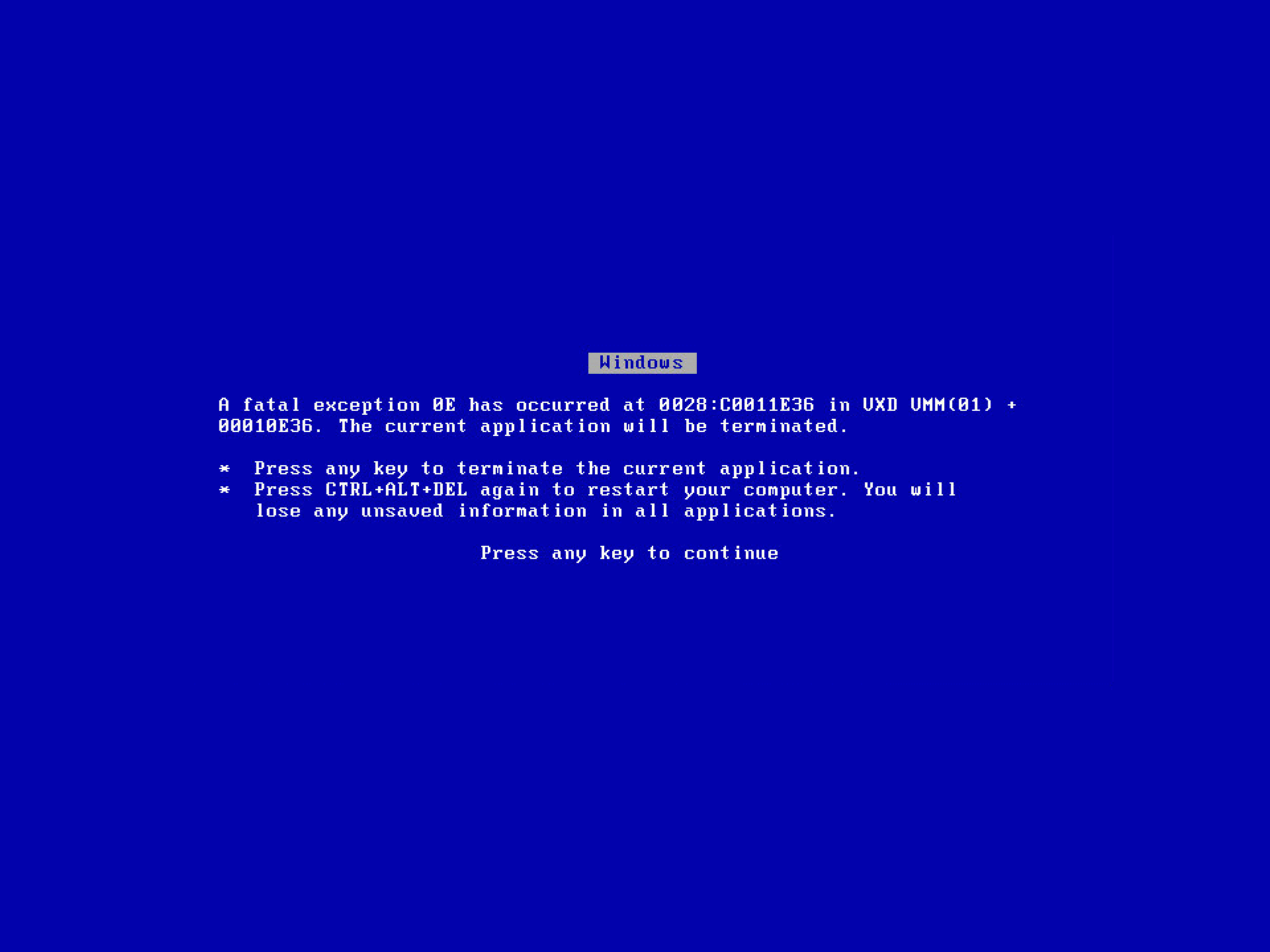
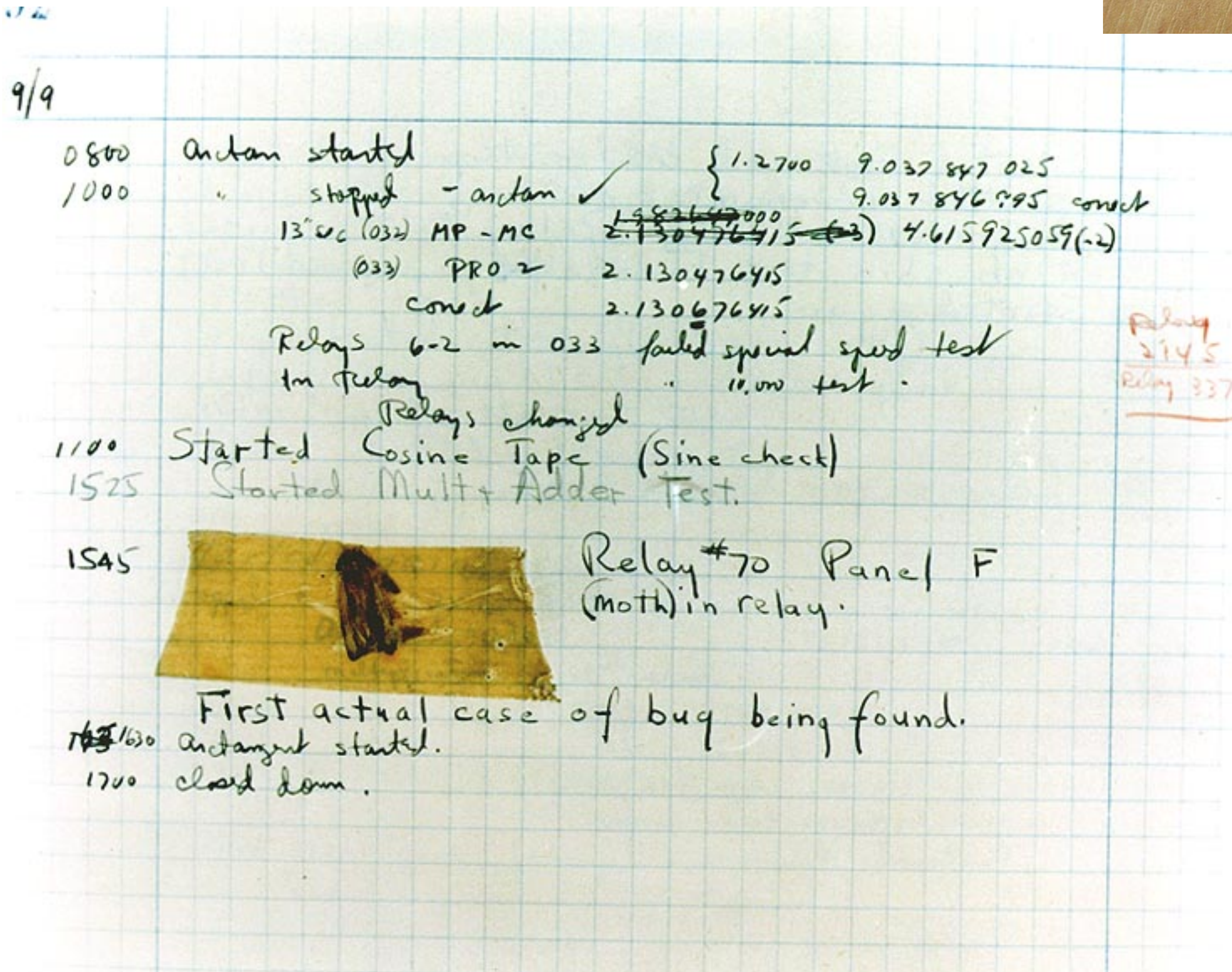
EECE 310 – Software Engineering

Software Testing:

Fundamentals of software testing


Windows
A fatal exception 0E has occurred at 0028:C0011E36 in VXD VMM(01) +
00010E36. The current application will be terminated.
* Press any key to terminate the current application.
* Press CTRL+ALT+DEL again to restart your computer. You will
lose any unsaved information in all applications.
Press any key to continue

Bug

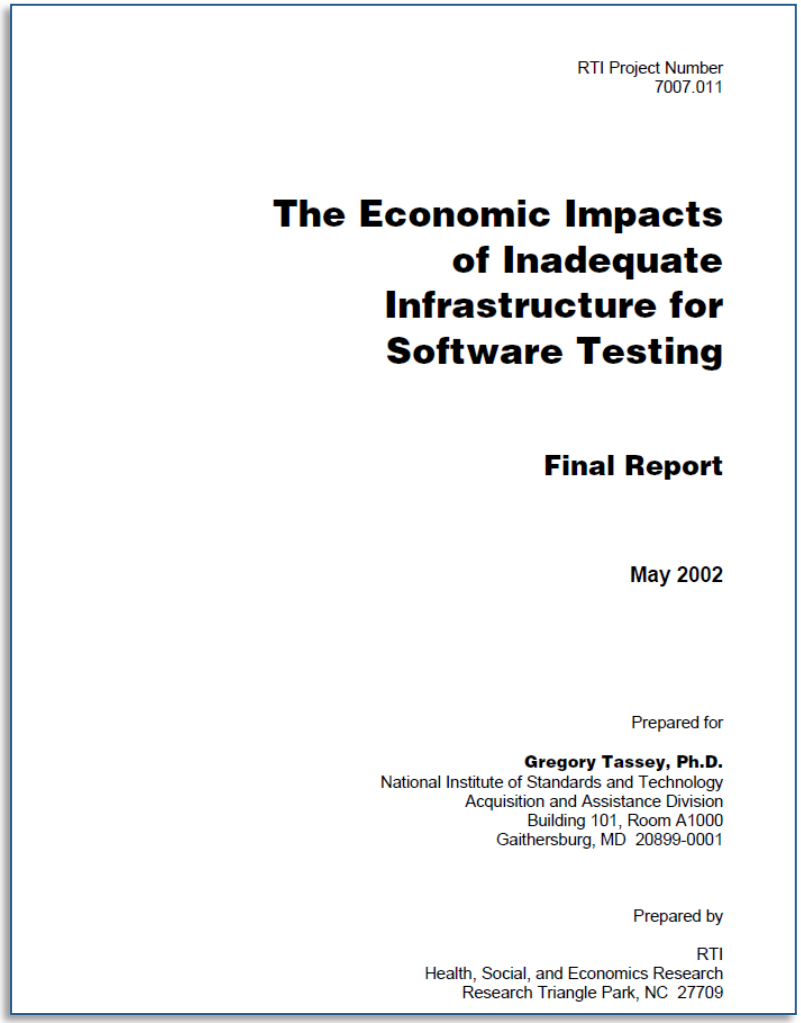

9/9
0800 Anctan started
1.2700 9.037 847 025
1000 stopped - anctan
9.037 846 995 correct
13"uc (032) MP - MC
4.615925059(-2)
(033) PRO 2 2.130476415
correct 2.130676415
Relays 6-2 in 033 failed special speed test
In relay 11,000 test.
Relays changed
1100 Started Cosine Tape (Sine check)
1525 Started Mult + Adder Test.
1545 Relay #70 Panel F
(moth) in relay.
First actual case of bug being found.
1630 Antangent started.
1700 closed down.

“Our civilization runs on software”

Bjarne Stroustrup

Can you estimate the cost of failing software systems?

Examples, Cost Analysis



RTI Project Number
7007.011

The Economic Impacts of Inadequate Infrastructure for Software Testing

Final Report

May 2002

Prepared for

Gregory Tassey, Ph.D.
National Institute of Standards and Technology
Acquisition and Assistance Division
Building 101, Room A1000
Gaithersburg, MD 20899-0001

Prepared by

RTI
Health, Social, and Economics Research
Research Triangle Park, NC 27709

US alone:
59 billion per year

Basics of program *correctness*

- Relatively few programs are **proven correct**: hard, expensive,…
- Make precise the meaning of programs
- In a logic, write down pre (*P*) and post (*Q*) conditions (specification)
- Associate precise (logical) meaning to each construct in programming language
- Reason that logical conditions are satisfied by program
  - A Hoare triple is a predicate {*P*} S {*Q*} that is true whenever *P* holds and the execution of S guarantees that *Q* holds

    {x <> 0} y := x * x; {y > 0}

A foundation for thinking about programs more precisely.

Can testing help to prevent such failures?
How much would that cost?

Software Testing

- An activity to assess the quality of a system
- Using simple scenarios that can be understood

Testing vs. Proving

- Dynamic
- Builds confidence
  - Can only show the presence of bugs, not their absence!
- Used widely in practice
- Costly

- Static
- It's a proof
  - Proofs are human processes that are not foolproof
- Applicability is practically limited
- Extremely costly

Should be considered to be complementary, not competitive. Testing is by far more dominant approach to assess software products.

Question

- In which software system is **proving** correctness preferable to **testing**?

  A. Safety-critical software.

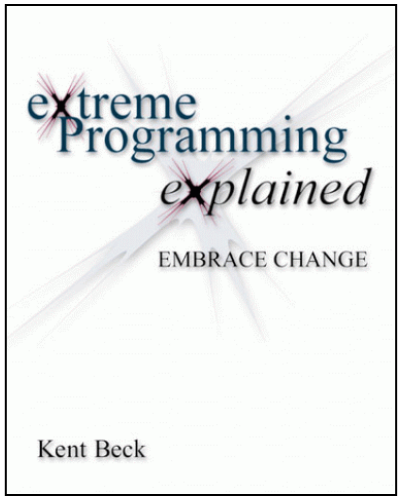
  B. Applications for iOS or Android.

  C. Microsoft Office.

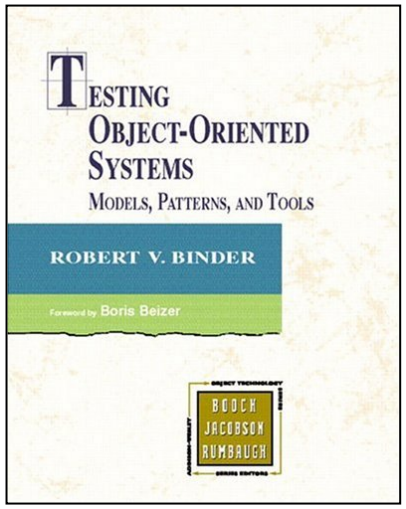
Our focus: Testing

- Basics of testing in EECE 210 and EECE 310
- Advanced testing topics in
  - "EECE 416: Software Testing and Analysis"

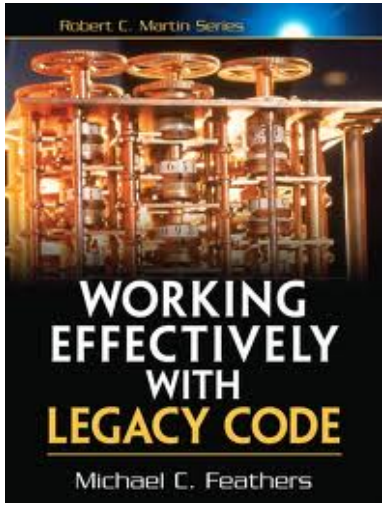
Software Testing Books



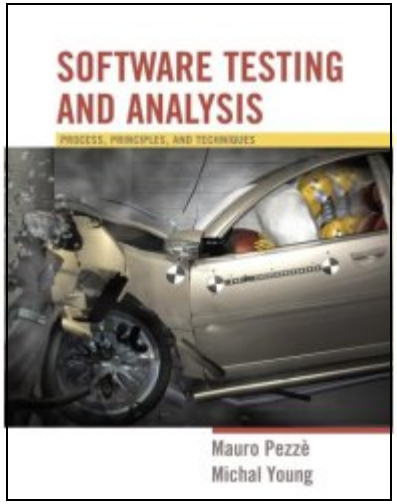
1999



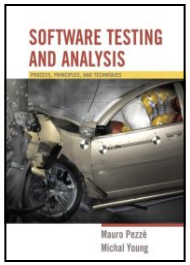
2000



2004



2007



The Book: Pezzè & Young

- Goal is to
  - *achieve balance of **costs**, **schedule**, and **quality***
- Testing and analysis are
  - *integral to modern software engineering practice*
  - *equally important and technically demanding as other aspects of development*

What is Software Testing?

Testing consists of

- the *dynamic* verification of the behavior of a program
- on a finite *set of test cases*
- suitably *selected* from the usually infinite executions domain
- against the specified *expected* behavior

Bertolino, www.swebok.org

Goals of Testing: Verification and Validation

- Validation: does the software system meet the user's real needs?

  Are we building the right software?

- Verification: does the software system meet the requirements specifications?

  Are we building the software right?

Static versus Dynamic Analysis

Static Analysis:

- Determines or estimates software quality ***without*** reference to actual executions
- Techniques: code inspection, program analysis, symbolic analysis, and model checking.

Dynamic Analysis:

- Ascertains and/or approximates software quality ***through*** actual executions, i.e., with real data and under real (or simulated) circumstances
- Techniques: synthesis of inputs, testing procedures, and the automation of testing environment generation.

A (Depressing) Truth

"Testing can show the *presence*, but not the *absence* of errors"

– Dijkstra's Law

Testing does help nonetheless!

First Code, then Test?

- Developers of the software should do no testing at all.
- Software should be "tossed over a wall" to strangers who will test it mercilessly.
- Testers should be involved with the project *only* when testing is about to begin.

First Code, then Test?

- Developers of the software should do no testing at all.
- Software should be "tossed over a wall" to strangers who will test it mercilessly.
- Testers should be involved with the project *only* when testing is about to begin.

WRONG!

Testing Phase

- **Waterfall**
- After requirements phase
- Write code then test
- Test thoroughly before release

- **Agile: TDD**
- Immediately at the start of the project
- Write test then code
- Test thoroughly continuously

Testing Terminology

- **Failure/fault/error** (or bug)
- **Test plan**: test specification
- **Test case**: a single unique unit of testing code
- **Test suite**: collection of test case
- **Test oracle**: expected behavior
- **Test fixture** (or test data)
- **Test harness:** collection of all the above + conf.

Functional versus non-functional Properties

Behavior:

- Structural testing
- Unit testing
- Regression testing
- Acceptance testing
- Integration testing
- UI Testing

Operation:

- Security testing
- Accessibility testing
- Performance testing
- Scalability testing
- Usability testing

Testing Tactics

- Tests based on *spec*
- Treats system as atomic
- Covers as much *specified* behaviour as possible

- Tests based on *code*
- Examines system internals
- Covers as much *implemented* behaviour as possible

Functional versus Structural Testing (black box versus white box)

Functional Testing:

- software program or system under test is viewed as a "**black box**".
- emphasizes on the **external** behavior of the software entity.
- selection of test cases for functional testing is based on the **requirement** or design **specification** of the software entity under test.

Functional versus Structural Testing

Structural Testing:

- the software entity is viewed as a "**white box**".
- emphasizes on the **internal** structure of the software entity.
- goal of selecting such test cases is to cause the execution of specific **statements**, program **branches** or **paths**.
- expected results are evaluated on a set of **coverage criteria**. Examples: path coverage, branch coverage, and data-flow coverage.

Identify 3 main testing challenges

Discuss with your neighbor

Testing: An easy job?

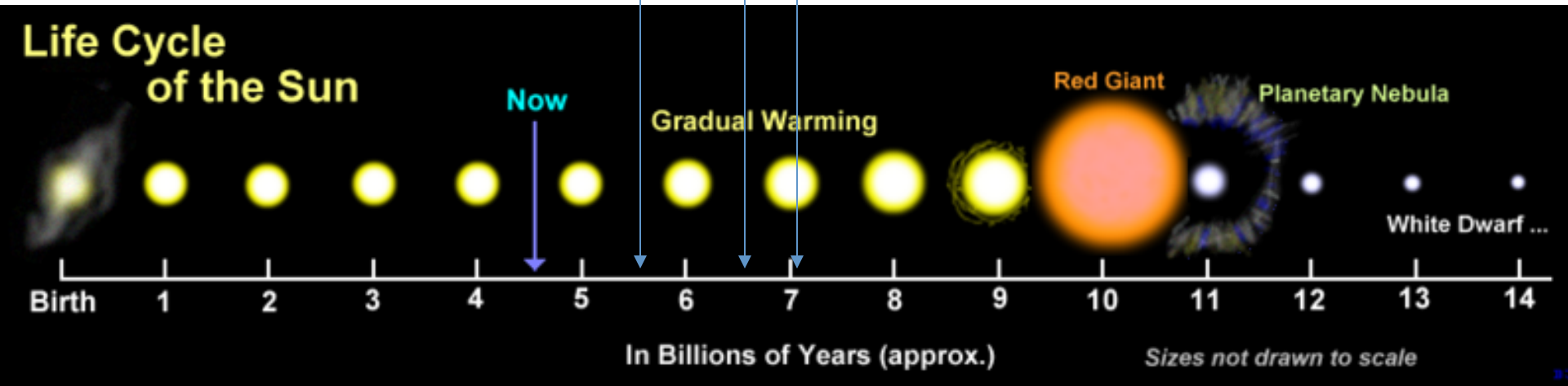
- A simple program:
  3 inputs, 1 output
- a,b,c: 32 bit integers
- trillion test cases. If each test case takes 1 second:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

All oceans dry

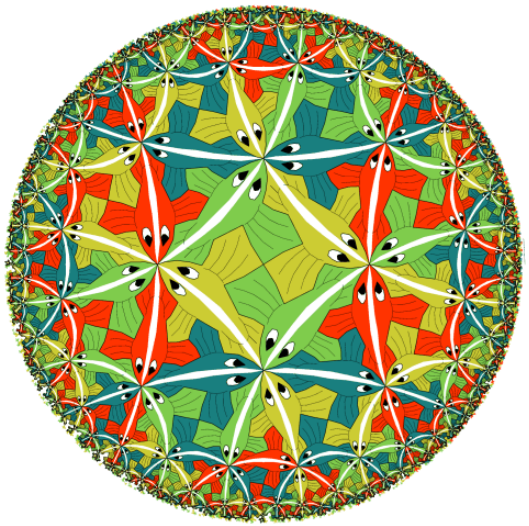
All plants dead

Tests done (2.5 bill. y)



The Set of Examples is Incomplete

- Too much data
- Too many combinations
- Too many paths



- Properties of interest fundamentally undecidable

Analysis versus Testing

- Does (desirable) property *P* hold for system S?
  - P = "Does S terminate"?

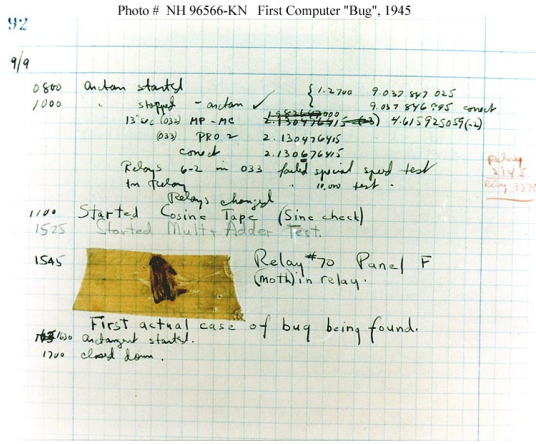
- **Software Testing: *Optimistic Accuracy***
  - S terminates on 4 inputs: it will always terminate.

- **Static analysis: *Pessimistic Accuracy***
  - We can only be sure S terminates if it doesn't contain a "while" or other looping construct
  - S contains a while, hence it doesn't terminate.

Testing Challenge: Faults May Hide

- **Failure:**
  - Manifested inability of a system to perform required function.
- **Fault:**
  - missing / incorrect code
- **Error**:
  - human action producing fault

"bug"



- **And thus**:
  - Testing: triggering failures
  - Debugging: finding faults given a failure

What should x and y be to find the bug?

```
function example(x, y) {
 if ( y > 5) {
    z = x * x; \\bug should be x + x
    if (0 < z <= 4) {
       print( z );
    }
 }
}
```

Challenge: RIP
Reach, Infect/Trigger, Propagate

You *reach* the fault only if y > 5

x * x should be x + x.
x = 0 or x = 2 won't trigger the fault

Fault only propagated if x * x <= 4.

```
if ( y > 5) {
    z = x * x;
    if (z <= 4) {
        print ( z );
    }
}
```

Bug: x * x should be x + x

Testing Challenges

- Explosion of input combinations
- Explosion of path combinations
- Most interesting properties *undecidable*
- Faults may be hard to reach (RIP)
- Faults may hide (observability)
- Faults may be hard to execute at will (controllability)
- Collected data & the external environment

Different types/levels of testing

- Who has some examples of testing from co-op or lab assignments?

Types of Testing

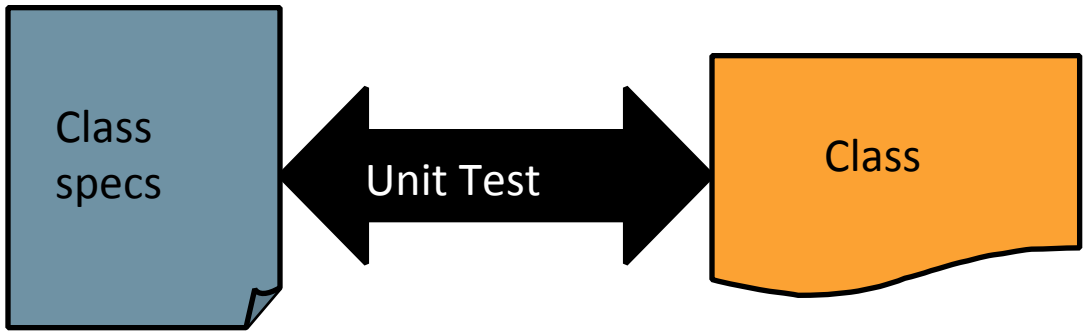
- Unit
- Integration
- Regression
- System
- Acceptance

Unit Testing – Definition

- A method of testing that verifies the individual units of source code are working properly. A unit is the **smallest testable part** of an application.
- Process of testing individual components in isolation
- Units may be:
  - functions / methods
  - classes
  - composite components with defined interfaces used to access their functionality

Unit Tests

- Uncover errors at module boundaries
- Typically written by the programmer himself/herself
- Frequently, fully automatic (see *regression*)



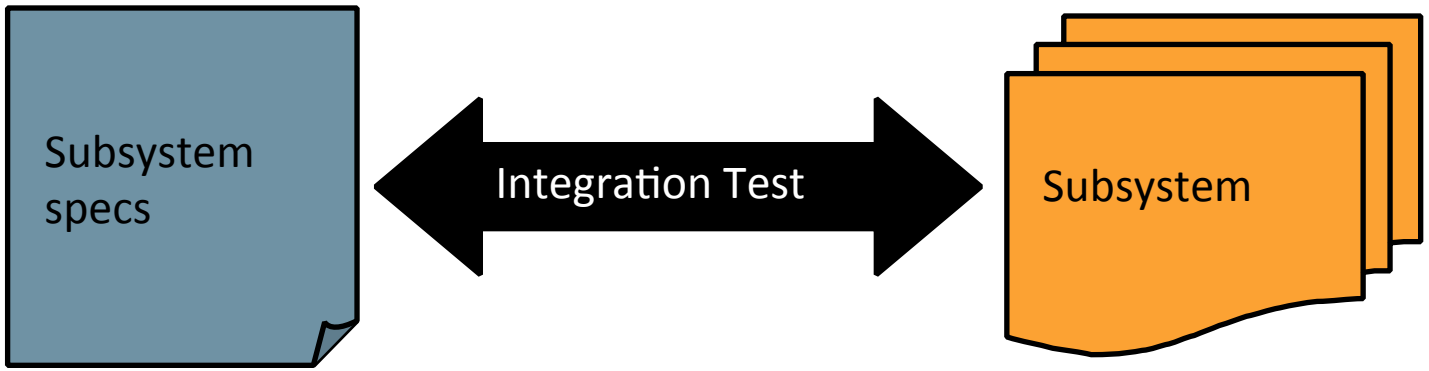
Integration Testing – Definition

- The phase of software testing in which individual software **modules are combined** and tested as a group. It follows unit testing and precedes system testing.

- Shows that major subsystems that make up the project work well together.

Integration Tests

- General idea:
  - Constructing software while conducting tests
- Options:
  - *Big Bang:* no stubs; do unit testing, then throw all parts together
  - Incremental Construction



Mock Objects (Stubs)

- Often we rely on portions of the system which are
  - non-deterministic
  - slow
  - not yet complete/built
- Mocks adhere to the contract (interface) but simulate behavior (no actual impl)
- Real implementation is substituted later when available

Mock example

```
public class SystemEnv implements Env {
   public long getTime() {
      return System.currentTimeMillis();
   }
   //other methods
}
```

```
public class MockSystemEnvironment implements Env {
   private long current_time;
   public long getTime() { return current_time; }
   public void setTime(long aTime) {
      current_time = aTime; }
}
```

from Pragmatic Unit Testing book;
see **JMock** library as an example

Regression Testing

From Sommerville's slides

Regression Testing

- Testing the system to check that changes have not ‘broken’ previously working code.
- In manual testing process, **regression** testing is expensive but with automated testing it is simple and straightforward. All tests rerun every time a change is made to the program.
- Tests must run ‘successfully’ before the change is committed.

From Sommerville’s slides

Challenges in Regression Testing

Challenges in Regression Testing

- Rerunning all the test cases after each change is costly
- Techniques:
  - Test Selection: only run affected tests
  - Test Prioritization: rank tests first, start running from top

System Testing – Definition

- Testing large parts of the system, or the whole system, for functionality.
- Involves integrating components to create a version of the system and then testing the integrated system.
- Tests the emergent behavior of a system.

System Tests – Deploy the system!

- Recovery testing
  - Forces the software to fail in various ways and verifies that recovery is properly performed.
- Security testing
  - Verifies that the protection mechanism will protect the software against improper penetration
- Stress testing
  - Executes the system in a manner that demands resources in abnormal quantity, frequency or volume

System Tests

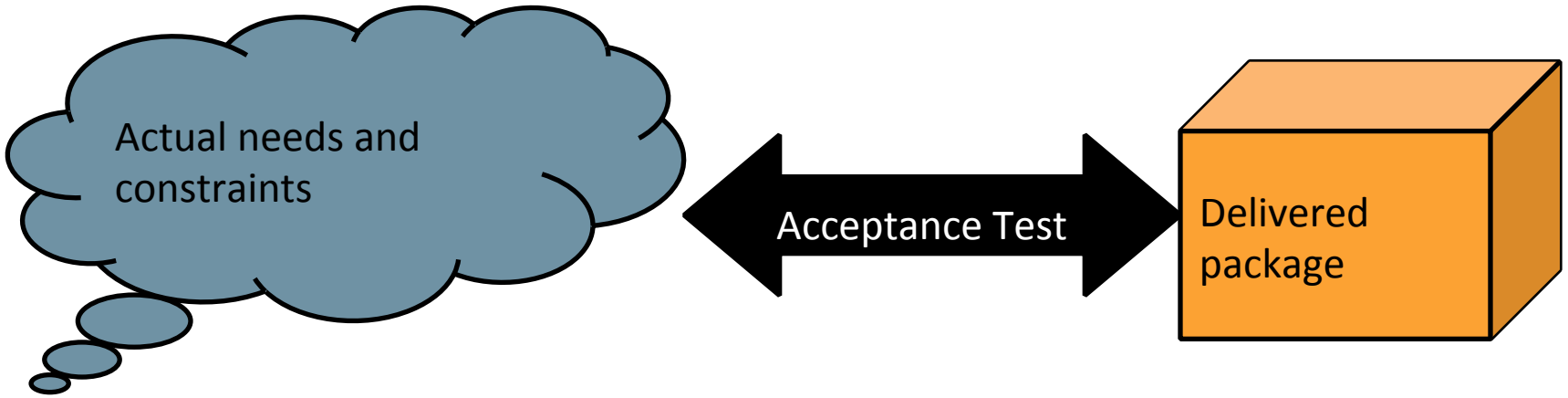
- *Capacity* testing
  - Evaluates the upper limit of some operational parameters (*i.e.* max number of users)
- *Usability* testing
  - Determines if the chosen design is efficient for the desired task (*i.e.* GUI usability)
- *Performance* testing
  - Test the run-time performance of the software in the context of a fully integrated system (CPU, memory, energy)

Acceptance Testing – Definition

- Formal testing with respect to **user needs and requirements** conducted to determine whether or not the software satisfies the acceptance criteria.
- Does the end product solve the problem it was intended to solve?

Acceptance Tests

- Checks if the *contractual requirements* are met
- Typically incremental
  - Alpha test: At production site
  - Beta test: At user's site
- Work is over when acceptance testing is done



When to stop?

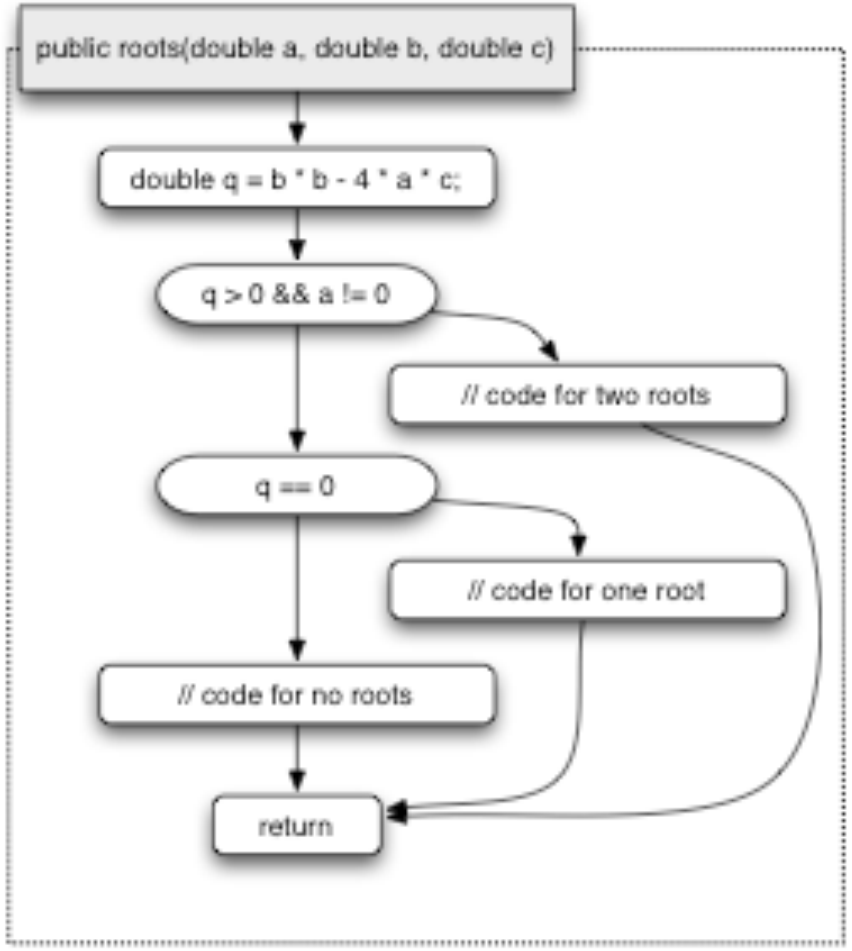
- Stopping criteria
  - Have I tested enough?

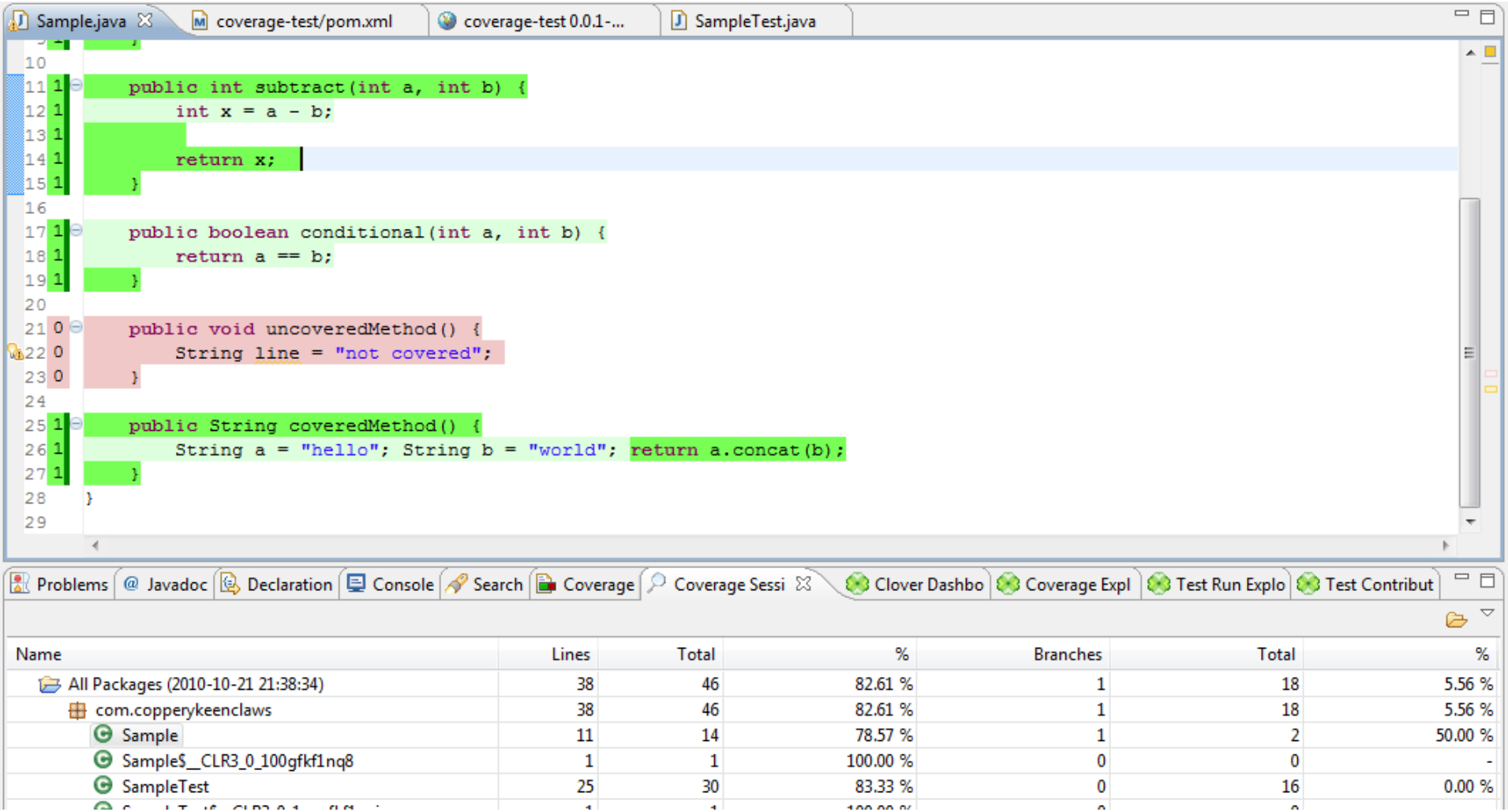
Stopping Criteria (when to stop testing)

- Unit test for every method?
- Number of tests?
- Coverage?
  - Each Statement?
  - Each Branch?
  - Each Path?
- Equivalence Partitioning?
- Boundary Tests?

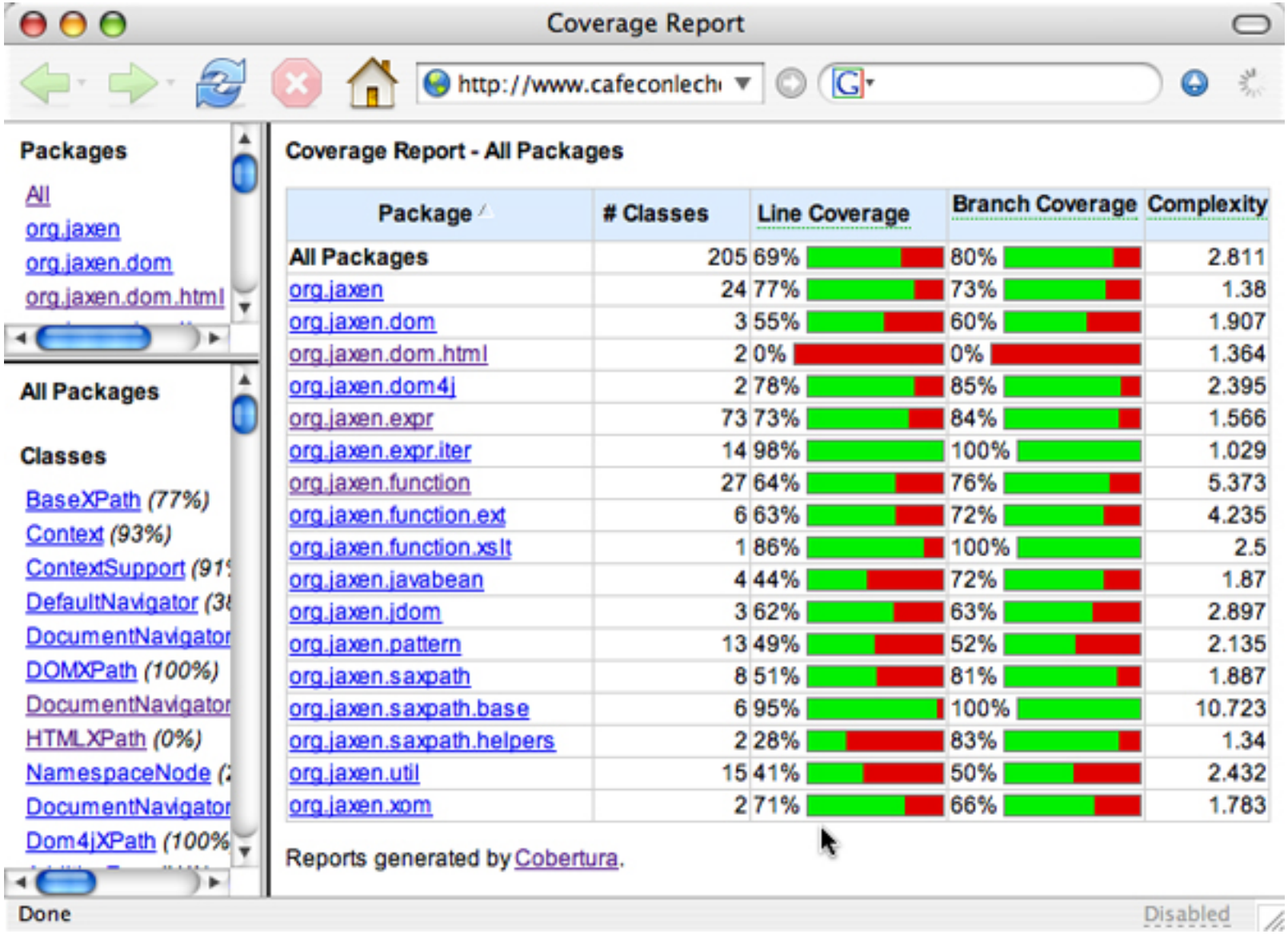
What is Coverage?

- The more ***parts*** are executed, the higher the chance that a test uncovers a defect.
- ***Parts*** can be nodes, edges, paths, conditions…
- Each lead to a different definition of ***coverage***.




Sample.java
coverage-test/pom.xml
coverage-test 0.0.1-...
SampleTest.java
10
11 1 public int subtract(int a, int b) {
12 1 int x = a - b;
13 1
14 1 return x;
15 1 }
16
17 1 public boolean conditional(int a, int b) {
18 1 return a == b;
19 1 }
20
21 0 public void uncoveredMethod() {
22 0 String line = "not covered";
23 0 }
24
25 1 public String coveredMethod() {
26 1 String a = "hello"; String b = "world"; return a.concat(b);
27 1 }
28 }
29
Problems
Javadoc
Declaration
Console
Search
Coverage
Coverage Sessi
Clover Dashbo
Coverage Expl
Test Run Explo
Test Contribut
Name
Lines
Total
%
Branches
Total
%
All Packages (2010-10-21 21:38:34)
38
46
82.61 %
1
18
5.56 %
com.copperykeenclaws
38
46
82.61 %
1
18
5.56 %
Sample
11
14
78.57 %
1
2
50.00 %
Sample$__CLR3_0_100gfkf1nq8
1
1
100.00 %
0
0
-
SampleTest
25
30
83.33 %
0
16
0.00 %


Coverage Report
Coverage Report - DoSomething
Classes in this File
Line Coverage
Branch Coverage
DoSomething
75% 6/8
75% 3/4
1
2 4 public class DoSomething {
3
4 /**
5 *
6 * Given a string and a started index, this function will count number of charact
7 * from a given started index.
8 *
9 *
10 * @param str a string containing number of characters.
11 * @param startedIndex location where we start counting number of characters.
12 * @return number of characters contained in the string str from the startedIndex
13 */
14 public int countCharacter(String str, int startedIndex){
15 4 if (str.length() > 0)
16 {
17 2 str = str.substring(startedIndex);
18 2 return str.length();
19 }
20 1 if (str.length() > 100)
21 {
22 0 str = str.substring(startedIndex);
23 0 return str.length() + 5;
24 }
25 1 return 0;
26 }
27 }
Report generated by Cobertura 1.9 on 8/16/08 11:03 AM.


Coverage Report
http://www.cafeconlech
Packages
All
org.jaxen
org.jaxen.dom
org.jaxen.dom.html
All Packages
Classes
BaseXPath (77%)
Context (93%)
ContextSupport (91
DefaultNavigator (3
DocumentNavigator
DOMXPath (100%)
DocumentNavigator
HTMLXPath (0%)
NamespaceNode (2
DocumentNavigator
Dom4jXPath (100%
Coverage Report - All Packages
Package
Classes
Line Coverage
Branch Coverage
Complexity
All Packages 205 69% 80% 2.811
org.jaxen 24 77% 73% 1.38
org.jaxen.dom 3 55% 60% 1.907
org.jaxen.dom.html 2 0% 0% 1.364
org.jaxen.dom4j 2 78% 85% 2.395
org.jaxen.expr 73 73% 84% 1.566
org.jaxen.expr.iter 14 98% 100% 1.029
org.jaxen.function 27 64% 76% 5.373
org.jaxen.function.ext 6 63% 72% 4.235
org.jaxen.function.xslt 1 86% 100% 2.5
org.jaxen.javabean 4 44% 72% 1.87
org.jaxen.jdom 3 62% 63% 2.897
org.jaxen.pattern 13 49% 52% 2.135
org.jaxen.saxpath 8 51% 81% 1.887
org.jaxen.saxpath.base 6 95% 100% 10.723
org.jaxen.saxpath.helpers 2 28% 83% 1.34
org.jaxen.util 15 41% 50% 2.432
org.jaxen.xom 2 71% 66% 1.783
Reports generated by Cobertura.
Done
Disabled

Testing: An easy job?

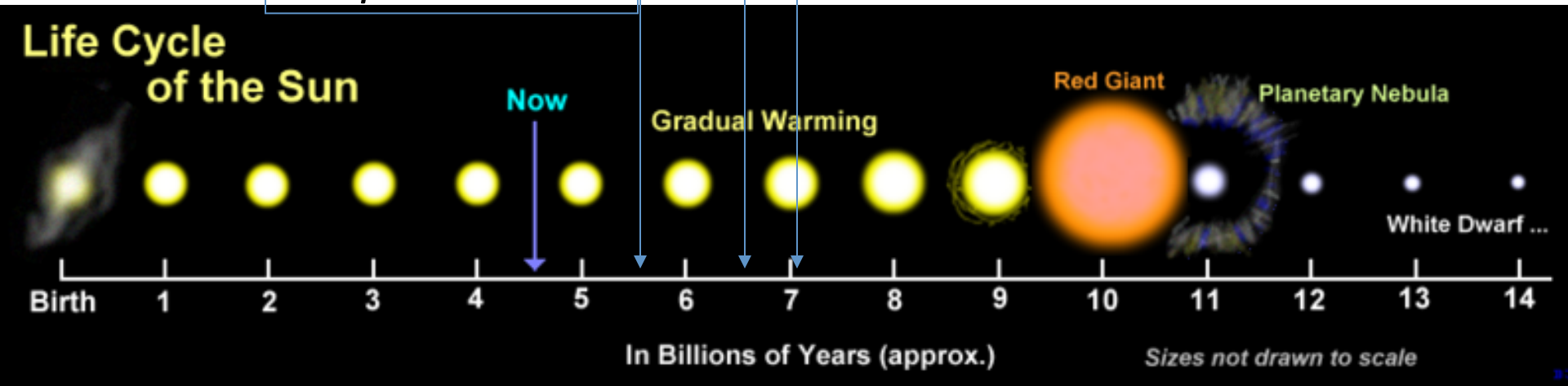
- A simple program:
  3 inputs, 1 output
- a,b,c: 32 bit integers
- trillion test cases. If each test case takes 1 second:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

All oceans dry

All plants dead

Tests done (2.5 bill. y)



Challenges in Testing

- **Too much data**
- **Too many combinations**
- **Too many paths**

Equivalence Class Partitioning

- ***Identify*** which types of inputs are likely to be processed in a similar way (equivalent behaviour)
  - Coverage, disjointedness, representation
- This creates ***equivalence partitions***
- Each test should exercise ***one and only one*** equivalence partition

Equivalence Partitioning Example

- Assume our application accepts an integer between 100 and 999
  - foo(int x) // 100 <= x <= 999

- How would you partition this into equivalence classes for testing?

Equivalence Partitioning Example

foo(int x) // 100 <= x <= 999

- Valid Equivalence Class partition: 100--999 inclusive. *(test with 350)*
- Non-valid Equivalence Class partitions:
  - less than 100 *(test with 5)*
  - more than 999 *(test with 1250)*

Boundary Value Testing

- Boundary value testing fault model:
  - Values close to partition **boundaries** are often *troublesome*
  - System has incorrectly implemented a boundary
- **On point:** value that lies on a boundary
- **Off point:** not on a boundary
  - ***But as close as possible to the boundary!***

A Simplified Domain-Testing Strategy

BINGCHIANG JENG
Sun Yat-Sen University
and
ELAINE J. WEYUKER
New York University

A simplified form of domain testing is proposed that is substantially cheaper than previously proposed versions, and is applicable to a much larger class of programs. In particular, the traditional restrictions to programs containing only linear predicates and variables defined over continuous domains are removed. In addition, an approach to path selection is proposed to be used in conjunction with the new strategy.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms: Reliability, Theory, Verification

Additional Key Words and Phrases: Domain testing, software testing

1. INTRODUCTION

Domain testing is a fault-based software-testing strategy proposed by White and Cohen [1978]. Testers have frequently observed that subdomain boundaries are particularly fault-prone and should therefore be carefully checked. Domain testing was proposed as a relatively sophisticated form of boundary value testing, and is applicable whenever the input domain is subdivided into subdomains by the program's decision statements.

In this paper, a simplified version of domain testing is described which removes several limitations associated with earlier domain-testing strategies. In particular, our strategy is applicable to arbitrary types of predicates, detects both linear and nonlinear errors, for both discrete and continuous variable spaces. In addition, we will show that our new technique requires much smaller test suites than earlier versions, and will argue that its effectiveness is comparable to, and in some cases superior to, the others.

This work was supported in part by NSF grant CCR-8920701 and by NASA grant NAG-1-1238
Authors' addresses: B. Jeng, Department of Information Management, National Sun Yat-Sen University, Kaohsiung, Taiwan 80424, R.O.C.; E. J. Weyuker, Courant Institute of Mathematical Sciences, New York University, 251 Mercer Street, New York, NY 10012, and AT & T Bell Laboratories, Room 2b-442, 600 Mountain Ave., Murray Hill, NJ 07974.

ACM Transactions on Software Engineering and Methodology, Vol. 3, No. 3, July 1994, Pages 254–270

Jeng & Weyuker. A simplified domain-testing strategy. *ACM TOSEM*, 1994. Also in Binder, 2000.

Open/Closed Boundaries

- **Open boundary**:
  - Strict inequality operator: **a > 3**
  - On point: (a=3) makes the condition *false*
  - false is covered, we need to cover true now
  - Off point: (a=4) makes it *true,* and is an *in point (4 is in the accepted range)*

Open/Closed Boundaries

- **Closed boundary:**
  - Strict equality: **a <= 100**
  - On point: (a=100) makes the condition *true*
  - *true is covered, we need to cover false now*
  - Off point: (a=101) makes it false, and is an *out point (101 falls out of the accepted range)*

Fill in the boundary values for x and y

Boundary conditions for " x > 0 && x <= 10 && y >= 1.0"								
Boundary			**Test Cases**					
Variable	**Condition**	**type**	t1	t1	t3	t4	t5	t6
x	> 0	on						
		off						
	<= 10	on						
		off						
	typical	in						
y	>= 1.0	on						
		off						
	typical	in						
Expected result								

Filled With Values

Boundary conditions for " x > 0 && x <= 10 && y >= 1.0"								
Variable	**Condition**	**type**	t1	t1	t3	t4	t5	t6
x	> 0	on	0					
		off		1				
	<= 10	on			10			
		off				11		
	typical	in					4	6
y	>= 1.0	on					1.0	
		off						0.9
	typical	in	10.0	16.0	109.3	2390.2		
Result			R	ok	ok	R	ok	R

Emerging philosophies of testing

- Exploratory testing
  - Test plan is pointless
  - Testing is a creative pursuit but a disciplined one.
- Acceptance tests as frameworks
  - Fit, RSpec
  - Let the customer “understand” what the acceptance criteria are, and link that to the unit tests.