Managing the Test Execution Process

Robin F. Goldsmith, JD

robin_goldsmith@iist.org

Who am I?

- Works directly with and trains professionals in quality and testing, requirements, software acquisition, project and process management, metrics, and ROI.
- Previously a developer, systems programmer/DBA/QA, and project leader with the City of Cleveland, leading financial institutions, and a "Big 4" consulting firm.
- Degrees: Kenyon College, A.B.; Pennsylvania State University, M.S. in Psychology; Suffolk University, J.D.; Boston University, LL.M. in Tax Law.
- Member of IEEE Std. 829 and 730 revision Working Groups.
- Subject expert for IIBA BABOK and for TechTarget.
- Author of Artech House book, *Discovering REAL Business Requirements for Software Project Success*.

This course counts as one day towards the CSTP Body of Knowledge area #4 or as a one-day elective towards the CSQM, CTM, and CSTAS requirements

You should have received an email with a link to the certification exam. Please make sure to complete the exam within 30 days of starting the course. Also, please make sure to complete the exam within that time limit indicated on the exam.

The exam for this course is a Closed book exam. That is why you only receive the course written material for future reference after you pass the exam.

For details on the CSQM, CTM, CSTP, CSTAS, and other IIST certifications, visit www.iist.org

Warm-Up

	Disagree Agree
We spend a significant portion of time on testing	1 2 3 4 5
Systems go into production with no errors or bugs	1 2 3 4 5
We follow written test plans for our projects	1 2 3 4 5
Spontaneous tests find lots of bugs	1 2 3 4 5
Defects are categorized, tracked, and analyzed	1 2 3 4 5
We count on users to do a lot of the testing	1 2 3 4 5
We measure and reward effective testing	1 2 3 4 5
I have enough time to test well	1 2 3 4 5

The value of improving our test execution management

To my organization...

To me...

My objectives for this course are...

Those taking the recorded version of this course: Please feel free to pause the course and email your answers to me at Robin_Goldsmith@IIST.org

Your Objectives

- .

Objectives

Participants should be able to:

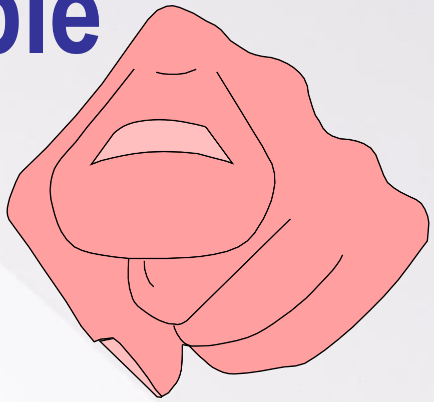
- Describe the essential elements of a test case, additional test documentation, and a structure to manage large volumes of testware
- Identify types of automated tools for supporting a managed testing environment and for executing tests
- Isolate and report defects so they are addressed
- Write effective testing status case reports
- Apply methods to reliably keep testing efforts on track and economical
- Measure both testing of particular software and overall test process effectiveness

Course Topics

1. Defining Test Cases
2. Planning and Designing the Best Tests
3. Testing Infrastructure—Technical
4. Isolating and Reporting Defects
5. Relating Testing Project and Process

You will learn more by pausing during the class to do the exercises and emailing your answers to me for feedback at Robin_Goldsmith@IIST.org

YOU Are Responsible for Your Results!

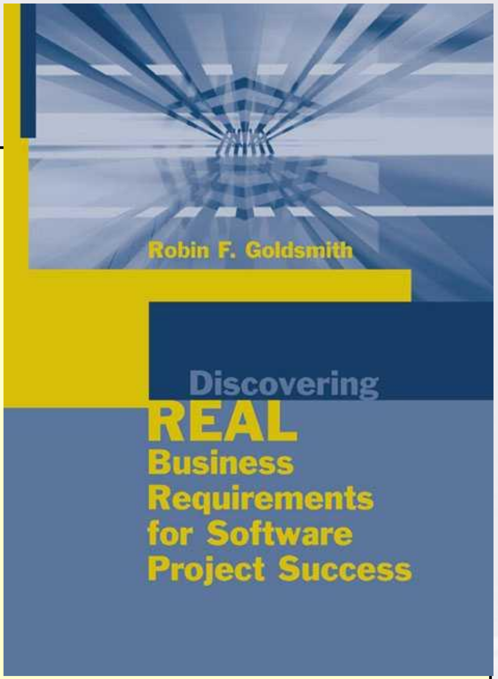


Essential belief for managing projects

- Only ***YOU*** can learn/use the ideas
- ***YOUR*** active interest and openness are essential; "can it work?" not just does it fit your current mold?
- I try to be open too and answer as best I can.
  - If you're not getting what you want, or you're having trouble with *any* aspect of the class, ***YOU*** must act to make me aware while there's time to address your need.
  - That's how you get what you need and I improve.

Holding issues for the evaluations at the end is not appropriate

Some Testing Resources


Robin F. Goldsmith
Discovering REAL Business Requirements for Software Project Success

Glenford J. Myers *The Art of Software Testing* John Wiley
William Hetzel *The Complete Guide to Software Testing* John Wiley
William E. Perry *A Structured Approach to Systems Testing* John Wiley
Boris Beizer *Software System Testing and Quality Assurance*
Software Testing Techniques Van Nostrand Reinhold
Cem Kaner, Jack Falk, Hung Nguyen *Testing Computer Software*
International Thomson Computer Press
Daniel P. Freedman & Gerald M. Weinberg *Handbook of Walkthroughs, Inspections, and Technical Reviews* Little, Brown and Company
Rex Black *Managing the Testing Process* Microsoft Press
Robin F. Goldsmith *Discovering REAL Business Requirements for Software Project Success*
Artech House http://www.artechhouse.com/Default.asp?Frame=Book.asp&Book=1-58053-770-7
IEEE (800) 678-IEEE Computer Society (800) CS-BOOKS (714) 821-8380 www.ieee.org
American Society for Quality, Software Division (800) 248-1946 (414) 272-8575 www.asq.org
STAR (Software Testing Analysis & Review) Conference and *Better Software* (*Software Testing & Quality Engineering, STQE)* magazine (800) 423-TEST (904) 278-0707 www.stickyminds.com
SearchSoftwareQuality.com http://searchsoftwarequality.techtarget.com/
Software Quality and Test Management Conference (SQTM) www.qualitymanagementconference.com
International Institute for Software Testing and Software Dimensions (800) GET-IIST www.iist.org

Quizzes and Certification Exam

- *For those attending the recorded session*, multiple-choice quizzes for almost each chapter are accessible at points within the course
- Quizzes do not affect your certification and are only for your learning assistance, so guide your use accordingly
- Directions for accessing the certification exam at the end of the course should have been sent with your course registration

1. Defining Test Cases

A *test case* is the fundamental element of testing

What Is a Test Case? How Much to Write?

- Essential: Inputs/conditions and expected results (outputs and changes to stored data, environment, state)
- Test case identification
  - ID, version number, name, description, resp. person
  - Cross-refs to features, requirements; category
- Pre-conditions (system/configuration, repeatable initial state, environment)
- Test procedure [advisable to keep separate]
  - Set-up, environment, tools and facilities, execution steps, results capture, environment restoration

See http://itknowledgeexchange.techtarget.com/software-quality/top-ten-software-quality-tips-of-2010/

Estimation Rules of Thumb: Get Consensus--What is a Test Case? 1 of 2

Which level is the test case: A, B, C, D, or E? e.g., B=7 test cases

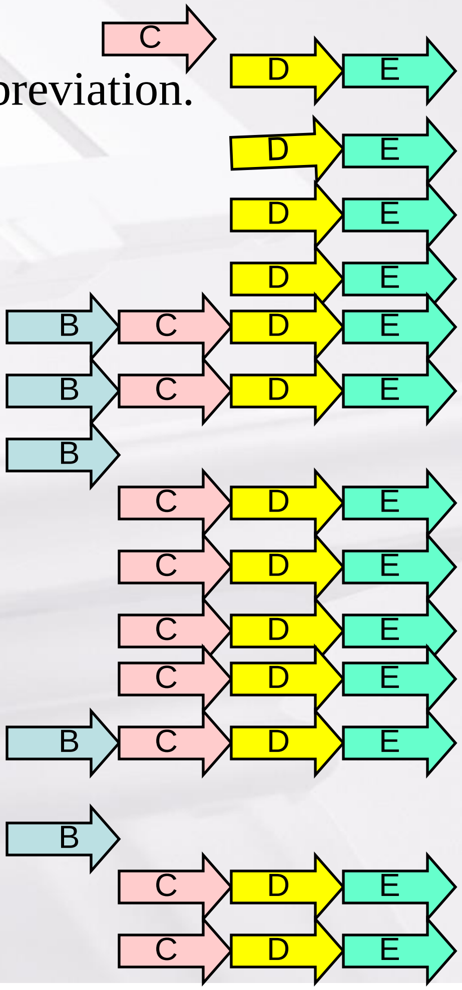
A- 1	A
B- 7	B
C-14	C
D-20	D
E-24	E

1.	Enter an order for a customer.	A
1.a	Existing customer.	B
1.a.1	Valid existing customer ID, customer is found.	C D E
1.a.2	Invalid customer ID, customer is not found.	C D E
1.b	New customer.	B
1.b.1	Valid name and address, added.	C
1.b.1.1	Valid state abbreviation.	D
1.b.1.1.1	First state (AK).	E
1.b.1.1.2	Last state (WY).	E
1.b.1.1.3	Short state name (IA).	E
1.b.1.1.4	Long state name (NC).	E
1.b.1.1.5	Delete and re-enter (MI,MN).	E
1.b.1.2	Invalid state abbreviation (MM).	D E
1.b.2	Valid name and address, not added.	C
1.b.2.1	Customer already exists.	D E
1.b.2.2	No disk space.	D E
1.b.2.3	Violates business rule, e.g., won't sell to PO Box.	D E

Estimation Rules of Thumb: Get Consensus--What is a Test Case? 2 of 2

Take the survey at www.gopromanagement.com

1.b.3 Invalid state abbreviation, not added.
1.b.3.1 Two alpha characters, but not a real state abbreviation.
1.b.3.2 Lower case not accepted.
1.b.3.3 One alpha character.
1.b.3.4 Blank.
1.c Cancel the transaction, nothing ordered.
1.d Order an item (valid item number and quantity).
1.e Fail to order an item.
1.e.1 Invalid item number.
1.e.2 Invalid quantity.
1.e.3 Valid item number and quantity, none on hand.
1.e.4 Cancel the transaction.
1.f Submit the completed order (valid customer and item/quantity), ordered.
1.g Fail to complete the order.
1.g.1 Submit without valid item/quantity.
1.g.2 System crashes.



How Much to Write: Keystroke-Level Procedure Embedded Within Test Case

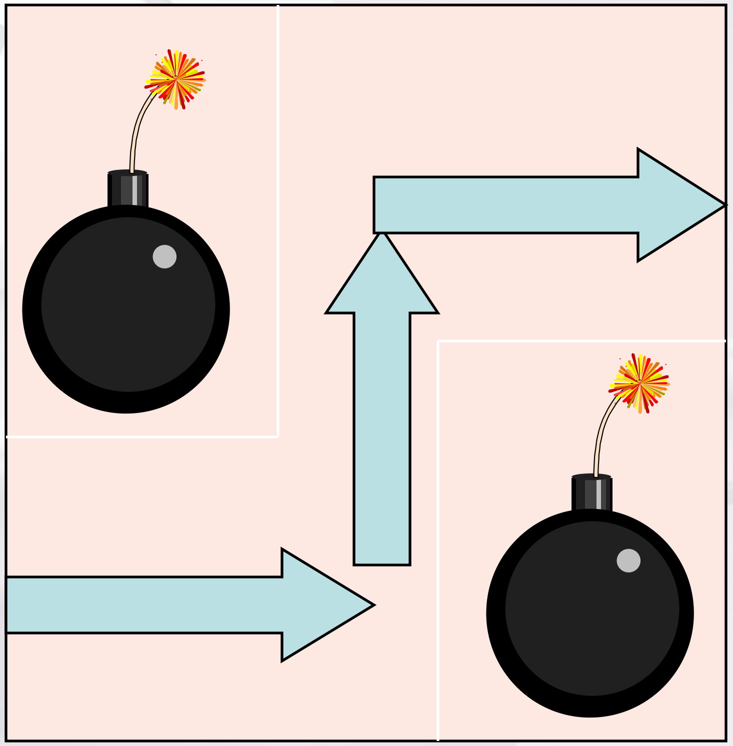
- **Pro**
  - Enables execution by low-priced people with negligible knowledge
  - Increases chances of precise repetition
- **Con**
  - Lots of high-priced time to create and maintain
  - Time spent writing reduces number of tests and time for executing tests
  - Impedes automation
  - Forces execution unlike a user's use
  - Virtually assures finding the least amount of errors

An automated test execution tool can do both: faster, cheaper, and more reliably

Exploratory Testing, Error Guessing

"Gurus" vs. Journeymen's Use

- Experienced testers find two-three times as many errors with same script (Cem Kaner)
- Test Manager's challenges
  - Inefficient, illusory (no right answer), not reliably repeatable
  - Focuses on what was written (mainly code), not what should have been (design)
  - *Analyze findings to supplement and refine structured test design*



Minefield effect, esp. for regression testing and excessive scripting

Typical Test Case Definition

Test Case Specification

Input, Condition

Operator enters customer number at location *X*.

Expected Result

System looks up customer in database and displays customer name at location *Y*.

What else do you need to perform this test?

Defining Test Cases This Way ...

Test Case Specification

<u>Input, Condition</u>

Operator enters customer number at location *X*.

<u>Expected Result</u>

System looks up customer in database and displays customer name at location *Y*.

- Interruptions, delay to find data
- Possibility of errors
  - Finding input data values
  - Checking validity of results
- Limitations on who can find data

Specify Exact Input, Expected Result

Test Case Specification

Input, Condition

Operator enters customer number at location *X*.

Expected Result

System looks up customer in database and displays customer name at location *Y*.

Low overhead

Test Case Values

Customer Number	Customer Name
C123	*Jones, John P.*
C124	*not found*

Test Script

<u>Input</u>	<u>Expected Result</u>	<u>Actual</u>
Menu=*Find Customer*	Customer entry screen	
Cust. No. = *C123*	Cust. Name *Jones, John P.*	
Cancel button	Menu	
Menu=*Find Customer*	Customer entry screen	
Cust. No. = *C124*	Cust. Name *Not Found*	
Cancel button	Menu	

Low overhead

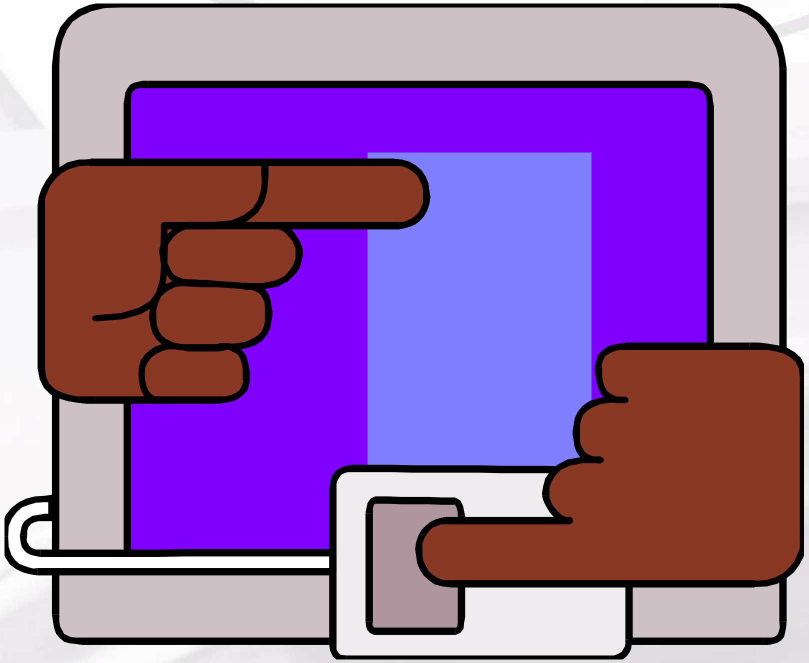
Could be viewed as six simple test cases or one complex test case

Test Matrix

Test No.	Input Cust. No.	Type	Active	Expected Results Cust. Name	Actual
1	*C123*	*10*	*A*	*Jones, John P.*	
2	*C124*	*10*	*A*	*not found*	

Low overhead

Other Common Test Case Formats



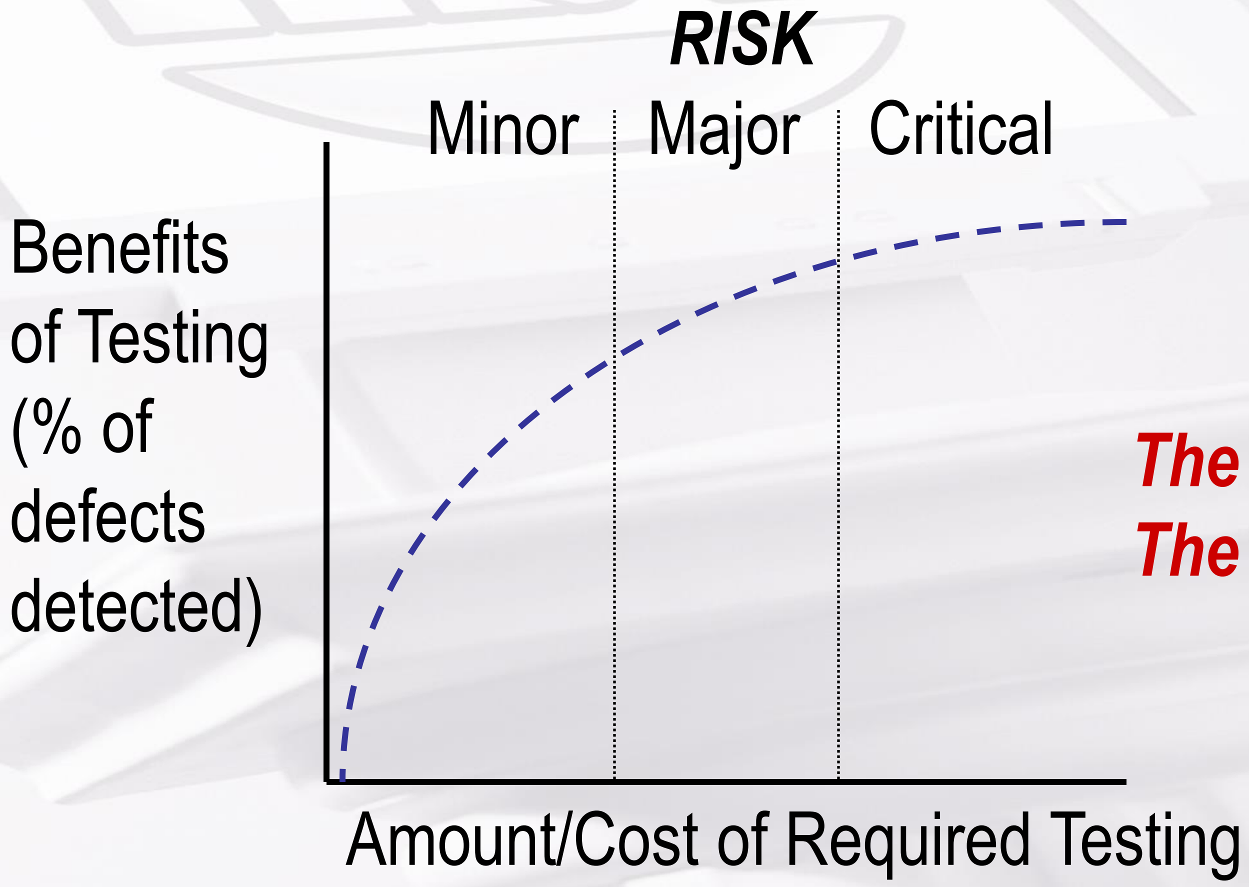
- Screen images
- Data file layouts
- Printed reports
- Source documents
- Input files/transmissions

Other test case dimensions:

- *Load, duration, frequency*
- *Sequence, concurrency*
- *Configuration, background*

Creating the test situation/inputs/conditions is often a challenge

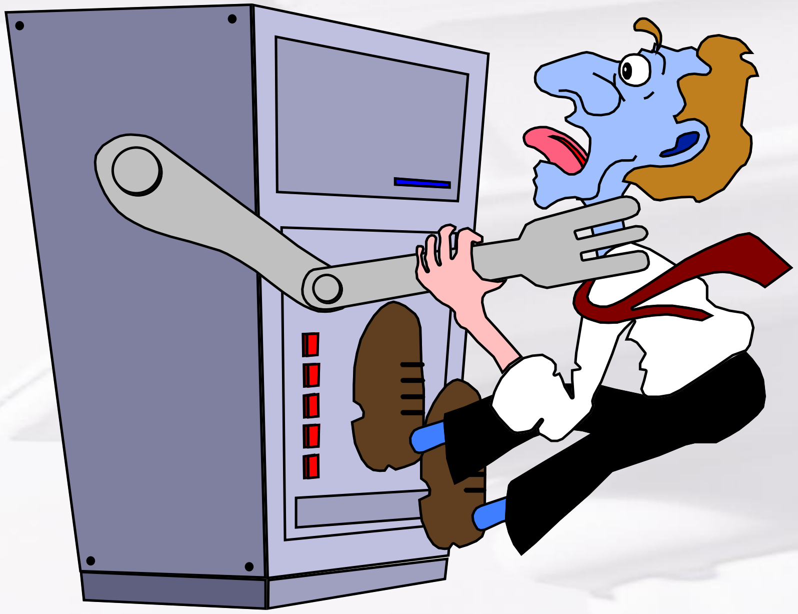
Testing Is Our Main Method of Reducing Risks



The more risk,
The more testing

Any Issues with Typical Risk Approach?

- Create test cases
- Analyze and prioritize risks they address
- Run the higher risk ones



Say you create 100 test cases and have time to run 10 of them.

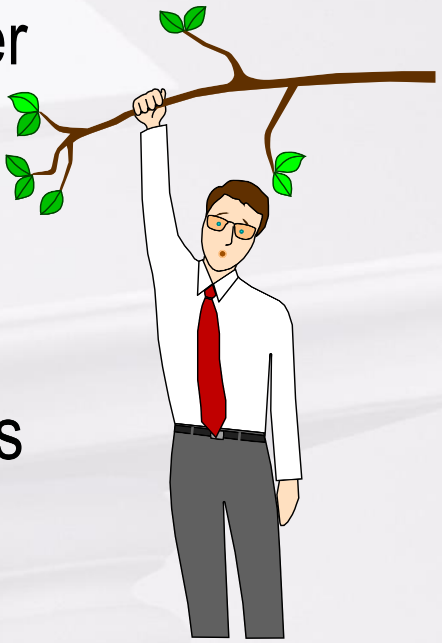
- *What's the value of the time spent on the other 90 that you don't run?*
- *Where did you prioritize the other test cases you didn't think of?*
- ***Were these 100 test cases even testing the most important things?***

2. Planning and Designing the Best Tests

Using limited test execution time and resources most effectively

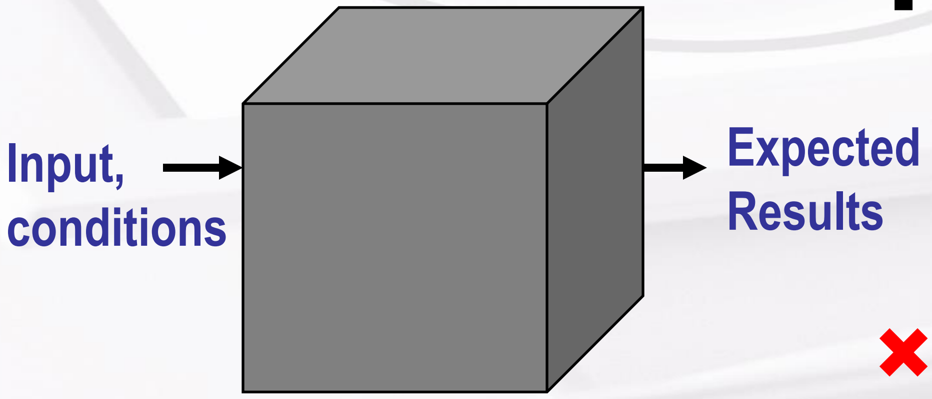
We Can Analyze Risk at Each Level of Test Planning/Design and Execution

- Traditional reactive approach rates features and components in the design and tests the riskier ones **more**
- Business impact and management factors indicate amount, not which, testing needed
- Proactive Testing™ anticipates and prioritizes by levels *too* so we test the highest risks
  - **More** intensely, more focused, more CAT-Scans
  - **Earlier**, builds strategy



Special Proactive Testing™ risk techniques spot missed requirements

Requirements-Based Tests Are Black Box (Functional) and Risk-Based

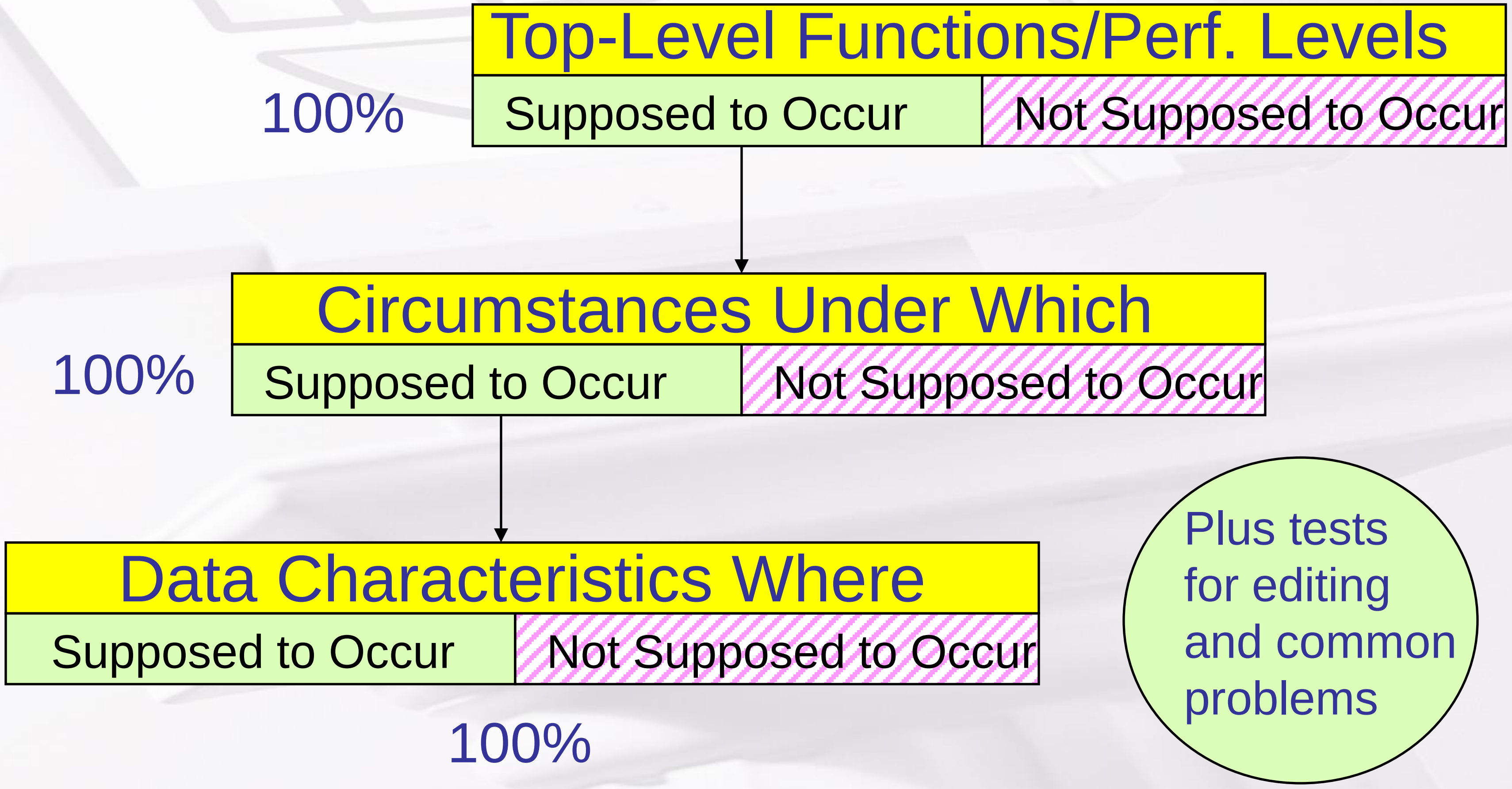


- ✚ Strengths
  - ✚ Intuitive
  - ✚ **Most important, relates to what software is supposed to do**
- ✖ Weaknesses
  - ✖ No single, certain right test
  - ✖ **Often testing design, not reqs**
  - ✖ Illusory sense of adequacy
    - ✖ Often done ad hoc, inefficiently
    - ✖ Overlooks things, input-driven

Many techniques apply →

Confirms each requirement is met

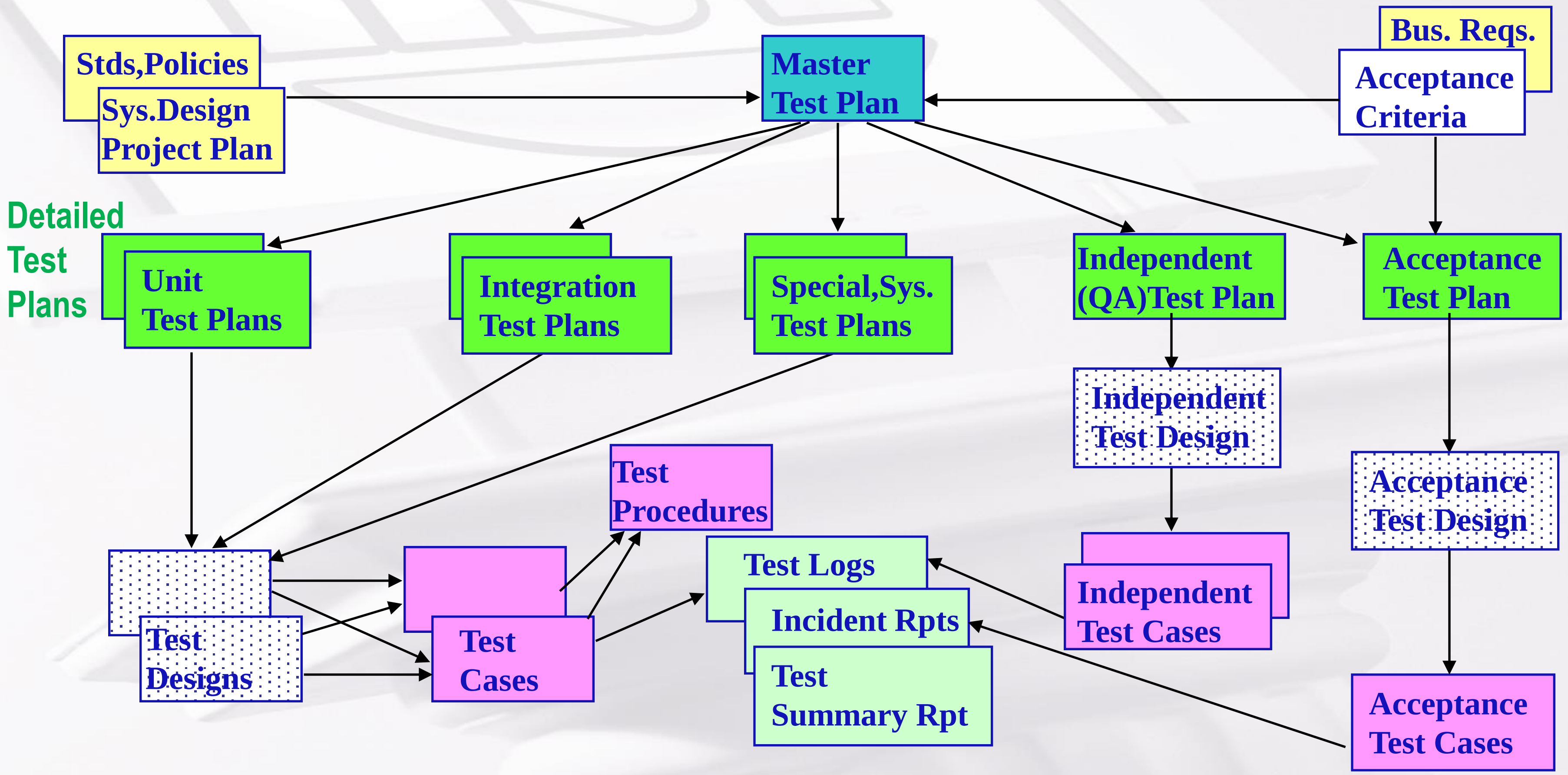
Structured Approach to Designing Functional Tests



Testware--Test (Plan) Documentation per ANSI/IEEE Std. 829-2008

- Controversial standard
- Frequently interpreted as mandating lots of documentation apparently for its own sake
  - Instead, view it as a way to organize thinking
  - Write just enough to be helpful, but no less
- Prior version hard to read, no diagrams
  - My diagram, phrase not in standard but fit it

Testware--Test (Plan) Documentation per ANSI/IEEE Std. 829-2008



What must we demonstrate to be confident it works?

Test Plans

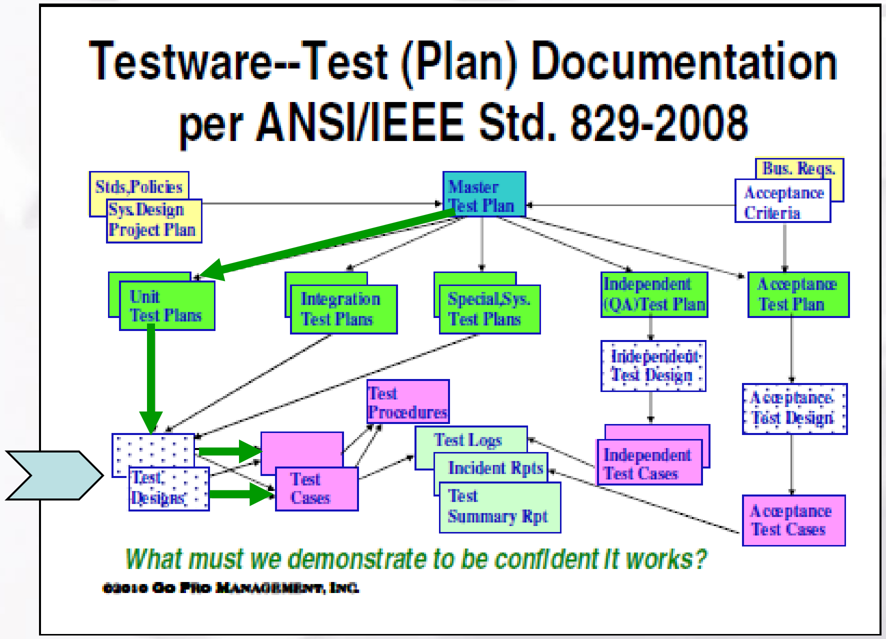
- Project plans for the Testing (sub) Project
- Objectives, strategies, guidelines, standards
- Identify testing tasks, resources, effort, duration →schedule, budget Based on:
  - The set of tests to demonstrate (detailed test plans in Master Test Plan, test design specifications in Detailed Test Plans)
  - Test support, environment, hardware, software, facilities
  - Ancillary and administrative activities

Test Design	Test Case	Test Procedure
• Identifies a set (list) of test cases (specifications) that taken together demonstrate the feature, function, or capability works • Can be reusable or application-specific	• Input/condition and expected result • What is executed • Specification (in natural language) and data values (which actually are input and expected) • Can be reusable, especially specification	• Step-by-step instructions for executing test cases • Includes set-up, establishing pre-conditions • Can get to keystroke level • Often embeds input and expected result data values, which increases maintenance difficulty
One →	***Many***	← ***One***

Testing Structure's Advantages 1 of 3

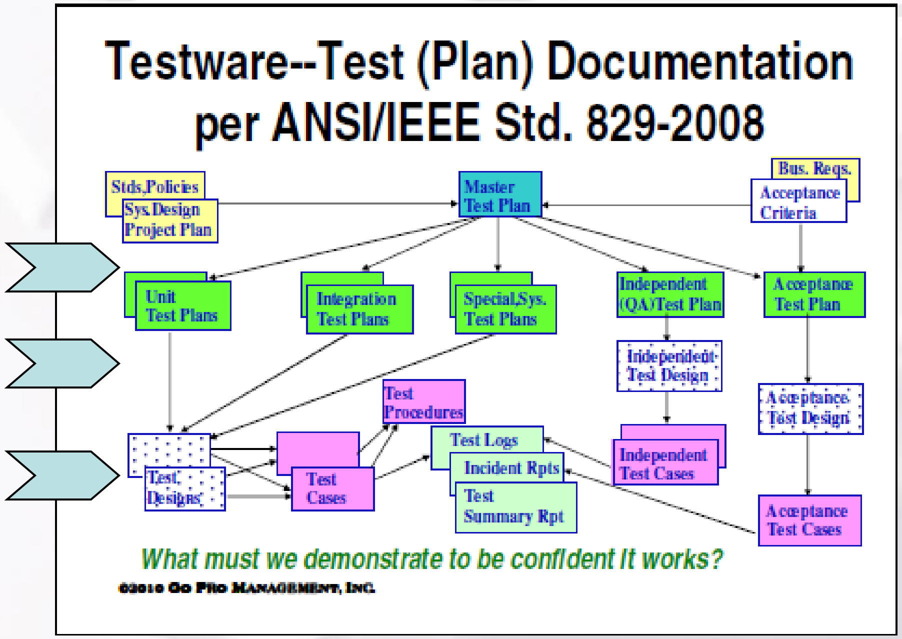
Reactive and Proactive

Testware--Test (Plan) Documentation per ANSI/IEEE Std. 829-2008



- ✓ Systematically decompose large risks into smaller, more manageable pieces
- ✓ Organize and manage large set of test cases
- ✓ Facilitate thorough test data recreation

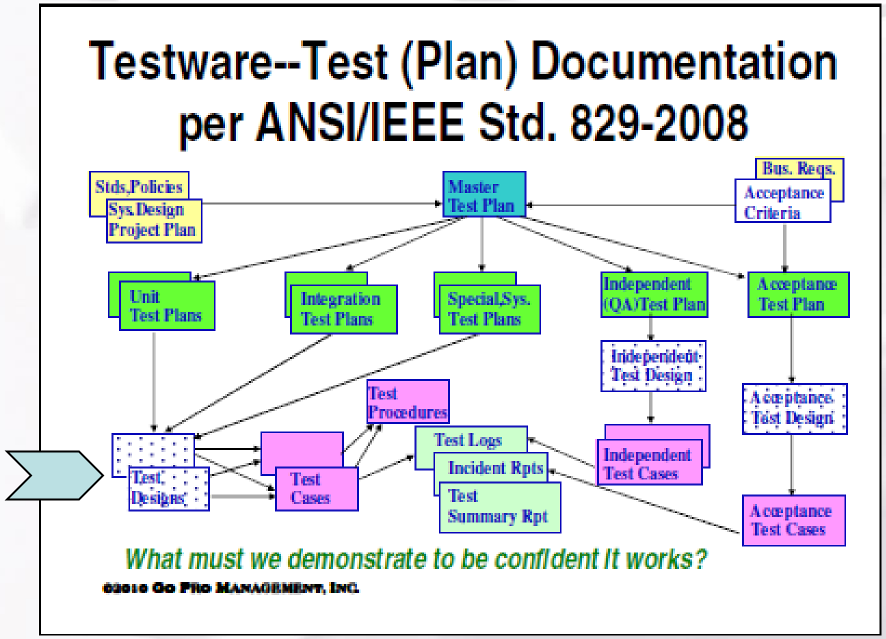
Testing Structure’s Advantages 2 of 3



Proactive

- ✓ Show the choices for meaningful prioritization
- ✓ Use powerful Proactive Testing™ techniques to spot ordinarily-overlooked risks
- ✓ Test the biggest risks more thoroughly
  - ✓ And ***earlier***
  - ✓ Focus first on larger issues, drill down later to detail

Testing Structure’s Advantages 3 of 3



Proactive

- ✓ Facilitate reuse
  - ✓ Where to find
  - ✓ Where to put
  - ✓ How to make reusable
- ✓ Test cases, typically for regression tests ***Reactive***
- ✓ Test design specifications
  - ✓ Higher leverage

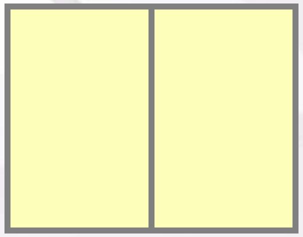
Keys to Effective Testing

➢ Define Correctness Independently of Actual Results	➢ You Must Know What the "Right Answer" Is
➢ Follow Independent Guidelines to Avoid Overlooking Things	➢ Systematically Compare Actual to Expected Results

Test Input	Actual Results	Expected Results
Cust. #123	John P. Jones	
New Cust's name,address	Redisplays screen with fields cleared	
10 Widgets	$14.99	

It is Impossible to Test All Inputs, So Testing Involves Sampling

Enter State Abbreviation



How many possible inputs?

Pick the tests that find the most
with the least time and cost

Might writing them down be a good idea?

Importance of Writing Test Plans/Designs as Part of Design (Before Coding)

- Written plans/designs are repeatable and refinable
- Time to write is same, but pre-coding adds value
  - More thorough, what should be, not just what is
  - More efficient
    - Less delay
    - Fewer interruptions
  - Actually cuts the developer's time and effort
    - Test planning/design is one of 15+ Proactive ways to test the design
    - Prevents rework

The Real Coding Process

Simple Spec:

Operator enters customer number

Computer looks up and displays customer name

Could it be coded wrong? What effect?

Test Planning/Design Can Reduce Development Time and Effort

Proactive

Simple Spec:
Operator enters customer number **C123** Computer looks up and displays customer name **Jones, John P.**

Rework ≅ ~~40~~% of Developer Time

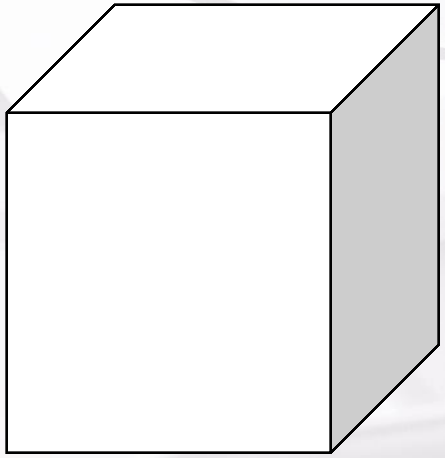
WIIFM

Could it be coded wrong? What effect?

Testing Special Conditions

- Quality factor testing
  - Don’t test them directly
  - Test by creating conditions that would cause the quality factor not to be met
- Database contents and structure testing
  - Can test somewhat using the application…
  - Need tool to look physically at contents and
  - Confirm with independent retrieval

White Box (Structural) Tests Based on Code



- Demonstrates system as written has been tried out
- Approach developers generally use (informally and unconsciously)
- Can be used at module level and between modules, as well as within a module (code level)
- Test cases generally more complex

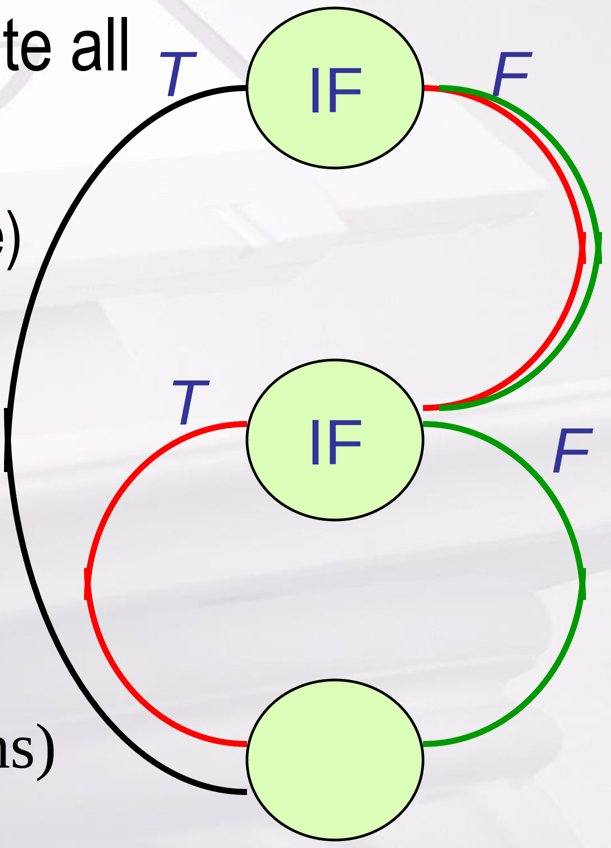
Degrees of Structural Coverage

1. Execute module/routine from entry to exit

   Within a module/routine, execute all

2. Lines of code, steps
3. Branches (basis paths, complete)
4. Logic paths (exhaustive)

- Flowgraph: ***Node***
  - Module's Entry, Exit
  - Where logic branches
  - Junctions (logic returns)
- Flowgraph: ***Edge***
  - all logic between nodes



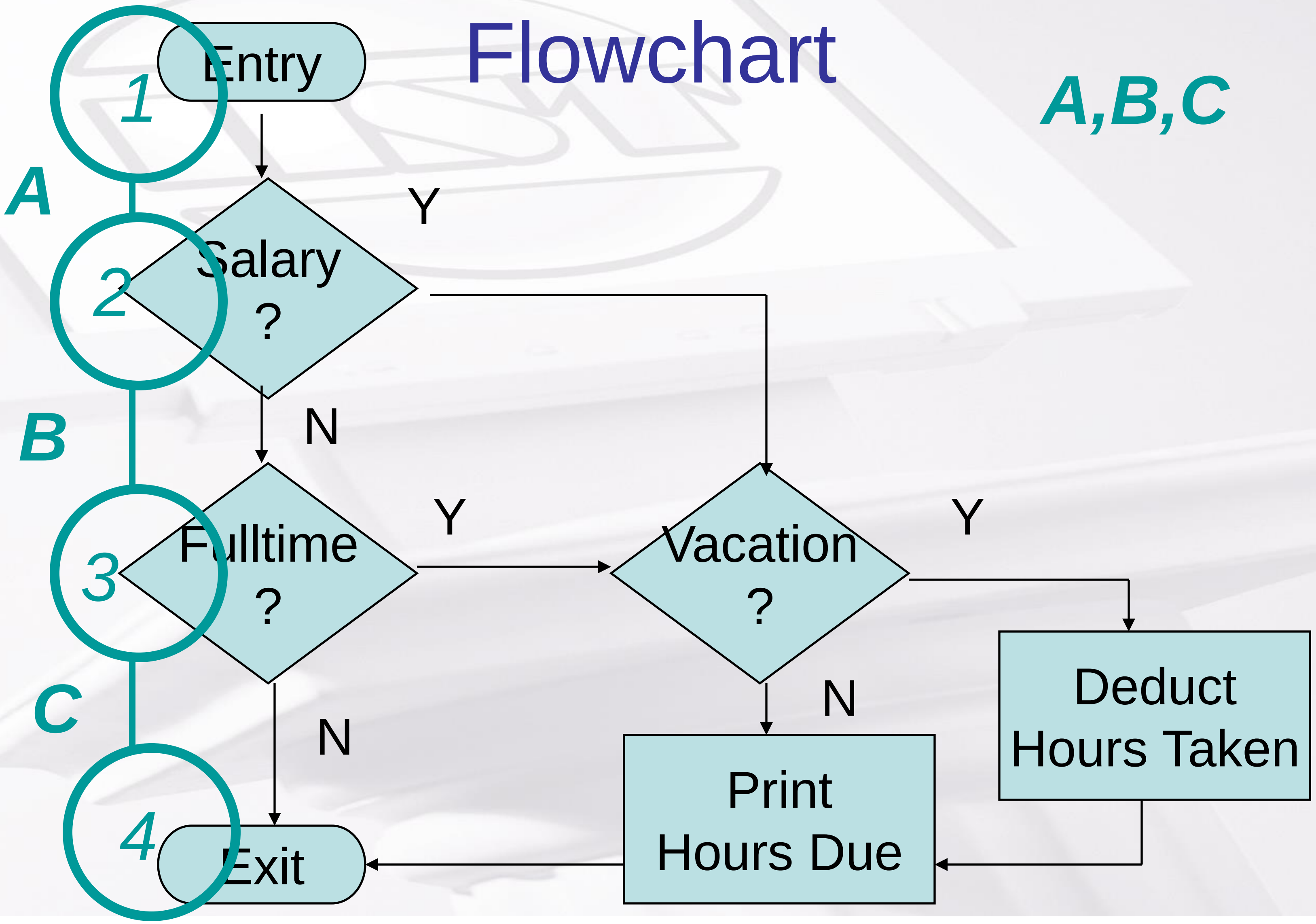
"Flowgraph"

Example: Vacation Routine, Code

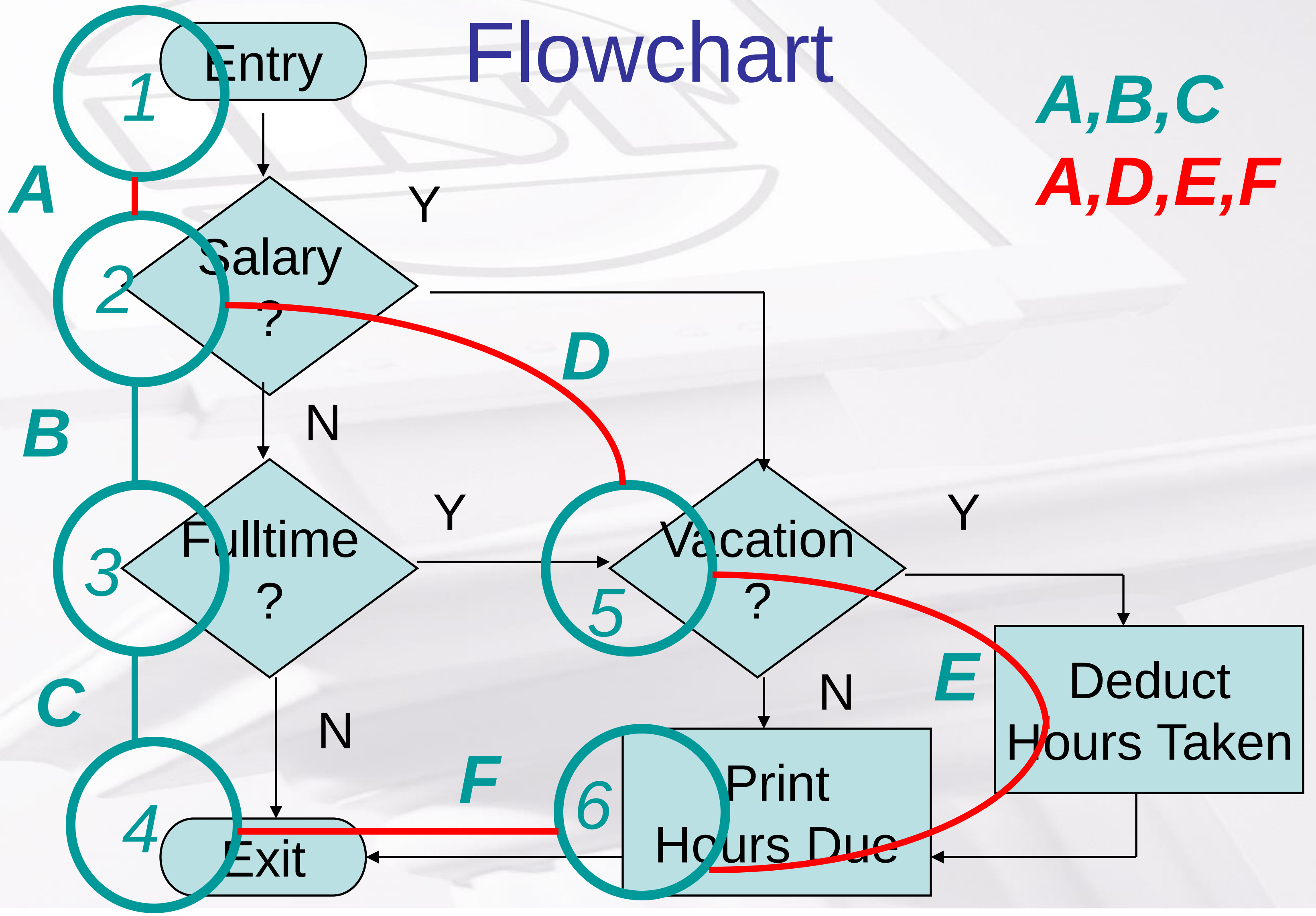
```
VACATION-RTN.
        IF NOT SALARIED
                IF NOT HRLY-FULL-TIME
                        GO TO VACATION-RTN-EXIT.
        IF TRAN-TYPE-VACATION
                SUBTRACT TRAN-HRS FROM VAC-HRS-LEFT.
        MOVE EMP-NAME TO P-EMP-NAME.
        MOVE VAC-HRS-LEFT TO P-VAC-HRS-LEFT.
        WRITE PRINT-REC AFTER ADVANCING 1 LINE.
        ADD 1 TO LINE-COUNT.
VACATION-RTN-EXIT.
        EXIT.
```

Example: Vacation Routine, Flowchart

A,B,C



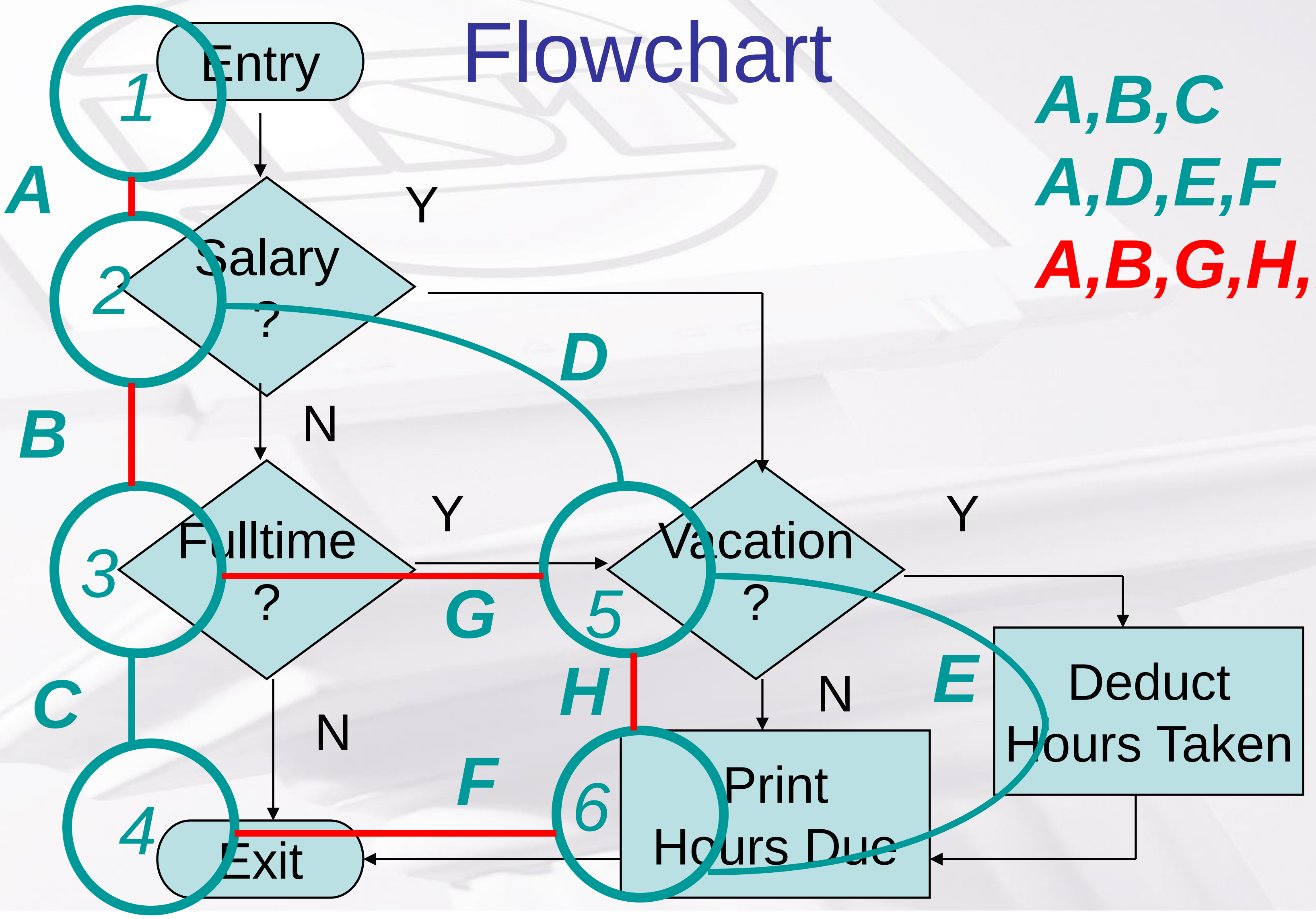
Example: Vacation Routine, Flowchart



A,B,C

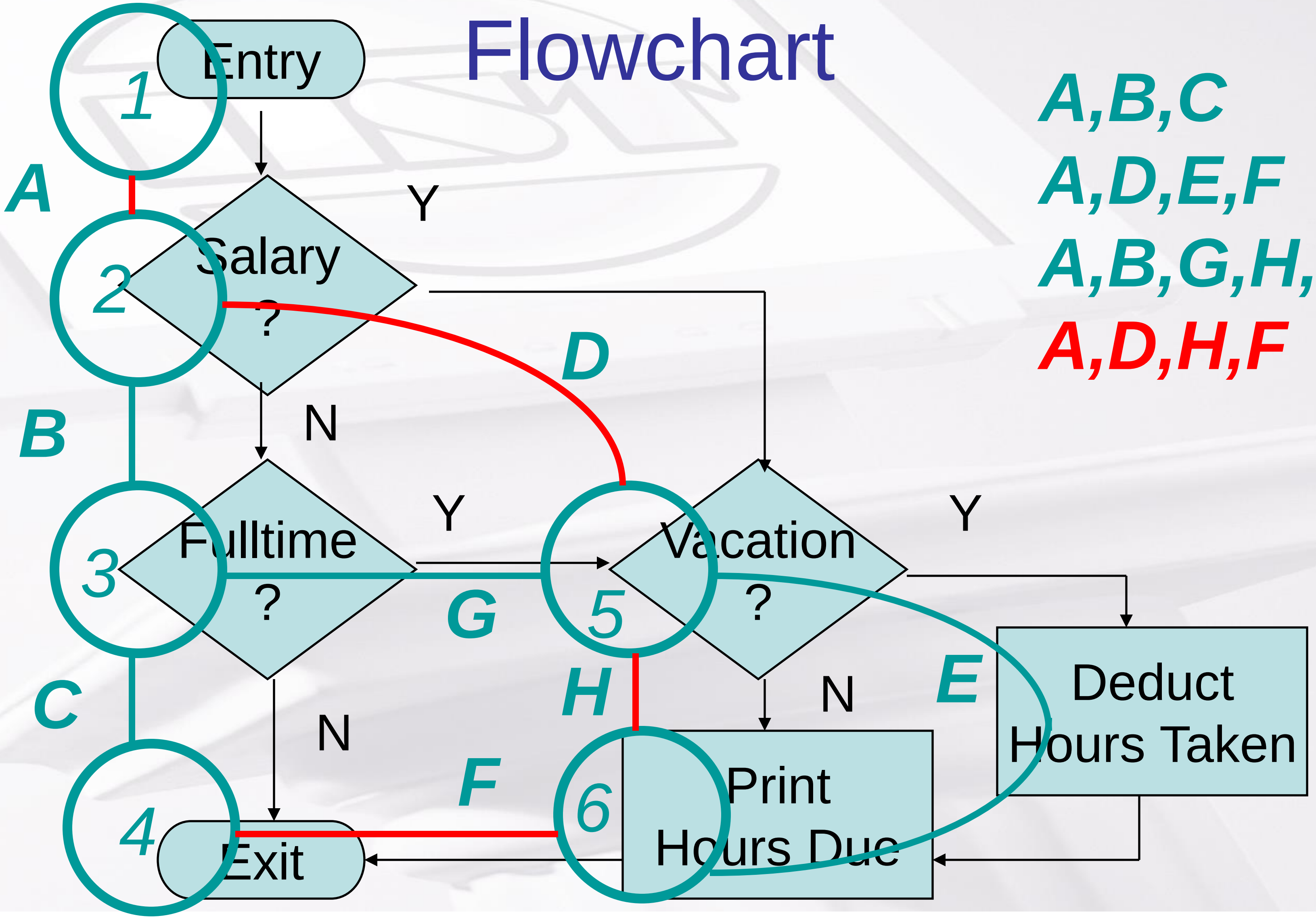
A,D,E,F

Example: Vacation Routine, Flowchart



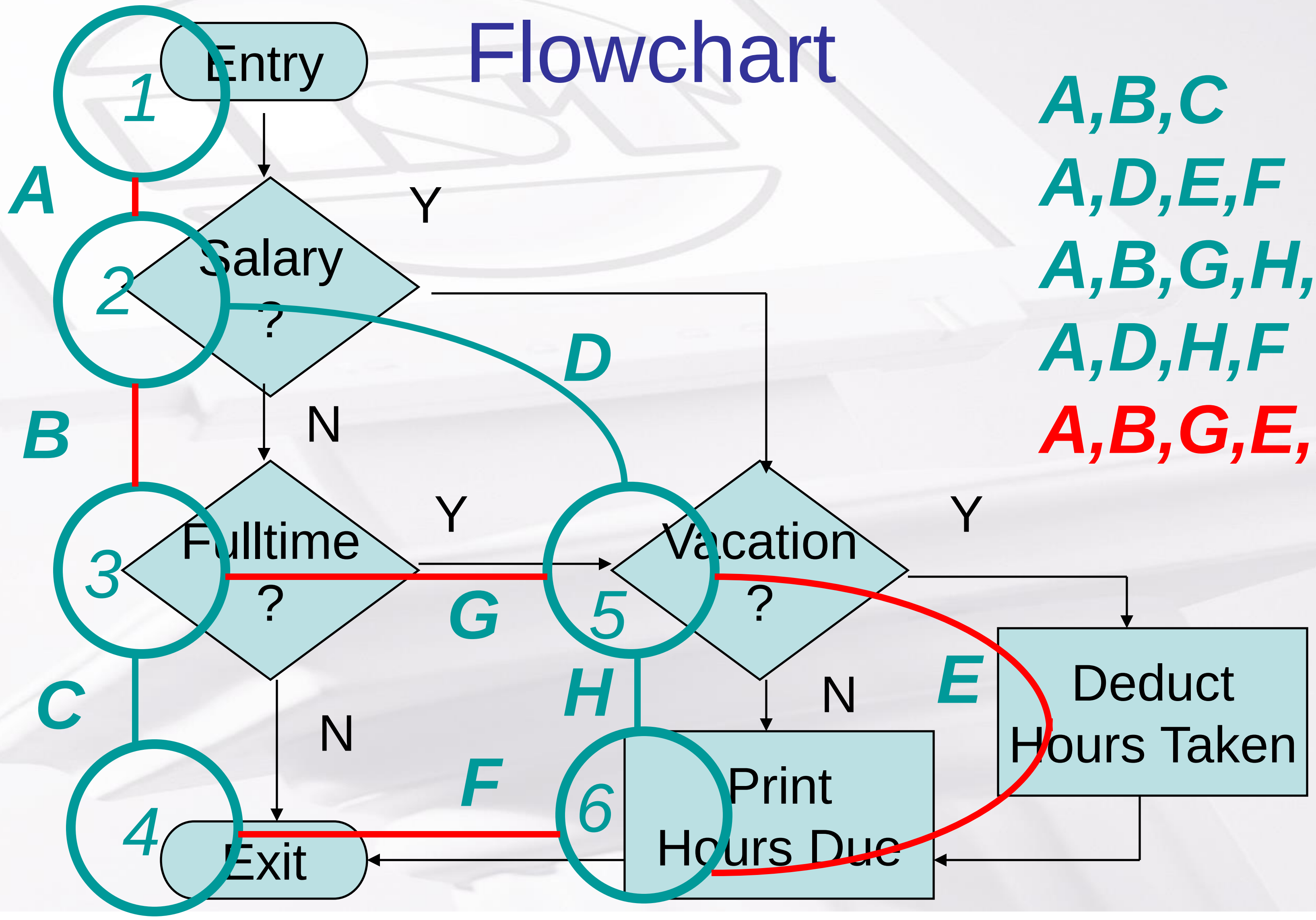
A,B,C
A,D,E,F
A,B,G,H,F

Example: Vacation Routine, Flowchart



A,B,C
A,D,E,F
A,B,G,H,F
A,D,H,F

Example: Vacation Routine, Flowchart



A,B,C
A,D,E,F
A,B,G,H,F
A,D,H,F
A,B,G,E,F

Use Cases, Usually Are Defined as Requirements, also Are Test Cases

Defined as "How an actor interacts with the system." The *actor* is usually the user, and the *system* is what developers expect to be programmed. Therefore, use cases really are white box/design rather than black box/business requirements.

Pause and Flowgraph this Use Case. *Path=Test Case*

U1. Enter customer number	R1.1.	Customer is found (U4)
	R1.2	Customer is not found (U2)
U2. Enter customer name	R2.1	Select customer from list (U4)
	R2.2	Customer is not in list (U3)
U3. Add customer	R3	Customer is added (U4)
U4. Enter order	R4	Order is entered (Exit)

Exercise: Flowgraph of Use Case

Exercise: More Use Case Tests

Write all the different inputs/conditions (specification in words, not with data values) that cause a specified use case path to be executed.

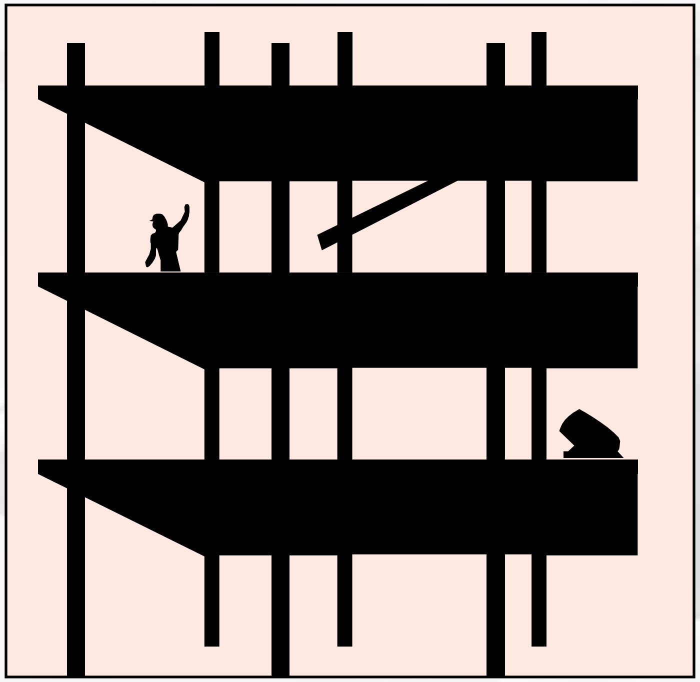
Note: taken together all these inputs/conditions need to be demonstrated to give confidence that the specified use case path works.

Please feel free to pause the course and email your answers to me at Robin_Goldsmith@IIST.org

What does this tell us about the premise that you need only a single test case for each use case scenario (path)?

Quiz 1

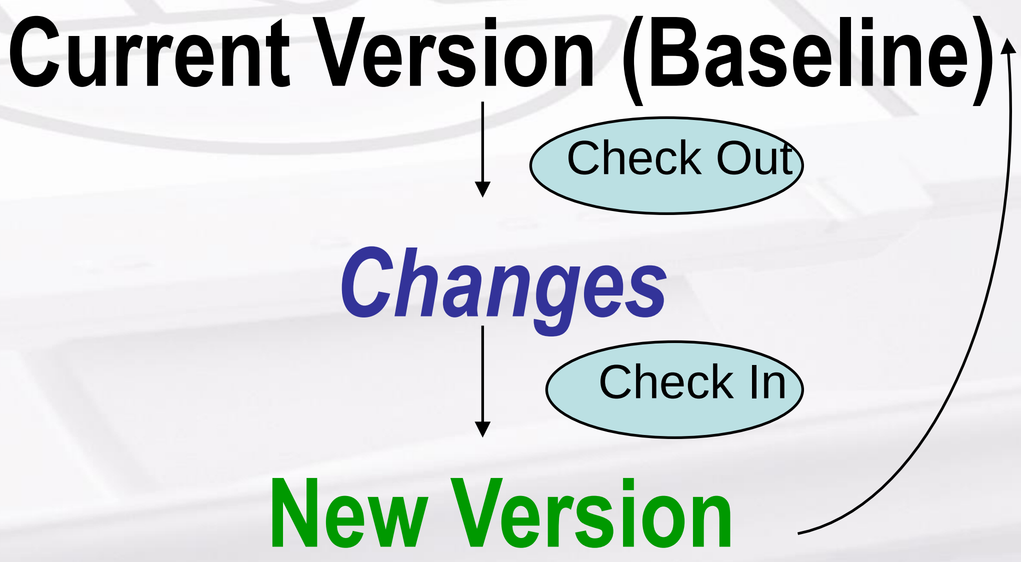
3. Testing Infrastructure--Technical



Establishing and Maintaining the Technical Test Bed/Environment

- Technology inventory
  - Needed vs. received
  - Location, owner, condition
  - Version, features/specs, doc., maintenance source
- Hardware and network/ operating system versions to match expected usage
- Interfacing software apps.
- Testing tools, install & support
- Software under test
  - Turnover procedures
  - Installation, updates
  - Requirements, design, release notes, user doc.
- Test data, sources, and receipt formats/methods, set-up, tear-down, refresh
- Access and usage rules, procedures, measures

Configuration Control/Management Implementation Levels



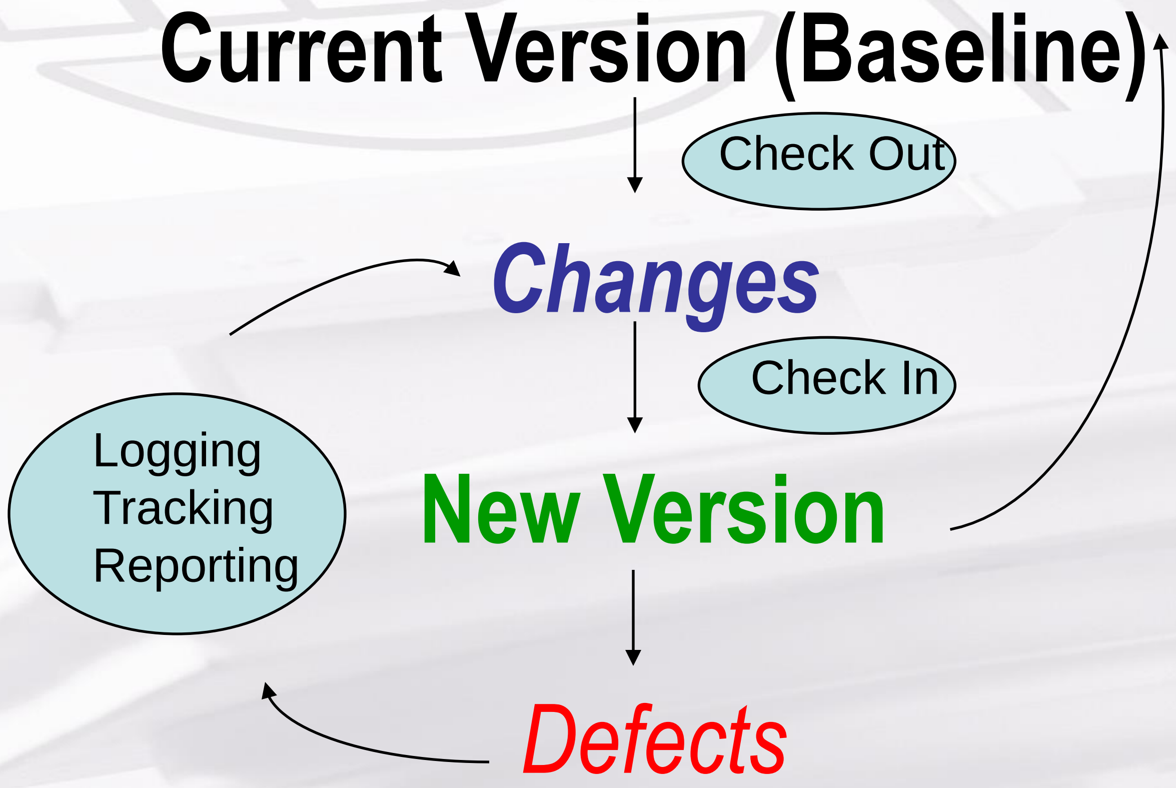
Protected Archived

Change Control Board

Controlled Transfer to Production

0 - None
1 - Source and Executables
2 - Requirements, Design, Documentation
3 - Testware

Configuration Control/Management & Defect Tracking--Automate First



Protected Archived

Change Control Board

Controlled Transfer to Production

Tools for a Managed Environment

Defect Tracking

- Logging, categorization
- Tracking, reporting
  - Assignment to fix
  - Open, closed, status
  - Statistics, trends
- Commercial products
  - Standardized
  - Documented
  - Web access
  - Integrated with C.M. tools

Configuration Management

- Control versions
  - Protect baseline
  - Capture changes
  - Backup and archive
- Issues
  - Ease of use
  - Environments, capacity
  - Features: enforce standards, capture statistics
  - Integrated with Defect tools

Examples of some better known test tools, not intended to be exhaustive or recommendations.

Defect Tracking

PVCS Tracker
Synergex International Corporation
2330 Gold Meadow Way
Gold River, CA 95670
800 366-3472
www.synergex.com

ClearQuest
Rational software from IBM
18880 Homestead Road
Cupertino, CA 95014
800 728-1212 408 720-1600
www.rational.com

JIRA
Atlassian
1098 Harrison Street
San Francisco, CA, 94103
415 701-1110
www.atlassian.com

Configuration Management

PVCS Version Manager
Synergex International Corporation
2330 Gold Meadow Way
Gold River, CA 95670
800 366-3472
www.synergex.com

ClearCase
Rational software from IBM
18880 Homestead Road
Cupertino, CA 95014
800 728-1212 408 720-1600
www.rational.com

Visual SourceSafe
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052-6399-
www.msdn.microsoft.com

Tools for Test Coverage Analysis

Static Analyzers

- Scans program/JCL source code
  - Identifies logic paths, flow chart or flowgraph; can match to expected logic
  - Flags trouble spots
- Issues
  - Language, platform
  - Ease of use, report volume and readability

Dynamic Analyzers

- Tracks program execution
  - Shows lines, paths executed
  - Logs number of executions, time per section
  - Measures coverage of test scripts, spots untested code
- Issues
  - Language, platform
  - Usability, reports' readability and interactivity

Examples of some better known test tools, not intended to be exhaustive or recommendations.

Static Analyzers

McCabe QA
9861 Broken Land Parkway
Columbia, MD 21046
800 638-6316 410 995-1075

Logiscope Code Checker
Verilog
3010 LBJ Freeway Suite 900
Dallas, TX 75234
800 424-3095 972 241-6595
www.verilogusa.com

CAST APPviewer
CAST Software
415 296-1300
www.castsoftware.com

Dynamic Analyzers

McCabe Test
9861 Broken Land Parkway
Columbia, MD 21046
800 638-6316 410 995-1075

Cantata (C and C++)
Quality Checked Software, Ltd. (US distributor
PO Box 6656
Beaverton, OR 97007-0656
503 645-5610
www.qcsltd.com

Software Testing Toolkit
IntegiSoft, Inc.
44 Airport Parkway
San Jose, CA 95110
800 867-8912 408 441-0200
www.integrisoft.com

Tools to Assist Test Execution

Debuggers

- General purpose probe, show program execution
  - Line by line
  - Stop at checkpoints
  - Display variable values
  - Modify variables online
- Most included with language/compiler; also stand-alone tools, e.g., for CICS, SQL

Memory Checkers

- Spot memory leaks and aid garbage collection
- Detect allowable language usage that produces undesirable effects
- Language, platform specific

Examples of some better known test tools, not intended to be exhaustive or recommendations.

Debuggers

DevPartner Studio
Borland (Micro Focus)
400 Interstate North Parkway #1050
Atlanta, GA 30339
770 937-7900
www.borland.com

Memory Checkers

Bounds Checker
Borland (Micro Focus)
400 Interstate North Parkway #1050
Atlanta, GA 30339
770 937-7900
www.borland.com

Purify
Rational software from IBM
18880 Homestead Road
Cupertino, CA 95014
800 728-1212   408 720-1600
www.rational.com

Insure++
ParaSoft
2031 S. Myrtle Avenue
Monrovia, CA 91016
888 305-0041 626 305-3036
www.parasoft.com

Tools to Plan, Design Tests

Test Planners

- Guide and capture test plans and test designs
  - Master and detail levels
  - Provide linkages, indexes
- Issues
  - Ease of use
  - Linkage to actual test cases data values

Test Design/Case Generators

- Based on program code
  - Each field, button, menu choice
  - Forces every branch
  - Includes boundary values
- Based on data field attributes, exhaustive input-driven data
- Based on functional specs
  - Highly structured depiction
  - Minimum rigorous test set
  - High overhead learning, using

Examples of some better known test tools, not intended to be exhaustive or recommendations.

Test Planners

T-Plan
Quality Checked Software, Ltd.
PO Box 6656
Beaverton, OR 97007-0656
503 645-5610
www.qcsltd.com

TracQA
DLB Quality Associates, LLC
61 Coach Road
Glastonbury, CT 06033-3237
860 659-2412
www.dlbqa.com

- -

Test Design/Case Generators

JUnit www.JUnit.org

Jtest, C++test, .test
Parasoft
101 E. Huntington Drive.
Monrovia, CA 91016
888 305-0041 www.parasoft.com

BenderRBT (Caliber-RBT, SoftTest)
Bender & Associates
17 Cardinale Lane
Queensbury, NY 12804
518 743-8755
www.benderRBT.com

Certify
WorkSoft, Inc
2008 E. Randol Mill Rd. #102
Arlington, TX 76011
877 961-9487
www.worksoft.com

Tools to Help Administer Tests

Utilities

- General purpose programs that can
  - Create, update test data
  - Observe, analyze, report test results and abends
  - Monitor system/network internals; show resource usage, performance
- May require assistance of technical specialists

Requirements Managers

- Captures and tracks itemized requirements
  - Cross-references to design and tests
  - Provides back and forth indicators of change impacts
- Issues
  - Document or database based
  - Interfaces with modeling/code generation and test execution tools

Examples of some better known test tools, not intended to be exhaustive or recommendations.

Utilities	**Requirements Managers**
FILE AID	Requisite Pro
Compuware	Rational software from IBM
1 Campus Martius	18880 Homestead Road
Detroit, MI 48226	Cupertino, CA 95014
313 227-7300	800 728-1212 408 720-1600
www.compuware.com	www.rational.com
ABEND AID	DOORS
Compuware	Rational software from IBM
1 Campus Martius	9401 Jeronimo Road
Detroit, MI 48226	Irvine, CA 92618
313 227-7300	877 275-4777 949 830-8022
www.compuware.com	www.telelogic.com
Easytrieve	Caliber-RM
CA Technologies	Borland (Micro Focus)
One CA Plaza	400 Interstate North Parkway #1050
Islandia, NY 11749	Atlanta, GA 30339
800 225-5224	770 937-7900
www.ca.com/	www.borland.com

Tools to Help Execute Tests

Test Drivers

- Software that
  - Sets up test conditions and loads test files
  - Feeds input (runs tests)
  - (May) capture outputs
- Often done in the code via stubs and scaffolding or as part of another tool
- Essential for testing most embedded code

Data Comparators

- File/database based
  - Compares two files record by record and flags differences
  - Can ignore fields intended to be different, e.g., time/date
  - Limited ability to adjust for insertions/deletions
- Screen (GUI) based
  - Essential to detect pixel level
  - Need both bitmap and script

Examples of some better known test tools, not intended to be exhaustive or recommendations.

Test Drivers, Unit Testers

JsTestDriver

testQuest (simulates input devices)
TestQuest, Inc.
7566 Market Place Drive
Minneapolis, MN 55344
952 936-7887
www.testquest.com

Jtest
Parasoft Corp.
101 E. Huntington Drive
Monrovia, CA 91016
888 305-0041
www.parasoft.com

Data Comparators

Comparex
SERENA Software, Inc.
11130 Sunrise Valley Drive #140
Reston, VA 20191
800-457-3736
www.serena.com
- - - - - - - - - -

Test Mentor for Visual Age Java
SilverMark Incorporated
5511 Capital Center Drive Suite 510
Raleigh, NC 27606
888 588-0668 919 858-8300
www.silvermark.com

Vectorcast (for embedded software)
Vector Software, Inc.
1351 South County Trail #310
East Greenwich, RI 02818
401.398.7185
www.vectorcast.com

Test Execution, Load Testing Tools

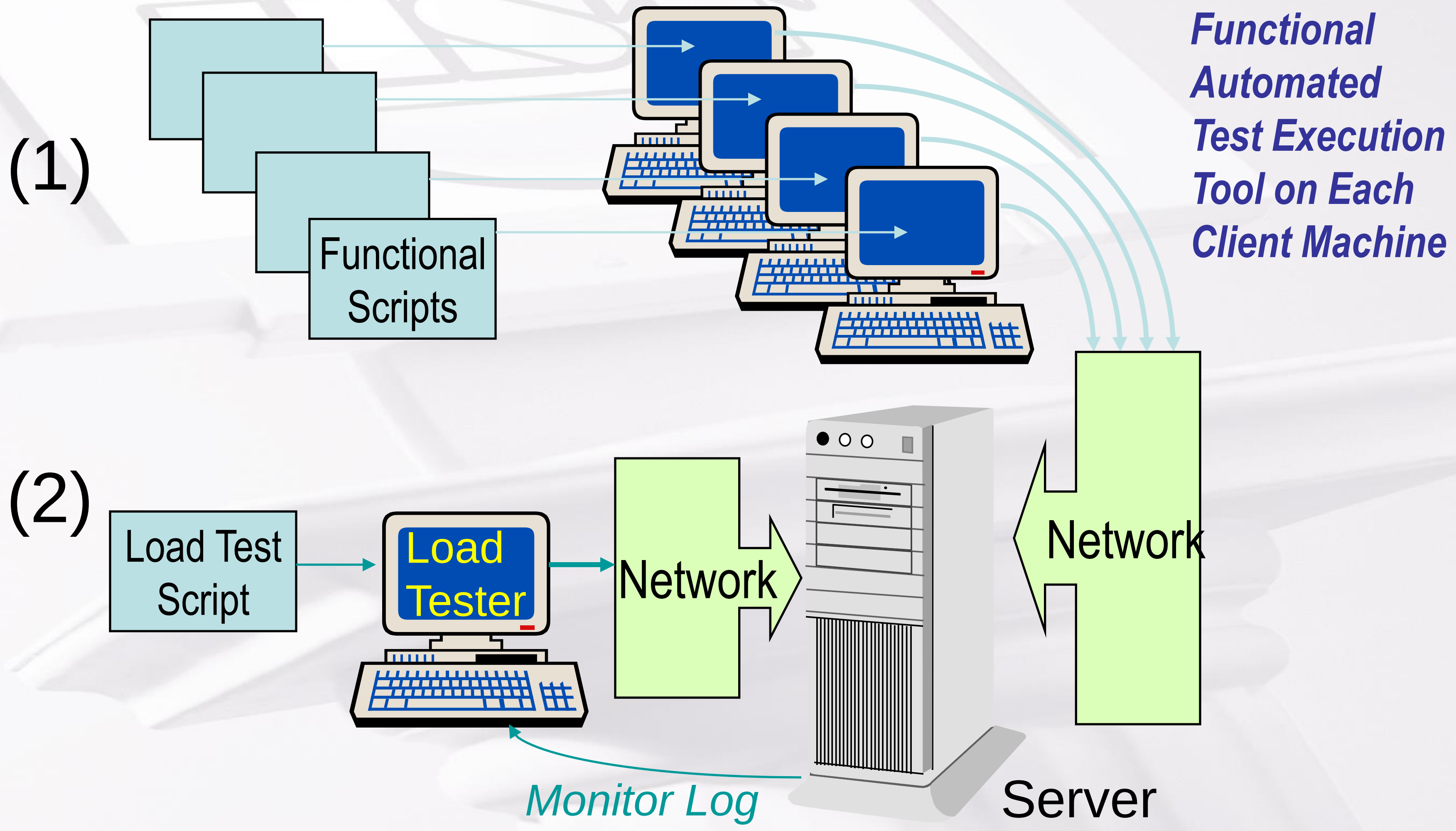
Mainframe

- Character-based X-Y input, result definition
- Feeds input into data fields, captures output; can generate “loads;” compares to saved “expected results”
- Expensive, cumbersome, and time-consuming to keep tests up-to-date

GUI Capture/Playback

- Keyboard/mouse capture and/or coding into script
  - Adjusts for scaling, location
  - Playback, match unattended
  - Integrate with defect tracker and test case manager
- Issues
  - Platform, tool suite, overhead
  - Scripting, ease of use, bugs
  - Non-standard controls , web

Automated Load Testing Strategies



Examples of some better known test tools, not intended to be exhaustive or recommendations.

Load Testing

LoadRunner
HP (Mercury Interactive Corp.)
1325 Boegas Avenue
Sunnyvale, CA 94089
800 TEST-911 408 822-5200
www.merc-int.com

Silk Performer
MicroFocus (Borland, Segue Software)
400 Interstate North Parkway #1050
Atlanta, GA 30339
877 772-4450
www.borland.com

NeoLoad
Neotys USA, Inc.
203 Crescent Street
Building 25, Suite 106
Waltham, MA 02453
781 899-7200
www.neotys.com

Test Management Tools, Services

Test Managers

- Integrated with GUI capture/playback suites
- Keep track of scripts
- Can incorporate manual test descriptions too
- String scripts together for unattended execution
- Integrate defect tracking and test status reporting

Performance Monitoring

- For Web sites, usually a service, but can do internally
- Can integrate with your load tests initiated around the world at different times
- Ongoing, sample your web site with typical transactions periodically to track performance over time, detect patterns of problems

Examples of some better known test tools, not intended to be exhaustive or recommendations.

<u>Test Managers</u>

Quality Center (Test Director)
Hewlett-Packard (Mercury Interactive)
1325 Boegas Avenue
Sunnyvale, CA 94089
800 TEST-911 408 822-5200
www.merc-int.com

Test Manager
Rational software from IBM
18880 Homestead Road
Cupertino, CA 95014
800 728-1212 408 720-1600
www.rational.com

Silk Central
MicroFocus (Borland, Segue Software)
400 Interstate North Parkway #1050
Atlanta, GA 30339
877 772-4450
www.borland.com

<u>Performance Monitors</u>

Active Test
Hewlett-Packard (Mercury Interactive Corp.)
1325 Boegas Avenue
Sunnyvale, CA 94089
800 TEST-911 408 822-5200
www.merc-int.com

KeyReadiness
Keynote Systems, Inc.
777 Mariners Island Blvd.
San Mateo, CA 94404
800 KEYNOTE 650 403-3458
www.keynote.com

ResponseWatch
Atesto Technologies, Inc.
39355 California Street, Suite 305
Fremont, CA 94538
510 405-5900
www.atesto.com

Examples of some better known test tools, not intended to be exhaustive or recommendations.

Mainframe Full Test Managers

Hiperstation
Compuware Corporation
1 Campus Martius
Detroit, MI 48226
313 227-7300
www.compuware.com

Verify
Computer Associates
One Computer Associates Plaza
Islandia, NY 11788
516 342-5224
www.cai.com
- - - - - - - - - - - - -
QAComplete
SmartBear Software
100 Cummings Center, Suite 234N
Beverly, MA 01915
978-236-7900
www.smartbear.com

GUI Capture/Playback, Load Testers

QuickTest Pro, Winrunner, Xrunner, LoadRunner
Hewlett-Packard (Mercury Interactive Corp.)
1325 Boegas Avenue
Sunnyvale, CA 94089
800 TEST-911 408 822-5200
www.merc-int.com

SQA Robot, Team Test, Rational Suite TestStudio
Rational software from IBM
18880 Homestead Road
Cupertino, CA 95014
800 728-1212 408 720-1600
www.rational.com

Silk Test, QA Partner, SilkTest Performer
MicroFocus (Borland, Segue Software)
400 Interstate North Parkway #1050
Atlanta, GA 30339
877 772-4450
www.borland.com

Example Linear Script

```
select window "Logon"
enter text "username", "administrator"
enter text "password", "testonly"
push button "Ok"
select window "Main"
push button "New Customer"
expect window "Customer Information"
select field "Name"
type "Jones"
select field "First Name"
type "John"
select field "Address"
type "54321 Space Drive"
    .
    .
```

Example Structured Scripting

```
Function EnterCustomerJones
        Logon
        Press "New Customer"
        Enter Field "Name", "Jones"
        Enter Field "First Name", "John"
        Enter Field "Address", "54321 Space Drive"
        . . .
        Logoff
End Function

Function OrderPremiere
        Logon
        Press "New Order"
        Enter Field "Product", "Adobe Premiere"
        Enter Field "Amount", "35"
        Enter Field "Delivery", "asap"
        . . .
        LogOff
End Function

. . .
```

Example Scripted Variables

```
Function EnterCustomerJones
        Logon
        Press "New Customer"
        Enter Field "Name", &Name
        Enter Field "First Name", &FirstName
        Enter Field "Address", &Address
        . . .
        Logoff
End Function

Function OrderPremiere
        Logon
        Press "New Order"
        Enter Field "Product", &Product
        Enter Field "Amount", &Amount
        Enter Field "Delivery", &Delivery
        . . .
        LogOff
End Function

. . .
```

after Hans Buwalda

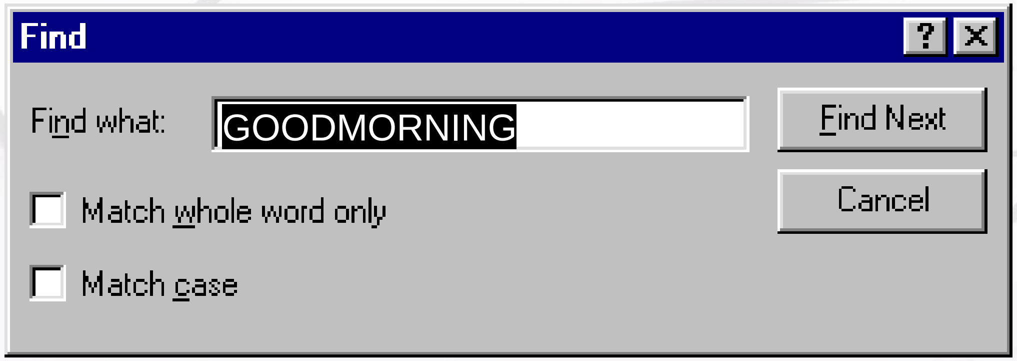
Customers

FirstName	**Name**	**Address**
John	Jones	54321 Space Drive
Mary	Smith	444 Maple Street
Larry	Smyth	21 Main Avenue

NewOrders

Product	**Amount**	**Delivery**
Adobe Premiere	35	asap
MS Word	198	COD
Quicken	95	included

A Data Driven Example



GOODMORNING

- Text file input to the test:

```
goodmorning
GOODMORNING
GoodMorning
GoodMorningVietnam
```

- Searched file with test data:

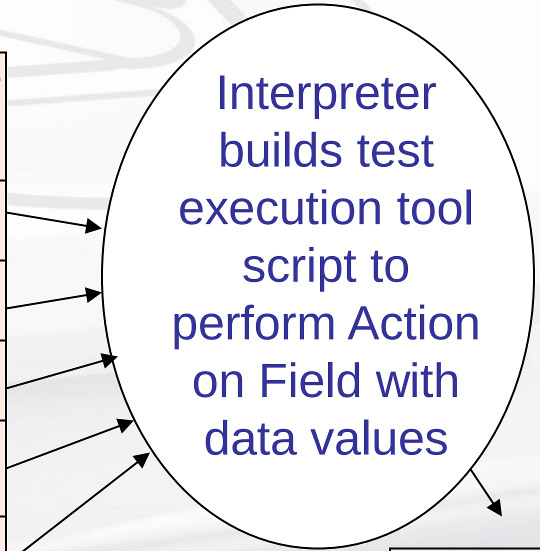
nr	case	whole	pattern	matches
1	off	off	GOODMORNING	4
2	off	on	GOODMORNING	3
3	on	off	GOODMORNING	1
4	on	on	GOODMORNING	1
5	off	off	goodmorning	4
6	off	on	goodmorning	3
7	on	off	goodmorning	1
8	on	on	goodmorning	1
9	off	off	GoodMorning	4
10	off	on	GoodMorning	3
11	on	off	GoodMorning	2
12	on	on	GoodMorning	1

- Data driven script:

```
for each line in the text file do
   open the find dialog
   read a line from the text file
   use the values to fill dialog
   press find button
   check amount of matches
   close the dialog
```

Automating the Automation (Low-Level Field-Based Interpreters)

Action Word	Field	Data Value
Window	*Cust*	
Enter	*ID*	*C123*
Match	*Name*	*John Doe*
Listbox	*Status*	
Select	*Status*	*Active*


Interpreter builds test execution tool script to perform Action on Field with data values

Tool Script:
Confirm window is "Cust"
EnterText, ID, "C123"
MatchText, Name, "John Doe"
ErrorMessage, Name, "not John Doe"
PullDownList, "Status"
SelectListEntry, "Status", "Active"

Most are home-grown.
See WorkSoft's Certify
www.fitnesse.org

Action Based Testing
Originated by Hans Buwalda in 1994

Tester builds a sheet, with Action Words and their data values

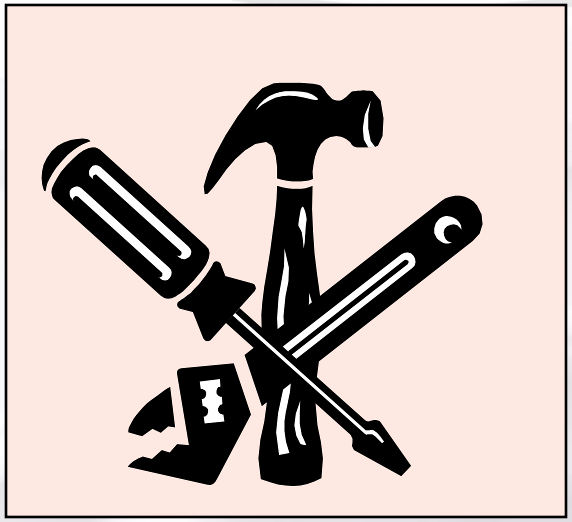
Action Word	*Arguments….*		
Open Account	John	Jones	458973458
Check Account	458973458	Jones, John	$0.00
Deposit	458973458	$214.37	
Check Balance	458973458	$214.37	

Tool reads Action Word and its data values (arguments)

per Action Word's logic,
- places data values into relevant input fields as if entered by operator and/or
- compares expected result data values to actual values in relevant output fields

See ***Integrated Test Design and Automation*** *Using the TestFrame Method*
By Hans Buwalda, Dennis Janssen, Iris Pinkster Addison-Wesley 2002

Key Test Automation Issues



What issues would affect use of a Test Manager tool?

- Tools should be part of a structured, planned test process or they'll lead to ineffective, inefficient tests
- Tools require knowledge
  - Tool operation
  - Underlying test methods
- Automating a test takes 3-20 times as long
- Tools don't replace staff

Exercise: Other Infrastructure

Infrastructure is both **Physical**

Facilities

Technology hardware and software

And **Conceptual**

Processes—rules, actions, beliefs, customs

Skills and knowledge--acquiring and maintaining

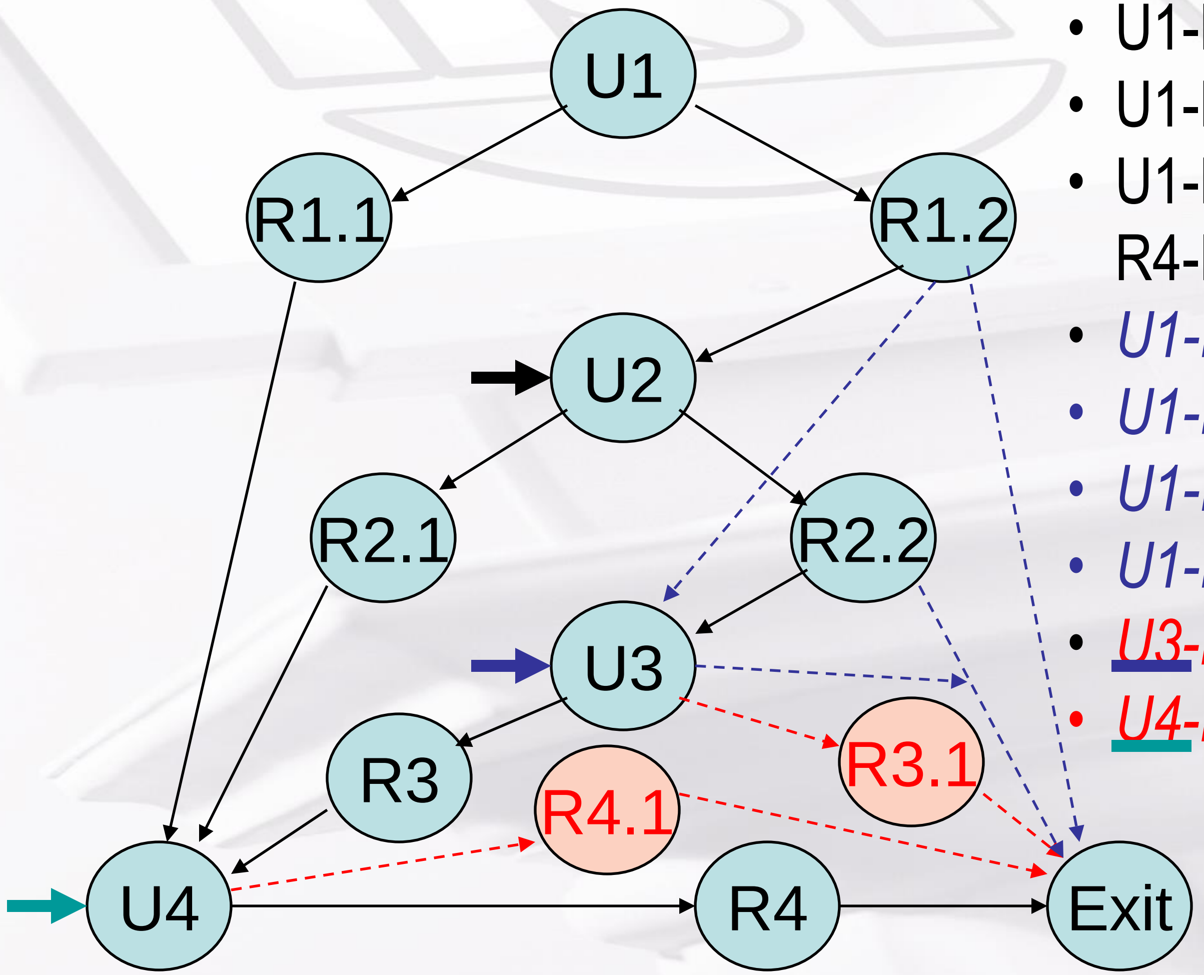
Identify Conceptual Infrastructure that test management needs to address in conjunction with automating test execution. Issues?

Please feel free to pause the course and email your answers to me at Robin_Goldsmith@IIST.org

Exercise: Other Infrastructure

Quiz 2

Exercise: Flowgraph of Use Case



- U1-R1.1-U4-R4-Exit
- U1-R1.2-U2-R2.1-U4-R4-Exit
- U1-R1.2-U2-R2.2-U3-R3-U4-R4-Exit
- *U1-R1.2-U3-R3-U4-R4-Exit*
- *U1-R1.2-Exit*
- *U1-R1.2-U2-R2.2-Exit*
- *U1-R1.2-U2-R2.2-U3-Exit*
- *U3-R3.1-Exit*
- *U4-R4.1-Exit*

4. Isolating and Reporting Defects

Key Defect Detection, Reporting, and Management Issues

- Isolating the bug to be sure
  - It really is a bug
  - Developer can reproduce it
- Describing the bug so it will be addressed
- Categorizing to spot trends
- Tracking to assure it's fixed
- Monitoring/reporting status
- Improving

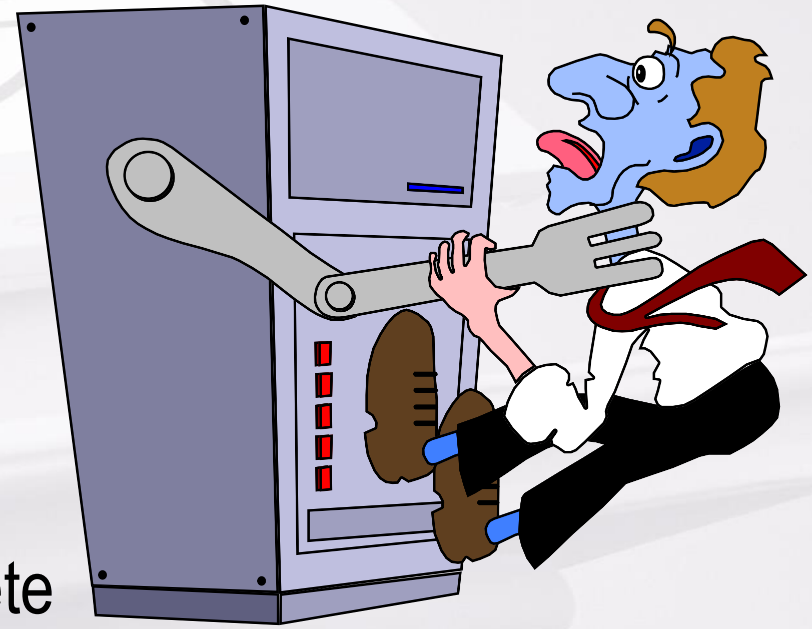
Bug Isolation vs. Debugging-- Tester Must Take Time to Determine

NOT: Identifying internal factors that responded wrong to these external factors

- What the steps were that led to the problem
  - Not an error in the test
  - So it can be reproduced, even if not 100% of time
- Variables (especially in the technical environment) that seem to affect it
  - HW/SW configuration
  - Other actions

Writing an Effective Bug Report: Description that Catches Attention

- What happened, when
- **Impact** on user and/or customer
- Clear, concise, correct



- ➢ Order modification is incomplete
- ➢ Once an item has been entered on the order, going back to it and modifying it can affect the price
- ➢ Customers aren't charged for modified items on orders

Exercise: Bug Description Title

Think of an important bug you have encountered recently. Write down the behaviors that cause you to say it is a bug: what it is supposed to do and what it does instead.

Then, write a one-line description of the bug that would cause managers to consider the bug worth fixing.

How well does your one-line description capture the problem and make the defect important enough to be given priority to be fixed? How improve it?

Please feel free to pause the course and email your answers to me at Robin_Goldsmith@IIST.org

Exercise: Bug Description Title

Steps to Reproduce the Problem (described on slide 4)

- Create an order for a customer
- Add at least one item to the order and go to add another
- Go back to an item that has been entered and modify it (item number, color, size, or quantity)

➢ Extended price => zero

- **Variables I tried which don't seem to matter**
  - First, second, or third item entered
  - Going to previous item by mouse (scroll bar), arrow, or function key
  - Quantity 1, 10, 100
  - Items with unit prices under $10, $10-100, over $100

Exercise: Defect Isolation

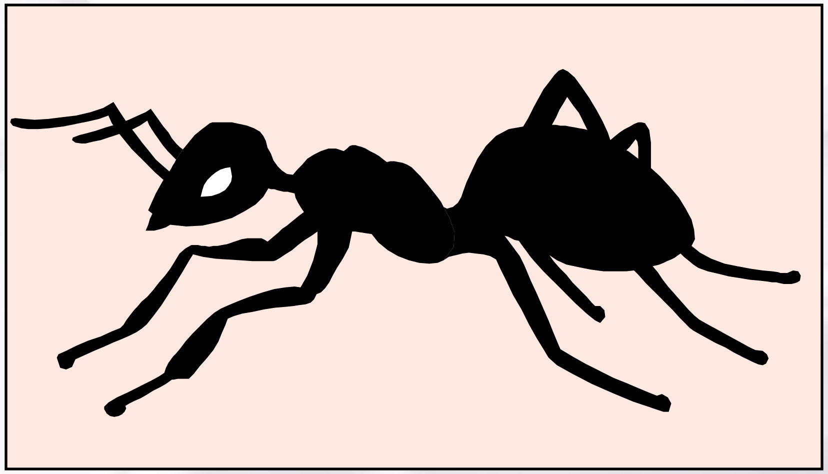
The developer has received your bug report (on previous page) but cannot reproduce the problem. ("It works on my machine.")

Identify additional variables which you could have checked to isolate the problem.

What if you now can't reproduce the problem either?

Please feel free to pause the course and email your answers to me at Robin_Goldsmith@IIST.org

Tracking and Categorizing Defects



"Bugs live in colonies"

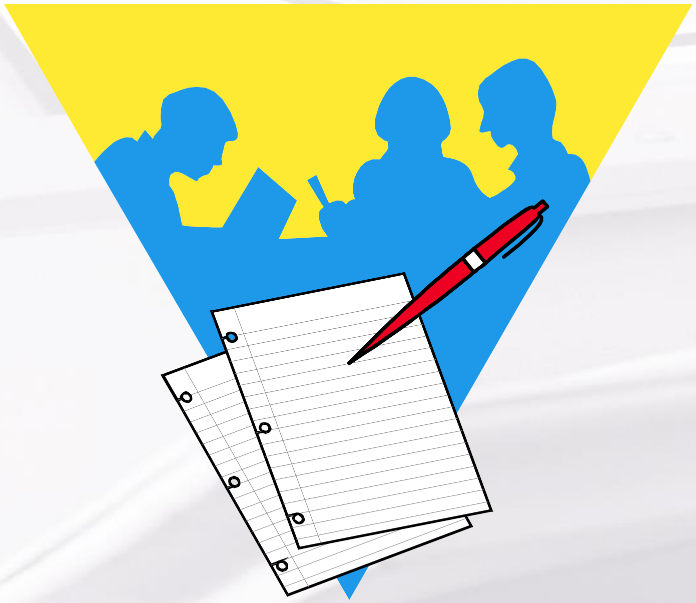
- Track each status change--open, assigned, deferred, fixed, canceled, closed
- Include sufficient specific info to reproduce problem
- Cross-reference to test ID, level revealing the bug
- Capture estimates, actuals
- Identify trends, causes

Common Defect Categories

- ✓ Criticality
  - ✓ Showstopper
  - ✓ Injury, damage
  - ✓ Failure to function
  - ✓ Impeded effectiveness
  - ✓ Cosmetic
- ✓ Nature of problem, e.g.,
  - ✓ Hardware
  - ✓ Communications
  - ✓ User error
- ✓ System, module, screen
- ✓ Symptom, e.g., lockup, miscalculation, no space
- ✓ Impact on business, breadth and depth
- ➢ Cost, effort, risk to fix
- ➢ Priority--likelihood, workaround

➔Age: detection - injection

Defect Categorization Issues



- Who decides each of the above categories?
- Which status data do we need, from whom, and how do we get it in a timely manner?
- How do we identify and account for duplicate defect reports?
- What about "Not a defect, implemented as designed"?

Some Test Measures

- Tests
  - Planned
  - Constructed
  - Executed
  - Passed
  - Failed
  - Blocked
- Test cycles
  - Number, duration, yield
  - Test vs. debug time

- Defect density, errors per
  - KLOC (K=1000, Lines of Code)
  - Function Point
  - Object, class
  - Module
  - Fix
  - Test (errors in the test)
- Mean time to fail
- Mean time to fix

Exercise: Test Status Reports

Identify the contents and other attributes of the asssigned status report which would be most useful to the following stakeholders:

A-C The Test Manager
D-M The Project Manager
N-R The IS Director/VP
S-Z The Customer/User Executive

Please feel free to pause the course and email your answers to me at Robin_Goldsmith@IIST.org

When Testing Is "Good Enough"

When the costs/risks of further testing exceed the risk reduction benefits likely from the further testing

How Most Decide It’s “Good Enough”

- Ad hoc, during testing
  - Claimed “necessary” for market-driven software
  - Criteria tend to shift as deadline approaches
  - Politics & persuasiveness
  - Prominent champions
  - Illusory soundness
  - Self-perpetuating mindset, prevents suitable feedback

Issues?

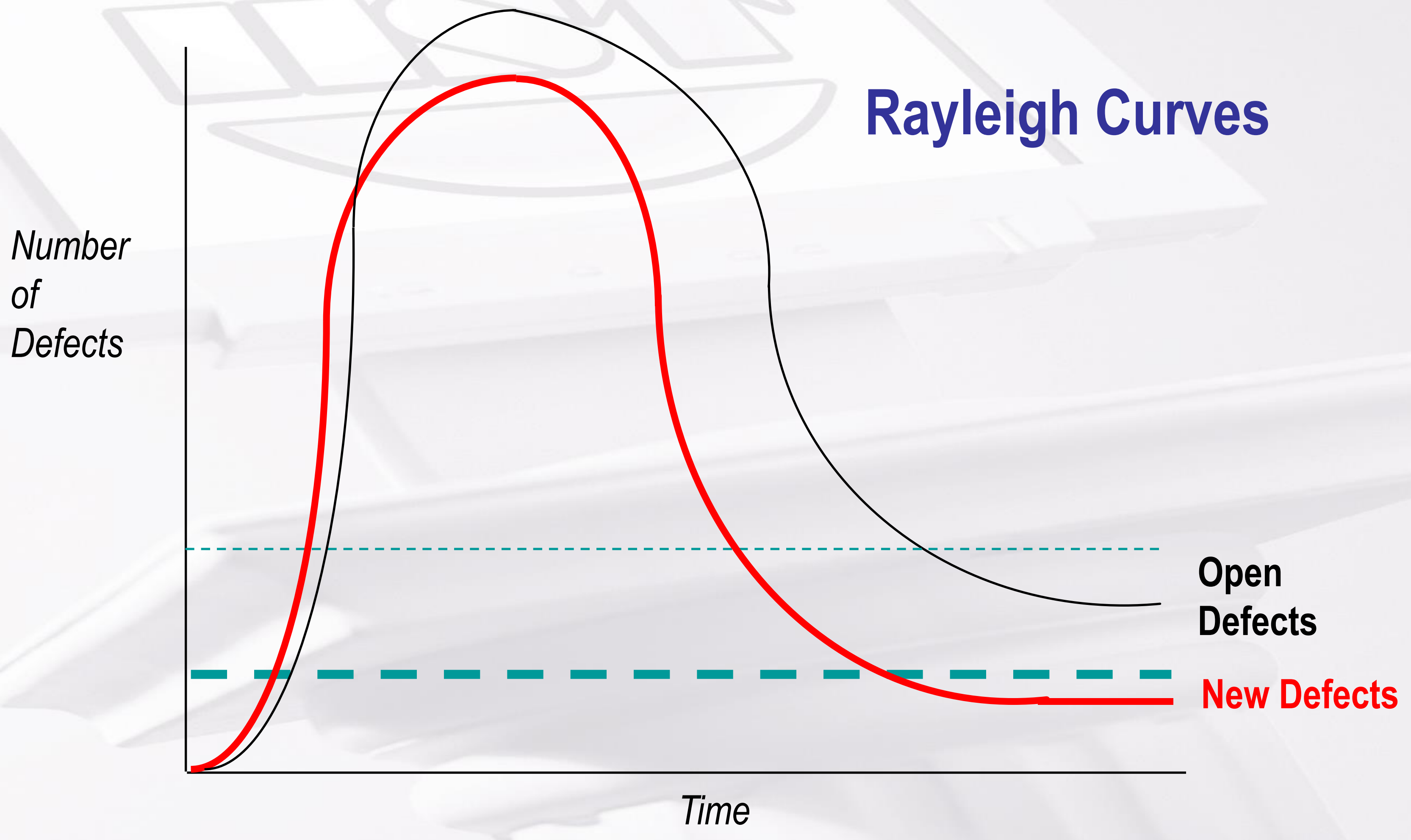
A Better Approach?

- Gain prior agreement
  - Testing to be performed
  - Measures of testing adequacy
  - Measures of quality
  - Required quality levels
  - Objective standards for dealing with changes

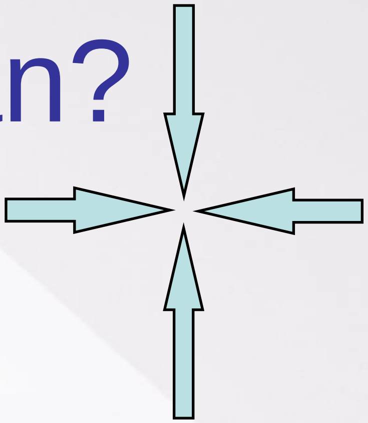
Part of an effective test plan

Tests are often our best approximation of the requirements

Projecting Ship Date from Defects



Should We CAT-Scan?

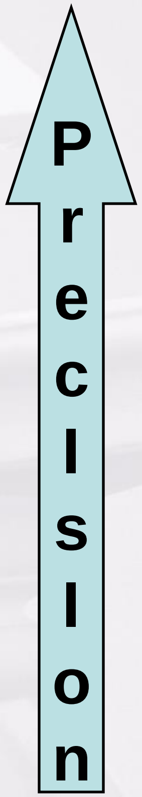


When the current testing method no longer reveals additional errors

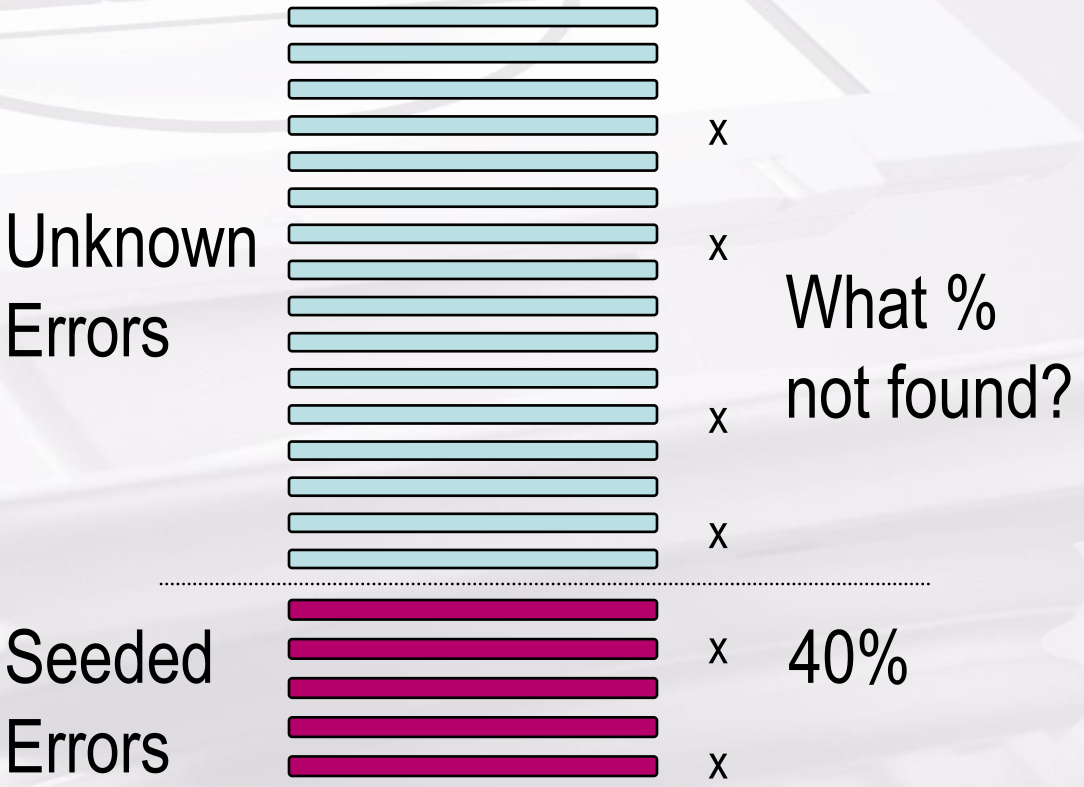
- Existence of additional errors is likely
- Chances of a different test finding them are reasonable
- Benefits of finding more errors now outweighs cost, time, risk of additional test

Estimating Remaining Defects: Statistics

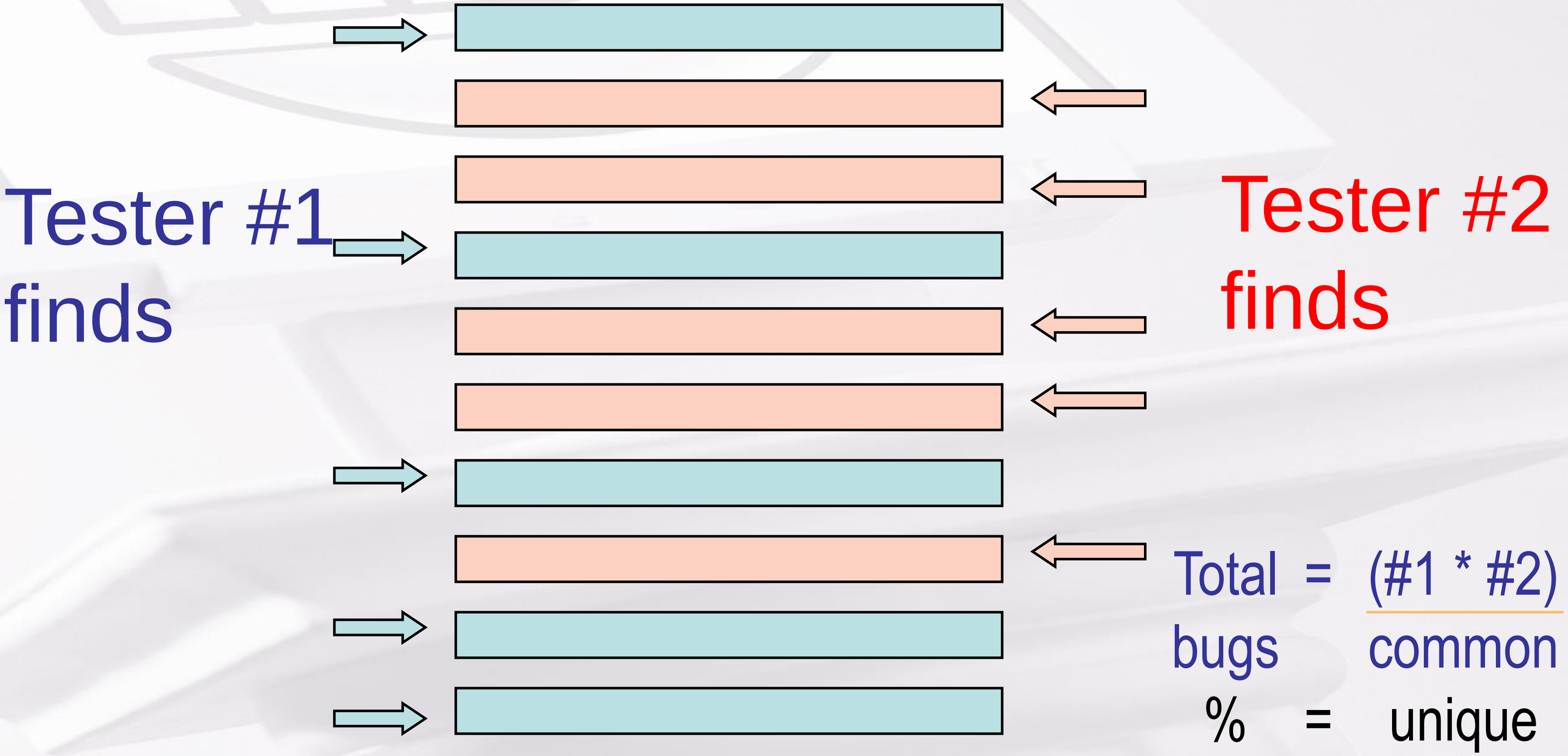
- ➢ Individual’s history
- ➢ Company, group history
- ➢ Database profiles
- ➢ General industry rules of thumb



Specific Estimates for This Software: Error Seeding



Specific Estimates for This Software: Two Independent Testers



Total bugs = (#1 * #2) / common

% found = unique / total bugs

What % not found?

Quiz 3

5. Relating Testing Project and Process

How well is our Testing doing?

Exercise: Test Project Effectiveness

Identify what measures test management needs in order to know reliably and communicate meaningfully how effective testing of a particular system has been?

What measures would give you confidence that the product is suitable for implementation?

Please feel free to pause the course and email your answers to me at Robin_Goldsmith@IIST.org

Exercise: Test Process Effectiveness

Identify what measures test management needs across projects to evaluate your testing process' effectiveness and guide specific improvements.

Please feel free to pause the course and email your answers to me at Robin_Goldsmith@IIST.org

Exercise: Formal Technical Reviews

Formal technical reviews, especially inspections, have been found to be the single most effective method for finding defects economically. Inspecting requirements and designs can save hundreds of hours by preventing many coding errors; and each hour of code inspection can find as many defects as up to ten hours of dynamic testing.

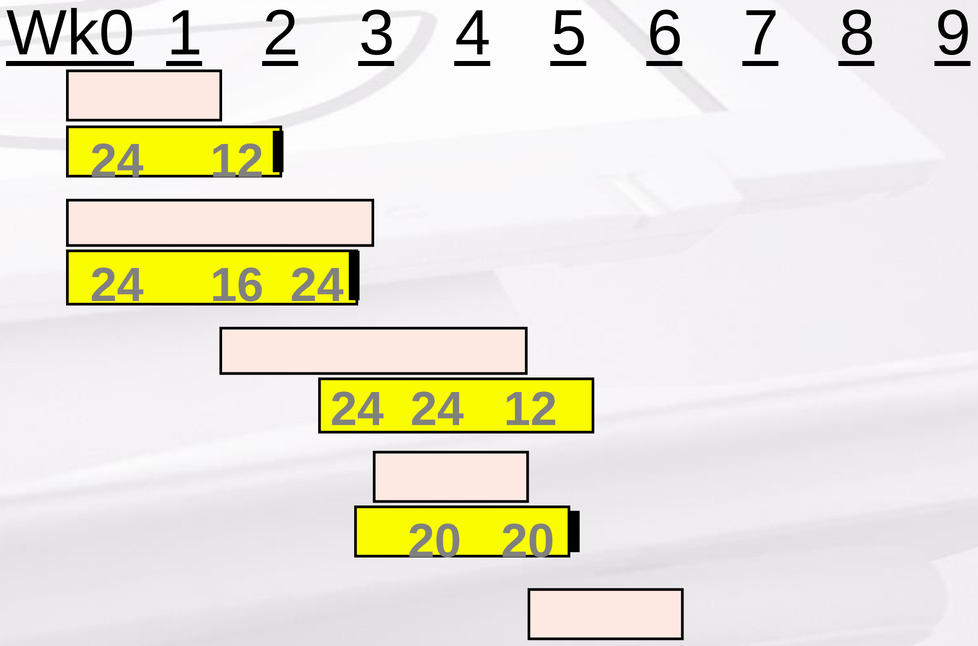
Identify measures test management would you need to counter resistance to reviews and show the extent such impacts occur in your own organization.

Please feel free to pause the course and email your answers to me at Robin_Goldsmith@IIST.org

Exercise: Formal Technical Reviews

Monitoring Milestones and Checkpoints

Task	Resp	Budg
Unit 1	Joe	40
		36
Unit 2	Sue	48
		64
Unit 3	Joe	56
		60
Unit 4	Sue	40
		40
Integ	TM	16



Projects fall behind one day at a time

What Percentage Completed?
Other Indicators of Testing Completion?

Unit 3? **90%** $\frac{60}{56}$ **= 107%** $\boxed{?}$

Project? **90%**

		Accountant's		**Earned Value**	
Unit 1	Joe	***36***	40	40	40
Unit 2	Sue	***64***	48	48	48
Unit 3	Joe	*60*	56	0	56
Unit 4	Sue	***40***	40	40	40
Integ	TM		16	0	16
		—	—	—	—
		200/200=100%		128/200 =64%	

Measuring Process Via Projects-- Get Baselines from Historical Data

- Track across projects by key test process points
  - Estimated and actual Development & Test effort, duration, and results (number of test cases, defect detection yield rates) relative to effort and project size (LOC, FP)
  - Defect density reduction
  - Defect age
  - Defect detection percentage
  - Cost of fixing defects
- Perform causal analysis and defect predictions
  - Project size, complexity (profile--application type, language, platform, team)
  - Development practices utilized
  - Test practices/tools used, including *formal technical reviews* (requirements, design, code, doc., tests)

Measuring Test Effectiveness

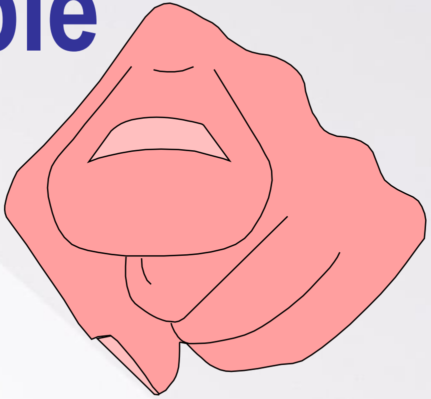
	Defects Found	Defects Missed	Detection Percent
Unit Test			
Integration Test			
System Test			
Acceptance Test			
Production			

WIIFM

Proactive

Quiz 4

YOU Are Responsible for Your Results!



Essential belief for managing projects

- Only ***YOU*** can learn/use the ideas
- ***YOUR*** active interest and openness are essential; "can it work?" not just does it fit your current mold?
- I try to be open too and answer as best I can.
  - If you're not getting what you want, or you're having trouble with *any* aspect of the class, ***YOU*** must act to make me aware while there's time to address your need.
  - That's how you get what you need and I improve.

Holding issues for the evaluations at the end is not appropriate

Objectives

Exam & Eval

Participants should be able to:

- Describe the essential elements of a test case, additional test documentation, and a structure to manage large volumes of testware
- Identify types of automated tools for supporting a managed testing environment and for executing tests
- Isolate and report defects so they are addressed
- Write effective testing status case reports
- Apply methods to reliably keep testing efforts on track and economical
- Measure both testing of particular software and overall test process effectiveness

Testing: I use as a memorable working definition: Demonstrating that a system or software does what it's supposed to do and doesn't do what it's not supposed to do. Testing involves confirming, not merely assuming, relying on someone else's opinion, or taking for granted, that a system works.

I also subscribe to the IEEE Std 729-1983 Glossary definition: The process of exercising or evaluating a system or system component by manual or automated means to verify that it satisfies specified requirements or to identify differences between expected and actual results.

Test plan: A test plan guides the preparation for and conduct of testing so that the testing can be carried out in an orderly, economical, and effective manner. Test plans ordinarily identify what is being tested (**SUT**—software/system under test); the approach or strategy to be taken to accomplish the testing; the specific things that must be demonstrated to be confident it works; the methods, standards, environment, and constraints to be applied; pass/fail, entry, and exit criteria; sources of data; workplan tasks sequence; required resources, effort, and duration; and work/administrative procedures including methods for modifying the plan and reporting results.

As described in IEEE Std. 829-1998, there are four levels of test planning documents. These may be separate or included within the same physical document. They differ mainly with respect to the size of the subject whose testing is being planned. The important part of them is their content, not their format. They should be written to allow reference, reuse, and refinement. They should include enough information to make sure that important things are remembered and not so much information as to obstruct the effective use of the important information. The determination of which planning documents to include and the extent of their formality and contents should be made based on size of the subject and risk.

A **Master Test Plan** is at a system- or project-level. It is a management agreement between the business/user and the technical organization senior executives about how the system will be tested. It sets defaults for lower-level test planning documents and identifies that set of Detailed Test Plans that taken together will give confidence the system as a whole works properly.

A **Detailed Test Plan** is a technical document that describes how to conduct each:

Unit Test (test of the smallest pieces of executable code, often a program, module, or object, usually performed by the developer),
Integration Test (testing combinations of units and other integrations),
System Test (testing the end-to-end integration in a production-like environment),
User Acceptance Test (the business/user's determination that the delivered system meets their
- **Business Requirements** [in business language, what business results must be delivered to provide value] as opposed to
- **System/Software/Functional Requirements/Specifications** [what technical people, and many others through lack of awareness,

generally consider to be the requirements, but which really describe the externally observable behavior—i.e., high-level design--of a particular product that is expected to be built/acquired; which is one of usually many possible ways to meet the Business/User Requirements, even though frequently the Business/User Requirements have not been defined explicitly and are simply assumed to be the System/Software/Functional Requirements/Specifications]

when operated in the manner and by the people who will be using it in production),

Independent QA Test (often the final test by the technical organization before the system is released to the users, conducted by a QA or Testing function that is independent of the developers and which endeavors to represent the users in the testing), or

Special Test (tests which are demonstrating features, functions, and capabilities that tend to be relevant to considerations outside of the application's particular functionality, such as performance or security; Special Tests often are considered part of System Tests and usually are referred to individually rather than as a group, which is why there is no widely used term for them—some people refer to them as Environmental Tests of Nonfunctional Tests).

Each Detailed Test Plan identifies the set of features, functions, and capabilities that taken together demonstrate the subject of the Detailed Test Plan works.

For each such feature, function, or capability (or consolidations thereof) can be a **Test Design Specification** which identifies the set of conditions that taken together demonstrate that it works plus information relevant to carrying out the testing. The Test Design Specification also identifies the set of Test Case Specifications that taken together will demonstrate that the software works under these conditions.

A **Test Case Specification** describes in words the executable test's conditions/inputs, expected results, and any procedural information necessary to perform the test. To be executed, **Test Case Data Values**--the actual input(s) and expected result(s)--also are needed.

.

Test Case: A Test Case is an input and its expected result, plus procedural information necessary for executing the Test Case, including dependencies with other Test Cases. Input can include environmental factors, such as state, version, and presence/absence of other hardware/software. A Test Case includes a Specification, which describes the input and expected result in words, and the actual Data Values for each. A Test Case can be Simple—one input, one expected result—or Complex, consisting of a sequence of inputs and results.

Test Scenario: A Test Scenario describes the set of actions necessary to carry out a business event. A Test Scenario is often an individual path through a Use Case. A single action could be a single Test Case, or a combination of actions could be a single Test Case. Thus, a Test Scenario will involve executing one or more Test Cases.

Test Condition: The term "Test Condition" can be used in a wide variety of ways. The two most common in my perception are: (1) any factor influencing or affecting the conduct of a test, such as, a customer in fact exists on the database and the tester has access to the database; and (2) what a test is intended to demonstrate, such as, being able to identify an existing customer. I try to stick with meaning (2) and try to limit my use of this generic term to what a Test Design Specification needs to have demonstrated to be confident it works, which means that as I endeavor to use the term, a Test Condition is at a sufficiently low level of detail that it can be demonstrated by a Test Case or small set of related Test Cases.

Test Script: A Test Script is a common format for documenting Test Cases. The Script implies a sequence: first this is input, and this is the result expected; then this next thing is input, and this next result is expect; etc. Each input-result combination can be considered a separate (Simple) Test Case, or a sequence of several input-result combinations together can be considered a separate (Complex) Test Case. Scripts are good for showing navigation, contingencies, and

Positive (or Valid) Testing (testing that the system/software does what it is supposed to do, that is, that it performs the functions that presumably produce business value which it is intended to perform in the manner intended); but they become cumbersome for describing multiple tests of the same set of variables, such as when doing

Negative (or Invalid) Testing (testing that the system/software in fact does not do what it is not supposed to do, often that the software detects and responds appropriately to an Invalid input).

Automated test execution tools create Scripts in a programming language that the tool understands. Such tools then execute the tests by reading the Script, generating the inputs to the SUT as if a user had entered them, capturing the SUT's outputs and comparing them to the expected results included within the Script. Each automated Script ordinarily is stored as a separate file and frequently is referred to as a separate Test Case.

Test Matrix: A Test Matrix is another common format for documenting Test Cases. The Matrix has a column for each input and result field/variable. Each row of the Matrix represents an individual Test Case with a unique combination of Data Values for the respective input and result fields/varaibles. This format is efficient for capturing a large number of Test Cases involving the same set of fields/variables; and it is the format embodied in batch processing testing, where each record in the input file is a separate Test Case. This format is not good for showing navigation or Test Cases involving differing inputs/expected results. Sequences/dependencies between Test Cases may be implied or indicated but often are not.

Test Procedure: A Test Procedure describes the steps or actions necessary for carrying out a test (i.e., executing one or more Test Cases). Such steps may include set-up and tear-down of the Test Environment, gaining access and positioning within an application, and entering/creating inputs, as well as capturing, evaluating, and reporting results.

Test Design: As a verb, Test Design is the process of identifying that set of Test Cases that will be adequate for demonstrating that a Test Condition is met. The Test Design activity also includes identifying Test Case effectiveness and efficiency considerations along with necessary Test Procedures to carry out the tests.

As a noun, the term Test Design is shorthand for and equivalent to Test Design Specification.

Use Case: A Use Case describes step-by-step how an **Actor** (usually the user, but also a piece of hardware or software) interacts with the system. Often a Use Case describes a specific business event, as does a Test Scenario. However, Use Cases ordinarily are created during the development process and characterized as being the User's Requirements; whereas in fact Use Cases really usually are high-level design (i.e., System/Software/Functional Requirements/Specifications rather than Business/User Requirements) describing the expected usage of the system/software product the developers intend to build or acquire. A major by-product advantage of Use Cases is that they also translate very directly into Test Cases. That is, the functionality described in the Use Case can be tested by executing the system in the manner described by the Use Case. Each path through the Use Case is generally considered a separate Test Scenario, and often also each path is considered a separate Test Case. The **Happy Path** is the usual way (Test Scenario) that the Use Case will be used and is typically a Positive Test. **Alternative or Exception Paths** tend to be Negative Tests and frequently are not executed fully. There are three common formats for Use Cases, all of which are essentially Test Script formats too:

- **Two-Column Use Cases** show the Actor's action in the left column (i.e., the input) and the System's responses—the System could respond in more than one way--(i.e., expected results) next to it in the right column.
- **One-Column Use Cases** show the Actor's actions and the System's responses in a single column, so the distinction is not always so clear; the format also tends to be inconsistent as to whether a System response is shown as a separate step or included in the same step as the Actor's action.
- **Narrative-Format Use Cases** are essentially one-column use cases where the steps are not listed separately in a column; rather each step is a separate sentence or clause of a sentence, which makes this format hardest to interpret.

Defect: An error. A difference between actual and reasonably expected results, including omissions and failure to perform in reasonable manners.

Error: A defect. Also, the thing that was wrong that caused the observable defect.

Early and

You Can Seize Control Of Your Testing Process With Better Use Of Resources

By Robin Goldsmith

Testing is our primary means of controlling system and software risks. Every testing authority explicitly states that testing should be in proportion to the degree of risk involved: The greater the risk, the more testing is needed. For example, the IEEE SWEBOK software engineering body-of-knowledge section on testing states, "Testing must be driven based on risk; i.e., testing can also be seen as a risk management strategy." (Version 1.0, May 2001, page 75)

Conducted correctly, risk-based testing enables better use of limited time and resources to ensure that the most important work is completed and that any tests that are skipped or short-changed are of lesser import.

Planning Your Process

To determine which test cases are the most deserving of your available time and resources, your testing process should be planned and designed, largely through identifying, prioritizing and addressing potential risks.

To be confident in your testing process, you must think systematically about what needs to be demonstrated, rather than depend on the busywork paper-pushing that many seem to confuse with planning. Value is maximized by economically and appropriately documenting risks so they can be remembered, shared, refined and reused. Write no more than is useful—and no less.

Risk exposure represents the combination of impact (damage, if the risk does occur) multiplied by likelihood (that the risk will indeed occur). Much of risk literature involves variations on a useful but mechanical method for quantifying risk: assigning respective values to impact and likelihood, such as 1 = Low, 2 = Medium, and 3 = High, and then multiplying to produce a risk exposure score of 1 (very low) through 9 (very high). Figure 1 (page 26) shows the respective exposures graphically: Low = 1 box, Medium = 4 boxes and High = 9 boxes. As with most images, these graphic depictions can enhance understandability.

However, pictures can be misleading: Some testers may adjust the scale as a form of gamesmanship. For instance, values of 1 = Low, 3 = Medium and 5 = High yield risk exposure scores of 1 (very low) through 25 (very high). As Figure 2 (page 27) suggests, an exposure of 25 boxes can seem far greater than an exposure of nine boxes, although a maximum 25 score on a 1–25 scale actually represents a risk identical to a maximum 9 score on a 1–9 scale.

Virtually everyone agrees on the concept of basing testing on risk impact and likelihood. However, testers continually encounter difficulties in effectively applying risk-based testing in practice. In explanation, many testing authorities imply that the way to identify and prioritize risks is obvious, and that too often, overly simplistic examples make a task seem deceptively easy—until you try to put the techniques in practice with a real system.

An even more significant hidden risk to the development and testing process occurs when testers are unaware of the possibility that they may not be adequately identifying and prioritizing risk-based tests. Risks that aren't identified

Robin F. Goldsmith, JD, is President of Needham, Mass.–based consultancy Go Pro Management, Inc. A prominent speaker and author of "Discovering REAL Business Requirements for Software Project Success" (Artech House, 2004), he advises and trains systems and business professionals. He can be reached at robin@gopromanagement.com.

Effective: The Perks Of Risk-Based Testing

can't be controlled, and risks that are identified too late can be overly difficult and expensive to mitigate. The complacency arising from unwitting overreliance on flawed processes can be a thornier problem than the flaws in the processes themselves.

To more fully gain the advantages of risk-based testing, testers and others involved in the development process must learn to thoroughly identify and reliably prioritize potential risks.

Picking Priorities

Accurately identifying potential risks is the first, most important and most difficult step in controlling risk through testing or other means.

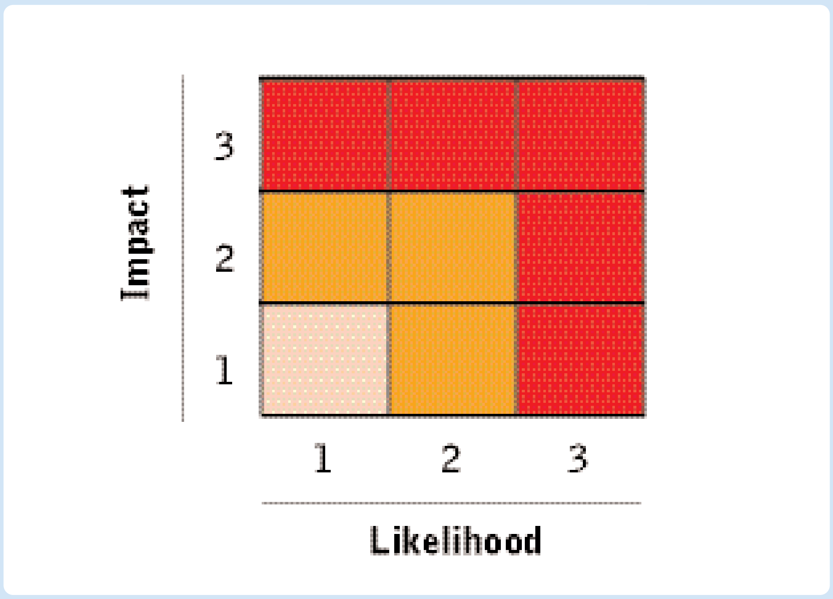
Risk literature is full of checklists of risks that commonly afflict various situations. While these lists can serve as helpful reminders of things to look for, they can also create a number of usually unrecognized problems. And, though they're intended to guide thinking, checklists can easily become a substitute for thoughtfulness. Many checklist users miss things because they overrely on their lists, uncritically assuming they address everything they need to cover, or failing to understand how to apply lists to their own situation.

Again, evaluating risk identification effectiveness to avoid mindless oversights is at least as important as having defined procedures, guidelines and checklists to follow in identifying potential risks.

Moreover, most checklists help with identifying potential risks, but not with prioritizing those risks. Thus, if you've found a checklist helpful, it may be worth the effort to expand the list items to aid prioritization. For example, a typical risk checklist item might be "Interfaces with other systems," which invites possibly unreliable subjective interpretations. For a more objective

Photograph by A. Navrin

FIG. 1: RISK PRIORITIZATION



analysis of interface risks, the item could be rewritten as:

Low risk = 0 interfaces with other systems
Medium risk = 1–2 interfaces with other systems
High risk = 3 or more interfaces with other systems

Checklists have a tendency to inadvertently intermix perspectives, which can seriously interfere with the prioritization process. Apples-versus-oranges confusion can be reduced by consciously conceptualizing risk identification perspectives and systematically approaching each perspective individually. Many published risk checklists reflect a dominant perspective that could be refined to facilitate apples-versus-apples comparisons.

The bulk of the literature on risk comes from a management perspective. People with engineering and operations orientations, including testers and non-engineering folks such as many business users, tend to concentrate on risks from a product/technical perspective. In part because generalized lists are unlikely to address an individual situation's specifics, business effects comprise the perspective that is most commonly neglected—and which, ironically, carries the most significant risks of all.

Business Effect Risks

The common failure to sufficiently recognize, understand and appreciate the importance of business effect risks virtually guarantees that all other risk-control efforts (such as testing) will be less effective.

Therefore, the first and most important test-planning and design risk-identification step is to determine the effects on your business if the system/software doesn't work as needed. Note that the issue is *what* the business requires, not product/system/software requirements that dictate *how* a presumed product or system solution is intended to work. Business effect risks answer the question, "So what if such and such happens—or doesn't?"

Not only are we unaccustomed to thinking in business terms, even when we do, we may have difficulty thinking about effects. Thus, despite conscious direction to identify business *effect* risks, people tend to instead identify actions and events that are presumably *causes* of some damaging effects, though those effects are generally not articulated.

Intermingling risk causes and effects, which is a common checklist approach, also inhibits meaningful prioritization, because comparisons must be made among comparables. Since risk is an effect, it's essential that impact/likelihood prioritization be focused entirely on potential risk effects. On the other hand, mitigation approaches are based on those effects' causes, which in turn must be clearly understood.

Business effects are the most important of all risks because they provide a context for analyzing all risks. Without such a context, it's easy to miss the most important management and techni cal risks, and divert your attention to more visible but perhaps less important areas.

Management Risks

Much of the risk literature deals with management risks, such as lack of resources and time, long duration, large projects and faulty development processes, including poor estimation, inadequate testing and requirements changes. Since 9/11, articles on risk have tended to deal with security breaches, although in the aftermath of Hurricane Katrina, the emphasis has shifted somewhat to the risks of natural disasters.

The most common published checklists come from a management risk perspective, and almost all such lists unwittingly intermingle causes and effects. Attempting to use these checklists for risk-based testing poses several additional pitfalls.

Checklists can inadvertently mix perspectives, which can interfere with the prioritization process.

First, except for a limited set of specifics such as security and disaster recovery, testing may not be a relevant response to mitigate many management risks. For instance, an understaffed project with an overly aggressive deadline will undoubtedly encounter quality issues, partly because testing has probably already been squeezed. More testing in general will help, but such general management risks don't help you determine *which* testing to increase.

Second, risk involves statistical probability. However, for most organizations, many of the typical management-risk checklist's items aren't risks, but certainties. Practically every project is underestimated, usually by a large percentage that grows as requirements appear to change, always in ways that necessitate additional time and resources. Incorrect estimates result in allocating insufficient or inappropriate time and resources, which regularly leads to deliveries that are late (or nonexistent), over budget, and wrong.

Inadequate testing is both a symptom and a cause of the predictable problems of typical development practices. Squeezing testing is a common management response to underestimated projects that are running late, but being late also stresses development and produces still more errors, which the squeezed testing is in turn incapable of catching. By definition, such faulty processes will certainly continue to create the conditions that produce problems. Traditional risk-based testing is highly unlikely to be the means for ensuring the time and resources necessary to break this cycle.

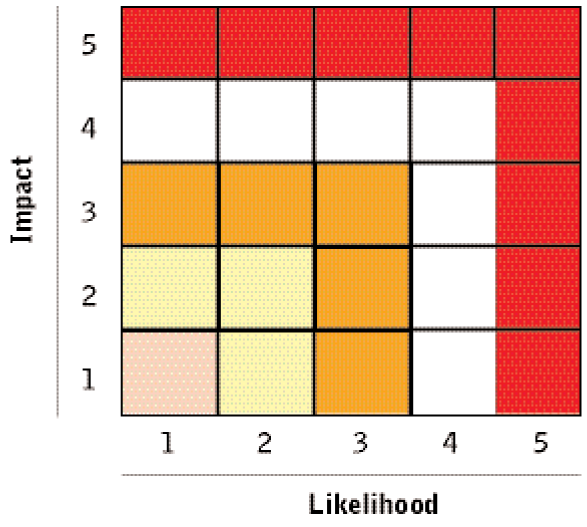
Third, testers tend to approach matters from the bottom up, often mainly in terms of executable test cases, and checklists seldom recognize the interrelationships of various management risks. This often quashes awareness about the ways that testing could in fact help mitigate the conditions cited on management risk checklists.

For example, the main reason for inadequate budgets, resources and schedules is inadequate task definition and estimation, which stems mainly from inadequate understanding of the work to be done, which is generally caused by inadequate product/system design, which is mainly due to inadequate definition of the real business requirements.

Failure to adequately discover the real business requirements is often caused by a lack of awareness of their nature and of the vital need to identify them in detail. Instead, conventional wisdom concentrates on the system/software/functional requirements, which actually comprise a high-level design of the product/system that is *presumed* to be what is needed to satisfy the *presumed* but usually undefined real business requirements. Table 1 summarizes these distinctions. Focusing solely on any downstream issue such as high-level design addresses symptoms rather than causes.

By recognizing how these management risks are interrelated and largely stem from inadequate requirements, appropriate testing can be a powerful aid to mitigation through detecting requirements and design issues early, when they're easiest and cheapest to fix.

FIG. 2: RISK PRIORITIZATION GAMESMANSHIP



Reviews

Reviews are the method used for "testing" or evaluating the adequacy of earlier development deliverables, such as requirements and designs. Many organizations consider reviews to be quality assurance rather than testing. Regardless, while certainly helpful, conventional requirements and design reviews tend to be far weaker than is usually recognized.

Some organizations don't review their requirements and designs at all; those that do ordinarily use only one or two weaker-than-realized techniques. Most conventional reviews miss many of the most important issues because they erroneously focus on product/system/software requirements' high-level design, rather than on real business requirements that provide value. Also, conventional reviews often concentrate mainly on matters of form, such as clarity and testability, rather than substance. And too often, the rest of the organization disregards such format-oriented review results as trivial nitpicking.

In contrast, Proactive Testing (see "Mindset and Methodology" sidebar, page 30) can break this vicious cycle, with more than 21 ways to evaluate the adequacy of business requirements, and more than 15 ways to evaluate designs. Many of these are more powerful, content-oriented techniques that also spot the overlooked and incorrect requirements and designs that format-oriented reviews tend to miss.

Overlooked and incorrect business requirements often relate to significant business effect risks. Business and management who appreciate the importance of these issues can look to Proactive Testing to spot red flags before they become predictable problems.

TABLE 1: TWO TYPES OF REQUIREMENTS

Business/User Requirements	Product/System/Software Requirements
Business/user perspective and language, conceptual.	*Human-defined* product or system perspective.
Exists within the business environment and thus must be discovered.	One of the possible ways (high-level design) to accomplish presumed business requirements.
When delivered, accomplished or satisfied, provides value by serving business objectives, typically solving expected problems, meeting challenges or taking opportunities.	Often phrased in terms of the external functions that the product/system is expected to perform; thus also called *functional specifications/requirements.*

Moving Beyond Reactive Testing

Testers tend to come at risks from the perspective of the product or system being developed. Conventional risk-based testing, as espoused by essentially every testing authority, analyzes a system's design to identify risks related to the system's functionality/features (which are sometimes called *requirements risks*) and/or the system's technical structure (its components and how they're put together), and then tests the higher risks more frequently.

Such risk identification may be done individually or in a group. Ordinarily, the system's functionality is defined based on user interface design, where each menu or GUI button choice is deemed a separate feature to analyze for risk. To identify structural risks, organizations often have a wider variety of technical design artifacts. For example, architecture and network diagrams, database structures and object models each may be reviewed to identify different types of component risks.

While sometimes guided by checklists, product/technical risk reviewers are often simply directed to apply their experienced judgment to identify those features and components that will have a significant impact if they go wrong, as well as those that seem likely to go wrong.

For each test-design specification to be tested, a set of test-cases is identified that will prove that the feature, function or capability really works.

Product/technical risk checklists generally prompt attention to things that are new or changed, complex or large, or used frequently. They also focus on interfaces, dependencies and historical problems. While definitely good advice, such checklists are often difficult to apply to particular systems, especially with regard to functionality.

This conventional risk-identification approach is necessary and important, but is generally less effective than realized because it's reactive. One reason for this reactivity is that, much to their frustration, testers often don't come on board until the end of the development process, after problems have already been incorporated in the code. And, in a side effect of such reactive late involvement, risk identification can become overwhelming, with masses of details dumped on the testers all at once. Without effective ways to get a handle on the whole thing, it's easy to get mired in detail and lose sight of bigger issues.

In addition, the greatest weakness of most conventional testing is that risk identification merely reacts to and therefore tends to be limited to whatever's in the design or code. Being guided by the design—or, even worse, the code— can lead risk identification away from recognizing errors and omissions.

The Proactive Plus

In addition to more effective reviews of requirements and design, Proactive Testing includes but goes beyond conventional testing's reactive product risk identification. Taking business and management as well as product perspectives, Proactive Testing uses specific techniques that help identify many important risks that reactive approaches miss. Moreover, Proactive Testing addresses risks earlier and at multiple points in the development process, scaled to need.

Consequently, whereas conventional reactive testing enables more frequent testing of higher risks, Proactive Testing enables testing higher risks not just more often, but earlier, when problems are easier to fix.

Proactive Testing can employ the approach suggested by IEEE Std. 829-1998 to organize the test planning and design process. When applying this proven project-management practice, testing is systematically broken down into smaller, more manageable pieces. At each point in the process, risks are identified and prioritized to guide a successively narrower focus on the most important areas to test.

This structure provides a way of thinking, not a mandate for generating paperwork, as some people (mis)interpret the Standard. However, it's important to write information down so it can be remembered, refined and reused. In Proactive Testing, you write as much as is helpful—no more, but no less.

The earliest and most important Proactive Testing risk analysis occurs as part of master test planning, which is performed most effectively as soon as you have a high-level system design.

The master test plan is the project plan for the testing project, which is a subproject within the overall development project. Each key risk to the system being developed is addressed in a detailed test plan—for each unit, integration, system and special (such as load or security) test. Detailed test plans dealing with the highest risks should be tested more often and earlier in the life cycle.

Powerful Proactive Testing techniques identify the biggest risks, including up to 75 percent of the showstoppers that conventional reactive testing ordinarily overlooks because typical development processes have failed to address these critical risks in the system design. Moreover, Proactive Testing identifies the risks early enough to let testing drive development. That is, while it's important to merely catch showstoppers prior to production, Proactive Testing can increase value dramatically by catching the showstoppers in time to prevent writing affected related code that otherwise must be thrown out and rewritten.

The value of spotting these ordinarily overlooked risks is greatest in conjunction with high-level design, but remains significant throughout development. Payback from preventing a showstopper risk is always valuable, even if it's not found until the day before the show would have been stopped.

While checklists can help spot generic types of risks that afflict many systems, Proactive Testing also emphasizes understanding the specifics in order to reveal the system's unique big risks, which are perhaps the most commonly overlooked. The written risk findings can become a checklist for future risk analyses, but beware that overreliance on checklists can prevent the meaningful, content-based risk identification that Proactive Testing uses to spot so many otherwise overlooked risks.

It's especially valuable to involve a broad range of participants in identifying these big risks, since Proactive Testing facilitation prompts each different view to identify risks that other participants would probably miss. Experiencing the discovery of big risks wins instant advocates for early, Proactive Testing involvement. In addition, a group representing a broad mix of involved stakeholders provides the most effective risk prioritization.

Because Proactive Testing identifies big risks early, test plans dealing with the higher risks can be tested not only more often, but earlier in the life cycle.

Scaling and Carrying Through

Each of these big-risk areas can be addressed in a detailed test plan, which should be just elaborate enough to be helpful; no more, but no less. The higher the risk involved, the more analysis will be needed. Lower-risk areas can be given respectively less attention, even if only to the point of documenting the conscious decision not to analyze them further.

Because Proactive Testing identifies these big risks early, detailed test plans dealing with higher risks can be tested not only more often, but earlier in the life cycle.

To do this, development must structure and schedule its work to create the higher-risk components earlier so they can be tested earlier. One powerful way to do this is to define the sequence of builds based at least in part on Proactive Testing risk prioritization.

Introducing New

yKAP® 2.4

Your Kind Attention Please

The Antidote for Over-Priced, Under-Featured, Hard-to-Use Bug-Tracking Software

yKAP 2.4
The next generation Web-based Defects and Issues Tracking Software

- Superior Functionality.
- Powerful Performance.
- Easy Customization.
- Effortless Collaboration.
- Unbeatable Value Pricing

Compare yKAP 2.4 and Save Up to 45% Off Your Purchase Price!
Visit, download & save:
www.yKAPbugTracker.com

DCom solutions

MINDSET AND METHODOLOGY

Proactive Testing is as much a mindset as a methodology for risk-based software test planning, design and management. Positioning testing within an overall quality and business value context, Proactive Testing emphasizes WIIFMs (What's In It For Me) that win over users, managers and developers. Rather than the traditional adversarial perception of testing as an obstacle to delivery, Proactive Testing can save you aggravation while helping you deliver better systems cheaper and quicker.

The Proactive Testing life cycle embraces true agility to minimize wasted effort by enabling the development process to build more right in the first place. At the same time, by intertwining more thorough and effective tests of each development deliverable at multiple key points, it can economically detect more errors.

Proactive Testing uses higher-payback test-planning and design methods that augment conventional testing techniques to find numerous test conditions commonly overlooked by traditional testing methods, allowing more effective scaled risk prioritization and repeated refocusing on the most important tests. Moreover, anticipation can promote reuse and let testing drive development to prevent problems or catch them earlier, when they're easier to fix.

By maintaining a real process perspective, Proactive offers a fresh take on conventional User Acceptance Testing (UAT). Rather than a subset of technical testing (unit, integration, system/special), UAT should be kept independent. With truly proactive, user-centered planning/design of UAT up-front for execution at the end, Proactive Testing can help improve requirements and designs while providing a more thorough and effective double-check of the developed system.

Finally, Proactive Testing accepts responsibility for results, which includes meaningful measurement and active management.

Each detailed test plan identifies the set of test design specifications that, taken together, give assurance that the subject of the detailed test plan works. Each test-design specification describes the set of test cases (inputs/conditions and expected results) that must be conducted to ensure confidence that a feature, function or capability works.

Just as Proactive Testing methods identify numerous otherwise-overlooked big-risk areas, additional special techniques successively identify many risks in ordinarily overlooked features, functions and capabilities. Conventional methods that fail to identify these risks can't include them in risk prioritization or mitigation.

The fully identified set of test design specifications is then prioritized according to the risks they address. Those dealing with the highest risks are tested earlier and more often. In addition, by structuring test-design specifications to be reusable, Proactive Testing offers an additional way to increase the amount of testing in the time available.

For each test-design specification to be tested, a set of test cases is identified that will prove that the feature, function or capability works. The test cases are ranked based on the risks they address, with the highest-risk test cases executed earlier and more often.

Prioritization Alternatives

It's common to use the well-known impact-multiplied-by-likelihood high/medium/low ratings risk-prioritization approach described above. However, this method frequently takes too much work and is too difficult to achieve consensus on, especially with a broadly representative group of stakeholders.

A further concern with the technique is that rating each risk individually (as does the commonly used mandatory/desirable/ideal differentiation technique) can actually prevent effective prioritization because each risk viewed in isolation tends to be rated high.

By definition, prioritization demands distinctions among risk choices, so it's advisable to use a risk prioritization technique that produces a ranking. As a technique, ranking raises two further issues.

Effective risk-based testing doesn't end with risk identification and prioritization.

First, for meaningful ranking, your selections must be limited to a manageable number, typically no more than 15 and preferably closer to seven (a range that numerous scientific studies have found to be the parameters of the typical human attention span).

Second, since rank alone doesn't convey relative differences in importance, it's more valuable to use a risk analysis technique that enables ranking based on quantified importance. Such methods can be gross or precise; formal or informal.

The simplest technique is to gain group consensus on a relatively small subset of risks that are very important relative to the fuller set of all identified risks. The subset of highest risks can then be ranked, or ranking can be skipped if it's certain that all the subset members will be addressed.

You can also try a more formal technique: the *100-point must system*. Each participant is given 100 points that he must allocate among the risk choices in proportion to importance. All 100 points must be allocated, and at least one point must be allocated to each risk choice. Wide discrepancies are then discussed and adjusted as appropriate, followed by arithmetic averaging of respective risks.

Managing the Process

Effective risk-based testing doesn't end with risk identification and prioritization. Tests of the highest risks must be planned, designed and executed. As time and resources allow, lower risks must also be tested.

Most importantly, testing effectiveness must be monitored and measured, issues must be detected in a timely manner, and appropriate actions taken.

Ultimately, the time and cost of testing must be weighed against its benefits. Did the testing catch problems that would have made a significant impact? Would the problems have been found without risk identification and analysis? Would the costs of those problems' occurrence exceed the costs of detecting them? What kinds of damage occurred in spite of risk-based testing? And what additional/different risk analysis and/or testing would have prevented the damage? ☒

ACTION PLAN

Concept or technique I can apply	Benefit to Me and my organization	My next step to make it happen

The Forgotten Phase

Development models often consider testing an afterthought, but there *are* models that focus on testing. This month, we'll examine the V Model—but is it flawed, too? Part 1 of 4. BY ROBIN F. GOLDSMITH AND DOROTHY GRAHAM

THE SOFTWARE WORLD HAS LONG FOUND IT helpful to define the phases of the development lifecycle. In addition to the classic waterfall approach, there are spiral or iterative processes, rapid application development (RAD) and—more recently—the Unified Process (known as RUP because of its origins at Rational Software). But testing often gets short shrift.

Just as development has its models, so, too, does testing. However, these processes tend to be less well known, partly because many testers have gained most of their expertise on the job. After all, even though testing constitutes about half of development time, most formal academic programs that purport to prepare students for software careers don't even offer courses on testing.

***Robin F. Goldsmith** is the president of Go Pro Management Inc. consultancy in Needham, Mass., which he cofounded in 1982. A frequent conference speaker, he trains business and systems professionals in testing, requirements definition, software acquisition and project and process management. Reach him via www.gopromanagement.com. **Dorothy Graham** is the founder of Grove Consultants (www.grove.co.uk) in the U.K., which provides consultancy, training and inspiration in software testing, test automation and inspection. With Tom Gilb, she is coauthor of* Software Inspection *(Addison-Wesley, 1993) and with Mark Fewster, coauthor of* Software Test Automation *(Addison-Wesley, 1999). In 1999, she was awarded the IBM European Excellence Award in Software Testing.*

Experts keep proposing new models because developers find value in them, but also feel limited by existing models. For example, no approach has been presented more fully than the RUP, yet even it has significant gaps—so many, in fact, that *Software Development* contributing editors Scott Ambler and Larry Constantine filled four volumes with the collected articles from the magazine that describe best practices and gap-filling in the RUP. Incidentally, one of those gaps is the RUP's failure to address test planning (see "Plan Your Testing," by Robin Goldsmith, in *The Unified Process Inception Phase* [CMP Books, 2000]).

The V Model

The V Model, while admittedly obscure, gives equal weight to testing rather than treating it as an afterthought.

Initially defined by the late Paul Rook in the late 1980s, the V was included in the U.K.'s National Computing Centre publications in the 1990s with the aim of improving the efficiency and effectiveness of software development. It's accepted in Europe and the U.K. as a superior alternative to the waterfall model; yet in the U.S., the V Model is often mistaken for the waterfall.

In fact, the V Model emerged in reaction to some waterfall models that showed testing as a single phase following the traditional development phases of requirements analysis, high-level design, detailed design and coding. The waterfall model did considerable damage by supporting the common impression that testing is merely a brief detour after most of the mileage has been gained by mainline development activities. Many managers still believe this, even though testing usually takes up half of the project time.

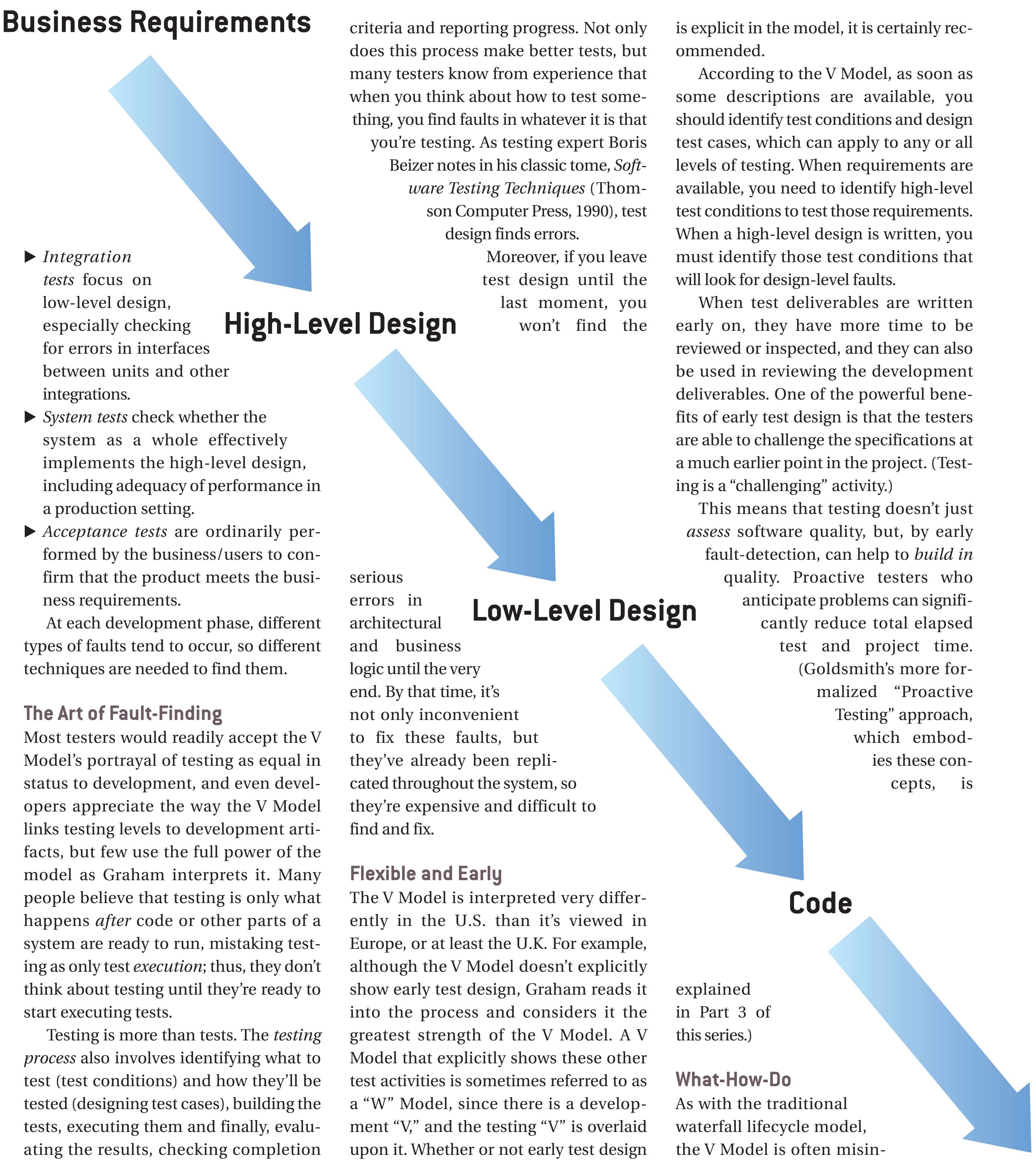
Testing Is a Significant Activity

The V Model portrays several distinct testing levels and illustrates how each level addresses a different stage of the lifecycle. The V shows the typical sequence of development activities on the left-hand (downhill) side and the corresponding sequence of test execution activities on the right-hand (uphill) side (see next page). (Note that some organizations may have different names for the various development and test activities.)

On the development side, we start by defining business requirements, then successively translate them into high- and low-level designs, and finally implement them in program code. On the test execution side, we start by executing unit tests, followed by integration, system and acceptance tests.

The V Model is valuable because it highlights the existence of several *levels of testing* and delineates how each relates to a different development phase:

- *Unit tests* focus on the types of faults that occur when writing code, such as boundary value errors in validating user input.



- *Integration tests* focus on low-level design, especially checking for errors in interfaces between units and other integrations.
- *System tests* check whether the system as a whole effectively implements the high-level design, including adequacy of performance in a production setting.
- *Acceptance tests* are ordinarily performed by the business/users to confirm that the product meets the business requirements.

At each development phase, different types of faults tend to occur, so different techniques are needed to find them.

The Art of Fault-Finding

Most testers would readily accept the V Model's portrayal of testing as equal in status to development, and even developers appreciate the way the V Model links testing levels to development artifacts, but few use the full power of the model as Graham interprets it. Many people believe that testing is only what happens *after* code or other parts of a system are ready to run, mistaking testing as only test *execution*; thus, they don't think about testing until they're ready to start executing tests.

Testing is more than tests. The *testing process* also involves identifying what to test (test conditions) and how they'll be tested (designing test cases), building the tests, executing them and finally, evaluating the results, checking completion criteria and reporting progress. Not only does this process make better tests, but many testers know from experience that when you think about how to test something, you find faults in whatever it is that you're testing. As testing expert Boris Beizer notes in his classic tome, *Software Testing Techniques* (Thomson Computer Press, 1990), test design finds errors.

Moreover, if you leave test design until the last moment, you won't find the serious errors in architectural and business logic until the very end. By that time, it's not only inconvenient to fix these faults, but they've already been replicated throughout the system, so they're expensive and difficult to find and fix.

Flexible and Early

The V Model is interpreted very differently in the U.S. than it's viewed in Europe, or at least the U.K. For example, although the V Model doesn't explicitly show early test design, Graham reads it into the process and considers it the greatest strength of the V Model. A V Model that explicitly shows these other test activities is sometimes referred to as a "W" Model, since there is a development "V," and the testing "V" is overlaid upon it. Whether or not early test design is explicit in the model, it is certainly recommended.

According to the V Model, as soon as some descriptions are available, you should identify test conditions and design test cases, which can apply to any or all levels of testing. When requirements are available, you need to identify high-level test conditions to test those requirements. When a high-level design is written, you must identify those test conditions that will look for design-level faults.

When test deliverables are written early on, they have more time to be reviewed or inspected, and they can also be used in reviewing the development deliverables. One of the powerful benefits of early test design is that the testers are able to challenge the specifications at a much earlier point in the project. (Testing is a "challenging" activity.)

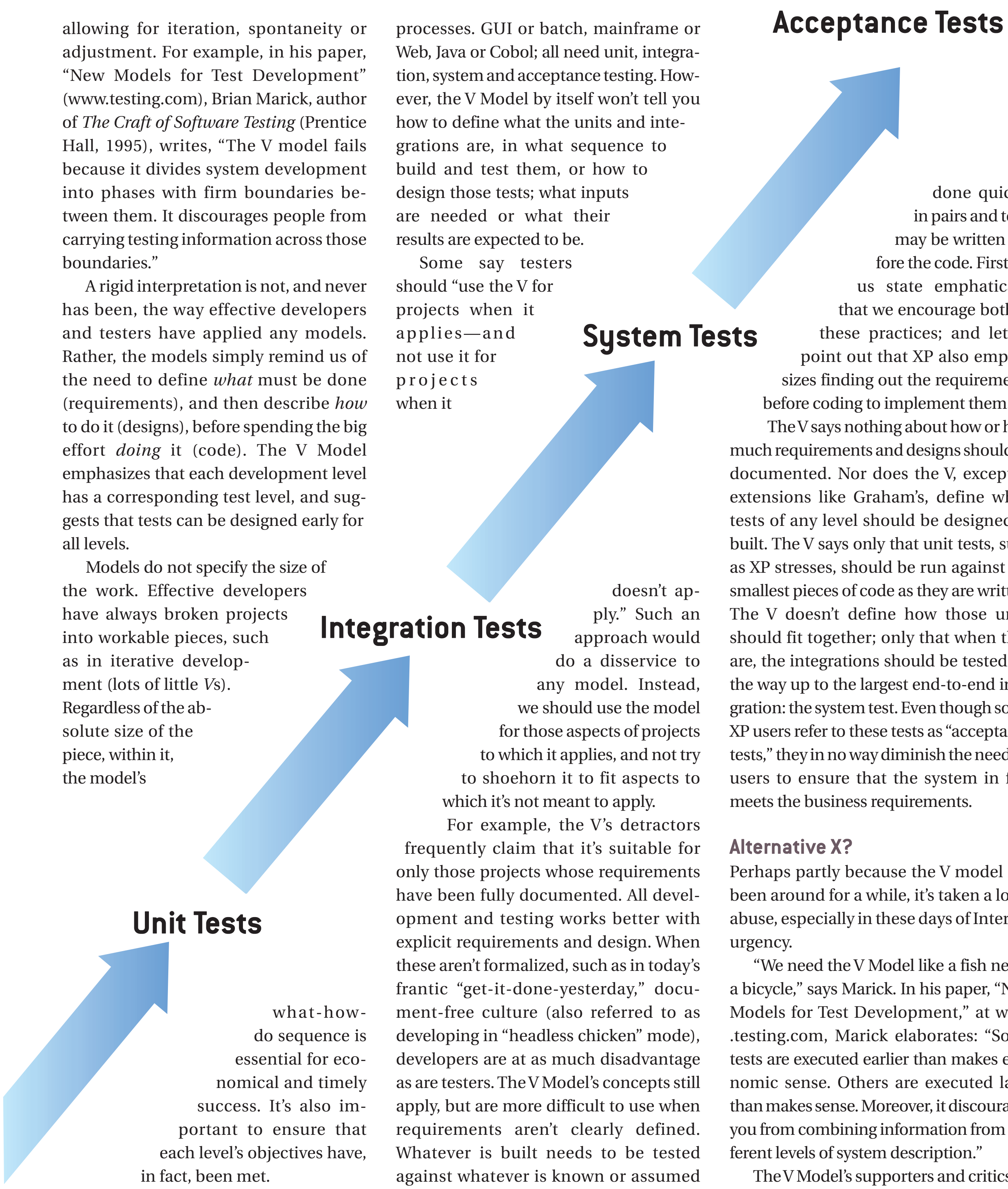
This means that testing doesn't just *assess* software quality, but, by early fault-detection, can help to *build in* quality. Proactive testers who anticipate problems can significantly reduce total elapsed test and project time. (Goldsmith's more formalized "Proactive Testing" approach, which embodies these concepts, is explained in Part 3 of this series.)

What-How-Do

As with the traditional waterfall lifecycle model, the V Model is often misinterpreted as dictating that development and testing must proceed slavishly in linear fashion, performing all of one step before going on to perform all of the next step, without

The V Model

The V shows the typical sequence of development activities on the left-hand (downhill) side and the corresponding sequence of test execution activities on the right-hand (uphill) side.



allowing for iteration, spontaneity or adjustment. For example, in his paper, "New Models for Test Development" (www.testing.com), Brian Marick, author of *The Craft of Software Testing* (Prentice Hall, 1995), writes, "The V model fails because it divides system development into phases with firm boundaries between them. It discourages people from carrying testing information across those boundaries."

A rigid interpretation is not, and never has been, the way effective developers and testers have applied any models. Rather, the models simply remind us of the need to define *what* must be done (requirements), and then describe *how* to do it (designs), before spending the big effort *doing* it (code). The V Model emphasizes that each development level has a corresponding test level, and suggests that tests can be designed early for all levels.

Models do not specify the size of the work. Effective developers have always broken projects into workable pieces, such as in iterative development (lots of little *V*s). Regardless of the absolute size of the piece, within it, the model's what-how-do sequence is essential for economical and timely success. It's also important to ensure that each level's objectives have, in fact, been met.

What V Won't Do

The V Model *is* applicable to all types of development, but it is not applicable to all *aspects* of the development and testing processes. GUI or batch, mainframe or Web, Java or Cobol; all need unit, integration, system and acceptance testing. However, the V Model by itself won't tell you how to define what the units and integrations are, in what sequence to build and test them, or how to design those tests; what inputs are needed or what their results are expected to be.

Some say testers should "use the V for projects when it applies—and not use it for projects when it doesn't apply." Such an approach would do a disservice to any model. Instead, we should use the model for those aspects of projects to which it applies, and not try to shoehorn it to fit aspects to which it's not meant to apply.

For example, the V's detractors frequently claim that it's suitable for only those projects whose requirements have been fully documented. All development and testing works better with explicit requirements and design. When these aren't formalized, such as in today's frantic "get-it-done-yesterday," document-free culture (also referred to as developing in "headless chicken" mode), developers are at as much disadvantage as are testers. The V Model's concepts still apply, but are more difficult to use when requirements aren't clearly defined. Whatever is built needs to be tested against whatever is known or assumed about what the code is supposed to do and how it's supposed to do it.

Detractors also might say the V is unsuited for techniques like Extreme Programming (XP), in which programming is done quickly in pairs and tests may be written before the code. First, let us state emphatically that we encourage both of these practices; and let us point out that XP also emphasizes finding out the requirements before coding to implement them.

The V says nothing about how or how much requirements and designs should be documented. Nor does the V, except in extensions like Graham's, define when tests of any level should be designed or built. The V says only that unit tests, such as XP stresses, should be run against the smallest pieces of code as they are written. The V doesn't define how those units should fit together; only that when they are, the integrations should be tested, all the way up to the largest end-to-end integration: the system test. Even though some XP users refer to these tests as "acceptance tests," they in no way diminish the need for users to ensure that the system in fact meets the business requirements.

Alternative X?

Perhaps partly because the V model has been around for a while, it's taken a lot of abuse, especially in these days of Internet urgency.

"We need the V Model like a fish needs a bicycle," says Marick. In his paper, "New Models for Test Development," at www.testing.com, Marick elaborates: "Some tests are executed earlier than makes economic sense. Others are executed later than makes sense. Moreover, it discourages you from combining information from different levels of system description."

The V Model's supporters and critics do agree on one thing: Although no model is a perfect representation of reality, in practice, some models are useful. Next month, we'll examine the X Model, which we've based on concepts Marick espouses.

This or That, V or X?

The X Model purports to fill the gaps left open by the V Model. But does it really handle all the facets of development, such as handoffs, frequent builds and so on? Part 2 of 4. BY ROBIN F. GOLDSMITH

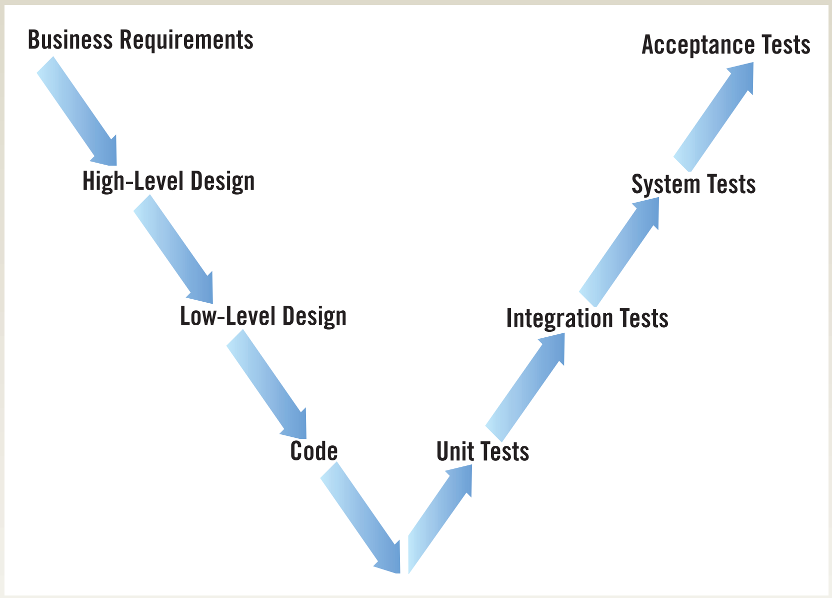
THE V MODEL IS PROBABLY THE BEST-KNOWN testing model, although many practicing testers may not be familiar with it—or any testing model. It has been around a long time and shares certain attributes with the waterfall development model—including a propensity to attract criticism. Since Part 1 of this series, "The Forgotten Phase" (July 2002), dealt in detail with the V Model, I'll give only a brief overview of it here.

The V proceeds from left to right, depicting the basic sequence of development and testing activities. The model is valuable because it highlights the existence of several *levels of testing* and depicts the way each relates to a different development phase. *Unit testing* is code-based and performed primarily by developers to demonstrate that their smallest pieces of executable code function suitably. *Integration testing* demonstrates that two or more units or other integrations work together properly, and tends to focus on the interfaces specified in low-level design. When all the units and their various integrations have been tested, *system testing* demonstrates that the system works end-to-end in a production-like environment to provide the business functions specified in the high-level design. Finally, when the technical organization has completed these tests, the business or users perform *acceptance testing* to confirm that the system does, in fact, meet their business requirements.

Although many people dismiss the V Model, few have done so with the careful consideration that Brian Marick, author of *The Craft of Software Testing* (Prentice Hall, 1995), has demonstrated. In a debate with Dorothy Graham at the STAR (Software Testing Analysis and Review) 2000 East Conference and on his Web site (www .testing.com), Marick argued against the V Model's relevance.

The X Model

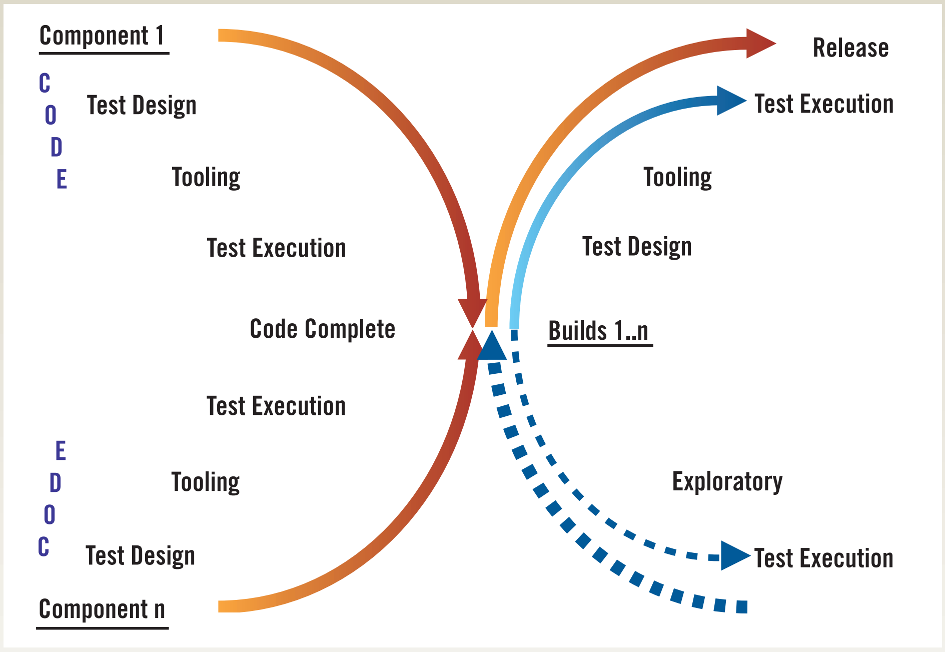
Marick raised issues and concerns, but is the first to acknowledge that he doesn't really propose an alternative model. I've taken the liberty of capturing some of Marick's thinking in what I've chosen to call, for want of a better term, the "X Model." Other than the sheer literary



The V Model

The V proceeds from left to right, depicting the basic sequence of development and testing activities. The model is valuable because it highlights the existence of several levels of testing and depicts the way each relates to a different development phase.

Robin F. Goldsmith *is the president of Go Pro Management Inc. consultancy in Needham, Mass., which he cofounded in 1982. A frequent conference speaker, he trains business and systems professionals in testing, requirements definition, software acquisition and project and process management. Reach him via www.gopromanagement.com.*



The X Model

The left side of the X Model depicts separate coding and testing of individual components, which then hand off frequently for integration into builds (upper-right quadrant) that also must be tested. Builds that pass their integration tests can be released either to users or for inclusion in larger integration builds.

brilliance of using *X* to contrast with *V*, there are some other reasons for choosing this particular letter: *X* often represents the unknown, and Marick acknowledged that his arguments didn't amount to a positive description of a model, but rather mainly to some concerns that a model should address. Prominent among these concerns is exploratory testing. Furthermore, with advance apologies lest the literary license offends, many of those sharing Marick's concerns undoubtedly could be described as members of Generation X. In addition, with further apologies to Marick, I've actually drawn an X-shaped diagram (see above) that I believe reasonably embodies the points Marick described in a somewhat different format.

Since an X Model has never really been articulated, its content must be inferred primarily from concerns raised about the adequacy of the V Model during the debate and in Marick's paper, "New Models for Test Development" (www.testing .com). My coverage of the X Model is brief because it hasn't been articulated to the extent of the V Model, and because I don't want to repeat some of the applicable general testing concepts discussed in Part 1.

Marick's quibble with the V Model centers largely on its inability to guide the total sequence of project tasks. He feels that a model must handle all the facts of development, such as handoffs, frequent repeated builds, and the lack of good and/or written requirements.

Coping With Handoffs and Frequent Build Cycles

Marick says a model should not prescribe behaviors that are contrary to actual acceptable practices, and I agree. The left side of the X Model depicts separate coding and testing of individual components, which then hand off frequently for integration into builds (upper-right quadrant) that also must be tested. Builds that pass their integration tests can be released either to users or for inclusion in larger integration builds. The lines are curved to indicate that iterations can occur with respect to all parts.

As seen in the graphic above, the X Model also addresses exploratory testing (lower-right quadrant). This is the unplanned "What if I try this?" type of ad hoc testing that often enables experienced testers to find more errors by going beyond planned tests. Marick didn't specifically address how exploratory testing fits in, but I'm sure he'd welcome its inclusion.

However, focusing on such low-level activities also creates concerns. A model isn't the same as an individual project plan. A model can't, and shouldn't be expected to, describe in detail every task in every project. It should, though, guide and support projects with such tasks. Surely, a code handoff is simply a form of integration, and the V Model doesn't limit itself to particular build cycle frequencies.

Marick and Graham agree that tests should be designed prior to execution. Marick advises, "Design when you have the knowledge; test when you have deliverables." The X Model includes test designing steps, as well as activities related to using various types of testing tools, whereas the V Model doesn't. However, Marick's examples suggest that the X Model isn't really a model in this regard; instead, it simply allows test-designing steps whenever, and if, one may choose to have them.

Planning Ahead

Marick questions the suitability of the V Model because it's based on a set of development steps in a particular sequence that may not reflect actual practice.

The V Model starts with requirements, even though many projects lack adequate requirements. The V Model reminds us to test whatever we've got at each development phase, but it doesn't prescribe how much we need to have. It could be argued that without requirements of any kind, how do developers know what to build? I would contend that the need for adequate requirements is no less an issue for the X or any other model. Though it should work in their absence, an effective model encourages use of good practices. Therefore, one of the V Model's strengths is its explicit acknowledgment of the role of requirements, while the X Model's failure to do so is one of its shortcomings.

Marick also questions the value of distinguishing integration tests from unit tests, because in some situations one might skip unit testing in favor of just running integration tests. Marick fears that people will blindly follow a "pedant's V Model" in which activities are performed as the model dictates them, even though the activities may not work in practice. While it may not be readily apparent in the graphic on page 44, I have endeavored to incorporate Marick's desire for a flexible sense of activities. Thus, the X Model doesn't require that every component first be unit tested (activities on the left side) before being integration tested as part of a build (upper-right quadrant). The X Model also provides no guidelines for determining when skipping unit testing would and would not be suitable.

Out of Phase

A model's major purpose is to describe how to do something well. When the elements the model prescribes are missing or inadequate, a model helps us recognize the deficiency and understand its price. By not only acknowledging, but perhaps advocating that system development can proceed without adequate requirements, the X Model may often promote practices that actually require extra effort. Similarly, one can choose to skip unit tests in favor of integration tests. However, benefits may be illusory. Generally, it takes much more time and effort to fix errors detected by integration tests than those found by unit tests.

A model that is predicated upon poor practices, just because they are common, simply ensures that we'll keep repeating those poor practices without chance for improvement. Moreover, people then assume that the poor practices are not only unavoidable, but necessary; and ultimately end up regarding them as a virtue.

For example, instead of learning how to better define business/user requirements, many developers simply declare it an impossible task because "users don't know what they want" or "requirements are always changing." Then, these developers may further declare that it's not only OK, but preferable, not to know the requirements because they're using a superior development technique, such as prototyping. While such iterative techniques are indeed valuable for confirming comprehension, in practice, they're often merely a way to skip straight to coding. Even iteratively, coding is at best a highly inefficient, ineffective, labor-intensive way to discover real business/user requirements. Iteration is far more valuable when used in conjunction with overall project planning and more direct requirements discovery methods.

Similarly, the X Model and exploratory testing were developed in part as reactions against methods that seem to involve excessive time writing various test documents, with a resulting shortage of time actually executing tests. (Somehow, some testing "gurus" have turned lack of structure and avoidance of written test plans into virtues). I'd be the last person to encourage documentation, or any form of busywork, for its own sake. I've seen too many examples of voluminous test plans that weren't worth the time to write. That doesn't mean that writing test plans is a bad practice, only that writing poor test plans is a bad practice. On the other hand, writing down important information can pay for itself many times over. It makes us more thorough, less apt to forget, and able to share more reliably with others.

In the next two segments of this article, I'll reveal how suitable structure can in fact get software developed quicker, cheaper *and* better, and present what I call the Proactive Testing Model, which my clients, students and I have found useful. I believe it fills the gaps in the V and X Models, and also provides significant additional benefits for both testers and developers.

Proactive Testing

Not V, not X: This combination model that intertwines testing with development provides a happy medium that can expedite your projects. Part 3 of 4. BY ROBIN F. GOLDSMITH AND DOROTHY GRAHAM

TESTING, WHILE AN INTEGRAL PART OF THE development process, doesn't often get the attention cast on other activities. Consequently, most testing models, despite their importance, are relatively unknown outside the communities that use them. The first two articles in this series, "The Forgotten Phase" (July 2002) and "V or X, This or That" (Aug. 2002) described the strengths and weaknesses of the time-honored V Model and the X Model we defined to describe the approaches currently being emphasized by some testing experts. This month, we introduce the Proactive Testing Model. We, as well as our clients and students, have found it to be useful for understanding and guiding effective software testing. Proactive testing draws value from both V and X Models, while reconciling their weaknesses.

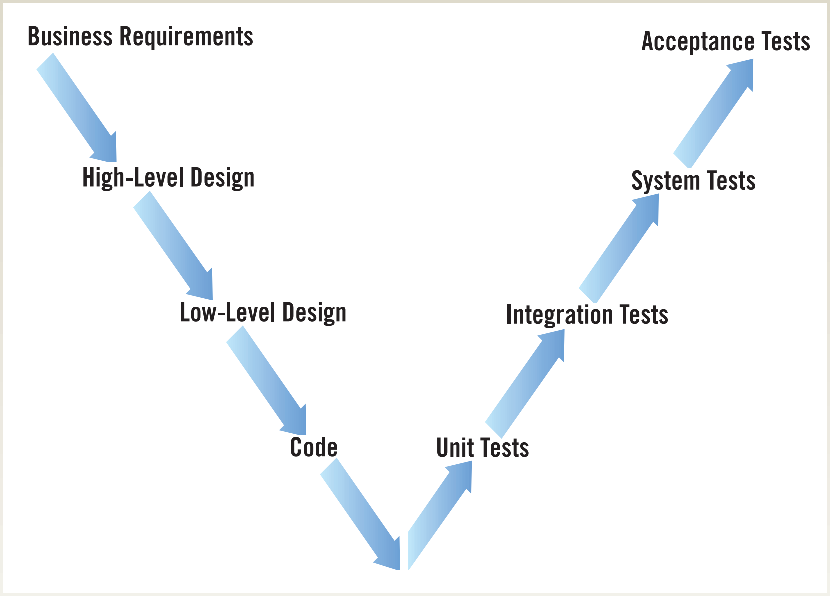
While proactive testing isn't perfect, it can provide significant benefits. Instead of waiting for perfection, we encourage using what you've got to gain as many benefits as you can. Moreover, we continually update the model as we discover issues that need to be addressed.

Because a single diagram isn't sufficient, we use three different graphical representations to represent the full range of concepts covered by the Proactive Testing Model. One of those charts appears on page 44. The other two charts will be published next month in Part 4, the final article in this series.

Development and Testing Combined

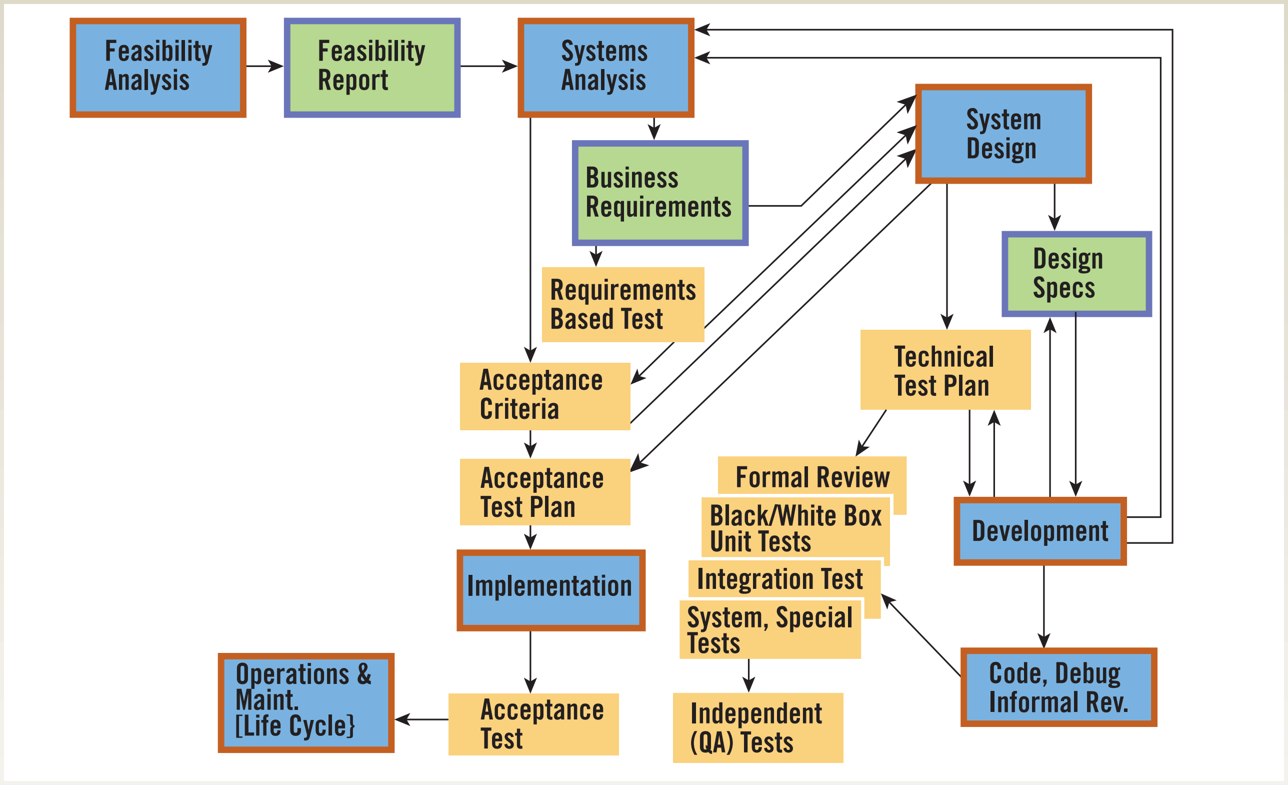
The Proactive Testing Model (page 43)

***Robin F. Goldsmith** is president of Go Pro Management Inc., Needham, MA, which he cofounded in 1982. A frequent speaker at leading conferences, he works directly with and trains business and systems professionals in testing, requirements definition, software acquisition, and project and process management. Reach him at robin@gopromanagement.com. **Dorothy Graham** is the founder of Grove Consultants in the UK, which provides consultancy, training and inspiration in software testing, test automation and Inspection. She is coauthor with Tom Gilb of* Software Inspection *(Addison Wesley, 1993) and coauthor with Mark Fewster of* Software Test Automation *(Addison Wesley, 1999). In 1999, she was awarded the IBM European Excellence Award in Software Testing. She is on the board of* STQE *magazine, testing conferences and the British Computer Society Information Systems Examination Board (ISEB) Software Testing and Business System Development Boards.*



The V Model

The V proceeds from left to right, depicting the basic sequence of development and testing activities. The model is valuable because it highlights the existence of several levels of testing and depicts the way each relates to a different development phase.



The Proactive Testing Model

Proactive testing brings together development and testing lifecycles, identifying key activities from inception to retirement, and delineating the point in the project's lifecycle at which they provide the optimum value.

brings together development and testing lifecycles, identifying key activities from inception to retirement. When the activities are not performed or are performed at inappropriate points, the likelihood of success is compromised.

Thus, for example, systems are developed more effectively when business/user requirements are defined than when they are not. In fact, we'd suggest that it's impossible to develop a meaningful system without knowing the requirements. Moreover, systems are developed more effectively when business/user requirements are defined prior to (as opposed to after) design and development of the affected system components.

The Proactive Testing Model embodies several key concepts:

Test Each Deliverable

Each development deliverable should be tested by a suitable means at the time it is developed. Program code is not the only deliverable that needs to be tested. The shaded box outlines (see diagram above) are meant to show that we also need to test feasibility reports, business requirements and system designs. This is consistent with, but more explicit and expansive than, the level-to-level verification that is sometimes implied on the left (development) side of the V Model (see page 42).

For example, coauthor Goldsmith has identified and presents a popular seminar describing more than 21 techniques for testing the accuracy and completeness of business/user requirements. If they use any at all, most organizations use one or two weak techniques. The Proactive Testing Model includes two test-planning techniques that also happen to be among the more powerful of the 21-plus ways to test requirements.

One of these methods focuses on developing requirements-based test cases. This not only creates an initial set of tests to run against the code when it's delivered later on, but also confirms that the requirements are testable. These tests may be employed by users for acceptance testing and/or by the development organization for technical testing. Many in the testing community consider testability the primary, if not the only, issue with requirements, even to the extent of saying that writing a test case for each requirement is all that is needed. While testing each requirement is necessary, it isn't sufficient. Requirements-based tests are only as good as the requirements. A requirement can be completely wrong, yet still be testable. Moreover, one can't write test cases for requirements that have been overlooked.

The second technique involves defining acceptance criteria, which are what the business/user needs to have demonstrated before they're willing to rely on the delivered system. Acceptance criteria don't just *react* to defined requirements. Instead, they come *proactively* from a testing perspective and can help to reveal both incorrect and overlooked requirements.

Similarly, system designs should be tested to ensure accuracy and completeness before implementing them by writing code. Organizations tend to be better at testing designs than requirements, but still generally use only a few of the more than 15 ways to test designs that Goldsmith presents. One of the most powerful design tests comes from planning how to test the delivered system, which is most effective when performed during the design phase, prior to coding.

Plan and Design Tests During the Design Phase

The design phase is the appropriate time to do test planning and design. Many organizations either don't plan or design their testing or do it immediately prior to test execution, in which case, tests tend to demonstrate only that programs work the way they were written—not the way they *should* have been written.

Effective test planning can increase the number of errors that can be found economically.

There are two major types of tests, each of which needs a plan. As in the V Model, *acceptance tests* are defined earliest to be run latest, demonstrating that the delivered system meets the business/user requirements from the business/user perspective.

Unlike the V Model, the Proactive Testing Model recognizes three components of an acceptance test plan. Two of these components come in conjunction with defining the business/user requirements: defining requirements-based tests and defining acceptance criteria. However, the third component must wait until the system is designed, because the acceptance test plan really is formed by identifying how to use the system as designed to demonstrate that the acceptance criteria have been met and that requirements-based tests have been executed successfully.

Technical tests are tests of the developed code, such as the dynamic unit, integration and system tests identified in the V Model. In addition, proactive testing also reminds us to include static reviews (such as walk-throughs and inspections) and independent QA tests (which typically follow the system test to provide a final check from the technical organization's notion of the user's perspective), as well as special tests. We use the term *special tests* as a catch-all definition for tests such as load, security and usability testing that aren't specifically directed toward the application's business functionality.

Technical tests primarily demonstrate that the code as written conforms to the design. Conformance means that the system does what it should and doesn't do what it shouldn't. Technical tests are planned and designed during the design phase, to be carried out during development, primarily by the technical organization (not by users).

Intertwine Testing with Development

Proactive testing intertwines test execution with development in a code-test, code-test pattern during the development phase. That is, as soon as a piece of code is written, it should be tested. Ordinarily, the first tests would be unit tests, since the developer can carry out these tests to find errors most economically. However, as with the X Model, proactive testing recognizes that a piece of code may also need relevant integration tests and possibly some special tests shortly after the code is written. Thus, with respect to a given piece of code, the sequence of these various tests follows the V Model; but they're intertwined with development of the particular code components, not segregated into separate test phases.

The technical test plans should define this intertwining. The major approach and structure of the testing should be defined during the design phase and supplemented and updated during development. This especially affects code-based tests, which could be testing units or integrations. In either case, the test will be more efficient and effective with a bit of planning and design prior to execution, which of course also facilitates reuse.

Test planning and design are shown on diagrams in next month's article, which addresses additional aspects of this intertwining. Intertwining enables us to detect and eject (fix) more errors sooner after they are injected (or created), reducing error correction time, effort and cost. Note that the test plans themselves, not the Model, specifically determine which tests to run, and in which sequences. Similarly, effective test planning can increase the number of errors that can be found economically, but the plans should leave room for the additional ad hoc exploratory testing that often enables experienced testers to find more defects than their written test scripts would have found alone.

Keep Acceptance and Technical Testing Independent

Acceptance testing should be kept independent of technical testing so that it serves as a double-check to ensure that the design and its implementation in code meet business needs. Acceptance testing is run either as the first step of the implementation phase or the last step of the development phase—the same point in the process, regardless of its name.

The Proactive Testing Model advocates that acceptance and technical testing follow two independent paths. Each defines how to demonstrate that the system works properly from its own perspective. When the independently defined business/user acceptance tests corroborate the design-oriented technical tests, we can be confident that the system is correct.

Develop and Test Iteratively

Development and testing proceed together iteratively. At any point in the process, you can revisit prior phases and steps to correct their deliverables. This could entail fixing errors, eliminating superfluous elements or adding newly discovered ones; the Model does not dictate the size of the system portion involved. This is consistent with informed use of the V Model. However, proactive testing makes the iteration explicit.

Proactively Find Real WIIFMs

Proactive testing consciously seeks to identify and deliver the WIIFM (What's In It For Me) values that make users, developers and managers want to use the testing techniques. The WIIFMs that proactive testing reveals are different from the rationales traditionally given for doing reactive testing.

Instead of merely prioritizing, citing the lower costs of finding errors earlier or emphasizing the fact that testing is important to ensure better quality, proactive testing represents a wholly different attitude toward testing. (Heaven forbid, a paradigm shift!) Throughout the development process, we repeatedly use (and communicate about) testing in ways that enable developers, managers and users to save time and make their jobs easier.

Traditionally, developers often consider (reactive) testing to be an added burden that gets in the way of meeting programming deadlines. However, when we proactively define how to test the program *before* it's written, I've found that the developer can reasonably finish the program in at least 20 percent less time. Although developers are seldom conscious of how they actually spend their time, on reflection they usually realize that a large part of new development time is spent reworking the code they've already written. Conservatively, designing tests before coding can prevent at last half of the rework in the following ways:

- Designing tests is one of the more powerful ways to test a design, and thus help the developer avoid writing code that will need to be changed. Often the tests make design logic flaws apparent. In other instances, writing tests can reveal ambiguities. After all, if you can't figure out how to test the program, the developer probably won't be able to determine how to correctly program it.
- Tests created prior to coding show how the program ought to work, as opposed to tests created after the program has been written, which often just show that the program works the way the developer wrote it. Such tests help the developer catch and correct errors right away.
- When tests are defined prior to coding, the developer can begin testing as soon as the code is written. Moreover, she can be more efficient, executing more tests at a time, because her train of thought won't be interrupted repeatedly to find test data.
- Even the best programmers often interpret seemingly clear design specifications differently from the designer's intentions. By having the test input and expected result along with the specification, the developer can see concretely what is intended, and thereby code the program correctly the first time.

Proactively defining how to test a program before coding is a real WIIFM that developers greatly appreciate once they experience the technique. Not only does it *save* them time, but it reduces the rework they tend to hate the most.

Next month, in the final article of this series, we'll describe how Proactive test planning and design can prevent showstoppers and overruns by letting testing drive development.

Test-Driven Development

What part of the system do you test first? The Proactive Testing Model helps you prioritize, manage a project-level plan and steer clear of risks. Part 4 of 4. BY ROBIN F. GOLDSMITH AND DOROTHY GRAHAM

AS DO DEVELOPERS, MOST TESTERS RECOGnize that models are valuable aids; however, testing doesn't often get the attention cast on other activities in the development lifecycle. The V Model is probably the best-known testing model, but many testers are unfamiliar with it. And the V Model isn't free from criticism: One of the more vocal critics of the V Model is Brian Marick, author of *The Craft of Software Testing* (Prentice Hall, 1995). In part 2 of this series, "V or X, This or That" (Aug. 2002), we described the "X Model," covering points that Marick felt should be present in a suitable testing model. Part 1, "The Forgotten Phase" (July 2002), discussed the V Model itself.

In Part 3, "Proactive Testing" (Sept. 2002), we introduced the Proactive Testing Model, which we believe incorporates the strengths and addresses the weaknesses of both V and X Models. In this final installment, we delineate the benefits of letting testing drive development.

Proactive Test Planning

The diagram "A Sophisticated Standard" (see page 53) depicts the oft-followed test-planning structure suggested by IEEE standard 829-1998. Some critics view the standard as a counterproductive paper-generator. While such practices are common, they need not be. The graphic overview adds value to the text-only standard, making it easier to understand and apply.

The standard suggests conceptualizing test plans at several different levels, either as individual documents or sections within a larger document. The master, overall system test plan is a management document that ultimately becomes part of the project plan.

We use a simple heuristic to drive down to lower and lower levels of test-planning detail: "What must we demonstrate to be confident that it works?"

The master test plan identifies a set of test plans that, when considered together, demonstrate that the system as a whole works properly. Typically, a detailed test plan describes each unit, integration, special (a catch-all term for tests that aren't specifically driven by the application, such as stress, security and usability tests) and system test. In turn, each detailed test plan identifies a set of features and functions that, in concert, demonstrate that the unit, integration, special function or system is working properly. For each feature and function, a test design specification describes how to demonstrate that the feature or function works properly. Each test design specification identifies the set of test cases that together indicate that the feature or function works.

The test-planning structure provides a number of benefits. While it's easy to see why many testers think that the IEEE standard requires voluminous documentation for its own sake, that approach provides no value, and we don't endorse it. However, we don't reject written test plans out of hand, as some X Model advocates seem to do. Instead, we suggest recording important test-planning information as a memory aid and to facilitate sharing, reuse and continual improvement.

For many people, test plans are primarily a collection of test cases—a collection that can grow quite large and unmanageable. One can see from the diagram, though, that the standard's structure provides immediate value by helping to organize and manage the test cases.

We can proactively define reusable test design specifications, identifying how to test common situations. These specifications can prevent enormous amounts of duplicated effort, enabling us to start credible testing with little delay. Similarly, the structure helps us define reusable test cases and selectively allocate resources. Moreover, the structure and test design specifications make it easier to reliably recreate test cases when necessary.

Proactive Prioritization

While risk prioritization should be a part of any test approach, neither the V nor X Models makes it explicit. Moreover, the proactive risk analysis methods we use are far more effective than the traditional ones we've seen applied elsewhere.

Robin F. Goldsmith *is president and co-founder of Go Pro Management Inc., in Needham, Mass. Reach him at www.gopromanagement.com.*

Dorothy Graham *is the founder of Grove Consultants in the UK. In 1999, she was awarded the IBM European Excellence Award in Software Testing.*

The proactive method improves testing in two ways. First, traditional approaches tend to assign risk to each test as it's identified. With nothing to compare to, each test tends to be denoted as high risk. The proactive test-planning structure, however, quickly reveals the available options, so that we can prioritize with respect to all our choices. Second, we can eliminate the common reactive technique of rating the risks of the tests that have been defined. Obviously, no risks are assigned to tests that have been overlooked, and overlooked tests are often the biggest risks. In contrast, proactive risk analysis enables us to identify critical risks that the reactive approach overlooks—and *then* to define tests to ensure that these risks don't occur.

While we use proactive test planning at each level, the most important task involves identifying and prioritizing project-level risks in the master test plan. In particular, this proactive technique helps us anticipate many of the traditional showstoppers. Every developer we know can readily name unexpected problems that stopped the project at the worst possible time—usually right before or after implementation.

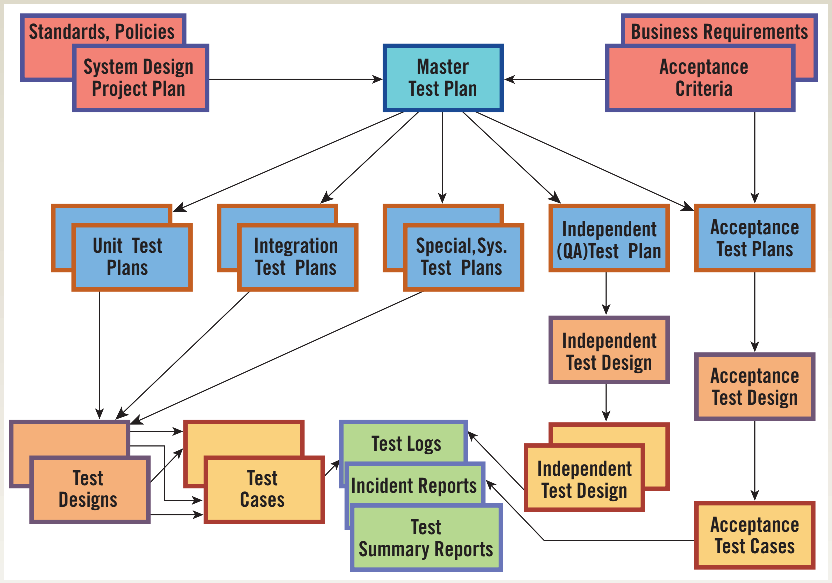
When working with a project team, we invariably find that they're able to use proactive risk analysis to identify a large number of potential showstoppers. The typical group reports that traditional project and test planning would have overlooked about 75 percent of these.

Once we've identified and prioritized risks, we can define the pieces of the system that need to be tested earlier. These often aren't the elements that the organization would ordinarily have scheduled to build early.

Whereas the typical development plan develops entire programs, often in job execution sequence, the Proactive Testing Model defines the system's parts—units or integrations—that must be present to test the high risks. By coding these modules first, *testing drives development*.

Building and testing these high-risk pieces early helps developers to catch problems before they've done additional coding that would have to be redone.

The effect is even more profound with respect to special tests. Rather than being employed within the system test (such as load and security tests), or often overlooked entirely (such as tests of training, documentation and manual procedures), special tests also merit detailed test plans and resulting identity. Therefore, each build may contain components that traditionally would have been addressed by unit, integration and system tests.



A Sophisticated Standard

The test-planning structure suggested by IEEE standard 829-1998 suggests conceptualizing test plans at different levels, either as individual documents or sections within a larger document.

An A-B-C Example

To get a feel for test-driven development, let's look at a simplified example. Assume that we have a system consisting of three programs, A, B and C, that in production would be executed in the A-B-C sequence. It's likely that the programs would also be developed and tested in the same sequence.

However, our risk analysis reveals that the integration of B and C poses the highest risk. Therefore, programs B and C should be built and unit-tested first. Since problems discovered in B or C could also affect A, finding these problems before A is coded can help prevent having to rewrite parts of A.

The next highest risk is program A's message capacity. In a typical development plan, we might build all of program A and then unit test it—but as a whole, A could be large and difficult to test and debug. By breaking A into two units, one dealing with the messaging and the other with the remainder of the program's functions, we can more immediately test the messaging capacity while it's still relatively easy to find and fix errors.

Next, we need an integration test to ensure that the two parts of program A function together correctly. And finally, now that the B–C and A–subparts integrations have been tested, we can address the third-highest risk: testing the integration of all three programs.

Note that the Proactive Testing Model doesn't dictate the specific sequence of tests. Rather, it guides us to plan the sequence of development and testing based on the particulars of each project to quickly and inexpensively arrive at the desired result: higher-quality software.

Developers who've worked with the Proactive Testing Model can immediately identify the nature and magnitude of problems that this approach helps them avoid. They know that these problems are often the cause of delayed, over-budget projects. When managers and developers realize how proactive testing helps them become aware of these risks—and prevent them—it's a WIIFM (What's In It For Me?) they can readily embrace.