COSC 340: Software Engineering

Validation & Verification

Adapted from slides by Ravi Sethi, University of Arizona

Software Quality

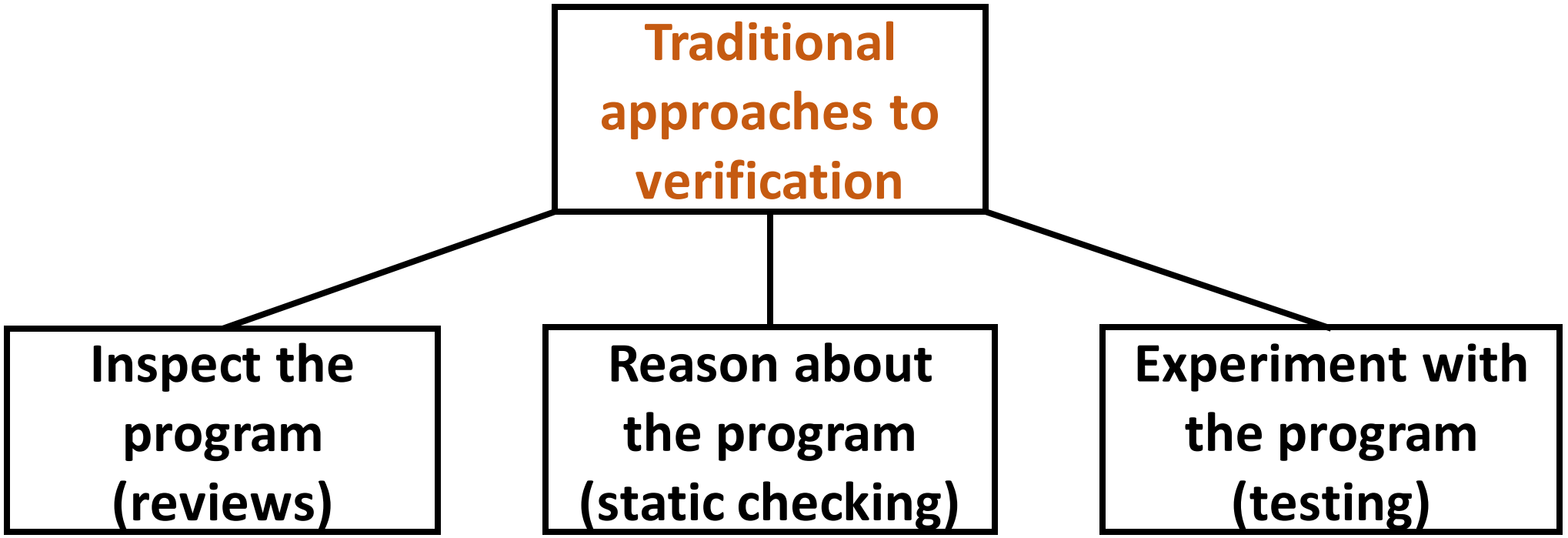
- **Questions to consider**
  - What is software quality?
  - What is quality as perceived by the customer?
  - What is a defect?
  - How do you measure program size?
- **Verification can improve software quality**
  - How effective are different verification techniques

What is Validation and Verification (V&V)?

- Both are processes and both have many variations
- **Validation: Have I built the right product?**
  - Confirm the product conforms to requirements
  - **Architecture reviews** are an approach to validation
- **Verification: Have I built the product right?**
  - Confirmation by examination and through the provision of objective evidence that specified requirements have been fulfilled (ISO 9000)
  - **Design and code reviews** are an approach to verification

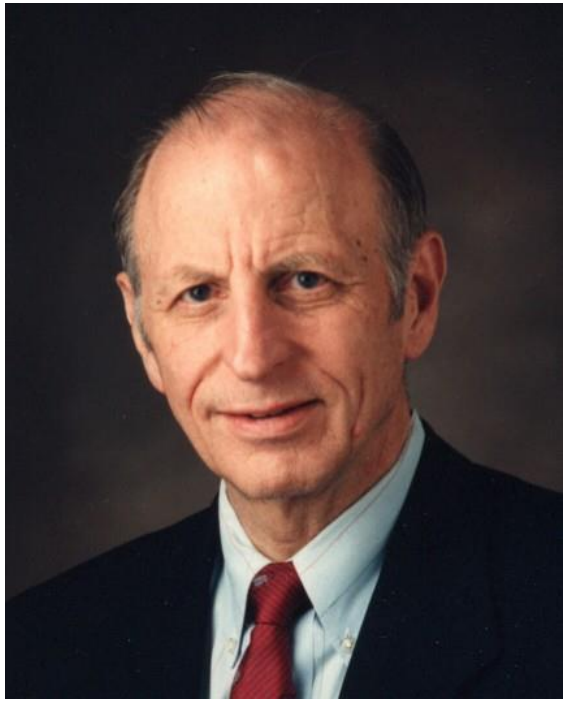
Verification, as widely practiced

- **Verification if building the product right**



Product Defects

- "While the classical definition of product quality must focus on the customer's needs, … I will concentrate on only the defect aspect of quality. This is because the cost and time spent in removing software defects currently consumes such a large proportion of our efforts that it overwhelms everything else."
  – Humphrey, Watts. *Defective Software Works*. (2004)

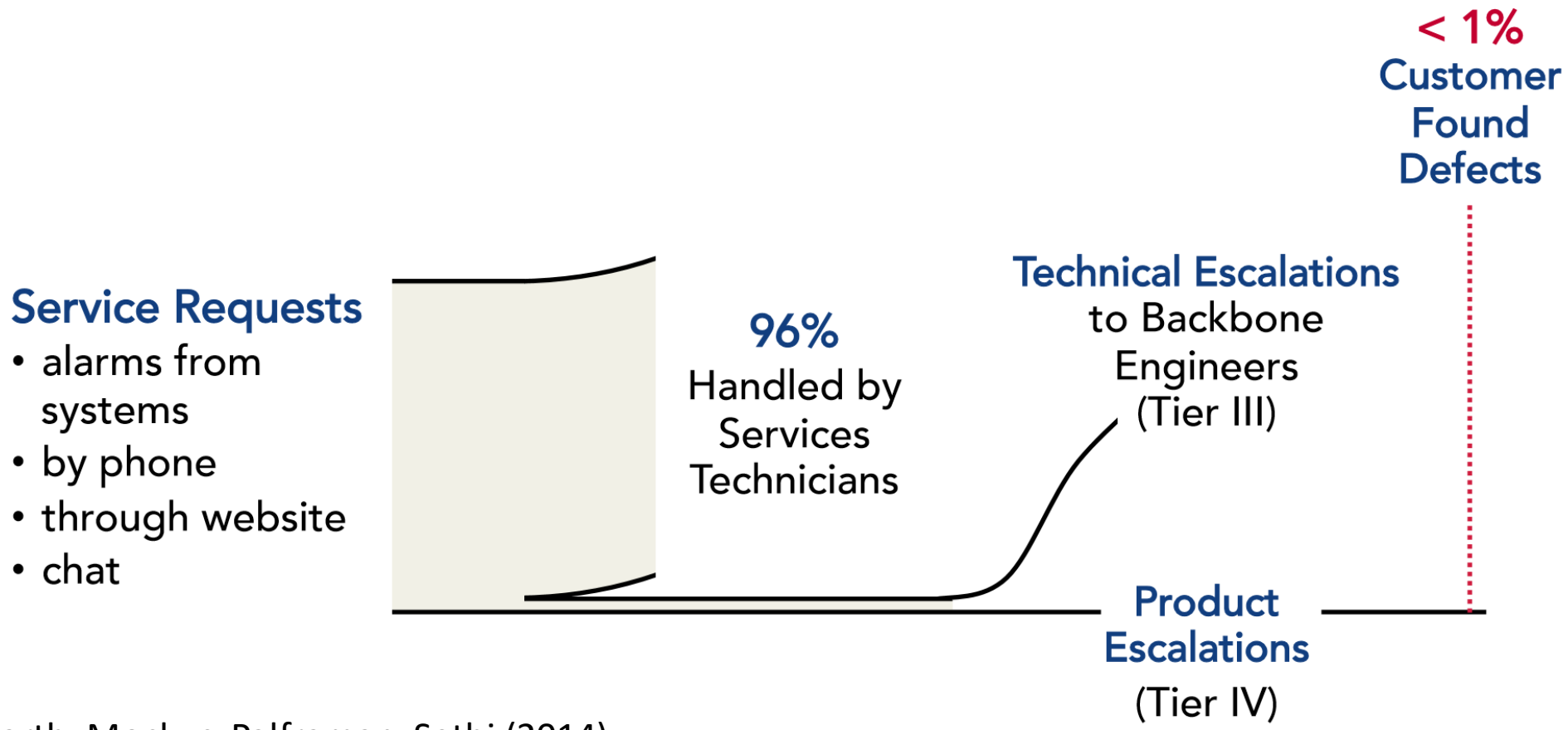


What is a defect?

Definitions vary …

- **Defect severity levels (ISTQB)**
  - Severity 1: Total stoppage
  - Severity 2: Major error
  - Severity 3: Minor error
  - Severity 4: Cosmetic error
- **Typical definition of customer found defect (CFD)**
  - Customer service report that has gone through levels of screening
  - Identified as a Severity 1 or 2 defect in a product/system
  - And, is the first report of the defect
  - Subsequent (duplicate) reports are counted separately

What is a Customer Found Defect (CFD)?



Source: Hackbarth, Mockus, Palframan, Sethi (2014)

How to Measure Size?

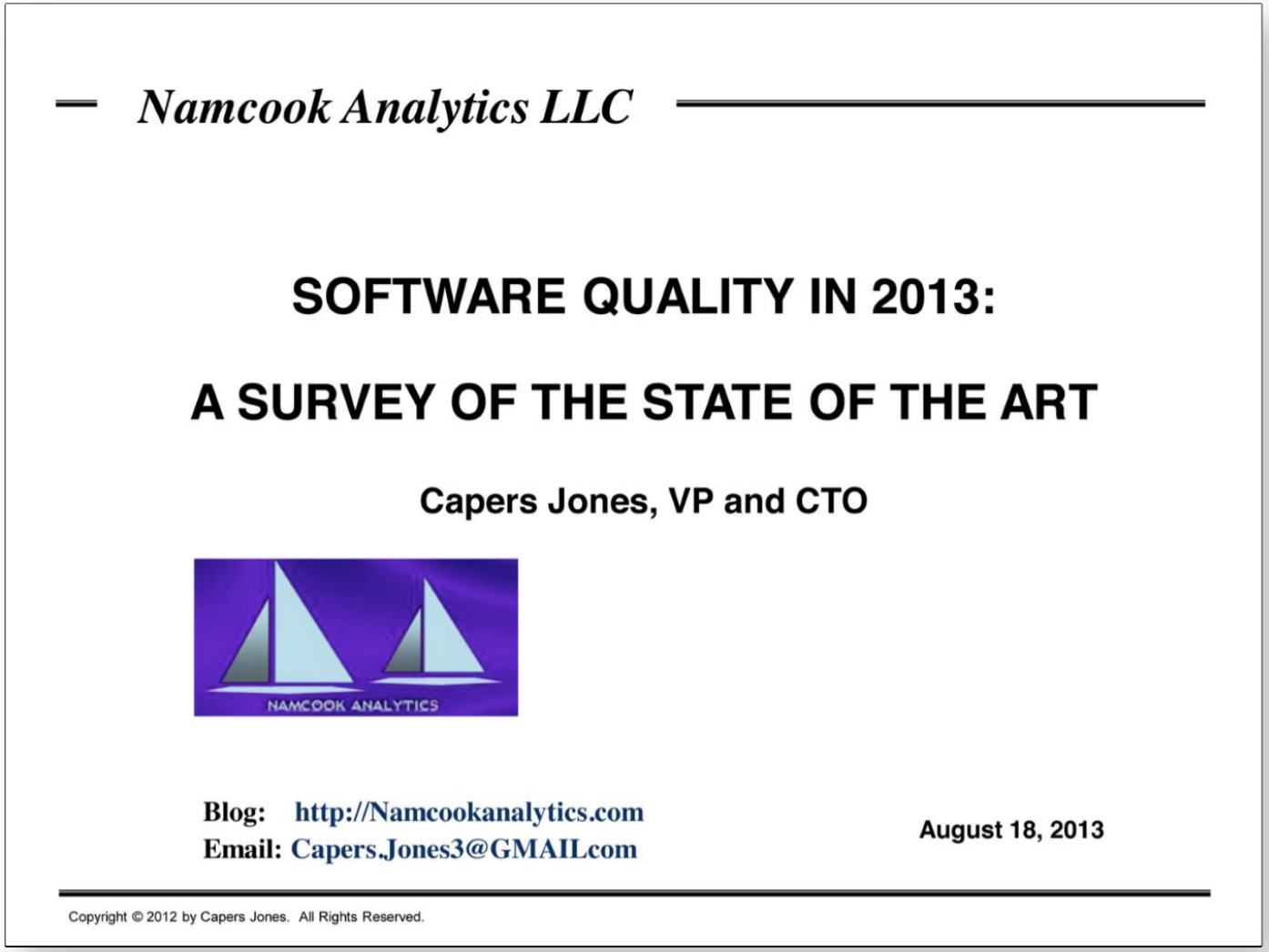
- Despite its limitations, lines of code (LOC) is still the primary metric
- Issues with lines of code
  - For the same functionality, lines of code vary with language
  - Across languages, productivity in lines of code is relatively constant
- Function Points
  - Metric introduced in the 1970s at IBM to measure functionality
  - Issue: different people come up with different answers
  - Correlates with lines of code
  - So, why consider Function Points? Because easier to compare across languages and projects

Issues with Metrics

- Metrics can be subverted!

Data from 13,500 Projects, 1984 – 2013

Published in Capers Jones's books & "200 journal articles and monographs"



Namcook Analytics LLC

SOFTWARE QUALITY IN 2013:

A SURVEY OF THE STATE OF THE ART

Capers Jones, VP and CTO

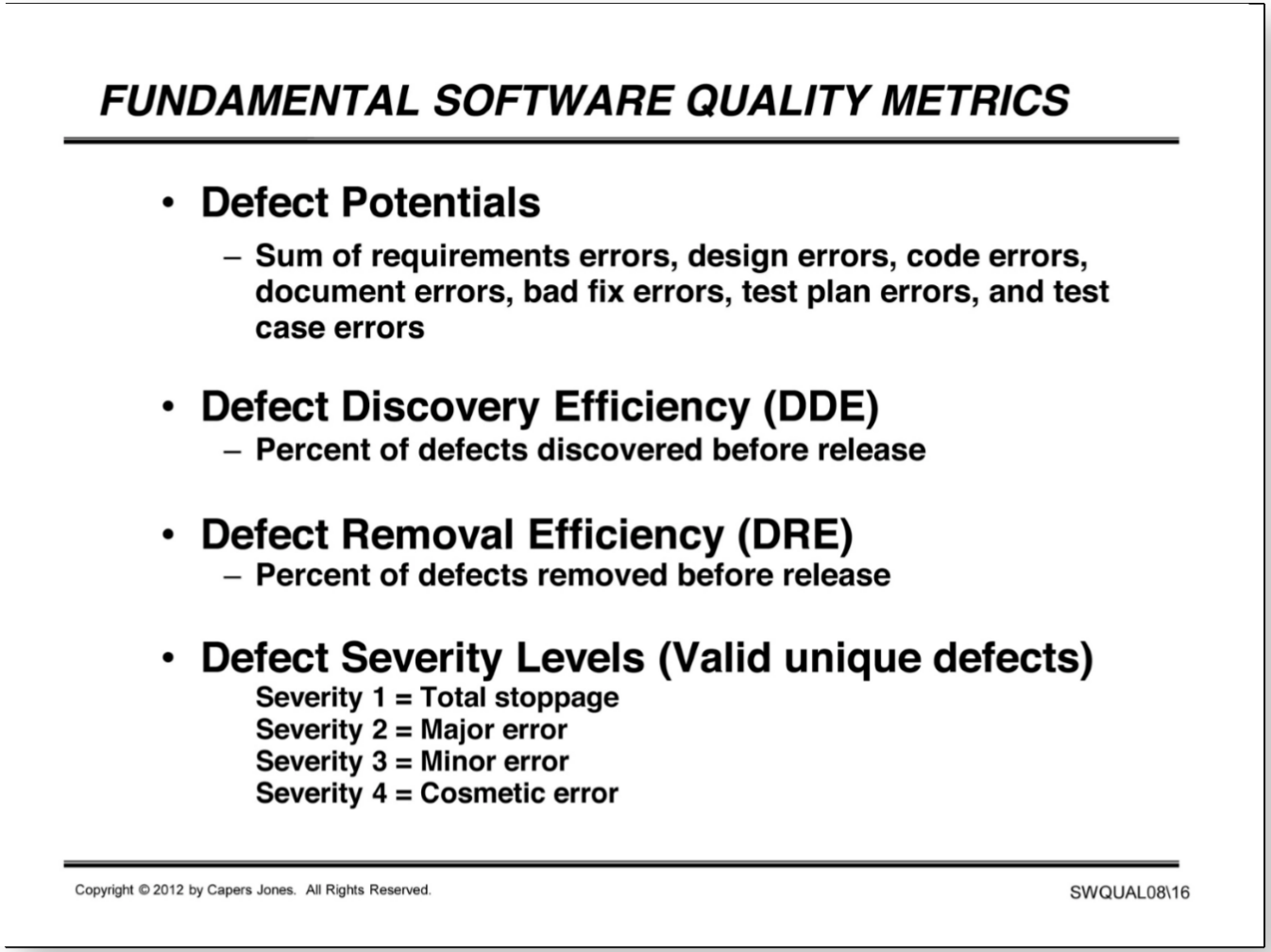
NAMCOOK ANALYTICS

Blog: http://Namcookanalytics.com
Email: Capers.Jones3@GMAILcom

August 18, 2013

Software Quality: State of the Art

Measures focus primarily on development



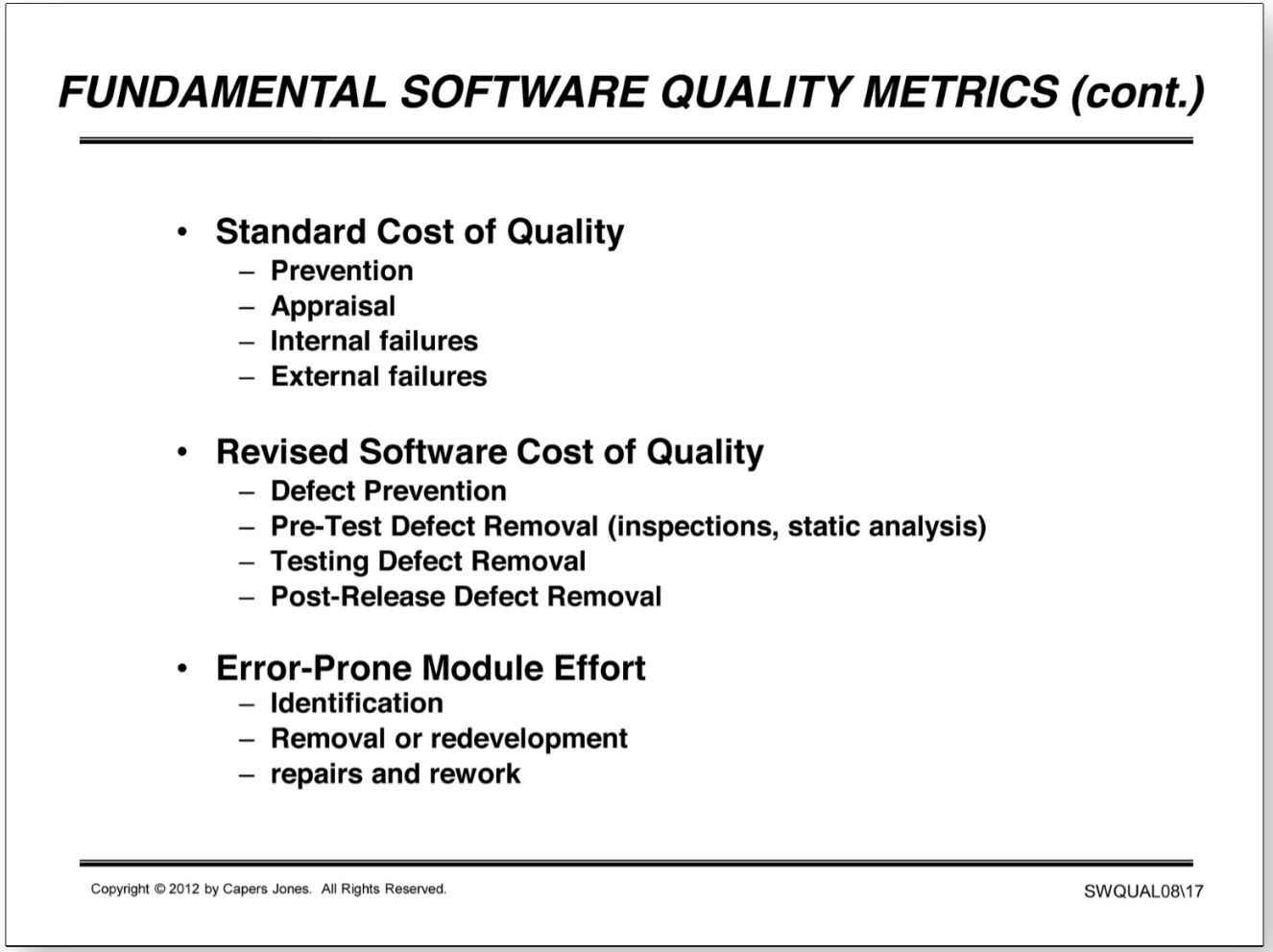
FUNDAMENTAL SOFTWARE QUALITY METRICS

- **Defect Potentials**
  - Sum of requirements errors, design errors, code errors, document errors, bad fix errors, test plan errors, and test case errors
- **Defect Discovery Efficiency (DDE)**
  - Percent of defects discovered before release
- **Defect Removal Efficiency (DRE)**
  - Percent of defects removed before release
- **Defect Severity Levels (Valid unique defects)**
  - Severity 1 = Total stoppage
  - Severity 2 = Major error
  - Severity 3 = Minor error
  - Severity 4 = Cosmetic error

SWQUAL08\16

Software Quality: State of the Art

Measures focus primarily on development

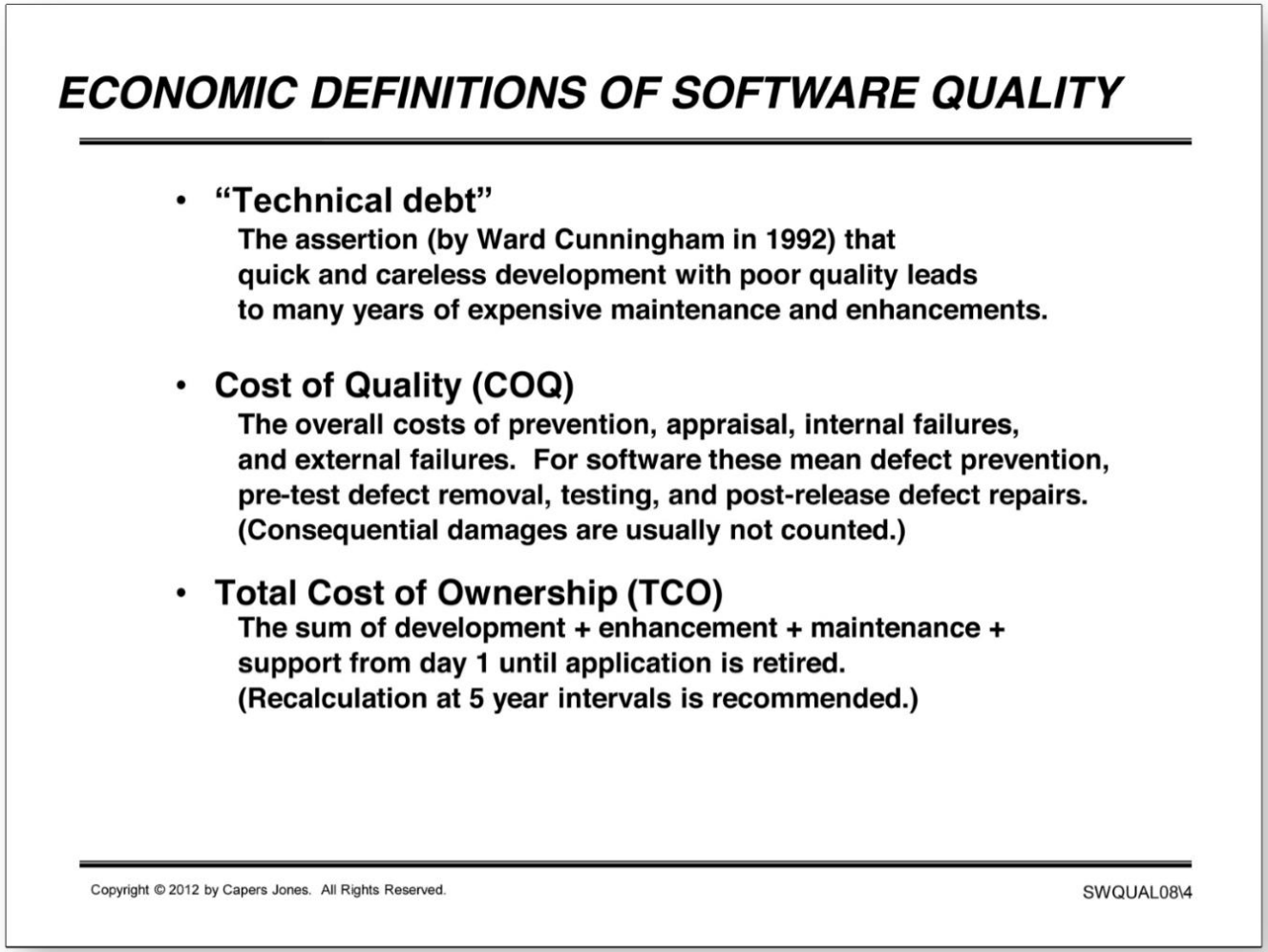


FUNDAMENTAL SOFTWARE QUALITY METRICS (cont.)

- **Standard Cost of Quality**
  - Prevention
  - Appraisal
  - Internal failures
  - External failures
- **Revised Software Cost of Quality**
  - Defect Prevention
  - Pre-Test Defect Removal (inspections, static analysis)
  - Testing Defect Removal
  - Post-Release Defect Removal
- **Error-Prone Module Effort**
  - Identification
  - Removal or redevelopment
  - repairs and rework

SWQUAL08\17

Terms you might encounter



ECONOMIC DEFINITIONS OF SOFTWARE QUALITY

- **"Technical debt"**
  The assertion (by Ward Cunningham in 1992) that quick and careless development with poor quality leads to many years of expensive maintenance and enhancements.

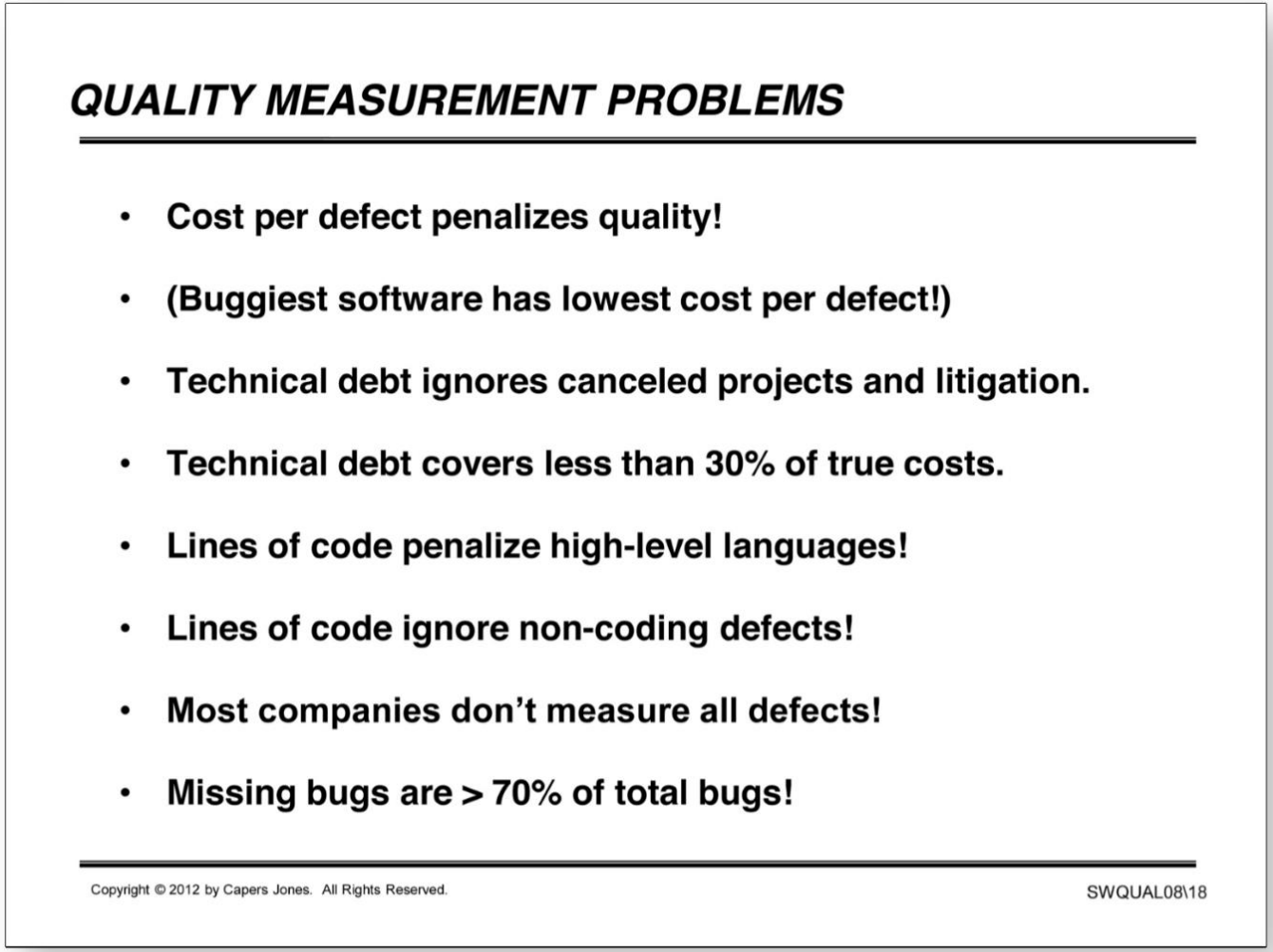
- **Cost of Quality (COQ)**
  The overall costs of prevention, appraisal, internal failures, and external failures. For software these mean defect prevention, pre-test defect removal, testing, and post-release defect repairs. (Consequential damages are usually not counted.)

- **Total Cost of Ownership (TCO)**
  The sum of development + enhancement + maintenance + support from day 1 until application is retired. (Recalculation at 5 year intervals is recommended.)

SWQUAL08\4

Quality Metrics in Practice

To build high-quality software, you need to know how to measure it



Comparing High and Low-Level Languages

Do not use lines of code …

LINES OF CODE HARM HIGH-LEVEL LANGUGES

	Case A JAVA	Case B C
KLOC	50	125
Function points	1,000	1,000
Code defects found	500	1,250
Defects per KLOC	10.00	10.00
Defects per FP	0.5	1.25
Defect repairs	$70,000	$175,000
$ per KLOC	$1,400	$1,400
$ per Defect	$140	$140
$ per Function Point	$70	$175
$ cost savings	$105,000	$0.00

SWQUAL08\9

Benchmark Data on Post-Install Defects

More users will find your bugs more quickly

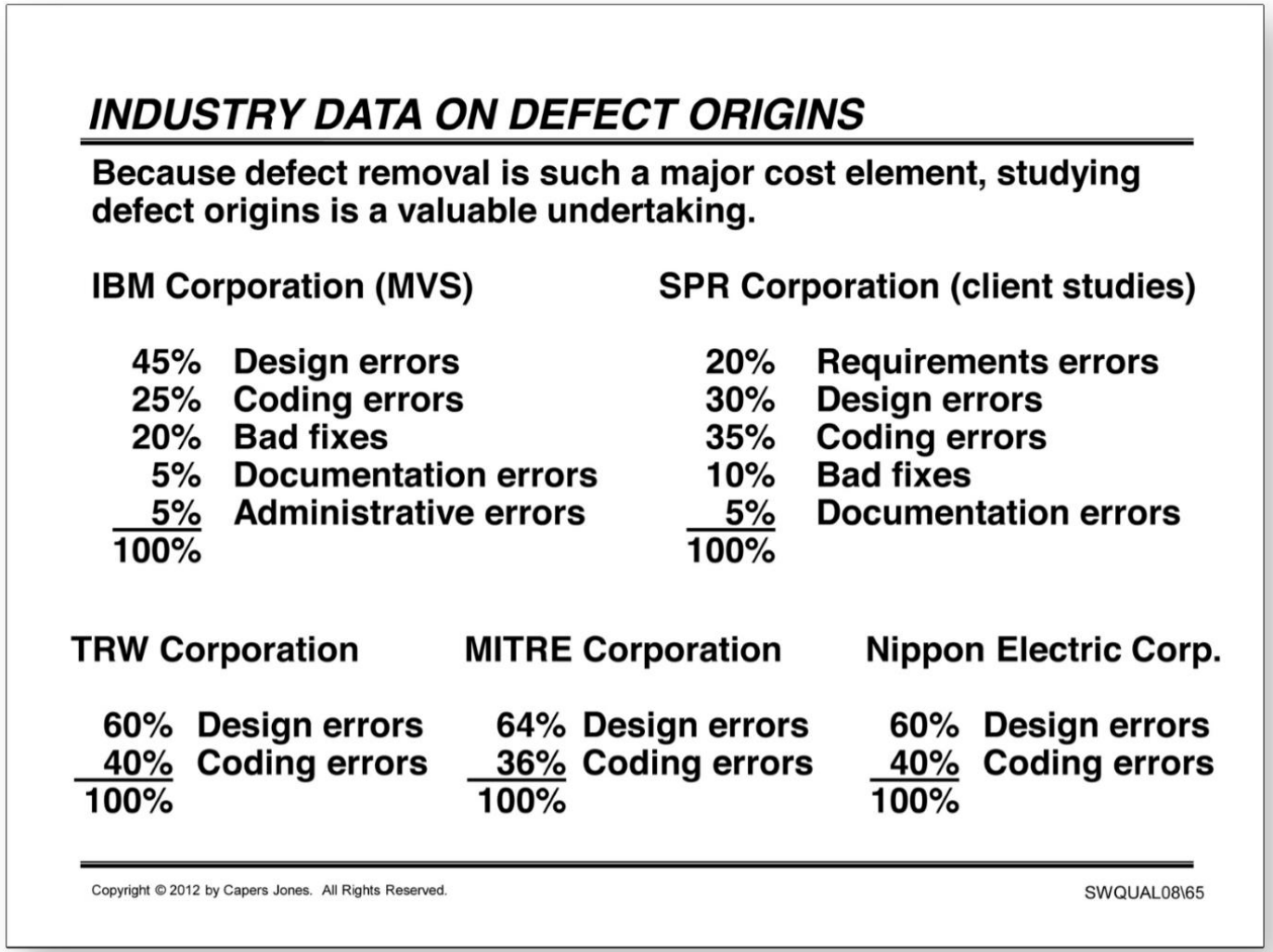
DEFECT REPORTS IN FIRST YEAR OF USAGE

Function Points	10	100	1000	10,000	100,000
Users					
1	55%	27%	12%	3%	1%
10	65%	35%	17%	7%%	3%
100	75%	42%	20%	10%	7%
1000	85%	50%	*27%*	12%	10%
10,000	95%	75%	35%	20%	12%
100,000	99%	87%	45%	35%	20%
1,000,000	100%	96%	77%	45%	32%
10,000,000	100%	100%	90%	65%	45%

SWQUAL08\41

Benchmark Data

Source: Jones (2013)



INDUSTRY DATA ON DEFECT ORIGINS

Because defect removal is such a major cost element, studying defect origins is a valuable undertaking.

IBM Corporation (MVS)

45%	Design errors
25%	Coding errors
20%	Bad fixes
5%	Documentation errors
5%	Administrative errors
100%	

SPR Corporation (client studies)

20%	Requirements errors
30%	Design errors
35%	Coding errors
10%	Bad fixes
5%	Documentation errors
100%	

TRW Corporation

60%	Design errors
40%	Coding errors
100%	

MITRE Corporation

64%	Design errors
36%	Coding errors
100%	

Nippon Electric Corp.

60%	Design errors
40%	Coding errors
100%	

SWQUAL08\65

Benchmark Data

Sorted by delivered defects

U.S. AVERAGES FOR SOFTWARE QUALITY

(Data expressed in terms of defects per function point)

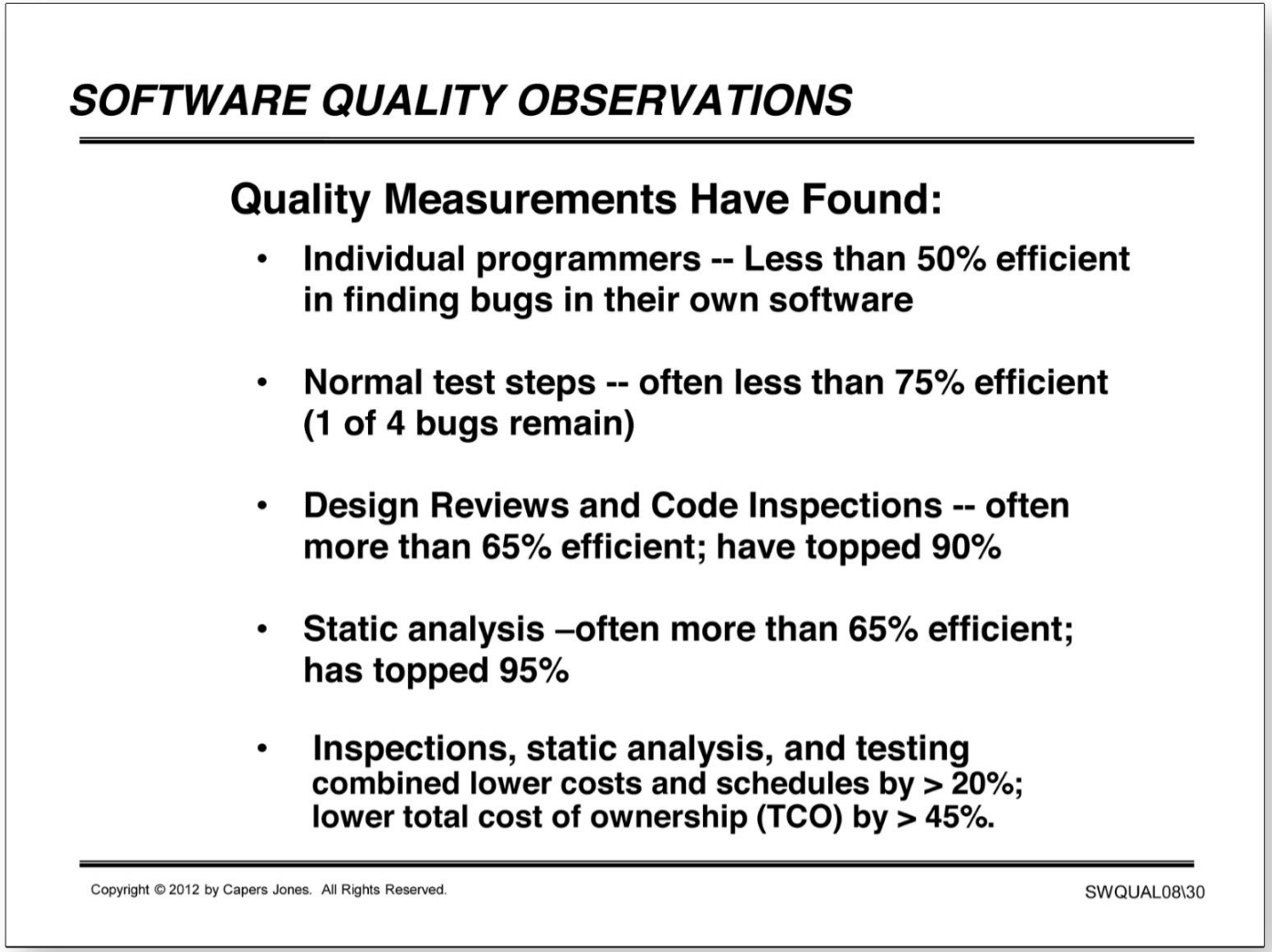
Defect Origins	Defect Potential	Removal Efficiency	Delivered Defects
Requirements	1.00	77%	0.23
Design	1.25	85%	0.19
Coding	1.75	95%	0.09
Documents	0.60	80%	0.12
Bad Fixes	0.40	70%	0.12
TOTAL	5.00	85%	0.75

(Function points show all defect sources - not just coding defects)
(Code defects = 35% of total defects)

SWQUAL08\23

Software Quality

Observations: Jones (2013)



SOFTWARE QUALITY OBSERVATIONS

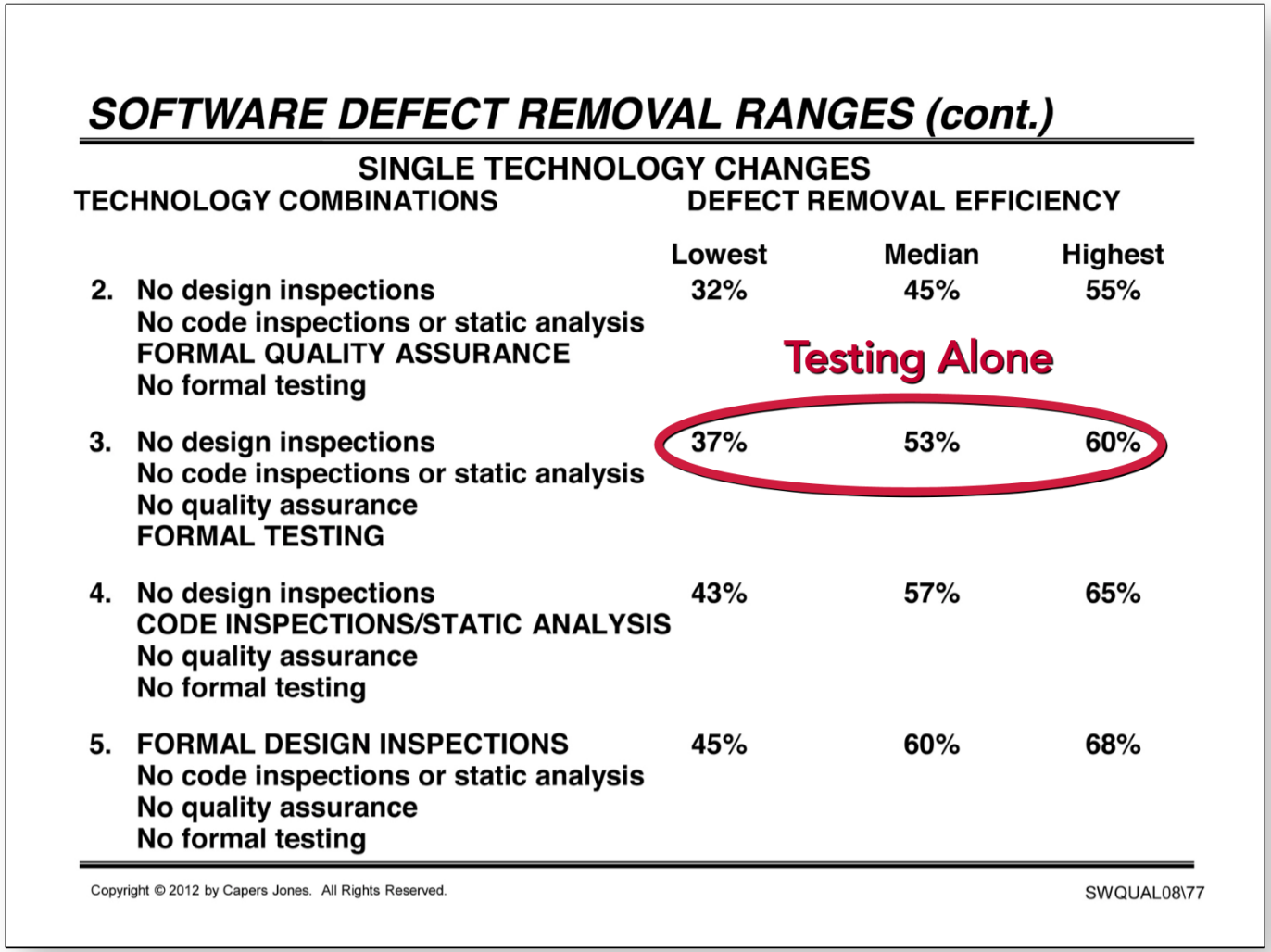
Quality Measurements Have Found:

- **Individual programmers -- Less than 50% efficient in finding bugs in their own software**
- **Normal test steps -- often less than 75% efficient (1 of 4 bugs remain)**
- **Design Reviews and Code Inspections -- often more than 65% efficient; have topped 90%**
- **Static analysis –often more than 65% efficient; has topped 95%**
- **Inspections, static analysis, and testing combined lower costs and schedules by > 20%; lower total cost of ownership (TCO) by > 45%.**

SWQUAL08\30

No Single Verification Technique is Enough

Benchmark Data: Jones (2013)



SOFTWARE DEFECT REMOVAL RANGES (cont.)

TECHNOLOGY COMBINATIONS	DEFECT REMOVAL EFFICIENCY		
	SINGLE TECHNOLOGY CHANGES		
	Lowest	Median	Highest
2. No design inspections No code inspections or static analysis FORMAL QUALITY ASSURANCE No formal testing	32%	45%	55%
3. No design inspections No code inspections or static analysis No quality assurance FORMAL TESTING	37%	53%	60%
4. No design inspections CODE INSPECTIONS/STATIC ANALYSIS No quality assurance No formal testing	43%	57%	65%
5. FORMAL DESIGN INSPECTIONS No code inspections or static analysis No quality assurance No formal testing	45%	60%	68%

Testing Alone

SWQUAL08\77

Combinations of Verification Techniques

Benchmark Data: Jones (2013)

SOFTWARE DEFECT REMOVAL RANGES (cont.)

THREE TECHNOLOGY CHANGES

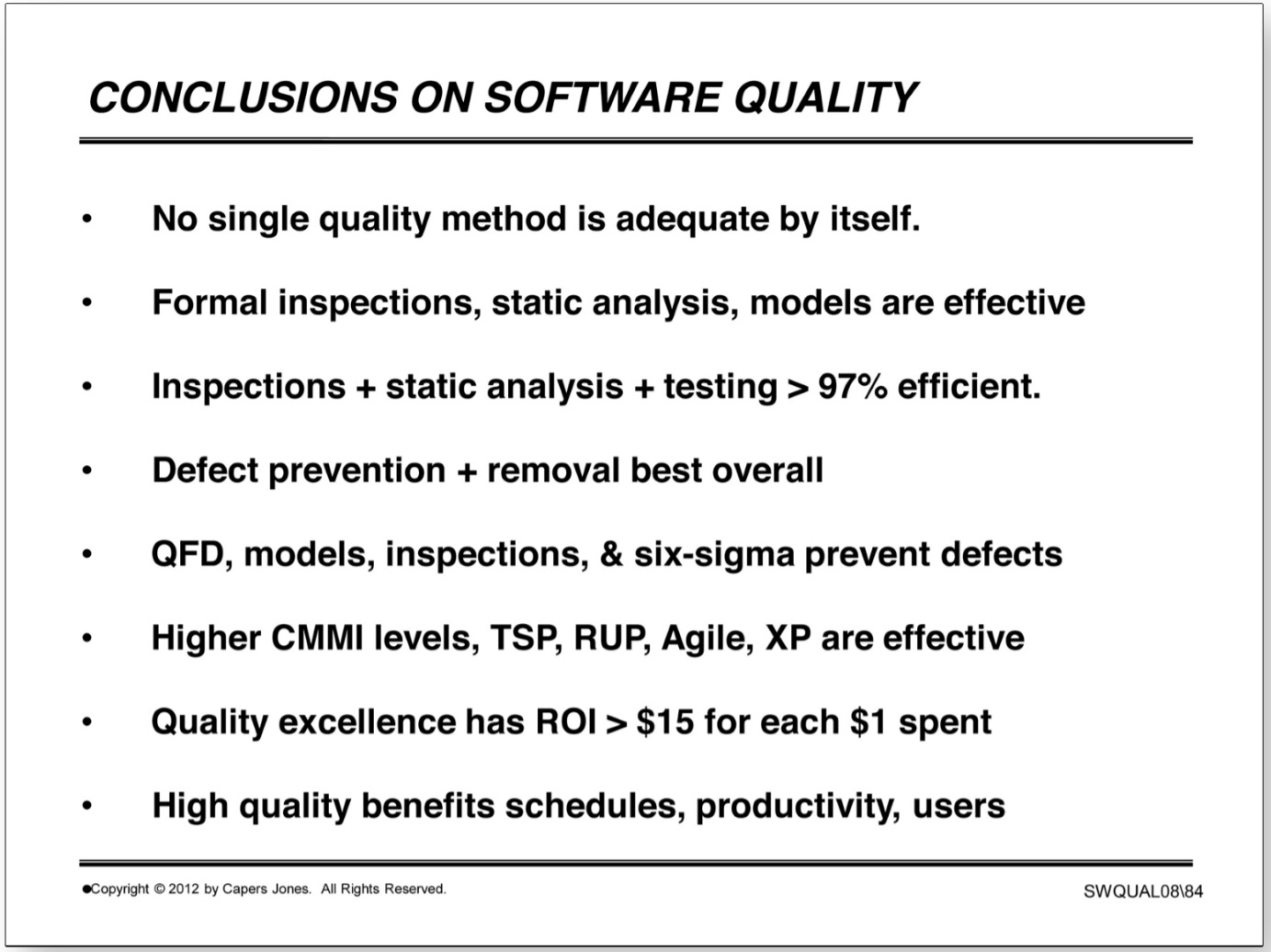
TECHNOLOGY COMBINATIONS	DEFECT REMOVAL EFFICIENCY Lowest	Median	Highest
12. No design inspections FORMAL CODE INSPECTIONS/STAT.AN. FORMAL QUALITY ASSURANCE FORMAL TESTING	75%	87%	93%
13. FORMAL DESIGN INSPECTIONS No code inspections or static analysis FORMAL QUALITY ASSURANCE FORMAL TESTING	77%	90%	95%
14. FORMAL DESIGN INSPECTIONS FORMAL CODE INSPECTIONS/STAT. AN. FORMAL QUALITY ASSURANCE No formal testing	83%	95%	97%
15. FORMAL DESIGN INSPECTIONS FORMAL CODE INSPECTIONS/STAT.AN. No quality assurance FORMAL TESTING	85%	97%	99%

Reviews, Static Analysis, Testing

SWQUAL08\80

Software Quality Conclusions

Jones (2013)



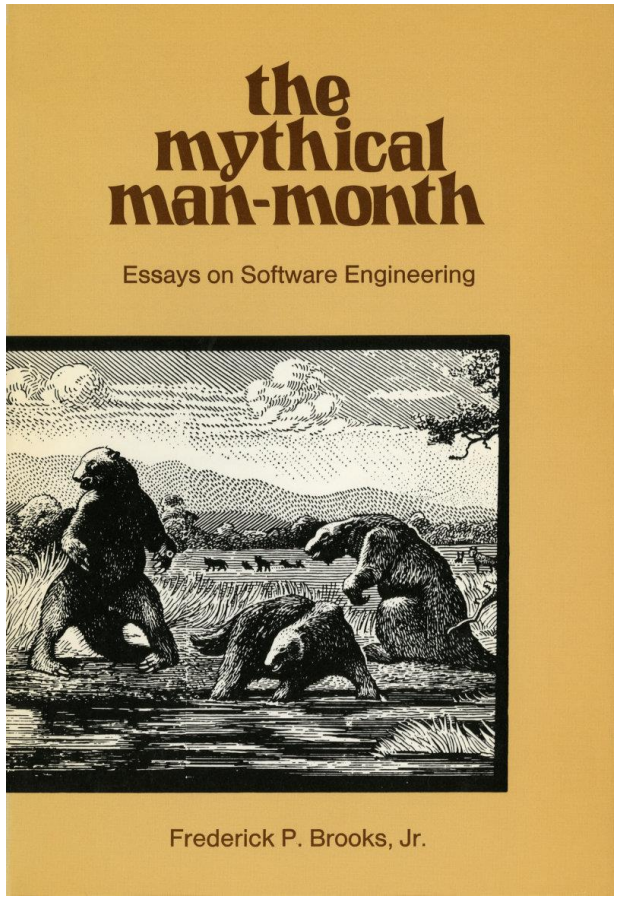
CONCLUSIONS ON SOFTWARE QUALITY

- **No single quality method is adequate by itself.**
- **Formal inspections, static analysis, models are effective**
- **Inspections + static analysis + testing > 97% efficient.**
- **Defect prevention + removal best overall**
- **QFD, models, inspections, & six-sigma prevent defects**
- **Higher CMMI levels, TSP, RUP, Agile, XP are effective**
- **Quality excellence has ROI > $15 for each $1 spent**
- **High quality benefits schedules, productivity, users**

SWQUAL08\84

Motivation for Static Analysis

- "Software products are among the most complex of man-made systems, and software by its very nature has intrinsic, essential properties (e.g., complexity, invisibility, and changeability) that are not easily addressed"
  - Brooks, F.P. The Mythical Man Month (1995)



Apple SSL Security Vulnerability (2014)

- **What happened**
  - SSL security vulnerability in Apple products announced in Feb 2014
  - Left users vulnerable to “man-in-the-middle” attacks
  - Potentially affects secure browsing, credit card info, etc.
- **Schedule of systems and security updates**

– iOS 6.x	6.1.5	since 9/19/12	fixed in 6.1.6	on 2/21/14
–	7.x – 7.0.5	9/18/13	7.0.6	2/21/14
– OS X	10.9 – 10.9.1	10/22/13	10.9.2	2/25/14

- **Acronyms:**
  - SSL: Secure Sockets Layer
  - TLS: Transport Layer Security

Apple SSL Security Vulnerability (2014)

- Extra `goto` in code for the handshake algorithm
  - Sequence duplicated six times!
  - As it appears in `SSLVerifySignedServerKeyExchange()`:

```
if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
    goto fail;
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
    goto fail;
```

What is Static Analysis?

- Analyzes code without executing it
- Doesn't require an understanding of the intent of the code
- Detects violations of reasonable programming practices
- Not a replacement for testing, but good at finding problems on untested paths
- May report many false positives
- There are many defects that static analysis cannot find (more can be detected in conjunction with dynamic analysis)
- A variety of checkers (Coverity) or types (Findbugs) are used to look for specific kinds of defects

What Static Analysis is Not

- It does not ensure code is correct or of good quality
- It is not a replacement for good design, code reviews (though it may augment reviews), or unit testing
- Does not find more sophisticated errors that can only be detected by running or simulate running of the code (e.g. dynamic analysis)

Static Analysis Tools for Java

- **FindBugs**
  - Analysis based on bytecode
  - Has a large number of patterns for potential defects
- **JLint**
  - Finds inconsistencies and synchronization problems in Java programs
- **PMD**
  - Programming Mistake Detector
  - Focuses on inefficient code, e.g. over-complex expressions
- **ESC/Java and ESC/Java2**
  - ESC: Extended Static Checker
  - Tools use theorem provers to improve diagnostics

Which bug is worse?

```
int x = 2, y = 3;
if (x == y)
       if (y == 3)
              x = 3;
else
       x =4;
```

```
String s = new (“hello”);
s = null;
System.out.println(s.length());
```

Which bug is worse?

```
int x = 2, y = 3;
if (x == y)
       if (y == 3)
              x = 3;
else
       x =4;
```

Detected by PMD
(when using certain rule sets)

Not detected during testing

```
String s = new (“hello”);
s = null;
System.out.println(s.length());
```

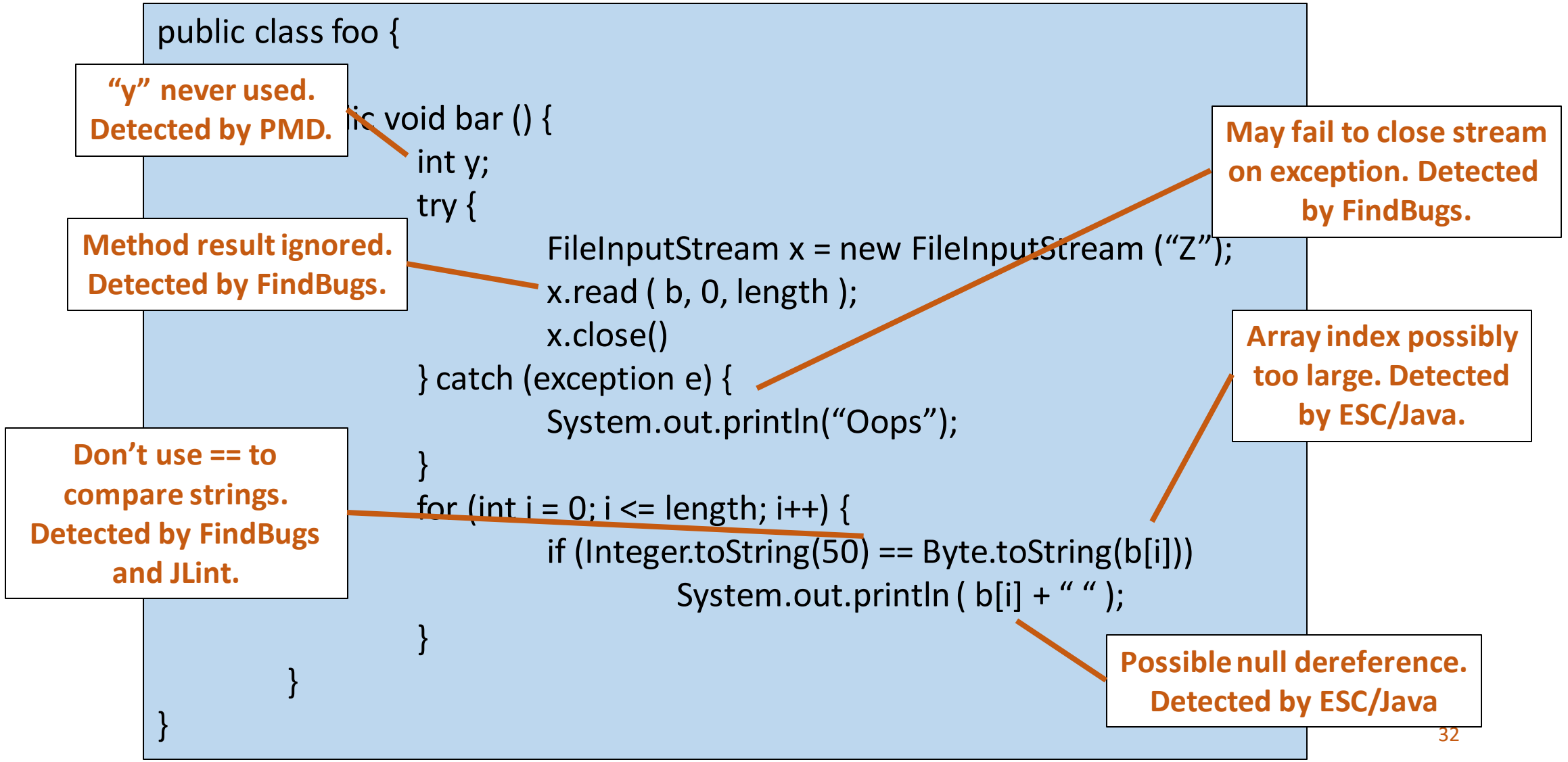
Detected by
FindBugs, JLint, ESC/Java

Also detected during testing

Different Tools Find Different Bugs

```
public class foo {
        …
        public void bar () {
                int y;
                try {
                        FileInputStream x = new FileInputStream (“Z”);
                        x.read ( b, 0, length );
                        c.close()
                } catch (exception e) {
                        System.out.println(“Oops”);
                }
                for (int i = j; I <= length; i++) {
                        if (Integer.toString(50) == Byte.toString(b[i]))
                                System.out.println ( b[i] + “ “ );
                }
        }
}
```

Different Tools Find Different Bugs



Limitations of Static Analysis

- **Potentially, large numbers of false positives**
  - Tool reports large number of things that aren't bugs
  - Programmer must manually review the list and decide
  - Sometimes too many warnings to sort through
- **False negatives**
  - Types of bugs the tool won't report
  - (increased risk if we filter results to remove false positives?)
- **Harmless bugs**
  - Many of the bugs will be low priority problems
  - Cost / benefit: is it worth fixing these bugs?

Coverity Checkers

Coverity supports more than 135 checkers, each for a particular defect

Category	#	Types of Defects Detected
C Headers	1	Inclusion of unnecessary header file
C/C++	41	Memory leaks, Stack corruptions, Buffer overruns, Use after free, Uninitialized variables, Pointer memory allocation defects, Unchecked dereferences of NULL return values, Dereferences of NULL pointers, Misuses of negative integers, inconsistencies in handling return values, return pointer to a local stack variable, bounds-checking
C++ only	18	Overriding virtual functions, Deleting an array, uses of STL iterators that are either invalid or past-the-end, function parameters too large, C++ exception thrown and never caught
C#	16	Similar categories to C/C++
JAVA	44	Similar to C/C++ checkers, plus errors detected by FINDBUGs, and thread issues.
Concurrency	5	Double and missing locks, Incorrect lock ordering, Blocking functions causing locks to be held too long
Security	14	Improper validation of tainted strings, Strings not null-terminated, Failure to size-check strings, Failure to bounds-check strings, String overflows, Buffer overflows, Time-of-check-time-of-use errors, Use of insecure temporary file creation routines
Rules	4	Non-conformance with a project's coding standards
Warnings	8	Complier parse warnings that are detected by the Coverity compiler

Example: Use of Static Analysis

Identifying Critical Impact Defects

- **Criteria for classifying a defect as a Critical Impact Defect**
  - **The error is in the critical path of execution**
    - As determined by dynamic analysis or execution traces
  - **If encountered in execution the result would be severe**
  - **The error is important as understood by the uniqueness of the checker**

Example: Use of Static Analysis

Identifying Critical Impact Defects

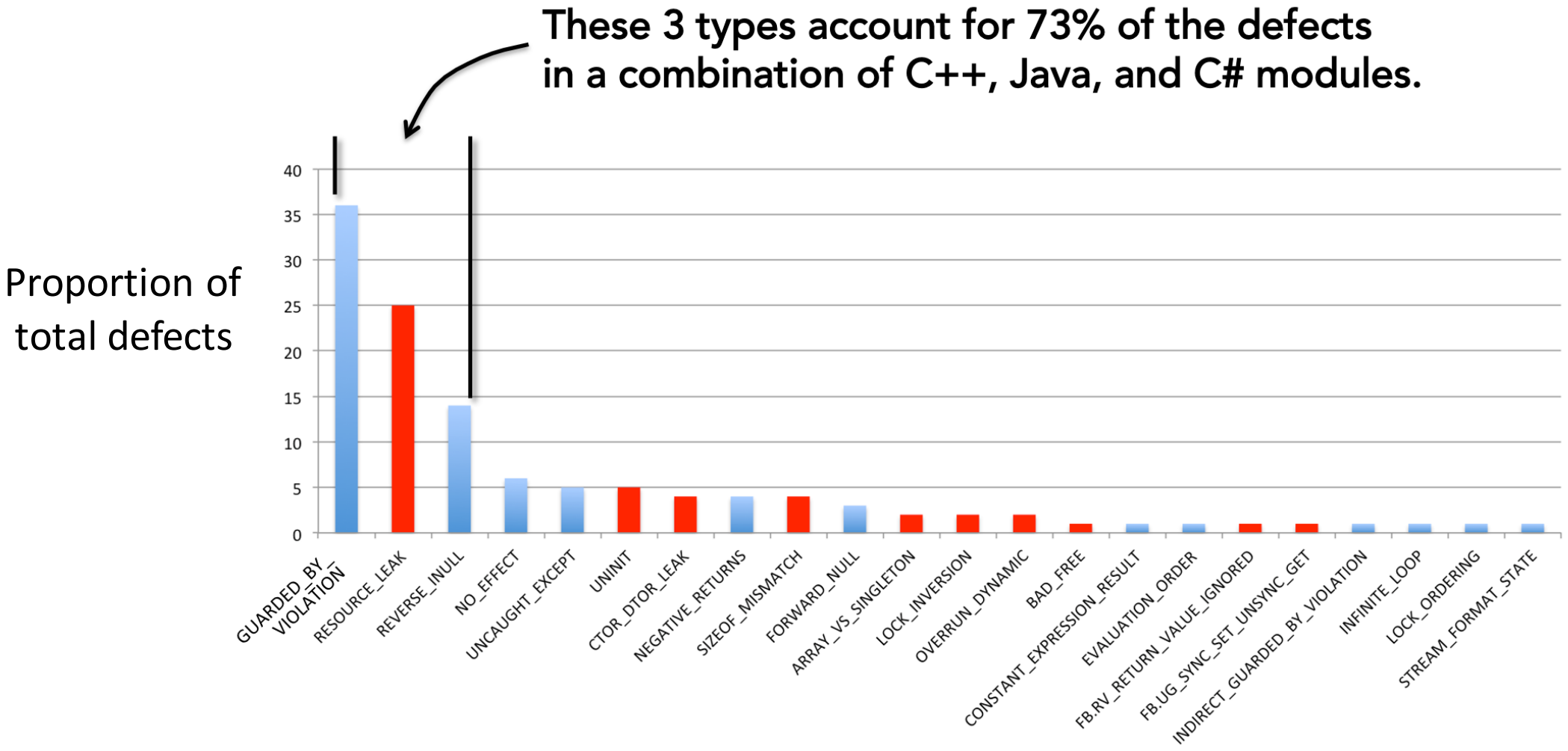
- **Criteria for classifying a defect as a Critical Impact Defect**
  - **The error is in the critical path of execution**
  - **If encountered in execution the result would be severe**
    - System Crash; e.g. overwrite data; object used before defined or initialized
    - Hanging or large delays due to infinite loops
    - Indeterminate behavior due to race conditions
    - Poor performance or crash due to resource leak
  - **The error is important as understood by the uniqueness of the checker**

Example: Use of Static Analysis

Identifying Critical Impact Defects

- **Criteria for classifying a defect as a Critical Impact Defect**
  - **The error is in the critical path of execution**
  - **If encountered in execution the result would be severe**
  - **The error is important as understood by the uniqueness of the checker**
    - Some checkers don't report defects very often, but when they do it is usually important (e.g., in C++, arithmetic on a pointer to a singleton)

Defects in Coverity



High-impact defects are in RED, medium-impact in BLUE

Coverity Checker: INFINITE_LOOP

- **Description**
  - INFINITE_LOOP finds instances of loops that never terminate
  - Checks whether the control variables of a loop are properly updated with respect to the relational operator in the variable's loop conditions
- **C/C++ Examples**
  - `for (j = 0; j < backup; j++) { … }`
  -
```
while (true) {
   … if (x == 55) break; …     // x never updated
}
```

Coverity Checker: RESOURCE_LEAK

Checks variables that go out of scope while "owning" a resource

- **File descriptor and socket leaks**
  - Can cause crashes, denial of service, inability to open more files
  - OS limits the number of FD's and sockets for each process
  - If leaked, cannot be reclaimed until process ends
- **Memory leaks**
  - Often occur on error paths where one forgets to free memory after encountering an error condition
  - Even small leaks are problematic for long-running processes
  - Can cause security issues (denial-of-service using a leaky program)

Coverity Checker: RESOURCE_LEAK

Checks variables that go out of scope while "owning" a resource

- **C++ Memory Leak Example**

```
int leak_example(int c) {
   void *p = malloc(10);
   if (c) return -1;
   /* … */
   free(p);
   return 0;
}
```

Coverity Checker: REVERSE_INULL

Checks for null checks after dereferences (C++, C#, Java)

- **Why is it important?**
  - A process may crash because of access to non-existent memory
  - Even if an exception is caught, there is no way to fix the situation
- **Notes on false positives**
  - REVERSE_INULL can report false positives if it determines that a pointer is null when that pointer can never be null
  - If the analysis incorrectly reports that a pointer is checked against null or that there is a defect due to a non-feasible path, you can suppress the event with a code annotation.

Coverity Checker: REVERSE_INULL

Checks for null checks after dereferences (C++, C#, Java)

- **C++ Example**

```
void foo(struct buf_t *request_buf) {
        …
        *request_buf = some_function()
        if (request_buf == NULL) return
        …
}
```

- **What if some_function() returns NULL?**

Testing Overview

"Microsoft, in terms of this quality stuff – we have as many testers as we have developers. And testers spend all their time testing, and developers spend half their time testing."

"We're more of a testing, a quality software organization than we're a software organization."

- Bill Gates, Information Week Interview, May 2002

Testing Overview

“… program testing can be used very effectively to show the presence of bugs but never to show their absence.”

- E. W. Dijkstra in EWD303

What is Testing and What is it Not?

- **Testing is a process for finding semantic or logical errors as a result of executing a program**
  - A run-time process, not a compile-time process
- **Testing is not aimed at finding syntactic errors**
  - The code is expected to be working code
  - Static checking, prior to run time, is separate
- **Testing does not improve software quality**
  - Test results are a measure of quality, but tests don't improve quality
- **Testing can reveal the presence of errors, not their absence**
  - If your tests don't find errors, then get more effective tests

Software Testing in Practice

- **Testing amounts to 40% to 80% of total development costs***
  - 40% for information systems
  - 80% for real time embedded systems
- **Testing receives the least attention and often not enough time and resources**
  - Testers are often forced to abandon testing efforts because changes have been made
- **Testing is (usually) at the end of the development cycle**
  - Often rushed because other activities have been late

*Source: David Weiss, Iowa State

Appropriate Testing

Imagine you are testing a program that performs some calculations

- **Three different contexts**
  - It is used occasionally as part of a computer game
  - It is part of an early prototype of a commercial accounting package
  - It is part of a controller for a medical device
- **For each context**
  - What is your mission?
  - How aggressively would you hunt for bugs?
  - How much will you worry about …
    - Performance, precision, user interface, security and data protection …?
  - How extensively will you document your tests?
  - What other information will you provide to the project?

Good tests have …

Desirable properties of tests

- **Power:** if there is a problem, the tests will find it
- **Validity:** the problems found are genuine problems
- **Value:** the tests reveal things that clients want to know
- **Credibility:** the test is a likely operational scenario
- **Non-redundancy:** each test provides new information
- **Repeatability:** easy and inexpensive to re-run
- **Maintainability:** test can be revised as the product is revised
- **Coverage:** exercises the product in a way not already tested for
- **Ease of Evaluation:** results are easy to interpret
- **Diagnostic Power:** help pinpoint the cause of the problems
- **Accountability:** you can explain, justify, and prove you ran it
- **Low Cost:** reasonable time and effort to develop and time to execute
- **Low Opportunity Cost:** better use of your time than other things you could be doing

Testing Strategies

Only a partial list

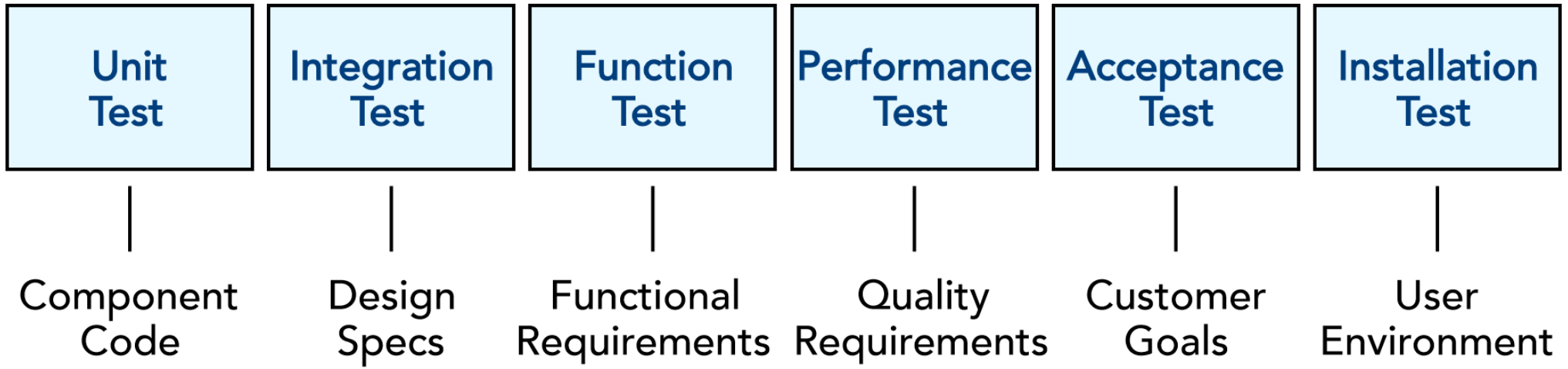
- **Black Box or Functional Testing**
  - Based on knowledge of the functionality, but not the implementation
  - For modules, based on knowledge of the interface (API), available methods, but not of the code that implements them
  - Base test cases on knowledge of how users will use the system
  - Can be performed independent of developers
- **White Box or Structural Testing**
  - Based on knowledge of the code
  - For modules, based on knowledge of their inner workings
  - Ensure coverage of statements, branches, conditions, …
- **Regression Tests**
  - Based on previously run tests
  - Repeat all tests, every time the system is modified

Black Box Testing: Limitations

- **Requirements may not be complete and accurate**
  - May not describe all behaviors of the system
- **Requirements may not have sufficient details for testing**
  - Design decisions may be left to the implementation
- **The system may have unintended functionality**

- **Conclusion: supplement Black Box (Functional) Testing with White Box (Structural) Testing**

Types of Testing
and what each tests …



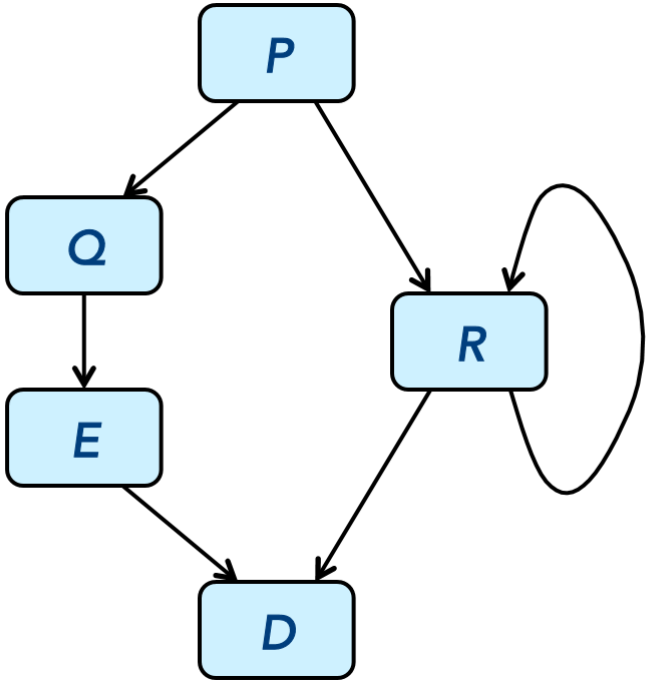
*** Not a comprehensive list**

Integration Testing

- **Follows Unit Testing**
  - Each unit is tested separately to check if it meets its specification
- **Integration Testing**
  - Units are tested together to check whether they work together
- **Integration Testing Challenges**
  - Problems of scale
  - Tends to reveal specification rather than integration errors

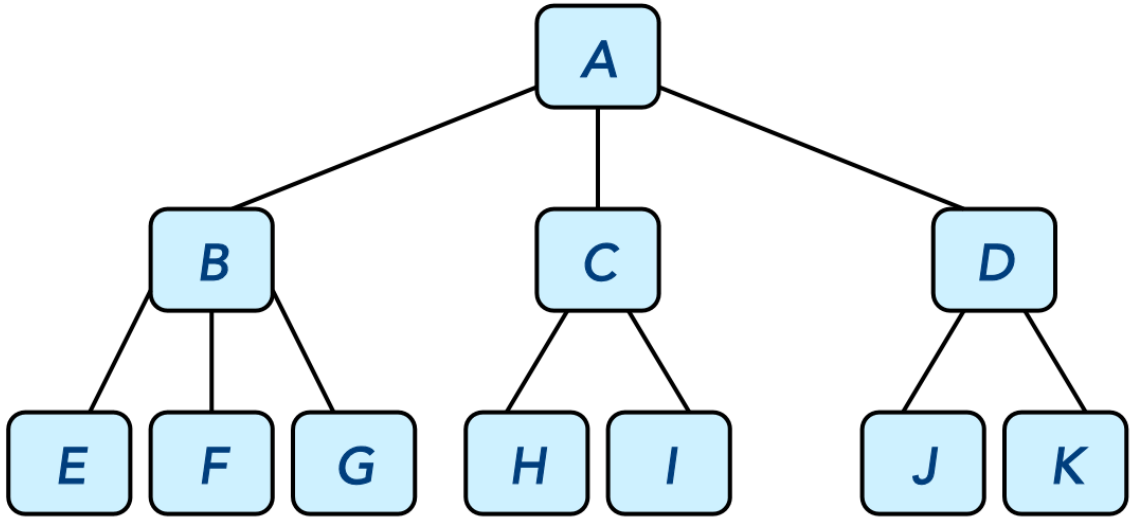
Integration Testing: Bottom Up

- **For this dependency graph, bottom up test order is:**
  1. **D**
  2. **E and R**
  3. **Q**
  4. **P**



Integration Testing: Top Down

- **For this module hierarchy, the top-down test order is:**
  1. **Test A with stubs for B, C, and D**
  2. **Test A + B + C + D with stubs for E, …, K**
  3. **Test the whole system**



Test Coverage

- **Definition: the extent to which a given verification activity has satisfied its objectives**
- **Email from Gilman Stevens of Avaya (2014):**

  "What we found was that we only tested 1/10 of 1% of what they use. And of course it was a bad release in the customer eyes. We changed the automation tools and manual testing to test what the customer used, about 1% of the code at the application layer ... and the next release was a fantastic release in the customers eyes. Through this tool we also learned that the customer base rarely (aka almost never) used new features it was all about everything that they currently use still working in the new release."

  "So regression testing is much more important than the new feature testing in the customer eyes.."

Coverage: Functional

```
int maximum equal (list a) {
/* requires: a is a list of integers
   effects:  returns the maximum element in the list
*/
      ...
}
```

- **Naïve test strategy**
  - Generate lots of tests and test for maximum
- **But**
  - That might not test non-normal cases; e.g. empty list, non-integers
- **So**
  - Need enough tests to cover every kind of input

Coverage: Functional

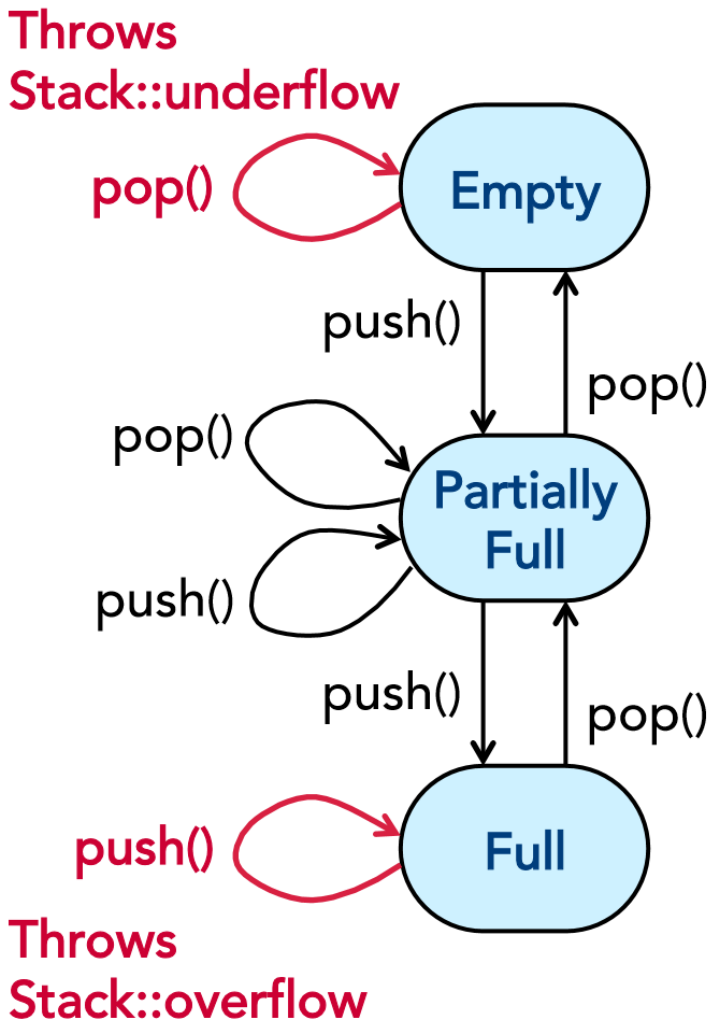
Testing for the maximum element in a list

Input	Output	Correct
3 16 4 32 9	32	Yes
9 32 4 16 3	32	Yes
22 32 59 17 88 1	88	Yes
1 88 17 59 32 22	88	Yes
1 3 5 7 9 1 3 5 7	9	Yes
7 5 3 1 9 7 5 3 1	9	Yes

- **Is this a good test set?**

Coverage: Behavioral

- **Naïve test strategy**
  - Push and pop items off the stack
- **But**
  - That might miss full and empty stack exceptions
- **So**
  - Need enough tests to exercise every event that can occur in each state the program can be in



Coverage: Structural

```
boolean equal (int x, int y) {
/* effects: returns true if x = y, false otherwise */
      if( x == y )
            return TRUE;
      else
            return FALSE;
}
```

- **Naïve test strategy**
  - Pick random values for x and y and test for equality
- **But**
  - That might not test the first branch of the if statement
- **So**
  - Need enough tests to cover every branch of the code

Structural Basis Testing

The minimal set of tests to cover every branch

- **How many tests?**
  - Start with 1 for the straight path
  - Add 1 for each of these keywords: if, while, repeat, for, and, or
  - Add 1 for each branch of a case statement
- **Example**
  - Count of test cases: 1+3 = 4
    (3 for each of the if key words)
  - Now choose the cases to exercise the paths
    - x = 3, y = 2, z = 1
    - X = 3, y = 2, z = 4
    - X = 2, y = 3, z = 2
    - X = 2, y = 3, z = 4

```
int midval(int x, y, z) {
/* effects:  returns the median
   value of the three inputs
*/
   if (x > y) {
      if (x > z) return x
      else return z
   } else {
      if (y > z ) return y
      else return z
   }
}
```

Boundary Checking

- **Boundary Analysis**
  - Every boundary needs 3 tests
    - Each side of the boundary
    - On the boundary
- **Example**
  - `if (x < MAX) { … }`
  - Sample test cases: x = MAX – 1, x = MAX, x = MAX + 1

Dataflow Testing

- **Dataflow Concepts**
  - Defined — The value of a variable is defined
  - Used — A value is used
  - Killed — A variable definition is overridden or space is released
  - Entered — Working copy created on entry to a method
  - Exited — Working copy released on exit from a method
- **Normal Life**
  - Defined once, used multiple times

Dataflow Testing

- **Potential Defects**
  - Def-Def Variable redefined
  - Def-Kill Defined but not used
  - Def-Exit Defined but not used
  - Entry-Use Variable used before it is defined
  - Kill-Use Used after it has been destroyed
  - …

Testing All Def-Use Paths

The minimal set of tests to cover all def-use paths

- **How many tests?**
  - A test for each path from each def to each use of the variable
- **Example**
  - Structural Basis Testing:

    2 test cases are enough
    - Cond1 = true, cond2 = true
    - Cond1 = false, cond2 = false
  - Def-Use Testing:

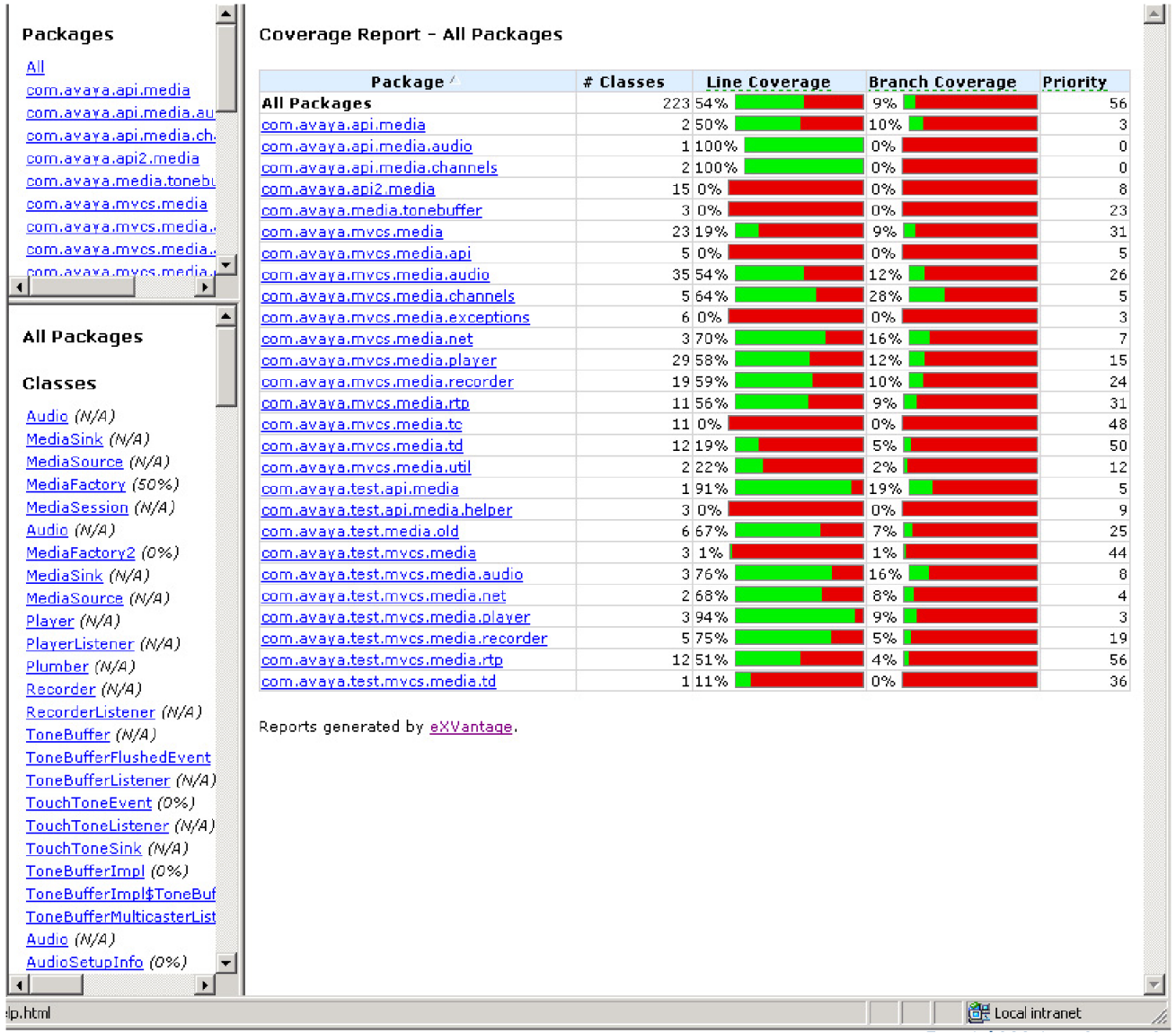
    Need 4 test cases
    - One for each Def-Use combination

```
if (cond1) {
    x = a;          Def 1
}
else {
    x = b;          Def 2
}
if (cond2) {
    y = x + 1;      Use 1
}
else {
    y = x - 1;      Use 2
}
```

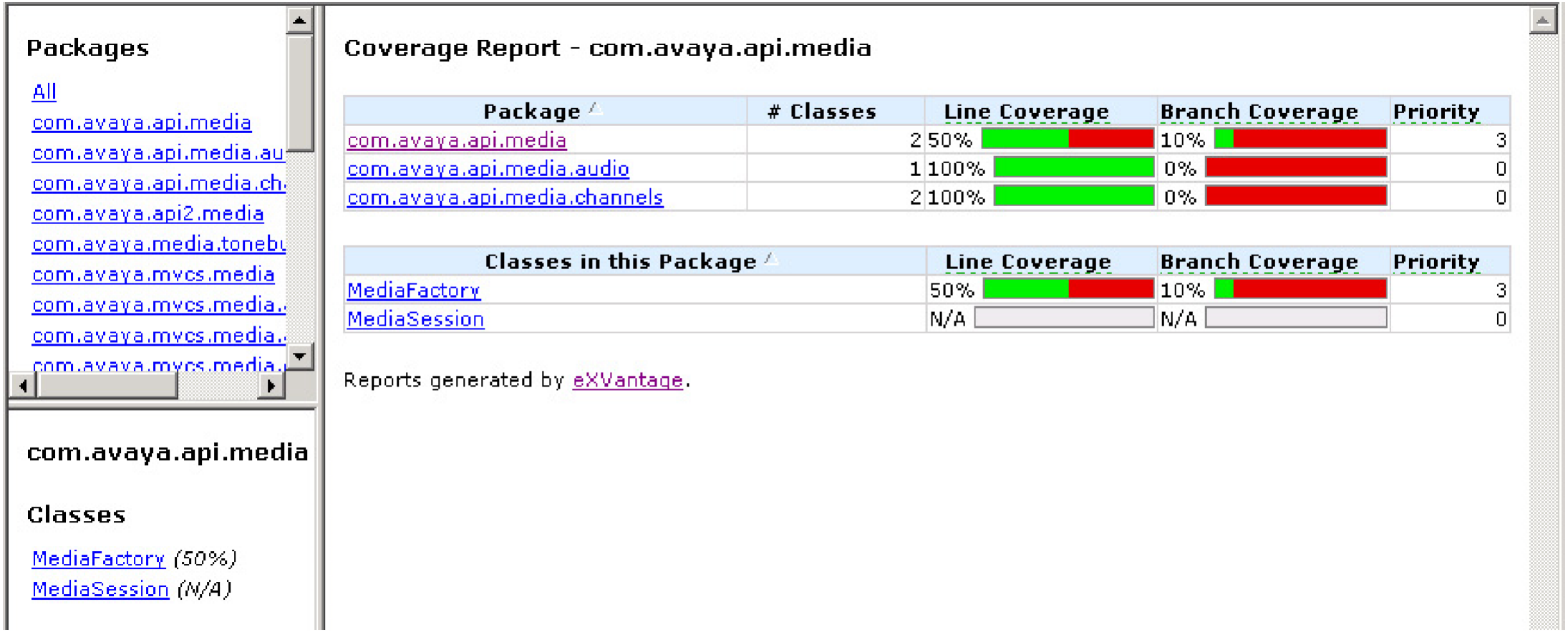
Code Coverage Tools

- **There exists a wide range of open source and proprietary tools for measuring and reporting test coverage**
  - Java: JCov, JaCoCo, EMMA
  - C/C++: COVTOOL, BullseyeCoverage, CppUnit
  - C#: Visual Studio, NCover, OpenCover
  - Python: coverage.py
  - …

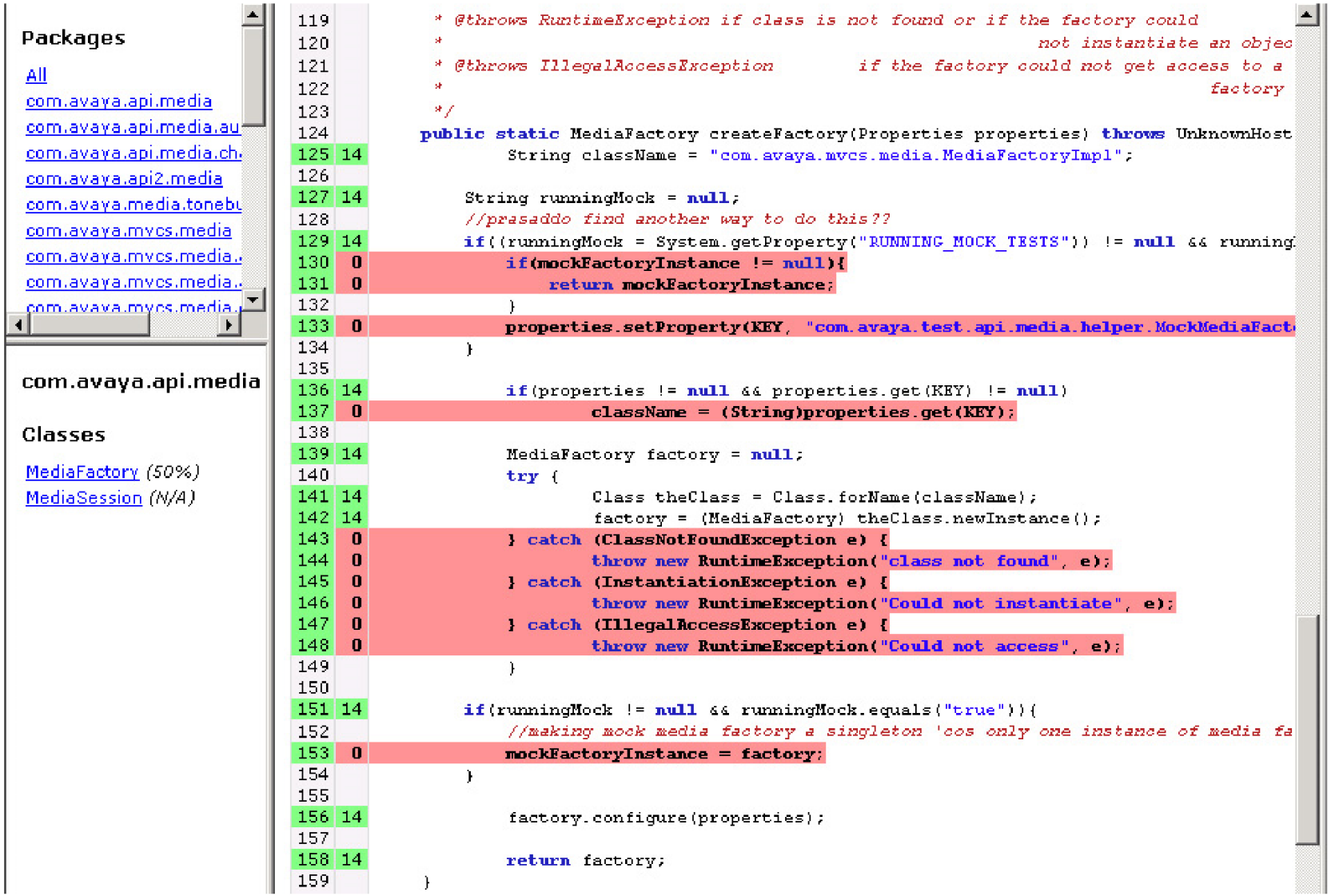
Coverage Reports: Packages



Coverage Reports: Classes



Coverage Reports: Source Code



MC/DC: Modified Condition / Decision Coverage

- **Advantages**
  - Shown to uncover important errors not detected otherwise
  - Linear growth in the number of tests required
  - Ensures coverage of the code
- **Mandated by the FAA (Federal Aviation Administration)**
  - Complex Boolean expressions are common in avionics
  - Real example: 2,262 decisions with 2 conditions, …, 219 with 6 – 10
- **Expensive; e.g. Boeing 777 (1st commercial fly-by-wire plane)**
  - Approx. 4M lines of code; 2.5M new; 70% Ada, rest C or assembly
  - Total cost of aircraft development: $5.5B
  - Cost of testing to MC/DC criteria: $1.5B

Condition Coverage

- **Each condition in a decision must take on all possible outcomes at least once**
- **Consider (a|b)**

	a	b
2	T	F
3	F	T

  - **Test cases 2 and 3 test both a and b**

Condition Coverage

- **Each condition in a decision must take on all possible outcomes at least once**
- **Consider (a|b)**

	a	b	(a|b)
2	T	F	T
3	F	T	T

  - **Test cases 2 and 3 test both a and b**
  - **However, the effect of (a|b) is not fully tested by these test cases**

Decision Coverage

- **Requires two test cases:**
  - **One for each of the true and false outcomes of the decision**
- **But, consider again (a|b)**

	a	b	(a|b)
2	T	F	T
4	F	F	F

  - **Test cases 2 and 4 cover both true and false outcomes for (a|b)**
  - **However, the effect of b is not tested by these test cases**

Modified Condition / Decision Coverage

- **Requires enough test cases to ensure:**
  - Each entry and exit point is invoked
  - Each decision takes every possible outcome
  - Each condition in a decision takes every possible outcome
  - Each condition in a decision is shown to independently affect the outcome of the decision
- **Consider again (a|b)**
  - Need test case 4 to test false outcome for (a|b)
  - Holding **a** fixed at F, flipping **b** affects the decision
  - Holding **b** fixed at F, flipping **a** affects the decision

	a	b	(a|b)
2	T	F	T
3	F	T	T
4	F	F	F

Modified Condition / Decision Coverage

Building Blocks: Boolean Operators

	a	b	a & b
1	T	T	T
2	T	F	F
3	F	T	F
4	F	F	F

Modified Condition / Decision Coverage

Building Blocks: Boolean Operators

	a	b	a & b
1	T	T	T
2	T	F	F
3	F	T	F
4	F	F	F

- **Tests for outcome T**
  - Test 1
- **Tests for outcome F**
  - Tests 2 and 3
  - Don’t need Test 4

Modified Condition / Decision Coverage

Building Blocks: Boolean Operators

	a	b	a | b
1	T	T	T
2	T	F	T
3	F	T	T
4	F	F	F

Modified Condition / Decision Coverage

Building Blocks: Boolean Operators

	a	b	a | b
1	T	T	T
2	T	F	T
3	F	T	T
4	F	F	F

- **Tests for outcome F**
  - Test 4
- **Tests for outcome T**
  - Tests 2 and 3
  - Don’t need Test 1

Modified Condition / Decision Coverage

Building Blocks: Boolean Operators

	a	b	a & b
1	T	T	T
2	T	F	T
3	F	T	T
4	F	F	F

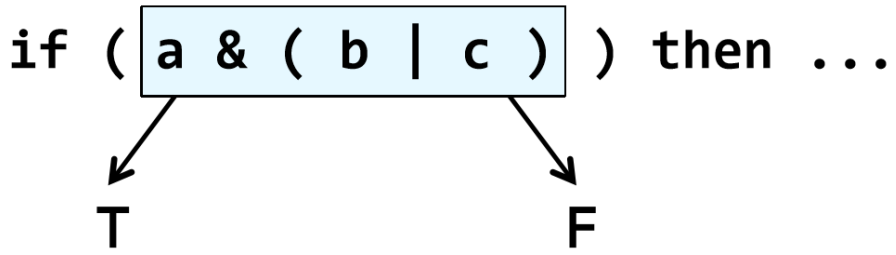
- **Tests for outcome T**
  - Test 1
- **Tests for outcome F**
  - Tests 2 and 3
  - Don’t need Test 4

	a	b	a | b
1	T	T	T
2	T	F	F
3	F	T	F
4	F	F	F

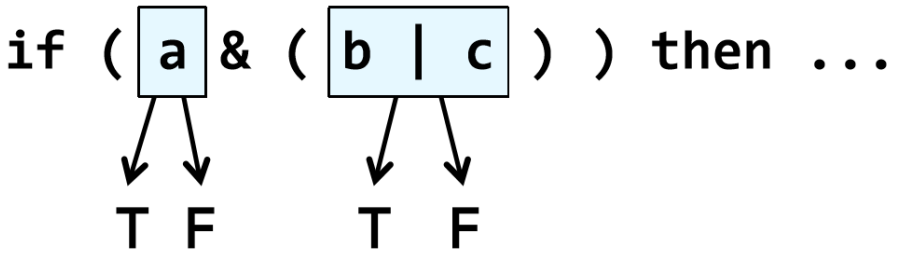
- **Tests for outcome F**
  - Test 4
- **Tests for outcome T**
  - Tests 2 and 3
  - Don’t need Test 1

Complex Decisions

- Decision Coverage



- But does each condition have an independent effect on the outcome?



Complex Decisions

- **Start with a truth table:**

	a	b	c	a & (b|c)
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

Complex Decisions

- **Truth table:**

	a	b	c	a & (b|c)
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

- If you flip **a**:
  - … in Test 1, you get Test 5
  - … in Test 2, you get Test 6
  - … in Test 3, you get Test 7

T T T
F T T

T T F
F T F

T F T
F F T

- **In tests 4 and 8, flipping a does not change the outcome of the decision**

Complex Decisions

- **Add a column for condition a. In the column, mark the tests where flipping a changes the outcome of the decision**

	a	b	c	a & (b|c)	a
1	T	T	T	T	5
2	T	T	F	T	6
3	T	F	T	T	7
4	T	F	F	F	
5	F	T	T	F	1
6	F	T	F	F	2
7	F	F	T	F	3
8	F	F	F	F	

- In Test 1 (T T T), flipping **a** gives Test 5 (F T T), and vice versa
- So put 5 in the column for Test 1 and put 1 in the column for Test 5
- Similarly, mark the pairs of tests 2,6 and 3,7

- **In tests 4 and 8, flipping a does not change the outcome of the decision**

Complex Decisions

- **Add a column for b and mark the tests where flipping b changes the outcome of the decision**

	a	b	c	a & (b|c)	a	b
1	T	T	T	T	5	
2	T	T	F	T	6	4
3	T	F	T	T	7	
4	T	F	F	F		2
5	F	T	T	F	1	
6	F	T	F	F	2	
7	F	F	T	F	3	
8	F	F	F	F		

- In Test 2, flipping **b** gives Test 4 and vice versa

- **In the other tests, flipping b does not change the decision**

Complex Decisions

- **Finally, add a column for c and mark the tests where flipping c changes the outcome of the decision**

	a	b	c	a & (b|c)	a	b	c
1	T	T	T	T	5		
2	T	T	F	T	6	4	
3	T	F	T	T	7		4
4	T	F	F	F		2	3
5	F	T	T	F	1		
6	F	T	F	F	2		
7	F	F	T	F	3		
8	F	F	F	F			

- In Test 3, flipping **c** gives Test 4 and vice versa

- **In the other tests, flipping c does not change the decision**

MC/DC with Complex Decisions

- **Need a test set where for every condition**
  - **There is a pair of tests where the condition is flipped and**
  - **The tests result in different outcomes for the decision**

	a	b	c	a & (b|c)	a	b	c
1	T	T	T	T	5		
2	T	T	F	T	6	4	
3	T	F	T	T	7		4
4	T	F	F	F		2	3
5	F	T	T	F	1		
6	F	T	F	F	2		
7	F	F	T	F	3		
8	F	F	F	F			

MC/DC with Complex Decisions

- **Need to include the following pairs of tests:**
  - **The pair 2,4 to test the effect of b on the outcome**
  - **The pair 3,4 to test the effect of c on the outcome**
  - **Thus, tests 2, 3, and 4 are required to test the effects of b and c**

	a	b	c	a & (b|c)	a	b	c
1	T	T	T	T	5		
2	T	T	F	T	6	4	
3	T	F	T	T	7		4
4	T	F	F	F		2	3
5	F	T	T	F	1		
6	F	T	F	F	2		
7	F	F	T	F	3		
8	F	F	F	F			

MC/DC with Complex Decisions

- **Need to include the following pairs of tests**
  - **The pair 2,4 to test the effect of b on the outcome**
  - **The pair 3,4 to test the effect of c on the outcome**
  - **Thus, tests 2, 3, and 4 are required to test the effects of b and c**

- For **a**, we could use any of pair 1,5, pair 2,6, or pair 3,7

	a	b	c	a & (b|c)	a	b	c
1	T	T	T	T	5		
2	T	T	F	T	6	4	
3	T	F	T	T	7		4
4	T	F	F	F		2	3
5	F	T	T	F	1		
6	F	T	F	F	2		
7	F	F	T	F	3		
8	F	F	F	F			

MC/DC with Complex Decisions

- **Need to include the following pairs of tests**
  - **The pair 2,4 to test the effect of b on the outcome**
  - **The pair 3,4 to test the effect of c on the outcome**
  - **Thus, tests 2, 3, and 4 are required to test the effects of b and c**

- For a, we could use any of pair 1,5, pair 2,6, or pair 3,7
- Since 2, 3, 4 are already required, we save a test by choosing 2,6 or 3,7

	a	b	c	a & (b|c)	a	b	c
1	T	T	T	T	5		
2	T	T	F	T	6	4	
3	T	F	T	T	7		4
4	T	F	F	F		2	3
5	F	T	T	F	1		
6	F	T	F	F	2		
7	F	F	T	F	3		
8	F	F	F	F			

MC/DC with Complex Decisions

- **Need to include the following pairs of tests**
  - **The pair 2,4 to test the effect of b on the outcome**
  - **The pair 3,4 to test the effect of c on the outcome**
  - **Thus, tests 2, 3, and 4 are required to test the effects of b and c**

- For a, we could use any of pair 1,5, pair 2,6, or pair 3,7
- Since 2, 3, 4 are already required, we save a test by choosing 2,6 or 3,7

	a	b	c	a & (b|c)	a	b	c
1	T	T	T	T	5		
2	T	T	F	T	6	4	
3	T	F	T	T	7		4
4	T	F	F	F		2	3
5	F	T	T	F	1		
6	F	T	F	F	2		
7	F	F	T	F	3		
8	F	F	F	F			

- Four tests are enough to test the effect of the three conditions on the outcome:
  - **Either 2, 3, 4, 6**
  - **Or 2, 3, 4, 7**
- The alternative, 1, 2, 3, 4, 5 uses five tests

Combinatorial Testing

Need to test configurations

An app must run on any combination of OS, browser, protocol, CPU, etc.

- **Configuration coverage is perhaps the most developed form of combinatorial testing**
  - **Common form is pairwise testing (varying pairs of parameters)**

Test	OS	Browser	Protocol	CPU	DBMS
1	XP	IE	IPv4	Intel	MySQL
2	XP	Firefox	IPv6	AMD	Sybase
3	XP	IE	IPv6	Intel	Oracle
4	OS X	Firefox	IPv4	AMD	MySQL
5	OS X	IE	IPv4	Intel	Sybase
6	OS X	Firefox	IPv4	Intel	Oracle
7	RHL	IE	IPv6	AMD	MySQL
8	RHL	Firefox	IPv4	Intel	Sybase
9	RHL	Firefox	IPv4	AMD	Oracle
10	OS X	Firefox	IPv6	AMD	Oracle

Interaction Failures

How does an interaction fault manifest in code?

```
if (pressure < 10) {
    //do something
    if (volume > 300) {
        faulty code! BOOM!
    } else {
        good code, no problem
    }
} else {
    // do something else
}
```

- A test that includes **pressure = 5** and **volume = 400** triggers the failure
- Together, **pressure < 10 & volume > 300** represent 2-way interaction

Interaction Failures

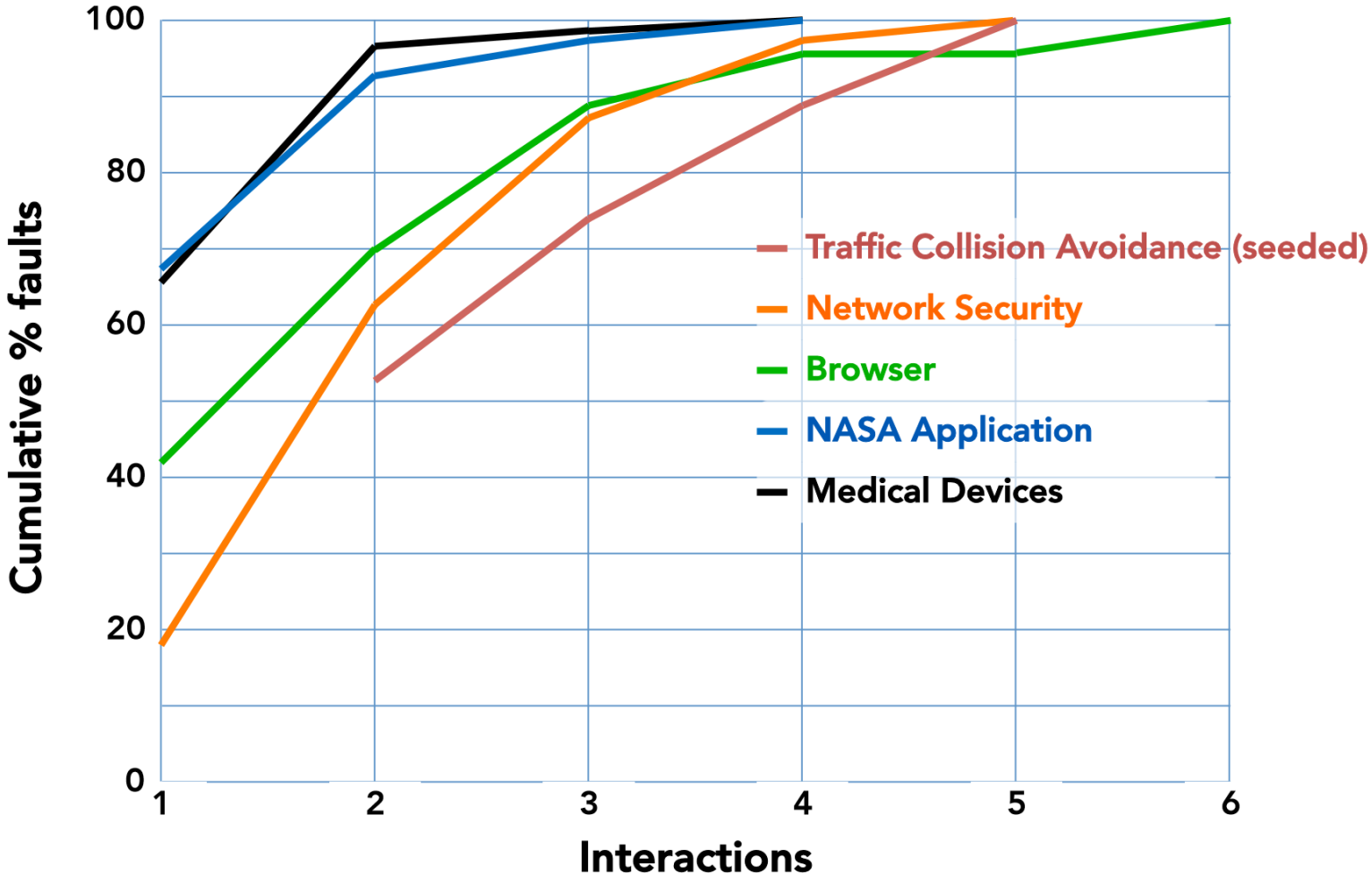
Branches from if, while, etc. can lead to interaction faults

- **Based on Empirical Data**
  - Most failures are induced by single factor faults
  - or by the joint combinatorial effect (interaction) of two factors
  - with progressively fewer failures induced by interactions between three or more factors
- **Example: NASA application**
  - 67% of the failures were triggered by only a single parameter value
  - 93% by 2-way combinations, and
  - 98% by 3-way combinations

But that's just one kind of application …

Interaction Failures: data from NIST

Browser faults are more complex than those for medical devices



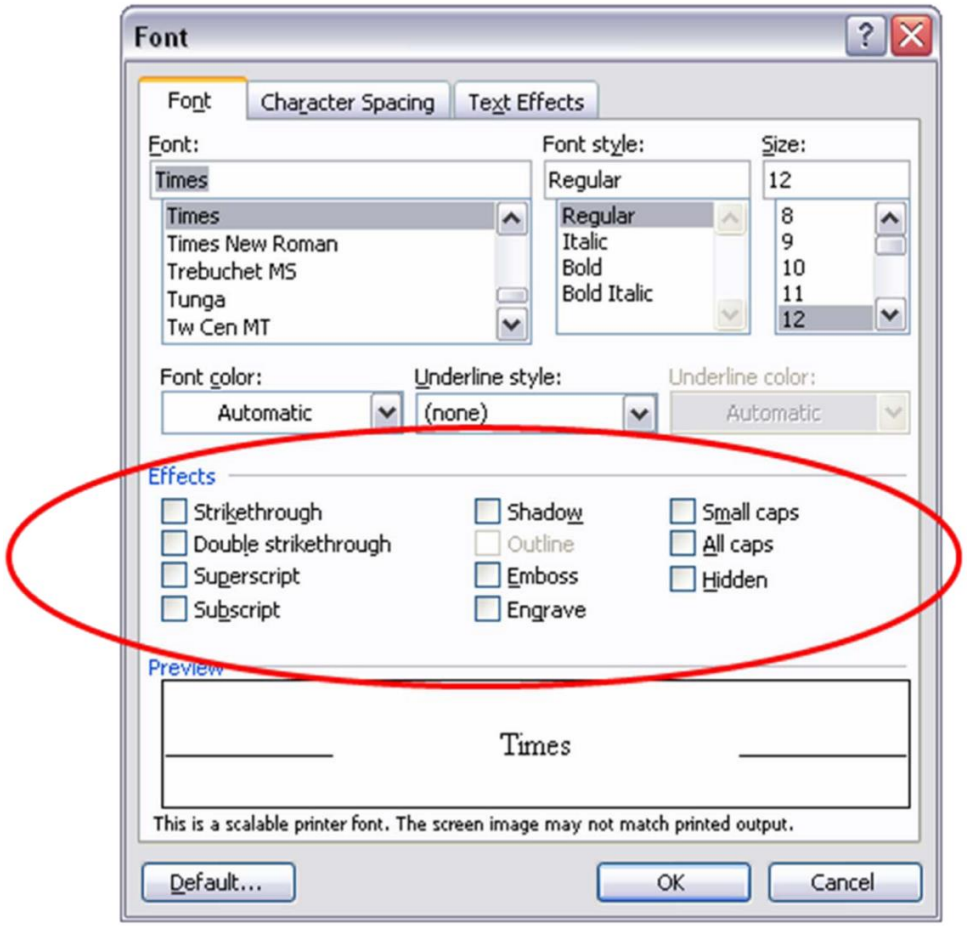
Interaction Failures

- **How is this knowledge useful?**
- **“Central Dogma”**
  - If all faults are triggered by the interaction of t or fewer variables then testing all t-way combinations can provide strong assurance

Combination of Effects

Text formatting example

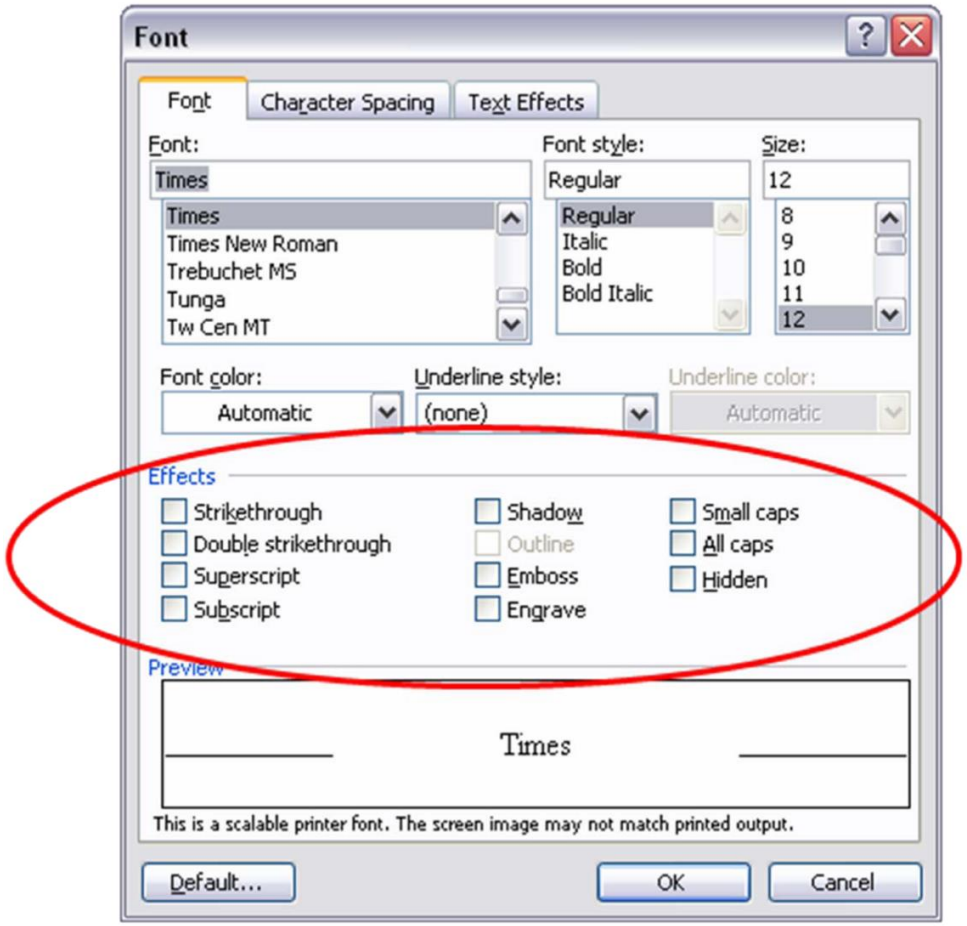
- 10 effects, each can be turned on or off
- How many tests to test all combinations?



Combination of Effects

Text formatting example

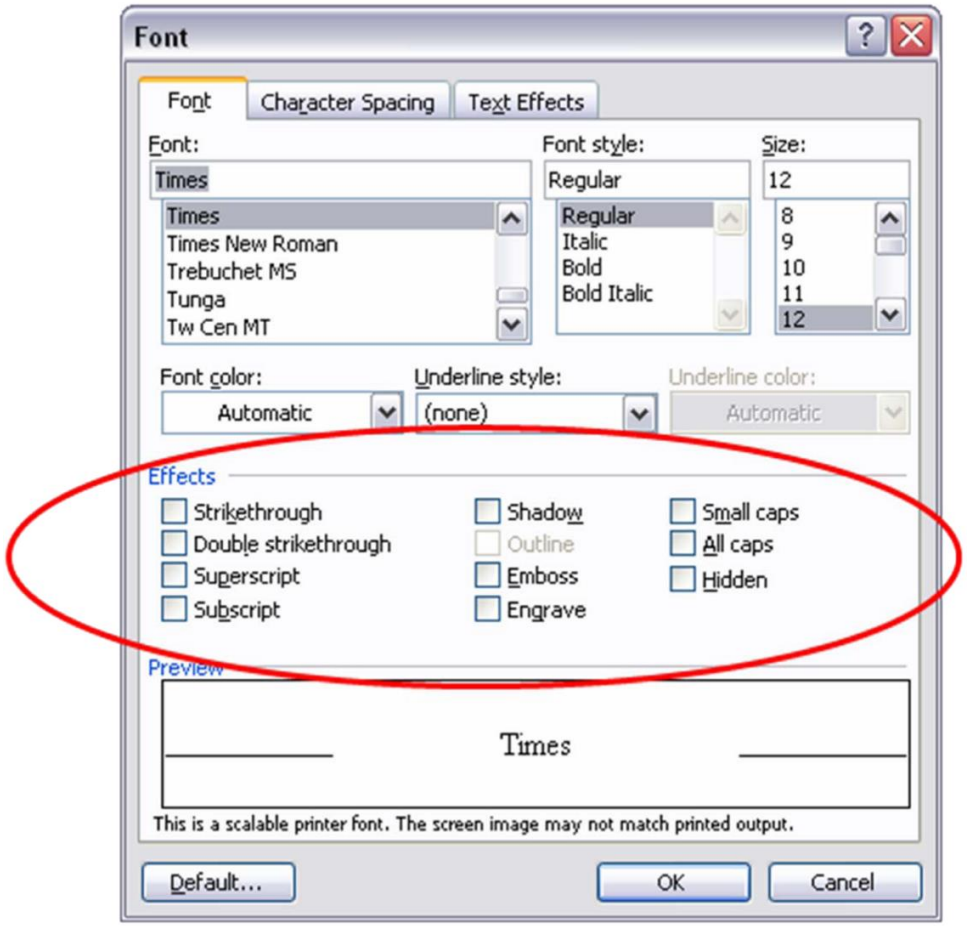
- 10 effects, each can be turned on or off
- How many tests to test all combinations?
  - All combinations is 2^{10} = 1,024
- Might not be able to do so many tests
- Instead, look only at 3-way interactions



Combination of Effects

Text formatting example

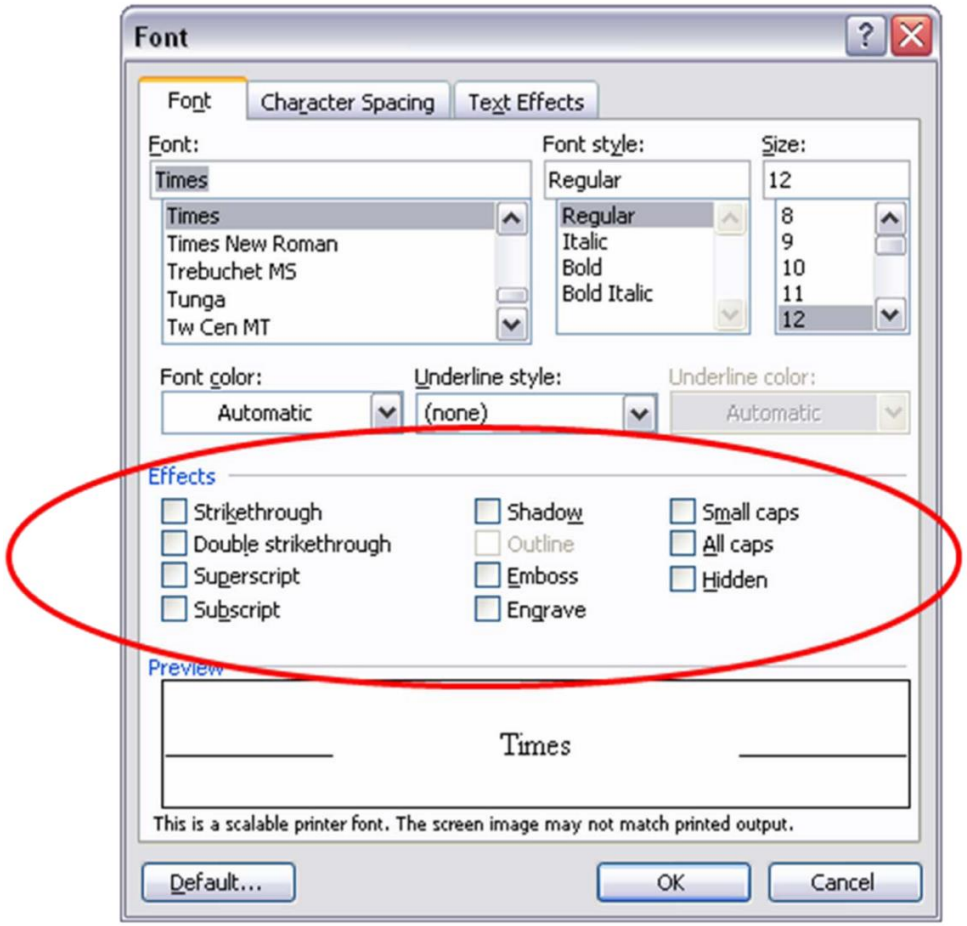
- **How many tests would it take to test all 3-way interactions?**
  - **How many combinations of 3 effects?**



Combination of Effects

Text formatting example

- **How many combinations of 3 effects?**
  - 10 ways to pick the first effect, 9 for the second, 8 for the third
  - $10 \times 9 \times 8 = 720$
  - But, the order in which you pick an effect does not matter
  - 3 ways to place the first one (1st, 2nd, or 3rd), 2 for the second
  - $3 \times 2 = 6$
  - So, $720 / 6 = 120$

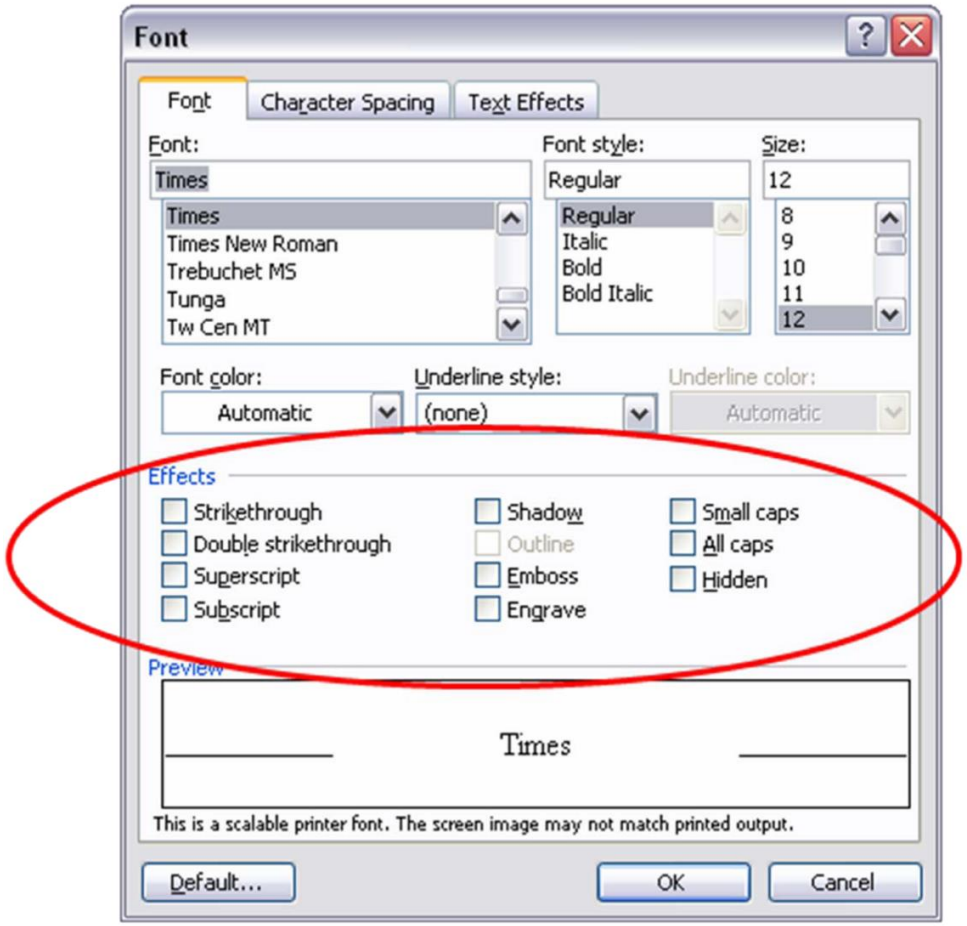


Combination of Effects

Text formatting example

- **Naïvely, 120 x 2^3 = 960 tests**
  - Each effect can be on or off, hence 2^3 tests for each 3-way interaction
- **Pack 3 triples into each test, so no more than 320 tests**
  - In each test, 10 effects to turn on or off

0 1 1 0 0 0 0 1 1 0



Covering Arrays

Text formatting example: 13 tests handle all 3-way interactions!

Each column represents an effect

Each row represents a test

0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1
1	1	1	0	1	0	0	0	0	1
1	0	1	1	0	1	0	1	0	0
1	0	0	0	1	1	1	0	0	0
0	1	1	0	0	1	0	0	1	0
0	0	1	0	1	0	1	1	1	0
1	1	0	1	0	0	1	0	1	0
0	0	0	1	1	1	0	0	1	1
0	0	1	1	0	0	1	0	0	1
0	1	0	1	1	0	0	1	0	0
1	0	0	0	0	0	0	1	1	1
0	1	0	0	0	1	1	1	0	1

Covering Arrays

Text formatting example: 13 tests handle all 3-way interactions!

- Any 3 columns have all 2^3 combinations

0 0 0
0 0 1
0 1 0
0 1 1
1 0 0
1 0 1
1 1 0
1 1 1

0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1
1	1	1	0	1	0	0	0	0	1
1	0	1	1	0	1	0	1	0	0
1	0	0	0	1	1	1	0	0	0
0	1	1	0	0	1	0	0	1	0
0	0	1	0	1	0	1	1	1	0
1	1	0	1	0	0	1	0	1	0
0	0	0	1	1	1	0	0	1	1
0	0	1	1	0	0	1	0	0	1
0	1	0	1	1	0	0	1	0	0
1	0	0	0	0	0	0	1	1	1
0	1	0	0	0	1	1	1	0	1

Covering Arrays

Text formatting example: 13 tests handle all 3-way interactions!

- Any 3 columns have all 2^3 combinations

0 0 0
0 0 1
0 1 0
0 1 1
1 0 0
1 0 1
1 1 0
1 1 1

0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1
1	1	1	0	1	0	0	0	0	1
1	0	1	1	0	1	0	1	0	0
1	0	0	0	1	1	1	0	0	0
0	1	1	0	0	1	0	0	1	0
0	0	1	0	1	0	1	1	1	0
1	1	0	1	0	0	1	0	1	0
0	0	0	1	1	1	0	0	1	1
0	0	1	1	0	0	1	0	0	1
0	1	0	1	1	0	0	1	0	0
1	0	0	0	0	0	0	1	1	1
0	1	0	0	0	1	1	1	0	1

Covering Arrays

Text formatting example: 13 tests handle all 3-way interactions!

- **A test suite that covers all k-way interactions is called a covering array**
- **Finding covering arrays is an NP-hard problem**
- **Tools are available**

0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1
1	1	1	0	1	0	0	0	0	1
1	0	1	1	0	1	0	1	0	0
1	0	0	0	1	1	1	0	0	0
0	1	1	0	0	1	0	0	1	0
0	0	1	0	1	0	1	1	1	0
1	1	0	1	0	0	1	0	1	0
0	0	0	1	1	1	0	0	1	1
0	0	1	1	0	0	1	0	0	1
0	1	0	1	1	0	0	1	0	0
1	0	0	0	0	0	0	1	1	1
0	1	0	0	0	1	1	1	0	1