a box by describing how the initial state is transformed into the output state. The clear box defines further internal functionality by defining a procedure to carry out state box transitions (the program that implements the corresponding state box) [1]. Using the box structure "… enforces completeness and precision in design of software systems…" [4]. An important feature of box structure design is referential transparency, which means that the specification of each box is enough to design and implement it without depending on the implementation of any other box.

The cleanroom process overall is aimed at creating small incremental pieces of work which allow for faster feedback loops both between internal teams and with the customer. The specification team defines small components of work that will be integrated with the overall system as they are developed. These incremental pieces of work are passed to the development team once they are completely defined.

Development Team

After the specification team has defined the specifications and requirements, their work is handed over to

the development team. The development team designs and implements the increment specified by the specification team using their box structured design and functional requirements [4].

Before actual coding begins, the team does iterations of design reviews and intended functions reviews. Intended functions are a "…definition of the full functional effect on data of the control structure…" [3] where it is present (often defined at the clear box level). The goal of reviews is to make the code as simple as possible through using common services and limiting data scope while still verifying the mathematical correctness of the component. Once the design has been approved by the team, implementation begins. Implementation should be straightforward and follow the design agreed upon by the team. To ensure the correctness of implementation at this phase, team code reviews are done to check the code [3]. Once the development team is complete, the work is handed off to the certification team without doing any debugging (or even compiling in some cases). If there are errors found by the certification team, the work is returned to the development team to be fixed.

Certification Team

The certification team receives the code from the development team and is tasked with ensuring it is correctly implemented. To verify the code, the team uses statistical testing and analysis in a stratified test design process according to the usage specification that the specification team defined [4].

Usage specification defines a probability distribution of all possible usage patters and scenarios (including erroneous and unexpected usages) and the probabilities that each will occur [1].
This idea of statistical usage specification for every possible internal state and the probability of each state occurring from the previous state is based on Markov chain usage models [4]. The Markov model can be shown as a directed graph where nodes (stages) are connected by arcs of each possible transition. By traversing the graph different usage scenarios are generated. The Markov model "…reflects the stochastic nature of software use and permits analysis of usage…" [3] so that sufficient test cases can be created. Some additional test cases may be added, such as those called out by a customer in the specification document or those required by standards or laws.

Once the set of test cases are determined they are implemented by the certification team. An emphasis is placed on automated test cases to allow for reuse of the tests after each increment of software is delivered. After running the tests, the results are assessed and quality measures such as performance are gathered. Any errors discovered or metrics not met result in the increment being returned to the development team to fix.

The interaction between the certification and development teams allows for feedback and quality control. This quick feedback loop is helpful because the smaller increments of development make it much easier to go back and fix any bugs that are encountered.

Examples of use and results

The cleanroom process has been used for the development of many industrial strength software systems. Overall, the results are very positive. Projects using cleanroom are completed on time, with less lines of code, safer design, more productive developers, higher moral for programmers and contain less

bugs [4]. Less than 5 errors per thousand lines of code are found in cleanroom projects compared to 50 per thousand lines of code in normal development process projects [4].

The first instance of cleanroom use was in an IBM project, completed in 1988, called the IBM Cobol Structuring Facility which was a product for automatically restricting COBOL programs [2]. The project was 85 KLOC with an error rate of 3.4 errors per KLOC [1]. The average productivity per person-month was 704 LOC which when compared to other IBM team averages is five times higher. Amazingly in the first three years of use only 7 minor errors were reported. The cleanroom design process resulted in a system that was far simpler and smaller than was originally estimated.

Following the publication of the paper on the use of cleanroom in IBM projects, many other development teams were inspired to try out cleanroom. For example, NASA used cleanroom for their satellite control project in 1989 [1]. The project on a whole was 40 KLOC and their error rate was 4.5 errors per KLOC. NASA claimed that productivity increased by 80% and their team completed 780 LOC per person-month. When looking at testing 60% of programs compiled correctly on the first attempt [1].

In 1993 the Ericsson Telecom AB Switching Computer OS32 operating system was developed using the cleanroom process [2]. The project overall was 350 KLOC and had a failure rate of 1 error per KLOC. Development productivity was also increased by 70% [2]. The project was written in PLEX and C and had a large 73-person development team.

Cleanroom has continued to be used in further projects (see table 1 for more), but it has not been widely adopted despite its proven success [5].

Year	Project	Quality
1991	IBM Language Product First increment 21.9 KLOC [2]	Failure rate: 2.1 errors/KLOC [2]
1993	University of Tennessee: Cleanroom tool 20 KLOC [2]	Failure rate: 6.1 errors/KLOC [2]
1988	IBM Cobol Structuring Facility. 85 KLOC [1]	6 person team. 3.4 errors/KLOC and 740 LOC per person-month. 7 minor errors in first 3 years of use [1]
	IBM 3090E tape drive. 86 KLOC [1]	1.2 errors/KLOC and 5 person team. [1]
1992	Martin Marietta Automated Documentation System. 1,802 LOC [1]	4 person team. 0.0 errors/KLOC, no compilation errors and no errors in testing or certification. [1]
1993	Ericsson Telecom OS32 Operating System. 350 KLOC. [1]	73-person team, 18 month project. Error rate of 1.0 errors/KLOC. Productivity increase of 70% for development [1].

Table 1: A selection of projects that used the cleanroom process.

Assessment

Positives	Negatives
• Many less errors overall • Less Debugging • Errors found are easier to fix • More productive developers • On time deliveries • Statistically relevant test cases • Reduced maintenance costs	• Context switching when errors are found • More time spent in specification and design at the start of the process • Formal specification training needed • Not test driven development

Overall, the cleanroom process seems to have positive outcomes. The cleanroom process boasts, many less errors overall, faster development times, easier to fix errors (when they are found) and reduced maintenance times. The highest value benefit gained from cleanroom development is high quality software (written the first time) with very few bugs. The debugging process is very inefficient and error-prone [2] so, it is beneficial to avoid debugging whenever possible. Sometimes by fixing one bug in code, developers mistakenly introduce another bug. By writing correct code from the start, this debugging process can be avoided. The formal specification process also means that the errors that are found are often much easier to resolve [2]. Because the functionality and design have already been proven out by the specification process, any errors found during development or certification are often small mathematical errors which are much easier to fix than large scale design flaws. The design process is set up to address this issue by making sure that each component is independent and works without requiring a certain implementation.

The testing process is also designed to combat the most important errors, by using statistically based testing. Statistically based test cases are aimed at finding the most critical errors first [5]. This allows for higher confidence in the output product's quality and also allows for a prediction of the reliability of the software based on mean time to error metrics [5]. Unit testing is common practice in the software industry, but statistical testing based on user probability distributions may be more efficient. Statistical testing is widely used on production lines where samples are picked off the line and tested for quality [5]. The negative side of statistical based testing is that some developers wonder if it is enough test coverage. However, the counter to this argument is that the test coverage is directly related to how a user will use the software [5]. Normal unit testing uncovers errors, but the errors uncovered are not always highly important or errors a customer would uncover. Statistical testing is aimed at quickly identifying the most important highly critical errors based on the usage model [5].

Another benefit to the cleanroom process is that many of the projects that used the cleanroom process were highly efficient and delivered on schedule [1]. Since increments are planned out by the specification team and encompass all work, it is easier to plan the timeline for delivery of a cleanroom project. The predictable timeline is very important to management teams and to customers. Since software is often developed for a customer, it is important to think about their timeline needs.

The idea of satisfying a customer through valuable software deliveries brings up the agile development process. Cleanroom uses many ideas which overlap with the principles of agile

development. For example, one agile principle is that "working software is the primary measure of progress" [6]. In cleanroom, each iteration of code that is completed and certified, is integrated with the product from the start, and each new iteration of code brings new value to the product. Also, another agile principle is "simplicity- the art of maximizing the amount of work not done - is essential" [6]. This relates to the specification process in cleanroom. By iterating on the specification, and design, with customer input, cleanroom is able to identify the optimum design that reuses code as much as possible. Boxes in a specification can be created so that they are reused in multiple parts of a system's flow. Customer input into the design also ensures that the product that is being developed matches what the customer actually wants and needs. This can help prevent costly rework if required changes are discovered later in the project.

Agile is a design philosophy that focuses on delivering working software by collaborating with the customer and building empowered teams. The philosophy highlights maintaining a sustainable development pace while being able to adapt to changing requirements. Agile defines 12 principles that improve the software development process. [6]

One negative to the cleanroom process is that training will be required for both developers, and management, to understand the process. Additionally, new tools may be required to create probability distributions and automated test suites. However, despite the extra training involved, many teams have found the learning and new tools to be worthwhile [5]. For example, the US army used the cleanroom process and found that they had a "return on investment of at least 11:1" on the first project they applied cleanroom techniques to [7]. The army used formal, classroom-style training courses followed by coaching and demonstration of examples from team members [7]. Once the teams learned the new approach, they were able to adapt it to their specific projects. So, although there was some initial learning time required to use cleanroom, it was worth it for previously completed projects.

The biggest potential downside for the cleanroom process is the separation of developers from the testing process. Developers hand their finished code to the certification team without any testing or even, in some cases, without compiling their code. The certification team then runs through their testing process and returns the code back to the development team if errors are identified. Because the certification team is separate from the development team, once the developers finish one increment of work and hand it to the certification team, they will start a new increment of work. Then, if errors are found they have to switch back to the old piece of work to debug errors. This could create a lot of context switching for the developers. Also, there would need to be lots of communication between the teams about where bugs were found. Context switching is generally harmful to productivity because as attention switches between tasks there is some time required to remember what the task entailed. If instead you focus on one task at a time, this relearning time period is not required. In an ideal cleanroom process, this would not happen very often because there would not be many errors, but it does seem to be a potential flaw. The cleanroom process intentionally keeps the certification team separate from the development team to promote "black box" testing, but it could add additional time for context switching and learning. For example, the development team has to learn about a specific code area to implement the code, and then the certification team also has to learn about that functional code area to be able to test it. This duplication of learning effort could be saved if the development team was also responsible for testing the code they developed.

An idea to combat this duplication of learning is the process of test-driven development. Test driven development has developers write small tests, watch them fail and then implement the code that makes the tests pass [8]. This ensures simple design by only implementing what is required to make each test pass. By writing incremental tests, you also ensure test coverage and generate documentation (in the form of test case names) to make it easy to remember what code was for if you refer to it later. The cleanroom response to this argument would be that if you never have errors after development is complete, then there is no need to return to documentation or even the code itself after the software is released. The only time you may need to return to the code, and documentation, is if you were to add a new feature to an existing piece of software.

Lastly, the specification process seems extensive and although it is not waterfall, it seems to stack a lot of design at the start. The specification team defines increments of work and box diagrams for how each component interacts with the next. These diagrams are hierarchical and are defined at a high level first before planning each individual component's functional requirements. However, it seems like a lot of up-front work is required. The upfront design work required by the specification team seems like a detriment to developers because they lose some of the freedom to create their own design and define the way each component acts with the next. On the other hand, this may be a positive because the rigid structure of the specification ensures each requirement will be met. This rigidity is beneficial because it reduces errors and adds reliability to the software, but the process may be cumbersome if major changes were required. For example, if the customer changed their mind late in the development process, new requirements and specifications would need to be made. Cleanroom tries to address this by getting feedback from the customer frequently in the specification process, and by designing components to be independent from each other. Despite these mitigation strategies, the emphasis on up front design seems like it might cause delays if major changes were needed.

Waterfall Design Process: Developed by Winston Royce in 1970, this process involves a sequential flow through phases (requirements, design, implementation, verification, and maintenance). It is rigid and requires completion of one phase before moving to the next [10].

Conclusion and Future Research

Overall, the cleanroom development process has shown that more reliable and error-free software is possible. Unlike in other engineering fields, the math first based approach of cleanroom development has not been widely adopted. However, since the software engineering field is relatively young it seems likely that an approach similar to cleanroom will be adopted in the future as the demand for more reliable software increases. Today bugs are thought of as common place, but if higher quality software is developed that is free from errors, customers will begin to realize the importance of reliable software and insist on it from companies.

One area for future research would be to compare using cleanroom against other development methods on the same project. Previous research says that the time to develop a project using cleanroom is reduced because of the decrease of debugging and rework time. However, it seems that the specification time of the design process could be longer than with other development processes. Measuring the time for each section of the development process could be useful. It would be interesting

to run an experiment with multiple teams of similarly skilled developers. To test the effectiveness of cleanroom, some of the teams would develop using cleanroom and the others would use other development processes. Then the experiment would measure the time of each phase (design, specification, testing, development and debugging) to compare between processes. Furthermore, it would be interesting to study the long-term appearance of bugs once the completed software is in use. This comparison would provide a more accurate comparison of the effectiveness of cleanroom. Previous studies use cleanroom but often do not have a baseline to compare against for the unique project that they are working on. This direct comparison experiment would be interesting given that the project would be consistent across teams. Refer to figure 2, to see a flow diagram of the comparison project and key points of measurement.

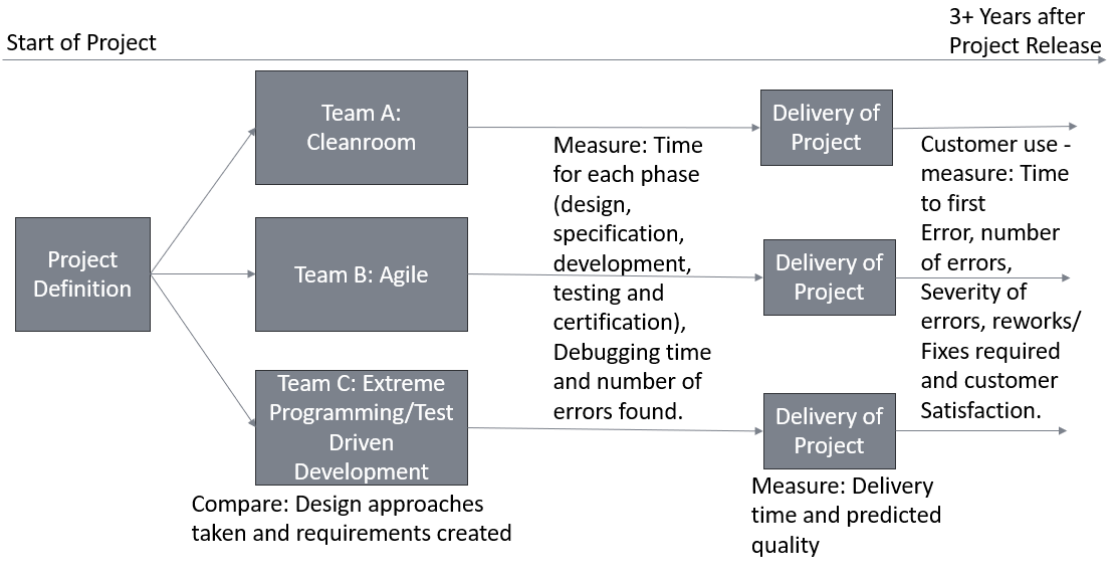

Figure 2: Diagram of research project comparing development design approaches.

Another area for potential future research would be automating function specification and correctness proofs. The theoretical mathematical specification is often cited as one of the reasons why cleanroom is not more widely adopted [5]. If there was some way to make the specification or proof portion easier by automating it or building a tool to handle some of that effort it might make the barrier to adoption lower for cleanroom.

In conclusion, there are further areas of study that could be conducted to convince developers to embrace the cleanroom process. Cleanroom is a proven process for eliminating critical errors in code, and it would be a useful tool for enhancing the reliability of software across the industry.

References

[1] R. C. Linger, "Cleanroom process model," *IEEE Software,* pp. 50-58, March 1994.

[2] P. A. Hausler, R. C. Linter and C. J. Trammell, "Adopting Cleanroom software engineering with a phased approach," *IBM Systems Journal,* 1994.

[3] R. C. Linger and C. J. Trammell, Cleanroom Software Engineering Refrence Model, Pittsburgh: Software Engineering Institute , 1996.

[4] H. D. Mills, "Certifying the correctness of software," *IEEE,* vol. 2, pp. 373-381, 1992.

[5] J. Henderson, "Why Isn't Cleanroom the Universal Software Development MEthodology," *Loral Space Information Systems .*

[6] K. Beck, J. Grenning and R. Martin, "Principles behind the Agile Manifesto," 2001. [Online]. Available: https://agilemanifesto.org/principles.html. [Accessed 17 Febuary 2024].

[7] S. W. Sherer, P. G. Arnold and A. Kouchakdjian, "Experience Using Cleanroom Software Engineering in the US Army".

[8] L. Copeland, "Extreme Programming," *Computerworld,* 2001.

[9] D. Parnas, "On the Criteria To Be Used in Decomposing Systems into Modules," *Association for Computing Machinery, Inc.,* p. Volume 15, December 1972.

[10] Atlassian, "Waterfall Methodology: A Comprehensive Guide," Atlassian, [Online]. Available: https://www.atlassian.com/agile/project-management/waterfall-methodology. [Accessed 17 February 2024].