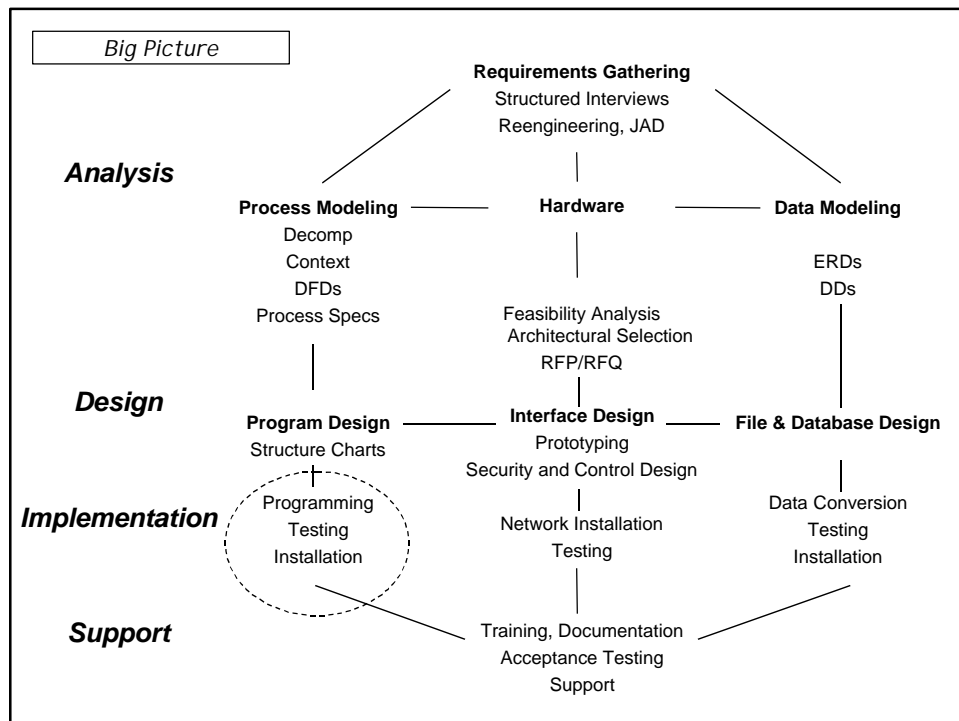


Implementation

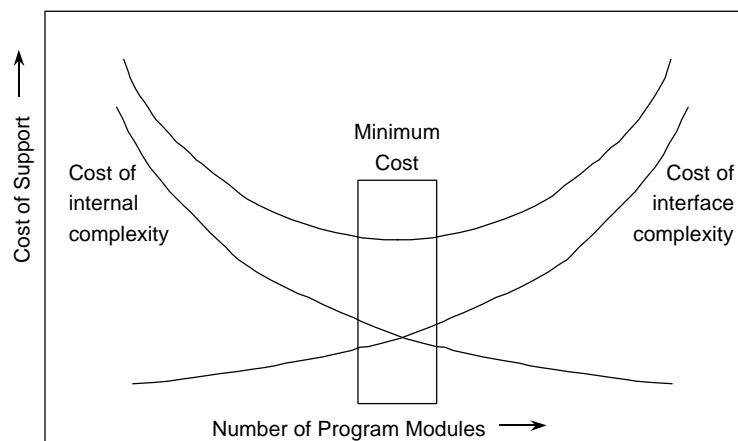
Overview
Packaging
Programming
Conversion
Testing



Implementation Activities

- Program Construction
- Validation and Testing
 - Programs, Systems
- Conversion, Installation
 - Data, Hardware, Software
- Training
- Documentation

Consider balancing number and size of modules before program construction



Refining your Program Design

- eliminate excessively small modules
 - unless they perform a common function
- factor out reusable modules
 - look for common code
- beware of RAM constraints
 - benchmark on typical and “minimum” systems

Finalizing your Design

- At this point, the analysis and design models are repackaged into two reports
- functional specifications
- technical specifications
- make sure they are “current”

Contents of Functional Specifications

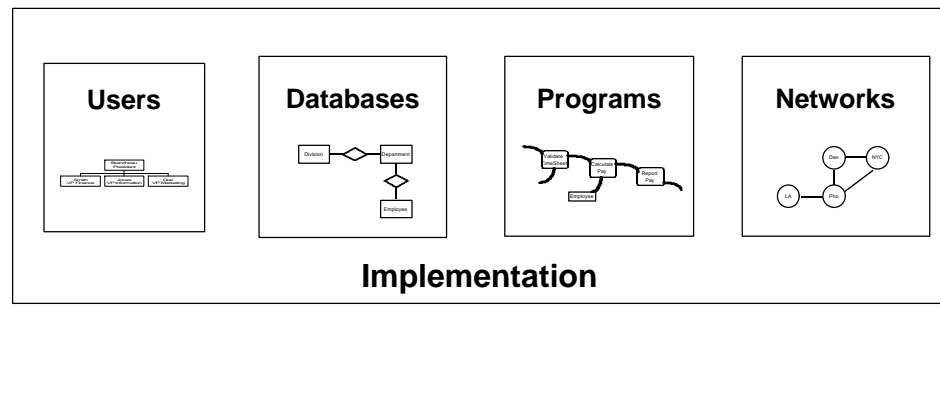
- general design
- processing functions
- forms, screens, reports
- manual & automated procedures
- conversion and installation plan
- high level technical requirements

Contents of Technical Specifications

- network design
- file/database design
- program/module design
- testing and conversion procedures
- detailed hardware & software requirements

Implementation Phase

- Entering the implementation phase, the models finally give way to the real thing.



Implementation Phase

- Build/Test Networks
- Build/Test Databases
- Build/Test Programs
- Install/Test System
- Deliver System

Build/Test Networks

- Infrastructure may already be in place
- Often done by specialists
- Network skills becoming critical in 1990s
- TCP/IP is current standard

Build/Test Databases

- May be part of infrastructure (often not)
- Analyst/designer always involved
- Analyst/designer sometimes leads effort
- Relational DBMS are current standard

Build/Test Programs

- Analyst/designer always involved
- Check for reusable software components
- For each module:
 - algorithm design
 - coding
 - testing
- Cobol, C, VB, C++, Smalltalk

Install/Test System

- Install software packages
- Test packages
- Conduct system test
- Prepare conversion plan

Deliver System

- Load files and databases
- Train client/users
- Wrap-up documentation
- Install (or convert to) new system
- Evaluate project and system

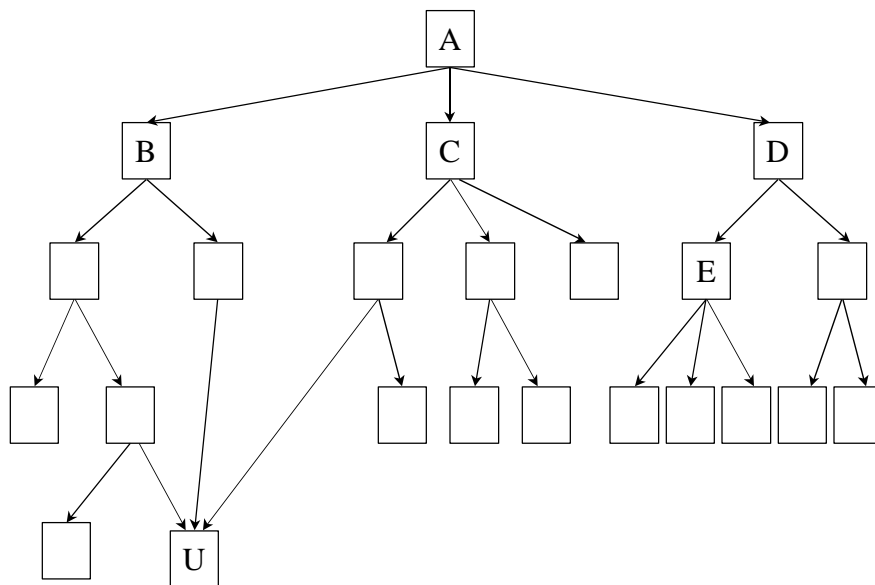
Key Deliverables from Implementation

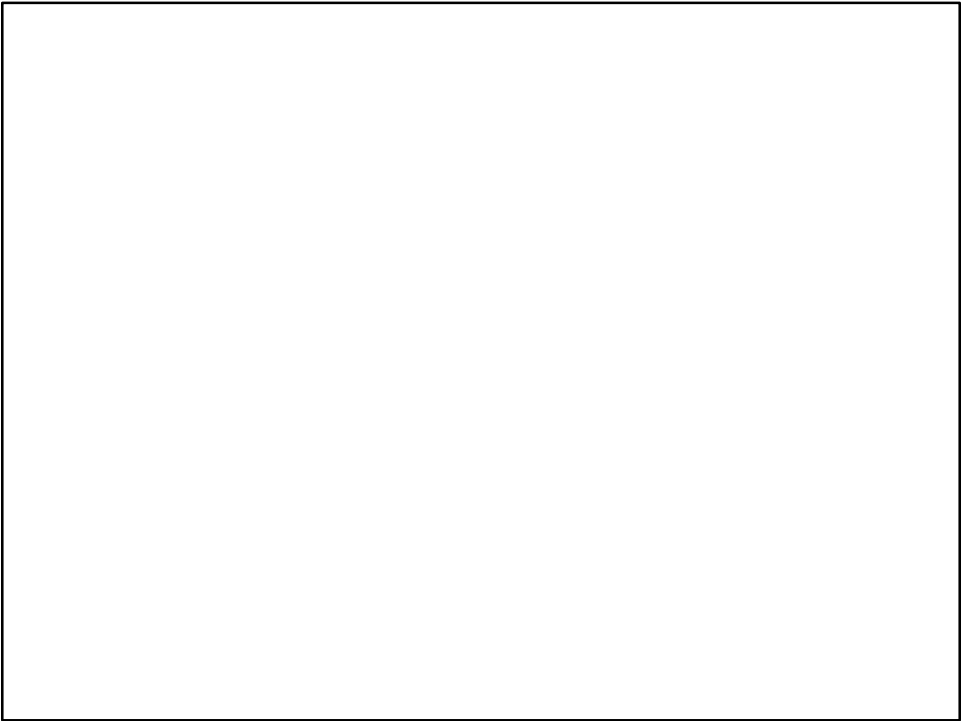
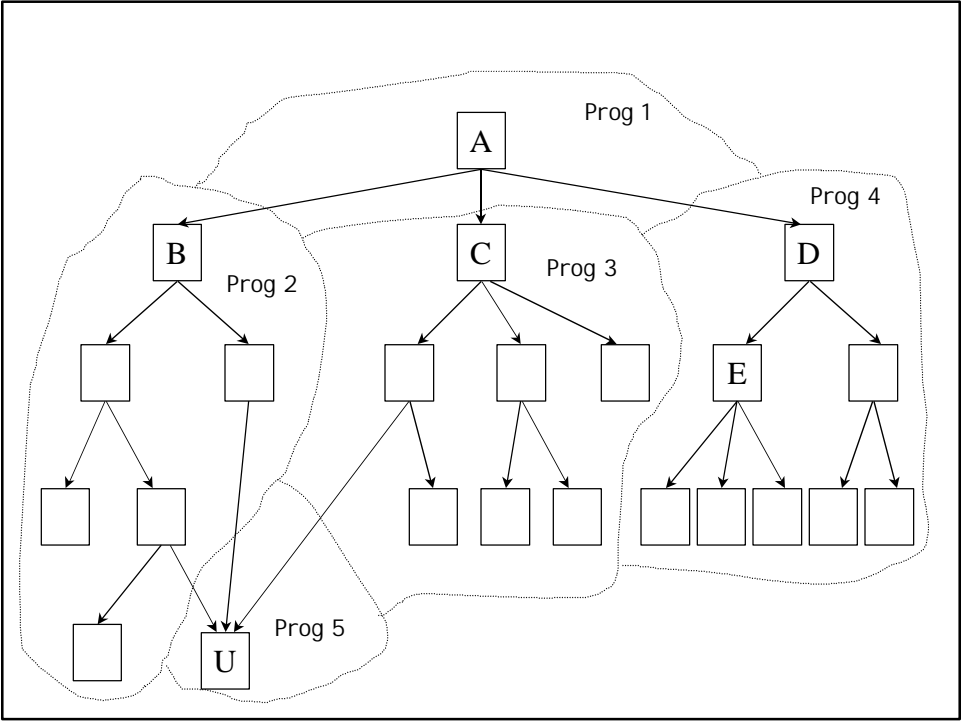
- operational network
- loaded files and databases
- tested programs and OS commands
- trained users
- documented system
- project evaluation

Packaging

- Every module should become a separate program (?): what is the problem with this approach?
- So, packaging into program units
- (In visual programming) program units may correspond to user interfaces or controls

Packaging





Planning the Programming Effort

- Review design specification
- Organize programming team
- Develop detailed programming plan
- Beware of our eternal optimism
 - Brook's law
 - mythical man month
 - 80/20 and 90/10 rules
 - inch pebbles vs. milestones

Organizing the Programming Effort

- chief programmer approach
 - organize around "superprogrammer"
- "pool" approach
 - specifications assigned to programming "pool"
- team approach
 - programmers assigned to project team
 - can be enhanced with "development center"

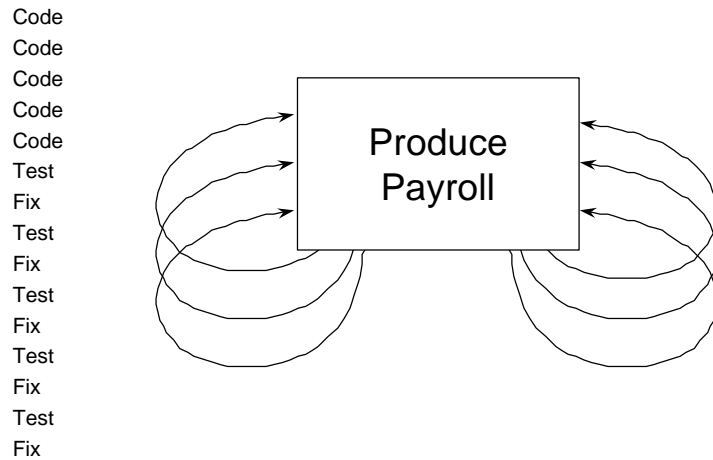
Programming Specifications

- In general, the more experienced the programmer, the less detailed the spec
 - program context and functions
 - unit test plan
 - file/database i/o
 - parameter i/o
 - program design
 - unit test data
 - psuedo code

Programming Approaches

- Monolithic
 - delays discovery of serious problems
 - multiplies need for resources late in project
 - pushes problems into operation/support
- Incremental
 - allows testing to start earlier
 - makes partial version available earlier
 - distributes work load more evenly

Big Program (Monolithic) Approach



Incremental Approaches

- bottom-up
 - a "concatenation" process
 - requires "drivers"
 - facilitates generation test data
 - facilitates broader delegation of programming
- top-down (versioned or phased)
 - a "refinement" process
 - requires use of "stubs"
 - tests most critical modules early

Structured Programming

- objective
 - saves money over the long-run
- definition (more than avoiding GOTO)
 - basic constructs
 - hierarchical structure
 - style conventions
 - internal documentation

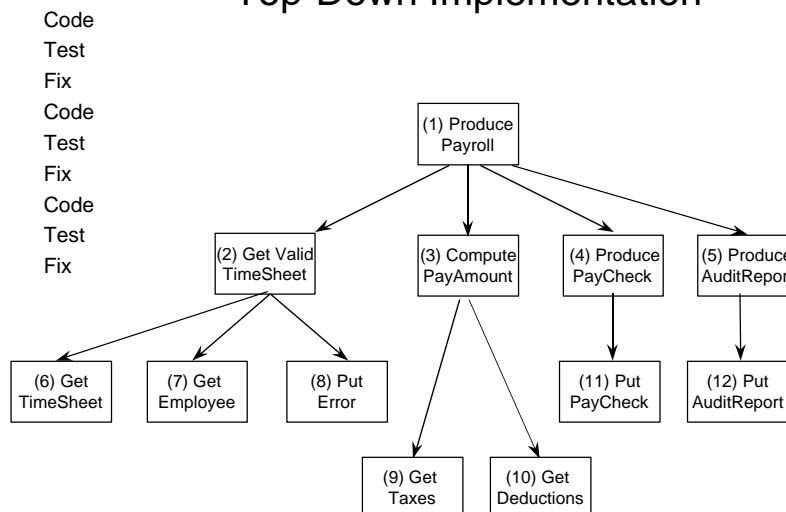
Structured Programming Constructs

- sequence
 - any code block with single entry, single exit
- condition
 - if ... then ... else ... endif
 - if ... then ... elseif ... elseif ... endif
 - case ... endcase
- repetition
 - do while ... enddo (with test before)
 - do until ... enddo (with test after)

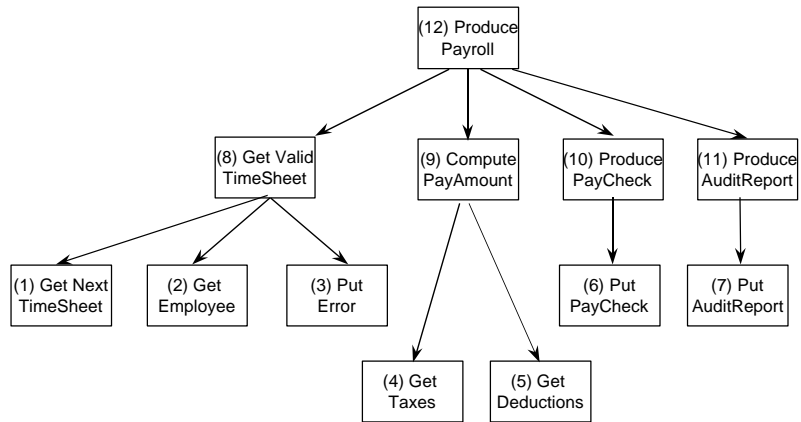
Effective Programming Style

- write one instruction per line
- use meaningful names
- use common prefixes for related elements
- avoid unnecessary labels
- use indentation and spatial alignment
- use parentheses to show precedence
- avoid programming tricks
- use comments
- follow a standard

Top-Down Implementation



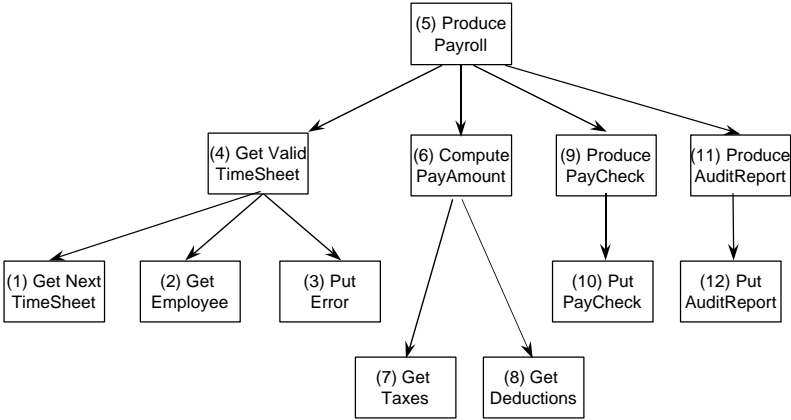
Bottom-up Implementation



Top-Down vs. Bottom-Up

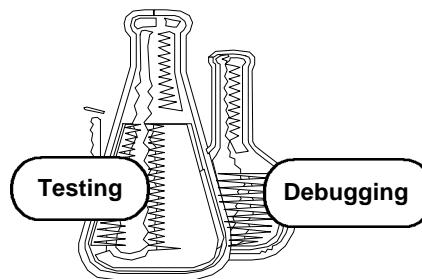
- should we build from top-down or bottom-up?
 - both attempt to order the programming
 - both encourage modular programming
- combination approach is often best
 - bottom-up on input-side
 - top-down on output-side

Combination Approach



Testing

Definitions



- ★ Testing is the process of exercising a program with the intent of finding errors
- ★ Debugging .NE. Testing
- ★ Debugging is diagnosing and correcting known errors; testing, therefore, is the process of locating and correcting presently unknown errors

Levels of Testing

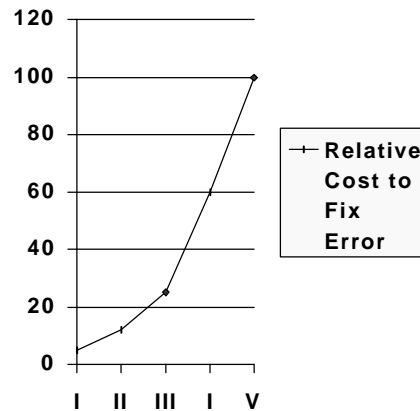
- Module testing -- testing a single module out of context, with no interfaces to other modules
- Integration testing -- verifying the interfaces among modules
- Function testing -- verifying that the system meets specs
- System testing -- verifying that the system meets user objectives

Psychology of Testing

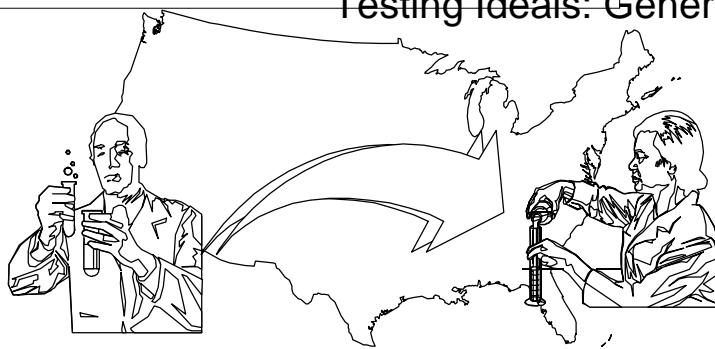
- Testing is the process of executing a program with the intent of finding errors
- A successful test finds errors
- Remember that programmers have ego involvement (they don't want to find errors in their own code, and don't want others to find them either)

Economics of Testing

- The earlier you can catch the error, the better

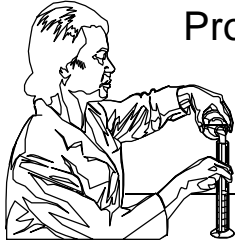


Testing Ideals: General



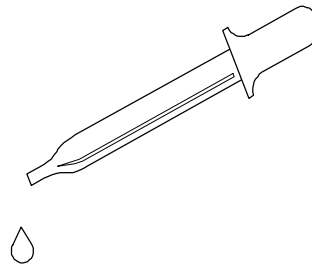
- The programming team should not examine its own programs
- A program should be examined to ensure that they:
 - do what they are supposed to do
 - do only what they are supposed to do

Testing Ideals: Programmers & End Users



- End user involvement is critical
- Programmers are blind to their errors
- End users are (unintentionally) a hostile environment for programmers: possibly on different paradigm

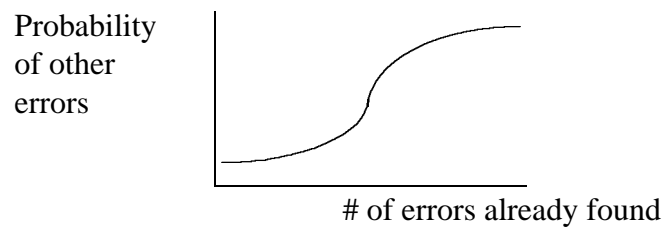
Testing Ideals: Test Cases



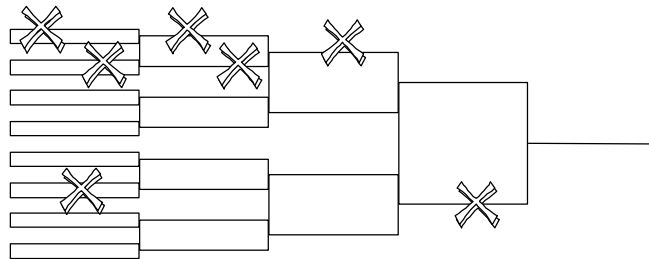
- ✓ Test case design begins with a definition of the possible and/or expected outputs or results
- ✓ Nevertheless, test cases should be planned for unexpected and invalid inputs/user actions as well as valid and expected conditions
- ✓ Test cases should not be designed or created expecting that no errors will be found

Testing Principles

- The probability that there are more errors in a section is proportional to the number of errors already found in that section (errors are not evenly distributed)

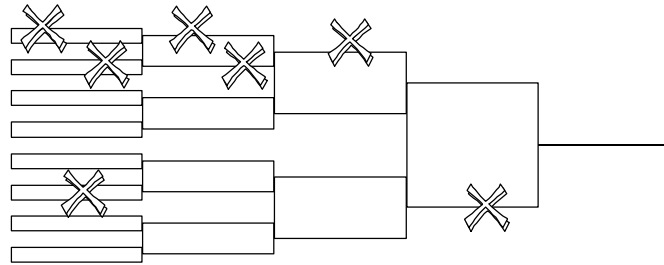


Programming Errors: Where Errors Occur



- ✓ The probability that there are more errors in certain program units--objects in a hierarchy, form, page, group of objects, or a section of code--is proportional to the number of errors already found in that unit
 - i.e., errors are not evenly distributed
- ✓ 20% of these program units have 80% of the errors

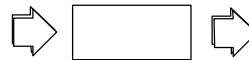
Programming Errors: Why Errors Occur in this Way



- ✓ Sloppy programmers tend to make a larger number of errors than more careful programmers
- ✓ Some program units are more frequently used than others

Testing Methods

✓ **Black Box (Input-Driven Method)**



✓ **White Box (Logic-Driven Method)**



✓ **Error Guessing**

|



White Box Testing

- Also Known As Logic Coverage Testing
- Consists of
 - Statement coverage: execute all statements at least once
 - Decision coverage: do all decisions at least once
 - Condition coverage: do all conditions at least once
 - Condition-Decision coverage: previous coverage plus all points of entry are invoked at least once
 - Multiple condition coverage -- all possible combinations of condition outcomes

Logic Coverage Testing

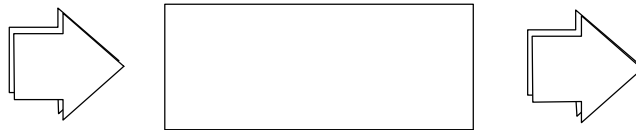
- Disadvantages
 - Does not detect specification errors
 - Cannot detect absence of necessary paths
 - Insensitive to data since it only tests logic

Black Box Testing

- Compares outputs with what is expected
- Does the module do what it is supposed to do, and nothing else
- Infinite number of inputs which must be tested



Black Box Testing



- Input Driven
- Types
 - Equivalence Partitioning
 - Boundary Analysis
 - Cause Effect Graph

Equivalence Partitioning

- Both valid and invalid

| <i>Classes of Conditions</i> | <i>Sample Requirement</i> | Valid | Invalid |
|--|---|--------------|--------------------|
| Item in range--test 1 within range and 1 on each end | Item count can be from 1 to 999 | 3, 6, 9 | 0, 1062, -100 |
| Multiple values-- test several & 1 on each end | Car can have up to 6 owners | Jones, Smith | no owner, 9 owners |
| Limited set of values-- test 1 valid & 1 invalid | Type of vehicle limited to: bus, truck, passengercar, cycle | Bus | Taxicab |
| Required positions-- test 1 that is and 1 that isn't | Customer id must begin with a letter | B8 | 83 |

Boundary Analysis

- A variant of equivalence partitioning:
Equivalence partitioning at the boundary

| <i>Classes of Conditions</i> | <i>Sample Requirement</i> | Valid | Invalid |
|---|---|-----------|--------------------|
| Item in range-- 1 within range & 1 on each end | Item count can be from 1 to 999 | 1, 999 | 0, 1000 |
| Multiple values-- test several & 1 on each end | Car can have up to 6 owners | Jones | no owner, 7 owners |
| Limited set of values-- test 1 valid & 1 invalid | Type of vehicle limited to: bus, truck, passengercar, cycle | --- | --- |
| Required positions--test 1 that is & 1 that isn't | Customer id must begin with a letter | A, A1, ZZ | 0A, 9Z |

Cause-Effect Graph

- Test all combinations of causes as inputs against the valid effects

Causes

In-State Student
Graduate Student
Professional Student

Effects

Low Tuition
Moderate Tuition
High Tuition
Outrageous Tuition

| A | B | C | D | E | F | G | H |
|---|---|---|---|---|---|---|---|
| Y | Y | Y | Y | N | N | N | N |
| Y | Y | N | N | Y | Y | N | N |
| Y | N | Y | N | Y | N | Y | N |
| | | | | | | | |
| X | X | X | X | | | | |
| | | | | | | | X |
| | | | | | X | X | |
| | | | | X | | | |

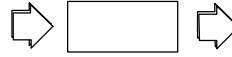
Error Guessing



- Intuitive, based on experience
- May be incomplete, but useful supplement to other testing techniques
- May involve language dependent problems: eg. dBase III has a limitation on the number of memory variables

Conclusively, Heuristics of Testing

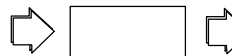
✓ If combinations of input conditions or end user actions are a likely problem, start with cause-effect graphing.



✓ Always use boundary analysis.



✓ Use one logic coverage technique.



✓ Add some test cases using error guessing.



System Testing

- Volume testing (normal volume)
- Stress testing (peaks and surges)
- Usability testing (human factors)
- Security testing
- Performance testing
- Storage testing
- Reliability testing (mtbf)
- Recovery testing

Debugging

- Brute force (print stmts, dumps)
- Induction (hypothesize based on data, test)
- Deduction (list possible causes, test each)
- Backtracking
- Testing
- Error analysis (when was error made, who made it, what was done incorrectly, could it have been prevented, why wasn't it detected earlier)

Debugging Principles

- Think. If you reach an impasse, sleep on it
- Describe the problem to someone else
- Avoid experimentation
- Where there is one bug, there are others
- Fix the error, not the symptom
- The probability of the fix being correct is not 100%
- The probability of the fix being correct drops as the program gets larger
- Error correction may create new errors