The approach to be discussed :- 'Cleanroom Strategy'

→ formal modeling & verification method.
→ specialized specification approach.
→ unique verification method
→ rigorous

→ Difference b/w traditional reviews & testing and formal modeling & verifica^{n} ?

- Reviews & testing begin once s/w models and codes have been developed.
- Formal modeling & verification (FMV) incorporate specialized modeling methods that are integrated with prescribed verifica^{n} approaches

→ Characteristics / Features / Advantages of Cleanroom S/W Engg

Aim / Goal / Idea → Do it right the first time.

Advantage? - saved from unnecessary s/w rework.
- less effort expended.
- reduced cost

Both cleanroom software engg (CSE) help a s/w team to build a s/w right the first time by providing a mathematically based approach to program

modeling and verification that of the correctness of the model.

CSE emphasizes mathematical verification of correctness before program construction commences and certification of s/w reliability as part of the testing activity.

Bottom-line → creation of s/w with extremely low failure rates.

CSE requires a specially trained software engineer.

Representation :-

Box structure representation of the system or some part of the system.

↓ Purpose

Encapsulation of the system or some aspect of the system at a specific level of abstraction

Step 1 :- Box structure designing

Step 2 Correctness verification.

Step 3 Statistical usage testing

O/P → A specialized formal model of requirements is developed. The results of correctness proofs and statistical use tests are recorded.

Traditional Strategy

→ Build & then test at the end

→ costly defect removal process

→ Analysis → Design → Code → Test → Debug cycle.

Cleanroom Strategy

↓

Incorporate correctness as the s/w built.

→ generate code increments

- model formally & verify the correctness of each increment before proceeding further

→ a specialized version of the incremental s/w model.

↓

pipeline of s/w increments, each developed by an independent team. The ~~in~~certified increments are integrated into the whole

→ The cleanroom process Model :

Task 1 → Increment Planning
- project plan developed having incremental strategy.
To a each increment create :-
- projected size.
- cleanroom development schedule.
- Integration of certified increments timely.

2) Requirements Gathering (RG)
- more detailed description of customer-level requirements for each increment is developed.

3) Box Structure Specification (BSS) :-
- Box structures describe functional specifications
- Box structures - isolate & separate the creative definition of :-
  - behaviour
  - data
  - procedures.

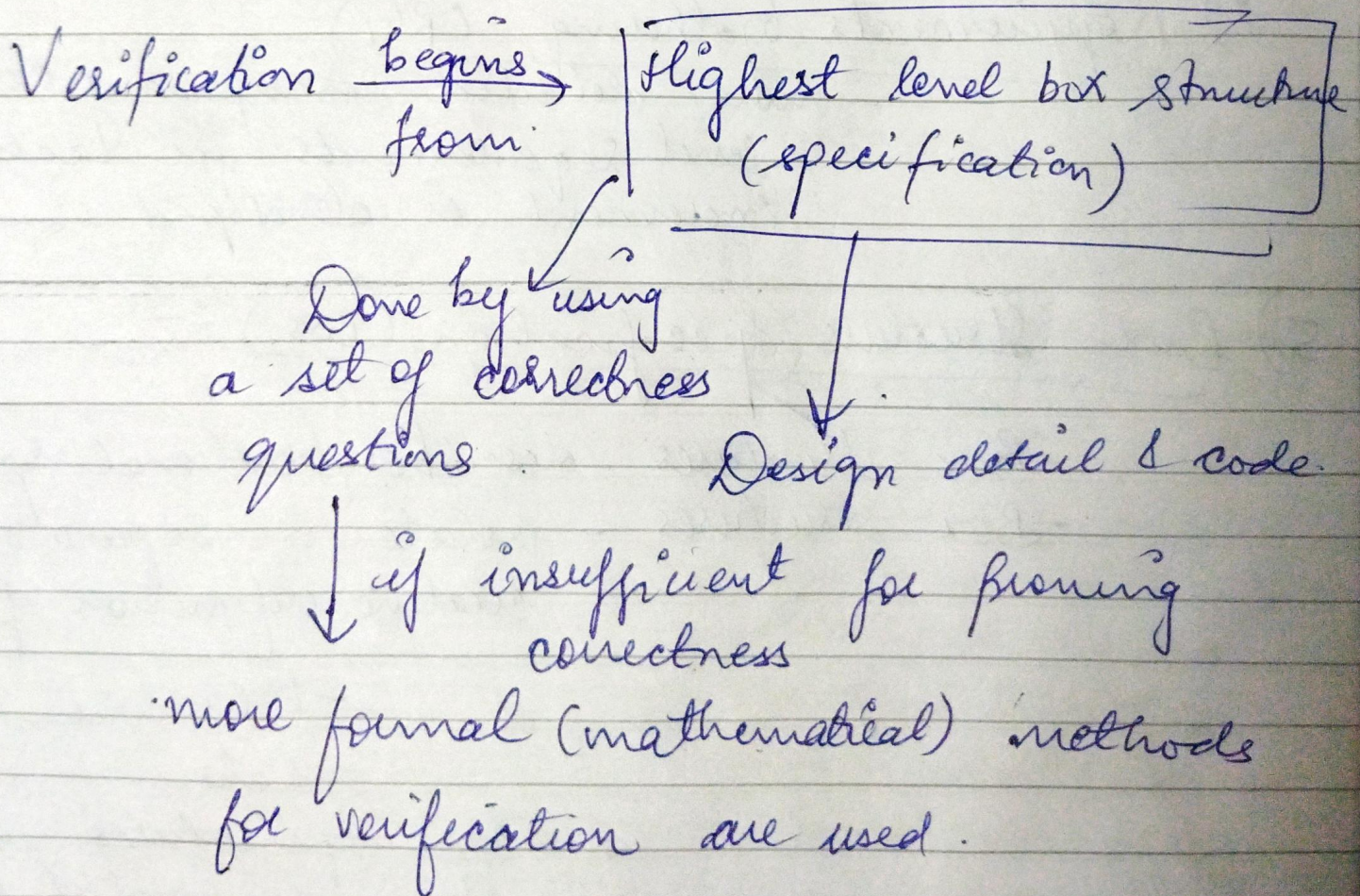
at each level of refinement

4) Formal Design -

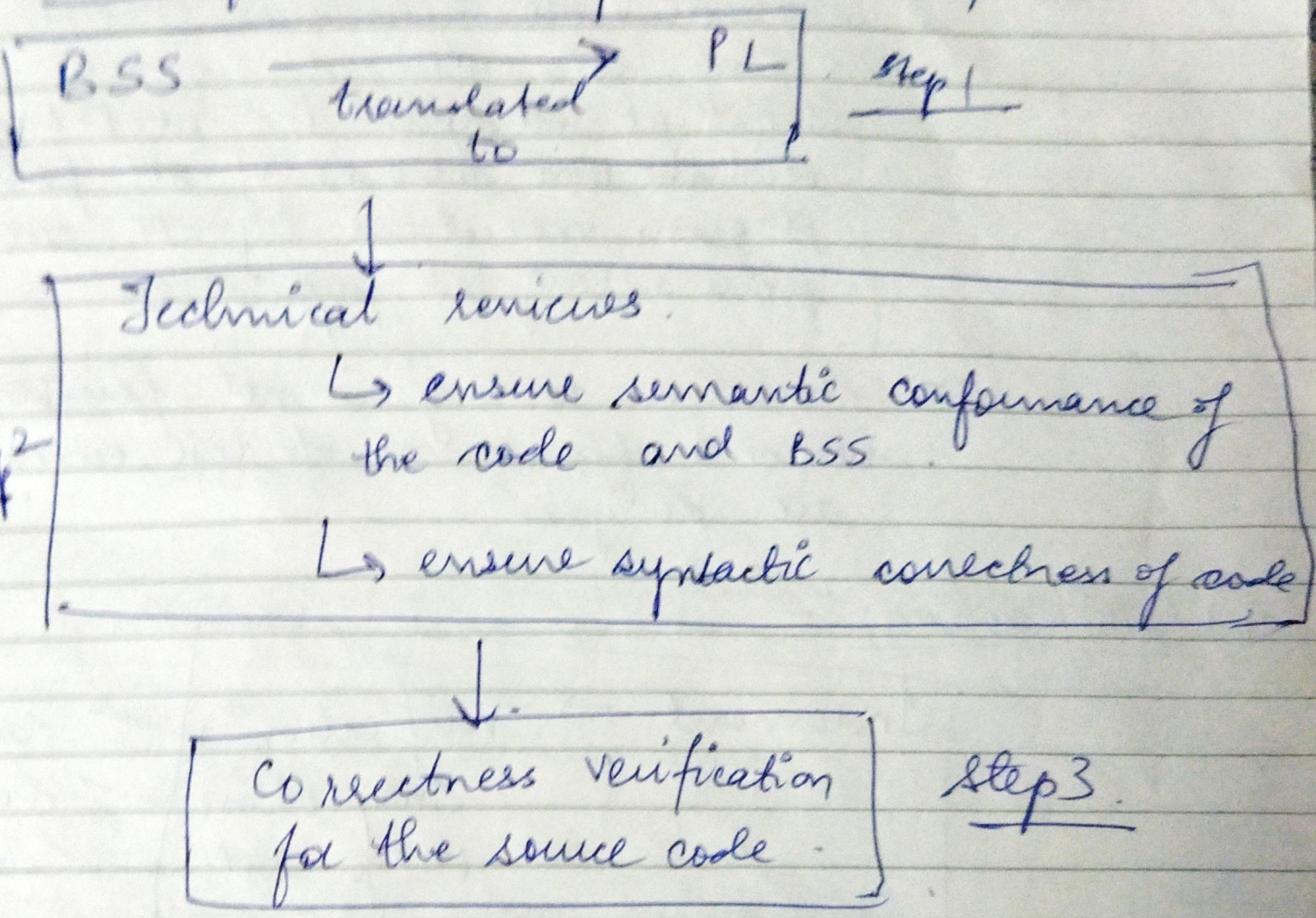
- Specifications (called ~~box~~ black boxes) are iteratively refined within an increment to become analogous to architectural designs (state boxes) and component level designs (clear boxes).

5) Correctness verification

- Series of rigorous correctness verification activities on the design and then the code.



6) Code generation, Inspection & Verification.



7) Statistical Test Planning

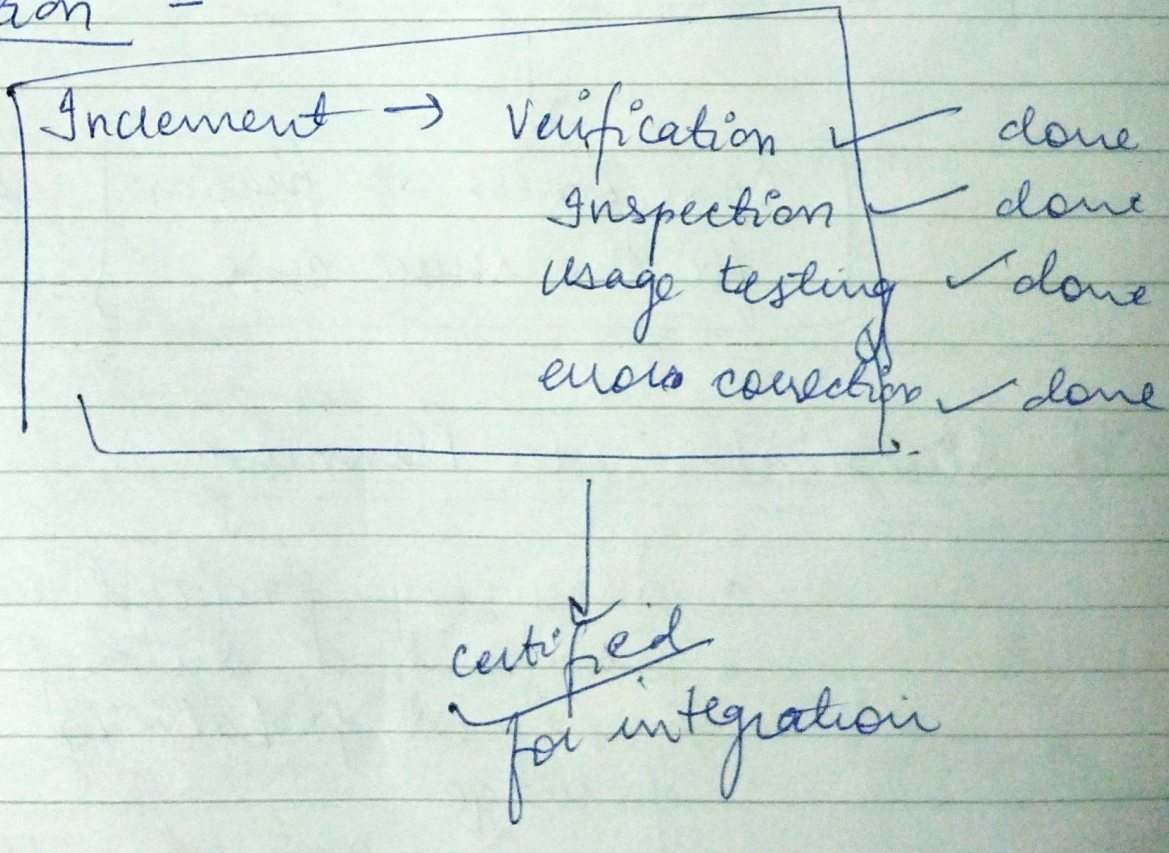
- Analyzing projected use of the S/w.
- Designing a suite of test cases based on probability distribution of usage
- done in parallel with specification, verification & code generation.

8) Statistical use testing -

- Statistical use techniques execute a series of tests derived from a statistical sample (the probability distribution of one) of all possible program executions by all users from a targeted population

- exhaustive testing is not possible, hence a finite no. of test cases are designed.

9) Certification -



Increment Planning

↓

Requirements Gathering

↓

BSS

↓

Formal Design

↓

Correctness verification

↓

Code Generation, inspection verification

↓

Statistical Test Planning

↓

Statistical Use Testing

↓

certification

Modeling activities

Formal verification activities

↓

The Cleanroom Process

(This entire sequence of steps is done for each increment)

→ FUNCTIONAL SPECIFICATION.

→ done by BSS
→ modeling activity
→ Box → encapsulation of the system or some aspect of the system at some level of calistraction.

Initial Box ⟶ Essential Representation

↓ Elaboration / stepwise refinement

Final Box → Implementation specific detail.

Black Box. → system behaviour or part of a system.

↓

State Box. → state data and services (operations).

↓

clear Box. → definitions of transition function.

Hierarchy of Box refinement

Adv → Enables the analyst to partition the system hierarchically.

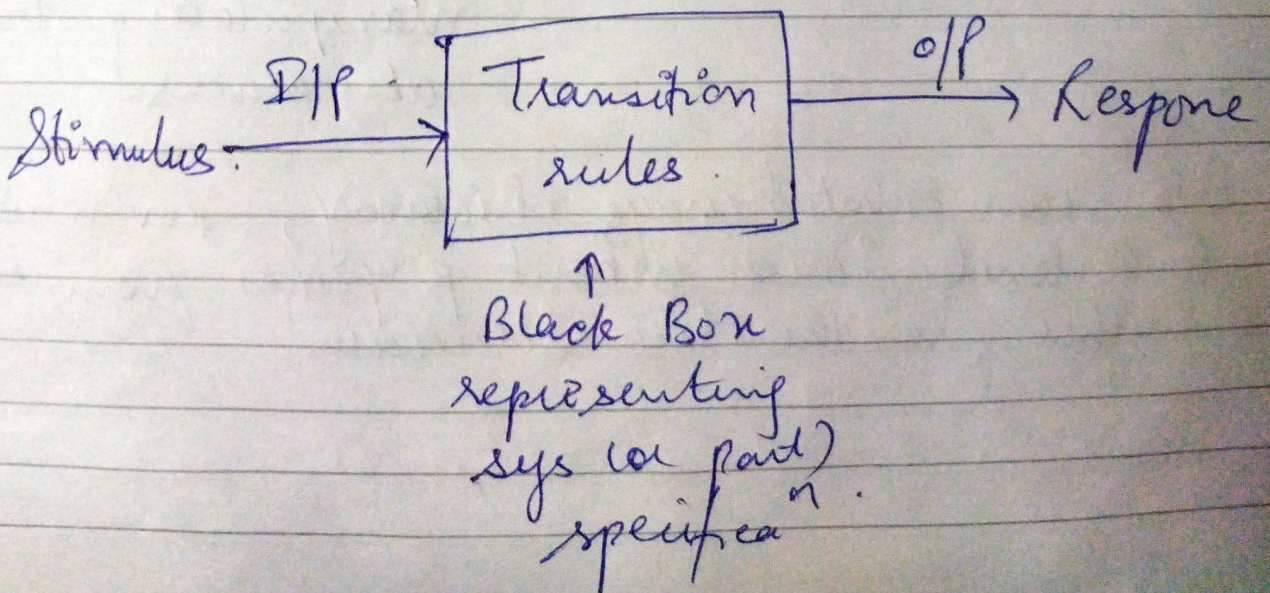
Property of each box → **Referential transparency.**

↓

info content of each box spec. is sufficient to define its refinement, without depending on the implementation of any other box.

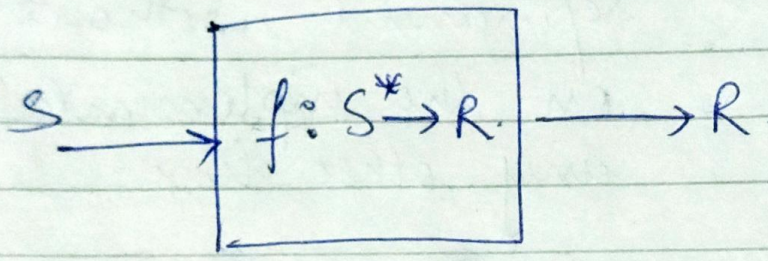
1) **Black Box (BB)**

- specifies system (or part) behaviour.
- The system (or part) responds to specific stimuli (or events) by applying a set of transition rules that map the stimulus into a response.



→ Black Box Specification (BBS)

It describes
- abstraction
- stimuli
- response



Black Box Specification.

The function f. can be a mathematical function for simple s/w components, but in general, f is described using natural language or a formal specification language (FSL)

BB encapsulates
① Data abstractions
② operations manipulating the abstractions

BB can exhibit usage hierarchy, in which low level boxes inherit properties from boxes higher in the tree structure.

2) State Box (SB) → simple generalization of a state machine

- encapsulates
  - ↳ state data
  - ↳ services (operations)

  s.t the specification is analogous to objects.
- represents stimulus history of the black box.

↓

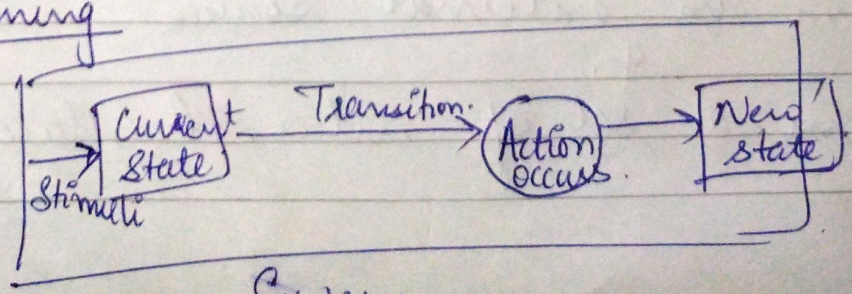
includes data encapsulated in the state box that must be retained b/w the transitions implied.

→ State Box Specification (SBS)

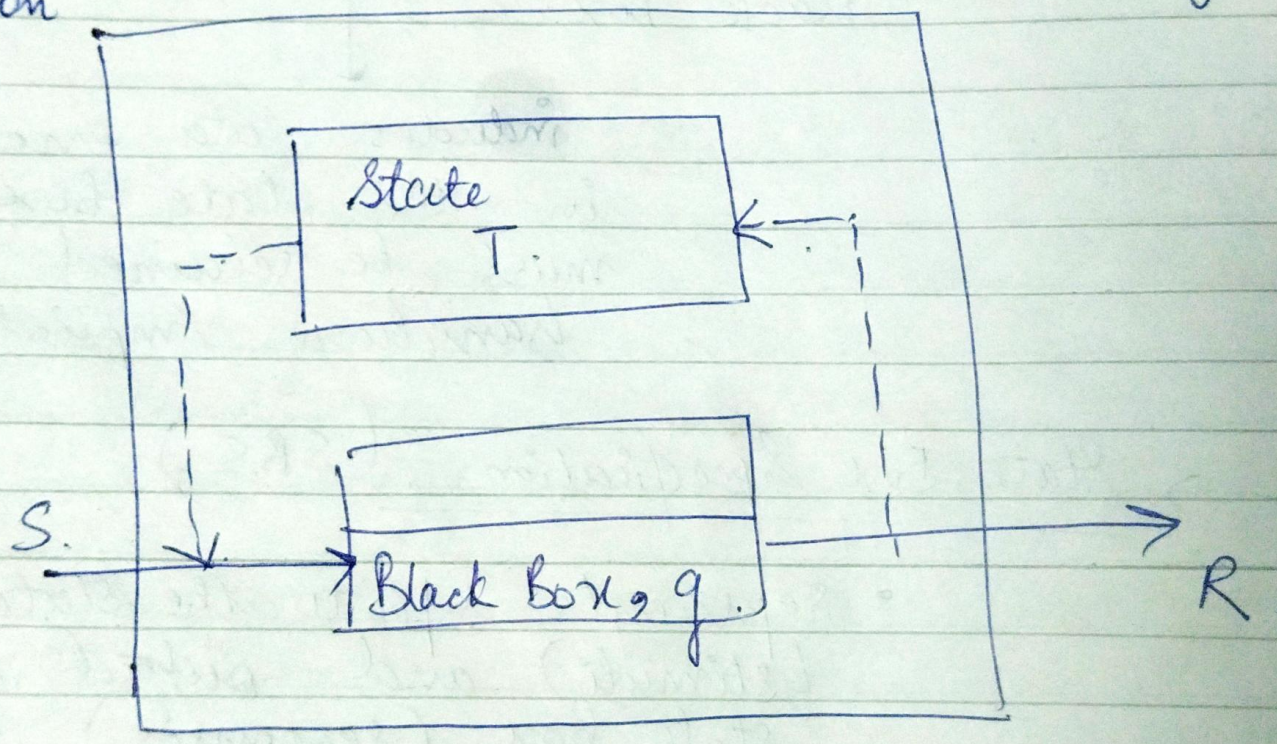
- represents I/P to the state box (stimuli) and outputs of the state box (responses).

State → An observable mode of system behaviour.

Functioning



- As processing occurs, a system responds to events (stimuli) by making a transition from the current state to some new state.
- As the transition is made, some action may occur.
- The state box uses data abstraction to determine the transition to the next state and the action (response) that will occur as a consequence of the transition.



A State Box Specification

- A SB incorporates a BB. (q)
- S → stimulus to BB.
  - → from an external source
  - + .
  - from a set of internal states T.

Given a specific state t, q is a BB sub-function tied to that state t.

$$q: S^* \times T^* \longrightarrow R \times T$$

For each state t, we can define a subfunction

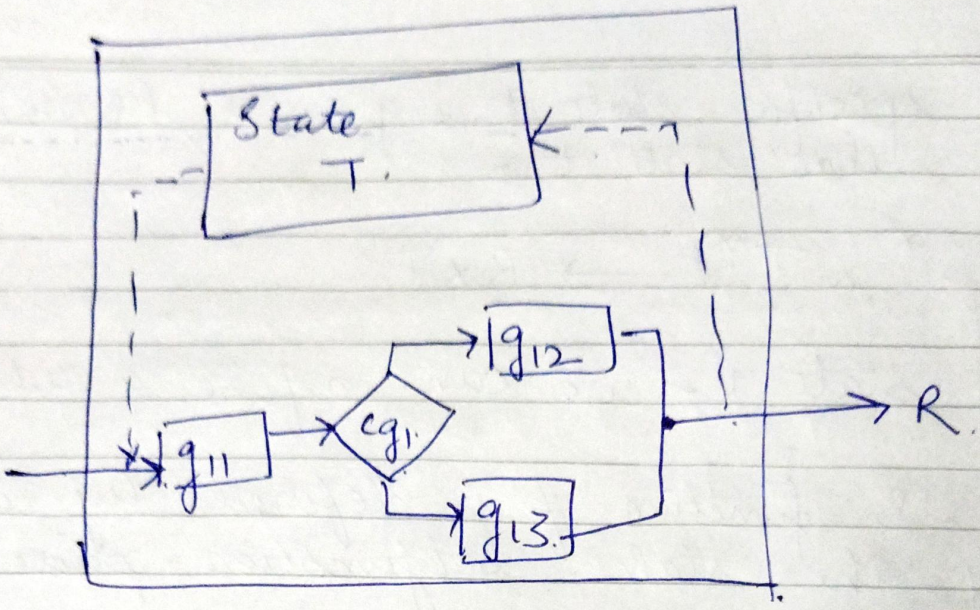
The entire BB function f is defined by collectively considering the state – subfunction pairs (t, q)

3) Clear Box (CB)

- defines the transition functions implied by the SB.
- contains procedural design for the SB

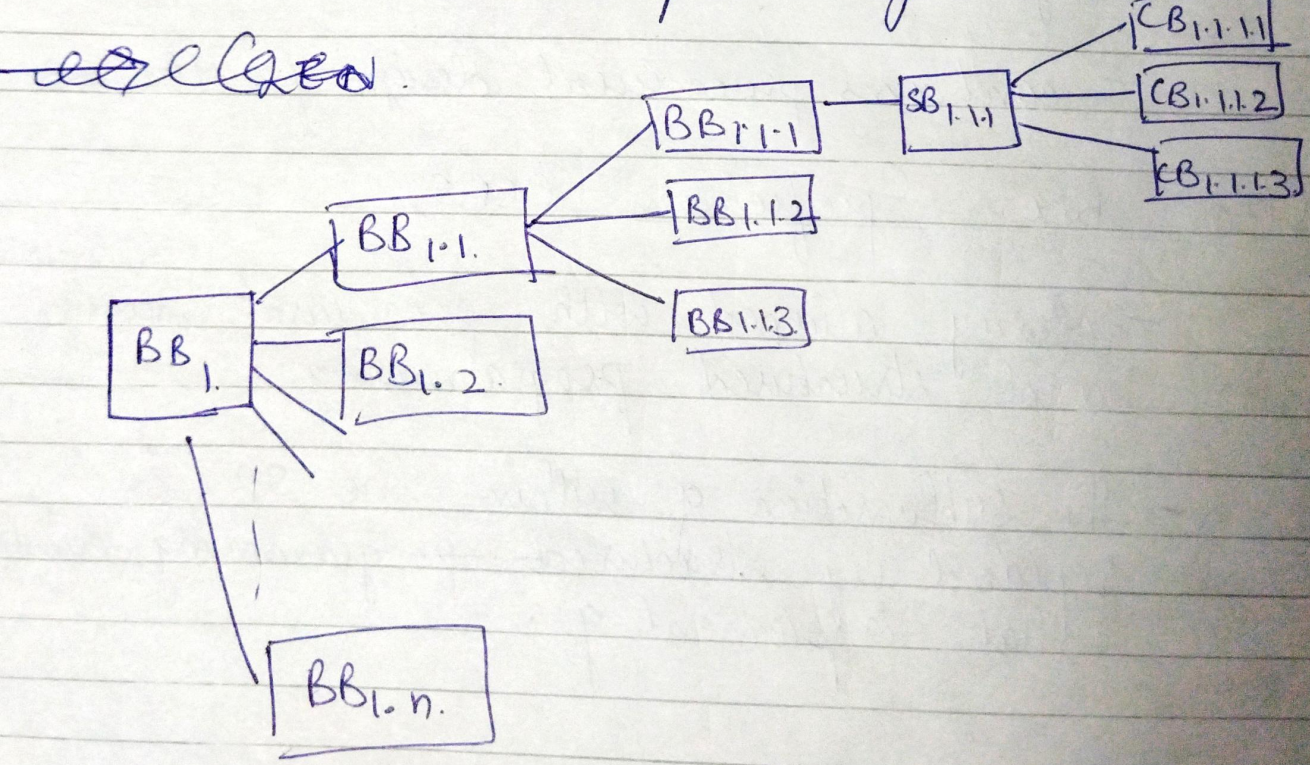
→ Clear Box Specification (CBS)

- closely aligned with procedural design and structured programming.
- The subfunction q within the SB is replaced by structured programming constr that implement q.



A clear box specification -

A CBS can be further refined into lower-level clear boxes with stepwise refinement.



Refinement Approach using BSS

- BB_1 defines responses for a complete set of stimuli
- BB_1 can be refined into $BB_{1.1}$ to $BB_{1.n}$ each of which addresses a class of behavior.
- Refinement continues until a cohesive class of behaviors is identified like $BB_{1.1.1}$.
- SB $SB_{1.1.1}$ is then defined for $BB_{1.1.1}$
- $SB_{1.1.1}$ contains all data and services required to implement $BB_{1.1.1}$
- $SB_{1.1.1}$ is refined into CBs $CB_{1.1.1-3}$ and procedural design details are specified.

- With each refinement step, verification of correctness also occurs.
- SBSs are verified to ensure that each conforms to the behavior defined by the parent BBS. Similarly, CBSs are verified against the parent SB.

→ CLEANROOM DESIGN

→ Design refinement

→ Design verification

- uses rigorous structured programming.

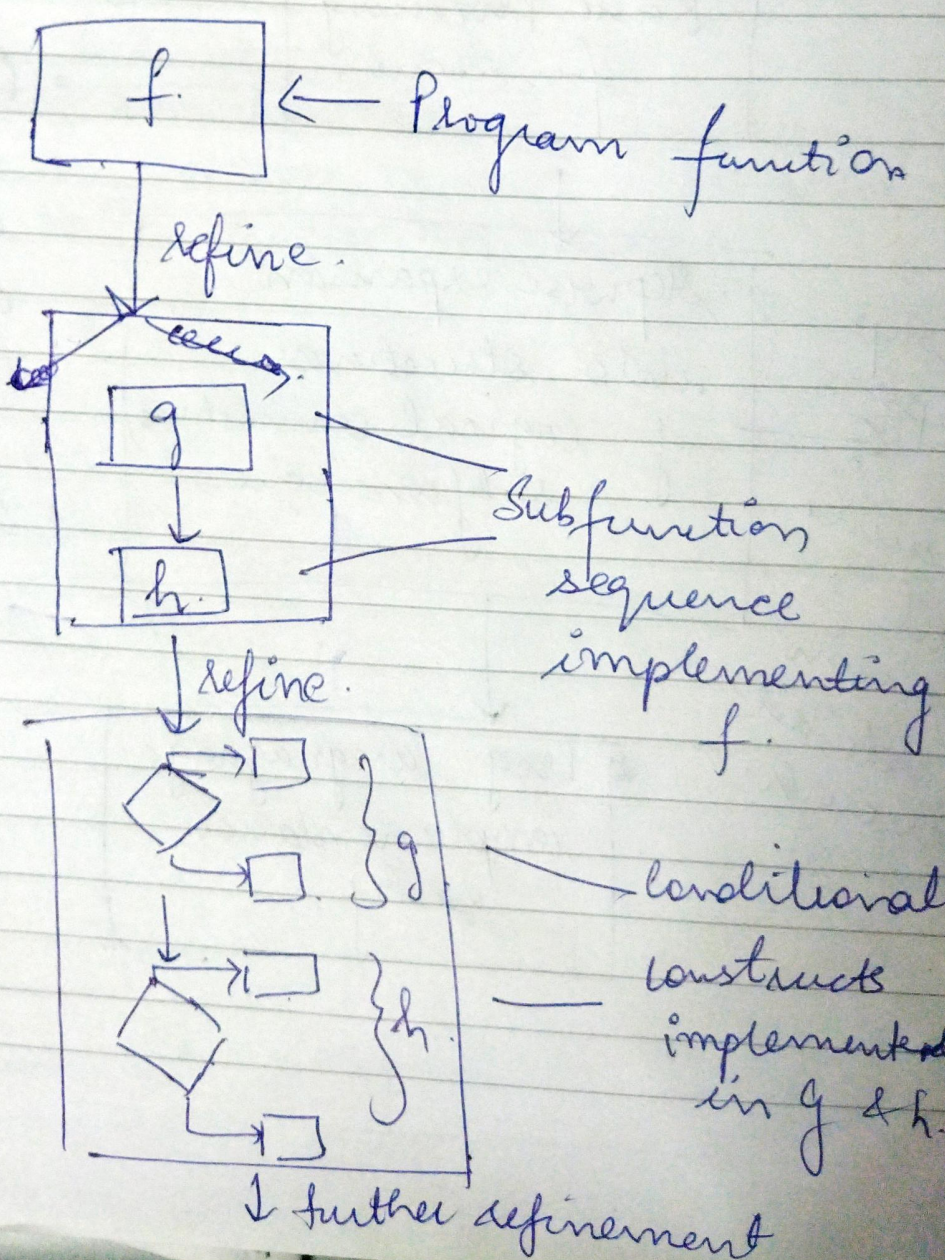
2 things need to be refined

- functions
  - Basic Processing functions
  - Stepwise expansion into structures of logical connectives, subfunctions
    - This step keeps on repeating until expansion is detailed enough.
  - Prog. language implementation ready.
- data
  - Data designing
    - Program data are encapsulated as a set of abstractions serviced by functions.
    - Uses concepts of
      - ↳ encapsulation
      - ↳ data hiding
      - ↳ data typing

1) Design Refinement

- done using CBS
- CBS represents the design of a subfunction required to accomplish a a SB transition.
- CB uses structured programming constructs and stepwise refinement

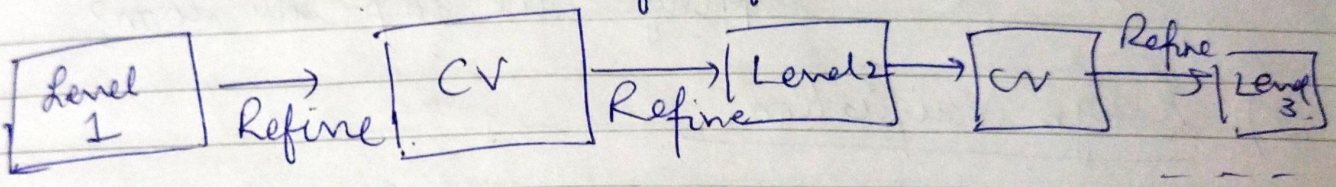
Eg.



Continue refinement until implementation required level of procedural detail is attained

Incorporating correctness verification (CV)

- formal correctness verification done at each level of refinement.



How? Generic correctness conditions are attached to the various structured programming constructs.

Examples. 1) function f expanded as a sequence of subfunctions g & h.

only 1 cond checked for sequence

Correctness Condition (CC) → For all input to f is :-
does g followed by h do f

2 conditions checked for if-then-else

2) function p refined as a conditional:
if <c> then q, else r.

<c> → For all input to p :-
- whenever condition <c> is true, does q do p?
- whenever <c> is false, does r do p?

Advantage: structured prog. constructs constrains the no of correctness tests.

3) function m refined as a loop

For all input to m :-
- is termination guaranteed?
- whenever <c> is true, is m accomplished?
- whenever <c> is false, does skipping the loop still do m?

<c> → 3 cond^n checked for loops.

2) Design Verification

- It does correctness verification for a procedural design.
- The correctness tests designed in the previous phase are verified.
- Verification done at each level of refinement.
- The entire cleanroom team is involved in the verification process, hence it is less likely that an error will be made in conducting the verification itself.

Steps for Design Verification: (DV)

Procedural Design → Add correctness conditions to the design → Prove the conditions are true for all cases.

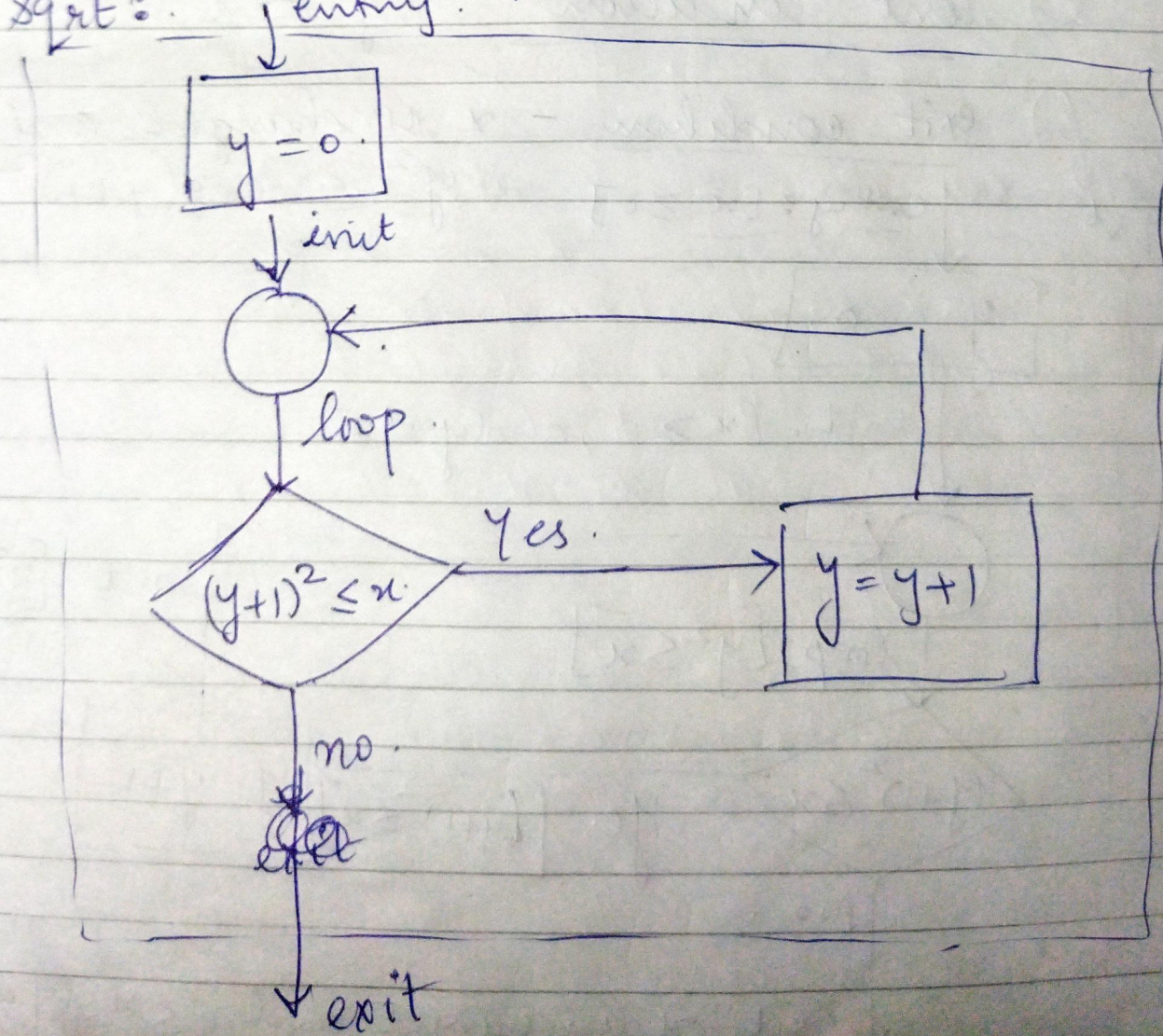
These separate proofs of all conditions in all cases are called subproofs.

DV part is the construction of these subproofs.

Example

Design and verify a small program that finds the integer part y of a square root of a given integer x.

Sol^n 1) Procedural Design :-



2) Adding correctness conditions to the design:-

We add 6 conditions :-

a) entry condition - $x \geq 0$.

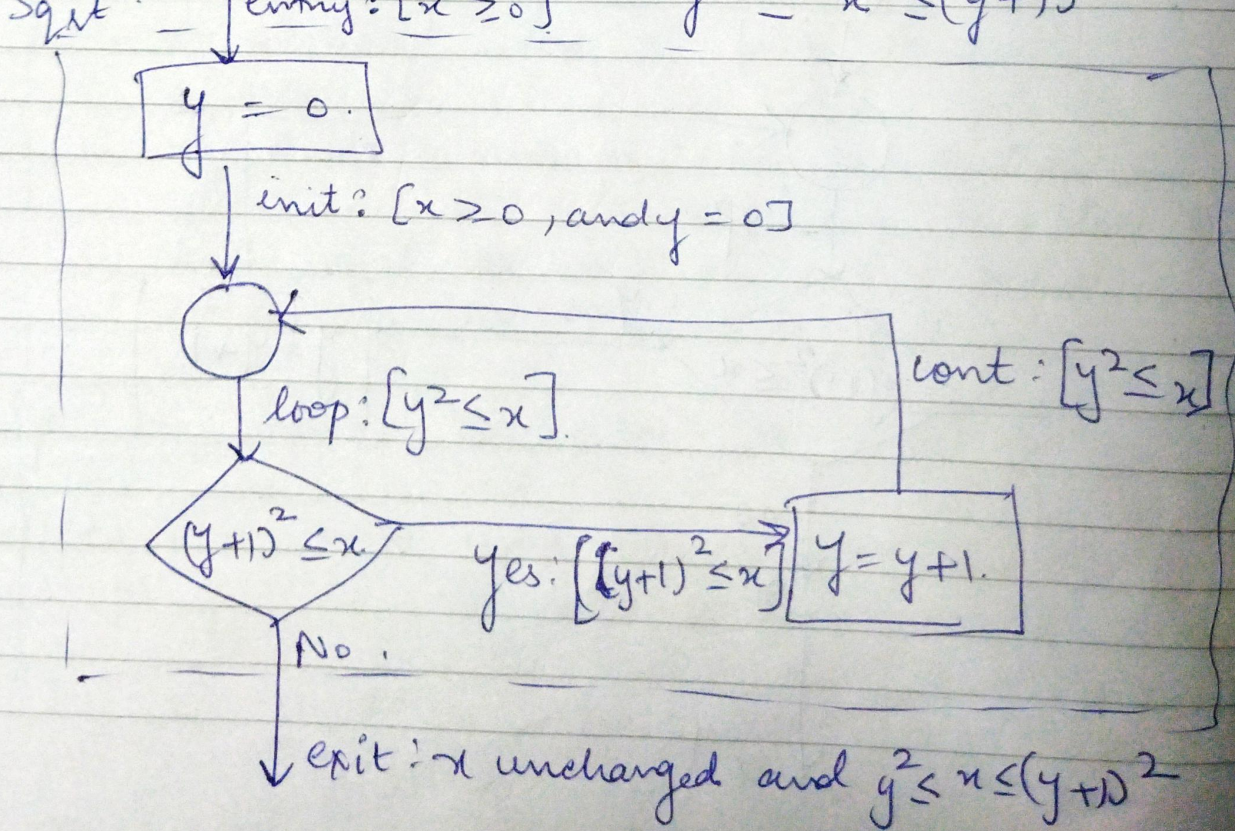
b) init condition - $x \geq 0$ and $y = 0$

c) loop condition - $y^2 \leq x$.

d) yes condition - $(y+1)^2 \leq x$.

e) cont. condition - $y^2 \leq x$

f) exit condition - x unchanged and $y^2 \leq x \leq (y+1)^2$



3) Design verification
- This will contain subproofs for each of the above added conditions.

a) Entry condition → [$x \geq 0$] → In the given context, a negative value for a square root has no meaning.

b) Init condition → [$x \geq 0$ and $y = 0$]

The first part of init condition, i.e $x \geq 0$ is satisfied based on the entry condition.

Refering to the flowchart, the statement immediately preceding the init condition sets $y = 0$. ∴ The second part of the init condition is also satisfied.

Hence, init is true. ✓

c) Loop condition :- [$y^2 \leq x$]

The loop cond^n can be encountered in 2 ways.

(i) directly from init - in this case the loop cond^n is satisfied directly! [∵ $x \geq 0$ and $y = 0$ is the init cond^n]

(ii) via flow control that passes through condition cont :- since the cont cond^n is identical to the loop cond^n, loop cond^n is true.

d) Cont condition → $[y^2 \leq x]$

This is encountered only when the value of y is incremented by 1. Also, the cont condition is encountered only when the yes condition $[(y+1)^2 \leq x]$ is also true.

$[(y+1)^2 \leq x] \Rightarrow y^2 \leq x$.

∴ Cont condition is satisfied.

e) Yes condition → $[(y+1)^2 \leq x]$

It is tested already in the conditional logic shown. The control flow will reach the yes path only if the condition is true.

f) Exit condition → x unchanged and $y^2 \leq x \leq (y+1)^2$

- first demand – x unchanged.

An examination of the design indicates that x appears nowhere to the left of an assignment operator. There are no function calls that use x. Hence, it is unchanged

- Second part – $y^2 \leq x \leq (y+1)^2$.

Since, the conditional construct test $(y+1)^2 \leq x$ must fail to reach the exit condition ∴ It implies that $(y+1)^2 \geq x$.

Also, the loop condition $[y^2 \leq x]$ must still be true (if it were not true, then exit condition must have been reached in the previous iteration of loop itself for value y.) ∴ $\Rightarrow y^2 \leq x$

Both of these can be combined to satisfy the second exit condition $y^2 \leq x \leq (y+1)^2$.

7) Loop termination :- An examination of the loop cond° indicates that because y is incremented and $x \geq 0$, there will eventually come a point when the loop condition $[y^2 \leq x]$ will be false for a value of y. Hence, loop will terminate.

Hence, the 7 subproofs above constitute a proof of correctness of the design of the sqrt algo. We can now be certain that the design will, in fact, compute the integer part of a square root.

→ 1. Statistical use Testing

→ CLEANROOM TESTING → Certification.

Traditional Testing Approach.

↓

Derive a set of test cases to uncover design & coding errors.

Cleanroom Testing Approach.

↓

Validates software requirements by demonstrating that a statistical sample of use cases have been executed successfully.

Basically doesn't let errors reach the coding phase (or tries to minimise or eliminate errors in as early in the s/w development cycle as possible)

→1) Statistical Use Testing. (SUT)

What is SUT? SUT amounts to testing software the way users intend to use it.

Since, in a complex system, the possible spectrum of inputs and events (i.e. use cases) can be extremely broad, SUT uses a subset of use cases that will adequately verify program behaviour.

How is it done?

Step1 → Certification teams determine a usage probability distribution for the s/w.

Step2 → The specification (BB) for each increment is analyzed.

Step 3 → A set of stimuli is defined that cause change in behavior.

Step 4 → Interview potential users.
↓
Create usage scenarios
↓
Get a general understanding of the application domain
↓
Define a probability of use for each element of the set of stimuli.

Step 5 → Generate test cases for each stimuli acc. to the usage prob. distribution.

2) Certification

- The CSE approach allows s/w components and entire increments to be certified.
- Certification implies that the reliability (measured by mean-time-to-failure MTTF) can be specified for each component.

- MTTF → The testing team executes the use cases and verifies s/w behavior against system spec.
  - Timing for tests is recorded so that internal times can be determined.
  - Using internal times, the certification team can compute MTTF.
  - If a long sequence of tests is conducted without failure, MTTF is low and s/w reliability is assumed to be high.

Value of Certification :-

- Reusable s/w components can be stored along with their usage scenarios, program stimuli & prob. distributions.
- Each component will have a certified reliability.

→ 5 Step Certification Process :-

a) Usage scenarios must be created.
b) Usage profile is specified.
c) Generate test cases from the profile.
d) Execute tests. Record & analyze failure data.
e) Compute reliability & certify.

Certification requires the creation of 3 models.

Sampling model.
↓
- S/w testing executes n random test cases.
- Certified if no failures occur or a specific no of failures occur.
- The value of n is derived mathematically to ensure that the required reliability is achieved.

Component model.
↓
- Given a system that is to be certified, contains n components.
- The component model enables the analyst to determine the probability that component i will fail prior to completion.

Certification model.
↓
It is used to project and certify the overall reliability of the system.

A certified MTTF is computed using each model for the software.