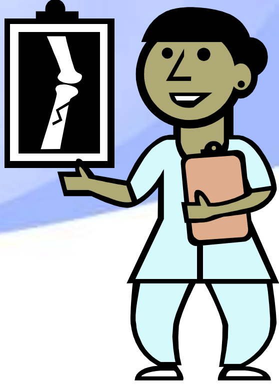
Software testing

Main issues:

- **There are a great many testing techniques**
- **Often, only the final code is tested**

Nasty question



- Suppose you are being asked to lead the team to test the software that controls a new X-ray machine. Would you take that job?
- Would you take it if you could name your own price?
- What if the contract says you'll be charged with murder in case a patient dies because of a mal-functioning of the software?

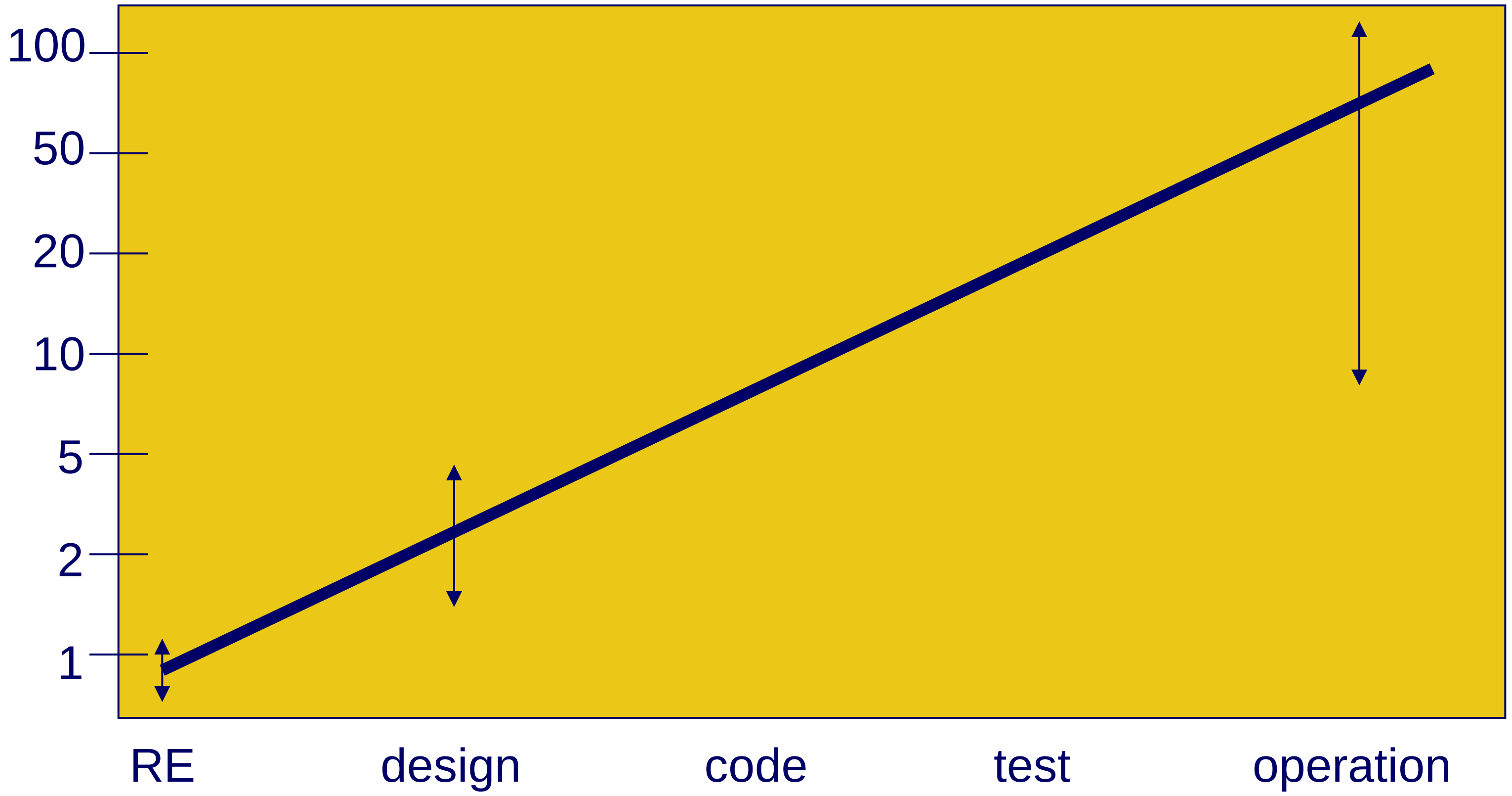
Overview

- **Preliminaries**
- All sorts of test techniques
- Comparison of test techniques
- Software reliability

State-of-the-Art

- 30-85 errors are made per 1000 lines of source code
- extensively tested software contains 0.5-3 errors per 1000 lines of source code
- testing is postponed, as a consequence: the later an error is discovered, the more it costs to fix it.
- error distribution: 60% design, 40% implementation. 66% of the design errors are not discovered until the software has become operational.

Relative cost of error correction



Lessons

- Many errors are made in the early phases
- These errors are discovered late
- Repairing those errors is costly
- ⇒ It pays off to start testing real early

How then to proceed?

- Exhaustive testing most often is not feasible
- Random statistical testing does not work either if you want to find errors
- Therefore, we look for systematic ways to proceed during testing

Classification of testing techniques

- **Classification based on the criterion to measure the adequacy of a set of test cases:**
  - coverage-based testing
  - fault-based testing
  - error-based testing
- **Classification based on the source of information to derive test cases:**
  - black-box testing (functional, specification-based)
  - white-box testing (structural, program-based)

Some preliminary questions

- What exactly is an error?
- How does the testing process look like?
- When is test technique A superior to test technique B?
- What do we want to achieve during testing?
- When to stop testing?

Error, fault, failure

- an *error* is a human activity resulting in software containing a fault
- a *fault* is the manifestation of an error
- a *fault* may result in a failure

When exactly is a failure a failure?

- Failure is a relative notion: e.g. a failure w.r.t. the specification document

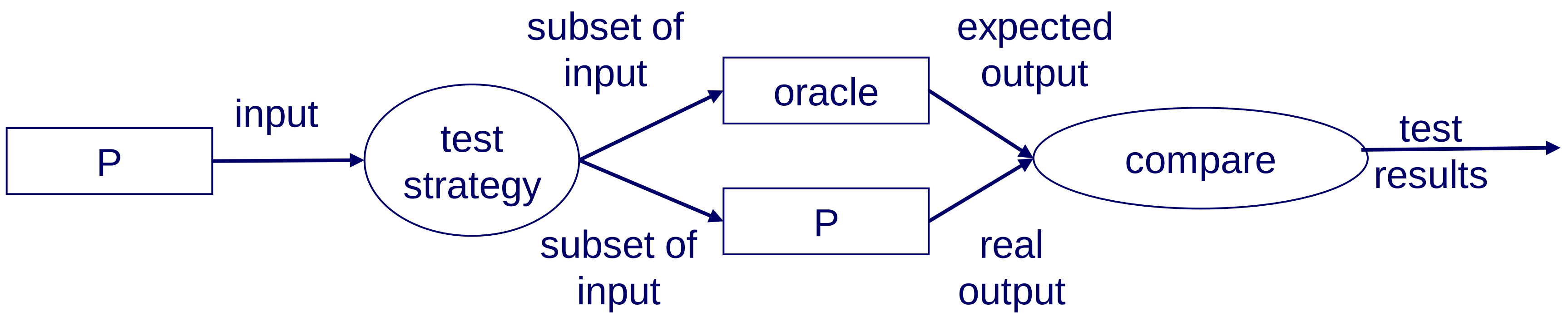
- *Verification*: evaluate a product to see whether it satisfies the conditions specified at the start:

  Have we built the system right?

- *Validation*: evaluate a product to see whether it does what we think it should do:

  Have we built the right system?

Testing process



Test adequacy criteria

- Specifies requirements for testing
- Can be used as *stopping rule*: stop testing if 100% of the statements have been tested
- Can be used as *measurement*: a test set that covers 80% of the test cases is better than one which covers 70%
- Can be used as *test case generator*: look for a test which exercises some statements not covered by the tests so far
- A given test adequacy criterion and the associated test technique are opposite sides of the same coin

What is our goal during testing?

- Objective 1: find as many faults as possible
- Objective 2: make you feel confident that the software works OK

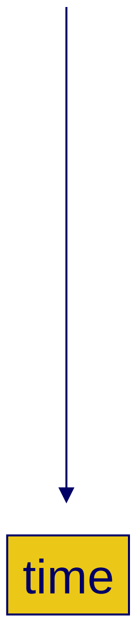
Example constructive approach

- **Task: test module that sorts an array A[1..n]. A contains integers; n < 1000**
- **Solution: take n = 0, 1, 37, 999, 1000. For n = 37, 999, take A as follows:**
  - A contains random integers
  - A contains increasing integers
  - A contains decreasing integers
- **These are *equivalence classes*: we assume that one element from such a class suffices**
- **This works if the partition is perfect**

Testing models

- *Demonstration*: make sure the software satisfies the specs
- *Destruction*: try to make the software fail

- *Evaluation*: detect faults in early phases
- *Prevention*: prevent faults in early phases



Testing and the life cycle

- **requirements engineering**
  - criteria: completeness, consistency, feasibility, and testability.
  - typical errors: missing, wrong, and extra information
  - determine testing strategy
  - generate functional test cases
  - test specification, through reviews and the like
- **design**
  - functional and structural tests can be devised on the basis of the decomposition
  - the design itself can be tested (against the requirements)
  - formal verification techniques
  - the architecture can be evaluated

Testing and the life cycle (cnt’d)

- **implementation**
  - check consistency implementation and previous documents
  - code-inspection and code-walkthrough
  - all kinds of functional and structural test techniques
  - extensive tool support
  - formal verification techniques
- **maintenance**
  - regression testing: either retest all, or a more selective retest

Test-Driven Development (TDD)

- ***First*** **write the tests,** ***then*** **do the design/implementation**
- **Part of agile approaches like XP**
- **Supported by tools, eg. JUnit**
- **Is more than a mere test technique; it subsumes part of the design work**

Steps of TDD

1. Add a test
2. Run all tests, and see that the system fails
3. Make a small change to make the test work
4. Run all tests again, and see they all run properly
5. Refactor the system to improve its design and remove redundancies

Test documentation (IEEE 928)

- Test plan
- Test design specification
- Test case specification
- Test procedure specification
- Test item transmittal report
- Test log
- Test incident report
- Test summary report

Overview

- Preliminaries
- All sorts of test techniques
  - **manual techniques**
  - coverage-based techniques
  - fault-based techniques
  - error-based techniques
- Comparison of test techniques
- Software reliability

Manual Test Techniques

- **static versus dynamic analysis**
- **compiler does a lot of static testing**
- **static test techniques**
  - reading, informal versus peer review
  - walkthrough and inspections
  - correctness proofs, e.g., pre-and post-conditions: {P} S {Q}
  - stepwise abstraction

(Fagan) inspection

- **Going through the code, statement by statement**
- **Team with ~4 members, with specific roles:**
  - moderator: organization, chairperson
  - code author: silent observer
  - (two) inspectors, readers: paraphrase the code
- **Uses checklist of well-known faults**
- **Result: list of problems encountered**

Example checklist

- Wrong use of data: variable not initialized, dangling pointer, array index out of bounds, …
- Faults in declarations: undeclared variable, variable declared twice, …
- Faults in computation: division by zero, mixed-type expressions, wrong operator priorities, …
- Faults in relational expressions: incorrect Boolean operator, wrong operator priorities, .
- Faults in control flow: infinite loops, loops that execute n-1 or n+1 times instead of n, ...

Stepwise Abstraction

```
procedure binsearch
    (A: array [1..n] of integer; x: integer): integer;
var low, high, mid: integer; found: boolean;
begin low:= 1; high:= n; found:= false;
    while (low ≤ high) and not found do
        mid:= (low + high) div 2;
        if x < A[mid] then high:= mid - 1 else
        if x > A[mid] then low:= mid + 1 else
            found:= true
        endif
    enddo;
    if found then return mid else return 0 endif
end binsearch;
```

Figure 13.8 A search routine

$result = 0 \leftrightarrow x \notin A[1 .. n]$

$1 \leq result \leq n \leftrightarrow x = A[result]$

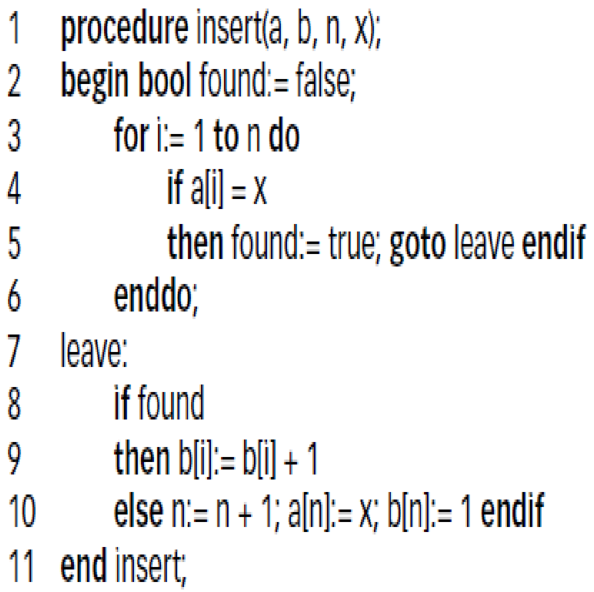
Overview

- Preliminaries
- All sorts of test techniques
  - manual techniques
  - **coverage-based techniques**
  - fault-based techniques
  - error-based techniques
- Comparison of test techniques
- Software reliability

Coverage-based testing

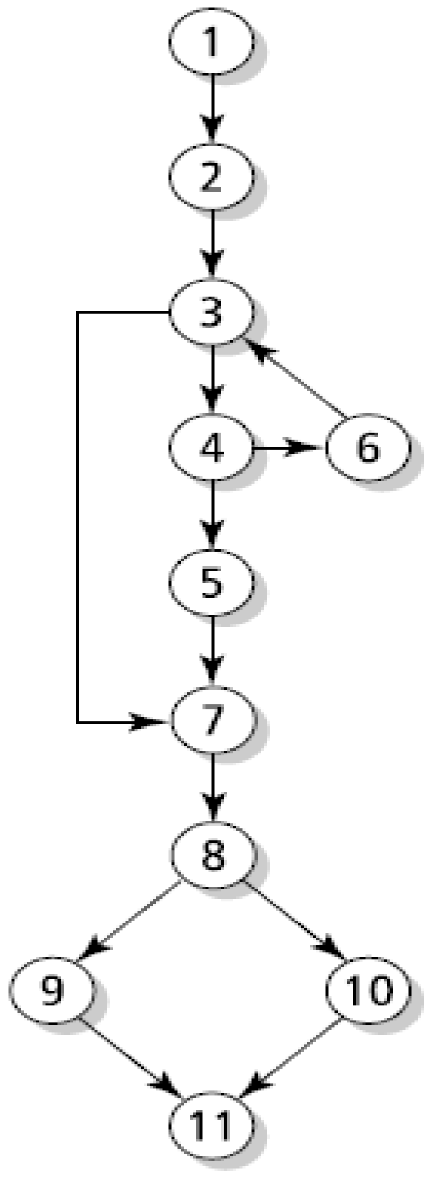
- **Goodness is determined by the coverage of the product by the test set so far: e.g., % of statements or requirements tested**
- **Often based on control-flow graph of the program**
- **Three techniques:**
  - control-flow coverage
  - data-flow coverage
  - coverage-based testing of requirements

Example of control-flow co

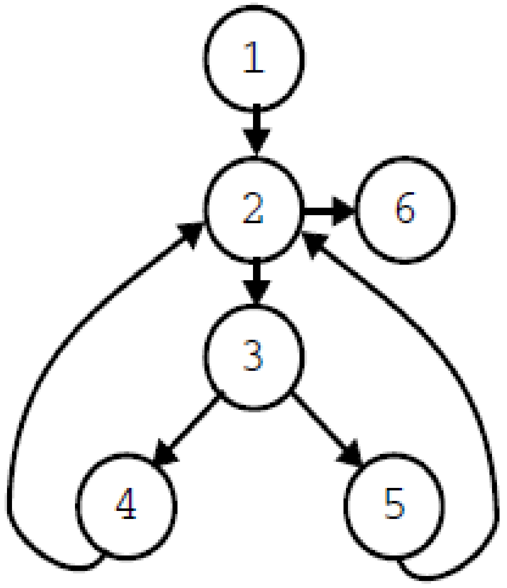


```
1   procedure insert(a, b, n, x);
2   begin bool found:= false;
3       for i:= 1 to n do
4           if a[i] = x
5           then found:= true; goto leave endif
6       enddo;
7   leave:
8       if found
9       then b[i]:= b[i] + 1
10      else n:= n + 1; a[n]:= x; b[n]:= 1 endif
11  end insert;
```

Figure 13.10 An insertion routine



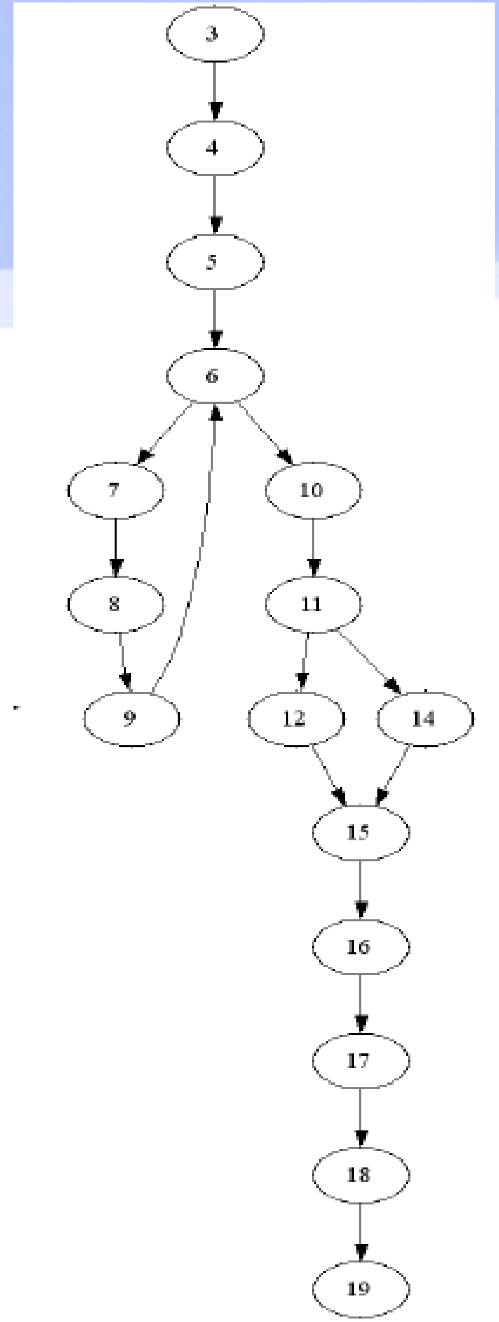
```
1: int gcd (int a, int b) {
2:    while (a != b) {
3:       if (a > b)
4:          a = a - b;
         else
5:          b = b - a;
         }
6:    return a;
     }
```



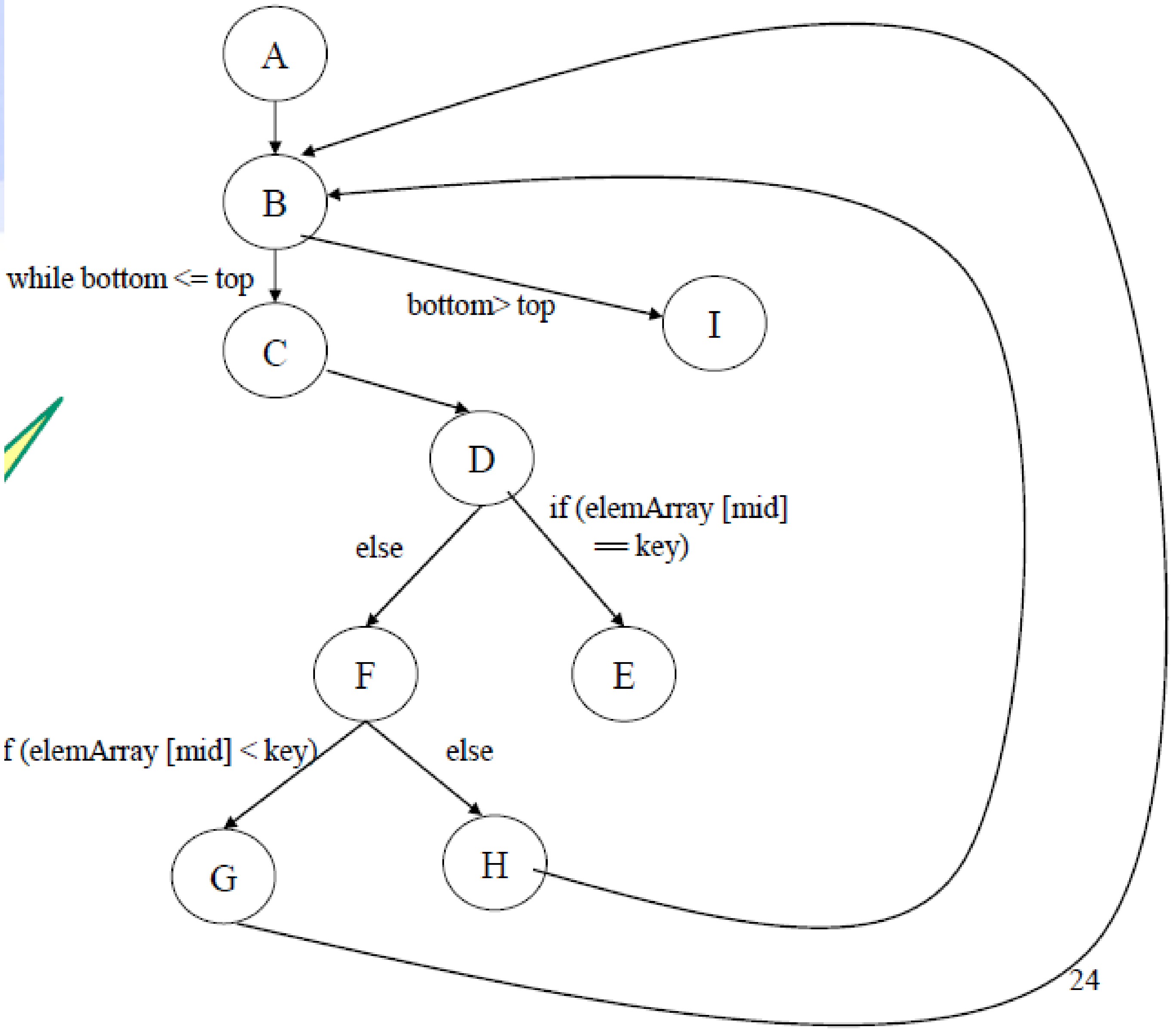
Control-flow coverage

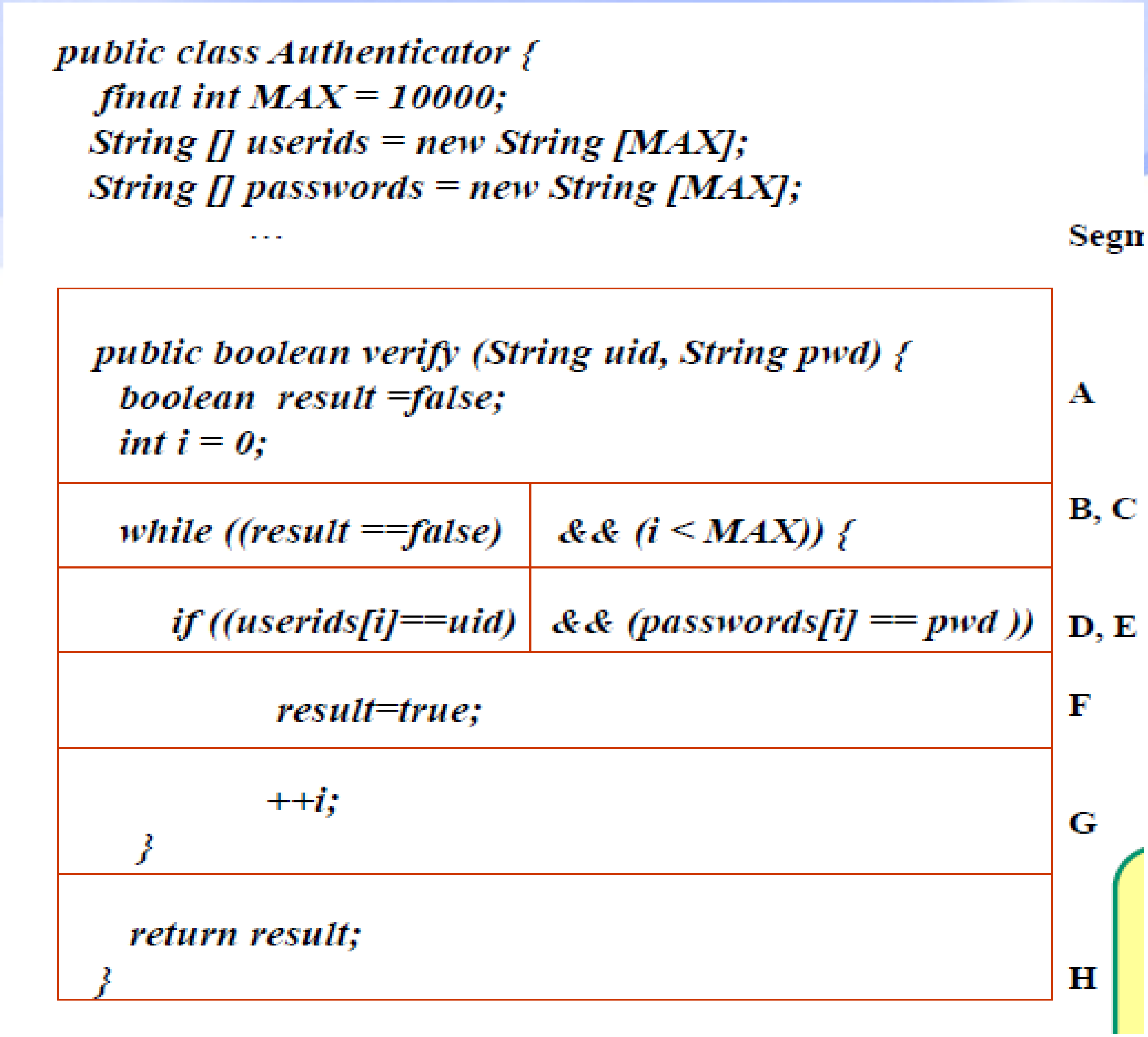
- ***All-Nodes coverage/statement coverage: All statements are executed once***
- **A stronger criterion: *All-Edges coverage/branch coverage***
- **At each branching node of the CG, all possible branches are chosen at least once**
- **Strongest: *All-Paths coverage* (≡ exhaustive testing)**
- **Special case: all linear independent paths, the *cyclomatic number criterion***

```
program Example()

var staffDiscount, totalPrice, finalPrice, discount, price

staffDiscount = 0.1
totalPrice = 0

input(price)
while(price != -1) do
 totalPrice = totalPrice + price
 input(price)
od

print(“Total price: ” + totalPrice)

if(totalPrice > 15.00) then
 discount = (staffDiscount * totalPrice) + 0.50
else
 discount = staffDiscount * totalPrice
fi

print(“Discount: ” + discount)
finalPrice = totalPrice - discount
print(“Final price: ” + finalPrice)

endprogram
```


3
4
5
6
7
8
9
10
11
12
14
15
16
17
18
19

```
Class BinSearch {

        public static void search(int key, int[] elemArray, Result r) {      A
                int bottom = 0;
                int top = elemArray.length – 1;
                int mid;
                r.found = false; r.index=-1;
--------------------------------------------------------------------------------
                while (bottom <= top)    {                                   B
--------------------------------------------------------------------------------
                        mid = (top + bottom)/2;                              C
--------------------------------------------------------------------------------
                        if (elemArray [mid] == key) {                        D
--------------------------------------------------------------------------------
                                r.index = mid;
                                r.found = true;                              E
                                return;
                        }
--------------------------------------------------------------------------------
                        else {                                               F
                                if (elemArray[mid] < key)
--------------------------------------------------------------------------------
                                        bottom = mid + 1;                    G
--------------------------------------------------------------------------------
                                else top = mid – 1;                          H
                        }
                }
--------------------------------------------------------------------------------
        }                                                                    I

}
```


A
B
C
D
E
F
G
H
I
while bottom <= top
bottom> top
if (elemArray [mid] == key)
else
f (elemArray [mid] < key)
else
24



```
public class Authenticator {
   final int MAX = 10000;
   String [] userids = new String [MAX];
   String [] passwords = new String [MAX];
            ...
```

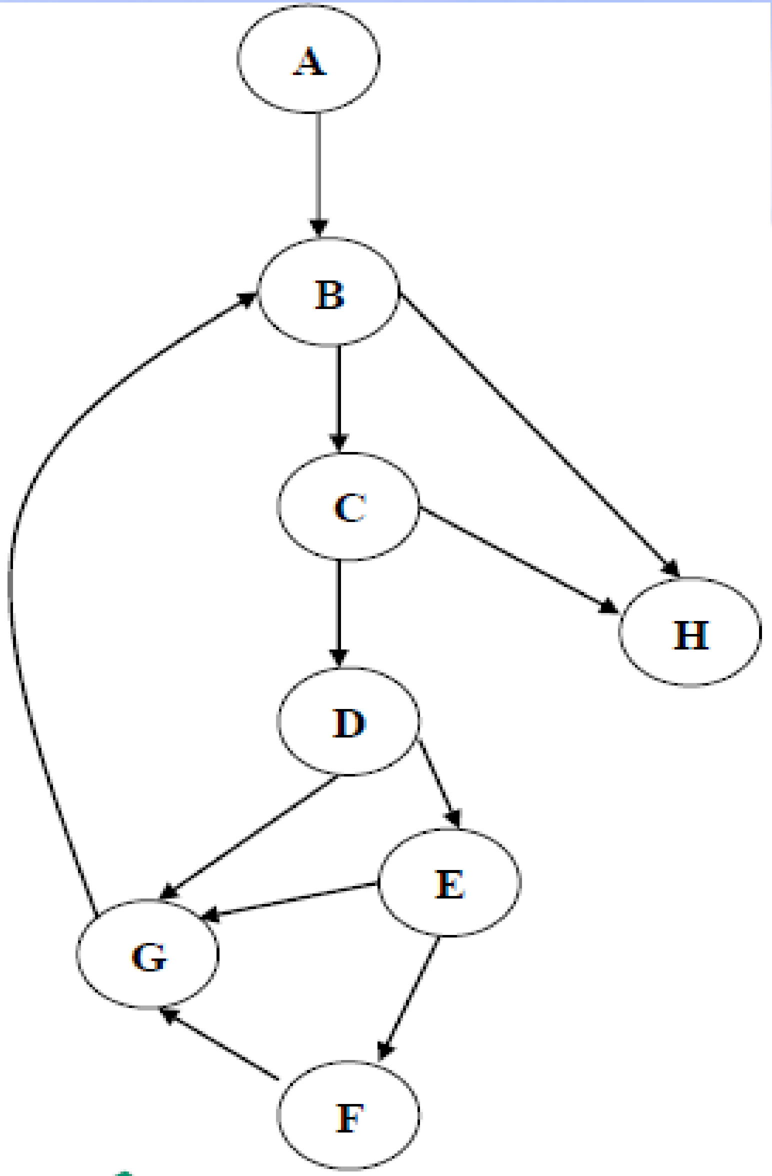
		Segn

`public boolean verify (String uid, String pwd) {`
`  boolean  result =false;`
`  int i = 0;`
		A

`  while ((result ==false)`	`&& (i < MAX)) {`	B, C
`      if ((userids[i]==uid)`	`&& (passwords[i] == pwd ))`	D, E
`          result=true;`		F

`          ++i;`
`  }`
		G

`  return result;`
`}`
		H


A
B
C
H
D
E
G
F

Data-flow coverage

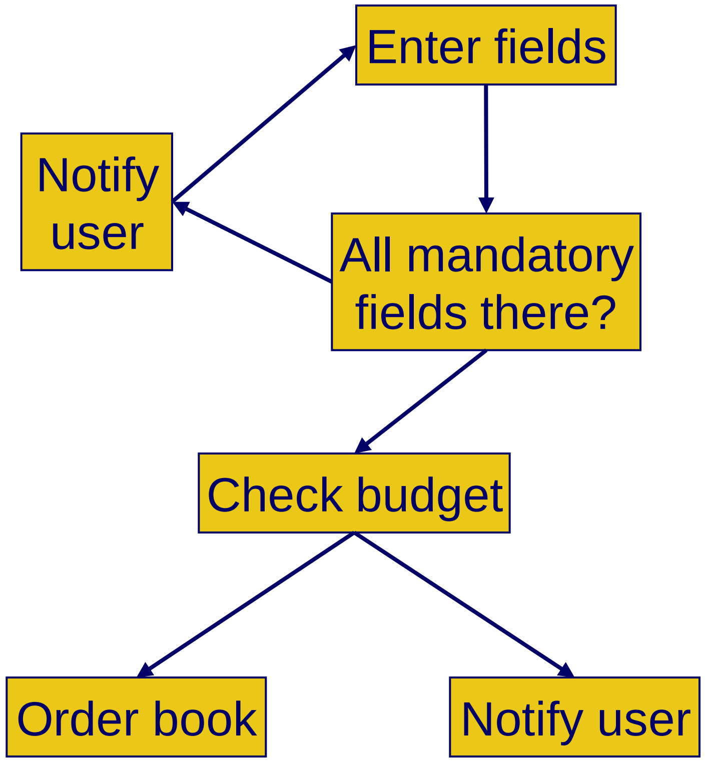
- Looks how variables are treated along paths through the control graph.
- Variables are *defined* when they get a new value.
- A definition in statement X is *alive* in statement Y if there is a path from X to Y in which this variable is not defined anew. Such a path is called *definition-clear*.
- We may now test all definition-clear paths between each definition and each use of that definition and each successor of that node: *All-Uses coverage.*
- *If each definition-clear path is required to be cycle free then it implies all-DU-paths coverage*

Coverage-based testing of requirements

- Requirements may be represented as graphs, where the nodes represent elementary requirements, and the edges represent relations (like yes/no) between requirements.
- And next we may apply the earlier coverage criteria to this graph

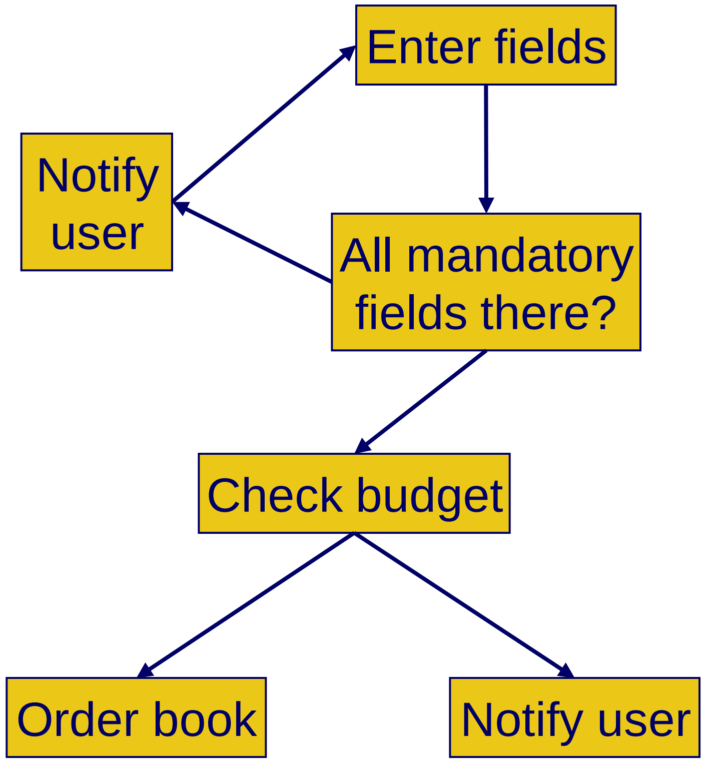
Example translation of requirements to a graph

A user may order new books. He is shown a screen with fields to fill in. Certain fields are mandatory. One field is used to check whether the department's budget is large enough. If so, the book is ordered and the budget reduced accordingly.



Similarity with Use Case success scenario

1. User fills form
2. Book info checked
3. Dept budget checked
4. Order placed
5. User is informed



Overview

- Preliminaries
- All sorts of test techniques
  - manual techniques
  - coverage-based techniques
  - **fault-based techniques**
  - error-based techniques
- Comparison of test techniques
- Software reliability

Fault-based testing

- **In coverage-based testing, we take the structure of the artifact to be tested into account**
- **In fault-based testing, we do not *directly* consider this artifact**
- **We just look for a test set with a high ability to detect faults**
- **Two techniques:**
  - Fault seeding
  - Mutation testing

Fault seeding

- **Conscious adding of errors to the source code**
  - One can try to estimate the number of ”real” errors in the code based on the number of seeded errors found
  - Evaluating the effectiveness of testing
- When the program is tested, we will discover both seeded faults and new ones.
- The total number of faults is then estimated from the ratio of those two numbers.

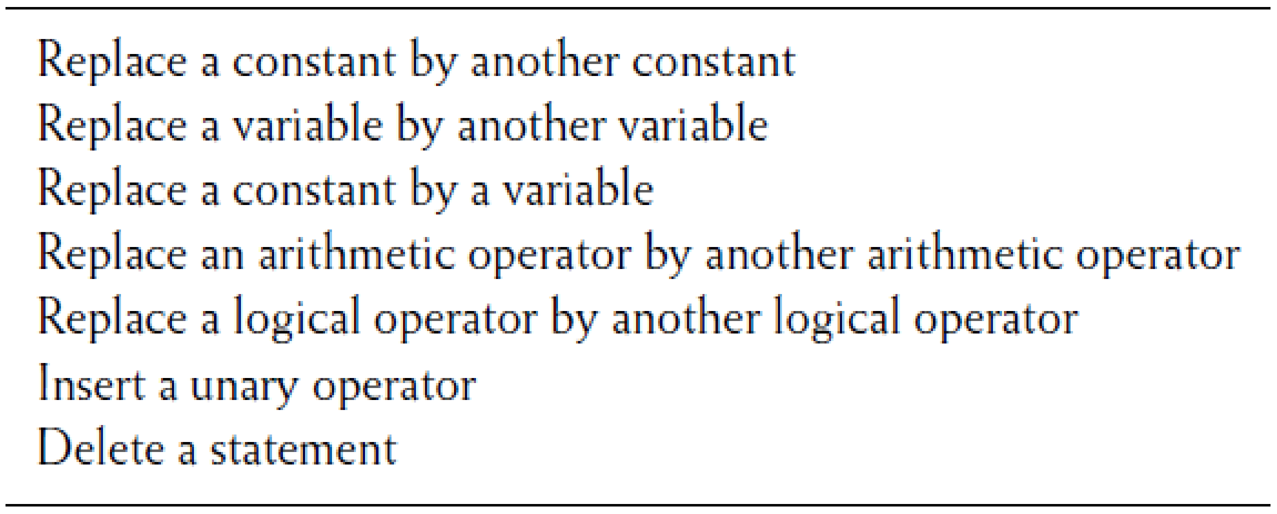
Mutation testing

```
procedure insert(a, b, n, x);
begin bool found:= false;
    for i:= 1 to $\cancel{n}$ do    $\boxed{n-1}$
        if a[i] = x
        then found:= true; goto leave endif
    enddo;
leave:
    if found
    then b[i]:= b[i] + $\cancel{1}$  $\boxed{2}$
    else n:= n$\cancel{+}$1; a[n]:= x; b[n]:= 1
    endif      $\boxed{-}$
end insert;
```

Mutation testing (cnt'd)

```
procedure insert(a, b, n, x);
begin bool found:= false;
    for i:= 1 to ~~n~~ do    n-1
        if a[i] = x
        then found:= true; goto leave endif
    enddo;
leave:
    if found
    then b[i]:= b[i] + 1
    else n:= n+1; a[n]:= x; b[n]:= 1
    endif
end insert;
```

Mutation operators



Replace a constant by another constant
Replace a variable by another variable
Replace a constant by a variable
Replace an arithmetic operator by another arithmetic operator
Replace a logical operator by another logical operator
Insert a unary operator
Delete a statement

Figure 13.15 A sample of mutation operators

- In mutation testing, a (large) number of variants of a program is generated.
- Each of those variants, or mutants, slightly differs from the original version.
- Usually, mutants are obtained by mechanically applying a set of simple transformations called mutation operators.
- All these mutants are executed using a given test set.
- As soon as a test produces a different result for one of the mutants, that mutant is said to be dead.
- Mutants that produce the same results for all of the tests are said to be alive.

How to use mutants in testing

- If a test produces different results for one of the mutants, that mutant is said to be *dead*
- If a test set leaves us with many live mutants, that test set is of low quality
- If we have M mutants, and a test set results in D dead mutants, then the *mutation adequacy score* is D/M
- A larger mutation adequacy score means a better test set

Strong vs weak mutation testing

- Suppose we have a program P with a component T
- We have a mutant T' of T
- Since T is part of P, we then also have a mutant P' of P

- In *weak mutation testing*, we require that T and T' produce different results, but P and P' may still produce the same results
- In *strong mutation testing*, we require that P and P' produce different results

Assumptions underlying mutation testing

- *Competent Programmer Hypothesis*: competent programmers write programs that are approximately correct

- *Coupling Effect Hypothesis*: tests that reveal simple fault can also reveal complex faults

Overview

- Preliminaries
- All sorts of test techniques
  - manual techniques
  - coverage-based techniques
  - fault-based techniques
  - **error-based techniques**
- Comparison of test techniques
- Software reliability

Error-based testing

- **Decomposes input (such as requirements) in a number of subdomains**
- **Tests inputs from each of these subdomains, and especially points near and just on the boundaries of these subdomains -- those being the spots where we tend to make errors**

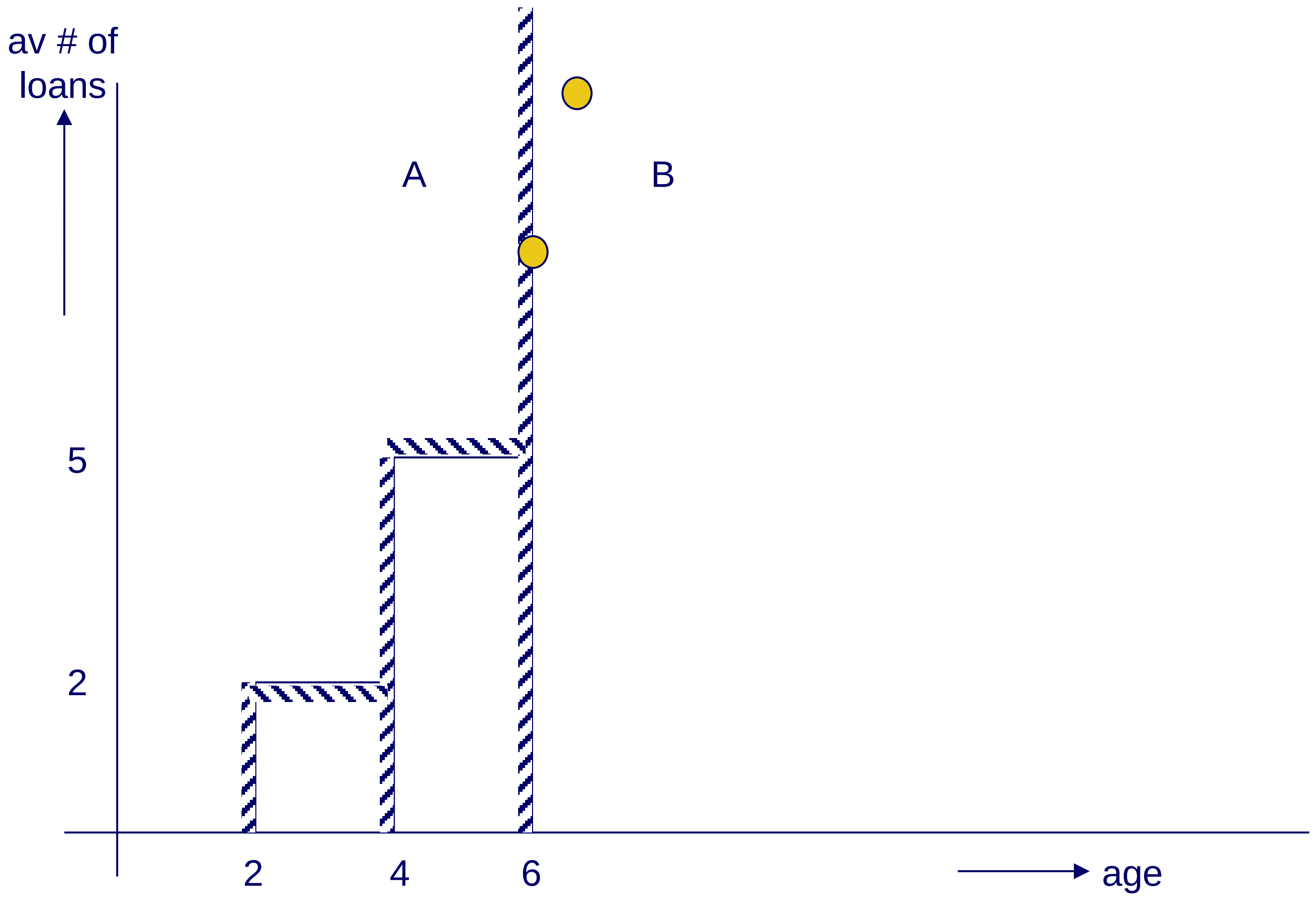
- **In fact, this is a systematic way of doing what experienced programmers do: test for 0, 1,** nil**, etc**

Error-based testing, example

Example requirement:

Library maintains a list of “hot” books. Each new book is added to this list. After six months, it is removed again. Also, if book is more than four months on the list, and has not been borrowed more than five times a month, or it is more than two months old and has been borrowed at most twice, it is removed from the list.

Example (cnt'd)



Strategies for error-based testing

- An ON point is a point on the border of a subdomain
- If a subdomain is open w.r.t. some border, then an OFF point of that border is a point just inside that border
- If a subdomain is closed w.r.t. some border, then an OFF point of that border is a point just outside that border

- So the circle on the line *age=6* is an ON point of both A and B
- The other circle is an OFF point of both A and B

Strategies for error-based testing (cnt'd)

- Suppose we have subdomains D_i, i=1,..n
- Create test set with N test cases for ON points of each border B of each subdomain D_i, and at least one test case for an OFF point of each border
- This set is called N$*$1 *domain adequate*

Application to programs

```
if x < 6 then …
elsif x > 4 and y < 5 then …
elsif x > 2 and y <= 2 then …
else ...
```

Overview

- Preliminaries
- All sorts of test techniques
  - manual techniques
  - coverage-based techniques
  - fault-based techniques
  - error-based techniques
- **Comparison of test techniques**
- Software reliability

Comparison of test adequacy criteria

- **Criterion A is stronger than criterion B if, for all programs P and all test sets T, X-adequacy implies Y-adequacy**
- **In that sense, e.g., All-Edges is stronger that All-Nodes coverage (All-Edges "subsumes" All-Nodes)**
- **One problem: such criteria can only deal with paths that can be executed (are feasible). So, if you have dead code, you can never obtain 100% statement coverage. Sometimes, the subsumes relation only holds for the *feasible* version.**

Desirable properties of adequacy criteria

- applicability property
- non-exhaustive applicability property
- monotonicity property
- inadequate empty set property
- antiextensionality property
- general multiplicity change property
- antidecomposition property
- anticomposition property
- renaming property
- complexity property
- statement coverage property

Experimental results

- There is no uniform best test technique
- The use of multiple techniques results in the discovery of *more* faults
- (Fagan) inspections have been found to be very cost effective
- Early attention to testing does pay off

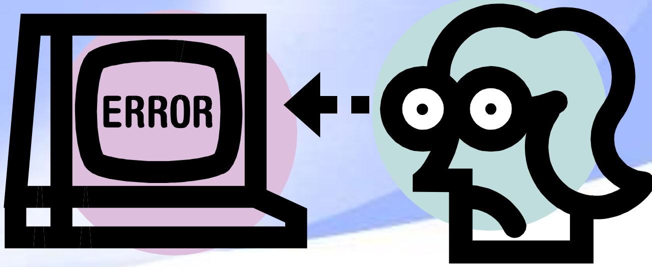
Overview

- Preliminaries
- All sorts of test techniques
  - manual techniques
  - coverage-based techniques
  - fault-based techniques
  - error-based techniques
- Comparison of test techniques
- **Software reliability**

Software reliability

- Interested in expected number of *failures* (not faults)
- … in a certain period of time
- … of a certain product
- … running in a certain environment

Software reliability: definition

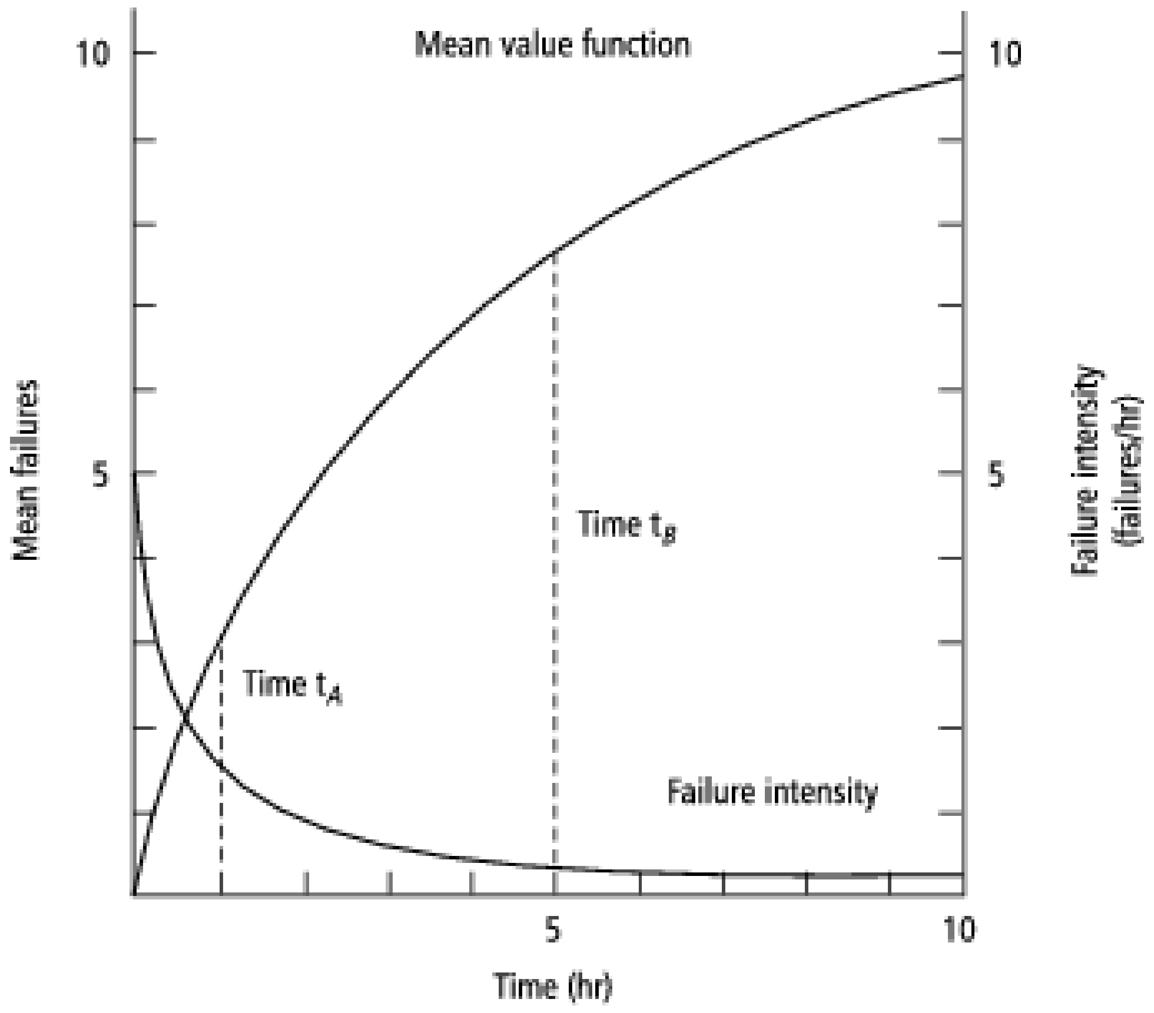


- Probability that the system will not fail during a certain period of time in a certain environment

Failure behavior

- **Subsequent failures are modeled by a stochastic process**
- **Failure behavior changes over time (e.g. because errors are corrected) $\Rightarrow$ stochastic process is *non-homogeneous***
- **$\mu(\tau)$ = average number of failures *until* time τ**
- **$\lambda(\tau)$ = average number of failures *at* time τ (failure intensity)**
- **$\lambda(\tau)$ is the derivative of $\mu(\tau)$**

Failure intensity $\lambda(\tau)$ and mean failures $\mu(\tau)$



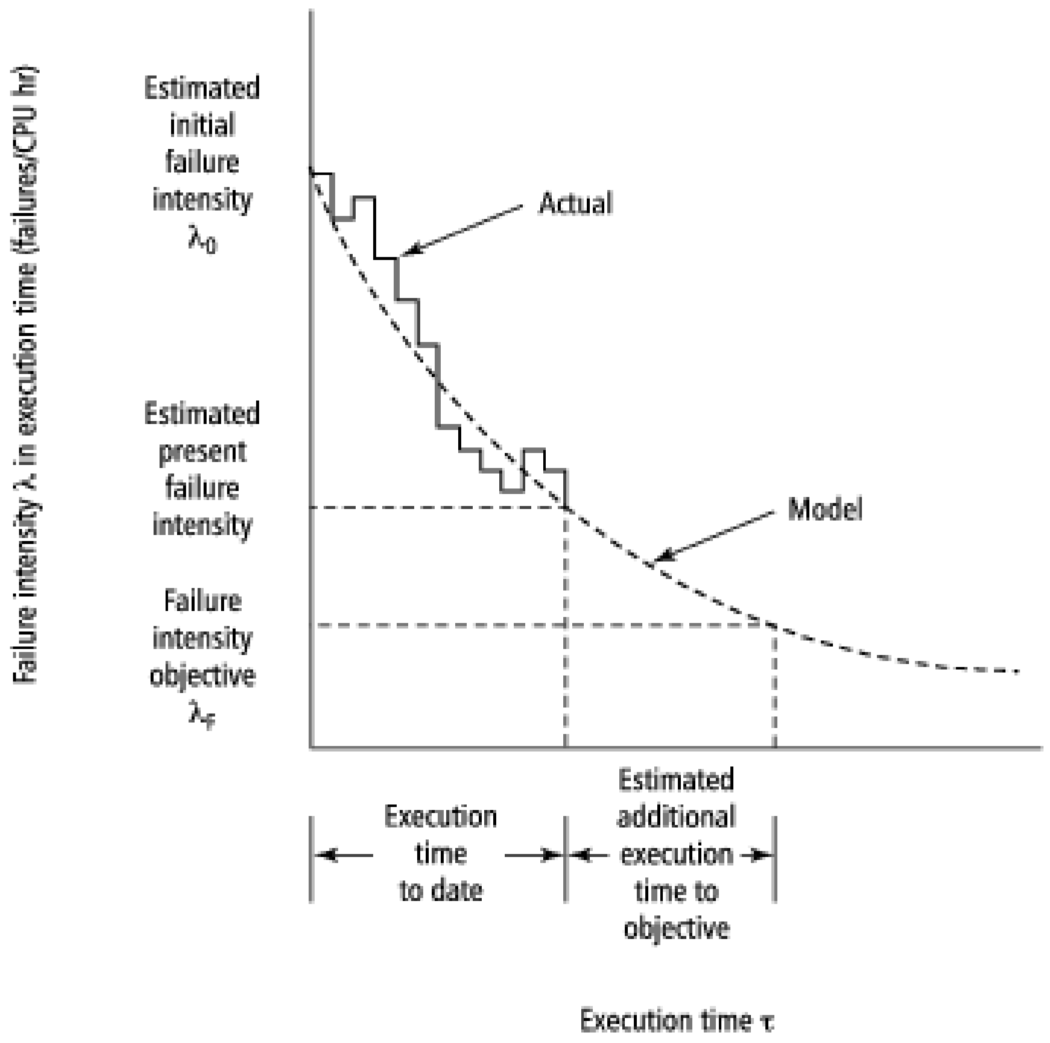
Operational profile

- Input results in the execution of a certain sequence of instructions
- Different input $\Rightarrow$ (probably) different sequence
- Input domain can thus be split in a series of equivalence classes
- Set of possible input classes together with their probabilities

Two simple models

- **Basic execution time model (BM)**
  - Decrease in failure intensity is constant over time
  - Assumes uniform operational profile
  - Effectiveness of fault correction is constant over time
- **Logarithmic Poisson execution time model (LPM)**
  - First failures contribute more to decrease in failure intensity than later failures
  - Assumes non-uniform operational profile
  - Effectiveness of fault correction decreases over time

Estimating model parameters (for BM)



Summary

- Do test as early as possible
- Testing is a continuous process
- Design with testability in mind
- Test activities must be carefully planned, controlled and documented.
- No single reliability model performs best consistently