

# Software Testing & Quality Assurance

## *Boundary Value Testing*

- ❖ Boundary value analysis
- ❖ Single fault assumption
- ❖ Robustness testing
- ❖ Worst case testing
- ❖ Special value testing
- ❖ Random testing

# Credits & Readings

The material included in these slides are mostly adopted from the following books:

- *Software Testing: A Craftsman's Approach*, by Paul Jorgensen, CRC PRESS, third edition, ISBN: 0-8493-7475-8
- Cem Kaner, Jack Falk, Hung Q. Nguyen, “*Testing Computer Software*” Wiley (see also <http://www.testingeducation.org/>)
- Cem Kaner, James Bach, Bret Pettichord, “*Lessons Learned in Software Testing*”, Wiley
- Paul Ammann and Jeff Offutt, “*Introduction to Software Testing*”, Cambridge University Press
- Kent Beck, “*Test-driven Development by Example*” Addison-Wesley
- Robert Binder, “*Testing Object-Oriented Systems: Models, Patterns, and Tools*” Addison-Wesley
- Glen Myers, “*The Art of Software Testing*”

# Introduction

- *Input domain testing* is the most commonly taught (and perhaps the most commonly used) software testing technique
- We will see a number of approaches to *boundary value analysis*
- We will then study some of the limitations of domain testing

# What is Boundary Value Analysis?

- Many programs can be viewed as a function  $F$  that maps values from a set  $A$  (its domain) to values in another set  $B$  (its range)

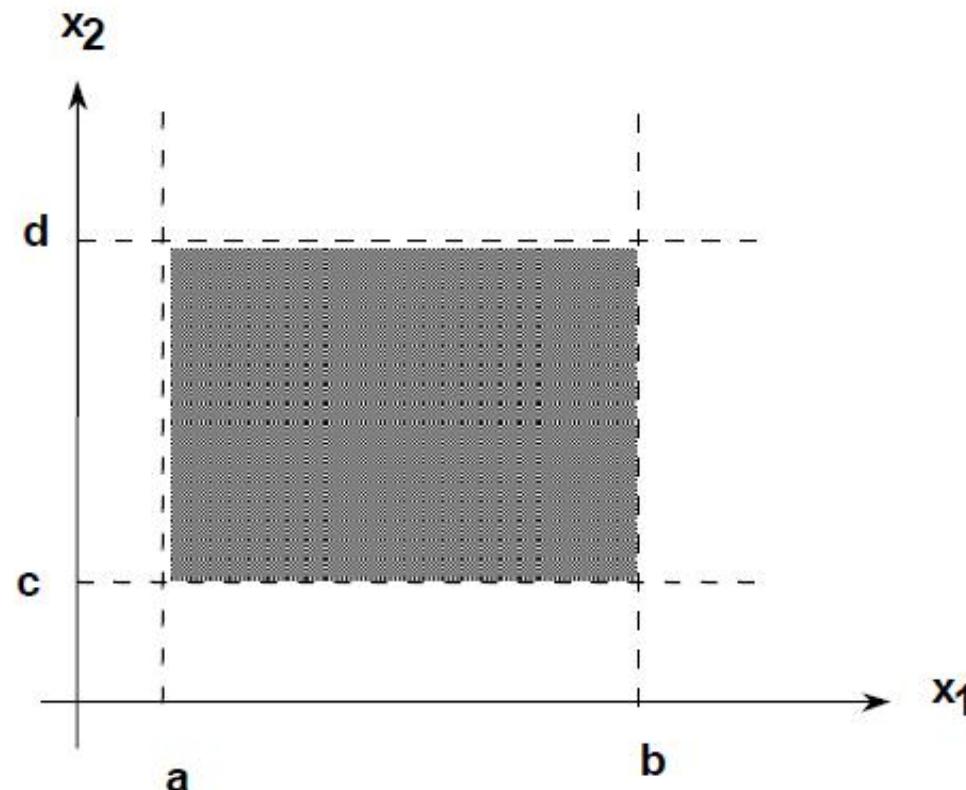
$$F : A \rightarrow B$$

- The input variables of  $F$  will have some (possibly unstated) boundaries:

$$a \leq x_1 \leq b$$

$$c \leq x_2 \leq d$$

# Input domain for variables $x_1$ and $x_2$



# What does BVA focus on?

- It focuses on the boundary of the input space to identify test cases
- The rationale behind this is that errors tend to occur near the extreme values of an input variable
  - Many times when a system is tested the function result is “off by one”

# What is the basic idea?

- To use input variable values at their...
  - Minimum
  - Just above the minimum
  - A nominal value
  - Just below their maximum
  - At their maximum

# The “single fault” assumption

- Failures are only rarely the result of the simultaneous occurrence of two (or more) faults

# How does BVA work?

- Boundary value analysis test cases are obtained by
  - holding the values of all but one variable at their nominal values, and
  - letting that variable assume its extreme values i.e.
    - Minimum
    - Just above the minimum
    - Nominal
    - Just below the maximum
    - Maximum

# Example: BVA test cases for a two-variable function

$\langle X_{1\text{nom}}, X_{2\text{min}} \rangle$

$\langle X_{1\text{nom}}, X_{2\text{min+}} \rangle$

$\langle X_{1\text{nom}}, X_{2\text{nom}} \rangle$

$\langle X_{1\text{nom}}, X_{2\text{max-}} \rangle$

$\langle X_{1\text{nom}}, X_{2\text{max}} \rangle$

$\langle X_{1\text{min}}, X_{2\text{nom}} \rangle$

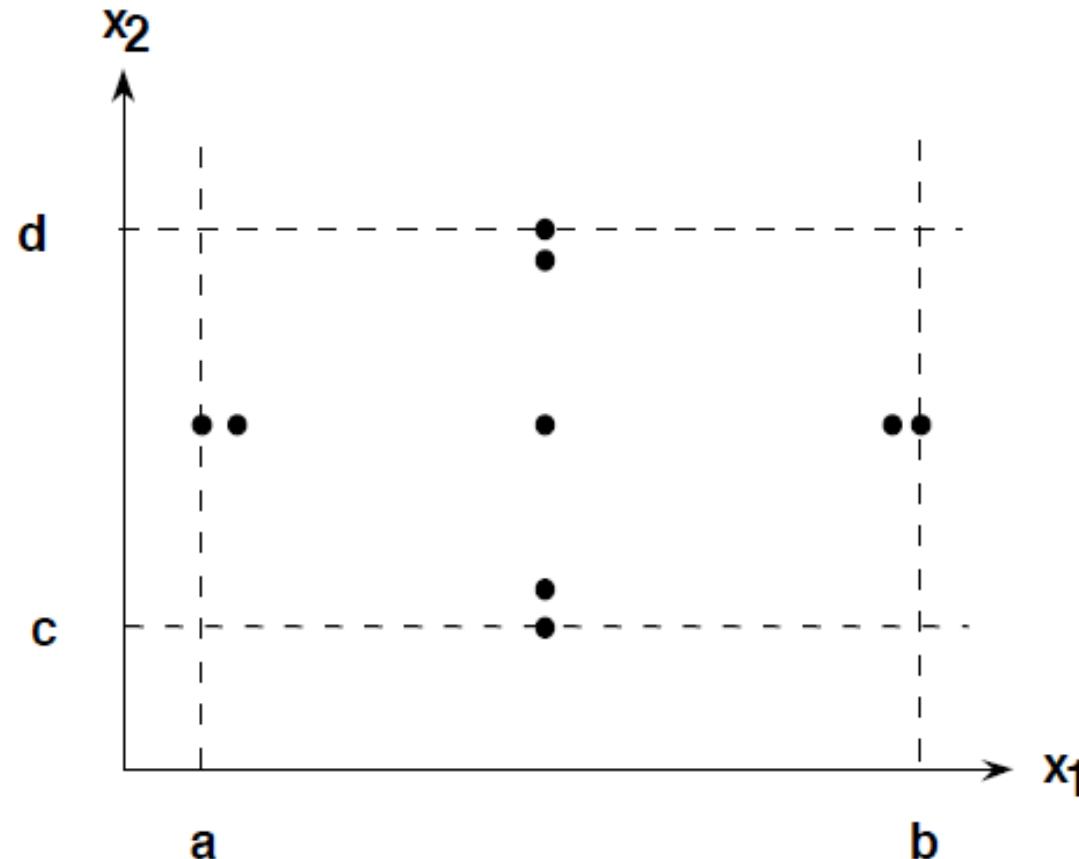
$\langle X_{1\text{min+}}, X_{2\text{nom}} \rangle$

$\langle X_{1\text{nom}}, X_{2\text{nom}} \rangle$

$\langle X_{1\text{max-}}, X_{2\text{nom}} \rangle$

$\langle X_{1\text{max}}, X_{2\text{nom}} \rangle$

# BVA test cases for $x_1$ and $x_2$



# In-class activity

- Let's apply BVA to the
  - Adder program
    - Input domain: 2 integers (2-digits each)
  - Triangle problem
    - Input domain: 3 sides (values 1-200 each)

# Limitations

- Boundary value analysis works well when the program is a function of several independent variables that represent bounded physical quantities
  - Physical boundaries can be extremely important
    - Airport in Phoenix shut down
      - Air temperature was 122 degrees
      - Instruments could only accept up to 120 degrees
- It won't work for the *NextDate* program because
  - dependencies exist among the *month*, *day* and *year* variables
- It does not work well for logical variables
  - Customer's PIN in ATM, transaction type
- It does not work well for Boolean variables
  - Boolean variables lend themselves to *decision table-based* testing (we will discuss later)

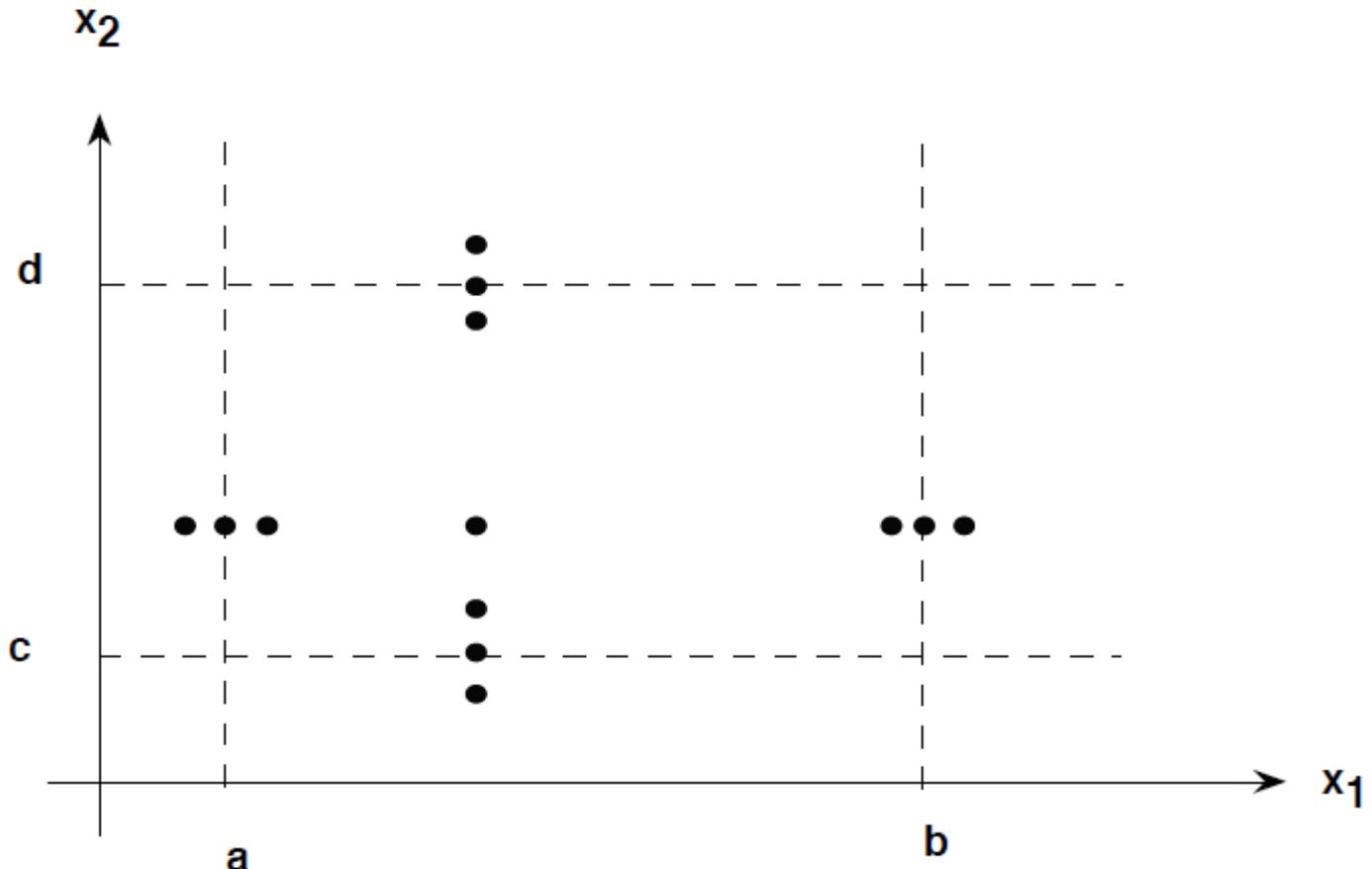
# How can BVA be generalized?

- It can be generalized in 2 ways:
  - By the number of variables
  - By the number of ranges
- Generalizing the number of variables is easy
  - Function of  $n$  variables, we hold all but one at the nominal values and let remaining variable assume min, min+, nom, max-, and max
    - It makes  $4n+1$  test cases

# Robustness testing

- A simple extension to boundary value analysis
- Add two more values per variable
  - Slightly greater than the maximum
  - Slightly less than the minimum
- What is the expected output?
  - Hopefully error message, system recovers

# Robust testing: test cases for $x_1$ and $x_2$



# In-class activity

- Let's apply robust testing to the
  - Adder program
    - Input domain: 2 integers (2-digits each)
  - Triangle problem
    - Input domain: 3 sides (values 1-200 each)

# Worst case testing

- Rejects the single fault assumption and tests all combinations of values
- Instead of  $5n$  test cases, we have  $5^n$
- Often leads to a large number of test cases with low bug-finding power
  - Usually better to apply special value testing (i.e. test cases based on the tester's intuition)
- Best application for worst-case testing when
  - physical variables have numerous interactions
  - failure of the function is extremely costly
  - Use of automated testing tools

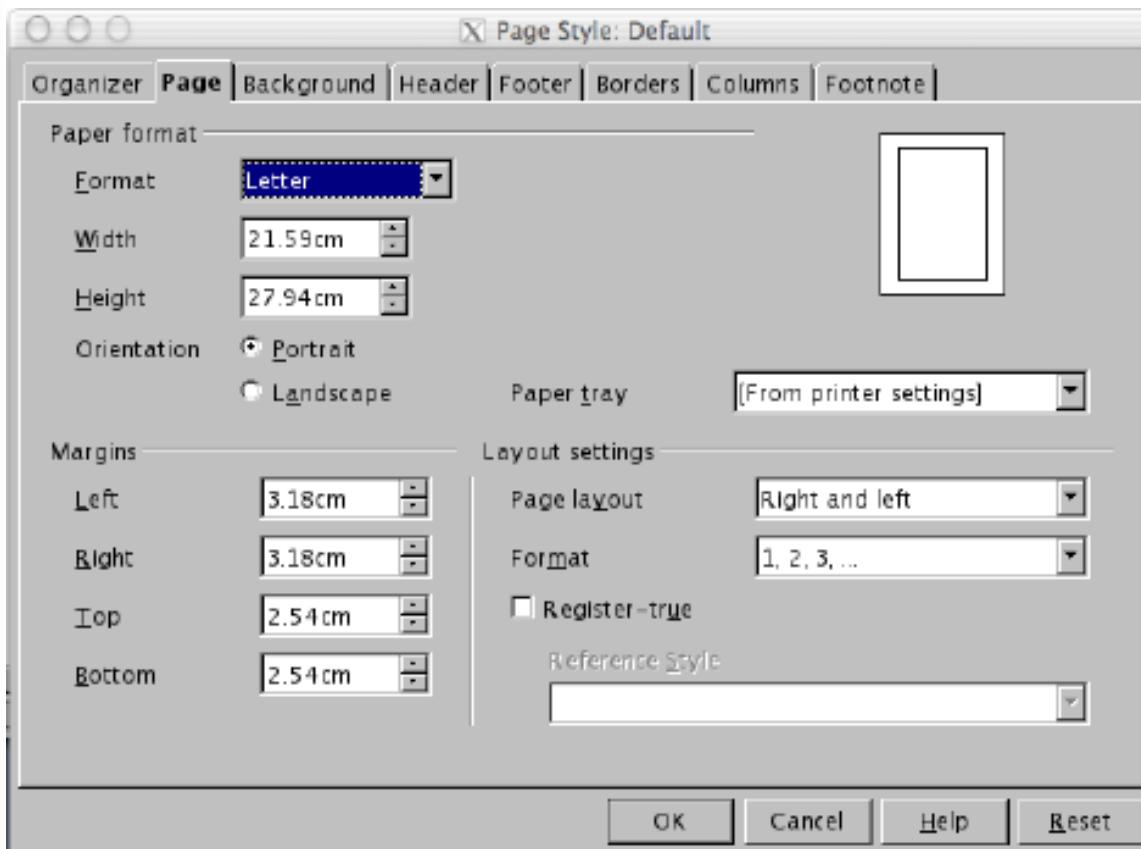
# Special value testing

- Most widely practiced form of functional testing
- Most intuitive and least uniform
- Occurs when a tester uses his/her domain knowledge, experience with similar programs, and information about “soft spots” to devise test cases
  - Also called “ad hoc testing” or “seat-of-the-pants” testing
  - No guidelines used other than best engineering judgment
  - Can be very useful, often more effective in revealing error results

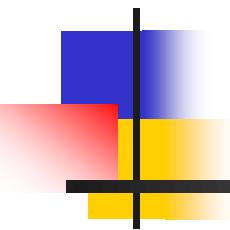
# Random testing

- Besides always choosing min, max, min+....
  - Use a random number generator to pick test case values
  - Avoids biases in testing

# In class activity



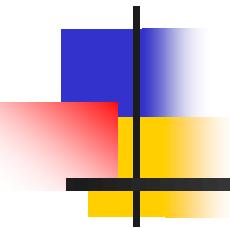
- Do a domain analysis on page width and height
  - BVA
  - Robust testing
  - Special value
- Assume the spec mentions that
  - Width values between 10cm and 60cm should be handled
  - Height values between 20cm and 100cm
- Can you identify any weaknesses of BVA?



# Software Testing & Quality Assurance

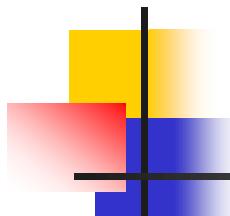
## *Reporting*

- Analyzing bugs
- Reproducing bugs and
- Reporting bugs effectively



# Credits & Readings

- The material included in these slides are adopted from the following resources:
  - Cem Kaner, Jack Falk, Hung Q. Nguyen, "*Testing Computer Software*" Wiley
  - The Testing Education & Research Center: <http://www.testingeducation.org/>



# Terminology review

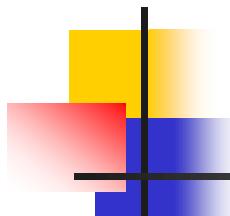
- Here's a defective program segment:

INPUT A

INPUT B

PRINT A / B

- What is the error? What is the fault?
- What is the critical condition (that triggers failure)?
- What will we see as the incident of the failure?



# Bug reporting

- Testers report bugs to software developers
- Problem (bug) report forms are commonly used
- If the report is not clear and understandable, the bug will not get fixed
- To write a fully effective report you must:
  - Explain how to reproduce the problem
  - Analyze the error so that it can be described with a minimum number of steps
  - Write a report that is complete, easy to understand, and non-antagonistic

# A typical problem report form\*

**PROBLEM REPORT #** \_\_\_\_\_

**REPORTER** \_\_\_\_\_

**DATE** \_\_\_\_ / \_\_\_\_ / \_\_\_\_

**PROGRAM** \_\_\_\_\_

**RELEASE** \_\_\_\_\_

**VERSION** \_\_\_\_\_

**CONFIGURATION** \_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_

**REPORT TYPE** \_\_\_\_\_

1 - Coding error 4 - Documentation  
2 - Design Issue 5 - Query  
3 - Suggestion

**SEVERITY** \_\_\_\_\_

1 - Fatal  
2 - Serious  
3 - Minor

**ATTACHMENTS (Y/N)** \_\_\_\_\_

*Description:* \_\_\_\_\_  
\_\_\_\_\_

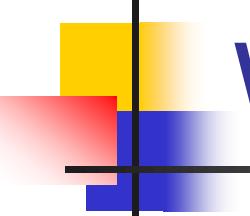
## PROBLEM SUMMARY \_\_\_\_\_

**CAN YOU REPRODUCE THE PROBLEM (Y/N/S/U)** \_\_\_\_\_

**PROBLEM AND HOW TO REPRODUCE IT** \_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_

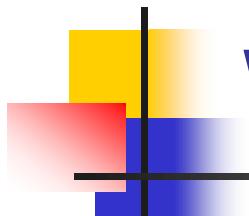
**SUGGESTED FIX (optional)** \_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_

\*Cem Kaner, Jack Falk,  
Hung Q. Nguyen,  
*“Testing Computer  
Software”*



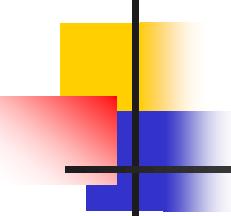
# What kind of errors to report?

- You may report any of the following:
  - Coding Error: The program doesn't do what the programmer expects it to do
  - Design Issue: It's doing what the programmer intended, but a typical customer would be confused or unhappy with it
  - More on the next slide...



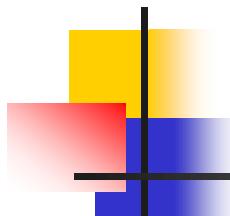
# What kind of error to report?

- Requirements Issue: The program is well designed and well implemented, but it won't meet some of the customer's requirements
- Documentation / Code Mismatch: Inconsistency between the code and related documentation
  - Report this to the programmer (via a bug report) and to the writer (usually via a memo or a comment on the manuscript)
- Specification / Code Mismatch: Sometimes the spec is right; sometimes the code is right and the spec should be changed



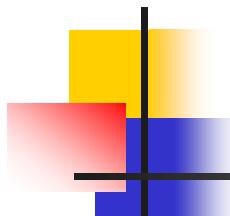
# Importance of bug reports

- A *bug report* is a tool that you use to sell the programmer on the idea of spending his/her time and energy to fix a bug
  - Motivate the buyer (i.e. make him/her WANT to fix the bug)
  - Overcome objections (i.e. get past his/her excuses and reasons for not fixing the bug)
- Bug reports are your primary work product as a tester
  - This is what people outside of the testing group will most notice and most remember of your work
- The best tester isn't the one who finds the most bugs or who embarrasses the most programmers
  - An effective tester is the one who gets the most bugs fixed



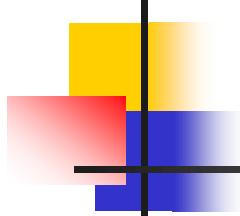
# Motivating the bug fixer

- Some reasons that will often make programmers want to fix the bug:
  - It looks really bad
  - It looks like an interesting puzzle and piques the programmer's curiosity
  - It will affect lots of people
  - Getting to it is trivially easy
  - It has embarrassed the company, or a bug like it embarrassed a competitor
  - Management (that is, someone with influence) has said that they really want it fixed



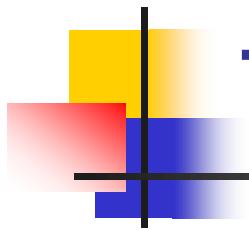
# Show the magnitude of faults

- Identifying the symptoms versus curing the disease
  - When you run a test and find a failure, you're looking at a symptom, not at the underlying fault
  - You may or may not have found the best example of a failure that can be caused by the underlying fault
- Therefore you should do some follow-up work to try to prove that :
  - *A defect can be more serious than it first appears*
  - *A defect is more general than it first appears*



# Follow-up testing

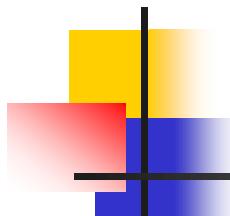
- Vulnerable program state
  - When you find a coding error, you have the program in a state that the programmer did not intend and probably did not expect
- Leverage vulnerability
  - Keep testing and you might find that the real impact of the underlying fault is a much worse failure, such as a system crash or corrupted data



# Types of follow-up testing

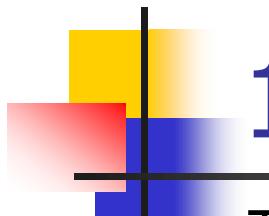
1. Vary the behavior (change the conditions by changing what the test case does)
2. Vary the options and settings of the program (change the conditions by changing something about the program under test)
3. Vary the software and hardware environment

Let's describe each one in more detail...



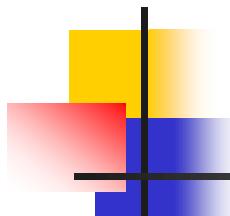
# 1. Vary the behavior

- Keep using the program after you see the problem (stay with it)
- Bring it to the failure state again and again (multiple times)
  - If the program fails when you do X, then do X many times
  - Is there a cumulative impact?
- Try things that are related to the task that failed
  - For example, assume that the program unexpectedly but slightly scrolls the display when you add two numbers
  - Then try doing X, see the scroll. Do Y then do X, see the scroll. Do Z, then do X, see the scroll, etc. (If the scrolling gets worse or better in one of these tests, follow that up, you're getting useful information for debugging)



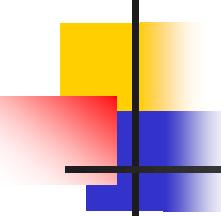
# 1. Vary your behavior--continued

- Try things that are related to the failure
  - If the failure is unexpected scrolling after adding then
    - Try scrolling first, then adding
    - Try repainting the screen, then adding
    - Try resizing the display of the numbers, then adding
- Vary the speed
  - Try entering the numbers more quickly or
  - Change the speed of your activity in some other way
- Also try other exploratory testing techniques
  - For example, you might try some interference tests
    - Stop the program or pause it just as the program is failing
    - Try it while the program is doing a background save
    - Does that cause data loss corruption along with this failure?



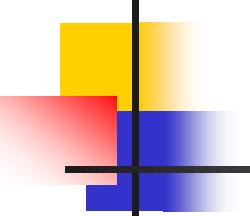
## 2. Vary options and settings

- Try to reproduce the bug when the program is in a different state:
  - Change the values of environment variables
  - Change how the program uses memory
  - Change anything that looks like it might be relevant that allows you to change as an option
- For example, suppose the program scrolls unexpectedly when you add two numbers
  - Maybe you can change the size of the program window, or the precision (or displayed number of digits) of the numbers



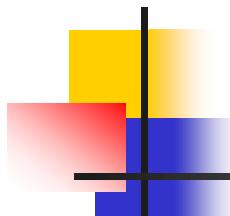
### 3. Vary the configuration

- A bug might show a more serious failure if you run the program with less memory, a higher resolution printer, more device interrupts coming in etc.
  - If there is anything involving timing, use a really slow (or very fast) computer, link, modem or printer, etc.
  - If there is a video problem, try other resolutions on the video card
  - Try displaying MUCH more (less) complex images



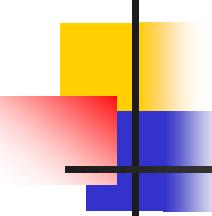
# Old bugs

- In many projects, an *old bug* (i.e. found in a previous release of the program) might not be taken very seriously, if there weren't lots of customer complaints
  - If you know it's an old bug, check its history
  - The bug will be taken more seriously if it is new
  - You can argue that it should be treated as new if you can find a new variation or a new symptom that didn't exist in the previous release
  - What you are showing is that the new version's code interacts with this error in new ways
  - *So, that's a new problem...*



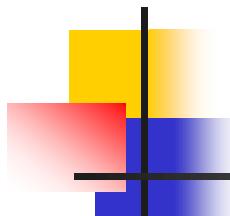
# Bugs credibility

- Bugs that don't fail on the programmer's machine are much less *credible* (to that programmer)
- Look for configuration dependence
- If they are configuration dependent, the report will be much more credible if it identifies the configuration dependence directly



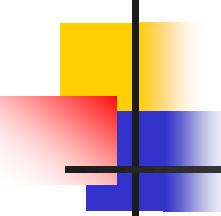
# Configuration dependence

- It is recommended to perform testing on two separate machines (standard in many companies)
  - Do your main testing on Machine 1
    - Maybe this is your powerhouse: latest processor, newest updates to the operating system, fancy printer, video card, USB devices, huge hard disk, lots of RAM, cable modem, etc.
  - When you find a defect, use Machine 1 as your bug reporting machine and replicate on Machine 2
    - Machine 2 is totally different. Different processor, different keyboard and keyboard driver, different video, barely enough RAM, slow, small hard drive, dial-up connection with a link that makes turtles look fast
- Some people do their main testing on the turtle and use the power machine for replication



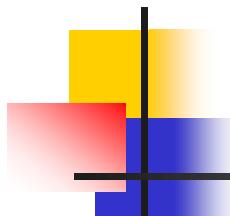
# Configuration dependence-continued

- Write the steps, one by one, on the bug form at Machine 1
- As you write them, try them on Machine 2
  - If you get the same failure, you've checked your bug report while you wrote it (a good thing to do)
  - If you don't get the same failure, you have a configuration dependent bug. Time to do troubleshooting



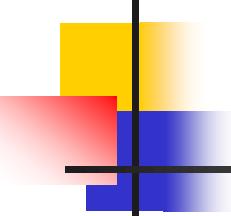
# Extreme vs. mainstream values

- Test at extreme values (most likely to show a defect)
  - If they yield failure, then do some troubleshooting around the extremes
  - Is the bug tied to a single setting or is there a small range of cases?
  - In your report, identify the narrow range that yields failures
    - The range might be so narrow that the bug gets deferred
    - That might be the right decision. Your reports help the company choose the right bugs to fix before a release, and size the risks associated with the remaining ones
- Try mainstream values
  - These are easy settings that should pose no problem to the program
  - Do you replicate the bug? If yes, write it up, referring primarily to these mainstream settings
  - This will be a very credible bug report



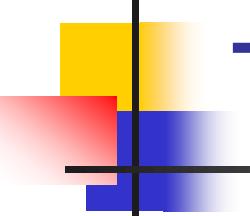
# Overcoming resistance

- Things that will make programmers resist spending their time on fixing the bug you have reported:
  - The programmer can't replicate the defect
  - Strange and complex set of steps required to induce the failure
  - Not enough information to know what steps are required, and it will take a lot of work to figure them out
  - The programmer doesn't understand the report
  - Unrealistic (e.g. "corner case")
  - It's a feature



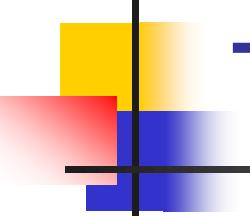
# Non-reproducible errors

- Always report non-reproducible errors
  - When you realize that you can't reproduce the bug, write down everything you can remember (do it now, before you forget even more)
  - As you write, ask yourself whether you're sure that you did this step (or saw this thing) exactly as you are describing it. If not, say so. Draw these distinctions right away. The longer you wait, the more you'll forget
  - Maybe the failure was a delayed reaction to something you did before starting this test or series of tests. Before you forget, note the tasks you did before running this test
  - Check the bug tracking system. Are there similar failures? Maybe you can find a pattern
  - If you report them well, programmers can often figure out the underlying problem
- You must describe the failure as precisely as possible
  - If you can identify a display or a message well enough, the programmer can often identify a specific point in the code that the failure had to pass through



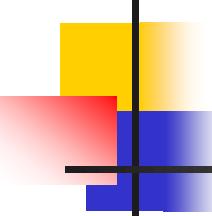
# The problem report form<sup>1</sup>

- A typical form includes many of the following fields
  - **Problem report number:** must be unique
  - **Reported by:** original reporter's name. Some forms add an editor's name
  - **Date reported:** date of initial report
  - **Program (or component) name:** the visible item under test
  - **Release number:** like Release 2.0
  - **Version (build) identifier:** like version C or version 20000802a



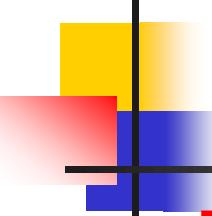
# The problem report form<sup>2</sup>

- **Configuration(s)**: h/w and s/w configurations under which the bug was found and replicated
- **Report type**: e.g. coding error, design issue, documentation mismatch, suggestion, query
- **Can reproduce**: yes / no / sometimes / unknown  
(Unknown can arise, for example, when the configuration is at a customer site and not available to the lab)
- **Severity**: assigned by tester. Some variation on small / medium / large



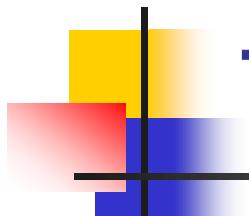
# The problem report form<sup>3</sup>

- **Priority:** assigned by programmer/project manager
- **Problem summary:** 1-line summary of the problem
- **Key words:** use these for searching later, anyone can add to key words at any time
- **Problem description and how to reproduce it:** concise step by step reproduction description
- **Suggested fix:** leave it blank unless you have something useful to say
- **Status:** Tester fills this in: Open / closed / resolved



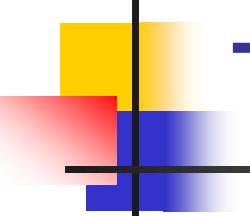
# The problem report form<sup>4</sup>

- **Resolution:** The project manager owns this field. Common resolutions include:
  - **Pending:** the bug is still being worked on
  - **Fixed:** the programmer says it's fixed. Now you should check it
  - **Cannot reproduce:** The programmer can't make the failure happen. You must add details, reset the resolution to Pending, and notify the programmer
  - **Deferred:** It's a bug, but we'll fix it later
  - **As Designed:** The program works as it's supposed to
  - **Need Info:** The programmer needs more info from you. He/she has probably asked a question in the comments
  - **Duplicate:** This is just a repeat of another bug report (XREF it on this report). Duplicates should not close until the duplicated bug closes
  - **Withdrawn:** The tester withdrew the report



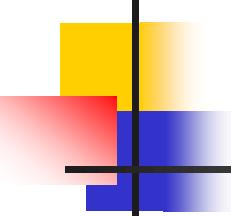
# The problem report form<sup>5</sup>

- **Resolution version:** build identifier
- **Resolved by:** programmer, project manager, tester (if withdrawn by tester), etc.
- **Resolution tested by:** originating tester, or a tester if originator was a non-tester
- **Change history:** date-stamped list of all changes to the record, including name and fields changed



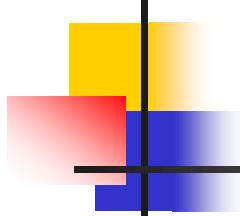
# The problem report form<sup>6</sup>

- **Comments:** free-form, arbitrarily long field, typically accepts comments from anyone on the project. Testers, programmers, tech support (in some companies) and others have an ongoing discussion of reproduction conditions, etc., until the bug is resolved. Closing comments (why a deferral is OK, or how it was fixed for example) go here
  - This field is especially valuable for recording progress and issues with difficult or politically charged bugs
  - Write carefully. Just like e-mail and web postings, it's easy to read a joke or a remark as a flame. Never flame.



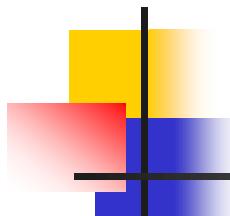
# Problem summary

- This one-line description of the problem is the most important part of the report
  - The project manager will use it in when reviewing the list of bugs that haven't been fixed
  - Executives will read it when reviewing the list of bugs that won't be fixed. They might only spend additional time on bugs with "interesting" summaries
- The ideal summary gives the reader enough information to help him/her decide whether to ask for more information. It should include:
  - A brief description that is specific enough that the reader can visualize the failure
  - A brief indication of the limits or dependencies of the bug (how narrow or broad are the circumstances involved in this bug)?
  - Some other indication of the severity (not a rating but helping the reader envision the consequences of the bug)



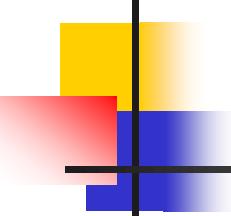
# Can you reproduce the bug?

- You may not see this on your form, but you should always provide this information
  - Never say it's reproducible unless you have recreated the bug (Always try to recreate the bug before writing the report)
  - If you've tried and tried but you can't recreate the bug, say "No". Then explain what steps you tried in your attempt to recreate it
  - If the bug appears sporadically and you don't yet know why, say "Sometimes" and explain
  - You may not be able to try to replicate some bugs. Example: customer-reported bugs where the setup is too hard to recreate



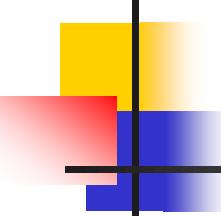
# Explain how to reproduce the bug

- First, describe the problem
- Don't rely on the summary to do this - some reports will print this field without the summary
- Go through the steps that you use to recreate this bug
  - Start from a known place (e.g. boot the program)
  - Then describe each step until you hit the bug
  - Number the steps. Take it one step at a time
  - If anything interesting happens on the way, describe it
  - Use landmarks/beacons
    - You are giving people directions to locate a bug
    - Especially in long reports, people need landmarks



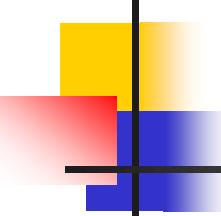
# How to reproduce the bug

- Describe the erroneous behavior and, if necessary, explain what should have happened (i.e. why is this a bug? Be clear)
- List the environmental variables (e.g. configuration) that are not covered elsewhere in the bug tracking form
- If you expect the reader to have any trouble reproducing the bug (special circumstances are required), be clear about them
- It is essential to keep the description focused
- The first part of the description should be the shortest step-by-step statement of how to get to the problem
- Add “Notes” after the description such as:
  - Comment that the bug won’t show up if you do step X between step Y and step Z
  - Comment explaining your reasoning for running this test
  - Comment explaining why you think this is an interesting bug
  - Comment describing other variants of the bug



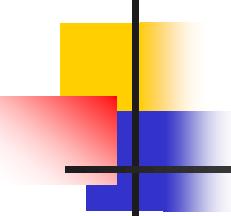
# Keeping the report simple

- If you see two failures, write two reports
- Combining failures creates problems:
  - The summary description is typically vague. You say words like "fails" or "doesn't work" instead of describing the failure more vividly. This weakens the impact of the summary
  - The detailed report is typically lengthened and contains complex logic like: "Do this unless that happens in which case don't do this unless the first thing, and then the test case of the second part and sometimes you see this but if not then that..."
- Even if the detailed report is rationally organized, it is longer (there are two failures and two sets of conditions, even if they are related) and therefore more intimidating
- You'll often see one bug get fixed but not the other
- When you report related problems on separate reports, it is a courtesy to cross-reference them



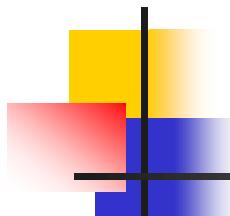
# Eliminate unnecessary steps<sup>1</sup>

- Sometimes it's not immediately obvious what steps can be dropped from a long sequence of steps in a bug
  - *Look for critical steps -- Sometimes the first symptoms of a failure are subtle*
- You have a list of the steps you took to show the error. You're now trying to shorten the list. Look carefully for any hint of a failure as you take each step
- A few things to look for:
  - Error messages (you got a message 10 minutes ago. The program didn't fully recover from the error, and the problem you see now is caused by that poor recovery)
  - Delays or unexpectedly fast responses
  - More on the next slide...



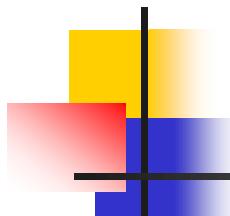
# Eliminate unnecessary steps<sup>2</sup>

- Display oddities, such as a flash, a repainted screen, a cursor that jumps back and forth, multiple cursors, misaligned text, slightly distorted graphics, etc.
- Sometimes the first indicator that the system is working differently is that it sounds a little different than normal
- An in-use light or other indicator that a device is in use when nothing is being sent to it (or a light that is off when it shouldn't be)
- Debug messages—turn on the debug monitor on your system (if you have one) and see if/when a message is sent to it
- If you've found what looks like a critical step, try to eliminate almost everything else from the bug report. Go directly from that step to the last one (or few) that shows the bug
- If this doesn't work, try taking out individual steps or small groups of steps



# Unrealistic cases

- Some reports are inevitably dismissed as unrealistic (having no importance in real use)
  - If you're dealing with an extreme value, do follow-up testing with less extreme values
  - Check with people who might know the customer impact of the bug:
    - -- Technical marketing
    - -- Human factors
    - -- Network administrators
    - -- In-house power users
    - -- Technical support
    - -- Documentation
    - -- Training
    - -- Maybe sales

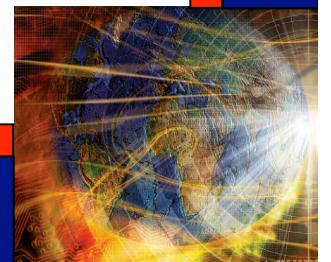


# Editing bug reports

- Some groups have a second tester (usually a senior tester) review reported defects before they go to the programmer
- The second tester:
  - checks that critical information is present and intelligible
  - checks whether she/he can reproduce the bug
  - asks whether the report might be simplified, generalized or strengthened
- If there are problems, she/he takes the bug back to the original reporter

# Chapter 1

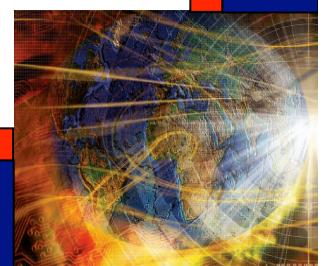
## Introduction



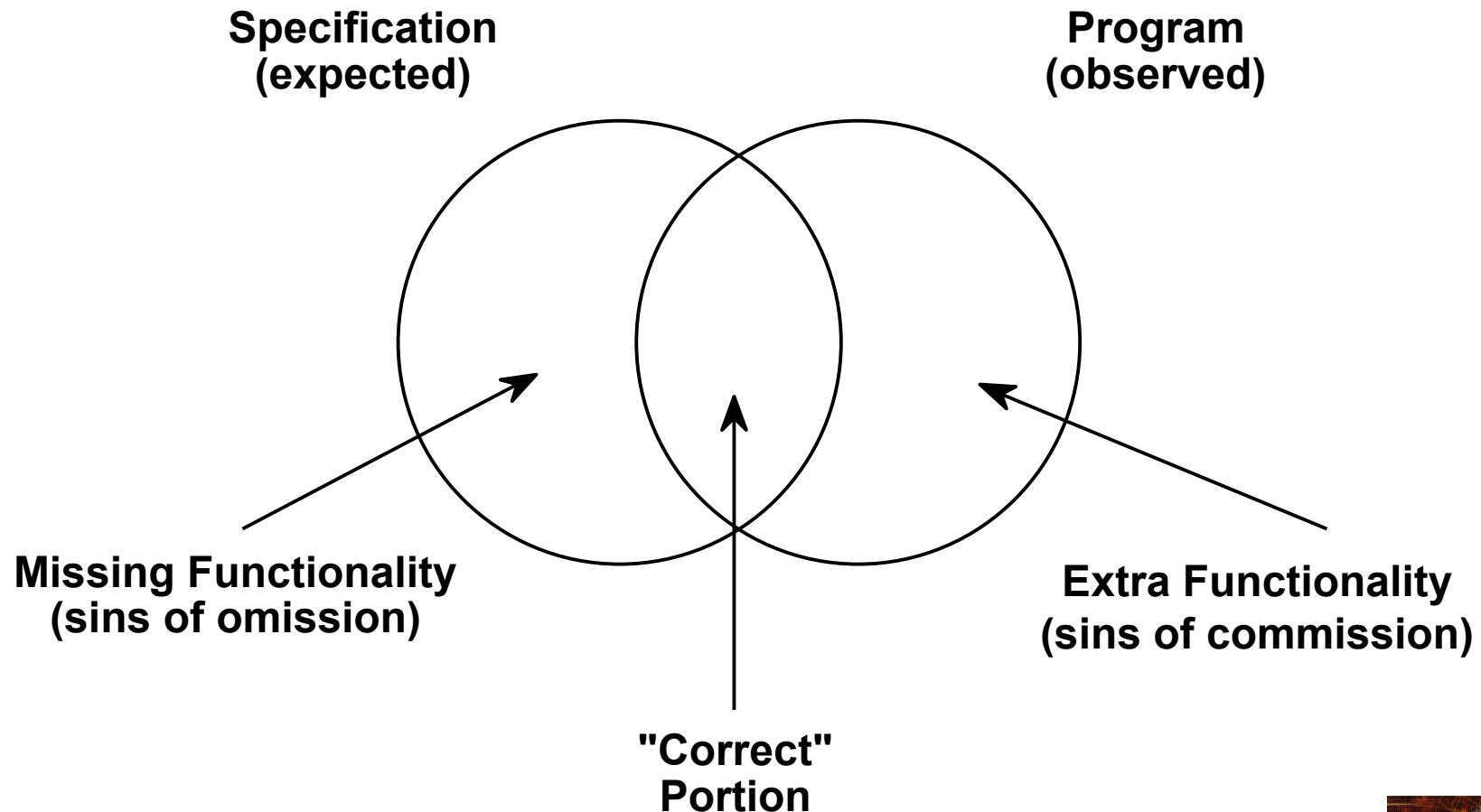
# Testing

" in the beginning of a malady it is easy to cure but difficult to detect, but in the course of time, not having been either detected or treated in the beginning, it becomes easy to detect but difficult to cure."

Nicolo Machiavelli  
*The Prince, 1513*

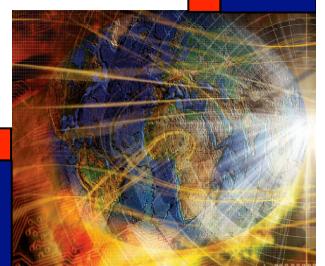


# Program Behaviors

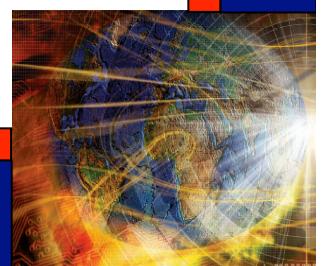
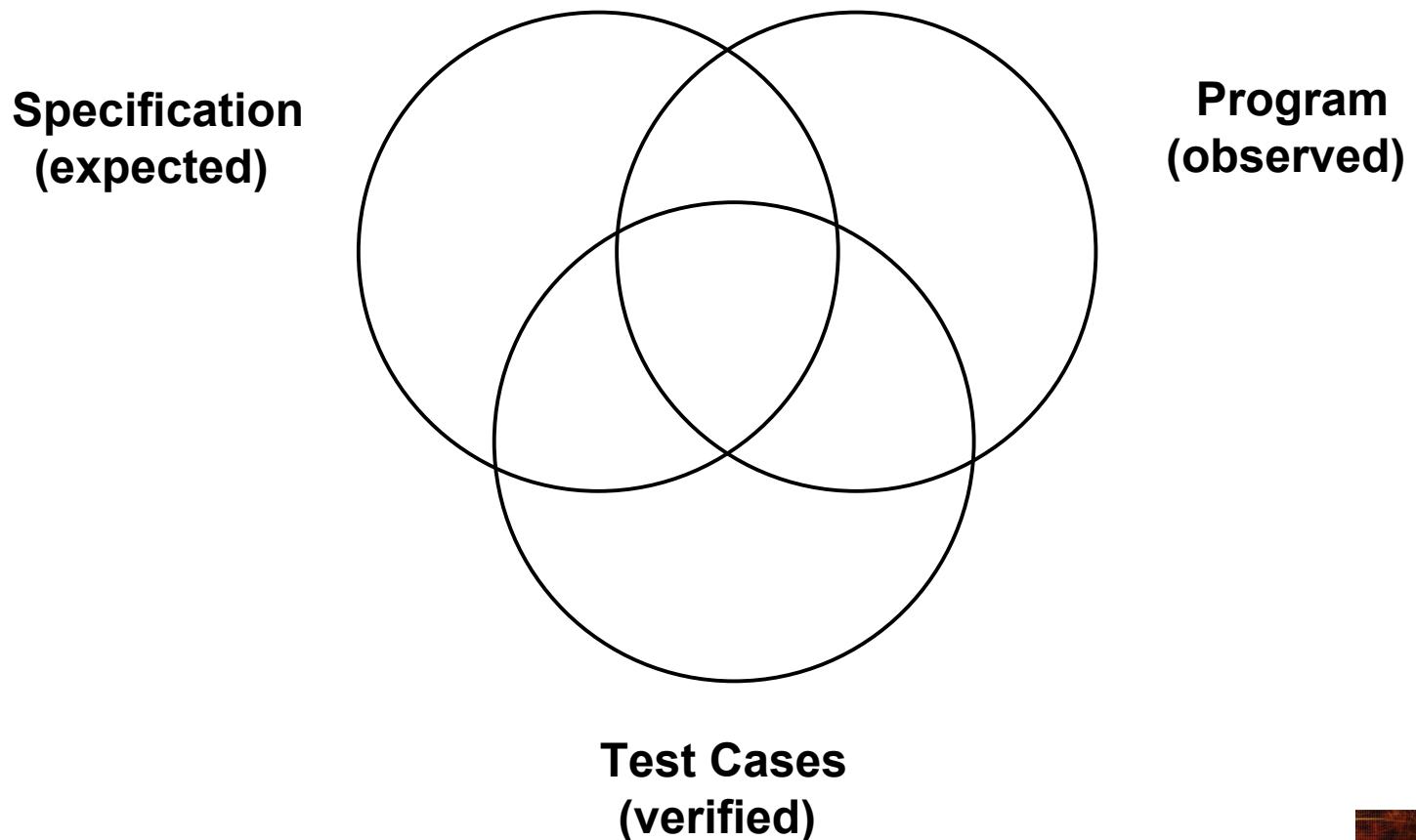


# Correctness

- **Impossible to demonstrate**
- **A term from “classical” computer science**
  - “proofs” derived from code
  - Not derived from specification
  - Can only prove that the code does what it does!
- **Better viewpoint: a relative term—program P is correct with respect to specification S.**
- **Bottom Line: do the specification and the program meet the customer/user's expectations?**

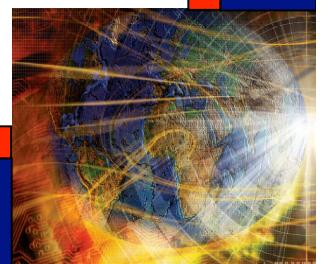
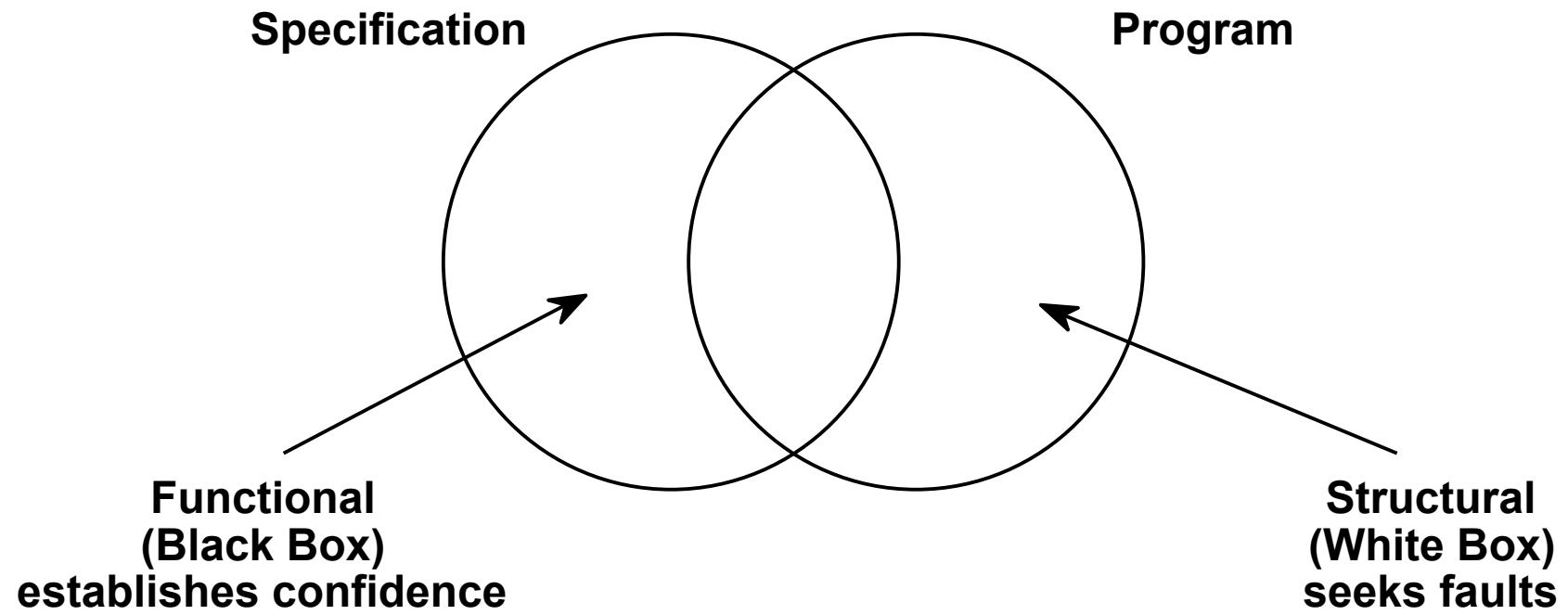


# Testing Program Behavior



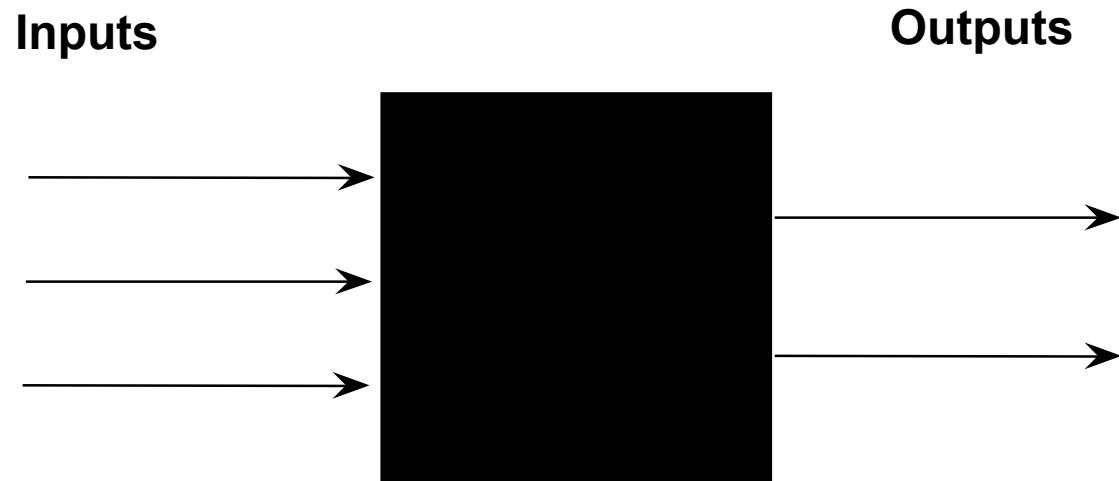
# Basic Approaches

Preferred terms: Specification-based and code-based testing

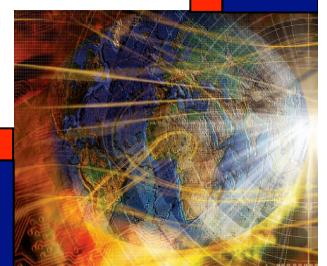


# Black Box

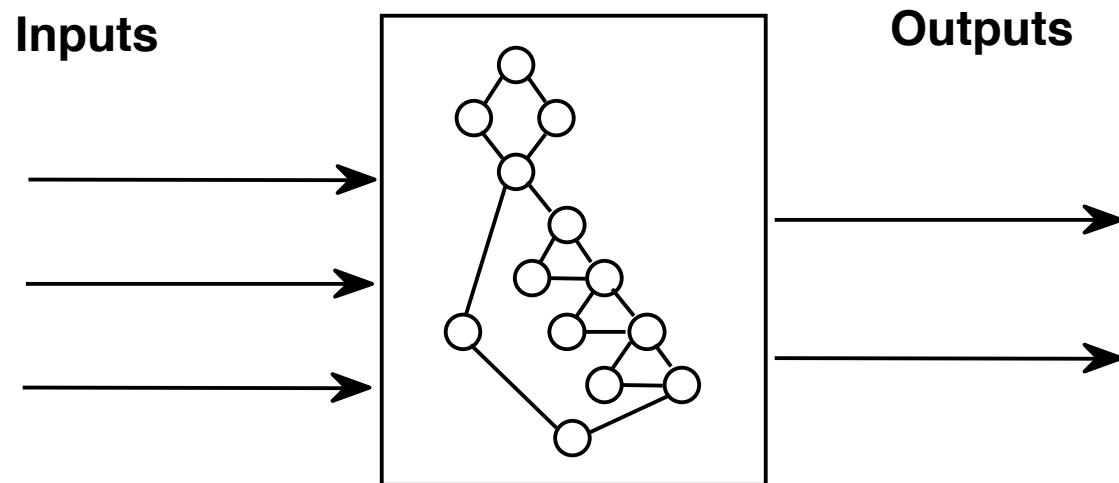
## A term borrowed from engineering



**Function is understood only in terms of its inputs and outputs, with no knowledge of its implementation.**



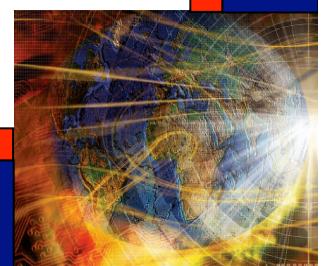
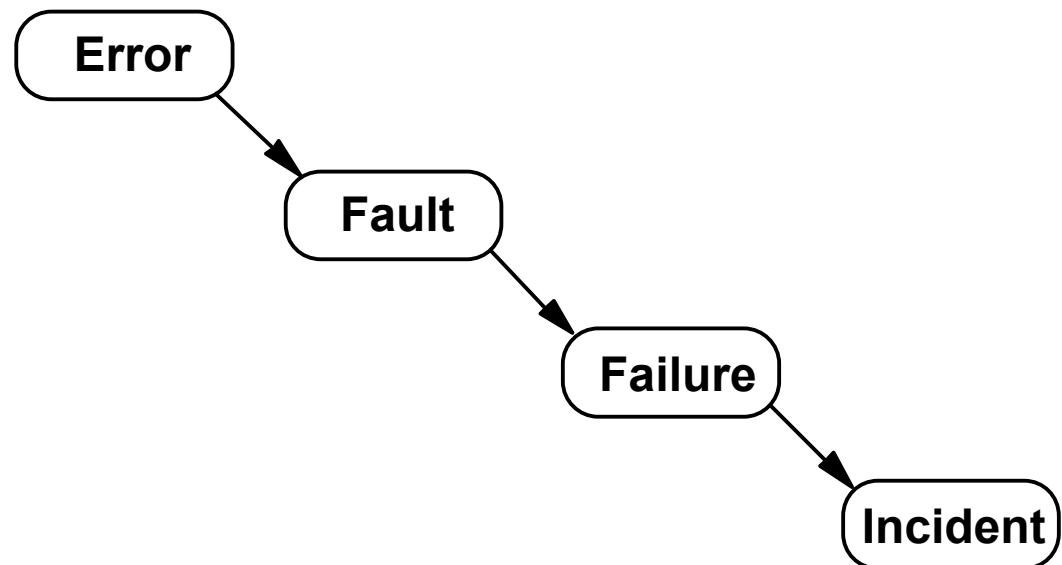
# White (Clear) Box



**Function is understood only in terms of its implementation.**



# IEEE Testing Terminology



# What is a software test?

- An experiment,
- Designed to reveal the presence of a fault,
- By causing a failure when executed by the code being tested.
- *n.b.* a test can never reveal the absence of a fault.
- A test method is a repeatable way to generate test cases.



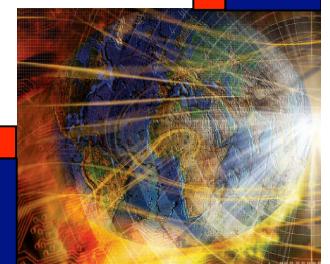
# Content of a Test Case

- "Boilerplate": author, date, purpose, test case ID
- Pre-conditions (including environment)
- Inputs
- Expected Outputs
- Observed Outputs
- Pass/Fail
  - Common usage: Pass when expected and observed outputs agree
  - BUT Myers [*The Art of Software Testing*] defines a successful test as one that reveals a fault, i.e., expected and observed outputs disagree.



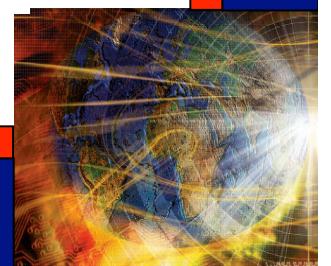
# How Might a Test Case Fail?

- **False positive?**
- **False negative?**
- **What do these really mean?**
- **Exercise**
  - Analyze false positive, false negative
  - Suggestion: use a decision table with conditions:
    - Expected output correct?
    - Observed output correct? (potential circularity here)
- **Compare with common medical usage**



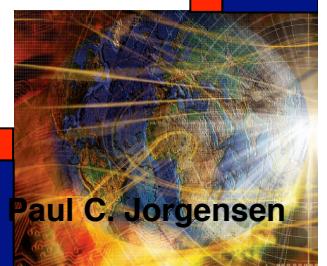
# Chapter 2

## Examples



# Examples

- **Triangle problem**
- **NextDate**
- **Commission problem**
- **Simple ATM system**
- **Currency converter**
- **Windshield Wiper controller**



# Triangle Problem

**Simple version:** The triangle program accepts three integers,  $a$ ,  $b$ , and  $c$ , as input. These are taken to be sides of a triangle. The output of the program is the type of triangle determined by the three sides: Equilateral, Isosceles, Scalene, or Not A Triangle.

**Improved version:** “Simple version” plus better definition of inputs:

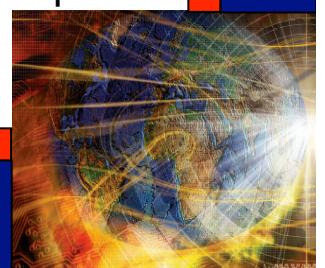
The integers  $a$ ,  $b$ , and  $c$  must satisfy the following conditions:

- |                         |                 |
|-------------------------|-----------------|
| c1. $1 \leq a \leq 200$ | c4. $a < b + c$ |
| c2. $1 \leq b \leq 200$ | c5. $b < a + c$ |
| c3. $1 \leq c \leq 200$ | c6. $c < a + b$ |

**Final Version:** “Improved version” plus better definition of outputs:

If an input value fails any of conditions c1, c2, or c3, the program notes this with an output message, for example, “Value of  $b$  is not in the range of permitted values.” If values of  $a$ ,  $b$ , and  $c$  satisfy conditions c1, c2, and c3, one of four mutually exclusive outputs is given:

1.      If all three sides are equal, the program output is Equilateral.
2.      If exactly one pair of sides is equal, the program output is Isosceles.
3.      If no pair of sides is equal, the program output is Scalene.
4.      If any of conditions c4, c5, and c6 is not met, the program output is NotATriangle.



# Triangle Problem Discussion

**Final Version:** “Improved version” plus better definition of outputs:

If an input value fails any of conditions c1, c2, or c3, the program notes this with an output message, for example, “Value of b is not in the range of permitted values.” If values of a, b, and c satisfy conditions c1, c2, and c3, one of four mutually exclusive outputs is given:

1. If all three sides are equal, the program output is Equilateral.
2. If exactly one pair of sides is equal, the program output is Isosceles.
3. If no pair of sides is equal, the program output is Scalene.
4. If any of conditions c4, c5, and c6 is not met, the program output is NotATriangle.

**Problems persist!**

**What output is expected for the input set (2, 2, 5)?**

- Isosceles because  $a = b$ ?
- NotATriangle because  $c > a+b$ ?



# Triangle Problem Exercise

**Fix the “Final Version” (sometimes testers must also be specifiers!)**

**The Really Final Version: “Improved version” plus better definition of outputs.**

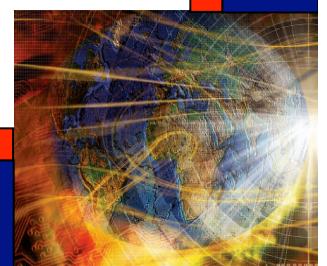


# NextDate

**NextDate** is a function of three variables: month, date, and year. It returns the date of the day after the input date. The month, date, and year variables have integer values subject to these conditions:

- c1.  $1 \leq \text{month} \leq 12$
- c2.  $1 \leq \text{day} \leq 31$
- c3.  $1812 \leq \text{year} \leq 2012$

If any of conditions c1, c2, or c3 fails, NextDate produces an output indicating the corresponding variable has an out-of-range value — for example, “Value of month not in the range 1..12”. Because numerous invalid day–month–year combinations exist, NextDate collapses these into one message: “Invalid Input Date.”



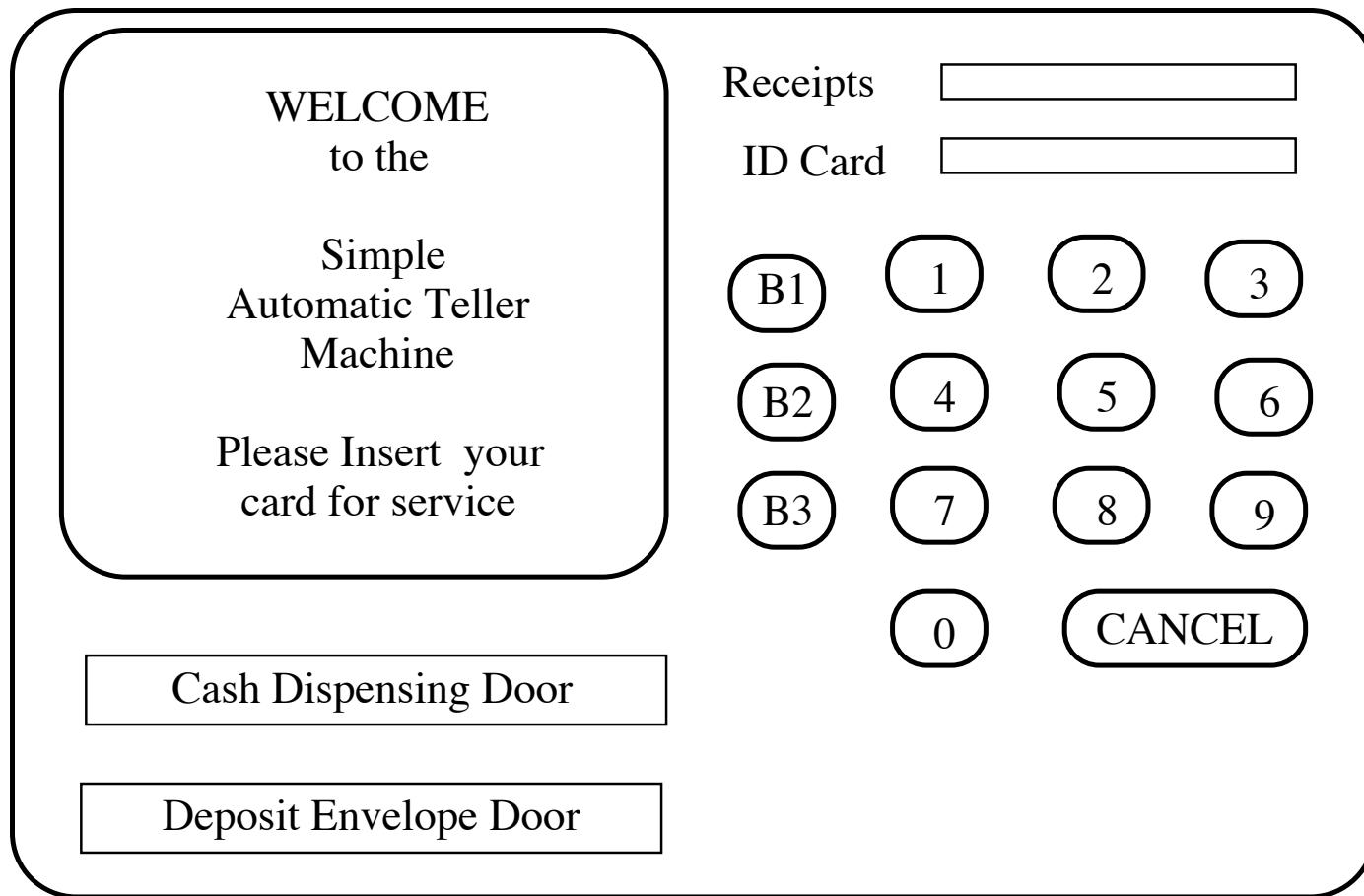
# The Commission Problem

A rifle salesperson in the former Arizona Territory sold rifle locks, stocks, and barrels made by a gunsmith in Missouri. Locks cost \$45, stocks cost \$30, and barrels cost \$25. The salesperson had to sell at least one complete rifle per month, and production limits were such that the most the salesperson could sell in a month was 70 locks, 80 stocks, and 90 barrels. After each town visit, the salesperson sent a telegram to the Missouri gunsmith with the number of locks, stocks, and barrels sold in that town. At the end of a month, the salesperson sent a very short telegram showing –1 locks sold. The gunsmith then knew the sales for the month were complete and computed the salesperson's commission as follows: 10% on sales up to (and including) \$1000, 15% on the next \$800, and 20% on any sales in excess of \$1800. The commission program produced a monthly sales report that gave the total number of locks, stocks, and barrels sold, the salesperson's total dollar sales, and, finally, the commission.



# The Simple ATM System

(see description in text)



# The Currency Converter

(see description in text)

## Currency Converter

U. S. Dollar amount

Equivalent in ...

- Brazil
- Canada
- European Community
- Japan

Compute

Clear

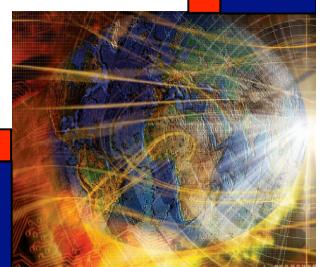
Quit



# The Saturn Windshield Wiper Controller

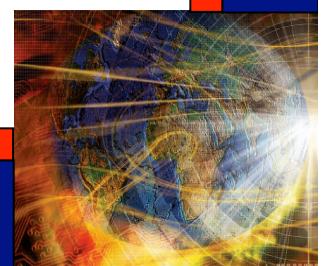
The windshield wiper on some Saturn automobiles is controlled by a lever with a dial. The lever has four positions, OFF, INT (for intermittent), LOW, and HIGH, and the dial has three positions, numbered simply 1, 2, and 3. The dial positions indicate three intermittent speeds, and the dial position is relevant only when the lever is at the INT position. The decision table below shows the windshield wiper speeds (in wipes per minute) for the lever and dial positions.

c1. Lever	OFF	INT	INT	INT	LOW	HIGH
c2. Dial	n/a	1	2	3	n/a	n/a
a1. Wiper speed is...	0	4	6	12	30	60



# Mainline Functional Testing Techniques

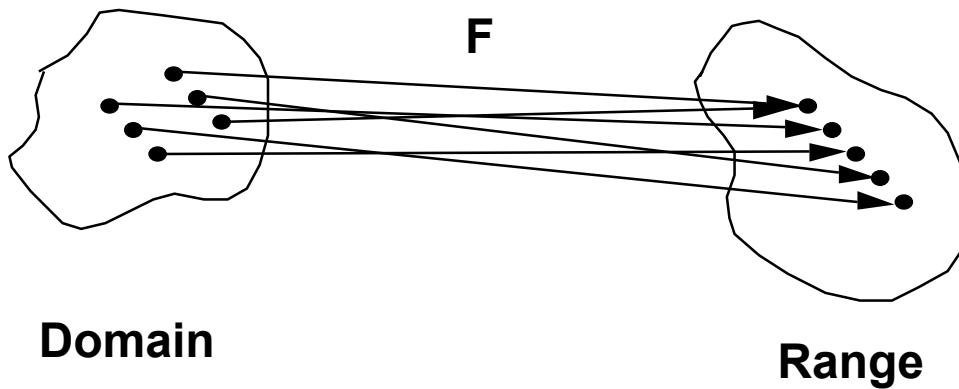
- **Boundary Value Testing (4 flavors)**
- **Equivalence Partitions (another 4 flavors)**
- **Special Value Testing**
- **Output Domain (Range) Checking**
- **Decision Table Based Testing (aka Cause and Effect Graphs)**



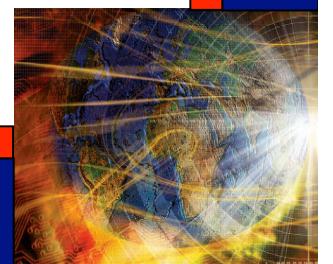
# Functional Testing

Ultimately, any program can be viewed as a mapping from its Input Domain to its Output Range:

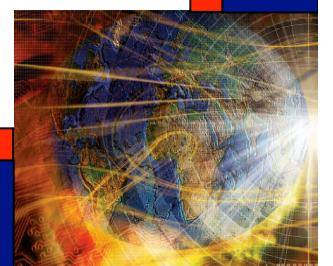
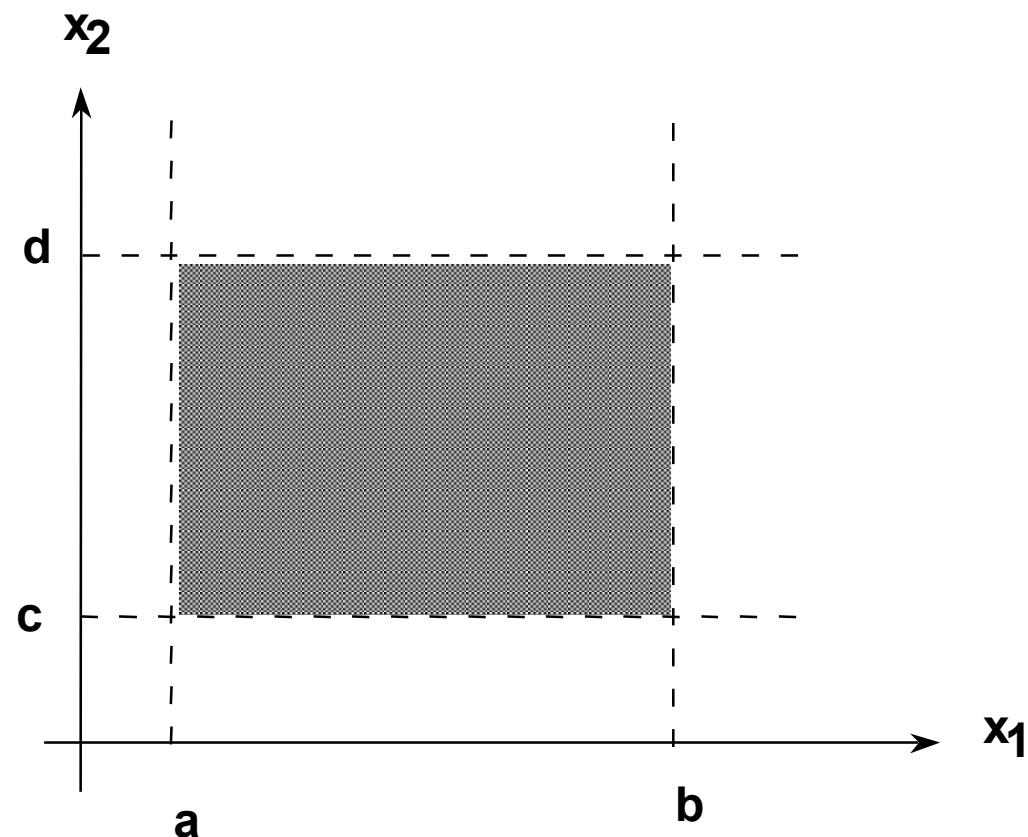
$$\text{Output} = F(\text{Input})$$



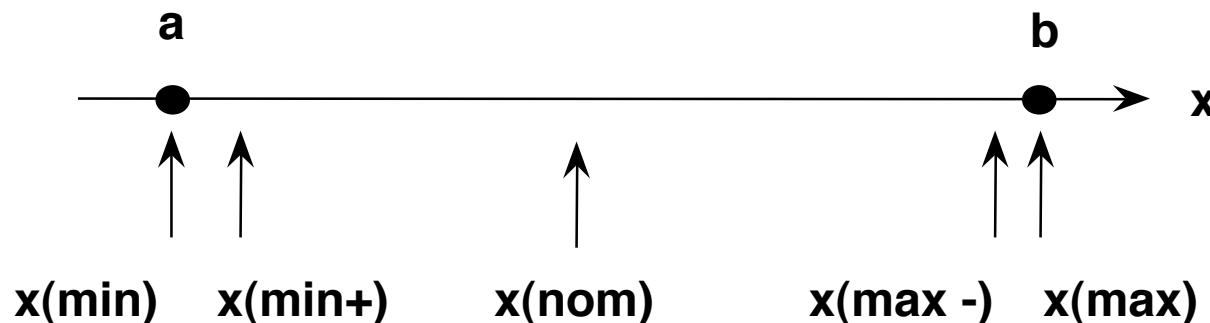
Functional testing uses information about functional mappings to identify test cases.



# Input Domain of $F(x_1, x_2)$

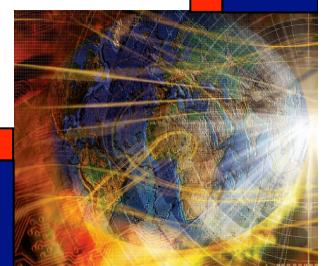


# Input Boundary Value Testing

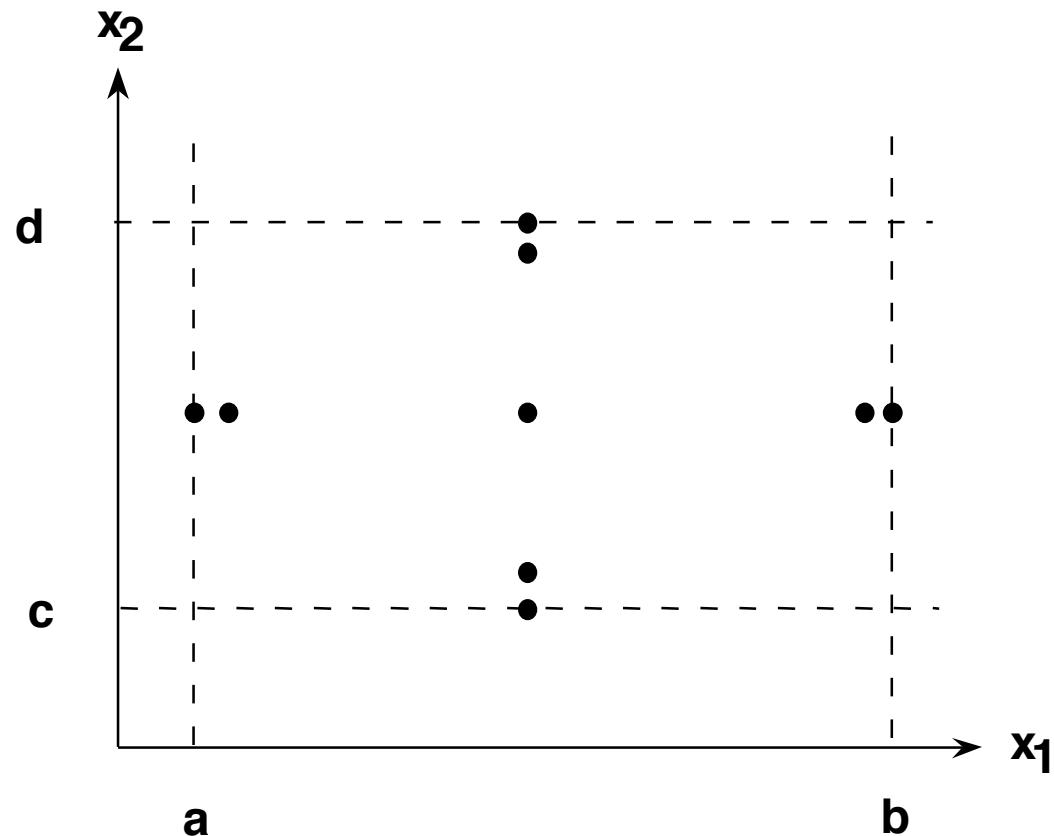


**test cases for a variable  $x$ , where  $a \leq x \leq b$**

**Experience shows that errors occur more frequently for extreme values of a variable.**



# Input Boundary Value Test Cases (2 Variables)



test cases for variables  $x_1$  and  $x_2$ , where  $a \leq x_1 \leq b$  and  $c \leq x_2 \leq d$

As in reliability theory, two variables rarely both assume their extreme values.



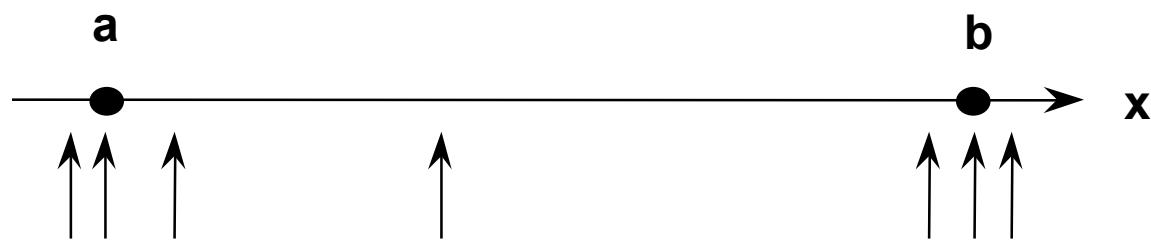
# Method

- Hold all variables at their nominal value.
- Let one variable assume its boundary values.
- Repeat this for each variable.
- This will (hopefully) reveal all faults that can be attributed to a single variable.

**Exercise: why might this not work?**

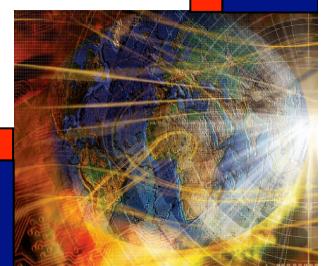


# Robustness Testing

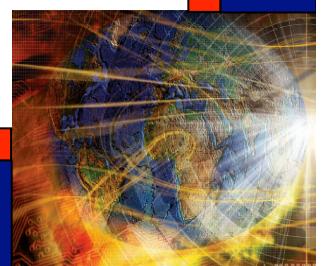
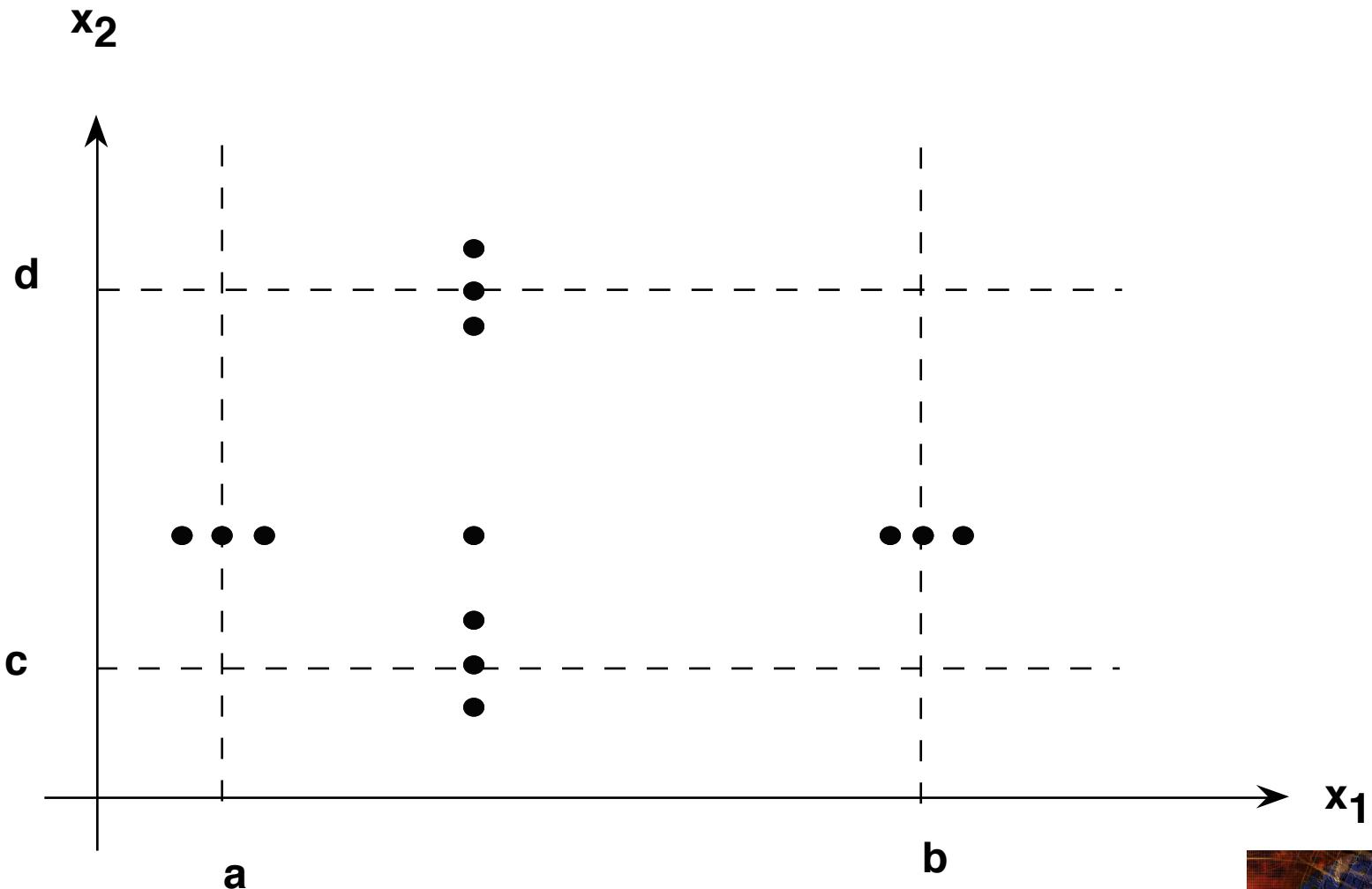


**test cases for a variable x, where  $a \leq x \leq b$**

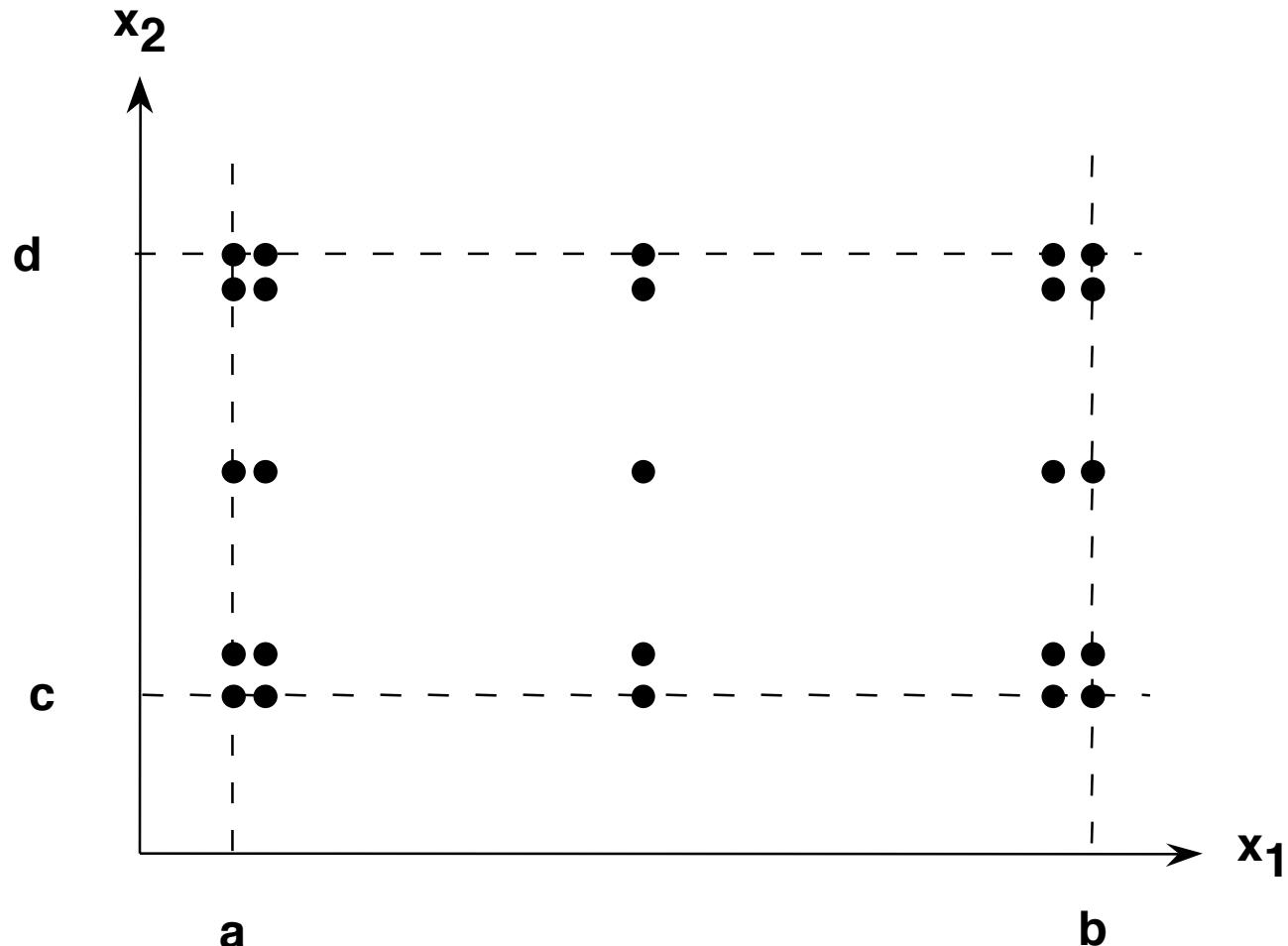
1. stress input boundaries
2. acceptable response for invalid inputs?
3. leads to exploratory testing (test hackers)
4. can discover hidden functionality



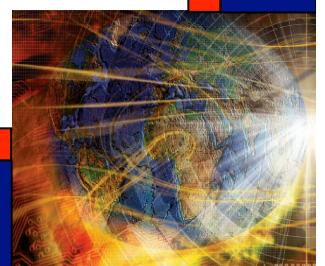
## Robustness Testing (2 Variables)



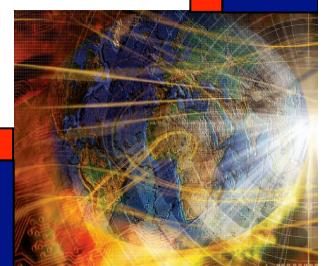
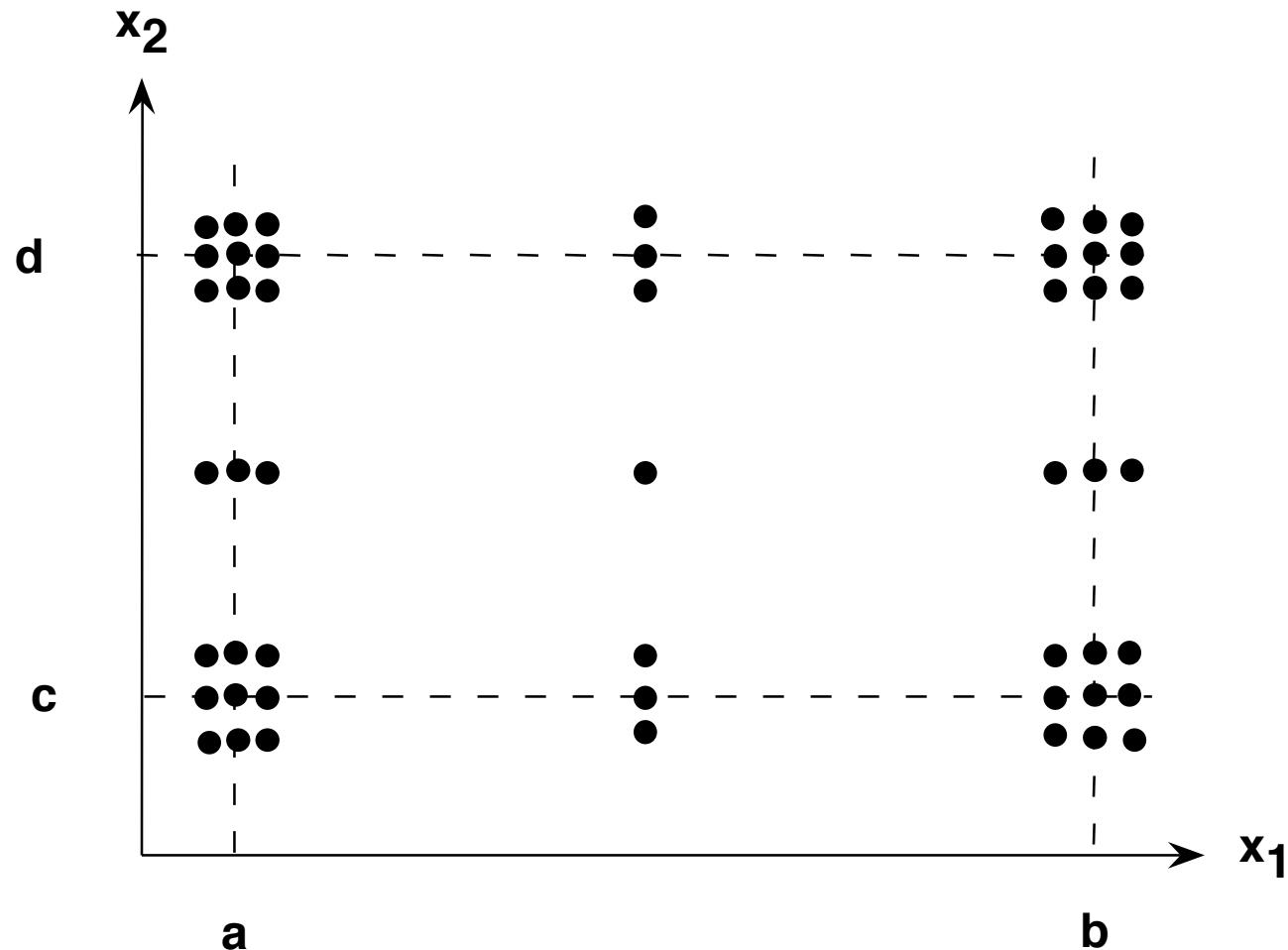
## Worst Case Testing (2 Variables)



Eliminate the "single fault" assumption.  
Murphy's Law?

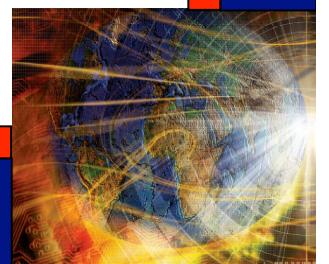


# Robust Worst Case Testing (2 Variables)



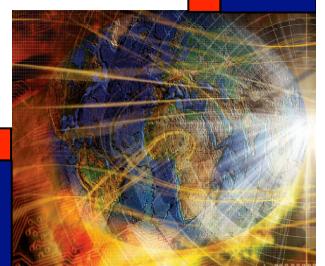
# Special Value Testing

1. complex mathematical (or algorithmic) calculations
2. worst case situations ( similar to robustness)
3. problematic situations from past experience
4. "second guess" likely implementations
5. experience helps
6. frequently done by customer/user
7. defies measurement
8. highly intuitive
9. seldom repeatable
10. (and is often most effective)

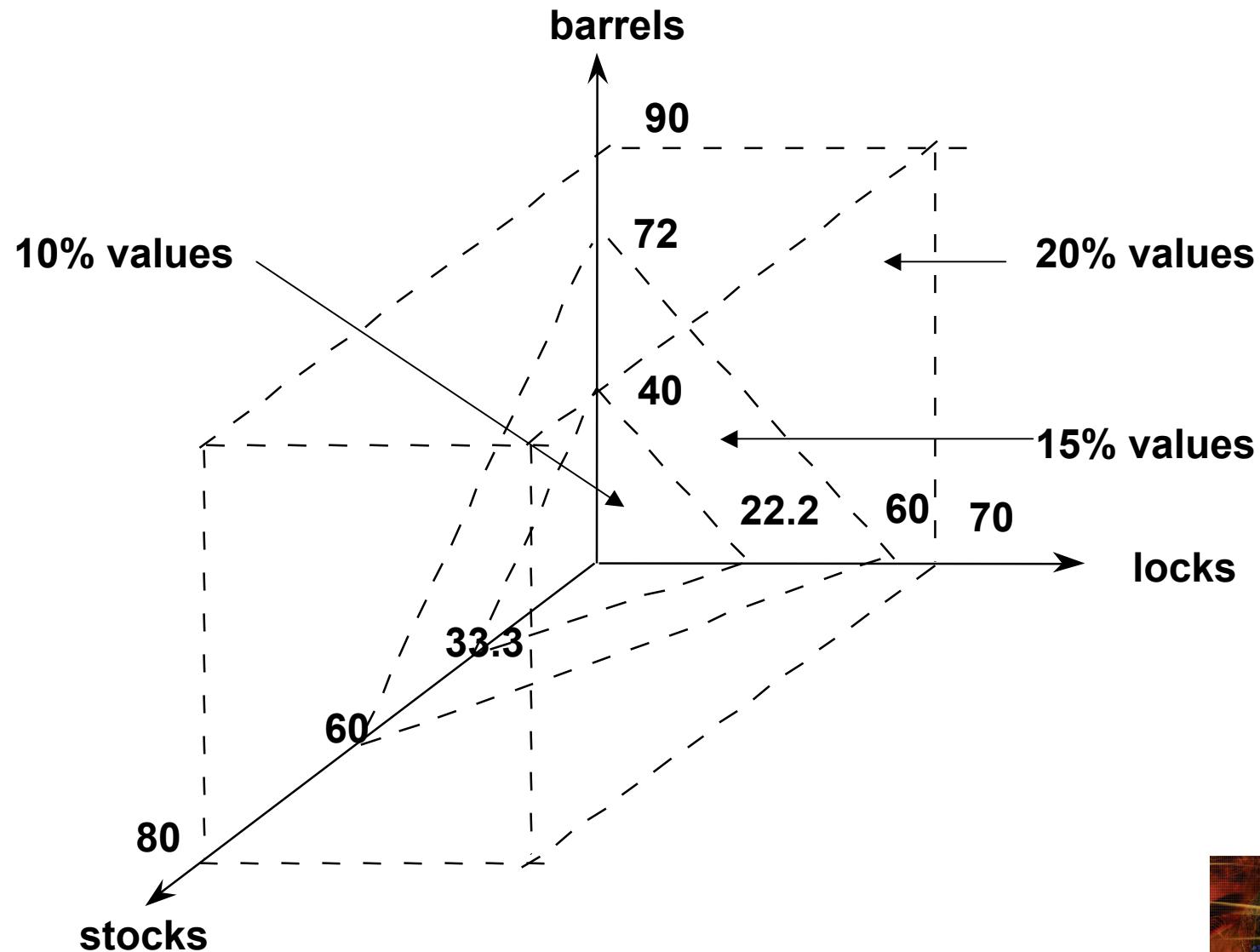


# Output Range Coverage

1. Work "backwards" from expected outputs (assume that inputs cause outputs).
2. Mirror image of equivalence partitioning (good cross check)
3. Helps identify ambiguous causes of outputs.



# Input Domain for Lock, Stock, and Barrel



# NextDate Function

**NEXTDATE** is a function of three variables: month, day, and year, for years from 1812 to 2012. It returns the date of the next day.

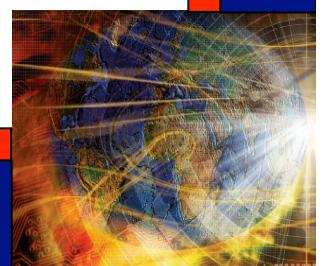
**NEXTDATE( Dec, 31, 1991)** returns Jan 1 1992

**NEXTDATE( Feb, 21, 1991)** returns Feb 22 1991

**NEXTDATE( Feb, 28, 1991)** returns Mar 1 1991

**NEXTDATE( Feb, 28, 1992)** returns Feb 29 1992

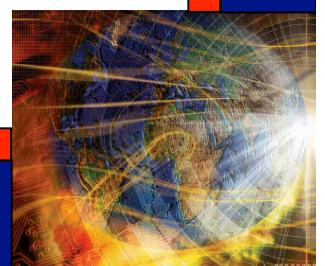
**Leap Year:** Years divisible by 4 except for century years not divisible by 400. Leap Years include 1992, 1996, 2000.  
1900 was not be a leap year.



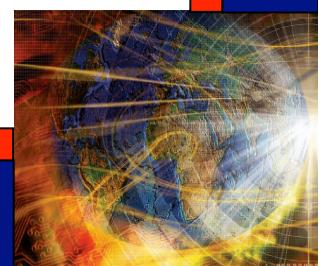
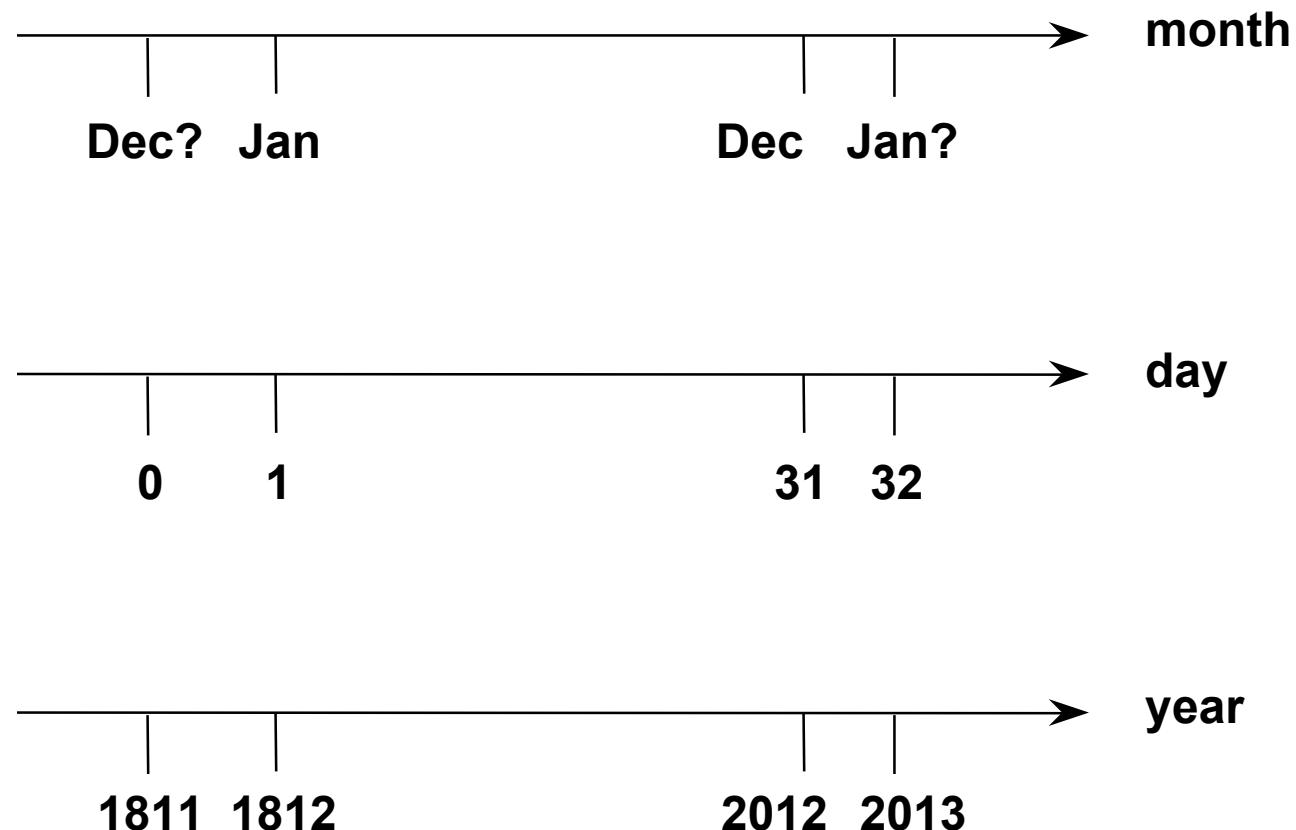
# Input Domain Test Cases

## Input Values

Test Case	Month	Day	Year	
ID-1	Jan	15	1912	Select test cases so that one variable assumes nominal and extreme values while others are held at nominal values.
ID-2	Feb	15	1912	
ID-3	Jun	15	1912	
ID-4	Nov	15	1912	
ID-5	Dec	15	1912	
ID-6	Jun	1	1912	Notice that Input Domain testing presumes that the variables in the input domain are independent; logical dependencies are unrecognized.
ID-7	Jun	2	1912	
ID-3	Jun	15	1912	
ID-8	Jun	30	1912	This typically results in "anomalies" like test case ID-9, which is logically impossible.
ID-9	Jun	31	1912	
ID-10	Jun	15	1812	
ID-11	Jun	15	1913	
ID-3	Jun	15	1912	
ID-12	Jun	15	2011	
ID-13	Jun	15	2012	



# Robust Input Domain Test Cases



# Robust Input Domain Test Cases

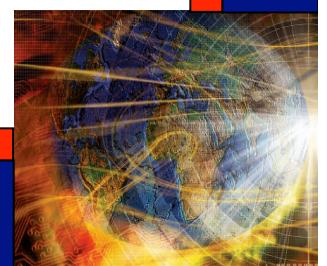
## Input Values

Robust Test Case	Test Case	Month	Day	Year	
RD-1	ID-5	Dec	15	1912	
RD-2	ID-1	Jan	15	1912	As with input domain testing,
RD-3	ID-2	Feb	15	1912	Robustness Testing presumes
RD-4	ID-3	Jun	15	1912	independent variables. The major
RD-5	ID-4	Nov	15	1912	difference is that robustness
RD-6	ID-5	Dec	15	1912	Testing also presumes that
RD-7	ID-1	Jan	15	1912	variables represent continuous
RD-8		Jun	0	1912	functions. Notice that the
RD-9	ID-6	Jun	1	1912	Robustness Test Cases for Month
RD-10	ID-7	Jun	2	1912	and Day sometimes don't make
RD-11	ID-3	Jun	15	1912	sense, but those for Year do.
RD-12	ID-8	Jun	30	1912	
RD-13	ID-9	Jun	31	1912	
RD-14		Jun	32	1912	
RD-15		Jun	15	1811	
RD-16	ID-10	Jun	15	1812	
RD-17	ID-11	Jun	15	1913	
RD-18	ID-3	Jun	15	1912	
RD-19	ID-12	Jun	15	2011	
RD-20	ID-13	Jun	15	2012	
RD-21		Jun	15	2013	



# Special Value Test Cases

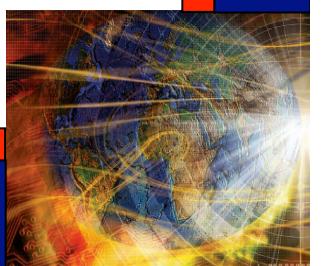
Test Case	Input Values			Reason for Test Case
	Month	Day	Year	
SV-1	Feb	28	1912	Leap day in a leap year
SV-2	Feb	28	1911	Leap day in a non-leap year
SV-3	Feb	29	1912	Leap day increment in a leap year
SV-4	Feb	28	2000	Leap day in the year 2000
SV-5	Feb	28	1900	Leap day in the year 1900
SV-6	Dec	31	1912	Change of year
SV-7	Jan	31	1912	Change of a 31 day month
SV-8	Apr	30	1912	Change of a 30 day month
SV-9	Dec	31	2012	Last day of defined interval



# Output Range Test Cases

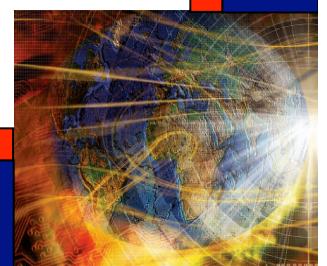
In the case of the NextDate function, the range and domain are identical except for one day . Nothing interesting will be learned from output range test cases for this example.

Part of the reason for this is that the NextDate function is a one-to-one mapping from its domain onto its range. When functions are not one-to-one, output range test cases are more useful.



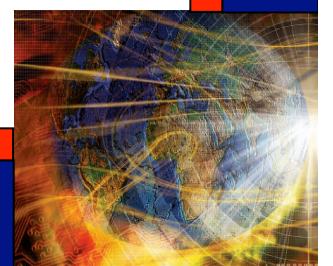
# Pros and Cons of Boundary Value Testing

- **Advantages**
  - Commercial tool support available
  - Easy to do/automate
  - Appropriate for calculation-intensive applications with variables that represent physical quantities (e.g., have units, such as meters, degrees, kilograms)
- **Disadvantages**
  - Inevitable potential for both gaps and redundancies
  - The gaps and redundancies can never be identified (specification-based)
  - Does not scale up well (Jorgensen's Law)
  - Tools only generate inputs, user must generate expected outputs.

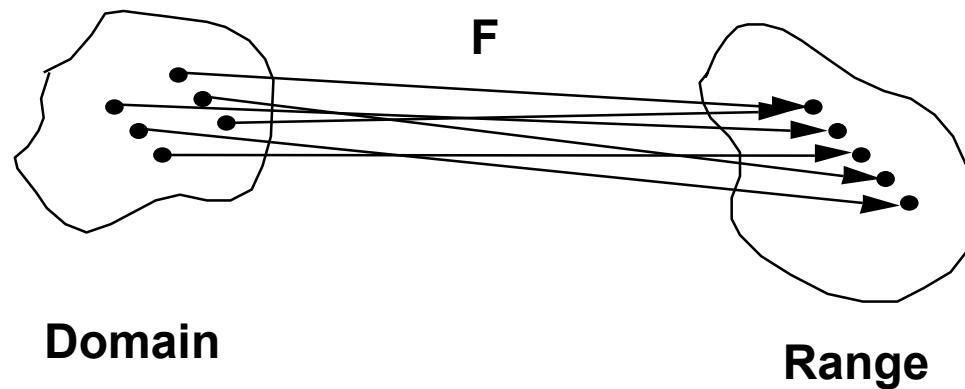


# Chapter 6

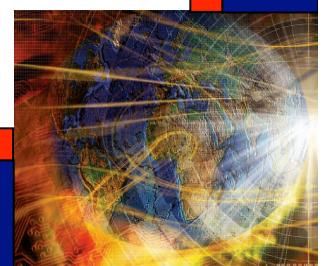
## Equivalence Class Testing



# Equivalence Class Testing

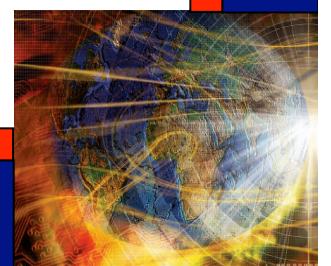


**Equivalence class testing uses information about the functional mapping itself to identify test cases**

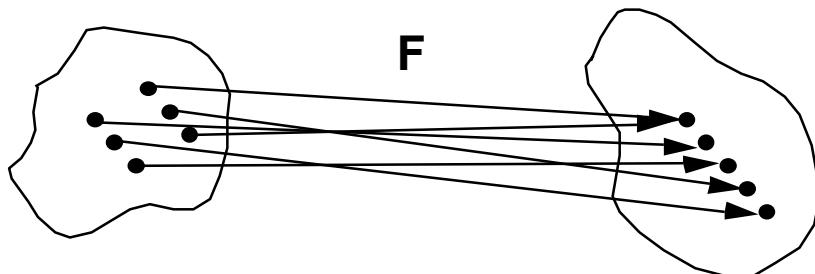


# Equivalence Relations

- Given a relation  $R$  defined on some set  $S$ ,  $R$  is an equivalence relation if (and only if), for all,  $x$ ,  $y$ , and  $z$  elements of  $S$ :
  - $R$  is reflexive, i.e.,  $xRx$
  - $R$  is symmetric, i.e., if  $xRy$ , then  $yRx$
  - $R$  is transitive, i.e., if  $xRy$  and  $yRz$ , then  $xRz$
- An equivalence relation,  $R$ , induces a partition on the set  $S$ , where a partition is a set of subsets of  $S$  such that:
  - The intersection of any two subsets is empty, and
  - The union of all the subsets is the original set  $S$
- Note that the intersection property assures no redundancy, and the union property assures no gaps.



# Equivalence Partitioning



Domain

F

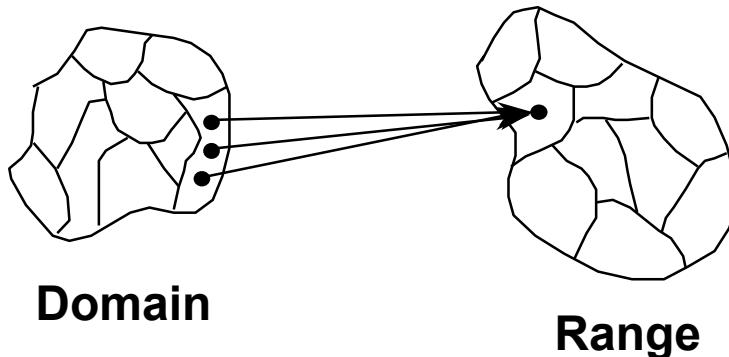
Range

Define relation R as follows:

for  $x, y$  in domain,  $xRy$  iff  $F(x) = F(y)$ .

Facts:

1. R is an equivalence relation.
2. An equivalence relation induces a partition on a set.
3. Works best when F is many-to-one
4. (pre-image set)



Domain

Range

Test cases are formed by selecting one value from each equivalence class.

- reduces redundancy
- identifying the classes may be hard



# Forms of Equivalence Class Testing

- **Normal: classes of valid values of inputs**
- **Robust: classes of valid and invalid values of inputs**
- **Weak: (single fault assumption) one from each class**
- **Strong: (multiple fault assumption) one from each class in Cartesian Product**
- **We compare these for a function of two variables,  $F(x_1, x_2)$**
- **Extension to problems with 3 or more variables is “obvious”.**

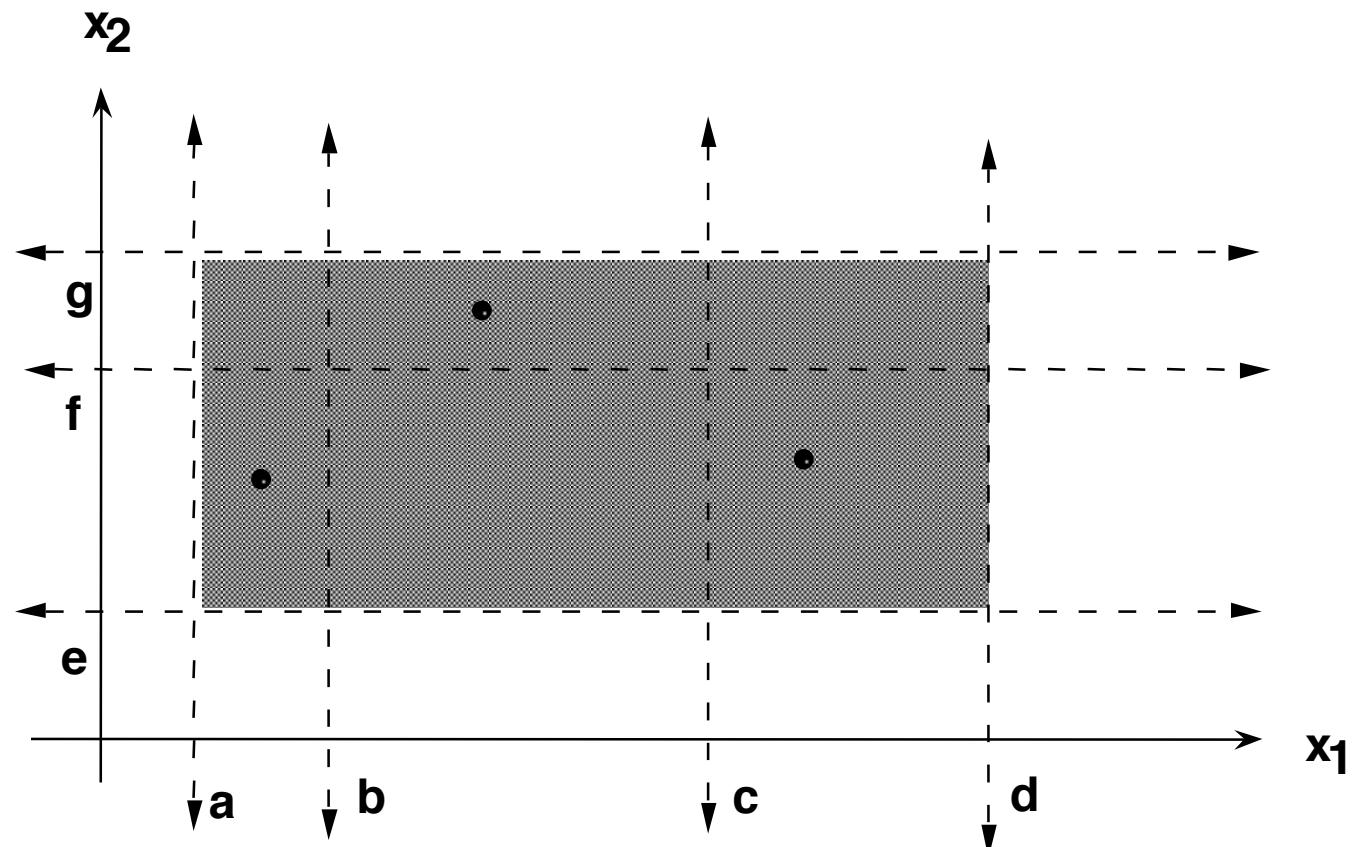


# Weak Normal Equivalence Class Testing

- Identify equivalence classes of valid values.
- Test cases have all valid values.
- Detects faults due to calculations with valid values of a single variable.
- OK for regression testing.



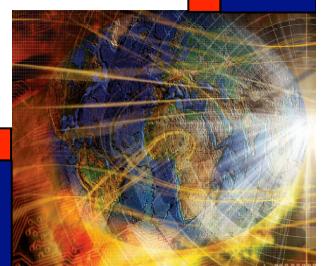
## Weak Normal Equivalence Class Test Cases



Equivalence classes (of valid values):

$\{a \leq x_1 < b\}$ ,  $\{b \leq x_1 < c\}$ ,  $\{c \leq x_1 \leq d\}$

$\{e \leq x_2 < f\}$ ,  $\{f \leq x_2 \leq g\}$

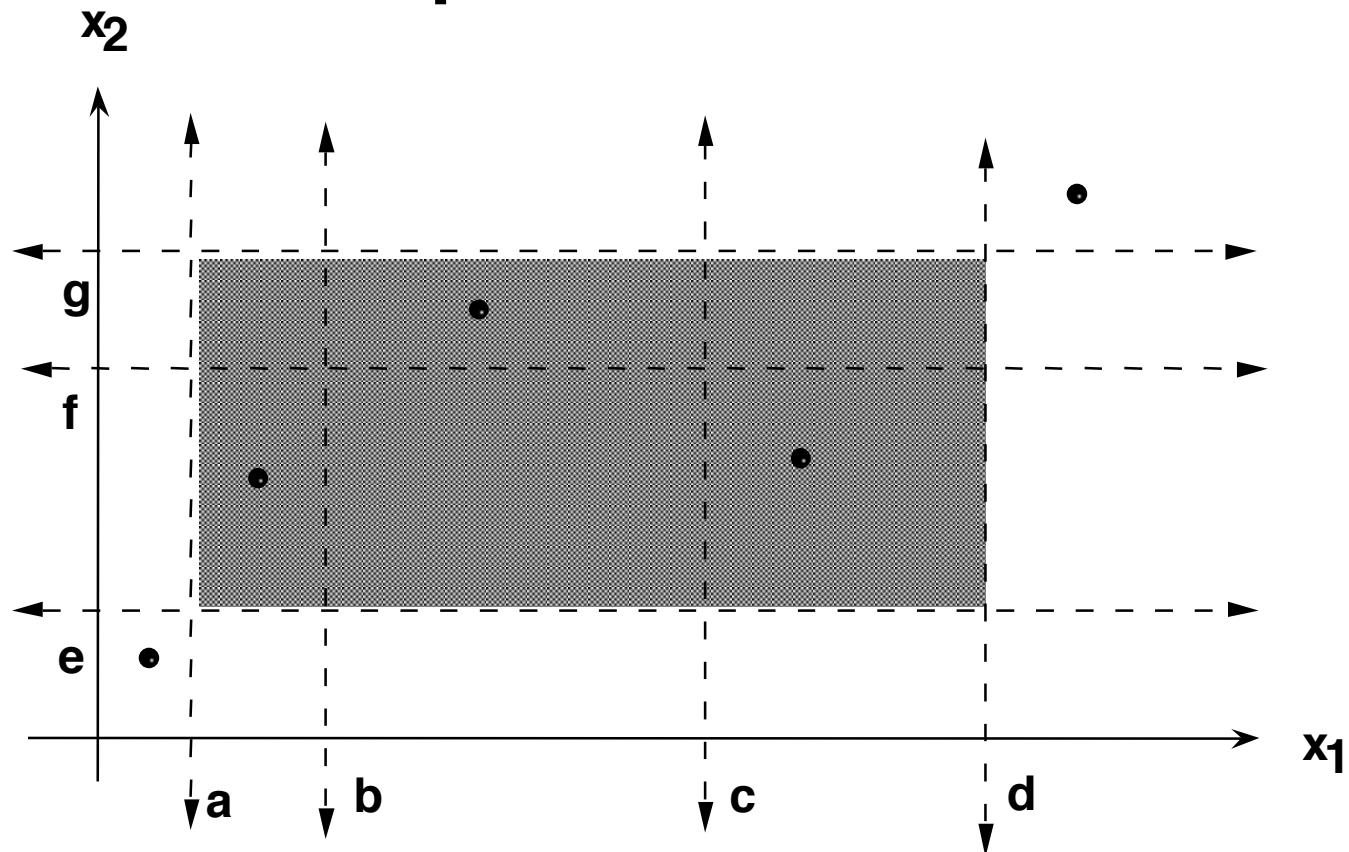


# Weak Robust Equivalence Class Testing

- Identify equivalence classes of valid and invalid values.
- Test cases have all valid values except one invalid value.
- Detects faults due to calculations with valid values of a single variable.
- Detects faults due to invalid values of a single variable.
- OK for regression testing.



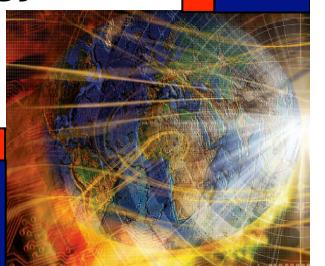
# Weak Robust Equivalence Class Test Cases



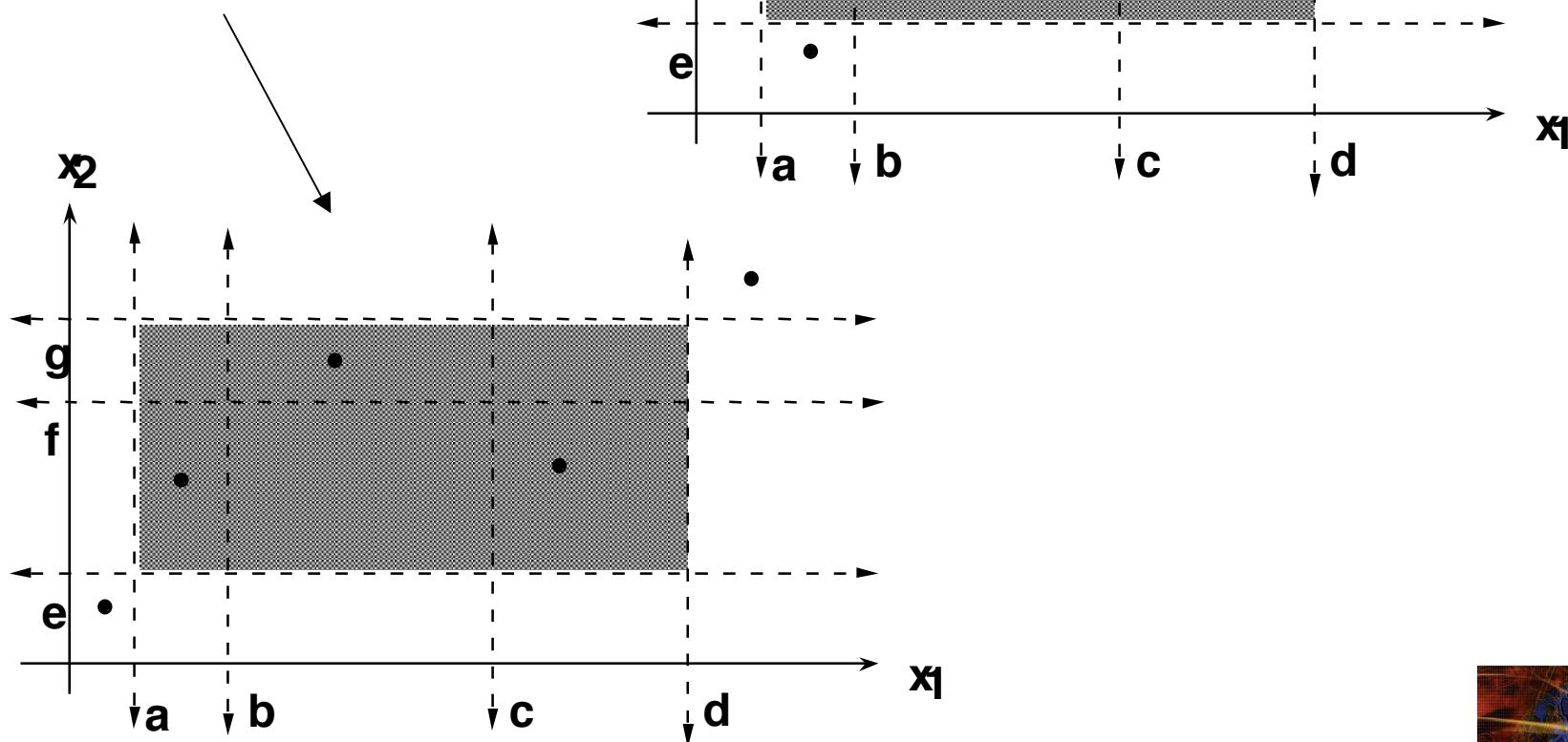
**Equivalence classes (of valid and invalid values):**

$\{a \leq x_1 < b\}$ ,  $\{b \leq x_1 < c\}$ ,  $\{c \leq x_1 \leq d\}$ ,  $\{e \leq x_2 < f\}$ ,  $\{f \leq x_2 \leq g\}$

**Invalid classes:**  $\{x_1 < a\}$ ,  $\{x_1 > d\}$ ,  $\{x_2 < e\}$ ,  $\{x_2 > g\}$

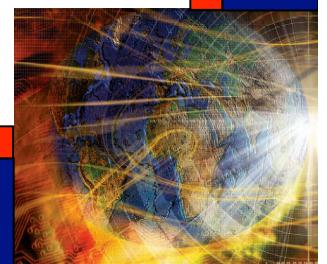


Is this  
preferable  
to this? Why?

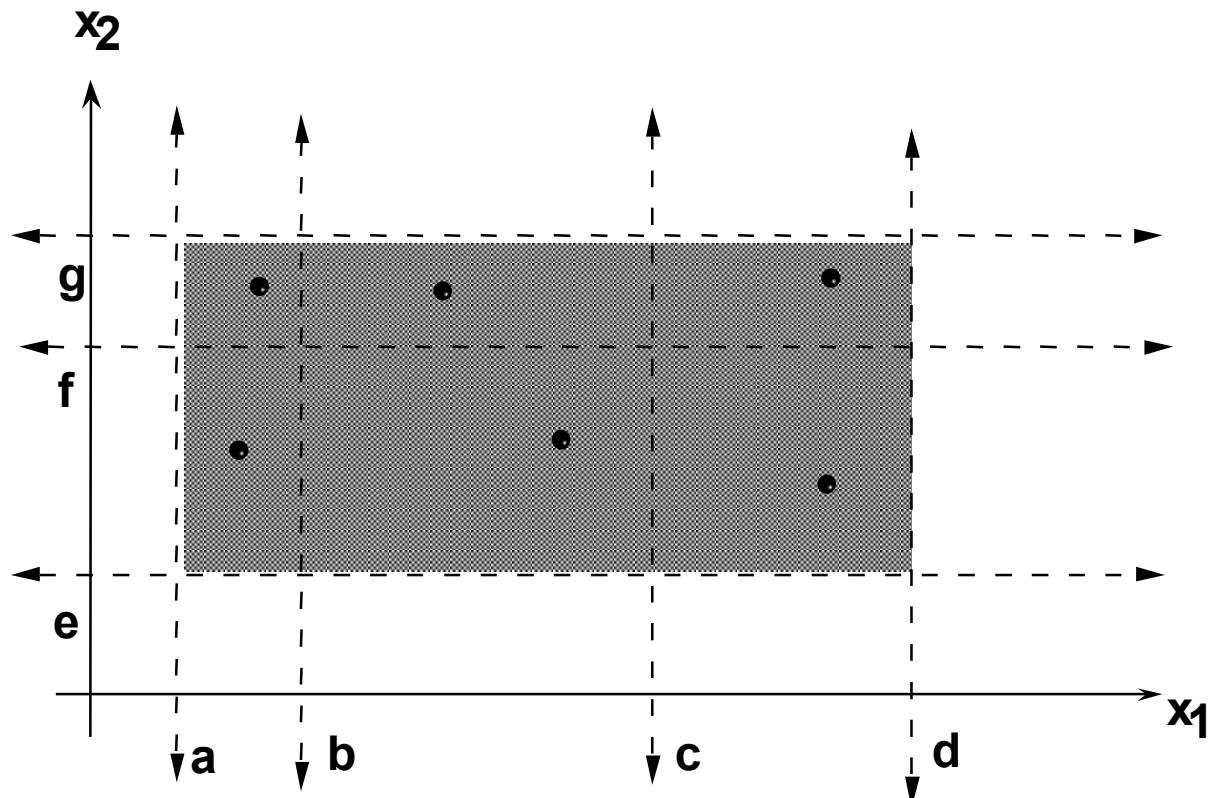


# Strong Normal Equivalence Class Testing

- Identify equivalence classes of valid values.
- Test cases from Cartesian Product of valid values.
- Detects faults due to interactions with valid values of any number of variables.
- OK for regression testing, better for progression testing.



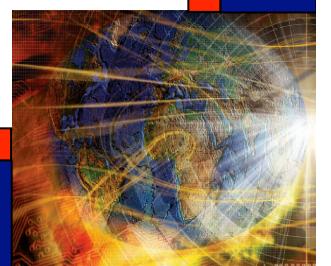
# Strong Normal Equivalence Class Test Cases



Equivalence classes (of valid values):

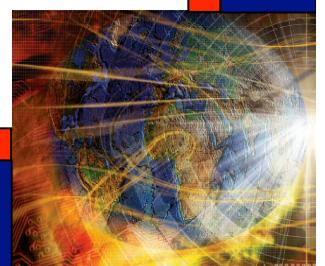
$\{a \leq x_1 < b\}, \{b \leq x_1 < c\}, \{c \leq x_1 \leq d\}$

$\{e \leq x_2 < f\}, \{f \leq x_2 \leq g\}$

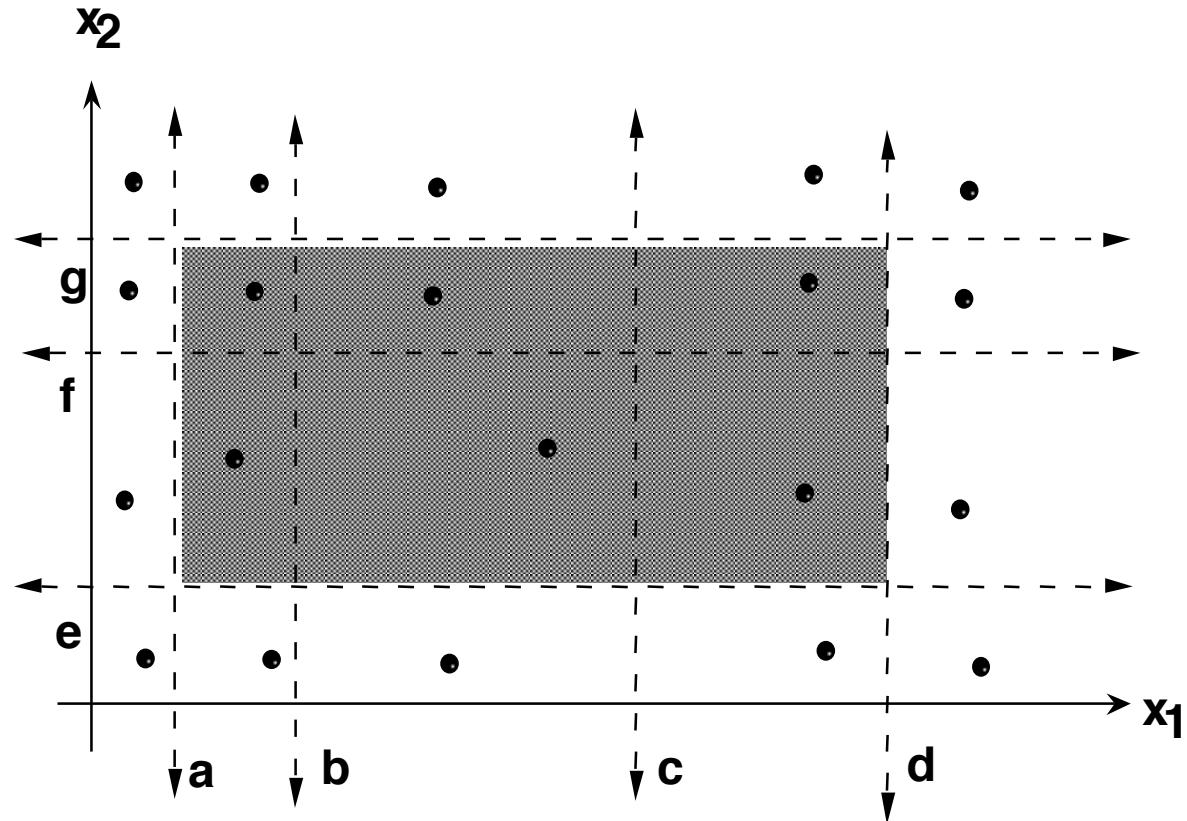


# Strong Robust Equivalence Class Testing

- Identify equivalence classes of valid and invalid values.
- Test cases from Cartesian Product of all classes.
- Detects faults due to interactions with any values of any number of variables.
- OK for regression testing, better for progression testing.  
(Most rigorous form of Equivalence Class testing, BUT,  
Jorgensen's First Law of Software Engineering applies.)
- Jorgensen's First Law of Software Engineering:
  - The product of two big numbers is a really big number.
  - (scaling up can be problematic)



# Strong Robust Equivalence Class Test Cases



Equivalence classes (of valid and invalid values):

$\{a \leq x_1 < b\}$ ,  $\{b \leq x_1 < c\}$ ,  $\{c \leq x_1 < d\}$ ,  $\{e \leq x_2 < f\}$ ,  $\{f \leq x_2 < g\}$

Invalid classes:  $\{x_1 < a\}$ ,  $\{x_1 > d\}$ ,  $\{x_2 < e\}$ ,  $\{x_2 > g\}$



# Selecting an Equivalence Relation

**There is no such thing as THE equivalence relation.  
If x and y are days, some possibilities for Nextdate are:**

- $x R y$  iff x and y are mapped onto the same year
- $x R y$  iff x and y are mapped onto the same month
- $x R y$  iff x and y are mapped onto the same date
- $x R y$  iff x(day) and y(day) are “treated the same”
- $x R y$  iff x(month) and y(month) are “treated the same”
- $x R y$  iff x(year) and y(year) are “treated the same”

**Best practice is to select an equivalence relation that reflects the behavior being tested.**



# NextDate Equivalence Classes

- Month:
  - M1 = { month : month has 30 days}
  - M2 = { month : month has 31 days}
  - M3 = { month : month is February}
- Day
  - D1 = {day : 1 <= day <= 28}
  - D2 = {day : day = 29 }
  - D3 = {day : day = 30 }
  - D4 = {day : day = 31 }
- Year (are these disjoint?)
  - Y1 = {year : year = 2000}
  - Y2 = {year : 1812 <= year <= 2012 AND (year ≠ 0 Mod 100)  
and (year = 0 Mod 4)}
  - Y3 = {year : (1812 <= year <= 2012 AND (year ≠ 0 Mod 4)



# Not Quite Right

- A better set of equivalence classes for year is
  - Y1 = {century years divisible by 400} i.e., century leap years
  - Y2 = {century years not divisible by 400} i.e., century common years
  - Y3 = {non-century years divisible by 4} i.e., ordinary leap years
  - Y4 = {non-century years not divisible by 4} i.e., ordinary common years
- All years must be in range:  $1812 \leq \text{year} \leq 2012$
- Note that these equivalence classes are disjoint.

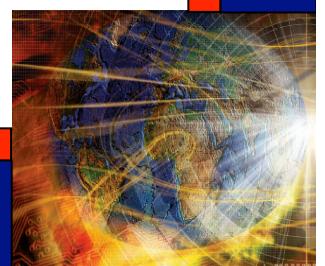


# Weak Normal Equivalence Class Test Cases

Select test cases so that one element from each input domain equivalence class is used as a test input value.

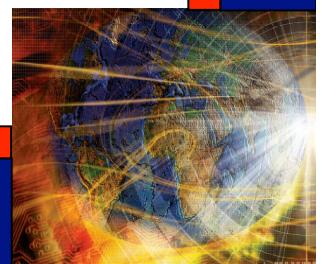
Test Case	Input Domain Equiv. Classes	Input Values	Expected Outputs
WN-1	M1, D1, Y1	April 1 2000	April 2 2000
WN-2	M2, D2, Y2	Jan. 29 1900	Jan. 30 1900
WN-3	M3, D3, Y3	Feb. 30 1812	impossible
WN-4	M1, D4, Y4	April 31 1901	impossible

Notice that all forms of equivalence class testing presume that the variables in the input domain are independent; logical dependencies are unrecognized.



# Strong Normal Equivalence Class Test Cases

- With 4 day classes, 3 month classes, and 4 year classes, the Cartesian Product will have 48 equivalence class test cases. (Jorgensen's First Law of Software Engineering strikes again!)
- Note some judgment is required. Would it be better to have 5 day classes, 4 month classes and only 2 year classes? (40 test cases)
- Questions such as this can be resolved by considering Risk.



# Revised NextDate Domain Equivalence Classes

- Month:

- M1 = { month : month has 30 days}
- M2 = { month : month has 31 days except December}
- M3 = { month : month is February}
- M4 = {month : month is December}

- Day

- D1 = {day : 1 <= day <= 27}
- D2 = {day : day = 28 }
- D3 = {day : day = 29 }
- D4 = {day : day = 30 }
- D5 = {day : day = 31 }

- Year (are these disjoint?)

- Y1 = {year : year is a leap year}
- Y2 = {year : year is a common year}

The Cartesian Product of these contains 40 elements.



# When to Use Equivalence Class Testing

- **Variables represent logical (rather than physical) quantities.**
- **Variables “support” useful equivalence classes.**
- **Try to define equivalence classes for**
  - **The Triangle Problem**
    - $0 < \text{sideA} < 200$
    - $0 < \text{sideB} < 200$
    - $0 < \text{sideC} < 200$
  - **The Commission Problem (exercise)**



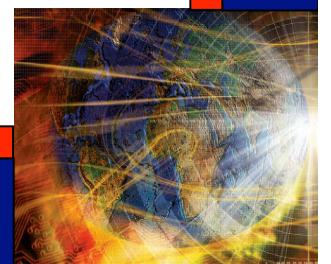
# Another Equivalence Class Strategy

- “Work backwards” from output classes.
- For the Triangle Problem, could have
  - {x, y, z such that they form an Equilateral triangle}
  - {x, y, z such that they form an Isosceles triangle with  $x = y$ }
  - {x, y, z such that they form an Isosceles triangle with  $x = z$ }
  - {x, y, z such that they form an Isosceles triangle with  $y = z$ }
  - {x, y, z such that they form a Scalene triangle}
- How many equivalence classes will be needed for x,y,z Not a triangle?



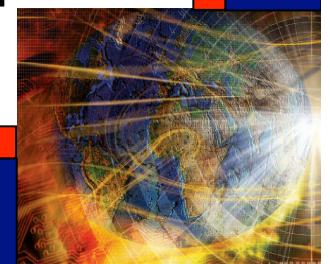
# In-Class Exercise

- **Apply the “working backwards” approach to develop equivalence classes for the Commission Problem.**
- **Hint: use boundaries in the output space.**



# Assumption Matrix

	<b>Valid Values</b>	<b>Valid and Invalid Values</b>
<b>Single fault</b>	<b>Boundary Value</b>  <b>Weak Normal Equiv. Class</b>	<b>Robust Boundary Value</b>  <b>Weak Robust Equiv. Class</b>
<b>Multiple fault</b>	<b>Worst Case Boundary Value</b>  <b>Strong Normal Equiv. Class</b>	<b>Robust Worst Case Boundary Value</b>  <b>Strong Robust Equiv. Class</b>



# Chapter 7

## Decision Table Based Testing



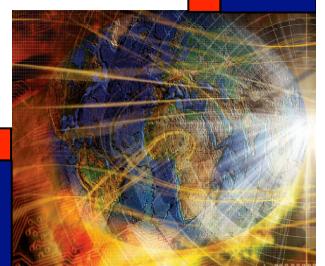
# Decision Table Based Testing

- Originally known as Cause and Effect Graphing
  - Done with a graphical technique that expressed AND-OR-NOT logic.
  - Causes and Effects were graphed like circuit components
  - Inputs to a circuit “caused” outputs (effects)
- Equivalent to forming a decision table in which:
  - inputs are conditions
  - outputs are actions
- Test every (possible) rule in the decision table.
- Recommended for logically complex situations.



# Decision Tables

- **Represent complex conditional behavior.**
- **Support extensive analysis**
  - Consistency
  - Completeness
  - Redundancy
  - Algebraic simplification
- **Executable (and compilable)**
- **Two forms: Limited and Extended Entry.**
- **“Don't Care” condition entries require special attention.**
- **Dependencies usually yield impossible situations**



# Decision Table Terminology

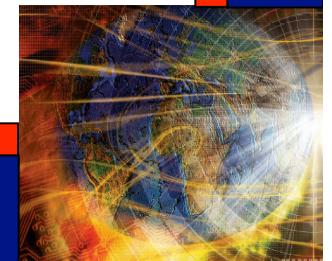
		Stub	Entry					
Conditions	c1	True			False			
		True		False	True		False	
		T	F	--	T	F	--	
Actions	a1	X	X		X			
	a2	X			X	X		
	a3		X		X	X		
	a4			X			X	
↑								
Rule								



# One Decision Table for the Triangle Problem

c1: a, b, c are a triangle?	N	Y	Y	Y	Y	Y	Y	Y	Y
c2: a = b?	--	Y	Y	Y	Y	N	N	N	N
c3: a = c?	--	Y	Y	N	N	Y	Y	N	N
c4: b = c?	--	Y	N	Y	N	Y	N	Y	N
a1: Not a triangle	X								
a2: Scalene									X
a3: Isosceles					X		X	X	
a4: Equilateral		X							
a5: Impossible			X	X		X			

Why are rules 3, 4, and 6 impossible?



# Decision Table with Mutually Exclusive Conditions

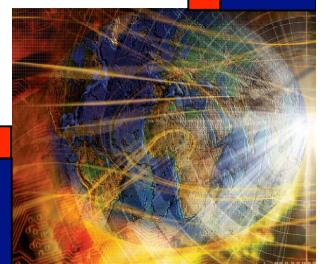
Conditions	Rule 1	Rule 2	Rule 3
c1: 30-day month	T	—	—
c2: 31-day month	—	T	—
c3: February	—	—	T



# Rule Counting to Check for Completeness

conditions	R1	R2	R3
c1: month in M1	T	--	--
c2: month in M2	--	T	--
c3: month in M3	--	--	T
<b>Rule count</b>	<b>4</b>	<b>4</b>	<b>4</b>

- A limited Entry decision table with n conditions has  $2^n$  rules
- A Don't Care entry doubles the count of a rule
- What are the possibilities when rule count is not  $2^n$  ?
- More precise to use F! (must be false) than -- (don't care)



# A Redundant Decision Table

conditions	1-4	5	6	7	8	9
c1:	T	F	F	F	F	T
c2:	--	T	T	F	F	F
c3:	--	T	F	T	F	F
<hr/>						
a1:	X	X	X	--	--	X
a2:	--	X	X	X	--	--
a3:	X	--	X	X	X	X

- Rule 9 is identical to Rule 4 (T, F, F)
- Since the action entries for rules 4 and 9 are identical, there is no ambiguity, just redundancy.



# Decision Table Exercise

## (revisit the false negative, false positive question)

- **Suggested conditions**
  - Expected output is correct
  - Observed output is correct
  - Expected and observed outputs agree
- **Suggested actions**
  - True pass
  - True fail
  - False pass
  - False fail
  - Impossible



# An Inconsistent Decision Table

conditions	1-4	5	6	7	8	9
c1:	T	F	F	F	F	T
c2:	--	T	T	F	F	F
c3:	--	T	F	T	F	F
<hr/>						
a1:	X	X	X	--	--	--
a2:	--	X	X	X	--	X
a3:	X	--	X	X	X	--

- Rule 9 is identical to Rule 4 (T, F, F)
- Since the action entries for rules 4 and 9 are different there is ambiguity.
- This table is inconsistent, and the inconsistency implies non-determinism.



# Nextdate Limited Entry Decision Table

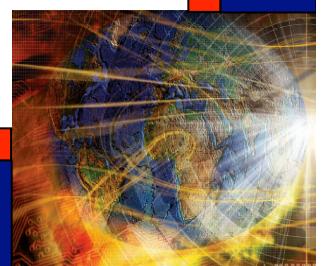
This decision table will have 256 rules,  
many of which will be logically impossible.

## Conditions

- c1: month in M1?
- c2: month in M2?
- c3: month in M3?
- c4: day in D1?
- c5: day in D2?
- c6: day in D3?
- c7: day in D4?
- c8: leap year?

## Actions

- a1: impossible
- a2: next date



# Decision Table Based Test Cases

1. Decision table testing begins with equivalence classes for conditions as in equivalence class testing.
2. The sparseness due to the assumption of independence is addressed by careful examination of elements in the cross product.
3. For the equivalence classes defined earlier, the cross product contains 36 elements. The corresponding decision table has 36 rules.

<M1, D1, Y1>, <M1, D2, Y1>, <M1, D3, Y1>, <M1, D4, Y1>,  
<M2, D1, Y1>, <M2, D2, Y1>, <M2, D3, Y1>, <M2, D4, Y1>,  
<M3, D1, Y1>, <M3, D2, Y1>, <M3, D3, Y1>, <M3, D4, Y1>,

<M1, D1, Y2>, <M1, D2, Y2>, <M1, D3, Y2>, <M1, D4, Y2>,  
<M2, D1, Y2>, <M2, D2, Y2>, <M2, D3, Y2>, <M2, D4, Y2>,  
<M3, D1, Y2>, <M3, D2, Y2>, <M3, D3, Y2>, <M3, D4, Y2>,

<M1, D1, Y3>, <M1, D2, Y3>, <M1, D3, Y3>, <M1, D4, Y3>,  
<M2, D1, Y3>, <M2, D2, Y3>, <M2, D3, Y3>, <M2, D4, Y3>,  
<M3, D1, Y3>, <M3, D2, Y3>, <M3, D3, Y3>, <M3, D4, Y3>.

4. Notice that many of these are impossible, e.g., <M1, D4, \* >



# NextDate Extended Entry Decision Table

## Conditions

c1: month in M1? M2? M3?  
c2: day in D1? D2? D3? D4?  
c3: year in Y1? Y2? Y3?

## Actions

a1: impossible  
a2: increment day  
a3: reset day  
a4: increment month  
a5: reset month  
a6: increment year

This decision table will have 36 rules, and corresponds to the cross product. Many of the rules will be logically impossible.

Many rules would collapse, except for considerations for December.



# Revised NextDate Domain Equivalence Classes

- Month:

- M1 = { month : month has 30 days}
- M2 = { month : month has 31 days except December}
- M3 = { month : month is December}
- M4 = {month : month is February }

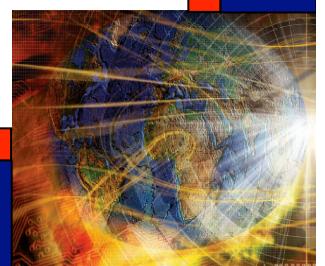
- Day

- D1 = {day : 1 <= day <= 27}
- D2 = {day : day = 28 }
- D3 = {day : day = 29 }
- D4 = {day : day = 30 }
- D5 = {day : day = 31 }

- Year (are these disjoint?)

- Y1 = {year : year is a leap year}
- Y2 = {year : year is a common year}

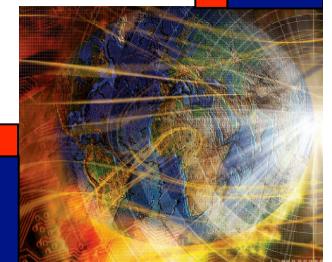
The corresponding decision table of these contains 40 elements.



# NextDate Extended Entry Decision Table

1 2 3 4 5 6 7 8 9 10										11 12 13 14 15 16 17 18 19 20 21 22												
c1: month in	M1					M2					M3					M4						
c2: day in	D 1	D 2	D 3	D 4	D 5	D 1	D 2	D 3	D 4	D 5	D 1,	D 2	D 3	D 4	D 5	D 1	D 2	D 2	D 3	D 3	D 4	D 5
c3: year in	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	Y 1	Y 2	Y 1	Y 2	-	-	
a1: impossible				X																		
a2: increment day	X	X	X			X	X	X			X	X	X	X		X	X			X	X	X
a3: reset day				X				X							X			X				
a4: increment month				X					X									X	X			
a5: reset month									X						X				X			
a6: increment year															X							

**Notice there are 40 rules in this decision table, corresponding to the 40 elements in the cross product of the revised equivalence classes.**



# NextDate Extended Entry Decision Table

**Algebraically Condensed to 13 rules (test cases)**

rules	1-3	4	5	6-9	10	11-14	15	16	17	18	19	20	21,22
c1: month in		M1		M2		M3		M4					
c2: day in	D1,D2,D3	D4	D5	D1,D2,D3,D4	D5	D1,D2,D3,D4	D5	D1	D2	D2	D3	D3	D4, D5
c3: year in	-	-	-	-	-	-	-	-	Y1	Y2	Y1	Y2	-
a1: impossible			X									X	X
a2: increment day	X			X		X		X	X				
a3: reset day		X			X		X			X	X		
a4: increment month		X			X					X	X		
a5: reset month							X						
a6: increment year							X						



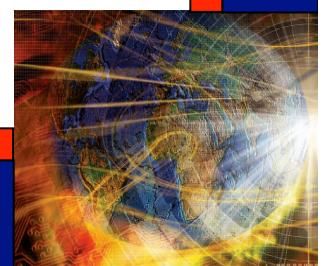
## Procedure for Decision Table Based Testing

- 1. Determine conditions and actions. (Might need to iterate)**
- 2. Develop a (the!) Decision Table, watching for**
  - completeness**
  - don't care entries**
  - redundant and inconsistent rules**
- 3. Each rule defines a test case.**



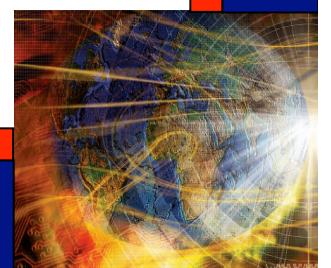
# Procedure for Decision Table Based Testing

- Determine conditions and actions. (Might need to iterate)
- Develop a (the!) Decision Table, watching for
  - Completeness
  - Don't care entries
  - Redundant and/or inconsistent entries
  - Impossible rules
- Each rule defines a test case.



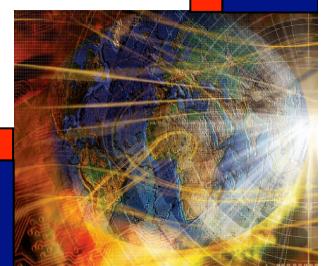
# Chapter 8

## Retrospective on Functional Testing



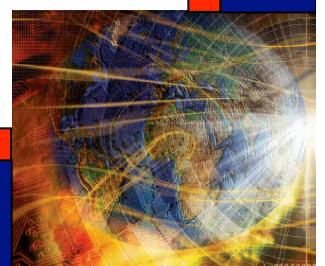
# Retrospective on Functional Testing

- **Test case development effort**
- **Test case effectiveness**
- **Test method selection guidelines**
- **Case study**

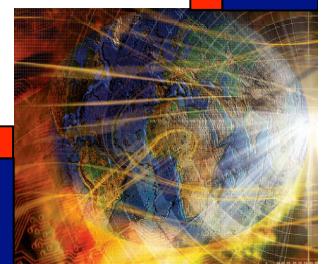
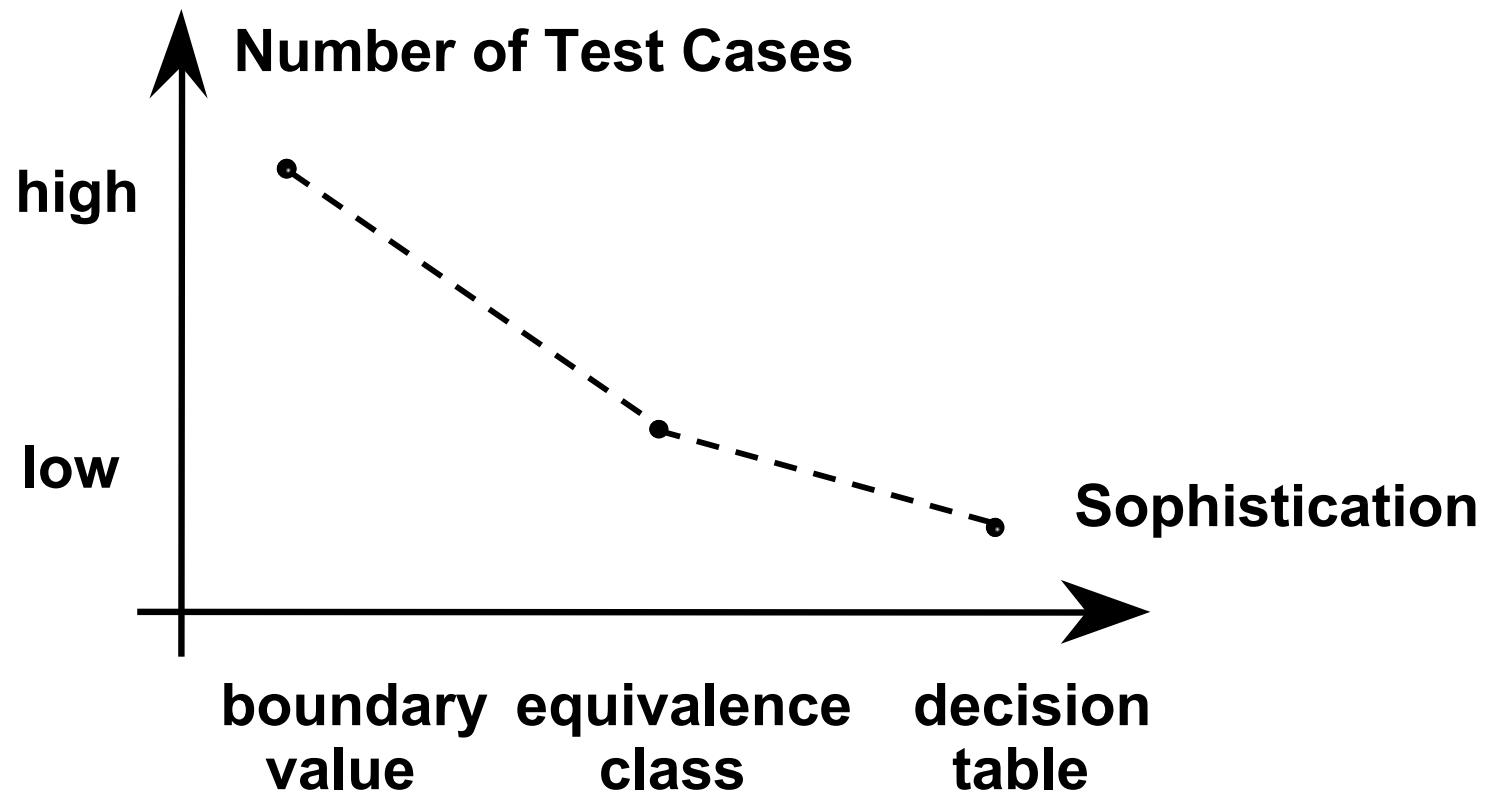


# Test Case Development Effort

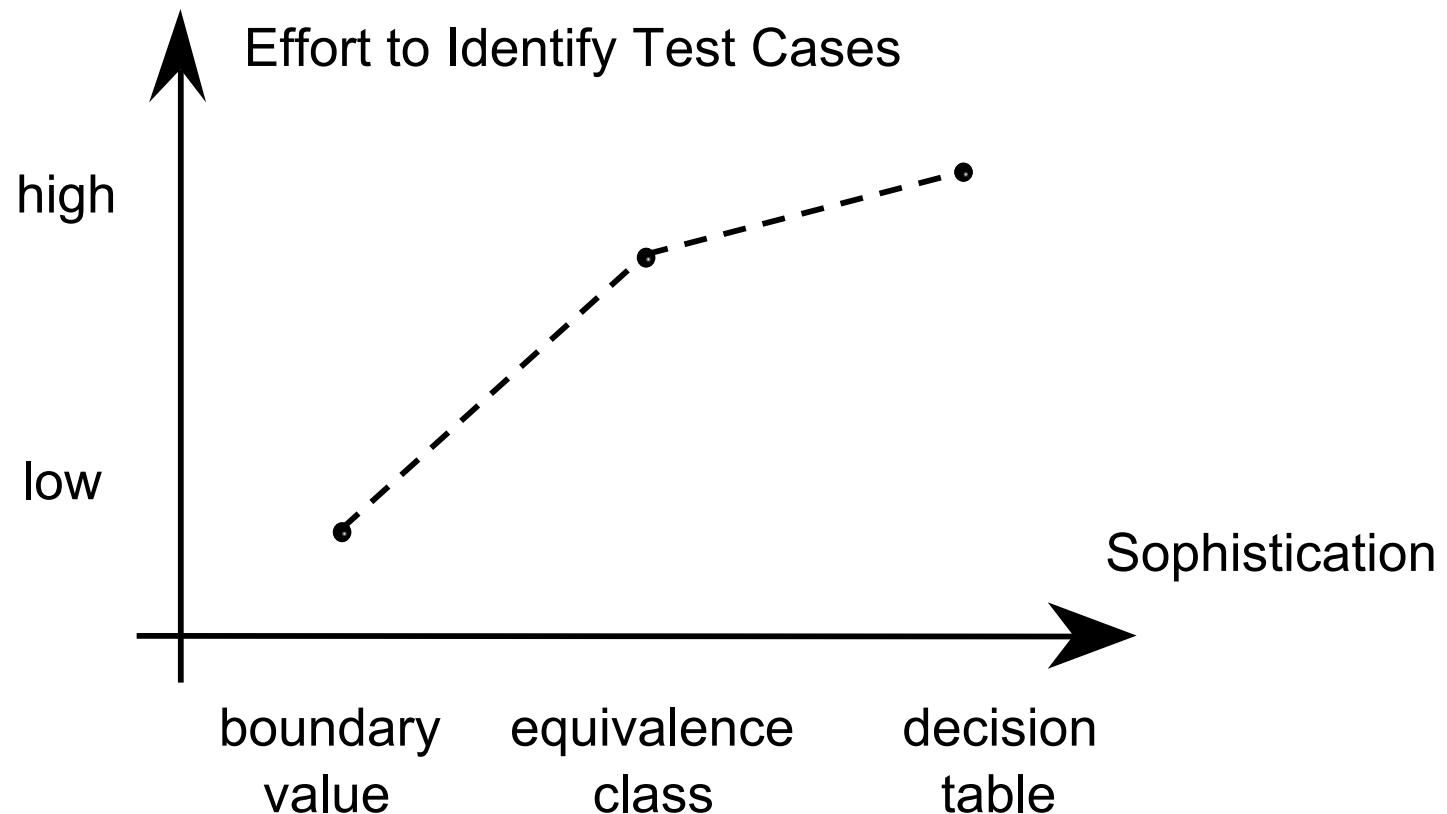
- As with so many things in life,  
“You get out of it what you put into it.” --Dad
- Boundary value: almost mechanical
- Equivalence class: effort to identify classes
- Decision table: still more effort



# Numbers of Test Cases

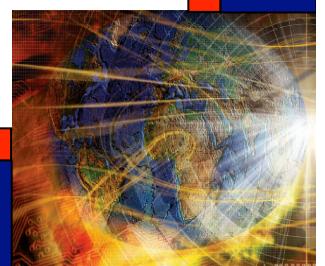


# Test Case Development Effort



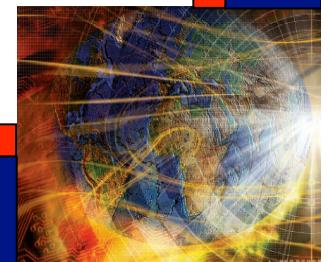
# Test Case Effectiveness

- True trade-off between development effort and number of test cases.
- Vulnerabilities
  - Boundary value testing has gaps and redundancies, and many test cases.
  - Equivalence class testing eliminates the gaps and redundancies, but cannot deal with dependencies among variables.
  - Decision table testing extends equivalence class testing by dealing with dependencies, and supports algebraic reduction of test cases.



# Appropriate Choices of Test Methods

c1. variables (P, physical, L, logical)	P	P	P	P	P	L	L	L	L	L
c2. independent variables?	Y	Y	Y	Y	N	Y	Y	Y	Y	N
c3. single fault assumption?	Y	Y	N	N	-	Y	Y	N	N	-
c4. exception handling?	Y	N	Y	N	-	Y	N	Y	N	-
a1. boundary value analysis (BVA)		x								
a2. robustness BVA	x									
a3. worst case BVA										
a4. robust worst case BVA										
a5. weak normal equiv. class		x					x			
a6. weak robust equiv. class	x					x				
a7. strong normal equiv. class			x					x		
a8. strong robust equiv. class			x				x			
a9. decision table					x					x



# Case Study

A hypothetical Insurance Premium Program computes the semi-annual car insurance premium based on two parameters: the policy holder's age and driving record:

**Premium = BaseRate\*ageMultiplier – safeDrivingReduction**

The ageMultiplier is a function of the policy holder's age, and the safe driving reduction is given when the current points (assigned by traffic courts for moving violations) on the policy holder's driver's license are below an age-related cutoff. Policies are written for drivers in the age range of 16 to 100. Once a policy holder has 12 points, his/her driver's license is suspended (hence there is no need for insurance). The BaseRate changes from time to time; for this example, it is \$500 for a semi-annual premium.



# Insurance Premium Program Data

<i>Age Range</i>	<i>Age Multiplier</i>	<i>Points Cutoff</i>	<i>Safe Driving Reduction</i>
16<= age < 25	2.8	1	50
25<= age < 35	1.8	3	50
35<= age < 45	1.0	5	100
45<= age < 60	0.8	7	150
60<= age <= 100	1.5	5	200

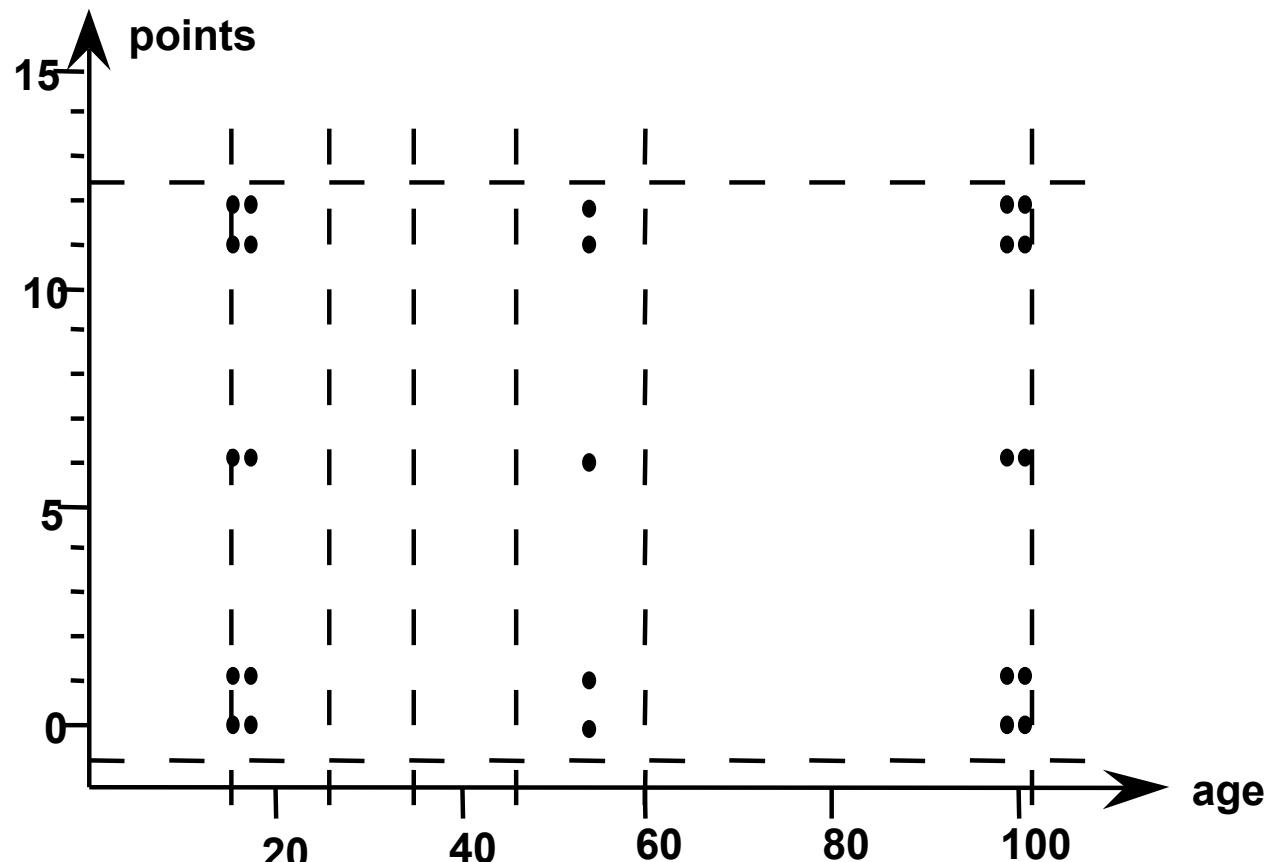


# Insurance Premium Program Calculations, Test Method Selection

- **Premium = BaseRate\*ageMultiplier – safeDrivingReduction**
- **ageMultiplier = F1(age)** [from table]
- **safeDrivingReduction = F2(age, points)** [from table]
- **age and safeDrivingReduction are physical variables, with a dependency in F2.**
- **Boundary values for age: 16, 17, 54, 99, 100**
- **Boundary values for safeDrivingReduction: 0, 1, 6, 11, 12**
- **Robust values for age and safeDrivingReduction are not allowed by business rules.**
- **Worst case BVA yields 25 test cases, and many gaps, some redundancy. Need something better.**



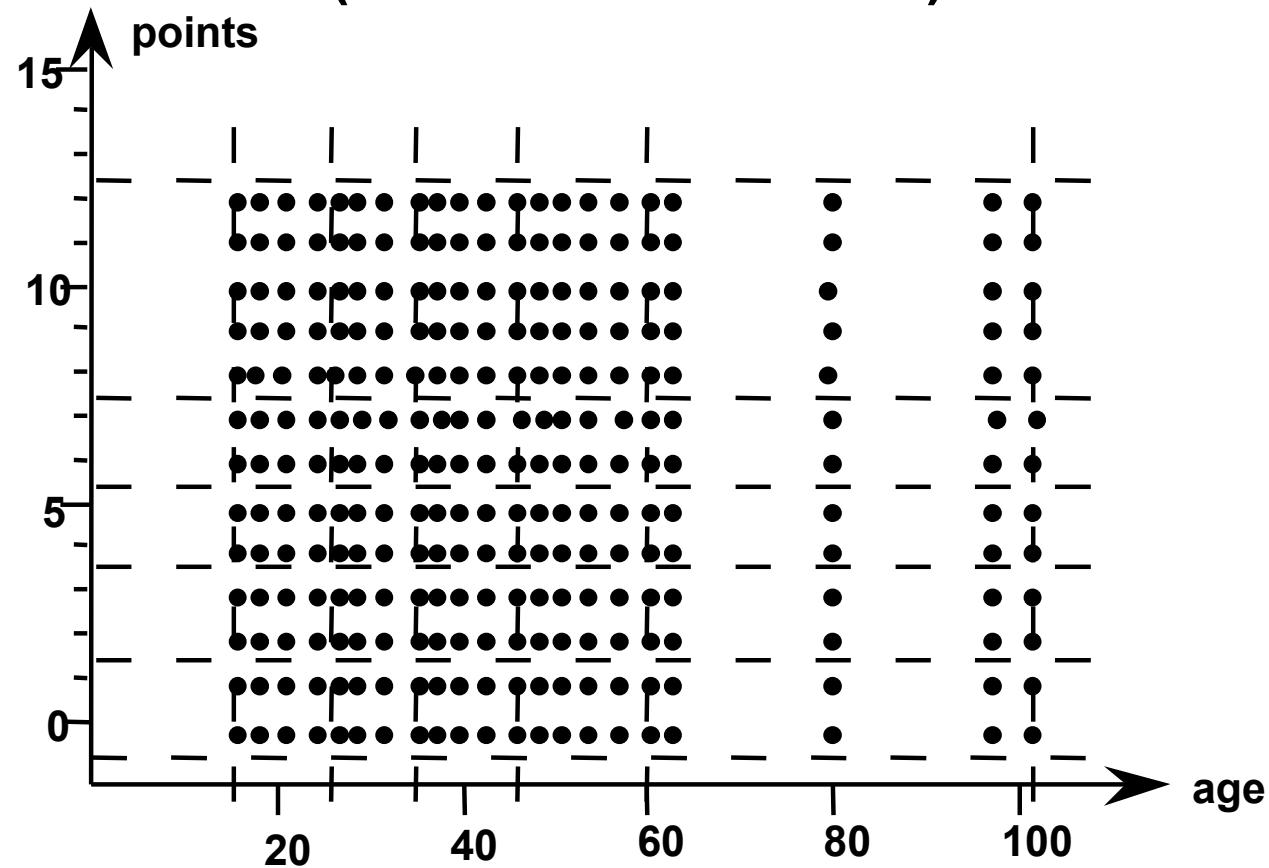
# Graph of Boundary Value Test Cases



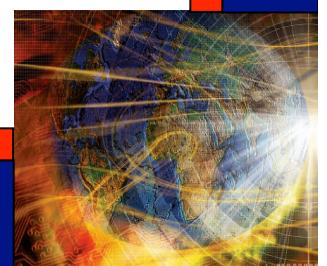
**Severe gaps!**



# Graph of Boundary Value Test Cases (refined boundaries)

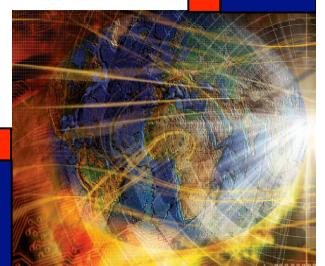


**Severe redundancy!**

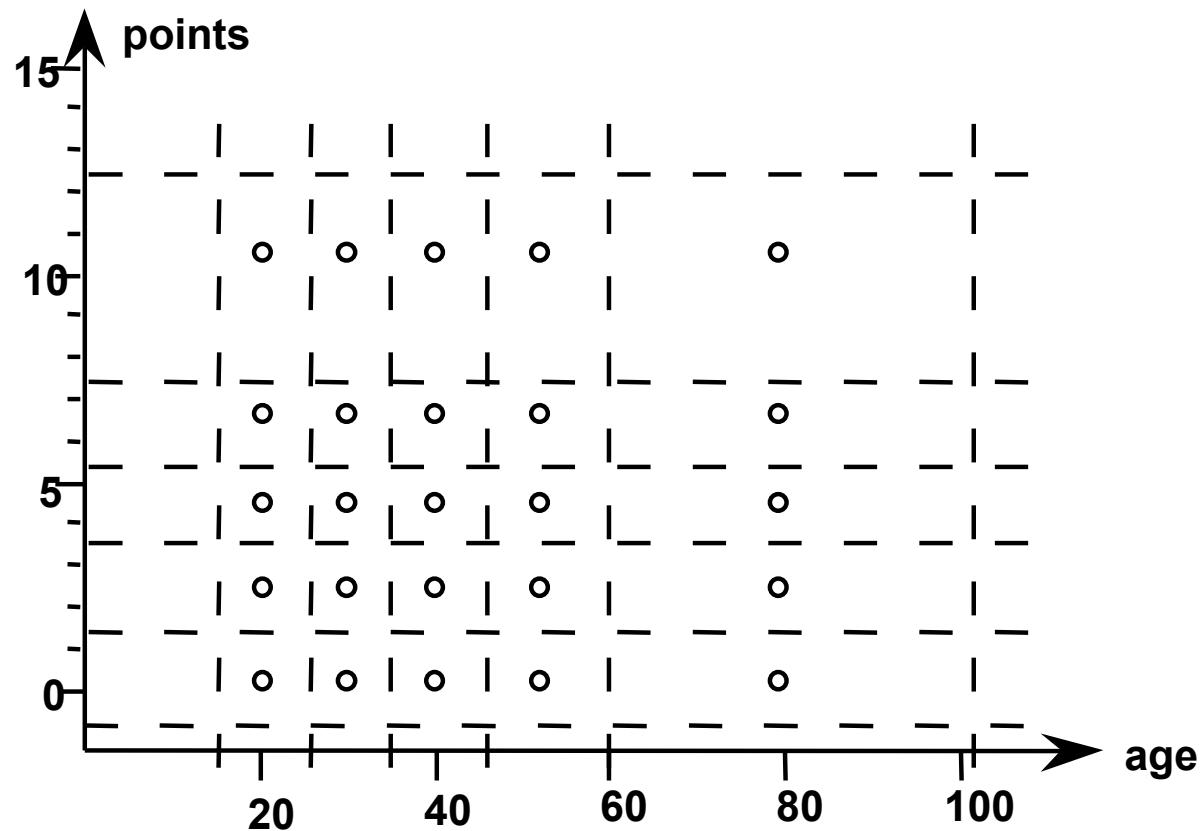


# Insurance Premium Program Test Method Selection

- **age has ranges that receive similar treatment. equivalence class testing is indicated.**
  - Age ranges per the data table
- **The points cutoff is also a range, further indication for equivalence class testing.**
  - Points {0, 1}
  - Points {2, 3}
  - Points {4, 5}
  - Points {6, 7}
  - Points {8, 9, 10, 11, 12}



# Insurance Premium Program Strong Normal Equivalence Class Test Cases



Still a lot of redundancy, try decision tables.

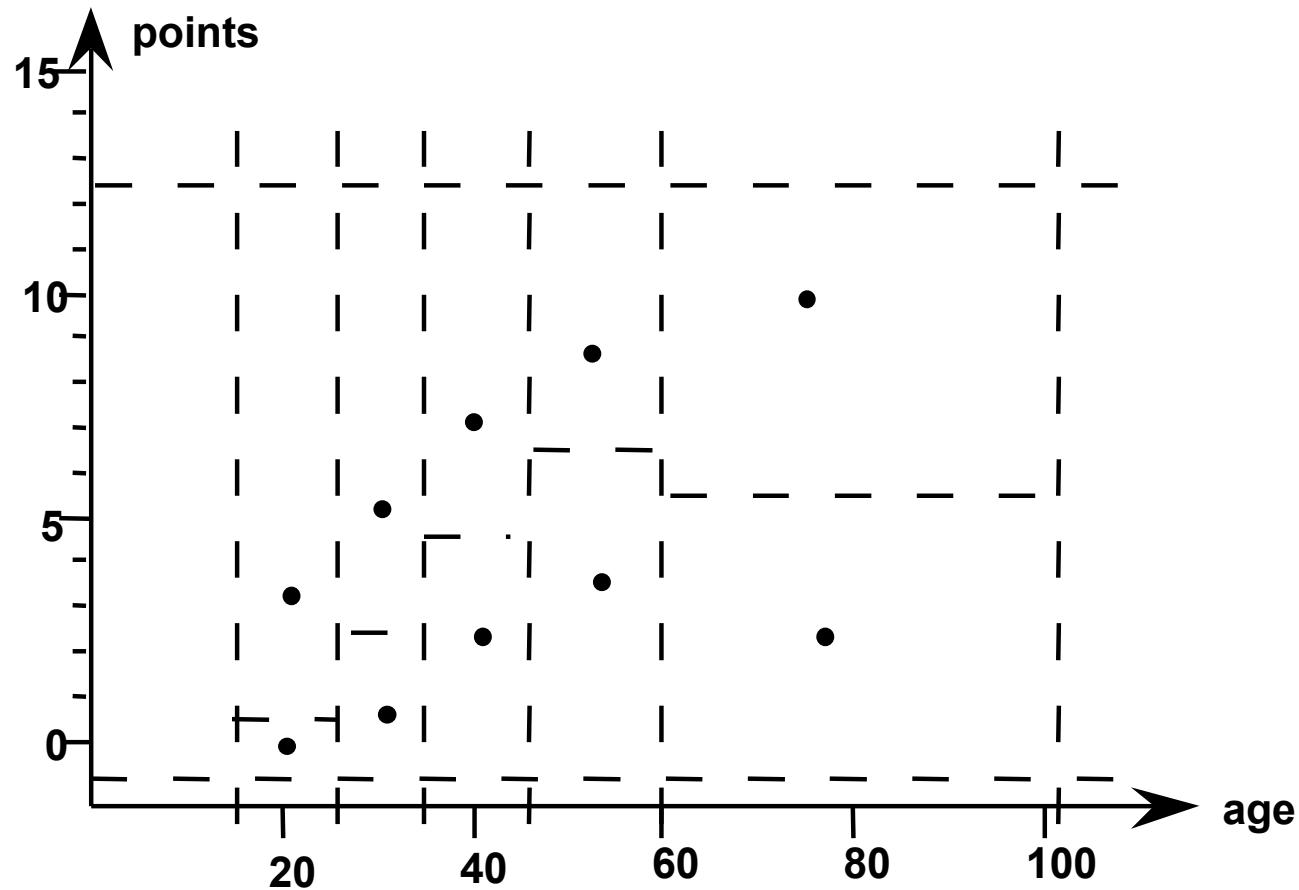


# Insurance Premium Program Decision Table Test Cases

c1. age is	16-25		25-35		35-45		45-60		60-100	
c2. points	0	1-12	0-2	3-13	0-4	5-12	0-6	7-12	0-4	5-12
a1. age multiplier	2.8	2.8	1.8	1.8	1.8	1.8	0.8	0.8	1.5	1.5
a2. safeDriving	50	--	50	--	100	--	150	--	200	--



# Insurance Premium Program Decision Table Test Cases

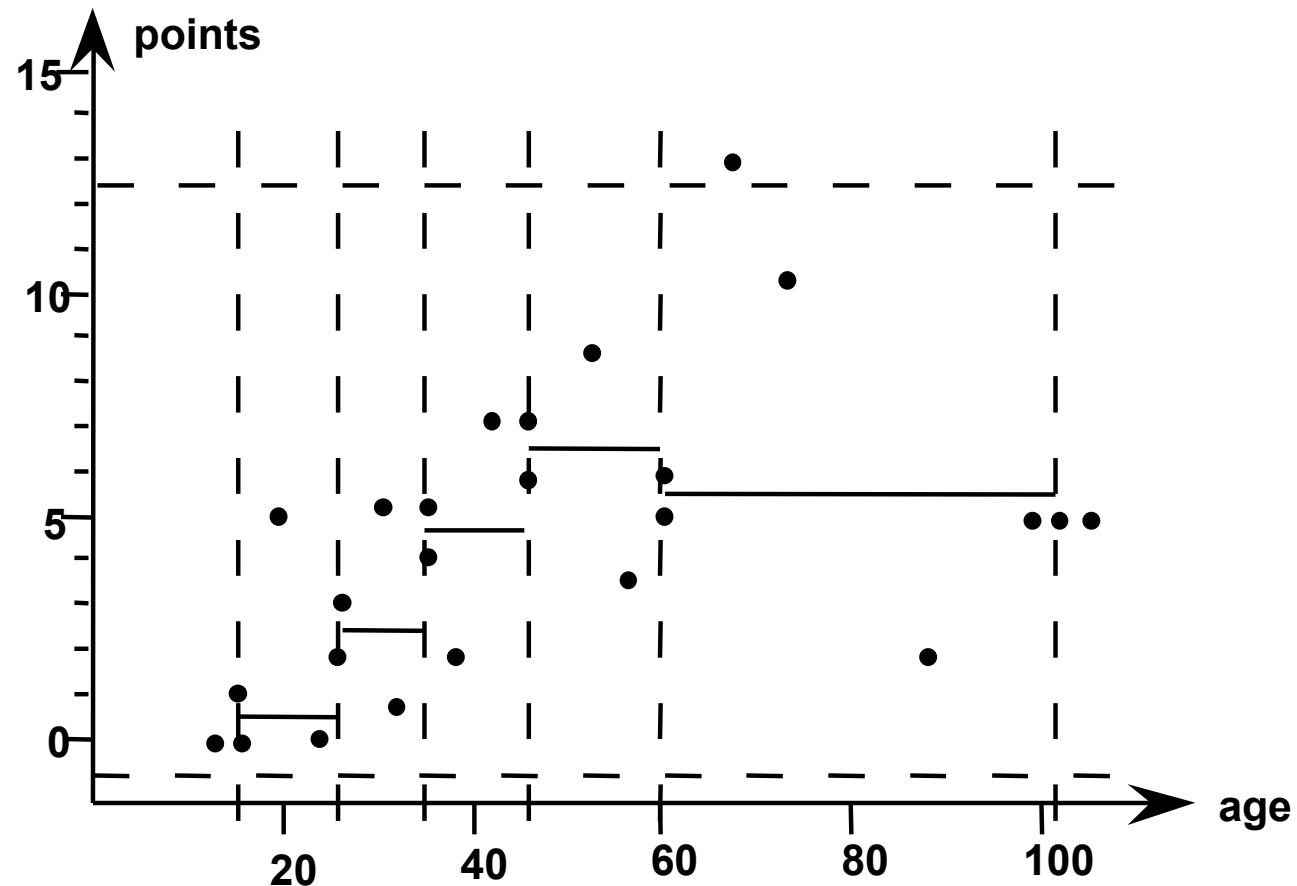


What about age range endpoints?

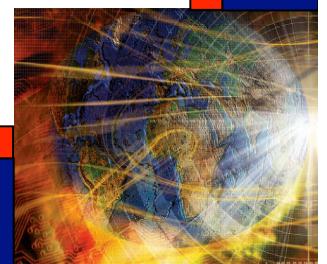


# Insurance Premium Program Test Cases

## (Decision table with boundary values hybrid)

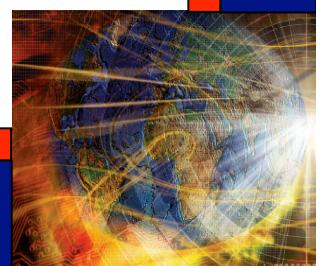


Ahhhh, at last!



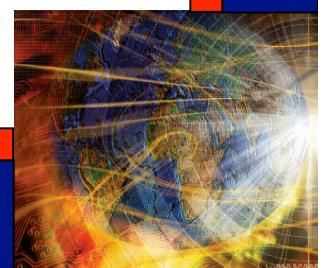
# Wrap Up

- **The inherent nature of the program being tested should dictate the test method.**
  - The decision table “expert system” (slide 7) recommendation is just a start.
  - Applications are seldom “chemically pure”.
- **Hybrid combinations of test methods can be very useful.**
- **Good judgment, based on insight, is a sign of a craftsman.**



# Chapter 9

## Path Testing–Part 1



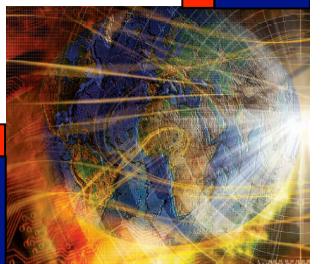
# Structural (Code-Based) Testing

- Complement of/to Functional Testing
- Based on Implementation
- Powerful mathematical formulation
  - program graph
  - define-use path
  - Program slices
- Basis for Coverage Metrics (a better answer for gaps and redundancies)
- Usually done at the unit level
- Not very helpful to identify test cases
- Extensive commercial tool support

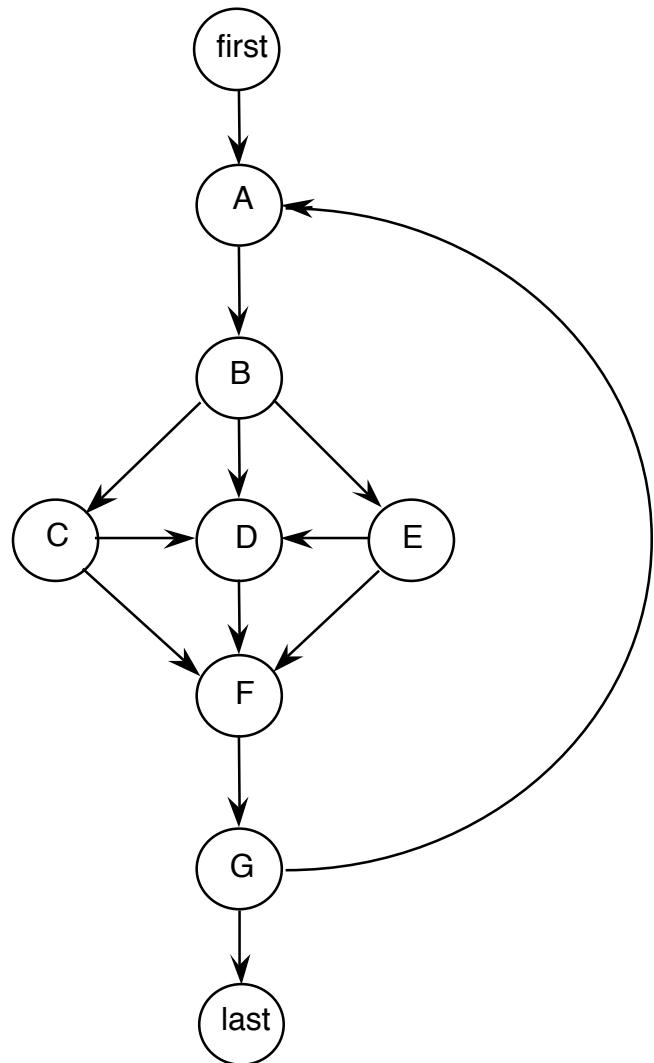


# Path Testing

- Paths derived from some graph construct.
- When a test case executes, it traverses a path.
- Huge number of paths implies some simplification is needed.
- Big Problem: infeasible paths.
- Big Question: what kinds of faults are associated with what kinds of paths?
- By itself, path testing can lead to a false sense of security.



# The Common Objection: Trillions of Paths



If the loop executes up to 18 times,  
there are 4.77 Trillion paths.

$(5^{18} = 3,814,697,265,625)$   
(Actually, it is 4,768,371,582,030 paths)

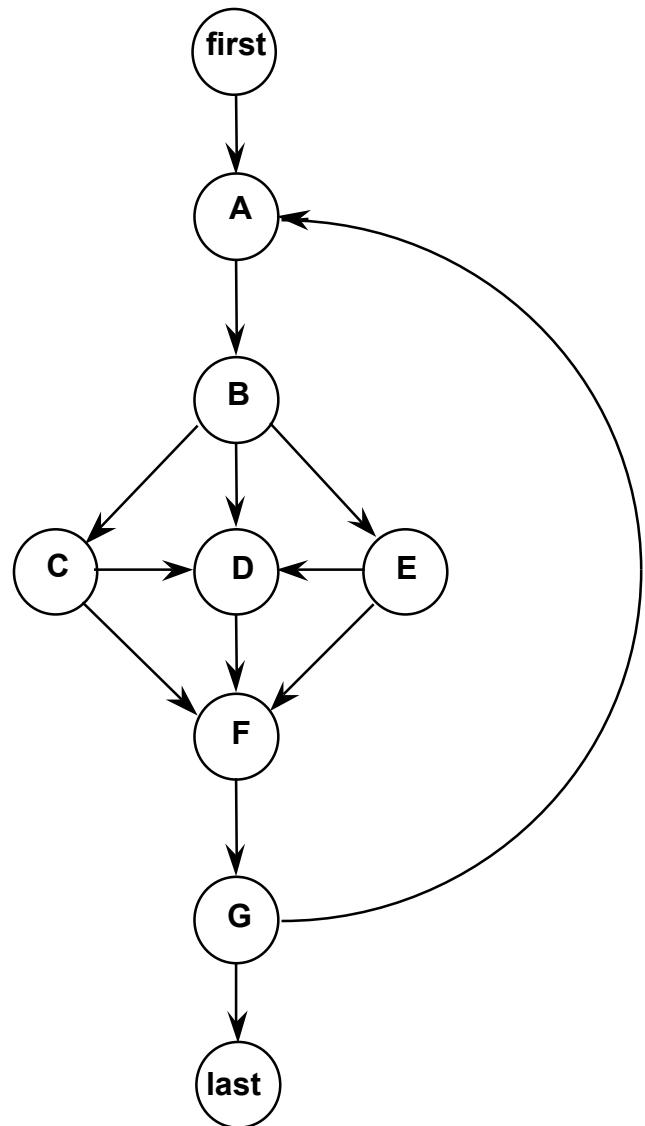
BUT

- What who would ever test all of these?
- We will have an elegant, mathematically sensible alternative

Stephen R. Schach, *Software Engineering*, (2nd edition) Richard D. Irwin, Inc. and Aksen Associates, Inc. 1993 (also in all later editions!)



# Test Cases for Schach's “Program”



1. First-A-B-C-F-G-Last
2. First-A-B-C-D-F-G-Last
3. First-A-B-D-F-G-A-B-D-F-G-Last
4. First-A-B-E-F-G-Last
5. First-A-B-E-D-F-G-Last

**These test cases cover**

- Every node
- Every edge
- Normal repeat of the loop
- Exiting the loop



# Program Graph Definitions

**Given a program written in an imperative programming language, its *program graph* is a directed graph in which...**

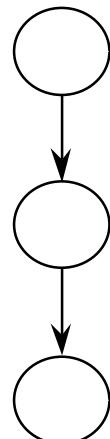
(Traditional Definition) nodes are program statements, and edges represent flow of control (there is an edge from node i to node j iff the statement corresponding to node j can be executed immediately after the statement corresponding to node i).

**(improved definition) nodes are either entire statements or fragments of a statement, and edges represent flow of control (there is an edge from node i to node j iff the statement (fragment) corresponding to node j can be executed immediately after the statement or statement fragment corresponding to node i).**

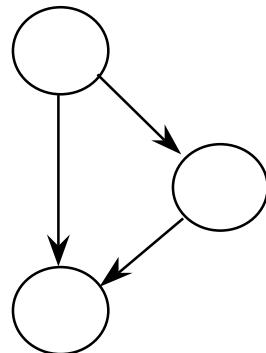


# Program Graphs of Structured Programming Constructs

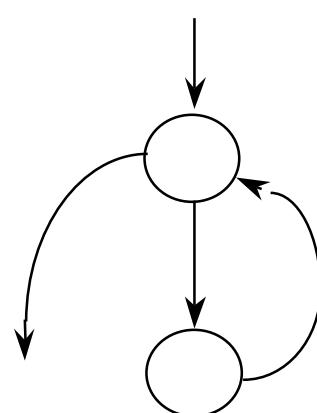
**Sequence**



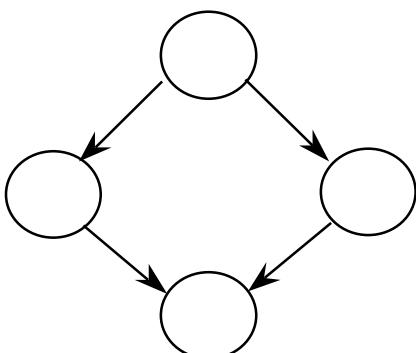
**If-Then**



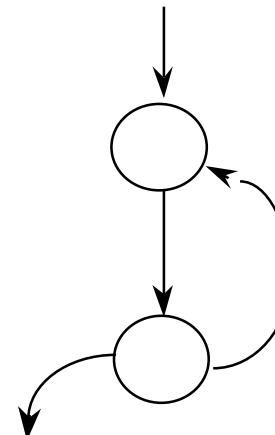
**Pre-test Loop**



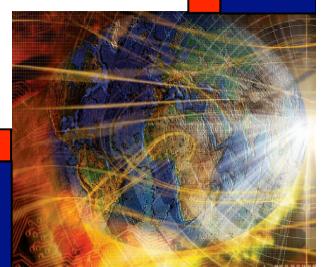
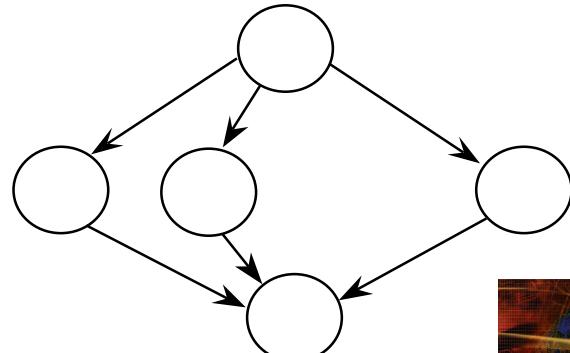
**If-Then-Else**



**Post-test Loop**

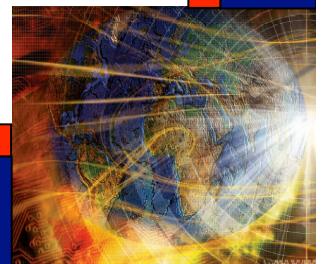


**Case**



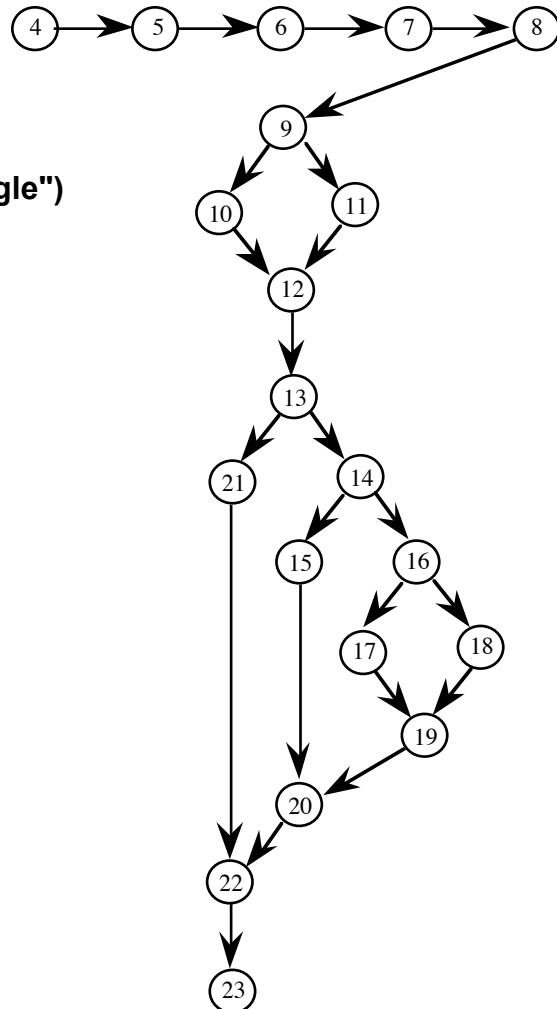
# Triangle Program Specification

- **Inputs:**  $a$ ,  $b$ , and  $c$  are non-negative integers, taken to be sides of a triangle
- **Output:** type of triangle formed by  $a$ ,  $b$ , and  $c$ 
  - Not a triangle
  - Scalene (no equal sides)
  - Isosceles (exactly 2 sides equal)
  - Equilateral (3 sides equal)
- **To be a triangle,  $a$ ,  $b$ , and  $c$  must satisfy the triangle inequalities:**
  - $a < b + c$ ,
  - $b < a + c$ , and
  - $c < a + b$



# Sample Program Graph

```
1. Program triangle
2. Dim a,b,c As Integer
3. Dim IsATriangle As Boolean
4. Output("Enter 3 integers which are sides of a triangle")
5. Input(a,b,c)
6. Output("Side A is ",a)
7. Output("Side B is ",b)
8. Output("Side C is ",c)
9. If (a < b + c) AND (b < a + c) AND (c < a + b)
10. Then IsATriangle = True
11. Else IsATriangle = False
12 EndIf
13 If IsATriangle
14. Then If (a = b) AND (b = c)
15. Then Output ("Equilateral")
16. Else If (a ≠ b) AND (a ≠ c) AND (b ≠ c)
17. Then Output ("Scalene")
18. Else Output ("Isosceles")
19. EndIf
20. EndIf
21. Else Output("Not a Triangle")
22. EndIf
23. End triangle2
```

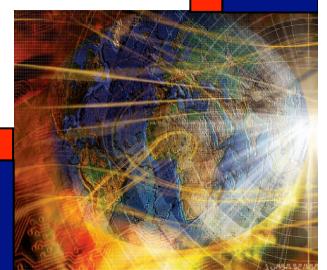
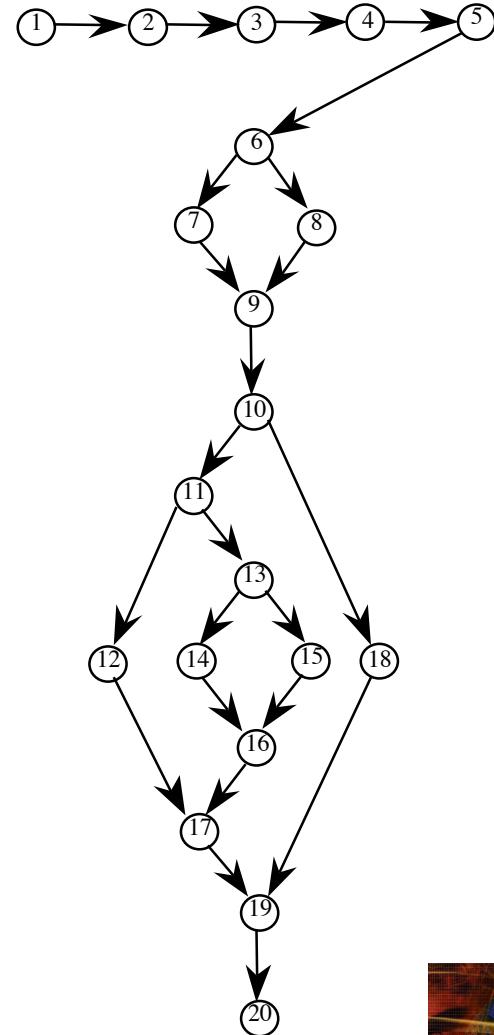


# Trace Code for $a = 5, b = 5, c = 5$

```

Program Triangle
Dim a, b, c As Integer
Dim IsATriangle As Boolean
'Step 1: Get Input
1. Output ("Enter 3 integers which are sides
   of a triangle")
2. Input (a,b,c)
3. Output ("Side A is ",a)
4. Output ("Side B is ",b)
5. Output ("Side C is ",c)
'Step 2: Is A Triangle?
6. If (a < b + c) AND (b < a + c) AND (c < a +
   b)
7. Then IsATriangle = True
8. Else IsATriangle = False
9. Endif
'Step 3: Determine Triangle Type
10. If IsATriangle
11. Then If (a = b) AND (b = c)
12.           Then Output
13.             ("Equilateral")
14.           Else If (a ≠ b) AND (a ≠
   c) AND (b ≠ c)
15.             Then Output ("Scalene")
16.             Else Output ("Isosceles")
17.           Endif
18. Else Output ("Not a triangle")
19. Endif

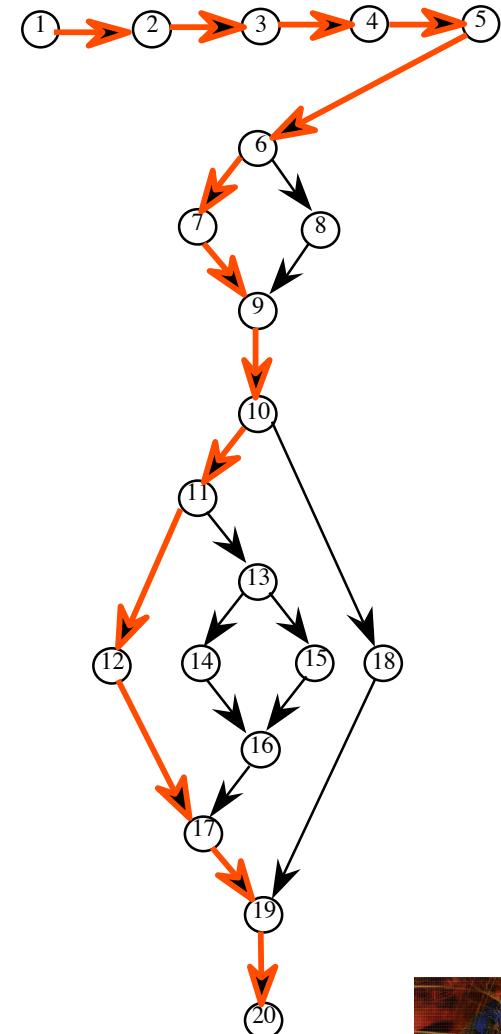
```



# Trace Code for a = 5, b = 5, c = 5

```

Program Triangle
Dim a, b, c As Integer
Dim IsATriangle As Boolean
'Step 1: Get Input
1. Output ("Enter 3 integers which are sides
   of a triangle")
2. Input (a,b,c)
3. Output ("Side A is ",a)
4. Output ("Side B is ",b)
5. Output ("Side C is ",c)
'Step 2: Is A Triangle?
6. If (a < b + c) AND (b < a + c) AND (c < a +
   b)
7. Then IsATriangle = True
8. Else IsATriangle = False
9. Endif
'Step 3: Determine Triangle Type
10. If IsATriangle
11. Then If (a = b) AND (b = c)
12.      Then Output
           ("Equilateral")
13.      Else If (a ≠ b) AND (a ≠
           c) AND (b ≠ c)
14.          Then
15.             Output ("Scalene")
16.             Output ("Isosceles")
17.         Endif
18.     Endif
19. Endif
Software Testing Approach (3rd Edition)
  
```

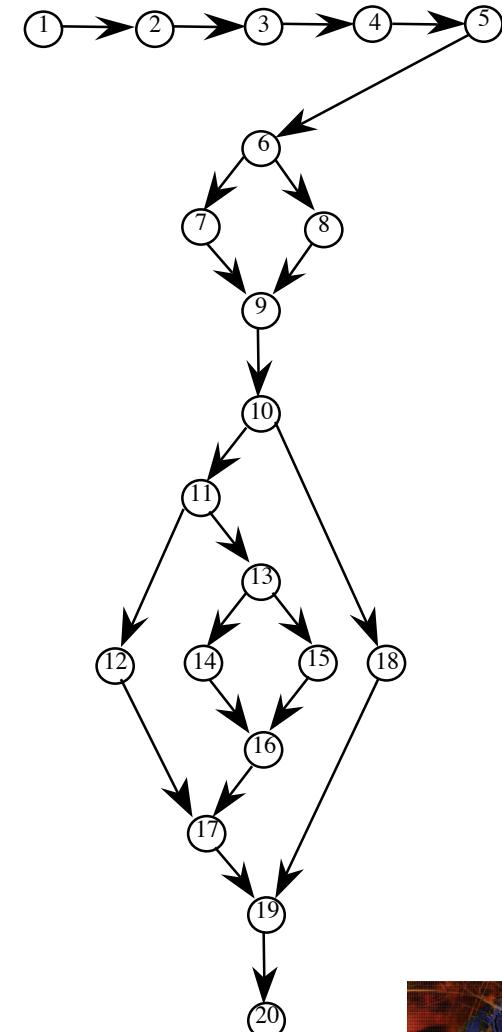


# Trace Code for $a = 2, b = 5, c = 5$

```

Program Triangle
Dim a, b, c As Integer
Dim IsATriangle As Boolean
'Step 1: Get Input
1. Output ("Enter 3 integers which are sides
   of a triangle")
2. Input (a,b,c)
3. Output ("Side A is ",a)
4. Output ("Side B is ",b)
5. Output ("Side C is ",c)
'Step 2: Is A Triangle?
6. If (a < b + c) AND (b < a + c) AND (c < a
   + b)
7. Then IsATriangle = True
8. Else IsATriangle = False
9. Endif
'Step 3: Determine Triangle Type
10. If IsATriangle
11. Then If (a = b) AND (b = c)
12.      Then Output
           ("Equilateral")
13.      Else If (a ≠ b) AND (a ≠
           c) AND (b ≠ c)
14.          Then
15.             Output ("Scalene")
16.             Output ("Isosceles")
17.         Endif
18.     Endif
19. Endif

```

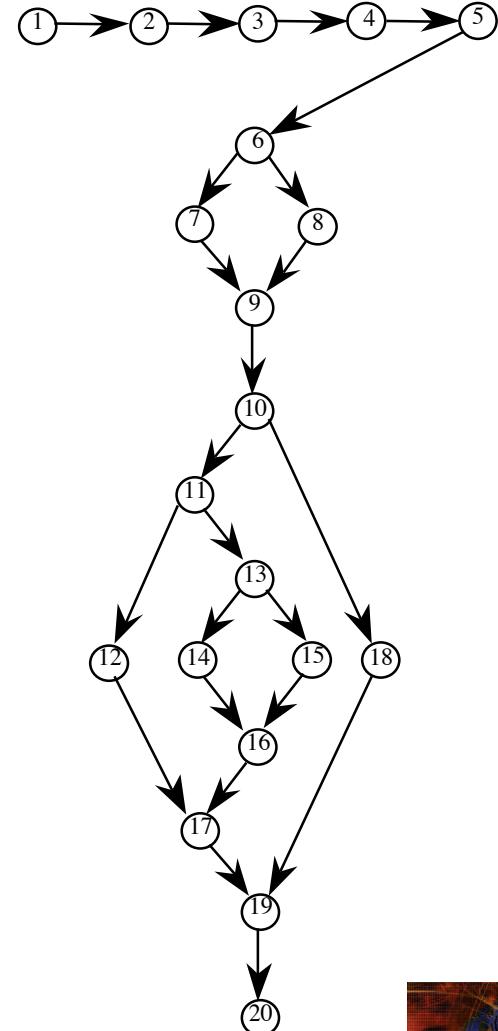


# Trace Code for $a = 3, b = 4, c = 5$

```

Program triangle
Dim a, b, c As Integer
Dim IsATriangle As Boolean
'Step 1: Get Input
1. Output ("Enter 3 integers which are sides
   of a triangle")
2. Input (a,b,c)
3. Output ("Side A is ",a)
4. Output ("Side B is ",b)
5. Output ("Side C is ",c)
'Step 2: Is A Triangle?
6. If (a < b + c) AND (b < a + c) AND (c < a
   + b)
7. Then IsATriangle = True
8. Else IsATriangle = False
9. Endif
'Step 3: Determine Triangle Type
10. If IsATriangle
11. Then IF (a = b) AND (b = c)
12.      Then Output
           ("Equilateral")
13.      Else If (a ≠ b) AND (a ≠
           c) AND (b ≠ c)
14.          Then
15.             Output ("Scalene")
16.             Output ("Isosceles")
17.         Endif
18.     Else Output ("Not a Triangle")
19. Endif

```

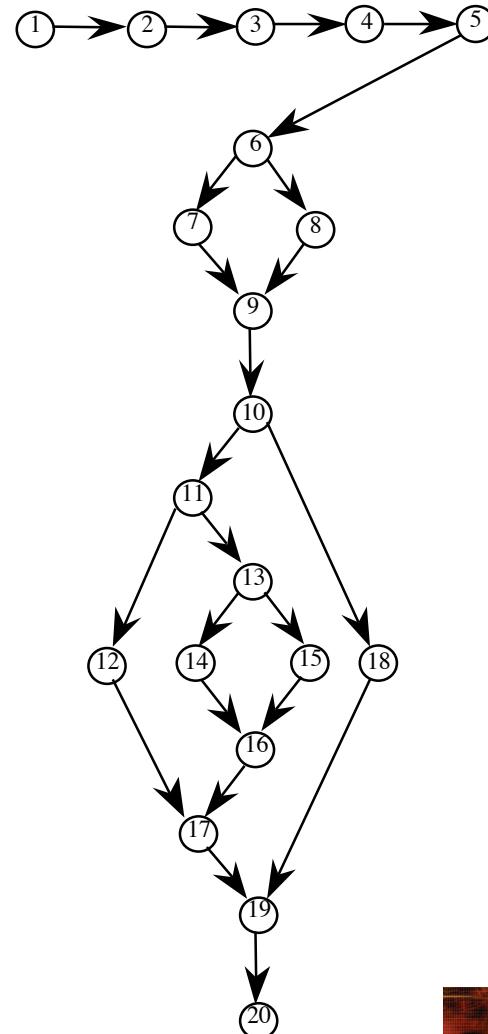


# Trace Code for $a = 2, b = 3, c = 7$

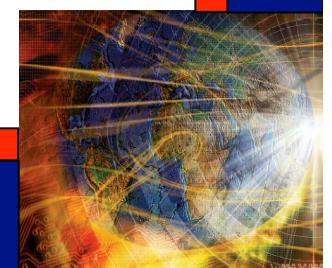
```

Program triangle
Dim a, b, c As Integer
Dim IsATriangle As Boolean
'Step 1: Get Input
1. Output ("Enter 3 integers which are sides
   of a triangle")
2. Input (a,b,c)
3. Output ("Side A is ",a)
4. Output ("Side B is ",b)
5. Output ("Side C is ",c)
'Step 2: Is A Triangle?
6. If (a < b + c) AND (b < a + c) AND (c < a
   + b)
7. Then IsATriangle = True
8. Else IsATriangle = False
9. Endif
'Step 3: Determine Triangle Type
10. If IsATriangle
11. Then IF (a = b) AND (b = c)
12.      Then Output
           ("Equilateral")
13.      Else If (a ≠ b) AND (a ≠
           c) AND (b ≠ c)
14.          Then
15.             Output ("Scalene")
16.             Output ("Isosceles")
17.         Endif
18.     Else Output ("Not a Triangle")
19. Endif
20. End Program triangle

```

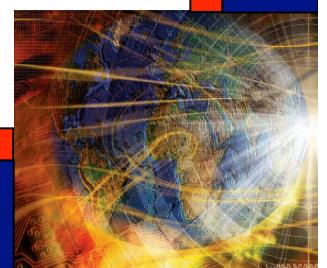
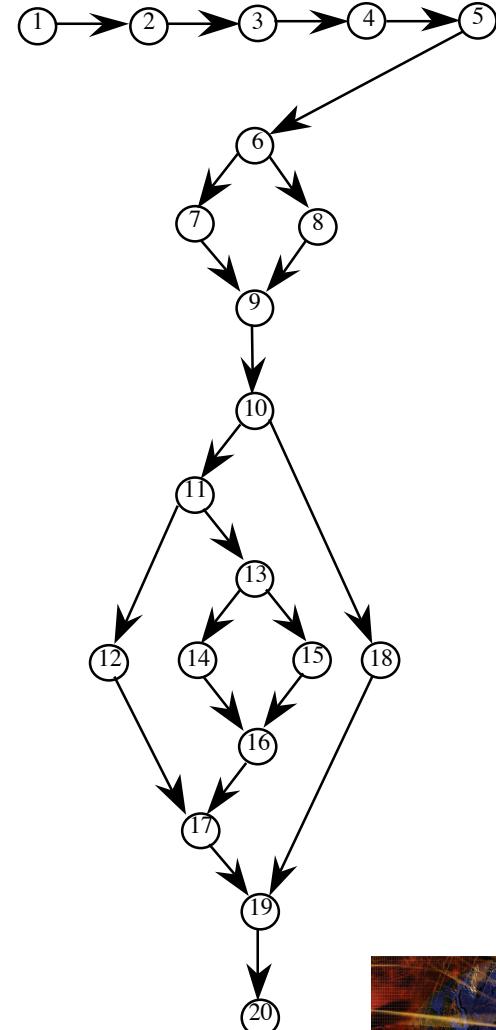


Path Testing I

*Software Testing: A Craftsman's Approach, 3<sup>rd</sup> Edition*

# Can you find values of a, b, and c such that the path traverses nodes 7 and 18?

```
Program triangle
Dim a, b, c As Integer
Dim IsATriangle As Boolean
'Step 1: Get Input
1.   Output ("Enter 3 integers which are sides
      of a triangle")
2.   Input (a,b,c)
3.   Output ("Side A is ",a)
4.   Output ("Side B is ",b)
5.   Output ("Side C is ",c)
'Step 2: Is A Triangle?
6.   If (a < b + c) AND (b < a + c) AND (c < a +
      b)
7.     Then IsATriangle = True
8.   Else IsATriangle = False
9.   Endif
'Step 3: Determine Triangle Type
10.  If IsATriangle
11.    Then IF (a = b) AND (b = c)
12.      Then Output
13.        ("Equilateral")
14.        Else If (a ≠ b) AND (a ≠
      c) AND (b ≠ c)
15.          Output ("Scalene")
16.          Output ("Isosceles")
17.        Endif
18.      Else Output ("Not a Triangle")
```



## DD-Paths

A DD-Path (decision-to-decision) is a chain in a program graph such that

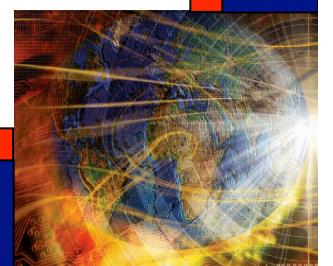
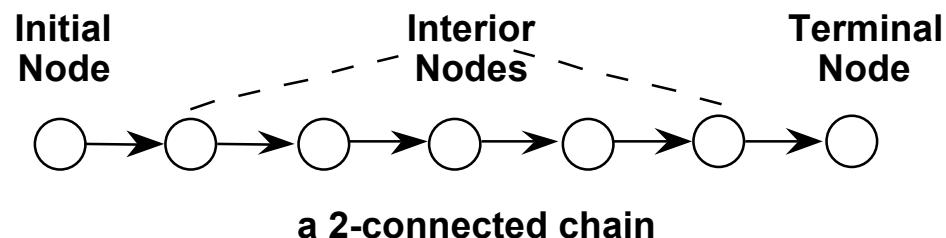
**Case 1:** it consists of a single node with indegree = 0, or

**Case 2:** it consists of a single node with outdegree = 0, or

**Case 3:** it consists of a single node with indegree  $\geq 2$  or outdegree  $\geq 2$ , or

**Case 4:** it consists of a single node with indegree = 1 and outdegree = 1, or

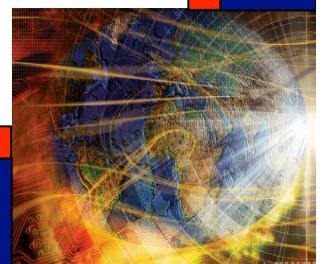
**Case 5:** it is a maximal chain of length  $\geq 1$ .



# DD-Path Graph

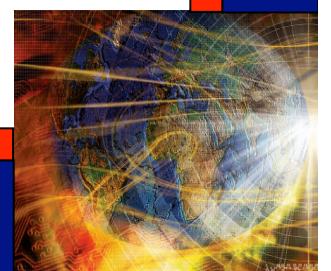
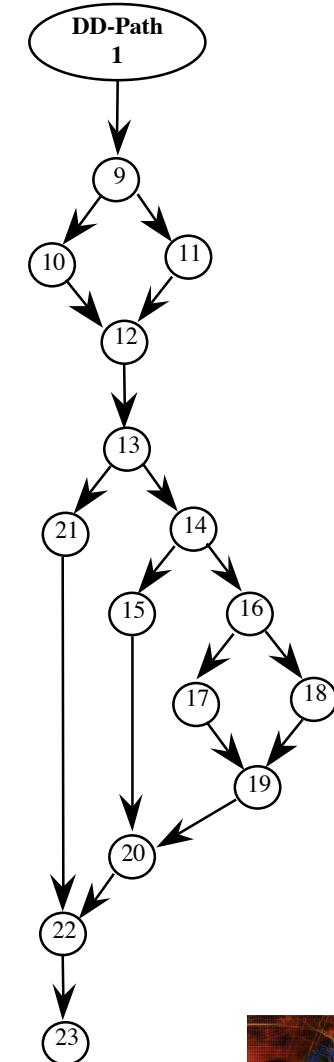
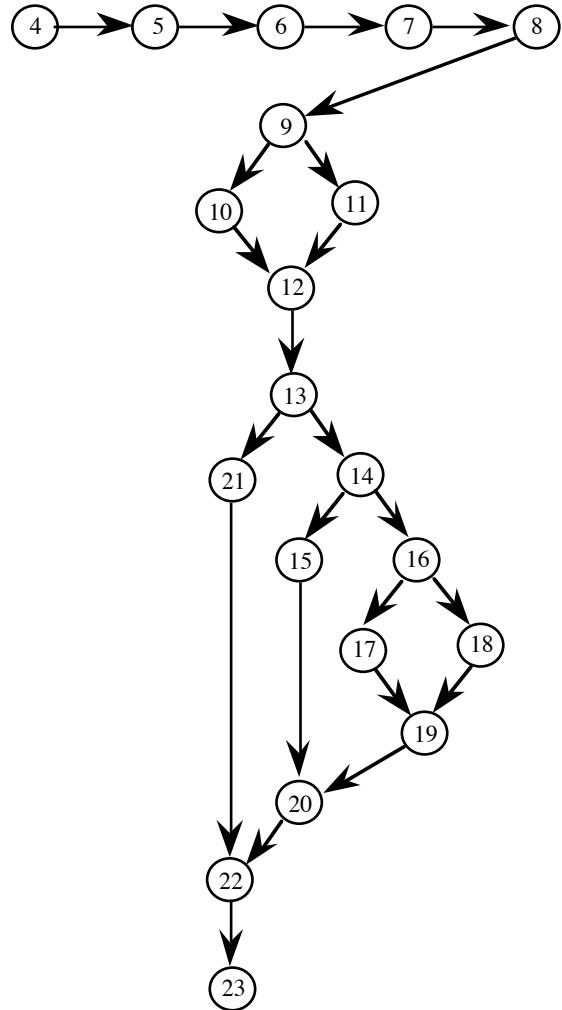
Given a program written in an imperative language, its *DD-Path graph* is the directed graph in which nodes are DD-Paths of its program graph, and edges represent control flow between successor DD-Paths.

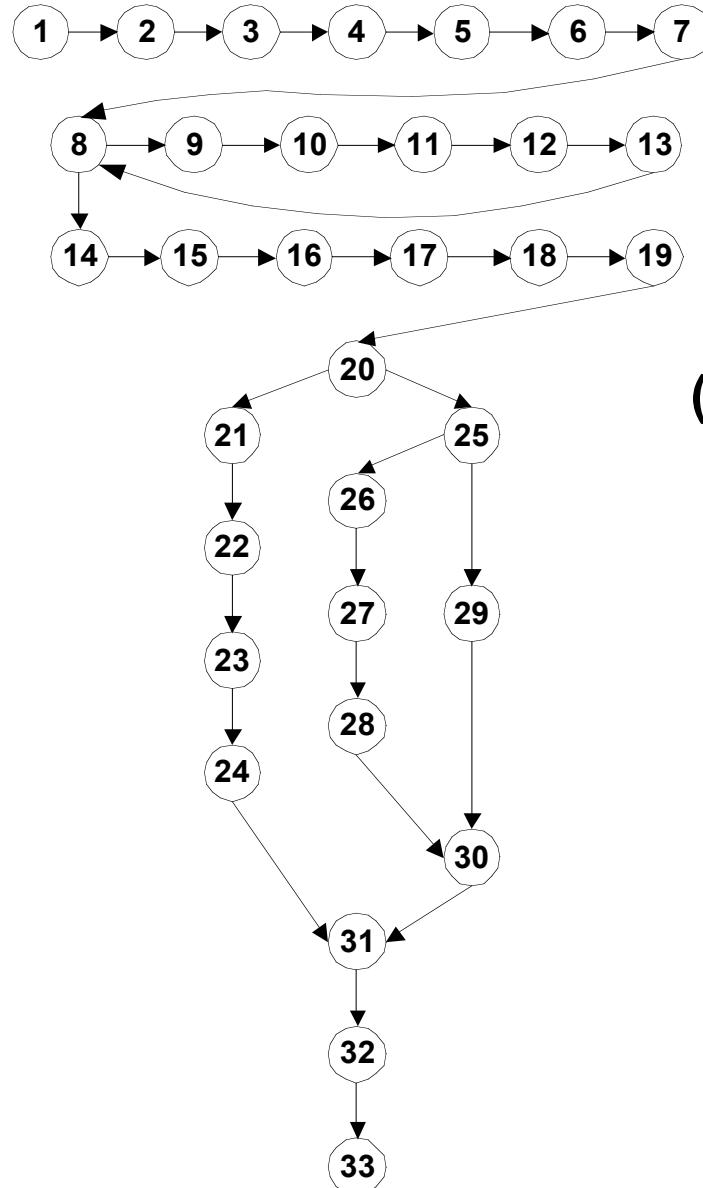
- a form of condensation graph
- 2-connected components are collapsed into an individual node
- single node DD-Paths (corresponding to Cases 1 - 4 ) preserve the convention that a statement fragment is in exactly one DD-Path



# DD-Path Graph

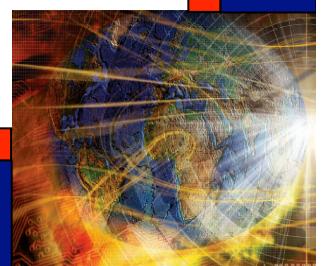
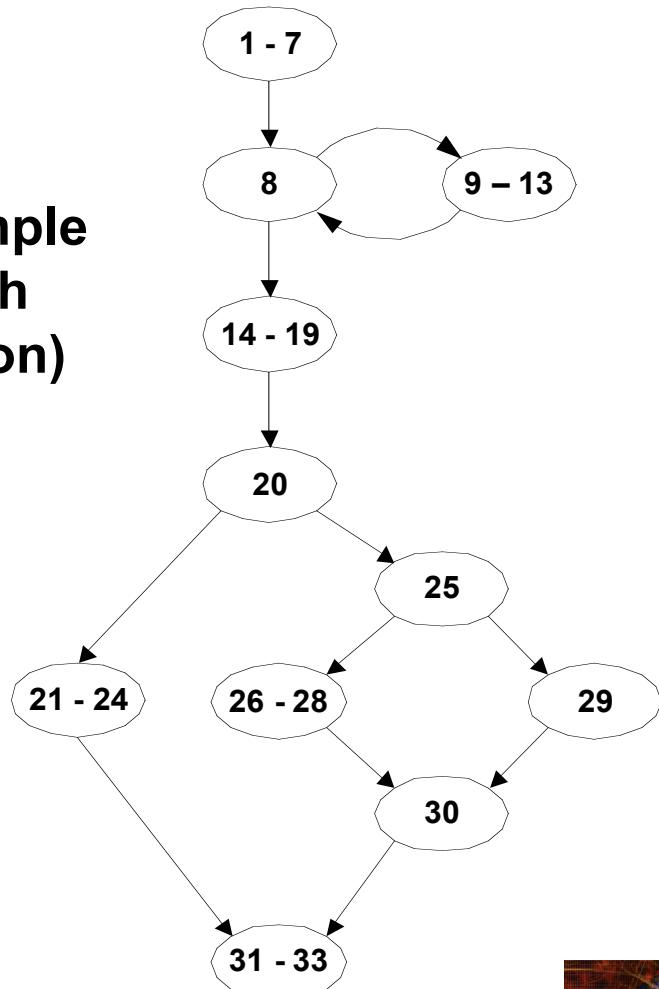
(not much compression because this example is control intensive, with little sequential code.)





## DD-Path Graph of Commission Problem

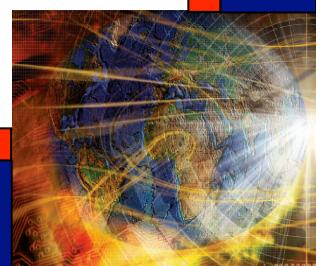
(better example  
of DD-Path  
compression)



# Structural Test Coverage Metrics

(E. F. Miller, 1977 dissertation)

- $C_0$ : Every statement
- $C_1$ : Every DD-Path
- $C_{1p}$ : Every predicate outcome
- $C_2$ :  $C_1$  coverage + loop coverage
- $C_d$ :  $C_1$  coverage + every pair of dependent DD-Paths
- $C_{MCC}$ : Multiple condition coverage
- $C_{ik}$ : Every program path that contains up to k repetitions of a loop (usually k = 2)
- $C_{stat}$ : "Statistically significant" fraction of paths
- $C_\infty$ : All possible execution paths



# Graph-Based Coverage Metrics

1. Every node
2. Every edge
3. Successive pairs of edges
4. Every path

**Exercise:** How do these compare with structural test coverage metrics?



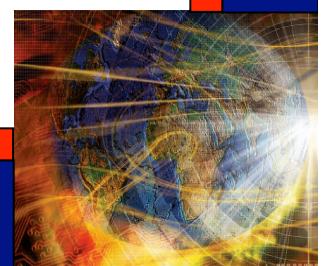
# Testing Loops

**Huang's Theorem: (Paraphrased)**

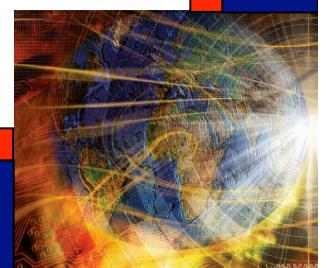
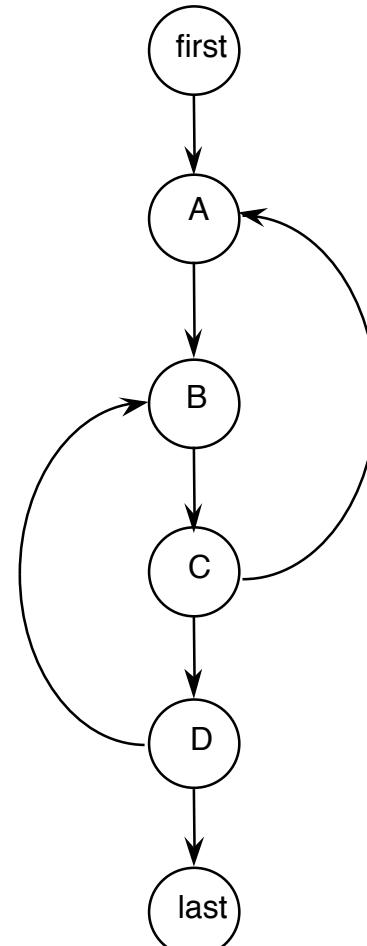
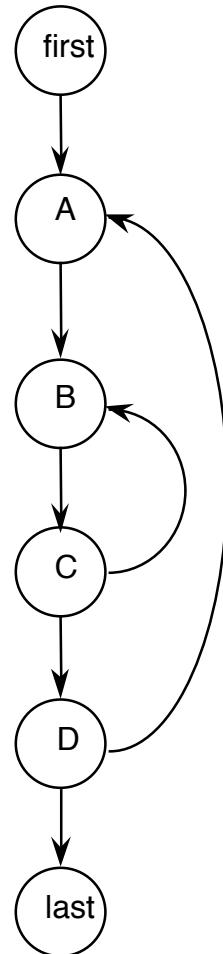
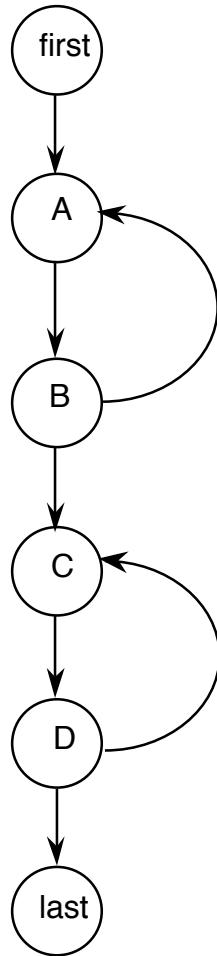
**Everything interesting will happen in two loop traversals: the normal loop traversal and the exit from the loop.**

**Exercise:**

**Discuss Huang's Theorem in terms of graph based coverage metrics.**

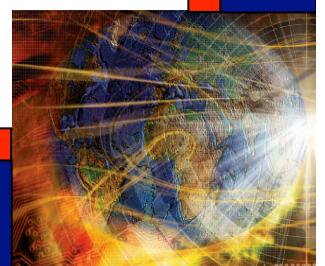


# Concatenated, Nested, and Knotted Loops



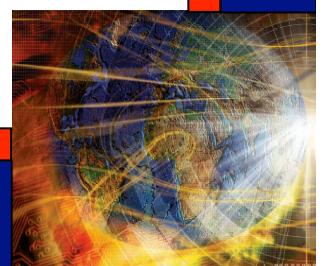
# Strategy for Loop Testing

- **Huang's theorem suggests/assures 2 tests per loop is sufficient.** (Judgment required, based on reality of the code.)
- **For nested loops:**
  - Test innermost loop first
  - Then “condense” the loop into a single node (as in condensation graph, see Chapter 4)
  - Work from innermost to outermost loop
- **For concatenated loops:** use Huang's Theorem
- **For knotted loops:** Rewrite! (see McCabe's cyclomatic complexity)



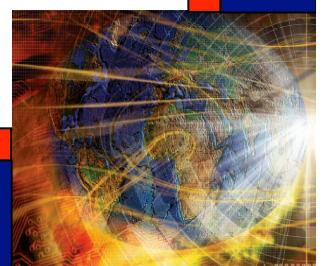
# Multiple Condition Testing

- Consider the multiple condition as a logical proposition, i.e., some logical expression of simple conditions.
- Make the truth table of the logical expression.
- Convert the truth table to a decision table.
- Develop test cases for each rule of the decision table (except the impossible rules, if any).
- Next 3 slides: multiple condition testing for  
**If (a < b + c) AND (b < a + c) AND (c < a + b)  
Then IsATriangle = True  
Else IsATriangle = False  
Endif**



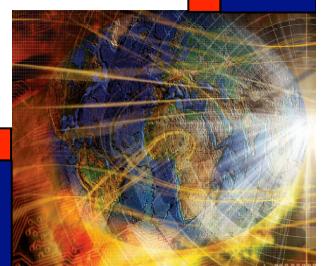
# Truth Table for ( $a < b+c$ ) AND ( $b < a+c$ ) AND ( $c < a+b$ )

$(a < b+c)$	$(b < a+c)$	$(c < a+b)$	$(a < b+c) \text{ AND } (b < a+c) \text{ AND } (c < a+b)$
T	T	T	T
T	T	F	F
T	F	T	F
T	F	F	F
F	T	T	F
F	T	F	F
F	F	T	F
F	F	F	F



## Decision Table for ( $a < b+c$ ) AND ( $b < a+c$ ) AND ( $c < a+b$ )

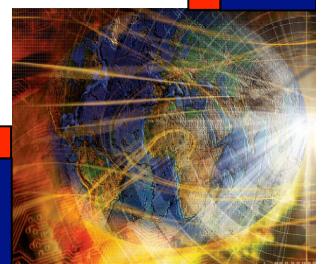
$c_1: a < b+c$	T	T	T	T	F	F	F	F
$c_2: b < a+c$	T	T	F	F	T	T	F	F
$c_3: c < a+b$	T	F	T	F	T	F	T	F
$a_1:$ <b>impossible</b>				X		X	X	X
$a_2:$ Valid test case #	1	2	3		4			



# Multiple Condition Test Cases for $(a < b+c)$ AND $(b < a+c)$ AND $(c < a+b)$

Test Case		a	b	c	expected output
1	all true	3	4	5	TRUE
2	$c \geq a + b$	3	4	9	FALSE
3	$b \geq a + c$	3	9	4	FALSE
4	$a \geq b + c$	9	3	4	FALSE

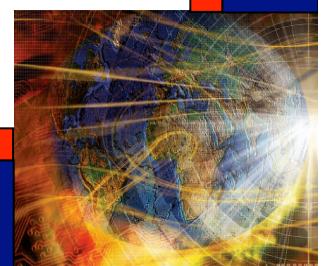
**Note:** could add test cases for  $c = a + b$ ,  
 $b = a + c$ , and  $a = b + c$ .



## Exercise: What test cases are needed for this code fragment ?

13. If ( $a \neq b$ ) AND ( $a \neq c$ ) AND ( $b \neq c$ )
14. Then Output (“Scalene”)
15. Else Output (“Isosceles”)
16. Endif

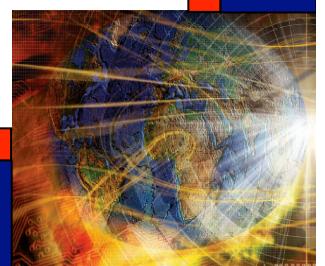
a	b	c	Expected Output
---	---	---	-----------------



# Dependent DD-Paths

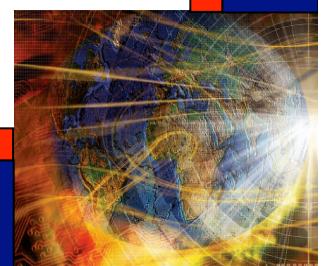
## (often correspond to infeasible paths)

- Look at the Triangle Program code in slide 8
- And the program graph and DD-Path graph in slide 11
- If a path traverses node 10 (Then IsATriangle = True), then it must traverse node 14.
- Similarly, if a path traverses node 11 (Else IsATriangle = False), then it must traverse node 21.
- Paths through nodes 10 and 21 are infeasible.
- Similarly for paths through 11 and 14.
- Hence the need for the  $C_d$  coverage metric.



# Code-Based Testing Strategy

- Start with a set of test cases generated by an “appropriate” (depends on the nature of the program) specification-based test method.
- Look at code to determine appropriate test coverage metric.
  - Loops?
  - Compound conditions?
  - Dependencies?
- If appropriate coverage is attained, fine.
- Otherwise, add test cases to attain the intended test coverage.



# Test Coverage Tools

- **Commercial test coverage tools use “instrumented” source code.**
  - New code added to the code being tested
  - Designed to “observe” a level of test coverage
- **When a set of test cases is run on the instrumented code, the designed test coverage is ascertained.**
- **Strictly speaking, running test cases in instrumented code is not sufficient**
  - Safety critical applications require tests to be run on actual (delivered, non-instrumented) code.
  - Usually addressed by mandated testing standards.

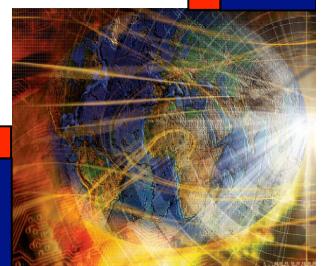


# Sample DD-Path Instrumentation

(values of array DDpathTraversed are set to 1  
when corresponding instrumented code is  
executed.)

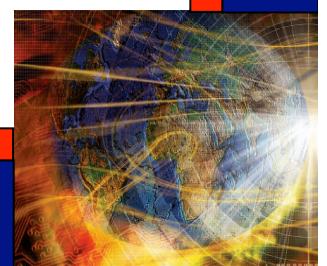
**DDpathTraversed(1) = 1**

4. Output("Enter 3 integers which are sides of a triangle")
5. Input(a,b,c)
6. Output("Side A is ",a)
7. Output("Side B is ",b)
8. Output("Side C is ",c)
- 'Step 2: Is A Triangle?  
**DDpathTraversed(2) = 1**
9. If (a < b + c) AND (b < a + c) AND (c < a + b)
10. Then **DDpathTraversed(3) = 1**  
IsATriangle = True
11. Else **DDpathTraversed(4) = 1**  
IsATriangle = False
- 12 EndIf



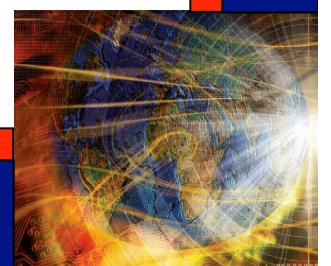
# Instrumentation Exercise:

**How could you instrument the Triangle Program to record how many times a set of test cases traverses the individual DD-Paths?**



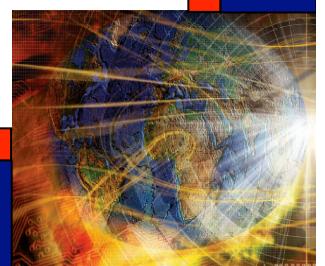
# Chapter 9

## Path Testing–Part 2



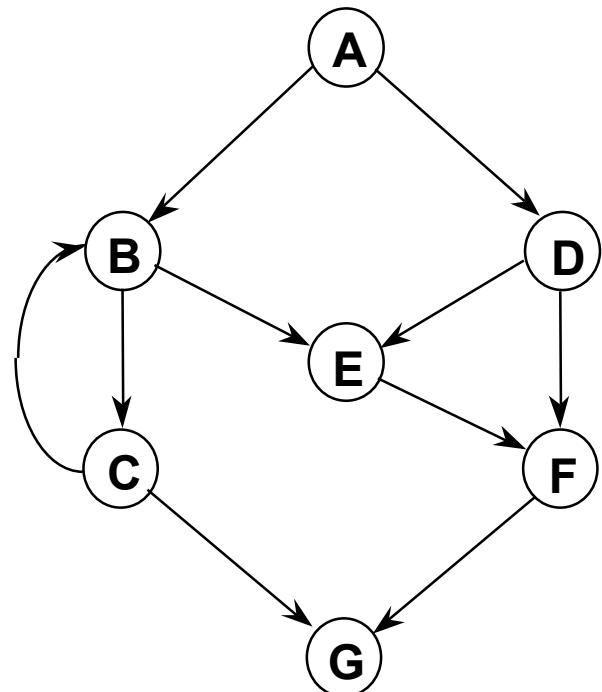
# (McCabe) Basis Path Testing

- in math, a basis "spans" an entire space, such that everything in the space can be derived from the basis elements.
- the cyclomatic number of a strongly connected directed graph is the number of linearly independent cycles.
- given a program graph, we can always add an edge from the sink node to the source node to create a strongly connected graph. (assuming single entry, single exit)
- computing  $V(G) = e - n + p$  from the modified program graph yields the number of independent paths that must be tested.
- since all other program execution paths are linear combinations of the basis path, it is necessary to test the basis paths. (Some say this is sufficient; but that is problematic.)
- the next few slides follow McCabe's original example.



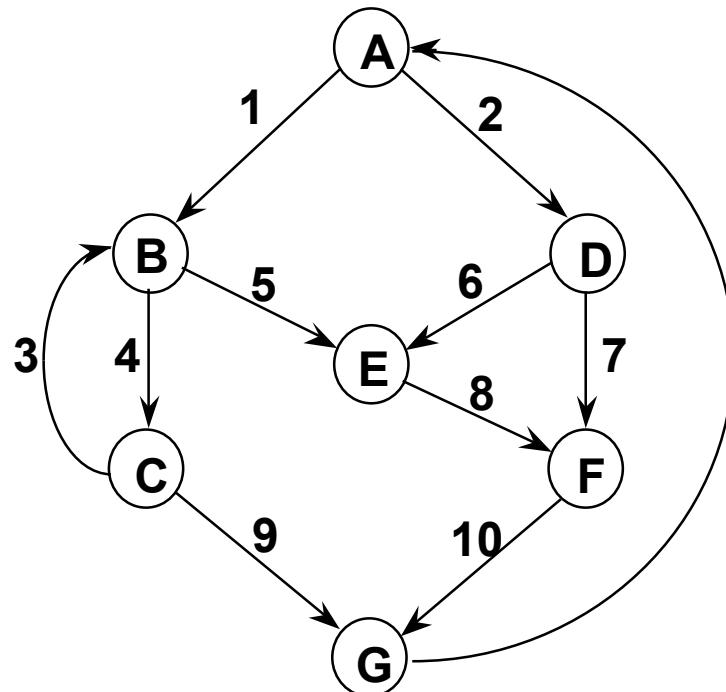
# McCabe's Example

## McCabe's Original Graph

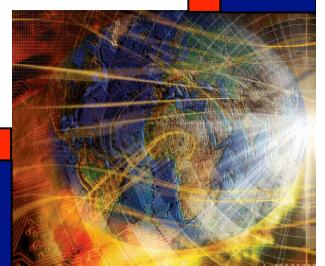


$$\begin{aligned}V(G) &= 10 - 7 + 2(1) \\&= 5\end{aligned}$$

## Derived, Strongly Connected Graph



$$\begin{aligned}V(G) &= 11 - 7 + 1 \\&= 5\end{aligned}$$

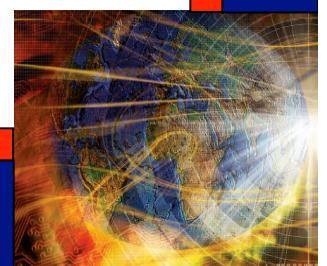


# McCabe's Baseline Method

To determine a set of basis paths,

1. Pick a "baseline" path that corresponds to normal execution. (The baseline should have as many decisions as possible.)
2. To get succeeding basis paths, retrace the baseline until you reach a decision node. "Flip" the decision (take another alternative) and continue as much of the baseline as possible.
3. Repeat this until all decisions have been flipped. When you reach  $V(G)$  basis paths, you're done.
4. If there aren't enough decisions in the first baseline path, find a second baseline and repeat steps 2 and 3.

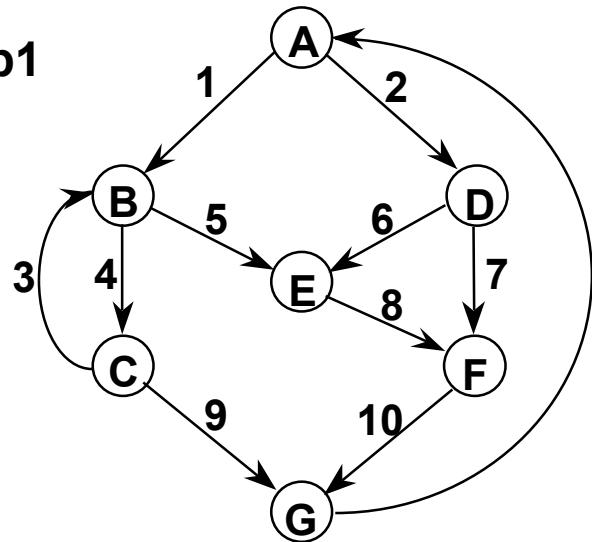
Following this algorithm, we get basis paths for McCabe's example.



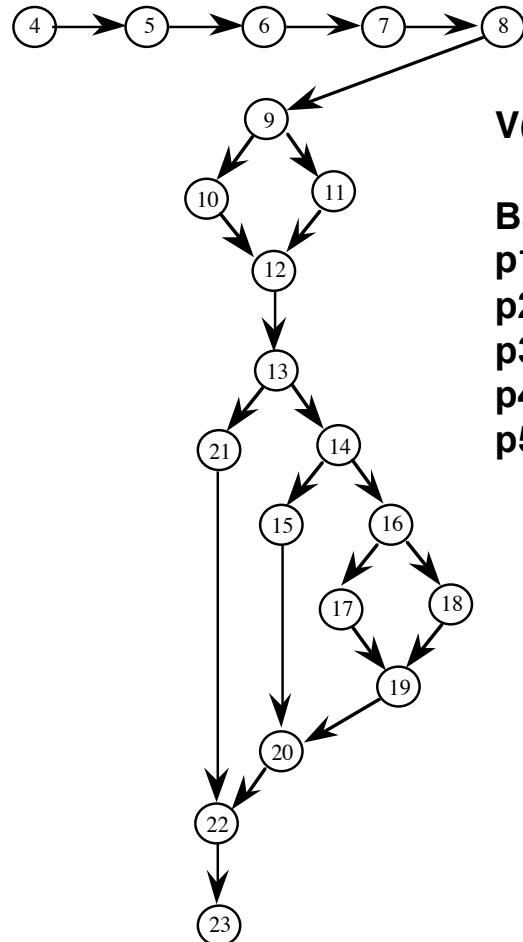
# Basis Paths

path \ edges traversed	1	2	3	4	5	6	7	8	9	10
p1: A, B, C, G	1	0	0	1	0	0	0	0	1	0
p2: A, B, C, B, C, G	1	0	1	2	0	0	0	0	1	0
p3: A, B, E, F, G	1	0	0	0	1	0	0	1	0	1
p4: A, D, E, F, G	0	1	0	0	0	1	0	1	0	1
p5: A, D, F, G	0	1	0	0	0	0	1	0	0	1
ex1: A, B, C, B, E, F, G	1	0	1	1	1	0	0	1	0	1
ex2: A, B, C, B, C, B, C, G	1	0	2	3	0	0	0	0	1	0

$$\begin{aligned} ex1 &= p2 + p3 - p1 \\ ex2 &= 2p2 - p1 \end{aligned}$$



# McCabe Basis Paths in the Triangle Program



$$V(G) = 23 - 20 + 2(1) = 5$$

Basis Path Set B1

- p1: 4, 5, 6, 7, 8, 9, 10, 12, 13, 14, 16, 18, 19, 20, 22, 23 (mainline)
- p2: 4, 5, 6, 7, 8, 9, 11, 12, 13, 14, 16, 18, 19, 20, 22, 23 (flipped at 9)
- p3: 4, 5, 6, 7, 8, 9, 11, 12, 13, 21, 22, 23 (flipped at 13)
- p4: 4, 5, 6, 7, 8, 9, 11, 12, 13, 14, 15, 20, 22, 23 (flipped at 14)
- p5: 4, 5, 6, 7, 8, 9, 11, 12, 13, 14, 16, 17, 19, 20, 22, 23 (flipped at 16)

**There are 8 topologically possible paths.  
4 are feasible, and 4 are infeasible.**

**Exercise: Is every basis path feasible?**

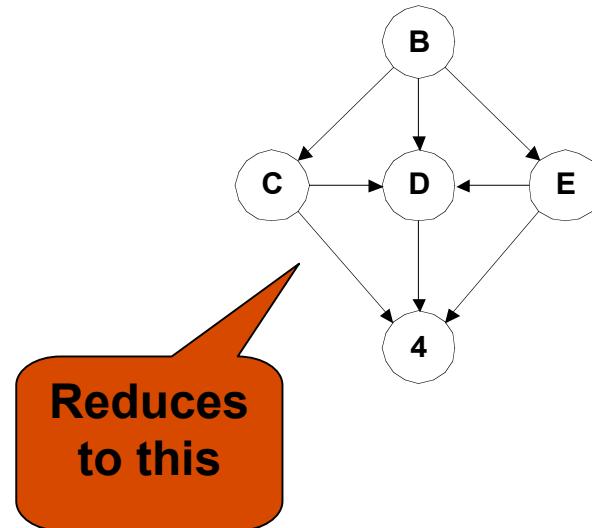
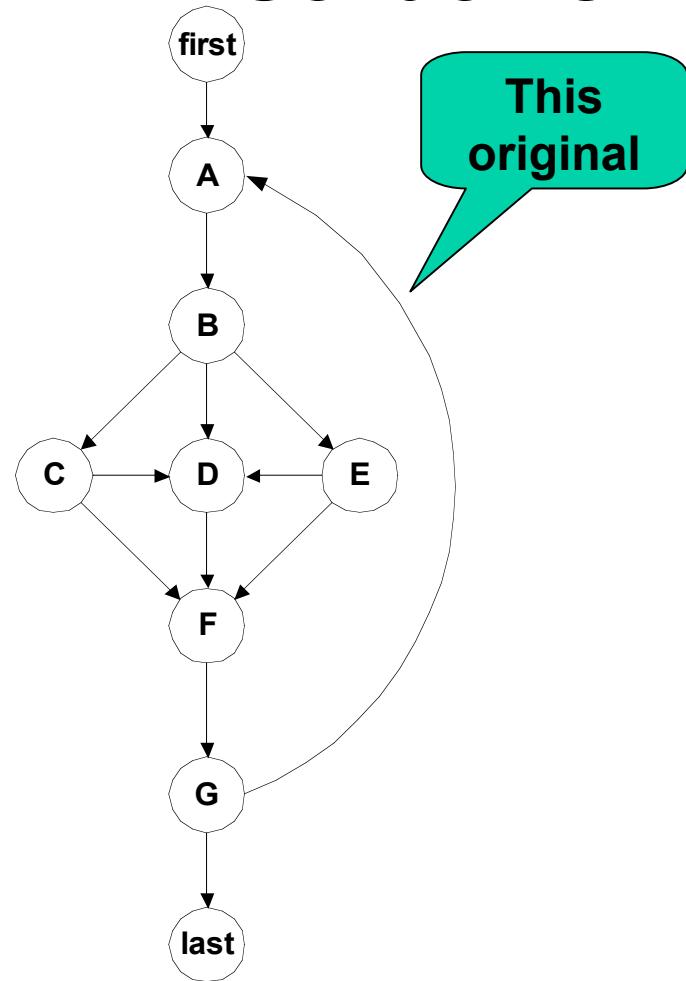


# Essential Complexity

- McCabe's notion of Essential Complexity deals with the extent to which a program violates the precepts of Structured Programming.
- To find Essential Complexity of a program graph,
  - Identify a group of source statements that corresponds to one of the basic Structured Programming constructs.
  - Condense that group of statements into a separate node (with a new name)
  - Continue until no more Structured Programming constructs can be found.
  - The Essential Complexity of the original program is the cyclomatic complexity of the resulting program graph.
- The essential complexity of a Structured Program is 1.
- Violations of the precepts of Structured Programming increase the essential complexity.

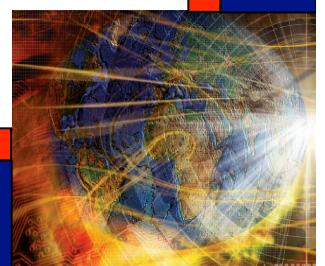


# Essential Complexity of Schach's Program Graph

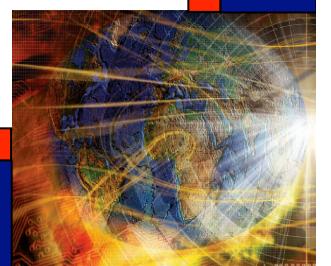
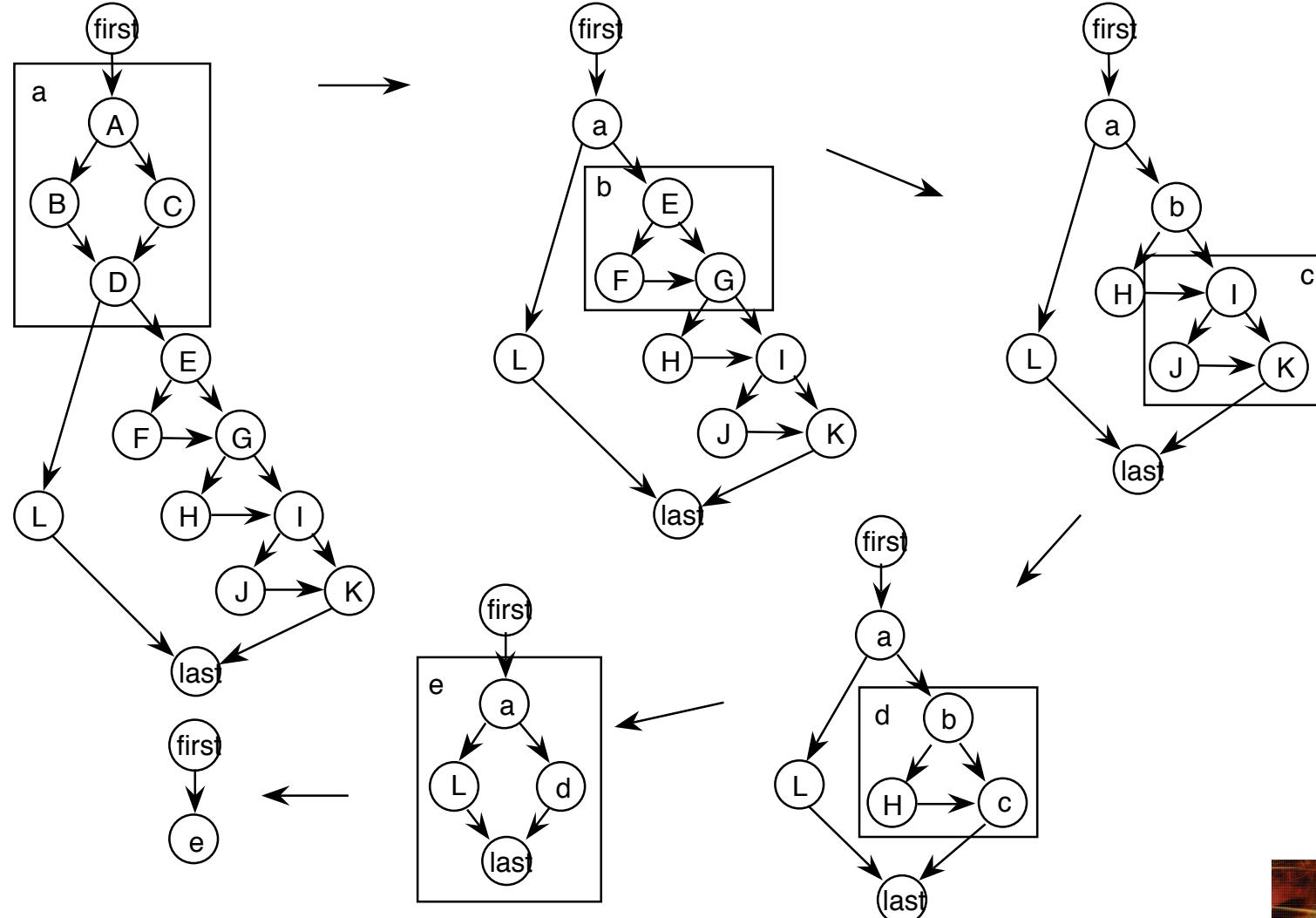


$$\begin{aligned}V(G) &= 8 - 5 + 2(1) \\&= 5\end{aligned}$$

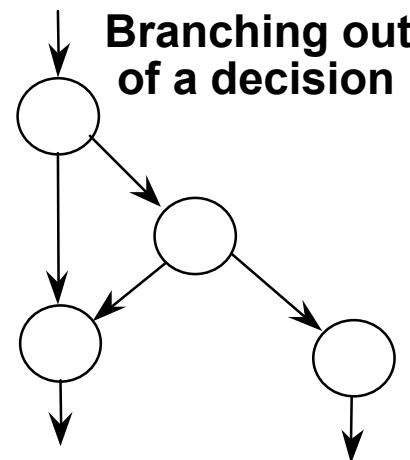
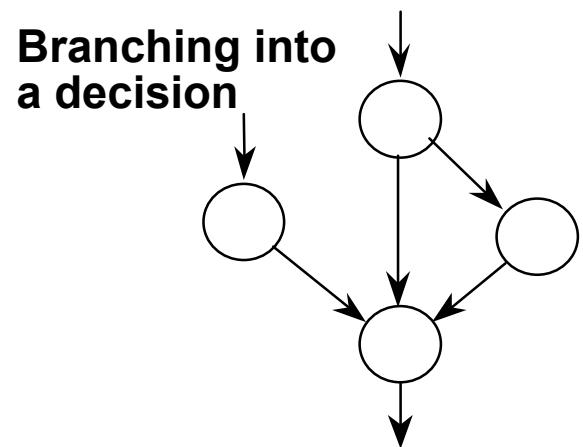
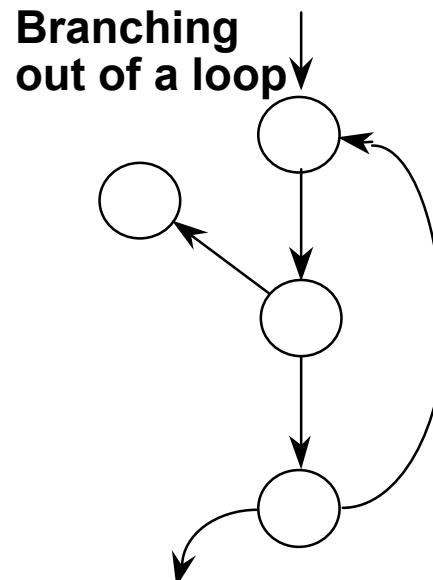
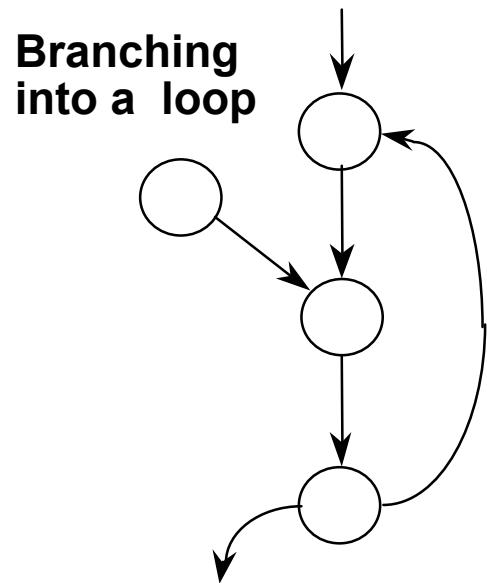
Essential complexity is 5



# Condensation with Structured Programming Constructs



## Violations of Structured Programming Precepts



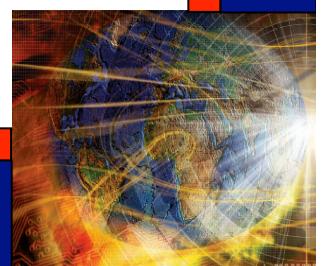
# Cons and Pros

- **Issues**

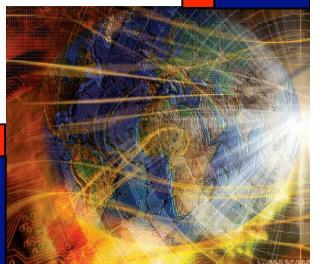
- Linear combinations of execution paths are counter-intuitive. What does  $2p_2 - p_1$  really mean?
- How does the baseline method guarantee feasible basis paths?
- Given a set of feasible basis paths, is this a sufficient test?

- **Advantages**

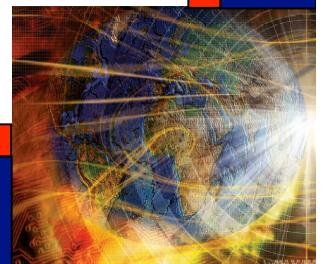
- McCabe's approach does address both gaps and redundancies.
- Essential complexity leads to better programming practices.
- McCabe proved that violations of the structured programming constructs increase cyclomatic complexity, and violations cannot occur singly.

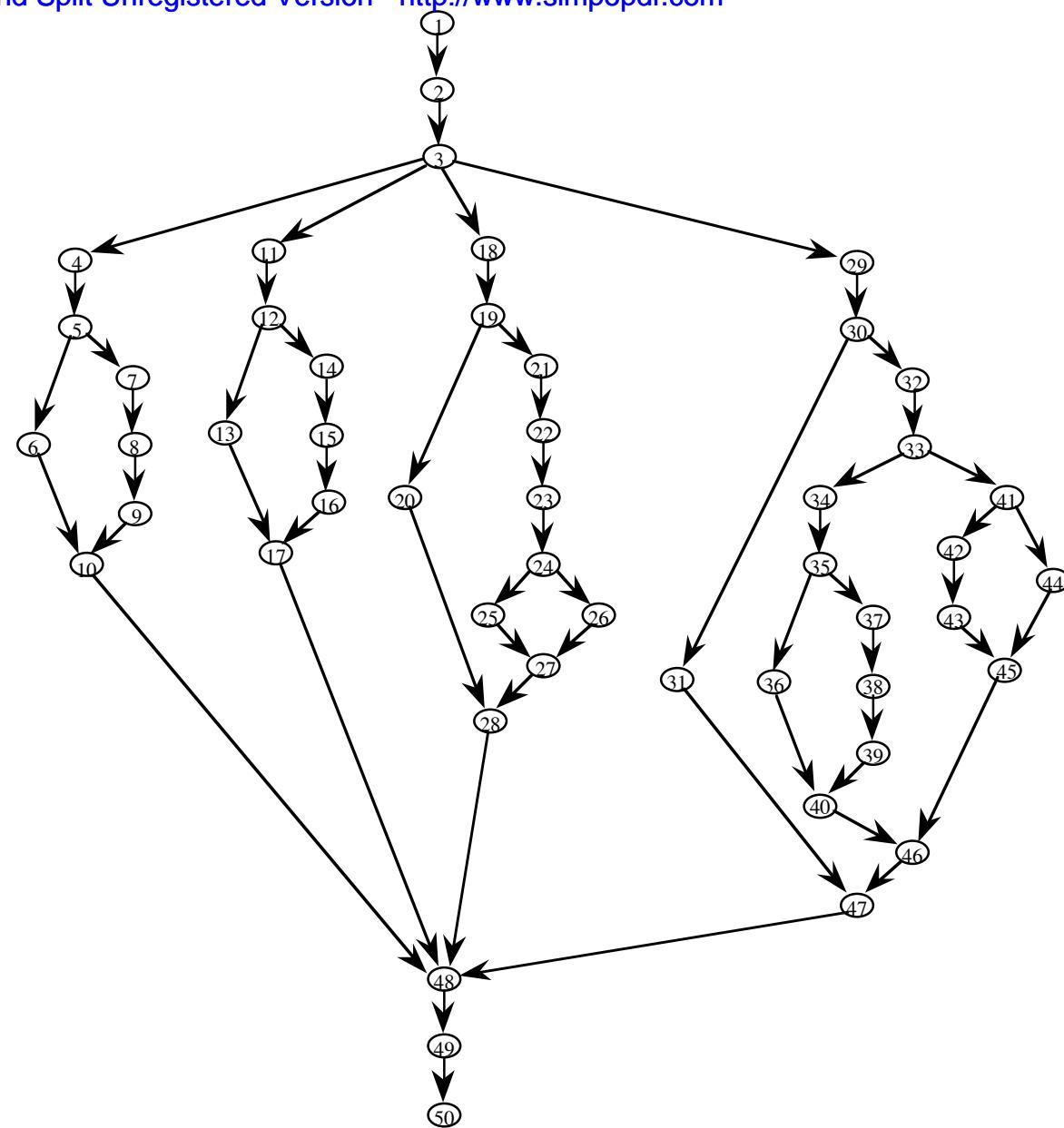


```
Program NextDate
Dim tomorrowDay,tomorrowMonth,tomorrowYear As Integer
Dim day,month,year As Integer
1. Output ("Enter today's date in the form MM DD YYYY")
2. Input (month,day,year)
3. Case month Of
4. Case 1: month Is 1,3,5,7,8,Or 10: '31 day months (except Dec.)
5.   If day < 31
6.     Then tomorrowDay = day + 1
7.   Else
8.     tomorrowDay = 1
9.     tomorrowMonth = month + 1
10.  EndIf
11. Case 2: month Is 4,6,9,Or 11 '30 day months
12.  If day < 30
13.    Then tomorrowDay = day + 1
14.    Else
15.      tomorrowDay = 1
16.      tomorrowMonth = month + 1
17.  EndIf
18. Case 3: month Is 12: 'December
19.  If day < 31
20.    Then tomorrowDay = day + 1
21.    Else
22.      tomorrowDay = 1
23.      tomorrowMonth = 1
24.      If year = 2012
25.        Then Output ("2012 is over")
26.        Else tomorrow.year = year + 1
27.      EndIf
28.  EndIf
```



```
29. Case 4: month is 2: 'February
30.   If day < 28
31.     Then tomorrowDay = day + 1
32.     Else
33.       If day = 28
34.         Then
35.           If ((year MOD 4)=0)AND((year MOD 400)≠0)
36.             Then tomorrowDay = 29 'leap year
37.             Else      'not a leap year
38.               tomorrowDay = 1
39.               tomorrowMonth = 3
40.             EndIf
41.           Else If day = 29
42.             Then tomorrowDay = 1
43.             tomorrowMonth = 3
44.             Else Output("Cannot have Feb.", day)
45.           EndIf
46.         EndIf
47.       EndIf
48.     EndCase
49.   Output ("Tomorrow's date is", tomorrowMonth,
        tomorrowDay, tomorrowYear)
50. End NextDate
```





# Commission Program Pseudo-Code

Program Commission

Dim lockPrice, stockPrice, barrelPrice As Real

Dim locks, stocks, barrels As Integer

Dim totalLocks, totalStocks As Integer

Dim totalBarrels As Integer

Dim lockSales, stockSales As Real

Dim barrelSales As Real

Dim sales, commission As Real

1 lockPrice = 45.0

2 stockPrice = 30.0

3 barrelPrice = 25.0

4 totalLocks = 0

5 totalStocks = 0

6 totalBarrels = 0

7 Input(locks)

8 While NOT(locks = -1)

9 Input(stocks, barrels)

10 totalLocks = totalLocks + locks

11 totalStocks = totalStocks + stocks

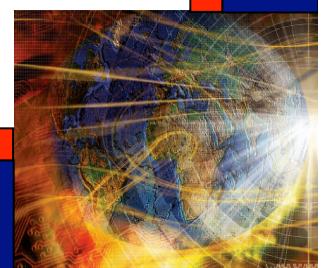
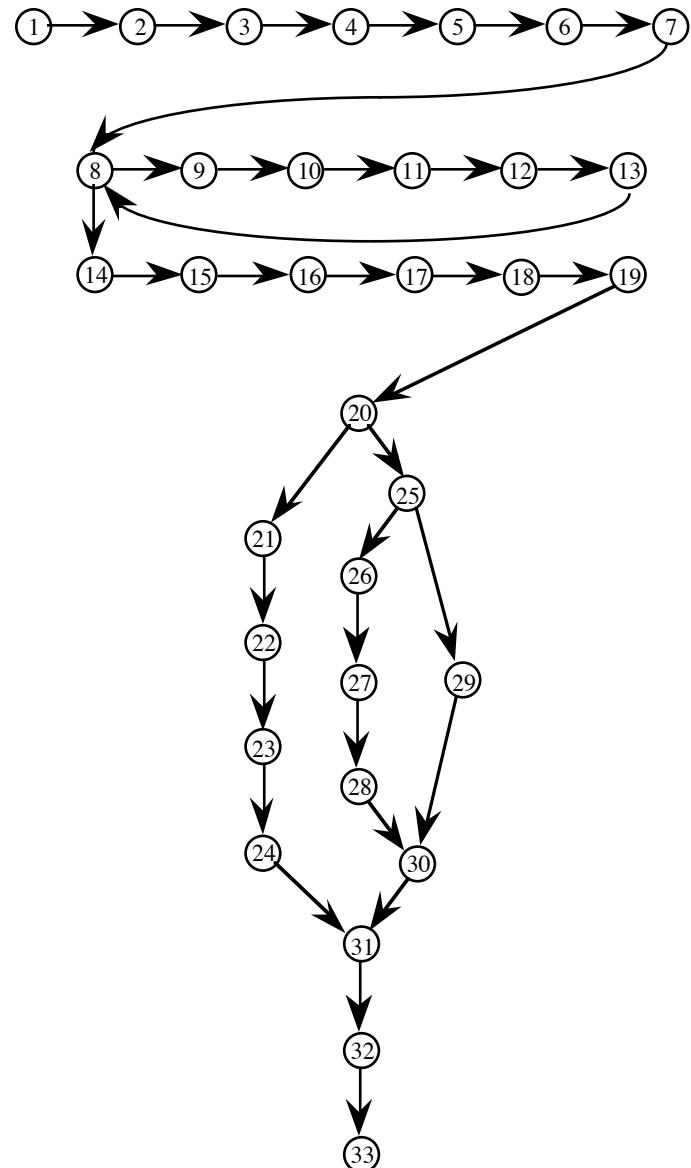
12 totalBarrels = totalBarrels + barrels

13 Input(locks)

14 EndWhile

15 Output("Locks sold: ", totalLocks)  
16 Output("Stocks sold: ", totalStocks)  
17 Output("Barrels sold: ", totalBarrels)  
18 sales = lockPrice\*totalLocks +  
stockPrice\*totalStocks + barrelPrice \* totalBarrels  
19 Output("Total sales: ", sales)  
20 If (sales > 1800.0)  
21 Then  
22 commission = 0.10 \* 1000.0  
23 commission = commission + 0.15 \* 800.0  
24 commission = commission + 0.20\*(sales-1800.0)  
25 Else If (sales > 1000.0)  
26 Then  
27 commission = 0.10 \* 1000.0  
28 commission = commission + 0.15\*(sales-1000.0)  
29 Else commission = 0.10 \* sales  
30 EndIf  
31 EndIf  
32 Output("Commission is \$", commission)  
33 End Commission





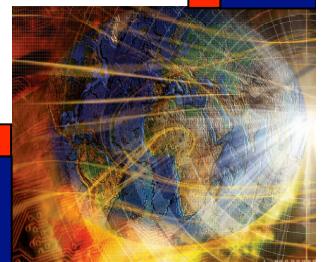
# Chapter 10

## Data Flow Testing Slice Testing



# Data Flow Testing

- Often confused with "dataflow diagrams".
- Main concern: places in a program where data values are defined and used.
- Static (compile time) and dynamic (execution time) versions.
- Static: Define/Reference Anomalies on a variable that
  - is defined but never used (referenced)
  - is used but never defined
  - is defined more than once
- Starting point is a program, P, with program graph  $G(P)$ , and the set V of variables in program P.
- "Interesting" data flows are then tested as mini-functions.



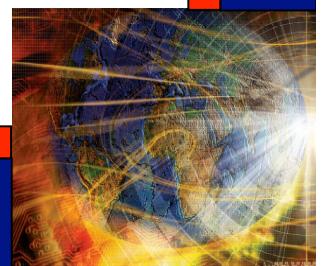
# Definitions

- Node  $n \in G(P)$  is a *defining node of the variable*  $v \in V$ , written as  $\text{DEF}(v, n)$ , iff the value of the variable  $v$  is defined at the statement fragment corresponding to node  $n$ .
- Node  $n \in G(P)$  is a *usage node of the variable*  $v \in V$ , written as  $\text{USE}(v, n)$ , iff the value of the variable  $v$  is used at the statement fragment corresponding to node  $n$ .
- A usage node  $\text{USE}(v, n)$  is a *predicate use* (denoted as P-use) iff the statement  $n$  is a predicate statement; otherwise,  $\text{USE}(v, n)$  is a *computation use* (denoted C-use).



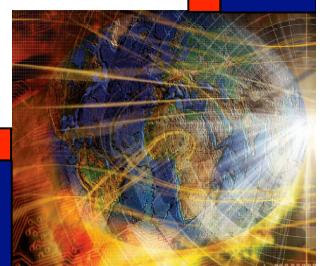
# More Definitions

- A *definition-use path with respect to a variable v* (denoted du-path) is a path in  $\text{PATHS}(P)$  such that for some  $v \in V$ , there are define and usage nodes  $\text{DEF}(v, m)$  and  $\text{USE}(v, n)$  such that  $m$  and  $n$  are the initial and final nodes of the path.
- A *definition-clear path with respect to a variable v* (denoted dc-path) is a definition-use path in  $\text{PATHS}(P)$  with initial and final nodes  $\text{DEF}(v, m)$  and  $\text{USE}(v, n)$  such that no other node in the path is a defining node of  $v$ .



# Example: first part of the Commission Program

1. Program Commission (INPUT,OUTPUT)
2. Dim locks, stocks, barrels As Integer
3. Dim lockPrice, stockPrice, barrelPrice As Real
4. Dim totalLocks, totalStocks, totalBarrels As Integer
5. Dim lockSales, stockSales, barrelSales As Real
6. Dim sales, commission As Real
7. lockPrice = 45.0
8. stockPrice = 30.0
9. barrelPrice = 25.0
10. totalLocks = 0
11. totalStocks = 0
12. totalBarrels = 0
13. Input(locks)
14. While NOT(locks = -1)
15.   Input(stocks, barrels)
16.   totalLocks = totalLocks + locks
17.   totalStocks = totalStocks + stocks
18.   totalBarrels = totalBarrels + barrels
19.   Input(locks)
20. EndWhile
21. Output("Locks sold: ", totalLocks)
22. Output("Stocks sold: ", totalStocks)
23. Output("Barrels sold: ", totalBarrels)

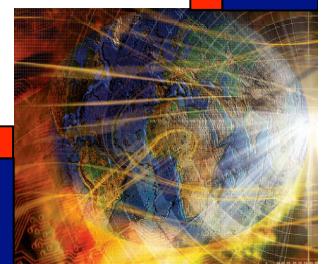
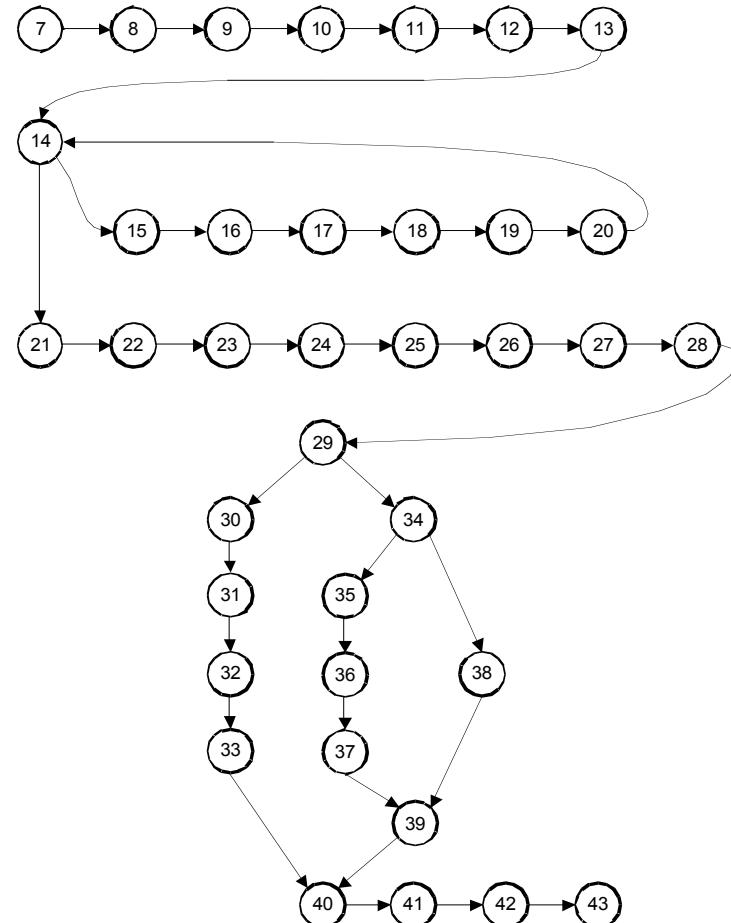


# Rest of Commission Problem

```
23. Output("Barrels sold: ", totalBarrels)
24. lockSales = lockPrice * totalLocks
25. stockSales = stockPrice * totalStocks
26. barrelSales = barrelPrice * totalBarrels
27. sales = lockSales + stockSales + barrelSales
28. Output("Total sales: ", sales)
29. If (sales > 1800.0)
30. Then
31.   commission = 0.10 * 1000.0
32.   commission = commission + 0.15 * 800.0
33.   commission = commission + 0.20 *(sales-1800.0)
34. Else If (sales > 1000.0)
35.   Then
36.     commission = 0.10 * 1000.0
37.     commission = commission + 0.15 *(sales-1000.0)
38.   Else
39.     commission = 0.10 * sales
40. EndIf
41. EndIf
42. Output("Commission is $", commission)
43. End Commission
```

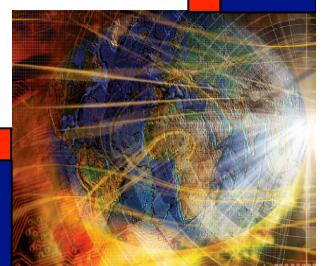


# Program Graph of Commission Problem



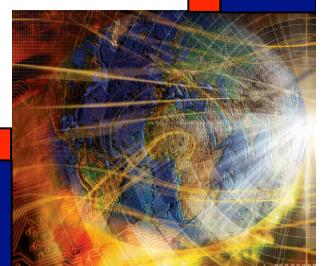
# Define/Use Test Cases

- **Technique:** for a particular variable,
  - find all its definition and usage nodes, then
  - find the du-paths and dc-paths among these.
  - for each path, devise a "suitable" (functional?) set of test cases.
- **Note:** du-paths and dc-paths have both static and dynamic interpretations
  - Static: just as seen in the source code
  - Dynamic: must consider execution-time flow (particularly for loops)
- **Definition clear paths are easier to test**
  - No need to check each definition node, as is necessary for du-paths



# Define and Use Nodes

<b>Variable</b>	<b>Defined at Node</b>	<b>Used at Node</b>
<b>locks</b>	<b>13, 19</b>	<b>14, 16</b>
<b>stocks</b>	<b>15</b>	<b>17</b>
<b>barrels</b>	<b>15</b>	<b>18</b>
<b>totalLocks</b>	<b>10, 16</b>	<b>16, 21, 24</b>
<b>totalStocks</b>	<b>11, 17</b>	<b>17, 22, 25</b>
<b>totalBarrels</b>	<b>12, 18</b>	<b>18, 23, 26</b>

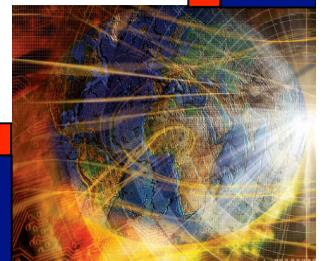


# Example (continued)

13. Input(locks)
14. While NOT(locks = -1)
15.   Input(stocks, barrels)
16.   totalLocks = totalLocks + locks
17.   totalStocks = totalStocks + stocks
18.   totalBarrels = totalBarrels + barrels
19.   Input(locks)
20. EndWhile

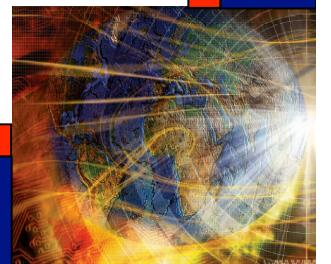
We have

- DEF (locks, 13), DEF (locks, 19)
- USE (locks, 14), a predicate use
- USE (locks, 16). A computation use
- du-paths for locks are the node sequences <13, 14> (a dc-path),  
<13, 14, 15, 16>, <19, 20, 14 >, < 19, 20, 14 , 15, 16>
- Is <13, 14, 15, 16> definition clear?
- Is < 19, 20, 14, 15, 16> definition clear? What about repetitions?



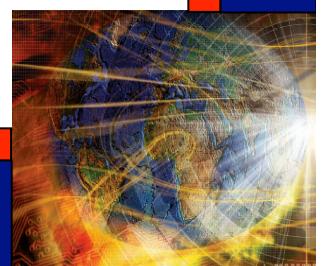
# Coverage Metrics Based on du-paths

- In the following definitions,  $T$  is a set of paths in the program graph  $G(P)$  of a program  $P$ , with the set  $V$  of variables.
- The set  $T$  satisfies the *All-Defs criterion* for the program  $P$  iff for every variable  $v \in V$ ,  $T$  contains definition-clear paths from every defining node of  $v$  to a use of  $v$ .
- The set  $T$  satisfies the *All-Uses criterion* for the program  $P$  iff for every variable  $v \in V$ ,  $T$  contains definition-clear paths from every defining node of  $v$  to every use of  $v$ , and to the successor node of each  $\text{USE}(v, n)$ .



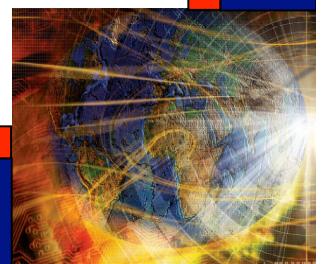
# Coverage Metrics Based on du-paths (continued)

- The set T satisfies the *All-P-Uses/Some C-Uses* criterion for the program P iff for every variable  $v \in V$ , T contains definition-clear paths from every defining node of v to every predicate use of v; if a definition of v has no P-uses, a definition-clear path leads to at least one computation use.



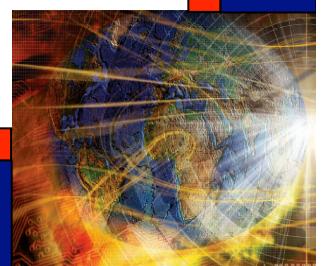
# Coverage Metrics Based on du-paths (continued)

- The set T satisfies the *All-C-Uses/Some P-Uses* criterion for the program P iff for every variable  $v \in V$ , T contains definition-clear paths from every defining node of v to every computation use of v; if a definition of v has no C-uses, a definition-clear path leads to at least one predicate use.

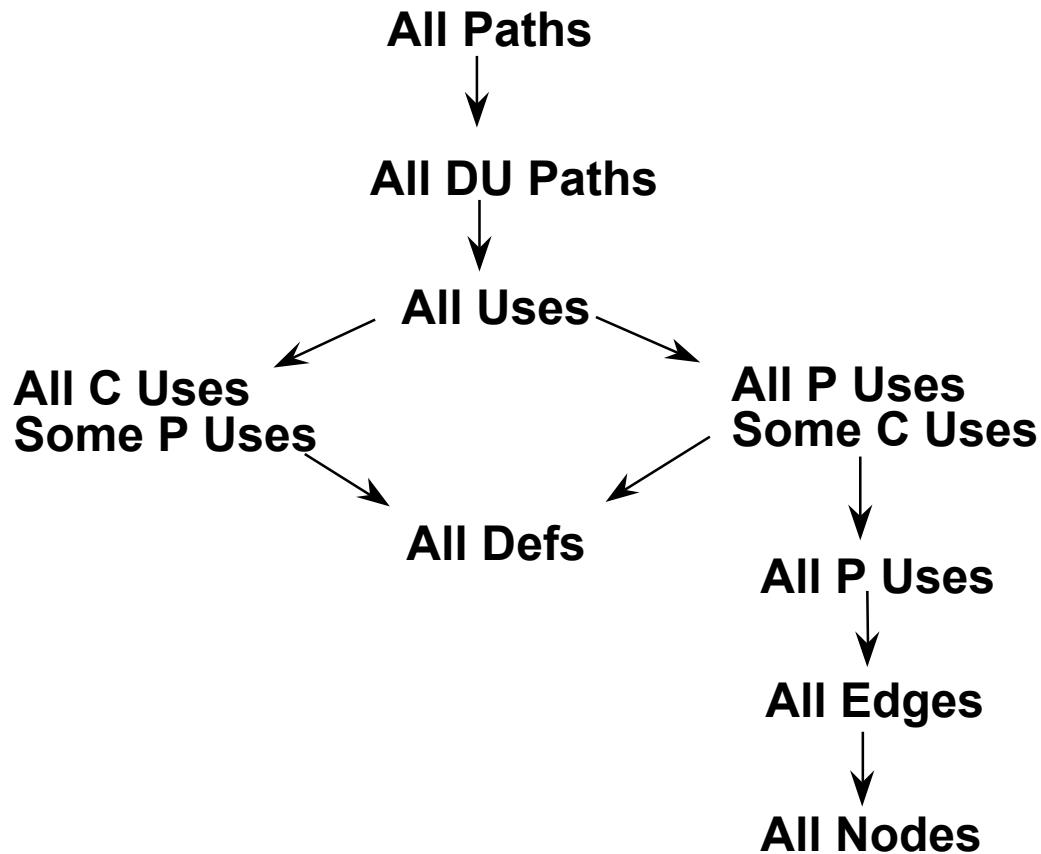


# Coverage Metrics Based on du-paths (concluded)

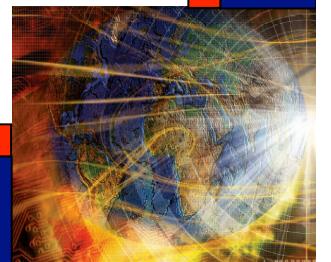
- The set  $T$  satisfies the *All-du-paths criterion* for the program  $P$  iff for every variable  $v \in V$ ,  $T$  contains definition-clear paths from every defining node of  $v$  to every use of  $v$  and to the successor node of each  $\text{USE}(v, n)$ , and that these paths are either single-loop traversals or cycle-free.



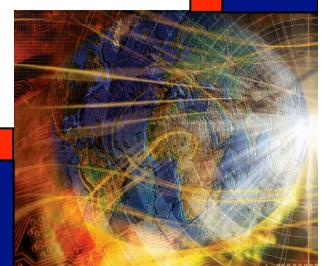
# Rapps-Weyuker Coverage Subsumption



S. Rapps and E. J. Weyuker  
"Selecting Software Test Data Using Data Flow Information"  
*IEEE Transactions on Software Engineering* vol 11 no 4 IEEE Computer Society Press, Washington, D. C. , April 1985, pp 367 - 375.

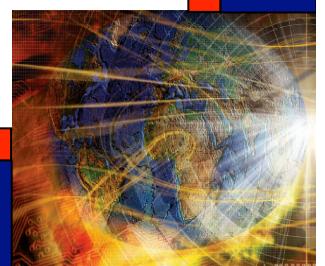


# Exercise: Where does the “All definition-clear paths” coverage metric fit in the Rapps-Weyuker lattice?



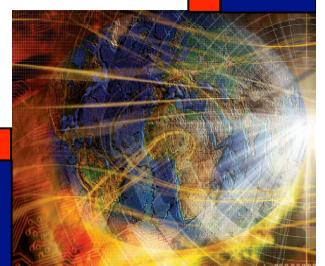
# Data Flow Testing Strategies

- **Data flow testing is indicated in**
  - Computation-intensive applications
  - “long” programs
  - Programs with many variables
- **A definition-clear du-path represents a small function that can be tested by itself.**
- **If a du-path is not definition-clear, it should be tested for each defining node.**



# Slice Testing

- Often confused with "module execution paths"
- Main concern: portions of a program that "contribute" to the value of a variable at some point in the program.
- Nice analogy with history -- a way to separate a complex system into "disjoint" components that interact:
  - European history
  - North American history
  - Orient history
- A dynamic construct.



# Slice Testing Definitions

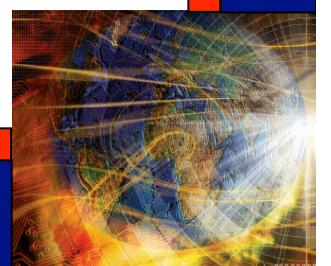
**Starting point is a program, P, with program graph  $G(P)$ , and the set V of variables in program P. Nodes in the program graph are numbered and correspond to statement fragments.**

- **Definition:** The *slice on the variable set V at statement fragment n*, written  $S(V, n)$ , is the set of node numbers of all statement fragments in P prior to n that contribute to the values of variables in V at statement fragment n.
- This is actually a “backward slice”.
- **Exercise:** define a “forward slice”.



# Fine Points

- "prior to" is the dynamic part of the definition.
- "contribute" is best understood by extending the Define and Use concepts:
  - P-use: used in a predicate (decision)
  - C-use: used in computation
  - O-use: used for output
  - L-use: used for location (pointers, subscripts)
  - I-use: iteration (internal counters, loop indices)
  - I-def: defined by input
  - A-def: defined by assignment
- usually, the set V of variables consists of just one element.
- can choose to define a slice as a compilable set of statement fragments -- this extends the meaning of "contribute"
- because slices are sets, we can develop a lattice based on the subset relationship.



## In the program fragment

```
13. Input(locks)
14. While NOT(locks = -1)
15.   Input(stocks, barrels)
16.   totalLocks = totalLocks + locks
17.   totalStocks = totalStocks + stocks
18.   totalBarrels = totalBarrels + barrels
19.   Input(locks)
20.EndWhile
```

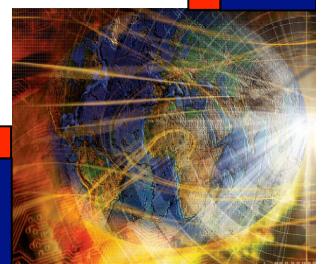
There are these slices on locks (notice that statements 15, 17, and 18 do not appear):

S1:  $S(\text{locks}, 13) = \{13\}$

S2:  $S(\text{locks}, 14) = \{13, 14, 19, 20\}$

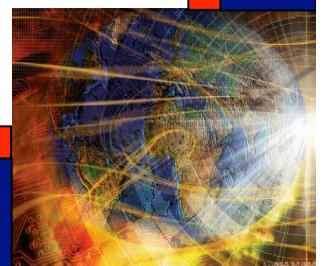
S3:  $S(\text{locks}, 16) = \{13, 14, 19, 20\}$

S4:  $S(\text{locks}, 19) = \{19\}$

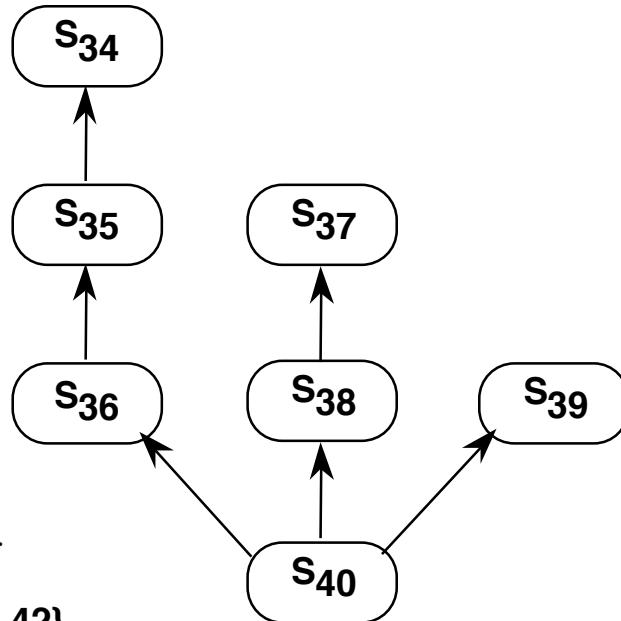


# Lattice of Slices

- Because a slice is a set of statement fragment numbers, we can find slices that are subsets of other slices.
- This allows us to “work backwards” from points in a program, presumably where a fault is suspected.
- The statements leading to the value of commission when it is output are an excellent example of this pattern.
- Some researchers propose that this is the way good programmers think when they debug code.



# Example Lattice of Slices



S34:  $S(\text{commission}, 41) = \{41\}$

S35:  $S(\text{commission}, 42) = \{41, 42\}$

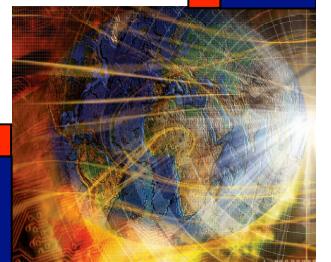
S36:  $S(\text{commission}, 43) = \{3, 4, 5, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 30, 36, 41, 42, 43\}$

S37:  $S(\text{commission}, 47) = \{47\}$

S38:  $S(\text{commission}, 48) = \{3, 4, 5, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 36, 47, 48\}$

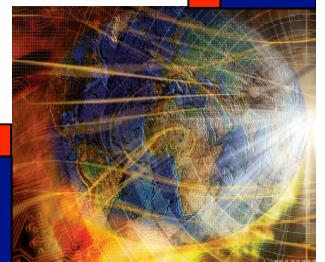
S39:  $S(\text{commission}, 50) = \{3, 4, 5, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 36, 50\}$

S40:  $S(\text{commission}, 51) = \{3, 4, 5, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 36, 41, 42, 43, 47, 48, 50\}$



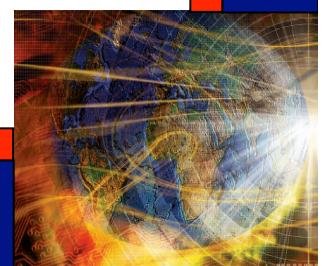
# Diagnostic Testing with Slices

- Relative complements of slices yield a "diagnostic" capability. The relative complement of a set B with respect to another set A is the set of all elements of A that are not elements of B. It is denoted as A - B.
- Consider the relative complement set S(commission, 48) - S(sales, 35):
  - $S(\text{commission}, 48) = \{3, 4, 5, 36, 18, 19, 20, 23, 24, 25, 26, 27, 34, 38, 39, 40, 44, 45, 47\}$
  - $S(\text{sales}, 35) = \{3, 4, 5, 36, 18, 19, 20, 23, 24, 25, 26, 27\}$
  - $S(\text{commission}, 48) - S(\text{sales}, 35) = \{34, 38, 39, 40, 44, 45, 47\}$
- If there is a problem with commission at line 48, we can divide the program into two parts, the computation of sales at line 34, and the computation of commission between lines 35 and 48. If sales is OK at line 34, the problem must lie in the relative complement; if not, the problem may be in either portion.



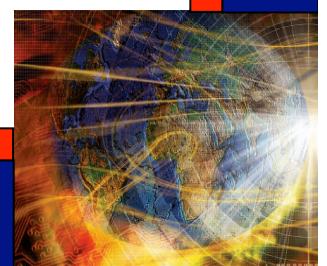
# Programming with Slices

- One researcher suggests the possibility of “slice splicing”:
  - Code a slice, compile and test it.
  - Code another slice, compile and test it, then splice the two slices.
  - Continue until the whole program is complete.
- Exercise: in what ways is slice splicing distinct from agile (bottom up) programming?



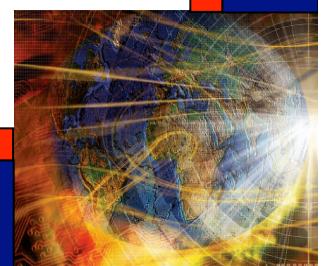
## Exercise/Discussion: When should testing stop?

- **when you run out of time?**
- **when continued testing causes no new failures?**
- **when continued testing reveals no new faults?**
- **when you can't think of any new test cases?**
- **when you reach a point of diminishing returns?**
- **when mandated coverage has been attained?**
- **when all faults have been removed?**



# Chapter 11

## Retrospective on Structural Testing



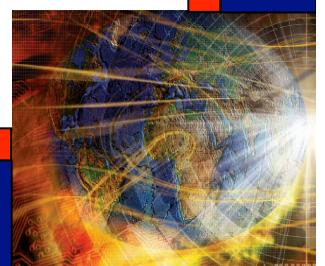
# Structural Testing Comparison

- How much testing is enough?
- Effort and size trendlines
- Metrics for test method comparison

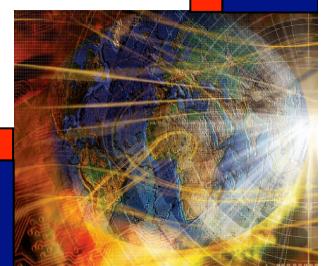
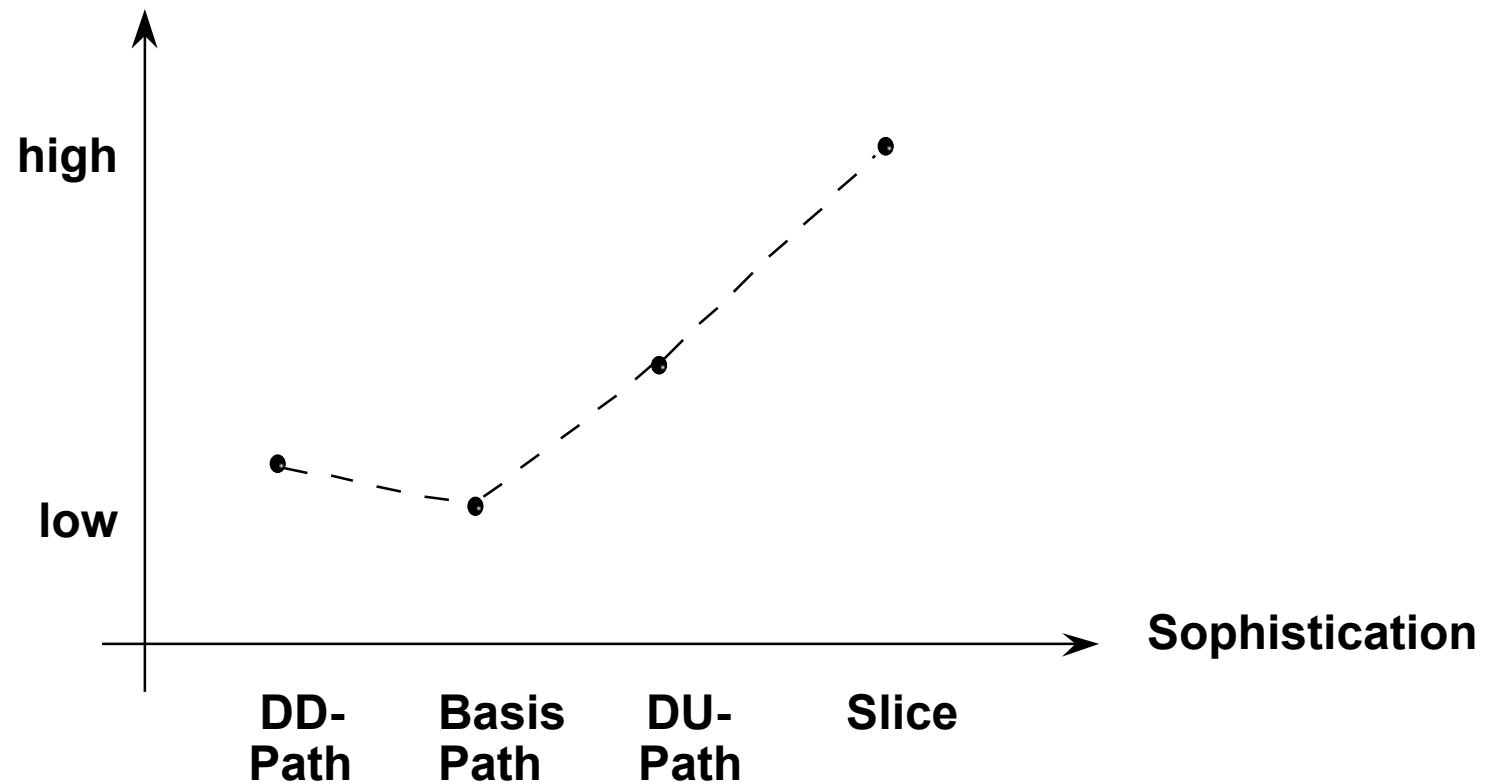


## Exercise/Discussion: When should testing stop?

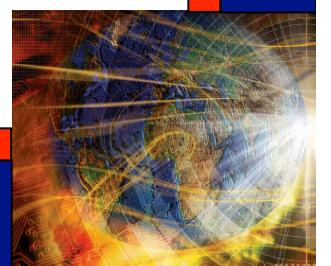
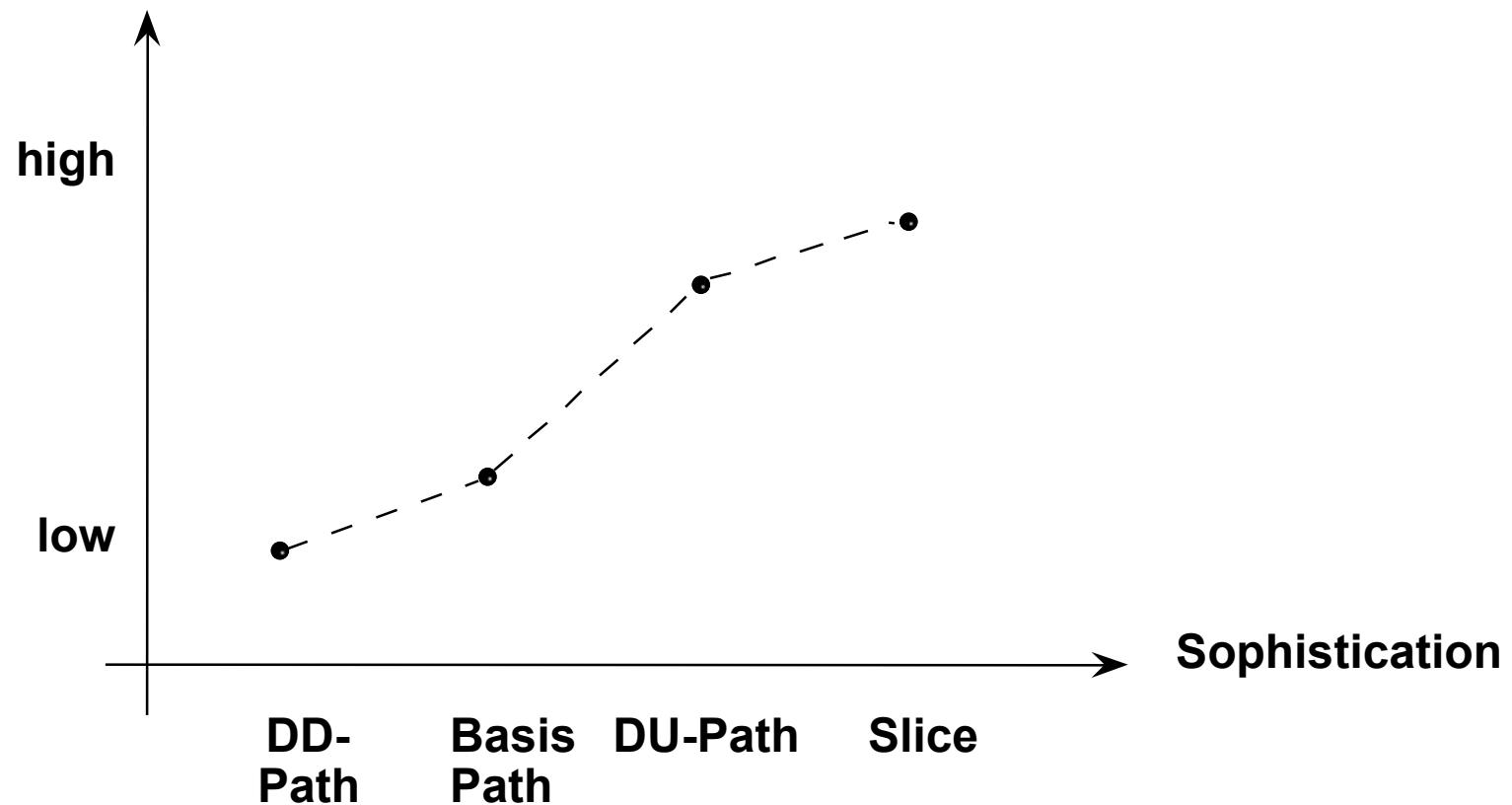
- **when you run out of time?**
- **when continued testing causes no new failures?**
- **when continued testing reveals no new faults?**
- **when you can't think of any new test cases?**
- **when you reach a point of diminishing returns?**
- **when mandated coverage has been attained?**
- **when all faults have been removed?**



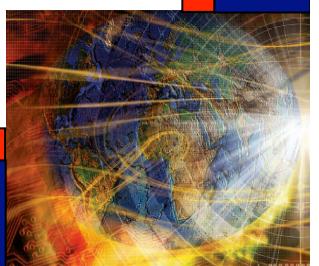
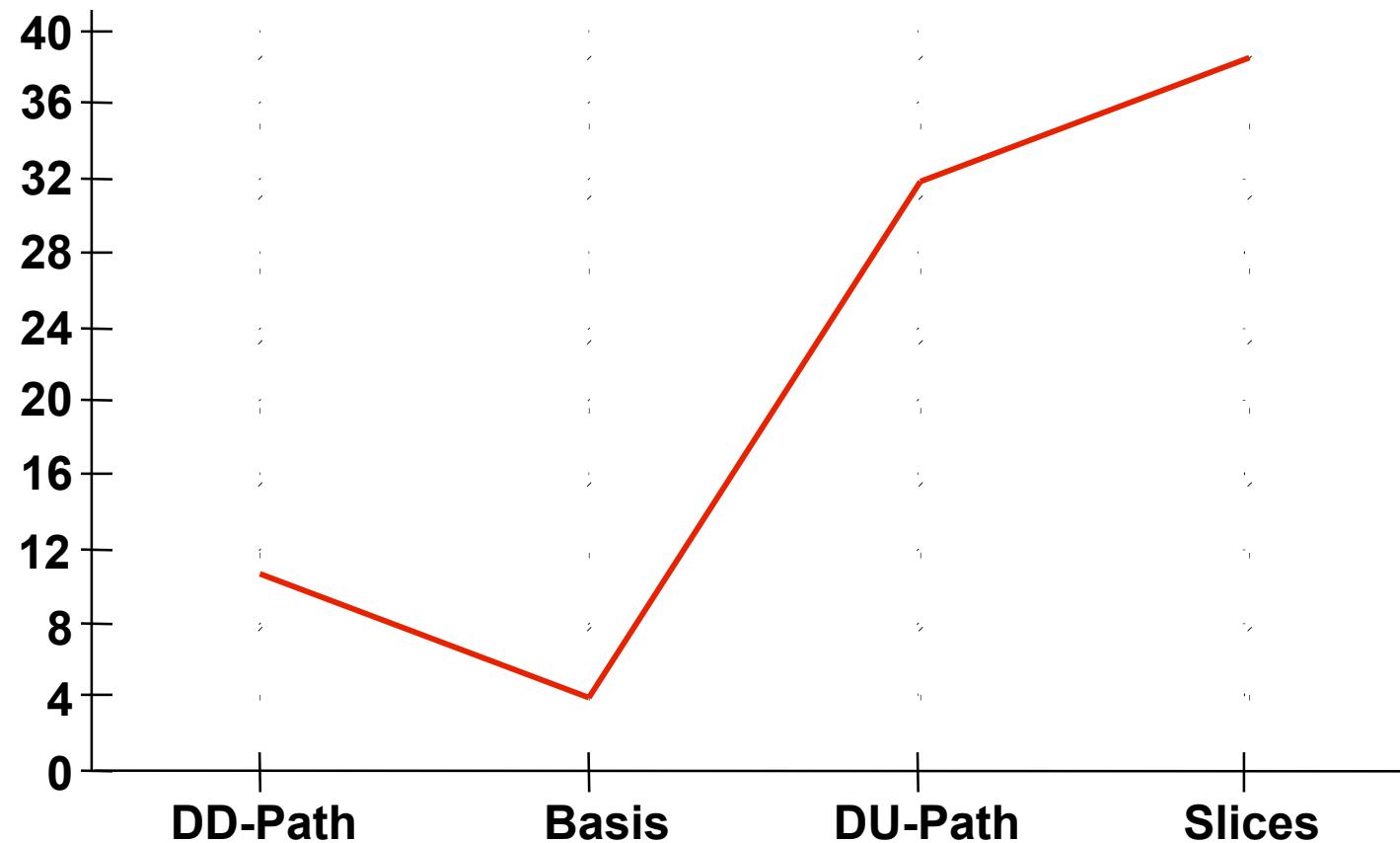
# Number of Test Coverage Items



# Effort to Identify Test Coverage Items

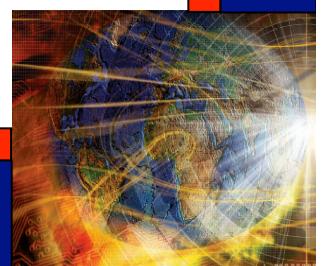


# Number of Coverage Items in the Commission Problem



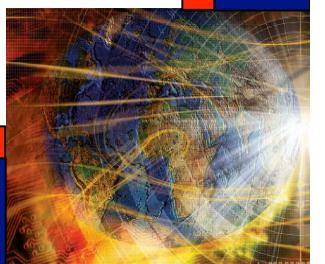
# Metrics for Test Method Comparison

- Assume that a functional testing technique M generates m test cases, and that these test cases are tracked with respect to a structural metric S that identifies s elements in the unit under test. When the m test cases are executed, they traverse n of the s structural elements.
- This framework supports the definition of metrics for testing effectiveness.



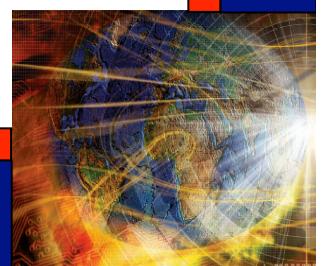
# Metrics for Testing Effectiveness

- The ***coverage*** of a methodology M with respect to a metric S is ratio of n to s. We denote it as  $C(M,S)$ .
- The ***redundancy*** of a methodology M with respect to a metric S is ratio of m to s. We denote it as  $R(M,S)$ .
- The ***net redundancy*** of a methodology M with respect to a metric S is ratio of m to n. We denote it as  $NR(M,S)$ .



# Sample Comparisons

Method	m	n	s	C(M,S)	R(M,S)	NR(M,S)
<b>Triangle Program</b>						
<b>BVA</b>	<b>15</b>	<b>7</b>	<b>11</b>	<b>0.64</b>	<b>1.36</b>	<b>2.14</b>
<b>Worst Case BVA</b>	<b>125</b>	<b>11</b>	<b>11</b>	<b>1.00</b>	<b>11.36</b>	<b>11.36</b>
<b>Commission Program</b>						
<b>Output BVA</b>	<b>25</b>	<b>11</b>	<b>11</b>	<b>1.00</b>	<b>2.27</b>	<b>2.27</b>
<b>Decision Table</b>	<b>3</b>	<b>11</b>	<b>11</b>	<b>1.00</b>	<b>0.27</b>	<b>0.27</b>
<b>DD-Path</b>	<b>25</b>	<b>11</b>	<b>11</b>	<b>1.00</b>	<b>2.27</b>	<b>2.27</b>
<b>DU-Path</b>	<b>25</b>	<b>33</b>	<b>33</b>	<b>1.00</b>	<b>0.76</b>	<b>0.76</b>
<b>Slices</b>	<b>25</b>	<b>40</b>	<b>40</b>	<b>1.00</b>	<b>0.63</b>	<b>0.63</b>



# Software Testing & Quality Assurance

## *Decision Table-Based Testing*

- Background and Definitions
- Related Techniques
- Decision Tables for the Triangle Program
- Decision Tables for the NextDate Program
- Test Case Design Guidelines
- Guidelines and Observations

# Credits & Readings

- The material included in these slides are mostly adopted from the following books:
  - *Software Testing: A Craftsman's Approach*, by Paul Jorgensen, CRC PRESS, third edition, ISBN: 0-8493-7475-8
  - Cem Kaner, Jack Falk, Hung Q. Nguyen, “*Testing Computer Software*” Wiley (see also <http://www.testingeducation.org/>)
  - Paul Ammann and Jeff Offutt, “*Introduction to Software Testing*”, Cambridge University Press
  - Glen Myers, “*The Art of Software Testing*”

# Decision Tables

- It's a tabular technique that has been used to represent and analyze complex logical relationships since the early 1960's
- Ideal for describing situations in which a number of combinations of actions are taken under varying sets of conditions

# Decision Table Anatomy

Stub	Rule 1	Rule 2	Rules 3,4	Rule 5	Rule 6	Rules 7,8
c1	T	T	T	F	F	F
c2	T	T	F	T	T	F
c3	T	F	-	T	F	-
a1	X	X		X		
a2	X				X	
a3		X		X		
a4			X			X

- Left of the bold vertical line is called the *stub portion*
- The part to the right is called the *entry portion*
  - Columns in the entry portion are rules
  - Rules indicate what actions need to be taken
- The part above the horizontal bold line is called the *condition portion*
- The part below is called the *action portion*

- Tables that have binary conditions are called truth tables
- Tables where all the conditions are binary are called limited entry decision tables
  - If  $n$  conditions exist, there must be  $2^n$  rules
- Tables that allow conditions to have several values are called extended entry decision tables
  - If  $n$  conditions exist with  $k$  number of values each then there must be  $k^n$  rules

# Redundancy in Decision Tables

conditions	1-4	5	6	7	8	9
c1:	T	F	F	F	F	T
c2:	--	T	T	F	F	F
c3:	--	T	F	T	F	F
<hr/>						
a1:	X	X	X	--	--	X
a2:	--	X	X	X	--	--
a3:	X	--	X	X	X	X

Rule 9 is identical to Rule 4 (T, F, F)

- Since the action entries for rules 4 and 9 are identical, there is no ambiguity, just redundancy.

# Inconsistency in Decision Tables

conditions	1-4	5	6	7	8	9
c1:	T	F	F	F	F	T
c2:	--	T	T	F	F	F
c3:	--	T	F	T	F	F
<hr/>						
a1:	X	X	X	--	--	--
a2:	--	X	X	X	--	X
a3:	X	--	X	X	X	--

Rule 9 is identical to Rule 4 (T, F, F)

- Since the action entries for rules 4 and 9 are different there is ambiguity.
- This table is inconsistent, and the inconsistency implies non-determinism.

# A Decision Table for Printer Troubleshooting

Conditions	Printer does not print?	Y	Y	Y	Y	N	N	N	N
	A red light is flashing?	Y	Y	N	N	Y	Y	N	N
	Printer is unrecognized?	Y	N	Y	N	Y	N	Y	N
Actions	Check the power cable			X					
	Check the printer-computer cable	X		X					
	Ensure printer software is installed	X		X		X		X	
	Check/replace ink	X	X			X	X		
	Check for paper jam		X		X				

# A Decision Table for a Billing System

When monthly bills are processed, it is noted whether the customer has made a payment. The payment may, or may not, equal what is owed. Also, it is possible that a customer owes money but makes no payment at all. For a customer that sends in money, a receipt must be sent. For a customer that owes money, a bill must be sent. Furthermore, a customer not owing money is to receive a special preferred-customer sales flyer. In other words, a decision has to be made concerning what actions to take and send a bill, a receipt, or a flyer based on certain conditions.

Payment made?	T		F	
Balance owed?	T	F	T	F
Send receipt	●	●		
Send bill	●		●	
Send flyer		●		●

- $k^n$  combinations where
  - $n$  is the number of conditions and
  - $k$  is the number of possible values for each condition
- In this case,  $n=2$  and  $k=2$

# Decision Table for a *Toy-Test* Activity

Make a decision table for a *toy-test* activity. Toy-test is an activity designed to test toys in different ways. If the toy is made of plastic, it should be stepped on. If it is made of medal, hit it with a hammer. If it has stickers on it, try to peel them off. If it is metal and painted, scratch the paint with a nail. If it is metal, has wheels, and has stickers, then submerge it in water for one hour.

# Decision Table for Toy-test

Made of plastic	T				F			
Has stickers	T		F		T		F	
It is painted	T	F	T	F	T	F	T	F
Has wheels	T	F	T	F	T	F	T	F
Step on it	X	X	X	X	X	X	X	X
Hit it						X	X	X
Peel stickers	X	X	X	X		X	X	X
Scratch paint					X	X		X
Submerge					X	X		

# Decision Table for Car Insurance

Create a decision table to assign risk categories and charges to applicants for car insurance. Use the following rules. If the applicant is under 21, apply a surcharge. If the applicant is male and under 26 and married, or male and over 26, assign him to risk category B. If the applicant is a single male under 26 or a female under 21, assign the applicant to risk category C. All other applicants are assigned to risk category A.

# Decision Table for Risk Categories

Age	<21				21-25				>26			
Gender	M		F		M		F		M		F	
Married	T	F	T	F	T	F	T	F	T	F	T	F
Surcharge	X	X	X	X								
Risk category A							X	X			X	X
Risk category B	X				X				X	X		
Risk category C		X	X	X		X						

# Identifying Test Cases Using Decision Tables

We interpret

- *Conditions* as inputs
  - Conditions refer to equivalence classes of inputs
- *Actions* as outputs
  - Actions refer to functional portions of the item being tested
- *Rules* to test cases
- Since a decision table can be mechanically forced to be complete, it produces a comprehensive set of test cases

## Techniques

- Adding an action to show when a rule is logically impossible
  - Sometimes conditions in the decision table refer to mutually exclusive possibilities
- The choice of conditions can greatly expand the size of a decision table

Stub	Rule 1	Rule 2	Rules 3,4	Rule 5	Rule 6	Rules 7,8
c1	T	T	T	F	F	F
c2	T	T	F	T	T	F
c3	T	F	-	T	F	-
a1	X	X		X		
a2	X					X
a3		X		X		
a4			X			X

# A Decision Table for the Triangle

C1: $a < b+c?$	F	T	T	T	T	T	T	T	T	T	T
C2: $b < a+c?$	-	F	T	T	T	T	T	T	T	T	T
C3: $c < a+b?$	-	-	F	T	T	T	T	T	T	T	T
C4: $a = b?$	-	-	-	T	T	T	T	F	F	F	F
C5: $a = c?$	-	-	-	T	T	F	F	T	T	F	F
C6: $b = c?$	-	-	-	T	F	T	F	T	F	T	F
A1: Not a Triangle	X	X	X								
A2: Scalene											X
A3: Isosceles						X		X	X		
A4: Equilateral				X							
A5: Impossible					X	X	X				

- Conditions C1-C6 represent equivalence input classes
- Actions A1-A5 represent output functionality of the program
- Rules portion represent possible test cases

- Notice the don't care entries
  - For instance, if the integers a, b and c don't constitute a triangle (C1-C3), we don't even care about possible equalities (C4-C6)
- Also, notice the impossible rule usage
  - For instance, if  $a=b$  and  $b=c$  then  $a=c$  must be true

# Derived Test Cases for Triangle

Case ID	a	b	c	Expected Output
DT1	4	1	2	Not a Triangle
DT2	1	4	2	Not a Triangle
DT3	1	2	4	Not a Triangle
DT4	5	5	5	Equilateral
DT5	?	?	?	Impossible
DT6	?	?	?	Impossible
DT7	2	2	3	Isosceles
DT8	?	?	?	Impossible
DT9	2	3	2	Isosceles
DT10	3	2	2	Isosceles
DT11	3	4	5	Scalene

- From the previous decision table we derive 11 test cases
- Each test case corresponds to an action taken (denoted by X in the decision table)
  - 3 impossible cases
  - 3 not valid triangle
  - 1 for an equilateral
  - 1 for scalene
  - 3 for isosceles

# The NextDate Problem

- The NextDate problem illustrates the problem of dependencies in the input domain
- Decision tables can highlight such dependencies
- Decision tables use the notion of “impossible action” to denote impossible combinations of conditions
- In NextDate, the impossible dates can be clearly marked as a separate action

We will try 3 refinement techniques

- Technique 1: focus is on identifying impossible combinations of inputs
- Technique 2: focuses on the leap year aspect
- Technique 3: focuses on specific days and months and clears up end-of-year considerations

## Equivalence Classes

M1= {month | month has 30 days}

M2= {month | month has 31 days}

M3= {month | month is February}

D1= {day |  $1 \leq day \leq 28$ }

D2= {day | day = 29}

D3= {day | day = 30}

D4= {day | day=31}

Y1= {year | year = 2000}

Y2= {year | year is a leap year}

Y3= {year | year is a common year}

# Partial Decision Table for *NextDate*

## (Technique 1)

C1: month in M1?	T	T	T	T	T	T	T	T	T	T	T	T	T
C2: month in M2?													
C3: month in M3?													
C4: day in D1?	T	T	T										
C5: day in D2?				T	T	T							
C6: day in D3?							T	T	T				
C7: day in D4?										T	T	T	
C8: year in Y1?	T			T			T			T			
C9: year in Y2?		T			T			T			T		
C10: year in Y3?			T			T			T			T	
A1: Impossible										X	X	X	
A2: Next Date	X	X	X	X	X	X	X	X	X				

- The goal is to show that many of these rules will be impossible
- Using the equivalence classes in the previous slide we can derive  $2^{10}=1024$  rules
- To see why these rules are impossible, we can expand our actions to include:
  - A1: Too many days in a month
  - A2: Cannot happen in a non-leap year
  - A3: Compute the next date

# Partial Decision Table for *NextDate*

## (Technique 2--Part I)

C1: month in	M1	M1	M1	M1	M2	M2	M2	M2
C2: day in	D1	D2	D3	D4	D1	D2	D3	D4
C3: year in	-	-	-	-	-	-	-	-
A1: Impossible				X				
A2: Increment day	X	X			X	X	X	
A3: Reset day			X					X
A4: Increment month			X					?
A5: Reset month								?
A6: Increment year								?

- The goal is to ensure that equivalence classes form a true partition of the input domain (i.e. disjoint subsets)
- We achieve that by creating an extended entry (rule) decision table
- We will consider only five possible manipulations needed to produce the next date:
  - Increment the day
  - Reset the day
  - Increment the month
  - Reset the month
  - Increment the year

➤ Using the cross product of equivalence classes M1-M3 X D1-D4 X Y1-Y3 we can derive  $1+2+3+5+\dots+10 = 55$  rules

Rest of table on next slide...

## Partial Decision Table for *NextDate* (Technique 2--Part II)

C1: month in	M3							
C2: day in	D1	D1	D1	D2	D2	D2	D3	D3
C3: year in	Y1	Y2	Y3	Y1	Y2	Y3	-	-
A1: Impossible				X		X	X	X
A2: Increment day		X						
A3: Reset day	X		X		X			
A4: Increment month	X		X		X			
A5: Reset month								
A6: Increment year								

# Partial Decision Table for *NextDate*

## (Technique 3--Part I)

C1: month in	M1	M1	M1	M1	M1	M2	M2	M2	M2	M2
C2: day in	D1	D2	D3	D4	D5	D1	D2	D3	D4	D5
C3: year in	-	-	-	-	-	-	-	-	-	-
A1: Impossible					X					
A2: Increment day	X	X	X			X	X	X	X	
A3: Reset day				X						X
A4: Increment month					X					X
A5: Reset month										
A6: Increment year										

### Updated Equivalence Classes

$M1 = \{\text{month} \mid \text{month has 30 days}\}$   
 $M2 = \{\text{month} \mid \text{month has 31 days}\}$   
 $M3 = \{\text{month} \mid \text{month is December}\}$   
 $M4 = \{\text{month} \mid \text{month is February}\}$

$D1 = \{\text{day} \mid 1 \leq \text{day} \leq 27\}$

$D2 = \{\text{day} \mid \text{day} = 28\}$

$D3 = \{\text{day} \mid \text{day} = 29\}$

$D4 = \{\text{day} \mid \text{day} = 30\}$

$D5 = \{\text{day} \mid \text{day}=31\}$

$Y1 = \{\text{year} \mid \text{year is a leap year}\}$

$Y2 = \{\text{year} \mid \text{year is a common year}\}$

➤ We consider a third set of equivalence classes in order to handle end-of-year issues

➤ Also, we focus on days and months

➤ We give no special attention to the year 2000

Rest of table on next slide...

# Partial Decision Table for *NextDate*

(Technique 3--Part II)

C1: month in	M3	M3	M3	M3	M3	M4							
C2: day in	D1	D2	D3	D4	D5	D1	D2	D2	D3	D3	D4	D4	D5
C3: year in	-	-	-	-	-	-	Y1	Y2	Y1	Y2	-	-	-
A1: Impossible											X	X	X
A2: Increment day	X	X	X	X		X	X						
A3: Reset day					X			X	X				
A4: Increment month								X	X				
A5: Reset month					X								
A6: Increment year					X								

# Commission Problem

- Test Cases for the Commission Problem
  - Not well served by a decision table
    - Very little decisional logic is used
    - There are no impossible rules that will occur in a decision table in which conditions correspond to the equivalence classes

# Test Case Design

- To identify test cases with decision tables, we interpret conditions as inputs, and actions as outputs
- Sometimes conditions end up referring to equivalence classes of inputs, and actions refer to major functional processing portions of the item being tested
- The rules are then interpreted as test cases

# Applicability

- The specification is given or can be converted to a decision table
- The order in which the predicates are evaluated does not affect the interpretation of the rules or resulting action
- The order of rule evaluation has no effect on resulting action
- Once a rule is satisfied and the action selected, no other rule need be examined
- The order of executing actions in a satisfied rule is of no consequence
- The restrictions do not in reality eliminate many potential applications
  - In most applications, the order in which the predicates are evaluated is immaterial
  - Some specific ordering may be more efficient than some other but in general the ordering is not inherent in the program's logic

# Decision Tables - Issues

- Before deriving test cases, ensure that
  - The rules are complete
    - Every combination of predicate truth values is explicit in the decision table
  - The rules are consistent
    - Every combination of predicate truth values results in only one action or set of actions

# Guidelines and Observations

- Decision Table testing is most appropriate for programs where
  - There is a lot of decision making
  - There are important logical relationships among input variables
  - There are calculations involving subsets of input variables
  - There are cause and effect relationships between input and output
  - There is complex computation logic (high cyclomatic complexity)
- Decision tables do not scale up very well
  - May need to
    - Use extended entry decision tables
    - Algebraically simplify tables
- Decision tables can be iteratively refined
  - The first attempt may be far from satisfactory

# Software Testing & Quality Assurance

## *Module 4: Equivalence Class Testing (ECT)*

- Background and Motivation
- Weak Normal ECT
- Strong Normal ECT
- Weak Robust ECT
- Strong Robust ECT

# Credits & Readings

- The material included in these slides are mostly adopted from the following books:
  - *Software Testing: A Craftsman's Approach*, by Paul Jorgensen, CRC PRESS, third edition, ISBN: 0-8493-7475-8
  - Cem Kaner, Jack Falk, Hung Q. Nguyen, “*Testing Computer Software*” Wiley (see also <http://www.testingeducation.org/>)
  - Cem Kaner, James Bach, Bret Pettichord, “*Lessons Learned in Software Testing*”, Wiley
  - Paul Ammann and Jeff Offutt, “*Introduction to Software Testing*”, Cambridge University Press
  - Kent Beck, “*Test-driven Development by Example*” Addison-Wesley
  - Robert Binder, “*Testing Object-Oriented Systems: Models, Patterns, and Tools*” Addison-Wesley
  - Glen Myers, “*The Art of Software Testing*”

# Motivation

- One weakness of Boundary Value Testing (BVT) is that it derives test cases with
  - Massive redundancy
    - For instance,  $<5,5,5>$ ,  $<6,6,6>$  and  $<100,100,100>$  are all *redundant* test cases for the triangle problem since they are “treated the same” i.e. “traversing the same execution path”
  - Serious gaps (i.e. sense of incomplete testing)
- Equivalence Class Testing (ECT) attempts to alleviate these problems
- It echoes the two deciding factors of BVT:
  - Robustness (i.e. handling invalid inputs effectively)
  - Single/multiple fault assumption (Weak vs. Strong ECT)

# Equivalence Class Testing

- Partition the set of all test cases into mutually disjoint subsets whose union is the entire set
- Choose one test case from each subset
- Two important advantages of ECT:
  - ❖ The fact that the entire set is represented provides a form of completeness
  - ❖ The disjoint-ness assures a form of non-redundancy

# Equivalence Class Selection

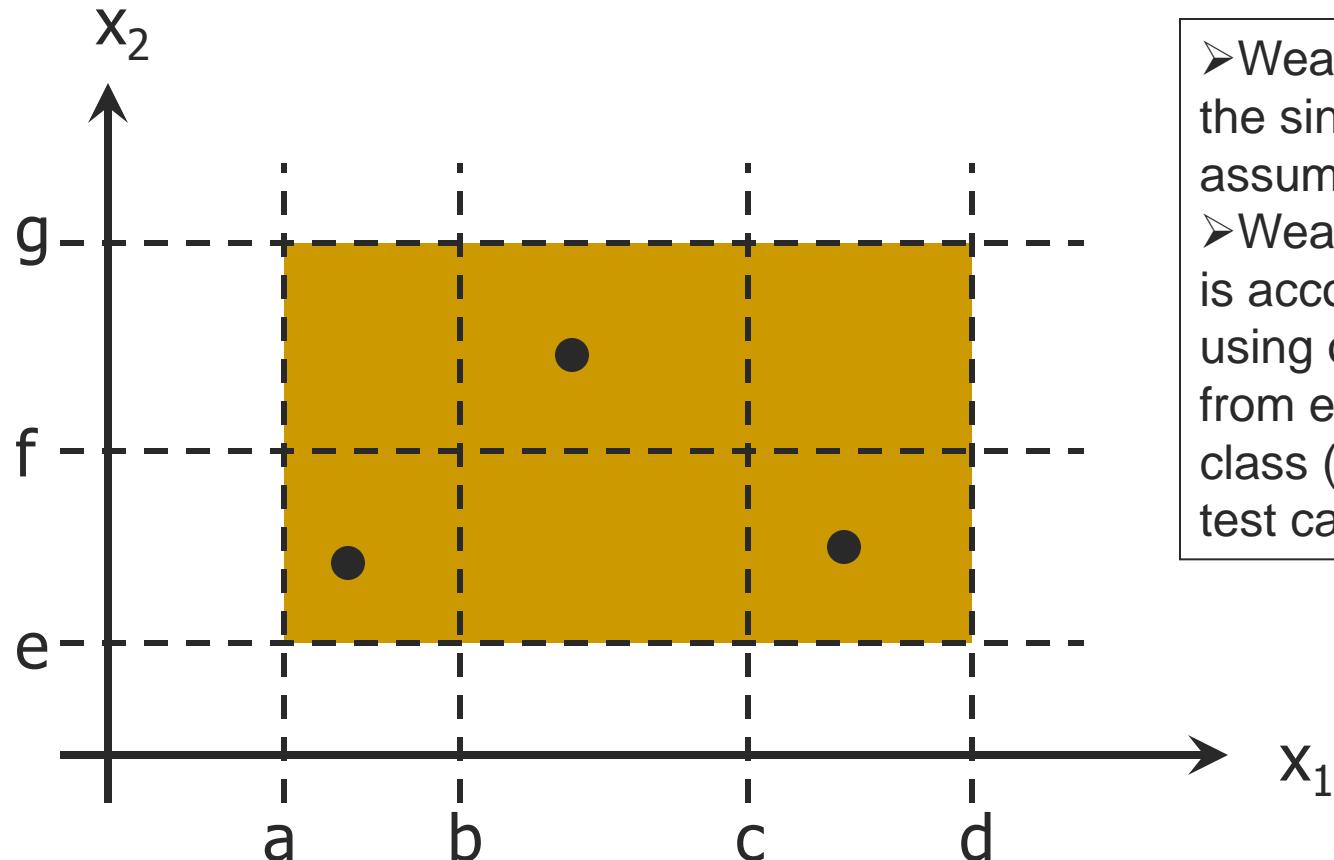
- The key point in ECT is the choice of the equivalence criteria (relation) that determine the classes
  - If the equivalence classes are chosen wisely, the potential redundancy among test cases is greatly reduced
  - When you define equivalence classes on the input domain watch for inputs being “treated the same” (they should belong to the same class)
  - Also, attempt to define equivalence classes on the output range of the program
- We will discuss four different types of ECT
  - Weak Normal ECT (“Weak/Strong” refers to the single/multiple fault assumption)
  - Strong Normal ECT
  - Weak Robust ECT (“Robust” relates to the consideration of invalid inputs)
  - Strong Robust ECT

# Applicability

- ECT is appropriate when the system under test can be expressed as a function of one or more variables, whose domains have well defined intervals (*i.e.* equivalence classes)
- Example: A two-variable function  $F(x_1, x_2)$ 
  - $a \leq x_1 \leq d$ , with intervals  $[a,b)$ ,  $[b,c)$ ,  $[c,d]^*$
  - $e \leq x_2 \leq g$ , with intervals  $[e,f)$ ,  $[f,g]$

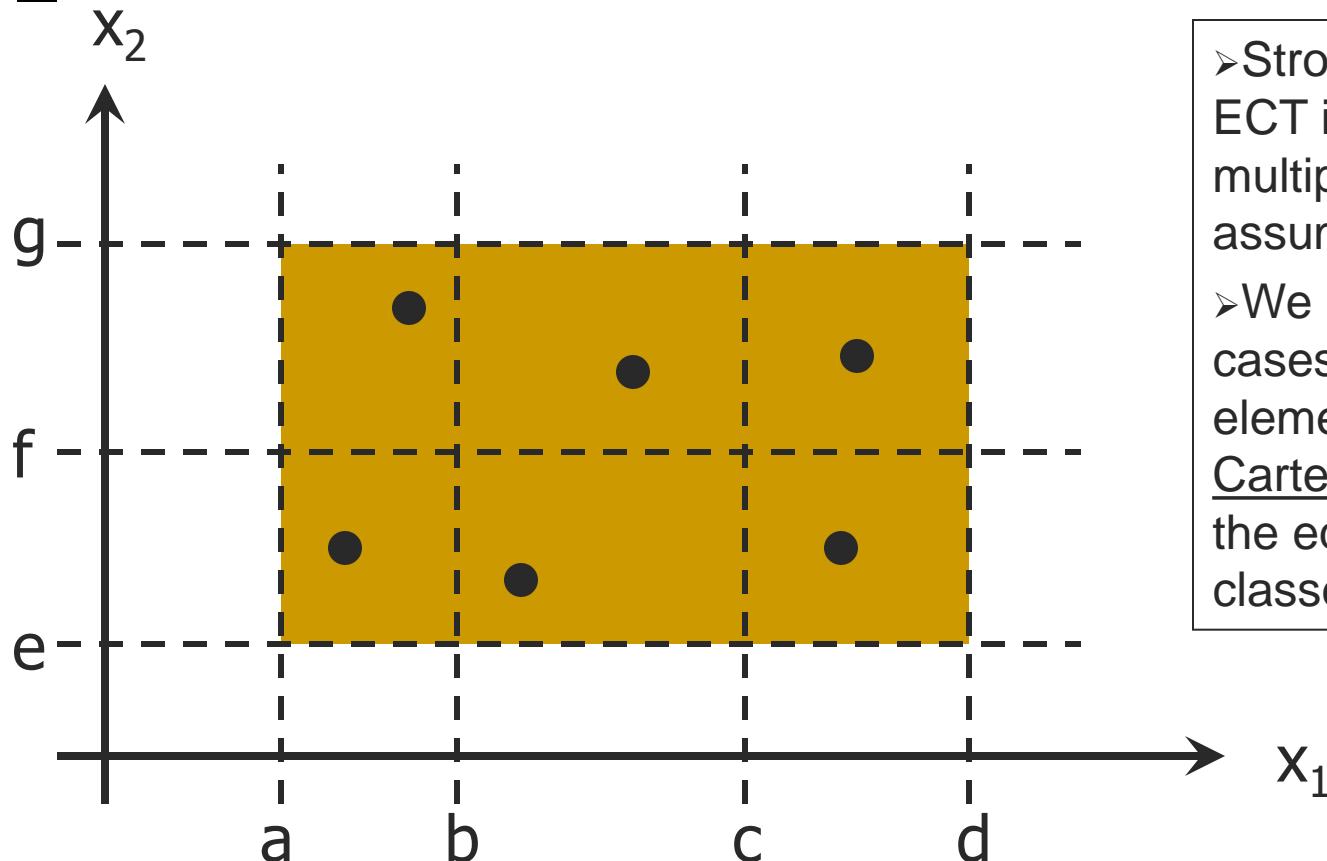
\* Where [ indicates a closed interval endpoint and ) indicates an open interval endpoint.

# Weak Normal ECT



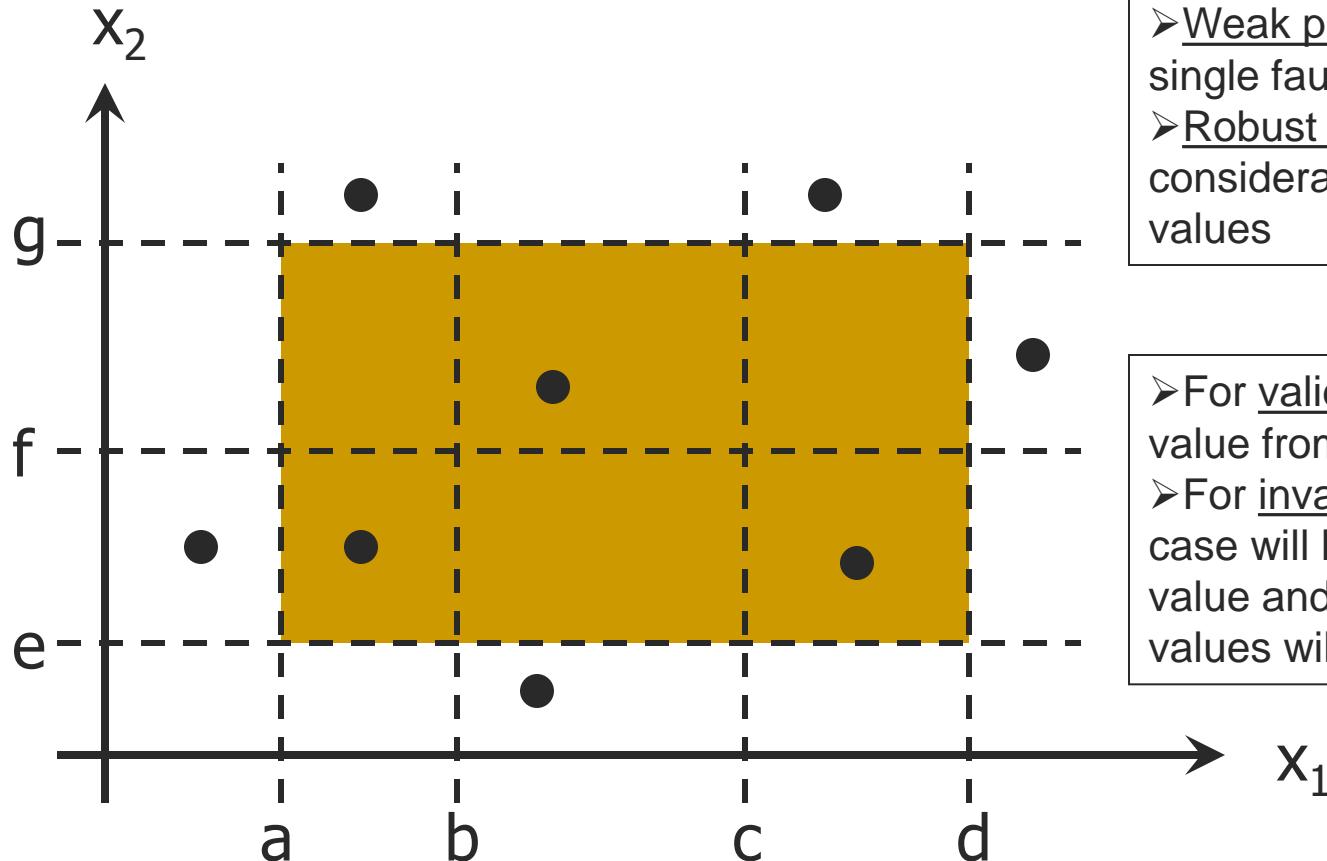
- Weak part refers to the single fault assumption
- Weak Normal ECT is accomplished by using one variable from each equivalence class (interval) in a test case

# Strong Normal ECT



- Strong Normal ECT is based on the multiple fault assumptions
- We need test cases from each element of the Cartesian product of the equivalence classes

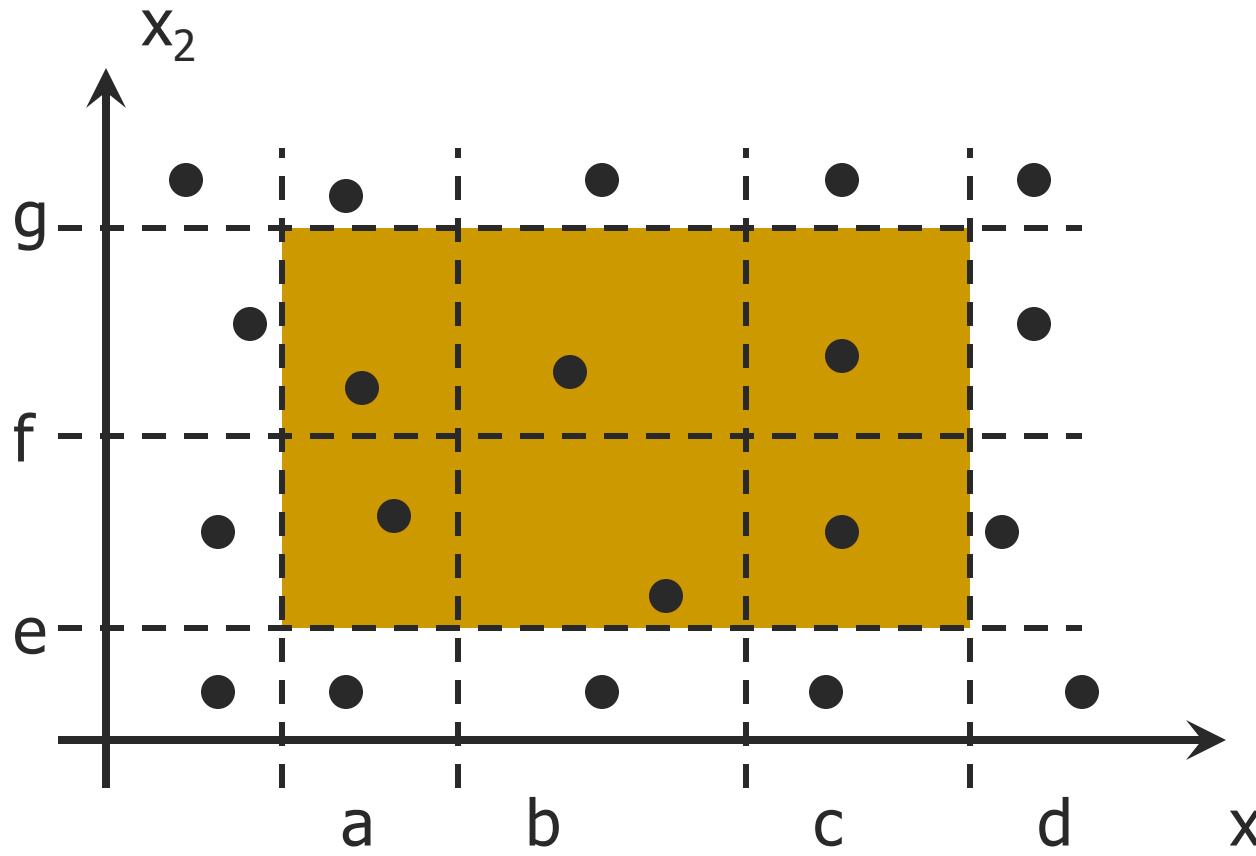
# Weak Robust ECT



➤ Weak part refers to the single fault assumption  
➤ Robust part comes from consideration of invalid values

➤ For valid inputs, use one value from each valid class  
➤ For invalid inputs, a test case will have one invalid value and the remaining values will all be valid

# Strong Robust ECT



➤ Robust part comes from consideration of invalid values  
➤ Strong part refers to the multiple fault assumption

➤ Test cases are obtained from each element of the Cartesian product of all the equivalence classes as shown in the figure

# Selection of ECT test cases

- The inputs are mechanically selected from the approximate middle of the corresponding equivalence class (interval)
- A mechanical selection of input values makes no consideration of the domain knowledge
  - This is a problem with “automatic” test case generation

# Limitations of Robust ECT

- Often, the specifications do not define what the expected output for an invalid test case should be
  - One could argue that this is a deficiency of the specs
  - Testers spend a lot of time defining expected outputs for these cases
- Strongly typed languages eliminate the need for the consideration of invalid inputs
  - ECT was introduced during the time when languages such as FORTRAN and COBOL were dominant and this error was common

# Triangle Problem: Output (Range) Equivalence Classes

- Four possible outputs:
  - Not a Triangle
  - Isosceles
  - Equilateral
  - Scalene
- We can use these to identify the following output (range) equivalence classes:

R1= {  $\langle a, b, c \rangle$ : the triangle with sides a, b, c, is equilateral}

R2= {  $\langle a, b, c \rangle$ : the triangle with sides a, b, c, is isosceles}

R3= { $\langle a, b, c \rangle$ : the triangle with sides a, b, c, is scalene}

R4= { $\langle a, b, c \rangle$ : sides a, b, c do not form a triangle}

# Weak Normal Test Cases

Test Case	a	b	c	Expected Output
WN1	5	5	5	Equilateral
WN2	2	2	3	Isosceles
WN3	3	4	5	Scalene
WN4	4	1	2	Not a Triangle

Weak Normal ECT contains one test case from each equivalence class

Note: the Strong Normal ECT is identical with the Weak Normal ECT because no valid subintervals of variables a, b and c exist

# Weak Robust Test Cases

Test Case	a	b	c	Expected Output
WR1	-1	5	5	a not in range
WR2	5	-1	5	b not in range
WR3	5	5	-1	c not in range
WR4	201	5	5	a not in range
WR5	5	201	5	b not in range
WR6	5	5	201	c not in range

- In addition to the weak part (previous WN1-4 test cases)
- The robust part considers some invalid values for variables a, b and c
- Each test case has one invalid value and the remaining values are all valid

# Strong Robust Test Cases

Test Case	a	b	c	Expected Output
SR1	-1	5	5	a not in range
SR2	5	-1	5	b not in range
SR3	5	5	-1	c not in range
SR4	-1	-1	5	a, b not in range
SR5	5	-1	-1	b, c not in range
SR6	-1	5	-1	a, c not in range
SR6	-1	-1	-1	a, b, c not in range

➤ Only a subset (a “corner” of the cube (3D-space) of the additional strong robust equivalence class test cases are shown in the table

# Triangle Problem:

## Input (Domain) Equivalence Classes

A richer set of test cases can be obtained if we base equivalence classes on the input domain as follows (R1-R4 vs. D1-D11):

D1= { $\langle a,b,c \rangle \mid a = b = c$ }      *all sides are equal*

D2= { $\langle a,b,c \rangle \mid a = b, a \neq c$ }      *exactly one pair is equal*

D3= { $\langle a,b,c \rangle \mid a = c, a \neq b$ }

D4= { $\langle a,b,c \rangle \mid b = c, a \neq b$ }

D5= { $\langle a,b,c \rangle \mid a \neq b, a \neq c, b \neq c$ }      *none is equal*

D6= { $\langle a,b,c \rangle \mid a > b+c$ }      *values don't form a valid triangle*

D7= { $\langle a,b,c \rangle \mid a = b+c$ }

D8= { $\langle a,b,c \rangle \mid b > a+c$ }

D9= { $\langle a,b,c \rangle \mid b = a+c$ }

D10= { $\langle a,b,c \rangle \mid c > a+b$ }

D11= { $\langle a,b,c \rangle \mid c = a+b$ }

# The NextDate Application: Input (Domain) Equivalence Classes

- The NextDate function returns the date of the day after the input date
- It uses 3 input parameters: *month*, *day*, *year* which have intervals

## Invalid input:

M1= {month < 1}  
M2= {month > 12}

D1= {day < 1 }  
D2= {day > 31}

Y1= {year < 1812}  
Y2= {year > 2012}

## ***Useful criteria for choosing equivalence classes:***

- If the input date is not at the end of the month the program will simply increment the day value, however
- If the input date is at the end of the month, it will force the program to change the day to 1 and increment the month
- If the input date is at the end of the year, it will force the program to reset both the day and the month to 1 and increment the year
- The leap year makes determining the last day of the month interesting

# Weak Normal Test Cases

Test Case	Month	Day	Year	Expected Output
WN1	6	14	2000	6/15/2000
WN2	7	29	1996	7/30/1996
WN3	2	30	2002	Invalid input date
WN4	6	31	2000	Invalid input date

One test case from each equivalence class

## Equivalence classes:

M1= {month | month has 30 days}  
 M2= {month | month has 31 days}  
 M3= {month | month is February}

D1= {day |  $1 \leq day \leq 28$ }  
 D2= {day | day = 29}  
 D3= {day | day = 30}  
 D4= {day | day=31}

Y1= {year | year = 2000 special treatment}  
 Y2= {year | year is a leap year}  
 Y3= {year | year is a common year}

# NextDate Discussion

- There are 36 strong normal test cases: (3x4x3) i.e. M1-M3 x D1-D4 x Y1-Y3 (see previous slide)
- Some redundancy creeps in
  - Testing February 30 and 31 for three different types of years seems unlikely to reveal errors
- There are 150 strong robust test cases (adding the invalid input – see previous slide) : M1-M5 x D1-D6 x Y1-Y5

# ECT for the Commission Problem

- More typical of commercial computing
- Contains a mix of computation and decision making
  - See problem statement in chapter 2

Valid classes of the input variables are:

$$L1 = \{\text{locks} : 1 \leq \text{locks} \leq 70\}$$

$$L2 = \{\text{locks} = -1\}$$

$$S1 = \{\text{stocks} : 1 \leq \text{stocks} \leq 80\}$$

$$B1 = \{\text{barrels} : 1 \leq \text{barrels} \leq 90\}$$

Invalid classes of the input variables are:

$$L3 = \{\text{locks} : \text{locks}=0 \text{ OR } \text{locks} < -1\}$$

$$L4 = \{\text{locks} : \text{locks} > 70\}$$

$$S2 = \{\text{stocks} : \text{stocks} < 1\}$$

$$S3 = \{\text{stocks} : \text{stocks} > 80\}$$

$$B2 = \{\text{barrels} : \text{barrels} < 1\}$$

$$B3 = \{\text{barrels} : \text{barrels} > 90\}$$

See chapter 6 for the detailed equivalence class test cases

# Guidelines and Observations

- Equivalence Class Testing is appropriate when input data is defined in terms of intervals and sets of discrete values
- Equivalence Class Testing is strengthened when combined with Boundary Value Testing
- Complex functions, such as the NextDate program, are well-suited for Equivalence Class Testing
- Several tries may be required before the “right” equivalence relation is discovered
- Strong equivalence takes the presumption that variables are independent
  - If that is not the case, redundant test cases may be generated

# In class activity

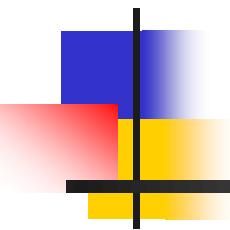
- For the triangle problem create
  - Weak Normal test cases
  - Strong Normal test cases
  - Weak Robust test cases
  - Strong Robust test cases

# Software Testing & Quality Assurance



## *Review of Functional Testing Techniques*

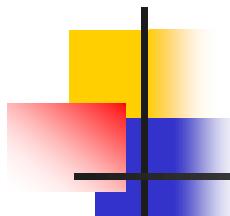
- Boundary Value Testing
- Equivalence Class Testing
- Decision Table-Based Testing
- Testing effort, efficiency and effectiveness issues



# Credits & Readings

The material included in these slides are mostly adopted from the following books:

- *Software Testing: A Craftsman's Approach*, by Paul Jorgensen, CRC PRESS, third edition, ISBN: 0-8493-7475-8
- Cem Kaner, Jack Falk, Hung Q. Nguyen, "*Testing Computer Software*" Wiley (see also <http://www.testingeducation.org/>)
- Paul Ammann and Jeff Offutt, "*Introduction to Software Testing*", Cambridge University Press
- Glen Myers, "*The Art of Software Testing*"



# Functional Testing

- We saw three types of functional testing
  - Boundary Value Testing
  - Equivalence Class Testing
  - Decision Table-Based Testing
- The common thread among these techniques is that they all view a program as a mathematical function that maps its inputs to its outputs
- We now look at questions related to
  - Testing effort
  - Testing efficiency
  - Testing effectiveness

# Boundary Value Test Cases

Test Case	a	b	c	Expected Output
1	100	100	1	Isosceles
2	100	100	2	Isosceles
3	100	100	100	Equilateral
4	100	100	199	Isosceles
5	100	100	200	Not a Triangle
6	100	1	100	Isosceles
7	100	2	100	Isosceles
8	100	100	100	Equilateral
9	100	199	100	Isosceles
10	100	200	100	Not a Triangle
11	1	100	100	Isosceles
12	2	100	100	Isosceles
13	100	100	100	Equilateral
14	199	100	100	Isosceles
15	200	100	100	Not a Triangle

- For each variable, select five values (keep the others constant)
  - Minimum
  - Just above the minimum
  - Nominal
  - Just below the maximum
  - Maximum

# Equivalence Class Test Cases

Test Case	a	b	c	Expected Output
WN1	5	5	5	Equilateral
WN2	2	2	3	Isosceles
WN3	3	4	5	Scalene
WN4	4	1	2	Not a Triangle
WR1	-1	5	5	a not in range
WR2	5	-1	5	b not in range
WR3	5	5	-1	c not in range
WR4	201	5	5	a not in range
WR5	5	201	5	b not in range
WR6	5	5	201	c not in range

- Weak Normal test cases (WN1-WN4) are derived by using one variable from each equivalence class (i.e. equilateral, isosceles, scalene, invalid)
- Weak Robust test cases (WR1-WR6) are derived by considering one invalid input and keeping the rest valid (i.e. a, b, c not in range)

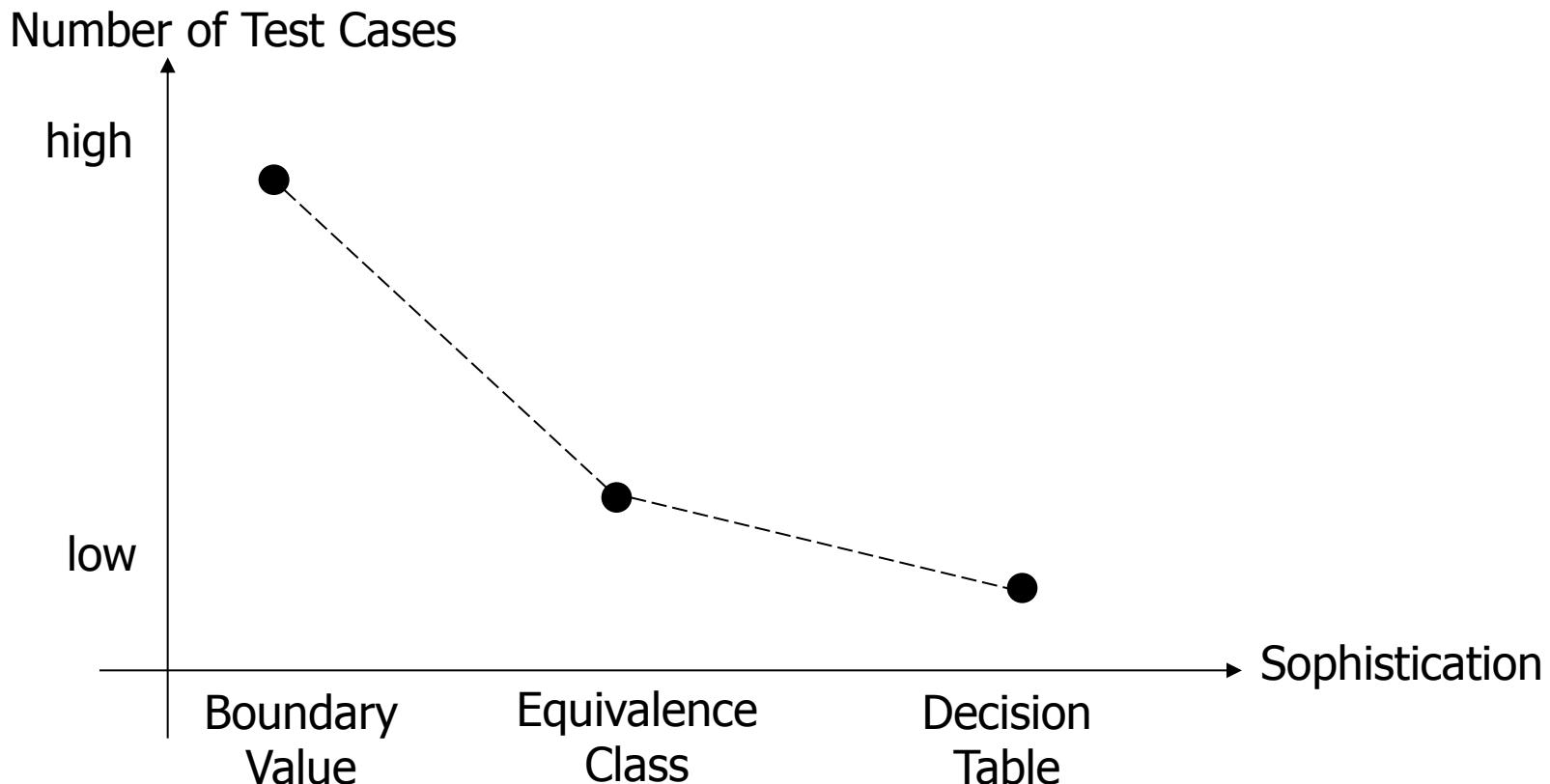
# Decision Table Test Cases

Test Case	a	b	c	Expected Output
DT1	4	1	2	Not a Triangle
DT2	1	4	2	Not a Triangle
DT3	1	2	4	Not a Triangle
DT4	5	5	5	Equilateral
DT5	?	?	?	Impossible
DT6	?	?	?	Impossible
DT7	2	2	3	Isosceles
DT8	?	?	?	Impossible
DT9	2	3	2	Isosceles
DT10	3	2	2	Isosceles
DT11	3	4	5	Scalene

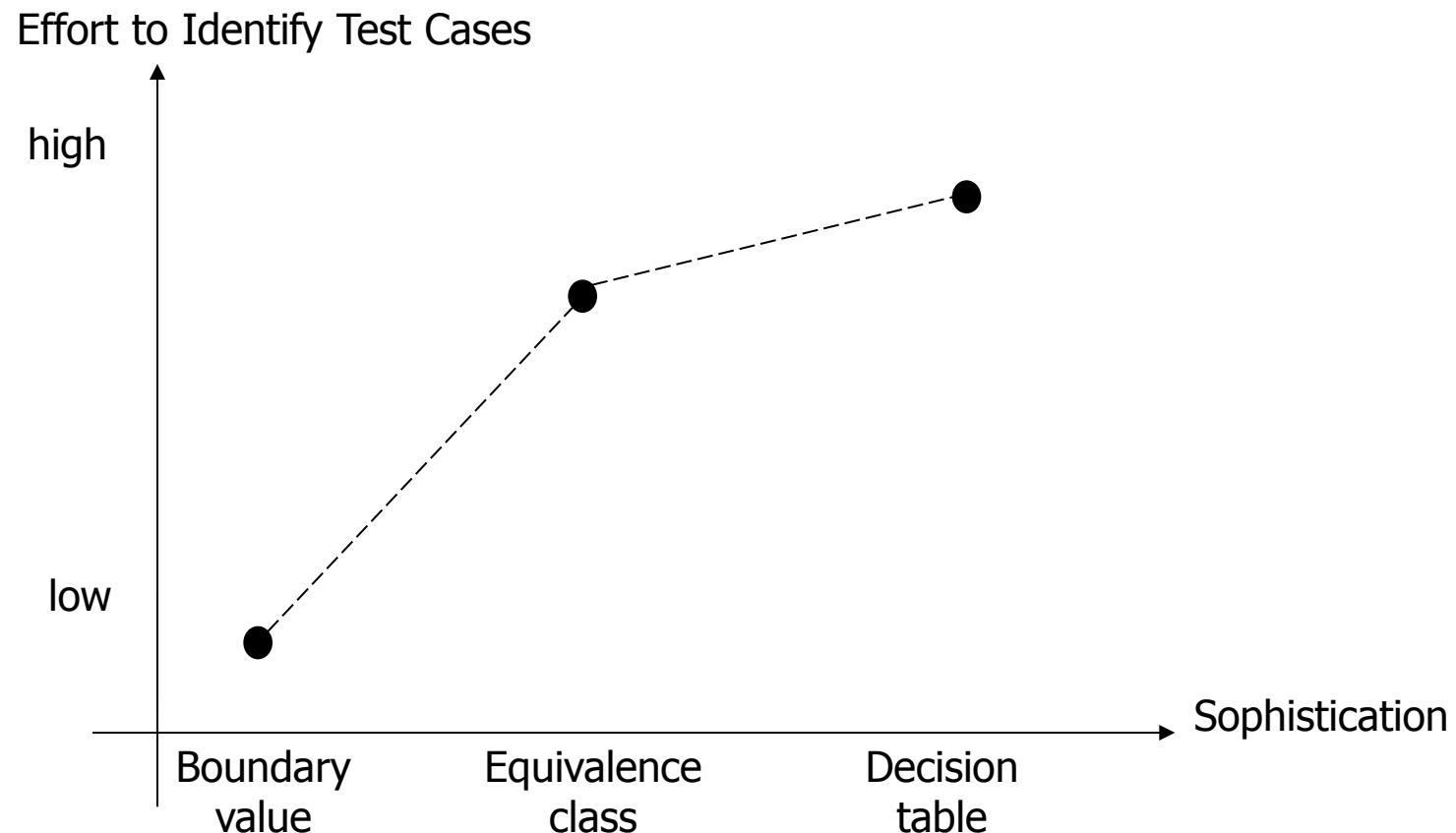
C1: $a < b+c$ ?	F	T	T	T	T	T	T	T	T	T	T
C2: $b < a+c$ ?	-	F	T	T	T	T	T	T	T	T	T
C3: $c < a+b$ ?	-	-	F	T	T	T	T	T	T	T	T
C4: $a = b$ ?	-	-	-	T	T	T	T	F	F	F	F
C5: $a = c$ ?	-	-	-	T	T	F	F	T	T	F	F
C6: $b = c$ ?	-	-	-	T	F	T	F	T	F	T	F
A1: Not a Triangle	X	X	X								
A2: Scalene											X
A3: Isosceles							X		X	X	
A4: Equilateral				X							
A5: Impossible					X	X		X			

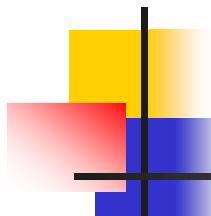
➤From the decision table we derive 11 test cases that correspond to actions taken (denoted by X in the decision table)

# Testing Effort<sup>1</sup>



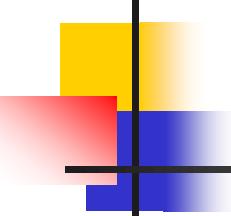
# Testing Effort<sup>2</sup>





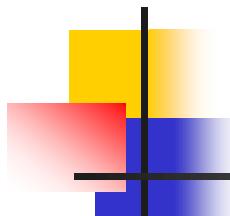
# Testing Effort<sup>3</sup>

- Boundary Value Testing has no recognition of data or logical dependencies
  - Mechanical generation of test cases
- Equivalence Class Testing takes into account data dependencies
  - More thought and care is required to define the equivalence classes
  - Mechanical generation after that



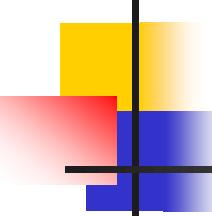
# Testing Effort<sup>4</sup>

- The decision table technique is the most sophisticated, because it requires that we consider both data and logical dependencies
  - Iterative process
  - Allows manual identification of redundant test cases
- Tradeoff between test identification effort and test execution effort



# Testing Efficiency

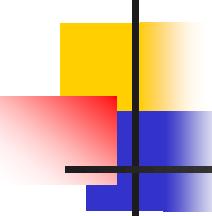
- Fundamental limitations of functional testing
  - Gaps of untested functionality
  - Redundant tests
- Testing efficiency question: *How can we create a set of test cases that is "just right"?*
  - Hard to answer. Can only rely on the general knowledge that more sophisticated techniques, such as decision tables, are usually more efficient
  - Structural testing methods will allow us to define more interesting metrics for efficiency



# Testing Efficiency Comparison

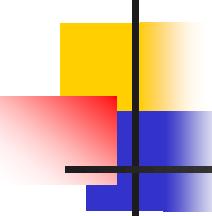
For the *NextDate* program

- The worst case boundary analysis generated 125 cases
  - These are fairly redundant (check January 1 for five different years, only a few February cases but none on February 28, and February 29, and no major testing for leap years)
- The strong equivalence class test cases generated 36 test cases 11 of which are impossible
- The decision table technique generated 22 test cases (fairly complete)



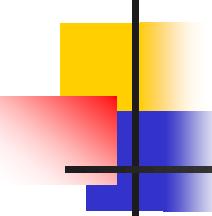
# Testing Effectiveness

- How effective is a method or a set of test cases for finding faults present in a program?
- Hard to answer because
  - It presumes we know all faults in a program
  - It is impossible to prove that a program is free of faults
- The best we can do is to work backward from fault types
- Given a fault type we can choose testing methods that are likely to reveal faults of that type
  - Use knowledge related to the most likely kinds of faults to occur
  - Track kinds and frequencies of faults in the software applications we develop



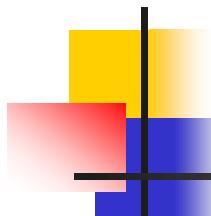
# Guidelines<sup>1</sup>

- Kinds of faults may reveal some pointers as to which testing method to use
- If we do not know the kinds of faults that are likely to occur in the program then the attributes most helpful in choosing functional testing methods are:
  - Whether the variables represent physical or logical quantities
  - Whether or not there are dependencies among variables
  - Whether single or multiple faults are assumed
  - Whether exception handling is prominent



# Guidelines<sup>2</sup>

- If the variables refer to physical quantities and/or are independent then we consider
  - domain testing or
  - equivalence testing
- If the variables are dependent then we consider
  - decision table testing
- If the single-fault assumption is plausible to assume then consider
  - boundary value analysis and
  - robustness testing

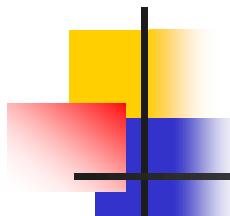


# Guidelines<sup>3</sup>

- If the multiple-fault assumption is plausible to assume then we consider
  - worst case testing
  - robust worst case testing
  - decision table testing
- If the program contains significant exception handling then we consider
  - robustness testing and
  - decision table testing
- If the variables refer to logical quantities, then consider
  - equivalence class testing and
  - decision table testing

# Functional Testing Decision Table

C1: Variables (P=Physical, L=Logical)?	P	P	P	P	P	L	L	L	L	L
C2: Independent Variables?	Y	Y	Y	Y	N	Y	Y	Y	Y	N
C3: Single fault assumption?	Y	Y	N	N	-	Y	Y	N	N	-
C4: Exception handling?	Y	N	Y	N	-	Y	N	Y	N	-
A1: Boundary value analysis		X								
A2: Robustness testing	X									
A3: Worst case testing				X						
A4: Robust worst case testing			X							
A5: Weak robust equivalence testing	X		X			X		X		
A6: Weak normal equivalence testing	X	X				X	X			
A7: Strong normal equivalence testing			X	X	X			X	X	X
A8: Decision table					X					X



# Case Study

- Apply and compare functional testing methods on the following example (page 123):
  - Consider an insurance premium program that computes the semi-annual car insurance premium based on two parameters
    - The policy holder's age
    - Driving record
  - It uses the following formula:

Premium = BaseRate \* ageMultiplier - safeDrivingReduction

ageMultiplier is a function of the policyholder's age

safeDrivingReduction is given when traffic points are below an age cutoff

# Software Testing & Quality Assurance

## *Introduction*

- ◆ Basic terminology and definitions
- ◆ Motivation, importance and limitations
- ◆ First look at some systematic testing techniques and tools

# Credits & Readings

◆ The material included in these slides are mostly adopted from the following books:

- *Software Testing: A Craftsman's Approach*, by Paul Jorgensen, CRC PRESS, third edition, ISBN: 0-8493-7475-8
- Cem Kaner, Jack Falk, Hung Q. Nguyen, "*Testing Computer Software*" Wiley (see also <http://www.testingeducation.org/>)
- Paul Ammann and Jeff Offutt, "*Introduction to Software Testing*", Cambridge University Press
- Glen Myers, "*The Art of Software Testing*"

# Why do we test software?

- ◆ What is your opinion?
- ◆ What would an IT professional say?
- ◆ Can you share your findings with the rest of the class?

# IT professionals say they test software in order to:

- ◆ Check programs against specifications
- ◆ Determine user acceptability
- ◆ Gain confidence that it works
- ◆ Show that a program performs correctly
- ◆ Demonstrate that errors are not present
- ◆ Understand the limits of performance
- ◆ Learn what a system is not able to do
- ◆ Insure that a system is ready for use
- ◆ Evaluate the capabilities of a system
- ◆ Verify documentation
- ◆ Find important bugs, to get them fixed
- ◆ Check interoperability with other products
- ◆ Help managers make ship/no-ship decisions
- ◆ Block premature product releases
- ◆ Minimize technical support costs
- ◆ Assess conformance to specification
- ◆ Conform to regulations
- ◆ Minimize safety-related lawsuit risk
- ◆ Convince oneself that the job is finished
- ◆ Find safe scenarios for use of the product

NOTE: Different objectives require different testing strategies and will yield different tests, different test documentation and different test results.

# Definition of software testing

“The process of exercising or evaluating a system by manual or automatic means to verify that it satisfies specified requirements or to identify differences between expected and actual results” [ANSI/IEEE std 729-1983]

# Course learning objectives

- ◆ Understand the motivation, importance and limitations of systematic testing
- ◆ Learn and understand the pros and cons of particular testing techniques and the situations in which they apply
- ◆ Practice with modern industrial best-practices, testing tools and frameworks
- ◆ Learn how to craft and execute test cases for large software systems
- ◆ Learn how to produce quality problem reports

# Basic definitions

- ◆ What is an *error*?
  - “It is a *mistake*, people make them” [Jorgensen]
  - Error of *omission*: something is missing
  - Error of *commission*: something is incorrect
- ◆ Other views
  - “A software error is present when the program doesn’t do what its user reasonably expects it to do” [Myers]
  - “A mismatch between the program and its specifications, if and only if the specifications exist and are correct” [Kaner]
  - “There can never be an absolute definition for *bugs*, nor an absolute determination of their existence. The extend to which a program has bugs is measured by the extend to which it fails to be useful” [Beizer]

# Basic definitions--continued

- ◆ What is a *fault*?
  - It is the result of an error, or the representation of an error (e.g. inaccurate requirements text, erroneous design, buggy source code etc.)
  - Synonyms used: *defect*, *bug*
- ◆ What is a *failure*?
  - The program's actual incorrect or missing behavior under the error-triggering conditions
  - A failure occurs when a fault executes
- ◆ A fault won't yield a failure without the *conditions* that trigger it
- ◆ What is an *incident*?
  - An incident is the symptom that alerts/indicates the occurrence of a failure (note: when a failure occurs, it may not be always apparent to the user/tester)

# Example

- ◆ Here's a defective program
  - INPUT A
  - INPUT B
  - PRINT A / B
- ◆ What is the error? What is the fault?
- ◆ What is the critical condition?
- ◆ What will we see as the incident of the failure?

# Example-answers

- ◆ **Error** (mistake): the mistake that the programmer made when he/she forgot to handle the special case where  $B=0$  (division by 0 not allowed)
- ◆ **Fault** (representation of error): the resulted buggy/faulty source code
- ◆ **Critical Condition**: if  $B=0$  then it triggers the error
- ◆ **Failure** (incorrect behavior): execution at halt or crash
- ◆ **Incident** (symptom): non-responsive screen (or error message displayed)

# Basic definitions--continued

- ◆ What is a *test*?

- The act of investigating/exercising software with an intent to
  - ◆ Find failures
  - ◆ Demonstrate correctness
  - ◆ Expose quality-related information

- ◆ What is a *test case*?

- Set of inputs and outputs
- Has an identity and is associated with a program behavior

# When are errors introduced?

## ◆ System Life Cycle (SLC)

- Requirements (*here*)
- Design (*here*)
- Implementation (*here*)
- Maintenance (*here*)

## ◆ Snowball effect

(errors introduced early can propagate into later stages and become deeper and harder to find)

## ◆ Testing Life Cycle (TLC)

- Fault detection
- Fault classification
- Fault isolation
- Fault resolution (*errors are possible here too!*)
- *See figure 1.1 in Jorgensen*

# Other related terminology\*

*Caution: related terminology found in the literature may vary depending on the context used*

- ◆ Software Problem = a discrepancy between a delivered artifact of a SD phase and its:
  - documentation
  - the product of an earlier phase
  - user requirements
- ◆ Problem Status = a problem can be
  - *open* (i.e. the problem has been reported)
  - *closed-available* (i.e. a tested fix is available)
  - *closed* (i.e. a tested fix has been installed)
- ◆ Error = A problem found during the review of the phase where it was introduced
- ◆ Defect = A problem found later than the review of the phase where it was introduced
- ◆ Fault = Errors and defects are considered faults
- ◆ Failure = Inability of software to perform its required function
  - It can be caused by a defect encountered during software execution (i.e. testing and operation)
  - When a failure is observed, problem reports are created and analyzed in order to identify the defects that are causing the failure

\*Motorola SQA and Measurements Program (Daskalantonakis, 1996)

# A broad taxonomy of software errors\*

- ◆ User Interface errors
  - customers complain about serious human-factor errors as much they complain about crashes
- ◆ Error handling
  - Failure to detect and handle an error in a reasonable way
- ◆ Boundary-related errors
  - Typically numeric values and memory size
- ◆ Calculation errors
  - Incorrect arithmetic due to truncation, misinterpreted formulas, incorrect algorithms
- ◆ Errors in handling or interpreting data
  - Passing corrupted data from one module to another
- ◆ Race conditions
  - Handling of asynchronous events (e.g. event B has to happen after event A for some reason)
- ◆ Load conditions
  - Program misbehaves when overloaded
- ◆ Hardware
  - Programs send bad data to devices, and try to use devices that are busy or don't exist
- ◆ Source and version control
  - Old problems reappear when we link an older version with the latest one
- ◆ Documentation
  - Poor documentation leads to mistrust of the software's capabilities
- ◆ Testing errors
  - Testers make mistakes too!

\*Cem Kaner et al, "Testing Computer Software" (<http://www.testingeducation.org/>)

# Anatomy of a test case

- ◆ Test case ID
- ◆ Purpose and objectives
- ◆ Inputs
  - Preconditions
  - Actual data inputs that were identified by some testing method
- ◆ Expected outputs
  - Post conditions
  - Actual outputs

# Views of testing

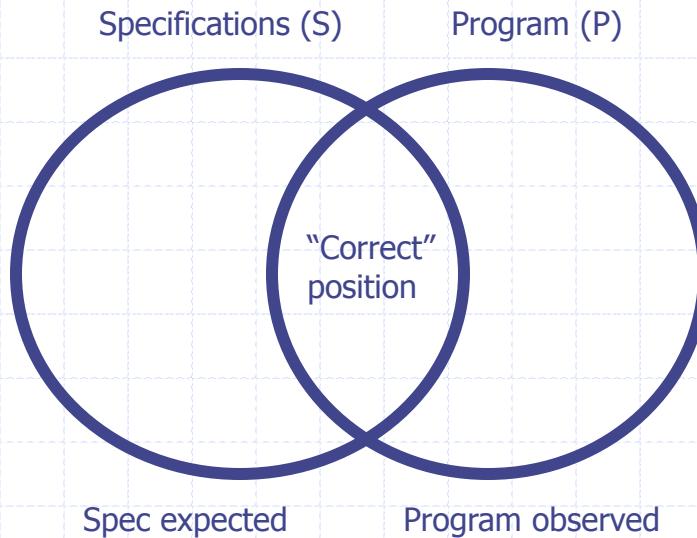
## ◆ Behavioral view

- It considers “what the code does i.e. how it behaves”

## ◆ Structural view

- It focuses on “what it is”
- Base documents are written by and for developers – the emphasis is on structural info rather than behavioral info

# Specified and implemented program behaviors



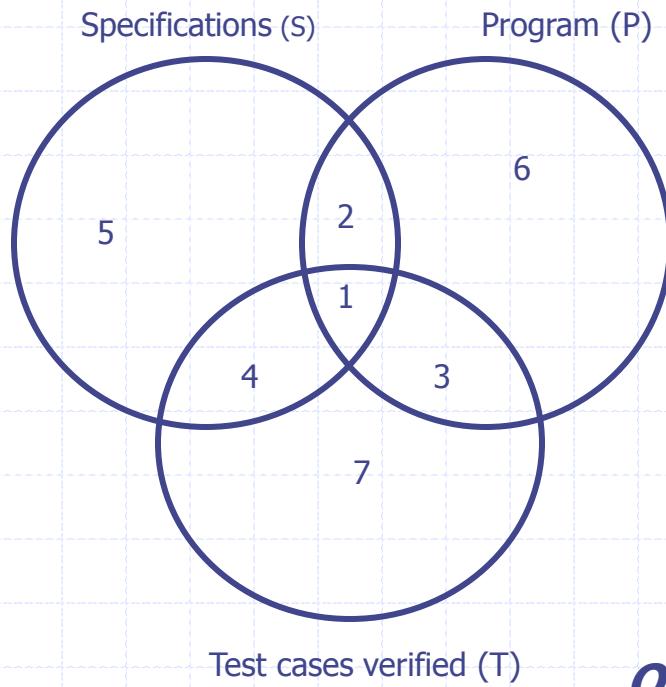
The Venn diagram shows

- 1) certain specified behaviors are not programmed
  - ◆ Faults of omission
- 2) certain programmed behaviors were never specified
  - ◆ Faults of commission

The *football* shape is the correct position

- Behaviors are both specified and implemented

# Specified, implemented and tested program behaviors



Regions 2 and 5 are specs that are never tested

Regions 1 and 4 are specified and tested

Regions 3 and 7 are test cases that are not specified

Regions 2 and 6 are program behaviors that are not tested

Regions 1 and 3 are program behaviors that are tested

Regions 4 and 7 are test cases that correspond to non programmed behaviors

**Our goal:** How can we make region 1 larger?

# A simple example

## Specs for an ADDER:

- Adds two numbers that the user enters
- Each number should be one or two digits
- The program echoes the entries, then prints the sum.
- Press <ENTER> after each number

## test run output

? 4

4

? 2

2

6

?

# First reactions? Observations?

- ◆ Can you list some potential areas for improvement on this application?

# First observations

- ◆ No on-screen instructions
- ◆ Nothing shows what this program is or does
- ◆ You don't even know if you run the right program
- ◆ How do you stop the program?
- ◆ The 6 should probably line up with the 4 and 2

# A first set of test cases

Can you suggest some test cases you would like to try in order to test this application?

# A first set of test cases

99 + 99

-99 + -99

99 + 56

56 + 99

99 + -14

-14 + 99

38 + -99

-99 + 38

-99 + -43

-43 + -99

9 + 9

0 + 0

0 + 23

-23 + 0

# Choosing test cases

- ◆ Not all test cases are significant
- ◆ Impossible to test everything (this simple program has tens of thousands of possible different test cases)
- ◆ If you expect the same result from two tests, they belong to the same class. Use only one of them
- ◆ When you choose representatives of a class for testing, pick the ones most likely to fail

# Further test cases

100 + 100

<Enter> + <Enter>

123456 + 0

1.2 + 5

A + b

<CTRL-C> + <CTRL-D>

<F1> + <Esc>

# Other things to consider

- ◆ Test cases with extra whitespace
- ◆ Test cases involving <Backspace>
- ◆ The order of the test cases might matter
  - E.g. <Enter> + <Enter>

# What is functional testing?

- ◆ Based on a view that any program can be considered to be a function that maps values from its input domain to values onto its output range
- ◆ *Black Box* testing
  - Systems are considered to be black boxes
  - Content or implementation is unknown
  - The function of the black box is completely understood in terms of its inputs and outputs

# What are the advantages?

- ◆ Functional test cases have two distinct advantages:
  - 1) They are independent of how the software is implemented
    - ♦ If implementation changes the test cases are still useful
  - 2) Test case development can occur in parallel with the implementation, thereby reducing overall project development interval

# What are the disadvantages?

- ◆ Functional test cases have the following disadvantages:
  - 1) Significant redundancies may exist among test cases
  - 2) There is a possibility of gaps of untested software

# What is structural testing?

- ◆ *White box or clear box* testing
- ◆ Implementation is known and used to identify test cases
- ◆ Testers identify test cases based on how the function is actually implemented

# Which approach is better?

- ◆ Neither approach alone is sufficient; both are needed
- ◆ The two approaches are complementary

# Software Quality Assurance (SQA) vs. Software Testing

◆ What is the difference in your opinion?

# Software Quality Assurance (SQA) vs. Software Testing

- ◆ What is a *process*?
  - how we do something
- ◆ What is a *product*?
  - The end result of a process
- ◆ Testing is clearly product-oriented
  - It finds *faults* in a product
- ◆ SQA tries to improve the product by first improving the process
  - It wants to fix *errors* in the development process

# Software anomalies\*

## ◆ Fault Severity

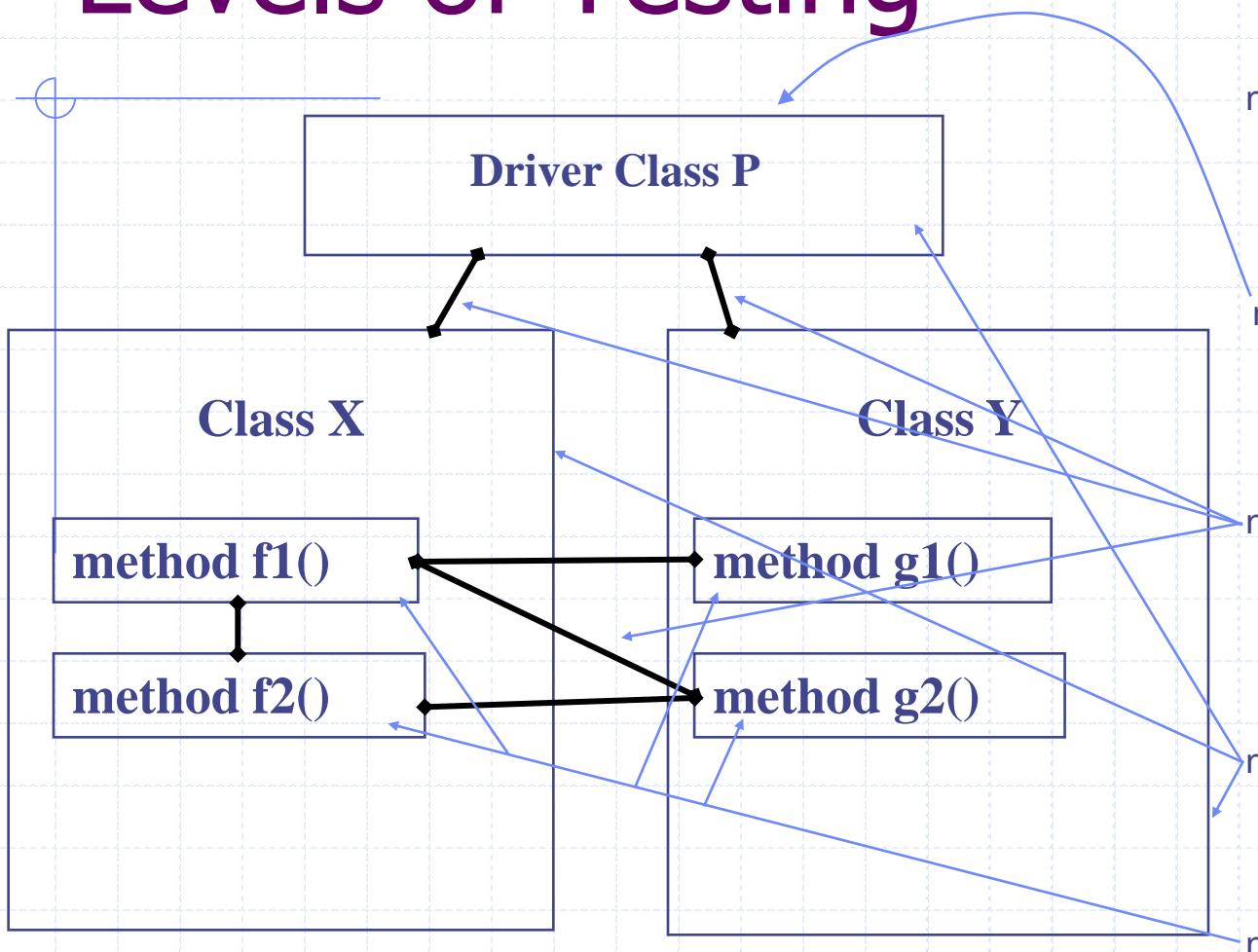
- Mild
- Moderate
- Annoying
- Disturbing
- Serious
- Very serious
- Extreme
- Intolerable
- Catastrophic
- Infectious

## ◆ Fault types

- Input/output
- Logic
- Computation
- Interface
- Data

\*ANSI/IEEE std 1044-1993 "Classification of Software Anomalies"

# Levels of Testing\*



n **Acceptance testing:** Is the software acceptable to the user?

n **System testing:** Test the overall functionality of the system

n **Integration testing:** Test how modules interact with each other

n **Module testing:** Test each class, file, module or component

n **Unit testing:** Test each unit (method) individually

\*Paul Ammann and Jeff Offutt,  
*"Introduction to Software Testing"*,  
Cambridge University Press

# Granularities of testing

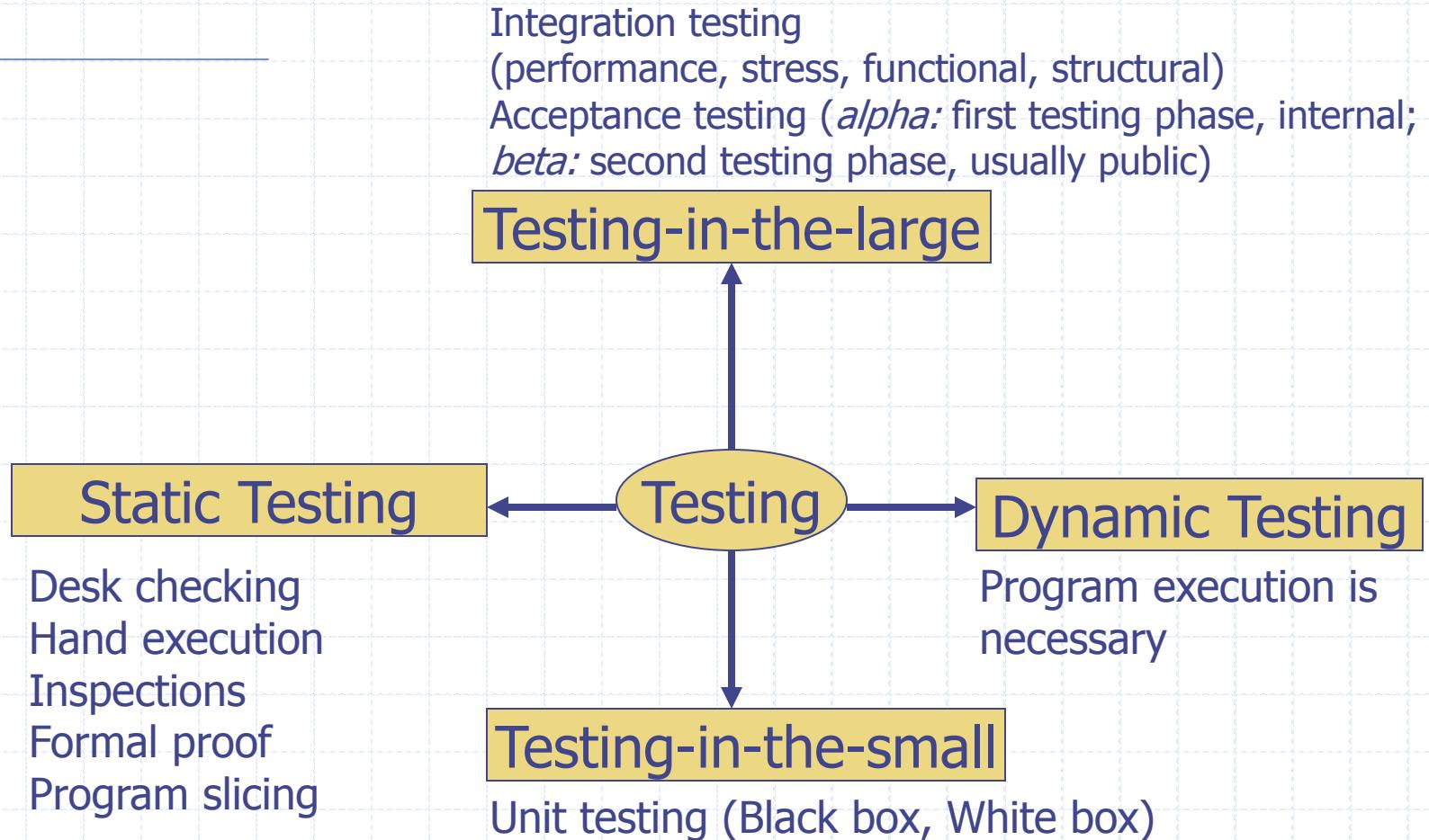
## ◆ Testing-in-the-small

- It refers to the testing activities of a single module/unit

## ◆ Testing-in-the-large

- It refers to the testing activities of groups of modules/units up-to and including the entire system (integration testing techniques)

# Other dimensions of software testing



# Typical steps involved in testing techniques

1. Select *what* is to be measured by the test
2. Decide *how* is to be tested (walkthrough, inspection, proof, BB/WB, etc.)
3. Develop a *test-bed* (a set of test cases)
4. Create the *test oracle* (predicted results for a set of test cases)
5. Execute the test cases
  - Prepare the *test harness* software
  - Compare the results with the test oracle
  - Identify any discrepancies between the predicted results and the actual results

# What is white box testing?

It is a test-bed design method, which uses the control-flow structure of the program to derive the necessary test cases.

# How does WB testing work?

- ◆ *Coverage* is a measure of how much of a module or system has been exercised (executed) by a test or series of tests
- ◆ To obtain coverage, we make sure every statement in the source code being tested, is executed at least once
  - Segments (sequential)
  - Decisions (if-then-else is executed twice)
  - Loops (provide test cases for skipping the execution of a loop, execute the body exactly once, or more than once)

# Tools

## *Eclipse*

- ◆ IDE for Java development
- ◆ Works seamlessly with Junit for unit testing
- ◆ Open source – Download from [www.eclipse.org](http://www.eclipse.org)
- ◆ Try it with your own Java code

## *Junit*

- ◆ A framework for automated unit testing of Java code
- ◆ Written by Erich Gamma and Kent Beck
- ◆ Download from [www.junit.org](http://www.junit.org)
- ◆ Related technical manuals available on web

# What is black box testing?

It is a test-bed design method, which focuses on the behavioral requirements of the program to derive the necessary test cases.

# How does BB testing work?

## ◆ *Equivalence partitioning*

- The goal is to reduce the number of necessary test cases to a manageable number
- ◆ Every input condition is divided into a number of *equivalence classes*.
  - Each class consists of a set of data items all of which are similar to each other on some relevant dimension
  - Example 1: If the input data is supposed to be valid across a range of values (e.g. an age, a price), there will be a minimum of three equivalence classes:
    - ◆ *Below, within and above* the range
  - Example 2: If the input data is valid when it is a value from a set of discrete or nominal values, (e.g. letter grades) there will be two equivalence classes:
    - ◆ One with valid discrete values
    - ◆ One with any other input values

# Glen Myers triangle problem

“The program reads three integer values from a card. The three values are interpreted as representing the lengths of the sides of a triangle. The program prints a message that states whether the triangle is scalene, isosceles, or equilateral.”

*[Taken From Glen Myers, "The Art of Software Testing"]*

# Classes of test cases for the triangle problem

- ◆ Can you find some candidate categories of test cases for this problem?

## Hint: Refined Specs

- ◆ Input: Three integers, a, b, c, the lengths of the side of a triangle
- ◆ Output: Scalene, isosceles, equilateral, or invalid triangle

# *Reminder: triangle properties*

- ◆ No side may have a length of zero
- ◆ Sum of two sides must be greater than the other side of the triangle
- ◆ Equilateral triangle: all three sides are equal
- ◆ Isosceles triangle: any two sides are equal
- ◆ Scalene triangle: all sides are unequal

# Classes of test cases for the triangle problem

- ◆ Valid triangle:
  - Test case for a valid scalene triangle
  - Test case for a valid equilateral triangle
  - Three test cases for valid isosceles triangles try ( $a=b, c$ ), ( $a, b=c$ ), ( $a=c, b$ )
- ◆ Invalid triangle:
  - All permutations of  $a + b = c$  (e.g.  $a=1, b=2, c=3$ ) try 3 permutations  
 $a+b=c$ ,  $a+c=b$ ,  $b+c=a$
  - All permutations of  $a + b < c$  (e.g.  $a=1, b=2, c=4$ ) try 3 permutations
  - All permutations of  $a = b$  and  $a + b = c$  (e.g.  $a=3, b=3, c=6$ )
- ◆ Invalid side values:
  - One, two or three sides has zero value (5 cases)
  - One side has a negative
  - MAXINT values
  - Non-integer
  - Wrong number of values (too many, too few)

# Caveat

- ◆ The triangle problem typifies some of the *incomplete definitions and assumptions* that impair communication among customers, developers, and testers
- ◆ Glen Myers specification assumes the developers know some details about triangles
  - ***Triangle properties:***
    - sum of two sides must be greater than the other side of the triangle
    - No side may have a length of zero

# Object-oriented implementation

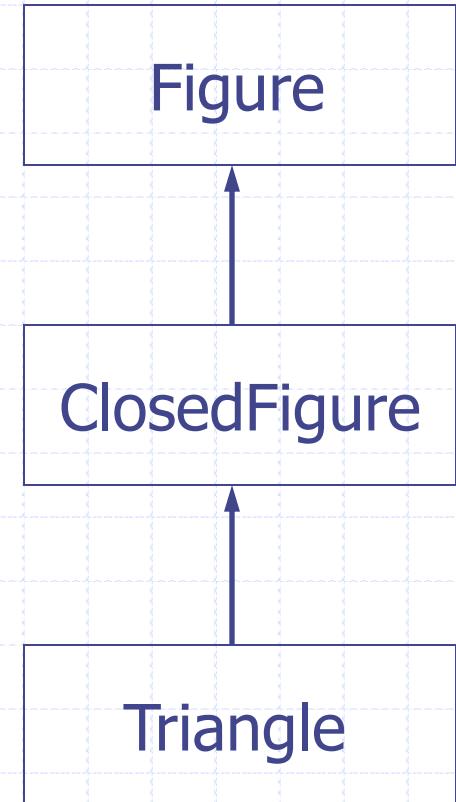
```
class Triangle{  
    public Triangle(LineSegment a, LineSegment b,  
                   LineSegment c)  
    public boolean is_isosceles()  
    public boolean is_scalene()  
    public boolean is_equilateral()  
    public void draw()  
    public void erase()  
}  
class LineSegment {  
    public LineSegment(int x1, int y1,  
                      int x2, int y2)  
}
```

# Extra Tests

- ◆ Is the constructor correct?
- ◆ Is only one of the is \_ \* methods true in every case?
- ◆ Do results repeat, e.g. when running is scalene twice or more?
- ◆ Results change after **draw** or **erase**?
- ◆ Segments that do not intersect

# Inheritance tests

- ◆ Tests that apply to all *Figure* objects must still work for *Triangle* objects
- ◆ Tests that apply to all *ClosedFigure* objects must still work for *Triangle* objects



# What is data-structure based testing?

- ◆ It involves looking at the data structures used by the module with an eye toward errors that might be related to the structure being used
- ◆ For instance, an array (or linked-list) is being passed to the module being tested. Then, four test cases should be designed as follows:
  1. Zero elements in the array/list
  2. Exactly one element in the array/list
  3. One less than the maximum number of elements
  4. Maximum number of elements in the array/list

# Integration testing activities

- ◆ Structure Testing

- Exercise all I/O parameters for each module
  - Exercise all module calls including utility routines

- ◆ Functional Testing

- Make sure that all functions are operational

- ◆ Performance Testing

- Determine the amount of execution time needed to carry out different routine functions (is it within reasonable amount of time?)

- ◆ Stress Testing

- Push integrated units to their limits (amount of load it can take)

# Testing limits

- ◆ Dijkstra: “Program Testing can be used to show the presence of defects, but never their absence”
- ◆ It is impossible to fully test a software system in a reasonable amount of time or money
- ◆ When is testing complete?
  - When you run out of time or money

# Complete testing?

- ◆ What do we mean by "complete testing"?
  - Complete *coverage* i.e. tested every line/path?
  - Testers not finding new bugs?
  - Test plan completed?
- ◆ Complete testing must mean that, at the end of testing, you know there are no remaining unknown bugs
- ◆ After all, if there are more bugs, you can find them if you do more testing. So testing couldn't yet be "complete"

# Complete coverage?

- ◆ What is *coverage*?
  - Extent of testing of certain attributes or pieces of the program, such as *statement* coverage or *branch* coverage or *condition* coverage
  - Extent of testing completed, compared to a population of possible tests
- ◆ Why is complete coverage impossible?
  - Domain of possible inputs is too large
  - Too many possible paths through the program
- ◆ Coverage measurement is a good tool to show *how far* you are from complete testing and not to show *how close* you are to completion

# Time-consuming test-related tasks

- ◆ Analyzing, troubleshooting, and effectively describing a failure
- ◆ Also
  - Designing tests
  - Executing tests
  - Documenting tests
  - Automating tests
  - Reviews, inspections
  - Training other staff

# The infinite set of tests

- ◆ There are enormous numbers of possible tests. To test everything, you would have to:
  - Test every possible input to every variable
  - Test every possible combination of inputs to every combination of variables
  - Test every possible sequence through the program
  - Test every hardware / software configuration, including configurations of servers not under your control
  - Test every way in which any user might try to use the program

# Testing valid inputs

- ◆ There are 39,601 possible valid inputs for the ADDER program
- ◆ In the Triangle example, assuming only integers from 1 to 10, there are  $10^4$  possibilities for a segment, and  $10^{12}$  for a triangle. Testing 1000 cases per second, you would need 317 years!

# Testing invalid inputs

- ◆ The error handling aspect of the system must also be triggered with invalid inputs
- ◆ Anything you can enter with a keyboard must be tried. Letters, control characters, combinations of these, question marks, too long strings etc...

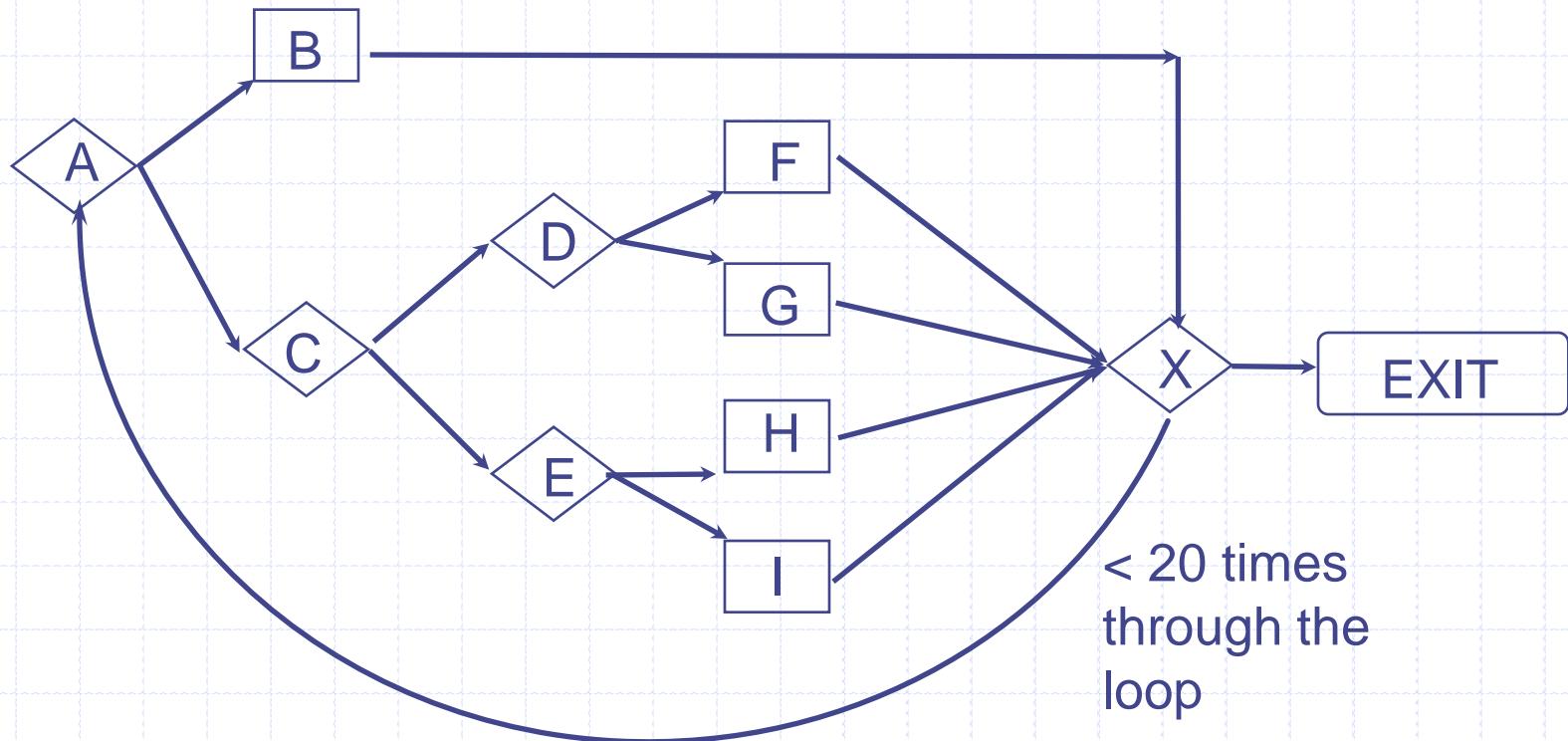
# Testing edited inputs

- ◆ Need to test that editing works (if allowed by the spec)
- ◆ Test that any character can be changed into any other
- ◆ Test repeated editing
  - Long strings of key presses followed by <Backspace> have been known to crash buffered input systems

# Testing input timing variations

- ◆ Try entering the data very quickly, or very slowly
- ◆ Do not wait for the prompt to appear
- ◆ Enter data before, after, and during the processing of some other event, or just as the time-out interval for this data item is about to expire
- ◆ Race conditions between events often leads to bugs that are hard to reproduce. For instance, handling of asynchronous events (e.g. event B has to happen after event A for some reason)

# Testing all paths in the system



Here's an example that shows that there are too many paths to test in even a fairly simple program. (Taken from Myers, *The Art of Software Testing*.)

# Number of paths

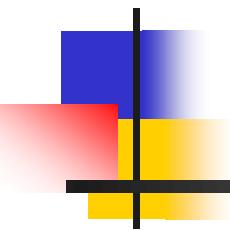
- ◆ One path is ABX-Exit. There are 5 ways to get to X and then to the EXIT in one pass.
- ◆ Another path is ABXACDFX-Exit. There are 5 ways to get to X the first time, 5 more to get back to X the second time, so there are  $5 \times 5 = 25$  cases like this.
- ◆ There are  $5^1 + 5^2 + \dots + 5^{19} + 5^{20} = 10^{14} = 100$  trillion paths through the program.
- ◆ It would take only a billion years to test every path (if one could write, execute and verify a test case every five minutes)

# Further difficulties for testers

- ◆ Testing cannot verify requirements.  
Incorrect or incomplete requirements  
may lead to spurious tests
- ◆ Bugs in test design or test drivers are  
equally hard to find
- ◆ Expected output for certain test cases  
might be hard to determine

# Conclusion

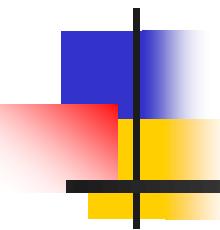
- ◆ It is impossible to completely test any non-trivial software module or system; therefore testers live and breathe tradeoffs
- ◆ Testing should be performed with the intention of finding errors
- ◆ Testing takes creativity and hard work
- ◆ Testing is best done by several independent testers



# Software Testing & Quality Assurance

## *Structural Testing*

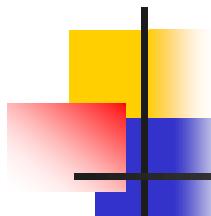
- Path Testing
- Test Coverage Metrics
- Cyclomatic and Essential Complexity
- Guidelines and Observations



# Credits & Readings

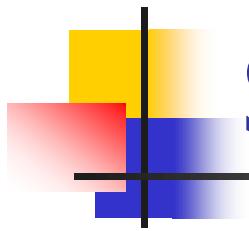
The material included in these slides are mostly adopted from the following books:

- *Software Testing: A Craftsman's Approach*, by Paul Jorgensen, CRC PRESS, third edition, ISBN: 0-8493-7475-8
- Cem Kaner, Jack Falk, Hung Q. Nguyen, "*Testing Computer Software*" Wiley (see also <http://www.testingeducation.org/>)
- Paul Ammann and Jeff Offutt, "*Introduction to Software Testing*", Cambridge University Press
- Beizer, Boris, "*Software Testing and Quality Assurance*", Van Nostrand Reinhold
- Glen Myers, "*The Art of Software Testing*"
- Stephen R. Schach, "*Software Engineering*", Richard D. Irwin, Inc. and Aksen Associates, Inc.



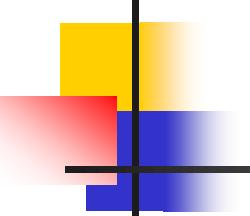
# Structural Testing

- Also known as glass/white/open box testing
- A software testing technique whereby explicit knowledge of the internal workings of the item being tested are used to select the test data
- Functional vs. Structural Testing
  - Functional Testing uses program specification
  - Structural Testing is based on specific knowledge of the source code to define the test cases and to examine outputs



# Structural Testing

- Structural testing methods are very amenable to:
  - Rigorous definitions
    - Control flow, data flow, coverage criteria
  - Mathematical analysis
    - Graphs, path analysis
  - Precise measurement
    - Metrics, coverage analysis

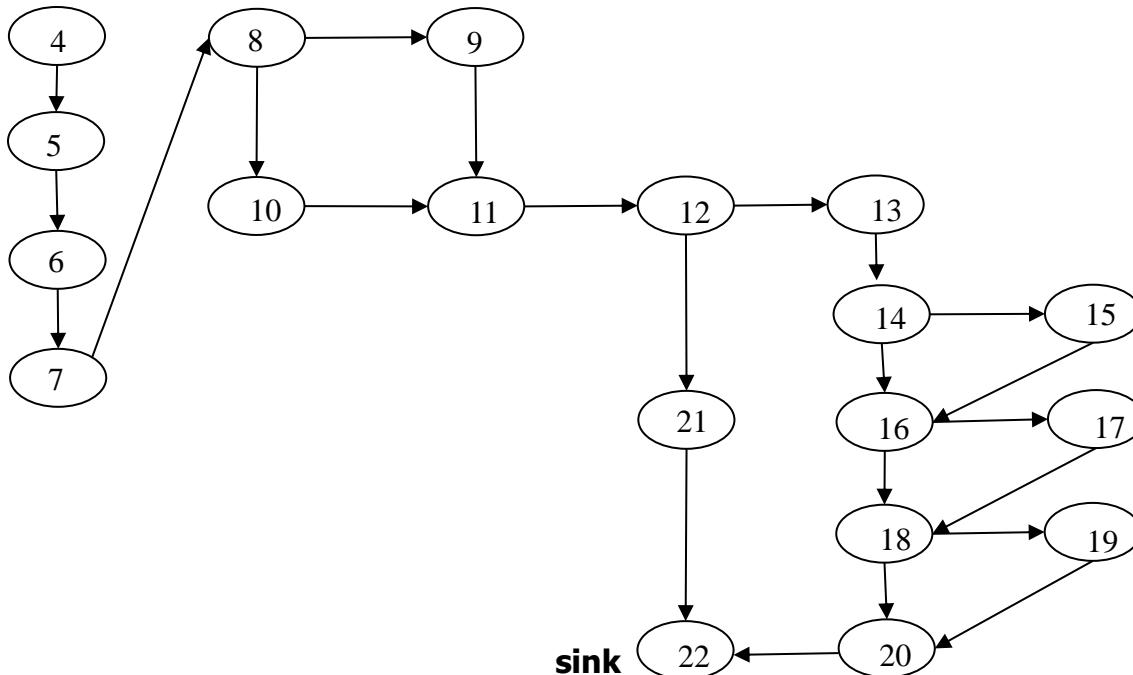


# Path Testing

- When a test case executes, it traverses a *path*
  - Some paths can be executed by many test cases
  - Some paths cannot be executed by any test cases
- Paths are derived from some graph construct (e.g. program graph)
- Issues:
  - huge number of paths; some simplification is needed
  - impossible paths
- Challenge: what kinds of faults are associated with what kinds of paths?

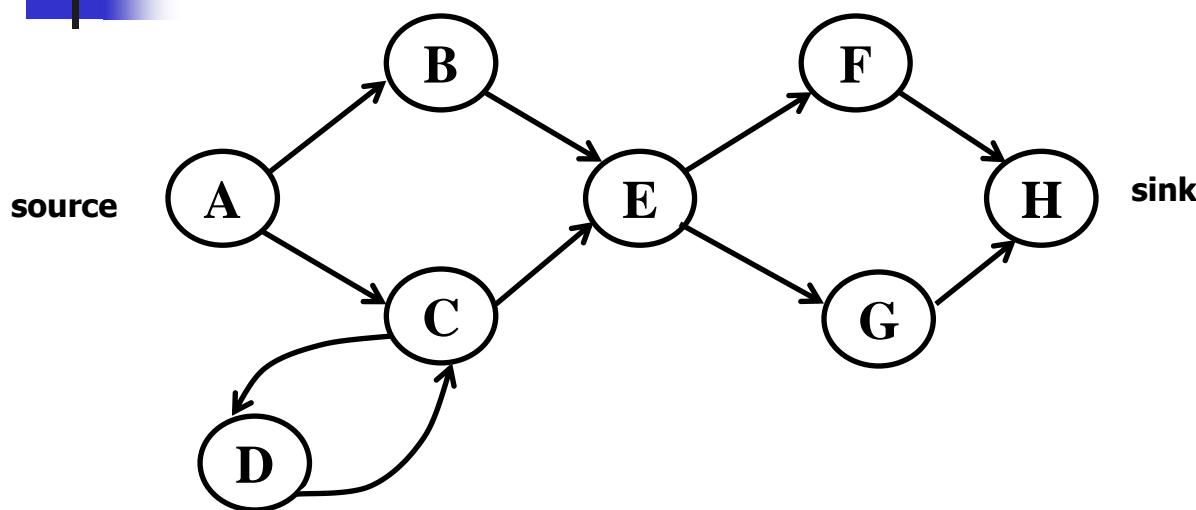
# Program Graph

source



- Testers use graphs to model software and then seek ways to provide "coverage"
- A program graph (a.k.a. control flow graph) is a directed graph in which
  - A node corresponds to entire statements or fragments of a statement
  - An edge represents flow of control (i.e. there is an edge from node i to node j iff the statement or statement fragment corresponding to node j can be executed immediately after the statement or statement fragment corresponding to node i)

# Program Graph Coverage



## Statement/Segment coverage:

*Cover every statement/segment (node)*

- ABEFH
- ACDCEGH

## Branch coverage:

*Cover every branch (edge)*

- ABEFH
- ACDCEGH

## Path Coverage:

*Cover every path*

- ABEFH
- ABEGH
- ACEGH
- ACEFH
- ACDCEFH
- ACDCEGH
- A(CD)\*EGH

# Path Expressions

AL

ABL

ABCDGL

ABCDEGL

ABC(DEF)\*DGL

ABC(DEF)\*DEGL

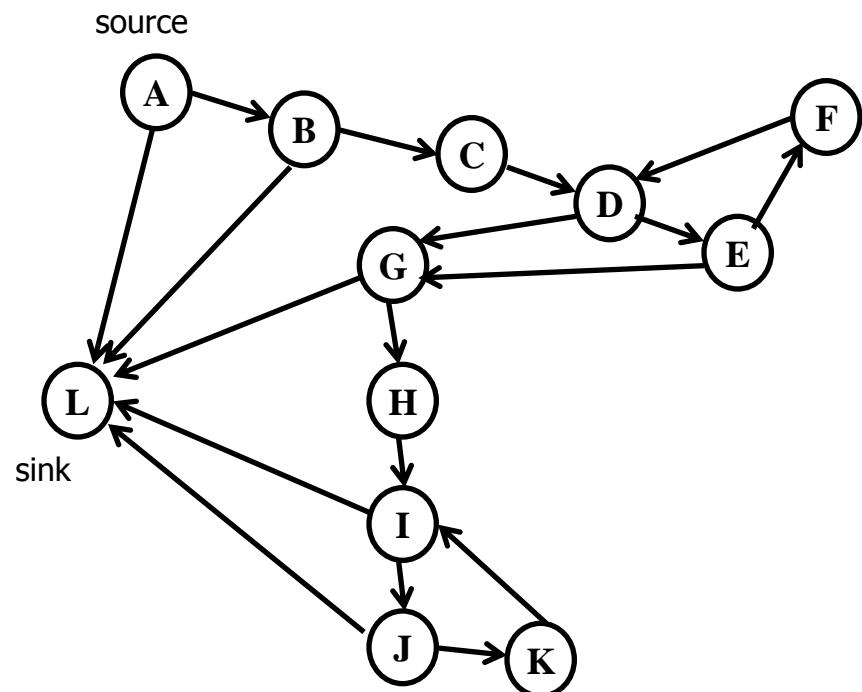
ABCDGHIL

ABCDGHIJL

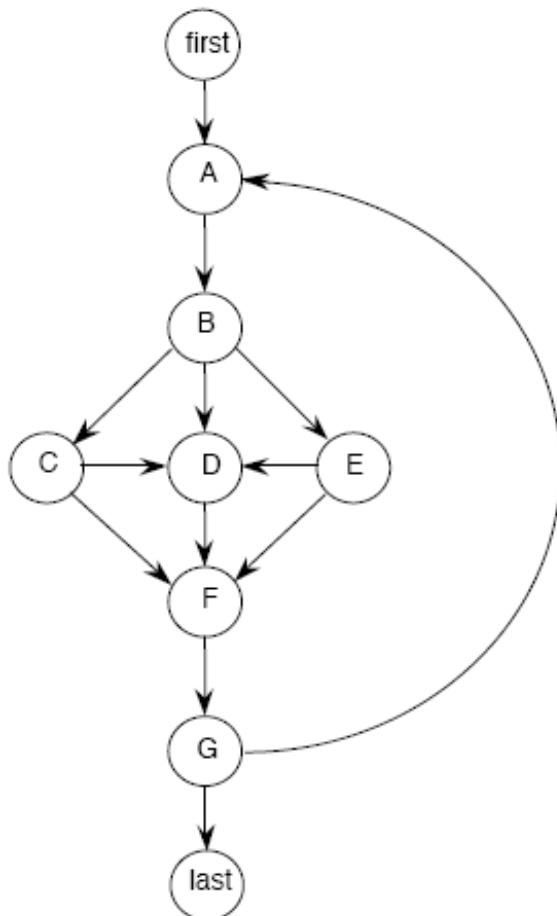
ABCDGH(IJK)\*IL

ABC(DEF)\*DEGH(IJK)\*IJL

...



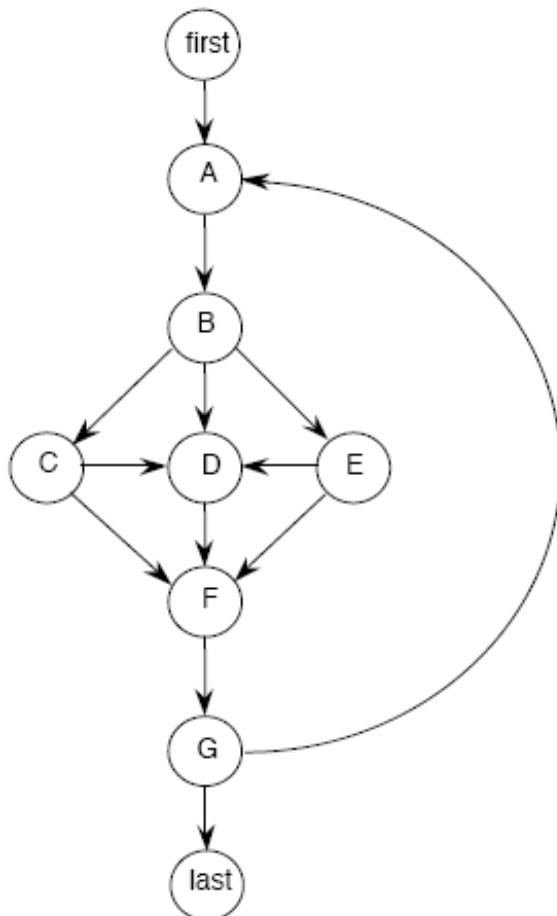
# Enormous Number of Paths



- If the loop executes up to 18 times, there are more than 4 trillion paths  $(5^1+5^2+5^3+\dots+5^{18})$
- BUT
  - Who could ever test all of these?
  - We need an elegant, mathematically sensible alternative

Taken from Stephen R. Schach, *Software Engineering*, (2<sup>nd</sup> edition) Richard D. Irwin, Inc. and Aksen Associates, Inc. 1993

# Test Cases for Coverage



## Some Test Cases

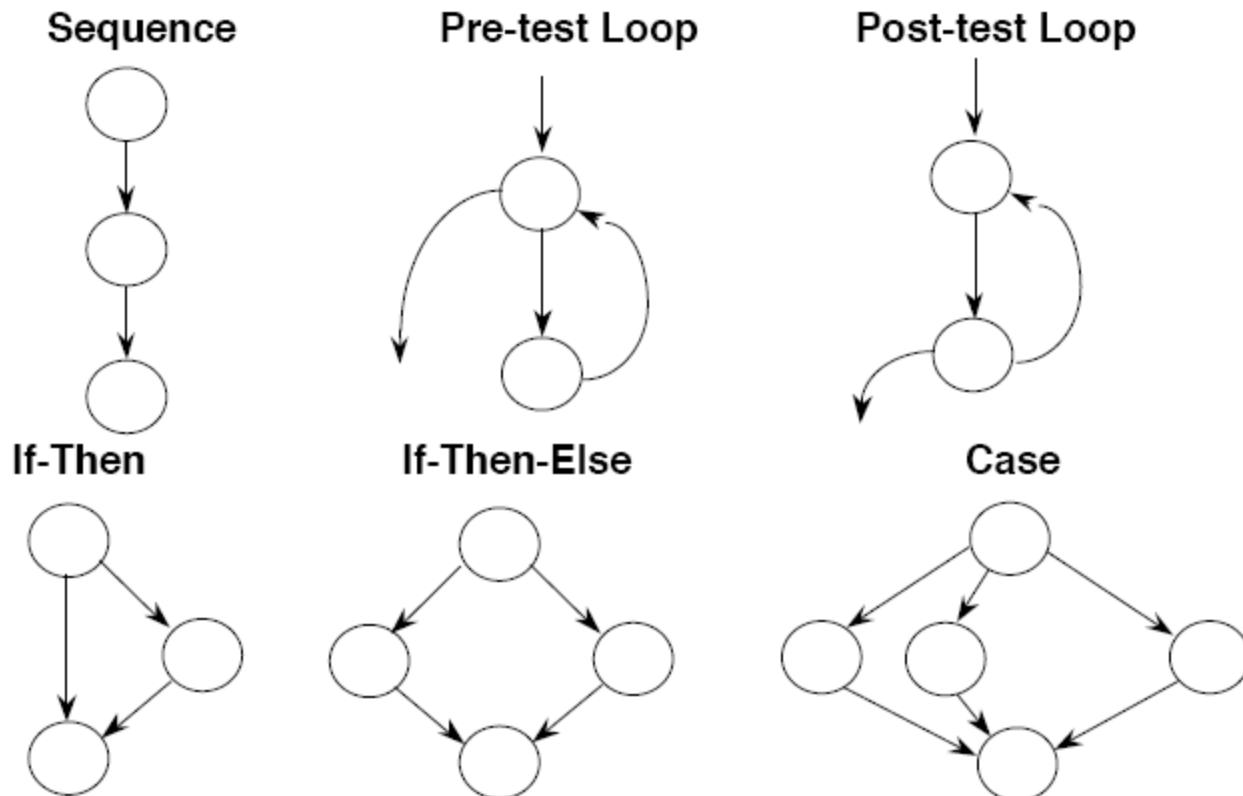
1. First-A-B-C-F-G-Last
2. First-A-B-C-D-F-G-Last
3. First-A-B-D-F-G-A-B-D-F-G-Last
4. First-A-B-E-F-G-Last
5. First-A-B-E-D-F-G-Last

These test cases cover

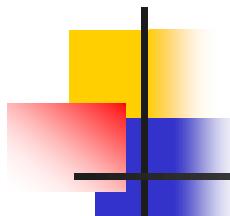
- Every node
- Every edge
- Normal repeat of the loop
- Exiting the loop

Taken from Stephen R. Schach, *Software Engineering*, (2<sup>nd</sup> edition) Richard D. Irwin, Inc. and Aksen Associates, Inc. 1993

# Program Graphs of Structured Programming Constructs



Adopted with permission from *Software Testing: A Craftsman's Approach*, by Paul Jorgensen, CRC PRESS, third edition.

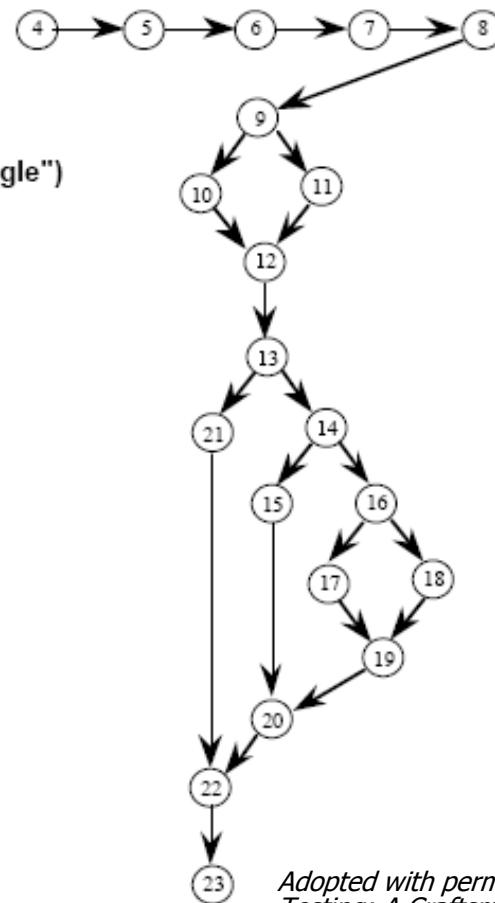


# Triangle Program Specification

- Inputs:  $a$ ,  $b$ , and  $c$  are non-negative integers, taken to be sides of a triangle
- Output: type of triangle formed by  $a$ ,  $b$ , and  $c$ 
  - Not a triangle
  - Scalene (no equal sides)
  - Isosceles (exactly 2 sides equal)
  - Equilateral (3 sides equal)
- To be a triangle,  $a$ ,  $b$ , and  $c$  must satisfy all three triangle inequalities:
  - $a < b + c$
  - $b < a + c$
  - $c < a + b$

# Program Graph for the Triangle Program

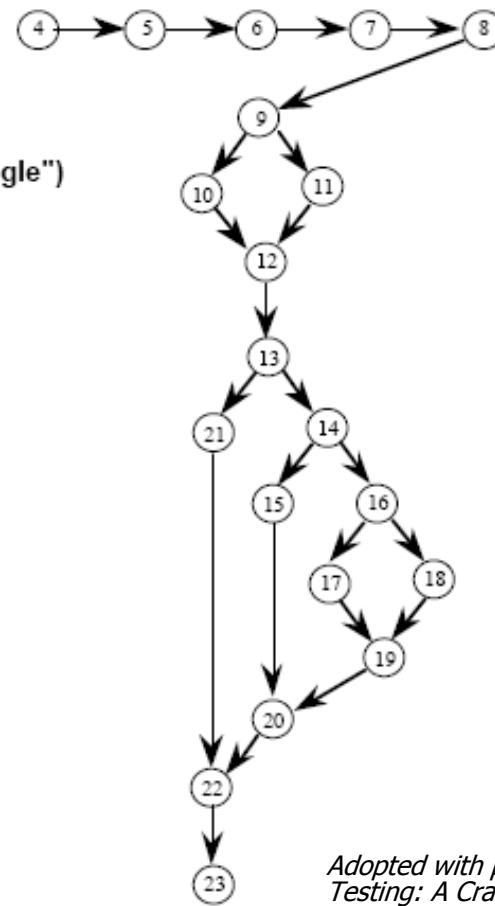
```
1. Program triangle
2. Dim a,b,c As Integer
3. Dim IsATriangle As Boolean
'Step 1: Get Input
4. Output("Enter 3 integers which are sides of a triangle")
5. Input(a,b,c)
6. Output("Side A is ",a)
7. Output("Side B is ",b)
8. Output("Side C is ",c)
'Step 2: Is A Triangle?
9. If (a < b + c) AND (b < a + c) AND (c < a + b)
10. Then IsATriangle = True
11. Else IsATriangle = False
12 EndIf
'Step 3: Determine Triangle Type
13. If IsATriangle
14. Then If (a = b) AND (b = c)
15. Then Output ("Equilateral")
16. Else If (a ≠ b) AND (a ≠ c) AND (b ≠ c)
17. Then Output ("Scalene")
18. Else Output ("Isosceles")
19. EndIf
20. EndIf
21. Else Output("Not a Triangle")
22. EndIf
23. End triangle2
```



Adopted with permission from Software Testing: A Craftsman's Approach, by Paul Jorgensen, CRC PRESS, third edition.

# Trace Code for $a = 5, b = 5, c = 5$

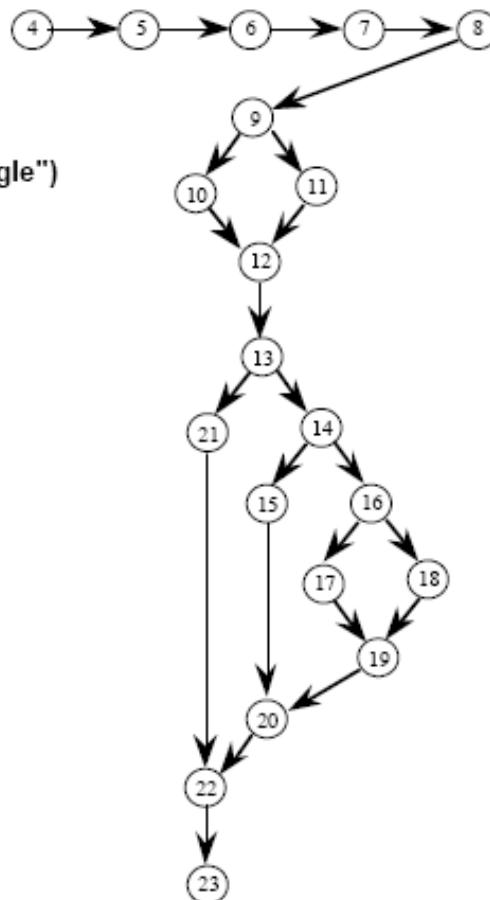
```
1. Program triangle
2. Dim a,b,c As Integer
3. Dim IsATriangle As Boolean
'Step 1: Get Input
4. Output("Enter 3 integers which are sides of a triangle")
5. Input(a,b,c)
6. Output("Side A is ",a)
7. Output("Side B is ",b)
8. Output("Side C is ",c)
'Step 2: Is A Triangle?
9. If (a < b + c) AND (b < a + c) AND (c < a + b)
10. Then IsATriangle = True
11. Else IsATriangle = False
12 EndIf
'Step 3: Determine Triangle Type
13 If IsATriangle
14 Then If (a = b) AND (b = c)
15 Then Output ("Equilateral")
16 Else If (a ≠ b) AND (a ≠ c) AND (b ≠ c)
17 Then Output ("Scalene")
18 Else Output ("Isosceles")
19 EndIf
20 EndIf
21 Else Output("Not a Triangle")
22 EndIf
23 End triangle2
```



*Adopted with permission from Software Testing: A Craftsman's Approach, by Paul Jorgensen, CRC PRESS, third edition.*

# In Class Activity

```
1. Program triangle
2. Dim a,b,c As Integer
3. Dim IsATriangle As Boolean
'Step 1: Get Input
4. Output("Enter 3 integers which are sides of a triangle")
5. Input(a,b,c)
6. Output("Side A is ",a)
7. Output("Side B is ",b)
8. Output("Side C is ",c)
'Step 2: Is A Triangle?
9. If (a < b + c) AND (b < a + c) AND (c < a + b)
10. Then IsATriangle = True
11. Else IsATriangle = False
12 EndIf
'Step 3: Determine Triangle Type
13 If IsATriangle
14. Then If (a = b) AND (b = c)
15. Then Output ("Equilateral")
16. Else If (a ≠ b) AND (a ≠ c) AND (b ≠ c)
17. Then Output ("Scalene")
18. Else Output ("Isosceles")
19. EndIf
20. EndIf
21. Else Output("Not a Triangle")
22. EndIf
23. End triangle2
```



Trace the code for following test cases:

T1: a = 2, b = 5, c = 5

T2: a = 3, b = 4, c = 5

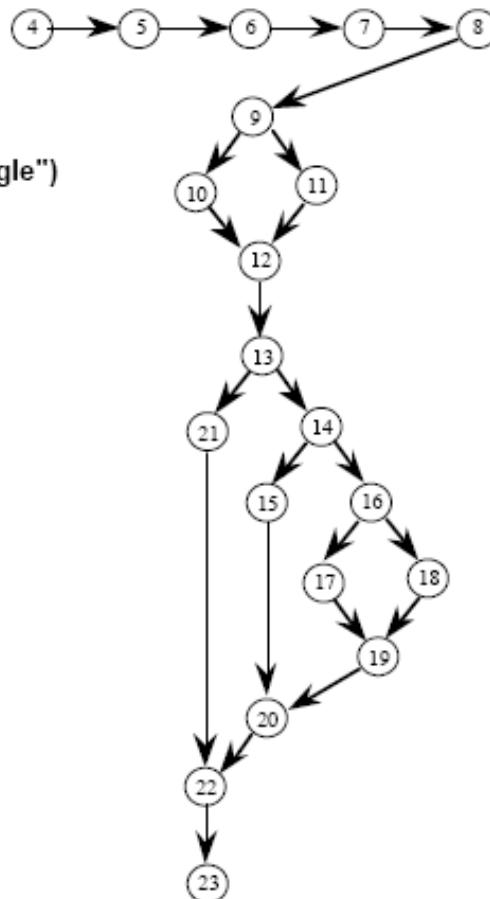
T3: a = 2, b = 3, c = 7

Show the path coverage on the program graph

*Adopted with permission from Software Testing: A Craftsman's Approach, by Paul Jorgensen, CRC PRESS, third edition.*

# Impossible Paths

```
1. Program triangle
2. Dim a,b,c As Integer
3. Dim IsATriangle As Boolean
'Step 1: Get Input
4. Output("Enter 3 integers which are sides of a triangle")
5. Input(a,b,c)
6. Output("Side A is ",a)
7. Output("Side B is ",b)
8. Output("Side C is ",c)
'Step 2: Is A Triangle?
9. If (a < b + c) AND (b < a + c) AND (c < a + b)
10. Then IsATriangle = True
11. Else IsATriangle = False
12 EndIf
'Step 3: Determine Triangle Type
13 If IsATriangle
14. Then If (a = b) AND (b = c)
15. Then Output ("Equilateral")
16. Else If (a ≠ b) AND (a ≠ c) AND (b ≠ c)
17. Then Output ("Scalene")
18. Else Output ("Isosceles")
19. EndIf
20. EndIf
21. Else Output("Not a Triangle")
22. EndIf
23. End triangle2
```



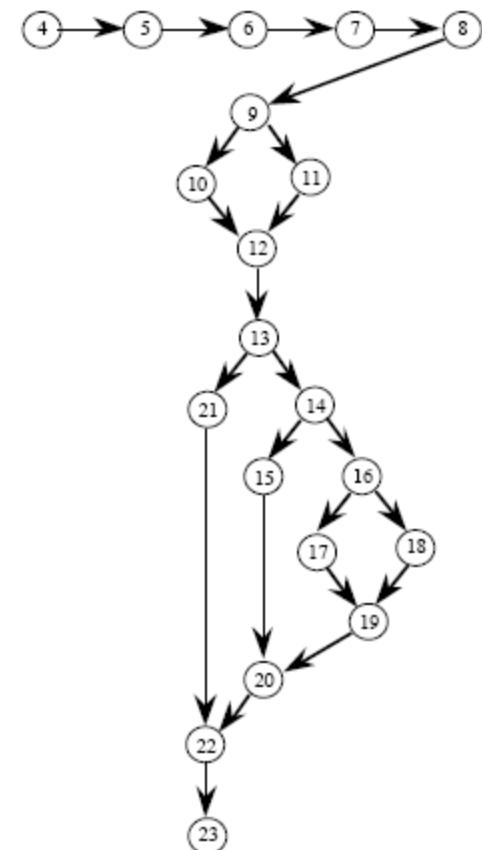
Can you find values of a, b, and c such that the path traverses nodes 10 and 21?

*Adopted with permission from Software Testing: A Craftsman's Approach, by Paul Jorgensen, CRC PRESS, third edition.*

# DD-Path definition

- A decision-to-decision path (DD-Path) is a chain in a program graph that satisfies any of the following cases:
  - Case 1: it consists of a single node with indegree=0 (E.g. node 4) or
  - Case 2: it consists of a single node with outdegree=0 (E.g. node 23) or
  - Case 3: it consists of a single node with  $\text{indegree} \geq 2$  or  $\text{outdegree} \geq 2$   
(E.g. node 9, 12, 13, 14, 16, 19, 20, 22) or
  - Case 4: it consists of a single node with  $\text{indegree} = 1$ , and  $\text{outdegree} = 1$ , or  
(E.g. nodes 10, 11, 15, 17, 18, 21) or
  - Case 5: it is a maximal chain of length  $\geq 1$   
(E.g. nodes 5-8)

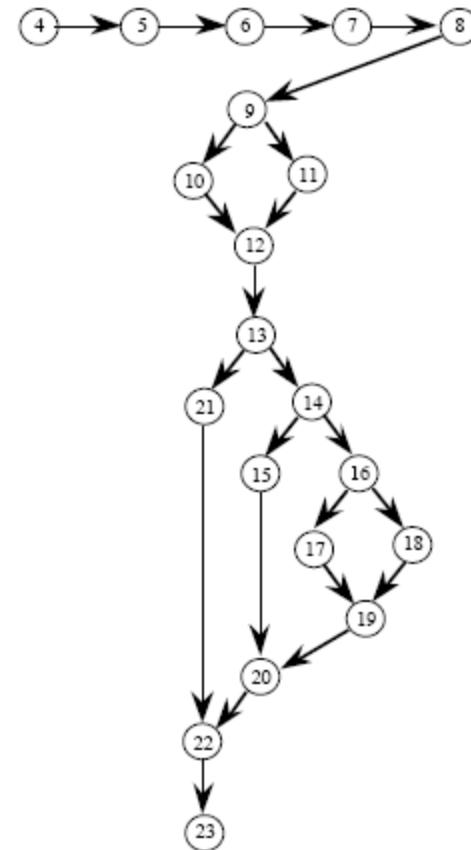
Program Graph for Triangle



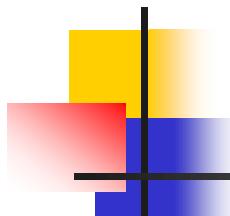
# DD-Paths for the Triangle

Program Graph Node	DD-path Name	Definition Case
4	First	1
5-8	A	5
9	B	3
10	C	4
11	D	4
12	E	3
13	F	3
14	H	3
15	I	4
16	J	3
17	K	4
18	L	4
19	M	3
20	N	3
21	G	4
22	O	3
23	Last	2

Program Graph for Triangle



Note: since the Triangle program is logic intensive and computationally sparse, it yields many short DD-Paths as shown in the this table. In other programs we will find longer chains.

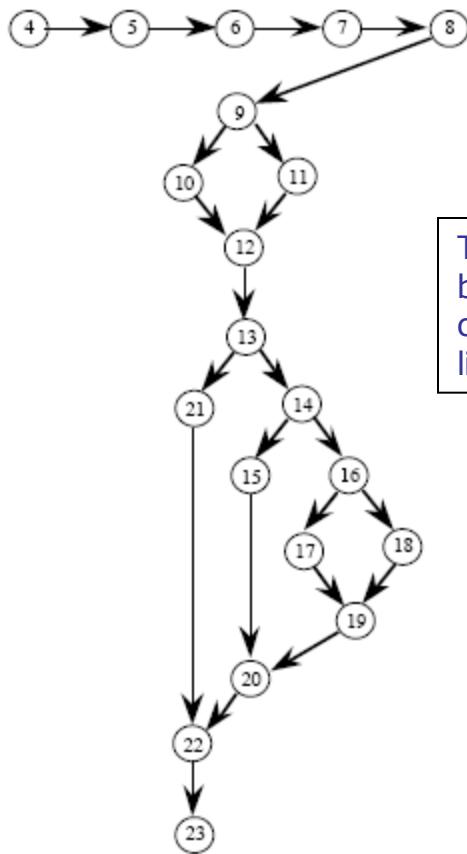


# DD-Path Graph

- A program's *DD-Path graph* is a directed graph in which
  - nodes are DD-Paths of its program graph, and
  - edges represent control flow between successor DD-Paths
- Also known as *Control Flow Graph*
  - it is a form of condensation graph
  - maximal chains are collapsed into an individual node (corresponding to Case 5)
  - single node DD-Paths (corresponding to Cases 1 - 4 ) preserve the convention that a statement (or a statement fragment) is in exactly one DD-Path
  - The process of generating DD-path graphs manually can be a tedious task (for larger programs)
  - There exist commercial tools that automate this process

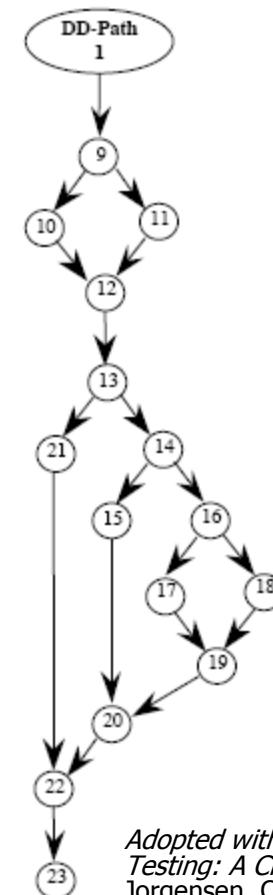
# DD-Path Graph for Triangle

Program Graph



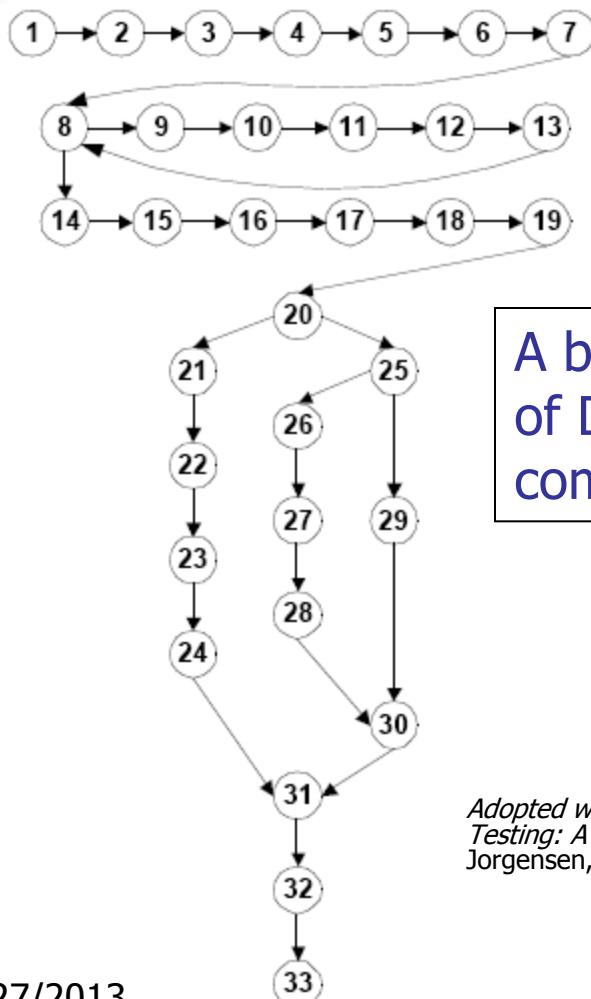
There is not much compression here because this example is control intensive, with little sequential code.

DD-Path Graph

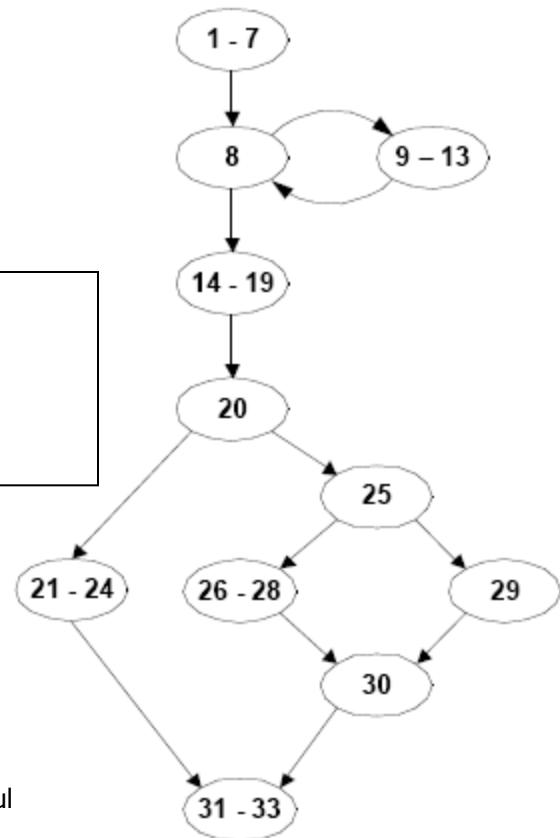


Adopted with permission from *Software Testing: A Craftsman's Approach*, by Paul Jorgensen, CRC PRESS, third edition.

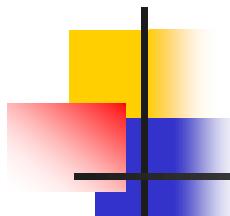
## DD-Path Graph for the Commission Problem



A better example  
of DD-Path  
compression



Adopted with permission from *Software Testing: A Craftsman's Approach*, by Paul Jorgensen, CRC PRESS, third edition.



# Program Graphs with Statement Fragments

- Create graph nodes when you encounter any of the following
  - a predicate
  - a loop control
  - a break
  - a method exit/return
- Let's try an example...

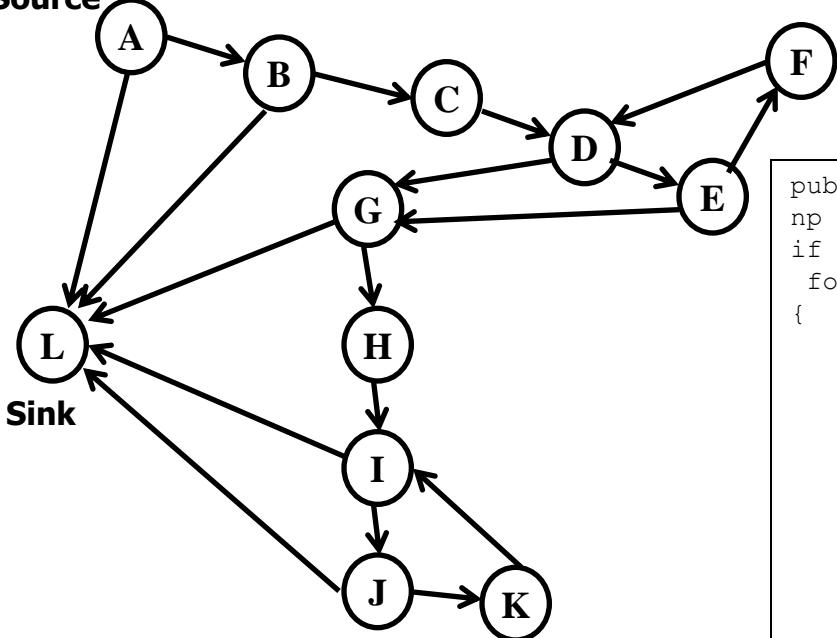
```
public int displayLastMsg(int nToPrint) {  
    np = 0;  
    if ((msgCounter > 0) && (nToPrint > 0)) {  
        for (int j = lastMsg; ((j != 0) && (np < nToPrint)); --j) {  
            System.out.println(messageBuffer[j]);  
            ++np;  
        }  
        if (np < nToPrint) {  
            for (int j = SIZE; ((j != 0) && (np < nToPrint)); --j) {  
                System.out.println(messageBuffer[j]);  
                ++np;  
            }  
        }  
    }  
    return np;  
}
```

Stop when you encounter any of the following

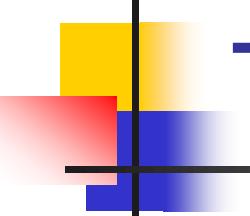
- a predicate
- a loop control/counter
- a break
- a method exit/return

# Program Graph

Source



```
public int displayLastMsg(int nToPrint) {  
    np = 0;          A  
    if ((msgCounter > 0) && (nToPrint > 0)) {  
        for (int j = lastMsg; ((j != 0) && (np < nToPrint)); --j) {  
            C System.out.println(messageBuffer[j]);  
            D ++np;  
        }          G  
        if (np < nToPrint) {  
            for (int j = SIZE; ((j != 0) && (np < nToPrint)); --j) {  
                H System.out.println(messageBuffer[j]);  
                I ++np;  
            }          J  
        }  
    }  
    return np;      L  
}
```



# Test Coverage Metrics

## Graph-based

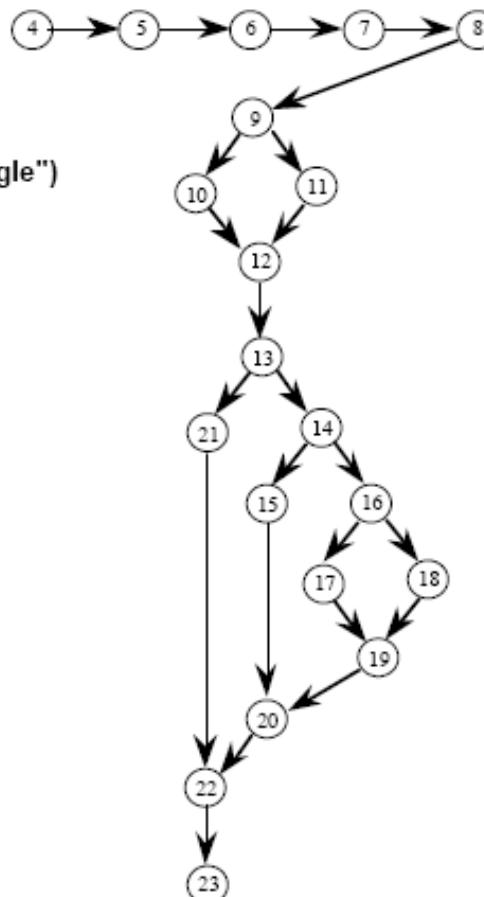
- Every node
- Every edge
- Every path

## Code-based (E. F. Miller, 1977 dissertation)

- $C_0$ : Every statement is executed at least once
- $C_1$ : Every DD-Path is executed at least once
- $C_{1p}$ : Every predicate outcome
- $C_2$ :  $C_1$  coverage + loop coverage
- $C_d$ :  $C_1$  coverage + every pair of dependent DD-Paths
- $C_{MCC}$ : Multiple condition coverage
- $C_{ik}$ : Every program path that contains up to k repetitions of a loop (usually k = 2)
- $C_{stat}$ : "Statistically significant" fraction of paths
- $C_\infty$ : All possible execution paths

# Dependent DD-Paths

```
1. Program triangle
2. Dim a,b,c As Integer
3. Dim IsATriangle As Boolean
4. 'Step 1: Get Input
5. Output("Enter 3 integers which are sides of a triangle")
6. Input(a,b,c)
7. Output("Side A is ",a)
8. Output("Side B is ",b)
9. Output("Side C is ",c)
10. 'Step 2: Is A Triangle?
11. If (a < b + c) AND (b < a + c) AND (c < a + b)
12. Then IsATriangle = True
13. Else IsATriangle = False
14. EndIf
15. 'Step 3: Determine Triangle Type
16. If IsATriangle
17. Then If (a = b) AND (b = c)
18. Then Output ("Equilateral")
19. Else If (a ≠ b) AND (a ≠ c) AND (b ≠ c)
20. Then Output ("Scalene")
21. Else Output ("Isosceles")
22. EndIf
23. Else Output("Not a Triangle")
24. EndIf
25. End triangle2
```



Often correspond to infeasible paths

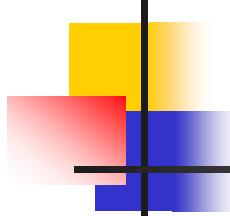
If a path traverses node 10 (Then IsATriangle =True), then it must traverse node 14

Similarly, if a path traverses node 11 (Else IsATriangle = False), then it must traverse node 21

Paths through nodes 10 and 21 are infeasible

Similarly for paths through 11 and 14

Hence the need for the Cd coverage metric

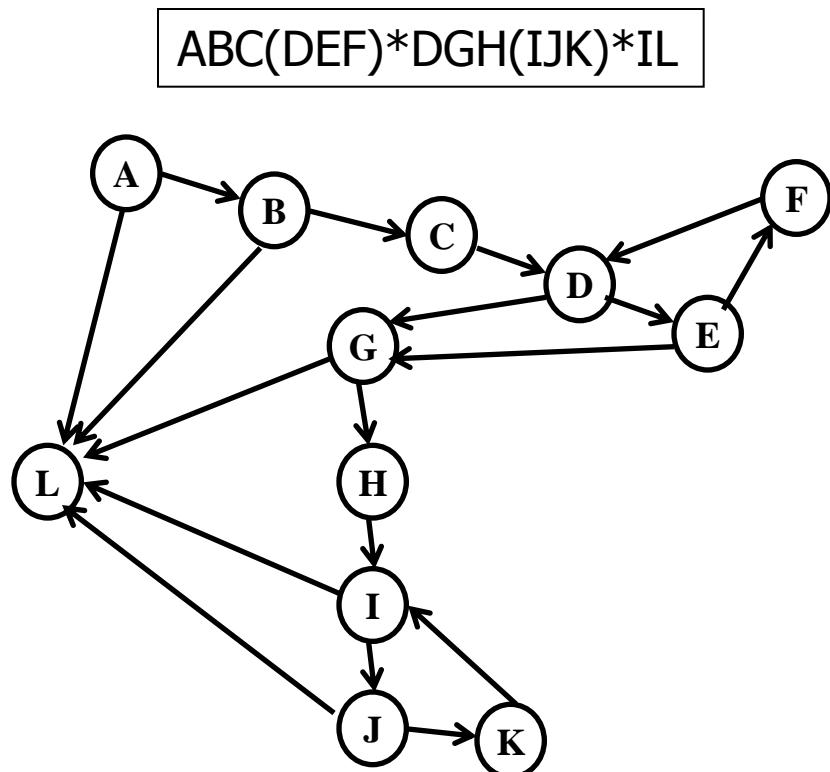


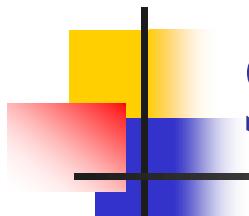
# Popular Coverage Metrics

- Statement Coverage
  - Every statement is executed at least once
- DD-Path Coverage
  - Every DD-Path is executed at least once
- Branch Coverage
  - Every predicate outcome (T/F) is executed at least once
- Multiple-Condition Coverage
  - All T/F combinations of simple conditions in compound predicates are considered at least once

# Statement Coverage

- Achieved when all statements in a method have been executed at least once
- A test case that will follow the path expression shown above will achieve statement coverage in this example
- One test case is enough to achieve statement coverage!

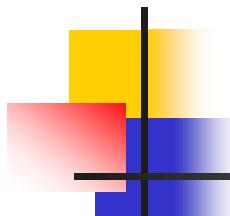




# Statement Coverage Issues

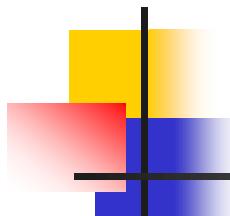
- A predicate may be tested for only one value (can miss many bugs)
- Loop bodies may only be iterated once
- Statement coverage can be achieved without branch coverage. Important cases may be missed

```
String s = null;  
if (x != y) s = "Hi";  
String s2 = s.substring(1);
```



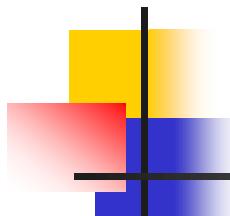
# DD-Path Coverage

- It covers segments not just single statements
- May produce drastically different numbers
  - Assume two segments P and Q
  - P has one statement, Q has nine
  - Exercising only one of the segments will give 10% or 90% statement coverage
  - Segment coverage will be 50% in both cases



# Branch Coverage

- At least one True and one False evaluation for each predicate
- Short-circuit evaluation means that many predicates might not be evaluated
- A compound predicate is treated as a single statement
  - If n clauses,  $2^n$  combinations, but only 2 are tested



# Multiple-Condition Coverage

- All True-False combinations of simple conditions in compound predicates are considered at least once
- A truth table may be necessary and can be constructed as follows:
  - Consider the multiple condition as a logical expression of simple conditions
  - Make the truth table of the logical expression
  - Convert the truth table to a decision table
  - Develop test cases for each rule of the decision table (except the impossible rules, if any)

# Apply Multiple-Condition Coverage to the Following Code

```
If (a < b + c) AND (b < a + c) AND (c < a + b)  
Then IsATriangle = True  
Else IsATriangle = False  
Endif
```

# Truth Table for ( $a < b+c$ ) AND ( $b < a+c$ ) AND ( $c < a+b$ )

$(a < b+c)$	$(b < a+c)$	$(c < a+b)$	$(a < b+c) \text{ AND } (b < a+c) \text{ AND } (c < a+b)$
T	T	T	T
T	T	F	F
T	F	T	F
T	F	F	F
F	T	T	F
F	T	F	F
F	F	T	F
F	F	F	F

Adopted with permission from Software Testing: A Craftsman's Approach, by Paul Jorgensen, CRC PRESS, third edition.

# Decision Table for ( $a < b+c$ ) AND ( $b < a+c$ ) AND ( $c < a+b$ )

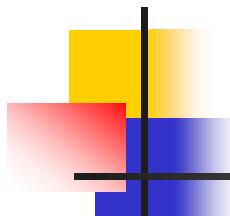
c1: $a < b+c$	T	T	T	T	F	F	F	F
c2: $b < a+c$	T	T	F	F	T	T	F	F
c3: $c < a+b$	T	F	T	F	T	F	T	F
a1: impossible				X		X	X	X
a2:Valid test case #	1	2	3		4			

Adopted with permission from *Software Testing: A Craftsman's Approach*, by Paul Jorgensen, CRC PRESS, third edition.

# Multiple Condition Test Cases for $(a < b+c)$ AND $(b < a+c)$ AND $(c < a+b)$

Test Case	True Conditions	a	b	c	Expected
1	All three conditions True	3	4	5	TRUE
2	$a < b + c$ $b < a + c$	3	4	9	FALSE
3	$a < b + c$ $c < a + b$	3	9	4	FALSE
4	$b < a + c$ $c < a + b$	9	3	4	FALSE

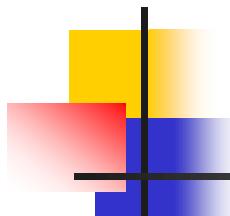
Adopted with permission from Software Testing: A Craftsman's Approach, by Paul Jorgensen, CRC PRESS, third edition.



## In Class Activity

What test cases are needed for this code fragment ?

```
If (a ≠ b) AND (a ≠ c) AND (b ≠ c)  
    Then Output ("Scalene")  
Else Output ("Isosceles")  
Endif
```

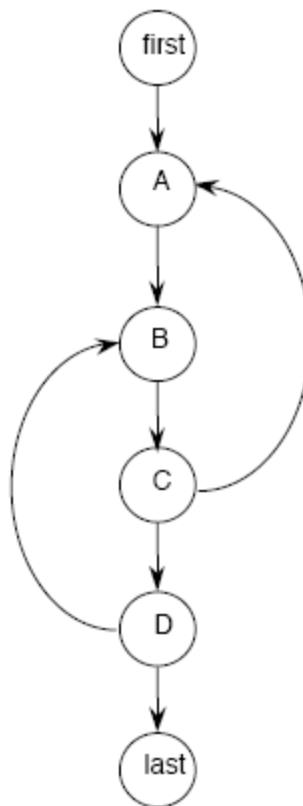
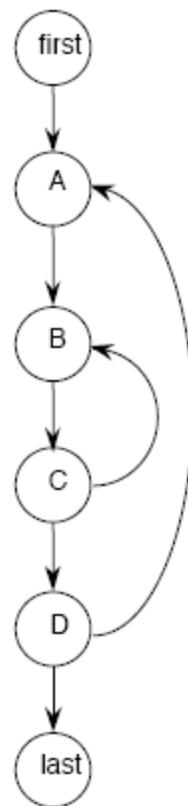
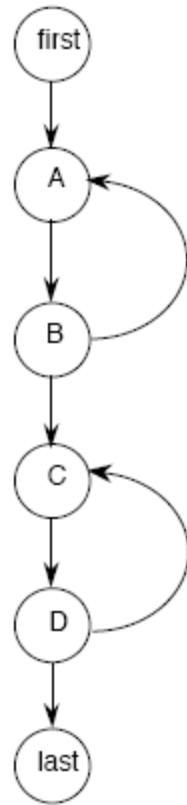


# Dealing with Loops

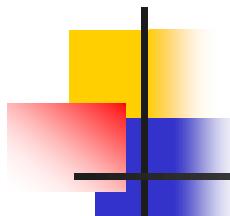
- Loops are highly fault-prone, so they need to be tested carefully
- A simple view: Every loop involves a decision to traverse the loop or not
- A better view: Boundary value analysis on the index variable
- Huang's Theorem: (Paraphrased)

*Everything interesting will happen in two loop traversals: the normal loop traversal and the exit from the loop*

# Concatenated, Nested, and Knotted Loops

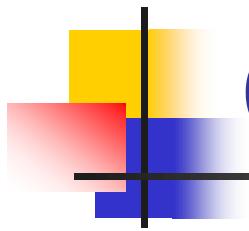


Adopted with permission from *Software Testing: A Craftsman's Approach*, by Paul Jorgensen, CRC PRESS, third edition.



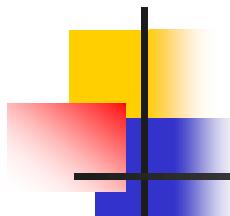
# Strategy for Loop Testing

- Huang's theorem suggests/assures 2 tests per loop is sufficient
- For nested loops:
  - Test innermost loop first
  - Then "condense" the loop into a single node (as in condensation graph)
  - Work from innermost to outermost loop
- For concatenated loops: use Huang's Theorem
- For knotted loops: Rewrite! (violates structure programming principles and results in high cyclomatic complexity)



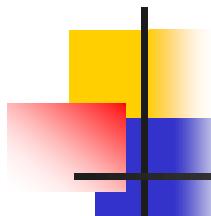
# Code-Based Testing Strategy

- Start with a set of test cases generated by an “appropriate” (depends on the nature of the program) specification-based test method
- Look at code to determine appropriate test coverage metric (e.g. statement, branch, etc.)
- If appropriate coverage is attained by the initial test cases, fine
- Otherwise, add test cases to attain the intended test coverage



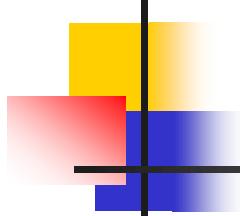
# Coverage Usefulness

- 100% coverage is never a guarantee of bug-free software
- Coverage reports can
  - point out inadequate test suites
  - suggest the presence of surprises, such as blind spots (gaps) in the test design
  - Help identify parts of the implementation that require further structural testing



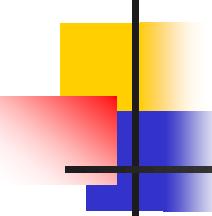
# How to Measure Coverage

- The source code is instrumented
  - Depending on the code coverage model, code that writes to a trace file is inserted in every branch, statement etc.
- Most commercial tools measure DD-Path and branch coverage



# Test Coverage Tools

- Commercial test coverage tools use “instrumented” source code
  - New code added to the code being tested
  - Designed to “observe” a level of test coverage
- When a set of test cases is run on the instrumented code, the designed test coverage is ascertained
- Strictly speaking, running test cases in instrumented code is not sufficient
  - Safety critical applications require tests to be run on actual (delivered, non-instrumented) code



# Sample DD-Path Instrumentation

Values of array DDpathTraversed are set to 1 when corresponding instrumented code is executed

**DDpathTraversed(1) = 1**

4. Output("Enter 3 integers which are sides of a triangle")
5. Input(a, b, c)
6. Output("Side A is ", a)
7. Output("Side B is ", b)
8. Output("Side C is ", c)

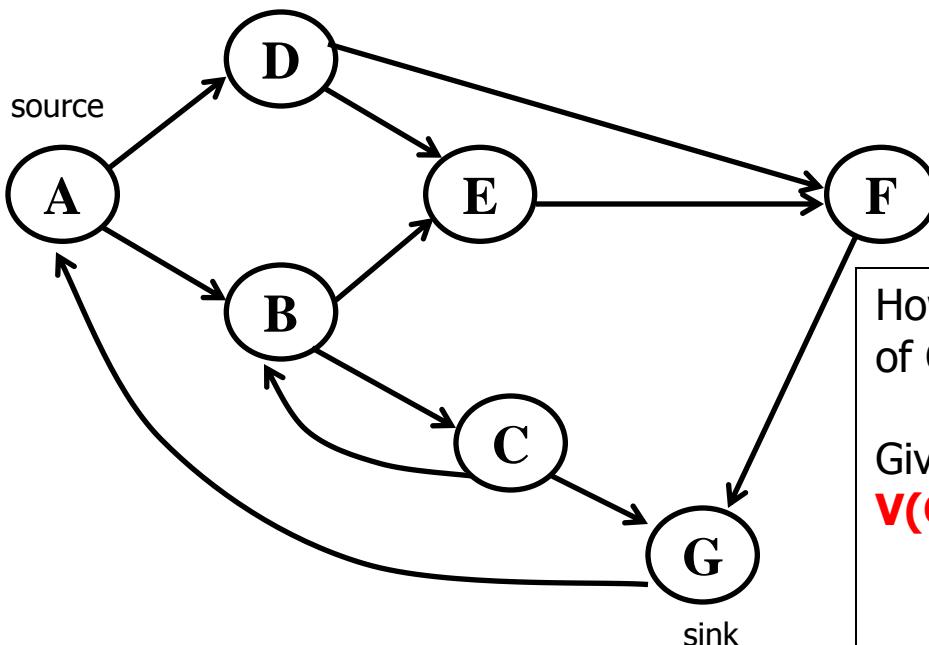
'Step 2: Is A Triangle?

**DDpathTraversed(2) = 1**

9. If (a < b + c) AND (b < a + c) AND (c < a + b)
10. Then **DDpathTraversed(3) = 1**  
IsATriangle = True
11. Else **DDpathTraversed(4) = 1**  
IsATriangle = False
- 12 EndIf

# Cyclomatic Complexity<sup>1</sup>

Consider the following strongly connected directed graph G (node G connects back to node A)



- Cyclomatic complexity is a metric used to measure the complexity of programs
- Programs with high cyclomatic complexity require more testing

How do we calculate the cyclomatic complexity V of G?

Given a strongly connected graph G:

$$V(G) = e - n + 1 = 11 - 7 + 1 = 5 \text{ where}$$

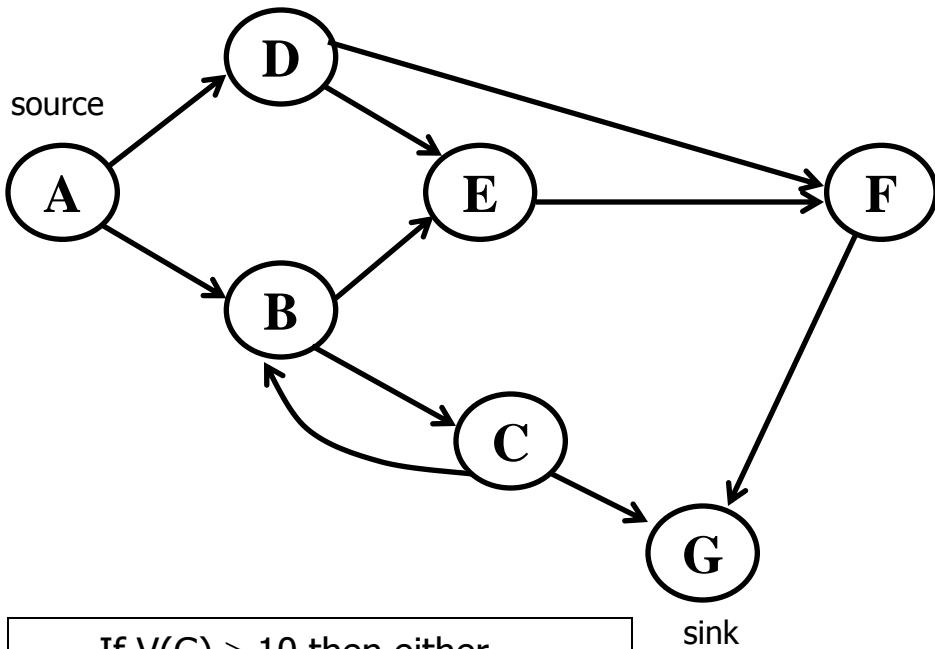
e = number of edges

n = number of nodes

*It means that G contains 5 independent circuits*

# Cyclomatic Complexity<sup>2</sup>

An arbitrary directed graph  $G(V,E)$   
(Note: violations of structured programming)



- If  $V(G) \geq 10$  then either simplify the unit or plan to do more testing

Another way to calculate  $V(G)$ :

Given an arbitrary directed graph  $G$ :

$$V(G) = e - n + 2p = 10 - 7 + 2(1) = 5$$

$e$  = number of edges

$n$  = number of nodes

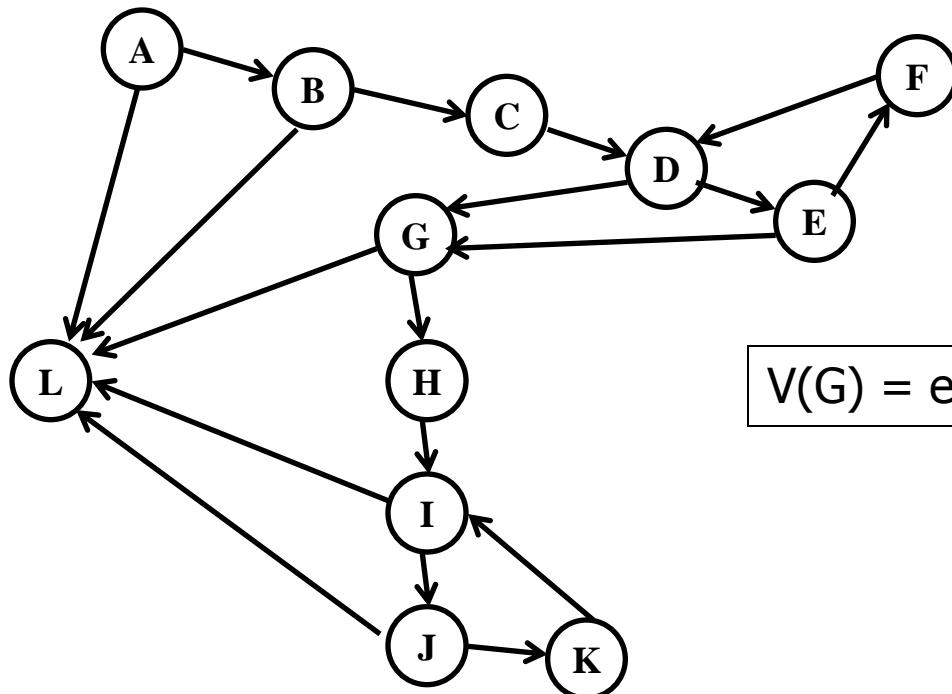
$p$  = number of connected regions  
(which is normally 1)

*It means that  $G$  has 5 linearly independent paths from node A to node G as follows:*

- P1: A, B, C, G
- P2: A, B, C, B, C, G
- P3: A, B, E, F, G
- P4: A, D, E, F, G
- P5: A, D, F, G

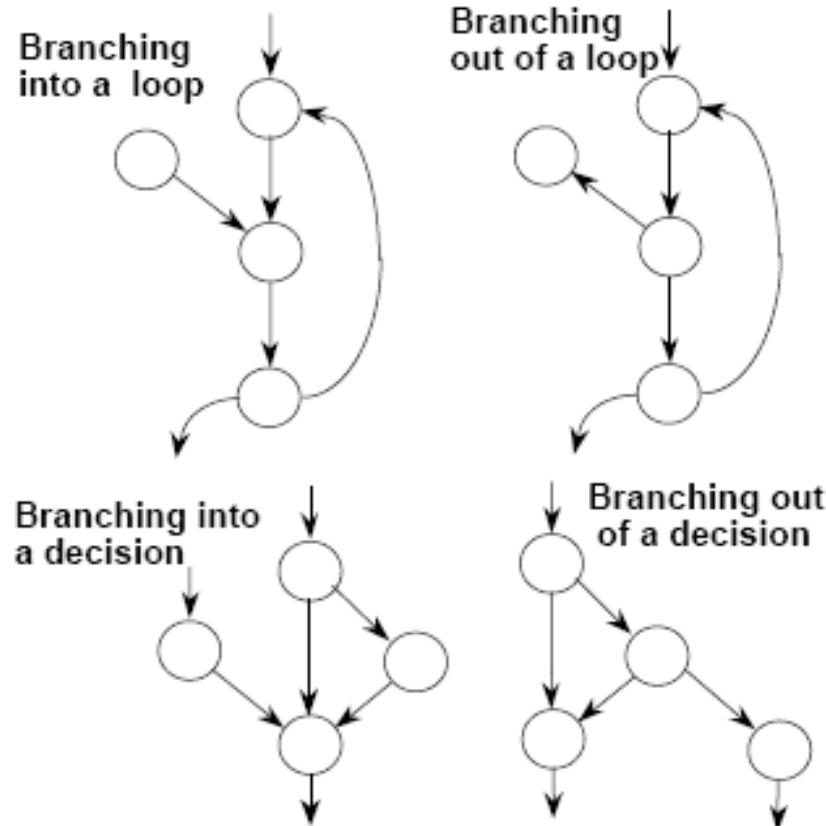
# In Class Activity

Can you calculate the Cyclomatic Complexity of this arbitrary directed graph?

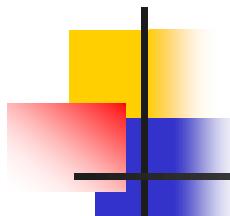


$$V(G) = e - n + 2p = 18 - 12 + 2(1) = 8$$

# Violations of Structured Programming



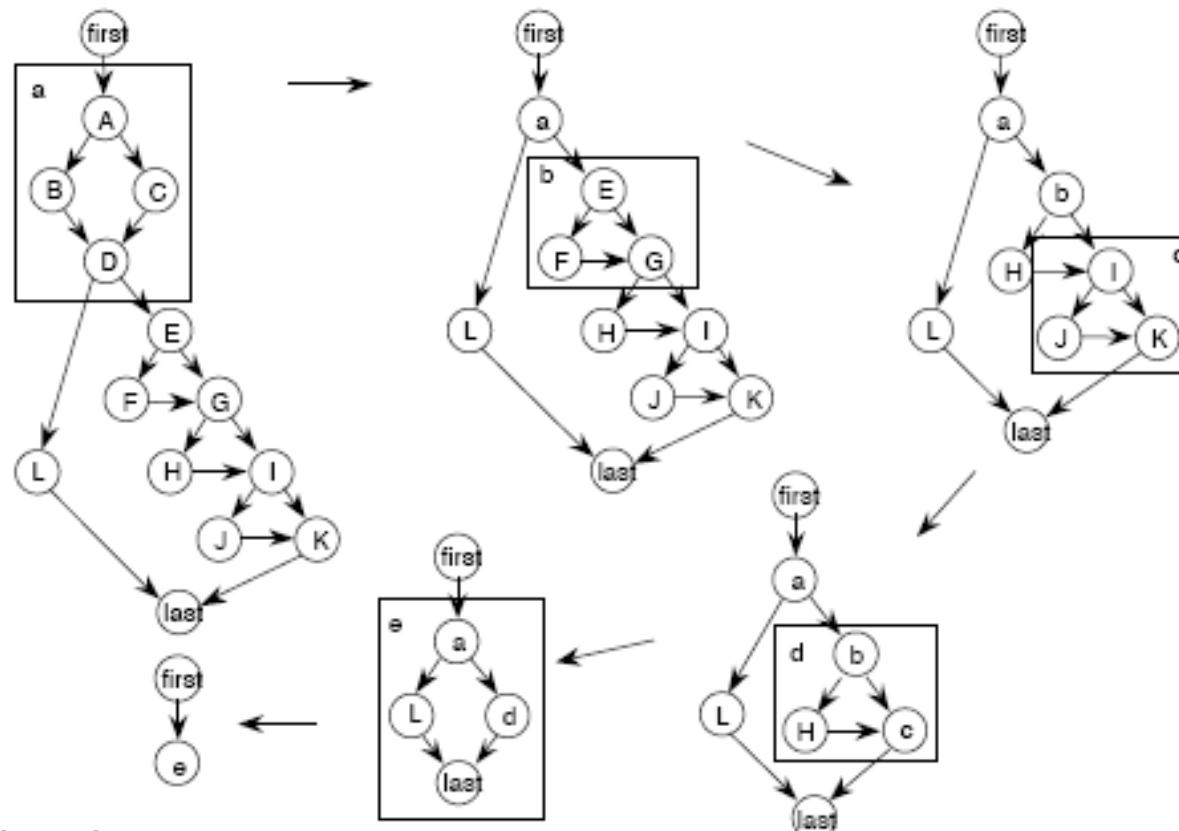
*Adopted with permission from Software Testing: A Craftsman's Approach, by Paul Jorgensen, CRC PRESS, third edition.*



# Essential Complexity

- McCabe's notion of Essential Complexity deals with the extent to which a program violates the principles of Structured Programming
- To find the Essential Complexity of a program graph:
  - Identify a group of source statements that corresponds to one of the basic Structured Programming constructs
  - Condense that group of statements into a separate node (with a new name)
  - Continue until no more Structured Programming constructs can be found
- The Essential Complexity of a Structured Program is 1
- Violations of Structured Programming increase the Essential Complexity

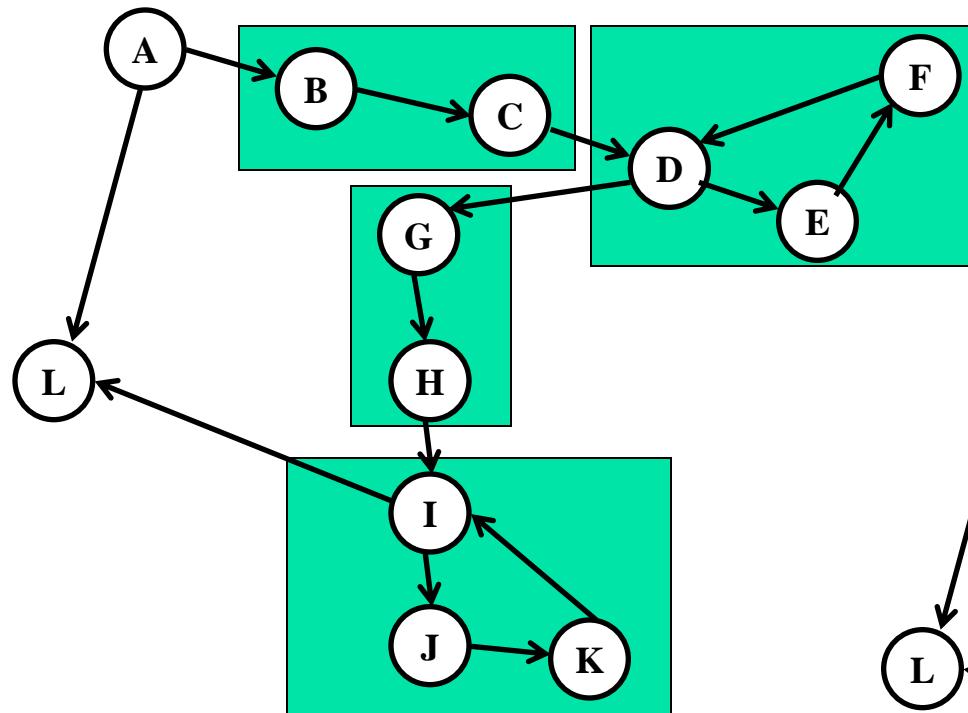
# Example



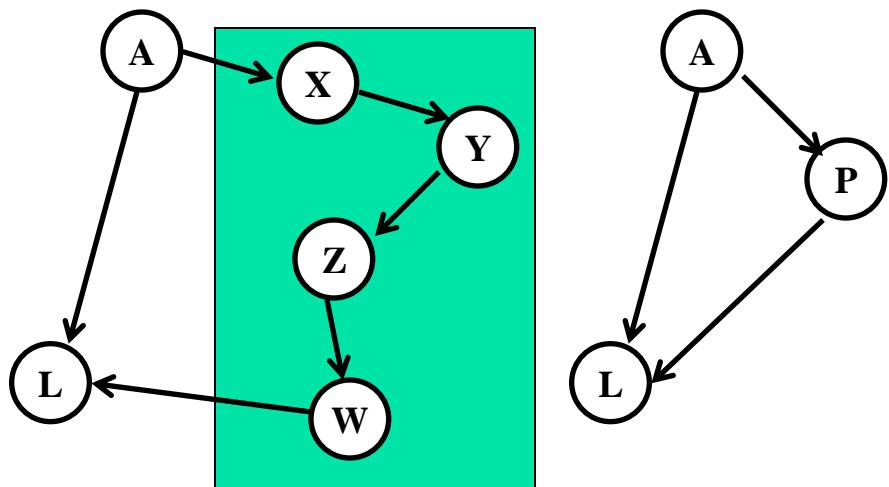
Adopted with permission from Software Testing: A Craftsman's Approach, by Paul Jorgensen, CRC PRESS, third edition.

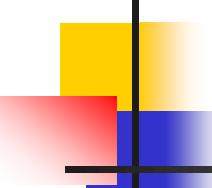
# In Class Activity

Can you condense the following directed graph and then calculate its Essential Complexity?



$$E(G) = e - n + 2p = 3 - 3 + 2(1) = 2$$





# NextDate Pseudo-code

Program NextDate

```

Dim tomorrowDay,tomorrowMonth,tomorrowYear As Integer
Dim day,month,year As Integer
1. Output ("Enter today's date in the form MM DD YYYY")
2. Input (month,day,year)
3. Case month Of
4. Case 1: month Is 1,3,5,7,8,Or 10: '31 day months (except Dec.)
5.   If day < 31
6.     Then tomorrowDay = day + 1
7.   Else
8.     tomorrowDay = 1
9.     tomorrowMonth = month + 1
10. EndIf
11. Case 2: month Is 4,6,9,Or 11 '30 day months
12. If day < 30
13.   Then tomorrowDay = day + 1
14.   Else
15.     tomorrowDay = 1
16.     tomorrowMonth = month + 1
17. EndIf
18. Case 3: month Is 12: 'December
19. If day < 31
20.   Then tomorrowDay = day + 1
21.   Else
22.     tomorrowDay = 1
23.     tomorrowMonth = 1
24.     If year = 2012
25.       Then Output ("2012 is over")
26.       Else tomorrow.year = year + 1
27.     EndIf
28. EndIf

```

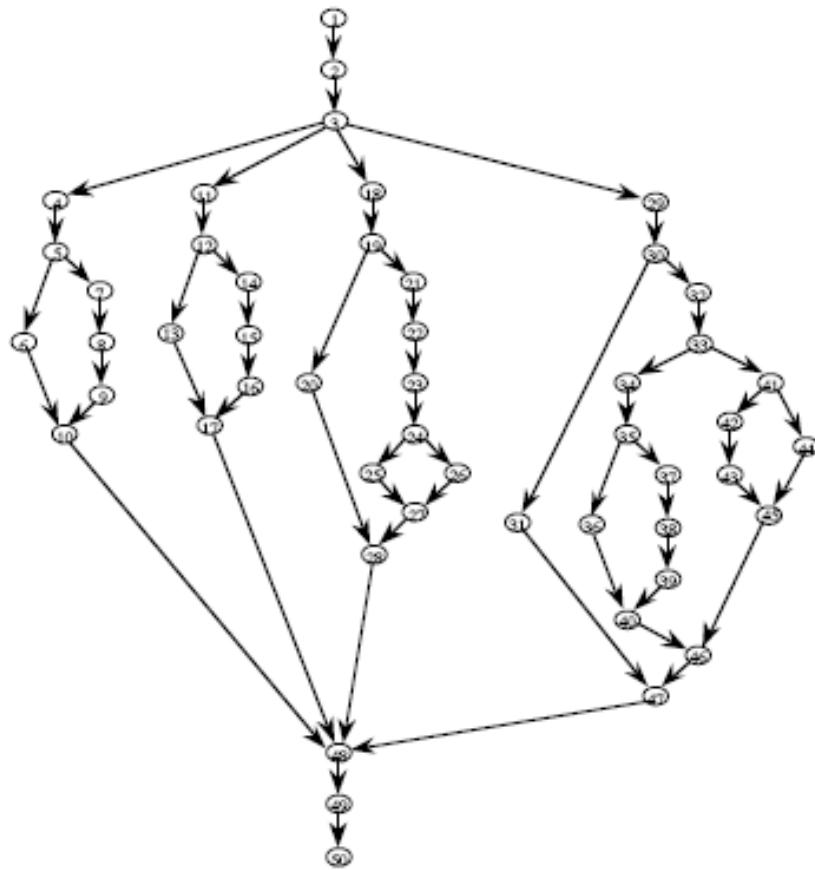
```

29. Case 4: month Is 2: 'February
30. If day < 28
31.   Then tomorrowDay = day + 1
32. Else
33.   If day = 28
34.     Then
35.       If ((year MOD 4)=0)AND((year MOD 400)≠0)
36.         Then tomorrowDay = 29 'leap year
37.       Else      'not a leap year
38.         tomorrowDay = 1
39.         tomorrowMonth = 3
40.     EndIf
41. Else If day = 29
42.   Then tomorrowDay = 1
43.   tomorrowMonth = 3
44. Else Output("Cannot have Feb.", day)
45. EndIf
46. EndIf
47. EndCase
48. EndCase
49. Output ("Tomorrow's date is", tomorrowMonth,
           tomorrowDay, tomorrowYear)
50. End NextDate

```

*Adopted with permission from Software Testing: A Craftsman's Approach, by Paul Jorgensen, CRC PRESS, third edition.*

# NextDate Program Graph



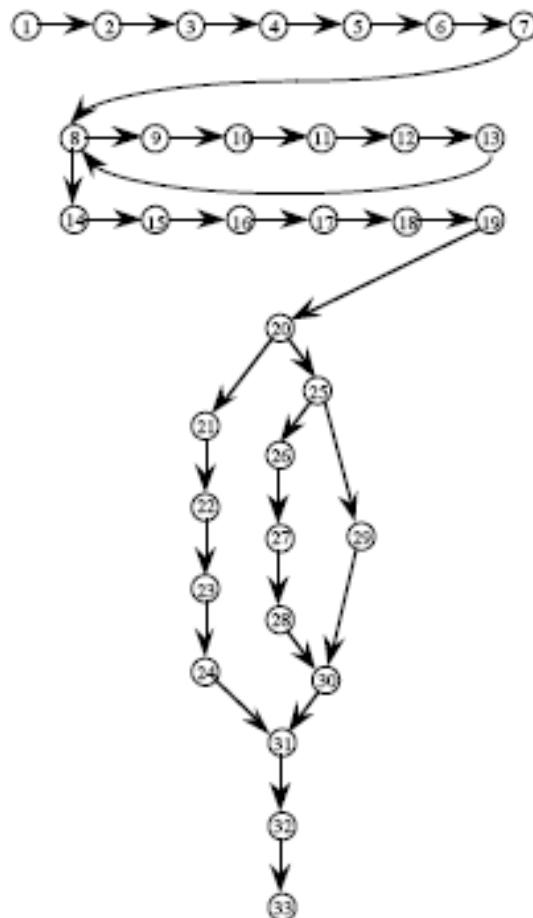
*Adopted with permission from Software Testing: A Craftsman's Approach, by Paul Jorgensen, CRC PRESS, third edition.*

# Commission Pseudo-code

Program Commission

```
Dim lockPrice, stockPrice, barrelPrice As Real
Dim locks, stocks, barrels As Integer
Dim totalLocks, totalStocks As Integer
Dim totalBarrels As Integer
Dim lockSales, stockSales As Real
Dim barrelSales As Real
Dim sales, commission As Real
1 lockPrice = 45.0
2 stockPrice = 30.0
3 barrelPrice = 25.0
4 totalLocks = 0
5 totalStocks = 0
6 totalBarrels = 0
7 Input(locks)
8 While NOT(locks = -1)
9   Input(stocks, barrels)
10  totalLocks = totalLocks + locks
11  totalStocks = totalStocks + stocks
12  totalBarrels = totalBarrels + barrels
13  Input(locks)
14 EndWhile
15 Output("Locks sold: ", totalLocks)
16 Output("Stocks sold: ", totalStocks)
17 Output("Barrels sold: ", totalBarrels)
18 sales = lockPrice*totalLocks +
           stockPrice*totalStocks + barrelPrice * totalBarrels
19 Output("Total sales: ", sales)
20 If (sales > 1800.0)
21   Then
22     commission = 0.10 * 1000.0
23     commission = commission + 0.15 * 800.0
24     commission = commission + 0.20*(sales-1800.0)
25 Else If (sales > 1000.0)
26   Then
27     commission = 0.10 * 1000.0
28     commission = commission + 0.15*(sales-1000.0)
29   Else commission = 0.10 * sales
30 EndIf
31 EndIf
32 Output("Commission is $", commission)
33 End Commission
```

# Commission Program Graph

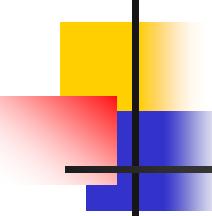


*Adopted with permission from Software Testing: A Craftsman's Approach, by Paul Jorgensen, CRC PRESS, third edition.*

# Software Testing & Quality Assurance

## *Data Flow Testing*

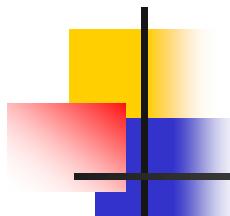
- Define/Use Testing
- Slice-Based Testing
- Guidelines and Observations



# Credits & Readings

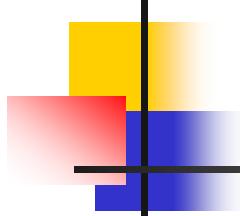
The material included in these slides are mostly adopted from the following books:

- *Software Testing: A Craftsman's Approach*, by Paul Jorgensen, CRC PRESS, third edition, ISBN: 0-8493-7475-8
- Cem Kaner, Jack Falk, Hung Q. Nguyen, "Testing Computer Software" Wiley (see also <http://www.testingeducation.org/>)
- Paul Ammann and Jeff Offutt, "Introduction to Software Testing", Cambridge University Press
- Beizer, Boris, "Software Testing and Quality Assurance", Van Nostrand Reinhold
- Glen Myers, "The Art of Software Testing"
- Stephen R. Schach, "Software Engineering", Richard D. Irwin, Inc. and Aksen Associates, Inc.



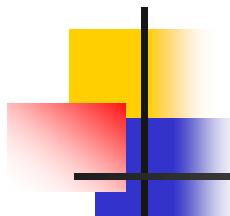
# Data Flow Testing

- Shortcomings of Path Testing
  - Testing All-Nodes and All-Edges in a control flow graph may miss significant test cases
  - Testing All-Paths in a control flow graph is often too time-consuming
- Can we select a subset of these paths that will reveal the most faults?
- Data Flow Testing focuses on the points at which variables receive values and the points at which these values are used/referenced



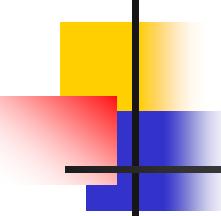
# Data Flow Analysis

- Can reveal interesting bugs such as
  - A variable that is defined but never used
  - A variable that is used but never defined
  - A variable that is defined twice before it is used
- Paths from the definition of a variable to its use are more likely to contain bugs



# Definitions

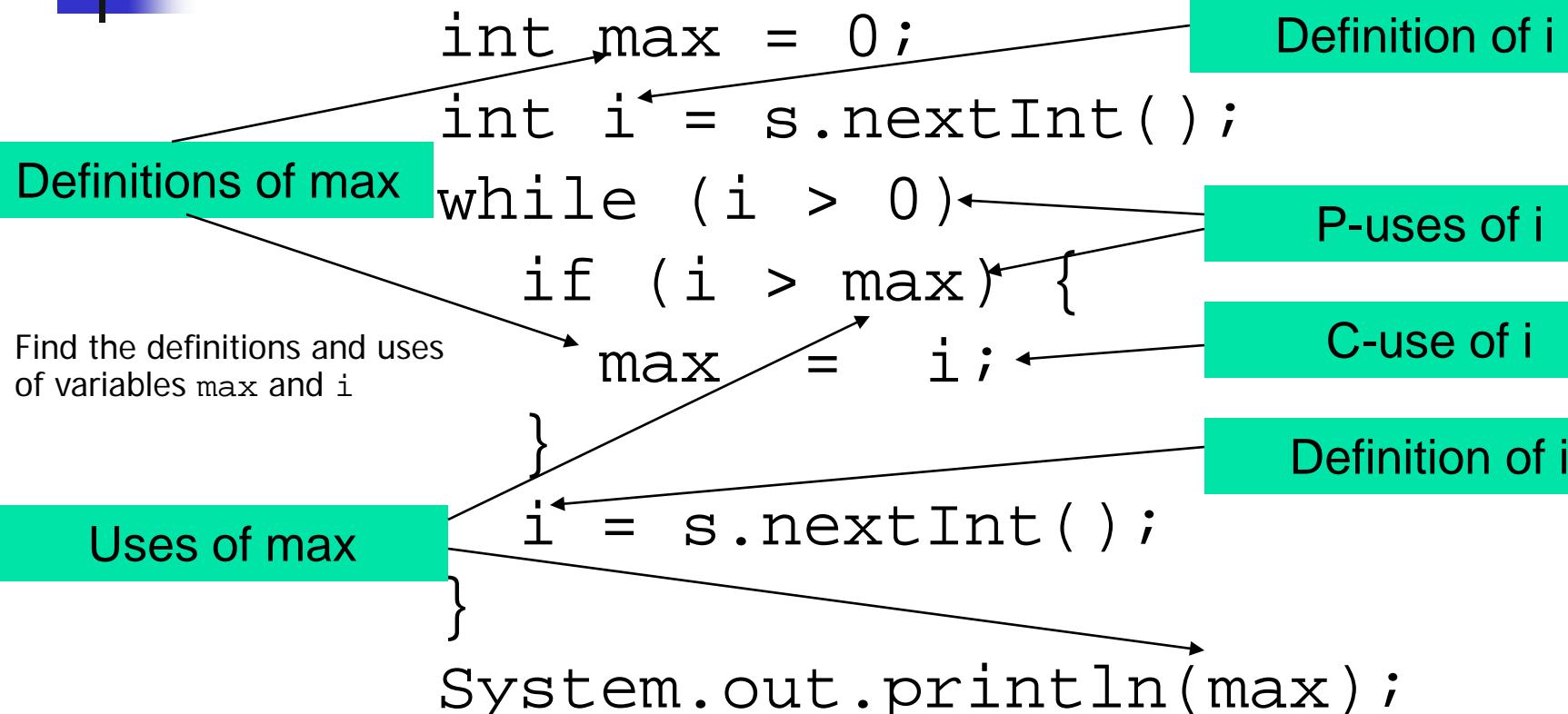
- A node in the program graph is a *defining* node for variable  $v$  if the value of  $v$  is defined at the statement fragment in that node
  - Input, assignment (left side), procedure calls
  - Contents of memory location are changed
- A node in the program graph is a *usage* node for variable  $v$  if the value of  $v$  is used at the statement fragment in that node
  - Output, assignment (right side), conditionals
  - Contents of memory location remain unchanged



# More Definitions

- A usage node is a predicate use (*P-Use*) if variable  $v$  appears in a predicate expression
- A usage node is a computation use (*C-Use*) if variable  $v$  appears in a computation
- A definition-use path (*du-path*) with respect to a variable  $v$  is a path whose first node is a defining node for  $v$ , and its last node is a usage node for  $v$
- A du-path with no other defining node for  $v$  is a definition-clear path (*dc-path*)

# An Example



A

```
int max = 0;  
int i = s.nextInt();
```

B

```
while (i > 0)
```

U

d

Find all du-paths  
of variable **max**

C

```
if (i > max)
```

U

E

```
i = s.nextInt();
```

U

D

```
max = i;
```

Show the  
definitions and  
uses of variable **i**

Find all du-paths  
of variable **i**

Variable **i**: A-B, A-C,  
A-D, E-B, E-C, E-D

Variable  
**max**:  
A-F, A-C,  
D-C, D-F

F

```
System.out.println(max);
```

# Commission Pseudo-code

```
1. Program Commission (INPUT,OUTPUT)
2. Dim locks, stocks, barrels As Integer
3. Dim lockPrice, stockPrice, barrelPrice As Real
4. Dim totalLocks, totalStocks, totalBarrels As Integer
5. Dim lockSales, stockSales, barrelSales As Real
6. Dim sales, commission As Real
7. lockPrice = 45.0
8. stockPrice = 30.0
9. barrelPrice = 25.0
10. totalLocks = 0
11. totalStocks = 0
12. totalBarrels = 0
13. Input(locks)
14. While NOT(locks = -1)
15.   Input(stocks, barrels)
16.   totalLocks = totalLocks + locks
17.   totalStocks = totalStocks + stocks
18.   totalBarrels = totalBarrels + barrels
19.   Input(locks)
20. EndWhile
21. Output("Locks sold: ", totalLocks)
22. Output("Stocks sold: ", totalStocks)
23. Output("Barrels sold: ", totalBarrels)
23. Output("Barrels sold: ", totalBarrels)
24. lockSales = lockPrice * totalLocks
25. stockSales = stockPrice * totalStocks
26. barrelSales = barrelPrice * totalBarrels
27. sales = lockSales + stockSales + barrelSales
28. Output("Total sales: ", sales)
29. If (sales > 1800.0)
30. Then
31.   commission = 0.10 * 1000.0
32.   commission = commission + 0.15 * 800.0
33.   commission = commission + 0.20 *(sales-1800.0)
34. Else If (sales > 1000.0)
35.   Then
36.     commission = 0.10 * 1000.0
37.     commission = commission + 0.15 *(sales-1000.0)
38.   Else
39.     commission = 0.10 * sales
40.   EndIf
41. EndIf
42. Output("Commission is $", commission)
43. End Commission
```

*Adopted with permission from Software Testing: A Craftsman's Approach, by Paul Jorgensen, CRC PRESS, third edition.*

# Defining & Usage Nodes for the Commission Problem

```
1. Program Commission (INPUT,OUTPUT)
2. Dim locks, stocks, barrels As Integer
3. Dim lockPrice, stockPrice, barrelPrice As Real
4. Dim totalLocks, totalStocks, totalBarrels As Integer
5. Dim lockSales, stockSales, barrelSales As Real
6. Dim sales, commission As Real
7. lockPrice = 45.0
8. stockPrice = 30.0
9. barrelPrice = 25.0
10. totalLocks = 0
11. totalStocks = 0
12. totalBarrels = 0
13. Input(locks)
14. While NOT(locks = -1)
15.   Input(stocks, barrels)
16.   totalLocks = totalLocks + locks
17.   totalStocks = totalStocks + stocks
18.   totalBarrels = totalBarrels + barrels
19.   Input(barrels)
20. EndWhile
21. Output("Locks sold: ", totalLocks)
22. Output("Stocks sold: ", totalStocks)
23. Output("Barrels sold: ", totalBarrels)
```

Variable	Defined at Node	Used at Node
locks	13, 19	14, 16
stocks	15	17
barrels	15	18
totalLocks	10, 16	16, 21, 24
totalStocks	11, 17	17, 22, 25
totalBarrels	12, 18	18, 23, 26

*Adopted with permission from Software Testing: A Craftsman's Approach, by Paul Jorgensen, CRC PRESS, third edition.*

# Defining & Usage Nodes for the Commission Problem (continued)

```
23. Output("Barrels sold: ", totalBarrels)
24. lockSales = lockPrice * totalLocks
25. stockSales = stockPrice * totalStocks
26. barrelSales = barrelPrice * totalBarrels
27. sales = lockSales + stockSales + barrelSales
28. Output("Total sales: ", sales)
29. If (sales > 1800.0)
30. Then
31.   commission = 0.10 * 1000.0
32.   commission = commission + 0.15 * 800.0
33.   commission = commission + 0.20 *(sales-1800.0)
34. Else If (sales > 1000.0)
35.   Then
36.     commission = 0.10 * 1000.0
37.     commission = commission + 0.15 *(sales-1000.0)
38.   Else
39.     commission = 0.10 * sales
40. EndIf
41. EndIf
42. Output("Commission is $", commission)
43. End Commission
```

Variable	Defined at Node	Used at Node
locks	13, 19	14, 16
stocks	15	17
barrels	15	18
totalLocks	10, 16	16, 21, 24
totalStocks	11, 17	17, 22, 25
totalBarrels	12, 18	18, 23, 26

*Adopted with permission from Software Testing: A Craftsman's Approach, by Paul Jorgensen, CRC PRESS, third edition.*

# Du-Paths for the “locks” Variable in the Commission Problem

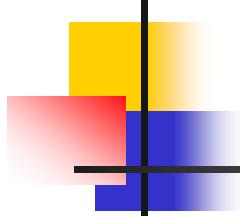
```
13. Input(locks)
14. While NOT(locks = -1)
15.   Input(stocks, barrels)
16.   totalLocks = totalLocks+locks
17.   totalStocks = totalStocks+stocks
18.   totalBarrels = totalBarrels+barrels
19.   Input(locks)
20. EndWhile
```

DEF (locks, 13), DEF (locks, 19)  
USE (locks, 14), a predicate use  
USE (locks, 16), a computation use

du-paths for locks are the node sequences  
 $\langle 13, 14 \rangle$  (a dc-path),  
 $\langle 13, 14, 15, 16 \rangle$ ,  
 $\langle 19, 20, 14 \rangle$ ,  
 $\langle 19, 20, 14, 15, 16 \rangle$

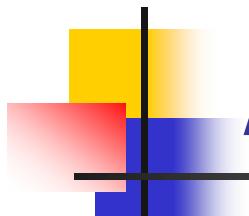
Is  $\langle 13, 14, 15, 16 \rangle$  definition clear?  
Is  $\langle 19, 20, 14, 15, 16 \rangle$  definition clear?

Adopted with permission from Software Testing: A Craftsman's Approach, by Paul Jorgensen, CRC PRESS, third edition.



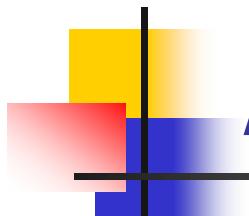
# Data Flow Coverage Metrics

- Based on these definitions we can define a set of coverage metrics for a set of test cases
- We have already seen
  - All-Nodes
  - All-Edges
  - All-Paths



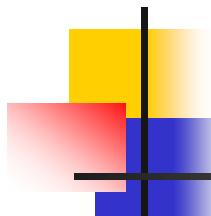
# All-Defs Criterion

- For every variable  $v$ 
  - For every defining node  $d$  of  $v$ 
    - There exists at least one test case that follows a path from  $d$  to a usage node of  $v$



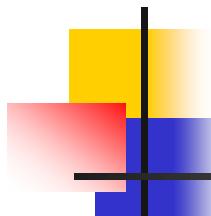
# All-Uses Criterion

- For every variable  $v$ 
  - For every defining node  $d$  of  $v$ 
    - For every usage node  $u$  of  $v$ 
      - There exists at least one test case that follows a path from  $d$  to  $u$



# All-P-Uses / Some-C-Uses

- For every variable  $v$ 
  - For every defining node  $d$  of  $v$ 
    - For every P-Use  $u$  of  $v$ 
      - There exists at least one test case that follows a path from  $d$  to  $u$
      - If there is no P-Use of  $v$ , there must exist at least one test case that follows a path from  $d$  to a C-Use of  $v$

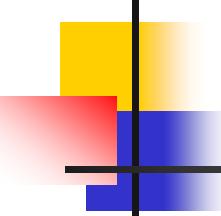


# All-C-Uses / Some-P-Uses

- For every variable  $v$ 
  - For every defining node  $d$  of  $v$ 
    - For every C-Use  $u$  of  $v$ 
      - There exists at least one test case that follows a path from  $d$  to  $u$
      - If there is no C-Use of  $v$ , there must exist at least one test case that follows a path from  $d$  to a P-Use of  $v$

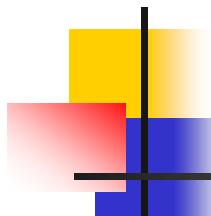
# Strategy for Generating Define/Use Test Cases

- For a particular variable:
  - find all its definition and usage nodes
  - find the du-paths and dc-paths among these
  - for each path, devise a "suitable" (functional) set of test cases
- Note: du-paths and dc-paths have both static and dynamic interpretations
  - Static: just as seen in the source code
  - Dynamic: must consider execution-time flow (particularly for loops)
- Definition clear paths are easier to test
  - No need to check each definition node, as is necessary for du-paths



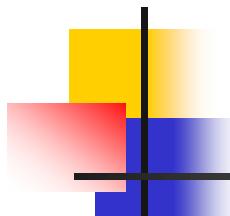
# Some Observations

- Data flow testing is indicated in
  - Computation-intensive applications
  - “Long” programs
  - Programs with many variables
- A definition-clear du-path represents a small function that can be tested by itself
- If a du-path is not definition-clear, it should be tested for each defining node
- Aliasing of variables causes serious problems
- Working things out by hand for anything but small methods is hopeless
- Compiler-based tools help in determining coverage values



# Slice-Based Testing<sup>1</sup>

- Informally, a program slice is a set of program statements that contribute to or affect a value for a variable at some point in the program
- The idea of a slice is to separate a program into components that have some useful (functional) meaning



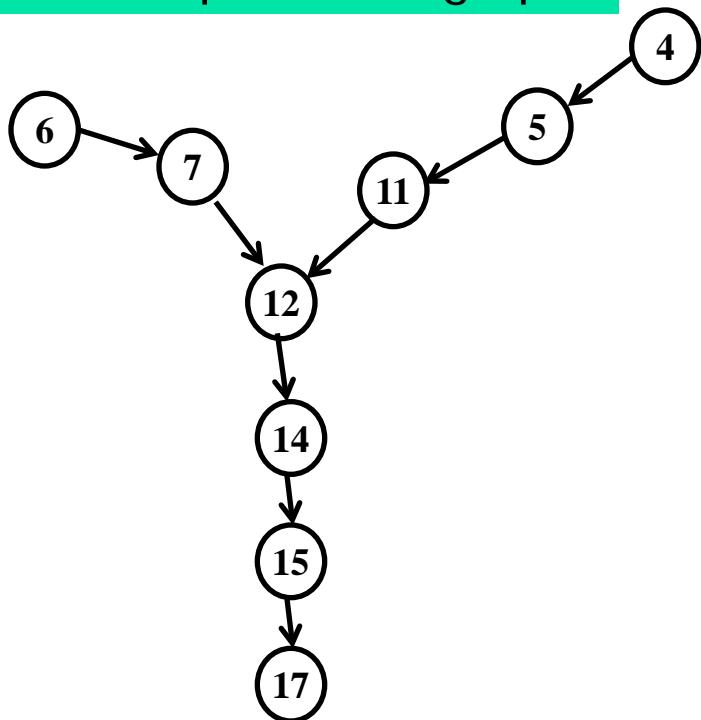
# Slice-Based Testing<sup>2</sup>

- More formally, given a program P and a set V of variables in P, a slice on the variable set V at statement n, written  $S(V, n)$ , is the set of all statements in P that contribute to the values of variables in V
- With slices we consider 5 forms of usage:
  - P-use used in predicate (decision)
  - C-use used in computation
  - O-use used for output
  - L-use used for location (pointers, subscripts)
  - I-use iteration (internal counters, loop indices)
- Also, we consider 2 forms of definition nodes
  - I-def defined by input
  - A-def defined by assignment

# An Example

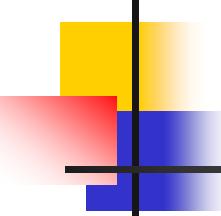
Find the (backward) slice on out1 at line 17 & the data dependence graph

Data dependence graph



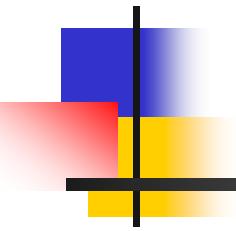
```

1. int n1, n2, n3;
2. int a, b, c, x, y, z;
3. int out1, out2, out3;
4. read(n1);
5. a = n1;
6. read(n2);
7. b = n2;
8. read(n3);
9. c = n3;
10. x = b + c;
11. y = a + 1;
12. z = y + b;
13. out2 = z + 1;
14. z = z + 2;
15. out1 = z;
16. out3 = x;
17. print(out1, out2, out3);
  
```



# Observations and Guidelines

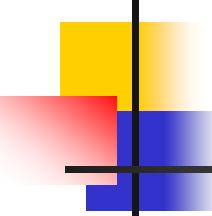
- Working “backwards” from points in a program (where a fault is suspected) is the way good programmers think when they debug code (according to some researchers)
- Make slices on a single variable
- Make slices for all A-def nodes
- Make slices for P-use nodes
- Slices on non-P-use usage nodes are not very interesting
- Consider making slices compilable
- Slices do not map nicely into test cases
- Slices provide a handy way to eliminate interaction among variables
- Use the slice composition approach to redevelop difficult sections of code, and test these slices before you *splice* (compose) them with other slices



# Software Testing & Quality Assurance

## *Structural Testing Review*

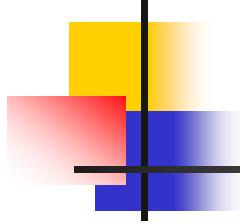
- Structural Metrics
- Test Coverage Metrics
- Coverage Usefulness & Measurement



# Credits & Readings

The material included in these slides are mostly adopted from the following books:

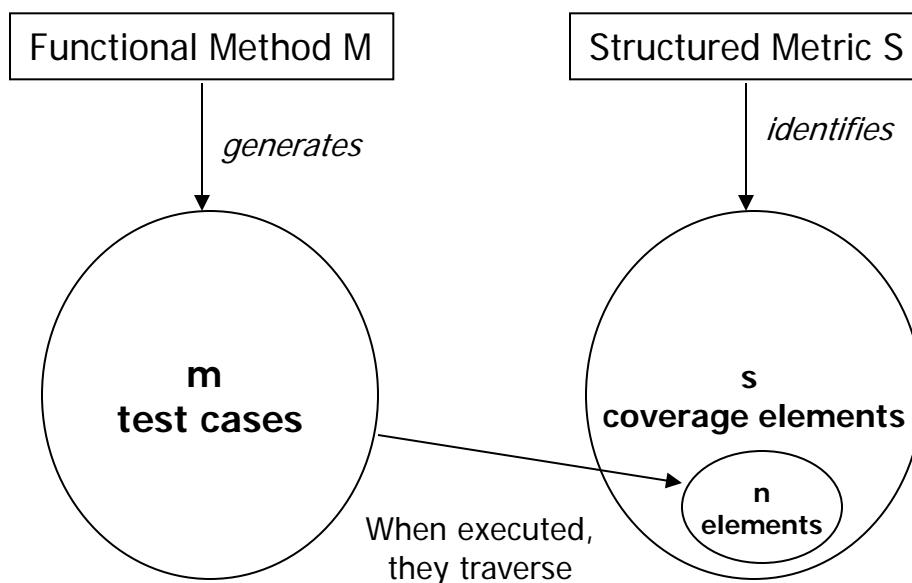
- *Software Testing: A Craftsman's Approach*, by Paul Jorgensen, CRC PRESS, third edition, ISBN: 0-8493-7475-8
- Cem Kaner, Jack Falk, Hung Q. Nguyen, "*Testing Computer Software*" Wiley (see also <http://www.testingeducation.org/>)
- Paul Ammann and Jeff Offutt, "*Introduction to Software Testing*", Cambridge University Press
- Beizer, Boris, "*Software Testing and Quality Assurance*", Van Nostrand Reinhold
- Glen Myers, "*The Art of Software Testing*"



# Measuring Gaps & Redundancy

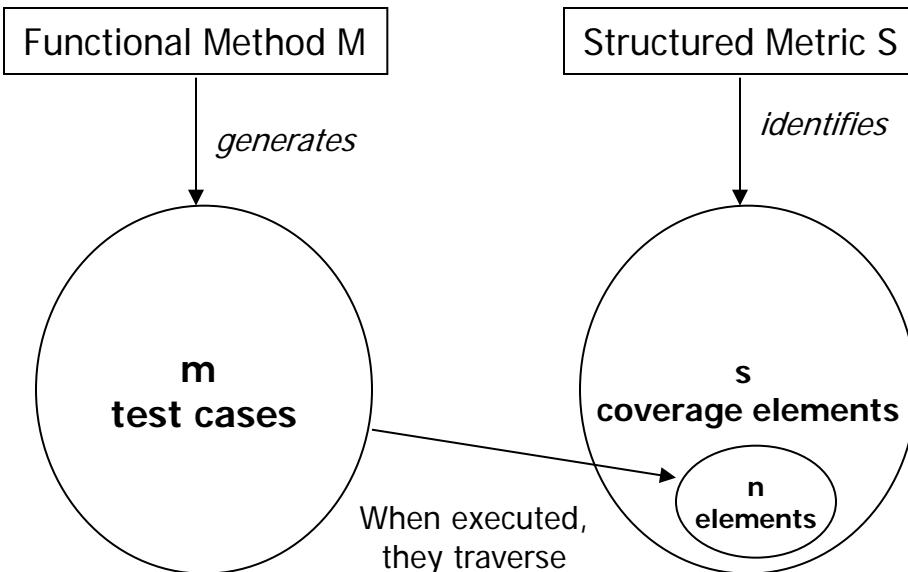
- We have seen that functional testing methods may produce test suites with serious gaps and a lot of redundancy
- Structural testing analysis allows to measure the extent of these problems
- Recall that the structural metrics are always expressed in terms of something countable, such as
  - The number of program paths
  - The number of decision-to-decision paths
  - The number of slices

# Evaluation of Functional Testing Based on Structural Metrics



- We assume that
  - A functional testing method **M** produces **m** test cases which are tracked with respect to
  - a structural metric **S** that identifies **s** coverage elements in the unit under test
- When the **m** test cases are executed, they traverse **n** (of the **s**) coverage elements

# Metrics for Functional Testing Effectiveness



All three metrics provide a quantitative way to evaluate the effectiveness of any functional testing method with respect to a structural metric

- Coverage of method M with respect to metric S as
  - $C(M,S) = n/s$
  - When it is less than 1, it means that that there are gaps in the coverage with respect to that metric
- Redundancy of method M with respect to metric S as
  - $R(M,S) = m/s$
  - The bigger it is , the bigger the redundancy of test cases
- Net redundancy of method M with respect to metric S as
  - $NR(M,S) = m/n$
  - It refers to things actually traversed, not to the total space of things to be traversed (more useful)
- Best possible value for these metrics is 1
- Note: when  $C(M,S) = 1$ , algebra forces  $R(M,S) = NR(M,S)$

# Metric Values for Triangle

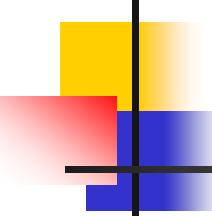
- The implementation has exactly 11 feasible paths shown in Table 11.1 of our textbook
- Table 11.2 shows the test cases generated using Boundary Value technique
  - Paths p1, p2, p3, p4, p5, p6 and p7 are covered and paths p8, p9, p10, p11 are missed
  - If Worst-Case Boundary Value testing was used, it would yield 125 test cases, providing full path coverage, but high redundancy

Method	m	n	s	C(M,S) n/s	R(M,S) m/s	NR(M,S) m/n
Boundary Value	15	7	11	0.64	1.36	2.14
Worst Case Analysis	125	11	11	1.00	11.36	11.36
WN ECT	4	4	11	0.36	0.36	1.00
Decision Table	8	8	11	0.72	0.72	1.00

# More Comparisons

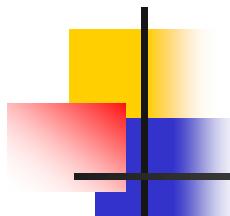
Method	m	n	s	C(M,S)	R(M,S)	NR(M,S)
Triangle Program						
BVA	15	7	11	0.64	1.36	2.14
Worst Case BVA	125	11	11	1.00	11.36	11.36
Commission Program						
Output BVA	25	11	11	1.00	2.27	2.27
Decision Table	3	11	11	1.00	0.27	0.27
DD-Path	25	11	11	1.00	2.27	2.27
DU-Path	25	33	33	1.00	0.76	0.76
Slices	25	40	40	1.00	0.63	0.63

Adopted with permission from Software Testing: A Craftsman's Approach, by Paul Jorgensen, CRC PRESS, third edition.



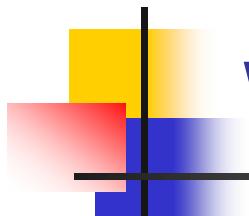
# Coverage Usefulness

- 100% coverage is never a guarantee of bug-free software
- Coverage reports can
  - point out inadequate test suites
  - suggest the presence of surprises, such as blind spots in the test design
  - Help identify parts of the implementation that require structural testing



# How to Measure Coverage?

- The source code is *instrumented*
- Depending on the code coverage model, code that writes to a trace file is inserted in every branch, statement, etc.
- Most commercial tools measure segment and branch coverage



# When Should We Stop Testing?

- When you run out of time?
- When continued testing reveals no new faults?
- When you cannot think of any new test cases?
- When mandated coverage has been attained?
- When all faults have been removed?