Cleanroom Software Engineering: An Overview

SIGAda 2000
Tutorial SF7
12 November 2000

Presented by:
William Bail
The MITRE Corporation and
The University of Maryland
Agenda

- Introduction and Overview
- Basic Principles
- History of Cleanroom
- Experience using Cleanroom (case studies)
- Foundations of Cleanroom
- Cleanroom Process Model
- Cleanroom and the CMM
- Cleanroom and Object-Oriented Development
- Conclusion
Introduction

- Developed by Dr. Harlan Mills while at IBM
- Cleanroom Software Development is a set of practices that, when followed, is said to improve the development experience and the resulting product
  - Higher quality
  - More efficient
  - Repeatable results
- Cleanroom has evolved from its original concept, as have other techniques
  - Your preconceptions about Cleanroom may be inaccurate
  - Currently viewed as a family of techniques, united by a common concept
Introduction (cont’d)

- The name was borrowed from the Cleanroom facility required for semiconductor manufacturing
  - Foundries have to be very very clean to avoid faults on the microcircuits
    » Keep the defects from occurring during manufacturing (e.g., dust)
  - The central idea behind Software Cleanroom is to create defect-free software
    » Avoid the defects rather than finding them after development via testing

But don’t worry - you don’t have to clean your desk
Overview of Tutorial

- MITRE, as a part of its continuing evaluation of software techniques, has identified Cleanroom as a key process suitable for adoption for the development of critical systems
- This tutorial is based on Cleanroom references that are readily available
- **Purpose** of this tutorial - to provide an introduction to gain familiarity and appreciation
- Does not provide enough detail to begin practice
  - Will provide an overview of the basic concepts and practices
  - Will tantalize and tease, but for deeper information, you will need to consult the references
- The material is scheduled for four 1 ½ hour sessions
Agenda

- Introduction and Overview
- Basic Principles
- History of Cleanroom
- Experience using Cleanroom (case studies)
- Foundations of Cleanroom
- Cleanroom Process Model
- Cleanroom and the CMM
- Cleanroom and Object-Oriented Development
- Conclusion
Basic Principles

- Primary focus is on **defect avoidance rather than defect removal**
  - Make it right the first time
  - Recognize that defects are not inevitable
  - Avoid the costs of rework which are more expensive than original construction

- Emphasis on software development as an exercise in mathematics
  - Exploit mathematics to facilitate the definition and verification of correct programs
    » Some view Cleanroom as being *semi-formal*
  - Avoid trial-and-error as a strategy
  - Employ statistics to certify quality

- Four key features -
  - Incremental development process
  - Formal specification and hierarchical design techniques
  - Formal reviews/ inspections to verify correctness
  - Statistical quality control
Key Features - Incremental Development

- Break problem down into multiple, smaller problems
  - Not just 2 or 3 - but 5, 10, and more
  - Each more manageable than the whole system
  - As a series of incremental builds

- With each increment a full product is developed, but with partial functionality
  - “100% of 20%”, rather than “20% of 100%”

- Each increment is cumulative, building to a full product

- Many small steps result in a more manageable development by reducing complexity
Key Features - Formal Specification / Hierarchical Design

- Based on three types of “Box Structures”, each of which provide different views of the same component
  - **Black Box** - external view of the component - precisely defines behavior
  - **State Box** - mixed external/internal view - defines behavior linked to a limited set of design features (state variables)
    - Intermediate step between behavior and design
    - Derived from Black Boxes by identifying states and state variables
  - **Clear Boxes** - internal view - defines the full design, with possible use of lower-level Black Boxes *(hence hierarchical)*
    - Based on Structured Programming principles to facilitate proofs of correctness
    - Derived from State Boxes by identifying algorithms needed to implement behavior

- Each Box is derived formally from the previous Box
  - and is proven to be semantically equivalent (validation)
Key Features - Formal Verification

- Formal reviews / inspections to verify correctness
  - Similar to Fagan inspections
    - Also developed out of IBM
  - Proofs based on design structure rather than embedded logic
    - Therefore simpler to perform than “traditional” proofs of correctness
      - which are hard to perform particularly for large programs / components
    - Underpinnings are formal, practice is “semi-formal”
      - Often called “Light-weight” formal methods
  - Performed after each development step
  - Made possible by restricting design / code to Structured Programming constructs, each with a defined semantics
    - <if-then-else>, <do-while>, <sequence>
  - Correctness for total product inferred from correctness of each component within the product
Key Features - Statistical Quality Control

- Statistical quality control
  - Achieved through reliability testing on increments
  - Supported by creation of Usage Models in the form of Markov Chains
    » Similar to Use Cases but more comprehensive
    » Requires understanding how the product will be used
  - Test data created by random sampling from the Usage Models
    » To recreate the operational environment
  - Results provide prediction of expected MTTF and of stopping point for testing
    » Only approximate of course
Overall Development Steps

Increment 3
Specification
(Black Box)

Increment 2
Specification

Increment 1
Specification
(Black Box)

State Box
Clear Box
Verification

Usage Model
Test Planning

Statistical Testing
Certification
Agenda

- Introduction and Overview
- Basic Principles
- History of Cleanroom
- Experience using Cleanroom (case studies)
- Foundations of Cleanroom
- Cleanroom Process Model
- Cleanroom and the CMM
- Cleanroom and Object-Oriented Development
- Conclusion
History

- Originated with Harlan Mills while he was at IBM
  - Along with codevelopers Michael Dyer and Richard Linger
  - Resulted from an evolution of ideas culminating in their article in *IEEE Software* in September 1987, “Cleanroom Software Engineering”

- Strongly based on previous work
  - Dijkstra - Structured Programming
  - Parkas - Modular design
  - Wirth - Stepwise refinement
Significant Milestones

- **1979** - *Structured Programming: Theory and Practice*, by Linger, Mills, and Witt - Documents formal approach to design and verification


- **1988** - DoD ARPA STARS program - Cleanroom selected as a key technology


- **1995** - SEI correlates Cleanroom to CMM - Maps Cleanroom process to Capability Maturity Model processes and standards
Agenda

- Introduction and Overview
- Basic Principles
- History of Cleanroom
- Experience using Cleanroom (case studies)
- Foundations of Cleanroom
- Cleanroom Process Model
- Cleanroom and the CMM
- Cleanroom and Object-Oriented Development
- Conclusion
Experience Using Cleanroom

- Cleanroom is not just a laboratory toy
- It has been applied to real projects, with great success
- We will look at 6 case studies
  - IBM Cobol Structuring Facility (Cobol/SF)
  - NASA Satellite-control project
  - NASA Satellite Control
  - IBM AOEXPERT/MVS
  - IBM 3090E Tape Drive
  - Ericsson Telecon S32 OS
Case Study 1 - IBM Cobol Structuring Facility (COBOL/SF)

- IBM’s 1st commercial Cleanroom product
  - Product converts “unstructured” Cobol code into structured code
- Complexity of product ≈ COBOL compiler
- Developed in five increments (with no unit testing)
- Complete product consisted of
  - New code - 52 KSLOC PL/I
  - Existing code - 28 KSLOC PL/I
- 179 defects observed
  - From first SW test through the end of statistical testing
- 3.4 defects/KSLOC overall (for new code)
- Productivity = 740 SLOC/sm
- Postdelivery experience - 10 defects discovered by users after release for first 3 years (0.2 defects/KSLOC)
Case Study 2 - NASA Satellite-Control Project

- CFADS - Coarse/Fine Attitude Determination System
  - Part of NASA Attitude Ground Support System (AGSS)
- First Cleanroom project performed by Software Engineering Laboratory (SEL)
  - Part of Goddard Space Flight Center
- 31 KSLOC Fortran in 6 increments
  - No unit testing
- 3.3 errors/KSLOC during statistical testing
- 780 SLOC/sm productivity
Case Study 2 - NASA Satellite-Control Project (cont’d)

- Where the defects were found

<table>
<thead>
<tr>
<th>Design Review</th>
<th>Code Inspection</th>
<th>Compile</th>
<th>Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>32%</td>
<td>53%</td>
<td>5%</td>
<td>10%</td>
</tr>
</tbody>
</table>

Data suggests that Cleanroom finds defects earlier

In general, the earlier you find defects, the less the cost impact to fix

- Note that 90% of the defects were discovered before the SW was executed
  - i.e., before testing

- What is typical? Approximately ...

<table>
<thead>
<tr>
<th>Design &amp; Code Inspections</th>
<th>Unit Test</th>
<th>Integration</th>
<th>System Test</th>
<th>Op Demo</th>
</tr>
</thead>
<tbody>
<tr>
<td>55%</td>
<td>10%</td>
<td>20%</td>
<td>10%</td>
<td>5%</td>
</tr>
</tbody>
</table>

Dyer
Case Study 4 - NASA Satellite Control

- 2 systems
  - Attitude determination subsystem (20 KSLOC)
  - Flight dynamics system (150 KSLOC)
- The 2nd and 3rd Cleanroom projects at the SEL
- 4.2 errors/KSLOC
Case Study 3 - IBM AOEXPERT/MVS

- A decision support system that predicts and prevents problems in an MVS environment
- Written in C, PL/I, Rexx, and TIRS - 107 KSLOC
- 2.6 errors/KSLOC
- 486 SLOC/sm
- Experienced no operational errors after beta test
Case Study 5 - IBM 3090E Tape Drive

- Device controller to support tape cartridge operations
- Consisted 86 KSLOC C code
- Software development revealed hardware errors due to requirements definition process
- 1.2 errors/KSLOC
Case Study 6 - Ericsson Telecon S32 OS

- 70 staff over 18 months
- 30 KSLOC OS for new switching computers
- Error rate of 1.0 errors/KSLOC
Range of Published Defect Densities

Data Source

- Pfleeger, Fenton & Page - 1994, subsystem
- Pfleeger, Fenton & Page - 1994, total system
- Business Week 1991 - Japan
- Business Week 1991 - USA
- Bazzana 1995 - Proj A
- Bazzana 1995 - Proj B
- Belford 1979
- Rubey 1975
- Alyama 1972

Average defect density in USA (M. Dyer, 1992)
Agenda

- Introduction and Overview
- Basic Principles
- History of Cleanroom
- Experience using Cleanroom (case studies)
- Foundations of Cleanroom
- Cleanroom Process Model
- Cleanroom and the CMM
- Cleanroom and Object-Oriented Development
- Conclusion
Foundations for Cleanroom

- Harlan Mills saw SW development as an exercise in mathematics
  - Using sound techniques, he believed that
    - Programmers could write correct programs
    - Verify that they are correct using proofs of correctness
    - Demonstrate their quality by showing how they work under realistic scenarios
  - Basing proofs on Structured Programming constructs
    - <sequence>, <if-then-else>, and <while-do>
    - Each of which has formally defined semantics
- Mills compares Cleanroom to long division where the answer is computed one step at a time and the correctness of each step verified before moving to next step
- Primary focus on defect avoidance rather than defect removal
  - Make it right the first time
  - Avoid rework costs which are more expensive than original construction
Key Features

- Incremental development process
- Formal specification and hierarchical design techniques based on *Box Structures*
- Formal reviews/ inspections to verify correctness
- Statistical quality control
General Cleanroom Life Cycle

1. Define Requirements
2. Define Incremental Builds
3. Define Black Box
4. Define State Box
5. Verify State Box
6. Define Clear Box
7. Verify Clear Box
8. Certify Increment

Performed once per increment
Key Features - Incremental Development Process

- Incremental development process
- Formal specification and hierarchical design techniques based on *Box Structures*
- Formal reviews/ inspections to verify correctness
- Statistical quality control
Incremental Development

- Construct product in small steps (*increments*)
- Developing with increments breaks the problem up into smaller, more manageable chunks
- With each increment a **full** product is developed, but with partial functionality
  ✷ “100% of 20%”
    » A fully executable and usable product is built
    » with 20% of the functionality that is planned for the whole product
  ✷ “20% of 100%”
    » A partial product that cannot be executed or tested as a whole
Cumulative Increments

- Each increment is cumulative, building to a full product
  - Each increment contains the full functionality of all previous increments, plus adds new capabilities
  - Avoid having to redo portions of earlier builds (too expensive)
- Cleanroom supports two approaches (*):
  - Incremental development - where the requirements are fixed at the beginning, and successive builds implement more and more requirements
  - Evolutionary development - where the requirements are revisited after each build

Increment 1

Increment 2

Increment 3

Increment 4

10 +

*MIL-STD-498
Benefits of Incremental Development

- Clear evidence of progress
  - Each increment represents a “complete” product that can be executed and used
  - Early demonstration builds confidence and verifies that progress is being made

- Support for early user feedback
  - Users can take the increments out for a spin and kick the tires (so to speak)
  - Opportunity for early feedback to redirect requirements

- Recursive testing enhances product quality
  - Each increment is tested as a full product
  - Early increments are tested repeatedly (with each successive increment), enhancing quality
  - Repeated testing provides early and extensive feedback on product and process quality
Benefits of Incremental Development (cont’d)

- Greater control for developers
  - Multiple increments defined with small increases of functionality with each increment
  - Allows developers to more carefully focus on current build and to manage change
  - Can decide when in product development the “hard” problems are to be encountered

- More effective system integration
  - Since increments can be integrated early and are integrated repeatedly, system integration has more opportunity to be successful
Increment Evolution

Increment 1  Increment 2  Increment 3

New and changed modules represent added functionality
Stubs represent subordinate Black Boxes and are defined
with full semantics but no implementation (Clear Boxes)
This semantics is necessary to support formal verification
at each step
Key Features - Formal Specification and Hierarchical Design

- Incremental development process
- Formal specification and hierarchical design techniques based on Box Structures
- Formal reviews/inspections to verify correctness
- Statistical quality control
Formal Specification and Hierarchical Design

- Based on *Box Structures*
  - An approach to system specification and design using stepwise refinement and verification
  - Small design steps with verification of each step

- Defined by Mills in

- Guided by 4 principles
  - **Referential Transparency** - All requirements for components should be fully defined (avoiding side effects)
    - Allowing change without affecting dependent components
  - **Transaction Closure** - The operations on a component must cover the state data, and the state data must be adequate to cover all operations
  - **State Migration** - Data should be encapsulated in the smallest possible component to ensure locality
  - **Common Services** - Components with multiple uses across the system should be made available as common services (“reuse”)
Box Structures

- Box Structures provide 3 views of each software component
  - **Black Box** view - a precise definition of the component’s behavior in terms of stimuli and responses
  - **State Box** view - definition of component’s behavior using state machines, and includes state variables to be used in the design
  - **Clear Box** view - defines the algorithms and processing flow, in PDL or code
    - Based on Structured Programming concepts (limited control constructs) to facilitate proofs of correctness
    - Each Clear Box may contain lower level Black Boxes
      - Each with a Black, State, and Clear Box views
    - Results in a decomposition hierarchy

- Each component is defined by all 3 box types
  - Which are semantically equivalent
  - Developed in the order - BB → SB → CB
    - Increasing design content
Box Structure Hierarchy

- Stepwise refinement produces a hierarchy of black boxes
  - Hierarchy defines where each Box is used (a usage hierarchy)
    - as opposed to where defined

```
          BB
         /   \
        SB   CB
       /     \
      BB     BB
     /   \
    SB   CB
   /     \
  BB   BB
 /     \
SB   SB
|     |
BB   BB
|     |
SB   SB
|     |
BB   BB
|     |
SB   SB
|     |
BB   BB
|     |
SB   SB
|     |
BB   BB
```
Box Structure Development

- Box structure design consists of 3 steps:
  - Define behavior by enumerating stimuli and responses - Black Box
  - Refine the Black Box by defining state variables and transitions - State Box
    » Must be shown to have the same semantics as the Black Box
  - Refine the State Box by defining computation necessary to implement the behaviors - Clear Box
    » Must be shown to have the same semantics as the State Box

- Each step requires making design decisions - not a mechanical process
  - Each Black Box can be equivalent to many possible State Boxes - one must be selected
  - Each State Box can be equivalent to many possible Clear Boxes - one must be selected
Box Structure Development (cont’d)

- After each step, the product is verified to be correct
  - The derived State Box is shown to be equivalent to the Black Box
  - The derived Clear Box is shown to be equivalent to the State Box

- There are many techniques that can be used to define behavioral requirements
  - Z
  - We will describe **Sequence-based Specification**
Black Box

- A description of the externally-visible behavior of a specific SW component
  - Whether the component is at the top-level (system) or at the low-level (module)
  - Its functional requirements
- A map of inputs to outputs
  - Stimuli are mapped to responses
  - Viewing software components as mathematical functions
- Inputs to be represented as stimulus histories
  - Since the outputs (responses) may vary depending on the sequence of stimuli that have been experienced (persistent memory)
- The Black Box view presents the behavior view of the component
Underlying Formalisms

A Black Box is a function that maps stimuli histories to responses

\[ S^* \rightarrow BB : S^* \rightarrow R \rightarrow R \]

A regular expression that denotes all possible strings constructed out of elements in the set \( S \)

Example: If \( S = \{ a, b, c \} \) then

\[ S^* = \{ a, b, c, \]
\[ \quad a, b, ac, ba, bb, bc, ca, cb, cc, \]
\[ \quad a, aab, aac, \]
\[ \quad aaaa, aaab, aaac, \ldots \} \]

Strings of length 1
Strings of length 2
Strings of length 3
Strings of length 4
etc.
A Protest from the Audience

Our goal is to enumerate all possible inputs and sequences of inputs, and, for each, define the expected output

Who are you kidding? Not in MY lifetime!!

Actually, it is not that bad...stay tuned...
Example

Let $S = \{a, b, c\}$ and let $R = \{Yes, No\}$

Define a function $f$ that returns a Yes whenever its input contains 2 a’s in a row and returns a No otherwise.

That is:

$$f(i) = \begin{cases} 
Yes & \text{if } i \text{ contains 2 a’s in a row} \\
No & \text{otherwise}
\end{cases}$$
Example (cont’d)

Strings of length 1:
- a
- b
- c

Strings of length 2:
- aa
- ab
- ac
- ba
- bb
- bc
- ca
- cb
- cc

Strings of length 3:
- aba
- abb
- aac
- acc
- bca
- bbc
- bba
- bbb
- bbb
- cca
- ccb
- ccc

Strings of length 4:
- aaaa
- aabb
- aacc
- abba
- abbb
- aacc
- abca
- abcb
- abcc

Input | Output
--- | ---
a | No
b | No
c | No

Input | Output
aaa | Yes
aab | Yes
aac | Yes
aba | No
abb | No
abc | No
aca | Yes
acb | No
acc | Yes
baa | Yes
bab | No
bac | No

Input | Output
bba | No
bbb | No
bbc | No
baa | Yes
bab | No
bac | No

Input | Output
cca | No
ccb | No
ccc | No

infinite number of possible strings
Some Considerations

- Stimuli sequences can be thought of as being processed
  - One-by-one, or
  - All-at-once

- All-at-once
  - Stimuli are individual input strings, perhaps selected from an infinite set
  - The function processes one input and provides an output
  - No stimulus history is needed (or is relevant)

- One-by-one
  - Each stimulus is input, processed, and a response is emitted
  - Stimulus histories may be necessary to define the function

- Our discussion focuses on one-by-one stimuli
  - More interesting
  - Allows us to focus on the state-based computation
  - Fully equivalent to the all-at-once view
Some Considerations (cont’d)

- Other techniques that we could have used to define sequences include:
  - Regular expressions - to describe stimuli histories
  - Grammars - to define stimuli histories
  - Others? ... join in with ideas ...
What about...?

- What happens if no output is necessary for a specific input?
  - Define a special output $\bullet$ (the null response) which denotes no output

- What about an empty input sequence?
  - Define a special symbol to denote this situation: $\lambda$
  - Note that $BB(\lambda) = \bullet$

- What about an input sequence that is not legal?
  - Unfortunately these happen all the time with real systems
  - Since our function definitions have to be complete (represent total functions), we need to handle these appropriately
    » Exception handling

- What about an input sequence that is impossible?
  - That is, physically impossible due to external conditions
  - Map these to the special symbol $\S$
  - That is, $BB( xxx ) = \S$ where $xxx$ is impossible
  - Not an output - just an indication of not_possible
Example Revisited

Define \( BB : S^* \rightarrow R \)
where
\[
S = \{ a, b, c, \ldots, z \} \\
R = \{ \text{Yes, No} \}
\]
Let \( \sigma \) denote a string from \( S^* \) - that is, \( \sigma \in S^* \)

<table>
<thead>
<tr>
<th>( \sigma )</th>
<th>( BB(\sigma) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \lambda )</td>
<td>Yes</td>
</tr>
<tr>
<td>a</td>
<td>No</td>
</tr>
<tr>
<td>b</td>
<td>No</td>
</tr>
<tr>
<td>c</td>
<td>No</td>
</tr>
<tr>
<td>aa</td>
<td>Yes</td>
</tr>
<tr>
<td>ab</td>
<td>No</td>
</tr>
<tr>
<td>ac</td>
<td>No</td>
</tr>
<tr>
<td>ba</td>
<td>No</td>
</tr>
<tr>
<td>bb</td>
<td>No</td>
</tr>
<tr>
<td>bc</td>
<td>No</td>
</tr>
<tr>
<td>ca</td>
<td>No</td>
</tr>
<tr>
<td>cb</td>
<td>No</td>
</tr>
<tr>
<td>cc</td>
<td>No</td>
</tr>
</tbody>
</table>

Note that certain sequences are equivalent.

Once an stimulus history has experienced an aa pattern, then the response will always be “Yes” for all subsequent stimuli.

\( \text{aaa, aab, aac, baa, caa} \)

are all equivalent to \( \text{aa} \)

etc...
Refinements

Identifying equivalent sequences

- Two stimuli sequences are equivalent iff they produce the same responses for all future stimuli
- If a sequence is equivalent to another one, it is reduced
  - In the example, stimulus aac is reduced to sequence aa
  - ccc can be reduced to cc
  - aba can be reduced to a
- If a sequence is not equivalent to any previous one, it is irreducible

In the sense of a sequence ordered by length
# New Enumeration Table

<table>
<thead>
<tr>
<th>$\sigma$</th>
<th>$BB(\sigma)$</th>
<th>Equiv</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\lambda$</td>
<td>$\bullet$</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>No</td>
<td>b</td>
</tr>
<tr>
<td>c</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>aa</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>ab</td>
<td>No</td>
<td>b</td>
</tr>
<tr>
<td>ac</td>
<td>No</td>
<td>b</td>
</tr>
<tr>
<td>ba</td>
<td>No</td>
<td>a</td>
</tr>
<tr>
<td>bb</td>
<td>No</td>
<td>b</td>
</tr>
<tr>
<td>bc</td>
<td>No</td>
<td>b</td>
</tr>
<tr>
<td>ca</td>
<td>No</td>
<td>a</td>
</tr>
<tr>
<td>cb</td>
<td>No</td>
<td>b</td>
</tr>
<tr>
<td>cc</td>
<td>No</td>
<td>b</td>
</tr>
<tr>
<td>aab</td>
<td>Yes</td>
<td>a</td>
</tr>
<tr>
<td>abc</td>
<td>Yes</td>
<td>a</td>
</tr>
<tr>
<td>aba</td>
<td>No</td>
<td>b</td>
</tr>
<tr>
<td>abb</td>
<td>No</td>
<td>b</td>
</tr>
<tr>
<td>abc</td>
<td>No</td>
<td>b</td>
</tr>
<tr>
<td>aca</td>
<td>No</td>
<td>a</td>
</tr>
<tr>
<td>acb</td>
<td>No</td>
<td>b</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\sigma$</th>
<th>$BB(\sigma)$</th>
<th>Equiv</th>
</tr>
</thead>
<tbody>
<tr>
<td>acc</td>
<td>No</td>
<td>b</td>
</tr>
<tr>
<td>baa</td>
<td>Yes</td>
<td>aa</td>
</tr>
<tr>
<td>bab</td>
<td>No</td>
<td>b</td>
</tr>
<tr>
<td>bac</td>
<td>No</td>
<td>b</td>
</tr>
<tr>
<td>bba</td>
<td>No</td>
<td>a</td>
</tr>
<tr>
<td>bbb</td>
<td>No</td>
<td>b</td>
</tr>
<tr>
<td>bbc</td>
<td>No</td>
<td>b</td>
</tr>
<tr>
<td>bca</td>
<td>No</td>
<td>a</td>
</tr>
<tr>
<td>bcb</td>
<td>No</td>
<td>b</td>
</tr>
<tr>
<td>bcc</td>
<td>No</td>
<td>b</td>
</tr>
<tr>
<td>caa</td>
<td>Yes</td>
<td>aa</td>
</tr>
<tr>
<td>cab</td>
<td>No</td>
<td>b</td>
</tr>
<tr>
<td>cac</td>
<td>No</td>
<td>b</td>
</tr>
<tr>
<td>cba</td>
<td>No</td>
<td>a</td>
</tr>
<tr>
<td>cbb</td>
<td>No</td>
<td>b</td>
</tr>
<tr>
<td>cbc</td>
<td>No</td>
<td>b</td>
</tr>
<tr>
<td>cca</td>
<td>No</td>
<td>a</td>
</tr>
<tr>
<td>ccb</td>
<td>No</td>
<td>b</td>
</tr>
<tr>
<td>ccc</td>
<td>No</td>
<td>b</td>
</tr>
</tbody>
</table>

Note that some of the sequences are irreducible, other can be reduced.

Any sequence that is legal and irreducible is called **canonical**

Note that once we have enumerated all sequences of length = 3, we are done, since they all have equivalent sequences that are shorter.
Other Refinements

- Identifying **equivalent sequences** (as we did in the example)
- **Abstractions** - Functions that shorten the stimulus histories
  - Avoid having to list out many repetitive stimuli
    - e.g., **10-Clock-pulses** - denotes a stimulus which is the 10th clock pulse
    - e.g., **phone-number** - denotes a fully-formed phone number
      > Abstract the primitive (atomic) inputs into compound data items

- **Specification functions**
  - Avoid having to provide detail on deterministic results by providing a function name for a set of responses
    - e.g., **sqrt, max(a_i), etc...**

But it gets better...
State Box

- Derived from the Black Box
  - Regards the stimulus history as a state
  - Based on the canonical sequences
- For each stimulus, use the black box function to compute the response from the stimulus history
- State box consists of a description of -
  - The **externally-visible behavior** of a specific SW component
  - Its **state variables**
- A map of inputs to states and outputs
  - Stimuli are mapped to state transitions and to responses
  - Expressing the same semantics as the Black Box but with some design information
- There are actually two stages to defining state boxes
  - **First**, identify the states based on the equivalent enumerations (still requirements)
  - **Second**, define state variables to capture the states (now some design)
State Box Definition

Specifies state data and transitions needed to achieve the behavior defined for the black box

where:
SB State box function
S Stimuli
R Response
T State space
Stage 1 - Identify State Machine

- Identify the stimulus histories that form equivalences
  - Canonical sequences
  - Denote them as states
- Sequences will have repeating patterns
  - I.e., two different input sequences will have the same response for the same input and for all future inputs
  - The different sequences end up in the same program state
- For each stimulus, define responses and state transition to define the same function as the enumerations
- Program behavior is then defined by current state and current input rather than stimuli history and current input

Note that this form of the State Box is still purely at the requirements level - expressing externally visible behavior
**Example**

Remember this table?

The canonical sequences now become our states

<table>
<thead>
<tr>
<th>( \sigma )</th>
<th>( \text{BB}(\sigma) )</th>
<th>\text{Equiv}</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \lambda )</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>ab</td>
<td>No</td>
<td>b</td>
</tr>
<tr>
<td>ac</td>
<td>No</td>
<td>b</td>
</tr>
<tr>
<td>ba</td>
<td>No</td>
<td>a</td>
</tr>
<tr>
<td>bb</td>
<td>No</td>
<td>b</td>
</tr>
<tr>
<td>bc</td>
<td>No</td>
<td>b</td>
</tr>
<tr>
<td>ca</td>
<td>No</td>
<td>a</td>
</tr>
<tr>
<td>cb</td>
<td>No</td>
<td>b</td>
</tr>
<tr>
<td>cc</td>
<td>No</td>
<td>b</td>
</tr>
<tr>
<td>aaa</td>
<td>Yes</td>
<td>aa</td>
</tr>
<tr>
<td>aab</td>
<td>Yes</td>
<td>aa</td>
</tr>
<tr>
<td>aac</td>
<td>Yes</td>
<td>aa</td>
</tr>
<tr>
<td>aba</td>
<td>No</td>
<td>a</td>
</tr>
<tr>
<td>abb</td>
<td>No</td>
<td>b</td>
</tr>
<tr>
<td>abc</td>
<td>No</td>
<td>b</td>
</tr>
<tr>
<td>aca</td>
<td>No</td>
<td>a</td>
</tr>
<tr>
<td>acb</td>
<td>No</td>
<td>b</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( \sigma )</th>
<th>( \text{BB}(\sigma) )</th>
<th>\text{Equiv}</th>
</tr>
</thead>
<tbody>
<tr>
<td>acc</td>
<td>No</td>
<td>b</td>
</tr>
<tr>
<td>baa</td>
<td>Yes</td>
<td>aa</td>
</tr>
<tr>
<td>bab</td>
<td>No</td>
<td>b</td>
</tr>
<tr>
<td>bac</td>
<td>No</td>
<td>b</td>
</tr>
<tr>
<td>bba</td>
<td>No</td>
<td>a</td>
</tr>
<tr>
<td>bbb</td>
<td>No</td>
<td>b</td>
</tr>
<tr>
<td>bbc</td>
<td>No</td>
<td>b</td>
</tr>
<tr>
<td>bca</td>
<td>No</td>
<td>a</td>
</tr>
<tr>
<td>bcb</td>
<td>No</td>
<td>b</td>
</tr>
<tr>
<td>bcc</td>
<td>No</td>
<td>b</td>
</tr>
<tr>
<td>caa</td>
<td>Yes</td>
<td>aa</td>
</tr>
<tr>
<td>cab</td>
<td>No</td>
<td>b</td>
</tr>
<tr>
<td>cac</td>
<td>No</td>
<td>b</td>
</tr>
<tr>
<td>cba</td>
<td>No</td>
<td>a</td>
</tr>
<tr>
<td>cbb</td>
<td>No</td>
<td>b</td>
</tr>
<tr>
<td>cbc</td>
<td>No</td>
<td>b</td>
</tr>
<tr>
<td>cca</td>
<td>No</td>
<td>a</td>
</tr>
<tr>
<td>ccb</td>
<td>No</td>
<td>b</td>
</tr>
<tr>
<td>ccc</td>
<td>No</td>
<td>b</td>
</tr>
</tbody>
</table>
### Example (cont’d)

A finite state machine

<table>
<thead>
<tr>
<th>Current State</th>
<th>Stimulus</th>
<th>Response</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>@</td>
<td>a</td>
<td>No</td>
<td>&lt;a&gt;</td>
</tr>
<tr>
<td>@</td>
<td>b</td>
<td>No</td>
<td>&lt;b&gt;</td>
</tr>
<tr>
<td>@</td>
<td>c</td>
<td>No</td>
<td>&lt;b&gt;</td>
</tr>
<tr>
<td>&lt;a&gt;</td>
<td>a</td>
<td>Yes</td>
<td>&lt;aa&gt;</td>
</tr>
<tr>
<td>&lt;a&gt;</td>
<td>b</td>
<td>No</td>
<td>&lt;b&gt;</td>
</tr>
<tr>
<td>&lt;a&gt;</td>
<td>c</td>
<td>No</td>
<td>&lt;b&gt;</td>
</tr>
<tr>
<td>&lt;b&gt;</td>
<td>a</td>
<td>No</td>
<td>&lt;a&gt;</td>
</tr>
<tr>
<td>&lt;b&gt;</td>
<td>b</td>
<td>No</td>
<td>&lt;b&gt;</td>
</tr>
<tr>
<td>&lt;b&gt;</td>
<td>c</td>
<td>No</td>
<td>&lt;b&gt;</td>
</tr>
<tr>
<td>&lt;aa&gt;</td>
<td>a</td>
<td>Yes</td>
<td>&lt;aa&gt;</td>
</tr>
<tr>
<td>&lt;aa&gt;</td>
<td>b</td>
<td>Yes</td>
<td>&lt;aa&gt;</td>
</tr>
<tr>
<td>&lt;aa&gt;</td>
<td>c</td>
<td>Yes</td>
<td>&lt;aa&gt;</td>
</tr>
</tbody>
</table>

@ = Initial state

A convenient renaming of the states

- One_a
- No_a
- Two_a
- No_a
- No_a
- No_a
- One_a
- No_a
- No_a
- No_a
- Two_a
- Two_a
- Two_a
A State Diagram

This is a complete definition of the functional behavior of this module

Note the convention we use here:
On each arc is an indication of the stimulus that causes the state transition, and the output that is produced.
Stage 2 - Identify State Variables

- At this stage, developers define variables that will capture the state of the machine.
- The State Box at this point becomes part of design.
  - Internal implementation detail
- For the example we could define three state variables:
  - Each with True / False values
    - No_a
    - One_a
    - Two_a
  - or we could define one state variable, with 3 values:
    - Num_a
      - = 0 if last symbol not an “a”
      - = 1 if last symbol is an “a” and symbol before last is not an “a”
      - = 2 if last 2 symbols are “a”s
Formalisms

- Derivation of state box is supported finite state automata theory
- Kleene’s Theorem
  - Any regular set can be recognized by some finite-state machine

$$SB : S \times T \rightarrow T \times R$$

where:
- SB: State box function
- S: Stimuli
- R: Responses
- T: State space

A set is regular if it can be defined by a regular expression, e.g. $S^*$

Here the stimuli history is captured in the current state
Discussion

Not so fast...

Not all input sets are as limited as regular expressions
Some are context-free, some are context-sensitive, and some are phrase-structure grammars
And these cannot be recognized by FSAs !!!

➢ So true, but consider …

By the way, are we losing anyone here? This may get a bit technical, but is included to satisfy the curiosity of some of the audience. We need to show just a tiny bit of the underlying theory. Trust us…this really works…
**Considerations**

- We need to define **all** possible inputs, not just **legal** inputs
  - If C is the set of all single inputs, then C* is the set of all possible input strings
    - Including legal and illegal
    - Even if the legal strings need to be defined by a context-free grammar
  - When defining input sequences, we partition the input space into classes of input strings
    - Legal, illegal, improbable, likely,... (looking ahead to usage models)

- Input strings can be defined in several ways, including:
  - **Regular expressions** - as we have seen
  - **Grammars** - such as for programming languages
    - These can be finite state, context free, context sensitive, or unlimited
    - *Note that creating a recursive descent parser is similar to the approach used by Cleanroom for developing programs*
      > The design is based on the input specification
Hard Problems

- Some problems we cannot solve
  - e.g., the halting problem

- Some may be hard to solve
  - NP-Complete
  - e.g. Hamilton’s problem - Given a graph, is there a path through the graph which visits each vertex precisely once? Is there a path which ends up where it started?

- What about input sequences that do not have patterns?
  - e.g., identifying irrational numbers

- These are all hard to solve using conventional techniques as well

- Many problems (as defined by their input strings) can be solved...thank goodness

- For interesting problems that are solvable, Cleanroom provides a powerful approach
Clear Box

Now that we have defined our State Box, what next?

Refine the State Box by defining the procedures that implement the functions

- May need to define lower-level Black Boxes to provide functionality

Within the Clear Boxes, we need to limit the control structures to the 7 structured programming constructs

- **Function** - f
- **Sequence** - f; g
- **If…then** - if p then f
- **If…then…else** - if p then f else g;
- **Whiledo** - while p do f ;
- **Do until** - do f until p;
- **Do…while…do** - do f while p do g ;
- Do wa diddy
Why?

- Why?
- Because these forms have well defined semantics that support formal verification
- These forms are prime programs
  - A prime program is a proper program that has no proper subprogram of more than one node
  - A proper program
    - has a single entrance and a single exit
    - all nodes have a path through that node from the entry to the exit
- The 7 listed are unique -
  - There are 15 possible prime programs containing 1 to 4 nodes
  - These 7 are the only ones that contain function nodes
    - i.e., can process data
So What?

- So each one has a single entry and a single exit
  - So no side effects and unintended consequences may occur
  - They have the form of a mathematical function
  - They supports referential transparency
- When we develop programs that contain only these 7 prime programs, we limit ourselves concerning side effects
- We also make things more scalable
  - Neat, well-defined building blocks
The Control Structures

sequence

if...then...

if...then...else
The Control Structures (cont’d)

- while...do

- do...until

- do...while...do
Intended Functions

- For each control structure, intended functions are attached.
- These express (as a comment) what the control structure is intended to do.
- For example:

\[
\text{intended function for the sequence should describe the function:} \quad h \left( g \left( i \right) \right)
\]

\[
\text{intended function for the if...then should describe the function:} \quad (p \rightarrow f = g \mid \neg p \rightarrow \text{nil})
\]
Code Generation

- Based on the state variables, control structures can be directly generated
  - Almost a mechanical process

- The form is

```plaintext
Procedure CB (inputs, outputs)
Declare state variables
Declare local variables
Determine state
Test for state
  Processing to be done in that state
Test for state
  Processing to be done in that state
Test for state
  Processing to be done in that state
Test for state
  Processing to be done in that state
```
State Box versus Clear Box

<table>
<thead>
<tr>
<th>Current State</th>
<th>Stimulus</th>
<th>Response</th>
<th>Next State</th>
<th>Num_a</th>
</tr>
</thead>
<tbody>
<tr>
<td>@</td>
<td>a</td>
<td>No</td>
<td>@</td>
<td>1</td>
</tr>
<tr>
<td>@</td>
<td>b</td>
<td>No</td>
<td>@</td>
<td>0</td>
</tr>
<tr>
<td>@</td>
<td>c</td>
<td>No</td>
<td>@</td>
<td>0</td>
</tr>
<tr>
<td>@</td>
<td>a</td>
<td>Yes</td>
<td>@</td>
<td>2</td>
</tr>
<tr>
<td>@</td>
<td>b</td>
<td>No</td>
<td>@</td>
<td>0</td>
</tr>
<tr>
<td>@</td>
<td>c</td>
<td>No</td>
<td>@</td>
<td>0</td>
</tr>
<tr>
<td>@</td>
<td>a</td>
<td>No</td>
<td>@</td>
<td>1</td>
</tr>
<tr>
<td>@</td>
<td>b</td>
<td>No</td>
<td>@</td>
<td>0</td>
</tr>
<tr>
<td>@</td>
<td>c</td>
<td>No</td>
<td>@</td>
<td>0</td>
</tr>
<tr>
<td>@</td>
<td>a</td>
<td>Yes</td>
<td>@</td>
<td>2</td>
</tr>
<tr>
<td>@</td>
<td>b</td>
<td>Yes</td>
<td>@</td>
<td>2</td>
</tr>
<tr>
<td>@</td>
<td>c</td>
<td>Yes</td>
<td>@</td>
<td>2</td>
</tr>
</tbody>
</table>

If Curr_Symb = “a” THEN
    Num_a := Num_a + 1;
    If Num_a > 2 THEN
        Num_a := 2;
    END IF;
ELSE
    Num_a := 0;
END IF;

IF Num_a = 0 THEN
    Put (“No”);
ELSIF Num_a = 1 THEN
    Put (“No”);
ELSIF Num_a >= 2 THEN
    Put (“Yes”);
END IF;
Summary of Box Structure Expansion Process

Eleven steps -

- **Black box definition**
  1. Define all possible stimuli and stimuli histories
  2. For each, define the required response of the black box

- **State box definition**
  3. Correlate for each response, the stimulus histories that cause that response
  4. Identify state data to define each state
  5. Identify the internal black boxes needed to create the state box
  6. Verify that the state box is correct relative to the black box definition

- **Clear box design**
  7. For each state datum, identify where it is referenced
  8. Define data abstractions for each state datum
  9. Define processing to replace internal black boxes
  10. Verify the clear boxes

- **Iteration**
  11. Repeat until complete
Key Features - Formal reviews/ inspections to verify correctness

- Incremental development process
- Formal specification and hierarchical design techniques based on Box Structures
- Formal reviews/ inspections to verify correctness
- Statistical quality control
Verification of Correctness

- Formal reviews/inspections to verify correctness
  - Proofs based on design structure rather than embedded logic
  - Made possible by restricting Clear Boxes to Structured Programming constructs, each with a defined semantics
  - Correctness for product inferred from correctness of each component

- Performed at each step of the decomposition
  - After defining the State Box for a Black Box, they must be shown to be equivalent
    - I.e., compute the same function
    - Technique - demonstrate that the state machine captures all stimuli sequences with the proper responses
  - After defining the Clear Box for a State Box, they must be shown to be equivalent
Basic Premise

- Verify that a program is equivalent to its intended function
  - Decomposed to proving correctness of parts and then by inference the whole
  - Based on Prime Programs
- Each control structure is abstracted to its function
  - and then compared to the intended function
- A specific set of questions are posed at each level, based on the specific control structure
  - Defined by Linger, Mills, and Witt in 1979
- The correctness demonstration proceeds from the prime programs upwards to the component level
- Since the code format strictly adheres to the state machine structure, the abstraction can be easily compared to the original state machine
- The process is effective even when performed semi-formally
  - Provides a focus to the formal inspection process
Verification Process Flow

\[ f = \text{Specification} \]

1. **Development**
2. **Abstraction**
3. **Compare**

Program function = \([P]\)
Sample Correctness Questions

Sequence

[ f ]
g;
h;

Composition question
Does \( h \left( g \left( i \right) \right) = f \)?

If...then

[ f ]
if
  if
    p
  then
    g
end if

Composition question
If \( p \) is true, does \( p \left( i \right) = f \)?
If \( p \) is false, does \( g(i) = f \)?
Key Features - Statistical Quality Control

- Incremental development process
- Formal specification and hierarchical design techniques based on Box Structures
- Formal reviews/inspections to verify correctness
- Statistical quality control
Certification

- Statistical quality control
  - Achieved through reliability testing on increments
  - Supported by creation of Usage Models in the form of Markov Chains
  - Test data created by random sampling from the Usage Models
  - Results provide prediction of expected MTTF and of stopping point

- Based on statistical testing

- Each increment is certified by challenging it with test data selected from its operational profile
Usage Models

- A model of how the system is to be used
  - Operational profiles

- Based on the State Boxes
  - Augmented by probabilities for each input / state transition
  - Markov chain

- Method -
  - Assign to each input a probability of occurrence corresponding to expected usage
  - Create separate models for classes of users / modes of usage / environment
    - Process called Stratification
  - Some failures are more critical than others
    - Stratification helps to focus on the more critical ones
### Example

Differences in user strata

<table>
<thead>
<tr>
<th>Current State</th>
<th>Stimulation</th>
<th>Response</th>
<th>Next State</th>
<th>Naïve User Probability</th>
<th>Expert Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>@</td>
<td>a</td>
<td>No</td>
<td>&lt;a&gt;</td>
<td>0.2</td>
<td>0.5</td>
</tr>
<tr>
<td>@</td>
<td>b</td>
<td>No</td>
<td>&lt;b&gt;</td>
<td>0.3</td>
<td>0.3</td>
</tr>
<tr>
<td>@</td>
<td>c</td>
<td>No</td>
<td>&lt;b&gt;</td>
<td>0.5</td>
<td>0.2</td>
</tr>
<tr>
<td>&lt;a&gt;</td>
<td>a</td>
<td>Yes</td>
<td>&lt;aa&gt;</td>
<td>0.1</td>
<td>0.5</td>
</tr>
<tr>
<td>&lt;a&gt;</td>
<td>b</td>
<td>No</td>
<td>&lt;b&gt;</td>
<td>0.3</td>
<td>0.3</td>
</tr>
<tr>
<td>&lt;a&gt;</td>
<td>c</td>
<td>No</td>
<td>&lt;b&gt;</td>
<td>0.6</td>
<td>0.2</td>
</tr>
<tr>
<td>&lt;b&gt;</td>
<td>a</td>
<td>No</td>
<td>&lt;a&gt;</td>
<td>0.2</td>
<td>0.5</td>
</tr>
<tr>
<td>&lt;b&gt;</td>
<td>b</td>
<td>No</td>
<td>&lt;b&gt;</td>
<td>0.3</td>
<td>0.3</td>
</tr>
<tr>
<td>&lt;b&gt;</td>
<td>c</td>
<td>No</td>
<td>&lt;b&gt;</td>
<td>0.5</td>
<td>0.2</td>
</tr>
<tr>
<td>&lt;aa&gt;</td>
<td>a</td>
<td>Yes</td>
<td>&lt;aa&gt;</td>
<td>0.2</td>
<td>0.1</td>
</tr>
<tr>
<td>&lt;aa&gt;</td>
<td>b</td>
<td>Yes</td>
<td>&lt;aa&gt;</td>
<td>0.3</td>
<td>0.4</td>
</tr>
<tr>
<td>&lt;aa&gt;</td>
<td>c</td>
<td>Yes</td>
<td>&lt;aa&gt;</td>
<td>0.5</td>
<td>0.5</td>
</tr>
</tbody>
</table>
How Do You Know?

- There are several techniques for selecting probabilities of stimuli
  - History
    - Based on experience with a previous system
  - Projected usage
    - Based on models of how the system will be used after deployment
    - e.g., Object Oriented Use cases and scenarios
  - Measurement
    - Based on measures of usage obtained from early increments
  - Approximation
    - Equally allocate probability to all alternatives
  - Continual refinement through each increment provides a converging set of estimates
Statistical Testing Rationale

- Reliability of SW “in the field” depends on two factors:
  - Location of defects in the code
  - Usage profile for the SW
    » Which dictates what parts of the code will be executed
- Defects are not, by themselves, harmful
  - They cause problems only when they are executed
    » And sometimes not even then!
- Goal of statistical testing is to test the software under the same conditions that it will experience when “in service”
  - Its behavior will then be similar to what users should experience after release
- Test cases are based on a random sample taken from the usage model
  - If usage model is accurate, random sample will closely approximate deployment experience
Data Collection

- When performing statistical testing using Markov chains, certain histories need to be kept
  - Number of states visited during test
  - Number of arcs visited during test
- Goal is to at least visit every state and every arc
- Also keep track of pass/fail histories over increments
  - Support the computation of stopping criteria
- Based on testing histories, reliability estimates can be computed
  - MTTF
  - Approximate but provide assessment of software behavior during deployment
**Defects and Reliability**

- Consider the following program
  
  ```ada
  PROCEDURE Sample (I : Integer) IS
    j : Integer;
  BEGIN
    Get (I);
    IF (I < 0) THEN
      xx;
    ELSIF (I < 1 THEN
      yy;
    ELSIF (I < 5) THEN
      zz;
    ELSE
      aa;
    END IF;
  END;
  
  Suppose that this program has defects in all 3 THEN clauses
  - xx, yy, and zz
  - Defect density = 3 defects/14 SLOC or 214 defects / kSLOC

- Suppose that when used, I is **always** > 5
  - No failures will be experienced
  - Reliability = 1.00
Defects and Reliability (cont’d)

- Consider the following program

```ada
PROCEDURE Sample (I : Integer) IS
  j : Integer;
BEGIN
  Get (I);
  IF (I < 0) THEN
    xx;
  ELSIF (I < 1 THEN
    yy;
  ELSIF (I < 5) THEN
    zz;
  ELSE
    aa;
  END IF;
END;
```

- Suppose that this program has a defect in aa
  - Defect density = 1 defects/1,013 SLOC or 0.99 defects / kSLOC

- Suppose that when used, I < 0 once per 1000 invocations (or once per hour) and I > 5 the rest of the time
  - The component will fail almost every time, and succeed once per hour
  - Reliability = 1 failure every 3.6 sec
Agenda

- Introduction and Overview
- Basic Principles
- History of Cleanroom
- Experience using Cleanroom (case studies)
- Foundations of Cleanroom
- Cleanroom Process Model
- Cleanroom and the CMM
- Cleanroom and Object-Oriented Development
- Conclusion
Cleanroom Process Model

- In 1996, the SEI released a Cleanroom Software Engineering Reference Model (CRM)

- The CRM is intended to serve as a guide for projects which intend to use Cleanroom by defining the life cycle model

- The CRM does not provide instruction in how to apply Cleanroom
  - Technical details, specific techniques,…

- This section of the tutorial summarizes CRM
  - Using material extracted from the Technical Report
Cleanroom Life Cycle Phases

- Requirements analysis - formal definition of product behavior
- Increment definition - sequence of requirements to be built into successive builds (increments)
- Black box definition - formal definition of product for current increment
- State box definition - identification of product states based on black box
- Clear box development - development of code
- Clear box verification - formal proof of correctness of the clear box
- Certification - statistical testing of the increment
# The 14 Cleanroom Processes

## Cleanroom Management Processes
- Project Planning Process
  - Cleanroom Engineering Guide
  - Software Development Plan
- Project Management Process
  - Project Record
- Performance Improvement Process
  - Performance Improvement Plan
- Engineering Change Process
  - Engineering Change Log

## Cleanroom Specification Processes
- Requirements Analysis Process
  - Software Requirements
  - Function Specification
- Usage Specification Process
  - Usage Specification
- Architecture Specification Process
  - Software Architecture
- Increment Planning Process
  - Increment Construction Plan

## Cleanroom Development Processes
- Software Reengineering Process
  - Reengineering Plan
  - Reengineered Software
- Increment Design Process
  - Increment Design
- Correctness Verification Process
  - Increment Verification Report

## Cleanroom Certification Processes
- Usage Modeling and Test Planning Process
  - Usage Models
  - Increment Test Plan
  - Statistical Test Cases
- Statistical Testing and Certification Process
  - Executable System
  - Statistical Testing Report
  - Increment Certification Report
Cleanroom Process Flow

Cleanroom Management Processes are concurrent across entire life cycle

Architecture Specification Process is concurrent across entire life cycle
Cleanroom Requirements Analysis is similar to Waterfall Requirements Analysis

Cleanroom Function Specification carries RA further by defining system level Black, State, and Clear Boxes which capture the detailed behavior plus the architectural design.

Cleanroom BB-SB-CB-Validation corresponds to Waterfall Detailed Design, Code/Unit Test, and some of Integration and Test.

Cleanroom Certification contains elements of Waterfall Integration and Test and FQT.
Cleanroom Management Processes

- **Project Planning**
  - Development of detailed plans for how the project will be conducted

- **Project Management**
  - Manage all activities involved with the Cleanroom Development

- **Performance Improvement Process**
  - On a continual basis
    - Monitor the development activities as well as key performance attributes
    - Identify shortfalls and failures
    - Determine causes
    - Implement process enhancement

- **Engineering Change**
  - Configuration management functions
  - Control and manage changes and updates to the work products
Architecture Specification

- This process is performed on a continuing basis
- Define the strategy and guidelines under which the architecture for the product will be developed
- Monitor architectural development and feedback improvements in a controlled manner
- Identify and monitor use of legacy, COTS, and NDI software resources
Architectural Concept

Inputs

System Functional Requirements

CSCI A

CSCI B

CSCI C

CSCI D

CSCI E

CSCI F

Hardware

Outputs

Capabilities

CSCI A Requirements

CSCI B Requirements

CSCI C Requirements

CSCI D Requirements

CSCI E Requirements

CSCI F Requirements

HW Requirements

CSCI A

CSCI B

CSCI C

CSCI D

CSCI E

CSCI F
Requirements Analysis

- This activity is similar to the “traditional” requirements analysis phase
- Goal is to capture
  - Capabilities to be provided
  - Functions to be implemented
  - Necessary performance (temporal and capacity)
  - Usage profile (use cases and scenarios)
  - The environment in which the system will be used
- Requirements are documented in a Software Requirements document
- Interaction with the customer is crucial to ensure acceptance
- Quality criteria:
  - Complete
  - Consistent
  - Feasible
  - Testable
Function Specification

- For most techniques, usually a part of requirements analysis
- The goals for Cleanroom are more ambitious
  - Requires more detail about the capabilities/functions
  Specify, the complete functional behavior of the software in all possible circumstances of use
  Mathematically precise, complete, and consistent
- Obtain agreement with the customer on the specified function
  
  But wait - we already do this as a part of requirements analysis!! Right?
- Not really, and not consistently
- There is a wide range of practice in developing software requirements documents
  - Current approaches tend to fall between capability descriptions and functional behaviors
Levels of Requirements

- Requirements can be specified at various levels of detail
  - Very general - e.g., “The product shall allow users to perform word processing”
  - to
  - Very detailed - “Pressing Ctrl and I at the same time results in the selected text being converted to an italics font within 0.5 sec”

- Regardless of the level and amount of detail, as long as the descriptions address external behavior, they are still requirements and not design

- When developing a system, requirements typically start at the general level
  - List of capabilities - what the users want to be able to do

- As development proceeds, the general requirements are refined until they become specific behaviors

or should be
Degrees of Specificity

General statement of intent
Word processing

Specific functional behavior
Handle italics
Use Ctrl-I for italicization
Each alternative requirements set represents a different approach to providing the capabilities.

The requirements analysis process involves defining some alternative sets of requirements, and choosing one.

The customer needs to verify that the selected alternative satisfies his needs.

Once an alternative is selected, the customer agrees that implementing that alternative will satisfy his needs.

When implemented, acceptance testing needs to focus on the implemented requirements. If successfully implemented, the capabilities are *de facto* provided.
“What” versus “How”

Yes, but the capabilities tell me what the customer wants, and the details tell me how, which is really design.

- “What” and “how” are misleading clichés
  - Often used to differentiate requirements from implementation
  - *What* “implies” requirements
  - *How* “implies” design

- Often confusing, certainly inaccurate

- Problems:
  - Fail to recognize hierarchical nature of requirements/design
    - Designs involve functional requirements of system components
  - Fail to recognize colloquial English
    - “*How* does the user interface provide *help* information?” - requirements
    - “*What* tasking structure does this component use?” - design
Externally Visible Behavior

- Cleanroom takes a more precise view of requirements
- Requirements cover all externally-visible behavior
  - Include functions, screen formats, GUIs, menus, interfaces, etc.
- Cleanroom defines the detailed behavioral requirements in a functional specification using a formal, mathematical approach
  - Considers every possible input, every possible user scenario
The approach taken by Cleanroom for testing is based on challenging the increments with tests based on how the software will be used in deployment

- Statistical testing

The Usage Specification activity develops the models of expected behavior
Increment Planning

- Define the phased implementation of requirements (functions) into successive incremental builds
- Coordinate with the customer to ensure a consistent and useful progression
- Ensure that components preserve referential transparency
- Design increments so that increasing functions are coordinated to permit a smooth architecture enhancement
- Guarantee that increments are
  - Small enough to facilitate a quality and efficient development
  - Large enough to represent effective plateaus
Software Reengineering

- This process is used when reused software is to be incorporated into the product
  - COTS
  - NDI
  - Legacy
- The purpose is to ensure that the semantics are appropriate for the design
- Restructure to ensure referential transparency if necessary
- Create BB level enumerations (or the equivalent) to support verification
- Incorporate into increments to ensure smooth enhancement
Correctness Verification

- Perform the verifications required
  - Verify that all State Boxes are equivalent to their predecessor Black Boxes
  - Verify that all Clear Boxes are equivalent to their predecessor State Boxes

- These are generally performed as formal inspections via peer reviews

- All defects are to be documented and traced to ensure correction
Usage Modeling and Test Planning

- Prepare Usage Models for each component
- Verify that the Usage Models are equivalent to the State Boxes relative to semantics
- Ensure that the probabilities are reasonable and based on appropriate criteria
- Construct usage strata to cover all expected modes of use of the product
- Develop a statistical test plan
Statistical Testing and Certification

- Create test cases by taking random samples over the Usage Models
- Apply the test cases to the increments and verify correct (or incorrect) results
- Compute the expected reliability
- Determine how much more testing is required to achieve the desired level of reliability and quality
Agenda

- Introduction and Overview
- Basic Principles
- History of Cleanroom
- Experience using Cleanroom (case studies)
- Foundations of Cleanroom
- Cleanroom Process Model
- Cleanroom and the CMM
- Cleanroom and Object-Oriented Development
- Conclusion
Cleanroom and the CMM

- The Capability Maturity Model was developed by the Software Engineering Institute to aid in assessing the level of maturity of an organization’s software development processes.
- It consists of a set of 18 key process areas (KPAs) and 5 levels of maturity.
- Organizations are assessed against these KPAs and are awarded a maturity level based on the assessment.
- In 1996, the SEI mapped the 18 KPAs to the 14 Cleanroom processes.

Why?

*If a project adopts Cleanroom, the mapping assists in determining the CMM level of maturity that can be expected.*

*It also shows the extent to which Cleanroom covers development activities (and where you need to augment)*

CMM - Five Levels of Maturity

For reference, the five levels are:

<table>
<thead>
<tr>
<th></th>
<th>Level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Initial</td>
<td>The software process is characterized as ad hoc, and occasionally even chaotic. Few processes are defined, and success depends on individual effort and heroics.</td>
</tr>
<tr>
<td>2</td>
<td>Repeatable</td>
<td>Basic project management processes are established to track cost, schedule, and functionality. The necessary process discipline is in place to repeat earlier successes on projects with similar applications.</td>
</tr>
<tr>
<td>3</td>
<td>Defined</td>
<td>The software processes for both management and engineering activities are documented, standardized, and integrated into a standard software process for the organization. All projects use an approved, tailored version of the organization’s standard software process for developing and maintaining software.</td>
</tr>
<tr>
<td>4</td>
<td>Managed</td>
<td>Detailed measures of the software process and product quality are collected. Both the software process and products are quantitatively understood and controlled.</td>
</tr>
<tr>
<td>5</td>
<td>Optimizing</td>
<td>Continuous process improvement is enabled by quantitative feedback from the process and from piloting innovative ideas and technologies.</td>
</tr>
</tbody>
</table>
For reference, the KPAs are correlated to the 5 levels as follows:

<table>
<thead>
<tr>
<th>Level 2</th>
<th>Level 3</th>
<th>Level 4</th>
<th>Level 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirements Management (RM)</td>
<td>Organization Process Focus (PF)</td>
<td>Quantitative Process Management (QP)</td>
<td>Defect Prevention (DP)</td>
</tr>
<tr>
<td>Software Project Planning (PP)</td>
<td>Organization Process Definition (PD)</td>
<td>Management (QM)</td>
<td>Management (TM)</td>
</tr>
<tr>
<td>Software Project Tracking and Oversight (PT)</td>
<td>Training Program (TP)</td>
<td>Integrated Software Management (IM)</td>
<td>Process Change Management (PC)</td>
</tr>
<tr>
<td>Software Subcontract Management (SM)</td>
<td>Software Product Engineering (PE)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Software Quality Assurance (QA)</td>
<td>Intergroup Coordination (IC)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Software Configuration Management (CM)</td>
<td>Peer Reviews (PR)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# Levels of Correlation

- Correlation is scored in 5 levels
  - From **highly correlated** to low correlation to **not included**

<table>
<thead>
<tr>
<th>Correspondence Category</th>
<th>Keyword</th>
</tr>
</thead>
<tbody>
<tr>
<td>The CMM element is consistent with Cleanroom processes, and implementation is high.</td>
<td><strong>High (H)</strong></td>
</tr>
<tr>
<td>The CMM element is consistent with Cleanroom processes, and implementation is partial.</td>
<td><strong>Partial (P)</strong></td>
</tr>
<tr>
<td>The CMM element is consistent with Cleanroom processes, and implementation is low.</td>
<td><strong>Low (L)</strong></td>
</tr>
<tr>
<td>The CMM element is consistent with Cleanroom processes, but is not implemented in the</td>
<td>Consistent (C)</td>
</tr>
<tr>
<td>Cleanroom processes, or is implemented in an indirect way.</td>
<td></td>
</tr>
<tr>
<td>The CMM element is not included in Cleanroom processes, and an alternative implementation is defined by Cleanroom processes.</td>
<td>Alternative (A)</td>
</tr>
</tbody>
</table>
### Level 2 KPAs: Repeatable

<table>
<thead>
<tr>
<th>Key Process Area</th>
<th>Cleanroom Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirements Management</td>
<td>High</td>
</tr>
<tr>
<td>Software Project Planning</td>
<td>High</td>
</tr>
<tr>
<td>Software Project Tracking &amp; Oversight</td>
<td>High</td>
</tr>
<tr>
<td>Software Subcontract Management</td>
<td>Consistent</td>
</tr>
<tr>
<td>Software Quality Assurance</td>
<td>Partial</td>
</tr>
<tr>
<td>Software Configuration Management</td>
<td>Partial</td>
</tr>
</tbody>
</table>

### Level 3 KPAs: Defined

<table>
<thead>
<tr>
<th>Key Process Area</th>
<th>Cleanroom Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Organization Process Focus</td>
<td>Consistent</td>
</tr>
<tr>
<td>Organization Process Definition</td>
<td>Partial</td>
</tr>
<tr>
<td>Training Program</td>
<td>Partial</td>
</tr>
<tr>
<td>Integrated Software Management</td>
<td>High</td>
</tr>
<tr>
<td>Software Product Engineering</td>
<td>High</td>
</tr>
<tr>
<td>Intergroup Coordination</td>
<td>High</td>
</tr>
<tr>
<td>Peer Reviews</td>
<td>High</td>
</tr>
</tbody>
</table>
## Cleanroom and the CMM - The Results (cont’d)

### Level 4 KPAs: Managed

<table>
<thead>
<tr>
<th>Key Process Area</th>
<th>Cleanroom Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quantitative Process Management</td>
<td>High</td>
</tr>
<tr>
<td>Software Quality Management</td>
<td>High</td>
</tr>
</tbody>
</table>

### Level 5 KPAs: Optimizing

<table>
<thead>
<tr>
<th>Key Process Area</th>
<th>Cleanroom Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Defect Prevention</td>
<td>High</td>
</tr>
<tr>
<td>Technology Change Management</td>
<td>Partial</td>
</tr>
<tr>
<td>Process Change Management</td>
<td>Partial</td>
</tr>
</tbody>
</table>
Cleanroom and the CMM - Conclusions

- The Cleanroom processes cover most of the important aspects of software development
  - as shown by the number of KPAs covered by Cleanroom
    - 10 of 18 have **High** correlation
    - 6 of 18 have **Partial** correlation
    - 2 have **Consistent** correlation (not implemented)
  - This suggests that Cleanroom is not a scattered and isolated technique
    - It focuses on the key processes

- If a project adopts all Cleanroom processes "faithfully", then the project has fulfilled most of the KPAs needed for CMM 4
  - And has partially satisfied CMM 5 needs
Agenda

- Introduction and Overview
- Basic Principles
- History of Cleanroom
- Experience using Cleanroom (case studies)
- Foundations of Cleanroom
- Cleanroom Process Model
- Cleanroom and the CMM
- **Cleanroom and Object-Oriented Development**
- Conclusion
Cleanroom and Object-Oriented Development

- After looking at Cleanroom, an initial observation might be:
  
  Nice, but clearly a structured analysis, functional decomposition style of design, unfortunately out of date compared to a modern, highly-effective object oriented development method.

- Philosophy aside, there is more correlation to OO that would appear in the surface

- The basic concept of a Black Box is similar to that of a class or object
  
  - State data and behavior encapsulated and managed
  - Internal data hidden from the outside
  - Functions achieved only through defined interfaces (methods)
  - Abstract Data Type theory underlies both Cleanroom and OO
OO / Cleanroom Similarities

- OO inheritance similar to increment growth
  - The concept of incremental development closely parallels OO inheritance
  - Refining component behavior can be accomplished via subclassing
    » Using inheritance to avoid recoding or redesigning
  - Necessary to ensure semantic consistency however
    » So subclassing should avoid redefining existing methods

- Use of state machines supported in UML
  - A consistent view across methods

- OO use cases and scenarios provide direct support to the creation of Usage Models

- An extensive study was performed blending OO methods with Cleanroom
  - [http://source.asset.com/stars/oral/cleanroom/oo/guidhome.htm](http://source.asset.com/stars/oral/cleanroom/oo/guidhome.htm)
Agenda

- Basic Principles
- Introduction and Overview
- History of Cleanroom
- Experience using Cleanroom (case studies)
- Foundations of Cleanroom
- Cleanroom Process Model
- Cleanroom and the CMM
- Cleanroom and Object-Oriented Development
- Conclusion
Conclusion

- Cleanroom is an effective and powerful technique that supports the development of quality and reliable software

- Any questions......
References


WEB Pages on Cleanroom

  - Cleanroom tutorial developed for The DoD Software Technology for Adaptable, Reliable Systems (STARS) program
  - Roger Pressman links on Cleanroom
  - Cleanroom SW Eng reference model
- http://www.cleansoft.com/
  - Cleanroom Software Engineering Inc. home page
  - DACS Cleanroom page
- http://source.asset.com/stars/loral/cleanroom/oo/guidhome.htm
  - Guide to integrating Cleanroom and OO
- http://www.sei.cmu.edu/str/descriptions/cleanroom.html
  - Cleanroom description