# Cleanroom Software Engineering

- Harlan Mills (Linger, Dyer, Poore), IBM, 1980
- Analogy with electronic component manufacture
- Use of statistical process control features
- Certified software reliability
- Improved productivity; zero defects at delivery

# Key Features

- Usage scenarios; statistical modeling
- Incremental development and release
- Separate development and acceptance testing
- No unit testing or debugging
  - Instead, formal reviews with verification conditions

# Cleanroom Projects

**Table 1.**
**Selected sample of Cleanroom projects.**
**(All other projects known to authors report substantial improvements in quality and productivity.)**

| Year | Applied technologies | Implementation | Results |
|---|---|---|---|
| 1980 | Stepwise refinement<br>Functional verification | Census, 25 KLOC (Pascal) | • No failure ever found<br>• Programmer received gold medal from Baldridge |
| 1983 | Functional verification<br>Inspections | Wheelwriter, 63 KLOC, three processors | • Millions of users<br>• No failure ever found |
| 1980s | Functional verification<br>Inspections | Space shuttle, 500 KLOC | • Low defect over entire function<br>• No defect in any flight<br>• Work received NASA's Quality Award |
| 1987 | Cleanroom engineering | Flight control, 33 KLOC (Jovial), three increments | • Completed ahead of schedule<br>• 2.5 errors/KLOC before any execution<br>• Error-fix effort reduced by a factor of five |
| 1988 | Cleanroom engineering | Commercial product, 80 KLOC (PL/I) | • Certification testing failure rate of 3.4 failures/KLOC<br>• Deployment failures of 0.1/KLOC<br>• Productivity of 740 lines/man-month |
| 1989 | Partial Cleanroom engineering | Satellite control, 30 KLOC (Fortran) | • Certification testing error rate of 3.3 failures/KLOC<br>• 50-percent improvement in quality<br>• Productivity of 780 lines/man-month<br>• 80-percent improvement in productivity |
| 1990 | Cleanroom engineering with reuse and new Ada design language | Research project, 12 KLOC (Ada and ADL) | • Certified to 0.9978 with 989 test cases; 36 failures found during certification (20 logic errors, or 1.7 errors/KLOC |

# Defect Rates

- Traditional
  - Unit testing: 25 faults / KLOC
  - System testing: 25 / KLOC
  - Inspections: 20 - 50 / KLOC
- Cleanroom
  - < 3.5 / KLOC delivered
  - Average 2.7 / KLOC between first execution and delivery

# Basic Technologies

1. Incremental Development
2. Box-Structured Specification
3. Function-theoretic verification
4. Statistical usage testing

# 1. Incremental Development

- Typical system < 100KLOC
- Increment:  2 - 15KLOC
- Team size < 14
- Each increment *End-to-End*
- Overlapped development of increments
- 12 - 18 weeks from beginning of specification to end of test
- Partitioning is difficult and critical

# 2. Formal Specification

- Box-structured design
  – Black box: stimulus-response
  – State box: formal model of system state
  – Clear box: hierarchical refinement
- Program functions
- Verification properties of control structures

# Box-Structured Specification and Design

- **Black Box**: stimulus / condition / response; organized into tasks; Z has been used for specification; top-down, stepwise refinement; concurrency supported
- **State Box**: data / history view; model oriented
- **Clear Box**: procedural control (sequence, alternation, iteration, concurrent; contains nested black boxes)
- Box Definition language

# State Boxes
## (Model-based Formal Specification)

- Description of system state in terms of *domains* (data structures without memory limitations
  - Sets, sequences, records, lists, maps, relations
- Specification of state *invariant*
- Specification of operations
  - Name
  - Arguments with domains
  - Validity condition (*precondition*)
  - Effect on state (*postcondition*)
- Each operation must maintain the invariant

# 3. Function-Theoretic Verification

- In Cleanroom, constructed programs can be checked by a parser for syntax errors, but may not be executed by the developer
  - No debugging $\Rightarrow$ cheap and predictable
- Verification is performed by a team review driven by a set of *verification conditions*
  - Questions to ask about the program code
  - Specific questions are asked about each kind of control structure
- Productivity: 3 - 5 x improvement in verification over debugging

# Formal Inspections

- Although program proving is always an option, this involves intensive work requiring mathematical sophistication
- An alternative, used by Cleanroom software engineering, is to structure a team code inspection in terms of program functions and verification conditions and then undertake an informal review confirming all verification conditions are satisfied

# Functional Verification Steps

1. Starting condition: program is specified by pre and post conditions
2. Program is parsed into prime programs
   - *Prime program decomposition*: parse program control flow into nested single entry/exit constructs (SESEs)
   - Usual SESEs are sequence, conditional, iteration
3. Proceeding top down, determine the program function for all SESEs
   - *Program function*: Description of the function of a prime program
   - Assertion placed before and after each SESE
4. Define verification conditions for each program point
   - *Verification Conditions*: things to check for each SESE
5. Inspect, answering all verification conditions

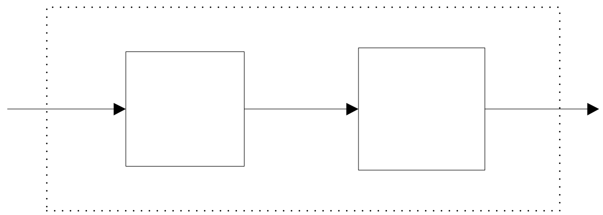
# Program Function

- Conditions under which the program can legally execute (preconditions)
- Expression of the effect of program execution on the state of the system (postconditions)
- Expressed in terms of the program's input arguments, return value, instance variables, global variables, and side effects on the environment (disk writes, printing, etc.) but not local program variables

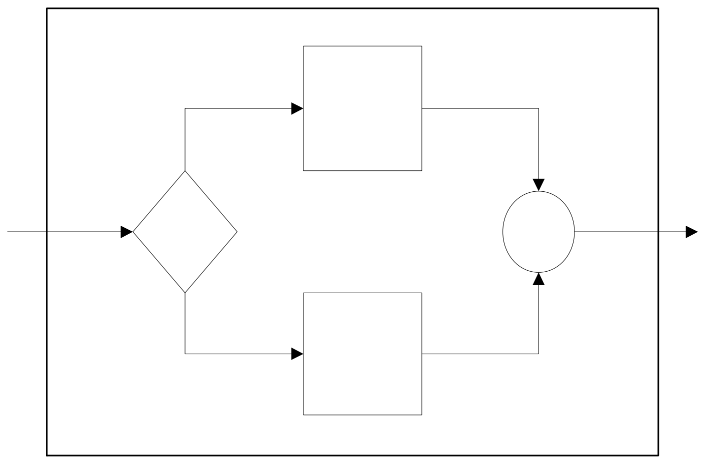
# Program Parse

- Modern programming languages support the concept of nested blocks
  - A block is normally enclosed in braces or keyword pairs (`begin-end`)
- In structured programs (programs without `GOTO` statements), the nesting is always well formed
  - That is, there is only ever one way for control to enter the block and one way to exit. That is, they have the property of being *single-entry, single exit* (SESE)
  - Programs with `GOTO`s can be handled using special methods
- The process of determining the SESEs for a program involves parsing its control flow graph.

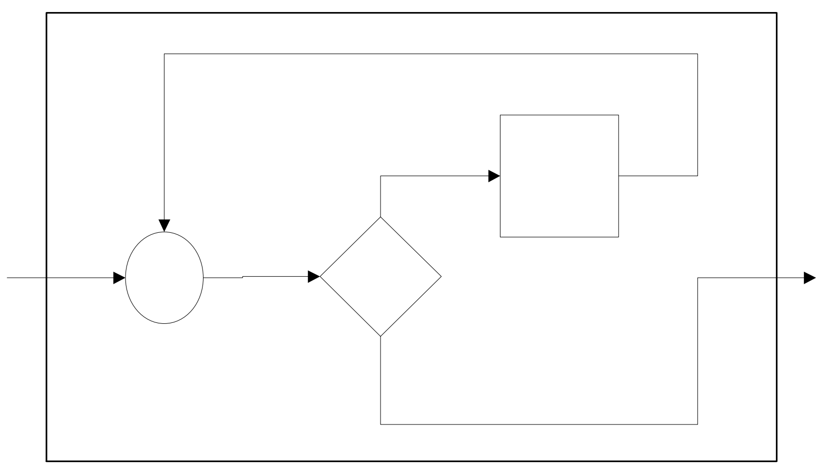
# Typical SESEs



Sequential Composition



Conditional Composition



Iterative Composition

# Composition of SESEs

- Each SESE can be thought of as being itself a small program with its own program function
- The overall program function is the logical composition of the program functions of its constituent SESEs
- The lowest level SESE is the single assignment statement

# Verification Conditions

- If we were proving a program correct, we would construct the proof by composing the proofs of each of the SESEs
- Instead of a proof, Cleanroom uses an informal review that examines each program statements to determine its logical validity
- In particular, each type of statement has a set of questions that should be asked about it every time that it occurs in the program
- There are three ways of composing SESEs
  - Sequence, conditional and iteration

# Sequence

- The simplest control structure is a sequence of two other statements or control structures
- There is one verification condition per sequence:
  – Do the constituent statements together accomplish the sequence's goal?
- This idea can readily be extended to three or more constituent statements

# Sequential Composition

1. Is the post assertion of the sequence equivalent to the logical composition of the first part followed by second part?

# Conditional

- An `if-then-else` has two arms
  - Does each arm acting by itself accomplish the control structure's post condition, assuming the control structure's precondition and that the tested condition is true (or false)?
- `If-then` is treated as `if-then-else` with a null arm

# Conditional

2. Does taking the `true` branch imply the post assertion?
   – The predicate of the conditional can be assumed to be `true`

3. Does taking the `false` branch imply the post assertion?
   – The predicate can be assumed to be `false`

# Iteration

- There are three questions to ask about an iterative construct such as a while loop:
  - Does it terminate in all circumstances?
  - Does it accomplish its purpose when it does not execute?
  - Does it accomplish its purpose when its body is executed followed by its own execution?
- `for` loops and `repeat` loops can be defined in terms of `while` loops

# Iteration

4. Does the loop terminate?

5. If the predicate is `false`, is the post assertion equivalent to the pre assertion?

6. If the predicate is `true`, is the post assertion of the loop equivalent to the post assertion of the body followed by the post assertion of the loop?
   - Recursive!
   - You may assume the predicate is `true`

# Implications

- As teams become more experience in Cleanroom, then begin to write their programs more directly
- This typically results in very small program segments with few control structures each
- Example:  3300 lines $\Rightarrow$ 600 control structures, 1000 correctness conditions

# 4. Statistical Usage Testing

- Certification of reliability
- Process control
- Cost-effective orientation
- Guidelines for test completion (desired reliability reached) or redesign (too many failures found)
- Stratification mechanism for dealing with critical situations
- But questions exist on how to feed back the results of testing to the development team

# Cost-Effective Testing

**Table 2.**
**Software failures for nine major IBM products, classified from rare to frequent.**

| | | Rare → | | | | | | | Frequent |
|---|---|---|---|---|---|---|---|---|---|
| Group | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| MTTF (years) | | 5,000 | 1,580 | 500 | 158 | 50 | 15.8 | 5 | 1.58 |
| Percent failures in class for product | 1 | 34.2 | 28.8 | 17.8 | 10.3 | 5.0 | 2.1 | 1.2 | 0.7 |
| | 2 | 34.3 | 28.0 | 18.2 | 9.7 | 4.5 | 3.2 | 1.5 | 0.7 |
| | 3 | 33.7 | 28.5 | 18.0 | 8.7 | 6.5 | 2.8 | 1.4 | 0.4 |
| | 4 | 34.2 | 28.5 | 18.7 | 11.9 | 4.4 | 2.0 | 0.3 | 0.1 |
| | 5 | 34.2 | 28.5 | 18.4 | 9.4 | 4.4 | 2.9 | 1.4 | 0.7 |
| | 6 | 32.0 | 28.2 | 20.1 | 11.5 | 5.0 | 2.1 | 0.8 | 0.3 |
| | 7 | 34.0 | 28.5 | 18.5 | 9.9 | 4.5 | 2.7 | 1.4 | 0.6 |
| | 8 | 31.9 | 27.1 | 18.4 | 11.1 | 6.5 | 2.7 | 1.4 | 1.1 |
| | 9 | 31.2 | 27.6 | 20.4 | 12.8 | 5.6 | 1.9 | 0.5 | 0.0 |
| Average percentage failures | | 33.4 | 28.2 | 18.7 | 10.6 | 5.2 | 2.5 | 1.0 | 0.4 |
| Probability of a failure for this frequency | | 0.008 | 0.021 | .044 | 0.079 | 0.123 | 0.187 | 0.237 | 0.300 |

# Testing Process

- Usage distribution models
  - From competitors, earlier versions, analysis
- Markov usage chain
  - State transition probability matrix
- Statistics
  - $\Pi$ (proportion of time spent in each state)
  - n (number of states visited before a given state is reached)
  - s (number of tests needed to reach a state).
- Random test generation
  - Design required
- Test execution and test chain generation, including failure states
- Statistics
  - R (reliability)
  - MTBF (mean time between failures)
  - D (divergence of test chain from usage chain)

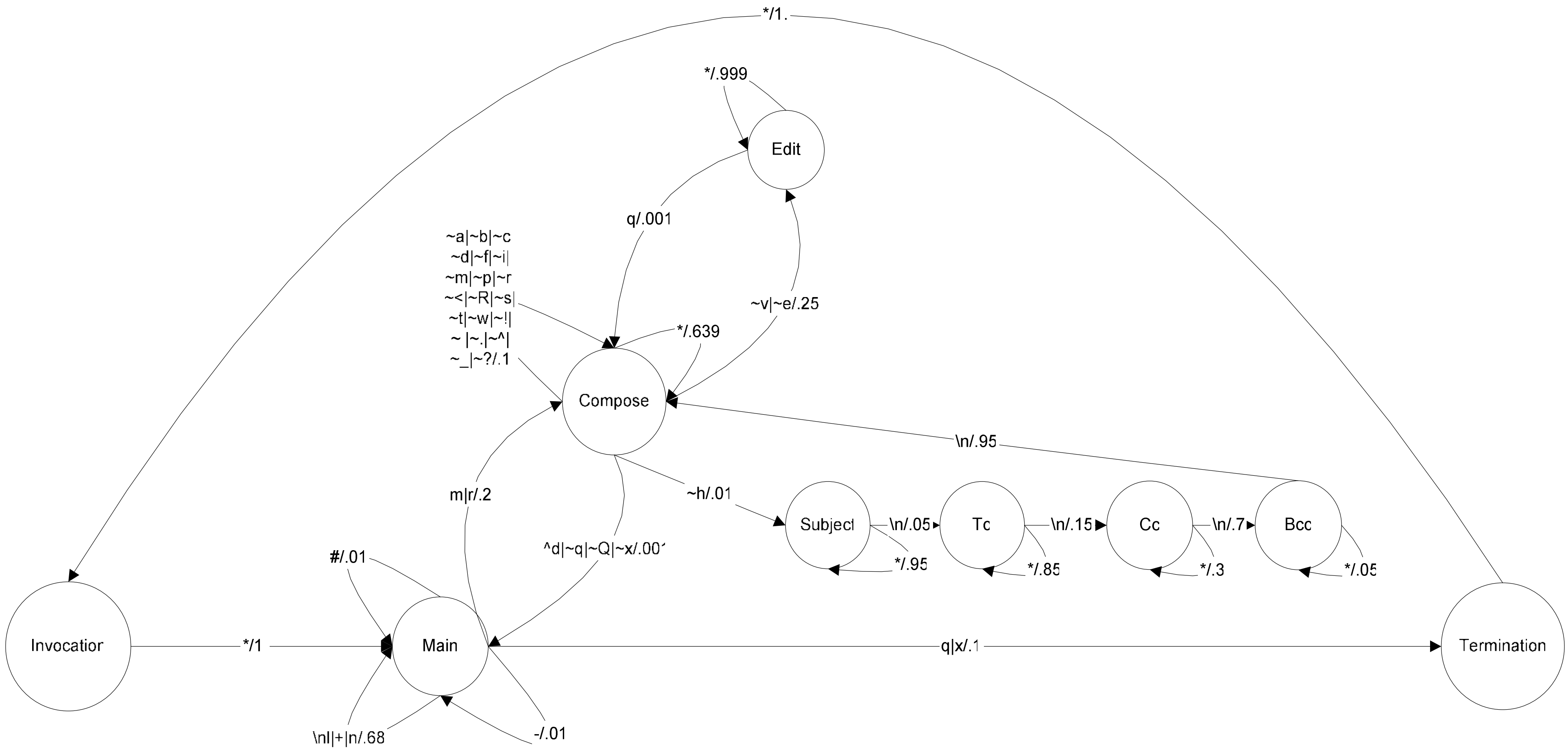
# Testing Process Overview

- Usage distribution models; other software, earlier versions, analysis
- Construct Markov usage chain / probability matrix
- Computations of $\Pi$ (proportion of time spent in each state), n (number of states visited before a given state is reached), and s (number of tests needed to reach a state).
- Random test generation (some design required here to deal with constraints)
- Test execution and test chain generation, including failure states
- Calculations of R (reliability), MTBF (mean time between failures), and D (divergence of test chain from usage chain)

# Testing Example

- COBOL / SF parser generator
- Four increments; 120 random tests
- Last 115 executions correct
- 12 failures in first five executions
- 3.9 faults / KLOC
- No new failures in four years of use

# Usage Model For Unix Mail



# Results Of Independent Empirical Evaluation

- 15 3-person teams; 10 of them used Cleanroom
- 6/10 delivered 91% of functionality
- Requirements better met and less failures
- More comments, less dense control flow
- Better adherence to schedule
- Developers expressed satisfaction with process

# Results

- Defects: 2 - 5 / KLOC versus 10-30 / KLOC for debugging
- Productivity: 3 - 5 × improvement in verification over debugging
- Reliability: statistical usage testing 20 × as effective as coverage testing

# Cleanroom Tools

- Test case generator
- Reliability analysis package
  - Spreadsheet
- Verification-based inspection syntax analyzer
  - Script for inspection
- Management assistant
  - Reports on process