Framework for Improvement in Cleanroom Software Engineering

Thesis

Submitted in the partial fulfillment of requirements for the award of the degree of

Master of Engineering

in

Software Engineering

By:

Nupur Chugh

(80731016)

Under the supervision of

Mrs. Shivani Goel

Lecturer, CSED,

Thapar University, Patiala

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT

THAPAR UNIVERSITY

PATIALA-147004

JUNE 2009

CERTIFICATE

I hereby certify that the work which is being presented in the thesis entitled, **"Framework for Improvement in Cleanroom Software Engineering"**, in partial fulfillment of the requirements for the award of degree of Master of Engineering in Software Engineering submitted in Computer Science and Engineering Department of Thapar University, Patiala, is an authentic record of my own work carried out under the supervision of **Mrs Shivani Goel** and refers other researchers' works which are duly listed in the reference section.

The matter presented in this thesis has not been submitted for the award of any other degree of this or any other university.

Nupur

(Nupur Chugh)

(Roll No. 80731016)

This is to certify that the above statement made by the candidate is correct and true to the best of my knowledge.

Shivani Goel

(Mrs. Shivani Goel)

Computer Science and Engineering Department

Thapar University,

Patiala.

Countersigned by:

24/06/09.

(Dr. Rajesh Kumar Bhatia)

Assistant Professor & Head

Computer Science & Engineering Department,

Thapar University,

Patiala.

R.K. Sharma 24/6/09

(Dr. R.K. Sharma)

Dean (Academic Affairs)

Thapar University,

Patiala.

ACKNOWLEDGEMENT

No volume of words is enough to express my gratitude towards my guide **Mrs. Shivani Goel,** Lecturer (CSED), Thapar University, who has been very concerned and has aided for all the material essential for the preparation of this thesis. She has helped me explore this vast topic in an organized manner and provided me with all the ideas on how to work towards a research-oriented venture.

I am thankful to **Dr. Rajesh Kumar Bhatia**, Head, Computer Science & Engineering Department, TU, Patiala, for the motivation and inspiration that triggered me for the thesis work. I would also like to thank the entire staff member and my colleagues who were always there at the need of the hour and provided with all the help and facilities, which I required for the completion of the thesis.

Most importantly, I would like to thank my parents and the almighty for showing me the right direction out of the blue, to help me stay calm in the oddest of the times and keep moving even at times when there is no hope.

Nupur Chugh
(80731016)
M.E. (Software Engineering)-2nd year
Computer Science & Engineering Department
Thapar University
Patiala –147004.

ABSTRACT

Cleanroom Software Engineering is a theory based team-oriented engineering process for developing very high quality software under statistical quality control. The Cleanroom process combines formal methods of object-based box structure specification and design, function theoretic correctness verification, and statistical usage testing for reliability certification on to produce software approaching zero defects. Cleanroom Software Engineering is a management and technical process that produces high quality software. The focus of Cleanroom Software Engineering involves moving from traditional, craft-based software development practices to rigorous, engineering-based practices. Cleanroom Software Engineering yields software that is correct by mathematically sound design, and software that is certified by statistically valid testing. In this report detail description of Cleanroom Software Engineering and Object Oriented Technology is given.

After studying the advantages and flaws of Cleanroom Software Engineering two new steps have been introduced in the existing Cleanroom Software Engineering. This report discusses the Collaborative Engineering and Mutation Testing that have been introduced to improve the Cleanroom Software Engineering. A Comparative study of Cleanroom Software Engineering and Object Oriented Technology is done and summarized in a table.

Table of Contents

.

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Cleanroom Software Engineering

Cleanroom software engineering is an engineering and managerial process for the development of high-quality software with certified reliability. The focus of Cleanroom Software Engineering involves moving from traditional, craft-based software development practices to rigorous, engineering-based practices. By detecting errors as early as possible, Cleanroom Software Engineering reduces the cost of errors during development and the probability of failures during operation; thus the overall life cycle cost of software developed under Cleanroom Software Engineering is reduced. A principal objective of the Cleanroom process is development of software that exhibits zero failures in use. The Cleanroom name is borrowed from hardware. The theoretical foundations of Cleanroom were established in the late 1970s and early 1980s, when Harlan Mills, an accomplished mathematician and IBM Fellow, related fundamental ideas in mathematics, statistics, and engineering to software. Cleanroom Software Engineering, with their emphasis on rigorous engineering discipline and focus on defect prevention rather than defect removal. Cleanroom Software Engineering combines mathematically based methods of software specification, design, and correctness verification with statistical, usage-based testing to certify software fitness for use. Cleanroom Approach is language, environment, and application-independent, and has been used to develop and evolve a variety of systems, including real-time, embedded, host, distributed, workstation, client-server, and microcode systems. Cleanroom Approach supports prototyping and object-oriented development, and enables reuse through precise definition of common services and component functional semantics, and certification of component reliability [1].

1.1 Cleanroom Technologies

Cleanroom software engineering is characterized by principal technologies: incremental development under statistical process control; precise specification and design; correctness verification; statistical testing and software certification; and software

reengineering. These technologies can be used separately or together, and can be introduced in any order to improve software practice.

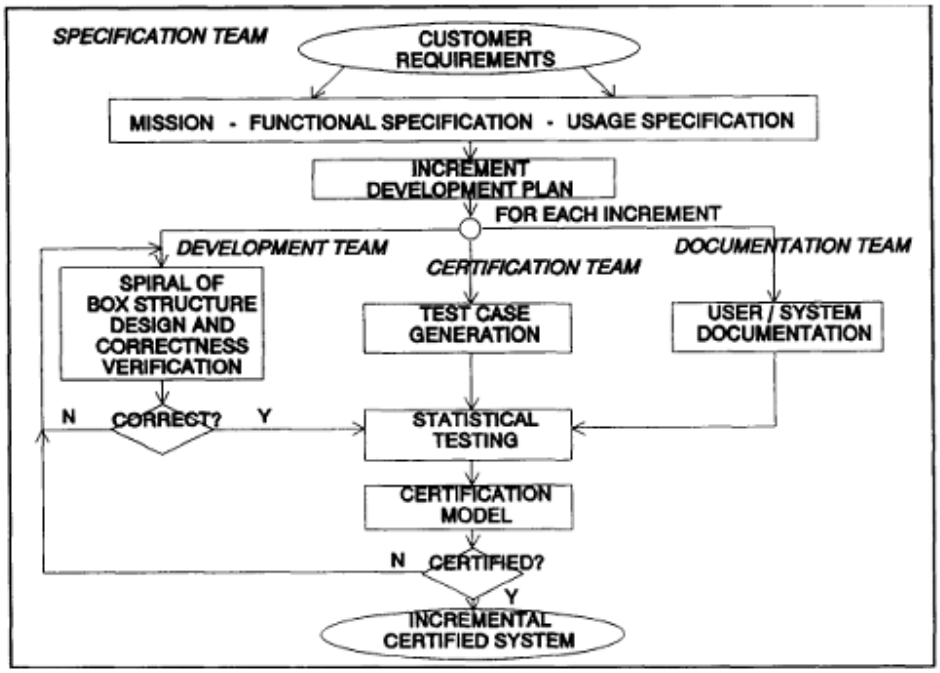

Figure 1.1 Cleanroom Development Process [18]

1.1.1 Incremental Development under Statistical Process Control

Cleanroom approach is based on development and certification of a pipeline of user-function software increments that accumulate into the final product. Incremental development enables early and continual quality assessment and user feedback, and facilitates process improvements as development progresses. The incremental approach avoids risks inherent in component integration late in the development cycle. Incremental development permits systematic management and incorporation of requirements changes over the development cycle. Incremental development as practiced in the Cleanroom process is based on the principle of referential transparency. Referential transparency means that the only thing that matters about an expression is its value and any sub expression can be replaced by any other that is equal in value. 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 [1]. When this property holds, a design can be shown to be correct with respect to its specification. In practice, the requirement for

referential transparency places constraints on the functional content and order of design decomposition of a software system. User functions are organized for development into a sequence of verifiable and executable software increments, each providing additional function. The functional content of the increments is defined such that they accumulate into the complete set of functions required for the final system. Architectural requirements and risk avoidance strategies place additional constraints on the increment content. For correctness, each increment must satisfy its parent specification through the functions it provides combined with the sub specifications it contains for future increments. For statistical testing and certification, each increment can contain stubs as placeholders for future increments to permit execution in the system environment. Each new increment replaces stubs in the evolving system and satisfies the sub specifications associated with it. In this way, referential transparency is maintained throughout the system development.

1.1.2 Precise Specification and Design

A key Cleanroom principle is that programs can be regarded as rules for mathematical functions (or relations). That is, programs carry out transformations from input to output that can be precisely specified as function mappings. Programs can be designed by decomposing their function specifications, and can be verified by abstracting and comparing their designed functions to their function specifications for equivalence. This concept is scale-free, with application ranging from large specifications for entire systems down to individual control structures, and to every intermediate decomposition and verification along the way. Three special types of mathematical functions are important in Cleanroom development because of their correspondence to useful system views, and their interrelationships in a stepwise decomposition and verification process. These functions are represented using Black box, State box and Clear box, collectively called Box Structures and are used by the specification team. Box structures map system stimuli and the stimulus histories into responses.

Black box: A black box specifies the external behavior of a system or system component. The transition function is:

((current stimulus, stimulus history) --> response) [1].

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. A black box definition is state-free and procedure-free, referencing only external, user- observable stimuli and responses. Black box definitions are often given in tables with columns for current stimulus, conditions on history, and responses.

State box: A state box is derived from and verified against a corresponding black box. The state box transition function is:

((current stimulus, current state) --> (response, new state)) [1].

That is, a state box maps the current stimulus and the current state into a response and a new state. In the state box, the stimulus history of the black box is replaced by retained state data necessary to achieve black box behavior. A state box definition is procedure-free, and isolates and focuses on state invention.

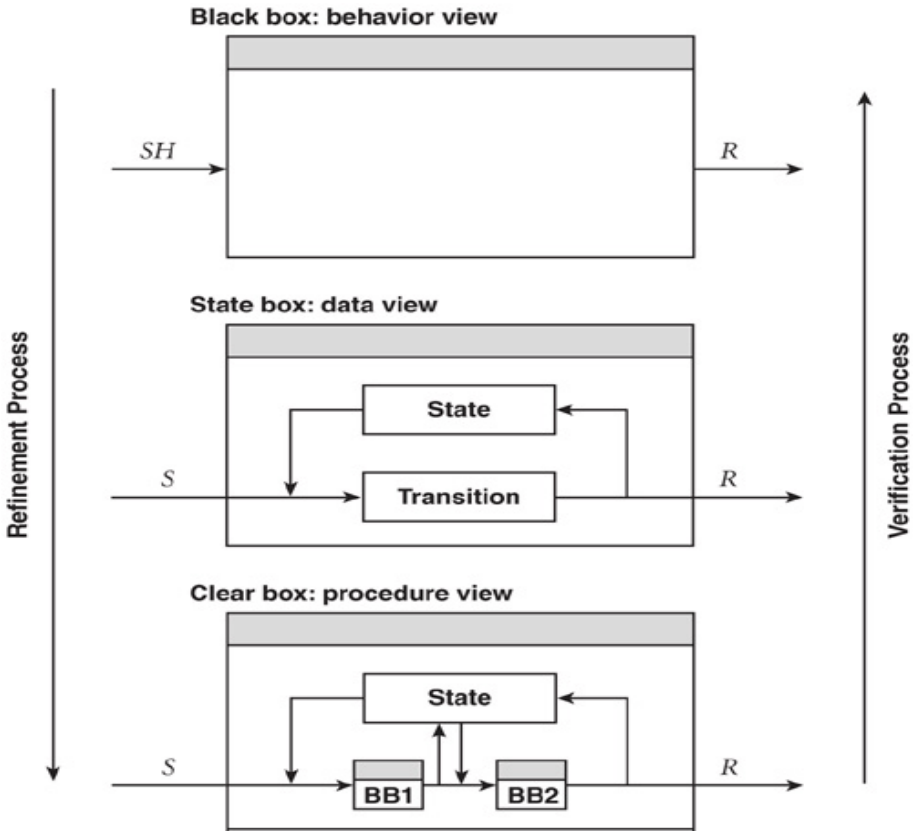

Figure 1.2 Box structure refinement and verification. BB = black box [2]

Clear box: A clear box is derived from and verified against a corresponding state box. The clear box transition function is:

((current stimulus, current state) --> (response, new state)) by procedures [1].

In the clear box, the procedures required to implement the state box transition function are defined, possibly introducing new black boxes for further decomposition into state and clear boxes. That is, a clear box is a program, or set of programs, that implements the state box and introduces and connects operations in a program structure for decomposition at the next level. Such connections are critical to maintaining intellectual control in large-scale software development.

A sequence structure is shown in Figure 1.2. This figure represents the relation between various Box Structures. The process is refined from black box to state box to clear box structures. The verification process takes reverse direction of the refinement process.

1.1.3 Correctness Verification

All Cleanroom-developed software is subject to function-theoretic correctness verification by the development team prior to release to the certification test team. The function-theoretic approach permits development teams to completely verify the correctness of software with respect to specifications. A correctness theorem defines conditions to be met for achieving correct software. These conditions are verified in mental or verbal proofs of correctness in development team reviews. Programs contain an infinite number of paths that cannot all be checked by path-based inspections or software testing. However, the correctness theorem is based on verifying individual control structures rather than tracing paths. Because programs contain a finite number of control structures, 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 is a major factor in the quality improvements achieved by Cleanroom teams.

1.1.4 Statistical Testing and Software Certification

The set of possible executions of a software system is an infinite population. All testing is really sampling from that infinite population. No testing process, no matter how

extensive, can sample more than a minute fraction of all possible executions of a software system. 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 of a software system produces scientific measures of product and process quality for management decision-making, just as has been done in hardware engineering for decades. The objective of the certification team is to provide scientific certification of software fitness for use, not to "test in" quality. The certification team creates usage models, which define all possible scenarios of use of the software, together with their probabilities of occurrence. 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. Usage can be defined in Markov models that permit substantial management analysis and simulation of test operations prior to software development, as well as automatic test case generation. Other methods may also be used to implement the Cleanroom principles of statistically based software testing and certification [1]. 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. Objective statistical measures of software reliability and fitness for use can be computed based on test results for informed management decision-making. Because statistical usage testing tends to detect errors with high failure rates, it is an efficient approach to improving software reliability.

1.1.5 Software Reengineering

The purpose of the Software Reengineering Process is to prepare reused software for incorporation into the software product. Non-Cleanroom-developed software can be incorporated into Cleanroom projects. Such software may require reengineering to enable developers to maintain intellectual control and to achieve Cleanroom levels of quality and reliability. Software may be reused as is, reused through interface controllers such as wrappers, or reused after reengineering.

Reused software must satisfy two principal Cleanroom requirements. First, the functional semantics and interface syntax of reused software must be understood and documented, to maintain intellectual control and to avoid unforeseen failures in execution. If

specification and design documentation for reused software is incomplete, its functional semantics can be recovered through function abstraction and correctness verification. The completeness and correctness of specifications for reused software must satisfy project specification standards.

1.2 Cleanroom Software Engineering Process

Cleanroom Software Engineering process comprises of four different processes:

- Management
- Specification
- Development
- Certification

A separate team is required for each of these processes to ensure the highest quality product.

1.2.1 Cleanroom Management Process

All the issues related to management during Cleanroom Software Engineering are divided into following subtasks:

i. **Project Planning:** A plan is prepared:
   - To tailor the Cleanroom Software Engineering processes for the project
   - To define and document plans for the Cleanroom project, and
   - To review the plans with customer, the project team, and peer groups.

   The need for this process is when there exists a new or revised statement or requirements; or when software development plan needs to be revised or when a new software increment begins.

ii. **Project Management:** This process is guided by the software development plan
   - To manage interaction with the customer and peer organizations.
   - To establish and train Cleanroom teams.
   - To initiate tracking
   - To control planned Cleanroom processes

- To eliminate or reduce risks, revise plans as necessary to accommodate changes and actual results, and continually improve Cleanroom team performance [15].

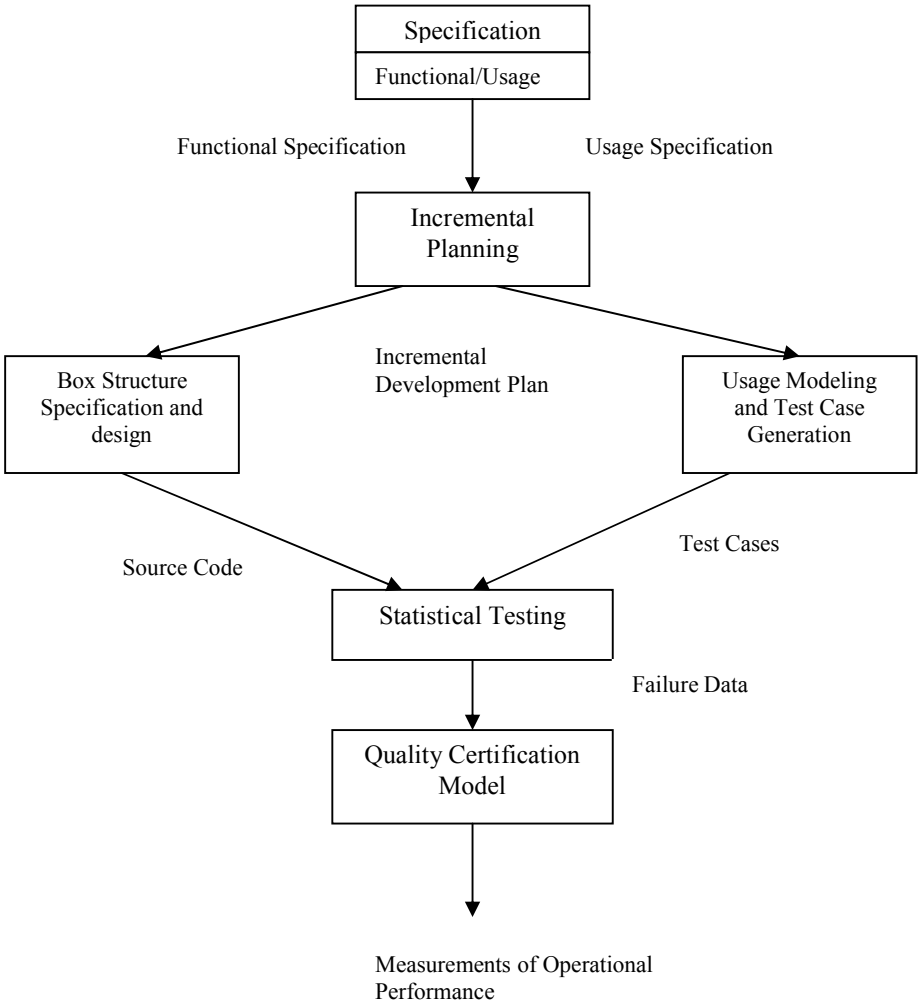

Figure 1.3 Cleanroom Software Engineering Process [3]

iii. **Performance Improvement:** This process is required:

- 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.
- To evaluate and introduce appropriate new technologies and processes.

iv. **Engineering Change:** This process is required 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 [4].

1.2.2 Cleanroom Specification Process

After planning, requirements are specified. The specification process in Cleanroom Software Engineering Process involves following processes:

i. **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.

ii. **Function Specification:** Specification in terms of functions is required 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.

iii. **Usage Specification:** Function specification alone may not cover all aspects of system requirements, so specification is done from usage point of view:

- 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
- To obtain agreement with the customer on the specified usage as the basis for software certification.

iv. **Architecture Specification**: The purpose here 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.

v. **Increment Planning:** All the requirements of function specification may not be fully incorporated into 1 increment so Cleanroom Software Engineering process plans for increments:

- 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.

1.2.3 Cleanroom Development Process

i. **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.

ii. **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.

iii. **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 recertified.

1.2.4. Cleanroom Certification Process

i. **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.

ii. **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 [22].

1.3 Object Oriented Technology

Object Oriented is a software development technology that focuses on objects. Objects model real world objects that are found in everyday life. An object can be compared to a

"black box" at the software level – it sends and receives messages. The object contains both code and data. A major concept in Object Oriented technology is the "class." Grady Booch defines a "class" as "a set of objects that share a common structure and a common behavior" [7]. Class defines abstract characteristics of object along with its attributes and operations.

1.3.1 Principles of Object Oriented Technology

The principles of Object Oriented Technology are as follows:

Abstraction: Abstraction refers to the act of representing essential features without including background details. "An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer"[7]. For example, a class Car would be made up of an Engine, Gearbox, Steering objects, and many more components. To build the Car class, one does not need to know how the different components work internally.

Encapsulation: The process of wrapping data into a single unit (class) is called encapsulation. It hides the design details from the implementation. Encapsulation can be described as "the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse". Encapsulation is generally achieved through information hiding, which is the process of hiding the aspects of an object that are not essential for the user to see. It is a technique for minimizing interdependencies among separately written modules.

Modularity: The process of partitioning a program into logically separated and defined components that possess defined interactions and limited access to data. Booch defines modularity as "property of a system that has been decomposed into a set of cohesive and loosely coupled modules"[7].

Inheritance: The process of inheriting all the existing attributes and operations of the other class. The class, which inherits the attributes, is called "subclass" (child class) and classes whose attributes are inherited are called "superclass" (parent class). One author

puts it this way, “Inheritance is a relationship among classes where a child class can share the structure and operations of a parent class and adapt it for its own use”[16]. Inheritance can be single inheritance or multiple inheritance. In single inheritance, the sub-class inherits the attributes and operations from a single superclass. In multiple inheritance, the sub-class inherits some attributes from one class and others from another class.

Polymorphism: Polymorphism comes from the Greek meaning “many forms.” It is the ability of objects belonging to different data types to respond to method calls of method of the same name. This allows different underlying implementations for the same command. For example, assume there exists a vehicle class that includes a steer-left command. If a boat object was created from the vehicle class, the steer-left command would be implemented by a push to the right on a tiller. However, if a car object was created from the same class, it might use a counter-clockwise rotation to achieve the same command.

Persistence: It is “the property of an object through which its existence transcends time (i.e., the object continues to exist after its creator ceases to exist) and/or space (i.e., the object’s locations moves from the address space in which it was created)”.

1.4 Motivation and Objective

Cleanroom Software Engineering is a theory based and team oriented process and used in development methodology? “The basic reason is the prohibition on unit testing is many projects. The Question arises “why isn’t Cleanroom the standard software too radical for most developers. The use of Mutation Testing and inclusion of some of the principles of Object Oriented Technology can solve the problem.

1.5 Organization of Thesis

Chapter 2 presents the Existing Cleanroom Software Engineering. The work that has already been done in this area is discussed in this chapter.

Chapter 3 describes the Object Oriented Technology and its features.

Chapter 4 describes the Problem Statement. This chapter draws a line of separation between our work and existing work and proper justification is given for the unanswered question.

Chapter 5 describes the Proposed Steps and a comparative study of Cleanroom Software Engineering and Object Oriented Technology.

Finally, Chapter 6 concludes the thesis work, includes summary of our contribution along with future scope.

CHAPTER 2

EXISTING CLEANROOM SOFTWARE ENGINEERING APPROACHES

Cleanroom development begins with a specification of required system functions. Without rigorous specification technology, it is difficult to devote time and effort to the specification process. In a box structure, it is important to define specifications correctly. Therefore there is a need to translate the natural language to a formal specification. Formal Methods are the techniques that are to be used at the specification phase to write all the user's requirements in a logical and mathematical language.

2.1 Formal Methods

Formal methods use mathematical and logical formalizations to find defects early in the software development lifecycle. Formal Methods is the use of ideas and techniques from mathematics and formal logic to specify and reason about computing systems to increase design assurance and eliminate defects. Formal Methods tools allow comprehensive analysis of requirements and design and complete exploration of system behavior, including fault conditions. Formal Methods provides a disciplined approach to analyzing complex safety critical system. Formal methods use a mathematical and logical formalization to prove that key properties of the system satisfy the expected behavior of the software system. Formal methods allow a software engineer to create a specification that is more complete, consistent, and unambiguous than those produced using conventional methods [4]. It reduces specification errors dramatically and, as a consequence; serve as the basis for software that has very few errors. Characteristics of formal methods include:

- Formal methods check the consistency of the system's descriptions. They make sure properties that the system analysts have defined meet the requirements of the system. Formal methods make it possible to find defects in the system early in the software lifecycle. Due to early defect detection the correct implementation through consistent requirements is possible.

- Formal methods avoid more testing. After applying these methods to high quality software systems, they find defects that may go undetected after extensive testing.
- Formal methods use mathematical notations to formalize the system's descriptions.
- Formal methods can be present in all phases of the software project. The software project manager decides when these methods should be used in the analysis, design and development phases, to detect more defects.

The step in the application of formal methods is to define the data invariant, state, and operations for a system function.

Data invariant: The data invariant is a condition that is true throughout the execution of a function that contains a collection of data.

State: The state is a stored data that function accesses and alters.

Operations: The operations are actions that take place in a system as it reads or writes data to a state.

	FSM	**VDM**	**VDM/CCS**	**Total Informal**	**Formal**
Number of faults discovered	43	184	11	238	487
Number of modules with this design type	77	352	83	512	692
Number of faults normalized by number of modules	0.56	0.52	0.13	0.46	0.70

Table 2.1 show faults discovered during unit testing for the delivered code based on different formal methods design types [4].

Source	Language	Failure per KLOC	Formal methods used?
Siemens operating system	Assembly	6-15	No
NAG scientific libraries	Fortran	3.00	No
CDIS air-traffic-control support	C	0.81	Yes
Lloyd's language parser	C	1.40	Yes
IBM Cleanroom development	Various	3.40	Partly
IBM normal development	Various	30.0	No
Satellite planning study	Fortran	6-16	No
Unisys communication software	Ada	2-9	No

Table 2.2 compares the failure rate of projects that used formal methods and those that did not use formal methods [4].

The benefits of having formal specifications are:

- Unambiguous language in comparison to natural language.
- Logic is able to define statements that consider all possible input values. This is significantly better than unit tests, which are usually able to test just a small subset of input data.
- Design choices can be formally verified before any implementation.
- Correctness verification can be done automatically through theorem proves and model checkers.
- Changes in software specifications can be handled more easily.

2.2 System Design

Cleanroom Software Engineering uses Box Structure to represent the data, state and transformation, which follows usage hierarchy.

2.2.1 Box Structures

It embodies important concepts of data encapsulation and information hiding. These structures exhibit identical external behavior but increasing internal visibility. Box structures are descriptions of functions that exhibit properties essential for effective system specification and design. Box structures separate three aspects of system development (specification of behavior, data, and procedures) yet relate them in a coherent process of refinement and verification. The box structures used in Cleanroom Software Engineering are:

Black box: The Black box specifies the behavior of a system or a part of a system. The system responds to specific stimuli (events) by applying a set of transition rules that map the stimuli to response [12]. The black box is the most abstract description of system behavior and can be considered as a requirements specification for a system or system part. A black box specification describes an abstraction, stimuli, and response using the notation as shown in figure 2.1.

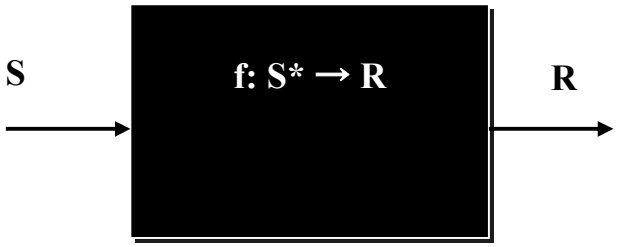

Figure 2.1 A Black box specification [5].

State box: The State box encapsulates state data and services (operations). The state box includes a designed state and an internal black box that transforms the stimulus and an initial state into the response and **a** new state. The required state is designed from an analysis of stimulus histories and responses for the system or system part. The state box incorporates a black box as shown in figure 2.2.

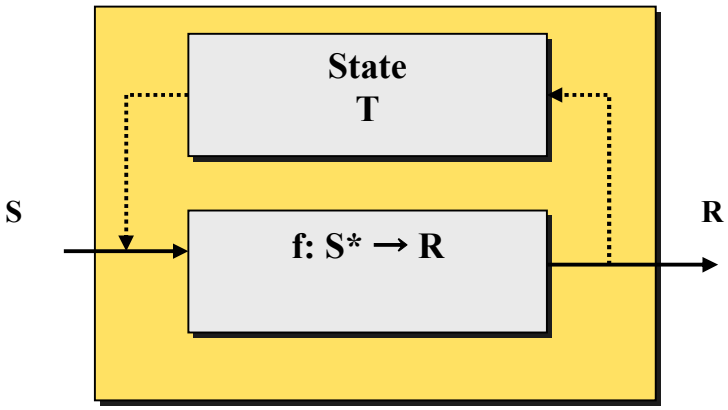

Figure 2.2 A State box specification [5].

Clear box: The clear box replaces the internal black box of the state box with the designed sequential or concurrent usage of other black boxes as subsystems. These new black boxes are in turn expanded at the next level of the system box structure usage hierarchy into state [12]. Consider the clear box as shown in figure 2.3. The Black box, g is replaced by a sequence construct that incorporates a condition.

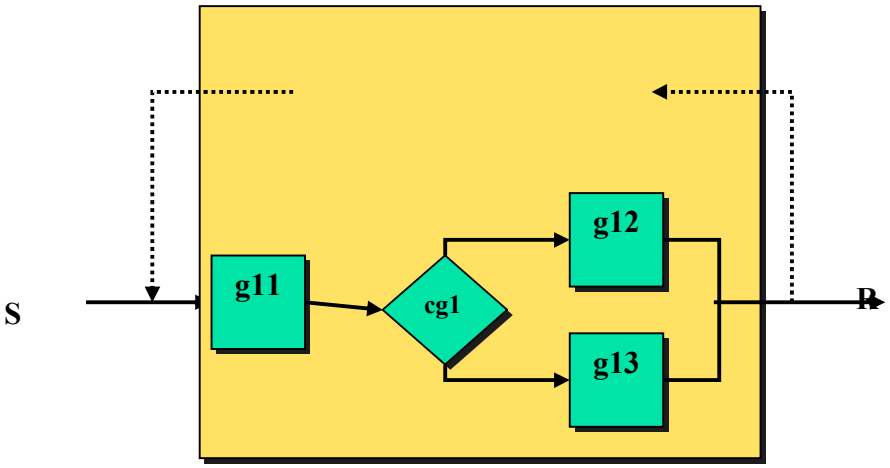

Figure 2.3 A Clear box specification [5].

The steps for box structure development are as follows:

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

2.3 Tool Support

Cleanroom Software Engineering has good automated tool support.

Hevner et al have categorized the case tools necessary for effective support of the Cleanroom System Development [10]. Case Tools for Cleanroom System Development on the basis of Processes are:

Requirement Determination

- Intelligent Editor
- Box Structure Graphics
- Decomposition Tool
- Usage Specification Tool

Increment Development

- Spreadsheets
- Scheduling Tools
- Decision Support Systems

Box Structure Analysis

- Box Structure Graphics
- Prototyping Tool
- Application Generator

- Performance Evaluation Tools

Statistical Testing

- Random Case Generator
- Statistical Packages
- CleanTest

CleanTest is a tool to support the statistical testing process used in Cleanroom software engineering. It was developed by Cleanroom Software Engineering, Inc., a company whose entire business is helping organizations use Cleanroom techniques. CleanTest solves the problem of recording and maintaining operational profiles (usage specifications) with an easy-to-use, graphical user interface [11]. CleanTest can keep track of what parts of the operational profile have been tested, a valuable aid to effective testing.

2.4 Statistical Usage Testing

Cleanroom Software Engineering uses Statistical Usage testing. In statistical testing, a model is developed to characterize the population of uses of the software, and the model is used to generate a statistically correct sample of all uses of the software. Usage-based testing implies a focus on detecting the faults that cause the most frequent failures, hence maximizing the growth in reliability. Software 'usage model' characterizes the population of intended uses of the software in the intended environment. Statistical usage testing is a black-box test technique that is based on the intended usage of the software. Several usage profiles may be set up, which means that several different user groups are considered [1]. Statistical testing based on a software usage model ensures that the failures that will occur most frequently in operational use will be found early in the testing cycle [6]. Statistical Usage Testing is used to certify the reliability of the software. Following are the steps for Statistical Usage Testing:

2.4.1 Method For Creating a Usage Model

1. Review and clarify the software specification.
2. Identify expected users of the software, expected uses of the software, and Expected system environment.

3. Define a stratification of user and environment parameters.
4. Determine the desired levels of usage model granularity.
5. Iteratively develop usage model structure.
6. Verify the correctness of the structure against the specification.
7. Iteratively develop a probability distribution for the model.
8. Verify the correctness of the probability distribution against any information available concerning intended usage of the software.

2.4.2 Use of the Usage Model

The following steps define the procedure for statistical test planning and software certification:

1. Perform cost–benefit analyses of the usage distribution with respect to the defined test environment.
2. Plan and implement the defined system environment.
3. Determine stopping criteria for testing.
4. Generate random test cases from the usage model. Where the test case includes a class of input variables, randomly generate input stimuli for the test case from the class of input variables.
5. Establish how each event of each test case will be evaluated to determine that it meets the specification.
6. Perform statistical testing of the software. Record any observed failure (i.e. any deviation from the specifications).
7. Analyze the test results using a reliability model to provide a basis for statistical inference of reliability of the software during operational use.

Usage modeling has been demonstrated to be an activity that improves the specification, gives an analytical description of the specification, quantifies the testing costs and, with statistical testing, provides a basis from which inferences of software reliability can be made. Statistical testing is based on a probabilistic generation of test data: structural or functional criteria serve as guides for defining an input profile and a test size. Previous work has confirmed the high fault revealing power. One of the engineering techniques

emphasized in Cleanroom is Statistical Usage Testing, which is a method for statistical control of the software reliability during the system or acceptance testing phase [6].

The example of calculator program is discussed:

Suppose inputs to a calculator program are:

	Input	Percentage	Number
A1	1st operand (correct)	22	0 – 21
A2	1st operand (incorrect)	3	22 – 24
B1	2nd operand (correct)	22	25 – 46
B2	2nd operand (incorrect)	3	47 – 49
C1	Calculation symbol (correct)	22	50 – 71
C2	Calculation symbol (incorrect)	3	72 – 74
D1	Equal symbol (correct)	22	75 – 96
D2	Equal symbol (incorrect)	3	97 – 99

Table 2.3 Inputs to Calculator Program [6]

Sequences of usage test cases that conform to the usage probability distribution are generated.

A series of random numbers are generated between 0 and 99 that corresponds to the probability of stimuli occurrence.

For example, the following random number sequences are generated:

14 – 95 – 26 – 44 – 45 – 76: A1; D1; B1; B1; B1; D1

81 – 19 – 31 – 69 – 45 – 9

38 – 21 – 52 – 84 – 86 – 97

The testing team executes the test cases noted above (and others) and verifies software behavior against the specification for the system.

For example for the test case

T1: A1; D1; B1; B1; B1; D1

The input sequence is:

1st operand (correct)

Equal symbol (correct)

2nd operand (correct)

2nd operand (correct)

2nd operand (correct)

Equal symbol (correct)

And the output should be: Error

2.4.3 Advantages of Statistical Usage Testing

1. Statistical Usage Testing determines a level of confidence that a software system conforms to a specification.
2. It is able to statistically evaluate and infer the quality of the software system to meet all requirements.
3. Statistical Usage testing is a quantitative approach that is verifiable.

2.4.4 Disadvantages of statistical usage testing

1. Testing is derived from a usage model that must be exhaustive in order select a subset for testing.
2. Statistical testing and verification will be more reliable if it is based on the some history data.
3. It would be effective if it could be integrated with other testing methods.

The Cleanroom Software Engineering and its approaches including Formal methods, Statistical Usage Testing are discussed in this chapter. The tools that are required for Cleanroom Software Engineering are mentioned. In the next Chapter the Object Oriented Technology is discussed in detail corresponding to the Cleanroom Software engineering.

CHAPTER 3

OBJECT ORIENTED TECHNOLOGY

Object oriented technology is a general term that describes tools, processes, and programming languages concerned with the development of systems that consist of chunks of data known as objects. The Object-Oriented process is an evolutionary approach to software engineering. The Object Oriented paradigm focuses on classes that encapsulate data and algorithms for manipulating the data. Object oriented classes promote reusability across applications.

3.1 Use cases

Object Oriented uses informal use case for characterizing usage. A use case in software engineering and systems engineering is a description of a system's behavior as it responds to a request that originates from outside of that system. In other words, a use case describes "who" can do "what" with the system in questionA use case is a specification of actions, including variants, which a system can perform, interacting with an actor of the system. A use case is a specific way of using the system by performing some part of the functionality. A use case instance is a specific sequence of actions as specified in a use case carried out under certain conditions. Use cases offer a way of identifying, through the description of scenarios of system usage, the classes of objects, and their interactions, which must be modeled in the process of system design and development [27]. The five types of use cases are as follows:

1. **Common use cases**: Common parts of use cases are factored out so that these can be (re) used by other use cases without repeating the description.
2. **Variant use cases**: In variant use cases, alternatives to the normal use case behavior are captured. They are also used for exceptions.
3. **Component uses cases**: In component use cases, parts of use cases are further refined leading to a hierarchical decomposition of use cases.
4. **Specialized use cases**: Use cases may be classified in more specialized versions.
5. **Ordered use cases**: Ordered use cases deal with situations where the completion of one use case is required before the following use case can be executed.

3.1.1 Use cases in Software Development Process

Use cases are described in natural language informally or in pseudocode. Use cases are used differently by current object-oriented methodologies. Use cases are mainly used to capture the requirements of a software system from a user-centered viewpoint. Uses cases are combined with domain analysis. Use cases have their impact on all models in the software development process. The use case model is expressed in terms of the domain object model, it is structured by the analysis model, it is realized by the design model, it is implemented by the implementation model, and it is tested by the testing model [28]. This approach to use cases has been incorporated in the Objectory CASE-tool.

The main idea behind use case modelling is to elicit and document requirements by discussing and defining specific contexts of system usage as they are anticipated by the different stakeholders in the requirements engineering process. Use cases can be viewed on different abstraction levels.

Environment Level: At the environment level, the use case is related to the entities external to intended system. On this level, a use case is viewed as an entity representing a usage situation.

Structure Level: At the structure level, the internal structure of a use case is revealed together with its different variants and parts.

Event Level: The event level represents a lower abstraction level where the individual events are characterized.

The conceptual framework for the presented use case modelling approach and their relations are illustrated in Figure 3. The figure represents the concept relations at the three different levels of abstraction that is, environment level, structure level, and event level described above.

a) **Environment Level**

   Users: The users belong to the intended target system's environment and can be either humans or other software/hardware based systems.

   Service: A service is a package of functional entities (features) offered to the users in order to satisfy one or more goals that the users have.

Actor: An actor represents a set of users that have some common characteristics with respect to why and how they use the target system.

Goals: Goals are objectives that users have when using the services of a target system.

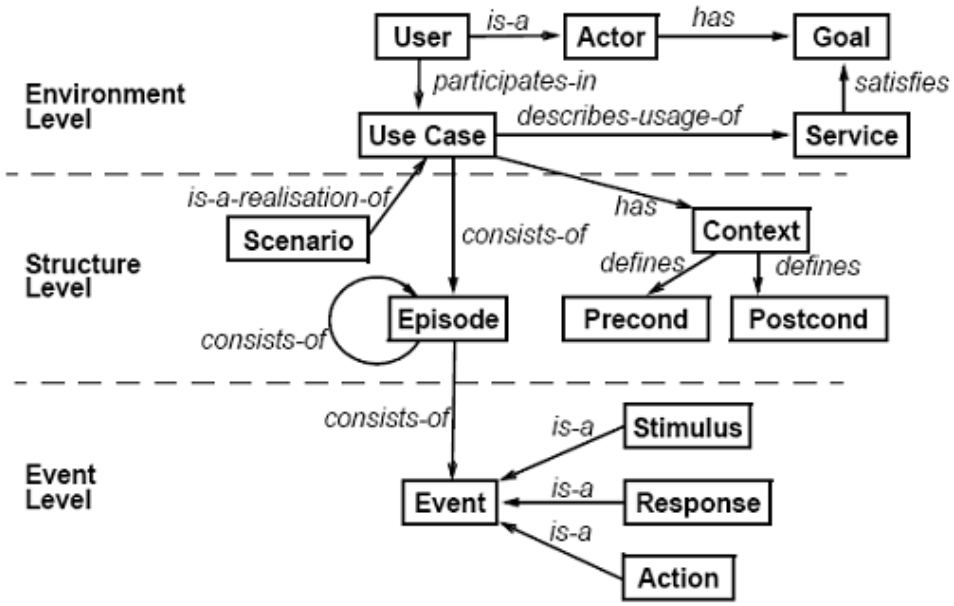

Figure 3.1 Concept relations and levels of abstraction [28].

b) Structure Level

Scenario: A scenario is a specific realization of a use case described as a sequence of a finite number of events. A scenario may either model a successful or an unsuccessful accomplishment of one or more goals.

Context: Context demarcates the scope of the use case and defines its pre-conditions and post-conditions.

Preconditions: Preconditions are the properties of the environment and the target system that need to be fulfilled in order to invoke the use case.

Postconditions: Postconditions are the properties of the environment and the target system at use case termination.

Episode: Episodes are the coherent parts of the use cases and the scenario.

c) **Event Level**

Stimulus: Stimuli are the messages from the user to the target system.

Response: Responses are the messages from the target system to the users.

Actions: Actions are target system intrinsic events, which are atomic in the sense that there is no communication between the target system, and the users that participate in the use case.

3.1.2 Use cases in Practice

Use cases have been used in practice to model almost anything within the system and even the business. For example, use cases are used to record the suite of actions between the actor and the system in an abstract manner to merely highlight the actions but with limited detail. Use cases have been documented using activity graphs to conduct business analysis and document functional requirements of the system [27]. Use cases can also be modeled with sequence diagrams to depict business scenarios (or instances). In the figure4.1 use case A includes use case B, use case C and use case D extends use case A.

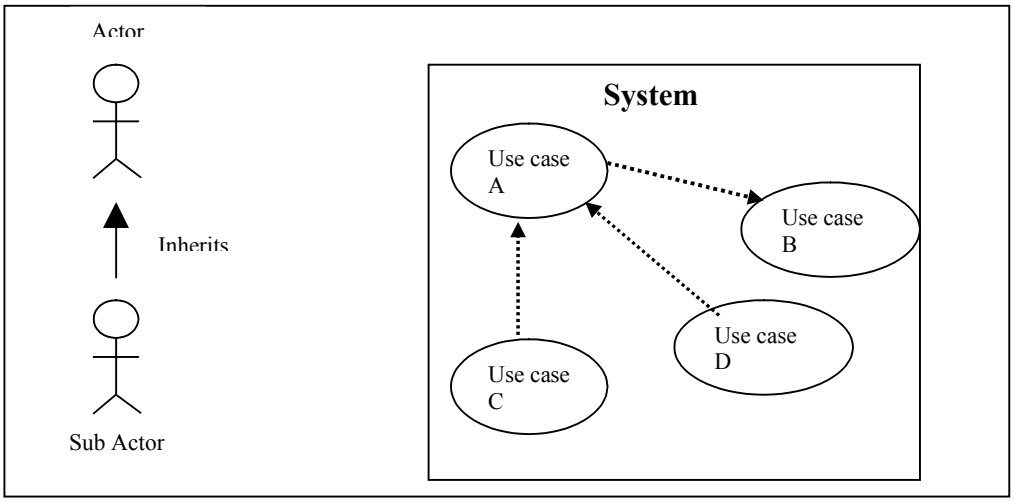

Figure 3.2 Example Use Case Diagram [29]

3.1.3 Advantages of Use Cases

1. Use cases provide the fundamental unit of the functionality required of the system. Therefore, through use cases, the functional requirements can be easily organized.

2. Use case diagramming can be an excellent mechanism for project managers to rank and priorities use cases thereby helping with the creation of the project plan.
3. Actors are the representation of the users as well as the stakeholders of the system. Therefore, actors provide maximum 'buy in' from stakeholders who may be invited to the use case workshops.
4. Use cases are an excellent starting point for business requirements because they start off by 'quizzing' the actors as to how they will use the system. Through use cases one is able to document the requirements that will be of immediate value to the 'real' users of the system.
5. Actors can have inheritance relationships and provide clarity on the use case diagrams, as the subactor need not be related with the use cases with which the actor itself is related. Through inheritance on the use case diagram the number of communication or association lines between the actors and the use cases is reduced.
6. Use cases provide opportunities for reuse of requirements as they enable inclusion and extension of other uses cases. This feature of use cases on the use case diagrams can be used in practice by identifying and modeling reusable requirements separately and then including them in the main use cases.

3.1.4 Limitations of Use Cases

1. Use case diagrams do not have a 'flow'. The use cases are related to each other through the <<include>> and <<extend>> relationship. These relationships themselves are not semantically strong and they can lead to more confusion and discussion than adding value to the model. In practical work, it is recommended that the relationships between use cases be minimally used.
2. Use cases were meant to be object oriented. However, the way they have been used in practical business analysis work is not object oriented at all. This needs to be kept in mind in their use in practice.
3. Use cases do not provide the entire suite of requirements as the focus is on interaction. However, creating another set of requirements document creates confusion and challenges in terms of maintaining the requirements. Hence aspects

of requirements such as business rules, mathematical formulae, printing and even user interfaces do not fit in properly with use cases. It is suggested that for each project the use case standard be discussed and modified to suit the requirements of the project.

4. Use cases are not a great way of modeling embedded systems, as there can be significant system behavior that does not have an actor available for such systems.

3.2 Inheritance

Object oriented uses inheritance hierarchy in form of classes. Inheritance is the concept of a child class (sub class) automatically inheriting the variables and methods defined in its parent class (super class). Inheritance is a way to form new classes (instances of which are called objects) using classes that have already been defined. The new classes, known as derived classes, take over (or inherit) attributes and behavior of the pre-existing classes, which are referred to as base classes (or ancestor classes) [17]. It is intended to help reuse existing code with little or no modification. Inheritance is also sometimes called generalization, because the is-a relationships represent a hierarchy between classes of objects. For instance, a "fruit" is a generalization of "apple", "orange", "mango" and many others. One can consider fruit to be an abstraction of apple, orange, etc [16].

3.2.1 Types of Inheritance

Single Inheritance: In "single inheritance," a common form of inheritance, classes have only one base class. Common attribute found in the design of most class hierarchies is that the derived class has a "kind of" relationship with the base class. In the figure, PaperbackBook is a kind of a book.

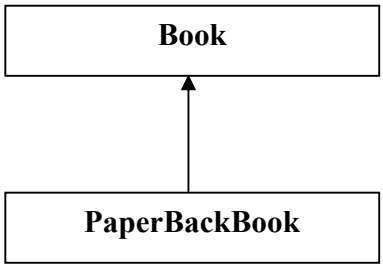

Figure 3.3 Simple Single Inheritance Graph

Multiple Inheritance: Multiple inheritance refers to a feature of some object-oriented programming languages in which a class can inherit behaviors and features from more than one superclass**.** In a multiple-inheritance graph, the derived classes may have a number of direct base classes. Consider the graph in the following figure.

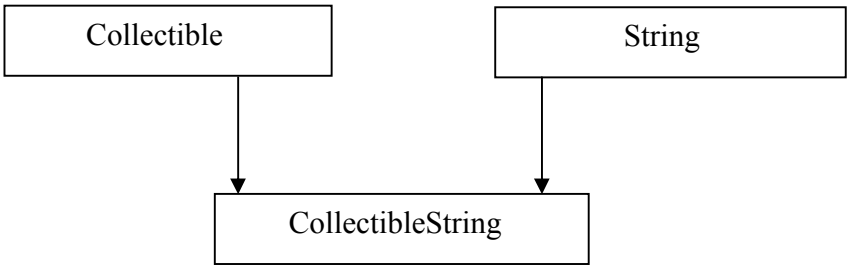

Figure 3.4 Simple Multiple-Inheritance Graph

Multilevel Inheritance: Multilevel inheritance is the mechanism by which one class derives its characteristic from the base class and another class is inherited from the derived class. For instance, a class A serves as a base class for class B which in turn serves as base class for another class C. The class B. that forms the link between the classes A and C is known as the intermediate base class.

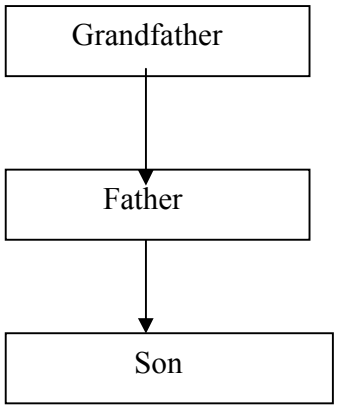

Figure 3.5 Multilevel Inheritance Graph

Inheritance implies both extension and contraction. Because a derived class behavior is, strictly speaking, broader than that of its parents we can say that the child class is an extension of its parent classes. And because the derived class can override some of its base class behavior, it is also a contraction.

The properties for the inheritance relationships, can be classified into different categories:

- **Specialization:** the derived class is a special case of its base class, it specializes its behavior and it is a subtype.
- **Generalization:** the base class is obtained as result of finding common behavior in different classes that become its children; these derived classes override some of the methods in the base class.
- **Specification*:*** the base class defines some behavior that it is only implemented in the derived class.
- **Extension*:*** the derived class adds some behavior but does not change the inherited behavior.
- **Combination*:*** the derived class inherits some behavior from more than one base class (multiple inheritance).
- **Construction*:*** the derived class uses the behavior implemented in the base class but it is not a subtype.
- **Limitation*:*** the derived class restricts the use of some of the behavior implemented in the base class.
- **Variance*:*** the derived and base class are variants one of the other and the relation class/subclass is arbitrary.

Inheritance provides a mechanism for code reuse. Inheritance in object-oriented design is used in accordance with the subsitution principle. During analysis and design, inheritance relationships between classes can be recognized in two general ways:

- As a specialization of some class.
- As a generalization of one or more classes.

An inheritance hierarchy allows to manage system complexity.

3.2.2 Advantages of Inheritance

1. **Reusability:** Inheritance helps the code to be reused in many situations. The base class is defined and once it is compiled, it need not be reworked. Using the concept of inheritance, the programmer can create as many derived classes from the base class as needed while adding specific features to each derived class as needed.

2. **Saves time and effort:** The above concept of reusability achieved by inheritance saves the programmer time and effort. Since the main code written can be reused in various situations as needed.
3. **Increases Program Structure**: The increase in program structure results in greater reliability.

3.3 Tool Support

Object oriented approach has substantial tool support for development[24]. Mario Van Damme identified three generations of object oriented case tools.

- **First Generation Object Oriented Case Tools- Diagramming Tool**
  Diagramming Tool only allow modeling of ideas- things like concepts, analysis artifacts etc. There's no link to code. Everything is maintained manually.
- **Second Generation Object Oriented Case Tools- Code Visualization Tool**
  A code visualization tool is a type of diagramming tools that knows how to harvest code and map it to model of classes and diagrams. These tools allow to start from a model and transform that into code.
- **Third Generation Object Oriented Case Tools- Computer Aided Software Modeling Tools**
  These tools assist the user in his/her choice regarding software architecture, data modeling, business modeling etc.

3.3.1 Case Tools for Object Oriented Analysis and Design

The case tools available for Object Oriented analysis and design are as follows:

- **Berard Object & Class Specifier (BOCS)**
  BOCS is an object-oriented analysis and design CASE tool for developing models of software & business systems and their underlying objects (classes, parameterized classes, and instances of classes). BOCS is used to create programming language independent specifications, then automatically generate formatted documentation combining text and graphics into popular publishing packages [25].

- **BridgePoint**

  The BridgePoint tool suite is an integrated set of automation tools specifically designed to support the Shlaer-Mellor Method of Object Oriented Analysis and Recursive Design through a detailed understanding of the underlying formalism.

- **Clyder**

  An object-oriented requirements engineering method, trying to combine rigor and usability [25]. Main characteristics are: few notations, native object orientedness, ability to range from informal but structured specifications to fully formal ones, formal semantics, sophisticated semantic checks, etc.

- **Ideogramic UML**

  A UML CASE tool with traditional features, e.g. the most important diagram types, XMI, printing, and reverse engineering, but with a user interface based on a quite different interaction principle: Instead of tool bars and menus, it uses gestural interaction. This makes it really easy and efficient to use, and uniquely also allows for use on a large electronic whiteboard and on Tablet.

3.4 Object Oriented Testing

Object Oriented Testing is done on four different levels depending on your approach, consisting of:

- Method Testing (Unit Testing)
- Class Testing (Unit Testing/Intraclass Testing)
- Interclass Testing (Integration Testing)
- System Testing

Full Life Cycle Object Oriented Testing methodology

The Full-Lifecycle Object-Oriented Testing (FLOOT) methodology is a collection of testing techniques to verify and validate object-oriented software. The FLOOT lifecycle is depicted in, indicating a wide variety of techniques are available throughout all aspects of software development.

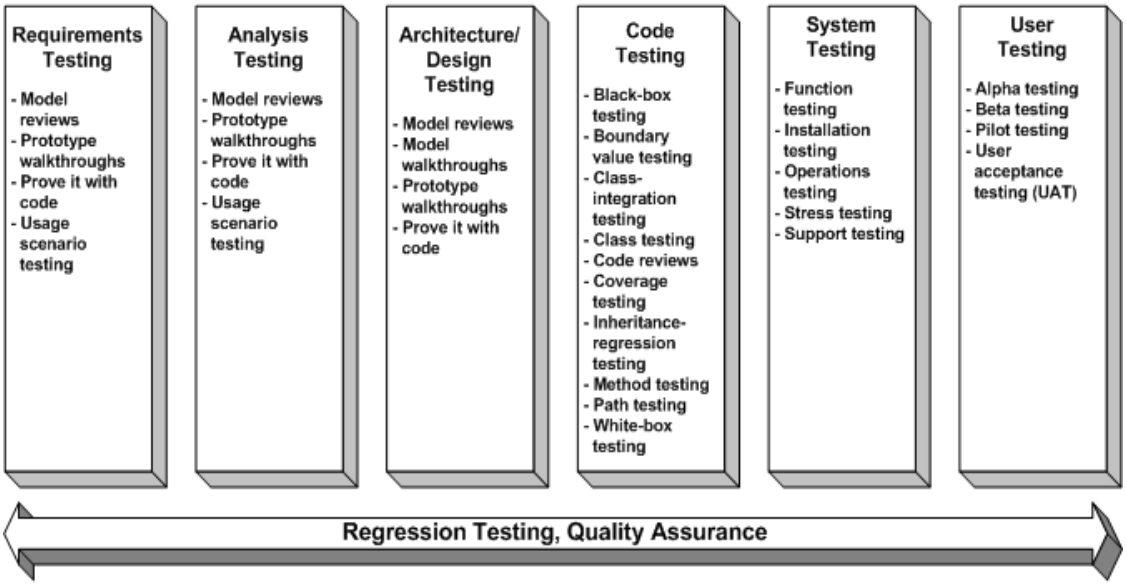

Figure 3.6 The FLOOT Lifecycle [23]

The table shows the various testing techniques used in Object Oriented Testing and its brief description.

FLOOT Technique	**Description**
Black-box testing	Testing that verifies the item being tested when given the appropriate input provides the expected results.
Boundary-value testing	Testing of unusual or extreme situations that an item should be able to handle.
Class testing	The act of ensuring that a class and its instances (objects) perform as defined.
Class-integration testing	The act of ensuring that the classes, and their instances, form some software performs as defined.
Code review	A form of technical review in which the deliverable being reviewed is source code.
Component testing	The act of validating that a component works as defined.
Coverage testing	The act of ensuring that every line of code is exercised at least once.

Design review	A technical review in which a design model is inspected.
Inheritance-regression testing	The act of running the test cases of the super classes, both direct and indirect, on a given subclass.
Integration testing	Testing to verify several portions of software work together.
Method testing	Testing to verify a method (member function) performs as defined.
Model review	An inspection, ranging anywhere from a formal technical review to an informal walkthrough, by others who were not directly involved with the development of the model.
Path testing	The act of ensuring that all logic paths within your code are exercised at least once.
Prototype review	Processes by which your users work through a collection of use cases, using a prototype as if it was the real system. The main goal is to test whether the design of the prototype meets their needs.
Prove it with code	The best way to determine if a model actually reflects what is needed, or what should be built, is to actually build software based on that model that show that the model works.
Regression testing	The acts of ensuring that previously tested behaviors still work as expected after changes have been made to an application.
Stress testing	The act of ensuring that the system performs as expected under high volumes of transactions, users, load, and so on.
Technical review	A quality assurance technique in which the design of your application is examined critically by a group of your peers. A review typically focuses on accuracy, quality, usability, and completeness. This process is often referred to as a walkthrough, an inspection, or a peer review.
Usage scenario testing	A testing technique in which one or more person(s) validate a model by acting through the logic of usage scenarios.
User interface testing	The testing of the user interface (UI) to ensure that it follows

	accepted UI standards and meets the requirements defined for it. Often referred to as graphical user interface (GUI) testing.
White-box testing	Testing to verify that specific lines of code work as defined. Also referred to as clear-box testing.

Table 3.1 Testing techniques [23].

Object Oriented System Testing is same as traditional testing and uses requirement specification. In Object Oriented Unit Testing two common structures are used that are Method and Class.

3.4.1 Object Oriented Integration Testing

Object Oriented Integration is the most complicated part of Object Oriented testing. It is the testing of the interaction of object-oriented components. It is based on composition in bottom-up approach. It makes use of clusters. The goal of integration testing is to find defects that arise when these fault free components interact with each other in an incorrect way.

3.4.1.1 Construct Definitions

The construct definitions for the Object Oriented Integration Testing are:

Method-Message Path (MM Path): A MM Path is a sequence of method executions linked by messages. It starts with a method and ends when it reaches a method that does not call another method. When there are no more subsequent method calls, the control returns to the start method. The start method has then executed its task and the system goes into a quiescence state. The execution in object-oriented software begins with an event. This event can be seen as an input port to the software, for example input port event B in the figure above. The input port event triggers the method message sequence of a MM-Path. This MM-Path may in its turn trigger other MM-Paths for completing its task. The MM-Path ends up with an output port event. This output port event might trigger a window to open or change the state of the system [19]. A MM Path is a small building block in scenario. Even if it is a small building block, a MM Path tests the integration and interaction of several objects. Several methods pass messages to each

other, their message interaction must work if a MM Path is to work.

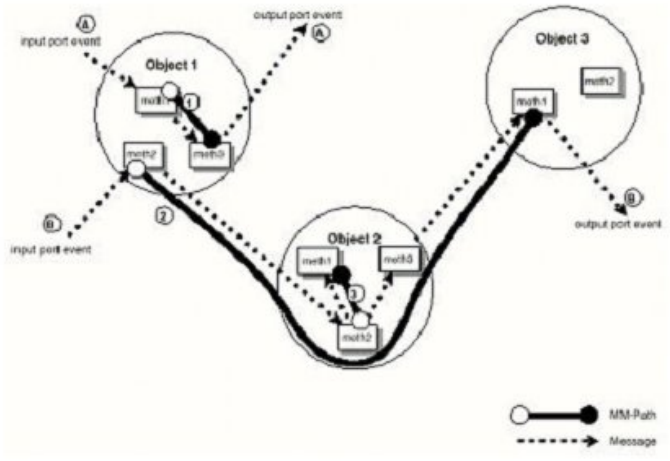

Figure 3.7 An Example of Method Message Path [17]

Atomic System Function: An ASF is an input port event followed by a set of MM-Paths and ends with an output port event. This sounds like a MM-Path consisting of other MM-Paths. The difference between an ASF and a MM-Path is that an ASF is an elemental function visible at the system level. A MM-Path on the contrary is not visible at the system level. The figure 3.8 shows the ASF of entering a Personal Identification Number (PIN) in an automated teller machine. It shows the interaction of objects when correct pin is entered.

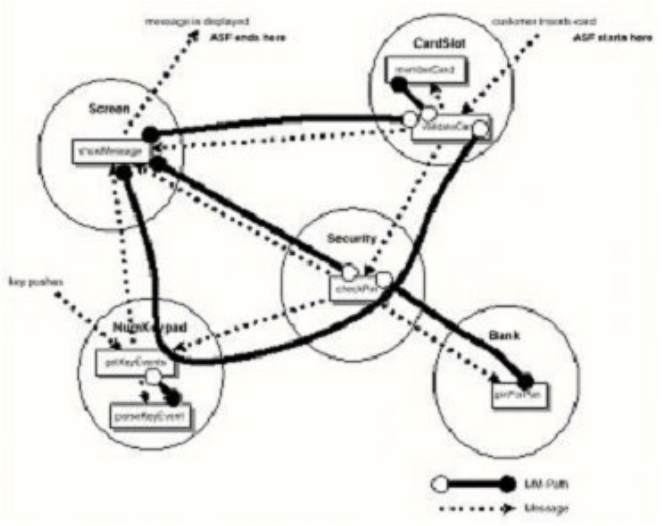

Figure 3.8 An Example of Atomic System Function Path [17]

3.4.2 Advantages of Object Oriented Testing

1. Once a class is testing thoroughly it can be reused without being unit tested again.
2. UML class state charts can help with selection of test cases for classes.
3. Classes easily mirror units in traditional software testing**.**

3.4.3 Disadvantages of Object Oriented Testing

1. Classes obvious unit choice, but they can be large in some applications.
2. Problems dealing with polymorphism and inheritance.

In this chapter we discuss Object Oriented Technology in detail including use cases, integration testing, inheritance, tool support required for the technology and advantages and disadvantages of use cases and object oriented testing. The next chapter describes the problem statement including similarities between the two approaches discussed above.

CHAPTER 4

PROBLEM STATEMENT

Cleanroom Software Engineering process evolved from concepts described over past many years by Harlan Mills and his colleagues. Cleanroom Technologies such as incremental development, precise specification and design, correctness verification and statistical testing have been applied successfully in commercial projects. Cleanroom approach is yet to become a common practice in software development industry because of various reasons that are as follows:

- Requires extra skills and knowledge.
- Ideal for safety critical system and not for ordinary commercial projects.
- Incomplete requirements cannot be resolved at the beginning to determine increments.
- Testing is not suitable for bug- hunting and would be effective if integrated with other testing methods.

The well-defined problems are as follows:

1. How the requirement problem can be resolved to determine increments.
2. Which testing is to be applied to make the process more effective?
3. How the flaws of Cleanroom Software Engineering can be overcome by using the advantages of Object Oriented Technology.

The similarities between Cleanroom Software Engineering and Object Oriented Technology are:

Life Cycle: Cleanroom Software Engineering follows the incremental development while the object-oriented approach follows the iterative development of the project.

Usage: The usage scenario involves the application of use case in Object Oriented Technology and the usage model in Cleanroom Software Engineering.

State machine Representation: Both use State Machine Representation for the behaviour of a design.

Reuse: The Object Oriented class in Object Oriented and the Cleanroom common service in Cleanroom Software Engineering are the units of reuse.

Although Cleanroom Software Engineering and Object Oriented Technology has many similarities and can be considered as complementary to each other, but still Cleanroom Software Engineering is not a common approach. 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. To further improve the Cleanroom Software Engineering Technology, a comparative study of Object Oriented and Cleanroom Software Engineering is performed and advantages of the object oriented are scrutinized that can be applied to Cleanroom Software Engineering.

CHAPTER 5

PROPOSED SOLUTION

5.1 The steps for existing Cleanroom Software Engineering

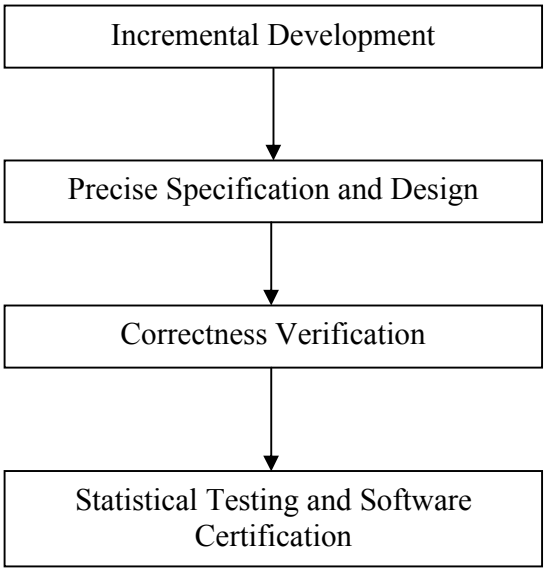

Figure 5.1 Existing Cleanroom Software Engineering Technologies

The steps for existing Cleanroom Software Engineering are described above in chapter1. The existing Cleanroom Software Engineering have some flaws that are described in chapter 4. To overcome the flaws of Cleanroom Software Engineering two new steps have been proposed for improving the technology.

5.2 Proposed Steps for the Cleanroom Software Engineering

The two new steps that is Collaborative Process for Requirement Engineering and mutation testing is discussed in detail in this chapter. Collaborative process is introduced for validating rapidly changing requirements. Mutation testing, a white box technique is introduced to prevent human errors in the software. The steps for proposed Cleanroom Software Engineering is shown in figure 5.2.

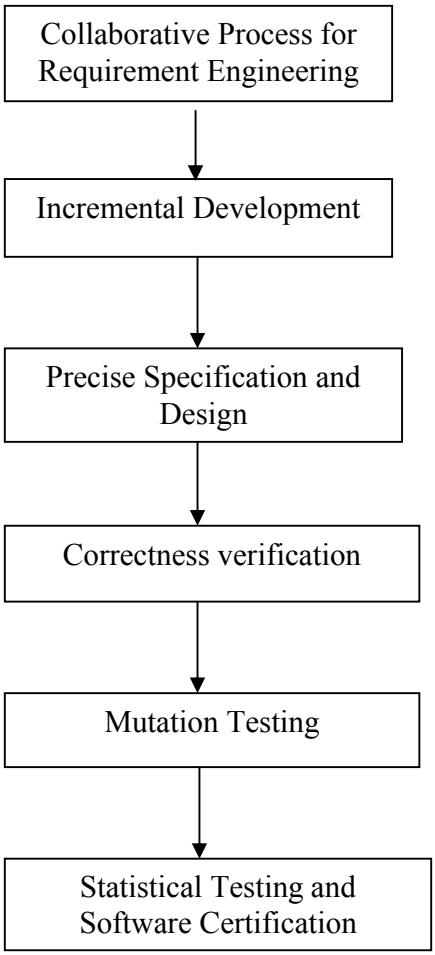

Figure 5.2 Proposed Cleanroom Software Engineering Technologies

5.3 Collaborative Process for Requirement Engineering

Collaborative Engineering is defined as a discipline that "studies the interactive process of engineering collaboration, whereby multiple interested stakeholders resolve conflicts, bargain for individual or collective advantages, agree upon courses of action, and/or attempt to craft joint outcomes which serve their mutual interests." The purpose of collaboration engineering is to quickly and repeatedly create an environment that promotes creativity and communication between participants. Collaboration engineering processes address the problem of having an unmanageable number of stakeholders by identifying and focusing on stakeholders critical to success. Collaboration engineering aims to simplify the process for practitioners by giving easy-to-use, repeatable and predictable techniques for efficiently synthesizing a large set of stakeholder wants and

needs into a workable set of requirements reached by consensus. Collaborative engineering denotes the joint practice of creating an engineering solution by a team. Organizations performing collaborative engineering can be regarded as a socio-technical system. Effective collaborative engineering requires both, guidance and best practice for team members when interacting in the team and matching technical mediation of these interaction processes [13]. Collaboration includes communication among team members, coordination of their activities. Collaborative Engineering focuses on high-value tasks, because improvements on those tasks can yield the most benefit to an organization. Collaborative Engineering focuses on recurring tasks for two reasons: 1) so that an organization can derive ongoing benefit from the investment in Collaborative Engineering; and 2) so that it is worth a practitioner's time to learn the process designed by a collaboration engineer. Collaborative Engineering has a fair degree of practical relevance in today's organizations with complex business processes addressing complex problems. Collaboration engineering helps in designing transferable, predictable and repeatable processes.

There are six general patterns of collaboration:

- **Generate**: Move from having fewer to having more concepts in the pool of concepts shared by the group
- **Reduce**: Move from having many concepts to a focus on fewer concepts that the group deems worthy of further attention
- **Clarify**: Move from having less to having more shared understanding of concepts and of the words and phrases used to express them.
- **Organize**: Move from less to more understanding of the relationships among concepts the group is considering
- **Evaluate**: Move from less to more understanding of the relative value of the concepts under consideration
- **Build consensus**: Move from having fewer to having more group members who are willing to commit to a proposal.

5.3.1 ThinkLets

ThinkLets is a pattern language for designing collaborative work practices. A thinkLet is the smallest unit of intellectual capital required to create one repeatable, predictable pattern of thinking among people working toward a goal.

ThinkLets are building blocks for designing collaborative processes. ThinkLets are useful because: they define which group support system or tool (the version of hardware and software technology) to use; how to configure it; and they provide a clear sequence of events and instructions (oral or written prompts) for the group to follow when using the tool [32]. ThinkLets are design patterns that can be used to create useful variations on the six general patterns of collaboration. They are prescriptive. ThinkLets are classified according to which of the six general patterns of collaboration they invoke. Table 5.1 lists the six general patterns of collaboration, and describes a corresponding thinkLet that can be used to invoke particular pattern of collaboration.

Pattern of Collaboration	**ThinkLet Example**	**Summary of ThinkLet**
Generate	LeafHopper	All participants view a set of pages, one for each of several discussion topics. Each participant hops among the topics to add ideas as inspired by interest and expertise.
Reduce	GoldMiner	Participants view a page containing a collection of ideas, perhaps from an earlier brainstorming activity. They work in parallel, moving the ideas they deem most worthy of more attention from the original page to another page
Clarify	Illuminator	Participants review a page of contributions for clarity. When a participant judges a contribution to be vague or ambiguous, s/he requests clarification. Other group

		members offer explanations, and the group agrees to a shared definition. If necessary, the group revises the contribution to better convey its agreed meaning.
Organize	PopcornSort	Participants work in parallel to move ideas from an unorganized list into to labeled categories, using a first-come-first-served protocol for deciding who gets to move each idea into a category.
Evaluate	StrawPoll	Moderator posts a page of unevaluated contributions. Participants are instructed to rate each item on a designated scale using designated criteria. Participants are told that they are not making a decision, just getting a sense of the group's opinions to help focus subsequent discussion.
Build consensus	Crowbar	After a vote, the moderator draws the group's attention to the items with the most disagreement. Group members discuss the reasons why someone might give an item a high rating, and why someone might give the item a low rating. The resulting conversation reveals unchallenged assumptions, unshared information, conflicts of goals, and other information useful to moving toward consensus.

Table 5.1 An Example of a ThinkLet for Particular Pattern of Collaboration [32]

5.3.2 Advantages of Collaborative Engineering ThinkLets:

1. As ThinkLet is technology independent they provide flexibility.

2. ThinkLets help to significantly reduce the cognitive load for practitioners by mapping facilitation techniques to the practitioner's own process.

5.4 Mutation Testing

Mutation testing is a method of software testing that is used to determine the correctness of software against functional verification. The process of mutation analysis can be automated to some extent therefore; it is used as white box method for unit testing. Mutation testing helps a user create test data by interacting with the user to iteratively strengthen the quality of test data.

Because program units are so much smaller, testers can find failures more efficiently during unit testing. In addition, it is easier to track down and solve faults (debug) in software units. Software units also tend to be generic code, which make them more amenable to testing by general-purpose, formal strategies. And what happens to the faults that we do not find when we skip unit testing? They are left in the software for the users to find, and for the maintainers to fix. To remove the faults in software through automated testing, mutation testing is used.

5.4.1 Mutation Process

During mutation testing, faults are introduced into a program by creating many versions of the program, each of which contains one fault. Test cases are used to execute these faulty programs with the goal of distinguishing the faulty programs from the original program. Faulty programs are called mutants of the original program and mutants are killed if the output of the mutant is different from the original program. Mutants follow coupling effect, which says that complex faults are coupled to simple faults in such a way that a test data set that detects all simple faults in a program will detect most complex faults. Mutation analysis provides a test criterion, rather than a test process. A testing criterion is a rule or collection of rules that imposes requirements on a set of test cases. Test requirements are specific things that must be satisfied or covered; for example, reaching statements are the requirements for statement coverage and killing mutants are the requirements for mutation [34].

The steps for mutation process are:

1. System creates mutant versions of the program when the same is submitted. Mutation operator is used to create the mutants. Mutation operators could be: replacing each operand with other legal operand, modify expressions by replacing operand or deleting the statement.
2. Test cases are supplied to the system to serve as input to the program. Each test case is executed on the original program and if the output is correct, it is executed on the mutant program. If the output is not correct bug is found and program is fixed before applying test case again. If the output of mutant program is different from the original mutant is marked as dead.
3. After all tests cases have been executed the mutation score is computed. The mutation score is the ratio of dead mutants over the number of non-equivalent mutants.
4. If mutants are not killed set of test cases are enhanced by supplying new inputs. The mutants that are not killed are called Equivalent Mutants.

   Equivalent mutants are the mutants that produce the same output as the original program in every case so they are not killed. Equivalent mutants are not counted in the mutation score. A mutation score threshold can be set as a policy decision to require testers to test software to a predefined level. Failures in the software are detected when test cases are executed against the original program. The tester must decide whether the output of the program on each test case is corrected.

   Solid Boxes represented steps that are automated and dashed boxes represent steps that are manual. The figure 5.1 shows the mutation testing process that can be used in Cleanroom Software engineering. Initially, a set of test cases is automatically generated and those test cases are executed against the original program, and then the mutants. The tester defines a threshold" value, which is a minimum acceptable mutation score. If the threshold has not been reached, then test cases that killed no mutants are removed. This process is repeated till the mutation score is achieved. At the end tester will examine the output of test cases and fix the bug if it is found. This process uses schema-based approach to execute the code and weak mutation is applied to reduce the amount of testing.

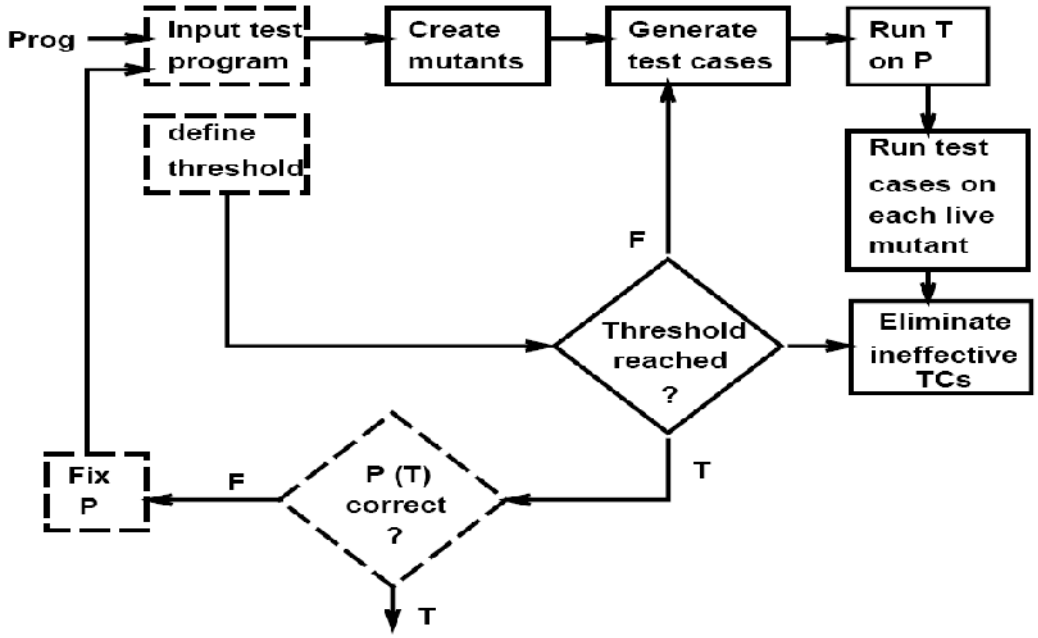

Figure: 5.3: Mutation Testing Process [34]

5.4.2 Advantages of Mutation Testing

1. Research has shown that there is a strong correlation between simple syntax errors and complex syntax errors in a program, i.e. usually a complex error will contain some simple error. By choosing a set of mutant programs that do a good job of distinguishing simple syntax errors, one has some degree of confidence that the set of mutant programs will also discover most complex syntax errors.
2. By proper choice of mutant operations, comprehensive testing can be performed. Research has shown that with an appropriate choice of mutant programs mutation testing is as powerful as Path testing or Domain analysis.
3. Mutation analysis is more easily automated that some other forms of testing. Robust automated testing tools have been developed at multiple universities and industrial testing groups.
4. Mutation testing lends itself nicely to stochastic analysis.

5.5 Comparison of Cleanroom Software Engineering and Object Oriented Technology

	Cleanroom Software Engineering	**Object Oriented Technology**
Data Objects	Data objects are created in decomposition and are reused when needed.	Data objects are identified first and then used in system design.
Methods	Markov Theory and Function Theory are formal methods that are used to the define completeness.	Informal Use case representations are used.
System Design	Cleanroom uses box structure that follows usage hierarchy	Object oriented uses inheritance hierarchy in form of classes
Tool Support	Cleanroom Software Engineering has good automated tool support for testing but little for development.	Object oriented approach has substantial tool support for development and even for testing.
Development Phase	It is used for life cycle application engineering.	It is used for front-end domain analysis.
Quality Control	The Cleanroom approach has both technical as well as management control.	The object-oriented approach has only technical control.
Development Approach	Cleanroom follows incremental approach.	Object oriented follows Bottom-up approach
Debugging	As box structures are used for verification so no debugging required.	Debugging is required for verification of code.
Representation Format	Cleanroom practitioners use tables and symbols formalism.	Object oriented practitioners use graphics for representation.

Testing	Statistical Usage Testing is done to indicate quality.	Integration testing is performed to indicate quality.
Software products	It is basically used for development of safety critical products and not for ordinary products.	It is used for ordinary products also.
Requirements	It cannot adapt quickly to rapidly changing requirements.	It can be adapt quickly to rapidly changing requirements.
Quality	Cleanroom Software Engineering produces almost zero defect software.	Object Oriented Technology not always produces zero defect software.
Configuration Management	Cleanroom Software Engineering requires more care about configuration management.	It does not require much care about configuration management when compared to Cleanroom Technology.

Table 5.2 Comparison of Cleanroom Software Engineering and Object Oriented Technology

CHAPTER 6
CONCLUSION AND FUTURE SCOPE

Cleanroom software engineering is used for life cycle application engineering. The Cleanroom approach has both technical as well as management control. It is basically used for development of safety critical products and not for ordinary products. Cleanroom follows incremental approach. Cleanroom practitioners use tables and symbols formalism. Here, usage testing is done to indicate quality. It uses box structures for verification, so no debugging is required. But it cannot adapt quickly to rapidly changing requirements. For improving this aspect of Cleanroom Software Engineering, we suggest the use collaborative engineering for requirements engineering. Also the use of formal languages makes it difficult to use for many developers. But the use of formal methods makes it more correct and verifiable. Inheritance hierarchy is useful in object oriented technology, so if we can include inheritance in box structure to improve. Unit testing is very important for debugging the human errors. But there is no provision for unit testing in Cleanroom Software Engineering. The inclusion of mutation testing can improve this aspect of Cleanroom Software Engineering.

The Cleanroom Software Engineering can be further improved by finding the gaps between Case tools defined for the Cleanroom Software Engineering and how they can be used to improve Cleanroom Software Engineering.

REFERENCES

[1] Richard C. Linger, Carmen, J. Trammell, Cleanroom Software Engineering Reference Model Version 1.0, Technical Report CMU/SEI-96-TR-022 ESC- TR-96-022, November 1996

[2] Linger, Mills, and Witt, Structured Programming: Theory and Practice, Addison-Wesley, 1979

[3] Hausler, P.A., Linger, R.C., Trammel, C.J.; Adopting Cleanroom Software Engineering With a Phased Approach, IBM Systems *Journal,* Volume 33, Number 1, 1994

[4] Pfleeger, S.L., Hatton, L. Investigating the Influence of Formal Methods, *IEEE Computer***,** Volume 30 Issue 2, Feb.1997, pp. 33 -43.

[5] Roger S. Pressman, Software Engineering A Practitioner's Approach fifth Edition,

[6] R. H. Cobb and H. D. Mills, Engineering Software Under Statistical Quality Control, IEEE Software, 7(6): 44–54, 1990

[7] Booch, Grady, Object-Oriented Analysis and Design. Addison-Wesley, 2nd edition, 1994

[8] Linger, R.C., Cleanroom Process Model, IEEE Sofhziare, March 1994

[9] Dyer, Michae1, The Cleanroom Approach to Quality Software Development, John Wiley and Sons, 1992.

[10] Cleanroom Software Engineering Tools
https://www.thedacs.com/databases/url/key/64/90.html

[11] Thomas J. Bergin, Computer-aided software engineering: issues and trends for the 1990s and beyond, Idea Group Inc (IGI), 1993

[12] A. R. Hevner and H. D. Mills, Box-Structured Methods for Systems Development with Objects, ZBM Systems Journal 32**,** No. 2, pages 232-251, 1993

[13] Ann Fruhling, Lucas Steinhauser, Gregory Hoff, Christopher Dunbar, Designing and Evaluating Collaborative Processes for Requirements Elicitation and Validation Proceedings of the 40th Hawaii International Conference on System Sciences – 2007

[14] Kemp, K., Formal Methods Specification and Verification Guidebook for Software and Computer Systems, Volume I: Planning and Technology Insertion, NASA, 1998.

[15] Deck M., An Introduction to Cleanroom Software Engineering for Managers, Cleanroom Software Engineering Inc., Boulder, CO, USA, 1995.

[16] Inheritance Hierarchies,
http://www.iua.upf.es/~xamat/Thesis/html/node11.html

[17] Christian Bucanac, Object Oriented Testing Report, 1998

[18] Meyer, Bertrand, Object-Oriented Software Construction, Prentice Hall, 2nd edition, 1997.

[19] Cem Kaner, An Introduction to Scenario Testing, Software Testing & Quality Engineering (STQE) magazine, 2003

[20] Pressmen and Associates, Cleanroom Engineering Resources.
http://www.rspa.com/spi/cleanroom.html

[21] H.S. Hong, Y.R. Kwon, and S.D. Cha. Testing of Object-Oriented Programs Based on Finite State Machines, Korea Advanced Institute of Science and Technology, Dept of Computer Science Technical Report, 1996.

[22] D. P. Kelly and R. S. Oshana, Improving Software Quality using Statistical Testing Techniques, Information and Software Technology, 2000

[23] The Full Life Cycle Object Oriented Testing (FLOOT) Method
www.ambysoft.com/essays/floot.html

[24] Object Oriented CASE Tools
http://www.developerdotstar.com/mag/articles/oo_case.html

[25] Case Tools for Object Oriented Analysis and Design
http://www.faqs.org/faqs/software-eng/part2/section-4.html

[26] J. A. Whittaker and J. H. Poore, Markov analysis of software specifications, ACM Transactions on Software Engineering and Methodology, pages 93–106, 1993

[27] Regnell, B., Andersson, M., & Bergstrand J., A Hierarchical Use Case Model with Graphical Representation, Proceedings ECBS'96, IEEE International Symposium and Workshop on Engineering of Computer-Based Systems, 1996

[28] B. Regnell, P. Runeson and C. Wohlin, Towards Integration of Use Case Modeling and Usage-Based Testing , Journal of Software and Systems, Vol. 50, No. 2, pages. 117-130, 2000

[29] Bhuvan Unhelkar, Use Cases: Good & Bad, 2009
http://alinement.net/index.php/component/content/article/69

[30] Poore J.H., H.D. Mills, and D. Mutchler, Planning and Certifying Software System Reliability, IEEE Software, pp 88-99, January, 1993

[31] Lijter M., Meyers S., and Reiss S. P., Support for maintaining object-oriented programs, IEEE Transactions on Software Engineering, 18(12), pp. 1045-1052, Dec. 1992

[32] Vreede G.J., Kolfschoten, G.L., ThinkLets: A Pattern Language for Facilitated and Practitioner-Guided Collaboration Processes, International Journal of Computer Applications in Technology, pages140-154, 2006

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

[34] B. Choi and A. P. Mathur, High-performance mutation testing, The Journal of Systems and Software, vol. 20, pp. 135-152, February 1993

[35] R. Untch, A. J. O_utt, and M. J. Harrold. Mutation analysis using program schemata, Proceedings of the 1993 International Symposium on Software Testing, and Analysis, pages 139-148,Cambridge MA, June 1993.

LIST OF PUBLICATIONS

Nupur Chugh, Shivani Goel, Introducing Mutation Testing and Collaborative Engineering in Cleanroom Software Engineering Process, In Proceedings of IEEE International Advance Computing Conference (IACC 09), Thapar University, Patiala, 6-7 March 2009