6th INTERNATIONAL SOFTWARE QUALITY WEEK

Quality Week 1993

Tutorial Notes

25-28 May 1993

The Sheraton Palace Hotel
2 New Montgomery Street
San Francisco, California 94105

Software Research Institute

ALL RIGHTS RESERVED. *No part of this document may be reproduced in any form, by photostat, microfilm, retrieval system, or by any other means known or hereafter invented without written permission from Software Research, Institute*

Copyright © 1993 by **Software Research, Institute**

Software Research Institute
625 Third Street
San Francisco, CA 94107-1997

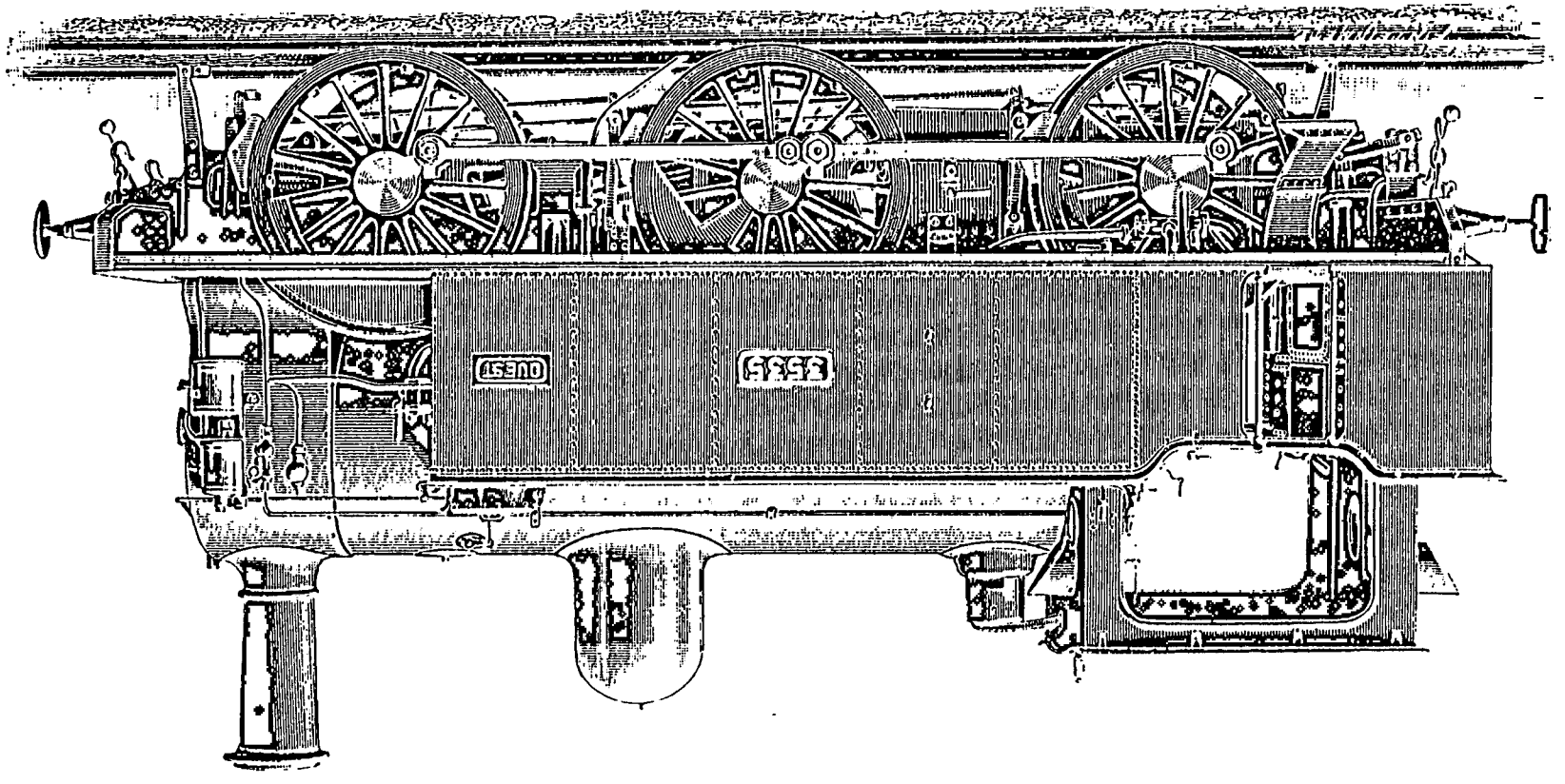
• Phone: (415) 957-1441 • Toll Free: (800) 942-SOFT
• FAX: (415) 957-1441 • E-mail: qw@soft.com

"Overview of Software Test Techniques"

TUTORIAL "A"

Dr. Boris Beizer
ANALYSIS

Dr. Boris Beizer is best known for his books on software testing and his seminars based on them: *Software Testing Techniques* and *Software System Testing and Quality Assurance*. He received a Ph.D. in Computer Science from the University of Pennsylvania and has been in the computer industry for more than 30 years. An independent consultant with Analysis, Inc., he works with leading software development organizations on software testing and quality assurance problems.


OUEST

READ THIS FIRST

1. OBJECTIVES.

This is an overview of the testing field. Its purpose is to provide you with the technical and conceptual vocabulary of testing. Testing has emerged as a field within software engineering and has acquired a big vocabulary. It has progressed, in the past 20 years, from intuition to science—from personal heuristics to well-understood practices rooted in theory and confirmed by use and experiments. The objectives of today's lecture are to:

1. Expose you to the ideas and terminology so you can begin learning the field.

2. Give you a perspective from which you can judge for yourself the parts of this field that are likeliest to be useful to you—in the short and long term.

3. Understand enough of the concepts and terminology so you can talk to, and understand those who know the details.

4. Help you plan how to best learn this technology.

5. Establish a basis that will enable you to communicate with people who are using this technology.

It is not the purpose of this overview to teach you the material in depth. This is an overview of an introductory testing course and an advanced testing course whose combined duration would be about 10 days. The material is based on my two books: *Software Testing Techniques—Second Edition,* and *Software System Testing and Quality Assurance*—published by Van Nostrand Reinhold.

2. TERMINOLOGY AND DEFINITIONS.

The terminology and definitions used are the mainstream terms of testing and follow the IEEE standard glossary for software engineering. Much of the time is spent on the testing vocabulary—the definitions come fast and heavy—some say that it's like drinking from a fire hose. A complete set of definitions is in the glossary of *Software Testing Techniques* (Second Edition). If I use a term with which you are not familiar or if you can't remember the definition, you'll find the definition in *Software Testing Techniques*. But don't be timid about asking.

3. QUESTIONS AND DISCUSSION.

There's a 1/2 hour discussion buffer built-in to this overview. The discussion time is distributed (e.g., used) throughout the session. Don't be timid about asking so-called "dumb" questions—they're the most important kind you can ask. If you don't understand, or if I misspoke, then others also may have misunderstood, and I'd rather clear up a problem as it occurs.

4. SCHEDULE AND BREAKS.

There is a 10 minute break every hour at 10 minutes to the hour (e.g., 8:50, 9:50, etc.) except the last hour, which runs until noon. I will start promptly on time, on the hour, so please cooperate.

5. NOTES VERSUS LECTURE.
Although I generally follow the notes, don't expect me to lecture precisely in that order. The notes are provided for your convenience and later review—they are not a script for the seminar. All the material in the notes will usually be covered but the exact sequence may vary depending on questions.

6. TRANSPARENCY NUMBERING.
Every transparency is numbered and dated. The order of the transparencies in your notes are not sequential because they are taken from the much larger set of transparencies that I use in my 3-5 day seminars on testing, advanced testing, integration testing, system testing, and software quality assurance. If there is any question about the order, please refer to the detailed index that follows. The pages are numbered in the lower right-hand side. If you have any questions or comments on these notes, please refer to the lecture note page number and the transparency number.

7. ERRORS AND TYPOS.
Please bring typographical or technical errors to my attention during the breaks or after the end. Mark them on your note set and show them to me so I can schedule a correction—unless, of course, the error is such that the transparency doesn't make sense or might be misleading: then tell me immediately. I thank you for all such errors brought to my attention. You are my test group and primary QA.

8. VIDEOTAPING AND/OR VOICE RECORDING.
Any form of recording is forbidden. The sole exception to this rule is made for blind students who can make audio tapes for their personal use. Such recordings may not be reproduced.

9. COPYRIGHT NOTICE. This lecture note set is the sole property of Boris Beizer and is a copyrighted document. Your copy has been provided under copyright law. Please note the copyright notification on each module and on each transparency. Copying these lecture notes without the author's permission is a violation of U.S. and International laws. Use of these lecture notes to present a course or seminar by other than the author is also a copyright violation.

OVERVIEW OUTLINE

MORNING	**08:00-12:00**	**or**	**13:00-17:00**
OVERVIEW, DEFINITIONS	08:00-08:50		13:00-13:50
BREAK	08:50-09:00		13:50-14:00
UNIT, INTEGRATION, AND SYSTEM TESTING	09:00-09:50		14:00-14:50
BREAK	09:50-10:00		14:50-15:00
TEST TECHNIQUES	10:00-10:50		15:00-15:50
BREAK	10:50-11:00		15:50-16:00
TOOLS AND PERSPECTIVE	11:00-12:00		16:00-17:00

SECOND LEVEL OUTLINE

OVERVIEW, DEFINITIONS	**08:00-08:50**
OBJECTIVES AND OVERVIEW	0540-1187
THE PURPOSE OF TESTING	0130.5-1235
DEFINITIONS	1205-1270
UNIT, INTEGRATION, AND SYSTEM TESTING	**09:00-09:50**
BEHAVIORAL VERSUS STRUCTURAL TESTING	1700-1715
UNIT TESTING OVERVIEW	1300-1340
INTEGRATION TESTING OVERVIEW	3030-3050
SYSTEM TESTING OVERVIEW	1500-1530
TESTING TECHNIQUES	**10:00-10:50**
CONTROL-FLOW TESTING (PATH TESTING)	1345-2260
TRANSACTION FLOW TESTING	2800-2831
DATA FLOW TEST STRATEGIES	4601-4640
TOOLS AND PERSPECTIVE	**11:00-12:00**
MANUAL TESTING LIMITATIONS	7900
TEST TOOLS OVERVIEW	7501
COVER CERTIFIERS	7800-7805
DRIVERS	7750-7806
TEST DESIGN AUTOMATION TOOLS	7705-7745
PERSPECTIVE ON TESTING	6900-6912

DETAILED OUTLINE

OVERVIEW, DEFINITIONS	**08:00-08:50**
OBJECTIVES AND OVERVIEW	**0540-1187**
Testing, a mature technology	1180
Seminar objectives	0540
Some signs of maturity	1180.1
The dimensions of testing	1181
That's Just Theory? (I)	1184.1
That's Just Theory (II)	1182
Fundamental testing techniques	1183
Auxiliary testing techniques	1184
Application scope	1185
Testability design is the goal	1186
Techniques are tool-intensive	1187
THE PURPOSE OF TESTING	**0130.5-1235**
The four phases of test awareness	1220
We must change objectives	1160
Risk-driven process, nightmare list	1161
Implications for clean and dirty tests	1230
The Pesticide Paradox	1235
Limitations of testing	1170
The testing triad and then some	1201
Overview of test techniques	0130.5
DEFINITIONS	**1205-1270**
Unit, component, integration	1205
Don't confuse integration with system testing	1208
System, structural, functional, testing	1210
Clean test, dirty test	1211
Equivalency, progressive, regression testing	1240, 1245, 1250
A universal experience (manual regression testing)	1251
Single-thread, multi-thread testing	1255, 1260
Stress testing defined	1271
Stress testing	1270
UNIT, INTEGRATION, SYSTEM TESTING	**09:00-09:50**
STRUCTURAL vs BEHAVIORAL	**1700-1715**
Behavioral versus structural testing	1700
Structural testing advantages	1706
Behavioral testing advantages	1711
Structural testing disadvantages	1707
Behavioral testing disadvantages	1712
Structural testing effectiveness	1708
Behavioral testing effectiveness	1713
There is no conflict	1715
UNIT TESTING OVERVIEW	**1300-1340**
Unit testing is the foundation	1300a, 1300b
Unit testing as a model	1310

Prerequisites to unit testing	1315
Test techniques and mental models	1335
Coverage concepts	1340
INTEGRATION TESTING OVERVIEW	**3030-3050**
Goals of integration testing	3030
What's this jazz about trust?	3035.1
The debugging process	3035.2
Integration with and without trust	3035.3
Integration is a process	3040
Prerequisites to integration (overview)	3050
SYSTEM TESTING OVERVIEW	**1500-1530**
System testing goals	1500
Prerequisites	1505
Bottom-line Behavioral testing	1515
System tests	1529
Other system tests	1530
TESTING TECHNIQUES	**10:00-10:50**
CONTROL-FLOW TESTING (Path Testing)	**1345-2260**
Why Testers Should know Path Testing	1345
Unit testing for systems testers?	1345.1
Path testing consists of selecting	2010
Control Flow testing strategies	2011
Parts of path test design	2015
Select covering paths	2015.1
Path selection criteria	2081
Sensitize the paths	2015.2
Instrument the paths	2015.3
Path Testing review	2260
Path Testing tools	2261
TRANSACTION FLOW TESTING	**2800-2830.5**
The concept	2800
Principles	2805
Transaction flow components	2815
Transaction flow example, chart	2822.2
Path Selection criteria	2825
Test design procedures--overview	2830
Execution and test driving	2830.5
Automation tools	2831
DATA FLOW TEST STRATEGIES	**4601-4640**
A question	4601
Data Flow Testing Criteria (structural)	4610
Data Flow Testing Criteria (behavioral)	4610.5
Data Flow Definitions	4605
Examples of Data Objects	4605.1
Data Flow Testing is Very Old	4611

Theory and Practice	4612
Data Flow Test Criteria Overview	4614
Data Flow Test Criteria Relative Strength	4621
Relative Criteria Cost-Effectiveness	4630.1
Tools and automation status	4640
TOOLS AND PERSPECTIVE	**11:00-12:00**
MANUAL TESTING LIMITATIONS	7900
Why Automated testing is mandatory	7901
TEST TOOLS OVERVIEW	7501
A basic tool kit	1188
COVER CERTIFIERS	**7800-7805**
Coverage Analyzers	7800
Path cover analyzer	7810
Side benefits of coverage tools	7801
Present status of commercial coverage tools	7805
DRIVERS	**7750-7806**
Test Drivers	7750
Overview of driver function	7760
Setup phase	7770
Execution phase	7780
Post-testing phase	7790
Status of execution automation tools	7806
TEST DESIGN AUTOMATION TOOLS	**7705-7745**
Prerequisites	7705
Objectives	7710
Test design automation tools	7715
Capture/replay tools	7920
Structural test generators (hypothetical)	7725
Structural test generator (real)	7725.1
Data flow test generators	7727
Specification Based Generators	7730
Regression and maintenance tools	7734
Not Enough Tools?	1191.1
Status of test design automation tools	7745
PERSPECTIVE ON TESTING	**6900-6912**
Perspective on testing	6900
Techniques are tool-intensive	6902
Tool building versus tool buying	6904
Tool Building and Buying	6905
Tools and automation importance (Japan survey)	6906
Realistic payoff projections	6908.1
The tool penetration problem	6910
The penetration problem solution	6912
The message recapitulation	1189

SOFTWARE TESTING AND QUALITY ASSURANCE SEMINARS—BORIS BEIZER

AN OVERVIEW OF TESTING

THIS IS A COPYRIGHT NOTICE. NO PART OF THESE GRAPHICS MAY BE REPRODUCED OR USED IN ANY FORM BY ANY MEANS SUCH AS GRAPHICS, ELECTRONIC, OPTICAL, MECHANICAL, INCLUDING BUT NOT LIMITED TO PHOTOCOPYING, RECORDING, SCANNING, CONVERSION TO OTHER MEDIA, TRANSMISSION, OR STORAGE IN A COMPUTER OR AN INFORMATION STORAGE AND RETRIEVAL SYSTEM — WITHOUT THE AUTHOR'S WRITTEN PERMISSION.

BORIS BEIZER, ANALYSIS – 1232 GLENBROOK ROAD, HUNTINGDON VALLEY, PA 19006
215-572-5580

054c 08/29/92 ©B²

TESTING--A MATURE TECHNOLOGY

- A STABLE TECHNICAL VOCABULARY-- *SOFTWARE TESTING TECHNIQUES--2ND EDITION, IEEE SOFTWARE ENGINEERING GLOSSARY.*
- RECOGNIZED BODY OF KNOWLEDGE, TECHNIQUES, CONCERNS.
- MANY TECHNICAL JOURNALS-- *IEEE-SE, ACM TOSEM, IEE-SEJ, ACM-SIGSOFT, ITEA, JSTVR...*
- GRADUATE AND UNDERGRADUATE COURSES, SEMINARS.
- 20+ YEARS OF APPLICATION AT IBM, AT&T, ETC.

1180 08/29/92 ©B²

OBJECTIVES

- **VOCABULARY:** INTRODUCE THE VOCABULARY AND THE CONCEPTS THAT TESTERS USE TO TALK ABOUT TESTING.
- **OVERVIEW OF TEST TECHNIQUES:** INTRODUCE THE BASIC TEST TECHNIQUES THAT RESEARCH AND PRACTICE HAVE DEMONSTRATED TO BE THE MOST PRODUCTIVE AND ON WHICH AUTOMATED TESTING TOOLS ARE BASED.
- **PARTS OF TESTING:** DISCUSS THE THREE MAIN PARTS OF TESTING—UNIT/COMPONENT, INTEGRATION, SYSTEM TESTING—AND THE TEST TECHNIQUES APPROPRIATE TO EACH.
- **TECHNOLOGY:** TOOLS, ALGORITHMS, AUTOMATION.

0540 08/29/92 ©B²

SIGNS OF MATURITY

COMPARISON OF 1ST AND 2ND EDITIONS OF SOFTWARE TESTING TECHNIQUES

PUBLICATION YEAR	1982	1990
PAGE COUNT	290	550
REFERENCES	132	406
TECHNICAL TERMS	95	350
LITERATURE %	75%	20%
MOTIVATIONAL CONTENT %	20%	2%
TEXTBOOK USE	0	50+
COMPETITORS	1	25+
FIRST PRINTING	1,500	4,000
ORIENTATION	MANUAL	AUTOMATION
INTEGRATION	COLLECTED IDEAS	INTEGRATED IDEAS

1180.1 08/29/92 © B²

THE DIMENSIONS OF TESTING

- FUNDAMENTAL TECHNIQUES--CONTROL FLOW, DATA FLOW, DOMAIN
- AUXILIARY TECHNIQUES--TRANSACTION FLOW, SYNTAX, STATE
- APPLICATION SCOPE--UNIT/COMPONENT, INTEGRATION, SYSTEM
- SPECIALIZED TESTING CONCERNS--RECOVERY, SECURITY, CONFIGURATION, PERFORMANCE
- DEVELOPMENT PHASE--FROM DESIGN TO MAINTENANCE
- ORIENTATION--BEHAVIORAL VERSUS STRUCTURAL

1181 08/27/92 © B²

"THAT'S JUST THEORY"???

- *EVERY* TECHNIQUE DISCUSSED HERE HAS BEEN AND IS BEING USED DAILY AT LEADING SOFTWARE DEVELOPERS.
- AT&T, BOEING, HP, HITACHI, IBM, INTEL, MICROSOFT, MOTOROLA, NCR, NOVELL, SAS, SIEMENS, SUN, TANDEM, XEROX.
- NO ONE USES EVERY TECHNIQUE, BUT EVERY TECHNIQUE IS USED BY SOMEONE.
- BRANCH & STATEMENT COVER IS THE DE-FACTO MINIMUM TEST STANDARD.
- SOME TECHNIQUES (BRANCH/STATEMENT, DATA FLOW, STATE) HAVE BEEN IN REGULAR USE FOR OVER 25 YEARS.

"THEORY"?? IT'S ALL PRACTICE, FOLKS.
PRACTICE BASED ON THEORY AND EXPERIENCE.

1184.1 08/29/92 © B²

"THAT'S JUST THEORY" ???

- All engineering advances by the interplay of experience and theory. Testing is no exception.
- Testing theory is mature and provides sound guidance to practices and tools.
- "Intuitive" testing is no better than intuitive aircraft design--would *you* fly in one?
- Practice predates theory by over two decades--and theory is being corroborated by practice and experience.
- The intuitive tester won't break new grounds, over-tests many areas, and misses some tests altogether.

1182 08/29/92 © B²

FUNDAMENTAL TECHNIQUES

- **CONTROL-FLOW TESTING**—based on the control flowgraph--a simpler and more mature version of the flowchart -- leads to statement and branch cover as fundamental strategies.
- **DATA-FLOW TESTING**—based on data flows, either in unit testing or supported by data-flow design diagrams.
- **DOMAIN TESTING**--ties numerical parameters to requirements, tests consistency and completeness of specifications.

These techniques are related to one another by a cohesive theory
All have deep testability design implications.

1183 08/29/92 © B²

AUXILIARY TECHNIQUES

- **TRANSACTION FLOW TESTING** -- is to system testing what control-flow testing is to unit/component testing.
- **SYNTAX TESTING** -- a technique especially suited to testing user and operator interfaces--beats key-pounding every time.
- **FINITE-STATE MACHINE TESTING** -- a technique especially suited to testing menu-driven interfaces .

These techniques create a detailed agenda for testing at all levels and have a direct bearing on testability design

1184 08/29/92 © B²

APPLICATION SCOPE

- UNIT/COMPONENT TESTING--BY PROGRAMMERS, FUNDAMENTAL TECHNIQUES, TEST TOOLS, SOLID THEORETICAL FRAMEWORK.
- INTEGRATION TESTING--DOMAIN TESTING, DATA-FLOW TESTING, CALL TREE COVER. PROBES THE INCONSISTENCIES AT COMPONENT INTERFACES. INTEGRATION STRATEGIES AND GUIDANCE.
- SYSTEM TESTING--TRANSACTION FLOW, SYNTAX TESTING, STATE TESTING, STRESS TESTING.
- MAINTENANCE--CURRENT AREA OF INTENSIVE R&D. PROTOTYPE TOOLS AND EARLY, USEFUL, THEORY RESULTS.
- DE-BUGGING--NEW ALGORITHMS PROMISE TO REVOLUTIONIZE DEBUGGING.

1185 08/29/92 © B²

TESTABILITY DESIGN IS THE GOAL

- BUG PREVENTION IS BETTER THAN BUG DETECTION
- TESTABILITY DESIGN IS THE NEW OBJECTIVE -- BEYOND STRUCTURE
- STRUCTURED DESIGN RULES FOLLOW FROM TESTABILITY
- TESTABILITY IS NOT A VAGUE WISH BASED ON INTUITION AND OPINION -- TESTING TECHNOLOGY GIVES CLEAR, QUANTITATIVE DIRECTIONS ON HOW TO DESIGN TESTABLE SOFTWARE

THIS IS THE TESTABILITY DECADE!!!!

1186 08/29/92 © B²

TECHNIQUES ARE TOOL-INTENSIVE

- ALL BUT THE SIMPLEST TECHNIQUES NEED TOOL SUPPORT
- "MANUAL TESTING" IS SELF-CONTRADICTORY
- TESTING IS NOT KEY-POUNDING
- THAT'S THE WAY IT WAS FOR HARDWARE TESTING. SHOULD WE EXPECT DIFFERENTLY FOR SOFTWARE?

IMPLICATION -- CAPITAL INVESTMENT, LONG-TERM PAYOFF, TOOL AND TECHNIQUES TRAINING, TOOL BUDGET, TOOLSMITHS.

1187 08/29/92 © B²

THE FOUR PHASES OF TEST AWARENESS

- PHASE 0 THINKING: NO DIFFERENCE BETWEEN TESTING AND DEBUGGING
- PHASE 1 THINKING: TO PROVE THAT THE SOFTWARE WORKS
- PHASE 2 THINKING: TO PROVE THAT THE SOFTWARE DOESN'T WORK
- PHASE 3 THINKING: PROVE NOTHING, BUT REDUCE THE ACTUAL AND PERCEIVED RISK OF USING THE SOFTWARE TO AN ACCEPTABLE LEVEL
- PHASE 4 THINKING: TESTING IS A STATE OF MIND AND PROCESS BY WHICH CODE IS DESIGNED SO AS TO REQUIRE LITTLE OR NO TESTING

1220 09/12/92 ©B²

WE MUST CHANGE OUR OBJECTIVES FROM:

- ABSOLUTE PROOFS TO PROBABILITIES
- DEDUCTIONS TO SEDUCTIONS
- FORMALISM TO A WARM TUMMY FEELING
- PROVING THAT THERE ARE NO BUGS TO DEMONSTRATING THAT THERE ARE

1160 09/19/92 ©B²

A RISK-DRIVEN PROCESS

THE PURPOSE OF TESTING IS TO GIVE MANAGEMENT THE INFORMATION IT NEEDS TO RATIONALLY EVALUATE RELEASE RISK

- THE NIGHTMARE LIST (AN ORDERED LIST OF CONCERNS)
- QUANTIFY (COST PER NIGHTMARE)
- DESIGN TESTS TO REDUCE THE NIGHTMARE'S PERCEIVED PROBABILITY
- STOP TESTING AND SHIP THE SOFTWARE WHEN YOU CAN LIVE WITH YOUR NIGHTMARES

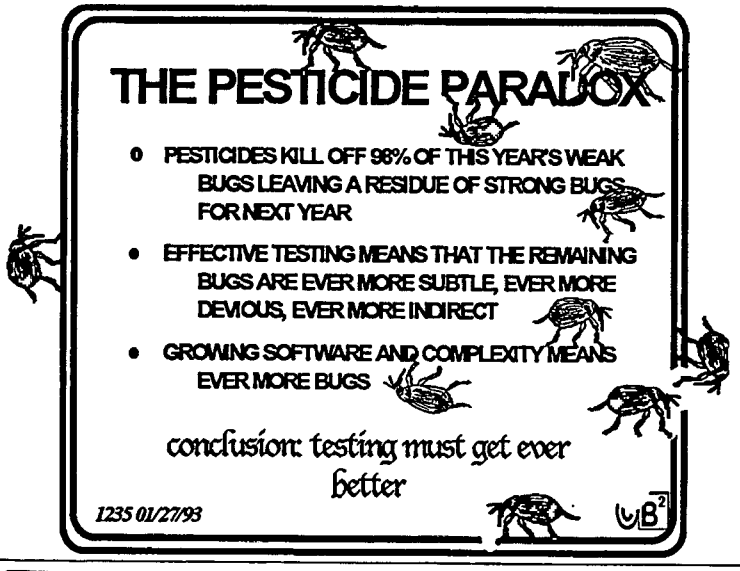
1161 08/19/92 ©B²

IMPLICATIONS FOR CLEAN AND DIRTY TESTS

- CLEAN:DIRTY RATIO CHANGES FROM 5:1 TO 1:5
  THE NEW PROBLEM IS HOW TO CULL TESTS INSTEAD OF HOW TO DESIGN TESTS
- TEST TECHNIQUES PROVIDE GUIDANCE TO REVEALING TESTS
- QA FEEDBACK ON TEST EFFECTIVENESS
- RISK-DRIVEN METHODOLOGY RATHER THAN INTUITION

1230 09/12/92 ©B²

THE PESTICIDE PARADOX



- PESTICIDES KILL OFF 98% OF THIS YEAR'S WEAK BUGS LEAVING A RESIDUE OF STRONG BUGS FOR NEXT YEAR
- EFFECTIVE TESTING MEANS THAT THE REMAINING BUGS ARE EVER MORE SUBTLE, EVER MORE DEVIOUS, EVER MORE INDIRECT
- GROWING SOFTWARE AND COMPLEXITY MEANS EVER MORE BUGS

conclusion: testing must get ever better

1235 01/27/93 ©B²

LIMITATIONS OF TESTING

- STATISTICAL QC OF WIDGETS vs. SOFTWARE QC/QA ISSUES.
- TESTING BLINDNESS
  - EVERY TECHNIQUE IS BLIND TO SOME KINDS OF BUGS.
  - NO TEST TECHNIQUE CAN FIND REQUIREMENTS BUGS.
  - EVERY TECHNIQUE ASSUMES SOMETHING ABOUT BUGS.
- CLEAN AND DIRTY TESTS

1170 08/19/92 ©B²

THE TESTING TRIAD AND THEN SOME

- THREE MAJOR TESTING PHASES
  - UNIT/COMPONENT TESTING (DEVELOPERS)
  - INTEGRATION TESTING (DEVELOPERS, INDEPENDENT
  - SYSTEM TESTING (INDEPENDENT)
- ADDITIONAL TESTING
  - ALPHA TESTING (SIMULATED USER IN-HOUSE)
  - BETA TESTING (FRIENDLY, EXTERNAL, REAL USER)
  - FIELD TESTING (REAL USERS, FRIENDLY TO HOSTILE)
  - USABILITY TESTING (INTERNAL, EXTERNAL)

1201 09/16/92 ©B²

OVERVIEW OF TEST TECHNIQUES

- UNIT/COMPONENT TESTING -- THE FOUNDATION
- PATH TESTING, SENSITIZING, COVERAGE
- TEST TECHNIQUES
  - SYNTAX TESTING
  - TRANSACTION FLOW TESTING
  - STATE TESTING
  - DOMAIN TESTING
  - DATA-FLOW TESTING

WHICH IS BEST??

0130.5 8/19/92 ©B²

DEFINITIONS

- UNIT TESTING: aimed at exposing bugs in the smallest component, the unit.
- COMPONENT TESTING: aimed at exposing bugs in integrated components of one or more units.
- INTEGRATION TESTING: aimed at exposing interface and interaction bugs between otherwise correct and component-tested components.
- FEATURE TESTING: aimed at exposing functional bugs in the features of an integration-tested system.

1205 8/21/92 ©B²

DON'T CONFUSE

- DON'T CONFUSE "INTEGRATION TESTING" WITH TESTING AN ALREADY INTEGRATED SYSTEM.
- THE LATTER IS CALLED "SYSTEM TESTING"
- INTEGRATION TESTING IS THE SPECIAL TESTING DONE TO SUPPORT THE PROCESS BY WHICH UN-INTEGRATED COMPONENTS BECOME INTEGRATED.

1208 09/16/92 ©B²

MORE DEFINITIONS

- SYSTEM TESTING: tests aimed at exposing bugs and conditions usually not covered by specifications, such as security, robustness, recovery, resource loss.
- STRUCTURAL TESTING: test strategies based on a program's structure--e.g., the code. Also called "white box" and "glass box" testing.
- BEHAVIORAL TESTING: test strategies based on a program's required behavior--e.g., specifications. Also called "functional testing" and "black-box testing."
- TESTING: the act of specifying, designing, testing, and executing tests.

1210 08/21/92 ©B²

CLEAN VERSUS DIRTY TESTS

- CLEAN TESTS: TESTS AIMED AT SHOWING THAT THE COMPONENT SATISFIES REQUIREMENTS. ALSO CALLED "POSITIVE TESTS".
- DIRTY TESTS: TESTS AIMED AT BREAKING THE SOFTWARE. ALSO CALLED "NEGATIVE TESTS".
- IMMATURE PROCESS: CLEAN:DIRTY = 5:1
- MATURE PROCESS: CLEAN:DIRTY = 1:5

OBTAINED BY INCREASING THE NUMBER OF DIRTY TESTS

1211 08/21/92 ©B²

TESTING MODALITIES
EQUIVALENCY TESTING

- **WHAT:** BEHAVIORAL REGRESSION TESTING OF ALL OLD FEATURES ON NEW VERSION.
- **WHY:** TO ESTABLISH A CORRECTABLE BASELINE.
- **PREREQUISITES:** UNIT, COMPONENT, INTEGRATION TESTING. SOFTWARE RUNS TO CLEAN OUTPUT FOR EASY CASES.
- **PREREQUISITES:** PERMANENT TEST PRODUCTS. TEST EXECUTION AUTOMATION.

1240 09/16/92 ©B²

TESTING MODALITIES
PROGRESSIVE TESTING

- **WHAT:** BEHAVIORAL TESTING OF NEW FEATURES ON NEW VERSION
- **WHY:** TO EXPOSE NEW FEATURE BUGS; INTER-ACTION BUGS OF NEW FEATURES WITH OLD FEATURES.
- **PREREQUISITES:** EQUIVALENCY TESTING.

1245 09/16/92 ©B²

TESTING MODALITIES
REGRESSION TESTING

- **WHAT:** RERUN OF TEST SUITE (BEST IF COMPLETE RERUN) AFTER ANY CHANGE OF: REQUIREMENTS, SOFTWARE, TEST, TARGET PLATFORM, OR CONFIGURATION.
- **WHY:** TO ESTABLISH A CORRECTABLE BASELINE, TO AVOID A RUNAWAY PROCESS.
- **WHEN:** WHENEVER THE THOUGHT OCCURS TO YOU.
- **PREREQUISITES:** PERMANENT TEST PRODUCTS, TEST EXECUTION AUTOMATION.

1250 09/16/92 ©B²

A UNIVERSAL EXPERIENCE

- THE (ALMOST) UNIVERSAL EXPERIENCE WITH REGRESSION TESTING IS THAT MANUAL REGRESSION TESTING NEVER GETS DONE.
  - EQUIVALENCY TESTING IS COMPROMISED
  - PROGRESSIVE TESTING IS COMPROMISED
  - BOTH ARE COMPROMISED
  - MANUAL RETEST ERROR RATES ARE HIGH
  - EVERYBODY HATES THE JOB

1251 09/16/92 ©B²

TESTING SIMULTANEITY SINGLE THREAD

- **WHAT:** ONE TESTER ON THE PRODUCT AT A TIME
- **SEVERITY:** THE GENTLEST OF ALL TEST MODES
- **BUGS CAUGHT:** SIMPLE, LOCALIZED, OBVIOUS
- **PREREQUISITES:** INTEGRATED, TESTABLE PRODUCT

1255 09/16/92 ©B²

TESTING SIMULTANEITY MULTI-THREAD

- **WHAT:** THREE OR MORE TESTER ON THE PRODUCT SIMULTANEOUSLY
- **SEVERITY:** TOUGH—BEGINS TO CATCH SUBTLE BUGS
- **BUGS CAUGHT:** REENTRANCE, INTERRUPTS, RESOURCE
- **PREREQUISITES:** DUAL-THREAD TESTING

1260 09/16/92 ©B²

STRESS TESTING DEFINED

- SUBJECTING SOFTWARE TO AN *UNREASONABLE* OVERLOAD WHILE DENYING IT THE RESOURCES NEEDED TO PROCESS THAT LOAD.
- MAXIMUM THROUGHPUT IS NOT STRESS--IT'S WHAT THE SYSTEM WAS DESIGNED TO DO.
- IF THE LOAD IS REASONABLE, IT'S NOT STRESS.
- IF THE RESOURCES ARE ADEQUATE, IT'S NOT STRESS.

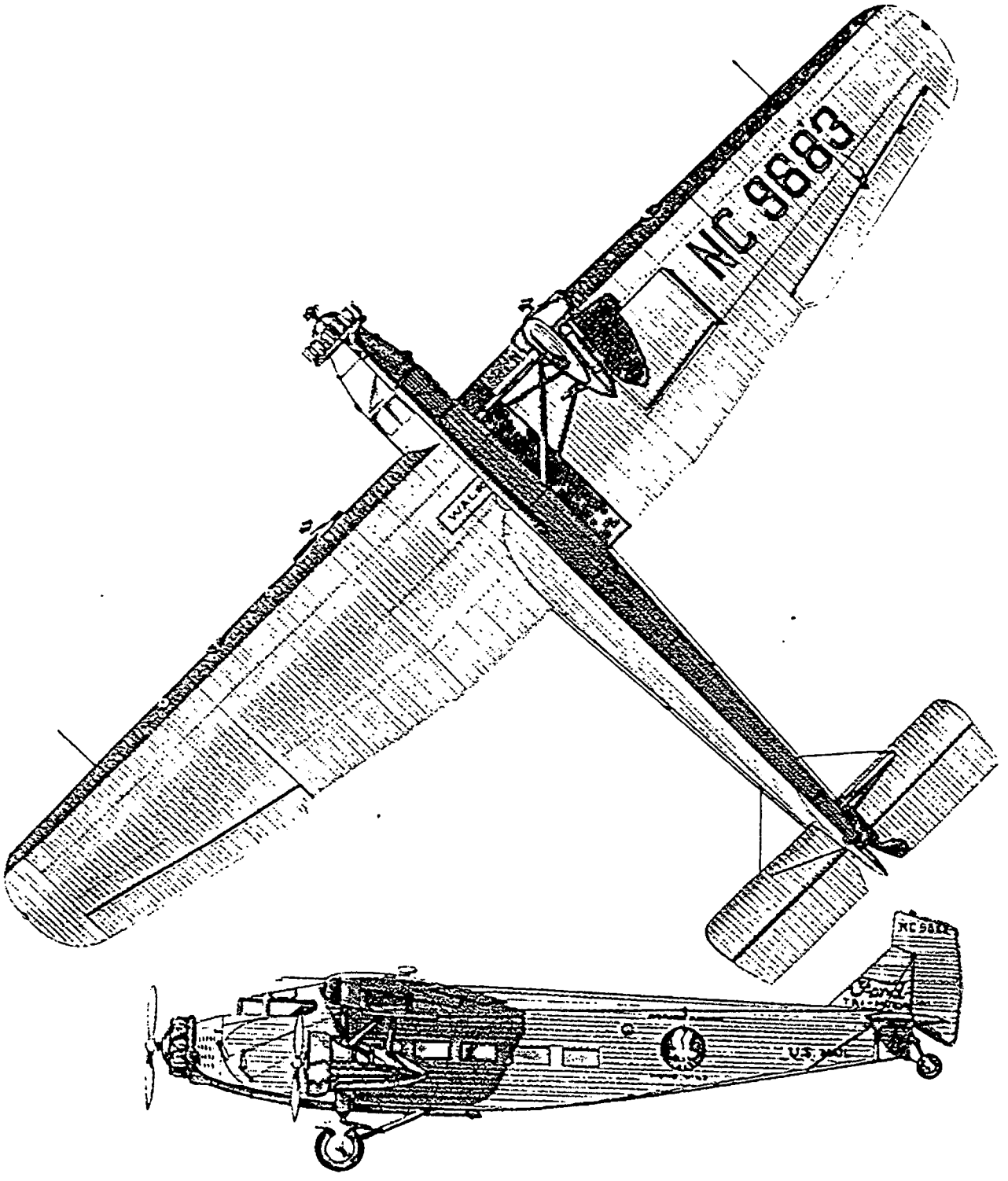
1271 09/16/92 ©B²

TESTING SIMULTANEITY STRESS TESTING

- **WHAT:** BACKGROUND OF STRESS TRANSACTIONS
- **SEVERITY:** MUCH WORSE THAN EXPECTED REALITY
- **BUGS CAUGHT:** REENTRANCE, INTERRUPTS, RESOURCE LOSS BUGS, COMPLEX INTERACTIONS
- **PREREQUISITES:** MULTI-THREAD WITH BACKGROUND

1270 09/16/92 ©B²

FORD TRI-MOTOR (U.S.A.)



34

Ford Model 5-AT-B of American Airways, 1931. *Engines:* Three 420 h.p. Pratt & Whitney Wasp C-1 or SC-1 nine-cylinder radials. *Span:* 77 ft. 10 in. (23.72 m.). *Length:* 49 ft. 10 in. (15.19 m.). *Wing area:* 835.0 sq.ft. (77.57 sq.m.). *Typical take-off weight;* 13,000 lb. (5,897 kg.). *Typical cruising speed:* 122 m.p.h. (196 km/hr.) *Service ceiling:* 18,500 ft. (5,639 m.). *Typical range:* 550 miles (885 km.).

BEHAVIORAL Vs. STRUCTURAL TESTING

THIS IS A COPYRIGHT NOTICE. NO PART OF THESE GRAPHICS MAY BE REPRODUCED OR USED IN ANY FORM BY ANY MEANS SUCH AS GRAPHICS, ELECTRONIC, OPTICAL, MECHANICAL, INCLUDING BUT NOT LIMITED TO PHOTOCOPYING, RECORDING, SCANNING, CONVERSION TO OTHER MEDIA, TRANSMISSION, OR STORAGE IN A COMPUTER OR AN INFORMATION STORAGE AND RETRIEVAL SYSTEM — WITHOUT THE AUTHOR'S WRITTEN PERMISSION.

Boris Beizer, ANALYSIS – 1232 Glenbrook Road, Huntingdon Valley, PA 19006
215-572-5580

17cx 09/01/92

©B²

BEHAVIORAL VS. STRUCTURAL TESTING

- **STRUCTURAL TESTING:** CONFIRM THAT THE ACTUAL STRUCTURE (E.G., CODE) MATCHES THE INTENDED STRUCTURE.
- **BEHAVIORAL TESTING:** CONFIRM THAT THE PROGRAM'S BEHAVIOR MATCHES THE INTENDED BEHAVIOR (E.G., REQUIREMENTS).
  INPUT → RESPONSE

1700 09/02/92

©B²

STRUCTURAL TESTING ADVANTAGES

- EFFICIENT
- THEORETICALLY COMPLETE
- CAN BE MECHANIZED (THEORETICALLY)
- INHERENTLY METHODICAL

1706 09/02/92

©B²

BEHAVIORAL TESTING ADVANTAGES

- INHERENTLY UNBIASED
- ALWAYS USEFUL AND MEANINGFUL
- CATCHES THE BUGS THE USERS SEE
- LESS ANALYSIS REQUIRED

1711 09/02/92 ©B²

STRUCTURAL TESTING DISADVANTAGES

- INHERENTLY BIASED BY DESIGN
- MAY NOT BE MEANINGFUL OR USEFUL
- CAN'T CATCH MANY IMPORTANT BUGS
- FAR REMOVED FROM USER

1707 09/02/92 ©B²

BEHAVIORAL TESTING DISADVANTAGES

- INEFFICIENT--TOO MANY BLANK SHOTS
- THEORETICALLY INCOMPLETE
- CANNOT BE FULLY AUTOMATED
- INTUITIVE RATHER THAN FORMAL

1712 09/02/92 ©B²

STRUCTURAL TESTING EFFECTIVENESS

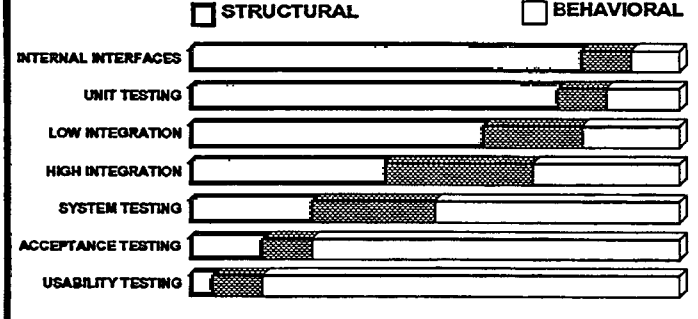
- CATCHES 50%-75% OF BUGS THAT CAN BE CAUGHT IN UNIT TESTING (25%-50% OF TOTAL)
- BUT THEY'RE THE EASIEST ONES TO CATCH
- AT MOST 50% OF TEST LABOR CONTENT

1708 09/02/92 ©B²

BEHAVIORAL TESTING EFFECTIVENESS

- CATCHES 10-30% OF BUGS THAT CAN BE CAUGHT IN UNIT TESTING (5%-15% OF TOTAL)
- CATCHES 50%-75% OF BUGS THAT CAN BE CAUGHT IN SYSTEM TESTING
- CATCHES TOUGH, EMBARRASSING BUGS
- ABOUT 50% OF TEST LABOR CONTENT

1713 09/02/92 ©B²



THERE IS NO CONFLICT!

1715 09/02/92 ©B²

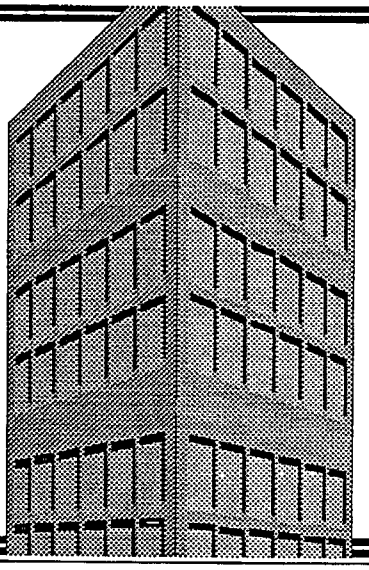
SOFTWARE TESTING AND QUALITY ASSURANCE SEMINARS---BORIS BEIZER

UNIT TESTING--THE FOUNDATION

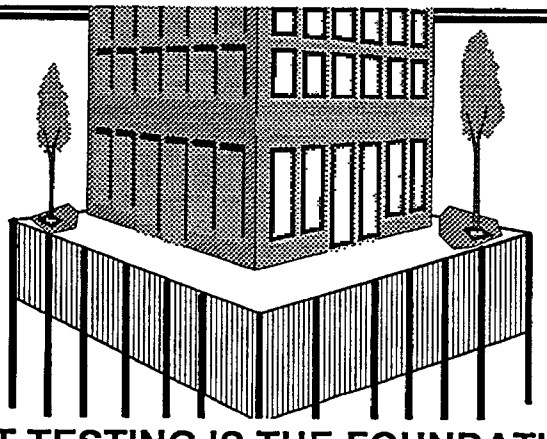
THIS IS A COPYRIGHT NOTICE. NO PART OF THESE GRAPHICS MAY BE REPRODUCED OR USED IN ANY FORM BY ANY MEANS SUCH AS GRAPHICS, ELECTRONIC, OPTICAL, MECHANICAL, INCLUDING BUT NOT LIMITED TO PHOTOCOPYING, RECORDING, SCANNING, CONVERSION TO OTHER MEDIA, TRANSMISSION, OR STORAGE IN A COMPUTER OR AN INFORMATION STORAGE AND RETRIEVAL SYSTEM --- WITHOUT THE AUTHOR'S WRITTEN PERMISSION.

BORIS BEIZER, ANALYSIS – 1232 GLENBROOK ROAD, HUNTINGDON VALLEY, PA 19006
215-572-5580

13xx 09/18/92 ©B²



1300a 09/18/92 ©B²



UNIT TESTING IS THE FOUNDATION

1300b 9/18/92 ©B²

UNIT TESTING IS A MODEL FOR ALL TESTING

- WHAT'S GOOD FOR SUBROUTINES IS GOOD FOR SUBSYSTEMS
- USES THE SAME, ALBEIT MODIFIED, TECHNIQUES
- THE SAME OBJECTIVE AND SUBJECTIVE GOALS

CONFESSION: I DON'T KNOW HOW TO DO SYSTEM TESTING
I ONLY KNOW HOW TO APPLY UNIT TESTING TECHNIQUES TO SYSTEM TESTING PROBLEMS

1310 09/18/92

©B²

PREREQUISITES TO UNIT TESTING

- BUILDER'S CONFIDENCE
- A TESTABLE COMPONENT
- INSPECTIONS
- THOROUGH PRIVATE TESTING
- A DESIGNED, DOCUMENTED, UNIT TEST PLAN
- TIME, PREREQUISITES, TOOLS, RESOURCES.

1315 09/18/92

©B²

MENTAL MODELS

- EVERY TEST TECHNIQUE IS BASED ON A DIFFERENT MENTAL MODEL OF WHAT'S IMPORTANT ABOUT A ROUTINE
- EACH SUCH MODEL ASSUMES SOMETHING ABOUT THE LIKELIHOOD OF CERTAIN BUG TYPES, FOR EXAMPLE:
  - CONTROL BUGS DOMINATE]
  - LOOPS ARE BUGGY
  - PROGRAMMERS CAN'T HANDLE INTERFACES
  - INITIALIZATION BUGS DOMINATE
- THERE ARE AS MANY POTENTIAL TEST TECHNIQUES AS THERE ARE WAYS OF LOOKING AT SOFTWARE'S BEHAVIOR OR STRUCTURE
- EACH POINT OF VIEW REPRESENTS A TEST TECHNIQUE

1335 09/18/92

©B²

COVERAGE CONCEPTS

- "Coverage" is a measure of testing completeness with respect to a particular testing strategy.
- "100% Coverage" *never never* means "complete testing", but only completeness with respect to a specific strategy.
- It follows that every strategy and therefore every associated test technique will have an associated coverage concept.
- An infinite number of strategies.
  An infinite number of associated techniques.
  An infinite number of coverage metrics.
- None is best, but some are better than others.

1340 11/03/92 ©B²

INTEGRATION AND INTERFACES

This is a copyright notice. No part of these graphics may be reproduced or used in any form by any means such as graphics, electronic, optical, mechanical, including but not limited to photocopying, recording, scanning, conversion to other media, transmission, or storage in a computer or an information storage and retrieval system -- without the author's written permission.

Boris Beizer, ANALYSIS -- 1232 Glenbrook Road, Huntingdon Valley, PA 19006
215-572-5580

30XX 09/18/92 ©B²

GOALS OF INTEGRATION TESTING

OBJECTIVE GOALS:

- DEMONSTRATE THAT SOFTWARE COMPONENTS ARE INCONSISTENT WITH ONE ANOTHER
- BUILD A HIERARCHY OF WORKING COMPONENTS

SUBJECTIVE GOALS:

- BUILD A HIERARCHY OF *TRUST*

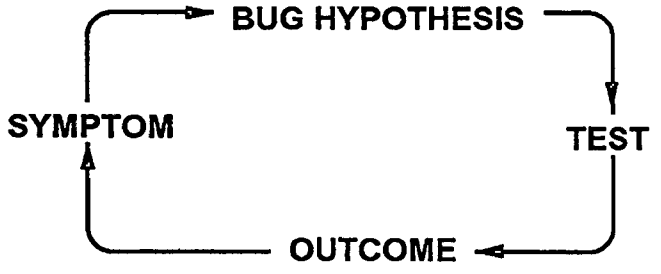
3030 09/18/92 ©B²

WHAT'S THIS JAZZ ABOUT "TRUST"???

- OCCAM'S RAZOR: FIRST TRY THE SIMPLEST HYPOTHESIS THAT FITS ALL THE FACTS.
- WHICH IS THE SIMPLER HYPOTHESIS?
  - THE BUG IS IN ONE OR THE OTHER COMPONENTS
  - THE BUG IS IN NEITHER COMPONENT, IT ARISES FROM THEIR UNPREDICTED INTERACTION.

3035.1 09/18/92 ©B²

THE DEBUGGING PROCESS



3035.2 09/18/92 ©B²

INTEGRATION WITH AND WITHOUT TRUST

- INTRA-COMPONENT BUGS ARE SIMPLER THAN INTER-COMPONENT BUGS
- WITH TRUST, DEBUGGER REJECTS THE SIMPLER UNIT BUG HYPOTHESIS AND GOES DIRECTLY FOR THE MORE COMPLEX INTEGRATION-INTERFACE BUG
- WITHOUT TRUST, THE DEBUGGER WASTES HOURS OR DAYS LOOKING FOR COMPONENT BUGS THAT DON'T EXIST

RIGHT DOWN TO THE SUBROUTINE LIBRARY, THE COMPILER, OR EVEN THE HARDWARE

3035.3 09/18/92 ©B²

INTEGRATION IS NOT AN EVENT
IT IS A PROCESS

A PROCESS THAT BEGINS WHEN THERE ARE TWO OR MORE TESTED COMPONENTS

AND ENDS WHEN THERE IS AN ADEQUATELY TESTED SYSTEM

3040 09/18/92 ©B²

PREREQUISITES TO INTEGRATION

- TRUSTED SUBCOMPONENTS
- INTERFACE STANDARDS
- CONFIGURATION CONTROL
- DATA DICTIONARY
- AN INTEGRATION PLAN
- TIME, TOOLS, RESOURCES

3050 09/18/92 ©B²

SOFTWARE TESTING AND QUALITY ASSURANCE SEMINARS—BORIS BEIZER

SYSTEM TESTING

THIS IS A COPYRIGHT NOTICE. NO PART OF THESE GRAPHICS MAY BE REPRODUCED OR USED IN ANY FORM BY ANY MEANS SUCH AS GRAPHICS, ELECTRONIC, OPTICAL, MECHANICAL, INCLUDING BUT NOT LIMITED TO PHOTOCOPYING, RECORDING, SCANNING, CONVERSION TO OTHER MEDIA, TRANSMISSION, OR STORAGE IN A COMPUTER OR AN INFORMATION STORAGE AND RETRIEVAL SYSTEM -- WITHOUT THE AUTHOR'S WRITTEN PERMISSION.

BORIS BEIZER, ANALYSIS -- 1232 GLENBROOK ROAD, HUNTINGDON VALLEY, PA 19006
215-572-5580

15XX 09/18/92 ©B²

SYSTEM TESTING GOALS

- BOTTOM-LINE BEHAVIORAL VERIFICATION
- FOCUS ON USERS' PERSPECTIVE AND CONCERNS
- RISK MINIMIZATION
- GLOBAL, INTERACTIVE, TIMING, BUGS

1500 09/18/92 ©B²

PREREQUISITES TO SYSTEM TESTING

- A SUFFICIENTLY FLESHED BACKBONE TO BEGIN TESTING
- A DETAILED, WRITTEN, FUNCTIONAL SPECIFICATION
- A DESIGNED, DOCUMENTED, SYSTEM TEST PLAN
- A CONTROLLED TEST DATA BASE
- A FAILURE REPORTING MECHANISM
- TIME, TOOLS, RESOURCES

1505 09/18/92 ©B²

BOTTOM-LINE BEHAVIORAL TESTING

- USUALLY INDEPENDENT TESTING
- DOMINATED BY BEHAVIORAL TESTING
- STRUCTURE KNOWLEDGE HELPS CUT USELESS TESTS
- THE USERS' POINT OF VIEW
- GETS RISK CONTROL DATA

1515 09/18/92 ©B²

SYSTEM TESTS

- FEATURE INTERACTION TESTING: SHOW THAT INTERACTING FEATURES DON'T HAVE UNWANTED BYPRODUCTS OR SIDE EFFECTS.
- CONFIGURATION TESTING: SHOW THAT THE SOFTWARE DOES NOT HAVE UNPLANNED INTERACTIONS WITH OTHER PACKAGES
- BOUNDARY INTEGRITY TESTING: SHOW THAT THE SOFTWARE RESPECTS THE MEMORY BOUNDARIES SET FOR IT BY OS.

1529 09/18/92

©B²

OTHER SYSTEM TESTS

- HARDWARE CONFIGURATION SENSITIVITY: SHOW THAT THE SOFTWARE WILL WORK FOR EVERY TARGET HARDWARE PLATFORM
- PERFORMANCE TESTING: SHOW THAT SYSTEM HARDWARE/SOFTWARE RESOURCES CAN MEET TARGET THROUGHPUTS AND DELAYS
- RECOVERY TESTING: SHOW THAT THE SYSTEM DOES NOT LOSE OR CORRUPT DATA OR ACCOUNTABILITY FOR DATA AFTER A FAILURE
- SECURITY TESTING: SHOW THAT THE SYSTEM CAN PROTECT ITSELF AGAINST POSTULATED ATTACKS

1530 09/18/92

©B²

CONTROL FLOW (PATH) TESTING

THIS IS A COPYRIGHT NOTICE. NO PART OF THESE GRAPHICS MAY BE REPRODUCED OR USED IN ANY FORM BY ANY MEANS SUCH AS GRAPHICS, ELECTRONIC, OPTICAL, MECHANICAL, INCLUDING BUT NOT LIMITED TO PHOTOCOPYING, RECORDING, SCANNING, CONVERSION TO OTHER MEDIA, TRANSMISSION, OR STORAGE IN A COMPUTER OR AN INFORMATION STORAGE AND RETRIEVAL SYSTEM -- WITHOUT THE AUTHOR'S WRITTEN PERMISSION.

BORIS BEIZER, ANALYSIS -- 1232 GLENBROOK ROAD, HUNTINGTON VALLEY, PA. 19006
215-572-5580

20xx 09/19/92

©B²

WHY ALL TESTERS SHOULD KNOW CONTROL FLOW (PATH) TESTING

- **FUNDAMENTAL TECHNIQUE** THAT ILLUSTRATES ASPECTS OF OTHER TEST TECHNIQUES.
- **PATHS EXIST** AND THEY'RE IMPORTANT EVEN IF YOU DON'T DO PATH TESTING.
- **DEVELOPERS' TESTING:** DESIGNERS OFTEN USE PATH TESTING METHODS IN UNIT TESTING. YOU MUST UNDERSTAND THEIR TESTS.
- **DOMAIN TESTING** IF USED AS A BEHAVIORAL TEST METHOD REQUIRES AN UNDERSTANDING OF THE UNDERLYING PROGRAM PATHS.
- **TRANSACTION FLOW TESTING:** A BEHAVIORAL TEST METHOD USED IN SYSTEM TESTING, IT IS ALMOST IDENTICAL TO PATH TESTING.
- **DATA-FLOW TESTING:** IN EITHER BEHAVIORAL OR STRUCTURAL FORM, PRESUPPOSES KNOWLEDGE OF PATH TESTING METHODS.

1345 09/19/92

©B²

UNIT TESTING FOR SYSTEM TESTERS?

WHY SHOULD A SYSTEM TESTER WHO WILL PROBABLY NEVER TEST INDIVIDUAL UNITS BE CONCERNED WITH UNIT TESTING ?

- HOW DO YOU PROPOSE TO TEST YOUR TESTING MACROS? MOST MACRO LANGUAGES ARE ACTUALLY FULL-BLOWN PROGRAMMING LANGUAGES.
- THE BASIC CONCEPTS OF TESTING--FLOWGRAPHS, BRANCH COVER, DATA-FLOW, ARE MOST EASILY EXPLAINED IN TERMS OF PATH (CONTROL-FLOW) TESTING. A BASIC UNIT TEST METHOD
- CAN YOU DO SQA AND BE AN EFFECTIVE CRITIC OF SOFTWARE DEVELOPERS IF YOU DON'T KNOW THEIR TEST METHODS?

1345.1 09/19/92

©B²

CONTROL FLOW TESTING CONSISTS OF SELECTING AND EXECUTING ENOUGH PATHS THROUGH A PROGRAM TO CONVINCE US THAT THE PROGRAM'S ACTUAL STRUCTURE MATCHES THE PROGRAMMER'S INTENDED STRUCTURE

CONTROL FLOW (PATH) TESTING CONSISTS OF SELECTING AND EXECUTING ENOUGH PATHS THROUGH A PROGRAM TO CONVINCE US THAT THE PROGRAM'S ACTUAL STRUCTURE MATCHES THE PROGRAMMER'S INTENDED STRUCTURE

2010 09/19/92 ©B²

CONTROL-FLOW (PATH) TESTING STRATEGIES

- **STATEMENT COVERAGE:** ENOUGH TEST CASES TO ASSURE THAT EVERY PROGRAM STATEMENT HAS BEEN TESTED AT LEAST ONCE. IEEE/ANSI STANDARD.
- **BRANCH COVERAGE:** ENOUGH TEST CASES TO ASSURE THAT EVERY PROGRAM BRANCH ALTERNATIVE HAS BEEN TESTED AT LEAST ONCE. (MINIMUM RATIONAL STANDARD)
- **PATH COVERAGE:** ENOUGH TEST CASES TO ASSURE THAT EVERY POSSIBLE PATH THROUGH THE PROGRAM HAS BEEN TESTED AT LEAST ONCE. IMPRACTICAL, INEFFICIENT, UNNECESSARY.

2011 10/26/92 ©B²

THE THREE PARTS OF PATH TEST DESIGN

- **SELECT THE COVERING PATHS IN ACCORDANCE TO THE CHOSEN STRATEGY**
- **SENSITIZE THE PATHS:** FIND INPUT VALUES THAT FORCE THE SELECTED PATHS.
- **INSTRUMENT THE PATHS:** CONFIRM THAT YOU ACTUALLY WENT ALONG THE CHOSEN PATH.

2015 09/21/92 ©B²

SELECT COVERING PATHS

- SELECT YOUR PATHS TO SATISFY THE CHOSEN STRATEGY
- USE FUNCTIONALLY SENSIBLE PATHS IF POSSIBLE
- USE COVERAGE TOOL TO MONITOR TEST COMPLETENESS

2015.1 09/21/92 ©B²

PATH SELECTION CRITERIA

- MANY SHORT PATHS BETTER THAN A FEW LONG, COMPLICATED PATHS
- FUNCTIONALLY MEANINGFUL PATHS BETTER THAN OBSCURE PATHS
  - ABILITY TO PREDICT OUTCOME
  - ABILITY TO SENSITIZE
- COVER MORE IMPORTANT THAN WHICH PATHS YOU PICK

2081 09/21/92 ©B²

SENSITIZE THE PATHS

- FIND A SET OF INPUT VALUES FOR EACH SELECTED TEST PATHS THAT WILL MAKE THE SOFTWARE GO DOWN THAT PATH
- THEORETICALLY DIFFICULT, BUT IN PRACTICE EASIER DONE THAN SAID
- SENSITIZATION DIFFICULTIES USUALLY INDICATE CONCEPTUAL OR DESIGN PROBLEMS

2015.2 09/21/92 ©B²

INSTRUMENT THE PATH

- BECAUSE OF BUGS, YOU MAY NOT TAKE THE PATH YOU INTENDED TO TAKE
- INSTRUMENTATION IS USED TO CONFIRM THE ACTUAL PATH TAKEN
- TODAY, USUALLY PART OF THE UNIT TEST TOOL

2015.3 09/21/92 ©B²

REVIEW OF PATH TESTING

- PICK A STRATEGY: STATEMENT COVER, BRANCH COVER, PATH COVER
- SELECT THE PATHS TO SATISFY THE STRATEGY
- SENSITIZE THE PATHS
- PREDICT THE OUTCOMES
- RUN THE TESTS
- CONFIRM THE OUTCOMES AND PATHS

2260 09/21/92 ©B²

PATH TESTING TOOLS

TYPICAL TOOL TODAY IS A MULTI-PURPOSE TOOL THAT INCLUDES

- COVERAGE RECORDING
- PATH CONFIRMATION
- TEST SET-UP
- TEST EXECUTION
- SMART OUTCOME COMPARISON

2261 09/21/92 ©B²

SOFTWARE TESTING AND QUALITY ASSURANCE SEMINARS—BORIS BEIZER

TRANSACTION FLOW TESTING

THIS IS A COPYRIGHT NOTICE. NO PART OF THESE GRAPHICS MAY BE REPRODUCED OR USED IN ANY FORM BY ANY MEANS SUCH AS GRAPHICS, ELECTRONIC, OPTICAL, MECHANICAL, INCLUDING BUT NOT LIMITED TO PHOTOCOPYING, RECORDING, SCANNING, CONVERSION TO OTHER MEDIA, TRANSMISSION, OR STORAGE IN A COMPUTER OR AN INFORMATION STORAGE AND RETRIEVAL SYSTEM — WITHOUT THE AUTHOR'S WRITTEN PERMISSION.

BORIS BEIZER, ANALYSIS — 1232 GLEFFERDOM ROAD, HUNTINGDON VALLEY, PA 19006
215-572-1580

28XX 09/21/92 ©B²

TRANSACTION FLOW TESTING

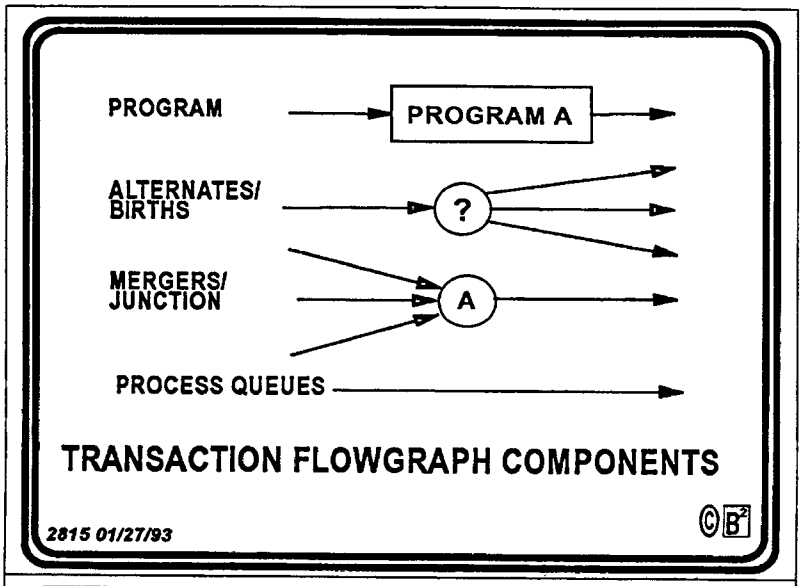
A BEHAVIORAL TEST TECHNIQUE BASED ON A STRUCTURAL MODEL (CONTROL FLOW TESTING)

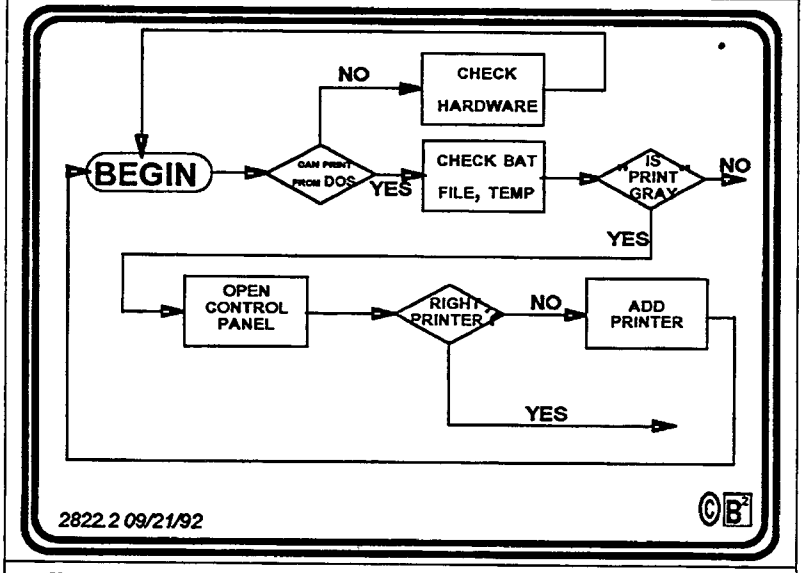
2800 09/21/92 ©B²

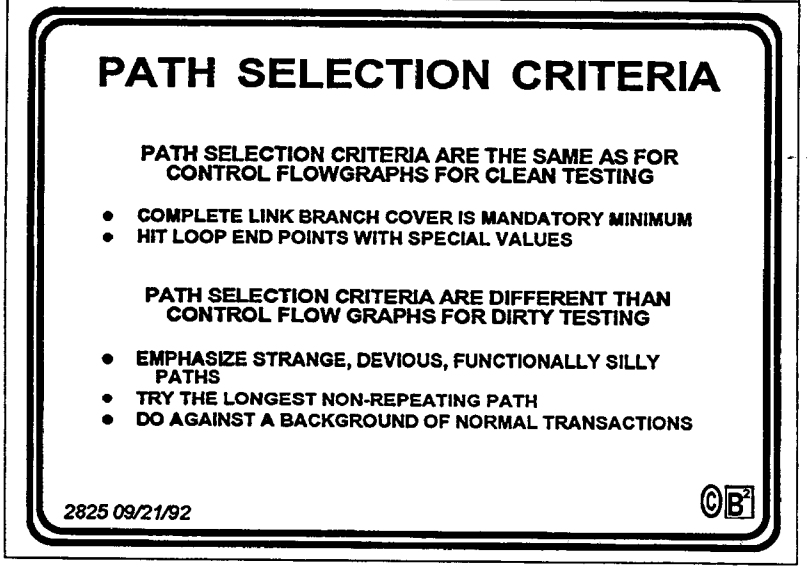
PRINCIPLES

- IDENTIFY THE MAJOR TRANSACTIONS
- DESCRIBE THE SEQUENCE OF OPERATIONS USING FLOWCHARTING (FLOWGRAPH) LANGUAGE
- TREAT THAT FLOWGRAPH AS IF IT IS AN ACTUAL PROGRAM (E.G., CONTROL FLOWGRAPH)
- APPLY STRUCTURAL TEST TECHNIQUES (E.G., PATH TESTING, LOOP TESTING, ETC.)

2805 09/21/92 ©B²


PROGRAM
PROGRAM A
ALTERNATES/
BIRTHS
?
MERGERS/
JUNCTION
A
PROCESS QUEUES
TRANSACTION FLOWGRAPH COMPONENTS
2815 01/27/93
©B²


NO
CHECK
HARDWARE
BEGIN
CAN PRINT FROM DOS
YES
CHECK BAT
FILE, TEMP
IS PRINT GRAY
NO
YES
OPEN
CONTROL
PANEL
RIGHT PRINTER
NO
ADD
PRINTER
YES
2822.2 09/21/92
©B²


PATH SELECTION CRITERIA
PATH SELECTION CRITERIA ARE THE SAME AS FOR CONTROL FLOWGRAPHS FOR CLEAN TESTING
• COMPLETE LINK BRANCH COVER IS MANDATORY MINIMUM
• HIT LOOP END POINTS WITH SPECIAL VALUES
PATH SELECTION CRITERIA ARE DIFFERENT THAN CONTROL FLOW GRAPHS FOR DIRTY TESTING
• EMPHASIZE STRANGE, DEVIOUS, FUNCTIONALLY SILLY PATHS
• TRY THE LONGEST NON-REPEATING PATH
• DO AGAINST A BACKGROUND OF NORMAL TRANSACTIONS
2825 09/21/92
©B²

TRANSACTION FLOW TEST DESIGN STEPS

1. FIND AND DEFINE A COVERING SET OF TRANSACTION FLOWS
2. SELECT THE TEST PATHS
3. SENSITIZE THE PATHS
   - PREPARE INPUTS
   - PREDICT OUTCOMES
4. INSTRUMENT THE PATHS
5. DEBUG AND RUN THE TESTS

2830 09/22/92 ©B²

TEST EXECUTION

- EXECUTION AUTOMATION ESSENTIAL
- SUPPORTING DATA AND STRUCTURES 25% TO 50% OF THE EFFORT
- INDIRECT VERIFICATION AS IMPORTANT AS DIRECT VERIFICATION
  - TRANSACTION CORRUPTION OR LOSS
  - RESOURCE CORRUPTION OR LOSS
  - PROGRAM SPACE OR OTHER MEMORY VIOLATIONS

2830.5 09/22/92 ©B²

AUTOMATION TOOLS

- CAPTURE/PLAYBACK: EASIEST WAY TO MAKE THE TRANSITION TO AUTOMATED TESTING.
- SCRIPTING LANGUAGE: SUCH AS CASL-IV FOR THOSE WHO'D RATHER SCRIPT THAN PLAY. PROVIDES BETTER TEST CONTROL.
- DEDICATED GENERATOR/DRIVER: FOR MASS TRANSACTIONS (E.G., BANK TELLER MACHINES) AND COMPLICATED SCENARIOS.
- PC/WORKSTATION: INDISPENSABLE.

2831 09/22/92 ©B²

DATA FLOW TESTING

THIS IS A COPYRIGHT NOTICE. NO PART OF THESE GRAPHICS MAY BE REPRODUCED OR USED IN ANY FORM BY ANY MEANS SUCH AS GRAPHICS, ELECTRONIC, OPTICAL, MECHANICAL, INCLUDING BUT NOT LIMITED TO PHOTOCOPYING, RECORDING, SCANNING, CONVERSION TO OTHER MEDIA, TRANSMISSION, OR STORAGE IN A COMPUTER OR AN INFORMATION STORAGE AND RETRIEVAL SYSTEM -- WITHOUT THE AUTHOR'S WRITTEN PERMISSION.

BORIS BEIZER, ANALYSIS -- 1232 GLEFFERDOK ROAD, HUNTINGDON VALLEY, PA 19006
215-572-5980

46XX 09/22/92 ©B²

A QUESTION

- COMMON SENSE DICTATES PATH TESTING BECAUSE IT DOESN'T MAKE SENSE TO RELEASE UNTESTED CODE
- DOES IT MAKE SENSE TO RELEASE CODE FOR WHICH DATA DEFINITIONS AND USE ARE UNTESTED?
- AT LEAST, WE SHOULD VERIFY THAT:
  - DATA ARE DEFINED AS WE INTEND
  - THEY ARE CREATED BEFORE NEEDED
  - THEY ARE USED (PROPERLY) IF CREATED

4601 09/22/92 ©B²

DATA FLOW TEST CRITERIA (STRUCTURAL)

- MORE GENERAL THAN PATH TESTING FAMILY
- STRONGER THAN BRANCH BUT WEAKER THAN ALL PATHS
- MUST BE DONE SEPARATELY FOR EACH DATA OBJECT
- BASED ON CONTROL FLOWGRAPH ANNOTATED WITH DATA FLOW RELATIONS

4610 09/22/92 ©B²

DATA FLOW TEST CRITERIA (BEHAVIORAL)

- HEURISTIC BUT SENSIBLE AND EFFECTIVE
- TRANSACTION FLOW TESTING IS A KIND OF DATA FLOW TESTING
- MUST BE DONE SEPARATELY FOR EACH DATA OBJECT IN YOUR DATA MODEL
- BASED ON DATA FLOWGRAPHS USED IN MANY DESIGN METHODOLOGIES

4610.5 09/22/92 ©B²

DATA FLOW DEFINITIONS

COMPUTATIONAL USE: VARIABLE USED IN A COMPUTATION; E.G., APPEARS ON THE RIGHT-HAND-SIDE OF AN ASSIGNMENT STATEMENT.

PREDICATE USE: VARIABLE USED IN A CONTROL FLOW PREDICATE.

USE: A COMPUTATIONAL AND/OR PREDICATE USE.

DEFINE: A VARIABLE IS DEFINED WHEN ANY OF THE FOLLOWING HAPPEN

- CREATED: E.G. IN A DECLARATION
- NEW VALUE ASSIGNED
- INITIALIZED TO A NEW VALUE

4605 09/22/92 ©B²

EXAMPLES OF DATA OBJECTS

- **EXPLICIT VARIABLES:** X, Y, PAYROLL_ENTRY
- **ARRAYS:** EACH ELEMENT IS A SEPARATE OBJECT
- **POINTERS:** TO ARRAYS, RECORDS, FILES
- **FILES:** FILES, RECORDS, DATA STRUCTURES
- **PROGRAMS:** WHEN MANIPULATED AS DATA (E.G., LOAD, INITIALIZE)
- **OBJECTS:** AS IN OBJECT ORIENTED PROGRAMMING

ANY OF THESE CAN BENEFIT FROM DATA FLOW TESTING

4605.1 09/22/92 ©B²

DATA FLOW TESTING IS VERY OLD

- USED AT IBM FOR OVER 25 YEARS
- IMPLICIT IN MANY DESIGN METHODS: HIPO, SOFTECH, BENDER
- A NATURAL ADJUNCT TO DATA FLOW DESIGN
- COMMON SENSE BASIS
  - KNOW WHAT YOU'RE DOING WITH DATA
  - TEST THAT WHAT YOU DID IS WHAT YOU WANTED TO DO
- TRANSACTION FLOW TESTING IS DATA FLOW TESTING

4611 09/22/92

©B²

THEORY AND PRACTICE

- PRACTICE: VERY OLD
  - INTUITIVE, HEURISTIC, COMMON SENSE
  - BEHAVIORAL TEST TECHNIQUE
- THEORY: 10 YEARS OLD
  - FORMALIZATION OF COMMON SENSE
  - STRUCTURAL TEST TECHNIQUE
  - TOOL INTENSIVE
- CONCLUSIONS
  - EITHER BEHAVIORAL OR STRUCTURAL
  - APPLICABLE TO ALL TESTING LEVELS
  - BASIC PART OF TESTERS' TOOL KIT

4612 09/22/92

©B²

DATA FLOW TEST CRITERIA

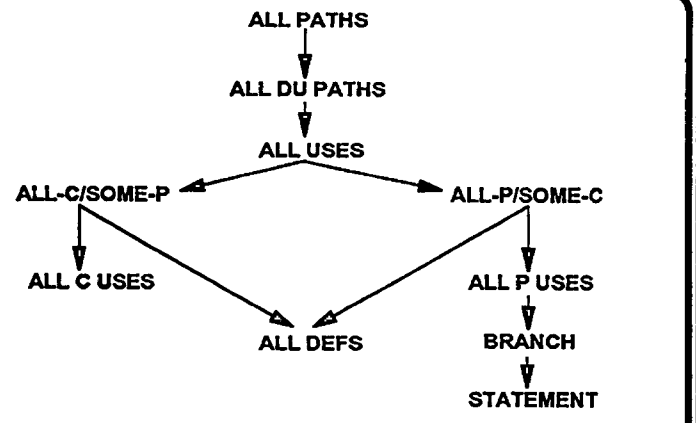
- ALL PREDICATE USES: ALL USES IN CONTROL-FLOW PREDICATES.
- ALL COMPUTATIONAL USES: ALL USES IN CALCULATIONS.
- ALL DEFINITIONS: ALL POINTS AT WHICH OBJECTS ARE DECLARED AND/OR ASSIGNED NEW VALUES.
- ALL USES: ALL POSSIBLE USES OF OBJECTS.
- ALL DU-SEGMENTS (DEFINED THEN USED): ALL SIMPLE PATH SEGMENTS ON WHICH AN OBJECT IS DEFINED AND THEN USED WITHOUT INTERVENING VALUE CHANGES.

NOTE: apply to every data object separately

NOTE: each criteria has its own coverage metric

4614 09/22/92

©B²

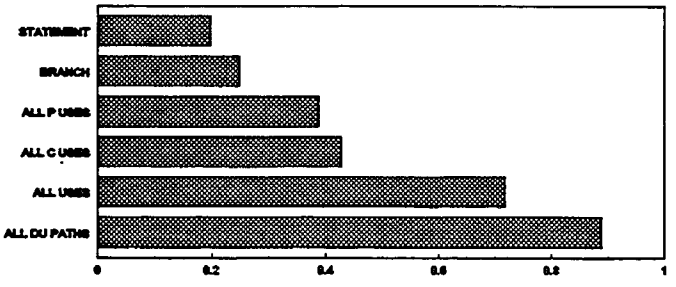


RELATIVE STRENGTH OF STRUCTURAL TEST STRATEGIES

4621 09/22/92

©B²

RELATIVE COST (TEST CASES PER BRANCH) OF VARIOUS STRUCTURAL STRATEGIES



CONCLUSION: FOR NOT MUCH MORE TESTING THAN EITHER STATEMENT OR BRANCH, ALL DU-PATHS IS ALMOST AS GOOD AS TESTING ALL POSSIBLE PATHS.

4630.1 09/22/92

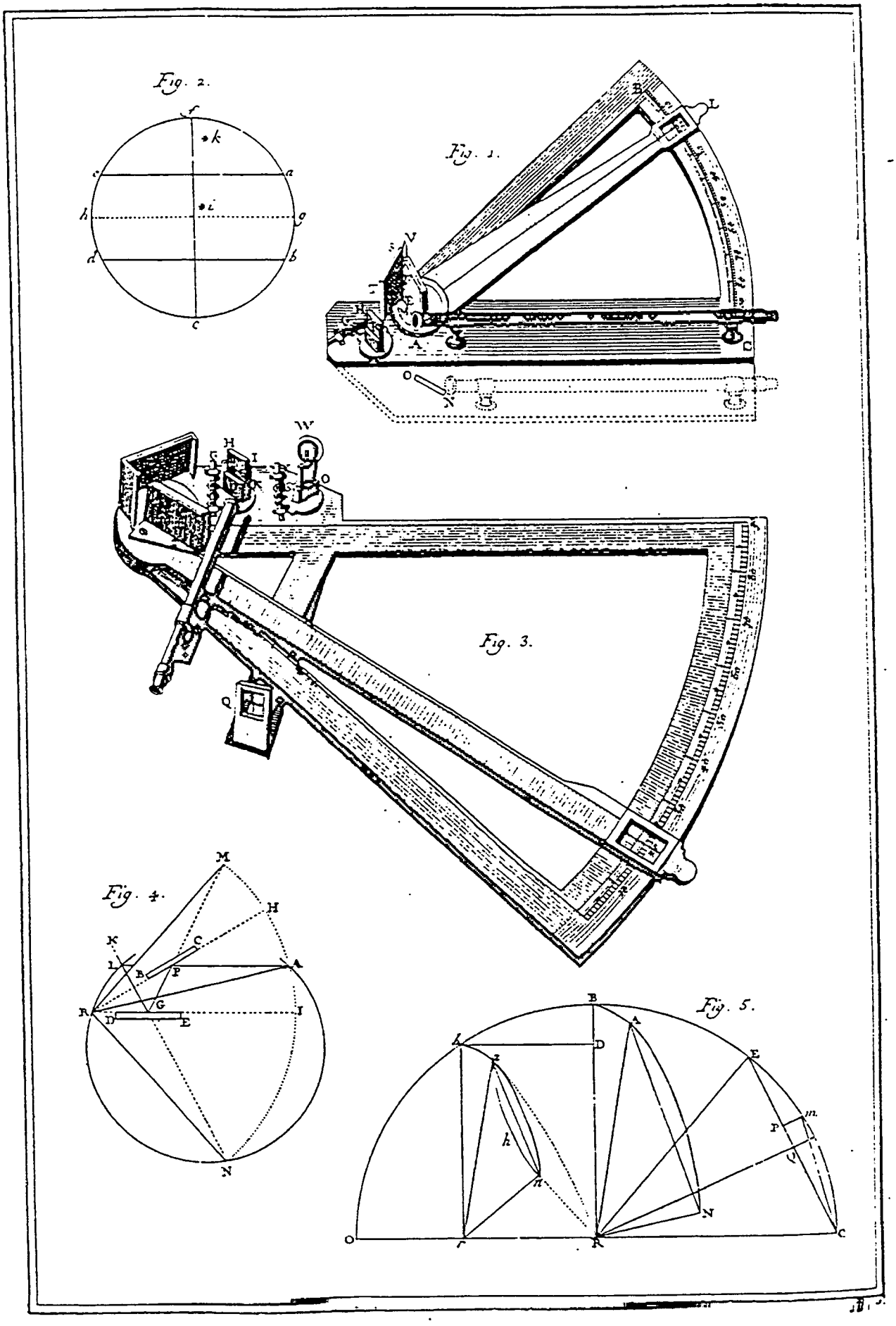
©B²

TOOLS AND AUTOMATION STATUS

- BEHAVIORAL DATA FLOW TESTING DOESN'T NEED SPECIAL TOOLS, ONLY THE WILL TO DO IT.
- STRUCTURAL DATA FLOW TESTING NEEDS TO MUCH BOOKKEEPING TO DO MANUALLY. IT IS TOOL INTENSIVE.
- MANY PRIVATE STRUCTURAL DATA FLOW TOOLS IN USE
- NO COMMERCIAL TOOLS AS OF 9/22/92

4640 09/22/92

©B²



Astronomie, Instrumens, Secteur de M. Eclse

TEST TOOLS

THIS IS A COPYRIGHT NOTICE. NO PART OF THESE GRAPHICS MAY BE REPRODUCED OR USED IN ANY FORM BY ANY MEANS SUCH AS GRAPHICS, ELECTRONIC, OPTICAL, MECHANICAL, INCLUDING BUT NOT LIMITED TO PHOTOCOPYING, RECORDING, SCANNING, CONVERSION TO OTHER MEDIA, TRANSMISSION, OR STORAGE IN A COMPUTER OR AN INFORMATION STORAGE AND RETRIEVAL SYSTEM -- WITHOUT THE AUTHOR'S WRITTEN PERMISSION.

BORIS BEIZER ANALYSIS -- 1232 GLENBROOK ROAD, HUNTINGDON VALLEY, PA 19006
215-572-9580

75XX 09/22/92

©B²

LIMITATIONS OF MANUAL TESTING

- NOT REPRODUCIBLE
- TESTING AND TESTER ERRORS
  - INITIALIZATION BUGS
  - DATA BASE AND CONFIGURATION BUGS
  - INPUT BUGS
  - VERIFICATION AND COMPARISON BUGS
  - INPUT "CORRECTIONS"
- VARIABLE REPORTS, NO SUPPORT FOR METRICS, POOR TRACKING
- VERY LABOR INTENSIVE: TESTERS SHOULD DESIGN TESTS, NOT POUND KEYS.

©B²

7900 09/22/92

WHY AUTOMATED TESTING IS MANDATORY

- MANUAL TEST EXECUTION ERROR RATES ARE MUCH HIGHER THAN THE SOFTWARE RELIABILITIES THE USERS DEMAND
- HOW CAN YOU TEST A 0.000001 PACKAGE WITH A 0.01 PROCESS?
- WHAT MUST BE JUSTIFIED IS CONTINUED USE OF MANUAL METHODS ON AUTOMATION PRODUCTS
- MOST COST-BENEFITS ANALYSES THAT CLAIM TO SHOW THAT MANUAL TESTING IS CHEAPER ASSUME NO TESTING BUGS--SILLY ASSUMPTION!

©B²

7901 09/22/92

TEST TOOLS OVERVIEW

- FUNDAMENTAL TOOLS: COMPILERS, SYMBOLIC DEBUGGER, DEVELOPMENT TOOLS, HARDWARE, HUMAN ENVIRONMENT.
- ANALYTICAL TOOLS: THAT TELL US SOMETHING ABOUT THE SOFTWARE: FLOWGRAPH GENERATORS, CALL TREE GENERATORS.
- TEST EXECUTION AUTOMATION TOOLS
- TEST DESIGN AUTOMATION AIDS
- CAST: COMPUTER AIDED SOFTWARE TESTING

7501 09/22/92 ©B²

A BASIC TOOLKIT

- CAPTURE/PLAYBACK (BEHAVIORAL TOOL)
- UNIT COVERAGE ANALYZER & DRIVER (STRUCTURAL TEST TOOL)
- REQUIREMENTS BASED TOOL (BEHAVIORAL TEST TOOL)

1188 09/10/92 ©B²

COVERAGE ANALYZERS

- EACH TECHNIQUE HAS A DIFFERENT NOTION OF COVERAGE
- EACH TECHNIQUE THEREFORE NEEDS A DIFFERENT COVERAGE ANALYZER
  - PATH TESTING: BRANCH AND STATEMENT COVER
  - DATA-FLOW TESTING: VARIOUS COVER METRICS
  - STATE TESTING: STATE AND TRANSITION COVER

7800 09/22/92 ©B²

PATH COVERAGE TOOL

- TRACE-LIKE OPERATION OR MODIFIES SOURCE CODE
- USUALLY INCLUDES TEST DRIVER
- USUALLY PROVIDES PATH CERTIFICATION
- EFFECTIVELY LIMITED TO UNIT AND LOW-LEVEL COMPONENT TESTING
- CAN MASK HEISENBUGS

7810 09/22/92 ©B²

SIDE BENEFITS OF COVERAGE TOOLS

- PROGRAMMERS (ESPECIALLY) HAVE INFLATED VIEWS OF THE COVERAGE THEY ACHIEVE IN TESTING
- THEY THINK THAT IT IS 95% BUT IN FACT, IT'S CLOSER TO 50%
- FUNDAMENTAL RISK ASSESSMENT DATA
- QUANTIFICATION–A METRIC OF COMPLETION

7801 09/22/92 ©B²

PRESENT STATUS OF COMMERCIAL COVERAGE TOOLS

- ONLY STATEMENT, BRANCH, CALL TREE, AND PATH COMMERCIALLY AVAILABLE
- MANY EXCELLENT PRIVATE DATA FLOW COVERAGE TOOLS
- HEURISTIC BEHAVIORAL COVERAGE TOOLS (E.G., FUNCTION POINT, SPECIFICATION COVER).
- COMMERCIALIZATION HINDERED BY LACK OF DEMAND, NOT TECHNOLOGY

7805 11/07/92 ©B²

TEST DRIVERS

- WHAT: TOOLS THAT AUTOMATE THE SETUP, INITIALIZATION, EXECUTION, OUTCOME RECORDING, AND CONFIRMATION OF TESTS, ESPECIALLY FOR UNIT TESTING
- WHY: ELIMINATION OF TEST EXECUTION ERRORS SIMPLIFIES TEST DEBUGGING AND MAKES REGRESSION TESTING POSSIBLE
- PREREQUISITES: FORMAL, DESIGNED TESTS UNDER CONFIGURATION CONTROL

7750 09/22/92 ©B²

OVERVIEW OF DRIVER FUNCTIONS

- SETUP PHASE
- EXECUTION PHASE
- ANALYSIS AND POST MORTEM PHASE

7760 09/02/92 ©B²

DRIVER SET-UP PHASE (IDEAL)

- CONFIRMS THE HARDWARE/SOFTWARE CONFIGURATION
- LOADS PRE- AND CO-REQUISITE COMPONENTS
- FETCHES AND INSTALLS STUBS
- INITIALIZES ALL HARDWARE AND SOFTWARE
- INITIALIZES PATH INSTRUMENTATION
- SETS UP ACCESS TO DATA STRUCTURES

7770 09/02/92 ©B²

DRIVER EXECUTION PHASE (IDEAL)

- RE-INITIALIZES TO BASE CONDITION FOR EACH TEST
- LOADS AND CONTROLS INPUTS
- EVALUATES ASSERTIONS
- RESETS PATH INSTRUMENTATION
- TEST DEBUG SUPPORT
- PERFORMANCE DATA GATHERING

7780 11/08/92 ©B²

ANALYSIS AND POST-MORTEM PHASE (IDEAL)

- VERIFY PATH CORRECTNESS
- COMPARE ACTUAL OUTCOME TO PREDICTED OUTCOME
- PASSES CONTROL TO DEBUG PACKAGE ON TEST FAILURE
- PASSES TEST STATUS TO COVERAGE TOOL
- CHECK FOR RESIDUES AND BYPRODUCTS

7790 09/02/92 ©B²

EXECUTION AUTOMATION STATUS

- MOST COMMERCIAL TOOLS ARE STRAP-ON
- SOME INTEGRATION WITH CASE TOOLS
- SOME INTEGRATION WITH CONFIGURATION CONTROL
- COMMON TOOL BACKPLANE ON THE WAY
- BUT STILL A LONG WAY TO GO

7806 05/12/92 ©B²

TEST DESIGN AUTOMATION

THIS IS A COPYRIGHT NOTICE. NO PART OF THESE GRAPHICS MAY BE REPRODUCED OR USED IN ANY FORM BY ANY MEANS SUCH AS GRAPHICS, ELECTRONIC, OPTICAL, MECHANICAL, INCLUDING BUT NOT LIMITED TO PHOTOCOPYING, RECORDING, SCANNING, CONVERSION TO OTHER MEDIA, TRANSMISSION, OR STORAGE IN A COMPUTER OR AN INFORMATION STORAGE AND RETRIEVAL SYSTEM — WITHOUT THE AUTHOR'S WRITTEN PERMISSION.

BORIS BEIZER, ANALYSIS — 1232 GLENBROOK ROAD, HUNTINGDON VALLEY, PA 19006
215-572-5580

770X 09/23/92 ©B²

PREREQUISITES TO DESIGN AUTOMATION

- TEST EXECUTION AUTOMATION: BECAUSE ALL TEST GENERATORS PRODUCE VAST NUMBERS OF TESTS
- TEST CONFIGURATION CONTROL
- UNDERSTAND GENERATOR'S METHODS
  - THE UNDERLYING TEST TECHNIQUE (IF ANY)
  - THE CAVEATS AND LIMITATIONS
  - THE HIDDEN ASSUMPTIONS

7705 09/23/92 ©B²

DESIGN AUTOMATION OBJECTIVES

- MORE COMBINATIONS OF FACTORS
- BETTER COVERAGE
- MORE REVEALING TESTS
- FULL CONFIGURATION SUPPORT
- REDUCE TEST DESIGN LABOR CONTENT

7710 09/23/92 ©B²

DESIGN AUTOMATION TOOLS

- CAPTURE/PLAYBACK (c)
- BEHAVIORAL/REQUIREMENTS BASED (c)
- SYNTAX TESTING (c)
- STATE TESTING (c)
- STRUCTURAL TEST "GENERATORS" (c)
- DATA FLOW TEST "GENERATORS"
- REGRESSION GENERATOR/ANALYZER

"C" MEANS COMMERCIAL TOOLS AVAILABLE

7715 09/23/92 ©B²

CAPTURE/PLAYBACK TOOL: WHAT

- HARDWARE AND/OR SOFTWARE
- INSERTED BETWEEN (OR IN) TEST TERMINAL AND TEST EXECUTION AUTOMATION SYSTEM
- CAPTURES TESTERS KEYSTROKES AND SYSTEM RESPONSES
- COMPARES OUTCOMES TO PREVIOUSLY RECORDED OUTCOMES
- REPORT BY EXCEPTION
- TEST EDIT FACILITY (E.G., LINK TO WORD PROCESSOR)

7920 09/23/92 ©B²

STRUCTURAL TEST GENERATOR

(HYPOTHETICAL)

- ~~GENERATES~~ COVERING TEST SET DIRECTLY FROM SOURCE CODE
- ~~PREDICTS OUTCOME~~
- ~~SENSITIZES PATHS~~
- COORDINATES WITH DRIVERS

7725 09/23/92 ©B²

STRUCTURAL TEST GENERATOR

(REAL)

- SUGGESTS COVERING TEST SET BASED ON SOURCE CODE
- ~~PREDICTS OUTCOME (RESEARCH TOOLS)~~
- ~~SENSITIZES PATHS (RESEARCH TOOLS)~~
- COORDINATES WITH DRIVERS
- MORE A METHOD THAN A TOOL
- REVERSE ENGINEERING AID

7725.1 09/23/92 ©B²

DATA FLOW TEST GENERATOR

- POINTS OUT UNCOVERED DATA FLOW RELATIONS AND SUGGESTS PATHS
- ~~OUTCOME PREDICTION, SENSITIZATION~~
- COORDINATION WITH DRIVER
- MULTIPLE STRATEGIES INCLUDED
- MANY PRIVATE TOOLS, NONE COMMERCIAL
- THE FUTURE OF UNIT TEST TOOLS

7727 09/23/92 ©B²

REQUIREMENTS-BASED GENERATOR

- CAPTURES REQUIREMENTS AND ANALYZES FOR COMPLETENESS AND CONSISTENCY
- GENERATES TEST SUITES BASED ON A VARIETY OF FORMAL AND HEURISTIC METHODS
- ESPECIALLY GOOD AT GENERATING COMBINATORIAL TEST CASES
- GOOD FUNCTIONAL COVERAGE

7730 09/23/92 ©B²

REGRESSION/MAINTENANCE TOOLS

- SELECTS OR CONSTRUCTS STRUCTURAL TESTS BASED ON SOURCE CODE AND CHANGES TO IT
- IDENTIFIES WHICH TESTS MUST BE MAINTAINED AND HOW
- REGENERATES PREDICTED OUTCOMES AND SENSITIZES NEW INPUT VALUES
- NO COMMERCIAL TOOLS
- AREA OF INTENSIVE RESEARCH THAT PROMISES A REVOLUTION IN THE NEXT GENERATION OF TEST DESIGN AUTOMATION TOOLS

7734 09/23/92

©B²

NOT ENOUGH TOOLS?

- 1200+ *TEST* TOOLS ON MARKET
- AT LEAST 50 OF THEM ARE GOOD
- 200+ PAPERS ON R&D TOOLS PUBLISHED 30-40 NEW ONES EACH YEAR
- UNTOLD NUMBER OF UNPUBLISHED INTERNAL TOOLS DEVELOPED AND USED
- TOOLSMITHS AT WORK WORLDWIDE

WE HAVE FAR MORE TOOLS THAN THE INDUSTRY CAN SUPPORT

1191.1 09/10/92

©B²

TEST DESIGN AUTOMATION STATUS

- WEAK EXECUTION AUTOMATION SUPPORT
- UN-INTEGRATED COMMERCIAL TOOLS
- AD-HOC, SPECIAL INTERFACES
- BIG GAP BETWEEN LABS AND PRACTICE
- HEAVY TRAINING INVESTMENT
- POOR INTEGRATION WITH CASE

7745 05/12/92

©B²

PERSPECTIVE ON TESTING

THIS IS A COPYRIGHT NOTICE. NO PART OF THESE GRAPHICS MAY BE REPRODUCED OR USED IN ANY FORM BY ANY MEANS SUCH AS GRAPHICS, ELECTRONIC, OPTICAL, MECHANICAL, INCLUDING BUT NOT LIMITED TO PHOTOCOPYING, RECORDING, SCANNING, CONVERSION TO OTHER MEDIA, TRANSMISSION, OR STORAGE IN A COMPUTER OR AN INFORMATION STORAGE AND RETRIEVAL SYSTEM -- WITHOUT THE AUTHOR'S WRITTEN PERMISSION.

BORIS BEIZER, ANALYSIS -- 1232 GLENDOON ROAD, HUNTINGDON VALLEY, PA 19006
215-572-5580

©B²

PERSPECTIVE ON TESTING

AMMUNITION TO USE ON NON-BELIEVERS

- ALL ADVANCED TEST TECHNIQUES ARE TOOL INTENSIVE
- IMPORTANCE OF TOOLS AND TEST AUTOMATION
- TOOL BUILDING VERSUS TOOL BUYING
- REALISTIC PAYOFF PROJECTIONS
- TOOL PENETRATION--REALITY VS ASPIRATIONS
- SOLUTION TO THE TOOLS PENETRATION PROBLEM

©B²

TECHNIQUES ARE TOOL-INTENSIVE

- MOST ADVANCED TEST TECHNIQUES, BE IT FOR UNIT TESTING, INTEGRATION TESTING, SYSTEM TESTING, REGRESSION TESTING--ARE TOOL-INTENSIVE
- MANUAL ALGORITHM USE IS POSSIBLE BUT NOT COST-EFFECTIVE
- SO WHAT! DOES IT MAKE SENSE THAT THE VERY HEART OF AUTOMATION TECHNOLOGY AND THE TESTING OF THAT TECHNOLOGY BE BASED ON MANUAL METHODS? WHY NOT STONE AXES?
- WE ARE JUST FOLLOWING THE PATH THAT OUR HARDWARE COUNTERPARTS STARTED ON 20 YEARS AGO. WHY SHOULD ANY ONE EXPECT IT TO BE DIFFERENT FOR SOFTWARE DEVELOPMENT?

©B²

TOOL BUILDING AND BUYING

- ABOUT 5% OF THE SOFTWARE DEVELOPMENT BUDGET SHOULD BE ALLOCATED TO TEST TOOL BUILDING AND/OR BUYING
- CENTRALIZING TOOL KNOW-HOW IN A TOOLSMITH GROUP IS A GOOD INVESTMENT
- EVEN IF YOU BUILD YOUR OWN, BUYING COMMERCIAL TOOLS IS A GOOD WAY TO LEARN WHAT YOU NEED
- SOME TOOLS, ESPECIALLY FOR SYSTEM TESTING SUPPORT, YOU WILL HAVE TO BUILD

6905 09/23/92 ©B²

TOOLS AND TEST AUTOMATION IMPORTANCE IN JAPAN

A SURVEY OF 200 JAPANESE SOFTWARE DEVELOPMENT GROUPS (T. NOMURA, "USE OF SOFTWARE ENGINEERING TOOLS IN JAPAN—9'TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, MONTEREY CA 3/30 - 4/2, 1987) RANKED SOFTWARE TEST TOOLS AS HAVING THE HIGHEST PRODUCTIVITY RETURN ON INVESTMENT.

- AHEAD OF ADOPTING A STANDARD METHODOLOGY
- AHEAD OF CASE
- AHEAD OF ANY SOFTWARE DEVELOPMENT TOOL
- AHEAD OF REVIEWS AND INSPECTIONS

IS THERE A LESSON IN THAT FOR US?

6906 09/10/92 ©B²

REALISTIC PAYOFF PROJECTIONS

- PAYOFF IS LONG-TERM ONLY. 2-4 YEARS AND SEVERAL PROJECTS
- A CAPITAL INVESTMENT—NOT JUST AN EXPENSE
- THE TRIAL PROJECTS NEED MORE TIME AND MORE LABOR, NOT LESS
- COST AND TIME BREAK-EVEN ON THE FIRST PROJECT IS A ROARING SUCCESS, BUT UNLIKELY
- SOFTWARE QUALITY WON'T IMPROVE AT FIRST—IT IS A CUMULATIVE EFFECT THAT REQUIRES THE BUILDUP OF AUTOMATED TEST SUITES AND A CONTROLLED TEST ENVIRONMENT.

6908.1 09/10/92 ©B²

THE TOOL PENETRATION PROBLEM

- THE PROBLEM: SURVEYS OF TOOL BUYERS, BUILDERS, AND USERS A YEAR OR TWO AFTER THE TOOL WAS BOUGHT SHOWED THAT THE TYPICAL USAGE WAS ONLY ABOUT 5%. USUALLY, THE TOOL WAS ONLY USED BY THOSE WHO SPONSORED IT, DESPITE THE OBVIOUS PAYOFF
- THE REASON: NO ALLOWANCE HAD BEEN MADE FOR THE TIME IT TAKES TO BECOME ADEPT AT USING THE TOOL. TESTERS AND PROGRAMMERS WERE EXPECTED TO LEARN IT ON THEIR OWN TIME AND STILL MEET TIGHT SCHEDULES. IT WAS EASIER TO CONTINUE MANUALLY

DESPITE TOP MANAGEMENT, MIDDLE MANAGEMENT, AND GRASS ROOTS SUPPORT AND ENTHUSIASM!

6910 0923/92 ©B²

PENETRATION PROBLEM SOLUTION

- ADEQUATE (COMPANY) TIME TO LEARN TOOL IS ESSENTIAL. A WEEK FOR SIMPLE TOOLS SUCH AS CAPTURE/PLAYBACK, MONTHS FOR ELABORATE TOOLS, TOOL KITS OR TESTING ENVIRONMENTS
- ADJUST SCHEDULES AND HUMAN RESOURCES ACCORDINGLY
- MANAGEMENT MUST ASSURE WORKERS THAT THE INITIAL PRODUCTIVITY DROP AND SCHEDULE DELAY IS EXPECTED AND WON'T BE HELD AGAINST THEM AT REVIEW TIME
- EXPECT BREAK-EVEN NO SOONER THAN SECOND OR THIRD PROJECT.

6912 09/23/92 ©B²

THE MESSAGE RECAP

- IT'S AN OLD AND MATURE TECHNOLOGY
- IT'S A TOOL-INTENSIVE TECHNOLOGY
- MORE THAN ENOUGH TOOLS OUT THERE TO GET STARTED WITH
- THERE'S A LOT MORE TO SOFTWARE TESTING THAN "BE SURE TO TEST THOROUGHLY"!
- TESTING IS NOT KEY-POUNDING!

1189 09/23/92 ©B²

"Three Ways to Improve your Testing"

TUTORIAL "B"

Mr. Brian Marick
Testing Foundations

Mr. Brian Marick is President of Testing Foundations which helps companies acquire essential testing skills and tools. Because practitioners are justifiably suspicious of those who talk about software development but never actually do any, he spends half his time building and maintaining tools. Mr. Marick worked in the industry for ten years as a tester, developer, and line manager, mostly on operating systems and compilers.

Note that slide 1 is on the bottom.
Software...

Topics - Going Backwards

- (Approximate) measures of test quality
- Effective test implementation
- Improving test design based on experience

Three Ways to Improve Your Testing

Brian Marick
Testing Foundations
809 Balboa, Champaign, IL 61820
(217)351-7228
marick@testing.com

Assumed Current Practice

- Developer unit testing
  - Somewhat incomplete
  - Often not repeatable
- Repeatable tests at subsystem or system level
- Based mostly on external interface

Structure of the Segments

- A practical goal
- What to do now
- How to get to the goal

More Detailed Coverage

```
% lc -} lc.c
                Pure Code    ...
lc.c            120          ...


% echo "main(){}" | lc
                Pure Code    ...
                1            ...
```


A Pattern for Using Coverage

- Design your tests as you do now
- Measure what they do
- As time allows, improve them
  - Coverage goal: Better than last time

Relational coverage

- A simple, though common, misuse

if (A > 5) if (A ≥ 5)

A	>	≥
0	false	false
10	true	true
5	false	true


```
while (—argc > 0 && (**++argv == '-'))
{
   if ( (*argv)[1] == LCURL || (*argv)[1] == RCURL)
      white_bracket = TRUE;
   else if (sscanf (*argv + 1, "%d", &page_size) == FALSE)
   {
      fprintf (stderr, "...", *argv);
      exit (BAD_FLAG);
   }
}
```

---

operator > might be >=. (L==R)

condition 1 (argc, 1) was taken TRUE 43, FALSE 0 times.

Branch Coverage

```
 option processing
 if (argc == 0)
{
        tally_file (stdin, &file_tally);
        show_header ();
        show_tally ("", &file_tally);
}
```

"lc.c", line 153: if was taken TRUE 0, FALSE 29 times.


```
while (—argc > 0 && (**++argv == '-'))
{
    if ((*argv)[1] == LCURL || (*argv)[1]==RCURL)
        white_bracket = TRUE;
    else if (sscanf (*argv + 1, "%d", &page_size) == FALSE)
    {
        fprintf (stderr, "...", *argv);
        exit (BAD_FLAG);
    }
}
```

---

condition 1 (<...>[...], 1) was taken TRUE 0, FALSE 14...

More Infeasible Coverage

```
if (fclose (fp) == EOF)
        panic (PANIC, "Fclose error.");
```

"lc.c", line 172: if was taken TRUE 0, FALSE 43 times.

Loop Coverage

```
if (argc == 0)
        ...
else
{
        show_header ();
        for (index = 1; index ≤ argc; index++)
        {
                ...
        }
```

---

loop zero times: 0, one time: 19, many times: 10.

Reasons

- Avoid repeating simple developer tests
- Avoid test clustering in input space
- Combinations suggest new test conditions

An Example

- Case of the input text
  - upper-case only
  - lower-case only
  - mixed case
  - no alphabetic characters
- argument A
  - empty list
  - contains matching count

What Can Be Done Now

- Make tests more complicated (within reason)
- Notice when stuck in a rut
- Seek ways to enlist chance
- Check design rules
- Problems
  - Bookkeeping
  - Combining rules hurt traceability
  - Diagnosing the bugs

Test Design Rules (Example)

- Initialize and check the background

fill(A, 10)

0	1	2	3	4	5	6	7	8	9	**X**

Test Implementation Difficulties

- How to force internal variables from external interface
- How to ensure that failures are visible
  1. **Add testing support code**
  2. Use smaller subsystems
  3. Manually check test conditions

The Scale of Testing

- Unit + integration testing is too expensive
  - Especially during maintenance
- Use a subsystem as the harness
  - Makes automated tests tractable

Growing the Testability of the System

- Begin with debugging code
- Modify for use in testing
- Retain in final product

A Practical Goal for Test Implementation

- Test design drives internal harness design
  - Induce special conditions
  - Expose internal state
  - Expose trace information
  - A coherent in-system debugger/harness is best
- Internal harness is part of the product
  - Also useful for customer service

Continued Growth

- Provide standards, examples, libraries
- (Don't make debugging harder.)
- Move toward in-system debugger/harness
- Potential eventual redesign
  - But only after proof-of-concept

Problems

- "We should test what we ship."
- "Cluttering up the system with testing crud"
  - Sometimes valid
  - Shifting work around
  - People issues

Practical Goal: Subsystem Test Conditions

- Use subsystem specification (not discussed)
- Use unit test conditions
- Use integration test conditions
- Combine all of them into complex tests
- Coverage as approximate measure of quality

Deciding What to Test Based on Past Experience

- Test conditions for subsystem testing
- Faults that testing misses
- System testing (race conditions)
  - Universality decreases with scale

A Cliche

```
for (i = 0 ; i < LENGTH; i++)
    if (array[i] == sought)
        break;
if (i == LENGTH)
    error("Not found");
do something useful to element i
```

- Observed in code or inferred from specification

Unit test conditions

- Using specification (not discussed)
- Scanning specification and code for cliches
- Skip details of code
  - Relational operators
  - Branches
  - Most will be automatic
  - Will catch remaining with coverage
- Hunches

Some Test Conditions

- Element found in last position
- Element not found
- More than one match
  - Low yield countered by combining rules
  - Useful for enlisting chance

Testing Cliches

- Programs are built by composing cliches
- Data "types" and operations
- Cliches have cliched faults
- Catalogued test conditions

Integration test conditions

- Concentrate on faults of omission
- Begin with intrinsically distinct cases of called function
- Test conditions from called subroutine
- Bug reports
- Test conditions for passed data (especially globals)

Using called function's specification

strncmp(passed, val, strlen(passed));

- result: **passed** is smaller
- result: **passed** is larger
- result: args are equal
- input: val shorter or equal
- input: val longer

Designing Tests Based on Experience

- **Look** at bugs testing missed
- **Ask** why they were missed
- **Invent** an improvement to design process

What Can Be Done Now

- Focus on subsystem specification
- As time allows, add more conditions
  1. Unit specifications (actual or inferred)
  2. Integration conditions
  3. Cliche conditions
- Measure coverage
- Improve tests as time allows

Making Allocation Bugs Testable

- Debugging malloc (comp.sources)
- Sentinel (703-430-9247, cpcahil@virtech.vti.com)
- Purify (415-903-5100, info@pure.com)

Directed Inspections

- Testing is poor at detecting some kinds of bugs
- Inspect for them
- May be done in context of other inspections/code reads
- Scan the inspection catalog, asking questions
- Application-specific questions also useful

THREE WAYS TO IMPROVE YOUR TESTING

Brian Marick
Testing Foundations

Everyone needs to improve quality. A few companies are in a quality crisis, where dramatic and expensive changes may be the only recourse. But most companies aren't. Most companies need to improve quality in ways that are largely non-disruptive, that don't jeopardize a steady stream of new releases and new products. They seek modest but sustainable improvements, with a minimal risk of wasted effort.

This paper discusses three inexpensive, continuous, incremental improvements that can be made in software testing. They are:

(1) Using coverage to measure what your tests do, in order to discover weaknesses in test design.

(2) Using defect analysis to record what your tests missed, then building on that experience with defect catalogs.

(3) Increasing testability via debuggability, thus reducing the cost of testing.

BIOGRAPHICAL SKETCH

Brian Marick worked in industry for ten years as a tester, developer, and line manager, mostly on operating systems and compilers. Joint research with the University of Illinois led to internal consulting and then the creation of Testing Foundations. It exists to help companies acquire essential testing skills and tools. Because practitioners are justifiably suspicious of those who talk about software development but never actually do any, he spends half his time building and maintaining tools, some freely available.

Author's address: Testing Foundations, 809 Balboa, Champaign, IL 61820
Phone: (217) 351-7228
Email: marick@testing.com

An earlier version of this paper was first presented at the 1992 Pacific Northwest Software Quality Conference, Portland OR, October 19-21. It is Copyright © Pacific Northwest Software Quality Conference and is reprinted with permission.

1. Measuring and Improving Effectiveness Through Coverage

A test suite would be perfectly effective if it revealed every bug[1] in a program. While we have no way of measuring this, we still need to judge test quality. One approach is to measure how tests exercise the code.

Suppose you're testing a large system, especially one that's undergone a lot of perfective and corrective maintenance. The following is a very real risk:

(1) Some part of the system, call it NS, is not used in normal testing.

(2) Changes to other parts of the system break NS. It doesn't work at all.

(3) A customer is the first person to use a feature of the system that exercises NS.

This should have a relatively low chance of happening (though I've seen it depressingly often), but it has a very high cost. (Having a basic feature simply not work is a wonderful way of losing repeat business.) High risk translates into a task for testing: make sure all of the system's features have been exercised. Of course, you do this based on the system's specification or user documentation, but it's easy to miss a feature in an "evolved" system. How can you detect your testing mistakes and omissions?

A tactic that works well is to *instrument* every routine in the system so that it records whether it's been entered. This can have very low runtime overhead. Run your test suite. Any unused function likely corresponds to an untested feature. You should deduce what feature that is and design tests for it.

This example establishes a pattern for testing:

(1) Design your tests however you do now.

(2) At small additional cost, measure what your tests do. This is referred to as *test coverage*.

(3) Use uncovered parts of the code as pointers to weak test design. Improve your tests as time allows.

This type of coverage, *routine coverage*, is appropriate for large-scale system testing. I find other types of coverage useful for system components.

Types of Path-Based Coverage Useful for Subsystems

All of the measures described below (and more) are implemented by my tool GCT, which is available via anonymous FTP. Fetch the file cs.uiuc.edu:pub/testing/GCT.README for instructions. I can also supply it on tape for a materials and handling fee.

Branch coverage requires every branch to be executed in both directions. For this code

```
if (arg > 0)
{
 counter = 0;      /* Reinitialize */
}
```

branch coverage requires that the if's test evaluate to both true and false. These two requirements are the two *coverage conditions* for branches. Taking the branch both ways *satisfies* them.

---

[1] Technically, I should use the terms *fault*, for an actual incorrectness in program text, *failure*, for the incorrect output resulting from a fault, or *error*, for the programmer's mistake that caused the fault. Often, the distinction is unimportant, so I'll use "bug" unless I mean one of the senses in particular.

Some kinds of faults may not be detected by branch coverage. Consider a test-at-the-top loop that is intended to sum up the elements of an array:

```
sum = 0;
while (i > 0)
{
 i = i - 1;
 sum = pointer[i];          /* Should be += */
}
```

This program will give the wrong answer for any array longer than one. This is an important class of faults: those that are only revealed when loops are iterated more than once. Branch coverage does not force the detection of these faults, since it merely requires that the loop test be true at least once and false at least once. It does not require that the loop ever be executed more than once. Further, it doesn't require that the loop ever be skipped (that is, that i initially be zero or less). *Loop coverage* [Howden78] [Beizer83] therefore requires that loops be iterated zero, once, and "many" times.

Multi-condition coverage [Myers79] is an extension of branch coverage. In an expression like

```
if (A && B)
```

multi-condition coverage requires that A be true in some test, A be false in some test, B be true in some test, and B be false in some test. Multi-condition coverage is stronger than branch coverage. These two inputs

$A == 1, B == 1$
$A == 0, B == 1$

satisfy branch coverage, but do not satisfy multi-condition coverage.

Multi-condition coverage provides a good illustration of how coverage can indirectly reveal bugs. Consider this common C fault:

```
if (A > 0 && B = 1)
```

The second clause is incorrect. Branch coverage does not force detection, since

$A == 1, B ==$ *anything*
$A == 0, B ==$ *anything*

will satisfy it. However, multicondition coverage is impossible to satisfy because of the fault - a fact that will be revealed when coverage results are examined. A similar observation was made by [Holthouse79].

Using Coverage

You would expect a thorough test suite to have very high coverage. After all, every branch in the program should correspond either to some system feature or some special case in the implementation, else why does it exist? Since any perfect test suite will test all the features and special cases, it must exercise all the branches.

However, high coverage alone does not necessarily mean the test suite is thorough. The "fault of omis-

sion" is a common type of bug ([Basili84], [Glass81], [Ostrand84]). It is exemplified by this code:

```
a = b / c;
```

which should read

```
if (c != 0)
        a = b / c;
```

Since you are testing the incorrect program, a branch coverage tool cannot tell you that you need a test where c=0 - it can't measure the coverage of a branch that doesn't exist. So a test suite that achieves perfect coverage may still not be good enough.

Using coverage as a test design technique will reduce your chance of finding this important class of bugs. Coverage should instead be a way of evaluating your existing tests, finding incompletenesses, and using those incompletenesses to focus attention on weaknesses in your test design. It is an incremental improvement to your normal design process. Here is what I do when GCT shows me a line like

> "lc.c", line 271: if was taken TRUE 0, FALSE 28 times.

(1) I ask if the branch is impossible (as, for example, in a sanity check or automatically generated code). If so, I ignore it.

(2) I ask what feature corresponds to this branch. Does it involve handling of quoted characters? Resource exhaustion? Collision handling in a hash table? I now know that that feature has been undertested in at least one way.

(3) If it's been undertested in one way, there's a good chance it's been undertested in other ways. I redo the design work I should have done in the first place, without worrying about the branch. Good tests will force the branch -- what I'm after are the tests that won't cause any increase in coverage but might find bugs. These are typically tests for omitted special-case code. This sounds like a lot of extra work, but it's rarely much harder than just satisfying the branch.

(4) I will sometimes decide that the branch is too much trouble; for example, it may be that the true case would be very hard to induce, the program's actions in that case are "obviously" correct, and I can think of no other related tests. I rarely choose to skip testing, though. In the past, I've almost done so, decided I'd test anyway, and found a bug.

What is the additional cost of measuring coverage and redesigning? In an experiment, I found that it increased the total cost of test design by 4% [Marick91]. Using coverage in my day-to-day testing shows a similarly small cost.

An Example

I spent roughly a month testing the essential core of GCT. The overall coverage of those tests was 86.44%. Of the 129 uncovered coverage conditions, 74 were impossible, 17 were from deliberately untested temporary code, 24 were due to a major feature that had never had regression tests written for it, and 14 corresponded to 10 untested minor features.

Completely testing the major feature required 4 hours of work for design and implementation. This testing discovered one bug. Note, though, that a test designed solely to achieve coverage would likely have missed the bug. (That is, the uncovered conditions could have been satisfied by an easy and obvious - but inadequate - test.)

Testing the minor features required 2 hours. Branch coverage discovered one pseudo-bug: dead code. A particular special case check was incorrect. It was testing a variable against the wrong constant. This check could never be true, so the special case code was never executed. However, the special case code turned out to have the same effect as the normal code.

After fixing the bugs, I reran the entire test suite to see if all expected coverage conditions had been satisfied. I found that one had not. A typo caused the test to miss checking what it was designed to test. Finding this test error and correcting it took 1/2 hour. (Running the test suite took none of my time, since

it's completely automated.)

This work - 6.5 hours - was a small part of the total testing effort. Of the two bugs, one could never cause a failure, and the other would cause a failure only in rather obscure C programs, so the effort of finding them was arguably not worth it. However, the knowledge that I had a thorough test suite made me quite confident that GCT produces correctly instrumented C code, and that confidence has been borne out.

Of course, the time spent was a small fraction only because the initial tests were so thorough. What do you do if your tests leave many test conditions unsatisfied -- 40%, say? You might decide to spend the time to achieve complete coverage. However, there's a considerable danger that an effort to add 40% coverage will degenerate into a blind writing of tests that satisfy coverage. Faults of omission will be missed.

It would be better to improve your test process so that each new testing effort naturally results in higher coverage than the one before it. But this begs the question of *how* you improve your test design so that these higher coverages are inevitable. Coverage results - more particularly, careful thinking about the types of coverage that are missed - can lead to improvements in test design. However, remember that coverage is only an approximation of effectiveness. For process improvement, you'll do better to deal directly with effectiveness by examining what your testing misses. This is the next topic.

2. Designing tests based on experience

This section describes several examples of how you can use the bug history of a system to learn how to design better tests and better measures of their quality. The basic process is always the same:

(1) LOOK at the bugs that testing missed.

(2) ASK why they were missed.

(3) INVENT an improvement to the testing process so that they're less likely to be missed next time.

Simple faults that allow coverage

Programmers often misuse relational operators. They write code like

```
if (A < 5)
  return 1;
else
  return 0;
```

when they should write

```
if (A <= 5)
  return 1;
else
  return 0;
```

In this particular case, the only value of A that causes a failure is A=5. For all other values, the incorrect program yields the same value as the correct one. Because testers know that this type of misuse is likely, testing "boundary conditions" is one of the basic test design rules [Myers79].

GCT can measure whether we've tested boundary conditions. Just as it can measure whether the branch was taken true or false, it can measure whether A=5 during the evaluation of the if's test. This type of coverage is called "relational operator coverage", and I've found it useful in testing. For example, after writing tests for part of GCT, I found that one relational operator was not covered. It turned out to be impossible to cover because the code was wrong. On my development machine, it caused no failure. On another machine, GCT might not have worked at all. That's a pretty severe bug; finding it paid for all the time spent on coverage.

Relational operator coverage is a subset of *weak mutation coverage* [Howden82]. GCT measures other types of weak mutation coverage, but relational coverage is the only one I'm sure is worthwhile.

You should use relational operator coverage just like other coverage: design your tests first, then measure coverage, then upgrade your tests.

Widely Applicable Test Condition Catalogs

"Testing for inequality" is included in every programming language because it's something every programmer wants to do. It's a universal cliche. Many other cliched actions are not built into every language. Some languages implement them, others provide versions of them as library routines, and some leave it to the programmer to implement them from scratch. Here are some examples:

(1) traversing an array.

(2) searching an array.

(3) sorting a list.

(4) using a temporary file.

The same is true of data types. All programming languages have integers; most have floating point numbers, a few have lists, a very few have hash tables. A data type gets put into a language (or defined by users) when enough people have found the abstraction useful - when it becomes a cliche.

The important thing about cliches is that they are mis-implemented or misused in cliched ways. We've already seen how the relational operator cliche is typically misused. That misuse often becomes a cliched mis-implementation of traversing an array - how many such traversals contain off-by-one errors? How many list operations fail on circular lists? How much floating point code uses exact equality when inexact is better? How many routines that take pointers fail when given a null pointer, or when given two pointers that happen to point to the same object?

Cliches can be catalogued, as in textbooks on algorithms. So can common faults in those cliches. [Johnson83] and [Spohrer85] are collections of such faults for novice programmers.

For purposes of testing, a fault catalog isn't quite what we want. For A<5, we don't care that A<=5 is the correct program - let the person fixing it worry about that. All we really need to know is that A=5 is a good *test condition* to use to find misuses of relational operators.

The distinction between faults and test conditions is useful. For elementary cliches like <, it's easy to exhaust the plausible alternatives. This is much more difficult when considering all the ways "searching" could be misused or incorrectly implemented.[2] Fortunately, relatively few test conditions suffice to detect almost all these faults, because people make cliched mistakes (and because one good test condition may detect many possible faults).

To find these few test conditions, you examine bug reports, asking what cliche the bug involves (if any) and what test condition would have detected it. Then you write that information down in a *test condition catalog*. [Marick92a] is such a catalog. I started it by writing down test conditions that are common knowledge (such as boundary conditions) or that I've absorbed in my experience as a tester. I then looked for new cliches in introductory programming texts and reference manuals for higher-level programming languages like Smalltalk. When I found a cliche, I asked myself how I'd test it. After that, I augmented the catalog by examining bug fixes (mostly those posted on Internet bug newsgroups).

Here's an example of an entry, one for a program that searches. (There are more specific entries for special kinds of searches.)

- Match not found
- Match found (more than one element, exactly one match)
- More than one match in the collection
- Single match found in first position INLINE
  (it's not the only element)
- Single match found in last position
  (it's not the only element)

The first three check for misuses of "searching". The first is used because programs that search sometimes incorrectly assume that the search will never fail. The second just ensures that you design a test where the program does search successfully. The third checks for another common bug, where a program should

[2] See [Morell90] and [Howden87], though, for analytical work on larger code fragments.

search for several matches but incorrectly searches for only one.

The last two test conditions check boundary conditions. Most faults they'll discover are the direct result of relational operator misuses, but it's inefficient to worry about those details if you can find the faults without them. (The INLINE notation is explained in the appendix.)

Some of these conditions do not apply to all programs - for example, it may truly be impossible for a match not to be found. Some may apply to a particular program, but not be useful. For example, if the program only uses a first match, why try a list with more than one match? However, it's usually easy to include such a test condition in a test you're writing anyway, so it costs almost nothing to use it.[3]

An expert tester has a catalog very like this in his or her head. By writing it down, you make it available to novices and experts with imperfect memories. The result is an improvement in test effectiveness. The catalog will grow as new bugs prompt additions.

Application-specific faults

Of course, some bug reports will have nothing to do with commonly used cliches. Instead, they'll report bugs particular to your system. Many such bugs will be due to subtle interactions between subsystems. Others will actually be errors in the specification of the system, not its implementation.

There's certainly every reason to think about these bugs as hard as cliched bugs. You will be able to use some of them to build an application-specific test condition catalog (though these faults seem to be harder to categorize). At the very least, you'll discover what parts of the system are bug-prone, so you know where to test harder. You'll also discover how customers use the system, so you'll be able to make system tests more representative.[4]

You may also discover that this focusing of attention suggests not only test designs, but ways of measuring the quality of tests. For example, in multiprocessor systems with locking, locking errors are a major risk (because they often cause unacceptable failures). Three types of bugs are:

(1) A processor fails to acquire a lock before updating a data structure. If another processor is simultaneously updating the data, it is likely to be corrupted.

(2) A processor fails to release the lock. The system will likely grind to a halt as other processors wait for that lock to be freed.

(3) The system may deadlock when one processor acquires lock 1 and attempts to acquire lock 2, while at the same time a second processor acquires lock 2 and attempts to acquire lock 1. Neither can proceed until the other gives up its lock.

The conventional approach to testing for locking errors is stress testing. By making sure that a shared datum is heavily used, you increase the chance that locking faults involving that datum will manifest themselves. The problem is with "making sure" -- despite your attempt to stress a particular part of the system, all the processors might be bottlenecked in some other part. Perhaps they are all waiting on a lock that's letting only one at a time through to the part of the system you're testing - and therefore you're not testing its locking at all.

This danger motivated *race coverage*. Race coverage counts the number of times a routine has had two or more processors (or threads of execution) in it at once. Race coverage thus provides direct evidence of whether stress tests are actually stressing particular routines. Those routines that have not been "raced" can be targeted with new stress tests.

Note that race coverage - like most coverage - is an indirect measure. It is not measuring contention for data structures, only entries to routines. (This allowed much faster implementation and tolerable runtime overhead.) Nevertheless, it was useful for generating tests that found locking bugs of all types.

---

[3] Test design - how to most effectively combine test conditions into tests - is outside the scope of this paper.

[4] [Petschenik85] contains a useful discussion of this type of system testing.

A note on inspections

Suppose your program allocates memory but fails to release it. Such "memory leaks" typically have no effect on the output, so they're not visible during a test. You only see them when the program runs out of memory after running for many hours. This is an example of a fault difficult to detect with testing. These are typically failures to honor what [Perry89] calls "obligations". That is, one part of a program uses the result of another part of the program. As a result, it incurs an obligation to perform some action later. The bug is that it fails to fulfill the obligation - it fails to free the memory, or it fails to close the opened file. (For the particular case of memory leaks, there are specialized tools that can make the failures more visible during a test, but they do not solve the general problem of obligations.)

Such faults may best be detected by inspecting the code. An inspection catalog can be built up alongside the test condition catalog, and in the same way - by examining faults. Here's an example from the "freeing data" section of [Marick92b]:

- Are you freeing already-freed storage?
- Does the freed storage still have pointers to it, pointers that could still be dereferenced?
- Are you freeing storage that is not supposed to be freed? (Often, allocated with a routine that says "do not free returned value" in a code header.)
- Is it possible for the pointer being freed to be null?
  (Note: POSIX-compliant implementations of free accept null pointers. Not all implementations conform.)
- Are you reusing the contents of freed data?
  (Some versions of free overwrite some parts of the freed data; most do not.)

Such a checklist is easily incorporated into whatever type of code reads or code inspections you do now.

Checklist items may also be specific to your application. For example, after a couple of years of working on GCT, I know to apply this checklist after I add a new type of coverage:

```
* 1. Have only copies of temporaries been used?
* 2. Has the original value of temporary_id been freed?
* 3. Does the instrumented tree have the same type as the original tree?
* 4. If locals are set pointing to parts of a tree, is the tree still the same when
*    the locals are used (else they might point to the wrong place)?
* 5. Are locals set to values derived from the root of an instrumented tree?
* 6. Do the mapfile entries read from left to right?
* 7. Is SELF built into the resulting instrumented tree?
* 8. Is any temporary_id that is an initialized non-static declared
*    OUTERMOST? (See gct-temps.c)
*/
```

(Note that this checklist is a comment at the head of the source file to which it applies, a convention I recommend.)

Detection and Prevention

Of course, you'd rather prevent faults than create them, find them, and correct them. The same examination of reported bugs that improves test design can also improve fault prevention. [Mays90] is an excellent description of this process. They report that examining faults and implementing preventative procedures costs only about 0.5% of total development effort, whereas they estimate a direct benefit of at least double and possibly triple the cost.

One type of fault prevention is simply to use test condition catalogs during development. Many of the faults those test conditions find are the results of unexamined assumptions; if the developer simply reads the test condition catalog and wonders if her code handles a condition, the fault may be forestalled. This of

course then raises the question of why you should bother writing the test.[5]

A prolonged discussion of this topic would take us far afield from ways of gradually improving testing. The answer, in brief, is that our fault prevention skills are as yet too shallow and too narrow.

(1) **shallow:** We're not good enough at preventing even those errors we think of. The developer may try to handle the condition, but get it wrong. The cost of testing is less than the risk of trusting prevention alone.

(2) **narrow:** There are too many errors we don't even try to forestall, because we don't think of them. Because testing is rather good at serendipitously finding bugs it wasn't "supposed" to find, it's a useful safety net for error prevention. This is especially true if the tests are made as complex as is tractable. Each should probe many test conditions at once. This increases the chance that you will accidentally probe some unsuspected test condition. [Knuth89] has a nice description of this approach applied to the TeX text processor.

As our prevention skills increase, they will reduce the need for testing. More importantly, the focus of testing will shift toward faults harder to prevent, especially design and specification faults caused by faulty or unquestioned assumptions.

3. Reducing Test Cost

It's not enough to design better tests; you must also reduce the cost of test construction.

The key to long-term reduction of testing costs is an automated regression test suite. Manual tests are simply too expensive to run often enough. Bugs are found long after they're caused, so the cost of diagnosing and fixing them is much higher. Further, bugs may not be found at all: if the tests are judged manually, by a person looking at the output to see if it's correct, many failures will be missed ([Basili87], [Myers78]).

While the long-term benefits of an automated test suite are enormous, the startup cost can be high. For example, testing individual routines in isolation (unit testing) is too expensive except for the most critical routines. You spend too much time writing test drivers and the stubs that emulate the subroutines the routine-under-test calls. Initial test development is bad enough; maintenance becomes a nightmare, and the unit tests are often abandoned.

To reduce the cost, separate test design from test implementation. The same unit tests can often be implemented using the system itself as the driver.

(1) The design calls for certain inputs to be delivered to the routine. Deduce what input to the system will cause that delivery.

(2) The design calls for certain expected results from the routine. From these, predict the expected results from the system, and use those results to judge the routine's correctness.

(Of course, the test design may be based not on individual routines, but on interfaces to larger subsystems. If test design is backed up with careful use of coverage, we may not lose much in effectiveness. This discussion applies to any instance of testing some body of code via the system that contains it.)

This approach usually makes at least part of the test suite easier to automate. There are two potentially serious problems:

(1) You may make a mistake designing inputs. Perhaps your chosen input is a special case for the system, handled by special-case code and never delivered to the routine at all. This is the lesser of the two problems, especially since coverage often points to it.

(2) It may be hard to determine correctness of an interior routine from correctness of the system. For example, suppose the routine produces a complicated linked list structure and the system produces 0 or 1. Information is lost in that transformation ([Voas91a], [Voas91b]). A correct and incorrect linked list structure may both produce 1, so you can't test the interior routine via the system.

This difficulty is usually because systems are not designed to be testable - they are not designed to be easy to force into known states, and they do not provide internal results in a way that allows them to be compared to expected results. Of course, redesigning systems for testability is hardly the sort of gradual improvement this paper is about. What is needed is a way of "growing" the testability of the system, so

[5] Hamlet has considered related questions. See, for example, [Hamlet92].

that every month you have a better prototype of a truly testable system. This can be done by taking advantage of what the programmers are already doing:

They're writing debugging code.

Consider the GNU Make program. It maintains an internal hash table of files. The programmer wrote debugging code to print out the entries in the hash table. The output looks like this (edited for conciseness):

```
# Files

a::
# Command-line target.
# Implicit rule search has not been done.
# Implicit/static pattern stem: ''
# Last modified Mon Jul 27 10:37:55 1992 (712251475)
# File has been updated.
# Successfully updated.
# 18 variables in 23 hash buckets.
# average of 78.3 variables per bucket, max 2 in one bucket.
```

This is almost what you would need to test the hash table manipulation routines, but there are several problems:

(1) The file's location in the hash table is not printed (in particular, we want output that shows how hash table collisions have been handled).

(2) Variable values are translated into text useful to the debugger (such as "successfully updated"). For testing, we'd prefer to see the exact values.

(3) The date in the output will cause spurious differences when expected output is compared to actual output.

Revising the routine to solve these is simple. An example of testing output is:

```
aa:: (at bucket 194, bucket pos 0, duplicate pos 0)
#  (update_status = 1) (allow_dups = 1)
aa: (at bucket 194, bucket pos 0, duplicate pos 1)
#  (update_status = 2) (allow_dups = 0)
aa: (at bucket 194, bucket pos 0, duplicate pos 2)
#  (update_status = 3) (allow_dups = 0)
# 2 files in 1007 hash buckets.
# average 0.2 files per bucket, max 1 files in one bucket.
```

The hash table routines are now testable at very little expense.

Surprisingly, some organizations discourage the inclusion of debugging code in the finished product. The code may then be unavailable for testing; at best, it will be less useful (because it's "throwaway" code in the first place). Eliminate this policy.

The same organizations probably also resist "cluttering up the code with testing crud". You often don't want testing code in the executable, usually for performance reasons. (However, if the testing code is modified debugging code, that inclusion may save a lot of time when answering phone calls from the field.) But there is rarely a reason not to include testing code in the source and compile it out in the delivered version. There are two arguments you may face:

(1) "We should test what we ship". This is absolutely true - for the last two runs of the test suite. During earlier development, there is very little risk in using a testing version. The small risk of the testing code obscuring a bug, causing it to be discovered late, is counterbalanced by the other bugs that will be found because more tests are feasible.

(2) "Too much testing code makes the system unmodifiable." If your testing modifications cause every third line to be

#ifdef TESTING

> you should certainly rethink them. Testing changes should be as modular as any other enhancement. But what this statement can also mean is that modifications that had to happen anyway are now visible. Whenever there is test support code, changes to the system can wreak havoc with it, causing wholesale reworking. But that havoc is usually invisible to the person making the change. When the test support code is inside the system, it causes more work for that person. That the *total* work required to effect a change is reduced will be no consolation. This is a situation requiring considerable tact. The greater the organizational separation between testing and development, the greater the tact required.

As converting debugging to testing code becomes more common, make it simpler by providing conventions, support code, and examples. Keep in mind that the purpose is not to make the debugging code harder to write, but to make it more adaptable for testing. As low-cost efforts prove their worth, move toward a more complete in-system debugger and test harness. I'll discuss an example.

I was the leader of a team of programmers porting a version of Common Lisp from an obscure workstation to an obscure minicomputer. The workstation was of the now-unfashionable type that allowed user-programmable microcode. That is, it did not exactly have a "native" instruction set; instead, each high-level language compiler could assume a machine language tailored to its requirements. The Common Lisp compiler assumed an instruction set much like that of the MIT Lisp Machines. For example, the Lisp Cons instruction was also a machine language instruction, implemented in microcode. The garbage collector was also implemented in microcode.

The first step of the port was to emulate all the microcoded instructions by C and assembly code on the minicomputer, turning it into a Lisp machine emulator. Garbage collection was the hardest instruction; I began with it. I implemented debugging code that allowed me to initialize the state of the imaginary Lisp machine, take a snapshot, garbage collect, take another snapshot, and compare the two states. For testing, all that was required was to allow the garbage collector debugger to be driven by a script. The remaining instruction emulations were debugged and tested using extensions of what I'd already done for garbage collection, mainly by adding more flexibility in setting up the starting state. As the system grew closer to completion, more debugging features were added as needed. (In particular, after every difficult bug fix, I asked myself what new debugging command would have made the problem more obvious. Then I added that command.) Because this increasingly capable harness was designed to be script-driven, each new debugging feature was also a new testing feature, if appropriate.

Of course, in other systems, such an elaborate test harness is not required. Almost all of GCT's tests use special testing code, but most of them depend only on one 10-line segment. Some of the others use relatively simple subsystem-specific routines that print out a portion of GCT's internal state. In a few cases, parts of GCT have been unit tested with stubs and drivers.

At some point, a gradual approach may be insufficient; eventually, you may find that you need an overhaul of the system to test as thoroughly as you like. But the costs are modest, the benefits immediate, and success at small changes can give you the experience you'll need to justify a more designed, less "grown", test harness.

4. Appendix: Excerpts from a Test Condition Catalog

To help you develop your own catalog, this appendix shows several more examples from [Marick92a][6]. When using a catalog, entries may directly match attributes of the program. Often, though, the match is indirect and takes some creativity to discover. These indirect matches lead to exclamations like "Hey! I can think about this file as a collection and this operation as a sort of search!" or "This test condition for trees has an analogue for my unique data structure!" The odder indirect matches can lead to some of the most productive test conditions.

There are two types of annotations that need to be explained.

INLINE

Suppose your program searches. It may either use a library routine, such as:

```
match = search(target, list);
```

or an inline implementation, such as:

```
for(rover = list; rover != NULL_NODE; rover=rover->next)
{
 if (rover->tag == target->tag)
        break;
}
```

Some test conditions in the catalog test the program's use of a cliche (such as "match not found"), whereas others test the implementation of the cliche (such as "match in first position"). Assuming the library routine has been well tested, the latter test conditions aren't useful. They're marked as INLINE so that you can ignore them if you know the implementation uses a library routine.

IN/OUT

Data type conditions are labelled as IN, OUT, or IN/OUT. An IN condition is good for arguments to the subsystem (or intermediate, internal variables constructed from those arguments). An OUT condition is a good value to give to the result of the subsystem (or an intermediate variable used to calculate the result).

For example, when two input values are pointers of the same type, an IN condition says to try a test case where they point to the same object. Programs often contain bugs because they assume that that doesn't happen. But suppose a subsystem has two result variables which are pointers -- it's much less likely that forcing them to point to the same object will discover a bug.

5. COUNTS

number of iterations, number of data items, etc.
Note that Counts are a special case of Intervals.

- -1 IN
- 0 IN/OUT
- 1 IN
- > 1 IN/OUT
- maximum possible value IN/OUT
- one larger than maximum possible value IN

6. GENERAL COLLECTIONS

Note: Spotting collections requires creativity. For example, the formal parameters to a C macro definition are a collection -- and an old bug in the GNU C preprocessor could be found using these rules.

---

[6] Portions of [Marick92a] are Copyright © 1991 by Motorola, Inc. Under an agreement between Motorola, Inc., and Brian Marick, Brian Marick is granted rights to these portions.

Note also that you might apply individual element rules to all the elements at once. For example, if an array is used in arithmetic, you would want all elements 0 in one test.

- empty IN/OUT
- a single element IN/OUT
- more than one element IN/OUT
- full IN/OUT
- duplicate elements IN/OUT
  (may be an error case, or impossible)

6.1. General Operations on Elements of Ordered Collections

Whenever you're doing something to particular elements of an ordered collection (one with a first and last element), use the following conditions:

- Doing it to the first element INLINE
  (There should be more than one)
- Doing it to the last element INLINE
  (There should be more than one)

6.2. Making subsets of a collection

Also generate test conditions from both the starting and ending collection (even if the ending collection is "virtual" - never actually present as a distinct variable in the code, but created by ignoring elements of the original collection).
In all cases, the starting collection should have more than one element.

- Removing no elements.
- Removing one element
- Removing all but one element
- Removing all elements
- Removing first element (if known) INLINE
- Removing last element (if known) INLINE

7. CONTAINERS

Containers are objects that can contain variable amounts of data. For example, a buffer of characters, a C-style string, etc. Containers are also Collections.

7.1. Appending to a container's contents

This includes the case where several elements are being added at once. Some are not applicable when adding elements one at a time.

- Container initially empty.
- Container initially not empty.
- Adding just as many elements as will fit.
- Full container, attempt to add one element.
- Not-full container, add one too many elements.
- Add just enough to leave room for exactly one element INLINE
  (often catches unchecked "small, harmless" additions)
- Add zero new elements.

8. CONSISTENCY RELATIONSHIPS

Two (or more) variables may stand in a particular relationship, where one of them may have a particular value only when the other has a particular value.

The variables are called V1, V2, V3...

8.1. Establishing the relationship

If the relationship is built up in a series of computational steps, all but the last temporarily violate the relationship.

- V1 is the last variable to be assigned a consistent value
- V2 is the last variable to be assigned a consistent value
- ...

8.2. Maintaining the relationship

- V1 gets a new, but consistent, value.
- V2 gets a new, but consistent, value.
- ...

- V1 gets a new and inconsistent value. (Expect ERROR)
- V2 gets a new and inconsistent value. (Expect ERROR)
- ...

9. STRINGS

This is included because it's easy to forget and it's very important.

- The empty string. IN/OUT

10. FILES

10.1. Opening a file for writing

- file does not exist but can be created
- file does not exist, can't be created
- file exists, but not writable.
- file exists and is writable

11. CONTINUED SEARCHING

Here, searching can be resumed from the last element found.

- Further match immediately adjacent to last match

12. GENERAL OPERATIONS ON PAIRS OF COLLECTIONS

- V1 and V2 empty IN
- V1 has 1 element, V2 none IN
- V2 has 1 element, V1 none IN
- Both V1 and V2 have N (>= 1) elements IN

REFERENCES

[Basili84] V. Basili and D. Weiss, "A Methodology for collecting valid software engineering data", *IEEE Transactions on Software Engineering*, vol. SE-10, pp. 728-738, November, 1984.

[Basili87] V. Basili and R.W. Selby. "Comparing the Effectiveness of Software Testing Strategies". *IEEE Transactions on Software Engineering*, vol. SE-13, No. 12, pp. 1278-1296, December, 1987.

[Beizer83] Boris Beizer. *Software Testing Techniques*. New York: Van Nostrand Reinhold, 1983.

[Glass81] Robert L. Glass, "Persistent Software Errors", *Transactions on Software Engineering*, vol. SE-7, No. 2, pp. 162-168, March, 1981.

[Hamlet92] Richard Hamlet. "Are We Testing for True Reliability?". *IEEE Software*, July, 1992.

[Holthouse79] Mark A. Holthouse and Mark J. Hatch. "Experience with Automated Testing Analysis". *Computer*, Vol. 12, No. 8, pp. 33-36, August, 1979.

[Howden78] W. E. Howden. 'An Evaluation of the Effectiveness of Symbolic Testing'. *Software - Practice and Experience*, vol. 8, no. 4, pp. 381-398, July-August, 1978.

[Howden82] W. E. Howden. "Weak Mutation Testing and Completeness of Test Sets". *IEEE Transactions on Software Engineering*, vol. SE-8, No. 4, pp. 371-379, July, 1982.

[Howden87] W.E. Howden. *Functional Program Testing and Analysis*. New York: McGraw-Hill, 1987.

[Johnson83] W.L Johnson, E. Soloway, B. Cutler, and S.W. Draper. *Bug Catalogue: I.* Yale University Technical Report, October, 1983.

[Knuth89] Donald Knuth. "The Errors of TeX". *Software - Practice and Experience*. Vol. 19, No. 7, July 1989, pp. 607-686.

[Marick91] Brian Marick. "Experience with the Cost of Test Suite Coverage Measures". *Pacific Northwest Software Quality Conference*, October, 1991.

[Marick92a] Brian Marick. *Test Condition Catalog*. Testing Foundations, 1992.

[Marick92b] Brian Marick. *A Question Catalog for Code Inspections*. Testing Foundations, 1992.

[Mays90] R.G. Mays, C.L Jones, G.J. Holloway, and D.P. Studinski, "Experiences with Defect Prevention", IBM Sys. J., Vol 29, No 1, 1990.

[Myers78] Glenford J. Myers. "A Controlled Experiment in Program Testing and Code Walkthroughs/Inspections". *Communications of the ACM*, Vol. 21, No. 9, pp. 760-768, September, 1978.

[Morell90] L.J. Morell. "A Theory of Fault-Based Testing". *IEEE Transactions on Software Engineering*, Vol. SE-16, No. 8, August 1990, pp. 844-857.

[Myers79] Glenford J. Myers. *The Art of Software Testing*. New York: John Wiley and Sons, 1979.

[Ostrand84] Thomas J. Ostrand and Elaine J. Weyuker, "Collecting and Categorizing Software Error Data in an Industrial Environment", *Journal of Systems and Software*, Vol. 4, 1984, pp. 289-300.

[Perry89] Dewayne E. Perry. "The Inscape Environment". *Proceedings of the 11th International Conference on Software Engineering*, pp. 2-12, IEEE Press, 1989.

[Petschenik85] Nathan H. Petschenik, "Practical Priorities in System Testing". *IEEE Software*, September, 1985.

[Spohrer85] J.C. Spohrer, E. Pope, M. Lipman, W. Scak, S. Freiman, D. Littman, L. Johnson, E. Soloway. *Bug Catalogue: II, III, IV.* Yale University Technical Report YALEU/CSD/RR#386, May 1985.

[Voas91a] Jeffrey Voas, Larry Morell, and Keith Miller. "Predicting Where Faults Can Hide from Testing". *IEEE Software*, March 1991.

[Voas91b] Jeffrey Voas. "Factors That Affect Software Testability". *Pacific Northwest Software Quality Conference*, October, 1991.

A SUMMARY OF SUBSYSTEM TESTING

Brian Marick
Testing Foundations

1. The Context

First, a joke:

A mathematician, a physicist, and an engineer are told, "All odd numbers are prime."

The mathematician says, "That's silly. Nine is a non-prime odd number."

The physicist says, "Let's see. Three is a prime, five is a prime, seven is a prime -- looks like it's true."

The engineer says, "Let's see. Three is a prime, five is a prime, seven is a prime, nine is a prime, eleven is a prime..."

Testing is like the joke. A tester is given a false statement ("the system works") and has the job of selecting, from an infinite number of possibilities, an input that contradicts the statement ("Then why does it crash when you type in a line longer than 72 characters?"). You have to avoid both the physicist's error (not trying the right value) and the engineer's (trying the right value, but not noticing the contradiction). You want to be like the mathematician, finding the right counterexample with a minimum of wasted effort. That's the topic of subsystem testing.[1]

Some unavoidable terminology

The term "bug" is not very precise. Sometimes the imprecision doesn't matter, but it's often useful to distinguish three different senses of the word.

error
: An *error* is a mistake made by a person. It might be a typographical error, a misreading of a specification, a misunderstanding of what a subroutine does, and so on. Errors are mostly located in people's heads.

fault
: An error may lead to one or more *faults*. Faults are located in the text of the program. More precisely, a fault is the difference between the incorrect program and a correct version. For example, we may describe a fault as "the program used < when it should have used <=" or "the program is missing the code that checks for an error return from the write subroutine".

failure
: The execution of the faulty code may lead to zero or more *failures*, where a failure is the difference between the results of the incorrect and correct program. If the write routine never returns an error indication, the faulty program will never fail. A particular fault may cause different failures, depending on how it's been exercised.

Failures are detected by comparing the actual output of the program to the expected output. In almost all cases, the expected output is gotten from the *specification*. For purposes of subsystem testing, the specification is any external, independent description of the program, including especially user documentation. Specifications are often incomplete, incorrect, ambiguous, or contradictory, so it may be the specification at fault, not the program. Finding specification bugs is another part of the testing task.

Copyright © 1994 by Brian Marick.

[1] The metaphor of software testing as searching for a counterexample to a conjecture is due to [Podgurski92].

Some helpful terminology

This section introduces some terminology from the theory of testing. You can use subsystem testing without reading this section, or any of the sections that use this terminology. But the theory is not complicated, and understanding it will make you a better tester.

A goal of testing is to execute the program with inputs that cause all faults to reveal themselves as failures. What does this require? Consider this program:

```
void
check_square(int do_square, double x)
{
   double y = x;
   if (do_square)
      y = x + x;              /* Should be x * x */
   printf("squared version larger? %d\n", y > x);
}
```

The fault is that + was used when * should have been.

reachability

To provoke a failure, the program's inputs must cause the faulty statement to be executed. Those inputs satisfy a *reachability condition.* In this case, the reachability condition is that do_square is true.

necessity

Next, the faulty statement must produce a different result than the correct statement. This is called the *necessity condition.* It describes the inputs that cause an incorrect internal state. In this example, X=1 will cause Y to have the incorrect value 2 (1+1), instead of the correct value 1 (1*1). However, X=2 will cause Y to have the correct value 4. The program calculated the right result the wrong way, so it won't fail on that input. The necessity condition for this fault is X!=2 && X!=0.

propagation

Finally, the incorrect internal state must propagate so that it becomes visible in the program's results. For example, if X is 4, Y will be 8, where it should have been 16. However, both of these are greater than 4, so the faulty program will print

```
squared version larger? 1
```

which is exactly what the correct program would have printed. The incorrect internal state has been "damped out" before it became visible. The *propagation condition* for the fault requires that x+x>x have a different truth value from x*x>x. That's true whenever x<=1.

These three conditions are called the *ideal fault conditions.* Their names are taken from [Offutt88] and [Morell83]. All three of them must be simultaneously satisfied for the bug to be revealed. That is, you must test this program with inputs satisfying

```
do_square true AND (X != 0 and x != 2) AND (X <= 1)
```

or

```
do_square true AND X <= 1 (except 0)
```

Only then will the incorrect program show a different result from the correct program.

Of course, the correct program is not available. That means you can't run it to check for a failure. That problem is solved by using the program's specification. Not having the correct program also means there's no way to discover the ideal fault conditions. Subsystem testing is about practical ways of approximating that ideal. The rest of this chapter will first summarize subsystem testing, then show its relationship to theory.

Fundamental Assumptions

Subsystem testing depends on these assumptions, which are described in more detail in this chapter.

(1) Most mistakes are not very creative. Since mistakes are cliched, methodical checklist-based approaches will have a high payoff.

(2) Faults of omission are the most important and most difficult type.

(3) Specification faults, especially omissions, are more dangerous than code faults. They're also harder to find.

(4) At every stage of testing, mistakes are inevitable. Later stages should either detect them or silently compensate for them.

(5) How thoroughly the tests exercise the code (code coverage) is a good *approximate* measure of their quality. Because it's an approximation, coverage must be used with extreme care.

Motivating the Technique

Tests can be derived from both the specification and the program. The derivation is done, in principle, by predicting likely programmer errors or likely program faults, then writing test cases to detect them. In practice, the prediction is less important than the test case. As long as the bug is found, the tester doesn't care what the underlying problem is. Therefore, testers use rules abstracted from common errors. One simple abstract rule is "always test boundary conditions". If the specification requires that argument A must be greater than zero, the tester would try the value zero. (We'll ignore the other side of the boundary for this example.)

What likely error justifies that choice? Programmers can't get their relational operators right, so they're forever writing

```
if (A <= 0) exit_with_error_message();        /* buggy */
```

when they should be writing

```
if (A < 0) exit_with_error_message();         /* correct */
```

Zero is the most likely value to detect this fault, because it's the only value that makes the buggy program take the true branch when the correct program would take the false branch. (A=0 is the necessity condition for this fault.)

You can do a lot of good testing armed with a few simple rules, the three most basic being:

(1) If the specification identifies a case to handle, there's a reasonable chance the programmer either forgot to handle it or handled it incorrectly. So you should try that case.

(2) If the code attempts to handle a particular case, there's a reasonable chance it fails. So you should try that case.

(3) For each case identified above, preferentially test the boundary values.

(In the testing literature, these rules are called "equivalence class partitioning with boundary values".)

The rules work well, but require either the program or the specification to identify all the relevant cases. They don't, not always. The specification writer may have missed a special case; if so, the programmer almost certainly did as well. Or the special case may be an implementation detail of the sort that isn't relevant to the specification. If the programmer missed it, the specification also won't identify it.

Missed special cases are called *faults of omission*. This is an example of a low-level fault of omission:

```
write(file_descriptor, buffer, amount);
```

This code does not check whether the write fails. The code should read

```
if (write(file_descriptor, buffer, amount) == -1)
  handle_the_problem();
```

As an example of a higher level fault of omission, a specification for the client side of a client-server application might describe the recovery procedure when the server crashes, but fail to anticipate the case that the server might resume transmissions before recovery is complete.

Faults of omission are critically important. For example, [Glass81] found them to be the most common type of fault in delivered avionics systems. And most maintainers will recognize the sinking feeling that comes when a customer calls with some completely unanticipated - but reasonable - special case, one that will require a major design change. Far better to find those bugs before the product is released.

How to do this? Let's first consider coding and low-level design omissions.

Every system is a mixture of the unique and the mundane. A compiler may have new and novel register allocation techniques. An operating system may have clever and complex virtual memory algorithms. But the cleverness is surrounded by ordinary routines doing ordinary things, and it's largely implemented using the same sort of code programmers write again and again in their careers. Test design builds on the ordinary, because ordinary code contains ordinary mistakes. Boundary-handling code and the associated off-by-one errors are the simplest example, one that every tester knows.

Much of the difference between the skilled tester and the novice tester lies in knowing how to test for other ordinary mistakes. The skilled tester can look at a program and rattle off sentences like these:

(1) "This routine takes a pointer argument. Such routines often fail to handle null pointers."

(2) "This command-line option makes the program search for a matching file. Code that searches sometimes breaks when the match isn't found. If there is a match, it's best to put it in the last position, or the first."

(3) "This command line option takes an argument. Programs often fail drastically when no argument is given and the option is the last word on the command line."

Experienced testers have a catalog of programming cliches and associated errors in their head. The Test Condition Catalog, included with this paper, lists many of these for the benefit of inexperienced testers and experienced testers without perfect memories. The catalog is effective at finding low-level omissions, because it was developed with those in mind.

Higher level specification omissions are much more difficult. To find them, you must understand the unique aspects of the system and match those against your understanding of the system's unique environment. It's a creative (or creatively destructive) task, and a general purpose testing technique can't provide that creativity.

What it can provide is room for creativity:

- by providing an orderly framework for test design, so that you don't get bogged down by inessentials.
- by getting the mundane tests out of the way efficiently, leaving you with time and brainpower to design the unique tests.
- by helping you avoid test design ruts. Every specification has an implicit "preferred reading" - its presentation encourages you to think about the problem the same way the specifiers did. A good test design technique will force you to think about the specification in a different way, by encouraging the combining of disparate specification elements. This often leads to "Aha!" moments, where you realize an incongruity between page 13 and page 43.

Generally - but not always - you discover specification problems during test design, well before the tests are implemented. In that case, you may not implement the test until the problem is fixed. You must still run the test, though, to check whether the fix was correctly implemented.

2. Test Design

The first requirement of test design is that it be methodical. Many programming errors are due to lapses of attention; some special case slips the programmer's mind. The cases that are easy for a programmer to overlook are just as easy for the tester, and a tester who does not proceed by careful, thorough steps will miss bugs in a flood of detail.

Subsystem testing divides test design into three stages: finding clues, expanding them into test conditions, and combining test conditions into test cases.

Three Elements in Test Design

test cases
: Test cases describe exact inputs to a program, together with exact expected outputs.

  LIST has the null string as its single element; expect return value 5.

test conditions
: Test conditions describe useful sets of input that should be tested.

  LIST must have a single element (value unspecified).
  Use the null string as some LIST element.

clues
: Clues are the sources for test conditions.

  LIST, a list of strings
  a list element, a string

Clues

The first stage in test design is searching for clues about what needs testing. These clues can be found in the program itself, in the specification, in bug reports filed against earlier versions of the system, from experience with similar systems, and so on. This paper will concentrate on the first two sources.

These clues are written down in a checklist. By listing the clues, you focus your attention and reduce the chance that you'll omit a needed test. Here are some representative clues:

> searching for an element equal to ELT in the input LIST.
> ELT < 0 is an error.
> D_ELT is a count.

That the program searches is a clue to write tests for likely searching errors. That the program classifies ELT is a clue to write tests checking for incorrect classification and incorrect error handling. That D_ELT is a count is a clue to write tests targeted to common errors involving counts.

Test Conditions

The clues describe what needs to be tested. The next step is to decide how to test them in order to discover likely errors. Off-by-one errors are always likely. In a search, such an error might cause the search to stop one element early. To detect the error, a matching element should be placed in the last position. Another likely error is to assume that a match will always be found. This leads to omitted code, and to catastrophic results when the search returns a "no match" indication. To detect the error, there should be no matching element in the LIST. Both of these descriptions of how to test - called *test conditions* - are useful enough often enough to be written down in the Test Condition Catalog.

Here are those test conditions for searching, together with some for the other clues:

searching for an element equal to ELT in the input LIST
    found in the last position.
    not found
ELT < 0 is an error.
    ELT = -1
D_ELT is a count.
    D_ELT = 0
    D_ELT = 1

Although test conditions are derived from likely errors or faults, the justification is not part of the test condition. A test condition is just a description of some input that's good at finding bugs.

Test Cases

Test conditions are general descriptions of how to test a program. The first searching test condition does not give precise values for ELT or LIST; it just says that the only match for ELT must be the last element of LIST. Further, it says nothing at all about the value of D_ELT. To execute the program, these variables must be given exact values in a *test case*. Here's an example:

TEST 1:
    ELT is 1
    LIST is [45, 5, 1]
    D_ELT = 1
EXPECT
    return value 2

Notice that the test case describes exact expected output as well as exact inputs. That's very important for effective testing.

A test case is created by picking inputs that satisfy one or more test conditions. The above test case satisfies two, one for searching and one for D_ELT:

found in last position
D_ELT = 1

In subsystem testing, test cases usually satisfy as many test conditions as possible. This leads to complex tests and test suites with more variety. Complexity and variety increase the chance of finding bugs you didn't think of looking for. Because we're all imperfect testers, we have to enlist chance on our side.

Although this test case satisfies test conditions, it does so poorly. Suppose the program uses D_ELT when it should use ELT, a not-unusual fault. Because this test gives D_ELT the same value as ELT, the program will produce the correct results the wrong way -- and the bug will be missed. You should satisfy the searching condition with some other value for ELT. Test case creation requires both a procedure for combining test conditions and rules to reduce the chance that a test will miss a failure.

3. Supplementary Code Reads

There are some types of faults that testing is poor at detecting. For example, suppose a program very occasionally fails to close an open file. A test that exercises that fault once would not cause a visible failure. Finding the fault by testing would require exercising it many times (until the program runs out of resources), at a much greater expense in time or hardware. It is better to augment test design with a focused kind of code reading. This reading is organized around a checklist of likely faults, the Question Catalog for Code Inspections. The catalog can be used to supplement the style of code reading or inspecting you already use. (The tradeoffs between testing and code reading are outside the scope of this paper.)

A Summary of Subsystem Testing

Build the test condition checklist
    Find clues
    Expand clues into test conditions
Design the tests
    Combine test conditions into test cases
    Check the test cases for common testing mistakes
Supplement testing with focused code reads
Implement test support code
Implement the tests
Evaluate and improve the tests
    Measure code coverage
    Use results to find undertested or missing clues
    Find more test conditions for those clues
    Write more test cases

4. Test Implementation and the Scale of Testing

The implemented tests should cost as little as possible over the life of the system. Reducing lifetime cost requires automation. Manual or semi-automated tests are cheaper to develop, but they're too time-consuming and hence expensive to run often. For this reason, it's not unusual to run a manual suite only once, just before a new release, with the desperate hope that no bugs are found. Manual testing is also error-prone; people easily miss obvious failures, especially late Friday afternoon after a solid week of running tests.

Automated tests usually require special-purpose test support code and tools. Reducing the cost of testing requires reducing the amount of that support code. This raises an issue: what amount of code should you consider when designing tests?

You could derive clues and test conditions from a single routine, combine them into test cases for that routine, then write test support code that allows you to test the routine in isolation. This approach, called *unit testing*, is too expensive. Not only is the initial development of the support code costly, maintenance becomes a nightmare. The problem is that relatively small changes to the system can require changes to the support code for many units. The result is that unit tests are often abandoned -- in which case, they're barely worth writing.

It is more efficient to test larger subsystems, because less support code needs to be written. In the most thorough application of this technique, you select a subsystem, derive clues and test conditions from all the internal routines, from the interfaces between the routines, and from the entire external interface, then combine them all into test cases. Based on those test cases, you design support code to allow you to test the entire subsystem in isolation. This code is substantially cheaper than writing support code for each routine. Alternately, you can adapt the system to provide the testing support for its subsystems.

Testing larger subsystems also has advantages in effectiveness. Because each routine participates in tests targeted to other routines, every routine is exercised more. Because test design emphasizes variety, all routines are exercised in many ways, including many unplanned ways. More bugs will be found by chance.

As subsystems increase in size, problems surface. It becomes harder to satisfy some test conditions, the amount of information to be managed increases the difficulty of test design, and the chance that a failure may escape detection grows. The approach of this technique is to test large subsystems and solve those problems, rather than keep subsystems small enough that they do not arise.

5. Testing the Tests

A good set of tests will thoroughly exercise a subsystem. The tests will force all branches in both directions, force all subparts of branch tests to be both true and false, force all loops to execute zero, one, and more than one times, and force certain boundary conditions.

During test design, you should pay no attention to these criteria. You should not write test conditions to force particular branches in particular directions. The test conditions gotten from other sources will usually do that anyway, so considering branches would be wasted thought.

After the tests are run, the criteria should be measured. This is called *test coverage*. Complete subsystem testing (not scaled down because of schedule pressure) will usually result in high coverage; for example, 95% of the branches might be taken in both directions.

The remaining branches might be considered test conditions to satisfy, but it is better to treat them as clues about weaknesses in the test design. For example, suppose this code is under test:

```
1     operation_failed = operation();
2     if (operation_failed)
3       switch (type_of_failure)
4         {
5         case FAILURE_TYPE_1: ...
6         case FAILURE_TYPE_2: ...
7         case FAILURE_TYPE_3: ...
8         }
```

A coverage tool might complain that the cases for failure types 1 and 3 had never been taken, with a report like

```
line 5: case was taken 0 times.
line 7: case was taken 0 times.
```

The strong temptation is to write tests to cause those two failures and be done with it. However, what this coverage really indicates is that the tests only weakly probe operation's failure cases. Rethinking that clue might reveal that operation can actually return a fourth failure type, one that the caller makes no effort to handle - a probable omission fault.

Thinking about what coverage means is more difficult than simply finding ways to satisfy it. However, after thorough initial test design, adding the tests that coverage suggests takes only a few percent of the total effort.

Coverage reveals test implementation mistakes as well as test design mistakes. For example, you may have thought you designed a test that passed a particular value to some internal routine, but in fact a higher-level routine intercepted that special case and handled it. The internal routine has not been correctly tested. Coverage is important in all testing; it is crucial for effective subsystem testing.

When coverage shows nothing more to do, stop testing. (When subsystem testing has been scaled down because of schedule pressure, perfect coverage is not a realistic goal, but coverage can still be used to guide test suite improvement.)

6. Summary: How Subsystem Testing Approximates the Theoretical Ideal

Subsystem testing's test conditions are most closely related to necessity conditions. Recall that necessity conditions describe what inputs are needed to cause an incorrect internal state. For a given program, there are three sets of necessity conditions: those for the faults actually present in the program, a much larger set for the faults plausibly present in the program, and an infinite set for the faults possibly present in the program. (Nit: an actual fault may not have been plausible. That is, you wouldn't have ever guessed the program could be wrong *that* way. Let's leave that topic aside for the moment.)

Test conditions should approximate the necessity conditions for the plausible faults, in the sense that tests satisfying the test conditions should also also satisfy the plausible necessity conditions. The test conditions are not identical to those necessity conditions because they're derived from different sources. They may be

derived from plausible errors - by asking what inputs would find most faults caused by this error, regardless of their actual manifestation. Or test conditions may be purely empirical - found by observing which inputs are good at provoking failures, regardless of either the underlying fault or error. If testing with the empty string catches bugs often enough cheaply enough, is it essential that we know why?

A second hope for the set of test conditions is that it be smaller than the set of plausible necessity conditions. On average, each test condition should satisfy several necessity conditions, reducing the cost of test design. This is accomplished by deriving them from "higher level" entities, such as specification and programming cliches. For example, the test conditions for a searching loop may detect several types of off-by-one errors, without having to consider each of these plausible faults separately. Or the empty string may exercise plausible faults in many locations, without having to consider each location individually.

But even if test conditions are perfect (have the effectiveness of plausible necessity conditions, while being few enough to allow affordable test design), necessity conditions are only one of the three ideal fault conditions. The other two are reachability, causing the fault to be exercised, and propagation, causing the incorrect internal state to affect the program's result. In subsystem testing, these two conditions are handled in three ways:

design
: Some of the test case design rules alert you to tests where reachability or propagation are doubtful.

implementation
: In some cases, you should modify the subsystem to permit reachability and propagation. For example, if the results of an internal function return a complicated list structure, but the entire system returns only TRUE or FALSE, there's a potential propagation problem. An incorrect list structure may yield the same eventual boolean value as the correct list structure. In this case, you should add test support code that prints out the list structure. Essentially, you've selectively made part of the intermediate state part of the subsystem's results. The propagation condition is much more likely to be satisfied.

coverage
: Coverage is good - though not infallible - at detecting failures to satisfy reachability conditions. If reachability isn't satisfied, some code may not be exercised. Examining coverage often leads to realizing that the test and program are such that a necessity condition doesn't reach a body of code where it would be useful.

Given these, reachability and propagation are not usually major problems in practice. But the set of test conditions may be inadequate in one of two ways:

overly general
: A test condition might not force a necessity condition. For example, a fault's necessity condition might be `X != 0 && X != 2`, while a test condition might be `X > 0`. If the test case, by sheer bad luck, selected `X=2`, the fault would not be detected. Another possibility is that you have the following test conditions:

    A true
    A false
    B true
    B false

  The fault's necessity condition is that A is true and B is true. Subsystem testing contains rules for discovering when test conditions interact and should be combined into compound test conditions like A is true and B is true. But when those rules fail, test case design does not require that all combinations of values for A and B be used. (It would be too expensive.) There is no guarantee that any test case exercises the necessity condition.

too small
: Because test conditions are not derived from actual faults, some test conditions might simply be missing. You might never notice an important clue.

These problems would be solved by enhancing the existing test conditions or discovering missing ones. Coverage often helps, but only when the inadequate test conditions correspond to unexercised code. Often,

though, they correspond to omitted code, and coverage says nothing about omitted code.

Test design that emphasizes variety and complexity helps counteract weak test conditions.

- Each test condition is used often, and used in different ways. Because of this, the test suite is more likely to satisfy X != 0 && X != 2.
- The chance of accidentally making a needed combination of two test conditions is increased. Because the test conditions for A and B will be satisfied together in a variety of tests, one of those tests is more likely to satisfy the compound test condition A true and B true.
- Because the tests are complicated, the chance of accidentally satisfying completely unanticipated necessity conditions increases. Those test conditions may be plausible ones missed during test design, or implausible ones that no cost-effective testing technique could reliably discover.

The test design rules attempt to strike a balance between too many tests and the likelihood of omitted necessity conditions. In all of testing, we must compare the unknown probability that a new test condition will find a bug to the better-known but still uncertain cost of that new test condition. There is too much uncertainty to give a deterministic rule. However, a basic premise of subsystem testing is that explicitly thinking about sources of both cost and effectiveness will lead to better decisions.

REFERENCES

[Glass81]
Robert L. Glass. "Persistent Software Errors". *Transactions on Software Engineering*, vol. SE-7, No. 2, pp. 162-168, March, 1981.

[Morell83]
Larry Morell. *A Theory of Error-Based Testing*. Ph.D. dissertation, University of Maryland, 1983.

[Offutt88]
A.J. Offutt. *Automatic Test Data Generation*. Ph.D. dissertation, Department of Information and Computer Science, Georgia Institute of Technology, 1988.

[Podgurski92]
Andy Podgurski. *The Role of Statistical Reliability Assessment*. 24th Symposium on the Interface: Statistics and Computing, College Station, Texas, March 1992.

TEST CONDITION CATALOG
STUDENT VERSION

Brian Marick
Testing Foundations
Revision 2.4

This version of the Test Condition Catalog has been pruned down so that people just learning subsystem testing will find it easier and quicker to use.

Copyright © 1994 by Brian Marick. Portions are Copyright © 1991 by Motorola, Inc. Under an agreement between Motorola, Inc., and Brian Marick, Brian Marick is granted rights to these portions.

CHAPTER 1

The Catalog

Some entries are marked "specification only". By this, I mean that the test conditions should be included only when they match an element in an external description of the program. For example, comparison operators used in the code shouldn't have test conditions written for them specially - they'll be well-tested via other test conditions (or, failing that, by coverage conditions).

PART 1A: SIMPLE DATA TYPES

1. BOOLEAN

• 1 (true)	IN, OUT
• 0 (false)	IN, OUT
• Some true value not 1	IN

2. NUMBERS

2.1. Comparison operators
Specification only
epsilon is 1 for integers.

<, >=
- V1 = V2 - epsilon
- V1 = V2

>, <=
- V1 = V2
- V1 = V2 + epsilon

==, !=
- V1 == V2
- V1 != V2

IF floating point numbers
- V1 = V2 + epsilon
- V1 = V2 - epsilon

3. COUNTS

number of iterations, number of data items, etc.
Note that Counts are a special case of Intervals.
Specification only.

• -1	IN
• 0	IN/OUT
• 1	IN
• > 1	IN/OUT
• maximum possible value	IN/OUT

• one larger than maximum possible value	IN

4. INTERVALS

Values selected from a range including its endpoints [A,B], or from a range that doesn't (A,B). Epsilon is 1 for intervals of integers.
Note: on computers, there's no such thing as an interval to infinity. Treat them as closed intervals ending in the largest or smallest possible numbers, or at least very large numbers.

[A, B]	
• V1 = A - epsilon	IN
• V1 = A	IN, OUT
• V1 = B	IN, OUT
• V1 = B + epsilon	IN
[A, B)	
• V1 = A - epsilon	IN
• V1 = A	IN, OUT
• V1 = B - epsilon	IN, OUT
• V1 = B	IN
(A, B]	
• V1 = A	IN
• V1 = A + epsilon	IN, OUT
• V1 = B	IN, OUT
• V1 = B + epsilon	IN
(A, B)	
• V1 = A	IN
• V1 = A + epsilon	IN, OUT
• V1 = B - epsilon	IN, OUT
• V1 = B	IN

4.1. Pairs Of Intervals

If there is more than one Interval-valued variable, consider adding these conditions:

- Let both V1 and V2 have the smallest possible values.
- Let both V1 and V2 have the largest possible values.

It doesn't matter if the variables describe the same interval or not. Restrict the pairs you list to variables likely to interact. Example:

page size, Interval [0,66]
number of pages, Interval [0, 1000]

If the amount of memory allocated is page size x number of pages, the two conditions would apply.

PART 1B: COLLECTIONS

5. GENERAL COLLECTIONS

Note: Spotting collections requires creativity. For example, the formal parameters to a C macro definition are a collection -- and an old bug in the GNU C preprocessor could be found using these rules.

Note also that you might apply individual element rules to all the elements at once. For example, if an array is used in arithmetic, you would want all elements 0 in one test.

- empty IN/OUT
- a single element IN/OUT
- more than one element IN/OUT
- full IN/OUT
- duplicate elements IN/OUT
  (may be an error case, or impossible)

5.1. General Operations on Elements of Ordered Collections

Whenever you're doing something to particular elements of an ordered collection (one with a first and last element), use the following conditions:

- Doing it to the first element INLINE
  (There should be more than one)
- Doing it to the last element INLINE
  (There should be more than one)

In most languages and programs, collections have an ordering, whether you know it or not. This ordering is often hidden from the specification. You may need to DEFER knowing which is the first element until you can look at the code.

5.2. Selecting elements from a collection

- No element selected
- One element selected
- More than one element selected

5.3. Removing single elements from a collection

- The collection has one element INLINE
  (If the collection is not an array.)
- The collection has no elements

5.4. Making subsets of a collection

Also generate test conditions from both the starting and ending collection (even if the ending collection is "virtual" - never actually present as a distinct variable in the code, but created by ignoring elements of the original collection).
In all cases, the starting collection should have more than one element.

- Removing no elements.
- Removing one element
- Removing all but one element
- Removing all elements
- Removing first element (if known) INLINE

- Removing last element (if known) INLINE

6. STREAMS (SEQUENTIALLY PROCESSED COLLECTIONS)

These are collections that are processed by taking each element in turn and doing something with it. Input from a file or a terminal or a network connection, for example. You can expect to see loops when you have streams. Particular operations on streams are covered later; this section covers special kinds of streams.

Notation: Streams are usually written as letters in parentheses, like (a b c). Identical letters mean that the elements are equal: (a b a).

When using streams, remember that streams are collections, so you should also look for test conditions there, and remember that several of the subcategories below may apply to any particular stream.

6.1. Streams that may contain repeated elements

These are streams in which repeated elements are expected to affect the processing. For example, if the elements are machines to open connections to, repeated elements might be ignored -- and that ignoring should be tested. On the other hand, if the elements of the stream are just integers to be added up, repeated elements don't make a difference and aren't worth testing.

Note that, in a lot of cases, these conditions may be combined. They're separated here for those cases where they can't be.

- No duplicates IN/OUT
  Examples: (a b c) (c a b d)
- All duplicates IN/OUT
  Examples: (a a) (b b b)
- An element is duplicated more than once IN/OUT
  Examples: (b b b) (a b a c a d)
- More than one element is duplicated IN/OUT
  Examples: (a a b b) (a b b a b c d c)
- An element is duplicated immediately IN/OUT
  Examples: (a a) (b a a c)
- An element is not duplicated immediately IN/OUT
  Examples: (a b a)

6.2. Streams with two kinds of values

This refers to either the actual elements (such as a stream of 1's and 0's) or the interpretation of the elements. For example, in a stream of machine names, the two values might be whether the machine was available or not available.

- Two or more elements, each with the first value IN/OUT
  Examples: (0 0) (0 0 0)
- Two or more elements, each with the second value IN/OUT
  Examples: (1 1) (1 1 1)
- Two or more elements, alternating IN/OUT
  Examples: (1 0) (0 1) (1 0 1) (0 1 0 1)

PART 1C: STATE AND RELATIONSHIPS

These conditions apply especially to global variables and arguments to functions.

7. INTERNAL STATE

Variables may contain internal state variables, typically fields in structures. Treat these just like cases, where the state may be any of (S1, S2, ..., Sn).

- V1 = each possible state. — IN, OUT
- V1 = some impossible state. — IN
  (If possible, choose boundaries, for example if states are represented by integers.)

8. EXTERNAL STATE

This section is most useful when using test conditions from an internal routine as part of the testing of a larger subsystem. That internal routine will be called ROUT.

External state may be imposed on a variable by its relationship to other variables. For example, a process on an operating system's run queue might be in either a "high priority", "normal priority" or "background" state depending on which run queue it's linked into, despite the fact that the process's priority isn't stored in the process structure. Or a message variable might be an error message or acknowledgement message, depending on its source.

- V1 = each possible state, and ROUT is executed on V1 — IN

The goal is to force the calling of the internal routine in many states, potentially from many locations. Most of the benefit is in finding problems in the caller.

9. CONSISTENCY RELATIONSHIPS

Two (or more) variables may stand in a particular relationship, where one of them may have a particular value only when the other has a particular value. In Part 1 of the book, these were called invariants.

The variables are called V1, V2, V3...

9.1. Establishing the relationship

If the relationship is built up in a series of computational steps, all but the last temporarily violate the relationship.

- V1 is the last variable to be assigned a consistent value
- V2 is the last variable to be assigned a consistent value
- ...

9.2. Maintaining the relationship

- V1 gets a new, but consistent, value.
- V2 gets a new, but consistent, value.
- ...

- V1 gets a new and inconsistent value. (Expect ERROR)
- V2 gets a new and inconsistent value. (Expect ERROR)
- ...

PART 1D: POINTERS

10. POINTERS

• null pointer	IN/OUT
• pointer to a true object	IN/OUT
• if several arguments are pointers of the same type, have them all point to the same object. (This catches bugs where the routine uses the argument when it should have made a copy.) Example: func(&object, &object)	IN

10.1. Pointer Equality Testing

A common bug is confusion about whether equality means equality of pointers or equality of what's pointed to.

• Two pointer arguments are equal	IN
• Pointers are not equal, but objects pointed to have identical contents	IN
• Neither pointers nor objects are equal	IN

PART 1E: DATA TYPES USING POINTERS

This section concerns itself with data types made up of structures linked together with pointers. Simple linked lists don't have anything special about them -- they're usually treated just like any other collection or sequentially processed collection, so you won't find anything about them here.

11. TREES

• The empty tree	IN, OUT
• A single-element tree	IN, OUT

11.1. Trees: Calculating their depth

The "first subtree" is whichever branch of the tree the traversal routine processes first

• The first subtree is not the deepest (because the code may only descend one branch)	INLINE
• The depth is not equal to the number of immediate children of root	INLINE

12. LINKED STRUCTURES

• The structure is circular	INLINE

The structure is circular if you can start at some node and follow some chain of pointers that brings you back to the starting node.

PART 1F: STRINGS AND TEXT

This part is concerned with text processing, both of small strings and larger groups of strings (text). Notice that strings are collections of characters, that they're often treated as streams of characters, and that programs use pointers to manipulate them. So lots of other catalog entries apply -- here are some special case ones.

13. STRINGS

This is included because it's easy to forget and it's very important.

- The empty string. IN, OUT

13.1. Comparing Strings

These are the same rules as for arrays, repeated here for convenience.

- V1 and V2 same length, differing in last element. INLINE
  Example: "abc" and "abd"
- V1 has same elements as V2, but one element shorter. INLINE
  Example: "ab" "abc"
- V2 has same elements as V1, but one element shorter. INLINE
  Example: "abc" "ab"

PART 1G: PATHNAMES, FILES, and OPEN FILE DESCRIPTORS

These are mostly UNIX-specific.

14. FILES

14.1. Checking whether a file exists

- file exists
- file does not exist

14.2. Opening a file for reading

- file does not exist
- file is readable
- file exists, but not readable
  (Note: if a setuid program, make it not readable by effective uid but readable by real [or vice-versa, if appropriate for the application])

PART 2A: SIMPLE OPERATIONS

15. EQUALITY COMPARISON

This applies to many data types, not just numbers. Some data types are indexed separately, because they have several false cases, or more refined false cases. See, for example, NUMBERS, STRINGS, and ARRAYS.

Specification only.

- Equal
- Not equal

16. OTHER COMPARISON OPERATORS

Some data types have a < or <= relationship defined for them, even though they're not numbers.

Specification only.

<, >=
- V1 < V2
- V1 = V2

>, <=
- V1 = V2
- V1 > V2

In the inequality cases, make the two variables "close together" whenever the ordering allows it.

For example, in string comparisons, "fooc" is both less than "food" and also nearly equal to it.

PART 2B: SEARCHING

17. GENERAL SEARCHING CONDITIONS

- Match not found
  If possible, a matching element should be placed just past the bounds of the search. If the boundaries are handled or specified incorrectly, this increases the chance of an observable failure.
- Match found (more than one element, exactly one match)
- More than one match in the collection
- Single match found in first position (it's not the only element) INLINE
- Single match found in last position (it's not the only element)

If the search is backward, the "last" element is actually the first in the list or array.

18. USING THE POSITION OF THE MATCHING ELEMENT

In general, we don't care where the element is found. In this case, the program uses the position. Essentially, what we do in this case is "undefer" the last two conditions of the general case.

- Single match found in first position (it's not the only element)
- Single match found in last position (it's not the only element)

18.1. CONTINUED SEARCHING

Here, searching can be resumed from the last element found.

- Further match immediately adjacent to last match

This is a good example of applying search conditions to text. For example, if searching for $-delimited keywords, one test case might be $key1$$key2$.

PART 2C: OPERATIONS ON PAIRS OF COLLECTIONS

19. GENERAL OPERATIONS

- V1 and V2 empty — IN
- V1 has 1 element, V2 none — IN
- V2 has 1 element, V1 none — IN
- Both V1 and V2 have N (>= 1) elements — IN

20. OPERATIONS ON EQUAL-SIZED COLLECTIONS

The collections are supposed to have the same number of elements.

- V1 has N (>= 1) elements, V2 N+1 — IN
- V2 has N (>= 1) elements, V1 N+1 — IN

21. ORDERED COLLECTIONS WITH MATCHES

Here, two collections are processed in order. Elements from one collection may match elements in the other. (The program may expect that all will match, or none, or some - these test conditions are useful regardless of the expectation; all that changes is whether they are error cases or not.)

In the test conditions below, variants of the word "match" indicate elements that are identical in the two collections; variants of "mismatch" indicate elements are not identical to their corresponding element in the other collection. You can freely make longer collections that satisfy several of the test conditions at once.

- V1 = (match another-match) V2 = (match another-match) — IN
- V1 = (match MISMATCH) V2 = (match mismatch) — IN
- V1 = (MISMATCH match) V2 = (mismatch match) — IN
- V1 = (MISMATCH ANOTHER-MISMATCH) V2 = (mismatch another-mismatch) — IN

These cases always apply to any longer collection. In the case where the collections are of different length, the "last element" is the last element of the shorter collection.

- First elements match — IN, INLINE
- First elements mismatch — IN, INLINE
- Last elements match — IN, INLINE
- Last elements mismatch — IN, INLINE

Any equality test conditions that apply to the elements (e.g., Comparing Strings) should be used. It is a good idea to refine both the "first elements" and "last elements" test conditions.

A QUESTION CATALOG FOR CODE INSPECTIONS

Brian Marick
Testing Foundations
809 Balboa, Champaign, IL 61820
(217) 351-7228
marick@testing.com
Revision 2.0

This document is a catalog to be used with *The Craft of Software Testing*, a text I use in on-site and correspondence courses.

For each procedure (or other chunk of code), scan the catalog, first checking whether the question applies anywhere, next whether the answer is yes. A yes answer means a probable bug. The questions are predominantly for bugs that dynamic testing is poor at discovering. Other bugs are better found via testing.

This catalog is to be kept *short*. A catalog with too many entries will not be used.

Copyright © 1992 by Brian Marick. Portions are Copyright © 1991 by Motorola, Inc. Under an agreement between Motorola, Inc., and Brian Marick, Brian Marick is granted rights to these portions.

You may copy or modify this document for personal use, provided you retain the original copyright notice and contact information. I would appreciate copies of your changes.

1. The Question Catalog

1.1. DECLARATIONS

1.1.1. LITERAL CONSTANTS

• If these are parameterized with #ifdefs, are any of the different values incorrect? For example:

```
#ifdef PDP10
#define BYTESIZE 7      /* Should be 6 */
#else
#define BYTESIZE 8
#endif
```

Such errors often slip past because all values are not tested (much less all combinations of values).

1.1.2. CHANGES TO EXISTING CODE

• If a new field is added to a declaration, have any initializing definitions not been changed? Consider

```
struct foo
{
 int i;
 int j;
 int m;
};

...
struct foo myfoo = { 1, 2, 3}
```

If the structure is changed to

```
struct foo
{
 int i;
 int k;
 int j;
 int m;
};
```

the insertion of K into the middle of the structure means that MYFOO's J will no longer be initialized to 2, but rather to 3.

• If a new field is added to a declaration, has any initializing code not been changed?

This is really the same as the above case, but worth noting specially. If, in the example above, instances of the structure are normally initialized with

```
s.i = 1; s.j = 2; s.m = 3;
```

all such initializations must be updated when k is added.

1.2. DATA ITEMS

1.2.1. STRINGS

• Can this string ever not be null-terminated?

1.2.2. BUFFERS

• Are there always size checks when copying into the buffer?

• Is this buffer ever too small to hold its contents?

For example, one program had no size checks when reading data into a 12-character buffer because the *correct* data would always fit. But when the file it read was accidentally overwritten with incorrect data, the program crashed mysteriously.

1.2.3. BITFIELDS

• Are there possible ordering problems (portability)?

1.3. INITIALIZATION

1.3.1. LOCAL VARIABLES

• Are local variables initialized before being used? (Common versions of 'lint' do *not* catch all cases of this -- only most of them.)

1.3.2. MISSING REINITIALIZATION

• Suppose the processing of a data element in a sequence causes a global to be set. For example, when a file is being read, some globals might be initialized for that file. Can that global be used for the next file in the sequence without being reinitialized?

1.4. MACROS

• If a macro formal parameter is evaluated twice, is the macro ever expanded with an actual parameter that has side effects?

For example, what happens in this code?

```
#define max(a,b) ((a)<(b)?(b):(a))
max(i++, j)
```

• If a macro is not completely surrounded by parentheses, is it ever invoked in a way that will cause unexpected results?

```
#define max(a,b) (a)<(b)?(b):(a)     /* Should be ((a)<(b)?(b):(a)) */
result = max(i, j)+3;
```

• If a macro's arguments are not surrounded by parentheses, will this ever cause unexpected results?

```
#define check(A)  (A && global)   /* Should be ((A) && (global))
result = check(i || j)
```

1.5. SIZING DATA

• Is the argument to sizeof an incorrect type?

A common error is using sizeof(p) instead of sizeof(*p).

1.6. ALLOCATING DATA

• Is too little (or too much) space allocated?

• Malloc and similar functions allocate uninitialized storage. Does the code assume it's been initialized to zero?

• When dynamically allocated structures or arrays are initialized, are any of the fields not set?

• Is allocated data never freed?

• If a function is called that returns static storage, is that function ever called again before the caller is finished using the first value?

```
give_storage(data1, data2)
{
 static char retval[SIZE];  /* Place return value in here. */
 /* Initialize contents of retval. */
 return retval;
}

 ...

storage1 = give_storage(upper1, upper2);
storage2 = give_storage(lower1, lower2); /* storage1 has just been overwritten. */
```

1.7. REALLOCATING DATA

- When allocating more space with realloc, is the additional space assumed to be initialized to zero?
- When allocating more space with realloc, are there pointers to the old copy of the data?

1.8. FREEING DATA

- Are you freeing already-freed storage?
- Does the freed storage still have pointers to it, pointers that could still be dereferenced? (Some versions of free overwrite some parts of the freed data; most do not.)
- Are you freeing storage that is not supposed to be freed? (Often, allocated with a routine that says "do not free returned value" in a code header.)
- Is it possible for the pointer being freed to be null?

(Note: POSIX-compliant implementations of free accept null pointers. Not all implementations conform.)

1.9. FILES

- Can a temporary file not be unique?

(This is a common design bug.)

1.10. OPEN FILES

- Is a file pointer reused without closing the file?

```
fid = open(...);
fid = open(...);
```

- Is a file not closed in case of an error return?

For example, a file is opened, processing continues, some error occurs, and the function returns without closing the file.

- Is a file never closed for any other reason?

1.11. COMPUTATION

• Is the parenthesization incorrect?

The following error is very common, of course:

```
        if (a = function() == 0)
should be
        if ((a = function()) == 0)
```

but there are more subtle cases:

```
        if (function(X,Y, (expression != test)))
instead of
        if (function(X,Y, expression) != test)


        malloc(strlen(name+1))
instead of
        strlen(strlen(name)+1)
```

• Sometimes, variables are used as caches for other values or must otherwise be updated in synchrony. If a value changes, are all synchronized variables updated?

1.12. CONDITIONALS

• If a test is an error-check, might the "error condition" actually be legitimate in some cases?

(People often are too quick to label possible return values as errors.)

• Are exact equality tests used on floating point numbers?

1.13. CASTING

• Are null pointers always cast to the correct type when passed as a function argument?

```
        execl(name, arg0, arg1, ..., argn, 0)
instead of:
        execl(name, arg0, arg1, ..., argn, (char *)0)
```

• Is an unsigned integer used when a signed one should be (or vice versa)?

1.14. POINTERS

• When dereferenced, can a pointer ever be null?

• Suppose this routine copies a pointer value. Should it instead have stored a copy of what that pointer points to?

For example, suppose ROUTINE1 takes a pointer as an argument and stores it in a global for later use by ROUTINE2. Then consider what happens in this situation:

```
routine1(&some_data);
some_data.count++;
routine2();        /* Will use updated value of some_data.count */
```

1.15. ASSIGNMENT

• Do the units of the expression and variable match?

For example, you might be calculating a number of bytes when the number of words was meant -- if the result is used for allocation, you'd be using too much memory.

1.16. FUNCTION CALLS

- Is this function correct? Should it be a different function with a similar name? (E.g., strchr instead of strrchr?)
- Could this function violate the preconditions of a called function?

1.17. PARTICULAR FUNCTIONS

1.17.1. PRINTF and friends

- Are there extra arguments? (Too few are likely to be detected, provided the printf is executed.)
- Is the FILE argument of fprintf missing? (This happens *all* the time.)

1.18. RACES (Multiple processes)

- After a fork, what happens if a child exits immediately?
- Is a shared variable ever improperly referenced without locking?
- If locking is not always required because a variable is expected to remain unchanged, does it always remain unchanged?
- Does this code set a variable that another thread expects to remain constant?

1.19. SIGNALS

- Will there be a problem if the same signal arrives inside the signal handler? (Often problem when porting BSD code to SysV, because old-style sysV signal handlers don't block.)
- Does the code expect interrupted system calls to be restarted?
(SysV again - they may instead return an EINTR, which I've seen cause programs to go into infinite input-handling loops.)

1.20. MISCELLANEOUS

- Has unintended debug code been left in?
- Does the program have a specific exit value (on UNIX) or does it just "fall off the end"?

"Introducing Change Into the Testing Process"

TUTORIAL "C"

Mr. Mike Migdoll
Bank of America

Mr. Mike Migdoll has over 20 years of Information Systems experience in a variety of consulting and management positions responsible for quality/productivity, system development and end-user organizations. He is the architect of Bank of America's preventative testing program and manager of the Testing Improvement Project for the past three years. He collaborated with Software Quality Engineering on design of Bank of America's systematic testing training and managed system integration testing for the flagship "Alpha Account" product, involving interfaces among 40+ individual applications systems. Mr. Migdoll is Vice President of the Bay Area Quality Assurance Association and member of the Software Testing Analysis and Review (STAR) advisory group. He has been a speaker at Apple Computer and Pacific Bell Quality Forums and speaker and track chairman at the 7th International Testing Conference. He has a B.S. with dual major in Psychology and English and three years graduate work in research Psychology. During this time he co-published an article on the mechanics of verbal learning and memory.

IMPLEMENTING CHANGE

IN

THE TESTING PROCESS

Mike Migdoll, Vice President
Bank of America

TUTORIAL OVERVIEW

- □ **Tutorial Objectives**
- □ **Foundation for Change**
- □ **Selling Change**
- □ **Methodology for Change**
- □ **Implementing Change**
- □ **Final Perspectives**

TUTORIAL OBJECTIVES

- **Purpose**

  Prepare testers to be change agents so they can facilitate testing improvements in their companies

- **Goals**
  - **Provide change agent tools and techniques**
  - **Provide practical exercises**
  - **Provide roadmap to improved testing**
  - **Share perspectives**
  - **Develop action plans for change**

WORKSHEET #1

PERSONAL LEARNING OBJECTIVES:

Using the space provided here, list two or three questions which, if answered, would allow you to consider our time together well spent. The questions should focus on the challenges of marketing change or delivering testing services to your client community. Typically, the questions begin with "how". e. g. How do you sell a client on ...? How do you get clients to buy...? How do you accomplish...?

Hard stuff is easy . . .
Soft stuff is hard . . .

Soft stuff is more important than hard stuff ! ! !

- Milliken
1990 Baldrige Winners

TUTORIAL OVERVIEW

- Tutorial Objectives
- Foundation for Change
- Selling Change
- Methodology for Change
- Implementing Change
- Final Perspectives

FOUNDATION FOR CHANGE

- **The Change Agent**
- **Challenges**
- **Pitfalls**
- **Determining Readiness for Change**
- **Stages and Phases of Change**

CHANGE AGENCY AND THE CHANGE AGENT

- **Change Agency is the process of identifying, planning and institutionalizing change with minimum disruption and culture shock.**

- **The Change Agent facilitates the process of change by influencing the culture to shape processes and policies for improvement.**

CHANGE AGENT CHARACTERISTICS

- ☐ Tempered zeal
- ☐ Credible
- ☐ Thick-skinned
- ☐ Flexible
- ☐ Facilitator
- ☐ Influencing skills
- ☐ Sense of humor
- ☐ Bold
- ☐ Balances action & analysis
- ☐ Tactful mediator
- ☐ Confident
- ☐ Resilient
- ☐ Sensitive
- ☐ Collaborator
- ☐ Process oriented
- ☐ Patient
- ☐ Self-sufficient
- ☐ Consultative
- ☐ Opinion leader
- ☐ Adaptive
- ☐ Communicator
- ☐ Negotiator
- ☐ Problem solver
- ☐ Tenacious
- ☐ Tolerance for chaos

CHANGE AGENT CHALLENGES

- Required to continuously change and place self at risk
- Change is threatening, therefore so is the Change Agent
- Every group, project and corporate culture is different
- Unlearning is difficult
- Change evokes emotional confrontations
- Change involves intangibles
- Requires cooperation and collaboration versus competition
- Benefits of change are hard to demonstrate before the fact
- Change involves ambiguity and ambivalence
- The needs of the individual, the project, division and corporation may be in conflict
- Change takes time

COMMON PITFALLS

- Lack of motivation (or urgency) to change
- Lack of proven methodology to guide the change process
- Lack of infrastructure support
- Failure to identify target objectives
- Failure to integrate with existing culture
- Failure to identify business need
- Inadequately trained change agents and sponsors
- Lack of tools and techniques
- Failure to perform up front assessment
- Inadequate education and training
- Lack of resources
- Lack of credibility or trust

DETERMINING READINESS FOR CHANGE

Formula: C = (abd) > R

Where:

C = Change

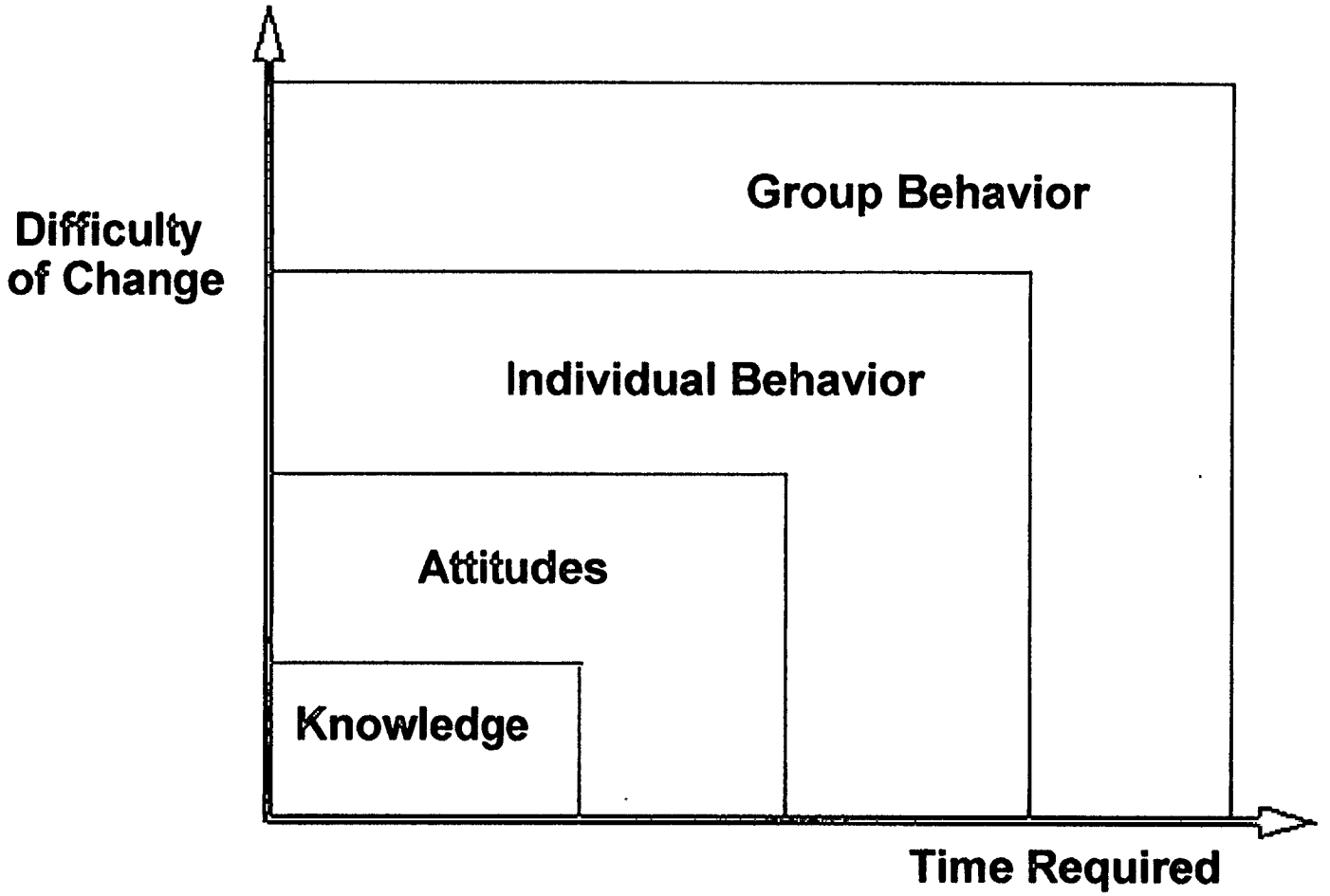
a = level of dissatisfaction

b = clear desired state

d = practical first steps toward the state

R = cost of changing (Resistance)

STAGES AND PHASES OF CHANGE:
Take it a step at a time



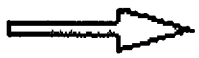
SOURCE: Software Quality Engineering

Progress is a nice word,
but change is its motivator.
And, change has its enemies.

- Robert F. Kennedy

Welcome to real world implementation!

TUTORIAL OVERVIEW

- Tutorial Objectives
- Foundation for Change
-  Selling Change
- Methodology for Change
- Implementing Change
- Final Perspectives

SELLING CHANGE

- **Identifying stakeholders**
- **Defining benefits**
- **Getting buy-in**

STAKEHOLDER INVOLVEMENT

- **Stakeholder is anyone who can impact the project or be impacted by it now or in the future**

- **Types of stakeholders**
  - **Sponsors**
  - **Change Agents**
  - **Targets**

AN OVERVIEW OF KEY STAKEHOLDERS

- **Sponsors (initiating)**
  - **Titles: EVP, SVP, RVP, Director**
  - **Key Characteristics:**
    - **Strategic planning; sets goals; clarifies direction/vision; lends support; breaks ties; legitimizes the change**

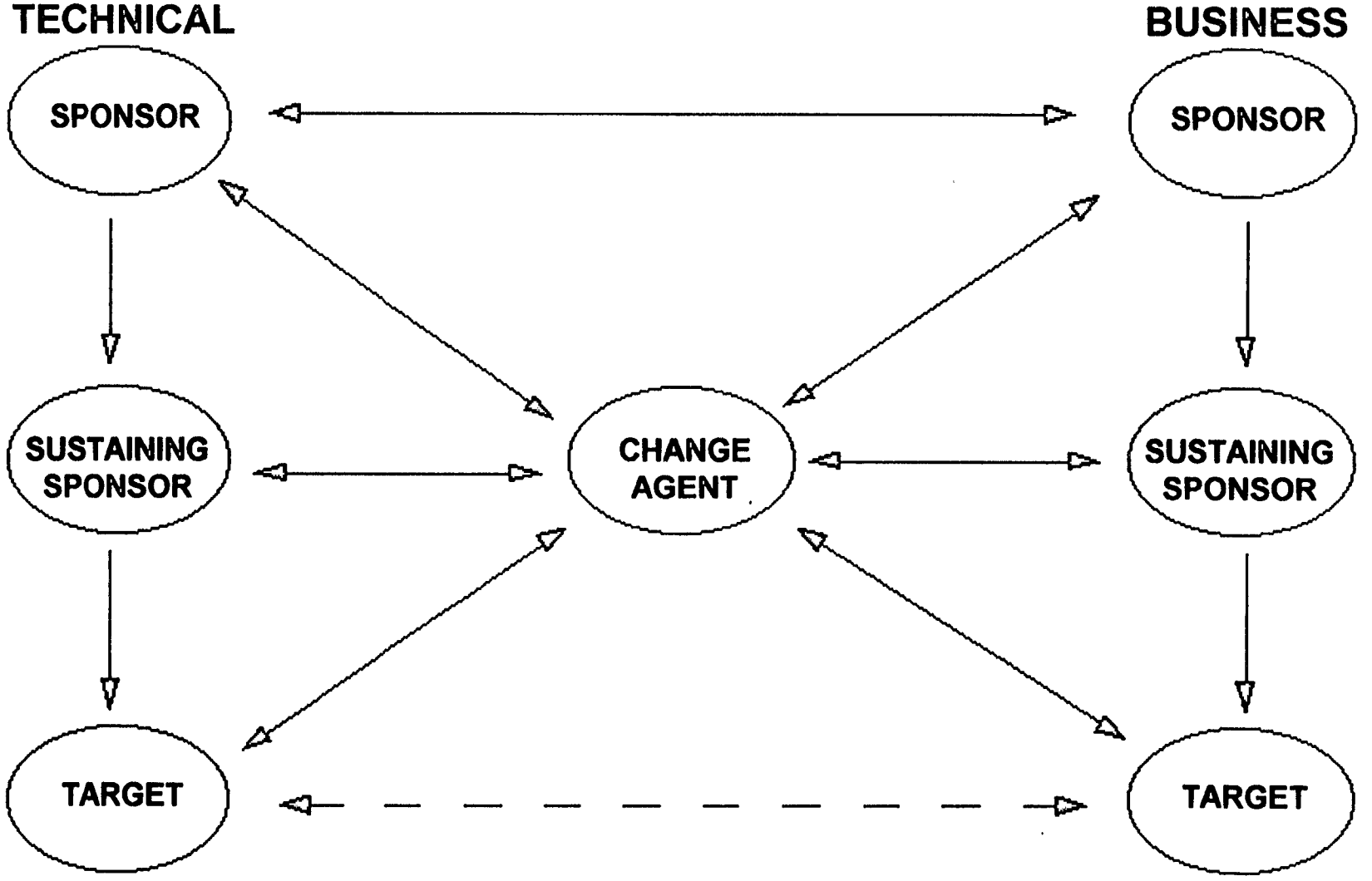
- **Sponsors (sustaining)**
  - **Titles: Division or Department Level Managers**
  - **Key Characteristics:**
    - **Identifies needs and problems; coordinates; markets; facilitates; ensures right priority; keeps people on track; takes issues back to Initiating Sponsor**

KEY STAKEHOLDERS (Continued)

- **Change Agents**
  - Titles: Support Organizations, i.e. quality assurance/testing groups, data administration, standards groups, technology implementation teams
  - Key Characteristics:
    - Performs on-going analysis of problem and adjusts plan as required; enables, facilitates, coordinates the change; brings required skill sets; ensures that target group has everything they need

- **Targets**
  - Titles: Project Leaders, technicians and business analysts
  - Key Characteristics:
    - Individuals or groups who actually must change

STAKEHOLDER RELATIONSHIPS



WORKSHEET #2

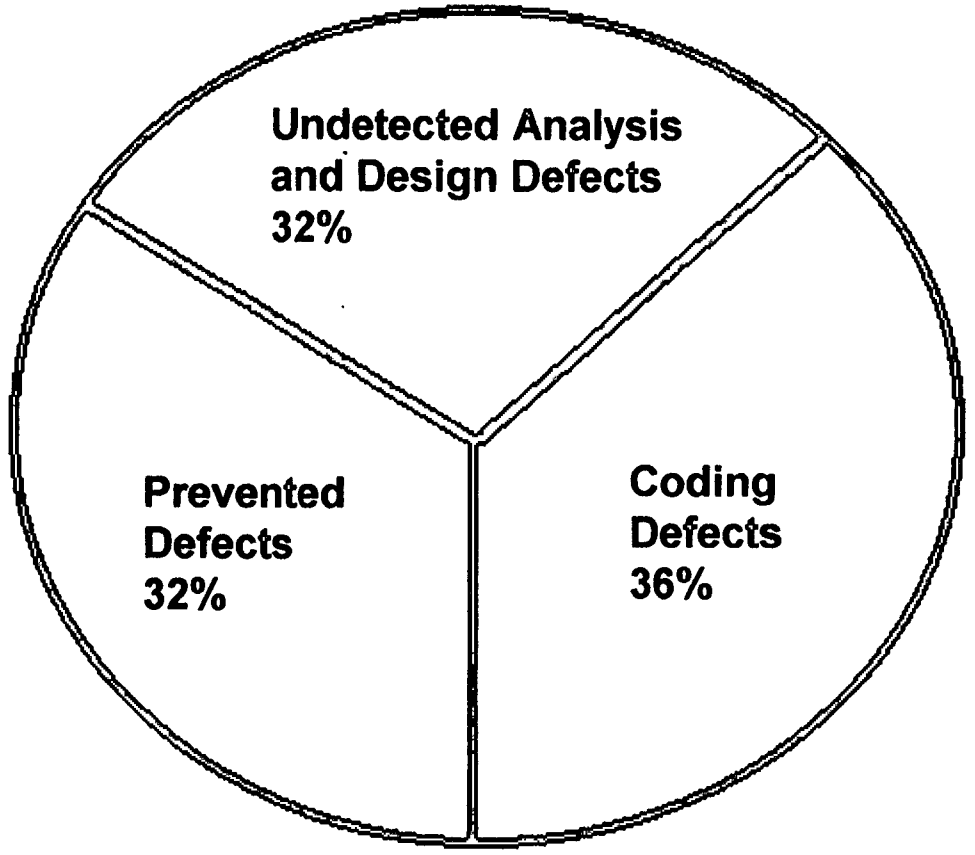
1. List SPONSORS

2. List SUSTAINING SPONSORS

3. List CHANGE AGENTS

4. List TARGETS

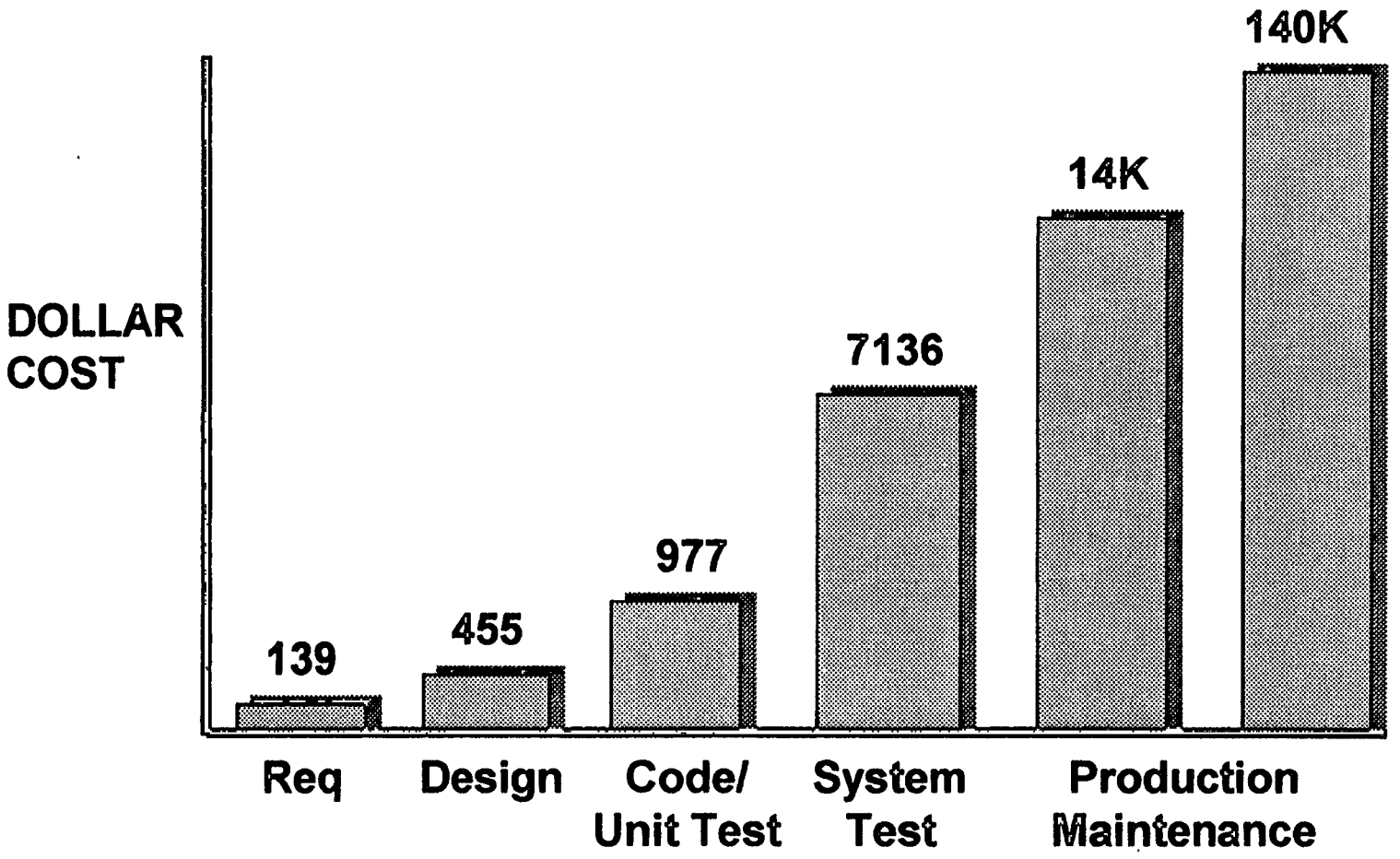
DEFINING BENEFITS: Early detection



1/3 of all defects prevented

Source: William E. Perry, A Structured Approach to Systems Testing

DEFINING BENEFITS:
Cost Reduction



IBM, ROCKWELL, TRW SURVEYS

GETTING BUY-IN

- Task forces
- Steering committees
- Nominal focus groups
- Surveys
- Interviews
- External consultants
- Prototypes

BANK OF AMERICA APPROACH : Getting buy-in

- **Resources and Methods**
  - **Internal Steering Committee**
    - **Draft policies and procedures**
    - **Distribute for review and approval**
    - **Present recommendations to management**
  - **External consultants**
    - **Conduct testing practices assessment**
    - **Collaborate on training improvements**
    - **Provide external perspective**
  - **Prototypes**

TUTORIAL OVERVIEW

- □ **Tutorial Objectives**
- □ **Foundation for Change**
- □ **Selling Change**
- ⇒ □ **Methodology for Change**
- □ **Implementing Change**
- □ **Final Perspectives**

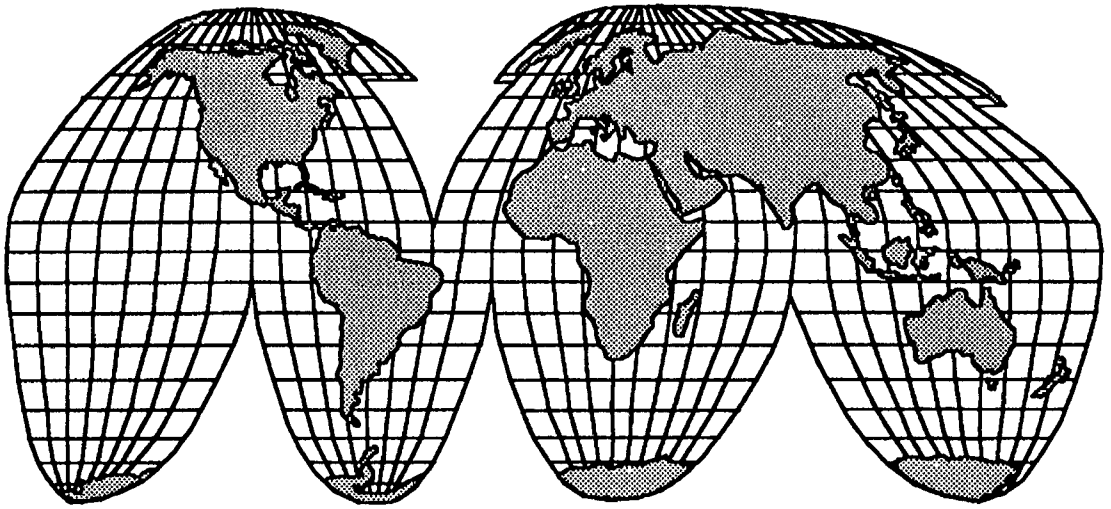
METHODOLOGY FOR CHANGE

- **Define goals and objectives**
- **Develop strategy for change**
- **Develop plan for change**

DEFINING GOALS AND OBJECTIVES

- **Assessments and problem analysis**
  - **Internal**
  - **External**
- **Stakeholder involvement**
  - **Steering committees**
  - **Nominal group process**
  - **Pilot projects**
- **Scoping and prioritizing**
  - **Weighted ranking of goals/objectives**

If you don't know where you're going, any road will do. - Chinese Proverb



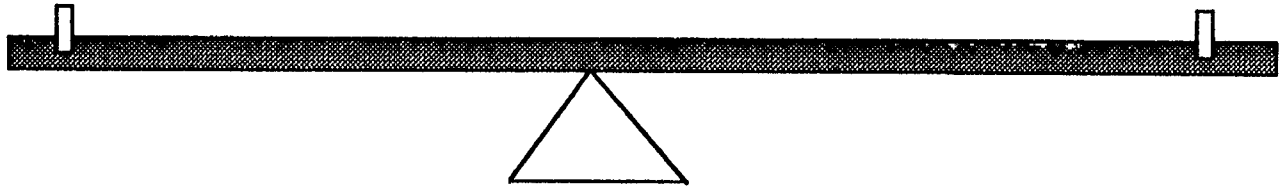
If you don't know where you are, a map won't help. **- Watts S. Humphrey**

DEVELOPING STRATEGY AND PLAN

- **Strategy**
  - The approach chosen to achieve a goal or objective
- **Plan**
  - A set of tasks to achieve a special goal or objective
- **Strategy versus Plan**
  - Plans are based on strategies
  - Plans should have clearly defined ends
  - A perfect plan with a bad strategy will usually fail
  - A good strategy can succeed even with poor execution

BEST FIT FOR MOST WORTH

	FIT	WORTH
First	Organization	Individual Employee
Second	Individual Department	Individual Department
Third	Individual Employee	Organization



SOURCE: Villard-Lindsay & Company

BANK OF AMERICA: Strategy for change

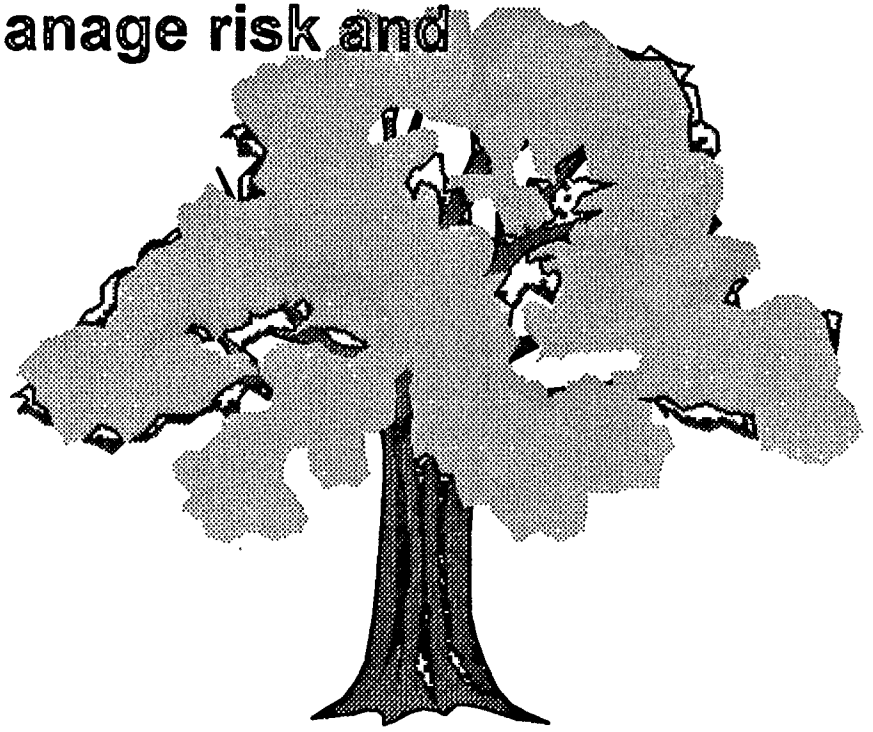
Testing Improvement Project (TIP)

- **Implement Testing task Force recommendations:**
  - **Four strategic objectives**
    - **Develop testing methodology**
    - **Enhance testing training**
    - **Increase use of testing tools**
    - **Automate testware management**
- **Build on existing culture**
  - **Testing process**
  - **Testing deliverables**

BANK OF AMERICA:
Testing Improvement Project

MISSION:

To provide support for early detection and correction of defects in all project deliverables in order to manage risk and minimize rework costs.



Products/Services:

Innovation

Consultation

Education

Methodology

Tools

TUTORIAL OVERVIEW

- ☐ **Tutorial Objectives**
- ☐ **Foundation for Change**
- ☐ **Selling Change**
- ☐ **Methodology for Change**
- ⟹ ☐ **Implementing Change**
- ☐ **Final Perspectives**

IMPLEMENTING CHANGE

- **Tie to strategy**
- **Products/Deliverables**
  - **Processes**
  - **Pro Formas**
- **Automation**
- **Organization**
  - **Resources and funding**
  - **Scheduling**
  - **Training**

BANK OF AMERICA: Strategy for change

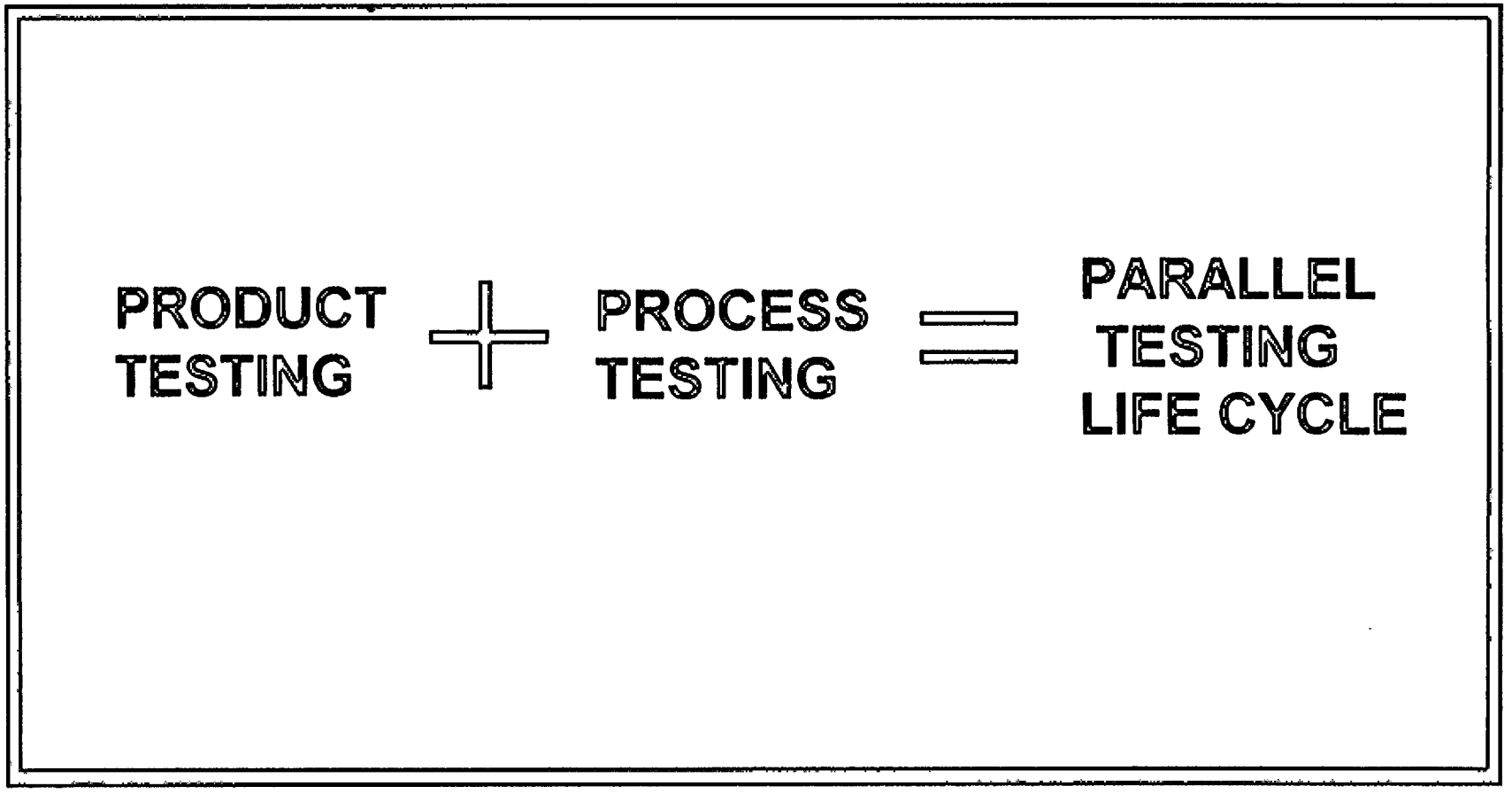
Testing Improvement Project (TIP)

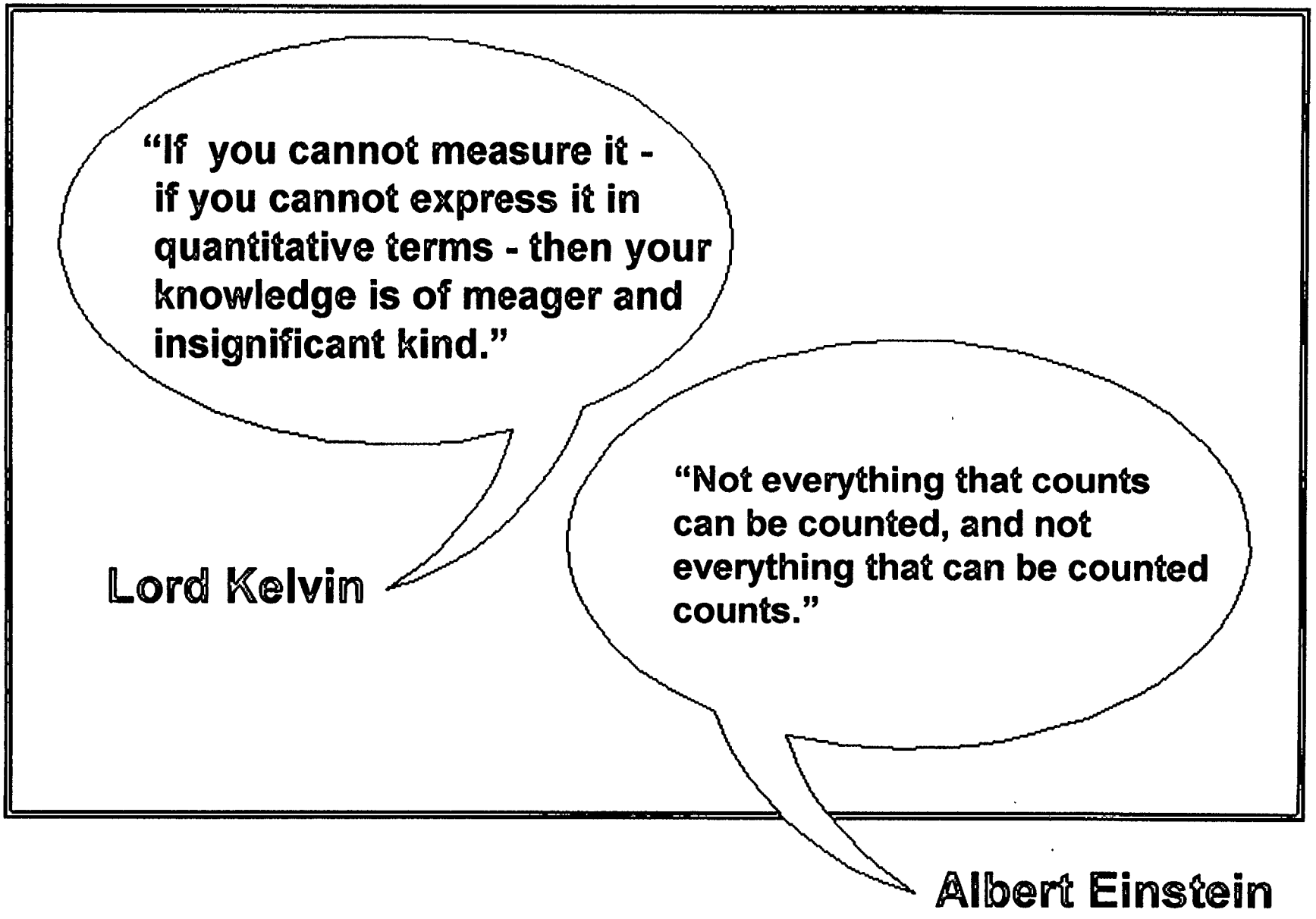
- **Implement Testing task Force recommendations:**
  - **Four strategic objectives**
    - **Develop testing methodology**
    - **Enhance testing training**
    - **Increase use of testing tools**
    - **Automate testware management**
- **Build on existing culture**
  - **Testing process**
  - **Testing deliverables**

PROCESS DELIVERABLES

- **Testing Lifecycle Methodology**
  - **Emphasis on prevention**
  - **Start at requirements time**
  - **Modeling with test cases**

- **Evaluation Processes**
  - **Formal review methodology**
  - **Test tracking**
  - **Defect recording and analysis**
  - **Measuring**


PRODUCT TESTING
PROCESS TESTING
PARALLEL TESTING LIFE CYCLE

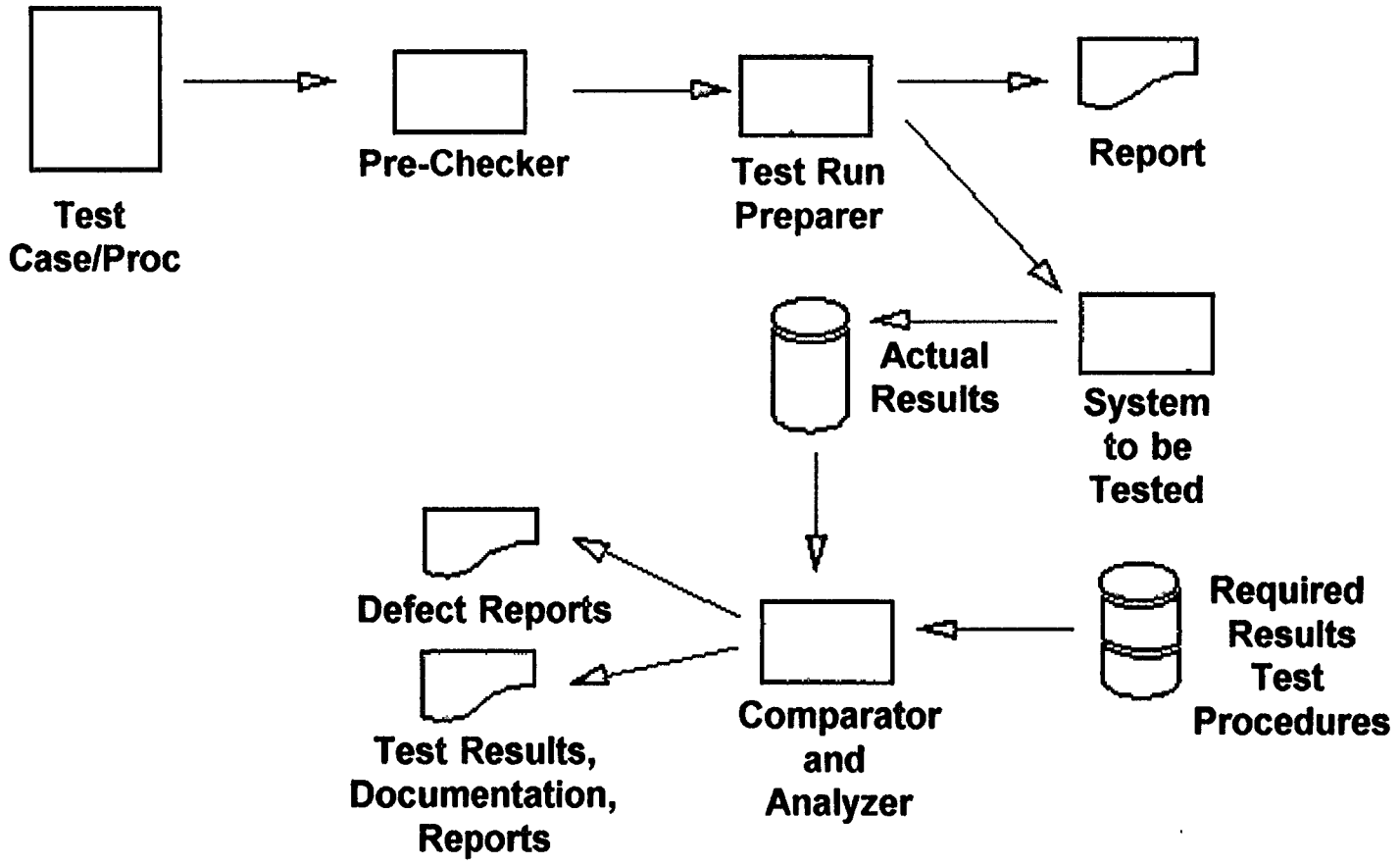

"If you cannot measure it - if you cannot express it in quantitative terms - then your knowledge is of meager and insignificant kind."
Lord Kelvin
"Not everything that counts can be counted, and not everything that can be counted counts."
Albert Einstein

STANDARDS AND PROCEDURE PRO FORMAS

- **Develop your own**

- **External models from . . .**
  - **ANSI 1012 - 1986**
  - **IEEE**
  - **DOD**
  - **National Bureau of Standards**
  - **External consultants**
  - **Tool vendors**
  - **Leading companies, e.g. banks, software firms, aerospace**

BANK OF AMERICA'S AUTOMATED TEST FACILITY : A Conceptual Framework



BANK OF AMERICA'S AUTOMATED TEST FACILITY : Objectives

- **Store and maintain test cases in machine readable form with random selection capability**
- **Automatically generate refreshable, controllable files and databases containing target records against which test cases are processed**
- **Provide flexibility and ease of change of test cases, files and databases**
- **Relate test cases to acceptance requirements, programs and data**
- **Integrate tools to assist in running tests and verification of coverage**
- **Provide automated test documentation and results reporting**
- **Assist test checkout and evaluation by providing easy vehicles for record and field display and validation**

SOURCES FOR TOOL INFORMATION

- **Test Conferences**
  - Vendor shows at most major events
- **Reference Publications**
  - DATAPRO reports and evaluations
  - International Directory of Software
  - Reifer Software Tools Directory
  - SQE Testing Tools Reference Guide
- **Databases**
  - NBS Tools Database
  - SOFSEARCH Software Database
- **User Groups**

ORGANIZATION

- **Structure**
- **Infrastructure**
- **Staffing**
- **Funding**
- **Scheduling**

ORGANIZATION STRUCTURE

Options . . .

- **Internal / External Consultants**
- **Test Unit / Independent Test Unit**
- **Coordinators**
- **Support Unit**
- **Quality Assurance**
- **Audit Group**
- **Other . . .**

ORGANIZATION INFRASTRUCTURE

- Project Management
- Metrics
- Methodology
- Tools and Support
- Quality Efforts
- Process Improvement Efforts
- Standards
- Managing Change Support

STAFFING:
Skills and experience

- ☐ **Experience as a programmer**
- ☐ **A thick skin**
- ☐ **A good sense of humor**
- ☐ **Tolerance for chaos**
- ☐ **Tenacity**
- ☐ **Skepticism**
- ☐ **Cunning**
- ☐ **Boldness**
- ☐ **Self-sufficiency**
- ☐ **Honesty**

TRAINING: Learning styles

Four basic learning profiles. . .

CONCRETE
EXPERIENCE
- Heavy reliance on feelings
- Learn best from specific examples
- Generally find theoretical approaches distasteful

ABSTRACT
CONCEPTUALIZATION
- Analytical conceptual approach to learning
- Learn best from systematic analysis
- Dislike unstructured discovery approaches

ACTIVE
EXPERIMENTATION
- Active doing orientation to learning
- Dislike passive learning, e.g. lectures
- Like projects and group activities

REFLECTIVE
OBSERVATION
- Reflective approach to learning based on observation
- Prefer lecture

FUNDING AND SCHEDULING

- **Funding Guideline: 3 - 5% of Development Budget**

- **Example:**

Estimated Cost of Improvement	**$ 3,000.**
Total Project Development Budget	**$100,000.**

3% Project Saving = R.O.I. Break Even Point

- **Schedule Phases and Stages Accordingly**

TUTORIAL OVERVIEW

- ☐ **Tutorial Objectives**
- ☐ **Foundation for Change**
- ☐ **Selling Change**
- ☐ **Methodology for Change**
- ☐ **Implementing Change**
- ⟹ ☐ **Final Perspectives**

FINAL PERSPECTIVES

- **Lessons learned**
  - Lessons learned at Bank of America
  - Successful changes at Bank of America
  - What is easy and what is hard
  - Costs versus benefits revisited
- **Next steps**
- **Summary and Recommendations**

LESSONS LEARNED AT BANK OF AMERICA

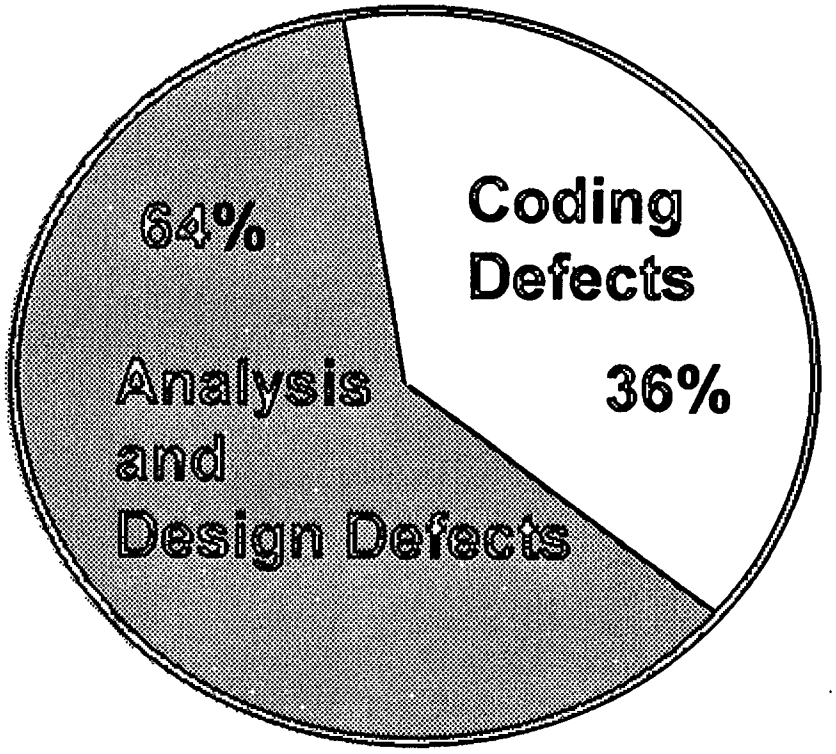
- **Have patience**
  - **Consensus building**
  - **Institutionalizing is a slow process**
  - **Evolution versus revolution**
  - **Tempered zeal**
  - **Education is crucial**
- **Emphasize spirit over letter**
- **Build credibility**
  - **Recruit a competent staff**
  - **Deliver tangible products**
  - **Keep selling**
- **Recognize metrics as a challenge**

SUCCESSFUL CHANGES AT BANK OF AMERICA

- **New methodology**
  - **Broadening acceptance**
  - **Shifting emphasis to prevention**
- **Enhanced training**
  - **System/Acceptance testing**
- **Increased use of testing tools**
  - **On-line tool catalog**
  - **Automated test facility**
- **Automated testware management**
  - **Repository strategy**
- **Requirements and metrics steering committee**

COSTS VERSUS BENEFITS REVISITED

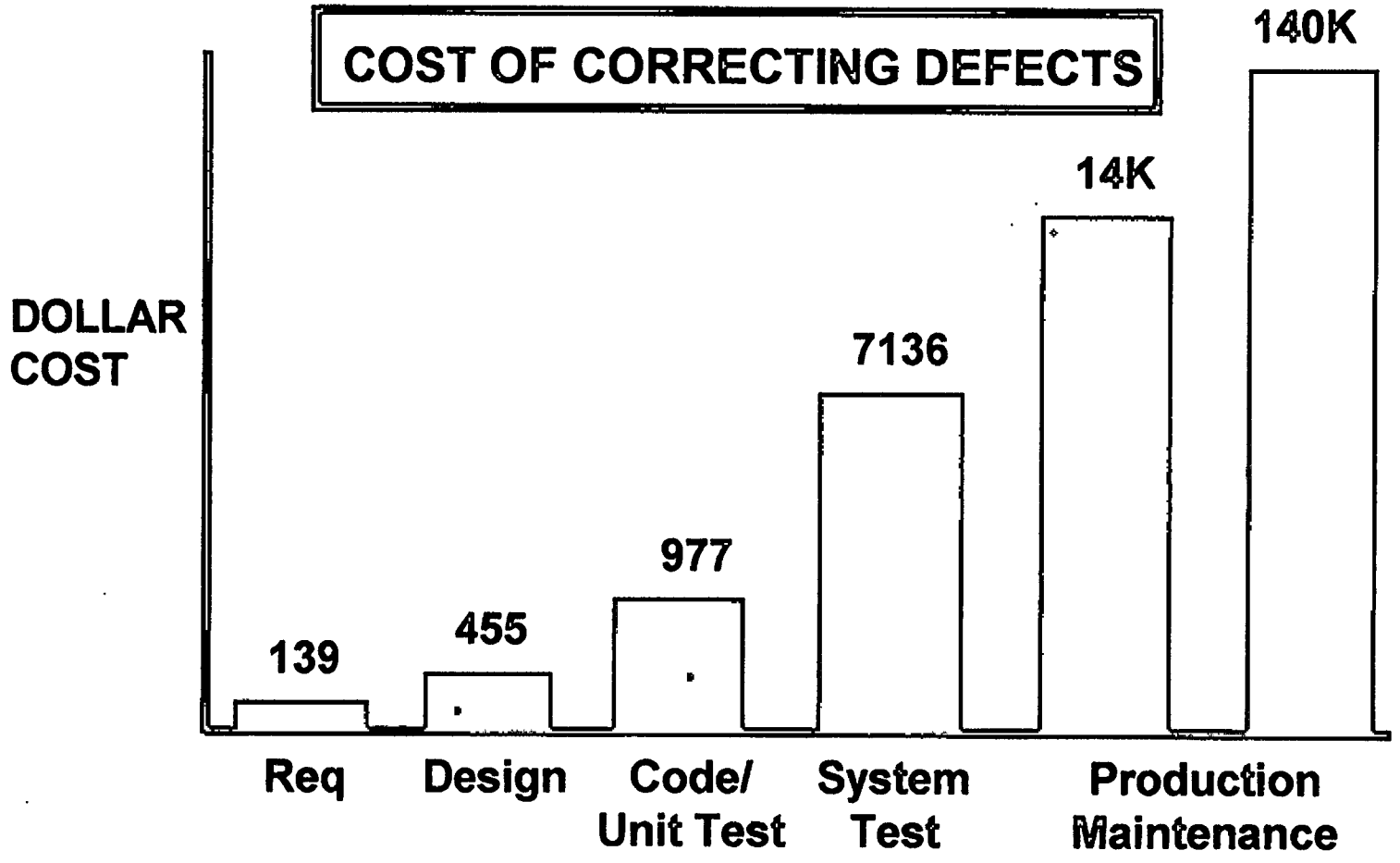
Prevention . . .



Typical Defect Distribution Today

SOURCE: J.L. Hancock, Sr VP, Chemical Bank of New York

EARLIER DETECTION



IBM, ROCKWELL, TRW SURVEYS

NEXT STEPS

- **Sell Need for Change**
- **Task Force/Steering Committee**
- **Sell Recommendations**
- **Assessment**
- **Sell Assessment Findings**
- **Develop Action Plans**
- **Sell Action Plans**
- **Implement Action Plans**
- **Evaluate**
- **Repeat, Repeat, Repeat**

SUMMARY AND RECOMMENDATIONS

- **Learn Change Agency techniques**
- **Create consensus**
- **Create partnerships with stakeholders**
- **Set clear objectives**
- **Ensure management support**
- **Build on current culture**
- **Sell and educate at all levels**
- **Be patient and flexible**
- **Get external perspective**
- **Track results and publicize stories**
- **Institutionalize change process**

"Total Quality Optimization in Real-World Testing"

TUTORIAL "D + I"

Mr. Shel Siegel and Mr. Bill Silver
YESS! Inc.

Mr. Shel Siegel is co-founder of the *Alliance for Software Quality* and the *Total Quality Optimization* (*TQO*) deployment methodology. His is an editor for *Software Quality World* and the author of numerous articles on the subjects of software quality, software engineering and testing techniques. A frequent presenter at conferences, he is also the president of YESS!, Inc., a professional support organization specializing in helping companies close the technology gap.

"Applying Total Quality Optimization to Real-World Testing"

TUTORIAL "I + D"

Mr. Bill Silver and Mr. Shel Siegel
ProQual, Inc.

Mr. Bill Silver is a software quality consultant and publisher and editor of *Software Quality World*. During Bill's 25 year career, he has held the positions of V.P. of Customer Service & Corp. Quality at Computer Corporation of America as well as Director of Software Quality for Prime Computer, Inc. and Atex (KODAK). He has given a wide range of presentations and seminars on software quality and has experience managing all phases of the software life cycle.

Welcome

to

Total Quality Optimization™

for Software

A Three-Day Seminar Authored by

Shel Siegel, President of YESS!, Inc.

&

Bill Silver, Publisher of *Software Quality World*

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Your TQO™ Course Developers:

Bill Silver

Bill Silver is president of ProQual and publisher of ***Software Quality World*** —a magazine dedicated to helping software practitioners apply quality principles to software. His 25+ year career includes:

VP of Customer Service & Corp. Quality
Computer Corporation of America

Director of Software Quality
Prime Computer; and Atex (KODAK)

Manager of Advanced Development

Multics operating system designer

Hands-on experience in all life cycle phases:

TQM	*testing*
metrics	*performance*
development	*product management*
hot-line support	*maintenance*
user documentation	*customer training*

Shel Siegel

Shel Siegel is president of YESS!, Inc.—specialists in helping companies close the quality gap. Together with Bill he co-founded the ***Alliance for Software Quality*** and co-developed ***Total Quality Optimization (TQO)***. Shel's 20+ year career includes:

VP Marketing
Cattier, Inc.

Director of Quality
Interactive Development Environments

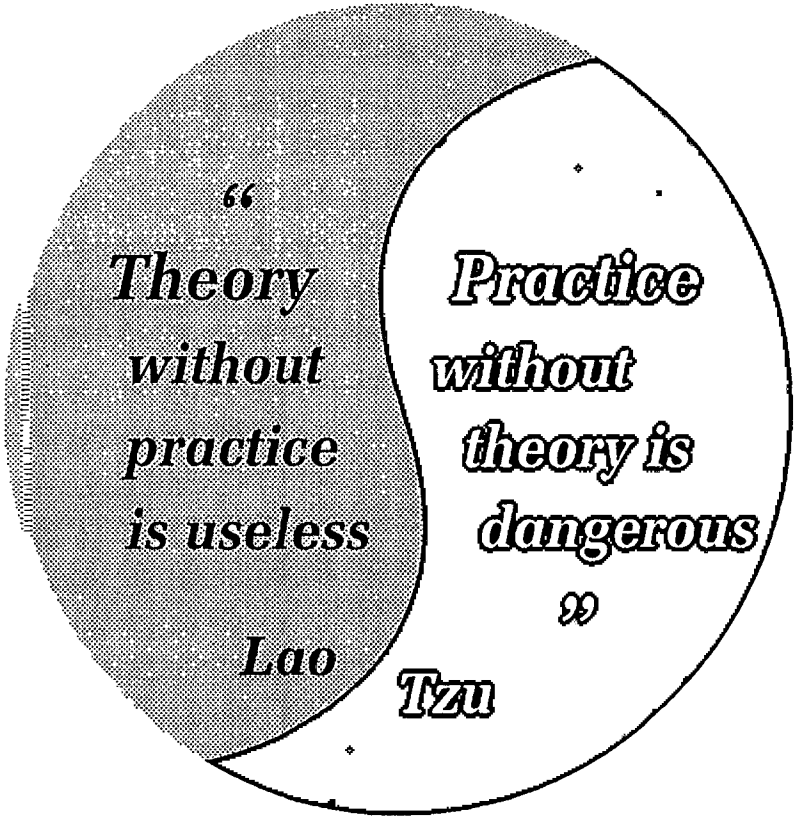
Chairman for Strategic Planning
Systems Evaluation Laboratory

Technical Staff: Testing and Q.A.
IBM Santa Teresa Labs

Shel earned an M.S. in Computer Science from the S.U.N.Y. Binghamton School of Advanced Technology.

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

The Wisdom of Giants

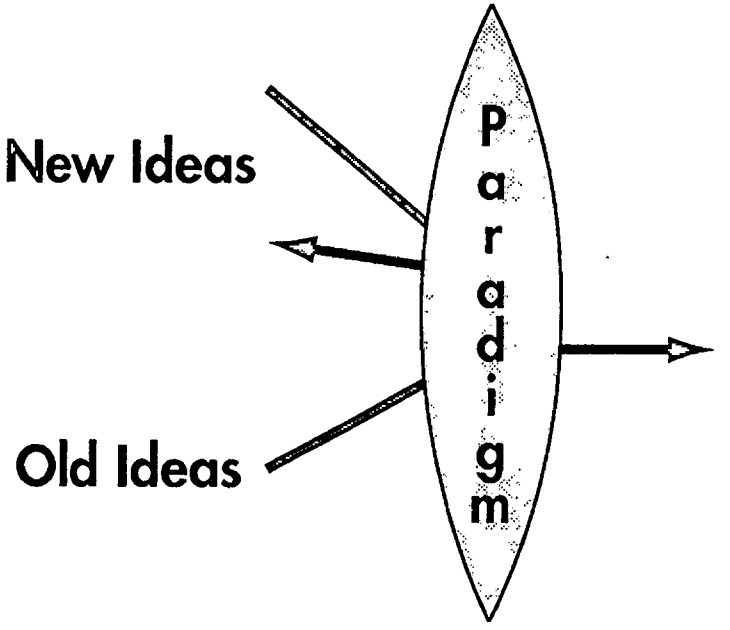


"There is nothing as practical as good theory!"

Dr. W. Edwards Deming

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Paradigm Principles

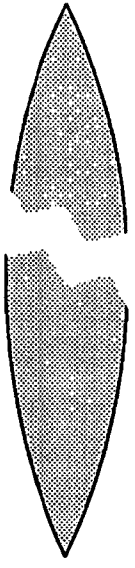


- A Lens not a Paragon
- Paradigm Paralysis
- Paradigm Peril

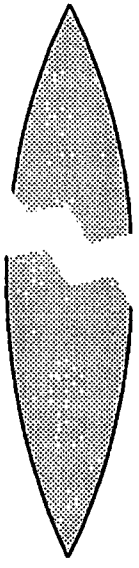
© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

TQO's Three Paradigm Shifts

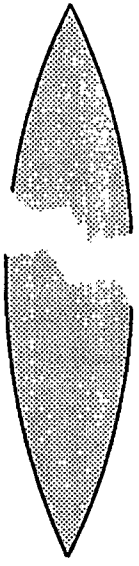
What is Quality?



How do you get quality?



Can you Apply Quality to Software?



© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Your TQO for Software Seminar

Day 1

① *Introduction*

② *The Quality Imperative*

③ *The Optimum Definition of Quality*

④ *The Quality Deployment Dilemma*

⑤ *Introduction to TQO*

⑥ *The Optimization Game* [Exercise]

Day 2

① *Brief Review of TQO*

② *TQO for a Sales Rep* [Case Study]

③ *TQO Measurement Paradigm*

④ *In-Depth TQO Review*

Values

Leadership

Quality Management, Process & Infrastructure

Optimization Cycles

⑤ *Breakthrough Bead Experiment* [Exercise]

Day 3

⑩ [Extended Case Study] *TQO at XYZ Software*

Values

Leadership Cycle

Organization Optimization Cycle

Software Process Optimization Cycle

Assessment

Measurement

⑫ [Extended Exercise] *TQO for Your Organization/Process*

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

TQO Training Objectives

	What	How	Softw.
Aware	B	B	B
Understand	TH	TH	TH
Value	TF	TF	TF
Apply	S2	S2	S3
Master	D	D	D

B = Briefing (1.0 hour)
TH = Tutorial (Half-Day)
TF = Tutorial (Full-Day)
S2 = Seminar (2-Day)
S3 = Seminar (3-Day)
D = Degapping

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Defining Quality

What is Quality

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

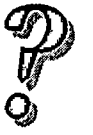
Obtaining Quality

How do you get Quality

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

What is the #1

Software Quality Issue



© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

The Optimum Definition of Quality

"

Quality is

doing the Optimum things

the Optimum way.

"

Bill Silver

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Attack the 3 Dimensions of a Company

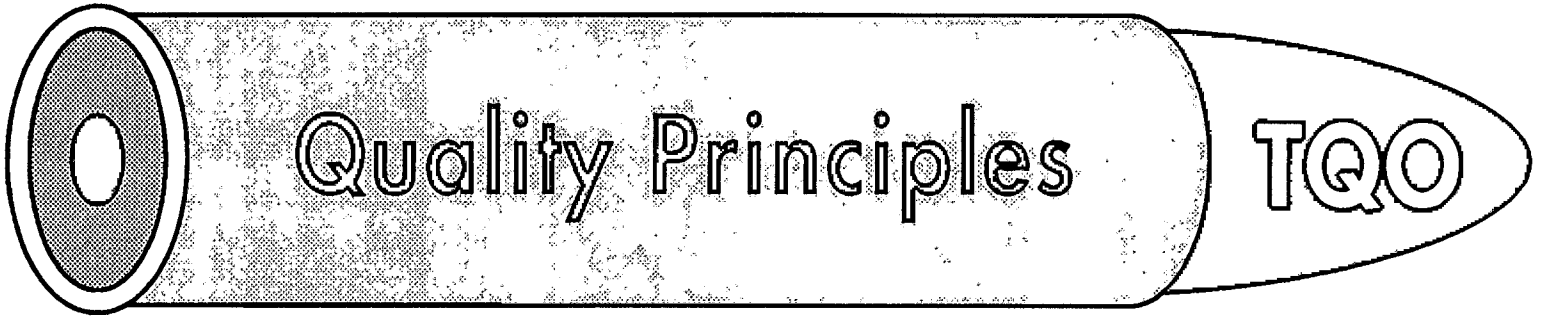
- Organization
- Process
- Infrastructure

Apply Chaos Science

- Iteration
- Fractal Scaling
- Optimization Attractors

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

The Only Software Quality Silver Bullet



"The # 1 software quality issue is quality."

Bill Silver

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

The Quality Imperative

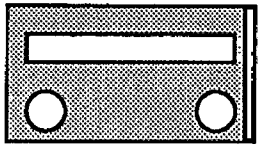
- ❑ A Brief History of Quality
- ❑ Quality and the Software Industry
- ❑ The Goal of Quality
- ❑ Productivity & Value
- ❑ Quality Losers
- ❑ The Cost and Loss of Poor Quality
- ❑ Quality Winners

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

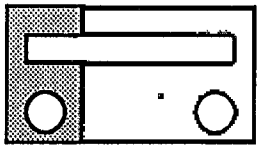
Declining U.S. Market Share

Consumer Electronics*

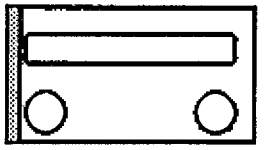
1955 96%



1965 30%

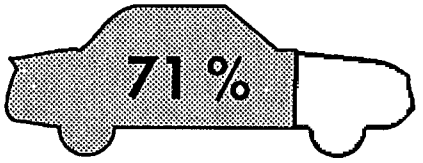


1975 5%



Automobiles*

1950



1980

1992

* Source: *Beyond Quality* by Bowles & Hammond

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

A Quality Tale of Two Nations / Companies

Japan

'46 JUSE Founded

'51 Deming Prize Founded

'56 K. Ishikawa TQM Book/Radio

United States

'46 ASQC Founded

'87 Baldrige Award Founded

'80 Deming TV Documentary

Toyota

'61 Ohno's 5 Whys

'70 Ohno's Just-In-Time

'90 Toyota $$$ = GM + Ford + $5B

Ford

'80 Ford Loses $1.54 Billion

'81 Ford Launches Team Taurus

'92 Taurus #1 Selling Car in U.S.

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

A Brief History of Quality in the U.S.

***1940's*:** World demand for quantity in an industrial vacuum

***1950's*:** U.S. is unchallenged industrial leader

***1960's*:** U.S. is preoccupied with social and political problems

***1970's*:** Oil crises and loss of some strategic industries

***1980's*:** For a few, a quality awakening
For many, continued denial and arrogance
Management believes technology is the answer

***1990's*:** General recession, declining competitive position noted
Growing quality gap between U.S. companies

***2000*:** A world economy in which the economic survivors have embraced and implemented *OPTIMUM* quality practices as a strategic imperative

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Software Industry Problems & Issues

Stable and complete REQUIREMENTS!

Not understanding value to the **customer** and the **user** leads to producing the wrong set (subset) of features.

A major impact to schedule are late and changing requirements.

Defects cause WASTE!

A major impact to schedule are endless test & rework cycles.

It costs a lot more to find and fix defects later in the life cycle.

When are you DONE?

What is the real market window of opportunity?

What level of quality does the market really require?

How to MEASURE quality?

You must measure user/customer satisfaction **AND** the cost to produce, maintain, and support.

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

The Goal of Quality

"

The goal of quality is *Business Success.*

This means providing all corporate stakeholders with genuine value in the form of:

Value at fair prices for *customers*

Pride and security for *employees*

Profits for *shareholders*

Long-term, stable relationships for *suppliers*

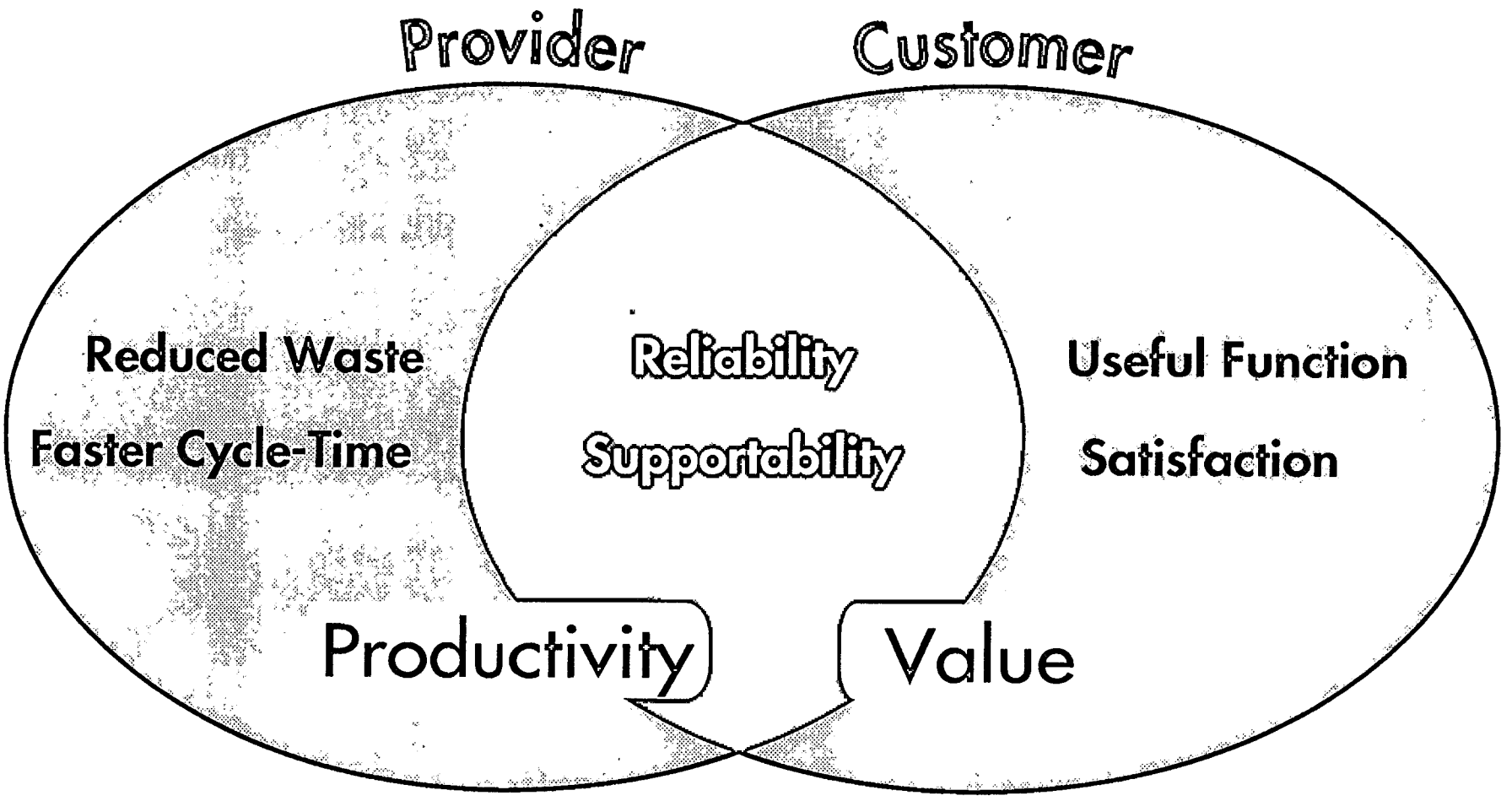
Benefits for the *community* at large.

"

Bill Silver and Shel Siegel

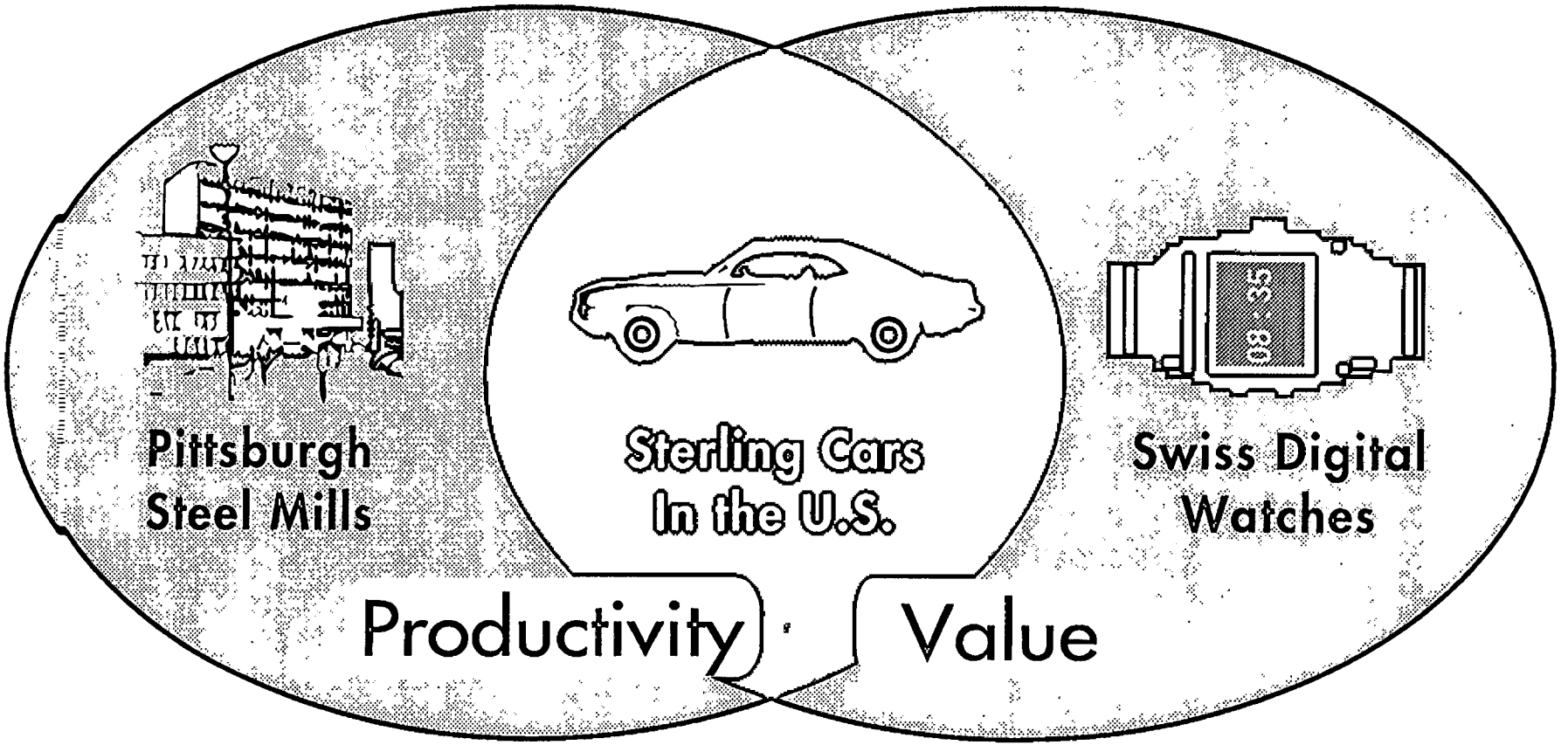
© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Quality Optimizes Productivity & Value



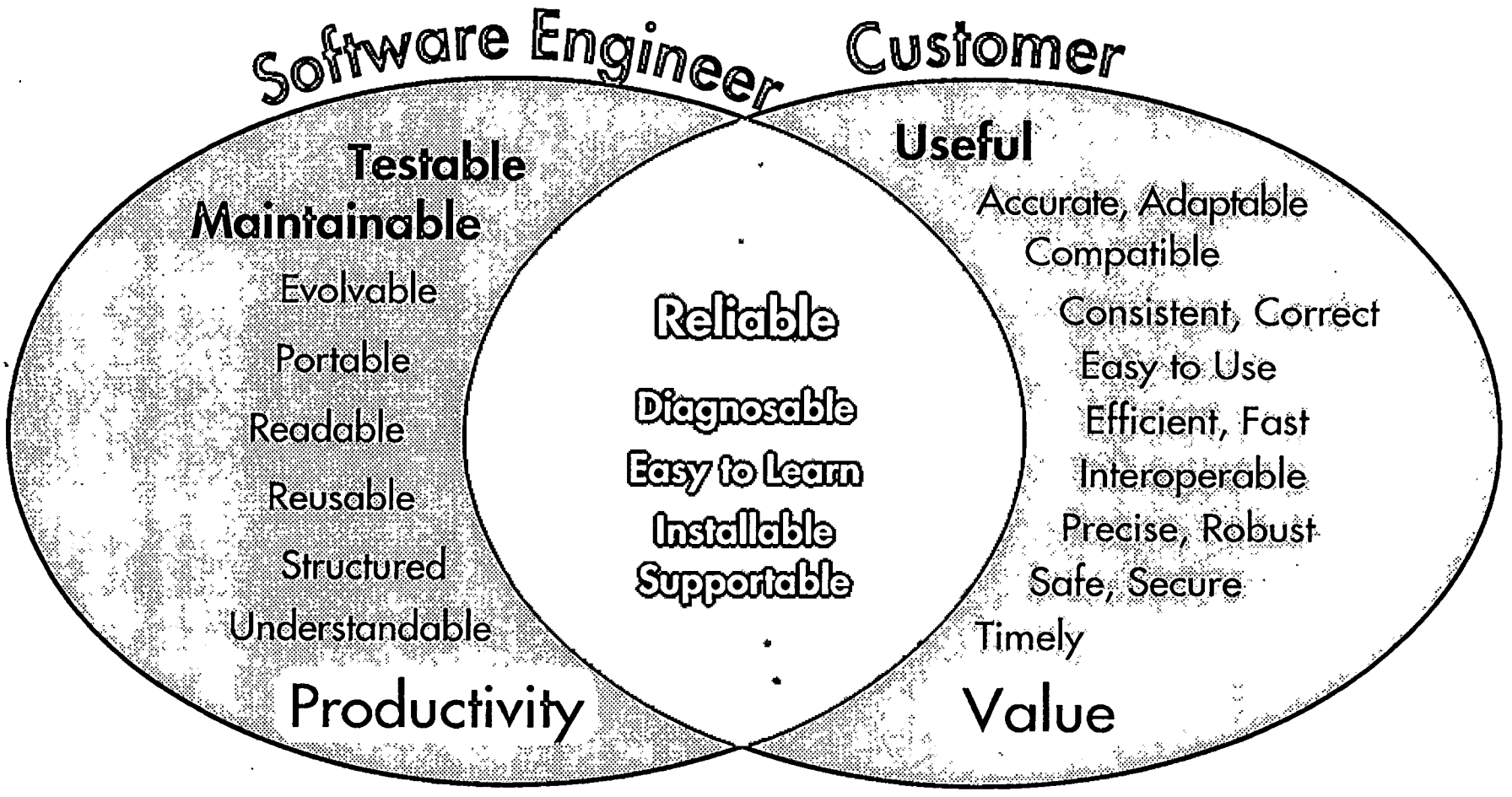
© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

The Fatal Attraction of Suboptimization



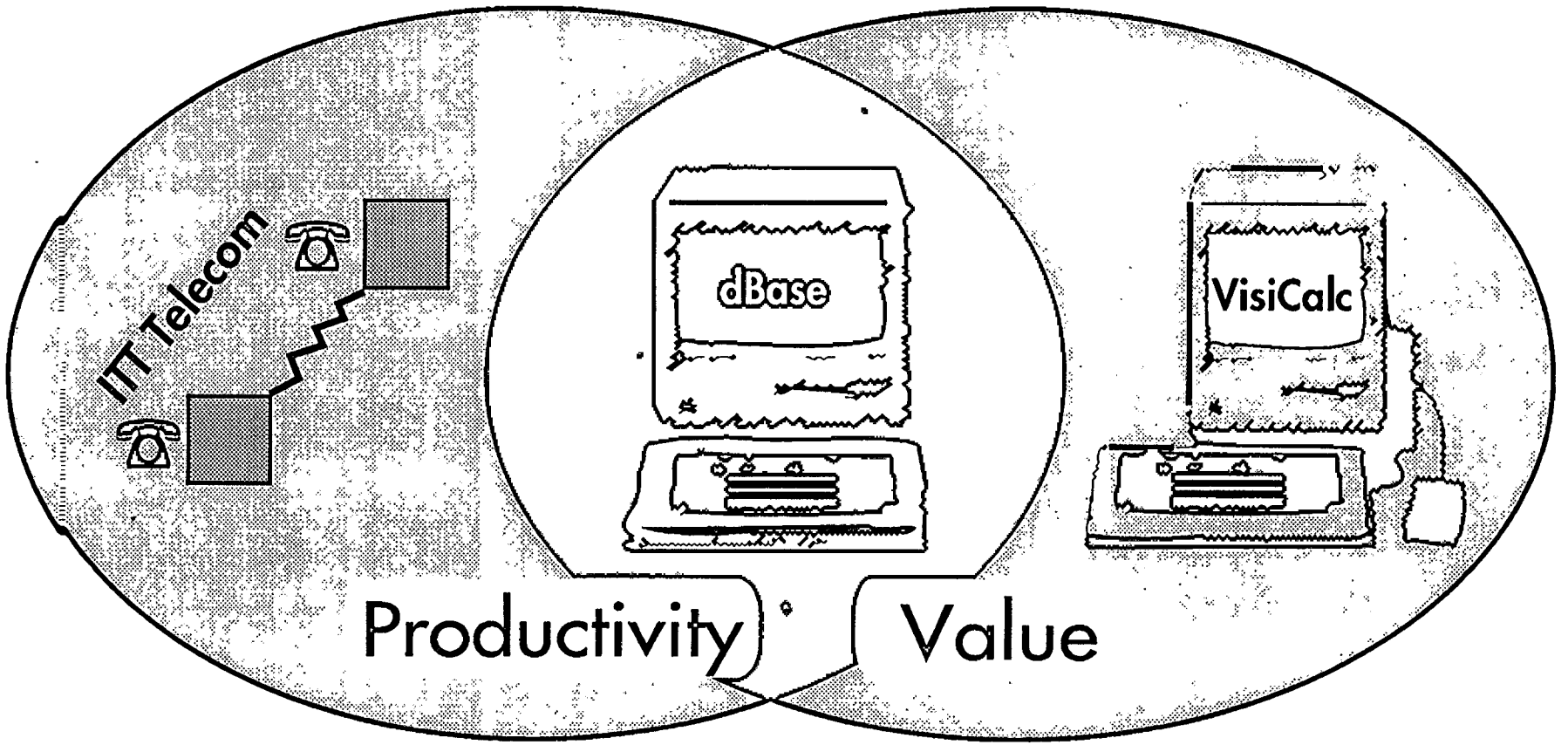
© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Optimizing Software Quality Attributes



© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

The Fatal Attraction of Suboptimization



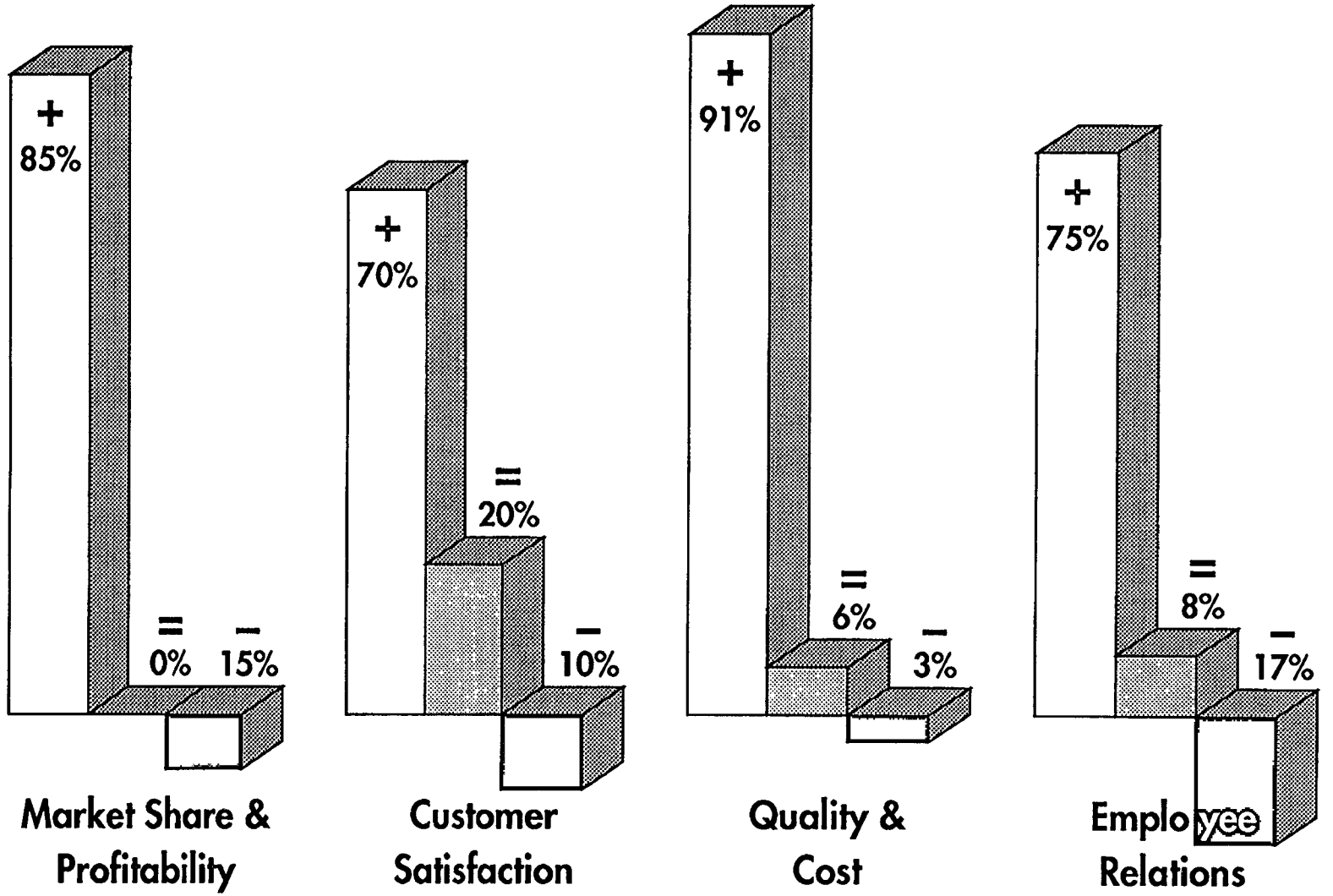
© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

The COST and LOSS of Poor Quality

- ***Cost of Defects***
  - **Detection**
  - **Rework**
  - **Support**
- ***Cost of Inefficiency***
  - **Wasted Steps**
  - **Wasted Resources**
- ***Loss of Revenue***
  - **Delayed Release**
  - **Poor References**
  - **Lost Sales Effort**
- ***Loss of Goodwill***
  - **Customers**
  - **Employees**

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

U.S. GAO Study Shows Quality is Good Business



Baldrige Finalists Improve Business Performance Indicators

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

The Optimum Definition of Quality

- How the Gurus Define Quality
- The Optimum Definition
- A Quality Taxonomy
- Operational Definitions
- Understanding Optimum

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

How the Gurus Define Quality

Noah Webster	that which makes something what it is basic nature degree of excellence, superiority
William E. Conway	waste reduction
Philip B. Crosby	conformance to customer requirements DIRFT: do it right the first time exceeding customer requirements zero defects
W. Edwards Deming	reduced variation
A. V. Feigenbaum	reduced cost of quality
Joseph M. Juran	fitness for use
Motorola	6 Sigma
Myron Tribus	love
Gerald M. Weinberg	value to customers

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

The Optimum Definition of Quality

"

Quality is

doing the Optimum things

the Optimum way.

"

Bill Silver

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

The Meaning of OPTIMIZE

"

Optimize:

① **to achieve the best possible under restrictions expressed or implied**

② **to realize to the utmost**

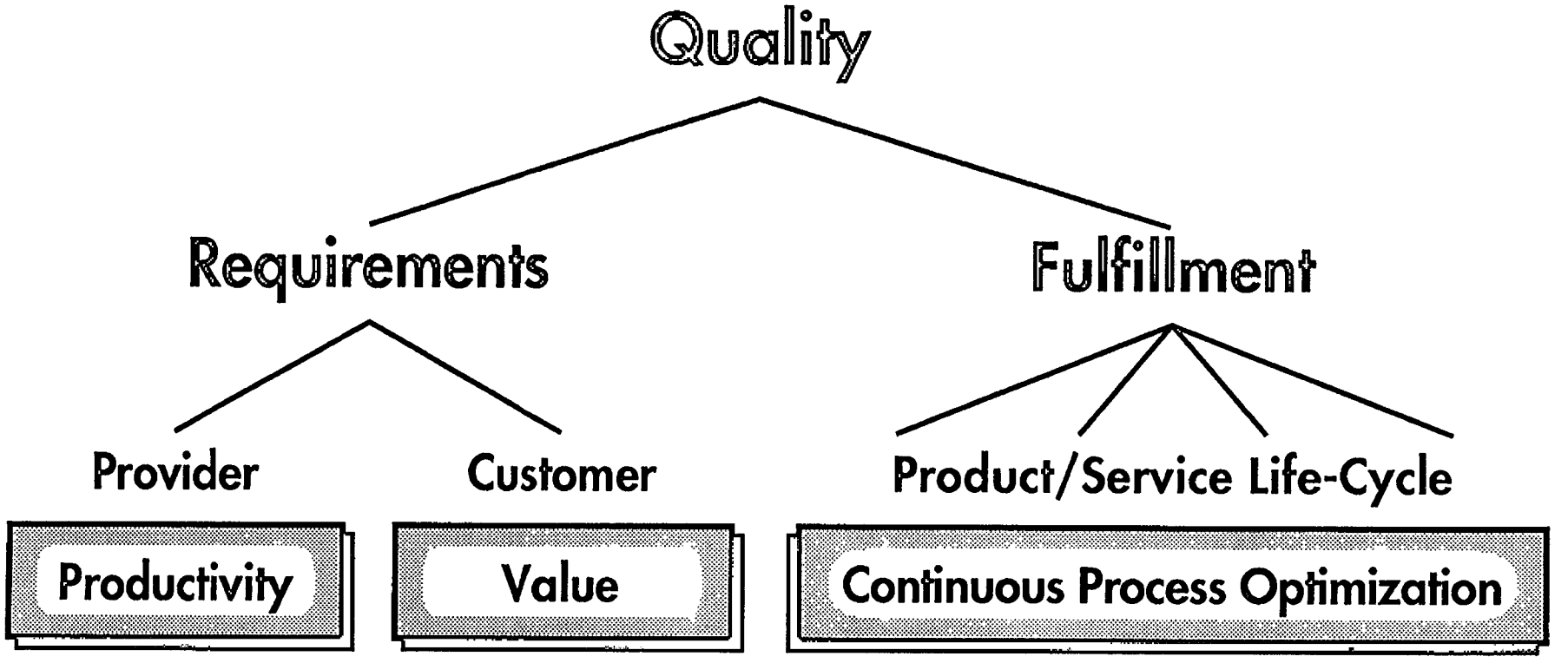
③ **to obtain the most efficient use of**

"

Webster

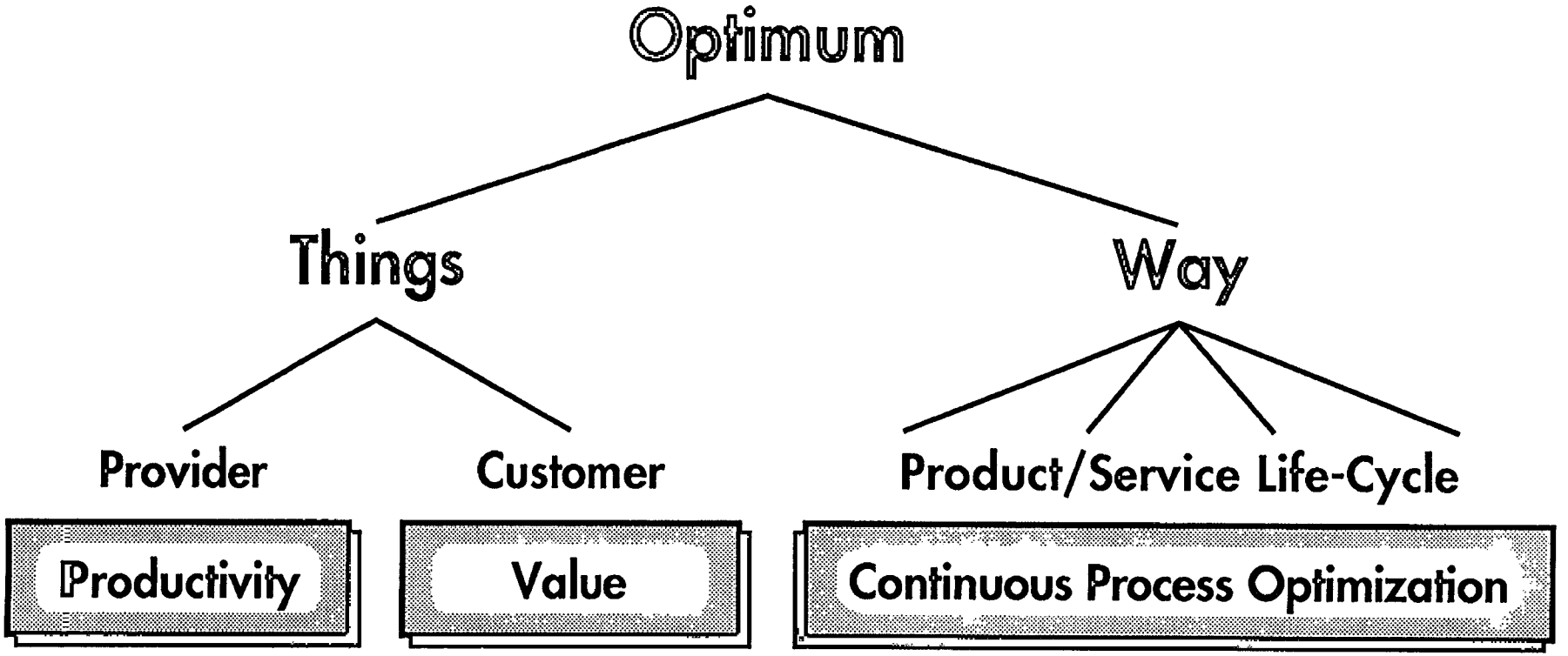
© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

A Quality Taxonomy



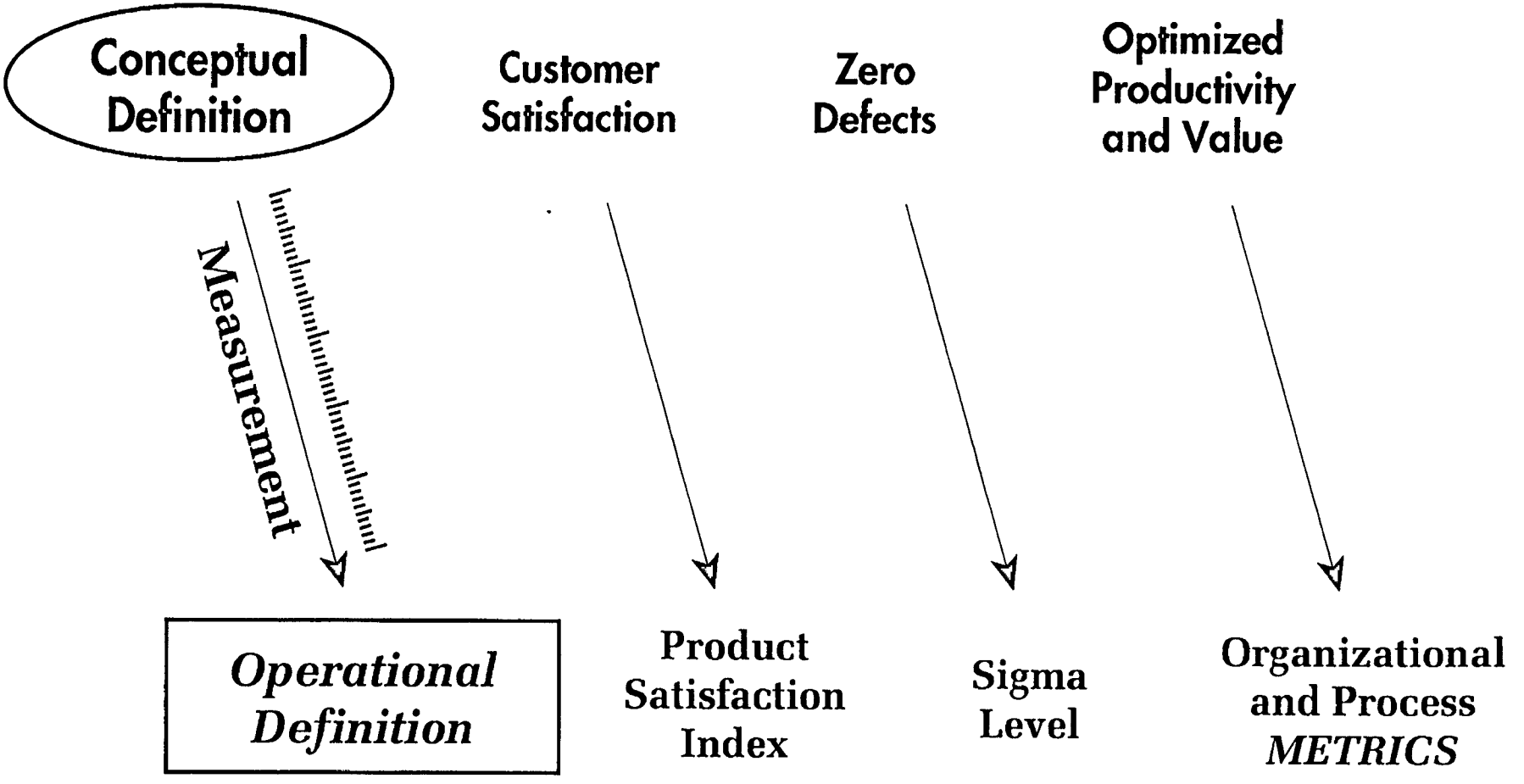
© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Optimum Quality



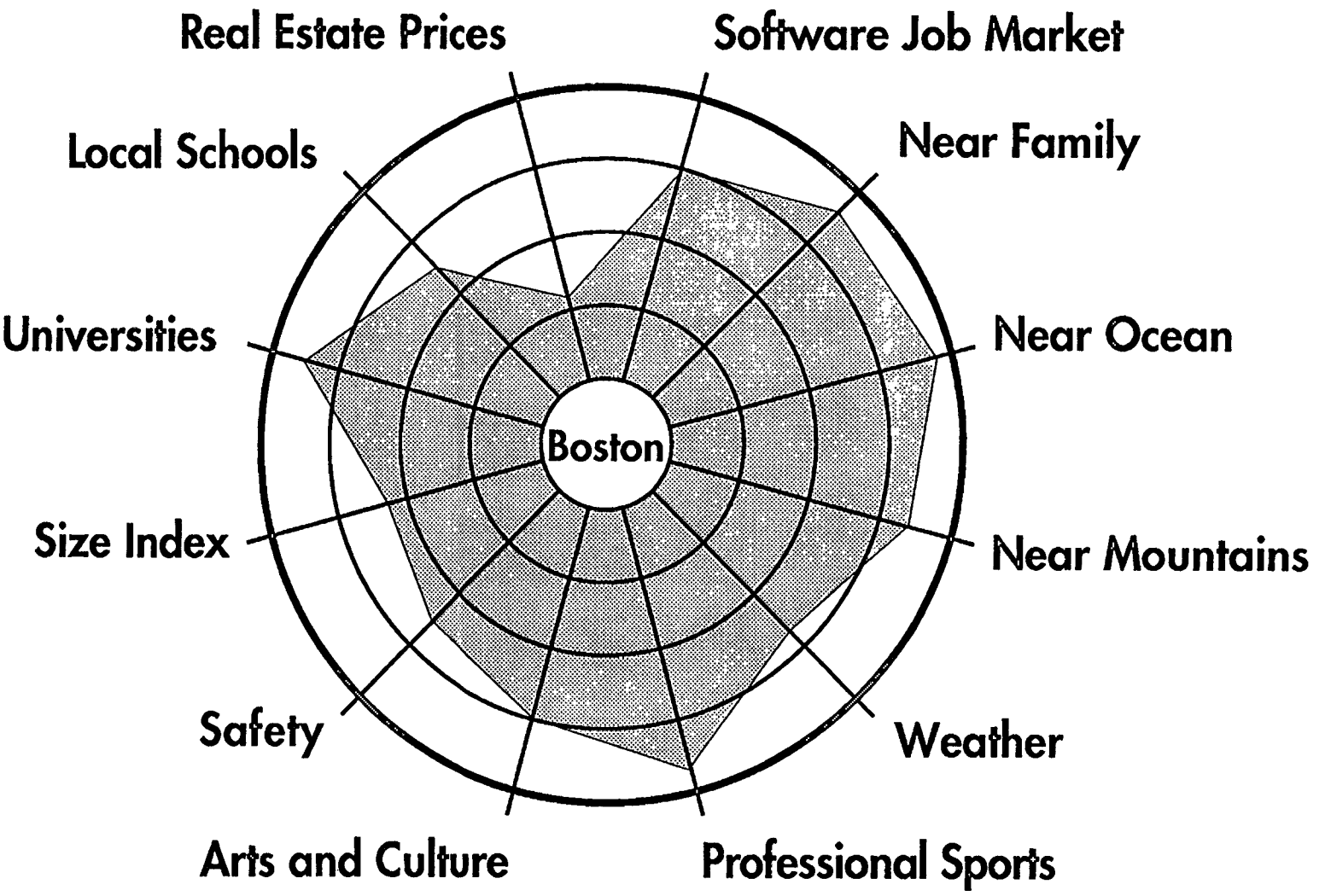
© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Operational Definitions



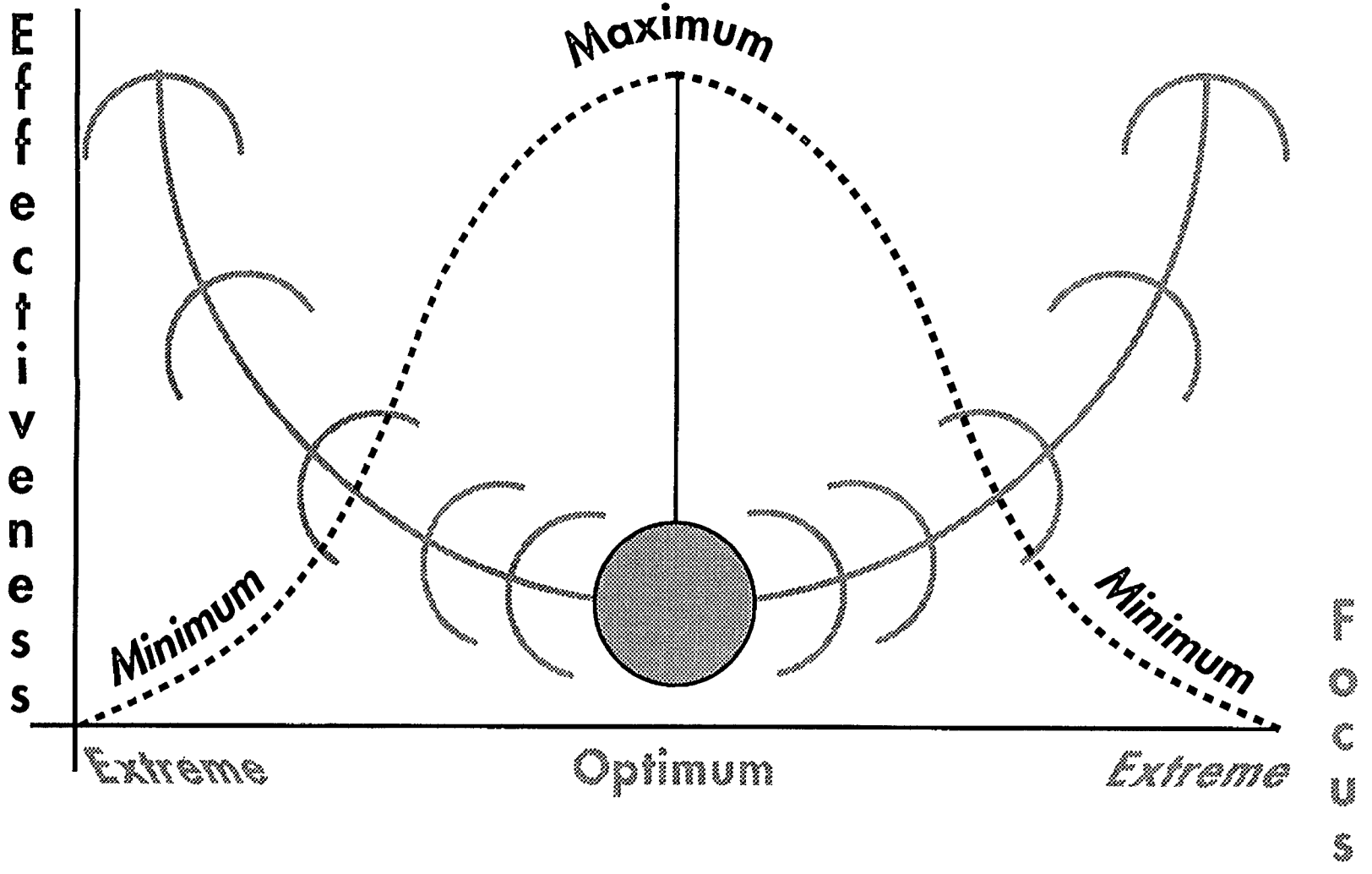
© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Selecting an Optimum City



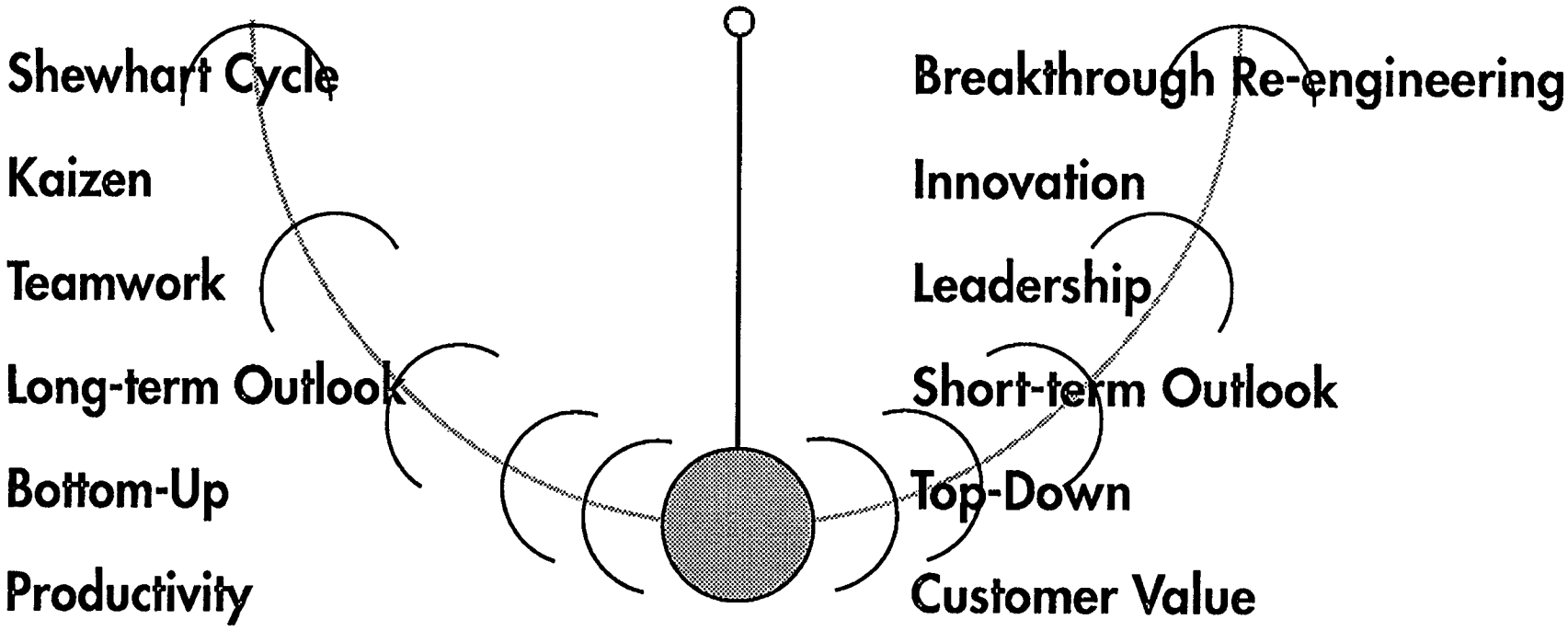
© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

The Optimum is Seldom at the Extremes



© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Even Quality Solutions Fail at the Extremes



Beware of single-focus solutions!

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

The Quality Deployment Dilemma

- ❑ Quality and Process
- ❑ Quality and People
- ❑ Three Dimensions of a Company
- ❑ Evolving Approaches to Quality
- ❑ The Quality Solution Spaghetti

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Evolving Approaches to Quality

① *QC* = ***Inspection***

② *QA* = ***Process Improvement***

③ *QO* = ***Total Optimization***

Focus on Culture

Organization

Process

Infrastructure

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Evolving Focus of Software Engineering

Tools:

CASE — Computer Aided Software Engineering

Integrated Development Environments

Object-Oriented Programming

Methods:

Structured Analysis

Information Modeling

Essential Systems Analysis

Object-Oriented Design

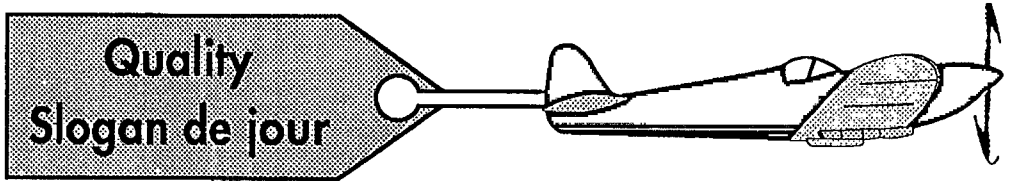
Process:

Waterfall, Spiral, Parallel, or Incremental

Software Engineering Institute (SEI)

Culture:

Top Down, Bottom Up, World Class, Slogan on the wall

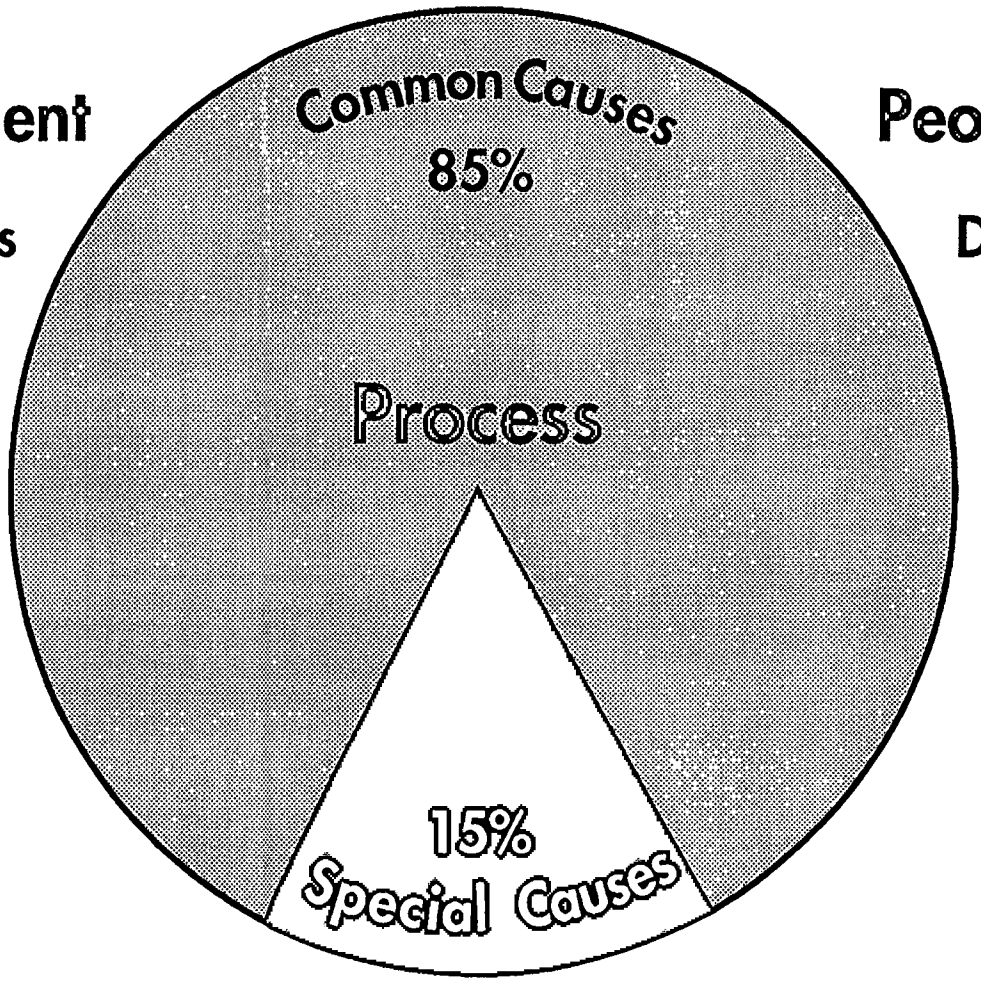


© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Juran/Deming Basic Quality Principle

Management

Owns the Process



People

Doing Their Best

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Quality is a PROCESS Issue

- ***Focus on Process—not problems***

  Continuous Process Optimization

- ***Managers Own the Process***
  (They Own the Resources)

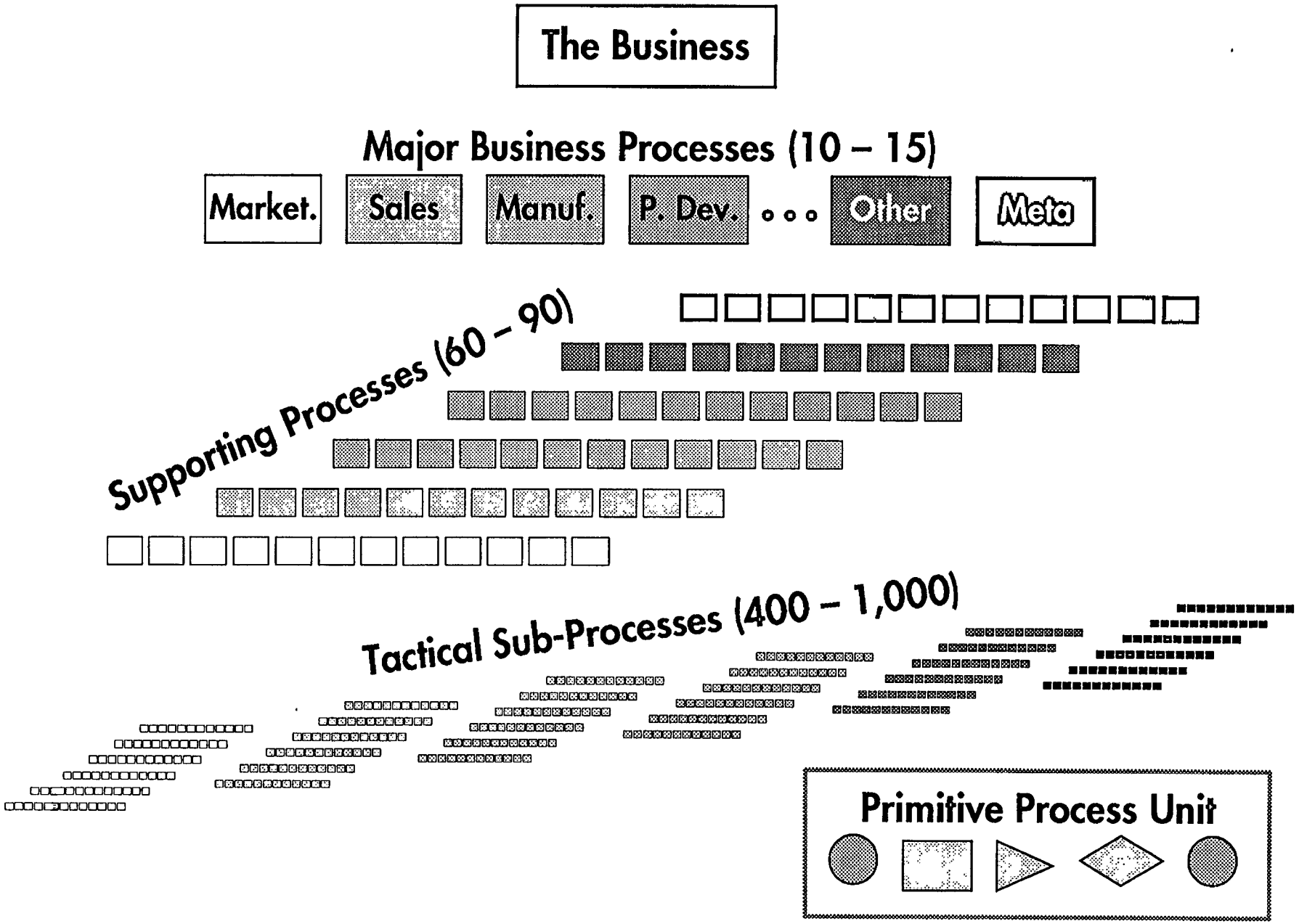
  Map Organizations ↔ Processes

- ***Use measurement to demonstrate productivity and value.***

- ***Everyone Empowered to Improve their Process***

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Quality Deployment: Process Hierarchy



© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Quality is a PEOPLE Issue

- ***Drive out Fear***
  - Safety and security
  - Reward the messenger
  - Exhortations and slogans are counterproductive
  - Trust required to effectively measure

- ***Unleash the power of Empowerment***
  - People doing a job often know best how to do it better
  - Access to timely Information
  - Support High Individual Quality

- ***Break down Barriers between people***
  - Product Teams
  - Process Improvement Teams
  - Executive teamwork

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Leaders are People

"

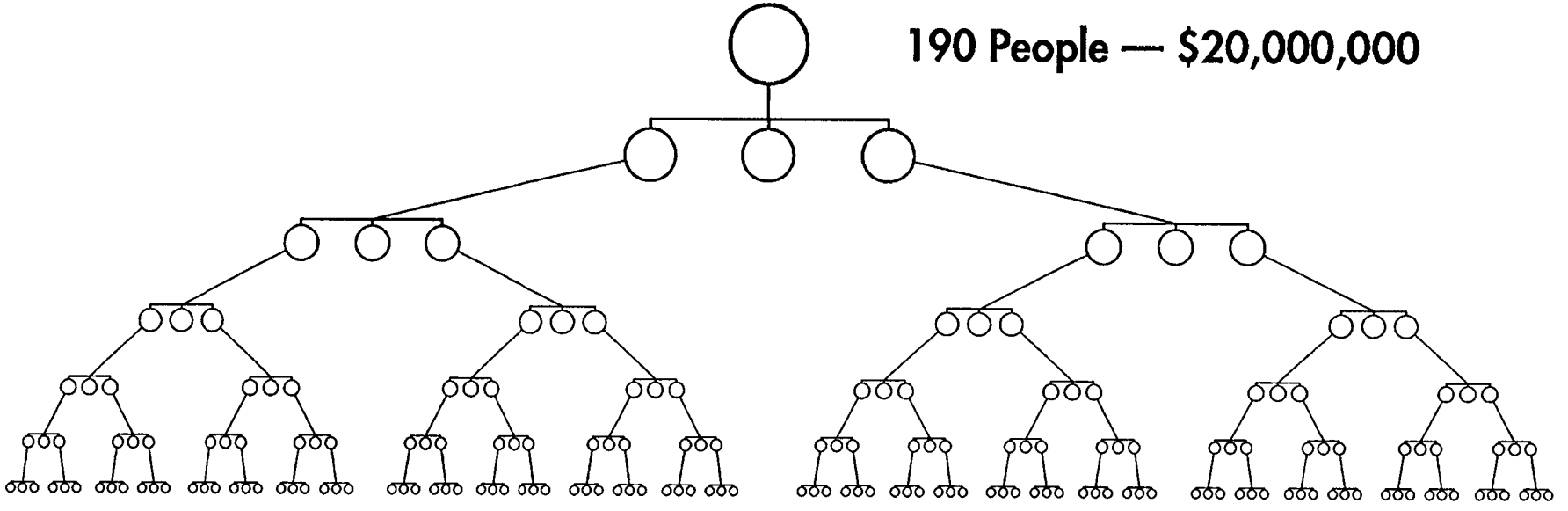
Support of top management is not sufficient. It is not enough that top management commit themselves for life to quality and productivity. They must know what it is that they are committed to—that is, what they must do. These obligations can not be delegated. Support is not enough: ...action is required.

"

Dr. W. Edwards Deming

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

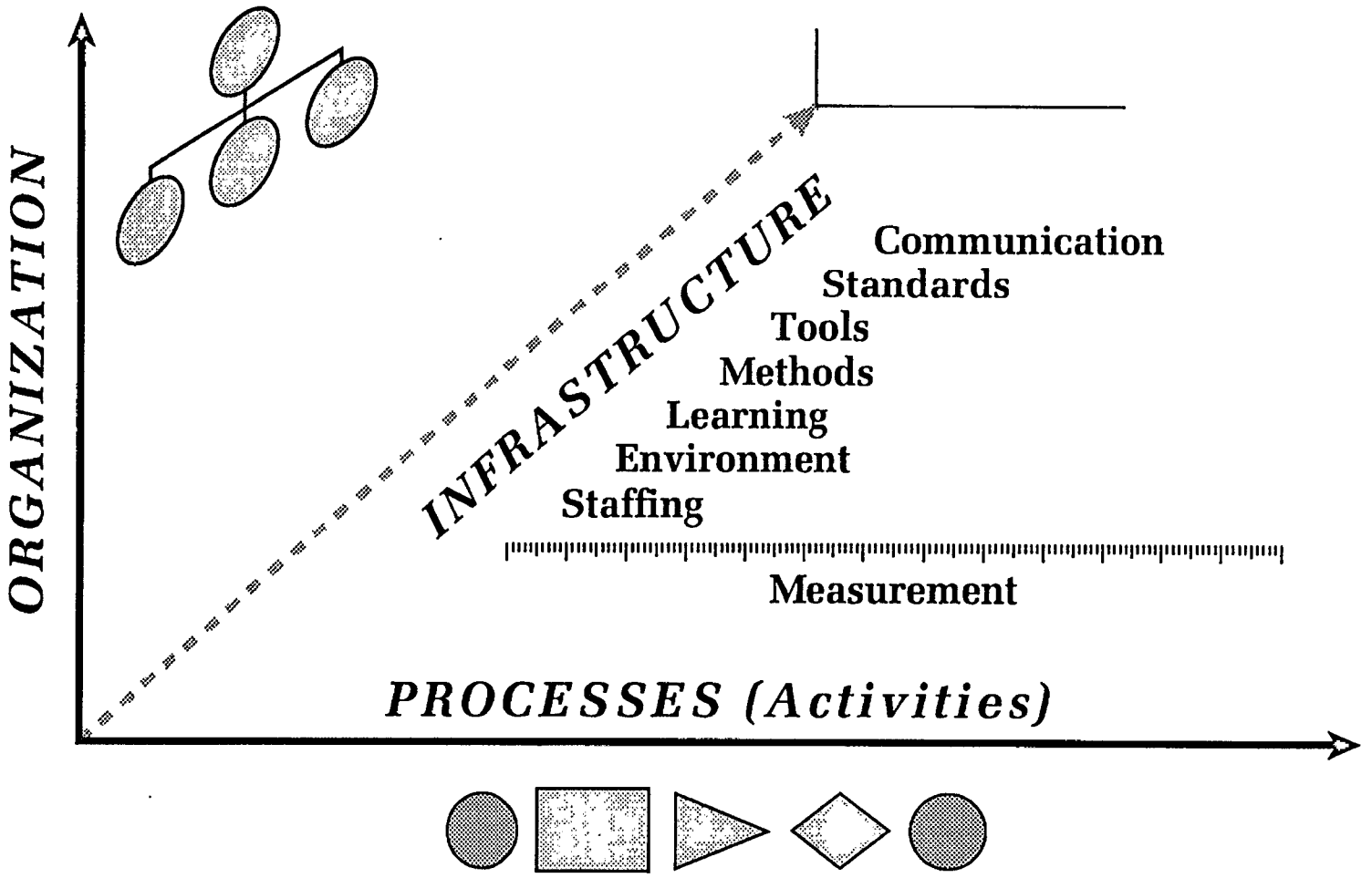
Quality Deployment: Organizational Hierarchy



Every manager and person in the company must believe in quality, understand quality, and consistently behave in a way that acts to improves quality.

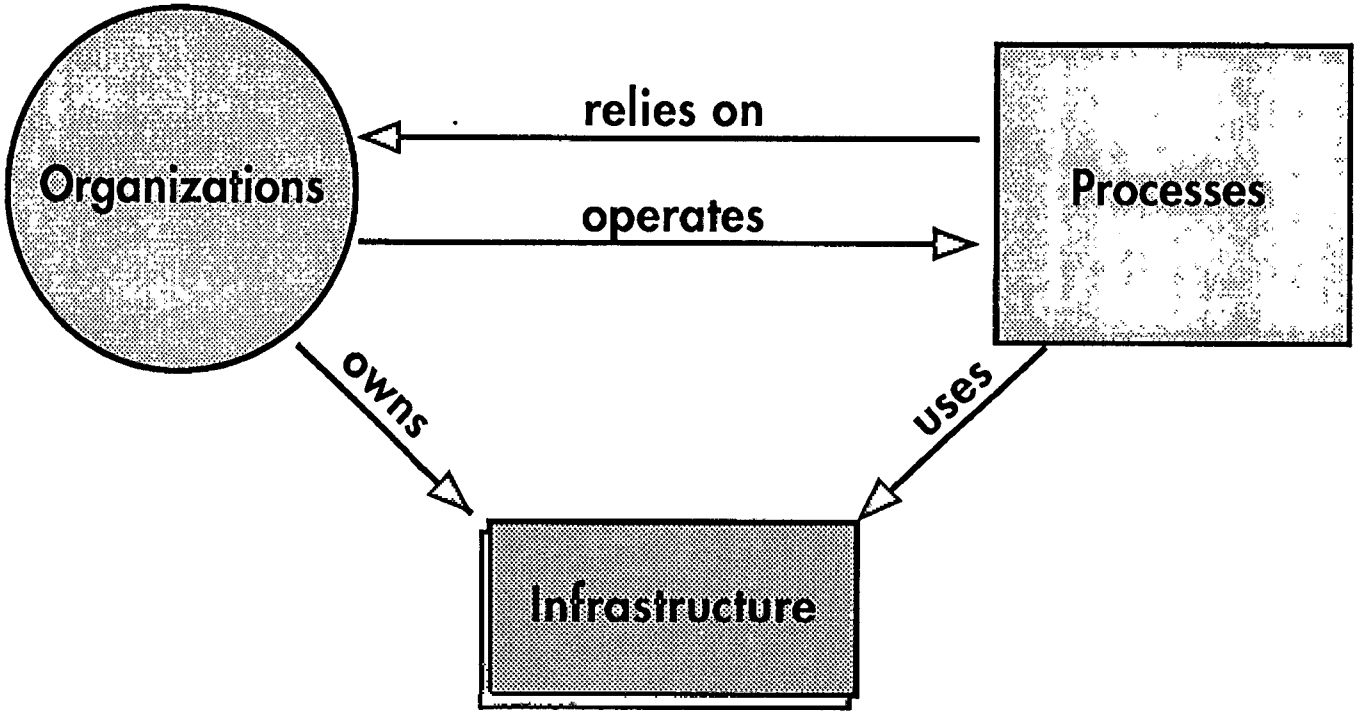
© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Three Dimensions of a Company



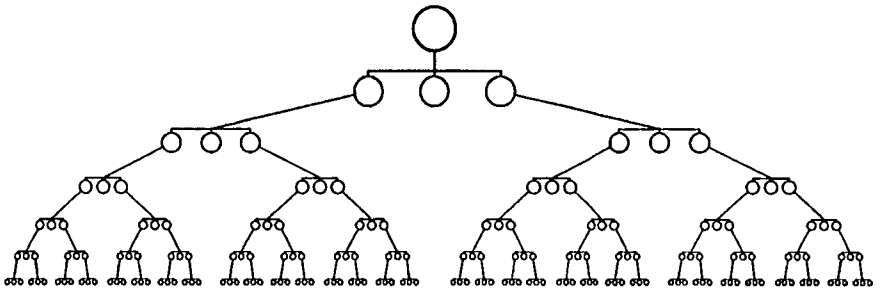
© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Relationships Between the Three Dimensions



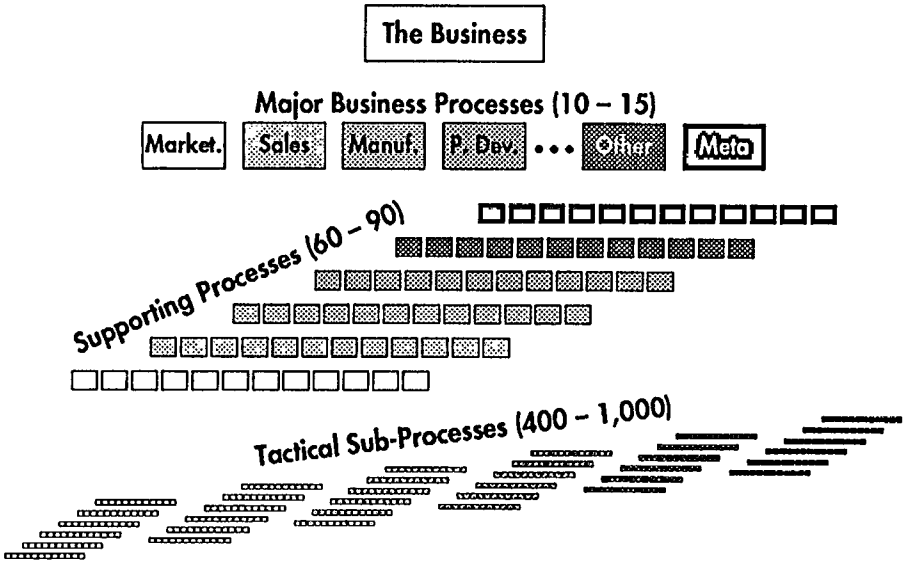
© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Corporate Complexity & Chaos



Companies are Dynamical Systems

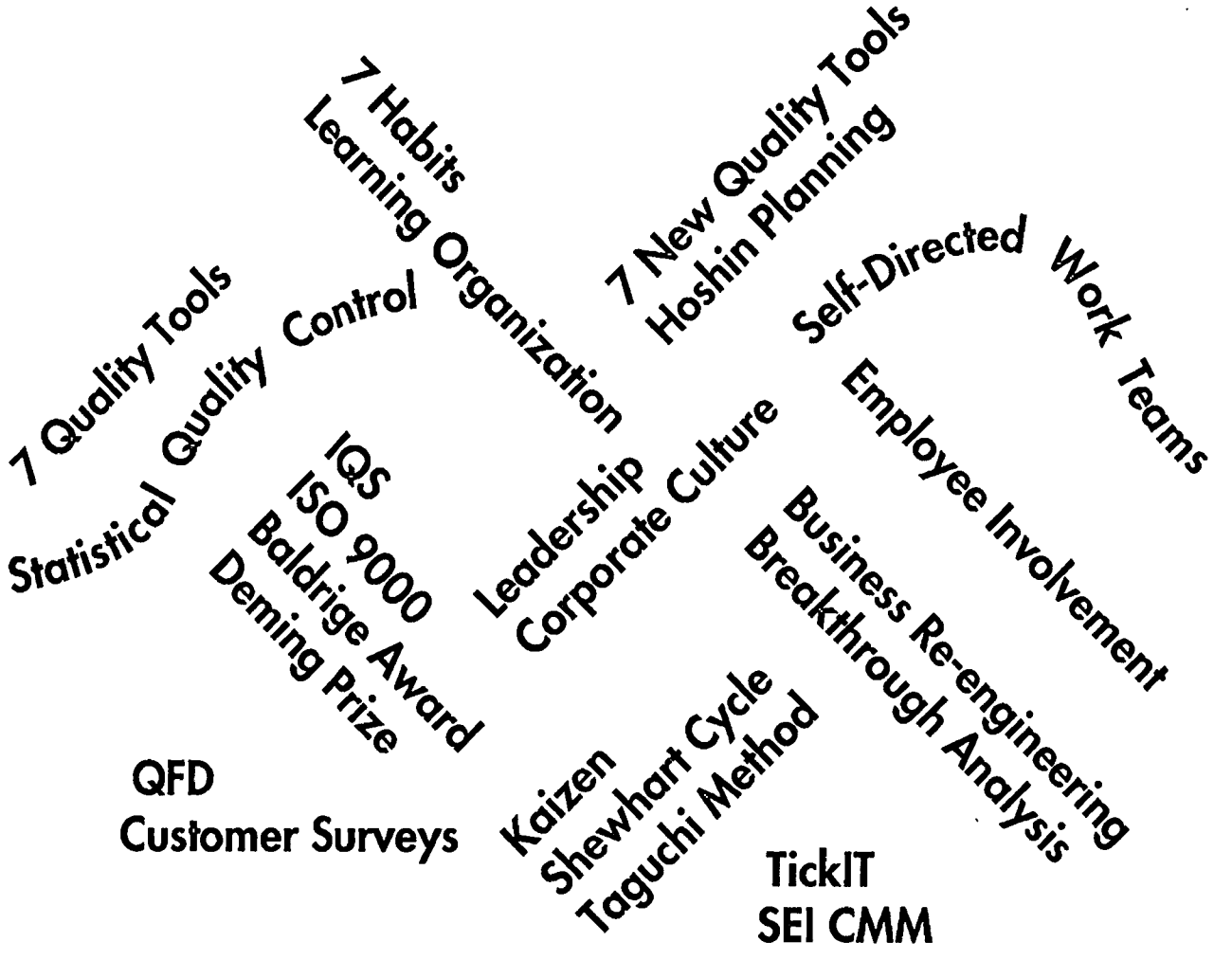
The ORG ↔ PROCESS Mapping is M x N



Corporate Complexity = Organizational Complexity X Process Complexity

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Quality Solutions Spaghetti



© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Introduction to TQO

- The TQO Methodology
- Models of TQO
- The Fractal Nature of TQO
  - The Power of Iteration
  - Fractal Deployment Strategy
  - Optimization Attractors
- TQO: an Organizing Principle

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

TQO

is a breakthrough,

fractal-based, measurement-intensive

methodology

for deploying and optimizing quality

throughout a company.

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Benefits of TQO

- **TQO is a TOTAL Quality Solution:**
  - Deals with Corporate Complexity and Culture
  - Integrates Organization, Process, and Infrastructure
  - TQO's Power comes from Iteration, not Complexity
- **TQO is the FRAMEWORK for Total Quality**
  - TQO Organizes all other Quality Solutions
  - TQO is a Methodology
  - TQO has a Graphical Model
- **You can DEPLOY TQO Companywide:**
  - Scales to all Organizations and Processes
  - Deployment Top-Down and Bottom-Up
- **TQO Works:**

Optimization	Breakthroughs
Customizable	Degapping

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Quality Programs Must be Customized

- **Companies & organizations differ greatly**

  Like dynamical systems, they have a sensitive dependence on initial conditions.

- **Canned quality solutions don't work**

  Consultants and benchmarks give ideas—not results.

  ***Degapping* Cycles Do Work!**

- **Quality programs must be customized**

  Internal quality leaders must adjust the methodology to the culture and practices of the company.

- **TQO is *Customizable***

  TQO requires some axiomatic elements

  but only recommends many others

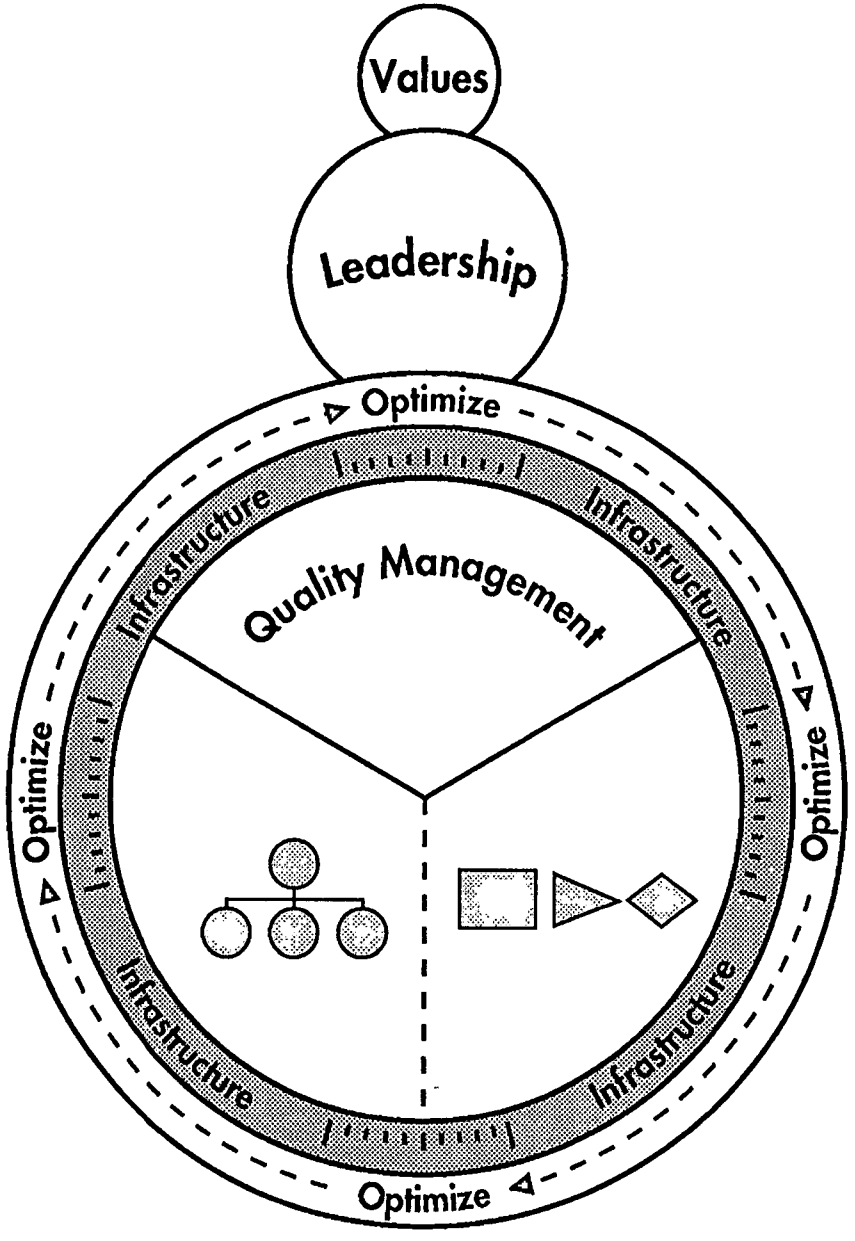
© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

TQO Methodology

- **INITIATE TQO:**
  - Start Somewhere
  - Study Quality & Define Values
- **LEADERSHIP CYCLES:**
  - Deploy Quality in Organizations & Processes
- **OPTIMIZATION CYCLES:**
  - Organizations & Processes
  - Optimize Infrastructure
  - Seek Breakthroughs
  - Leverage Results

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

The Axiomatic TQO Model



© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

INITIATE TQO

- ❑ ***START*** **Somewhere**
  Organization / Process / Project
  Top / Bottom / Middle
- ❑ ***LEARN*** **about Quality**
- ❑ ***DEFINE*** **Values**
- ❑ ***LEAD*** **the Quality Effort**
  (Become a Quality Fractal)

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Leadership Cycles

- Define the *Mission* / *Vision* of Your Organization / Process / Project
- Define a *Strategy* to Achieve the *Mission* / *Vision*
- *MAP* Organizations ↔ Processes
- Allocate or Obtain *RESOURCES* for Quality Staff Assignments / Process Improvement Teams
- *Begin OPTIMIZATION CYCLES* Organization / Process
- *ITERATE*

Parallel DEPLOYMENT Top↘Down / Bottom↗Up

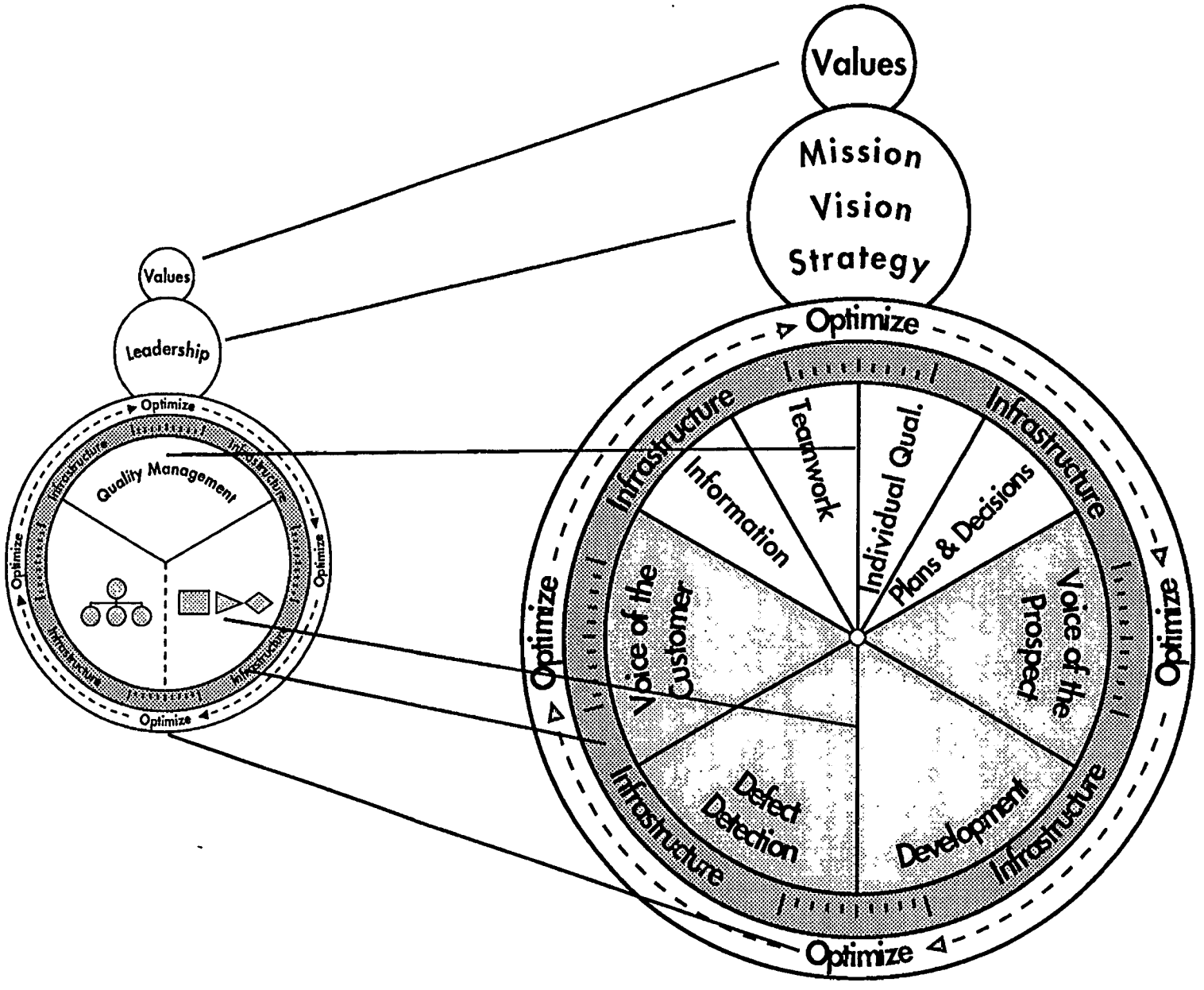
© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Process Optimization Cycle

PLAN:	① *Adopt / Adapt / Extend* higher-level Process Values, Mission, Vision, and Strategy ② *Observe, Understand* and *Document* the current **Process ↔ Infrastructure** ③ *Assess* and *Benchmark* Process Capability ④ *Identify* and *Prioritize* Changes ⑤ Define *Measurement* of Results ⑥ *Plan* and *Prepare* the Changes
DO:	⑦ *Install and Operate* new **Process ↔ Infrastructure**
CHECK:	⑧ *Evaluate* Quality Results
ACT:	⑨ *Leverage* to other parts of the company: Processes, Projects, and Organizations ⑩ *Iterate* Process Optimization Cycle

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

TQO Process Model



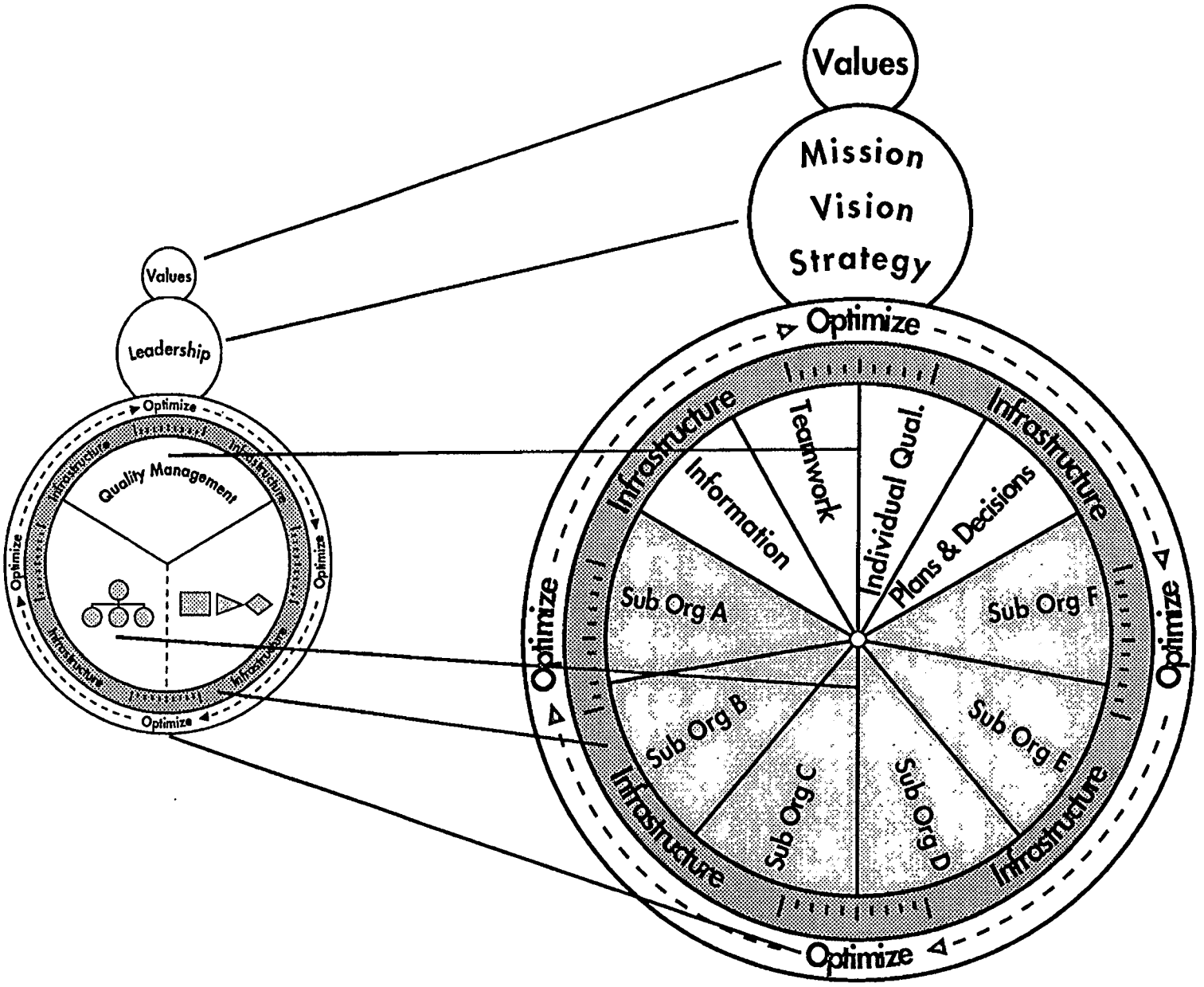
© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Organization Optimization Cycle

Plan:	❶ *Adopt / Adapt / Extend* higher-level Organization Values, Mission, Vision, and Strategy ❷ *Observe, Understand* and *Document* the current management and quality practices ❸ *Assess / Benchmark* Organization's Quality Capability ❹ *Identify* and *Prioritize* changes ❺ Define *Measurement* of Results ❻ *Plan* and *Prepare* the changes
Do:	❼ *Install and Practice* new quality management
Check:	❽ *Evaluate* Quality Results
Act:	❾ *Leverage* to other parts of the company: lower-level (and higher-level) organizations, processes, and projects ❿ *Iterate* Organization Optimization Cycle

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

TQO Organization Model



© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

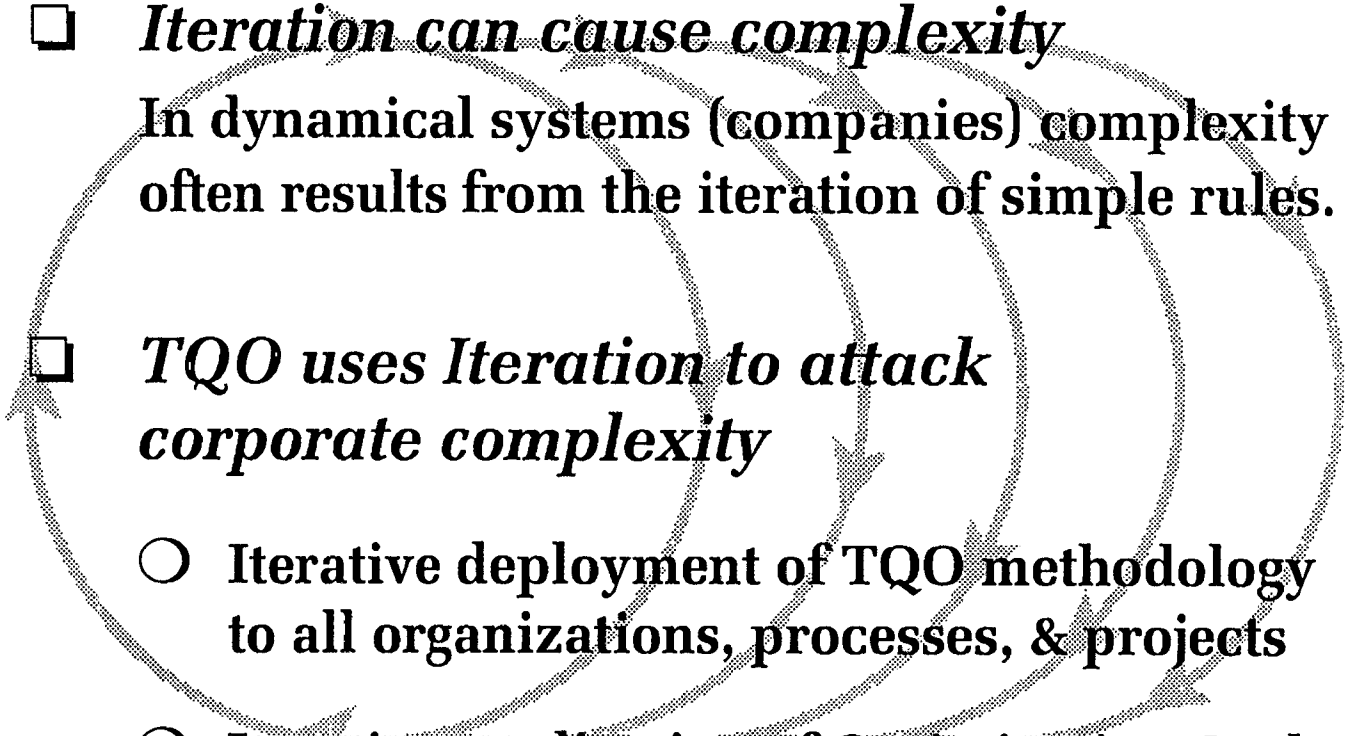
The Key Role of Iteration

- ***Iteration can cause complexity***

  In dynamical systems (companies) complexity often results from the iteration of simple rules.

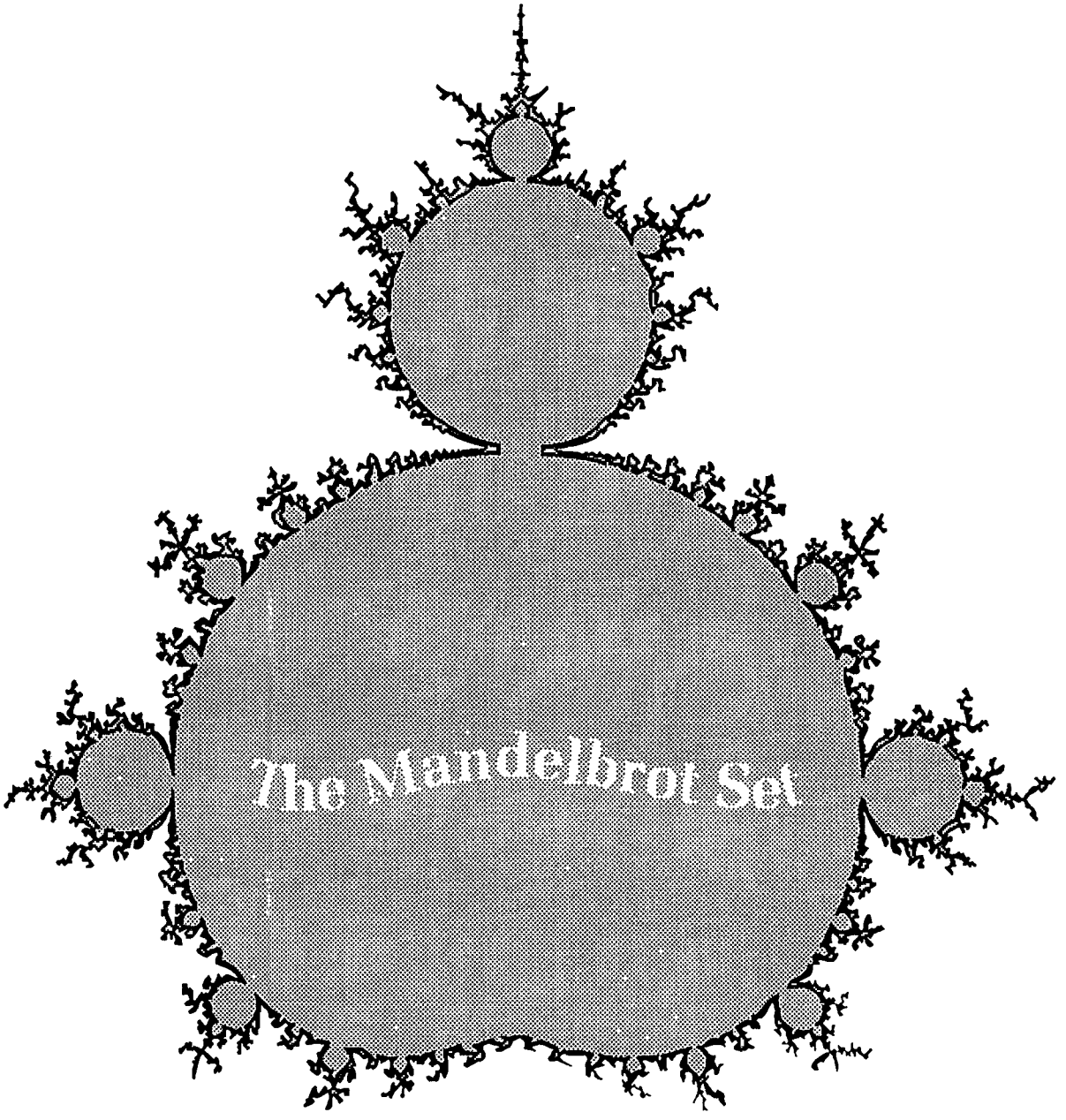
- ***TQO uses Iteration to attack corporate complexity***

  - **Iterative deployment of TQO methodology to all organizations, processes, & projects**
  - **Iterative application of Optimization Cycles**



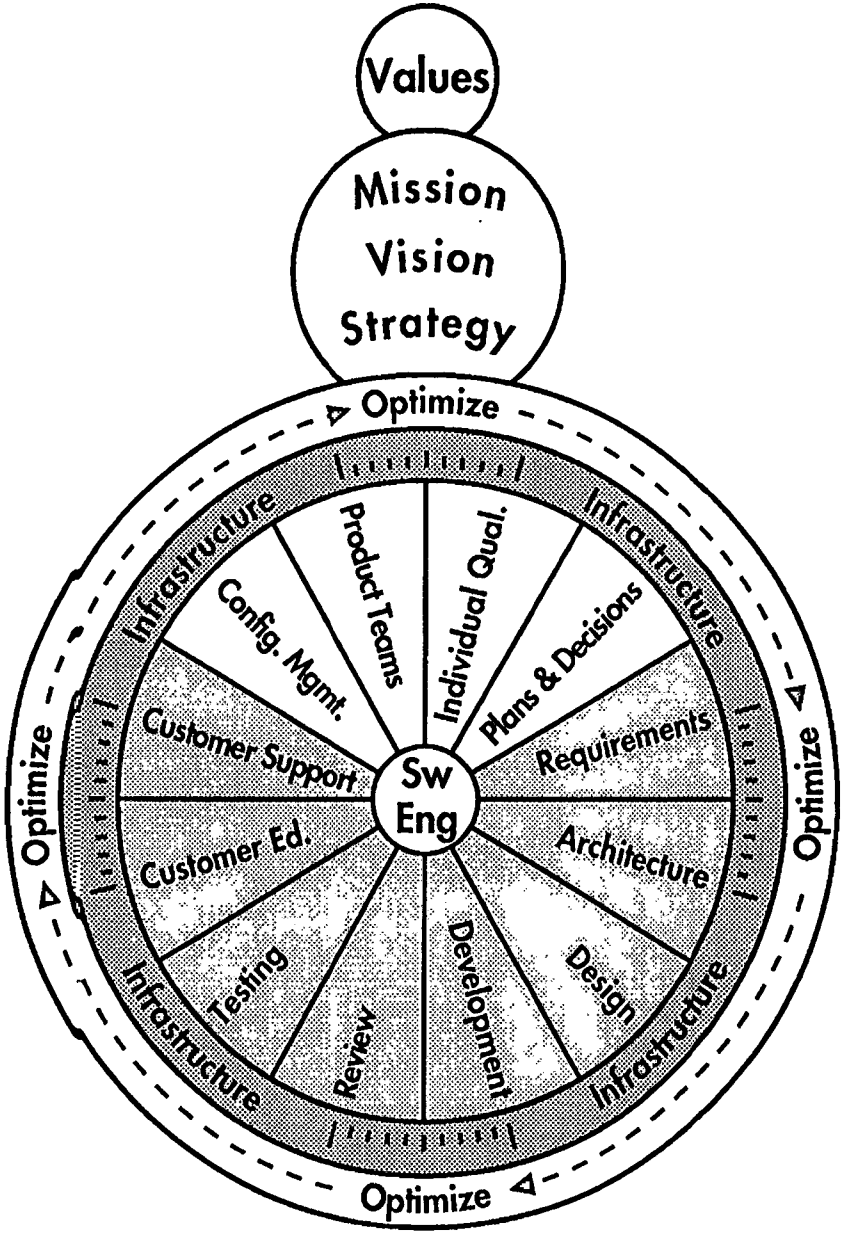
© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

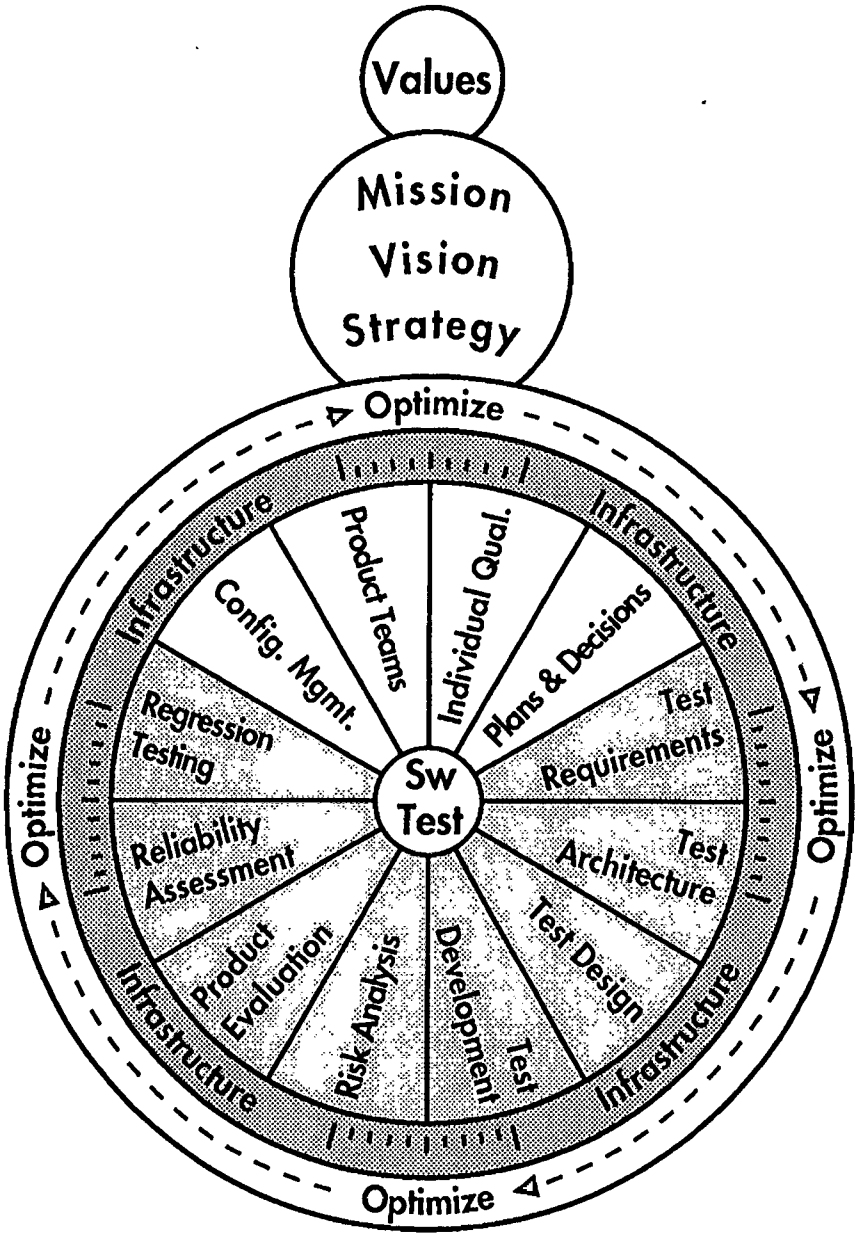
Fractals: Scaling & Self-Similarity



© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

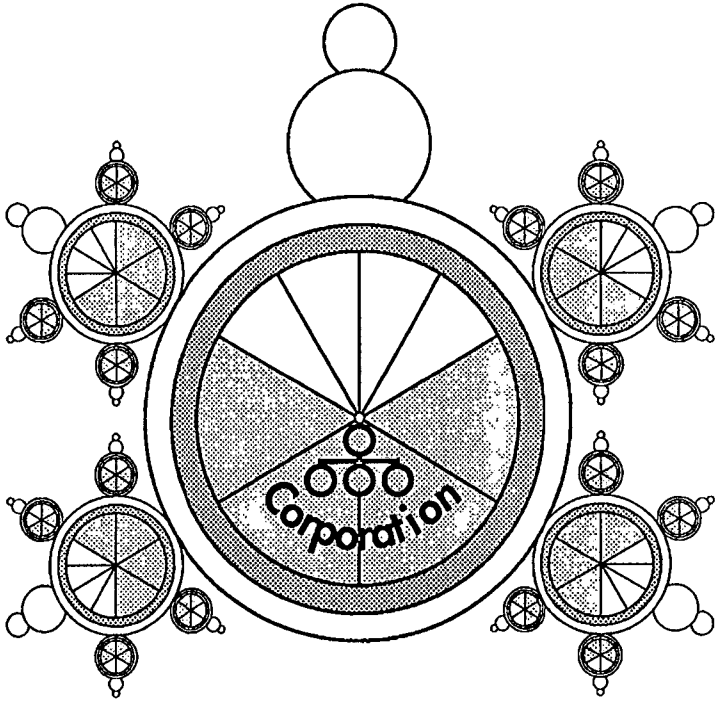
Deploying TQO to Software Engineering & Testing

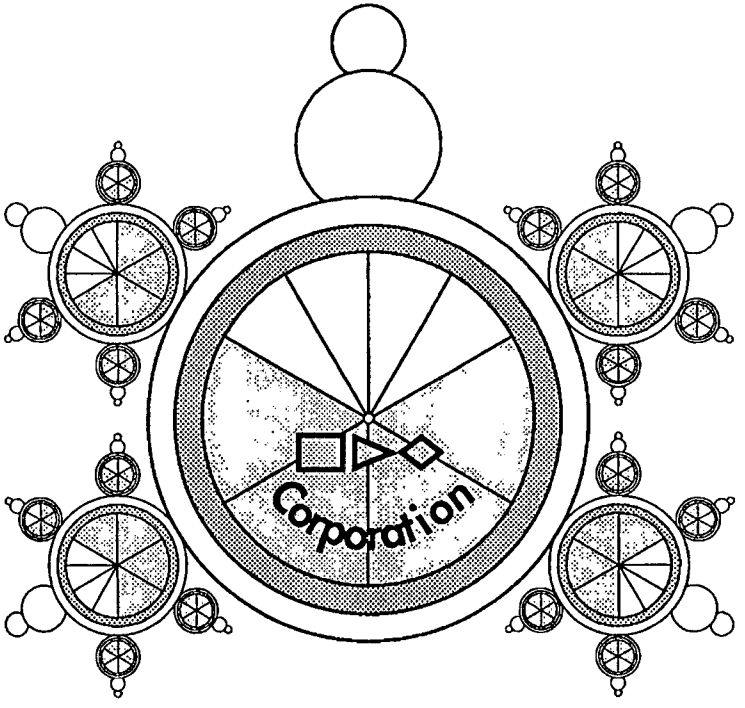




© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

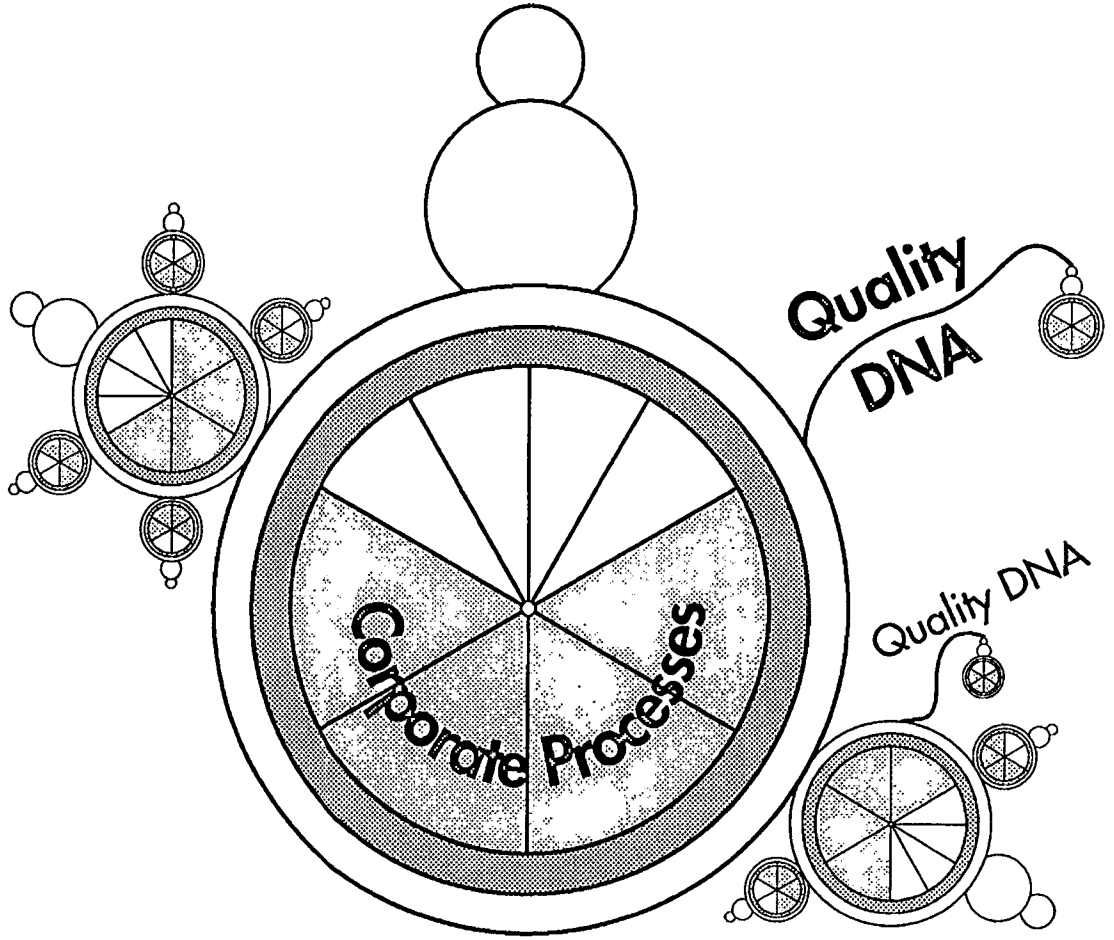
Deploying TQO: Ideal-World





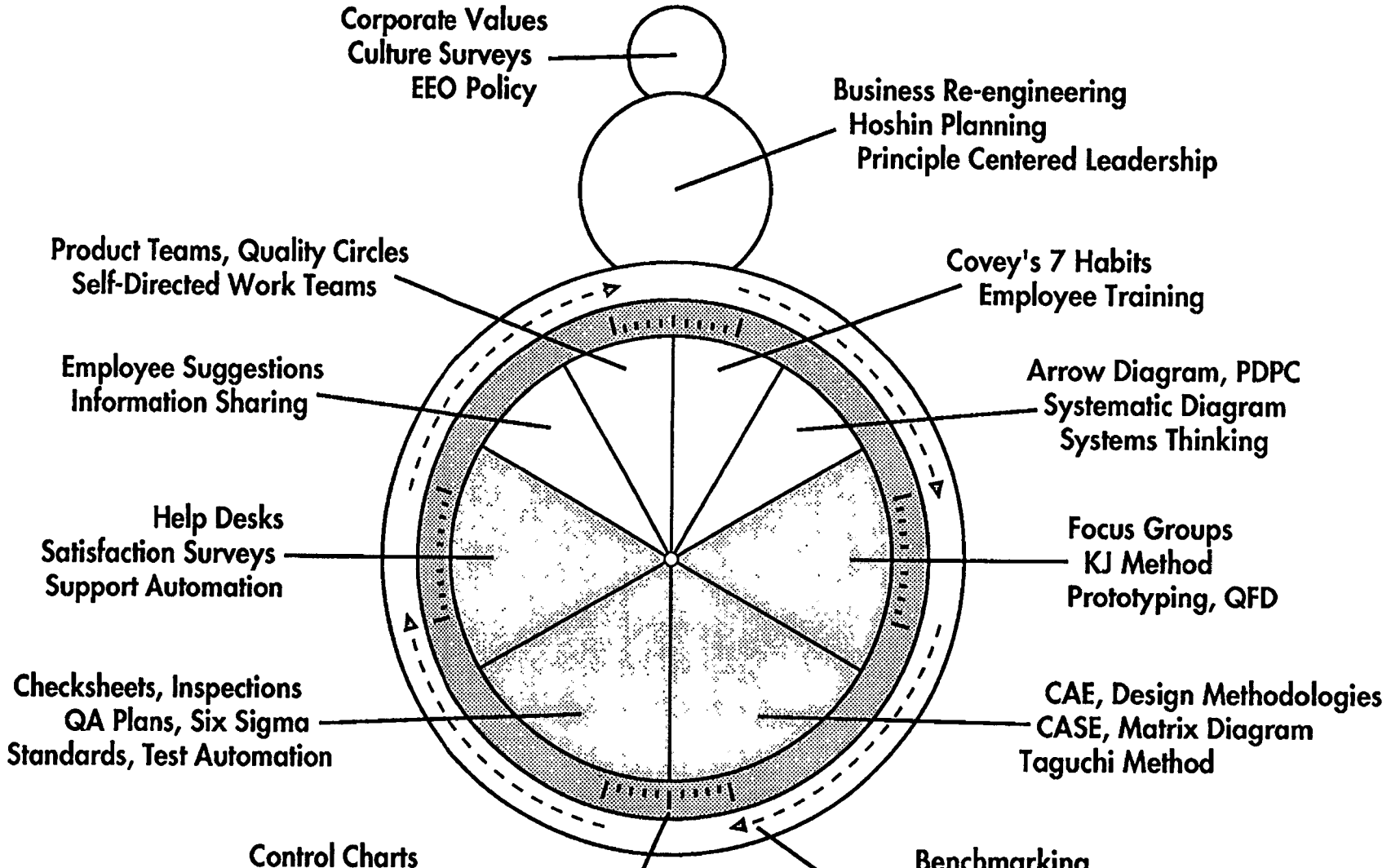
© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Deploying TQO: Real-World Quality Fractals



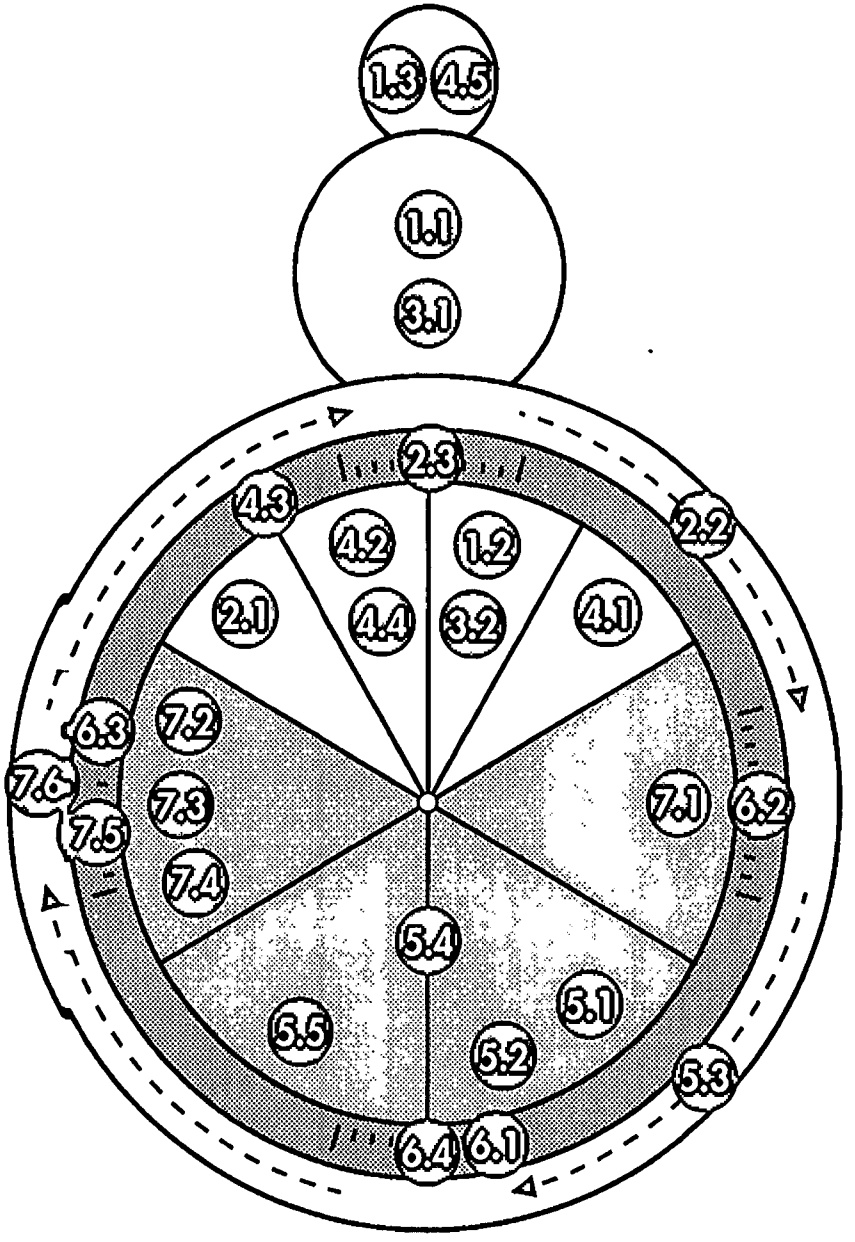
© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

TQO as an Organizing Principle



© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

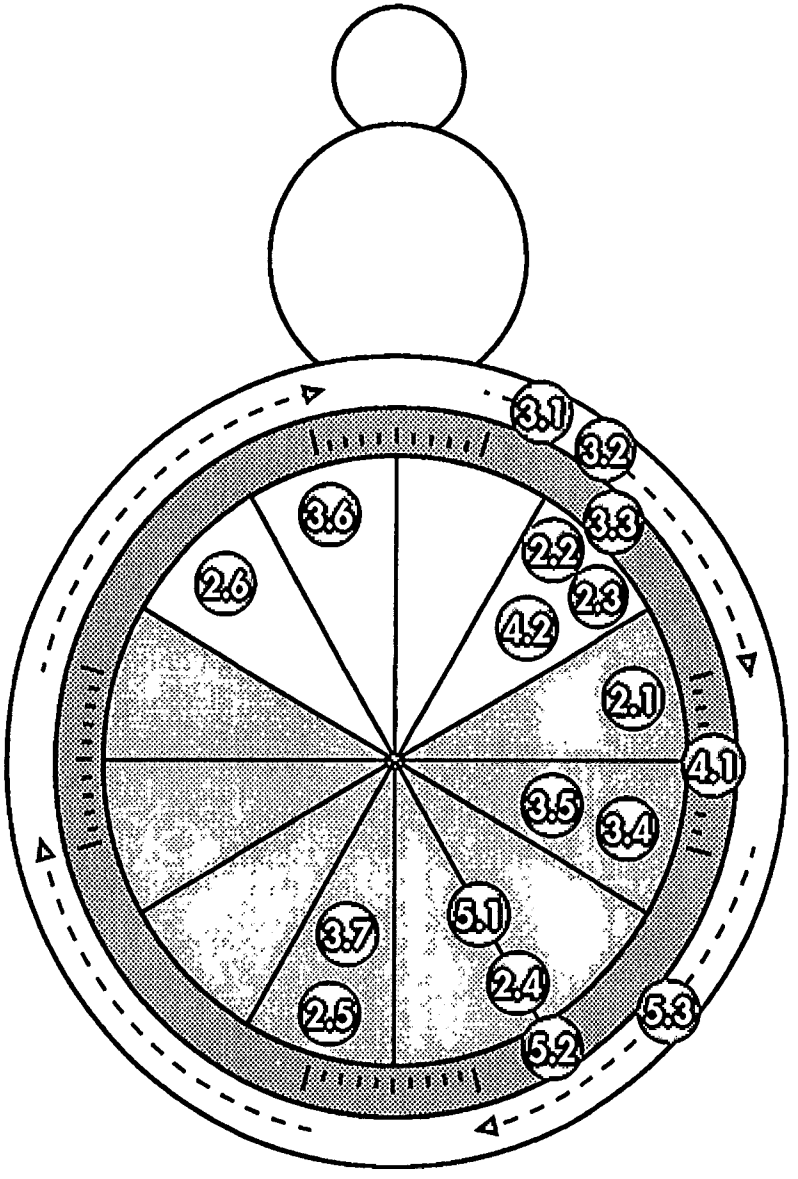
TQO ⟷ Baldrige Criteria



1.1	Senior Executive Leadership
1.2	Management for Quality
1.3	Public Responsibility & Corporate Citizenship
2.1	Scope & Management of ... Data
2.2	Competitive Comparisons & Benchmarks
2.3	Analysis and Uses of Company-Level Data
3.1	Strategic Quality ... Planning Process
3.2	Quality & Performance Plans
4.1	Human Resource Planning & Management
4.2	Employee Involvement
4.3	Employee Education & Training
4.4	Employee Performance & Recognition
4.5	Employee Well-Being & Satisfaction
5.1	Design & Intro of Quality Products & Services
5.2	Product & Service Production & Delivery
5.3	Business Processes & Supporting Processes
5.4	Supplier Quality
5.5	Quality Assessment
6.1	Product & Service Quality Results
6.2	Company Operational Results
6.3	Business Process & Support Service Results
6.4	Supplier Quality Results
7.1	Customer Expectations: Current and Future
7.2	Customer Relationship Management
7.3	Commitment to Customers
7.4	Customer Satisfaction Determination
7.5	Customer Satisfaction Results
7.6	Customer Satisfaction Comparison

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

TQO ⟷ SEI CMM



Optimized (5)
- 3–Process change management
- 2–Technology innovation
- 1–Defect prevention

Managed (4)
- 2–Quality management
- 1–Process measurement and analysis

Defined (3)
- 7–Peer reviews
- 6–Intergroup coordination
- 5–Software product engineering
- 4–Integrated software management
- 3–Training program
- 2–Organization process definition
- 1–Organization process focus

Repeatable (2)
- 6–Software configuration mgmt.
- 5–Software quality assurance
- 4–Software subcontract management
- 3–Software project tracking & oversight
- 2–Software project planning
- 1–Requirements management

Initial (1)

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Benefits of TQO

- **TQO is a TOTAL Quality Solution:**
  - Deals with Corporate Complexity and Culture
  - Integrates Organization, Process, and Infrastructure
  - TQO's Power comes from Iteration, not Complexity
- **TQO is the FRAMEWORK for Total Quality**
  - TQO Organizes all other Quality Solutions
  - TQO is a Methodology
  - TQO has a Graphical Model
- **You can DEPLOY TQO Companywide:**
  - Scales to all Organizations and Processes
  - Deployment Top-Down and Bottom-Up
- **TQO Works:**
  - Optimization
  - Customizable
  - Breakthroughs
  - Degapping

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Introducing The TQO Measurement Paradigm

- ❑ The Measurement Imperative
- ❑ Features of TQO Measurement
- ❑ The Measurement Process
- ❑ Measures, Metrics, Meters
- ❑ The Fractal Nature of Measurement
- ❑ Measurement and Optimization
- ❑ Controlling Variation
- ❑ An Example of Breakthrough Optimization

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

The Measurement Imperative

"You can't control what you can't measure."

Tom DeMarco

"Only through measurement
can you know what's optimum."

Bill Silver

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Most Measurement Programs Fail

1. They lacked a quality culture that provides a positive environment for measurement.
2. They did not focus on the needs of critical internal customers (especially practitioners).
3. They failed to give feedback to the people providing the data about how the measures are being used and the benefits of measurement.
4. They lacked basic measurement tools and expertise—the adoption rate of available tools is very slow.
5. Measurement fails at the extremes:

measure too much	measure too little
top down focus	bottom up focus

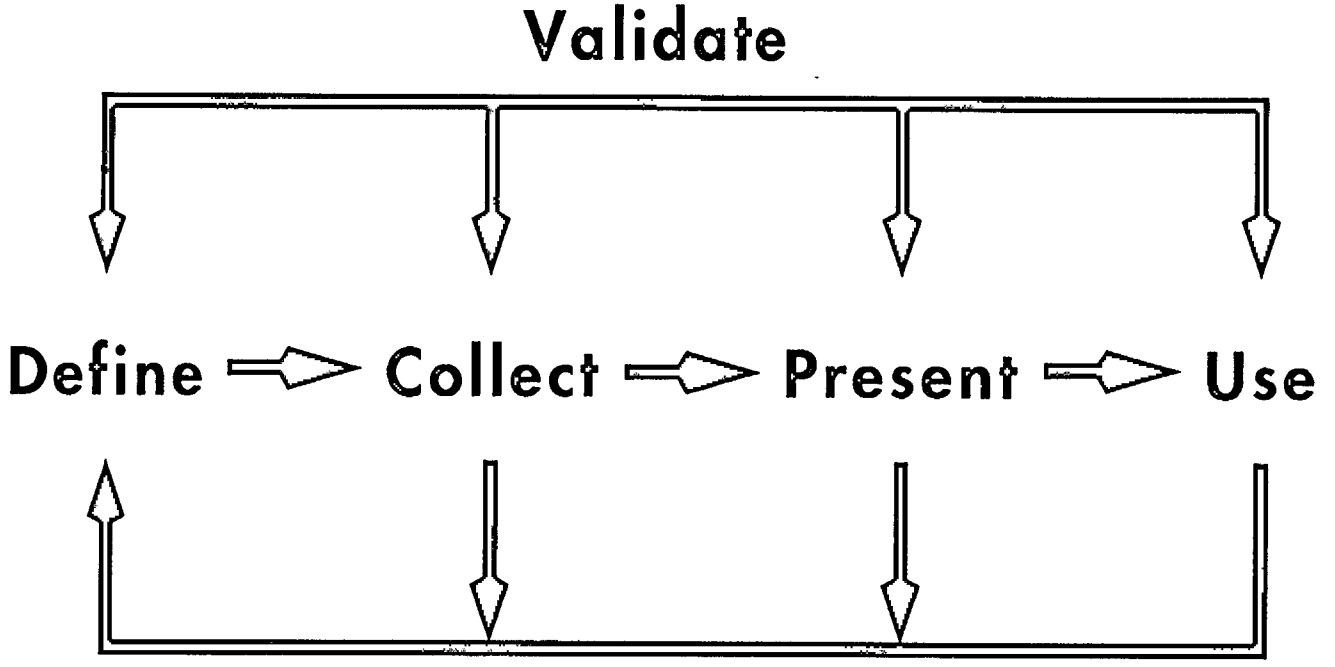
© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Features of TQO Measurement

- **Supports TQO methodology:**
  - Works top down & bottom up
  - Maps into TQO models
  - Maps into TQO assessment diagrams
- **Supports Optimization:**
  - Balanced organization & process metrics
- **Based on sound theory:**
  - Measurement process
  - Measures, Metrics, Meters
  - Measures from the 3 corporate dimensions

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

The Measurement Process



- Each step has feedback loop
- Validation is a constant part of the process

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Users / Uses of Measurement

Real People

- *Line Managers*
- *Project Managers*
- *Practitioners*
- *Process Improvement Teams*

Real Problems

- *Planning*
- *Control:*
  Financial
  Process
  Project
- *Analysis:*
  Comparisons
  Financial
  Risk Assessment
  Tradeoffs (Optimization)

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Presenting Measurement

- ***Classic Quality Tools***
  Histograms (Pareto diagrams)
  Scatter diagrams
  Run charts and control charts
- ***On-line, Operational Information***
  Available for people to do their jobs
- ***Quick Reports***
  Analyze data
  Answer questions
  Understand **new questions**
- ***Statistical Packages***
- ***Desktop Graphics***
  Kiviat Diagrams, etc.

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Collection: Things to Measure

Organization	**People** ○ Employees ○ Customer ○ Supplier **Capital Assets** ○ $$$$$
Process	**Activities / Operations** **Work Products (Input / Output)** ○ Expense Report ○ Purchased Part ○ Specification ○ Code
Infrastructure	**Tools** ○ Lathe ○ Compiler **Environment** ○ Office space

soft measures are hard

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Measurement Collection: Rules of Thumb

- ❑ **Build collection into everyday work processes**
- ❑ **Automate**
- ❑ **Get professional help for surveys**
- ❑ **Three basic *software* measurement tools:**
  1. **defect tracking**
  2. **configuration management**
  3. **time recording**

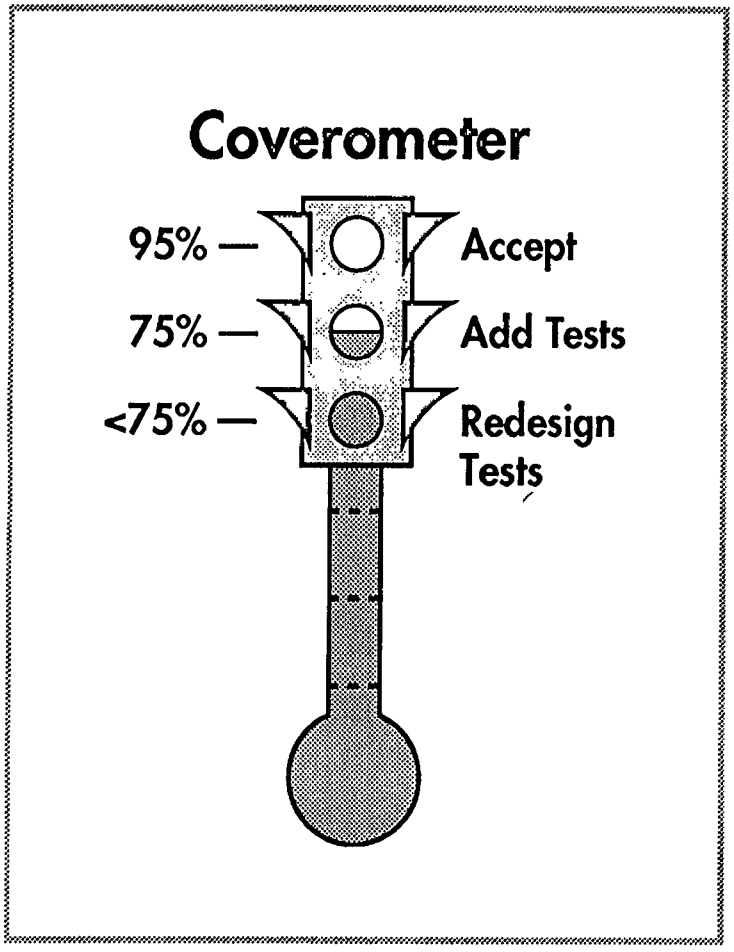
© 1993 ProQual, Inc. (508) 359-7273 & YESSI, Inc. (916) 944-1032

Defining Measurements

3 Levels of Measurements	Examples
① *Measure:* a dimension, attribute or amount of something	# lines of code in a module customer satisfaction response date and time of a failure revenue
② *Metric:* a primitive or computed set of measures used for comparison or prediction	asset turnover ratio function points module complexity density product satisfaction index productivity
③ *Meter:* a metric (or measure) that has been calibrated to help managers and practitioners make operational decisions	control charts pass/fail course grade troublesome module graph: complexity * structuredness that tells a project manager which models need inspection

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

An Example of a Software Meter



Measure: Line of code that was (or was not) executed

Metric: Statement Test Coverage

% of source statements executed during a test

Meter: Statement Test Coverometer

95 % => accept test suite

75 % => more tests needed

<75% => test redesign

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Fractal Measurement: Top-Down & Bottom-Up

Newtonian (Deductive)

① develop organization or project goals

② generate questions that define these goals

③ specify meters (or metrics) that determine if these goals have been achieved

④ specify the measures needed to calculate these meters

⑤ develop a plan for collecting & validating these measures

⑥ collect, validate, & analyze the meters or metrics to determine if the goals have been met

Darwinian (Inductive)

⑥ repeat based on the new insight gained

⑤ calibrate meters for use by managers and practitioners in day-to-day operations

④ analyze the metrics created from these measures

③ develop, validate and install a mechanism for collecting these measures

② define what can / should be measured

① study the process to be optimized

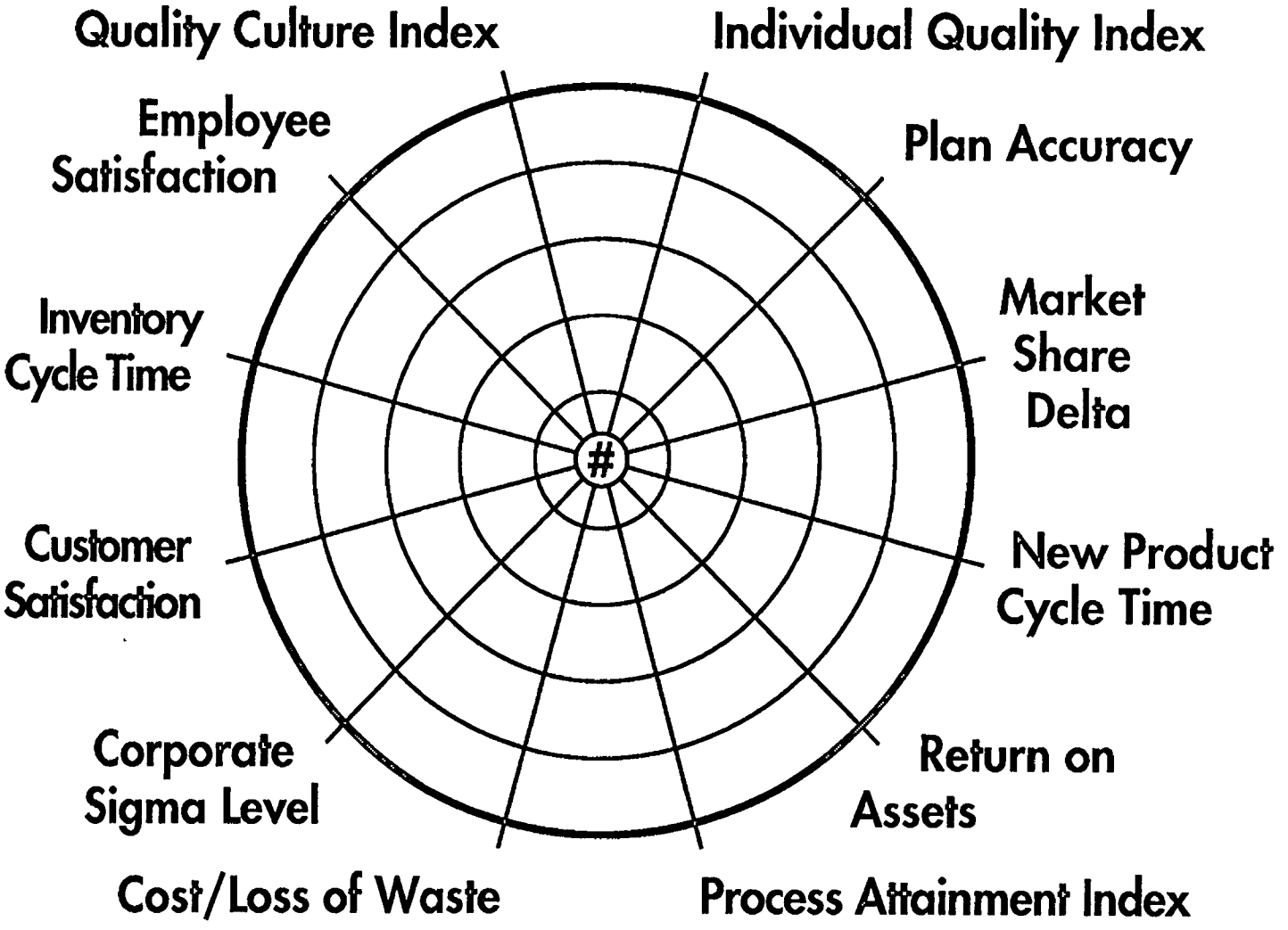
© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Fractal Measurement: Scaling

Culture Index	Defect Containment Index
company	company
division	product
department	release
project	load module
individual	source module

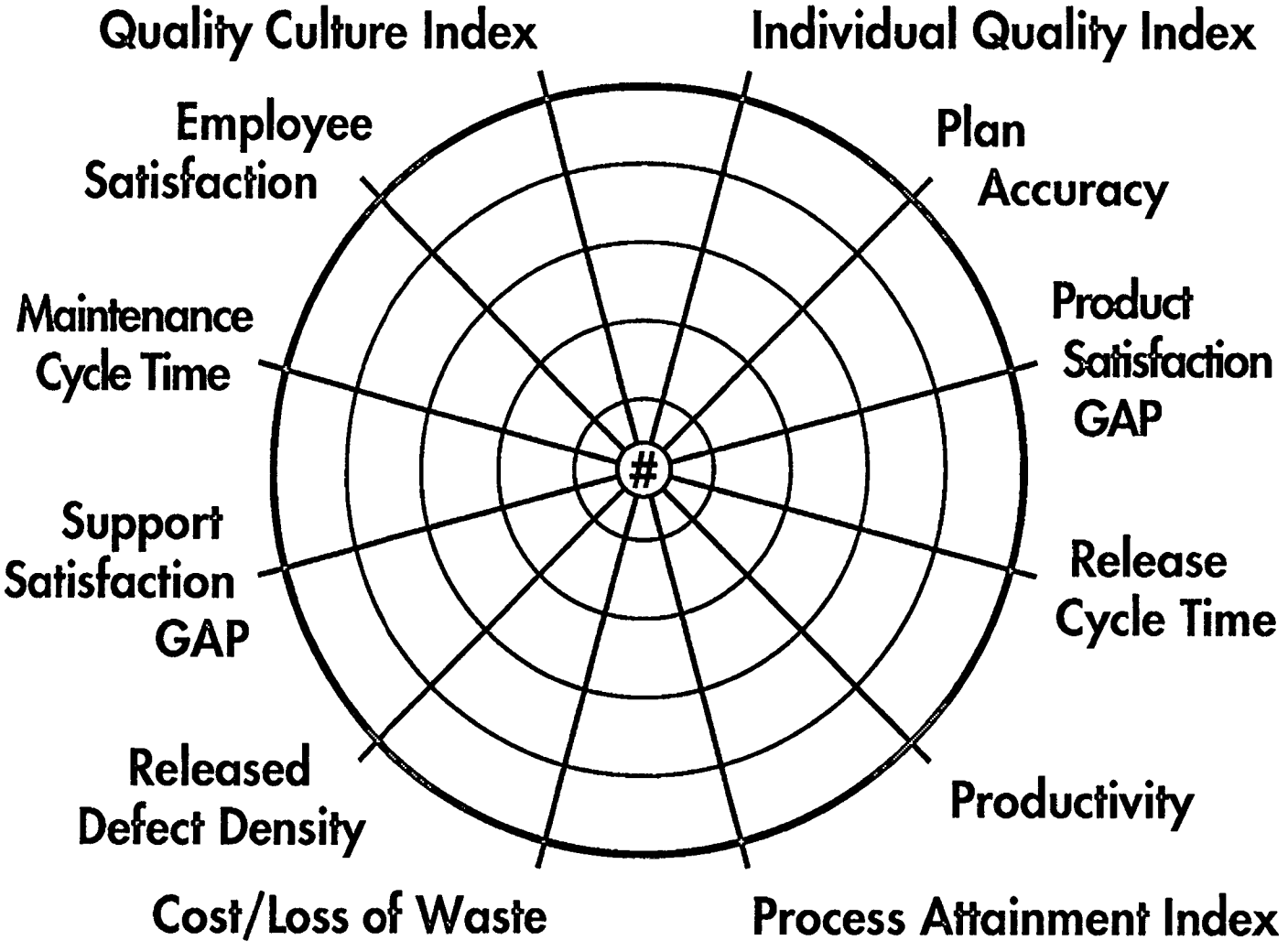
© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Corporate Business Process RESULTS



© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Software Engineering Process RESULTS



© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Controlling Variation

66

The central problem of management in all its aspects ... is to understand better the meaning of variation, and to extract the information contained in variation.

99

Dr. W. Edwards Deming

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

The Japanese Train Story

(From *Out of the Crisis* by W. Edwards Deming)

5:25 pm.....Leave Taku City

7:23 pm.....Arrive Hakata

Change Trains

7:24 pm.....Leave Hakata

What does this say about the Japanese train system?

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Using Coverage: Faults of Omission

- Actual code

```
a = b / c;
```

- Required code

```
if (c != 0)
    a = b / c;
```

- Dangerous to use coverage for test design

Handling Remaining Uncovered Conditions

- Is the branch impossible?
- Is the branch not cost-effective?
  - Errors + panics
  - Errors + recovery
- What feature corresponds to the branch?
- What other ways has feature been under-tested?
  - Coverage supplies clues, not commands

A Practical Goal for Coverage

- Consistent 90%+ feasible coverage from initial test design
- Easy to reach 100% feasible coverage
  - $< 4\%$ additional cost
  - Especially cheap to fix mis-implementations
- Goal: cost of exercising a branch less than risk of customer doing it

Recommended Types of Subsystem Coverage

- Branch
- Loop
- Multicondition
- Relational

GCT Availability

- Anonymous FTP:
  cs.uiuc.edu:pub/testing/GCT.README
- marick@testing.com

What Can Be Done Now

- Coverage low (20% - 80%)
- No time for thorough examination of coverage
- Look for large patterns and cheap tests
- Do not blindly try to achieve coverage
- Plan on doing better next time
  - Test implementation
  - Test design

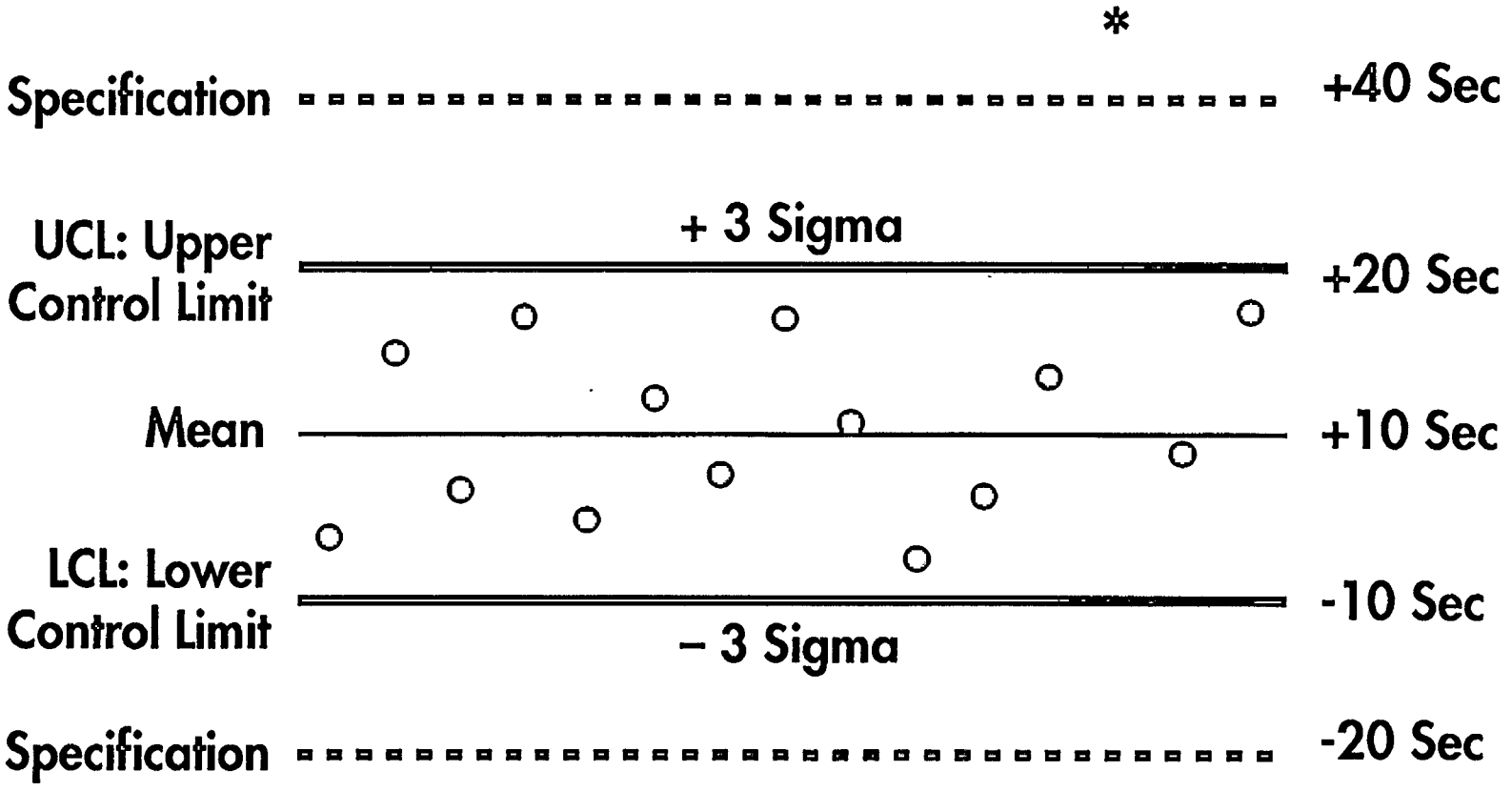
Combining Test Conditions

- Use many test conditions per test case
- Use test conditions fairly

Test Implementation Topics

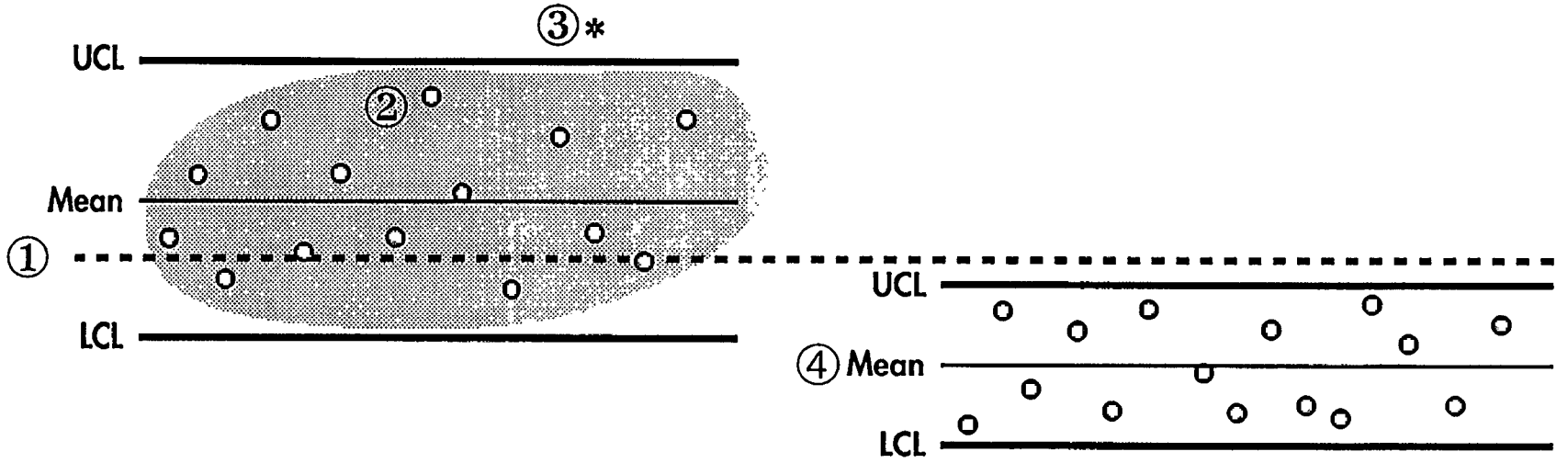
- Building Tests
  - Assume test conditions given
  - How to use them
- The scale of testing
  - Testing via the subsystem
  - Gradual implementation of testable subsystems

Are Japanese Trains on Time?



© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Understanding and Managing Variation



① Control limits are NOT goals

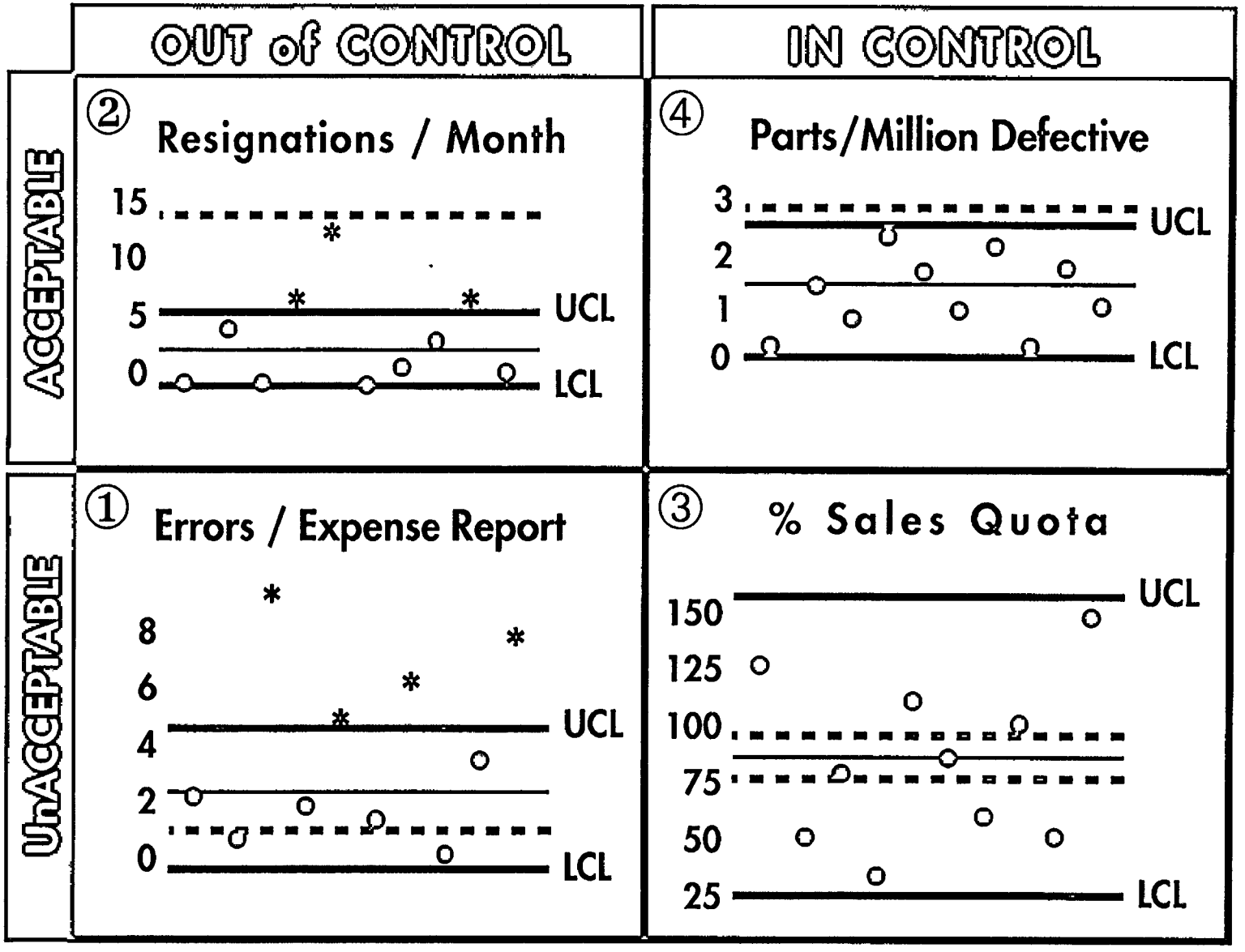
② IGNORE variation in control

③ REMOVE special causes to get/keep process in control

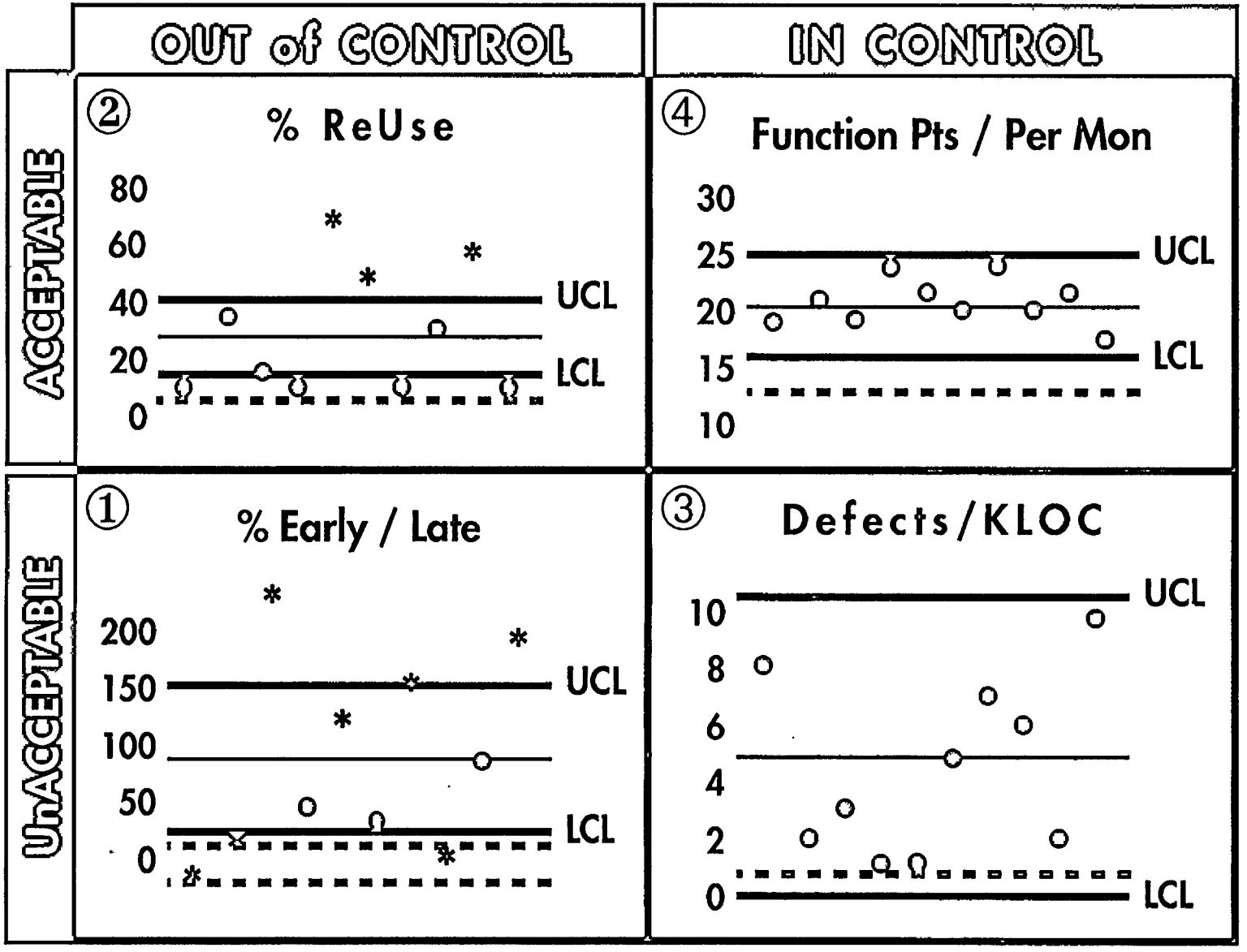
④ CHANGE process to achieve stable, breakthrough results

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Statistical Process Control



© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032



© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Breakthrough Bead Experiment

- ❑ Adapted from W. Edwards Deming by Bill Silver.
- ❑ What good and bad management behavior do you observe?
- ❑ Do you see a *breakthrough solution* to the reliability problem?

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Values

- ❑ What are Values?
- ❑ Who Do Values Affect?
- ❑ The Values in Deming's 14 Points

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

What Are Values?

- ❑ **Values are what matter to people.**
- ❑ **Values are social principles or standards of behavior held or accepted by people.**
- ❑ **Values are principles that guide the way you do business.**

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Who Do Values Affect?

- Values affect all organizations, processes and projects
- Values serve as guidelines for individual and group behavior for all the **stakeholders** of a company:
  - Customers
  - Employees
  - Shareholders
  - Suppliers
  - The Community

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Differences in Values Really Matter

"

**Ideas are much more fatal things than guns.
Why should any man be allowed to buy a printing press
and disseminate pernicious opinions
calculated to embarrass the government?**

"

Nikolai Lenin, 1920

"

**...and were it left to me whether we should have
a government without newspapers,
or newspapers without a government,
I should not hesitate a moment to choose the latter.**

"

Thomas Jefferson, 1787

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Deming's 14 Points

"

The 14 points are the basis for transforming American industry.

It will not suffice to solve problems, big or little.

Adoption and action on the 14 points are a signal that the management intends to stay in business and aim to protect investors and jobs.

"

Dr. W. Edwards Deming

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Deming's 14 Points

1. Create constancy of purpose toward improvement of product and service, with the aim to become competitive and to stay in business, and to provide jobs.

2. Adopt the new philosophy. We are in a new economic age. Management must awaken to the challenge, must learn their responsibilities, and take on leadership for change.

3. Cease dependence on mass inspection to achieve quality. Build quality into a product.

4. Stop awarding business on the basis of price tag. Minimize total cost. Move toward a single supplier for any one item, on a long-term relationship of loyalty and trust.

5. Improve constantly and forever the system of production and service, to improve quality and productivity, and thus decrease costs.

6. Institute training on the job.

7. Institute leadership. Supervisors should help people, machines and gadgets do a better job. Supervision of management and production workers needs an overhaul.

8. Drive out fear, so that everyone may work effectively for the company.

9. Break down barriers between departments. People in research, design, sales, and production must work as a team to foresee problems in production and in use that may be encountered with the product or service.

10. Eliminate slogans, exhortations, and targets for the work force asking for zero defects and new levels of productivity. Exhortations only create adversarial relationships.

11. Eliminate numerical quotas for the work force and numerical goals for management. Eliminate management by objective.

12. Remove barriers that rob people of pride of workmanship. Eliminate the annual rating or merit system.

13. Institute a vigorous program of education and self improvement for everyone.

14. Put everybody in the company to work to accomplish the transformation. The transformation is everybody's job.

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

The Values in Deming's 14 Points

Personal Values	Corporate Values
Self Improvement	*Constant Improvement*
Individual Responsibility	*Provide Jobs*
Leadership for Change	*Management as Leaders*
Loyalty	*Long-Term Business Partners*
Teamwork	*Being Competitive*
Pride of Workmanship	*Build Quality Into Products*
Trust	*Drive Out Fear*
On the Job Training	*Help People*
Education	*Minimize Total Costs*
Focus on People	*Improve Productivity*

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Leadership

- ❑ *Mission*
- ❑ *Vision*
- ❑ *Strategy*

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Leadership Through Mission

"Mission: A continuing responsibility." ... *Webster*

Organization

What an organization is destined or fitted to do or accomplish

Process

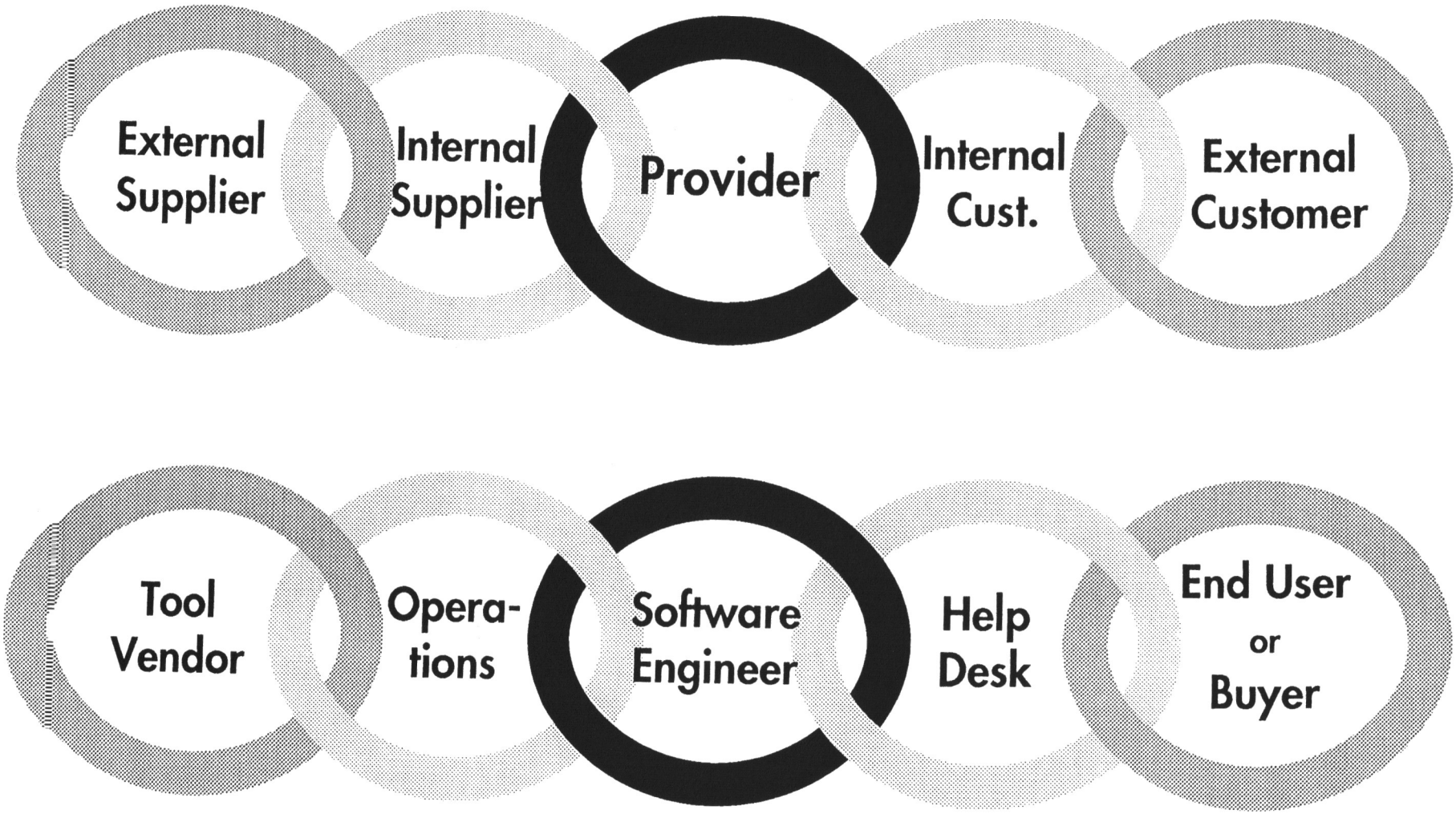
Generic deliverables, customers & suppliers of the process

Project

Specific deliverables in a specific timeframe

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Supplier ➦ Provider ➦ Customer



© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Leadership Through Vision

"Vision: The power of imagining the future." ... *Webster*

Organization

Define the future of the organization.

Vision can extend the mission of the organization.

Process / Infrastructure

Breakthroughs in:

Value: some*thing* new that will help or delight the customer.

Productivity: a faster, cheaper, less wasteful *way* of doing a job.

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Vision and Objectives

"

Effective visionaries bring their visions to life as concrete objectives. Objectives taken collectively fulfill the mission of an organization or process.

"

Shel Siegel

"I propose that this nation should commit to the goal of sending a man to the moon by the end of the decade and returning him safely to earth..." *JFK*

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Leadership Through Strategy

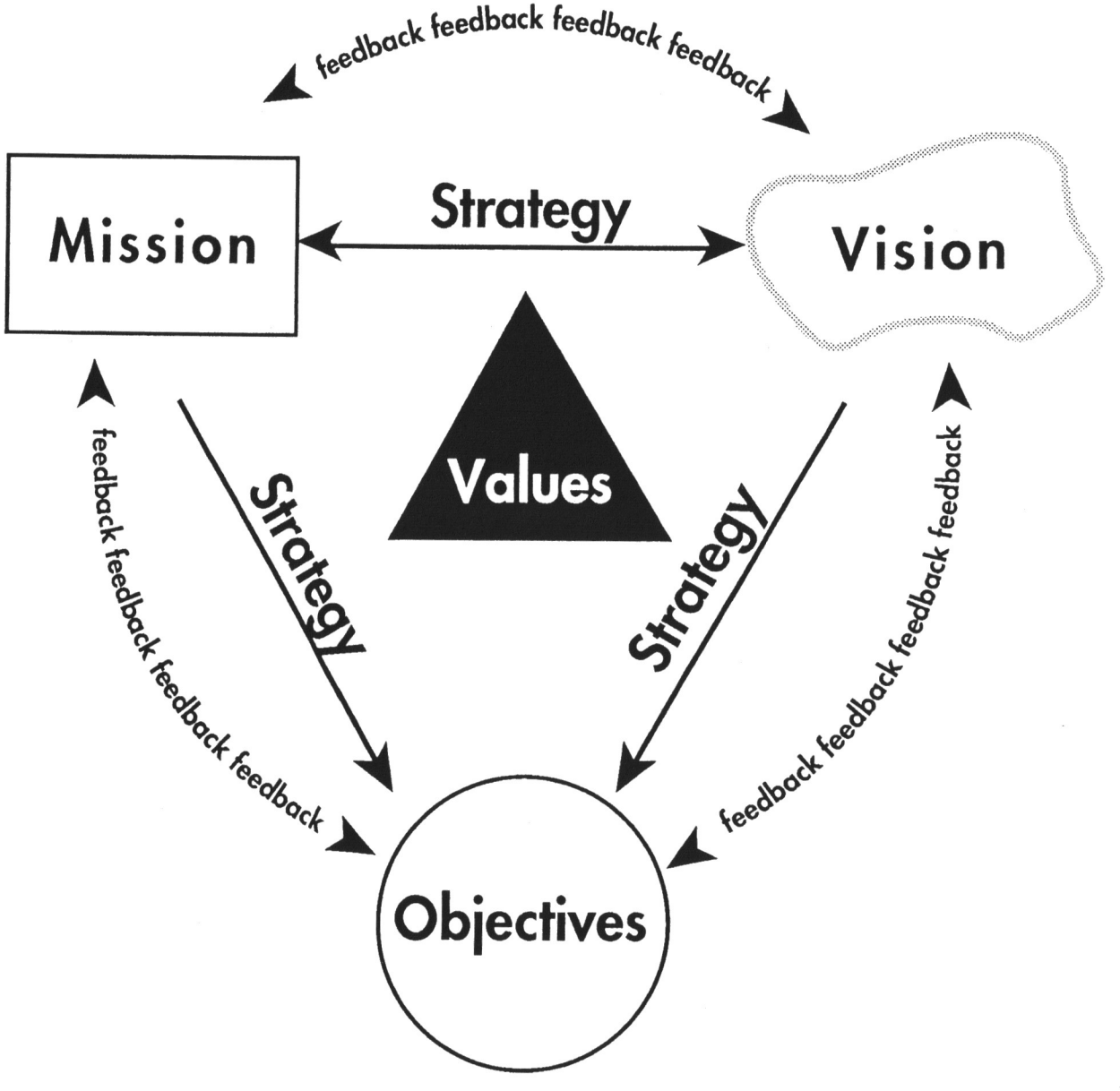
"Strategy: A high-level plan of action."
Webster

You need a strategy to:

- ❑ nurture and share your vision
- ❑ reach your objectives

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Leadership: The Dynamo of Quality



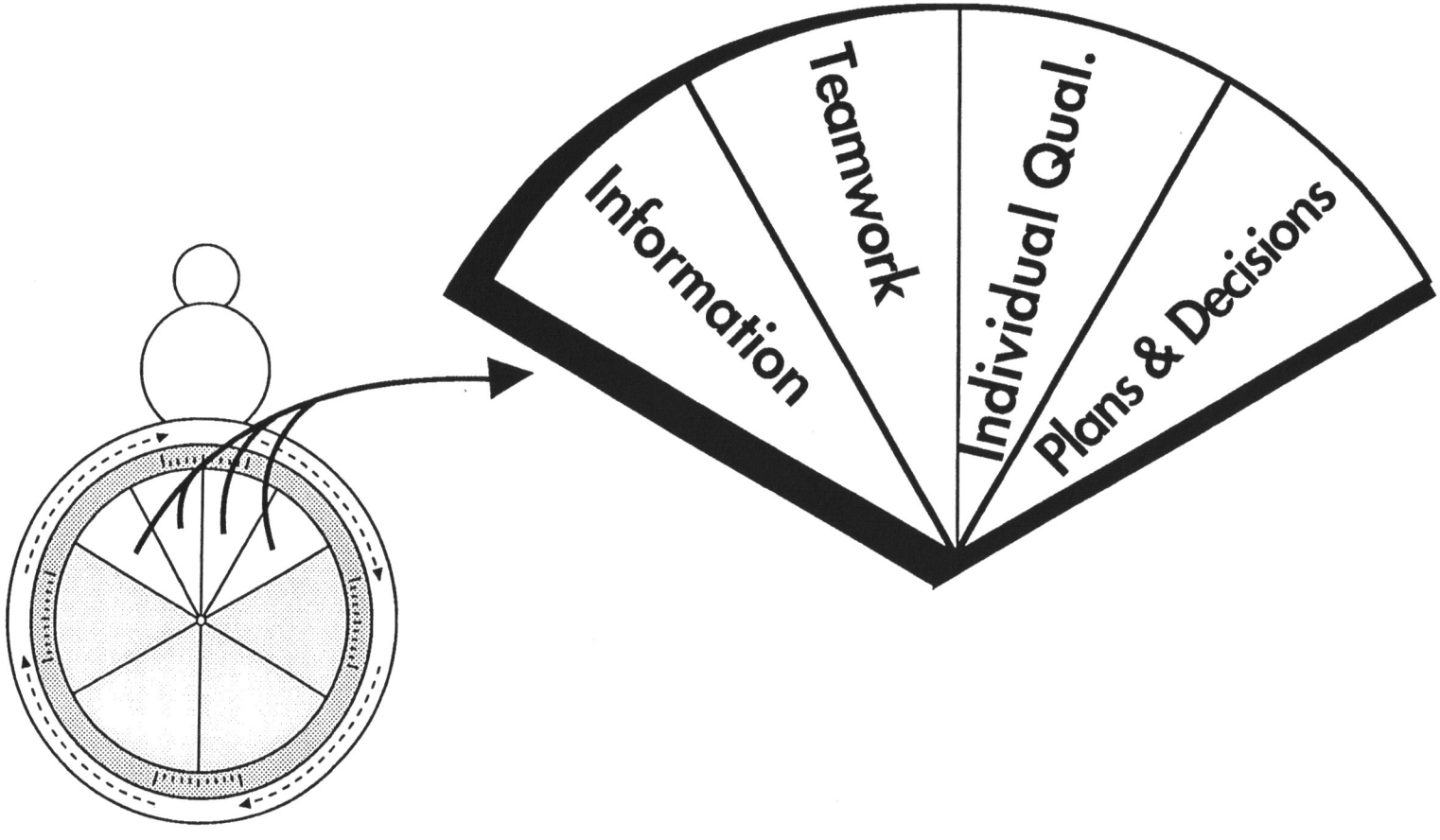
© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Quality Management, Process, & Infrastructure

- ❑ Quality Management
- ❑ Quality Drivers
- ❑ Process Life Cycle
- ❑ Infrastructure

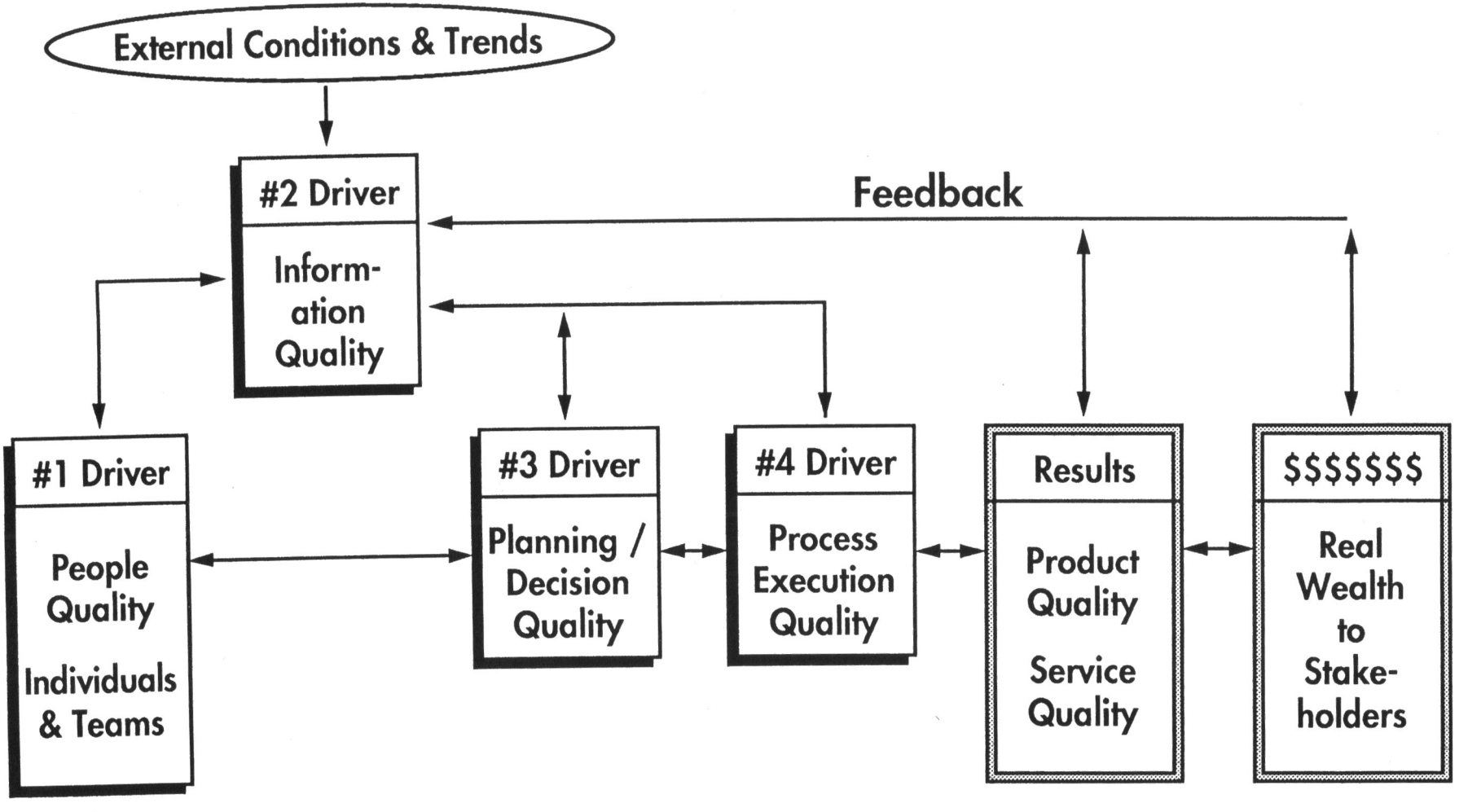
© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Quality Management



© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Quality Drivers



Presented with the permission of Dennis Grahn, Director of Corporate Quality Management, Menasha Corporation

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Individual Quality

Organization

- What people know about quality
- How they behave
- Stephen Covey's *Seven Habits of Highly Effective People*

Process

- What people know about how to do their jobs
- How they participate in process optimization

"Becoming the best at something ... motivates Americans; producing the best products or services does not. Thus, quality must be made personal; the focus must be on the individual."

G. Clotaire Rapaille, *Quality Archtype Study*
Quality Progress, February 1993

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Teamwork Quality

Organization

- Line staff

Process

- Product Team
  (Cross-functional)
- Project Teams
- Process Improvement Teams

Teamwork and team leadership can and must be taught, practiced, and measured.

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Information Quality

Organization

- Sharing information with the line staff and people in an organization
- Sharing information with other organizations

Process

- Cross-function information *(plans, status, issues)*
- Information about customers
- Sharing information with customers
- Sharing information with suppliers

The goal for information quality is for all stakeholders to have timely access to all the accurate information they need.

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Planning & Decision Quality

Organization

- Strategic planning
- Resource capability planning

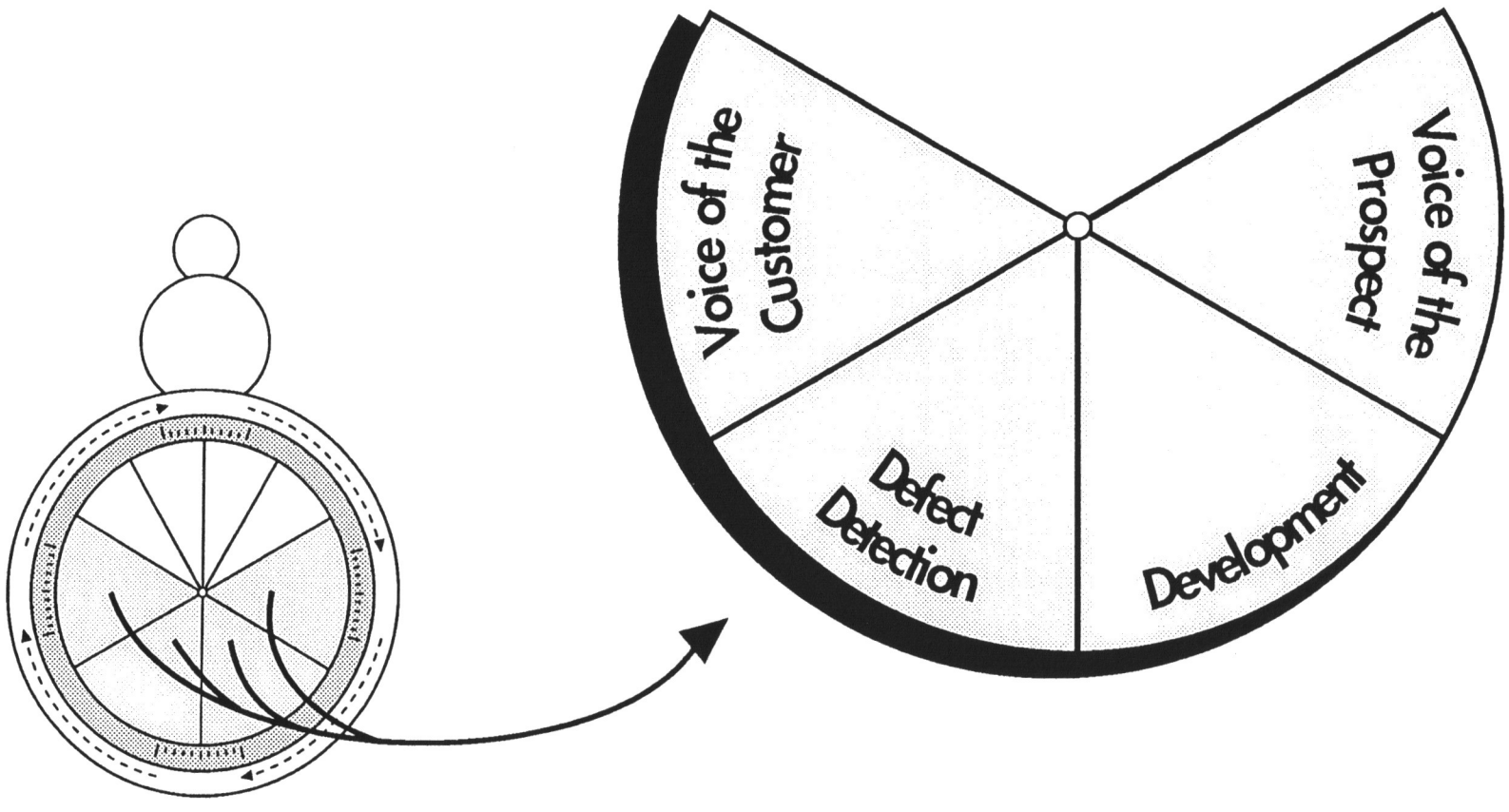
Process

- Product planning
  (Customer focused)
- Project planning
- Planning for process improvement

Planning and decision making must be overt activities, and performed with skill, tools, and accurate information.

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Process Life Cycle



© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Voices of the Customer / Prospect Activities

- Customers & Prospects are Similar but Different
- Optimized Quality Implies BALANCE Between

Support	↔	New Products
Compatibility	↔	Innovation

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

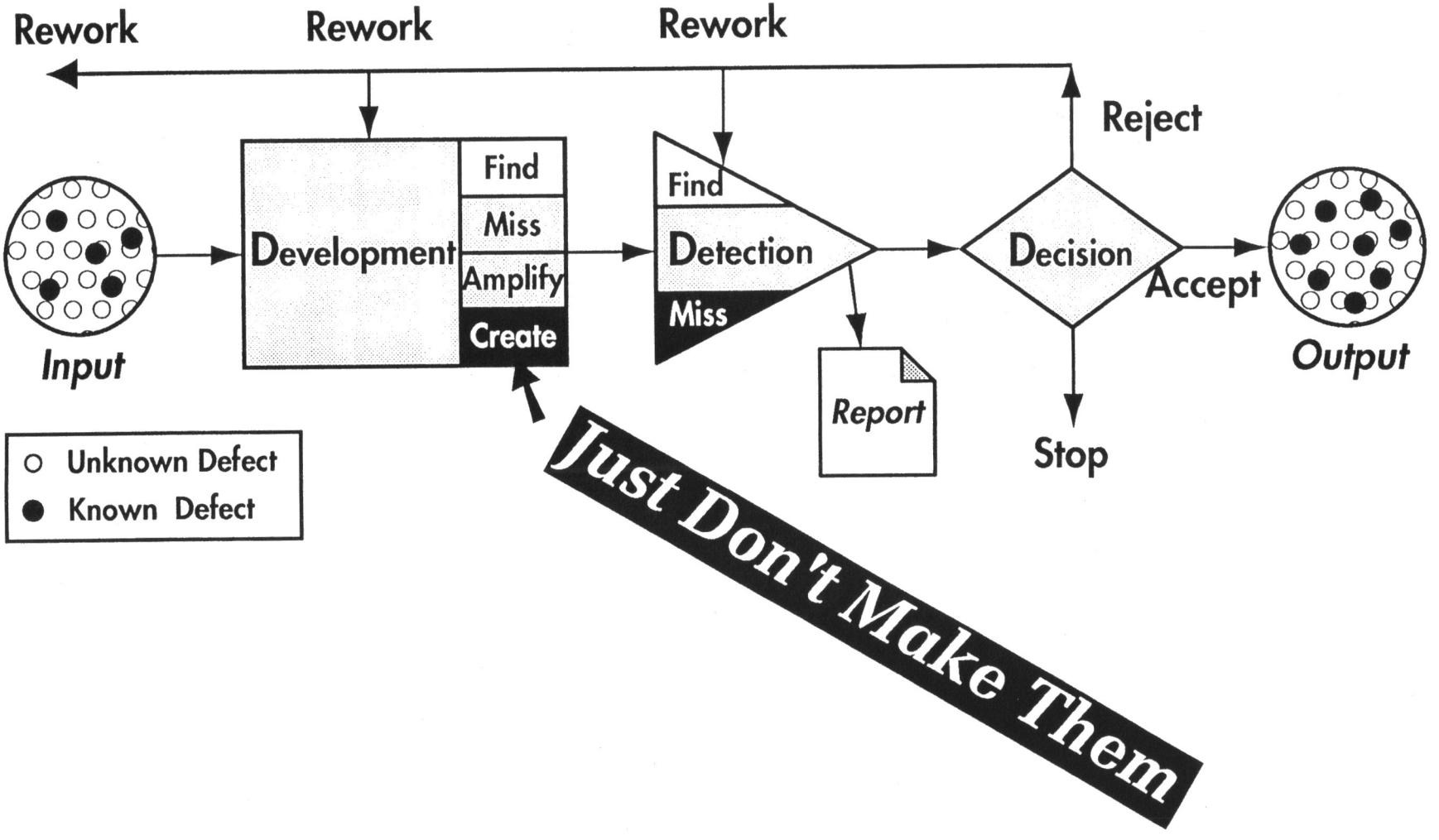
Development Activities

- These Activities CREATE (or BUY)
  Products or Services
  (Either in Final or Intermediate Form)

- These Activities are the Focus of Prevention
  (They are Where All Defects are Created)

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Defect Prevention



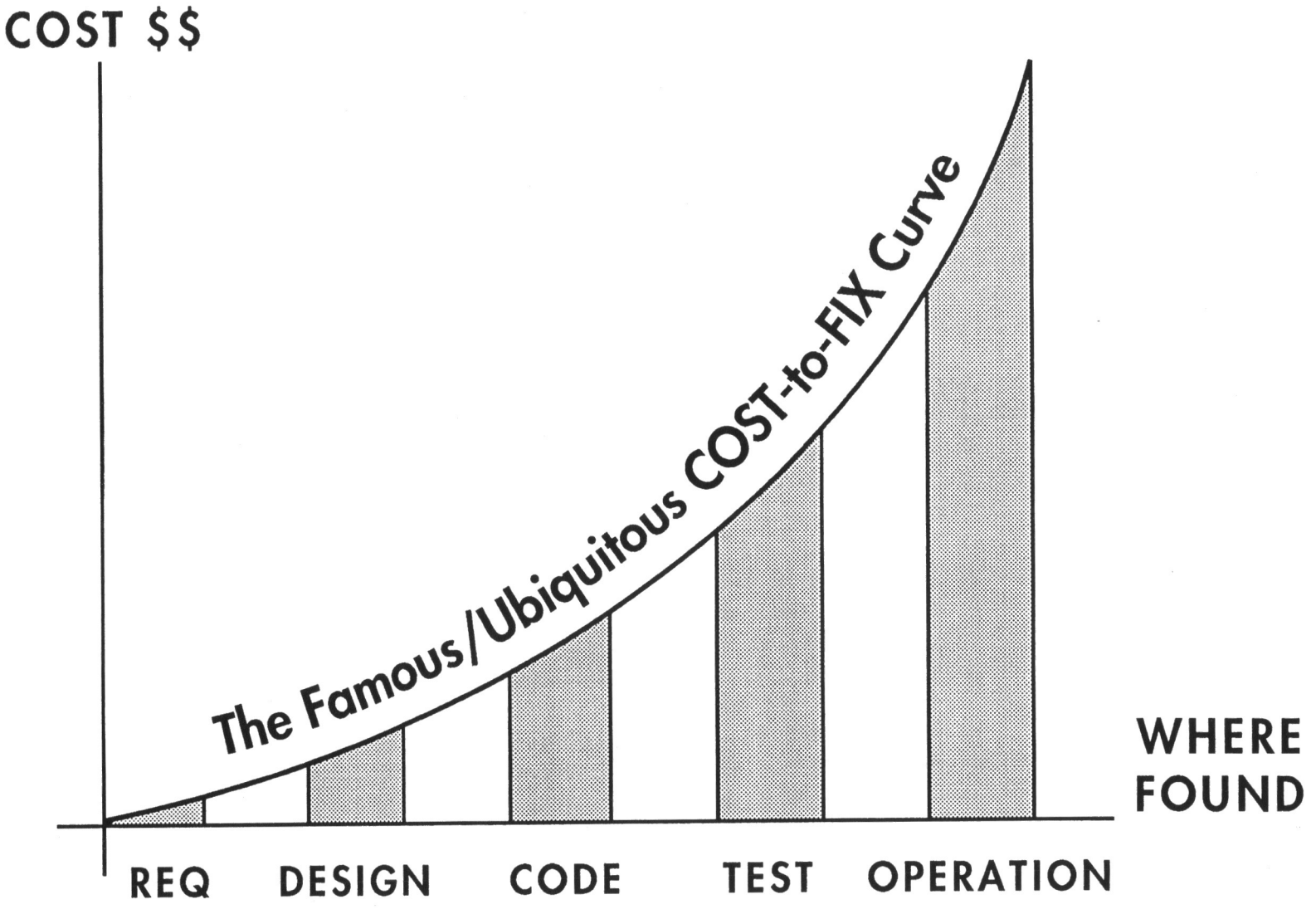
© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Defect Detection Activities

- Deming said *"Focus on Prevention"*
  He never said *"Don't Test"*
- Early DETECTION + REWORK = PREVENTION
  to the Rest of the Process & the Customer
- What is the Optimum Goal?
  Zero? 6-Sigma? ?????

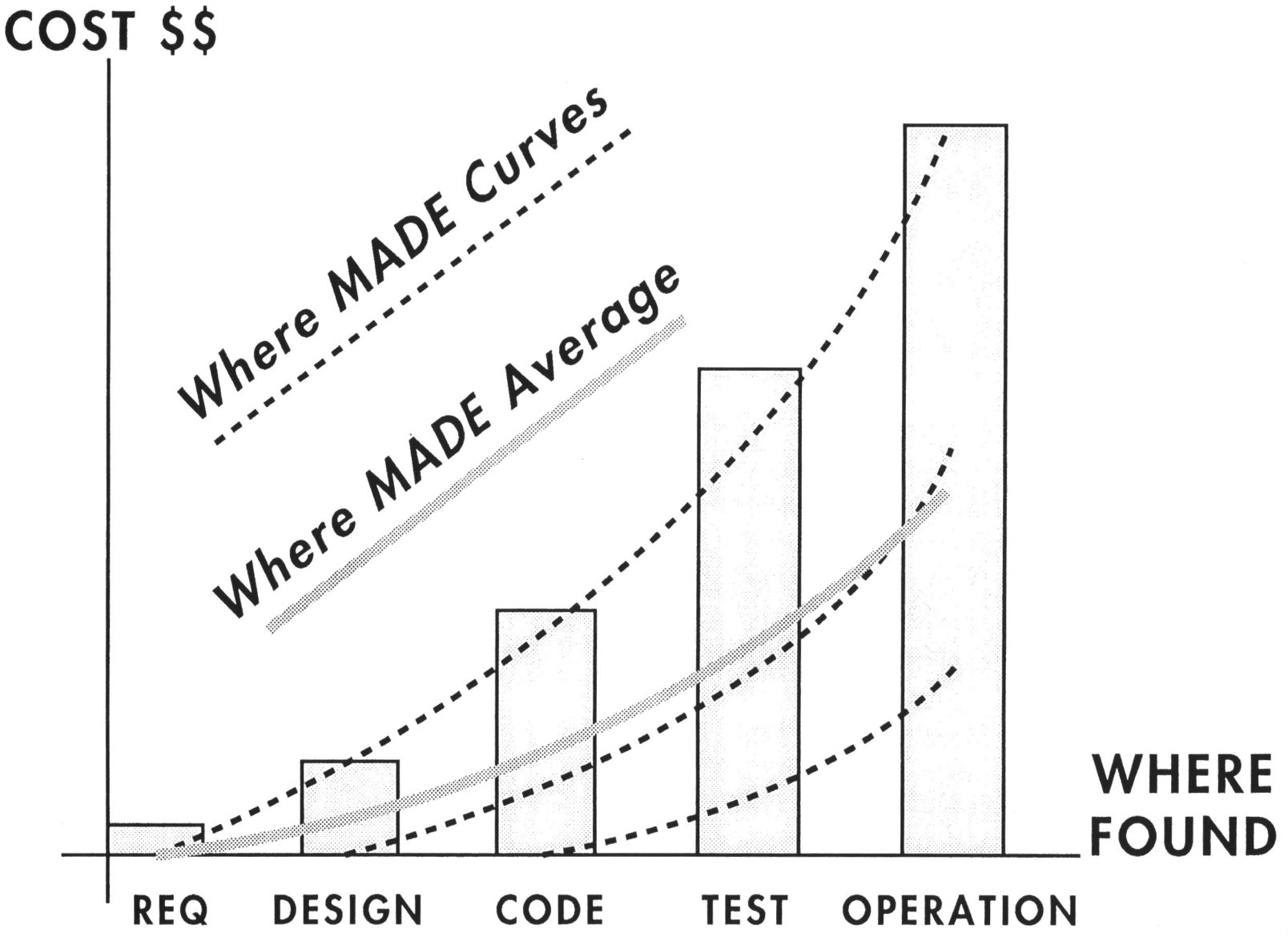
© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

The Value of Early Defect Detection



© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

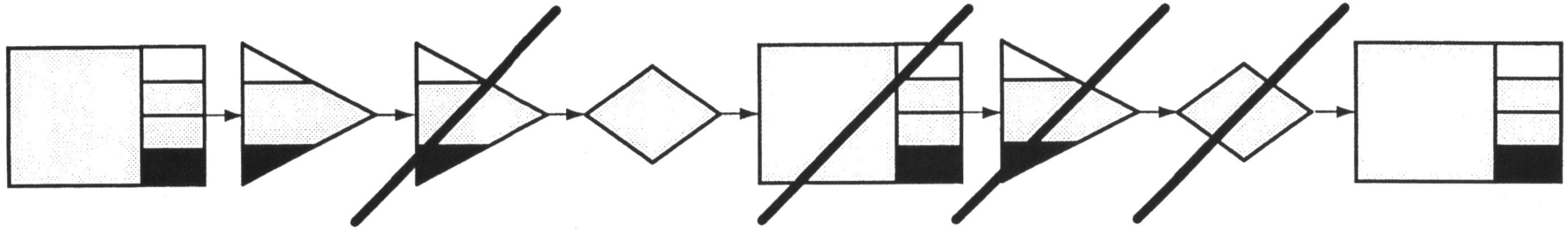
The Real Value of Early Defect Detection

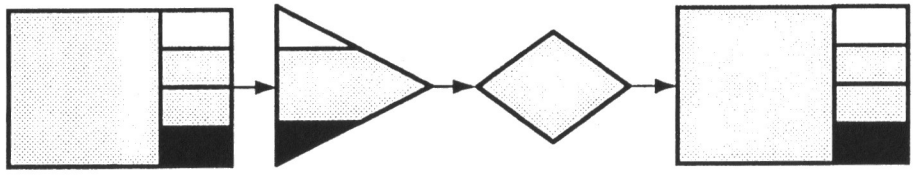


© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Waste Prevention: Process Re-Engineering

Old Process



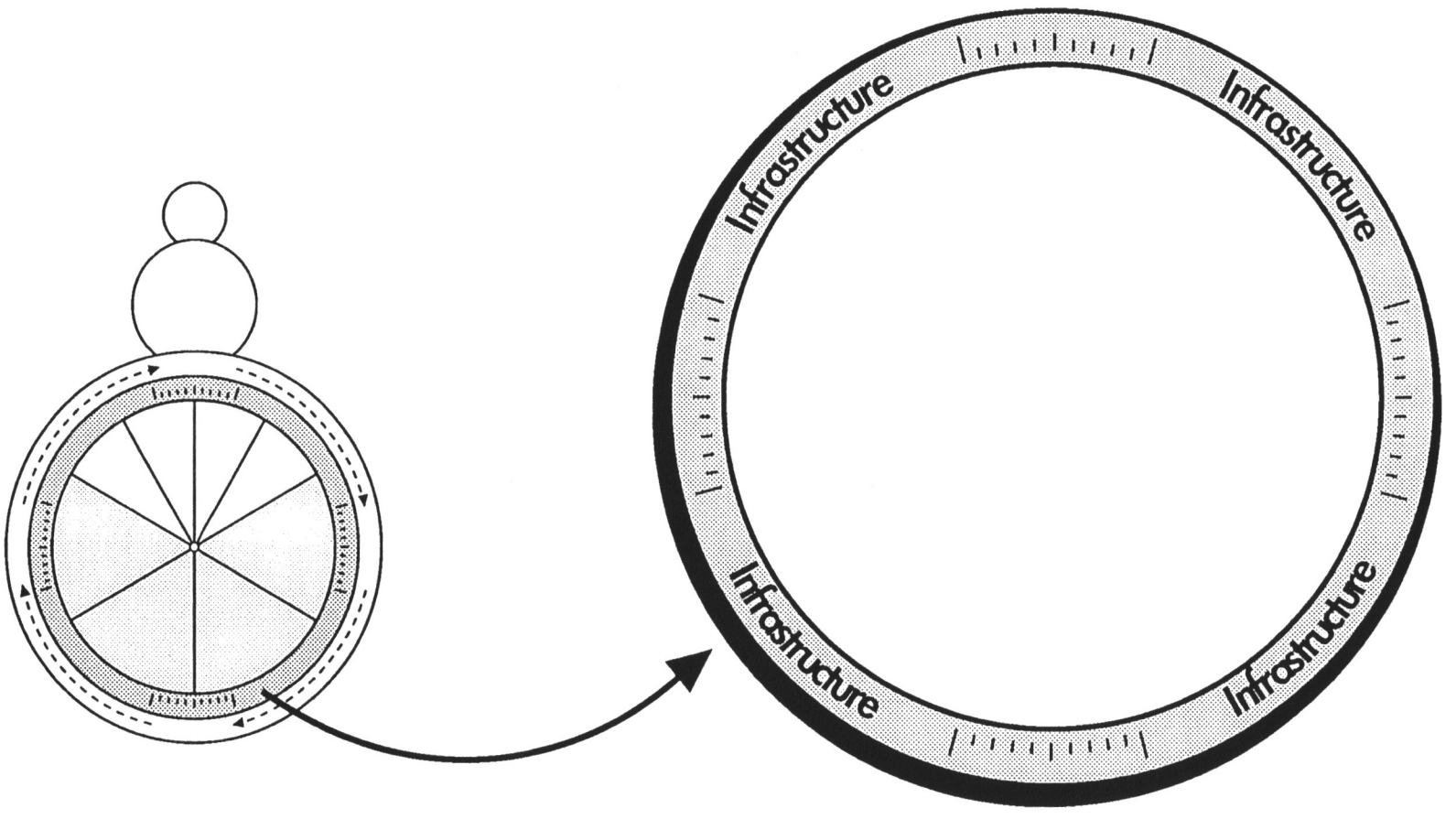


New Process

(After Breakthrough Re-engineering)

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Infrastructure



© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Infrastructure Items

Communication:

How people share the information they need to do their jobs

Standards:

External & internal references, guides & benchmarks

Tools:

Hardware, software, and physical implements

Methods:

Procedures and techniques

Learning:

On-going education, training & on-the-job experience that enhances people's ability to perform a quality job

Environment:

The physical work surroundings

Staffing:

Hiring people with the required skills and experience

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

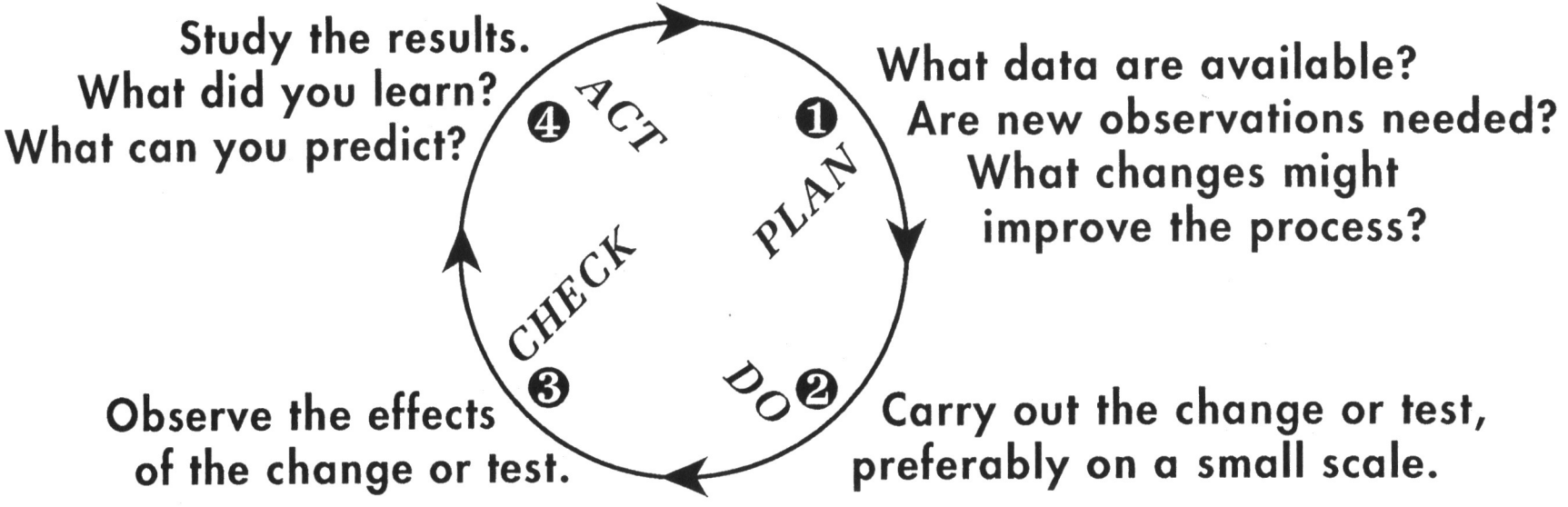
Optimization Cycles

- ❑ The Shewhart Cycle
- ❑ Incremental and Breakthrough Process Improvement
- ❑ The TEN Steps of the Process* Optimization Cycle

*Organization optimization cycles work the same way except where noted.

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

The Shewhart Cycle

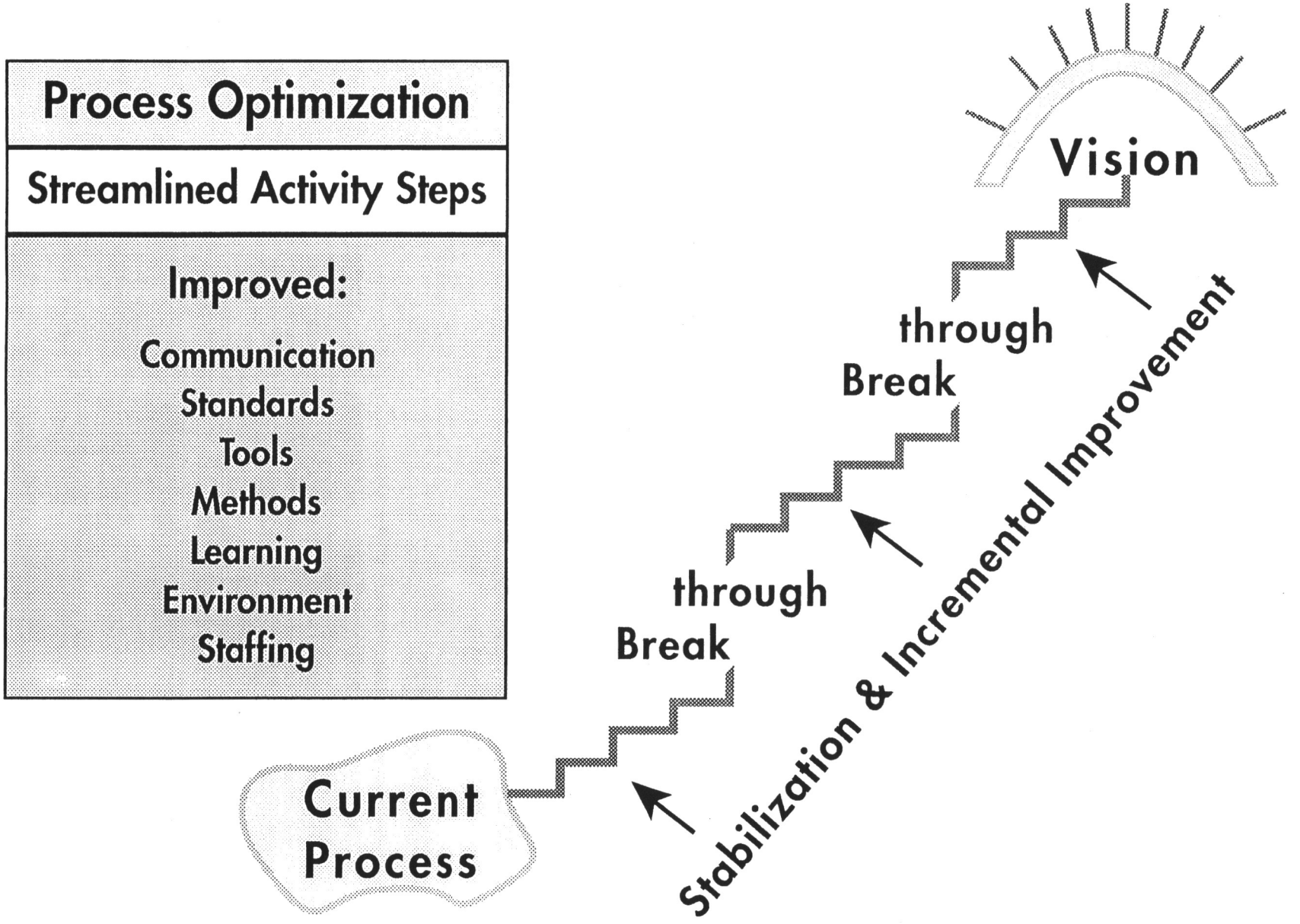


❺ Repeat Step 1, with knowledge accumulated.

❻ Iterate Step 2, and onward.

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Incremental & Breakthrough Improvement



© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Process Optimization Cycle

PLAN:	① *Adopt / Adapt / Extend* higher-level Process Values, Mission, Vision, and Strategy ② *Observe, Understand* and *Document* the current **Process ↔ Infrastructure** ③ *Assess* and *Benchmark* Process Capability ④ *Identify* and *Prioritize* Changes ⑤ Define *Measurement* of Results ⑥ *Plan* and *Prepare* the Changes
DO:	⑦ *Install and Operate* new **Process ↔ Infrastructure**
CHECK:	⑧ *Evaluate* Quality Results
ACT:	⑨ *Leverage* to other parts of the company: Processes, Projects, and Organizations ⑩ *Iterate* Process Optimization Cycle

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

① Values and Leadership

- ❑ Build on the Values and Leadership (Mission, Vision, and Strategy) provided by the higher-level processes.
- ❑ Adapt and extend them for this process.
- ❑ Who are the customers (and suppliers) of this process? What do they want?

If no values, mission, vision, or strategy have been defined at a higher level then you must define them yourself. Then work to leverage them—UP / ACROSS/ DOWN.

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

② Observe Current Process

- ❑ Observe and *Understand*—what you do now:
  - Process *Activities*
  - Supporting *Infrastructure*
- ❑ Document the current process:
  - Flow Charts
  - Structured Analysis
  - Object-Oriented Analysis
  - Process Cinemazation *(See Bill Silver)*

Like any navigation, the first step in navigating change is knowing where you are.

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

③ Assess & Benchmark Process CAPABILITY

Assessment:

Rating how capable you are vs. some standard.

For Software:

- **TQO Roadmaps**
- **SEI Capability Maturity Model**
- **ISO 9003 (TickIT)**

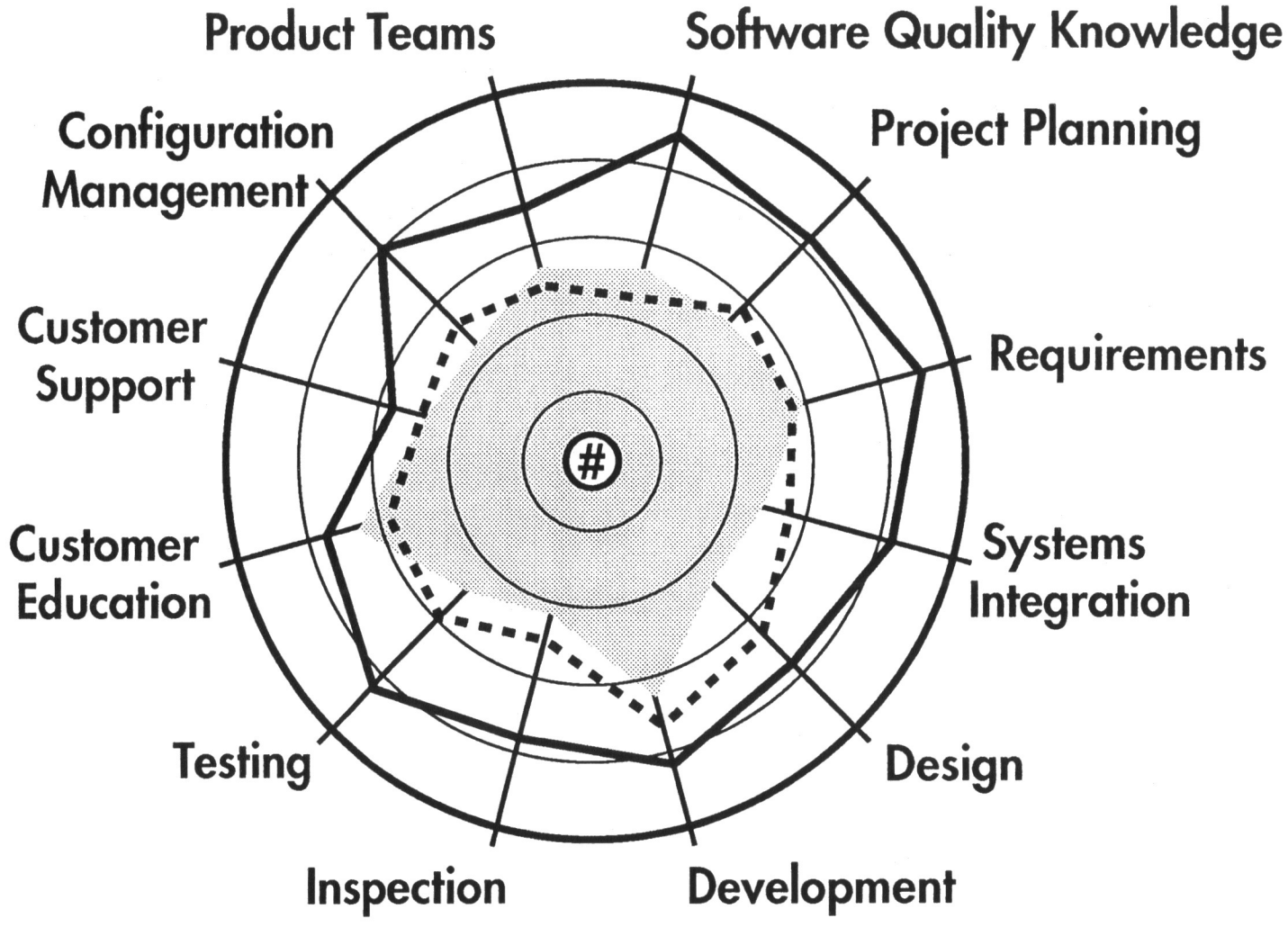
Benchmark:

Learning how others do the same job.

- ***Generic:* best anywhere in the world**
- ***Competitive:* your direct competitors**
- ***Internal:* others in your company**
- ***Functional:* those doing like things, but in different industries**

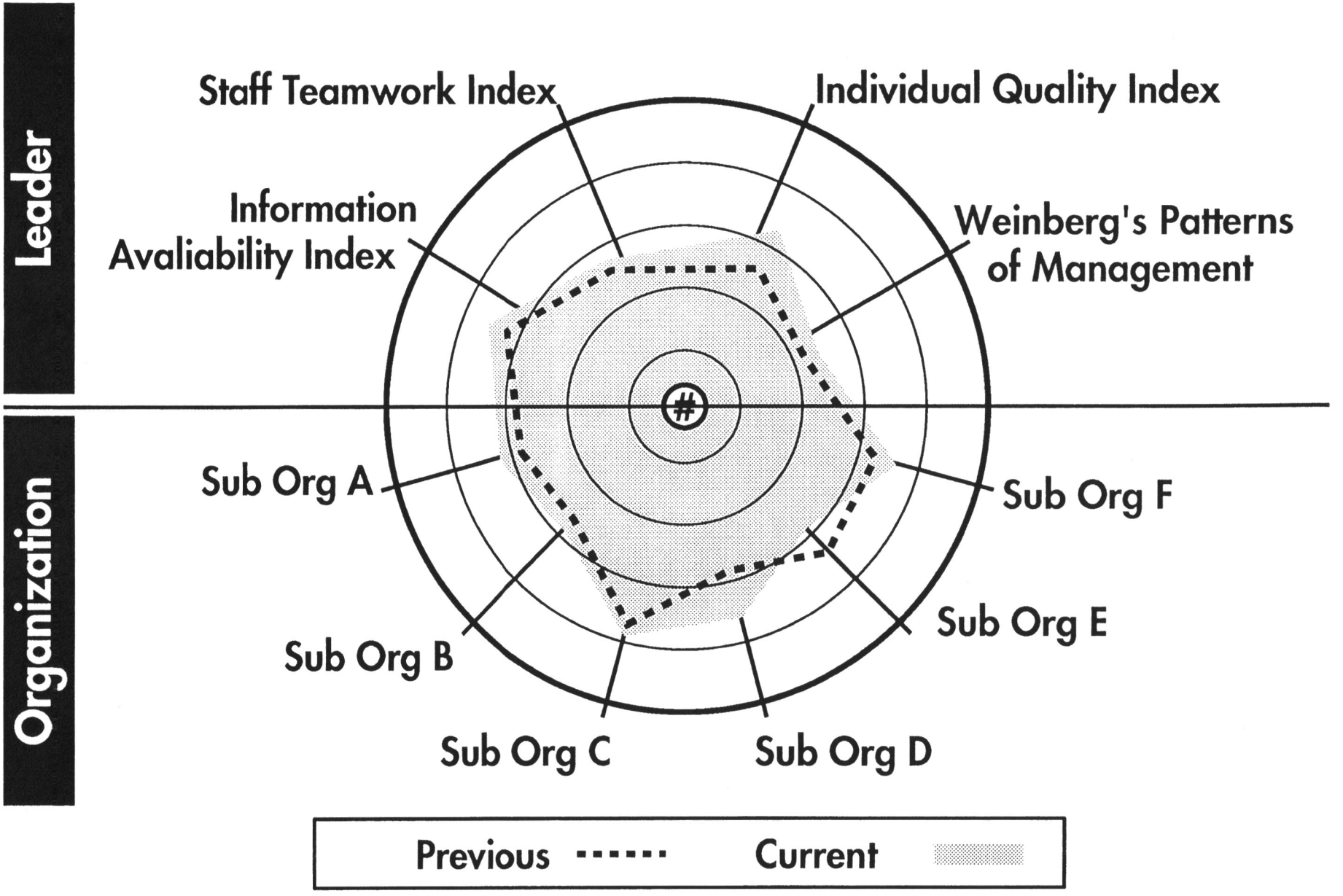
© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Software Engineering Process Assessment



© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Organization Assessment



© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

④ Identify and Prioritize Changes

- ❑ Brainstorming and breakthrough analysis
- ❑ Root Cause Analysis:
  - Ohno's 5 why's
  - Fishbone diagrams
- ❑ Pareto analysis:
  - Identify the high-leverage problems
- ❑ Optimize the balance between:
  - time, cost, difficulty, and payback

Note: changes in process often require changes in infrastructure—and vice versa.

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

⑤ Define Measures of Results

- ❑ What meters / metrics will tell you if the changes were successful?

- ❑ What new measures can / should be collected to instrument the new process activities and infrastructure?

Note: a person's subjective opinion is an objective measure.

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

⑥ Plan and Prepare

- ❑ Changes to process may require:
  - new methods and standards
  - retraining
- ❑ Changes to infrastructure may require:
  - tool requirement analysis
  - vendor and tool selection
  - retraining

Changes to process or infrastructure are themselves real projects.

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

⑦ Install and Operate

- ❑ Install the process and infrastructure changes according to your plan.
- ❑ Process and infrastructure changes sometimes require months (or more) before you can tell how well they are working.
- ❑ If the risk is high try a Pilot Project.

Murphy's law dictates that you expect to make adjustments to your plan.

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

⑧ Evaluate

- ❑ Use quality tools and statistical analysis to evaluate the improvement
- ❑ Communicate the effect of the change to management and to everyone who works with this process
- ❑ Have a plan to UNDO a change if it is not working

The most important standard of reference is how much you have improved.

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

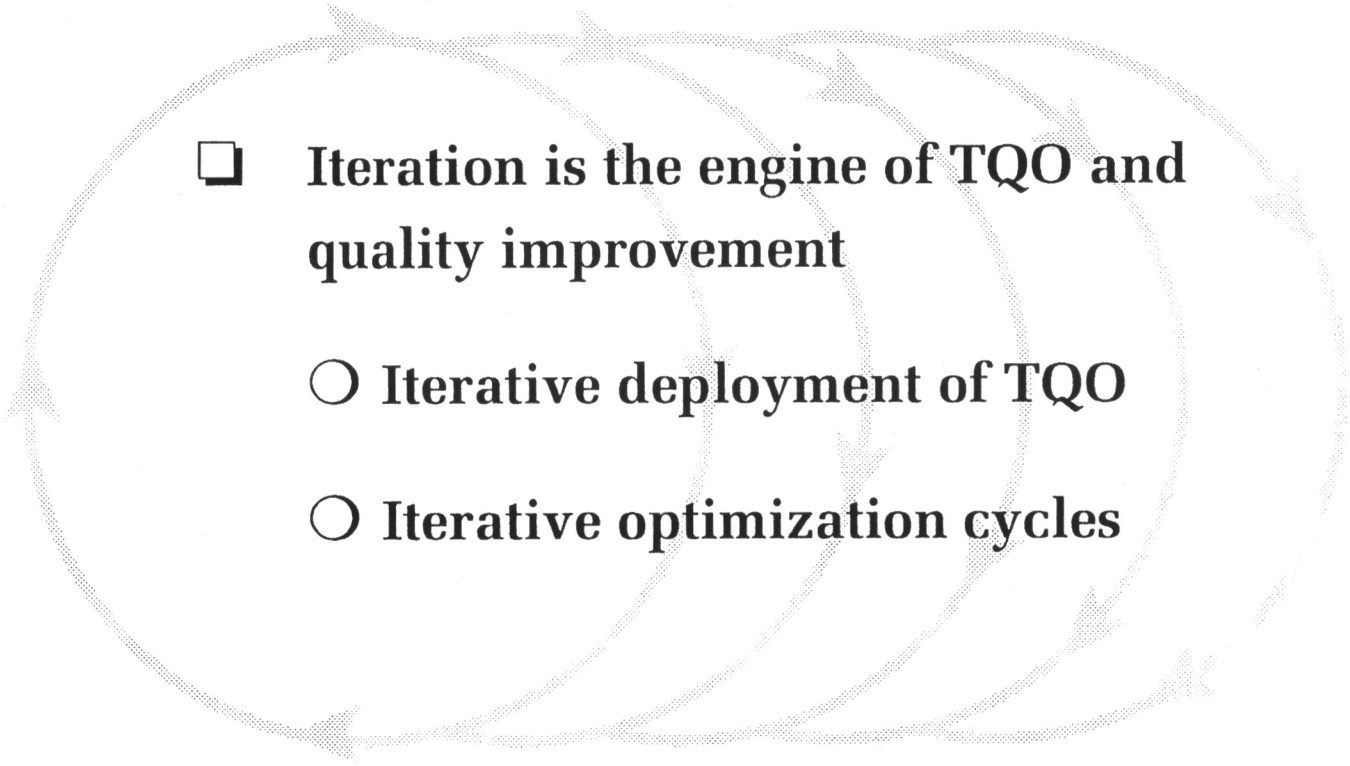
⑨ Leverage

- ❑ **Share your success with others:**
  - **quality council**
  - **higher-level process improvement team**
  - **similar projects**
  - **other organizations**
- ❑ **Serve as an internal benchmark**
- ❑ **Publicize in public and in private**

Note: sharing a failure is may be just as helpful to your peers as sharing a success.

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

⑩ Iterate



- Iteration is the engine of TQO and quality improvement
  - Iterative deployment of TQO
  - Iterative optimization cycles

Iteration with FEEDBACK is the only way to achieve continuous Total Quality Optimization.

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Case Study: Applying TQO

- ❑ Case Study Overview
- ❑ The Product
- ❑ The Organization
- ❑ Process & Infrastructure
- ❑ Worksheets

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

Case Study Overview

❶ You are the newly hired Vice President of Software Engineering at XYZ Software, Inc.

❷ On your first day at XYZ the CEO welcomes you and asks you, "What are you going to do to improve your organization and XYZ's product?" You agree to meet again on Friday afternoon and present your suggestions and plans.

As input to your planning the CEO gives you the official XYZ mission statement.

> "The mission of XYZ Software, Inc. is to provide high quality, accounting software packages to medium-to-large size businesses; and to provide excellent customer service and customer satisfaction; and to reward our shareholders with an outstanding financial return on their investment."

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

The Product: XPac

❶ XYZ's only product is an accounting package named XPac. XPac is used by financial analysts and accountants to consolidate financial information for strategic planning.

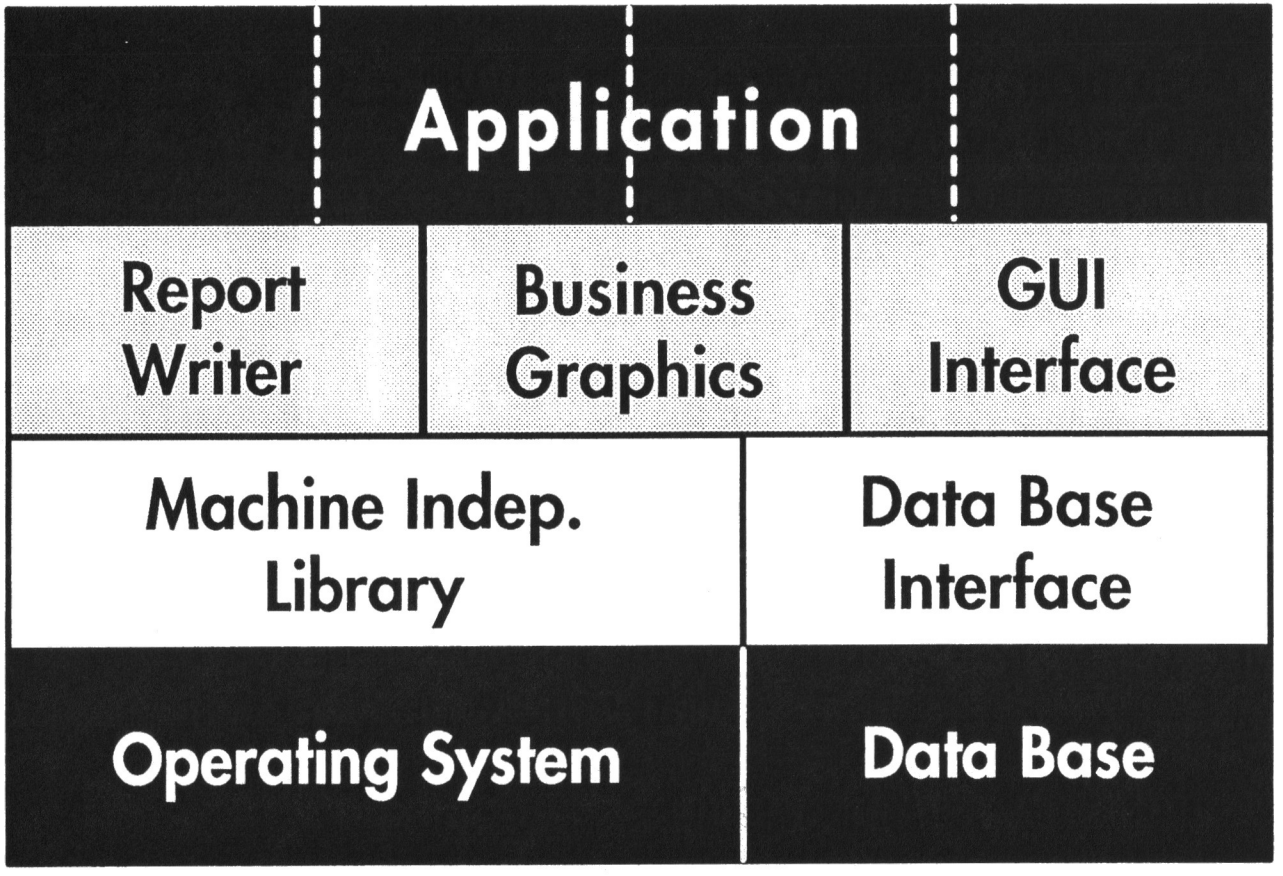
❷ The product has sold well in the past. In 1991 XYZ had 65% of this market. Although sales increased in 1992 by 8.3%, the XYZ's share of this market has slipped to 48.8%. Sales slumped in the first half of 1992 when ABC, Inc. introduced a new version of their competitive product. XPac Release 2.0 was to have shipped in Q1–92, but due to slow performance and a long test period (4 months) it did not ship until Q3–92.

❸ XPac 2.0 was well received in the marketplace. Sales rose in the second half of '92. But, there have been many problems (anomalies) reported by customers. Nine important customers have not paid their upgrade fee because of these problems.

❹ XPac runs on several different workstation platforms. It can integrate data found in databases on these workstations—and on a mainframe.

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

XPac Implementation Structure



© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

XYZ's Software Engineering Organization

❶ There are four departments in XYZ Software Engineering. Each department develops one of the four major functional areas of XPac.

❷ The size of the departments are (6, 5, 5, and 4). Seventeen of the 20 people are programmers with an average of 5 years experience in software engineering and an average of 10 months at XYZ.

❸ The other three people are tester (one for each department). Two of these testers are temporary contractors and are very junior. They have some experience using accounting packages like XPac. The other tester has 8 years of experience as a programmer and QA engineer. There are 2 open requisitions to hire more testers.

❹ There is an XPac product manager in Marketing. Documentation and customer training are done by Marketing. The Help Desk also reports to Marketing and has 5 people.

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

XPac Process and Infrastructure

- **Leadership:**

  The CEO says that quality is important, but there is no active corporate quality effort at XYZ. The CEO (a former CPA and Chief Financial Officer) talks about the XPac of the future—but few engineers understand or care about what the CEO says.

- **Individual Quality:**

  XYZ tries to hire the very best software engineers. However, they have receive no formal quality training—and only occasional technical training. There has never been a culture survey. The management of XYZ thinks that most employees are very happy to work at XYZ—but turnover rate for the last three years has averaged 16.4%.

- **Teamwork:**

  The executive staff (including you, the Director of Software Engineering) meet every week for 6 hours. Your predecessor (who left the company due to illness) worked 70 hours per week and didn't have time for staff meetings. There is a release team (various people around the company) who meet during final weeks of product release.

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

XPac Process and Infrastructure

- **Planning:**

  There is a release plan for XPac 3.0. It is scheduled for release in Q393. The product manager made the schedule—which was determined primarily by the date the new release was needed in the marketplace. The development department managers did commit to the date—but the senior QA engineer and the senior technical publications person both think that the date will be missed by 2-3 months. Unfortunately, the CEO assumes that by authorizing the hiring of two additional testing people the delay incurred by XPac 2.0 won't occur with XPac 3.0.

  A project management tool was used to do the initial planning for Release 2.0—but its use was abandoned in the middle of the release. There is no formal record of the past estimates or the actuals. XYZ does not use XPac for any strategic planning.

  All employees fill out weekly time sheets, but they are only used by Finance. They always fill in 40 hours/week regardless of how many hours they worked. (Most software engineers work about 50 hours/week—and more during crunch time.)

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

XPac Process and Infrastructure

- **Configuration Management:**

  Developers use a *MAKE*-like tool to keep track of source and object libraries and to do builds. There is no formal integration procedure. There is no configuration management of other project documents (designs, test plans, etc.). There is no product review board.

- **Requirements:**

  No formal requirements process or tools are used. The CEO often decides what will go into the next release. Marketing has conducted some customer meetings to find out what they want. Some customers call in enhancement requests to the Help Desk. These are given to marketing—no one knows exactly what happens to them.

- **Design:**

  No formal design tools are currently used. Typically, each development manager has informal design meeting with their own group to spec out the new features. There is no overall architecture document that shows how all the pieces of the product work together. There have been some design reviews, but no formal inspections.

  The previous development director wanted to get some CASE tools, but there was no agreement on what design methodology to use.

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

XPac Process and Infrastructure

- **Development:**

  Most (95%) of XPac approximately 120K lines of source code are written in C. There are a few COBOL and 4GL routines that run on the mainframe. All the software engineers are experienced C programmers. Sometimes developers share their code with a peer. But, there are no formal code inspections. There are no other code-oriented tools are used except the compiler.

- **Testing:**

  Developers are supposed to unit test their own code. Some do. There are no guidelines, procedures, or tools. The test engineers in each department work together to do a system test—with each one concentrating on their area of the application.

  There are a few documented test procedures. Sometimes the Help Desk people (if they have time) help out testing the product. There are no automated test tools or test suites.

  A developer (who has since left XYZ) built a crude error tracking data base. It is used by the testers and by the Help Desk.

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

XPac Process and Infrastructure

- **Help Desk:**

  There are many calls each day to the Help Desk asking how to use XPac. The users don't seem to read the documentation. And, most of them won't pay for the official training courses.

  There is a backlog of 517 unresolved defects in Release 1.0 and 2.0. The ones that development thinks should be fixed are planned to be in Release 3.0. The Help Desk people try to give the users "work-arounds" to hold them until Release 3.0.

  Some users are angry that they have to wait for Release 3.0. A few have been sent temporary fixes. The CEO made a speech at the 1992 User Convention and said, "Quality and customer satisfaction are very important to XYZ." As a sign of this, they were told that they could call the CEO whenever they had a problem. Some have.

© 1993 ProQual, Inc. (508) 359-7273 & YESS!, Inc. (916) 944-1032

"Advanced Software Inspections"

TUTORIAL "E"

Mr. Tom Gilb
Independent Consultant

Mr. Tom Gilb devoted 80 pages to the earliest book describing software inspection, *Software Metrics* (1976). He has actively taught and installed and followed up the method internationally since 1975. He has consulted to key pioneers of Inspection at IBM development labs. He is the author of *Principles of Software Engineering Management*, and the forthcoming *Software Inspections* (Addison-Wesley, late 1993).

Cover Sheet:

Tutorial Documentation for Quality Week 1993, SF CA

"Advanced Software Inspections"

By

Tom Gilb

Copyright Tom Gilb 1993

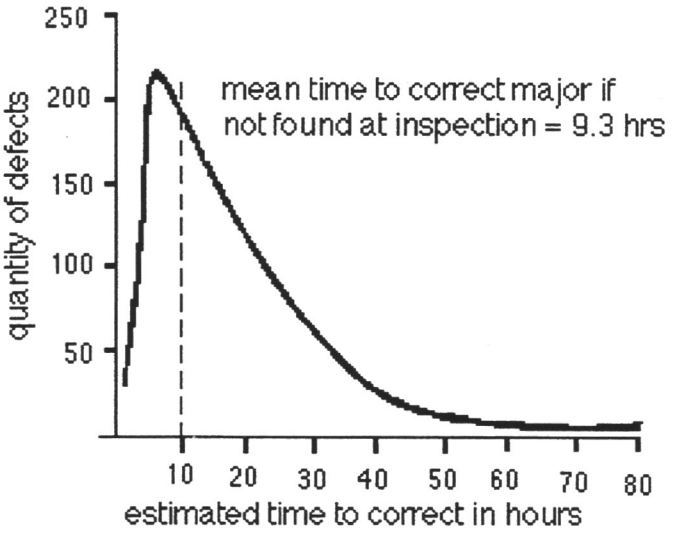
You are welcome to make non-profit use of these slides when proper credit is given.

© Tom Gilb, Iver Holtersvei 2, N-1410 Kolbotn, Norway. Tel. 011-47-66-801697
Extracted from (Summer 93) book "Software Inspections" Addison-Wesley

"Advanced Software Inspections" 1
Process Diagrams 2
The Cost of Quality Curve 2
The Inspection Process TIME COST Chart 3
The Inspection Process Overview Chart 4
The work product of sources and rules 5
Inspection Leader Tasks 6
The Planning Process Components 7
Kickoff Meeting Contents 8
Role specialization during checking 9
The Error to Failure Chain 10
Items Logged Concepts 11
Generating Checklist Questions 12
The Relative Volume of Checking Activity 13
Process Change Flow 14
Standards are 15
A procedure is task definition 16
Rules are ... 17
Rules serve a variety of functions 18
Checklist functions 19
Inspection Standards: Examples 20
Inspection Master Plan 20
Inspection Issue Log 21
Process Brainstorming Meeting Log 22
Change Request 23
Inspection Data Summary 24
Procedure: Leader During Logging 25
A Generic Software engineering Document Rule set. 26
RULES for Document Type: Test Plan 27
A One-page Inspection Handbook. 28
Sample generic entry criteria 29
Generic Exit Criteria 30
CHECKLIST for Document Type: Test Plan. 31
Inspection Policy 32

Process Diagrams

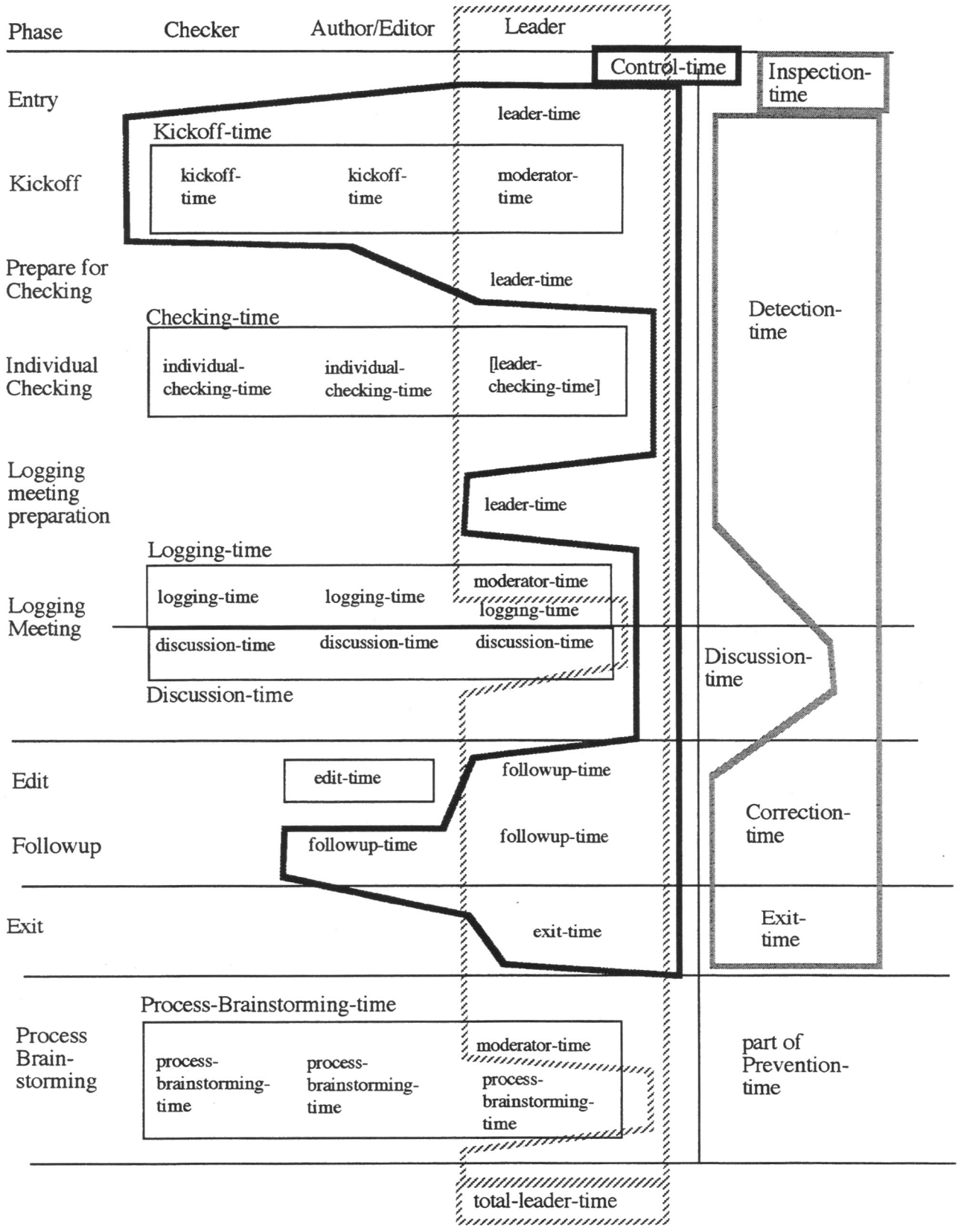
The Cost of Quality Curve



(Source: MEL inspection data, June 1989 - January 1990

Ill. The "cost of quality". The average saving for finding a Major was found to be 9.3 work hours, versus a cost of about one work hour to find and fix a major. Thus Majors save about eight work hours net each.

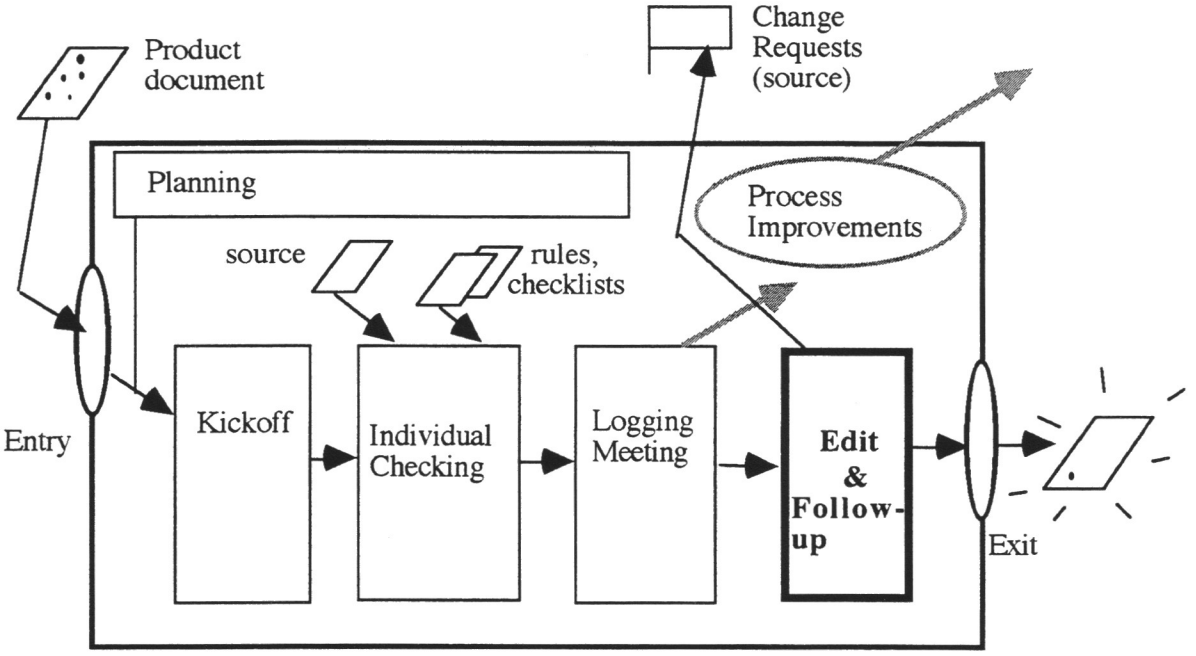
© Tom Gilb, Iver Holtersvei 2, N-1410 Kolbotn, Norway. Tel. 011-47-66-801697
Extracted from (Summer 93) book "Software Inspections" Addison-Wesley



The Inspection Process TIME COST Chart

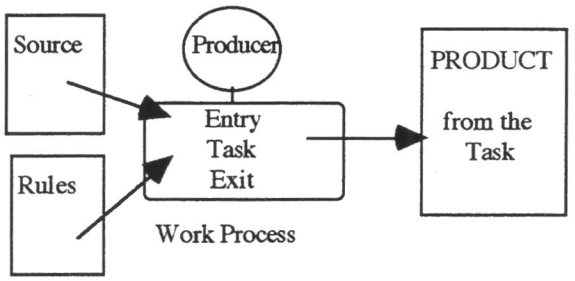
© Tom Gilb, Iver Holtersvei 2, N-1410 Kolbotn, Norway. Tel. 011-47-66-801697
Extracted from (Summer 93) book "Software Inspections" Addison-Wesley

The Inspection Process Overview Chart



© Tom Gilb, Iver Holtersvei 2, N-1410 Kolbotn, Norway. Tel. 011-47-66-801697
Extracted from (Summer 93) book "Software Inspections" Addison-Wesley

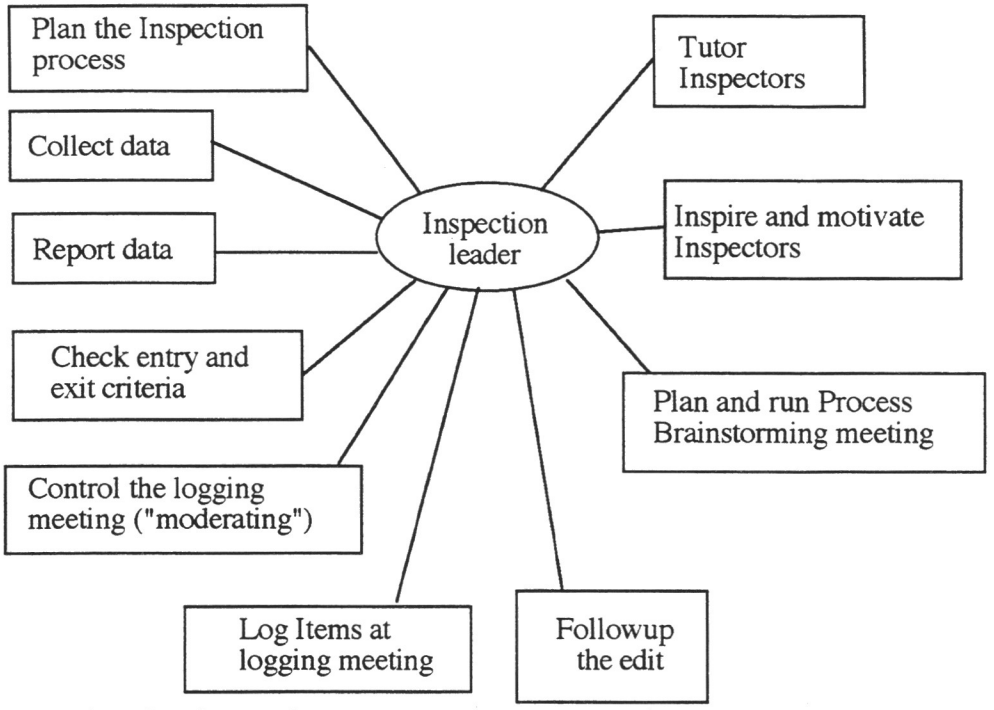
The work product of sources and rules



The Product is the result of the Author doing a process. Checkers check that this process has been done correctly.

© Tom Gilb, Iver Holtersvei 2, N-1410 Kolbotn, Norway. Tel. 011-47-66-801697
Extracted from (Summer 93) book "Software Inspections" Addison-Wesley

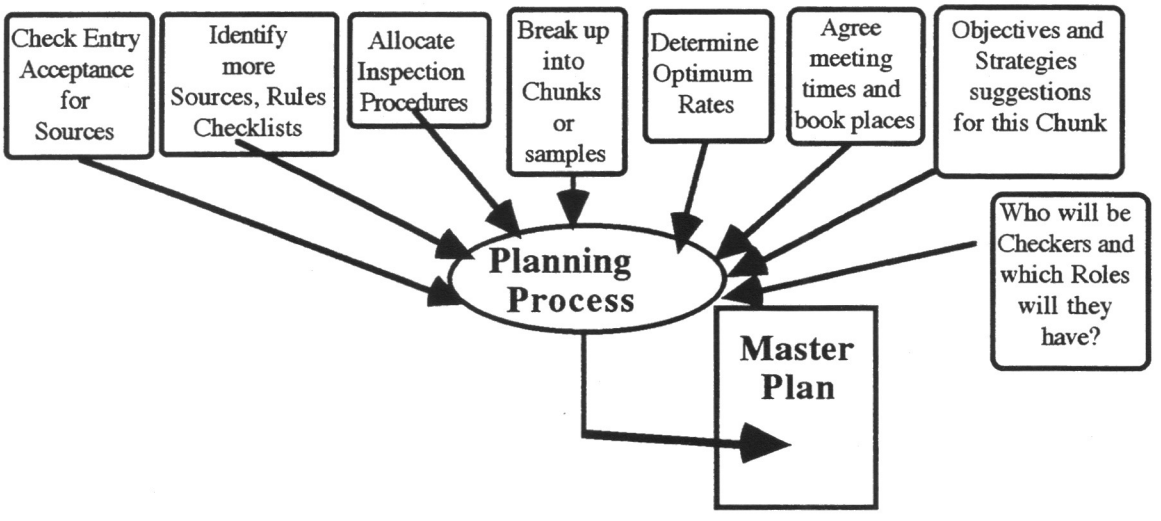
Inspection Leader Tasks



Inspection Leader tasks

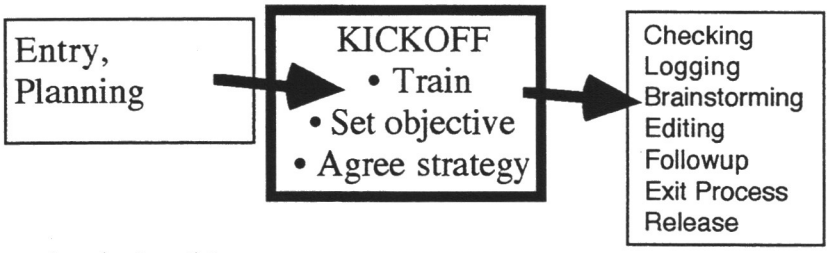
© Tom Gilb, Iver Holtersvei 2, N-1410 Kolbotn, Norway. Tel. 011-47-66-801697
Extracted from (Summer 93) book "Software Inspections" Addison-Wesley

The Planning Process Components



© Tom Gilb, Iver Holtersvei 2, N-1410 Kolbotn, Norway. Tel. 011-47-66-801697
Extracted from (Summer 93) book "Software Inspections" Addison-Wesley

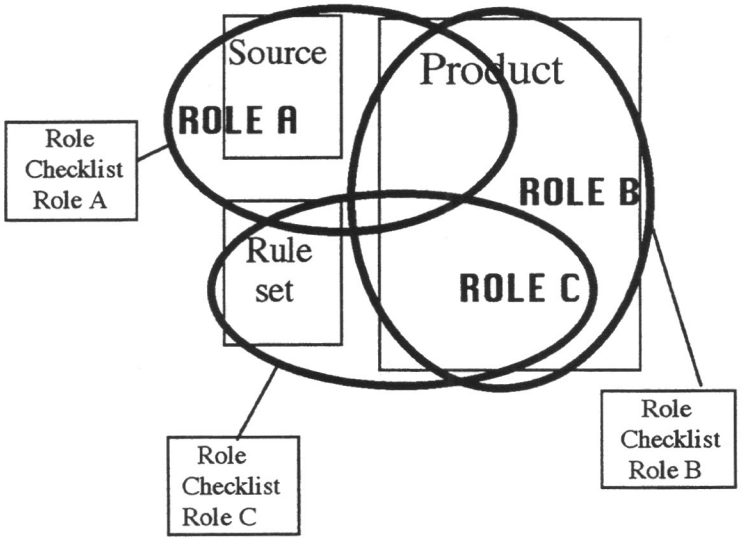
Kickoff Meeting Contents



Kickoff contents and relationships.

© Tom Gilb, Iver Holtersvei 2, N-1410 Kolbotn, Norway. Tel. 011-47-66-801697
Extracted from (Summer 93) book "Software Inspections" Addison-Wesley

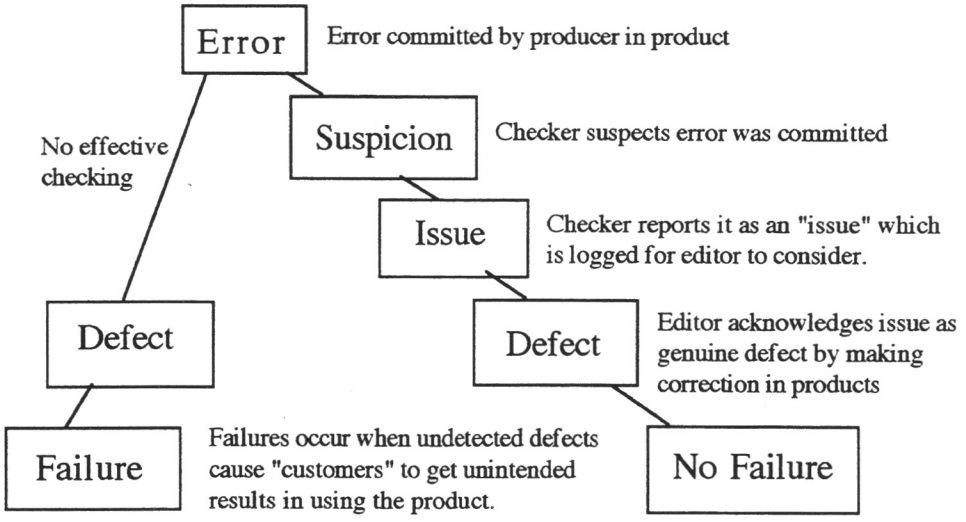
Role specialization during checking



Role allocation is the specialization of checkers in an attempt to improve team productivity.

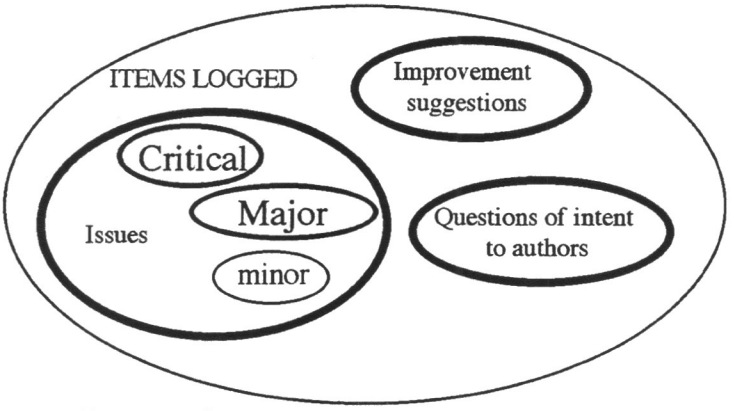
© Tom Gilb, Iver Holtersvei 2, N-1410 Kolbotn, Norway. Tel. 011-47-66-801697
Extracted from (Summer 93) book "Software Inspections" Addison-Wesley

The Error to Failure Chain



© Tom Gilb, Iver Holtersvei 2, N-1410 Kolbotn, Norway. Tel. 011-47-66-801697
Extracted from (Summer 93) book "Software Inspections" Addison-Wesley

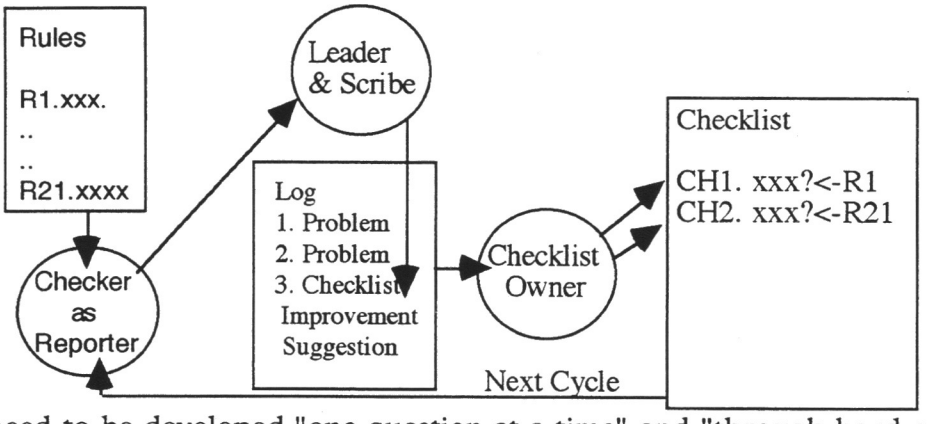
Items Logged Concepts



"Items" is a term covering all those things which a logging meeting writes in the log.

© Tom Gilb, Iver Holtersvei 2, N-1410 Kolbotn, Norway. Tel. 011-47-66-801697
Extracted from (Summer 93) book "Software Inspections" Addison-Wesley

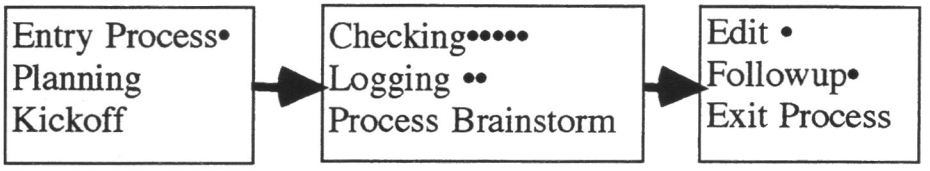
Generating Checklist Questions



Ill. Checklists need to be developed "one question at a time" and "through hard experience".

© Tom Gilb, Iver Holtersvei 2, N-1410 Kolbotn, Norway. Tel. 011-47-66-801697
Extracted from (Summer 93) book "Software Inspections" Addison-Wesley

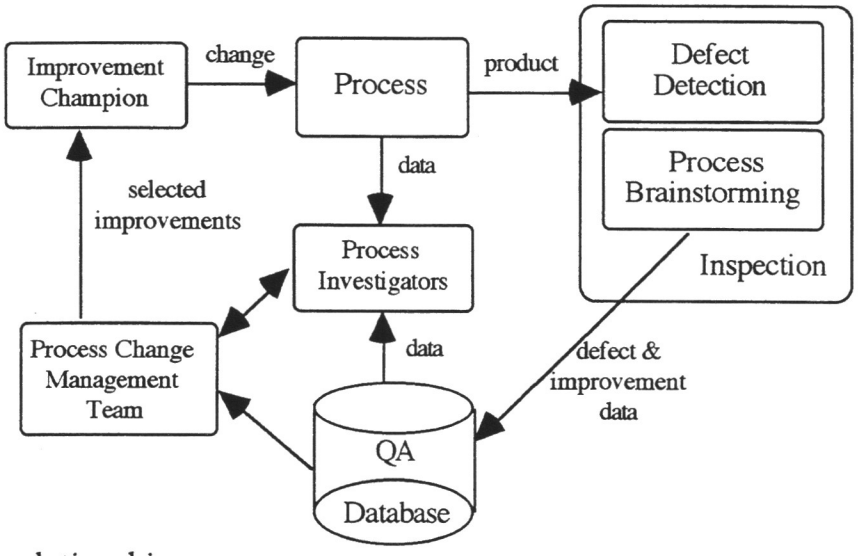
The Relative Volume of Checking Activity



The relative volume of checking activity in sub-processes (•=a little, ••••• = most of it).

© Tom Gilb, Iver Holtersvei 2, N-1410 Kolbotn, Norway. Tel. 011-47-66-801697
Extracted from (Summer 93) book "Software Inspections" Addison-Wesley

Process Change Flow

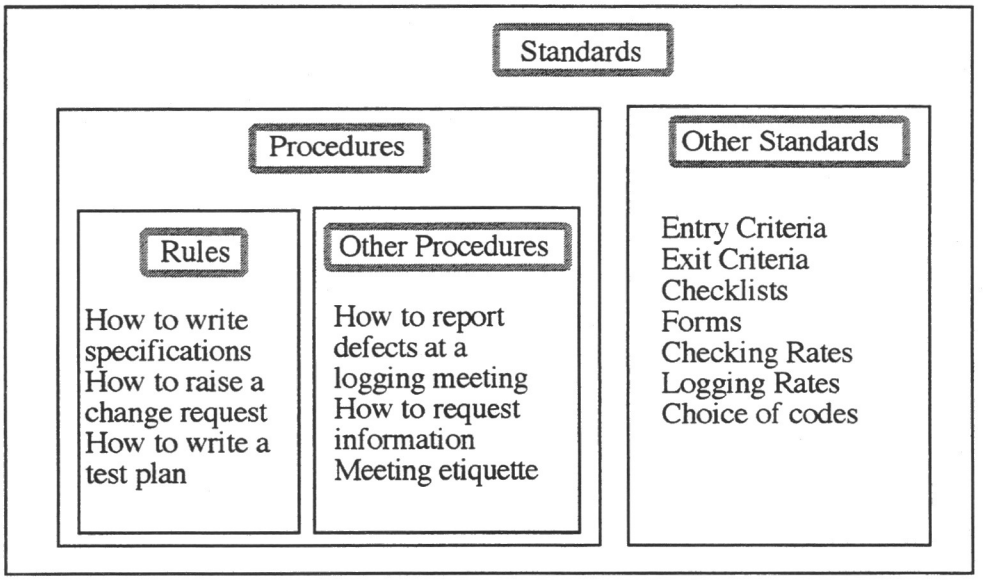


Process Change relationships.

© Tom Gilb, Iver Holtersvei 2, N-1410 Kolbotn, Norway. Tel. 011-47-66-801697
Extracted from (Summer 93) book "Software Inspections" Addison-Wesley

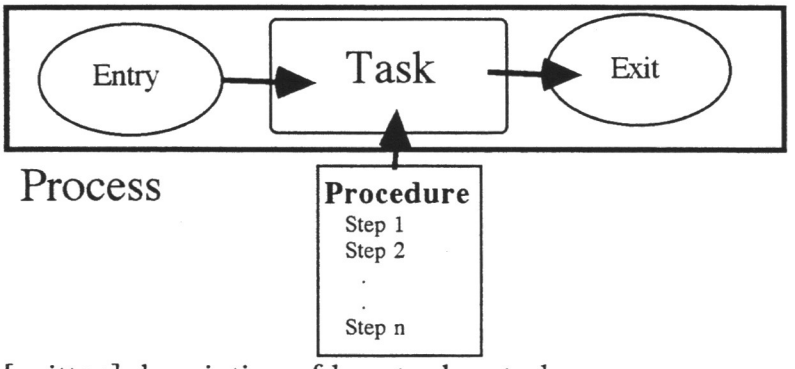
Standards are ....

standard: guidance in the way processes are performed. A standard guides people to do something in a particular way.



© Tom Gilb, Iver Holtersvei 2, N-1410 Kolbotn, Norway. Tel. 011-47-66-801697
Extracted from (Summer 93) book "Software Inspections" Addison-Wesley

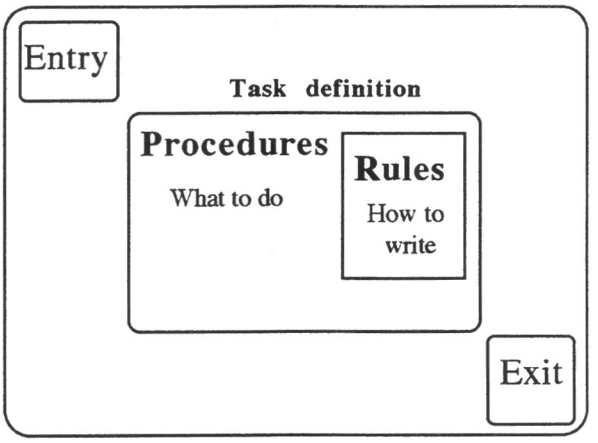
A procedure is task definition



A procedure is the [written] description of how to do a task.

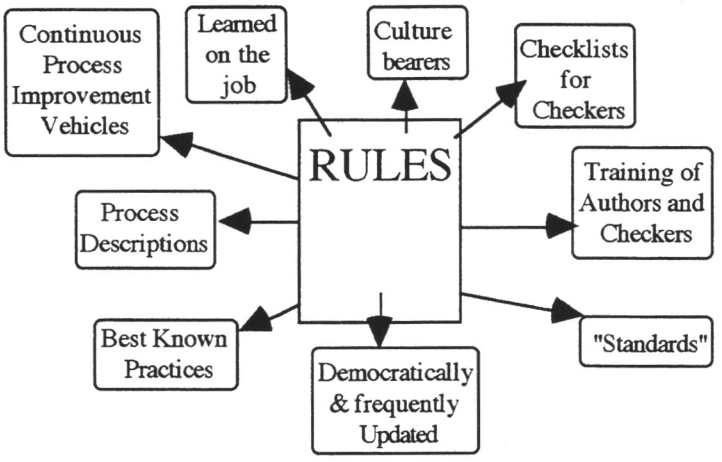
© Tom Gilb, Iver Holtersvei 2, N-1410 Kolbotn, Norway. Tel. 011-47-66-801697
Extracted from (Summer 93) book "Software Inspections" Addison-Wesley

Rules are ...



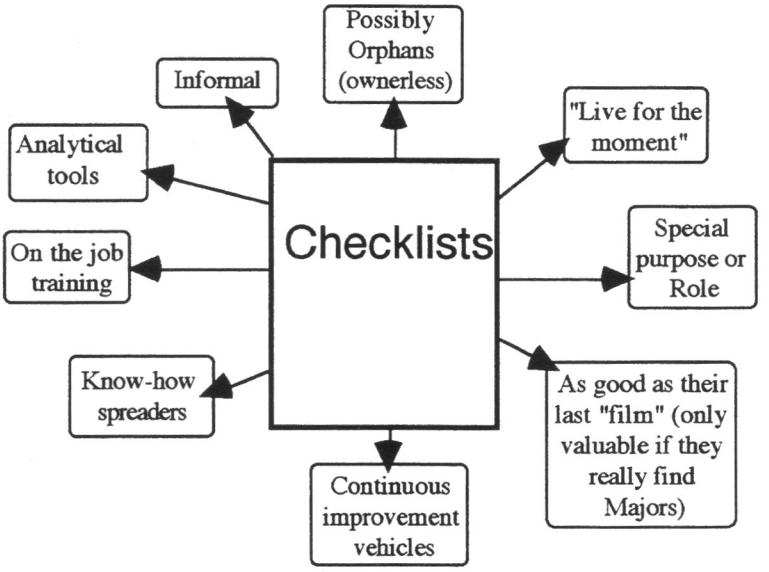
© Tom Gilb, Iver Holtersvei 2, N-1410 Kolbotn, Norway. Tel. 011-47-66-801697
Extracted from (Summer 93) book "Software Inspections" Addison-Wesley

Rules serve a variety of functions



© Tom Gilb, Iver Holtersvei 2, N-1410 Kolbotn, Norway. Tel. 011-47-66-801697
Extracted from (Summer 93) book "Software Inspections" Addison-Wesley

Checklist functions



Checklists have many links the Quality Improvement framework too.

© Tom Gilb, Iver Holtersvei 2, N-1410 Kolbotn, Norway. Tel. 011-47-66-801697
Extracted from (Summer 93) book "Software Inspections" Addison-Wesley

Inspection Standards: Examples

Inspection Master Plan

Inspection Leader: *Rod O'Mator* Tel *6279*.

Author(s): *Ginny Pigg* Date Inspection requested *11/June/93*.

Product *Graphical Obj Module Des GD.MAINSYS.GRMOD V2.0* No. pages *15* Status *Not Inspected*.

Entry Criteria which apply *Vol., Prod Mini-Inspected, Sys. Des Insp'd Waived: leader cert., Test Plan Insp'd*.

Current Entry Status *Passed 13/June/93*.

Exit Criteria which will be applied *Edit complete, CR raised, rates ±10% Standard, estimated remaining defects <= 1/pg.*

Meetings

Kickoff : Date *16/June/93* Location *Conference Room C9* Start Time *10:30* End time *11:00*.

Logging: Date *22/June/93* Location *Conference Room C4* Start Time *10:00* End time *12:00*.

Process Br: Date *22/June/93* Location *Conference Room C4* Start Time *12:00* End time *12:30*.

Documents

Sources (relevant pages) *Graphic System Design (pp 24 - 26), System Test Plan (3.7 - 3.12)*.

Product document(s) samples or chunk(s) *Graphic Object Module Design, Vers. 1.3, 11/6/93, CHUNK pp 1 - 4*.

Rules, procedures and standards *GIS: Graphic Interface Standard, MDRS: Module Design Rule Set*.

Checklists: For Tasks: *GDC: Graphic Design Checklist* For specific Documents: *DC: Des Chlst, SDC: Sys. Des Chlst.*

Participants: Name / tel.	Document Role Tags	Checklist or Procedure Role Tags	Inspection Procedure Tags
1 *Ginny Pigg*	RSGD: Rule Set	CI: Interfaces, PC3	PAK, PAC, PAL
2 *Penny Lane*	GDS: Source	CT: Tester	PCK, PCC, PCL
3 *Ivor Heddaicke*	GD: Graphic Des	CC: Coder	PCK, PCC, PCL
4 *Francis French*	GDT: Test Plan	CU: User	PCK, PCC, PCL
5 *Alex Shufflebottom*	GSD: System Des	CMA: Accuracy	PCK, PCC, PCL
6			

Standard Rates and Estimates

Individual checking rate: *1* pages per hour of product document
Logging meeting rate: *1.5* pages/hour
Estimated effort in individual checking: *4* work-hours (to nearest tenth)

Data Collection (by you the checker)

Actual work-hours spent *3.6* No. any pages studied (at optimum rate) *4*
No. any pages scanned (not at optimum rate) *6* No. pages used *19* (excluding master Plan, this page
No. items identified: Major Issues *16* minor Issues *25* Improvements *3* Questions *8*

© Tom Gilb, Iver Holtersvei 2, N-1410 Kolbotn, Norway. Tel. 011-47-66-801697
Extracted from (Summer 93) book "Software Inspections" Addison-Wesley

Inspection Issue Log

Inspection Identification *GD5* Page: *1* of *9* .

No	Document Reference Tag	Page	Line No./ Location	Type of Item (circle one)	Checklist/ Rule Tag	Description (if needed) (key offending words)	Editor Note
1	GRMOD 2.0	1	4	Major Imp minor ?	CI4	"interface spec"	
2	"	"	7	Major Imp minor ?	CT3		
3	"	"	15 - 17	Major Imp minor ?	CU2	New standard User Guidelines not followed	
4	"	"	6	Major Imp minor ?	GDC5		
5	GDT	"	-	Major Imp minor ?	GDT	Test Plan format would be better corresponding with GDC	
6	STP	2	3.7-12	Major Imp minor ? *NEW*	-	Test missing	
7	GRMOD 2.0	2	5	Major Imp minor ?	CMA 7	"response time"	
8	"	"	13	Major Imp minor ?	SDC 1	Wrong format	
9	"	"	44 - 46	Major Imp minor ?		What does it mean?	
10	"	3	25	Major Imp minor ?	GDC 6		

Subtotals:
New issues found in meeting: *1* .
Major issues logged: *6* minor issues logged *2* .
Improvement suggestions logged: *1* Questions of intent logged: *1* .

© Tom Gilb, Iver Holtersvei 2, N-1410 Kolbotn, Norway. Tel. 011-47-66-801697
Extracted from (Summer 93) book "Software Inspections" Addison-Wesley

Process Brainstorming Meeting Log

Date of process brainstorming meeting *22/June/93* Start time *12:30* .

Software Inspection Identification(s): *GD5* Inspection Leader *Rod O'Mater*

.

Candidate Document Reference *GD.MAINSYS.GRMOD Graphical Object Module Design V 2.0*

.

Other Document References *Graphic System Design (pp 24 - 26), System Test Plan (3.7 - 3.12)* .

Brainstorm Item No.	Issue Ref	Classification of Cause (tick)	Root Cause (keywords)	Improvement Suggestions (keywords)
1	6	Communication ->Oversight Transcription Education	different type of test only needed for graphic designs	Add checklist question to GDT
2	3	Communication Oversight Transcription -> Education	New User guidelines not widely known	Publicize in newsletter Memo to all graphic designers
3	7	->Communication Oversight Transcription Education	Old interface conventions not updated in current manual. Interface names very similar	Audit status of manual before beginning task Update project leader task Add checklist question CRS
4		Communication Oversight Transcription Education		
5		Communication Oversight Transcription Education		

© Tom Gilb, Iver Holtersvei 2, N-1410 Kolbotn, Norway. Tel. 011-47-66-801697
Extracted from (Summer 93) book "Software Inspections" Addison-Wesley

Change Request

Date of Change Request *24/June/93* Change Request Identification *GDI.54*

To: Name *Peter Principal*.

Location *Room 1106, Copse Building*.

From: Name *Alex Shufflebottom* Tel. *Ext 485*.

Location *Room 46B, HQ Building*.

Change Description
Document in which the change is requested *GD.STP V3.2 System Test Plan*

Details of the requested change

Add tests for the graphic objects being overlaid,

with combinations as in Section 3.8 for movement.

Authorized by *Thea Stickler, Test Manager* on (date) *24/June/93*.

Response (please tick)

Details of action required from document owner - reply required by (date) *2/July/93*.

O Change accepted. Date scheduled _____ Date performed _____.

O Change not accepted. Detail of reasons why it was not accepted.

O I am not the document owner. Document owner is Name _____

Location _____ Tel. _____

O I have forwarded a copy of this Change Request to the actual owner.

Time spent
I have spent _____ work-hours on this Change Request. Date _____

© Tom Gilb, Iver Holtersvei 2, N-1410 Kolbotn, Norway. Tel. 011-47-66-801697
Extracted from (Summer 93) book "Software Inspections" Addison-Wesley

Inspection Data Summary

Date *22/June/93* Software Inspection ID: *GD5* Inspection Leader *Rod O'Mator*.

Candidate Document Reference *GD.MAINSYS.GRMOD Graphical Object Module Design V 2.0*.

Date Inspection requested *11/June/93* Date Entry criteria passed _____.

Leader Planning Hours *1.5* Entry Time _____hours(tenths)

Individual Checking Results (to be reported during entry process for Logging meeting)

Inspector	Checking Hour (t)	Pages studied(P)	Major issues	minor issues	Improvements	Questions noted	Checking rate P/t
1	3.6	4	16	25	3	8	1.11
2	1.9	4	7	23	0	2	2.11
3	2.8	3.5	20	14	5	0	1.25
4	4.2	5	9	44	1	12	1.19
5	2.4	2.6	15	21	1	19	1.08

Checking-time (total) *14.9* Checking rate *1.28* average P/t)

Logging

Number of people 6 Meeting duration (hour) 1.7 Logging-time 10.3 hours

Major issues logged	minor issues logged	Improvement suggestions	Questions of intent	New items found in the meeting
27	30	8	22	3

Logging rate *.84* (Items/minute) Detection time(Plan+Kickoff+Check+Log) *27.7*.

Editing, Followup and Exit

No. Major defects 29 No. minor defects 54 No. Change Requests 3.

Edit-time 16.6 Followup time 1.5 Exit-time 6 Exit date 1/July/93.

Control-time 4.6 Inspection-time 46.4 Estimated remaining defects 7 Maj..

Effectiveness (%defects found/page) 80 % Efficiency (Maj./hr) 0.6.

Development time saved 191 hrs..

© Tom Gilb, Iver Holtersvei 2, N-1410 Kolbotn, Norway. Tel. 011-47-66-801697
Extracted from (Summer 93) book "Software Inspections" Addison-Wesley

Procedure: Leader During Logging. PLL

1. • Entry to Logging;
2. • gather individual checking data
3. • record it on data summary sheet
4. • evaluate if it is worth holding Logging meeting
5. • cancel meeting if necessary.
6. • Remind team of Kickoff objectives and strategy agreed earlier.
7. • Decide and announce a recording sequence and content (Majors only, Sources first for example).
8. • Assign logging task or take it on yourself.
9. • Remind Author to validate the written Log, and seat accordingly.
10. • Begin Logging process.
11. • Make sure that unique Majors not in checklist get evaluated for inclusion in an updated checklist.
12. • Keep recording pace high (one to four Logged per minute).
13. • Stop discussions, defensiveness: focus on Logging.
14. • Have fun, joke, help people to learn and enjoy.
15. • Announce results, in relation to Kickoff objective, at end.
16. • Decide how to handle lack of time;
17. • reschedule continuation
18. • re-chunk the remainder
19. • Consult with Author. Is this sample enough?

© Tom Gilb, Iver Holtersvei 2, N-1410 Kolbotn, Norway. Tel. 011-47-66-801697
Extracted from (Summer 93) book "Software Inspections" Addison-Wesley

A Generic Software engineering Document Rule set.

G1 (DEF.) A "Software engineering Document" is the Product of any Software engineering related task. It includes code, test cases, requirements, contracts and user manuals.
G2 (ONE) All documents shall be unambiguous to the intended readership, unless clearly marked (for example using the <angle parenthesis>).
G3 (NOTE). All form of comment, note, suggestion, idea which does not form an official part of the document shall be clearly distinguished as such (for example by "quotes", *italics*, footnotes· NB: prefacing words.)
G4 (EXTRA). All documents shall be as brief as possible, to support their purpose.
G5 (CLEAR). All documents shall be crystal clear to all possible readers as to intent. The burden is on the producer not the reader.
G6 (HEAD). All documents shall contain producer, Date, Page number, Source document codes, Rule set codes, quality status (Exited or not), identification tag, revision code.
G7 (UNIQUE). Ideas shall be stated <u>once only</u> in documents and thereafter referred to by their unique tag. Use comments (") to paraphrase ideas elsewhere.
G8 (SOURCE). Statements shall contain information about their exact and detailed sources. *Normally use the "<-" source arrow., but also "evidence" and comments.*
G9 (EL). Statements shall be broken up into their most elementary form *(to permit separate analysis, costing, implementation).*
G10 (TAG). All elementary statements shall have their own identity tag for direct reference from other parts of any larger document set. *Parameter name and qualifiers can be used as sub-tags. e.g. USABILITY.PAST[1991].*
G11 (HIER). Documents shall be organized into clear hierarchical structures, with tagging reflecting the structure.
G12 (CHANGE). Date and unique producer initials (or employee code) of any change shall be indicated at the level of change. *e. g.*
PLAN 60%<-{REQT-R, Tsg, FEB-15-91} "Inspection rework"
G13 (RISK). Any known or suspected uncertainty or risk shall be clearly indicated using devices such as { <vaguely defined>, ?, ??, 60->70, 70%± 20 } and suitable comments.
G14 (STATUS). A document shall show its Quality Improvement status clearly. In particular EXITED (from the Inspection process), NOT EXITED. Not Exited shall be assumed until other status is claimed. The number of <u>maximum Major Problems probably remaining per Non-commentary Full Page</u> shall be noted next to the Exit declaration.

© Tom Gilb, Iver Holtersvei 2, N-1410 Kolbotn, Norway. Tel. 011-47-66-801697
Extracted from (Summer 93) book "Software Inspections" Addison-Wesley

RULES for Document Type: Test Plan [1]

Document Tag: "TP".

Owner: DG Exited (Max. Mj=1/pg.)

TP1. Specify Test Plan Identifier as SYS.TP.n, where SYS is the System Name, and n is the sequential test plan number.

TP2. Introduction: Summarize software items to be tested with references to project plan, quality plan, and test design documents.

TP3. Identify test items including version level, hardware and software requirements, and references to requirements specifications and design documentation.

TP4. List features to be tested and features not to be tested (with reasons)

TP5. Describe test approach to be used, including techniques and level of comprehensiveness.

TP6. Specify item pass/fail criteria, and person responsible for the decision.

TP7. Specify test suspension and resumption criteria.

TP8. Specify test environment, including tools required, user involvement, and special equipment

TP9. Specify schedule, resources required, responsibilities, and contingencies.

---

[1]Adapted from ANSI/IEEE Standard for Software Test Documentation, Std 829 - 1983.

© Tom Gilb, Iver Holtersvei 2, N-1410 Kolbotn, Norway. Tel. 011-47-66-801697
Extracted from (Summer 93) book "Software Inspections" Addison-Wesley

A One-page Inspection Handbook.

1. The Inspection process is always managed by a trained and certified Team Leader.
2. The Leader is responsible for managing the process in all respects for productive results.
3. The first objective of Inspection is to identify and correct Major defects.
4. The second, but most important, objective of Inspection is to identify and remove the source of defects.
5. The fundamental measure of Inspection success is the quality-to-cost ratio of the total development and service life cycle.
6. Short term measures include Major Issues found per work-hour used, and defect causes identified and treated.
7. The productivity measure of Inspection is the net hours saved, estimated statistically, due to defects found and removed earlier than they otherwise would be.
8. ENTRY. The Inspection process is only entered when a specified set of entry criteria have been met.
9. PLANNING: The Leader selects a set of Source documentation, candidate documentation, checklists, rule sets, work rates, people, roles, and Logging Meeting rates to ensure maximum productivity.
10. KICKOFF. The Leader can elect to run a "KICKOFF" meeting prior to checking. Team improvement goals and corresponding strategies shall be adopted. Necessary material is taught.
11. CHECKING. A checking phase has a recommended time or rate, but checkers have instructions to deviate from that whenever individual ability, Role or situation dictates, in order to increase productivity
12. The objective of individual checking is to identify a maximum of unique Major Issues which no other checker will bring to the Logging meeting. To do this each checker should have at least one special "Role".
13. LOGGING MEETING: the team concentrates on Logging, in writing, Major issues at a rate of at least one per minute. The Leader permits little other verbal meeting activity. Meetings last a maximum of two hours (human tiredness limit) at the known optimum rate. If necessary, work must be chunked to avoid tiredness. Optimum rate time for the meeting is planned & used to aid issue-finding by additional checking effort.
14. EDIT. Issue analysis and correction action is undertaken by an Editor. Some written action must be taken on all Logged issues - if necessary by sending change requests to other Authors.
15. EDIT FOLLOWUP. The Leader shall determine that some appropriate written action has been taken on all logged issues. The Leader is not responsible for the correctness (the Editor is).
16. EXIT. The Leader determines whether the formal exit criteria have been met before signing off completion of the Inspection . These include followup completed, statistics delivered, planned rates kept to, and level of remaining issues within acceptable bounds.
17. PROCESS BRAINSTORMING. Immediately after each Logging meeting, time is used to brainstorm the process causes of Major defects, and to brainstorm Improvements to remove these causes. These opinions are stored in the QA database for the Process Change Management Team. This meeting shall last no more than half an hour. The finishing time will be announced. The objective shall be to maximize production of useful ideas and personal commitment to change within that time.

© Tom Gilb, Iver Holtersvei 2, N-1410 Kolbotn, Norway. Tel. 011-47-66-801697
Extracted from (Summer 93) book "Software Inspections" Addison-Wesley

Sample generic entry criteria

Here is an example of basic generic entry criteria:

Generic Entry criteria Tag=GE
GE1: The Author must agree to let the process proceed at all sub-stages of Inspection.
GE2: The Leader must agree to let the process proceed at all sub-stages of Inspection.
GE3: All Source documents are in writing and have successfully exited with a known level of maximum remaining defects. Failing this they will be mini-checked or sampled to determine their defects level before proceeding. Failing this they will be clearly marked as NOT EXITED on all pages.
GE4: The applicable set of generic and Specific rules for the Task which produced the Product are available in writing.
GE5: A formal Master Plan has been made for the rest of the checking Task where the Optimum Recommended checking Rate has been established and it is not faster than about one non-commentary full page per checking -hour.
GE6: The Leader has been trained for several days (3 theory, 2 practice) and is officially Certified as an able practitioner.
GE7: A cursory examination (<5 minutes) of a sample (page or so) of the Product document shows less than one Major per page findable. (Otherwise the Author cleans up)
GE8: Ordinary text documentation shall have been cleaned up by a spelling checker before submission. Other possible machine checks or diagnostics have been done (clean compile, cross referencing, indexing, grammar check).
GE9: The Author (or editor) agrees to participate as a checker.

© Tom Gilb, Iver Holtersvei 2, N-1410 Kolbotn, Norway. Tel. 011-47-66-801697
Extracted from (Summer 93) book "Software Inspections" Addison-Wesley

Generic Exit Criteria

GENERIC exit criteria:
GX1. EDIT COMPLETE: all edit work completed according to edit rules; validated by Leader at followup.
GX2. POTENTIAL CHANGE CONTROLLED: all necessary change request cross references are inserted in the candidate Product.
GX3. Do Author/editor and Leader agree to release the document?
GX4. OPTIMUM RATES. Was the deviation from optimum rates of Checking and Logging more than 30% or enough to raise the question of whether the Inspection was effective in finding the defects which would be found at a more optimum rate?
GX5. REMAINING DEFECTS: do the estimated remaining defects per page exceed a specified limit (default 0.25 Majors per page[2] maximum)? Compute remaining defects assuming that you have minimum 30%, maximum 50%, finding effectiveness. Assume 1 of 6 fixes fail.
GX6. STATS: have all statistical data been entered into the QA database?
GX7. All chunks have exited before document as whole can exit .

[2] This is an example of a limit at one part of IBM. See Watts Humphreys, Managing the Software Process, Addison Wesley. It is certainly too strict a criteria for beginners. Try 2.5 Majors remaining as an Exit criteria as an easier initial criteria. Ten times looser than 0.25 but much better than what you probably do today.

© Tom Gilb, Iver Holtersvei 2, N-1410 Kolbotn, Norway. Tel. 011-47-66-801697
Extracted from (Summer 93) book "Software Inspections" Addison-Wesley

CHECKLIST for Document Type: Test Plan.

Document Tag:CTP Owner:DG

Exited (max. 0.25 Mj/Pg).
CTP1. Does the Test Plan meet the rules on all counts?<-Generic Rule G1
CTP2. Do you fully understand the Test Plan ?<-Generic Rule G2
CTP3. Do you fully understand your Source documents?<-Generic Rule G2
CTP4 . Do you fully understand the checklist and the rules?<-Generic Rule G2
CTP5. Are there any inconsistencies between the Source documents and the Test Plan?<-Generic Rule
CTP6. Are there any internal inconsistencies in the Test Plan?<-Generic Rule
CTP7. Have all cross-references and versions been specified?<- {G6, G8}
CTP8. Do all given cross-references actually exist?<-G8
CTP9. Are all related company policies or directives met here? (should be supplied by Leader!)<-G8

© Tom Gilb, Iver Holtersvei 2, N-1410 Kolbotn, Norway. Tel. 011-47-66-801697
Extracted from (Summer 93) book "Software Inspections" Addison-Wesley

Inspection Policy

Version Nov 24 1992 TG. Tag=POL.I

IP1. All 'documents' costing over one work-hour to produce, or which affect one work week of cost, shall be "exited" correctly from Inspection before further use by us.

IP2. All paperwork processes shall be defined by written 'owned' rules and procedures, entry criteria and exit criteria.

IP3. Default exit condition for maximum remaining major defects per page is three [for this year] and 0.3 [three years from now].

IP4. Optimum checking and logging meeting (including further checking) rates shall be determined and used.

IP5. Inspection team leaders will be trained[5 days] and "certified", to permit them to take responsibility for Inspections. They shall then be continuously monitored for reasonable performance by their statistics.

IP6. Quality Assurance will provide all support functions for Inspection [training, database, audit, certification, forms, ownership of standards by default).

IP7. Defect prevention through process improvement will be given priority. [Kickoff Objectives & strategies, Quality Assurance database, Process brainstorming, Process Change management teams].

IP8. Inspection will be applied "upstream" first in development, and to maintenance and change control.

IP9. Inspection will be subject to regular, no less than annual, process audit.

IP10. Managers will lead by using Inspection on all their important documents (objectives, budgets, strategies, policies, presentations to the board and the like).

© Tom Gilb, Iver Holtersvei 2, N-1410 Kolbotn, Norway. Tel. 011-47-66-801697
Extracted from (Summer 93) book "Software Inspections" Addison-Wesley

"Formal Methods and Testing"

TUTORIAL "F"

Dr. Micheal Mac-an-Airchinnigh
K & M Technologies

Dr. Micheal Mac-an-Airchinnigh is Technical Director and co-founder of K&M Technologies Limited, an Irish Consultancy Company which specializes in Software Engineering, with particular emphasis on Process Modeling, Requirements Engineering, and Formal Methods. He has done consulting for Motorola and for Erricson Eurolab. He is also a Lecturer in Computer Science, Faculty of Engineering and Systems Sciences, in the University of Dublin, Trinity College, where he is responsible for the development of the Irish School of VDM, a model-theoretic formal method. In addition to formal methods, he lectures in compiler theory and practice, intelligent Computer-Aided Design, computer graphics and computer geometry. Dr. Mac-an-Airchinnigh holds a B.S in Mathematics (University of London), M.S in Computer Science and a Ph.D. in Computer Science (both from the University of Dublin).

Formal Methods & Testing

Mícheál Mac an Airchinnigh
Department of Computer Science
Trinity College, Dublin, Ireland
e-mail: Micheal.MacanAirchinnigh@cs.tcd.ie

25th May 1993

Abstract

The tutorial addresses one of the key technologies—formal methods—that is reaching maturity in the software engineering domain, even to the extent that developers of safety critical systems may be open to special litigation in the event of accident/failure, if it can be shown that they have not taken *all reasonable precautions* in building their system. The use of a formal method does not obviate the need for testing, but it can serve as an innovative framework for better testing.

Requirements/specification bugs are the most pernicious of all. The methods by which they may be detected, at the phase in which they occur, have not received the degree of attention they deserve. To redress this serious imbalance, evident from the literature, we dedicate ourselves exclusively, in this tutorial, to the use of applied constructive mathematics, strictly within the framework of an *Aformal Method*, the Irish School of the Vienna Development Method (*VDM*$^{\clubsuit}$), to the problem of bug detection at the requirements/specification level.

1 Prologue

The time is rapidly approaching, assuming that it is not yet already at hand, when a system failure traced to a software fault, and injurious to a third party, will lead to successful litigation against the developers of said system software. The litigation will be successful in spite of any of the usual software disclaimers currently in vogue and in spite of the intensive system and software testing currently undertaken by responsible organisations and developers. How could that be possible? What factor or factors might an attorney use in pressing the claim for compensation?

We will assume that the defendant, the development organisation, has in place an appropriate software development process, that adherence to the process has been assured by a vigorous quality programme, and that the best in testing techniques

have been applied. We will further assume that the developers have been 'certified', in some sense, to be software engineers. How then could the defendant possibly lose?

Unfortunately, the smart attorney for the claimant may politely inquire to what extent the defendant has made **use of formal methods** in the software development and testing processes, and could the defendant please supply the appropriate documentation to support her/his claim.

Attorney for the defence, inadequately briefed by the defendant, will, of course, plead that formal methods are not mature enough, that it is impossible to prove large system programs correct, that there are not enough instructors or courses in formal methods, and that the universities and other third level colleges do not include it in their curricula.

Attorney for the claimant will counter-attack with a charge of negligence, that there is sufficient material in the public domain, and has been for some time (since **circa** 1978), that the defendant ought to have, at least, made the effort to deploy formal methods in the organisation, and that the defendant is negligent to the extent that no effort was made to engage formal methods in the development and testing processes. Furthermore, our attacking attorney will rhetorically ask in what respect the software engineers can lay claim to that title and be ignorant of the mathematics of their field, or being knowledgeable, how could they be so negligent as to ignore what they know.

The final line of defence, in all probability, will be recourse to a plea that the pressures imposed by the **deadline**, the date of system delivery/handover, prohibited any 'experimentation' in formal methods. To which the defence, well-versed in the origins of words, will reply with a smirk: Whose deadline? That of the developers or the claimant who suffered from the negligence?

Lest it should be thought that the scenario, just outlined above, is nothing more than an 'attention-getter', then perhaps the formal methods debate, raging in the UK, may provide further food for thought [15]. Of particular note is the statement that

> "As a result of 00–55 [*The Procurement of Safety Critical Software in Defence Equipment*, UK MoD Defence Standard], formal methods cannot now be ignored as an option in the safety engineering tool-kit, an option moreover whose exclusion from—rather than inclusion in—the design process, must now be justified."[15] p. 269.

1.1 Formal Methods

There is an extraordinary wide variety of formal methods available to the software industry today, each of which requires a committment to a specific philosophy, is distinguished from another by notation, method and field of application, and which

needs for its continuing survival, a School of adherents which in turn has many of the attributes of a living organism, the most salient of which is, perhaps, self-perpetuation with a necessary inbuilt evolutionary mechanism which maintains its vitality and vibrancy.

But what are we understand by the term 'formal method'? The concensus, in Europe at least, is that it denotes both a (specification) language and a method that is *rooted intrinsically* in mathematics and logic (and there **is** a difference between these two).

The qualifier *formal* is a particular twentieth century word that calls for a committment to a very specific philosophy, the roots of which may be discerned in Hilbert's programme to formalise mathematics and in Russell's attempt to reduce mathematics to a body of formal logic, both of which failed utterly, and were subsequently shown to be theoretically impossible.

The subsequent work of Gödel, Kleene, Church and Turing, *inter alia*, heralded in a new era of formality which is today completely dominated by the computer scientists, one of whose first great achievements was the dual theories of *Formal Languages* and *Automata*, theories upon which depend all of our programming language technology depends.

The word 'formal' is derived from *form*, which is, of course, in opposition to *content*. The formalists seek to present a system that is, of itself, inherently independent of and devoid of meaning. An accessible account of the rise of formality in mathematics is given by Morris Kline in the last chapter of his famous work on *Mathematical Thought from Ancient to Modern Times* [9] p. 1182–211.

The form/content duality is also to be found in the syntax/semantics duality. Having mastered the theory of syntax and having built the machinery to process it, the formalists turned their attention to its dual, the theory of semantics—the meaning of the formal language. Therein was born the formal method that we know today.

It seems to the author, whose opinion is undoubtedly biased, that **the** first formal method of computer science [1], ready bedecked with mathematics and logic, was the Vienna Development Method (*VDM*), the specification language of which, replete with a suitable mathematical symbolism, was punningly called *Meta–IV*. That it deserves the name formal method may be further justified by the industrial concerns of its originators and the continued industrial concerns of all who espouse it. That is to say, the term 'formal method' has now acquired a connotation of industrial relevancy.

There are essentially two distinct classes of formal method:

1. the model-oriented formal methods, often called model-theoretic methods, of which the *VDM* [1] [2], and $\mathcal{Z}$ [17] [4], are, perhaps the most widely known exemplars, being also one must add, the most widely used in (European) industry;

2. the rest, which includes those based on axiomatics, the process algebras, etc.

The author begs the tolerance of all the other authors whose works have not been cited, and of the researchers and developers whose methods, being of equal importance, in their own way, to both the *VDM* and $\mathcal{Z}$, have not even been mentioned by name. **They** will appreciate the quandry. The formal literature is vast indeed.

Due to its longevity, as measured relative to the computer science era, it is not surprising that evolutionary mechanisms have forced the development of *VDM* dialects, each of which has its School. For the interested reader, a brief overview of these *VDM* dialects is given in [16] p. 12–5.

In this tutorial we shall present the *VDM*$^{\clubsuit}$ [10] [11], which is based on applied constructive mathematics and which relies on ordinary algebra as a basis for conducting proofs.

With the passing of time, and with experience gained in applying the method, we have become accustomed to think of our work as an **A**formal Method, where the Greek prefix 'α-' has the same meaning of negation that is found in common words such as **a**tom, **a**theist and **a**gnostic. That the *VDM*$^{\clubsuit}$ is within the formal methods community, is guaranteed by its philosophy of constructibility.

In addition, we strive to build a mathematics with which an engineer would feel comfortable. We have noted that formal logic is not used, in practice, by engineers, scientists, or mathematicians. We do not believe that computer scientists or software engineers are so distinguished by their field, that, of necessity, they must commit themselves to formal logic.

That said, we must confess, that even for those who are *au fait* with mathematics, such as engineers and scientists, the mathematics of the *VDM*$^{\clubsuit}$ and, especially its notational accoutrement, comes something of a shock, initially. On the other hand, computer scientists, software engineers, and programmers [**not** coders!], i.e., whose very world and culture is, to some extent, determined by the instrinsic duality of machine and language, and by the absolute necessity of constructibility, will find their thoughts and concepts reflected by and abstracted within the applied constructive mathematics that characterises methods such as the *VDM*$^{\clubsuit}$.

We do not wish that our remarks on formal logic, scattered throughout this tutorial, should be misconstrued as being in some sense derogatory of the work being done by those who have committed themselves to it. Rather, we have taken a deliberate scientific standpoint, a point of view, and propose that one place mathematics, logic, and method, on a par with, say, the experimental tools—the equipment—of the chemists, physicists and biologists. In other words, the very fact that most formal methodists espouse formal logic as the vehicle for reasoning and proof, is in itself, an incentive to take an alternative approach, thereby placing the School in opposition to the others in a very definite way, for which we make no apology. Where there is

such difference and diversity then one can expect strong growth among all—subject, of course, to the usual predator-prey mathematical model. In this respect, 'standardisation' of a formal method, whilst being in some sense necessary and essential for industrial engagement, especially for the purpose of tool building to assist in the management and control of the complexity of system specification, a complexity that is several orders of magnitude less than that of system code, is nevertheless a formidable obstacle to creativity, ingenuity, and evolutionary growth. One does not 'standardise' mathematics.

The tutorial text is **dead**, once created. It is there both for the benefit of those who participate personally in the tutorial, acting as an *aide mémoire* to the historical events and the dialogue of the tutorial itself, and for those who did not partake in the dialogue—the tutorial—for whom the event can not possibly have any direct influence. It is an historical imperative that the dead letter must be directed primarily at non-participants and it is, therefore, to them that this tutorial text must be primarily addressed.

Before presenting the specific mathematical material as applied to the chosen application, taken from the domains of medical information and network communication, it behoves us to say a few words on testing. After all, the word is included in the title of the tutorial.

1.2 TESTING

It will be assumed that the reader is familiar with an infrastructure and culture such as might have emerged in conformance with the Federal Information Processing Standard on the validation, verification, and testing of (computer) software [12], or any comparable national, international, or industrial set of guidelines for same.

It is surely self-evident from the title of the tutorial, in the placing of formal methods before testing, and in the very juxtaposition of the two terms, that our treatment of testing will differ considerably from current accepted practice or theory. We do not propose to do away with software testing techniques or to render a text such as that of Beizer[3] redundant. If one is searching for further software testing techniques, then this is not the tutorial for them. There are contemporaneous sibling tutorials to address this aspect. What **is** to be found, though taking second place, is a sound theoretical and foundational framework for testing, a sort of skeleton, upon which and with respect to which, testing techniques have a place.

Due to its precise mathematical nature, a formal method permits the term *verification*—the building of the system rightly—to attain the fullness of its meaning. Within the *VDM*, there are planes of abstractions, both vertical and horizontal, expressed in terms of models and their operations. A *downward* movement, by which models on one plane are developed into models on a lower plane, i.e., that which is nearer to the ultimate realisation of the system—the implementation, is usually

known as a *reification* or downward refinement. A *lateral* movement is an *elaboration* on the same plane of abstraction. In such an elaboration we move through lateral planes by adding information, often in an exploratory fashion.

The method (*M*) in the *VDM*, posits a variety of choices for both reification and elaboration, and provides the mathematical techniques, among which the **retrieve** functions feature prominently, by which one may prove rigorously the validity of a development (*D*). In such a context does verification take on its fullest meaning.

Validation—the building of the right system—is to be construed as the process by which it may be shown that the system to be built, and subsequently, the delivered system, satisfies the requirements and, therefore, ultimately the clients who need the system and who are bearing the costs. We have already exhibited, in considerable detail, the use of the *VDM*$^{\clubsuit}$ for verification [10] [11]. In this tutorial, we take the opportunity to make public for the first time an extensive treatment of the method of the *VDM*$^{\clubsuit}$ by which requirements modelling and requirements specification are to be validated, **within their phase** of the life-cycle.

In this context we are wont to refrain from using the term *testing*, since it has very strong connotations with software and system testing. In particular, in all aspects of telecommunications systems, it is almost impossible to speak about requirements and specification testing, so forcefully is the term bound to a 'testing culture' and the accompanying specific set of testing techniques used in practice. In consequence, we coined the phrase 'the determination of critical examination criteria for validity' to replace the term testing when we wish to be precise and unambiguous about the method by which we determine bugs in system requirements and system specification in the pre-development phases of the life-cycle.

To illustrate what we mean by the *determination of critical examination criteria*, in a context other than that of the mathematical methods to be presented, we have found the following real-life example to be particularly illuminating. The case in question concerns the terrible disaster that befell the space shuttle CHALLENGER on 28th January 1986 [6] p. 415, a failure that was subsequently determined to be due to (i) physical failure of the so-called O-rings, and (ii) managerial failure that permitted a launching deadline to overrule the results of the standard testing techniques [6] p. 426.

The test that led to exposure of the failure was as breathtakingly simple in its nature, as it was far reaching in its implications for **all** systems engineering [6] p. 423:

> DR. FEYNMAN: This is a comment for Mr. Mulloy [*project manager for the solid rockets*]. I took this stuff [*i.e., rubber*] that I got out of your seal [*i.e., the Viton O-ring*] and I put it in ice water, and I discovered that when you put some pressure on it for a while [*For the test, Feynman had earlier purchased a small C-clamp and pliers in a hardware store in Washington.*

> *The provision of a carafe of ice-cold water and glass completed the test equipment.*] and then undo it, it doesn't stretch back. It stays the same dimension. In other words, for a few seconds at least and more seconds than that, there is no resilience in this particular material when it is at a temperature of 32 degrees [*Fahrenheit*].
>
> I believe that has some significance for our problem.

One must place this test in context. Despite all of the resources available, and the testing techniques that must have been applied, why is it that the Feynman test, so cheap and so simple, should make (testing) history? The answer is as simple as the test itself. There are some very fundamental (physical) principles that can not be ignored, and consequent fundamental (engineering) principles that must not be violated without, as in this case, disasterous repercussions. Physical principles are well established and well known, with an astonishing variety of experimental data to support them, and a wealth of mathematics that brings coherence to the interpretation of the experimental results. Is that not also true of computing experiments—the programs that we write and the systems that we build? What then are the corresponding computing principles and where is the mathematics to support them?

It will be argued and demonstrated in this tutorial that formal methods supply at least some of the answers.

We shall conclude this prologue by addressing ourselves to systems requirements, and what better way than to give a definition of requirements verifiability from the ANSI/IEEE standard that sets the guidelines for Software Requirements [7] p. 26:

DEFINITION 1.1 *A requirement is verifiable if and only if there exists some finite cost-effective process with which a person or machine can check that the software product meets the requirement.*

Guidelines are one thing, definitions are another, and the reality is a third. In reality, many requirements are strictly *delta* requirements, an expression of the need for an enhancement, i.e., added functionality, to an existing system. Rarely is one given the opportunity of capturing, analysing, modelling, and specifying, the requirements for a system *ab initio*.

Those who have been responsible for such systems requirements, whether as deltas or for new systems, know that certain sentences, felicitous requirements statements, already trigger potentially realisable implementation strategies. This is all the more the case for the formal methodist. A brief statement of description is usually sufficient to trigger the model building activity, and having begun with the names of concrete domains drawn from that description, a flood of purely abstract models rushes in, each model proposing itself for consideration, and leading to a frenzy in reuse. For this indeed is in the very nature of mathematical modelling—many of the necessary

abstractions are already available. Nor is this activity localised to requirements modelling. It is a phenomenon familiar to all (applied) mathematicians, whatever the field, a phenomenon that leads one to speak of mathematical *discovery* rather than mathematical *invention*.

Thus, although we have argued elsewhere [11] against the inlining of formal methods in a deadlined development project, this was not due to some unstated inherent weakness in formal methods *per se*, but rather in the realistic appraisal of the amount of expertise available in the field.

This remark requires us to comment on what may realistically be expected of a formal methods course, or of what the likely impact will be on an organisation, if it should seriously embrace a formal methods culture, and integrate it into its software process.

Based on experience, we estimate that, if introduced into an organisation in 1993, formal methods will bear fruit in a tangible manner, three to four years hence. By that we mean to say that the formal methodists will by then have become integrated (and accepted) within the organisation, and their mathematical techniques will have become an integral part of the software process. A comparable time span is to be found in small software start-up companies from initial team-building activities and product idea brainstorming until first successful product release.

To continue with the company start-up analogy, we assert that there is a critical-mass factor which is absolutely essential for success. That is to say, we proclaim that perhaps as few as five formal methodists are necessary. This is based on both psychological and sociological factors, and the **key** sociological factor for formal methods, as for mathematics in general, is that both are intrinsically social activities. Proof is a social activity. Automation of proof may be desirable in certain cases ... but there follows the corollary that the proof is not interesting/significant mathematically.

But from where are all these formal methodists to come? A two-week course, for instance, is certainly not sufficient to train/educate someone in the exploitation of formal methods and their application to real projects. The two-week course **is** necessary to prepare the organisation, management, development team, etc., to be able to embrace the new technology, to alert the development teams to the potential richness and added-value, and to deflate potential perceived threats to 'the way things are done'!

In preparing for this tutorial, and having in mind the potential for litigation in the medical domain, and in view of the subject matter, the idea of focusing on a system that processed blood tests suggested itself. That such a system might exist, did not deter us from modelling one, and we considered the pun on test to be charmingly worthwhile.

Thus to set the scene, we hurriedly scribbled a few contextual statements and decided upon a requirement, the invalidation of which would prove not merely em-

barrassing to all concerned, but that would lead, almost certainly, to litigation in certain circumstances.

EXAMPLE 1.1 (BLOOD TEST SYSTEM) *Reflections of the author as they were scribbled in November 1992.*

1) A hospital laboratory performs blood tests for the patients of a group of medical practitioners who are affiliated to/registered with the hospital.

2) In the interests of privacy, a general practitioner has access only to the blood tests of her/his own patients.

3) By law (of MMA), the hospital has access to all blood tests to monitor community health.

4) The hospital does not have access to patient identifiers, only to the blood test ids (and results, of course), and to the corresponding general practitioner.

5) ...

REQUIREMENT 1.1 *Blood tests must not be mixed up.*

From this text a development is exhibited. It must be confessed that we thought to rewrite the description to reflect more accurately the mathematics that did eventually emerge. Such a re-polishing, both of description and mathematical formulation, is commonplace. However, we felt that an honest historical account would have a greater impact. In addition, it presents an interesting exercise to determine in what respect the mathematical development has diverged from the original description.

We begin in § **2** with a well-paced presentation of the notion of a model and its operations, and introduce some of the fundamentals of the notation. Embedded within this introduction is a very brief summary of some of the salient aspects of the algebraic theory underlying and supporting the notation. It will be observed that the mathematical theory is at the level of a first year or second year undergraduate course in mathematics at College or University.

Whereas § 2 is a direct outcome of the initial requirements text, with its emphasis on doctors, patients, and blood tests, § **3** introduces the model for a centre for disease control, inspired by those which are established in the U.S.A. and in France.

Already by § 3, the name blood test was being considered conceptually as a place holder for any test or other event that might be related to the notification of a communicable disease, including, for example, such events as the HIV serologic prescription used in France [5]. The focus has, however, now moved to the communciation network, which permits us to demonstrate (cf., Q.E.D. ≡ **quod erat demonstrandum**) what we mean by a proof in the *VDM*$^{\clubsuit}$ and to show how, in practice, theoretical

results are arrived at and used. This material of § **4** is, by far, the most technical of the tutorial.

Having introduced the transport medium for blood test data transmission, and with a view to a major generalisation of the application, we present some further aspects of network modelling in § **5**. It is, of course, completely independent of the medical application.

§ **6** completes the tutorial with a few succint observations and concluding remarks. We had originally intended to present an outline of a multimedia services application which naturally extended the medical information system and the network system. The blood test place holder was conceptually extended to embrace image, audio, and data, which doctors and labs might wish to transmit to the disease centre, or to one another, and, by inference, to a full general multimedia service application via total abstraction. Unfortunately, this work which is still in progress, could not be presented here in the tutorial text for a variety of reasons.

2 MODELS

A good recommendation in writing is to begin at the beginning. We take these words to heart, keeping in mind that there may be those for whom this introduction is exceedingly elementary, and try within the allotted space to be as overtly simple as possible. When we think of blood test, we naturally think of the relation that a doctor bears to her/his patient.

EXAMPLE 2.1 (DOCTOR-PATIENT RELATION) *Each doctor d has a set of patients $p_1, p_2, \ldots, p_n$. Such a doctor-patient relationship, viewed from the perspective of the doctor, will be designated a mapping, thus:*

$$[d \mapsto \{p_1, p_2, \ldots, p_n\}] \tag{1}$$

For some other doctor d', i.e., $d' \neq d$, we will have a similar mapping:

$$[d' \mapsto \{p'_1, p'_2, \ldots, p'_n\}] \tag{2}$$

The space of all such doctor-patient relationships is usually designated by

$$\text{DOCTOR} \xrightarrow{m} \mathcal{P}\,\text{PATIENT} \tag{3}$$

where the symbol $\mathcal{P}$ denotes a power set. A typical collection of doctors and their patients will be denoted by some variable name which is customarily a letter, say the Greek letter μ. For convenience, when specifying the model, we will usually attach such a variable name thus:

$$\mu \in \text{DOCTOR} \xrightarrow{m} \mathcal{P}\,\text{PATIENT} \tag{4}$$

Once we have written down a model, we may immediately write down mathematical expressions and provide possible **interpretations**. For example, given a collection of doctors and their patients μ, and some doctor d which is recorded in μ, then the expression

$$\mu(d) \tag{5}$$

designates the patients of doctor d. Alternatively, we may view the expression as the response to the question

QUESTION 2.1 *Who are the patients of doctor d?*

However, the expression is valid if and only if (which expresses both **necessary** conditions and **sufficient** conditions, and is often abbreviated to **iff**) doctor d is actually recorded in the given collection μ. We specify this validity condition by the expression

$$\chi[\![d]\!]\mu \tag{6}$$

which is our preferred form for the alternative $d \in dom\,\mu$.

In presenting our work to others, or for recording purposes in archival documents, it is customary to provide a heavy sprinkling of syntactic sugar for the simple mathematical expression. Thus, the query on the patients of some doctor may be considered to be a (table) lookup operation, which we might characterise in the form:

1 LookUp: DOCTOR $\longrightarrow$ ((DOCTOR $\xrightarrow{m}$ $\mathcal{P}$ PATIENT) $\longrightarrow$ $\mathcal{P}$ PATIENT)
.1 LookUp$[\![d]\!]\mu \triangleq \mu(d)$

and the validity constraint, i.e., that d is in the domain of μ, is expressed as a pre-condition of the lookup operation:

2 *pre*–LookUp: DOCTOR $\longrightarrow$ ((DOCTOR $\xrightarrow{m}$ $\mathcal{P}$ PATIENT) $\longrightarrow$ $\mathbb{B}$)
.1 *pre*–LookUp$[\![d]\!]\mu \triangleq \chi[\![d]\!]\mu$

But the expression $\chi[\![d]\!]\mu$ itself may also be wrapped up in syntactic sugar as an is-recorded operation:

3 isRecorded: DOCTOR $\longrightarrow$ ((DOCTOR $\xrightarrow{m}$ $\mathcal{P}$ PATIENT) $\longrightarrow$ $\mathbb{B}$)
.1 isRecorded$[\![d]\!]\mu \triangleq \chi[\![d]\!]\mu$

Being a predicate, it does not have a pre-condition. Use of this predicate name allows one to express the pre-condition of the lookup operation in a wonderfully verbose form:

.1 *pre*–LookUp$[\![d]\!]\mu \triangleq$ isRecorded$[\![d]\!]\mu$

With frequency of use it is customary to abbreviate operation names such as LookUp$[\![d]\!]$ by Lkp$[\![d]\!]$, or even the completely mathematical form $L[\![d]\!]$.

Syntactic sugaring may also be applied to parameter names, of course. Thus instead of the purely symbolic d, one might use *doc* or even *doctor*. However, such syntactic sugaring, whilst being 'user-friendly', is rarely if ever used in mathematics, tending, as it does, to obstruct the clarity of the expressions. Indeed, the very use of model names such as DOCTOR and PATIENT obscures the remarkable amount of reuse available via the abstract $X \xrightarrow{m} \mathcal{P}\, Y$ form.

Let us now explore further our model of doctors and patients. It will give us an opportunity to present other fundamental mathematical operations and expressions. First, a typical collection of doctors and their patients may be represented by

$$\mu = [d^1 \mapsto \{p_1^1, p_2^1, \ldots, p_j^1\}, d^2 \mapsto \{p_1^2, p_2^2, \ldots, p_k^2\}, \ldots] \tag{7}$$

Note our decision to use both superscript and subscript notation for indexing. It is included to indicate some of the experimental work currently being investigated in the *VDM*$^{\clubsuit}$ to try to adopt/adapt tensor notation for our own purposes.

For convenience we may exhibit the same information in 'matrix' form:

$$\mu = \left[\begin{array}{lcl} d^1 & \mapsto & \{p_1^1, p_2^1, \ldots, p_j^1\} \\ d^2 & \mapsto & \{p_1^2, p_2^2, \ldots, p_k^2\} \\ \ldots & & \ldots \\ d^m & \mapsto & \{p_1^m, p_2^m, \ldots, p_l^m\} \end{array}\right] \tag{8}$$

The quotes around matrix are necessary to emphasise the lack of ordering by column in such a map. Ordering is an essential concept in the normal matrix. However, now that we have made the comment, we feel free to use the word matrix unquoted to describe such a display.

If we wish to inquire as to the set of all the doctors recorded in the collection, then the answer is immediately given by

$$dom\, \mu = \{d^1, d^2, \ldots, d^m\}, \tag{9}$$

which is the column of the matrix. If, on the other hand, we inquire as to the set of all patients in the matrix, then the operator *rng* gives the second column:

$$rng\, \mu = \{\{p_1^1, p_2^1, \ldots, p_j^1\}, \{p_1^2, p_2^2, \ldots, p_k^2\}, \ldots, \{p_1^m, p_2^m, \ldots, p_l^m\}\} \tag{10}$$

But technically the question calls for the result

$$\{p_1^1, p_2^1, \ldots, p_j^1, p_1^2, p_2^2, \ldots, p_k^2, \ldots, p_1^m, p_2^m, \ldots, p_l^m\} \tag{11}$$

which may be obtained by flattening, i.e., applying a reduction with respect to set union, of the range result:

$$\begin{aligned} \cup/\, rng\, \mu &= \cup/\{\{p_1^1, p_2^1, \ldots, p_j^1\}, \{p_1^2, p_2^2, \ldots, p_k^2\}, \ldots, \{p_1^m, p_2^m, \ldots, p_l^m\}\} && (12) \\ &= \{p_1^1, p_2^1, \ldots, p_j^1\} \cup \{p_1^2, p_2^2, \ldots, p_k^2\} \cup \ldots \cup \{p_1^m, p_2^m, \ldots, p_l^m\} && (13) \\ &= \{p_1^1, p_2^1, \ldots, p_j^1, p_1^2, p_2^2, \ldots, p_k^2, \ldots, p_1^m, p_2^m, \ldots, p_l^m\} && (14) \end{aligned}$$

Let us contine to examine our model by proposing other obvious questions.

QUESTION 2.2 *What happens if we wish to register a new doctor d?*

In other words, we have an existing collection μ and some new doctor d is to be registered, i.e., we have the pre-condition $\neg\chi[\![d]\!]\mu$. In this case we extend the collection using a special extend operator, $\sqcup$, which is expressly provided for this purpose:

$$\mu \sqcup [d \mapsto \emptyset] \tag{15}$$

In particular, $\mu \sqcup [d \mapsto \emptyset]$ is valid, i.e., mathematically defined, iff d does not occur in the domain of the map, $\neg\chi[\![d]\!]\mu$.

Historical Note: The map extend operator, $\sqcup$, was (and is) used for the specification of the declaration of new variables within a *block* in a programming language.

Note that patients associated with the doctor are not registered at this time. For this purpose, we have another special override operator, $\dagger$. Specifically, let us suppose that the patient p of doctor d is to be recorded. We may specify this by

$$\mu \dagger [d \mapsto \mu(d) \cup \{p\}] \tag{16}$$

One will note that the occurrence of $\mu(d)$ in the above expression implies the necessary precondition $\chi[\![d]\!]\mu$.

Historical Note: The override operator, $\dagger$, was (and is) used to specify the semantics of the *assignment statement* in a programming language.

This override operator may be rigorously defined in terms of a deletion operator, $\lhd\, [\![d]\!]$, which removes all information associated with d from the map, and the extend operator:

$$\mu \dagger [d \mapsto \mu(d) \cup \{p\}] \triangleq \lhd\, [\![d]\!]\mu \sqcup [d \mapsto \mu(d) \cup \{p\}] \tag{17}$$

Such a rewriting turns out to be invaluable in carrying out certain types of proofs. Lest we forget, $\lhd\, [\![d]\!]$, is a convenient form for the more precise $\lhd\, [\![\{d\}]\!]$. In other words, we tend to simplify notation wherever possible.

Historical Note: Removal is necessary in specifying the semantics of local variables. Once control passes out of a block, any temporary variables created and the associated storage, must be removed. This is the case, for example, in *procedures*. The strange

form of the removal operator is due to $\mathcal{Z}$. That it is customarily expressed as a functional operator in the $VDM^{\clubsuit}$ ought to become clear in the context of more elaborate specifications, to be presentated later.

One will wonder whether a doctor and her/his patient could be registered at the same time. Well, of course, if that is the requirement, then the expression needed is

$$\mu \sqcup [d \mapsto \{p\}] \tag{18}$$

But we may also write this expression in the form

$$\mu \dagger [d \mapsto \{p\}], \tag{19}$$

a fact which may be checked by applying the definition given above for the override operator. Nevertheless, in system specification, clarity is essential and one ought to say what one means. Allowing the override operator to be a sort of 'catchall' is dangerous.

To the initiate, all of the mathematical material presented thus far, will have seemed to be bewildering. Of overriding concern will be the basic question: Why choose these particular operators? Before proceeding with a further analysis of our simple model, let us turn to the mathematical algebraic theory that provides the setting for all of the work.

2.1 Monoids

Everyone is familiar with the natural numbers

$$\mathbb{N} = \{0, 1, 2, 3, 4, \ldots\} \tag{20}$$

and that if one adds any two natural numbers m and n, then the result is a natural number. This basic fact is called the **law of closure:** for all m, n in $\mathbb{N}$, then

$$m + n \in \mathbb{N} \tag{21}$$

The zero, 0, is a *unique* natural number (with respect to addition) in the sense that it is the only number which when added to any other natural number n, gives the same natural number n. We say that zero is the (unique) **identity element** for addition:

$$n + 0 = n = 0 + n \tag{22}$$

We may, for convenience, speak of this fact as the **law of the identity**. Finally, we are familiar with the simple fact that the order in which we do additions of more than two natural numbers does not matter to the overall result. More formally, we say that for all m, n, and p in $\mathbb{N}$,

$$(m + n) + p = m + (n + p) \tag{23}$$

This is the **associative law**. Any set M, furnished with a binary operator, $\oplus$, and which satisfies the three laws given above is called a monoid, denoted $(M, \oplus, u)$, where $u \in M$ is the (unique) identity element for the operator $\oplus$. Thus the set of natural numbers under addition, $(\mathbb{N}, +, 0)$, is a monoid.

If, in addition to the three laws cited, we have the **commutative law**, i.e., for all m, n in M,

$$m \oplus n = n \oplus m \tag{24}$$

then we have a commutative monoid. The set of natural numbers under addition, $(\mathbb{N}, +, 0)$, is a commutative monoid. It just so happens that the monoid is fundamental to computer science. It turns up everywhere. Before we return to consider the doctor-patient model, it is important to note that the 'same' set may give rise to different monoids with respect to different operators. For example, if we delete the zero from the natural numbers, $\lhd [\![0]\!]\mathbb{N}$, we obtain the set of counting numbers

$$\{1, 2, 3, 4, 5, \ldots\} \tag{25}$$

which we may denote, for convenience, by $\mathbb{N}'$, or $\mathbb{N}_1$. The set of counting numbers forms a commutative monoid, $(\mathbb{N}', \times, 1)$, under multiplication.

Now let us consider our doctor-patient model in the abstract:

$$X \xrightarrow{m} Y \tag{26}$$

where X denotes DOCTOR and Y denotes $\mathcal{P}$ PATIENT. Then all the elements μ in $X \xrightarrow{m} Y$ forms a non-commutative monoid, $(X \xrightarrow{m} Y, \dagger, \theta)$, under the override operator, where θ denotes the null map. Just as in the case of natural numbers, we may provide another operator for the underlying set of maps that gives us a commutative monoid. Specifically, let us consider

$$X \xrightarrow{m} \mathcal{P}' Y \tag{27}$$

where X denotes DOCTOR and Y denotes PATIENT. Remove the null set from the range, i.e., $\mathcal{P}' Y = \lhd [\![\emptyset]\!] \mathcal{P} Y$. Then let us define a special $\textcircled{\cup}$ operator which combines the extend operator and the override operator as follows:

$$\mu \textcircled{\cup} [d \mapsto \{p\}] \triangleq \begin{cases} \mu \sqcup [d \mapsto \{p\}], & \text{if } \neg\chi[\![d]\!]\mu \\ \mu \dagger [d \mapsto \mu(d) \cup \{p\}], & \text{otherwise} \end{cases} \tag{28}$$

Then $(X \xrightarrow{m} \mathcal{P}' Y, \textcircled{\cup}, \theta)$ is a commutative monoid where the operator $\textcircled{\cup}$ **inherits** its commutativity from that of set union $\cup$. Thus the registration of a doctor and her/his patient might be succintly captured by

$$\mu \textcircled{\cup} [d \mapsto \{p\}] \tag{29}$$

This abstract monoid will prove to be of especially great significance in some of our later specifications.

2.2 MORPHISMS

Every engineer is surely familiar with the remarkable property of logarithms, embodied in the physical realisation of the slide rule, that the log of a product is the sum of the logs, and that the log of 1 is 0.

$$\log(xy) = \log(x) + \log(y) \tag{30}$$
$$\log(1) = 0 \tag{31}$$

Formally, we say that the log function is an **isomorphism** from the multiplicative group of strictly positive real numbers, $(\mathbb{R}^+\backslash\{0\}, \times, 1)$, to the additive group of real numbers, $(\mathbb{R}^+, +, 0)$. The morphism is called 'iso-' precisely because the two groups have exactly the same structure but with different operators. A group, $(G, \oplus, u)$, is a monoid with the additional law that every element $g \in G$ has a unique inverse $\bar{g} \in G$ such that

$$g \oplus \bar{g} = u = \bar{g} \oplus g \tag{32}$$

Let μ_j, μ_k in DOCTOR $\xrightarrow{m}$ $\mathcal{P}$ PATIENT be two collections of doctors and their patients such that μ_k is to be interpreted as a 'batch update' of μ_j, denoted $\mu_j \dagger \mu_k$. Then the operator *dom* which returns the set of doctors in the collection is a homomorphism from the monoid of maps, (DOCTOR $\xrightarrow{m}$ $\mathcal{P}$ PATIENT, $\dagger, \theta$), to the monoid of sets, ($\mathcal{P}$ DOCTOR, $\cup, \emptyset$).

$$dom(\mu_j \dagger \mu_k) = dom(\mu_j) \cup dom(\mu_k) \tag{33}$$
$$dom(\theta) = \emptyset \tag{34}$$

The domain operator is **not** an isomorphism. We may demonstrate this fact by showing that two distinct maps *glue* to the same set. A simple counter-example is

$$dom([d \mapsto S] \dagger [d \mapsto T]) = dom([d \mapsto S]) \cup dom([d \mapsto T]) \tag{35}$$
$$= \{d\} \cup \{d\} \tag{36}$$
$$= \{d\} \tag{37}$$

Both $dom([d \mapsto S])$ and $dom([d \mapsto T])$ map to the same result, $\{d\}$.

Although map extention, $\sqcup$, does not give a monoid, it is a specialisation of map override, $\dagger$, and, therefore, we also have

$$dom(\mu_j \sqcup \mu_k) = dom(\mu_j) \cup dom(\mu_k) \tag{38}$$

for **disjoint** maps, μ_j and μ_k, i.e., where $dom\,\mu_j \cap \mu_k = \emptyset$.

Unfortunately, the range operator on maps, *rng*, is not a monoid homomorphism:

$$rng(\mu_j \dagger \mu_k) \subseteq rng(\mu_j) \cup rng(\mu_k) \tag{39}$$
$$rng(\theta) = \emptyset \tag{40}$$

We may demonstrate inequality with the same counter-example we used above to illustrate that the domain operator was not an isomorphism.

$$\begin{aligned} rng([d \mapsto S] \dagger [d \mapsto T]) &= rng([d \mapsto T]) && (41)\\ &= \{T\} && (42)\\ &\neq \{S, T\} && (43)\\ &= \{S\} \cup \{T\} && (44)\\ &= rng([d \mapsto S]) \cup rng([d \mapsto T]) && (45) \end{aligned}$$

However, we do have the important result for disjoint maps:

$$rng(\mu_j \sqcup \mu_k) = rng(\mu_j) \cup rng(\mu_k) \tag{46}$$

Therefore, from the definition of the override operator in terms of the extend operator, we do have the equality

$$\begin{aligned} rng(\mu_j \dagger \mu_j) &= rng(\lhd [\![dom\, \mu_k]\!] \mu_j \sqcup \mu_k) && (47)\\ &= rng(\lhd [\![dom\, \mu_k]\!] \mu_j) \cup rng(\mu_k) && (48) \end{aligned}$$

This result is essential for the execution/construction of proofs. Before rejoining our model development there is yet another important theoretical aspect that needs to be covered, one that addresses the meaning of the removal or deletion operator.

2.3 Monoids with Operators

In considering the natural numbers, we identified two well-known monoids: the additive monoid, $(\mathbb{N}, +, 0)$, and the multiplicative monoid, $(\mathbb{N}', \times, 1)$. We wish now to focus on the monoid $(\mathbb{N}, +, 0)$ and ask in what manner the operation of multiplication may be 'integrated' into it. It comes as no surprise to learn that we are here speaking about **the distributive law**, the distribution of multiplication over addition. Specifically, for all m, n in $\mathbb{N}$ and for some k in $\mathbb{N}'$, we usually write the law in the form

$$k(m + n) = km + kn \tag{49}$$

Let us now expressly exhibit the multiplication operator $\times$ which was denoted by juxtaposition above.

$$k \times (m + n) = k \times m + k \times n \tag{50}$$

Finally, being good computer scientists, we will replace the infix $\times$ with a prefix functional name, say M, and wrap up the multiplier k in brackets:

$$\begin{aligned} M[\![k]\!](m + n) &= M[\![k]\!](m) + M[\![k]\!](n) && (51)\\ M[\![k]\!](0) &= 0 && (52) \end{aligned}$$

Well, what a pleasant surprise! The expression, $M[\![k]\!]$, which we will call an operator, behaves exactly like a monoid morphism. Specifically, it maps the monoid $(\mathbb{N}, +, 0)$ to itself. For this reason, we call it an endomorphism of $(\mathbb{N}, +, 0)$. Now we might ask what sort of structure operators of the form $M[\![k]\!]$ might have. Specifically, is there a binary operator that turns the set of all such operators into a monoid? The answer is affirmative. Denoting the binary operator by $\circ$, which usually denotes functional composition, we immediately have

$$\begin{aligned}
(M[\![j]\!] \circ M[\![k]\!])n &= M[\![j]\!](M[\![k]\!]n) && (53)\\
&= M[\![j]\!](k \times n) && (54)\\
&= j \times (k \times n) && (55)\\
&= (j \times k) \times n && (56)\\
&= M[\![j \times k]\!]n && (57)
\end{aligned}$$

Consequently, we may define the law of composition of the functional operators $M[\![j]\!]$ and $M[\![k]\!]$, for all j, k in $\mathbb{N}'$ as

$$M[\![j]\!] \circ M[\![k]\!] = M[\![j \times k]\!] \tag{58}$$

Therefore, the set of all functional operators is closed under this functional composition. It is rather easy to verify that the associative law holds. Finally, $M[\![1]\!]$ is the unique identity. We have just established that the set of functional operators is a monoid under functional composition as defined above. Moreover, the association

$$M[\![k]\!] \leftrightarrow k \tag{59}$$

demonstrates unequivocally that this monoid is isomorphic to $(\mathbb{N}', \times, 1)$.

With this background information, we are ready to show that map removal with respect to a set is a monoid endomorphism. Recall that

$$\lhd [\![d]\!]\mu \tag{60}$$

denotes the collection, $\mu \in \text{DOCTOR} \xrightarrow{m} \mathcal{P}\,\text{PATIENT}$, of doctors and their patients with information on doctor d removed/deleted. Let $S \in \mathcal{P}\,\text{DOCTOR}$ denote a set of doctors. Then we may show that

$$\begin{aligned}
\lhd [\![S]\!](\mu_j \dagger \mu_k) &= (\lhd [\![S]\!]\mu_j) \dagger (\lhd [\![S]\!]\mu_k) && (61)\\
\lhd [\![S]\!]\theta &= \theta && (62)
\end{aligned}$$

and, of course, that this endomorphism applies to the specialised extend of two disjoint maps

$$\lhd [\![S]\!](\mu_j \sqcup \mu_k) = (\lhd [\![S]\!]\mu_j) \sqcup (\lhd [\![S]\!]\mu_k) \tag{63}$$

The collection of such removal operators, $\lhd\!\!\lhd [\![S]\!]$, $S \in \mathcal{P}\,\text{DOCTOR}$, forms a monoid under functional composition defined by

$$\lhd\!\!\lhd [\![S]\!] \circ \lhd\!\!\lhd [\![T]\!] = \lhd\!\!\lhd [\![S \cup T]\!] \tag{64}$$

The functional operator, $\lhd\!\!\lhd [\![\emptyset]\!]$, is the unique identity element and the associate law holds for the composition. Setting up the formal association

$$\lhd\!\!\lhd [\![S]\!] \leftrightarrow S \tag{65}$$

demonstrates that the monoid of removal operators is isomorphic to the monoid of sets of doctors $(\mathcal{P}\,\text{DOCTOR}, \cup, \emptyset)$.

This concludes our very brief exposition of some of the algebraic theory underlying the formal method that we are using in this tutorial. Other theoretical results necessary in the context of the application being developed will be presented in their proper place.

Space prohibits inclusion of other significant foundational material here. Further details may be found in the author's doctoral thesis [10] and in a comprehensive tutorial presented in the Netherlands [11].

2.4 Complementary Models

Let us look at the relationship between our model and reality. We ask ourselves what interpretation is to be given to the model, in what way can it be said to be an abstraction of some reality. As computer scientists we will also naturally ask ourselves how we will implement the model. Does it involve hardware or software? Where will we draw the system boundary that delineates the implementable thing from the purely conceptual thing. The model

$$\mu \in \text{DOCTOR} \xrightarrow{m} \mathcal{P}\,\text{PATIENT} \tag{66}$$

says something about a collection of doctors and their patients. Now it is quite clear that we can not put these doctors and patients into a computer. What goes into a computer is basically encoded information. What information might we associate with our doctors and patients?

Example 2.2 (Doctors and Names) *Every doctor d has a name. Thus, we introduce the space of doctors and their names.*

$$\nu \in \text{DOCTOR} \xrightarrow{m} \text{DOC_NM} \tag{67}$$

A typical doctor-name map will have the form

$$[d_1 \mapsto n_1, d_2 \mapsto n_2, \ldots, d_j \mapsto n_k] \tag{68}$$

In the real world, we know that there is some non-zero probability that two different doctors will have the same name. For instance, it is possible that in some ν we might find entries of the form

$$d_j \mapsto \text{Beizer}, \quad \text{and} \quad d_k \mapsto \text{Beizer} \tag{69}$$

For this reason, names are considered to consist of surname, firstname and middle initials. But even here, chance may have it, that two distinct doctors will have the same (extended) names. Computer scientists overcome this particular problem by insisting on giving people **identifiers**. Therefore, let us introduce this new model

$$\alpha \in \text{DOCTOR} \xrightarrow{m} \text{DOC_ID} \tag{70}$$

Technically, this association is an exact analogue of the notion of coordinates in geometry. Formally, we make a distinction between points and their coordinates. If we let E^3 denote the 3-dimensional euclidean space of points, then we may choose to assign an orthonormal cartesian coordinate frame and associate with each point a triple of real numbers taken from $\mathbb{R}^3$. Such an assignment of cartesian coordinates may be modelled by the map

$$\varrho \in E^3 \xrightarrow{m} \mathbb{R}^3 \tag{71}$$

A particular point p in ϱ will have coordinates (x_p^1, x_p^2, x_p^3). Moreover, distinct points will have distinct coordinates, i.e., the map ϱ will be 1–1. We may express this fact formally by stating that the number of domain elements, points, is exactly the same as the number of range elements, cartesian coordinates:

$$|\, dom\, \varrho| = |\, rng\, \varrho| \tag{72}$$

On the other hand, to the same point we may assign, say, spherical coordinates, triples of numbers, (r_p, ϕ_p, θ_p), where r_p, ϕ_p, θ_p in $\mathbb{R}$, and $0 \leq \phi_p < 2\pi$, $0 \leq \theta_p < 2\pi$. Such an assignment may also be modelled by

$$\sigma \in E^3 \xrightarrow{m} \mathbb{R}^3 \tag{73}$$

Unlike the cartesian coordinate map, the spherical coordinate map is not 1–1. The problem arises at the origin point of the coordinate space. Specifically, when $r_p = 0$, we can not distinguish between $(r_p, 0, 0)$ and $(r_p, \pi/2, \pi)$, say. Is it not surprising, therefore, that coordinate functions should be complex in computer science?

We can certainly put identifiers into a computer. But what do we do about the doctor part of the association? Doctors will not go into computers. Consider our two models

$$\nu \in \text{DOCTOR} \xrightarrow{m} \text{DOC_NM} \tag{74}$$

$$\alpha \in \text{DOCTOR} \xrightarrow{m} \text{DOC_ID} \tag{75}$$

Since the latter is 1–1, we may **invert** it to obtain

$$\alpha^{-1} \in (\text{DOCTOR} \xrightarrow{m} \text{DOC_ID})^{-1} = \text{DOC_ID} \xrightarrow{m} \text{DOCTOR} \tag{76}$$

Composition of the inverted model and the doctor-name model gives us the basic information model

$$\begin{aligned} \nu \circ \alpha^{-1} = \delta \quad &\in \quad \text{DOC_ID} \xrightarrow{m} \text{DOC_NM} && (77) \\ &= \quad (\text{DOCTOR} \xrightarrow{m} \text{DOC_NM}) \circ (\text{DOC_ID} \xrightarrow{m} \text{DOCTOR}) && (78) \end{aligned}$$

where the composition $\nu \circ \alpha^{-1}$ is defined over the common intersection of the domains of ν and α^{-1}. Now we have arrived at a model that may be used in an implementation. The unique identifier represents the doctor and the name represents an attribute of the doctor, her/his name.

TECHNICAL ASIDE: The notion of inverse images is a key one in mathematics, and consequently, in the *VDM*♣. It plays a critical rôle in classical mathematics. It is not surprising, therefore, that in a model-theoretic formal method, such as the *VDM*♣, it should also prove to be of critical importance. For example, it was by the use of the inverse image that the annihilator form of the invariant for the bill of materials problem was discovered [11] p. 201, a major result that led to the tractability of proving that operations on a bill of material satisfied the invariant. A second major result was the discovery that the inverse image is just the appropriate construct to identify *aliases* in a system specification [11] p. 182. Again, in determining the *standard development* steps of the *VDM*♣, the inverse image proved essential in the parameterisation step [11] p. 189. That the inverse image should prove to be of considerable use in the *VDM*♣ is sufficient reason for us to study its properties at length.

Since the inverse image will re-occur frequently in this tutorial, it seem appropriate to mention briefly some of its uses in classical mathematics. We begin with a definition from classical algebra.

Every map μ from a set X to a set Y,

$$\mu \in X \xrightarrow{m} Y \tag{79}$$

determines an equivalence relation E_μ in X, [8] p. 12, by specifying

$$x_j E_\mu x_k \quad \text{if and only if} \quad \mu(x_j) = \mu(x_k) \tag{80}$$

The inverse image $\mu^{-1}(y)$ of the element $y \in Y$, is the set of all elements $x \in X$ which map under μ to y, formally

$$\mu^{-1}(y) = \{x \in X \mid \mu(x) = y\} \tag{81}$$

If $y = \mu(x)$ then the set $\mu^{-1}(y)$ is just the equivalence class $\bar{x}_{E_\mu} = \mu^{-1}(\mu(x))$ in X determined by the element x, which subset of X is sometimes referred to as the "fibre over the element" y [8] p.12.

The set of all such fibres constitutes a partition of X determined by E_μ, [8] p. 12, i.e., they are elements of the quotient set X/E_μ, which in turn is a subset of $\mathcal{P}(X)$.

If we let ν denote the natural map from X to X/E_μ, then there is a map $\bar{\mu}$ from X/E_μ to Y which is induced by μ, such that we have the factorisation

$$\mu = \bar{\mu} \circ \nu \tag{82}$$

It is, of course, quite natural to generalise the inverse image to apply to a subset S of Y rather than a simple element. Specifically,

$$\mu^{-1}(S) = \{x \in X \mid \mu(x) \in S\} \tag{83}$$

In analysis, the concept of inverse image plays a fundamental definitional rôle. For example, a function f from a measurable space X to a topological space Y,

$$f \in X \xrightarrow{m} Y \tag{84}$$

is defined to be measurable, [13] p. 8, provided that $f^{-1}(S)$ is a measurable set for every open set S in Y.

Such examples of the use of the inverse image in mathematics are intended solely to indicate its relevance therein. Due to the underlying philosophy of the *VDM*$^\clubsuit$, [10], it behoves us to explore the range of its use in the specification of systems and any properties that inverse images might have □

For those *au fait* with (relational) databases, it will be recognised that the uniqueness of the identifier is spelt out more precisely by adding extra information such as social security number, address, phone number, etc.

$$\delta' \in \text{DOC_ID} \xrightarrow{m} (\text{DOC_NM} \times \text{SOC_SEC_NUM} \times \text{ADDRESS} \times \text{PHONE_NUM} \times \ldots) \tag{85}$$

We may, if we so wish, extract the original indentifier-name structure from this model by iterating over δ' with a pair of functions, the identity function, $\mathcal{I}$, and a projection function, π_1, that selects the first component of the attribute tuple. Note that the projection function is not to be confused with the number that denotes the area of a unit disc, π! We denote this iteration by

$$(\mathcal{I} \xrightarrow{m} \pi_1)\delta' = \delta \tag{86}$$

All of these models may be manipulated by map operators that were introduced in the context of the doctor-patient collection: domain and range operators, map extend and map override, and removal with respect to a set. For each operator expression

we can find a meaningful interpretation pertinent to the problem at hand and **most importantly** conditions of validity—pre-conditions—guarding each operator. *Is it really necessary to state that the pre-conditions are precisely those things which need to be tested in an implementation?* There are other aspects of models which give rise to the determination of critical examination criteria for validity.

Before delving deeper into the application at hand, we would now like to introduce another model which is complementary to the basic doctor-patient model.

EXAMPLE 2.3 (PATIENT-DOCTOR RELATION) *From the perspective of the patient p, we may wish to record all the doctors that the patient consults, the eye doctor, the ear doctor, the cardiologist, etc. Such a patient-doctor relation may be denoted by*

$$[p \mapsto \{d_1, d_2, \ldots, d_m\}] \tag{87}$$

In other words we wish to look at the space

$$\mu^{-1} \in \text{PATIENT} \xrightarrow{m} \mathcal{P}\,\text{DOCTOR} \tag{88}$$

This model is the inverse of the doctor-patient relation and, consequently, we use μ^{-1} to denote a typical instance. It is not the sort of model that one would normally wish to implement directly (via the coordination technique presented) because the map is conceptually accessible only to a **homunculus**, an external (global) observer. Such a situation occurs frequently in formal modelling and is of great usefulness in system specification. Some questions that we might ask of μ may be answered directly by reference to this model. Specifically, if we ask who are the doctors that some patient p consults, then the result is immediate

$$\mu^{-1}(p) \tag{89}$$

where we have, of course, the necessary pre-condition $\chi[\![p]\!]\mu^{-1}$. However, many software engineers may feel uncomfortable about introducing such a complementary model, that somehow we are cheating. Therefore, let us provide an explicit inversion algorithm that converts a given μ into the corresponding μ^{-1}. This will give us an opportunity to introduce the style of algorithm that we normally use in the *VDM*$^{\clubsuit}$, which in this case is tail-recursive. Those familiar with programming in a logic programming language such as PROLOG or, especially, a functional programming language such as MIRANDA, will have no difficulty in following the reasoning behind the algorithm. Space prohibits us from giving a comprehensive treatment of the theory of tail-recursive functions, or of the more general theory of curried functions. Supplementary material may be found in [10] and [11]. Suffice it to remark that tail-recursive functions have exact while-loop equivalents in imperative programming languages and the strategies for the transformation of a tail-recursive function into the corresponding while-loop are so well-developed that complete automation is often possible.

The inversion algorithm follows:

4 Invert: (DOCTOR $\xrightarrow{m}$ $\mathcal{P}'$ PATIENT) $\longrightarrow$ (PATIENT $\xrightarrow{m}$ $\mathcal{P}'$ DOCTOR)

.1 $\text{Invert}(\mu) \triangleq \text{Invert}[\![\mu]\!]\theta$

5 Invert: (DOCTOR $\xrightarrow{m}$ $\mathcal{P}'$ PATIENT) $\longrightarrow$
((PATIENT $\xrightarrow{m}$ $\mathcal{P}'$ DOCTOR) $\longrightarrow$ (PATIENT $\xrightarrow{m}$ $\mathcal{P}'$ DOCTOR))

.1 $\text{Invert}[\![\theta]\!]\nu \triangleq \nu$

.2 $\text{Invert}[\![[d \mapsto S] \sqcup \mu]\!]\nu \triangleq \text{Invert}[\![\mu]\!] \circ \text{Invert}[\![d, S]\!]\nu$

6 Invert: DOCTOR $\times$ $\mathcal{P}'$ PATIENT $\longrightarrow$
((PATIENT $\xrightarrow{m}$ $\mathcal{P}'$ DOCTOR) $\longrightarrow$ (PATIENT $\xrightarrow{m}$ $\mathcal{P}'$ DOCTOR))

.1 $\text{Invert}[\![d, \emptyset]\!]\nu \triangleq \nu$

.2 $\text{Invert}[\![d, \{p\} \uplus S]\!]\nu \triangleq \text{Invert}[\![d, S]\!](\nu \textcircled{\cup} [p \mapsto \{d\}])$

Annotations

4 Note that the null set is excluded from the maps. This is a critical point. It is not possible to invert an entry such as $d \mapsto \emptyset$. A similar remark applies to the signatures in **5** and **6**.

4.1 The computation of the function to be called is handed over to another function of the same name. This latter function is the tail-recursive function whose signature (**5**) differs from the function to be called. The argument outside the parenthesis (which appears as ν elsewhere) is often known as an accummulator variable and is initialised to the null map. Entry to the tail-recursive function is equivalent to entry to a while-loop.

Note the overloading of the function name. The two distinct functions may be determined by signature, i.e., by the number of parameters and their type. This facility is availed of by PROLOG and ADA, for example.

This inversion function is a monoid homomorphism:

$$\text{Invert}(\mu_j \textcircled{\cup} \mu_k) = \text{Invert}(\mu_j) \textcircled{\cup} \text{Invert}(\mu_k) \quad (90)$$

$$\text{Invert}(\theta) = \theta \quad (91)$$

5.1 The tail-recursive function is a functional operator. In this, the base case, we have termination (exit from the while-loop) and $\text{Invert}[\![\theta]\!]$ is clearly the identity function.

5.2 We take advantage of the definition of the extend operator on maps to exhibit a non-null map explicitly as the extend of two disjoint maps. In this, the recursive case, we introduce a second tail-recursive function that will iterate over the set of patients.

The second invert function gives us a monoid of functional operators:

$$\text{Invert}[\![\mu_j \textcircled{\cup} \mu_k]\!]\nu = \text{Invert}[\![\mu_k]\!] \circ \text{Invert}[\![\mu_j]\!]\nu \quad (92)$$

$$\text{Invert}[\![\theta]\!]\nu = \nu \quad (93)$$

6.1 Clearly, $\text{Invert}[\![d, \emptyset]\!]$ is the identity function. However, its arguments are formally equivalent to an entry of the form $d \mapsto \emptyset$ which we have ruled out. There is, of course, no inconsistency. We have taken care of that problem by arranging the signature (**6**) appropriately.

6.2 Since ordinary set union, $\cup$, does not allow us to express the disjoint union of two sets and thus to exhibit a non-empty set in two disjoint parts, we have recourse to a special union operator, $\uplus$, which functions as such a disjoint union. This recursive case is where some actual modification of the accumulator variable take place. Note that the result is elegantly expressed in terms of the operator that gives us a monoid of patient-doctor relations.

Yet again do we have a monoid of functional operators. For convenience, we will use a slightly modified notational form:

$$\text{Invert}_d[\![S \cup T]\!]\nu = \text{Invert}_d[\![T]\!] \circ \text{Invert}_d[\![S]\!]\nu \quad (94)$$

$$\text{Invert}_d[\![\emptyset]\!]\nu = \nu \quad (95)$$

The algorithm is proven correct by construction! That is to say, in the *VDM*$^{\clubsuit}$, proofs are constructive. This philosophical position is poorly understood by the practitioners of other formal methods. In a sense it is the ideal that every programmer strives for. However, just as in programming, it is quite possible that the strong claim made about correctness, is in fact false. How is one to know? Therein lies the social aspect of mathematical proof. It is certainly not primarily an issue of **testing** the specification. Rather we emphasise that the specification is presented before others and the arguments for correctness are presented.

However, every expression in the *VDM*$^{\clubsuit}$ is constructive and, consequently, executable in practice. Hence, a prototype of the specifications written in, say MIRANDA, is readily feasible. System specifications grow in complexity but at a much lower rate than system code. There are other formal methods with tool support which will happily accommodate the *VDM*$^{\clubsuit}$. In this manner at least the syntactic complexity may be kept under control.

This completes our introductory treatment of models and their operators in the *VDM*$^{\clubsuit}$.

3 THE LAB AND THE DISEASE CENTRE

There is a well-know aphorism that "*familiarity breeds contempt*". Such aphoristic utterances convey kernels of wisdom that need to be elaborated upon to ascertain their precise meaning. We all know the counter-aphorism that "*absence makes the heart grow fonder*", a saying the meaning of which is usually immediately clear to those who return to the home hearth after prolonged visits to conferences and symposia. But what of the aphorism originally quoted? By what mechanism does familiarity breed contempt?

One reasonable theory that explains the saying has to do with *enlightenment*. In other words, the more familiar you are with the other, the more you get to know the other, warts and all. It is the very same with formal methods. Therefore, to reach the ultimate stage of perfect contempt, we first must become familiar and strive for enlightenment. In this tutorial the path to enlightenment is set out clearly within the framework of doctors, patients, hospital laboratories, blood tests and a centre for disease control. To this fascinating world we now return.

However, having elaborated in depth in § 2 on the style of development by which we arrive at the mathematical expressions which are the answers to our questions, we feel free hereafter to be more brief in our exposition in order to focus more on the application at hand. We trust that one has already sufficient information and insight that she/he will be able to expand on a given expression, where necessary. Again, for further assistance, one may consult the material in [10] and [11].

A hospital laboratory carries out blood tests for the patients of doctors who are registered with the hospital. The recorded results of the blood tests may be modelled by the space

$$\varrho \in \text{PATIENT} \xrightarrow{m} \text{B_TEST} \tag{96}$$

The hospital register which lists affiliated doctors and their patients is modelled by

$$\mu \in \text{DOCTOR} \xrightarrow{m} \mathcal{P}\,\text{PATIENT} \tag{97}$$

This model we have already seen before. Now it may be the case that we wish to ensure that the patients mentioned in ϱ are precisely the same patients listed in the register μ. To establish such a constraint we use an **invariant**.

7 $inv\colon (\text{PATIENT} \xrightarrow{m} \text{B_TEST}) \times (\text{DOCTOR} \xrightarrow{m} \mathcal{P}\,\text{PATIENT}) \longrightarrow \mathbb{B}$

.1 $inv(\mu, \varrho) \triangleq dom\,\mu = \bigcup / rng\,\varrho$

Instead of modelling the recorded information in this manner, we might prefer an alternative model:

$$\delta \in \text{DOCTOR} \xrightarrow{m} (\text{PATIENT} \xrightarrow{m} \text{B_TEST}) \tag{98}$$

One good reason for this alternative is simply that we have one model instead of two and the necessity for having an invariant of the form given above has been eliminated completely.

There is, as one would expect, a connection between the two approaches, a connection which can be captured formally. From the single model δ we may extract the model of patient-blood test information. If we **assume** that all of the patient-blood test maps in δ are mutually disjoint then we may use the following expression

$$\varrho = \sqcup / \, rng \, \delta \tag{99}$$

Clearly the very validity of our solution depends completely on the validity of the application of reduction with respect to map extend, $\sqcup$, and this operator may validly be used iff the patient-blood test maps in δ are mutually disjoint. What a very strong assumption! Earlier in the tutorial we were very liberal (and realistic) in allowing a person to be the patient of more than one doctor. If we wish to be true to that liberalism (and realism) then we certainly can not use the extend operator. Let us explore the problem in some detail, i.e., debug the specification models.

Suppose that we have a δ of the form

$$\delta = [d \mapsto [p \mapsto t_1], d' \mapsto [p \mapsto t_2]] \tag{100}$$

which records the fact that two different doctors have the same patient. Application of the range operator gives

$$rng \, \delta = \{[p \mapsto t_1], [p \mapsto t_2]\} \tag{101}$$

The inappropriateness of the extend operator is all too apparent. But if we *were* able to find a function that computes the corresponding μ, what would the result look like? Clearly, the only possible solution ought to be of the form

$$[p \mapsto \{t_1, t_2\}] \tag{102}$$

which is an element of the space

$$\text{PATIENT} \xrightarrow{m} \mathcal{P}\,\text{B_TEST} \tag{103}$$

We seem to have got off to a very bad start. However, taking this seemingly negative result and turning it around into a positive one, allows us to illustrate most clearly how we employ such functions to expose certain types of inadequacies in the formal models. Such functions are customarily called **retrieve** functions and they feature prominently in the verification process of formal modelling. How happy we must feel to have caught such a pernicious bug at this early modelling stage—the requirements modelling/specification stage—long before we even entered into considerations of design and ultimate implementation.

We will have other reasons to discard our patient-blood test model later. Apart from the detected bug, there is another potential subtle one embedded in the revised model of patients and blood tests:

$$\text{PATIENT} \xrightarrow{m} \mathcal{P}\,\text{B_TEST} \tag{104}$$

Its discovery will be left as an exercise. For the present we adopt the irrealistic position that everything is in order and abide by the decision to insist on the assumption of mutually disjoint patient-blood test maps. Now we will turn to the other retrieve function by which ϱ may be extracted from δ, keeping our fingers crossed lest we uncover another difficulty. Happily, everything is in order:

$$\varrho = (\mathcal{I} \xrightarrow{m} dom)\delta \tag{105}$$

Having noted the uncovered problem for later resolution, we turn our attention to the centre for communicable diseases.

By law, each hospital laboratory is required to notify the national centre for disease control of all blood test results. However, for the purposes of confidentiality, the centre is prohibited from obtaining the names of the corresponding patients. On the other hand, the centre must be able to notify any registered doctor on the basis of a blood test result of one of his or her patients. We shall begin by supposing that the centre's information is modelled by

$$\varsigma \in \text{LAB} \xrightarrow{m} \mathcal{P}\,\text{DOCTOR} \tag{106}$$

$$\kappa \in \text{DOCTOR} \xrightarrow{m} (\text{B_TEST} \xrightarrow{m} \mathbb{N}'), \tag{107}$$

and propose to investigate the operations: (i) lab notifies centre, (ii) centre notifies lab and doctor.

Again we have two models and, therefore, we look carefully to see if an invariant must be specified that constrains the models appropriately.

It is quite clear that the set of doctors occurring in both models must be the same and we may record this observation formally:

$$\cup/\,rng\,\varsigma = dom\,\kappa \tag{108}$$

The submodel

$$\beta \in \text{B_TEST} \xrightarrow{m} \mathbb{N}' \tag{109}$$

represents a bag which is used to count the number of occurrences of a given test result. It is a monoid $(\text{B_TEST} \xrightarrow{m} \mathbb{N}', \oplus, \theta)$ where the binary operator is given by

$$\beta \oplus [t \mapsto n] = \begin{cases} \beta \sqcup [t \mapsto n], & \text{if } \neg\chi[\![t]\!]\beta \\ \beta \dagger [t \mapsto \beta(t) + n], & \text{otherwise} \end{cases} \tag{110}$$

In other words, the operator $\oplus$ inherits its properties from the addition of natural numbers. We recognise the similarity of this construction with the definition of the $\textcircled{\cup}$ operator defined earlier. This leads us to formulate a general theorem:

THEOREM **3.1 (Indexed Monoids)** *Let $(M, *, u)$ denote an arbitrary monoid, which we shall call the* base *monoid, with unit u, and $(M', *)$ the corresponding semigroup, i.e., with $M' = \lhd [\![u]\!] M$. Then for a domain X, the structure $(X \xrightarrow{m} M', \circledast, \theta)$ is an indexed monoid which inherits its operator properties from $(M, *, u)$, where for μ in $X \xrightarrow{m} M'$,*

$$\mu \circledast [x \mapsto m] = \begin{cases} \mu \sqcup [x \mapsto m], & \text{if } \neg\chi[\![x]\!]m \\ \mu \dagger [x \mapsto \mu(x) * m], & \text{otherwise} \end{cases} . \tag{111}$$

For convenience, let us denote by

$$X \circlearrowleft (M, *, u) \tag{112}$$

the construction of the indexed monoid from the base monoid. With this notation, we may express the additive monoid of bags $(X \xrightarrow{m} \mathbb{N}', \oplus, \theta)$ as $X \circlearrowleft (\mathbb{N}, +, 0)$ and the monoid $(X \xrightarrow{m} \mathcal{P}' Y, \bigcirc, \theta)$ as $X \circlearrowleft (\mathcal{P} Y, \cup, \emptyset)$.

We might be tempted to break off the treatment of the disease centre model at this point and devote ourselves to a further elaboration of the theory that has been opened up by this new theorem. However, there is yet much to be done and we must forgo such a pleasure and return to the grindstone.

As in the case of the lab we may consider a single model for the disease centre information:

$$\psi \in \text{LAB} \xrightarrow{m} (\text{DOCTOR} \xrightarrow{m} (\text{B_TEST} \xrightarrow{m} \mathbb{N}')) \tag{113}$$

To obtain the first of our two models, ς, we write

$$\varsigma = (\mathcal{I} \xrightarrow{m} dom)\psi \tag{114}$$

To obtain the second model, κ, we proceed as in the case of the single model for the hospital lab and try a reduction with respect to map extend over the range of ψ:

$$\kappa = {}^{\sqcup}/\, rng\, \psi \tag{115}$$

Once more this reduction is valid iff the doctor-blood test maps are mutually disjoint. We ask ourselves under what conditions such maps might not be disjoint? Clearly, in analogy to the hospital lab case, the specified retrieve function is invalid if we permit the same doctor to be registered with at least two distinct labs. Once more we have uncovered a pernicious bug of the **same class** as before. It may very well be the case that a consultant is registered with more than one hospital lab in the real world where the system is supposed to work. Whether this bug infects the two separate models ς and κ, is left as an exercise.

But, of course, one may argue that an experienced systems analyst would have uncovered such bugs in the requirements negotiations with the clients. Still it is comforting to know that they can be detected easily with formal mathematical procedures.

Let us proceed then with the models as given and use the assumption of one doctor–one lab. Our next assignment is the consideration of the two notify operations.

3.1 LAB NOTIFIES CENTRE

We will use the pairs of models for the lab and the centre, each of which is repeated here for convenience.

$$\varrho \in \text{PATIENT} \xrightarrow{m} \text{B_TEST} \tag{116}$$

$$\mu \in \text{DOCTOR} \xrightarrow{m} \mathcal{P}\,\text{PATIENT} \tag{117}$$

$$\varsigma \in \text{LAB} \xrightarrow{m} \mathcal{P}\,\text{DOCTOR} \tag{118}$$

$$\kappa \in \text{DOCTOR} \xrightarrow{m} (\text{B_TEST} \xrightarrow{m} \mathbb{N}') \tag{119}$$

Upon determination of the results of a blood test t for the patient p of doctor d, the laboratory l notifies the centre. We expect the operation to have the form:

$$\text{Notify}[\![l, d, t]\!](\varrho, \mu, \varsigma, \kappa) \triangleq \ldots \tag{120}$$

We will certainly expect that the notify operation has a pre-condition which we shall now determine. First we must ensure that the blood test t was recorded at the lab l:

$$\chi[\![t]\!] rng\, \varrho \tag{121}$$

But the curious will note that we did not make use of the lab's identifier l. It is certainly required by the centre but the domain LAB is not mentioned anywhere in models ϱ, μ. This anomaly must be rectified. It is precisely to remove such anomalies that one introduces a homunculus model for the hospital laboratory, such as

$$\mu' \in \text{LAB} \xrightarrow{m} (\text{DOCTOR} \xrightarrow{m} \mathcal{P}\,\text{PATIENT}) \tag{122}$$

But now that we have identified the problem and its solution, we will proceed with the original models.

Continuing with the pre-condition analysis, we require that the doctor in question be registered with the lab

$$\chi[\![d]\!]\mu \tag{123}$$

and that the patient p in question is a patient of doctor d:

$$\varrho^{-1}(t) \cap \mu(d) \neq \emptyset \tag{124}$$

Our pre-condition thus far was concerned with the validity of data at the specific hospital lab. This involved checking the first pair of models. What about validity with respect to the models of the centre?

We immediately identify distinct cases. Suppose that the lab l is not registered at the centre and neither is the doctor. Then we would have

$$\text{Notify}[\![l, d, t]\!](\varrho, \mu, \varsigma, \kappa) \triangleq (\varrho, \mu, \varsigma \sqcup [l \mapsto \{d\}], \kappa \sqcup [d \mapsto [t \mapsto 1]]) \tag{125}$$

where the use of the extend operator indicates that we are adding the conditions $\neg\chi[\![l]\!]\varsigma$ and $\neg\chi[\![d]\!]\kappa$.

Now suppose that the lab is not registered, but that the doctor is registered with some other lab. The very nature of the model pair allows us to capture the specification accurately:

$$\text{Notify}[\![l,d,t]\!](\varrho,\mu,\varsigma,\kappa) \triangleq (\varrho,\mu,\varsigma \sqcup [l \mapsto \{d\}], \kappa \dagger [d \mapsto \kappa(d) \oplus [t \mapsto 1]]) \tag{126}$$

where the extra conditions are given by

$$\neg\chi[\![l]\!]\varsigma \wedge \chi[\![d]\!]\kappa \tag{127}$$

Or it may be the case that the lab is registered and the doctor is not ...

After an exhaustive analysis, it becomes quite clear that we need to bring some order to the variants of the notify operation. One obvious solution, which usually occurs to programmers, using conventional languages, is to distinguish the various cases by writing nested if-then-else statements. From experience we know that this route is fraught with difficulties and provides a golden opportunity for the independent testers.

For those who are encouraged to use modular programming lanuages or object-oriented languages, it will be apparent that the preferred route is to introduce a separate register operation. Thus, if a lab is not registered with the centre, it **must** do so before transmitting data. Such a solution would occur to programmers who are familiar with the model of cellular telephony, or with those engaged in advanced broadband communications services (whether multimedia or not). The register operation for the lab is simply

$$\text{Register}[\![l]\!](\varrho,\mu,\varsigma,\kappa) \triangleq (\varrho,\mu,\varsigma \sqcup [l \mapsto \emptyset], \kappa) \tag{128}$$

The occurrence of the empty set in the expression causes us to reflect whether or not one ought to register the lab's doctors D at the same time that one registers the lab. If we decide to go this route, then ς becomes modified to

$$\varsigma \sqcup [l \mapsto D] \tag{129}$$

and subsequent new affiliates of the lab may be registered as and when the need arises. But registering the doctors has an impact on the κ model and we must take into account whether or not the doctor to be registered for lab l is not, in fact, already registered with another lab l'. The resolution of this particular complication is left as an exercise, the solution of which will depend heavily on the degree of committment to the pairs of model specification or whether a different model is to be chosen.

3.2 CENTRE NOTIFIES LAB AND DOCTOR

Having determined that the results of a particular blood test t are significant, the centre notifies all the labs and affiliated doctors of the result. The specification will have the form:

$$\text{Notify}[\![t]\!](\varsigma, \kappa) \triangleq \ldots \tag{130}$$

For this specification we will be content to identify the doctors concerned, and then from this information to identify the corresponding labs. One may check the validity of the given significant blood test in a manner similar to the method used in the previous notify operation. Here we will concentrate on extracting the relevant information.

First, since the input is simply a blood test and no numerical information is supplied then we will construct a derived model:

$$\kappa' = (\mathcal{I} \xrightarrow{m} dom)\kappa \in \text{DOCTOR} \xrightarrow{m} \mathcal{P}\,\text{B_TEST} \tag{131}$$

Using the inversion algorithm given earlier, we may construct another derived model:

$$\kappa'' = \text{Invert}(\kappa') \in \text{B_TEST} \xrightarrow{m} \mathcal{P}\,\text{DOCTOR} \tag{132}$$

recalling that the validity of the inversion depends on prohibiting entries of the form $d \mapsto \emptyset$, a situation which can only arise if a doctor is registered with the centre but there is no record of a blood test for any of her/his patients. Hence, we must make sure that registration of doctors does not leave the centre's database in a state which would cause the inversion to fail!

The map κ'' may now be used to give the set of all doctors who have to be notified:

$$((\mathcal{I} \xrightarrow{m} dom)\kappa)^{-1}(t) \tag{133}$$

where we have chosen to eliminate the auxillary variables κ', κ'' and replace the word invert with the appropriate symbol. If one finds this expression to be too abstract then the recommendations on verbosity given in § 2 of the tutorial may be applied.

From the lab-doctors model, we apply the inversion once again to obtain the derived model

$$\varsigma' = \text{Invert}(\varsigma) \in \text{DOCTOR} \xrightarrow{m} \mathcal{P}\,\text{LAB} \tag{134}$$

Once again, validity depends on the non-occurrence of an entry of the form $[l \mapsto \emptyset]$ in ς. In other words, registration of a lab, without further information, will put the centre's database into a state which will cause the inversion to fail.

We may now use the set of doctors to be notified to restrict this derived model to give both the set of doctors and the corresponding labs that both need to be notified. Resorting to κ'', this may be expressed in the form:

$$\lhd\ [\![\kappa''(t)]\!]\varsigma' \tag{135}$$

For the stout of heart, we express the same result in terms of the original variables and eliminate the invert operation name to give

$$\lhd \llbracket((\mathcal{I} \xrightarrow{m} dom)\kappa)^{-1}(t)\rrbracket\varsigma^{-1} \tag{136}$$

Finally, it must be remarked that the restriction operator presented in this solution is another monoid endomorphism which is the dual of the removal monoid endomorphism. Rather than give the details here, the reader is referred to [10].

In specifying the notify operations we have tacitly assumed that communication is carried out directly from lab to centre and vice-versa in a simple abstract network that might be assumed to have a star configuration. We certainly did not enter into any discussion as to what the actual communications network might look like. To develop the application further we now look at a simple abstract model of such a network and use it as a vehicle to illustrate what proofs look like in the $VDM^{\clubsuit}$.

4 Proof

Earlier we made the statement that proofs are constructions in the $VDM^{\clubsuit}$. In most other formal models proofs are based on the use of formal logic. To illustrate the distinction we choose a simple model of a communications network with switching nodes, give a simple invariant in the formal logic style and remark upon its properties. Then an alternative constructive form of the invariant is supplied. Finally, the power of the method is exhibited by considering satisfaction of the invariant by an operation on the network and **calculation** of the necessary and sufficient pre-condition by constructive proof. Those who advocate formal logic for proof are invited to do likewise in their method.

The material in this section was first presented by the author at a guest lecture in the Technical University of Delft, December 1992, and the material appears here in print for the first time. The results are believed to be new.

4.1 The Network Model

Let us consider a distributed system whose network topology is a ring. Using a graph representation, one may identify the nodes with processes and edges with communication channels [14] p. 27. A graph may, in turn, be represented as a map μ in the space [11] p. 221:

$$\mu \in X \xrightarrow{m} \mathcal{P}\,X \tag{137}$$

But not every μ will represent a ring. We know that the same space models directed acyclic graphs, which are, for example, representations of bills of material [11] p. 218.

To identify precisely that subspace which models rings we supply an invariant. For the present we illustrate the model of an n-node unidirectional ring by

$$\mu = \left[\begin{array}{ccc} x^1 & & \{x^2\} \\ x^2 & & \{x^3\} \\ \vdots & \mapsto & \vdots \\ x^{n-1} & & \{x^n\} \\ x^n & & \{x^1\} \end{array} \right] \tag{138}$$

In a general network topology, it must be the case that every node x is connected to at least one other node. Supposing that the edges are to represent bidirectional channels, then

$$x \mapsto \{x^1, x^2, \ldots, x^j\} \tag{139}$$

would represent a typical node's connectivity. A more detailed analysis on the distinctions bewteen unidirectional and bidirectional links will be developed later when we make a distinction between a fixed network topology and the dynamic selection of routings based on sender-receiver pairs.

4.2 The Invariant

Whatever the intended network topology, it is clear that it is never the case that a node be self-connected, i.e., there are no 'direct' cycles in the network. Thus elements of the form

$$x \mapsto \{\ldots, x, \ldots\} \tag{140}$$

are definitely to be prohibited. We may, therefore, use this information to constrain the space $X \xrightarrow{m} \mathcal{P}X$ by imposing the invariant:

$$inv(\mu) \triangleq \forall x \in dom\,\mu, x \notin \mu(x) \tag{141}$$

There may well be other constraints that one might impose. We shall however, content ourselves with an analysis of that given.

From the perspective of the logician, the given invariant is concise, accurate and elegant. But from the perspective of the formal methodist who insists on the application of **formal** logic, then the calculation we are about to perform, may cause certain technical difficulties—a challenge!

In the *VDM*$^{\clubsuit}$ we eschew writing universal and existential quantifiers. Thus are we impelled to seek an alternative constructive form of the invariant. One solution follows.

The general strategy is to take an element of the form

$$x \mapsto S, \tag{142}$$

tuple the domain and range to give

$$x \mapsto (x, S), \tag{143}$$

and iterate over the resulting range with the characteristic function to produce

$$x \mapsto \chi(x, S), \tag{144}$$

which is formally equivalent to

$$x \mapsto \chi[\![x]\!]S. \tag{145}$$

If x is not a member of S, then our construction will produce $x \mapsto 0$. We now have some choices on how to proceed to establish the required invariant. Before illustrating these, let us first present the construction that achieves the above result.

For each $\mu \in X \xrightarrow{m} \mathcal{P}X$ we introduce the corresponding 'identity map' $\mathcal{I}_\mu$ which takes each element x in the domain of μ to itself:

$$\mathcal{I}_\mu \triangleq \{x \mapsto x \mid \chi[\![x]\!]\mu\} \tag{146}$$

Alternatively, since $\mathcal{I}$ denotes **the** identity map of $X \xrightarrow{m} \mathcal{P}X$, then

$$\mathcal{I}_\mu \triangleq \lhd\ [\![dom\,\mu]\!]\mathcal{I} \tag{147}$$

Now, for each $\mu \in X \xrightarrow{m} \mathcal{P}X$, we form the extended map

$$\mathcal{I}_\mu \bowtie \mu \tag{148}$$

where $\bowtie$ is the join or 'pairing operator' which tuples a pair of maps with common domains [10]. Iteration over the range with respect to the characteristic function is readily achieved:

$$(\mathcal{I} \xrightarrow{m} \chi)(\mathcal{I}_\mu \bowtie \mu) \tag{149}$$

which constructs a map in $X \xrightarrow{m} \mathbb{B}$.

Remark: One will note that $\mathcal{I}_\mu$ is referred to as an 'identity' map, the quotes being significant. The issue here is that we already have a unique identity function $\mathcal{I}$. In our effort to be rigorous we have written $\mathcal{I}_\mu$ for the construction of the required tupled map, but $\mathcal{I}$ for the iteration. The difference arises from our insistence that the operator $\bowtie$ is defined over a pair of maps if and only if said maps have the same domain.

The required invariant may now be given in either of the two forms

$$\Big((\mathcal{I} \xrightarrow{m} \chi)(\mathcal{I}_\mu \bowtie \mu)\Big)^{-1}(1) = \emptyset \tag{150}$$

where we use the inverse image to check for the non-occurrence of x in S, or

$$rng(\mathcal{I} \xrightarrow{m} \chi)(\mathcal{I}_\mu \bowtie \mu) = \{0\} \tag{151}$$

where we rely on the property of non-duplication of elements in a set to arrive at the equivalent result.

4.3 THE OPERATION

Let us suppose that nodes in the network are taken out of service, for whatever reason, and then brought back on line later. In a non-stop system, the outage may be constrained to be as little as 5 minutes per year! Let x' denote a node that is to be brought back on line. From the fixed network topology, it has been determined that x' is to be linked to node x. There will be other linkages to be restablished, of course. Confining ourselves to this single re-connection, we specify it as an enter command E as follows:

$$E[\![x, x']\!]\mu = \begin{cases} \mu \sqcup [x \mapsto \{x'\}], & \neg\chi[\![x]\!]\mu \\ \mu \dagger [x \mapsto \mu(x) \cup \{x'\}], & \text{otherwise} \end{cases} \tag{152}$$

Note the care that we take in considering whether or not x' has already been linked to some other node in the network. The mathematical expression we have seen before. Thus we might prefer to have written:

$$E[\![x, x']\!]\mu = \mu \textcircled{\cup} [x \mapsto \{x'\}] \tag{153}$$

No pre-condition is given. We shall now calculate the necessary and sufficient pre-condition by attempting to demonstrate that the enter operation satisfies the invariant.

4.4 THE CONSTRUCTIVE CALCULATION

For convenience, we write $\nu = E[\![x, x']\!]\mu = \mu \textcircled{\cup} [x \mapsto \{x'\}]$.

We are not yet able to reason efficiently with the $\textcircled{\cup}$ operator and must resort to its basic parts. In other words, we note that there is still work to be done. Hence, the proof will be accomplished by cases.

Case A: $\nu = E[\![x, x']\!]\mu = \mu \sqcup [x \mapsto \{x'\}]$, if $\neg\chi[\![x]\!]\mu$.

The first step is to construct the identity map and apply the join operator to tuple the resulting pair of maps:

$$\begin{aligned} 8 \quad & \mathcal{I}_\nu \bowtie \nu \\ .1 \quad & = (\mathcal{I}_\mu \sqcup [x \mapsto x]) \bowtie (\mu \sqcup [x \mapsto \{x'\}]) \\ .2 \quad & = (\mathcal{I}_\mu \bowtie \mu) \sqcup [x \mapsto (x, \{x'\})] \end{aligned}$$

Next we iterate over the result by applying the characteristic function to the range, to give:

$$\begin{aligned} 9 \quad & (\mathcal{I} \xrightarrow{m} \chi)(\mathcal{I}_\nu \bowtie \nu) \\ .1 \quad & = (\mathcal{I} \xrightarrow{m} \chi)((\mathcal{I}_\mu \bowtie \mu) \sqcup [x \mapsto (x, \{x'\})]) \\ .2 \quad & = (\mathcal{I} \xrightarrow{m} \chi)(\mathcal{I}_\mu \bowtie \mu) \sqcup (\mathcal{I} \xrightarrow{m} \chi)[x \mapsto (x, \{x'\})] \\ .3 \quad & = (\mathcal{I} \xrightarrow{m} \chi)(\mathcal{I}_\mu \bowtie \mu) \sqcup [x \mapsto \chi(x, \{x'\})] \\ .4 \quad & = (\mathcal{I} \xrightarrow{m} \chi)(\mathcal{I}_\mu \bowtie \mu) \sqcup [x \mapsto \chi[\![x]\!]\{x'\}] \end{aligned}$$

At this point we have a choice between application of a range operator, which readily establishes the pre-condition we are searching for, or application of the more exotic looking inverse image operator which is technically challenging. We choose the latter, exhibiting new theoretical results as a consequence, and leave the former as a simple exercise. Thus, application of the inverse image operator gives:

$$
\begin{array}{ll}
10 & ((\mathcal{I} \xrightarrow{m} \chi)(\mathcal{I}_\nu \bowtie \nu))^{-1} \\
.1 & = ((\mathcal{I} \xrightarrow{m} \chi)(\mathcal{I}_\mu \bowtie \mu) \sqcup [x \mapsto \chi[\![x]\!]x'])^{-1} \\
.2 & = ((\mathcal{I} \xrightarrow{m} \chi)(\mathcal{I}_\mu \bowtie \mu))^{-1} \circledcirc [x \mapsto \chi[\![x]\!]x']^{-1} \\
.3 & = ((\mathcal{I} \xrightarrow{m} \chi)(\mathcal{I}_\mu \bowtie \mu))^{-1} \circledcirc [\chi[\![x]\!]x' \mapsto \{x\}]
\end{array}
$$

Remark: The passage from **10.1** to **10.2** is accomplished with a new theorem on inverse image maps:

THEOREM **4.1** *For any pair of disjoint maps μ and ν in $X \xrightarrow{m} Y$,*

$$(\mu \sqcup \nu)^{-1} = \mu^{-1} \circledcirc \nu^{-1} \tag{154}$$

where μ^{-1}, ν^{-1}, $(\mu \sqcup \nu)^{-1}$ in $Y \xrightarrow{m} \mathcal{P}'X$, and, in addition, for an evaluation at some y

$$
\begin{aligned}
(\mu \sqcup \nu)^{-1}(y) &= (\mu^{-1} \circledcirc \nu^{-1})(y) && (155) \\
&= \mu^{-1}(y) \cup \nu^{-1}(y) && (156)
\end{aligned}
$$

Now we simply evaluate the result as required:

$$
\begin{array}{ll}
11 & ((\mathcal{I} \xrightarrow{m} \chi)(\mathcal{I}_\nu \bowtie \nu))^{-1}(1) \\
.1 & = ((\mathcal{I} \xrightarrow{m} \chi)(\mathcal{I}_\mu \bowtie \mu) \sqcup [x \mapsto \chi[\![x]\!]\{x'\}])^{-1}(1) \\
.2 & = ((\mathcal{I} \xrightarrow{m} \chi)(\mathcal{I}_\mu \bowtie \mu))^{-1}(1) \cup [\chi[\![x]\!]\{x'\} \mapsto \{x\}](1) \\
.3 & = \emptyset \cup [\chi[\![x]\!]\{x'\} \mapsto \{x\}](1) \\
.4 & = [\chi[\![x]\!]\{x'\} \mapsto \{x\}](1) \\
.5 & = \emptyset \text{ iff } \chi[\![x]\!]\{x'\} \neq 1 \\
.6 & \quad \Rightarrow x \notin \{x'\} \\
.7 & \quad \Rightarrow x \neq x'
\end{array}
$$

which establishes the pre-condition for Case A. Note the evaluation step in the passage from **11.1** to **11.2**.

In the course of the proof, there were many other steps that we ought to have annotated, in particular giving the details of the various lemmas and theorems that follow as a consequence. However, such a detailed exposition would have clouded the overall structure of the proof, and the pride of place given to the theorem highlighted, might have been obscured.

The proof of the first case was relatively simple as expected. Proofs which involve disjoint maps usually are. The real technical problems we expect to meet when we consider the override operator, to which we turn next. However, before doing so, it is quite proper to remark on the distinction between the two resultant forms of the pre-condition: $x \notin \{x'\}$ versus $x \neq x'$. The latter we must use if we adhere to the form of the parameters supplied to the enter operation. The former is, in fact, more elegant and more general, suggesting a different form of the parameter list that permits a more general enter operation.

Case B: $\nu = E[\![x, x']\!]\mu = \mu \dagger [x \mapsto \mu(x) \cup \{x'\}]$, if $\chi[\![x]\!]\mu$.

The procedure is exactly the same as for Case A. The noteworthy aspect is the rewriting of the override operator in terms of the extend operator:

$$
\begin{array}{ll}
12\ \mathcal{I}_\nu \bowtie \nu & \\
\quad .1 & = \mathcal{I}_\mu \bowtie (\mu \dagger [x \mapsto \mu(x) \cup \{x'\}]) \\
\quad .2 & = (\lhd [\![x]\!]\mathcal{I}_\mu \sqcup \mathcal{I}_x) \bowtie (\lhd [\![x]\!]\mu \sqcup [x \mapsto \mu(x) \cup \{x'\}]) \\
\quad .3 & = (\lhd [\![x]\!]\mathcal{I}_\mu \bowtie \lhd [\![x]\!]\mu) \sqcup (\mathcal{I}_x \bowtie [x \mapsto \mu(x) \cup \{x'\}]) \\
\quad .4 & = \lhd [\![x]\!](\mathcal{I}_\mu \bowtie \mu) \sqcup ([x \mapsto (x, \mu(x) \cup \{x'\})]) \\
\quad .5 & = (\mathcal{I}_\mu \bowtie \mu) \dagger ([x \mapsto (x, \mu(x) \cup \{x'\})])
\end{array}
$$

In the next stage the inter-conversion between the override operator and the extend operator and vice-versa has been condensed for simplicity. One may follow the paradigm established above to check the actual intermediate details, if necessary.

$$
\begin{array}{ll}
13\ (\mathcal{I} \xrightarrow{m} \chi)(\mathcal{I}_\nu \bowtie \nu) & \\
\quad .1 & = (\mathcal{I} \xrightarrow{m} \chi)(\mathcal{I}_\mu \bowtie \mu) \dagger [x \mapsto \chi[\![x]\!](\mu(x) \cup \{x'\})]
\end{array}
$$

Choosing once more the application of the technically difficult inverse image operator, gives:

$$
\begin{array}{ll}
14\ ((\mathcal{I} \xrightarrow{m} \chi)(\mathcal{I}_\nu \bowtie \nu))^{-1} & \\
\quad .1 & = ((\mathcal{I} \xrightarrow{m} \chi)(\mathcal{I}_\mu \bowtie \mu) \dagger [x \mapsto \chi[\![x]\!](\mu(x) \cup \{x'\})])^{-1} \\
\quad .2 & = (\mathcal{I} \xrightarrow{m} \lhd [\![x]\!])'((\mathcal{I} \xrightarrow{m} \chi)(\mathcal{I}_\mu \bowtie \mu))^{-1} \circledcirc [\chi[\![x]\!](\mu(x) \cup \{x'\}) \mapsto \{x\}]
\end{array}
$$

Remark: Justification of this result is based upon the more general theorem which is the counterpart of that given above:

THEOREM **4.2** *For any pair of maps μ and ν in $X \xrightarrow{m} Y$,*

$$(\mu \dagger \nu)^{-1} = (\mathcal{I} \xrightarrow{m} \lhd [\![\, dom\, \nu]\!])'\mu^{-1} \circledcirc \nu^{-1} \tag{157}$$

where μ^{-1}, ν^{-1}, $(\mu \dagger \nu)^{-1}$ in $Y \xrightarrow{m} \mathcal{P}' X$, and, in addition, for an evaluation at some y

$$
\begin{array}{rcll}
(\mu \dagger \nu)^{-1}(y) & = & ((\mathcal{I} \xrightarrow{m} \lhd [\![\, dom\, \nu]\!])'\mu^{-1} \circledcirc \nu^{-1})(y) & (158) \\
 & = & (\mathcal{I} \xrightarrow{m} \lhd [\![\, dom\, \nu]\!])'\mu^{-1}(y) \cup \nu^{-1}(y) & (159)
\end{array}
$$

The priming on $(\mathcal{I} \xrightarrow{m} \lhd [\![dom\, \nu]\!])$ denotes the fact that if as a result of the deletion we obtain an entry of the form $[y \mapsto \emptyset]$, then this must be deleted from the resulting inverse image map. We may establish this theorem by noting that

$$(\mu \dagger \nu)^{-1} = (\lhd [\![dom\, \nu]\!] \mu \sqcup \nu)^{-1} \quad (160)$$

$$= (\lhd [\![dom\, \nu]\!] \mu)^{-1} \circledcirc \nu^{-1} \quad (161)$$

Finally, for any set S in $\mathcal{P}\, Y$, it may be shown that

$$(\lhd [\![S]\!] \mu)^{-1} = (\mathcal{I} \xrightarrow{m} \lhd [\![S]\!])' \mu^{-1} \quad (162)$$

from which the theorem follows immediately.

It goes without saying that this result reduces to that given earlier for the inverse image of disjoint maps. Other significant theoretical consequences arise from the two theorems given, but this is not the place to present them.

Continuing with the rest of the calculation, we arrive at the expected result:

15 $((\mathcal{I} \xrightarrow{m} \chi)(\mathcal{I}_\nu \bowtie \nu))^{-1}(1)$

.1 $= (\mathcal{I} \xrightarrow{m} \lhd [\![x]\!])'((\mathcal{I} \xrightarrow{m} \chi)(\mathcal{I}_\mu \bowtie \mu))^{-1}(1) \cup [\chi[\![x]\!](\mu(x) \cup \{x'\}) \mapsto \{x\}](1)$

.2 $= ((\mathcal{I} \xrightarrow{m} \chi)(\mathcal{I}_\mu \bowtie \mu))^{-1}(1) \cup [\chi[\![x]\!](\mu(x) \cup \{x'\}) \mapsto \{x\}](1)$

.3 $= [\chi[\![x]\!](\mu(x) \cup \{x'\}) \mapsto \{x\}](1)$

.4 $= \emptyset$ iff $\chi[\![x]\!](\mu(x) \cup \{x'\}) \neq 1$

.5 $\Rightarrow x \notin (\mu(x) \cup \{x'\})$

.6 $\Rightarrow x \notin \mu(x) \vee x \notin \{x'\}$

.7 $\Rightarrow x \notin \{x'\}$

.8 $\Rightarrow x \neq x'$

which gives the pre-condition for Case B. Happily, it is the same pre-condition as in Case A. A different result would have been disasterous.

In this example of invariant satisfaction we were fortunate indeed to have two distinct forms of the invariant, both of which are suitable for calculating the pre-condition. That proofs using both forms give the same result, is pre-eminently satisfying.

Regretfully, space does not permit us to explore this issue further here. Work that remains to be done, includes a complete listing of all operations on the network and the accompanying pre-conditions, computed in the manner demonstrated. Although the details may (appear to) be tedious, it is comforting to know that once done, the results will endure forever. Therein lies the real benefit of the formal methods.

Having introduced the notion of a network topology, it seems fitting that we consider some further aspects of a communications network in connection with the medical part of our system. The next section is dedicated to the network view of the notify operations and other network issues.

5 Network Topology, Routing and Node storage

The communications network for the medical system was assumed to be a star configuration. Let us now, though still keeping that same configuration, introduce the distinction between information held at a site and information at a network node as illustrated in Fig. 1.

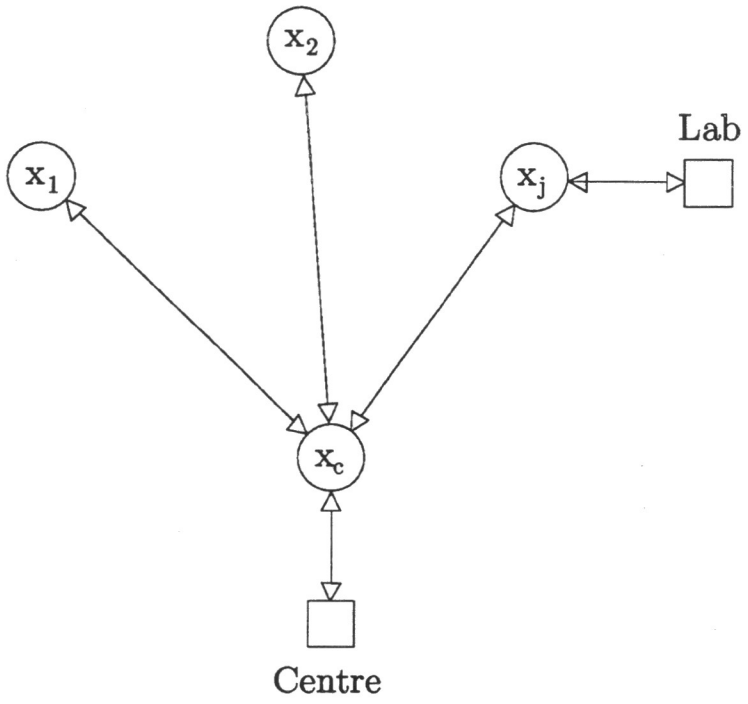


Figure 1: Network topology

For this section, we will use the more general single models for both lab site and centre:

$$\delta \in \text{DOCTOR} \xrightarrow{m} (\text{PATIENT} \xrightarrow{m} \text{B_TEST}) \tag{163}$$

$$\psi \in \text{LAB} \xrightarrow{m} (\text{DOCTOR} \xrightarrow{m} (\text{B_TEST} \xrightarrow{m} \mathbb{N}')) \tag{164}$$

The network topology is modelled, as in the previous section, by

$$\nu \in X \xrightarrow{m} \mathcal{P}\, X \tag{165}$$

subject to the invariant that there are no self-loops:

$$((\mathcal{I} \xrightarrow{m} \chi)(\mathcal{I}_\nu \bowtie \nu))^{-1}(1) = \emptyset \tag{166}$$

Such a model is inherently asymmetrical, recording essentially unidirectional connectivity. That there is connectivity in the opposite direction, is modelled by

$$\nu^{-1} \in X \xrightarrow{m} \mathcal{P}\, X \tag{167}$$

In addition we will specify an invariant to capture the notion that all of the links are bidirectional:

$$\nu = \nu^{-1} \tag{168}$$

Alternatively, we may introduce a new operation based on set intersection and formulate the invariant as

$$\nu \circledcap' \nu^{-1} = \nu \tag{169}$$

where $\circledcap$ is defined by

$$\nu \circledcap [x \mapsto S] = \begin{cases} \theta, & \text{if } \neg\chi[\![x]\!]\nu, \\ \nu \dagger [x \mapsto \nu(x) \cap S], & \text{otherwise} \end{cases} \tag{170}$$

As usual, the prime denotes the removal of resulting entries of the form $x \mapsto \emptyset$.

Then for a network with unidirectional links only, the invariant is

$$\nu \circledcap' \nu^{-1} = \theta \tag{171}$$

Networks with both bidirectional and unidirectional links will have an invariant of the general form

$$\theta \prec \nu \circledcap' \nu^{-1} \prec \nu \tag{172}$$

Given a sender-receiver pair, and the network topology, we may then extract a model of the possible routings, ignoring for the present any issue of link capacity. Such routings are also modelled by

$$\rho \in X \xrightarrow{m} \mathcal{P}\, X \tag{173}$$

Finally, to complete the conceptual framework, we need to give a model for node storage, since we are implicitly assuming a store and forward packet switched network. Each packet will contain a header, giving the send-receiver 'addresses' and the data, which we will model as:

$$\sigma \in X \xrightarrow{m} \mathcal{P}(X^2 \times (\text{DOCTOR} \xrightarrow{m} \mathcal{P}\, \text{B_TEST})) \tag{174}$$

5.1 Forward & Store

Whereas the lab and the centre view the transmission of data as notify operations, the network sees the same transmission as the forwarding of data from one node (which is a removal operation) to some other node (which is an enter operation). Such removal and enter operations will involve the network storage model only. In other words,

we will work entirely on the assumption of the star configuration, it being one of the simplest topologies. In particular, we can safely ignore the routing problem.

For convenience, we will work with a typical network storage example of the form

$$\sigma = \left[\begin{array}{lcl} x_1 & \mapsto & \{(\eta_1^1, \phi_1^1), (\eta_1^2, \phi_1^2)\} \\ x_2 & \mapsto & \{(\eta_2^1, \phi_2^1), (\eta_2^2, \phi_2^2)\} \\ \ldots & & \ldots \\ x_c & \mapsto & \{(\eta_c^1, \phi_c^1), (\eta_c^2, \phi_c^2), \ldots, (\eta_c^n, \phi_c^n)\} \end{array} \right] \tag{175}$$

where $\eta = (x_s, x_r)$ is the header, and $\phi \in \text{DOCTOR} \xrightarrow{m} \mathcal{P}\,\text{B_TEST}$ is the data. The example illustrates the fact that at each lab node, there are two possible types of data, those that it is sending to the centre, and those that has been sent to it by the centre. For information, we note that

$$(\mathcal{I} \xrightarrow{m} \mathcal{P}\,\pi_1)\sigma \tag{176}$$

extracts a derived map of all the header information. The derived map of data information is

$$(\mathcal{I} \xrightarrow{m} \mathcal{P}\,\pi_2)\sigma \tag{177}$$

Let us deal with the removal operation first. We expect the specification to have the form

16 $R\colon X^2 \times (\text{DOCTOR} \xrightarrow{m} \text{B_TEST}) \longrightarrow (\text{STORAGE} \longrightarrow \text{STORAGE})$

.1 $R[\![(x_j, x_k), [d \mapsto t]]\!]\sigma \triangleq \ldots$

where it is clear that the data part of the (virtual) packet is a doctor, bloodtest pair. We really ought to introduce another model for the virtual packet and bring some more structure into the conceptual framework.

As far as the remove operation is concerned, we might consider a pre-condition of the form $\chi[\![x_j]\!]\sigma = \chi[\![\pi_1\eta]\!]\sigma$, where we suppose that the forwarding is from node x_j to x_k.

But from the model of storage, and repeated forwarding from the same node, it is clear that we may arrive at the situation where

$$\sigma(x) = \emptyset \tag{178}$$

We may wish to interpret this to mean that there is no data at the x node for onward transmission. We will treat such states as bugs, preferring to delete the entry. Similarly, there is also the possibility of $\chi[\![(\eta, \theta)]\!]\sigma(x)$ occurring, which indicates that all the packets associated with the header η have been forwarded. Again this is interpreted as a bug. We resolve this as above.

Given a header $\eta = (x_j, x_k)$, we use it to pick out the corresponding ϕ from $\sigma(x_j)$ and give the specification as a recursive algorithm:

$$
\begin{array}{ll}
17 & S\colon X^2 \longrightarrow (X^2 \times (\text{DOCTOR} \xrightarrow{m} \mathcal{P}\,\text{B_TEST}) \longrightarrow (\text{DOCTOR} \xrightarrow{m} \mathcal{P}\,\text{B_TEST})) \\
.1 & S[\![(x_j, x_k)]\!]\sigma(x_j) \triangleq \\
.2 & \quad \text{let } (\eta, \phi) \in \sigma(x_j) \text{ in} \\
.3 & \quad\quad (x_j, x_k) = \eta \\
.4 & \quad\quad\quad \rightarrow \phi \\
.5 & \quad\quad\quad \rightarrow S[\![(x_j, x_k)]\!] \lhd [\![(\eta, \phi)]\!]\sigma(x_j)
\end{array}
$$

where, for validity, we must have the pre-condition $\chi[\![(x_j, x_k)]\!]\sigma(x_j)$. The recursive algorithm may be converted into an operator form such as $(x_j, x_k) \Downarrow \sigma(x_j)$, a process with which we are not very happy, leading as it does to a proliferation of needless operator symbols, and against which we often rail.

The next step in the specification is the removal of $[d \mapsto t]$ from $\phi = S[\![(x_j, x_k)]\!]\sigma(x_j)$. We will denote this by a primed removal:

$$R'[\![[d \mapsto t]]\!]\phi \tag{179}$$

taking care to delete entries of the form $d \mapsto \emptyset$. Such a removal is the analogue of bag diminution, which is formulated similarly to the subtraction of natural numbers—the predecessor function. Specifically, the bag diminution operator, $\ominus$, is defined by

$$\beta \ominus [x \mapsto 1] = \begin{cases} \lhd [\![x]\!]\beta, & \text{if } \beta(x) = 1, \\ \beta \dagger [x \mapsto \beta(x) - 1], & \text{otherwise} \end{cases} \tag{180}$$

The counterpart for the monoid of sets, $(X \xrightarrow{m} \mathcal{P}' Y, \textcircled{\cup}, \theta)$, may be defined similarly:

$$R'[\![[x \mapsto \{y\}]]\!]\mu = \begin{cases} \lhd [\![x]\!]\mu, & \text{if } \mu(x) = \{y\}, \\ \mu \dagger [x \mapsto \lhd [\![y]\!]\mu(x)], & \text{otherwise} \end{cases} \tag{181}$$

It should be quite obvious that the operator symbol, $\lhd$, and the function symbol, R, are inter-changeable.

To denote the updated information at node x_j as a result of forwarding $[d \mapsto t]$, we may now write

$$\lhd [\![(\eta, \phi)]\!]\sigma(x_j) \cup \{(\eta, R'[\![[d \mapsto t]]\!]\phi)\} \tag{182}$$

a specification that recalls the definition of the override operator in terms of extend. Hence, the specification of removal, which we have developed so far, reads

$$R[\![(x_j, x_k), [d \mapsto t]]\!]\sigma \triangleq \sigma \dagger [x_j \mapsto \lhd [\![(\eta, \phi)]\!]\sigma(x_j) \cup \{(\eta, R'[\![[d \mapsto t]]\!]\phi)\}] \tag{183}$$

But we are now very conscious of the bugs that may arise. In this case, by repeatedly forwarding the same doctor's blood test data, it is possible to obtain an entry of the form

$$x_j \mapsto \{(\eta, \theta), \ldots\} \tag{184}$$

Thus, in the event that $R'[\![[d \mapsto t]]\!]\phi = \theta$, we add the extra clause

$$R[\![(x_j, x_k), [d \mapsto t]]\!]\sigma = \sigma \dagger [x_j \mapsto \lhd [\![((x_j, x_k), \phi)]\!]\sigma(x_j)] \tag{185}$$

Finally, this new clause may lead to an entry of the form

$$x_j \mapsto \emptyset \tag{186}$$

giving us the bug on the higher level. This bug is eliminated by priming the outer removal operator. To complete the definition of the removal operation, we really ought to eliminate the temporary variable that we have been using, $\phi = S[\![(x_j, x_k)]\!]\sigma(x_j)$, and which we leave as an exercise.

Quite clearly, the definition of the forward part has resulted in a very complex mathematical expression. The complexity arose out of dealing with null structures, θ, and $\emptyset$, introduced as a direct result of the removal operator. As a result we are led to consider whether the specification can be lifted to a higher level of abstraction by introducing new monoids or new morphisms. Alternatively, we also consider changing the basic model representation. But all such activity is meat for the School of the *VDM*$^{\clubsuit}$. For all its complexity, the expression is at least manageable compared to the actual code that one might find in the real system.

Let us now look at the store part of the forward and store operation. Fortunately, the expression will not be so complex. Due to our policy of deleting (η, ϕ) entries, then it is quite possible that there is no destination entry in the model, i.e., $x_k \mapsto \emptyset$. Hence, we expect

$$E[\![(x_j, x_k), [d \mapsto t]]\!]\sigma \triangleq \sigma \sqcup [x_k \mapsto \{((x_j, x_k), [d \mapsto t])\}] \tag{187}$$

If there had been a destination entry, but no information associated with (x_j, x_k), i.e., $\neg\chi[\![((x_j, x_k), \phi)]\!]\sigma(x_k)$, then the enter operation would have read

$$E[\![(x_j, x_k), [d \mapsto t]]\!]\sigma \triangleq \sigma \dagger [x_k \mapsto \sigma(x_k) \cup \{((x_j, x_k), [d \mapsto t])\}] \tag{188}$$

From our previous work we are led to combine the two specifications using a new operator the meaning of which should be obvious from context:

$$E[\![(x_j, x_k), [d \mapsto t]]\!]\sigma \triangleq \sigma \textcircled{\cup}_2 [x_k \mapsto \{((x_j, x_k), [d \mapsto t])\}] \tag{189}$$

This form of the enter operation deals conclusively with the two situations where the two outermost null bugs would have arisen. In the other innermost null bug case, the specification is

$$E[\![(x_j, x_k), [d \mapsto t]]\!]\sigma \triangleq \begin{cases} \sigma \dagger [x_k \mapsto \lhd [\![(\eta, \phi)]\!]\sigma(x_k) \cup \{(\eta, \phi \sqcup [d \mapsto \{t\}])\}] \\ \sigma \dagger [x_k \mapsto \lhd [\![(\eta, \phi)]\!]\sigma(x_k) \cup \{(\eta, \phi \dagger [d \mapsto \phi(d) \cup \{t\}])\}] \end{cases} \tag{190}$$

where the distinction is based on whether or not $\phi(d) = \theta$. We recognise our monoid of sets in this expression and can condense it to the form

$$E[\![(x_j, x_k), [d \mapsto t]]\!]\sigma \triangleq \sigma \dagger [x_k \mapsto \lhd [\![(\eta, \phi)]\!]\sigma(x_k) \cup \{(\eta, \phi \circledcirc [d \mapsto \{t\}])\}] \quad (191)$$

The process leading to the specification of the enter part of the store and forward operation seems much shorter and easier than the removal part. This is due to two very important reasons. By beginning with the removal part we were forced to address the pernicious null bugs. That we took this approach is almost entirely due to our familiarity with certain monoids. The second reason is simply that remove and enter are the inverses of each other. Once the removal was complete, then we presented the specification of enter in the reverse order. There is, of course, a third reason—the use of the auxillary variables η and ϕ which guided us in the specification of the removal operation. We found it very convenient to rely on these for the last part of the enter operation specification.

What can we say about the specifications that have beem produced? The first question that must be asked is: Are they correct? We must confess that we can not be too sure. This text is the third time that the specifications have been developed/elaborated. They must still be regarded as work in progress. The usefulness of the η and ϕ variables suggest splitting up the original models. It appears that we ought to introduce some auxillary models in order that η and ϕ achieve a permanent status. We did hint at this in the beginning of the section when we referred to headers, η, and collections of data packets, ϕ.

Another way in which we might check the correctness of the specifications, is to introduce an invariant that captures the amount of information in the storage in the network at any instant, and then prove that appropriate forward, store operations conserves that invariant.

5.2 Routing

To complete this section, we will now address the distinction between the 'static' network topology and the dynamic network routing, in a completely general setting. The quotes around static are intended to emphasise that at any instant, the topology of the network is fixed, in contrast to the dynamic nature of routing extraction from that fixed topology. However, as nodes may be taken out of service, and later switched back in, or as communication links may fail, etc., then the topology itself will no longer be static but vary over time. It will be clear from the specification of the routing, that the same mathematics may be used to analyse and specify the fragility of the network topology itself, in the presence of failure.

For definiteness, we may work initially with a topology such as that shown in Fig. 2. In the *VDM* we may demonstrate that such a topology is a reification of the star configuration, together with extra connectivity between lab nodes. But that takes us

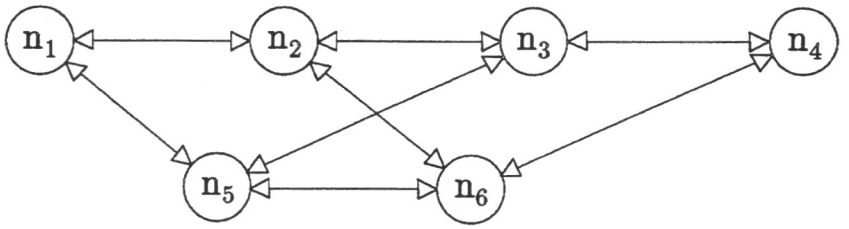


Figure 2: Network topology

into the verification part of the method. We are concerned with validation in this tutorial.

Now let us suppose that the node n_1 wishes to send data to node n_4. Then, from the diagram it is clear that we obtain the possible routings given in Fig. 3. Formally, we will extract the routings from the network topology by giving a recursive

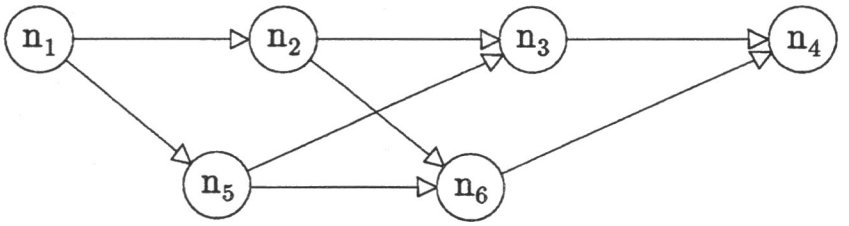


Figure 3: Network routing

algorithm, denoted by $\mathcal{E}$. Since both network topology and routing are essentially graphs, then the algorithm is based on graph traversal. We have modelled such graphs and graph traversals in the style of the *VDM*$^{\clubsuit}$ before and presented preliminary results in [11], where we noted our uncertainty as to the equivalence of two depth-first search algorithms, drawn from the literature.

The routing extraction problem has given us a fresh opportunity to look at the problem again *ab initio* and to obtain, what we believe to be, an elegant result. Relying on the development presented in [11], we propose to be very brief in our annotation of the following extraction algorithm. First we give the algorithm for the simple case that all the links are bidirectional.

18 $\mathcal{E}: X^2 \longrightarrow \text{TOPOLOGY} \longrightarrow (\text{TOPOLOGY} \times \text{ROUTING} \times \text{BACK_EDGES})$

.1 $\mathcal{E}[\![s,t]\!]\gamma \triangleq \mathcal{E}_t^s[\![\gamma(s)]\!](\gamma, [s \mapsto \gamma(s)], [s \mapsto \gamma(s)]^{-1})$

Annotations

18 For convenience we give names for the models rather than their structure. The TOPOLOGY and ROUTING models we have seen before. The BACK_EDGES model is exactly the same structure as the other two.

18.1 We hand over the computation to another extraction algorithm of the same name. The form of the right hand side is standard in the *VDM*♣. Of particular note is the inclusion of the topology γ as a read-only (data) structure. As well as the result to be computed, the routing, we will also compute its complement, the back edge structure.

The transformation of the source and target nodes, s, and t, from parameters to a superscript, and subscript, respectively, adds to the clarity of the functionality of the algorithm. Again this is standard.

19 $\mathcal{E}: X^2 \times \mathcal{P}\,X \longrightarrow$
$\quad$ ((TOPOLOGY × ROUTING × BACK_EDGES) $\longrightarrow$
$\quad\quad$ (TOPOLOGY × ROUTING × BACK_EDGES))

.1 $\mathcal{E}_t^s[\![\emptyset]\!](\gamma, \gamma_r, \gamma_b) \triangleq (\gamma, \gamma_r, \gamma_b)$

.2 $\mathcal{E}_t^s[\![\{n\} \uplus S]\!](\gamma, \gamma_r, \gamma_b) \triangleq$

.3 $\quad \chi[\![n]\!]\gamma_r \vee (n = t)$ — visited(n) or target(n)?

.4 $\quad\quad \rightarrow \mathcal{E}_t^s[\![S]\!](\gamma, \gamma_r, \gamma_b)$

.5 $\quad\quad \rightarrow \mathcal{E}_t^s[\![S]\!]\mathcal{E}_t^n[\![\lhd\,[\![\,dom\,\gamma_r]\!]\gamma(n)]\!](\gamma,$

.6 $\quad\quad\quad \gamma_r \sqcup [n \mapsto \lhd\,[\![\,dom\,\gamma_r]\!]\gamma(n)], \gamma_b \circledcirc [n \mapsto \lhd\,[\![\,dom\,\gamma_r]\!]\gamma(n)]^{-1})$

Annotations

19 The algorithm has the signature of a tail-recursive function. But from **19.5** we see that the algorithm is, in fact, binary recursive. Thus, this algorithm belongs to the more general class of curried functions.

19.1 $\mathcal{E}_t^s[\![\emptyset]\!]$ is the identity function.

19.3 If the new node n has already been visited, then we can ascertain this fact from the current routing. In addition, we check whether or not it is the target node ...

19.4 ... and, if either case is true, then no further depth searching is possible.

19.5 This is the heart of the algorithm. $\mathcal{E}_t^n$ denotes the depth-first search component. Space does not permit an extended treatment on the rationale for the result. The interested reader is urged to apply the algorithm to a concrete simple example and discern the reason for the choice of expressions. This is how the author developed them in the first place!

Of particular note is the fact that the algorithm computes all possible routings. It does not concern itself with optimal routings. For that additional structure and information is necessary.

Now let us turn to the general algorithm which deals with the extraction of routings from a network topology in which some of the links are unidirectional. Such may be the case, if there is a unidirectional ring embedded in a bidirectional network. The algorithm follows:

20 $\mathcal{E}: X^2 \longrightarrow$ TOPOLOGY $\longrightarrow$ (TOPOLOGY $\times$ ROUTING $\times$ BACK_EDGES)

.1 $\mathcal{E}[\![s,t]\!]\gamma \triangleq \mathcal{E}_t^s[\![\gamma(s)]\!](\gamma, [s \mapsto \gamma(s)], [s \mapsto \gamma(s)]^{-1} \textcircled{\cap}' \lhd [\![\gamma(s)]\!]\gamma)$

Annotations

20.1 The noteworthy difference from the previous algorithm is in the computation of the back edges. This is only to be expected since some of our links are unidirectional. Note the occurrence of the primed intersection operator that we have already introduced earlier—a nice surprise! In addition, a restriction operator is essential for correctness. Once more, the only advice we can offer for the present, is the exercising of the algorithm on a simple example (see below).

21 $\mathcal{E}: X^2 \times \mathcal{P}X \longrightarrow$
((TOPOLOGY $\times$ ROUTING $\times$ BACK_EDGES) $\longrightarrow$
(TOPOLOGY $\times$ ROUTING $\times$ BACK_EDGES))

.1 $\mathcal{E}_t^s[\![\emptyset]\!](\gamma, \gamma_r, \gamma_b) \triangleq (\gamma, \gamma_r, \gamma_b)$

.2 $\mathcal{E}_t^s[\![\{n\} \uplus S]\!](\gamma, \gamma_r, \gamma_b) \triangleq$

.3 $\quad \chi[\![n]\!]\gamma_r \vee (n = t) \vee \lhd [\![dom\,\gamma_r]\!]\gamma(n) = \emptyset$ — visited(n) or target(n)

.4 or *cul de sac*?

.5 $\quad \rightarrow \mathcal{E}_t^s[\![S]\!](\gamma, \gamma_r, \gamma_b)$

.6 $\quad \rightarrow \mathcal{E}_t^s[\![S]\!]\mathcal{E}_t^n[\![\lhd [\![dom\,\gamma_r]\!]\gamma(n)]\!](\gamma,$

.7 $\quad\quad \gamma_r \sqcup [n \mapsto \lhd [\![dom\,\gamma_r]\!]\gamma(n)],$

.8 $\quad\quad \gamma_b \textcircled{\cup} ([n \mapsto \lhd [\![dom\,\gamma_r]\!]\gamma(n)]^{-1} \textcircled{\cap}' \lhd [\![\lhd [\![\gamma_r]\!]\gamma(n)]\!]\gamma))$

Annotations

21.3 Of particular interest is the added test. Because of the unidirectional links, we can find ouselves in a *cul de sac*, a dead end. This was one of the most unexpected results discovered in the course of developing the algorithm.

To illustrate this case, we propose the example of a triangular unidirectional ring network (n_1, n_2, n_3), with bidirectional links ($n_1 \leftrightarrow n_4$, $n_2 \leftrightarrow n_5$, $n_3 \leftrightarrow n_6$) connecting each of the vertices to 'terminal nodes', (n_4, n_5, n_6). Computation of the routings for n_4 to n_6 produces a dead end example.

21.8 It took a full eight hours to arrive at this back edge computation and to obtain the given expression. Upon looking at it again, we must confess that we are astonished that we discovered it at all, and most important of all, we wonder whether there is not a simpler expression for the same result.

Fortunately, the corresponding encoded prototype may be exercised to determine, if indeed, the expression gives the desired result. In addition, the expression is given in terms of known operators and, therefore, amenable to subsequent theoretical analysis.

The presentation of the two extraction algorithms concludes the main body of the technical content of the tutorial. We feel that sufficient material has been presented to illustrate most of the issues involved in the validation of requirements and specifications via the use of formal methods. It is now time to conclude.

6 EPILOGUE

The text is dead, long live the dialogue!

Mathematics and Formal methods are something that one does, and when there are results, whether simple or great, one shares. How can one possibly convey the pain of labour, the excitement of discovery? This tutorial text has been written in a typical development style. A broad brush was used, where we boldly sketched out some of the models that arise in a typical application. We had intended to finish with a flourish by showing how the various domains could be brought together in a multimedia services application, incorporating the views of the customer, the service provider, and the network operator. Unfortunately, this was not to be. The tutorial participants will be the beneficiaries of that flourish, but not the readers, alas!

In the domain of testing, the most obvious technical contributions were presented as models and their invariants, operations and their pre-conditions, and the use of retrieve functions. Several classes of common pernicious bugs were carefully chosen and illustrated. But were we to draw the reader's attention to the most important aspect of the use of the mathematics for testing, we would have to mention the significance of the Feynman test.

All of our programming and system building, and all of our testing, does rest on a very sound theoretical basis. The mathematics is not (yet) very difficult. Only the notation looks strange. In our opinion, we can find no reason why software engineers ought not to be as well-versed in applied constructive mathematics, such as that of the *VDM*♣, as are 'ordinary' engineers in classical mathematics. One may not have to use the mathematics in daily work, but having the right mathematical engineering culture that is appropriate to computing, will surely imbue the software engineer with the appropriate spirit. When can we expect an analogous Feynman test for computing?

6.1 ACKNOWLEDGEMENTS

First and foremost, we express our gratitude to **K&M Technologies**, Limited (Ireland), of which the author is a Technical Director, for the considerable support in both time and other resources, without which this tutorial could not have been prepared.

We also owe a debt of gratitude to the other members of the School of the *VDM*♣, Dr. A. Butterfield, A. Donnelly, G. O'Reagan and E. McDonnell, who endured the development and presentations of many of the theoretical results, among which that of § 4 is to be singled out. Although the theory on inverse images had been developed in February 1991, it is clear from the published proof of satisfaction of the annihilator invariant by an enter operation [11], that the inverse image theory had not yet become an integral part of our methods.

For the specific problem on eliminating self-loops in a network presented in § 4, and the very idea of the calculation of the pre-condition, I am indebted to Dipl.-Ing. E. H. Dürr, who asked the author to review a paper which he is preparing, together with Prof. S. J. Goldsack, on the modelling of distributed systems based on work for the ESPRIT AFRODITE project.

Finally, we most sincerely thank the conference committee for accepting this tutorial for the 6th International Software Quality Week, San Francisco, 1993. They have shown extraordinary confidence in the 'new' technology and have permitted the author to publish many new results. We trust that their confidence was not misplaced, and that the participants of the tutorial will find new insights into the problem of debugging requirements and specifications via mathematical modelling.

REFERENCES

[1] Dines Bjørner and Cliff B. Jones, editors. *The Vienna Development Method: The Meta-Language, Lecture Notes in Computer Science* **61**. Springer-Verlag, Berlin, 1978.

[2] Dines Bjørner and Cliff B. Jones, editors. *Formal Specification and Software Development.* Prentice/Hall International, Englewood Cliffs, New Jersey, 1982.

[3] Boris Beizer. *Software Testing Techniques.* Van Nostrand Reinhold, New York, second edition, 1990.

[4] Antoni Diller. *Z: An Introduction to Formal Methods.* John Wiley & Sons, Chichester, England, 1990.

[5] Phillippe Garnerin et al. A Visualization Tool for Representing Epidemic Spread. *The Mathematica Journal*, 1(4):59–61, 1991.

[6] James Gleick. *Genius, Richard Feynman and Modern Physics.* Little, Brown and Company, London, 1992.

[7] IEEE Computer Society. IEEE Guide to Software Requirements Specifications, ANSI/IEEE Std 830–1984. In Merlin Dorfman and Richard H. Thayer, editors, *Standards, Guidelines, and Examples on System and Software Requirements Engineering*, pages 16–38. IEEE Computer Society Press, Los Alamitos, California, 1990.

[8] Nathan Jacobson. *Basic Algebra I.* W. H. Freeman, New York, second edition, 1985.

[9] Morris Kline. *Mathematical Thought from Ancient to Modern Times.* Oxford University Press, Oxford, 1972. The 1990 three volume paperback work is cited.

[10] Mícheál Mac an Airchinnigh. *Ph.D. Thesis: Conceptual Models and Computing.* Department of Computer Science, Trinity College, Dublin, Ireland, 1990.

[11] Míchéal Mac an Airchinnigh. Tutorial Lecture Notes on the Irish School of the VDM. In S. Prehn and W. J. Toetenel, editors, *VDM'91, Formal Software Development Methods Volume 2: Tutorials, Lecture Notes in Computer Science* **552**, pages 141–237. Springer-Verlag, Berlin, 1991.

[12] National Bureau of Standards (U.S.). Guideline for Lifecycle Validation, Verification, and Testing of Computer Software, FIPS PUB 101. In Merlin Dorfman and Richard H. Thayer, editors, *Standards, Guidelines, and Examples on System and Software Requirements Engineering*, pages 63–100. IEEE Computer Society Press, Los Alamitos, California, 1990.

[13] Walter Rudin. *Real and Complex Analysis.* McGraw-Hill, 1970.

[14] Gerard Tel. *Topics in Distributed Algorithms.* Cambridge University Press, Cambridge, 1991.

[15] Margaret Tierney. Software Engineering Standards: the 'Formal Methods Debate' in the UK. *Technology Analysis & Strategic Management*, 4(3):245–78, 1992.

[16] W. J. Toetenel. *Ph.D. Thesis: Model Oriented Specification of Communicating Agents.* Department of Computer Science, Technical University of Delft, The Netherlands, 1992.

[17] Jim Woodcock and Martin Loomes. *Software Engineering Mathematics, Formal Methods Demystified.* Pitman Publishing, London, 1988.

"Maintaining the Software Process Movement"

TUTORIAL "G"

Dr. Bill Curtis
Software Engineering Institute

Dr. Bill Curtis is the Director of the Software Process Program in the Software Engineering Institute at Carnegie Mellon University. His group is building methods and technology for helping organizations mature their process for building software systems. Previously, he was a Director in MCC's Software Technology Program, where he directed research on coordination and collaboration technology and on empirical studies of coordinating team design in large systems. Prior to joining MCC, Dr. Curtis held positions at ITT's Programming Technology Center, in GE Space Division, and in the Organizational Research Group at the University of Washington.

The CASE for Process

Bill Curtis
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213-3890
USA

I. The Disappointing Results from Programming Technology

The past decade witnessed a torrent of promises about dramatic productivity and quality growth in software development that awaited those who purchased and deployed any of the newly-developed Computer-Aided Software Engineering (CASE) tools. Long-suffering software organizations were desperate to solve their software crisis, a condition that appeared more chronic than acute. However, software development managers now find themselves searching for the slightest shred of evidence that can be used to justify the return on investment they promised when purchasing their CASE tools. The order-of-magnitude promise that was an obligatory justification to get a purchase request authorized, too often turns into an insignificant trend in the right direction when the data (if they are ever collected) are inspected.

Data reported in the early 1980s were not promising in their implications for the impact of CASE tools later in the decade. On the front cover of Barry Boehm's (1981) book, *Software Engineering Economics*, he describes the impact of numerous factors on software development productivity. The impact of software tools and practices were infinitesimal compared to the striking impact of the capability of the personnel assigned to the project. In developing correction factors to apply to the productivity estimate used in driving his COCOMO cost estimating model, Boehm's data drove him to assign a multiplier of only 1.49 to software tools (where 1.0 represents no impact), compared to a factor of 4.18 for personnel capability (the difference between a 15th percentile team and an 90th percentile team). In fact, by the time he finished evaluating all of the personnel-related factors, Boehm reported that "a project staffed with uniformly very low rated personnel on all capability and experience factors would require 10.53 times as much

effort to complete the project as would a project team with the highest rating in all the above factors" (1981, p. 431). — Hold it, this was the impact the ads in Computerworld and Datamation told us we would get from CASE tools, not people!

Vallett and McGarry (1989) reported similarly distressing findings in analyzing the data collected over a decade in NASA's Software Engineering Laboratory (NASA-SEL). The most salient factor in their data was the 20+ to 1 differences among individual programmers in productivity (cf. Curtis, 1981). Similarly, they found that the best predictor of defect prone modules, a major problem in software quality, was who programmed it. In trying to sort out the impact of technology on the data produced in the NASA-SEL, Card, McGarry, and Page (1987) reported no impact for tool use on either software productivity or quality. They did, however, find a significant impact on software quality for several process factors such as the percentage of possible product and process reviews that were held, the percentage of product and process documents that were actually produced, and the percentage of the code that was actually reviewed before testing. Quick to learn from industry experience such as NASA-SEL's, consultants realized that there was money in helping managers outline a process to support their newly purchased CASE tools.

Between 1980 and 1983 I worked in ITT's Programming Technology Center and built a worldwide software productivity and quality measurement system. In studying the factors that impacted software productivity throughout ITT, we discovered the curious effect represented in Figure 1 (Vosburgh, et al., 1984) . If projects did not use software tools or modern programming practices, their productivity was uniformly low. However, using software tools and practices did not guarantee high productivity. Although some projects using tools displayed exemplary results, many others were just as abysmally unproductive as those that used no tools at all. These projects were usually out of control, and software tools offered little assistance in correcting the problems they experienced. As I inspected other project data throughout the software industry during the 1980s, I found that this result was far from unique to ITT.

II. The Emphasis on Process

What do software development data such as those reported above tell us about the new world order promised from the use of CASE tools? After two decades of unfulfilled promises about productivity and quality

gains from applying new software methods and technologies, software development organizations are realizing that their fundamental problem is the inability to manage the software process. The lack of tools is an excuse to cover for the lack of management and engineering discipline. The benefits of better methods and tools cannot be realized in the maelstrom of an undisciplined, chaotic project where documents are not produced and reviews are not held. When projects are excessively late and double the planned budget, the organization is usually guilty of not providing the infrastructure and support necessary to help projects avoid these problems. CASE tools are not a remedy for this failure.

The software engineering community is learning, as has virtually every other area of engineering, that advances in productivity and quality do not materialize just because technology was thrown at a problem. Rather, improving the results of software development or maintenance operations requires attention to the processes by which these operations are performed. A software process movement emerged in the mid-1980s when shortcomings in managing the development process were recognized as prime inhibitors of growth in software productivity and quality.

Too often developers have purchased CASE tools or other technologies, and then tried to employ a process that fitted their newly acquired tools. Unfortunately, existing integrated programming support environments (IPSEs) only support a portion of the tasks included in a sophisticated software development or maintenance process. In the absence of a defined software process, it is difficult to identify:

- the complete suite of tools needed to support the entire process,
- how these tools should be integrated to support actual work, and
- how to design a software development environment that coordinates the work of many software engineers.

Putting the emphasis on technology first might be justified if radical gains in productivity and quality were being achieved. However, since the payback from inserting technology has been modest, greater leverage for improving project results appears to reside in better management of the software process. When the noise created by poorly defined or badly managed processes is removed from a project, the impact of technology is more easily observed in project outcomes (Humphrey, et al., 1991). Software development organizations are finding that they should first focus on defining the processes used in their software business, and only then carefully select tools and methods to support these processes.

III. A Process Maturity Framework

If the maturity of an organization's software development process influences the success with which it can introduce and use advanced software technologies such as CASE, then it is wise to understand what distinguishes mature from immature organizations. In an immature software organization, software processes are generally improvised by practitioners and their management during the course of the project. Even when a software process such as peer reviews or quality assurance has been specified, it is not followed or enforced. One of the most damaging characteristics of immature organizations is that unrealistic commitments are made by managers without consulting with the development staff on whether these commitments can be met. Schedules and budgets are then routinely exceeded, because they are not based on realistic estimates that a sound engineering process can support. When hard deadlines are imposed, product functionality is often compromised, and engineering disciplines such as reviews and testing are often curtailed or eliminated in order to meet the schedule.

The immature software organization is reactionary and managers are usually focused on fighting fires that a more mature process might have prevented. When success is obtained, and an occasional striking success can emerge from an immature organization, it is achieved through the heroic efforts of talented individuals. Repeating this success is solely dependent on getting the same individuals appointed to the next project. Thus, in immature organizations it is individuals rather than the organization that carries the capability for success. The organization has not created a mechanism for learning from previous project experience, and does not have the ability exercise effective project control. In such environments it is not surprising that we find individual talent being the best predictor of productivity and quality.

On the other hand, a mature software organization possesses an institutionalized capability for managing software development and maintenance processes. A defined, organization-wide software process is accurately communicated to both existing staff and new employees, and work activities are carried out effectively. The defined processes are trained and are consistent with the way the work actually gets done. These defined processes are updated when necessary, and improvements are developed through controlled pilot-tests and/or cost benefit analyses. Roles and responsibilities within the defined process are clear throughout the project and to others in the organization.

In a mature organization, managers monitor the quality of the software product quantitatively and ensure the customer is satisfied. There is an objective, usually quantitative, basis for judging quality and analyzing problems with both the product and process. Schedules and budgets are based on historical performance and are realistic; the predicted results for product cost, schedule, functionality, and quality are usually achieved. Based on the predictability of project results, reliable management decisions can be made about tradeoffs among these outcomes. When a software product is unprecedented in a mature organization, project management and staff know how to alter their standard software process in an orderly way to establish a basis for effective management planning, control, and decision-making.

Capitalizing on these observations about immature and mature software organizations requires construction of a process maturity framework. This framework must describe an evolutionary path from ad hoc, chaotic processes to a mature, disciplined software process. This framework is important because improvement goals must be set in stages that represent a sustainable growth path from an undisciplined, immature state to a mature, institutionalized software process. Attempts to improve project results through buying CASE technology that are not guided by a process improvement framework may prove ineffective because the necessary foundation for supporting successive advances has not been established. As will be become apparent in the next section, the initial problems to be attacked are not those solved by existing CASE technology.

IV. A Model of Process Maturity

One model of process maturity that has received extensive recent attention, and has been adopted by a large segment of the software development community, is the Capability Maturity Model (CMM) for software (Humphrey, 1989; Paulk, et al. 1991). This model provides software organizations with guidance on how to gain control of their process for developing and maintaining software and how to evolve their environment toward a culture of software engineering excellence. The CMM was designed to guide software organizations in selecting process improvement strategies based on the current maturity of their organization's software process. The CMM's benefit is in narrowing the scope of improvement activities to those software processes that provide the next foundational layer for the organization's growth. By focusing on a limited set of activities and working aggressively to achieve them,

organizations can steadily improve their software process to enable continuous and lasting gains in software project performance.

The staged structure of the CMM is based on product quality principles that have been evolved by W. Edwards Deming, Joseph Juran, and others over the last sixty years. The maturity framework into which the principles underlying the CMM have been adapted was first elaborated by Philip Crosby (1979) in his book *Quality is Free.* Crosby's quality management maturity grid describes five evolutionary stages in adopting quality practices. This framework was adapted to the software process by Ron Radice and his colleagues (Radice, et al., 1985) working under the direction of Watts Humphrey at IBM. Humphrey brought this framework to the Software Engineering Institute in 1986 and developed the foundation for its current use throughout the software industry.

The CMM provides a framework for identifying five levels of maturity that lay successive foundations for continuous process improvement. These five maturity levels define an ordinal scale for measuring the maturity of an organization's software development capability. Each maturity level is a well-defined evolutionary plateau on the path toward becoming an exceptional software organization. Each level comprises a set of process goals that, when satisfied, make a significant addition to the sophistication and capability of an organization's software development process. The five levels of the CMM and the key process areas constituting them are shown in Figure 2.

The Initial maturity level (level 1) is aptly characterized by the description of an immature software development organization provided in the previous section. Maturity levels 2 through 5 can be characterized through the activities performed by the organization to establish or improve their software process.

Level 2: the Repeatable level - The primary objective in achieving level 2 is to stabilize project commitment and management control processes. Project management policies and procedures must established by the organization, and an audit function independent of project management (i.e., a quality assurance function) must determine that they are being practiced. Planning and management of new projects is based on experience with similar projects. Project managers track software costs, schedules, and functionality. Problems in meeting commitments are identified when they arise. Software requirements and the artifacts developed to satisfy them are baselined, and their integrity is controlled. Project teams work with their customers, and their

subcontractors if any, to establish a stable, managed working environment. Realistic project commitments are established from results observed on previous projects that allow project staff to repeat successful practices developed on earlier projects.

Level 3: the Defined level - At the defined level an organization-wide standard software process is defined and documented, including both software engineering and software project management processes. Projects use the organization's software process as a basis for designing a process tailored to each project's unique characteristics. A defined process is enabled because the management discipline established at level 2 ensures that sound engineering principles will not be abandoned even under stress. Having a defined process allows management and engineering activities to be coherently integrated on each project. A software engineering process group is established to facilitate process definition and improvement activities. Each project institutes a peer review process to enhance product quality. An organization-wide training program is implemented to ensure that all practitioners and managers have the knowledge and skills required to carry out their tasks using the tools and methods they have selected. The capability established at level 3 is based on a common understanding of processes, roles, and responsibilities in a defined process that allows all project personnel to coordinate their activities more effectively and work from a common understanding of their mutual dependencies.

Level 4: the Managed level - At the managed level the organization sets quantitative quality goals for software products. In addition, productivity and quality are measured for important software process activities across all projects in the organization. These fine-grained measures would be impossible to effectively interpret without establishing the carefully defined process built at level 3. These measures establish the quantitative foundation for evaluating project processes (e.g., defect removal efficiency of reviews at different life cycle stages) and products (e.g., defects per thousand delivered source instructions during the first year of operation). Projects achieve control over their products and processes by narrowing the variation in their performance to within acceptable quantitative boundaries. When building software in unfamiliar application domains, the risks involved in moving up the learning curve during development are known and carefully managed. The organization is able to predict process and product quality trends within established quantitative bounds. When these bounds are violated, management action is taken to correct the situation.

Level 5: the Optimizing level - At the optimizing level the organization focuses on continuous process improvement. Based on the quantitative capabilities established at Level 4, it can identify weak process elements and strengthen them, with the goal of enhancing quality. Empirical evidence is available on process effectiveness and is used to perform cost/benefit analyses on new technologies. Technologies with the greatest payoff for the specific processes used by the organization (as per their defined process) are selected for a managed introduction into their process. Although the software process is mature, it is a target for continuous enhancement. Project teams analyze process performance to determine the causes of defects. Software processes are evaluated to prevent known types of defects from recurring, and the lessons learned are disseminated to other projects. An environment of software engineering excellence has been achieved.

Maturity levels 4 and 5 are unknown territory for the software development industry in 1992 (Humphrey, Kitson, & Kasse, 1989). There are a few examples of projects with level 4 and 5 characteristics, but no level 4 and 5 software organizations have been observed. Unfortunately, these projects are too few to draw general conclusions about other characteristics of level 4 and 5 organizations. Characteristics of level 2 and 3 organizations have been established through extensive assessment of existing software development organizations. The characteristics of organizations at levels 4 and 5, however, have been defined by analogy with other industries and by anticipating the next most likely steps to be taken by the most advanced level 3 organizations. Additional characteristics may be added to describe level 4 and 5 organizations as we study the few level 3 organizations currently on the verge of becoming level 4.

V. Implications of the CMM for Introducing CASE

The maturity of a software organization is a predictor of the benefit that will be observed from installing CASE and other software engineering technologies. The data reported during the 1980s mostly reflected organizations in their earliest stages of maturity. It is not surprising that there was little impact observed for deploying software tools. The fundamental problem in such organizations in the lack of management or process discipline. Thus, a software development project can look chaotic because there is little formal organization to the means by which it will

be executed. CASE tools will not add organization to a chaotic process, and their benefits will be difficult to observe above the noise.

Aficionados of CASE should not loose heart at these results. The promise of CASE and other technologies should be realized in the data at higher levels of process maturity. Once the process has come under management control, it is possible to begin defining the tools that will benefit the defined engineering process. However, this approach is a departure from the way many people deploy CASE tools. First they acquire the tools, and then they determine the process that fits the tools purchased. Making the process subservient to the tools would be appropriate if the tools provided complete coverage of the software development process. However, neither CASE tools nor IPSEs present a complete solution, and large gaps in the process are left uncovered by any technology. Therefore, the more effective path is to define the process first, and then select the tools that will aid it. In this way managers can more effectively determine the real likelihood that a tool will provide an observable cost benefit.

Unfortunately, the habit of too many managers is to invest in something once. If they see no immediate benefit, they swear off it for life. Such attitudes fail to recognize that organizations mature in their business processes, and that different interventions require a level of maturity to be achieved before they can be employed effectively. A two year old benefits little from instruction in highjumping.

The stages of the CMM provide a guide to acquiring CASE technology based on the specific problems to be solved in building a more effective process. That is, it provides a guide to staging the introduction of CASE tools. The model does not identify specific tools, but it does suggest which processes should be considered candidates for automated assistance at different stages. The process focus is on technology transition, rather than on specific technologies.

As organizations begin the trek toward level 2, where they gain management control over their project and commitments, some tools can aid in gaining management control. In particular, tools that provide management assistance such as project planning and tracking tools and cost estimating tools can help managers organize their activities, track their commitments and progress, and communicate plans and status more effectively to project staff and senior management. These tools are not those traditionally thought of as CASE tools. Rather they are considered project management tools. Similarly, configuration management tools are

critical in establishing manageable control over product baselines and changes. These tools are not the glamor tools of the CASE world. But then establishing a sound management infrastructure was never the glamorous aspect of running a business.

As organizations work toward level 3, they begin to establish a clear picture of their engineering process. Tools for modeling and representing process will prove invaluable for analyzing and communicating the process decisions made. Some existing CASE tools may provide support in this modeling activity,since processes are often represented with the same languages used for modeling in information systems development (Curtis, et al., 1992). Once a clearly defined engineering process has been defined, some software engineering tools suggest themselves immediately. They may be different from the tools that would have originally been purchased, but such is often the case when one reflects before one buys. Also, the organized training program that is required at level 3 will aid the introduction of CASE tools. Tools for which adequate prior training is not provided are tools that remain unused under the stress of a project schedule. Well planned schedules permit many things, but extensive free periods for learning and exploring a new tool set are not among them.

The defined process provides a roadmap for integrating tools to aid a more complete development process than is usually thought about when CASE tools are first purchased. Many of the most time consuming activities in software development such as reviews and test case development are forgotten when viewing a demonstration of a CASE tool that nicely documents design decisions. Without elaborating these other activities (i.e., those that support the production of programs rather than those that actually produce programs) many of the largest productivity and quality drivers are forgotten (Curtis, et al., 1988). These activities often offer the greatest benefits when automated.

At level 4 the emphasis shifts toward collecting detailed data on the process and product. Thus, the tools that would provide the greatest assistance at this level must have the ability to capture both process and product data. Unfortunately we are more speculative about implications for tools at levels 4 and 5; because we have been unable to study any organizations at this level (although a few are on the verge of reaching level 4). At level 2 an organization needs to collect the large-grained project outcome data that allow them to estimate project costs, schedules, and resources as part of the management planning activity. However, at level 4 the data become sufficiently fine-grained that the

efficiency of each step of the process can be analyzed (e.g., the defect removal efficiency of different reviews). The cost of such data would be prohibitive unless the collection of it is automated. Thus, at higher levels of maturity the CASE tools and IPSEs must support automatic data collection concerning process steps and the artifacts produced.

At level 5 the focus becomes continuous process improvement and defect prevention. The data on the efficiency of each step have stabilized in the statistical quality control sense, and the next objective is to drive the trends in the desired direction (lower costs and defects, higher quality and productivity). However, the detailed process data available at this level of maturity allow a thorough analysis of the potential benefits of a particular tool against the actual data for the process it is purported to improve. The cost-benefit analyses that managers are so fond of asking for become easy to develop at this level of maturity. Unfortunately, they were even more necessary at much lower levels of maturity, before the justifications for investment became so obvious. We find that by level 3, managers do not need return on investment arguments, since the returns on improving their development process have become obvious in the results already experienced. Rather, they become more demanding about functional analyses of which tools most effectively enhance the capabilities of the processes the organization has selected.

VI. Conclusions

The CMM provides one explanation of the disappointing performance of CASE tools in actual project data. But it is an explanation with hope. When CASE tools are tossed into a noisy, chaotic process, they have little hope of sorting out the problems that the organization is suffering. As the organization begins to improve its ability to manage the software development process, it also improves its ability to successfully deploy carefully selected tools — tools specifically selected to aid a defined process. The introduction of tools into software development must itself result from a defined transition process. Thus, at the initial levels of maturity the introduction of CASE tools would be as haphazard as any other process the organization is performing. The maturity required to successfully transition tools grows proportionately with the maturity of the software development process that they support.

The focus on process suggests that CASE tools and IPSEs are the wrong place to start in trying to upgrade the productivity and quality of software development. Rather, improve the management and execution of the software process to the point that its readily supports the

introduction of carefully selected tools. Until tools possess more silver bullets than we find in CASE tools on the market today (Brooks, 1987), we should lead with process to pave the way for the successful introduction of automation.

VII. References

Brooks, F.P. (1987). No silver bullet. *IEEE Computer*, 20 (4), 10-19.

Card, D.N., McGarry, F.E., & Page, G.T. (1987). Evaluating software engineering technologies. *IEEE Transactions on Software Engineering*, 13 (7), 845-851.

Crosby, P. (1979). *Quality Is Free.* New York: McGraw-Hill.

Curtis, B. (1981). Substantiating programmer variability. *Proceedings of the IEEE*, 69 (7), 846.

Curtis, B., Kellner, M., & Over, J. (1992). Software process modeling. *Communications of the ACM*, 35 (9), in press.

Curtis, B., Krasner, H., & Iscoe, N. (1988). A field study of the software design process for large systems. *Communications of the ACM*, 31 (11), 1268-1287.

Humphrey, W.S. (1989). *Managing the Software Process.* Reading, MA: Addison-Wesley, 1989.

Humphrey, W.S., Kitson, D.H., & Kasse, T.C. (1989). *The State of Software Engineering Practice: A Preliminary Report (Tech. Rept. CMU/SEI-89-TR-1).* Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University.

Humphrey, W.S., Snyder, T.R., & Willis, R.R. (1991). Software process improvement at Hughes Aircraft. *IEEE Software*, 8 (4), 11-23.

Paulk, M.C., Curtis, B., Chrissis, M.B., Averill, E.L., Bamberger, J., Kasse, T.C., Konrad, M., Perdue, J.R., Weber, C.V., & Withey, J.V. (1991). *The Capability Maturity Model for Software (Tech. Rep. CMU/SEI-91-TR-24).* Pittsburgh: Software Engineering Institute, Carnegie Mellon University.

Valett, J.D. & McGarry, F.E. (1989). A summary of software measurement experiences in the Software Engineering Laboratory. *Journal of Systems and Software*, 9 (2), 137-148.

Vosburgh, J., Curtis, B., Wolverton, R., Albert, B., Malec, H., Hoben, S., & Liu, Y. (1984). Productivity factors and programming environments. In *Proceedings of the 7th International Conference on Software Engineering*. Washington, DC: IEEE Computer Society, 143-152.

The Capability Maturity Model and Software Process Improvement

Dr. Bill Curtis

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213-3890
curtis@sei.cmu.edu
SEI Info: 412-268-5800

Promethean Technologies, Inc.
3644 Ranch Creek
Austin, Texas 78730-3701
512-346-2435
Fax: 512-346-4677

Outline

Why is software process the right focus?

SEI Capability Maturity Model (CMM) for Software

Software process improvement programs

Return on investment for software process improvement programs

Why is Software Process the Right Focus?

" If builders built like programmers program, a woodpecker could destroy civilization."

- Gerald Weinberg

The Cost of Software Quality

Telecommunications software in the news
- DSC forgets regression test—stock drops
- System 12 software doubles—ITT cut in half
- AT&T defect reporting system—the N.Y. Times

Good measures of software reliability
- market share
- executive tenure
- contract renewal
- return on equity
- company ownership

Crisis-Driven Development

Air Force red team study
- 17 projects (all major projects in one command)
- all projects late—average 75% over schedule

Most software organizations are fighting fires
- the fire is in control
- constantly reacting—no time to improve
- your only control—fire prevention

Sources of Crisis

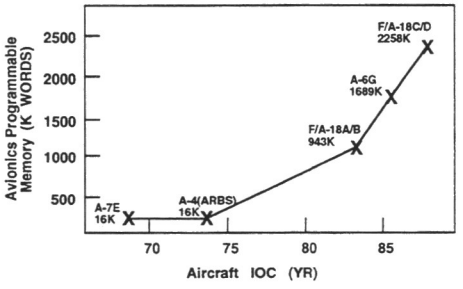
Inaccurate estimates
- explosive code growth
- unachievable schedules
- inadequate resources

Unmanaged work
- unplanned commitments
- untracked plans
- unruly subcontractors

Uncontrolled products
- fluctuating baselines
- volatile requirements

Software Growth in DoD Systems

Experience with Naval Weapon Center Aircraft



Size and Process Boundaries

Lines of Code		Development structure
1,000–5,000	➡	Individual programmer
5,000–25,000	➡	Small team
25,000–100,000	➡	Large subdivided team
100,000–1,000,000	➡	Several teams or divisions
1,000,000–10,000,000	➡	Several companies
10,000,000–100,000,000	➡	National undertaking

Scalability Problems

Order of magnitude growth in software size — every 5 years

Scaling requires fundamental process changes
- **can't run 60 mph by trying harder**
- **can't build skyscraper using carpenters**

Cannot scale up current techniques
- **"individual programmer" model is debilitating**
- **organizational limits on manageable complexity**
- **process models break at size boundaries**

Infant Mortality Among Start-Ups

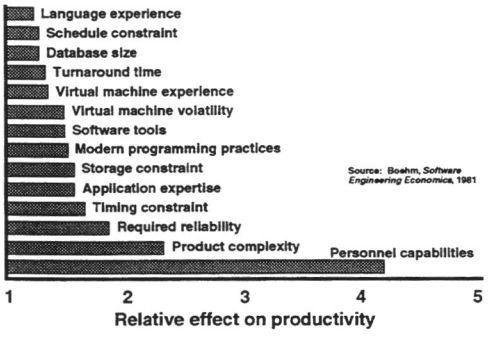
Statistically, most die

First product released in spite of dreadful process

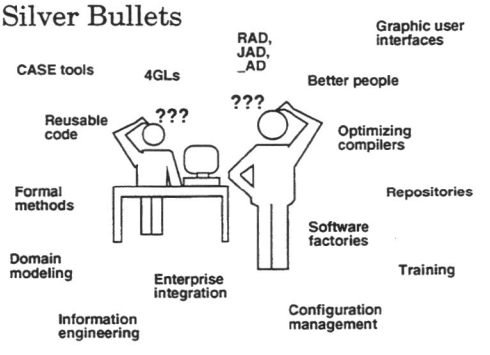
Software evolution becomes unprofitable
- **spaghetti code**
- **configuration tangles**
- **weak product planning**

Where does the second product come from?
- **everyone trapped in maintenance**
- **can't handle increased complexity**
- **past entrepreneurial stage—must mature**

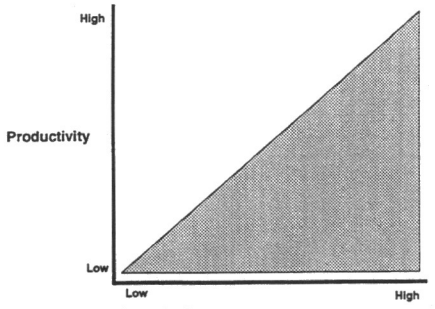
Productivity Factors at TRW



Silver Bullets



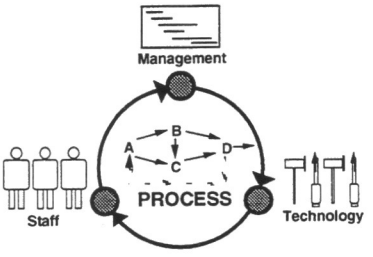
Productivity Variation for Practices



Source: Vosburgh, Curtis, et al., *International Conference on Software Engineering*, 1984

The Role of Process

Process: the set of activities, methods, practices, and transformations that integrate managers and software engineers in using technology to develop and maintain software.



The Focus on Process

Focusing on product alone misses
- scalability issues
- knowledge of how to do it better

Focusing on process predicts
- repeatability of outcomes
- project trends
- product characteristics

MIL-STD-2167A focuses on product artifacts, not process.

Process Versus Product

PRODUCT \ PROCESS	Bad	Good
Bad	typical level 1	unprecedented system
Good	heroic effort	mature software business

Immature Software Organizations

Processes are ad hoc and occasionally chaotic.

Processes improvised by practitioners.

Testing and reviews usually curtailed under stress.

Quality is unpredictable.

Costs and schedules are usually exceeded.

Reactionary management is usually firefighting.

Success rides on individual talent and heroic effort.

Technology benefits are lost in the noise.

Executive Complaints

Low visibility into progress and quality

Unpredictable performance

Constant surprises

Excessive maintenance costs

Mature Software Organizations

Processes are defined and documented.

Management plans, monitors, and communicates.

Roles and responsibilities are clear.

Product and process are measured.

Quality, costs, and schedules are predictable.

Time to market is reduced, useful life is increased.

Management committed to continuous improvement.

Technology used effectively within defined process.

Process Improvement Paradigm

Characterize current software practices

Establish measurable improvement goals

Set priorities for improvement activities

Assign dedicated resources

Implement an improvement plan

Manage organizational change

Measure progress

Periodically reassess baselined practices

SEI Capability Maturity Model (CMM) for Software

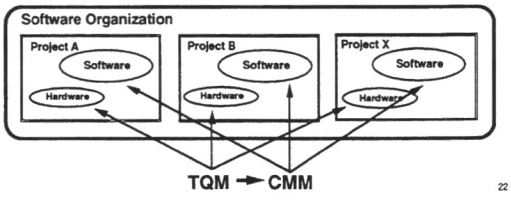
" The quality of a software system is governed by the quality of the process used to develop it"

- Watts Humphrey

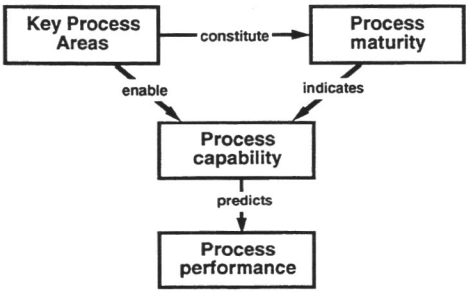
CMM Objectives

A conceptual framework based on state-of-the-art software engineering practices to help organizations

- characterize the maturity of their process
- establish goals for process improvement
- set priorities for immediate actions
- envision culture of software engineering excellence



Process Maturity Framework -1



Process Maturity Framework -2

Key Process Areas: a cluster of related activities that when performed collectively, achieve goals that enhance process capability.

Process Capability: the range of results expected from following a process; a predictor of future project outcomes.

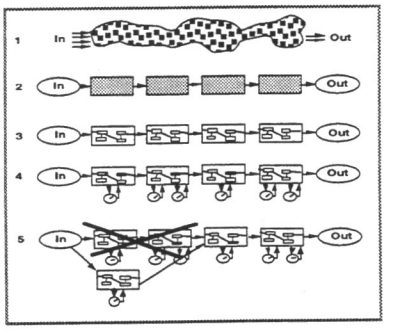
Process Performance: the actual results achieved from following a process.

Process Maturity: an organization's ability to consistently follow and improve its process.

CMM

Level	Focus	Key Process Areas
5 Optimizing	Continuous process improvement	Defect prevention Technology innovation Process change management
4 Managed	Product and process quality	Process measurement and analysis Quality management
3 Defined	Engineering process	Organization process focus Organization process defn. Peer reviews Training program Intergroup coordination Software product engineering Integrated software mgt.
2 Repeatable	Project management	Software project planning Software project tracking Software subcontract mgt. Software quality assurance Software configuration mgt. Requirements mgt.
1 Initial	Heros	

Management Visibility by Level



Genealogy of the CMM

"Characterizing the Software Process: A Maturity Framework"
- Watts Humphrey, IEEE Software, 1987

A Method for Assessing the Software Engineering Capability of Contractors
- Watts Humphrey et. al., CMU/SEI-87-TR-23, 1987

Assessment and Evaluation Training
- SEI 1988

Managing the Software Process
- Watts Humphrey, 1989

Key Practices of the Capability Maturity Model for Software
- Weber, et al.,CMU/SEI-91-TR-25, 1991

Level 1: The Initial Process

Unstable environment lacking sound management practices–commitments not controlled.

Practitioners argue against engineering discipline under the guise of "individual creativity."

Success rides on individual talent and heroic effort.

Standards and practices often sacrificed to management priorities–usually schedule.

Process capability is unpredictable–schedule, cost, and quality targets are rarely met.

Level 2 Key Process Areas

Initial level → **Software project planning**
Software project tracking and oversight
Software subcontract management
Software quality assurance
Software configuration management
Software requirements management
→ **Repeatable level**

Level 2: The Repeatable Process

Basic project management discipline is installed.

Reasonable commitments are planned and tracked.

Engineering baselines are enforced.

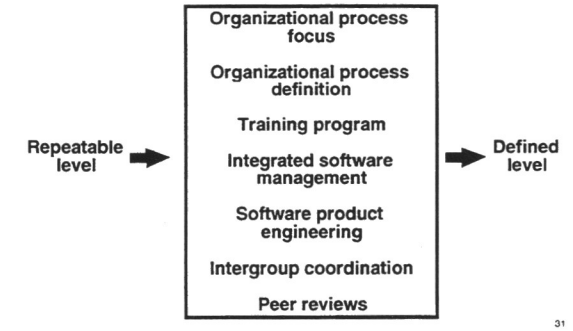
Management discipline ensures that sound practices are observed even during a crisis.

Previously successful processes are repeatable in a stable, managed environment.

Process capability exists for meeting schedules.

Carnegie Mellon University
Software Engineering Institute

Level 3 Key Process Areas



Carnegie Mellon University
Software Engineering Institute

Level 3: The Defined Process

Standard software processes are defined and applied throughout the organization.

Shared understanding of all roles and interfaces to other groups is established.

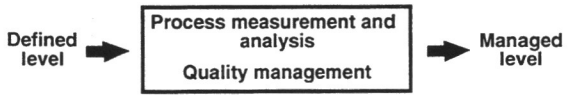
Software engineering process group (SEPG) created to focus on process improvement.

Organization-wide training needs are identified, provided for, and tracked.

Process capability to meet schedule, cost, and functionality targets exists.

Carnegie Mellon University
Software Engineering Institute

Level 4 Key Process Areas



Carnegie Mellon University
Software Engineering Institute

Level 4: The Managed Process

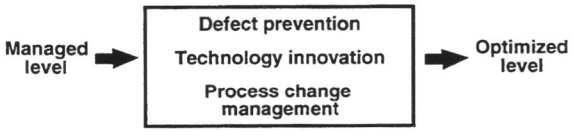
Statistical process control principles are used to address special causes of process variation.

Detailed process and product data are available–quality targets are set and tracked.

Process is capable of performing within narrowly defined quantitative limits–targets are predictable.

Carnegie Mellon University
Software Engineering Institute

Level 5 Key Process Areas



Carnegie Mellon University
Software Engineering Institute

Level 5: The Optimizing Process

Chronic causes of poor performance are identified and eliminated.

Software process is continually improved.

New technologies are prototyped, piloted, and if successful, introduced into the process.

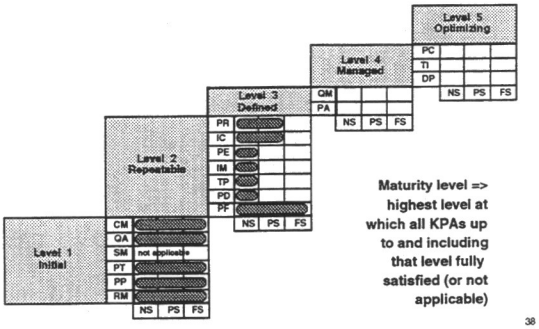
Process capability is continuously raised.

Key Process Area Satisfaction Summaries

A graph of the extent to which each Key Process Area at Level 2 (Repeatable) is satisfied

Key Process Area		Not Satisfied	Partially Satisfied	Fully Satisfied
[CM]	Software Configuration Management			
[QA]	Software Quality Assurance			
[SM]	Software Subcontract Management	not applicable		
[PT]	Software Project Tracking & Oversight			
[PP]	Software Project Planning			
[RM]	Requirements Management			

Maturity Profile



Maturity level => highest level at which all KPAs up to and including that level fully satisfied (or not applicable)

CMM Interpretations

Level 5 "Work the measures"

Level 4 "Measure the work"

Level 3 "Work the plan"

Level 2 "Plan the work"

Level 1 "Our heros have always been cowboys"

Maturity Levels Cannot be Skipped

Process improvements are implemented in stages, since some processes cannot be effective unless others are stable.

Each level provides foundation for improvements undertaken at the next level.

- **engineering processes are easily sacrificed without management discipline**
- **process measures uninterpretable without defined processes**
- **process innovation benefits lost in a noisy process**

Structure of a Maturity Level -1

Structure of a Maturity Level -2

Key Process Areas: principal contributing factors to process capability at each maturity level.

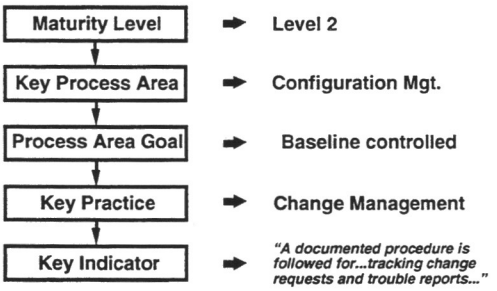
Key Process Area Goals: process objectives that, when achieved, enhance process capability.

Key Practices: policies and activities that collectively implement a key process area.

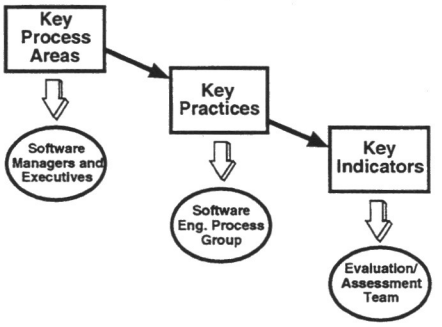
Key Indicators: most critical practices for implementing a key process area.

Maturity Questionnaire: probes that sample key indicators from each key process area in preparation for on-site interviews.

Structure of a Maturity Level -3



CMM Target Groups



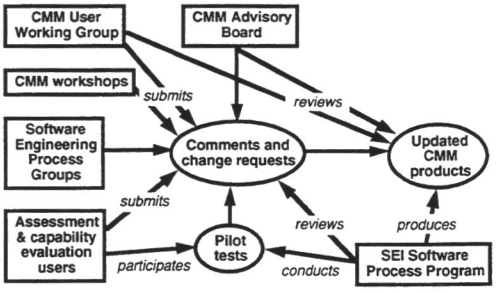
CMM Community

CMM provides a principled, public model for appraising software development capability.

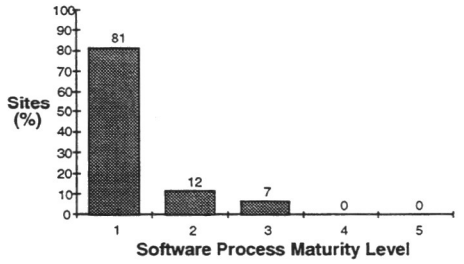
CMM owned by national software development community (and CMM being accepted abroad).

SEI exercises stewardship over the CMM.
- **evolves CMM from industry experience**
- **performs empirical research**
  - **validation**
  - **state of the practice**
- **develops CMM-based improvement methods**

Public Review Process for CMM



Maturity Distribution: 1991
59 Assessed Sites (296 Projects)



Source: Kitson & Masters, *An Analysis of SEI Software Process Assessment Results: 1987-1991*, CMU/SEI-92-TR-24, 1992

CMM Summary

Defines process capability at each maturity level

Supports assessments and capability evaluations

Defines coverage and content of Maturity Questionnaire

Provides foundation for process improvement

Specifies "what," not "how"

Software Process Improvement Programs

"Create constancy of purpose toward improvement of product and service ...Improve constantly and forever the system of production and service, to improve quality and productivity."

- W. Edwards Deming

Principles of Process Improvement

Commitment to improve must start at the top

Focus on the process, not the people

First understand the current process

Change must become a way of life

Improvement requires investment

Institutionalizing improvements requires vigilance and periodic reinforcement

Improvement Requirements

Senior management commitments

Organizational consensus on most important issues

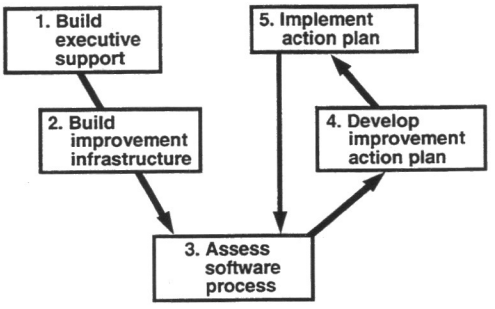
Prioritized improvements to undertake

Belief that continuous improvement will work

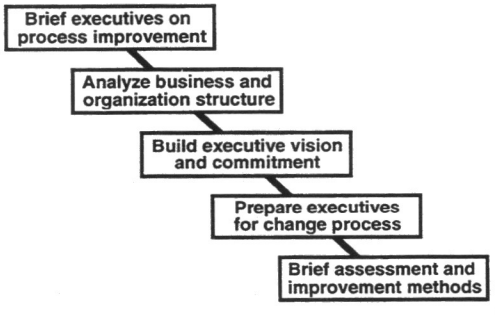
Action orientation with management attention

A dedicated focal point for the program

Initiating an Action Plan



1. Build Executive Support



Commitment Starts at the Top

Process is a management responsibility.

Management must
- set challenging goals
- provide necessary resources
- monitor progress
- insist on performance
- reward success, but not punish failure
- build an improvement culture

Do not start an improvement program without senior management commitment to action.

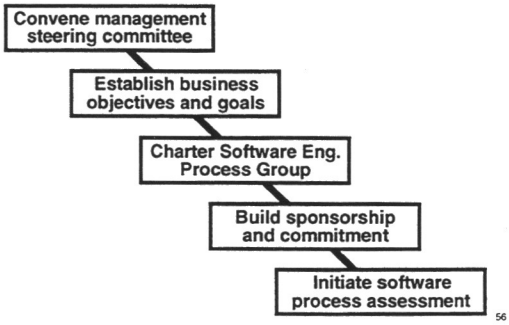
Organizational Analysis

Improvement strategy must be determined by the type of organization executives want to create.

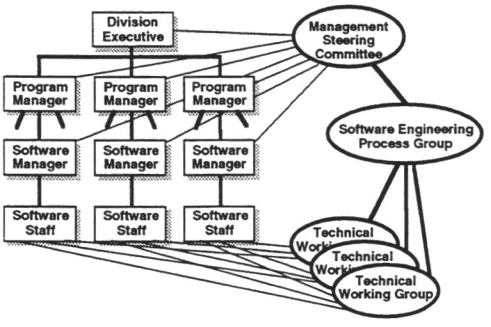
Issues in determining organizational focus
- who is source of improvement funding?
- how broad is improvement commitment?
- who will exercise review and oversight?
- what is the span of impact of the program?
- what work dependencies affect the program?
- what is the desired organizational structure?

Assessment and improvement program strategy must be designed around these decisions.

2. Build Improvement Infrastructure



Organizational Sponsorship



Organizational Roles in Improvement

Management Steering Committee
- provides management oversight
- sets policy and overall direction

Software Engineering Process Group (SEPG)
- small group of full-time staff
- facilitates overall process improvement program

Technical Working Groups
- part-time task forces of practitioners
- chartered to work on a specific process area

Responsibilities of the Management Steering Committee

Establish visible commitment to improvement

Build management teamwork for improvement

Oversee policy and strategic direction

Charter SEPG

Fund and oversee improvement program

Empower organization to act on assessment recommendations

Review progress and problems at least quarterly

Responsibilities of SEPGs

Establish process standards

Maintain the process database

Serve as the focal point for technology insertion

Provide key process education

Provide project consultation

Make periodic assessments and status reports

Characteristics of SEPGs

1% to 3% of professional software staff

Composed of seasoned professionals respected throughout the organization

Responsible to Management Steering Committee

Risks to success
- inadequate staffing
- inconsistent management support
- unrealistic expectations for improvement
- getting sidetracked with project support
- inadequate planning or infrastructure

Responsibilities of Technical Working Groups

Staff with senior people from individual projects

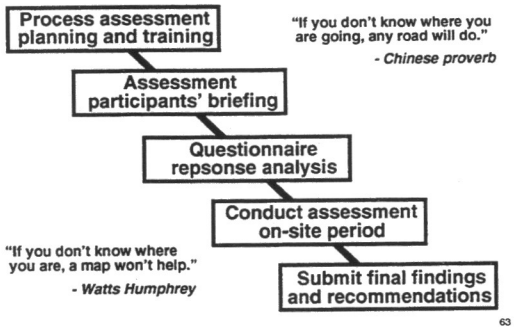
Spend dedicated time on improvement activities

Define action plan for implementing a recommended improvement across the organization

Define a tactical plan for implementing defined improvements on individual projects

Inidividual members become implementation agents on their project

3. Conduct A Software Process Assessment



What is an Assessment?

An appraisal of an organization's current software process using
- a trained team of 4–8 experienced software professionals
- review of 4 or 5 representative projects
- responses to the Maturity Questionnaire
- in-depth discussions with project managers and practitioners
- collective team knowledge and experience
- feedback on findings and recommendations

Assessment Objectives

Understand the organization's current software engineering practices

Identify key areas for process improvement

Develop consensus on improvement needs

Facilitate the initiation of process improvement by
- providing a framework for action
- helping obtain sponsorship and support

Assessment Principles

Start with a process maturity framework

Observe strict confidentiality

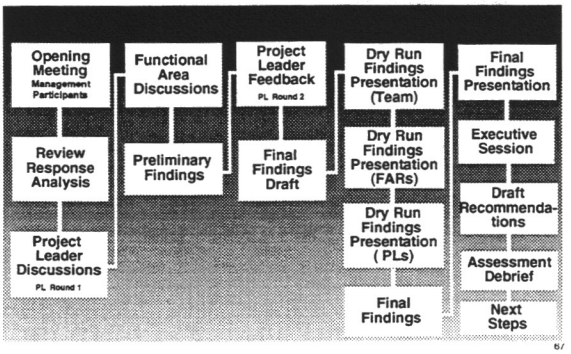
Involve senior management

Approach assessment collaboratively

Focus on improvement actions

Carnegie Mellon University
Software Engineering Institute

On-Site Assessment Process



67

Carnegie Mellon University
Software Engineering Institute

Process Assessment Summary

Process Assessment provides
- data on improvement needs
- consensus on most important problems
- motivation to undertake change

Judgement required to map key practices to specific project circumstances

Role of the Maturity Questionnaire
- springboard for assessment method
- helps organize scoring
- not all key indicators covered
- final maturity level may differ from initial questionnaire scoring

68

Carnegie Mellon University
Software Engineering Institute

What is a Software Capability Evaluation (SCE)?

A method to evaluate the capability of contractors by gaining insight into their software development processes.

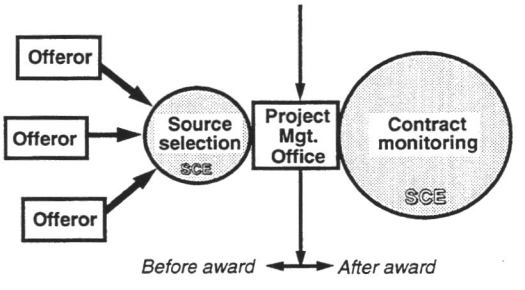
The method involves
- completed maturity questionnaires
- 5-person evaluation team
- 3–5 projects evaluated in 3-day site visit
- findings report

Findings report documents process capability against key process areas
- strengths
- weaknesses
- improvement activities

69

Carnegie Mellon University
Software Engineering Institute

SCE Method Application Areas



70

Carnegie Mellon University
Software Engineering Institute

SCE Role in Process Improvement

SCEs help acquisition organizations
- characterize contractor's current software practice
- determine resulting program risks
- set goals and priorities for risk reduction
- assess progress against goals

SCEs motivate contractors to
- assess their software development process
- establish a plan for process improvement
- provide the needed resources and management commitment

71

Carnegie Mellon University
Software Engineering Institute

Assessment vs. Evaluation

Issue	Process Assessment	Capability Eval.
Use	process improvement	source selection
Objective	assess current practice	substantiate practice
Improvement goal	catalyst for improvement	evaluate commitment
Output	input for action plan	performance risk
Range of findings	non-CMM findings possible	CMM findings only
Style	collaborative	audit-oriented
Focus of results	applies to organization	predict next project
Status of results	confidential	known to DoD

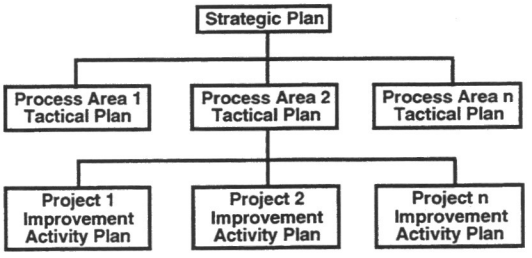
72

4. Develop Improvement Action Plan

"There is nothing more difficult to plan, more doubtful of success, nor more dangerous to manage than the creation of a new order of things."

— Niccolo Machiavelli

Types of Action Plans -1



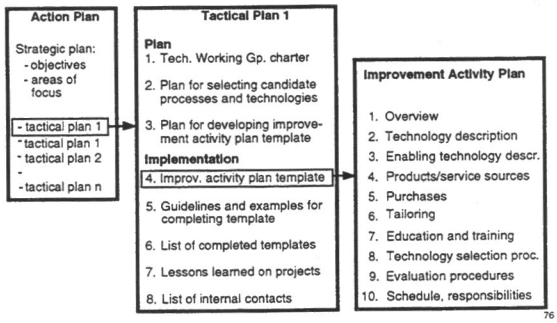
Types of Action Plans -2

***Strategic plan*: Defines process improvement goals from business perspective, consistent with senior management's strategy and vision.**

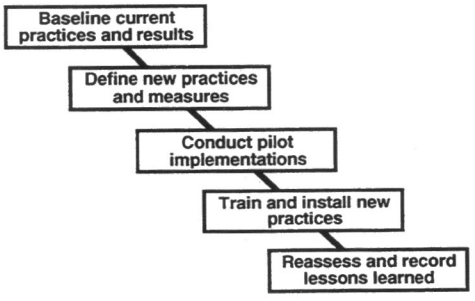
***Tactical plans*: Delineates activities for individual technical working groups assigned to each assessment recommendation.**

***Improvement activity plans*: Developed for a specific project working on improvements in one particular area.**

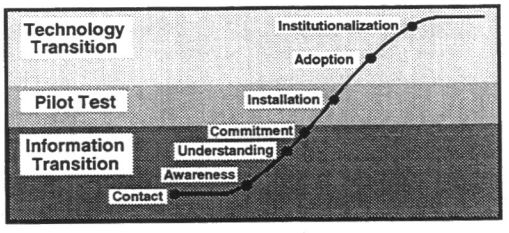
Types of Action Plans -3



5. Implementing the Action Plan



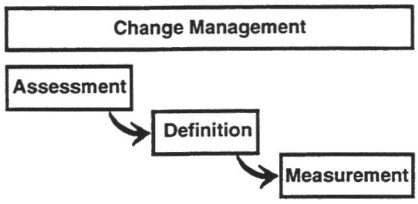
Technology Transfer Curve



Process Improvement Methods

Specific interventions designed to provide the foundation for continuous process improvement.

Techniques within each method are tailored to the maturity level of the organization.



Baselining the Current Practice

Process baseline
- how do we do it today?
- what processes are currently defined?
- how many of them are enforced?
- can we create a repository of process definitions?
- what do we have to build on?

Measurement baseline
- what measures are we collecting today?
- what is our current level of performance?
- can we create an adequate historical database?
- how can we measure the improvement effort?
- what measures can we build on?

Process Driven Development

A defined process is
- documented
- supported
- trained
- practiced as documented and trained
- enforced
- measured

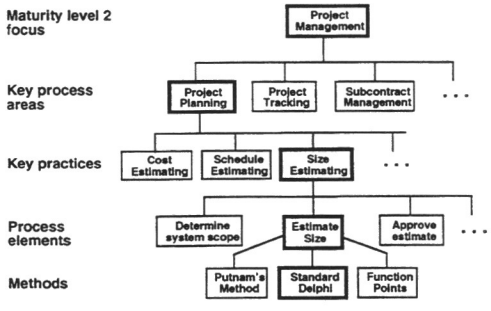
Organization-wide software process
- is tailored to meet project and product goals
- guides the selection of technology
- clarifies activities, roles, and dependencies
- is improved through measurement and feedback
- is permanent store of corporate know-how

Process Definition Focus by Level

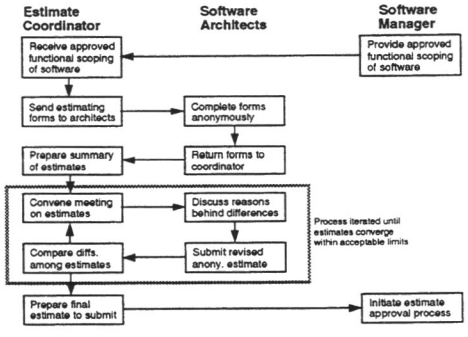
Myth: process definition occurs only at level 3

Level 5	**Innovative processes**
Level 4	**Quality management processes**
Level 3	**Engineering processes**
Level 2	**Management processes**
Level 1	**Waterfall diagram**

Project Mgt. Process Hierarchy



Process for Standard Delphi Method

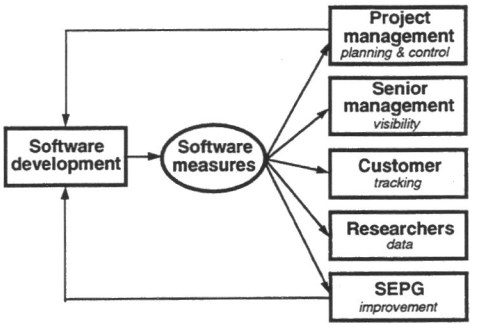


Quantitative Software Management

Measurement is necessary to sustain process improvement programs
- communicates seriousness of intent
- indicates progress and bottlenecks
- establishes good engineering practice

Baselining current performance provides
- historical data for estimating
- baseline for measuring improvement
- basis for return-on-investment decisions

Uses of Software Measurement



A Starting Point for Measurement

	Human Machine Interface (CSCI #1)
Size	• 10,247 total source lines coded to date
Effort	• 4029 staff-hours expended this month
Schedule	• CDR completed January 15, 1993
Defects	• 53 problem reports, 36 closed, 17 open

Measurement Focus by Level

Myth: measurement occurs only at Level 4

Level 5	Improvement and cost/benefit data
Level 4	Process and product quality data
Level 3	Process data
Level 2	Planning and tracking data
Level 1	Haphazard data

Roles in Managing Change

Sponsor	Authorizes and reinforces change
Champion	Vocally encourages new methods
Agent	Empowered to implement program
Target	Focus of change program

Executive Role in Managing Change

Set a vision of the desired state

Create new communication channels

Protect people from further disruptions

Monitor the course of events with SEPG

Encourage cohesion within the organization

Expect old issues to resurface and don't choke them off as irrelevant

Threats to Improvement Programs

Lack of senior management commitment

Middle management resistance

Inadequate resources dedicated to improvement

Inappropriate improvement goals

Termination before transition is institutionalized

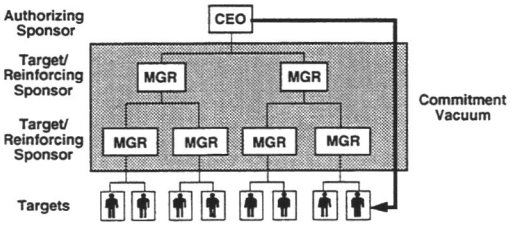
ROI fever

Lack of Management Commitment

Lack of senior management commitment
- turnover or merger
- disillusionment with results
- shifting investment priorities
- inadequate resource allocation

Middle management resistance
- overriding pressure for performance
- doubt about seriousness of senior leadership
- little confidence in new methods

Middle Management Resistance



Inappropriate Improvement Goals

Level 5 by 1995!

75 business units to be assessed by 4Q93!

Crash implementations
- no plans
- no long-term vision
- no change management

Unrealistic expectations
- the great productivity gap
- technology adoption curve

Return on Investment for Software Process Improvement Programs

"No price is too dear to pay for perfection."

- Andrew Carnegie

Return on Investment Opportunities

Lower maturity levels
- cost and schedule overruns
- project terminations
- expensive rework
  - 30% *(Boehm, IEEE Computer, Sept. 1987)*
  - 40% *(McGarry, NASA-SEL, 1987)*

Higher maturity levels
- maintenance costs
- time to market
- development effort
- software quality
- customer satisfaction

NASA: SEL Experience Reliability Factors

Factor	% improvement when in use	% of variation in reliability
Quality assurance	29	10
Code reading	29	10
Documentation	27	8
Chief programmer	8	1
Top-down dev.	6	1
Structured code	3	1
Tool use	3	1
Design schedule	-1	0

Source: Card, McGarry, & Page, *IEEE Trans. on Software Engineering*, July 1987 (45 flight dynamics projects)

Hughes Aircraft Experience: Background

Hughes Software Engineering Division
- software component of Ground Systems Group
- 500 professionals
  - 12% have > 20 years experience
  - 41% have 10–20 years experience

Software process assessment results
- 1987: Level 2
- 1990: Level 3

Source: Humphrey, Snyder, & Willis, *IEEE Software*, July 1991

Hughes Aircraft Experience: Return on Investment Data

Costs
- $45,000: process assessment
- $400,000: 2-year improvement program

Benefits
- cost perf. index (actual/budgeted cost of work)
  - .94 in 1987
  - .97 in 1990
- $2,000,000 annual reduction in cost overruns

Return on investment
- 5X first year versus initial investment
- 4.5X accrued annually against initial investment

Source: Humphrey, Snyder, & Willis, *IEEE Software*, July 1991

Hughes Aircraft Experience: Lessons Learned

Business impact
- cost estimate more credible in contract negotiation
- effect of volatile requirements is predictable
- fewer crises, less overtime

Management commitment
- delegation of responsibility is inadequate
- should be part of bonus determination

Culture change
- improvement results => pride => increased results
- common culture emerged at Level 3
- annual turnover < 10%

Source: Humphrey, Snyder, & Willis, *IEEE Software*, July 1991

Raytheon Experience: Background

Raytheon's Equipment Division
- 350 professionals

Software process assessment results
- 1988—Level 1
- 1990—Level 2

Source: Dion, *IEEE Software*, July 1992

Raytheon Experience: Return on Investment Data

Costs
- $1,100,000: 1990 cost of improvement program

Benefits
- $8,200,000: 1990 nonconformance reduction

Return on investment
- 7.5X return

Source: Dion, *IEEE Software*, July 1992

Raytheon Experience: Cost Analysis

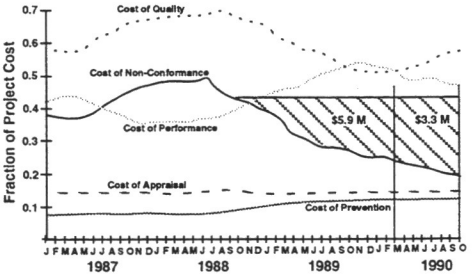
Cost types
- **Performance: cost of building it right first time**
- **Nonconformance: cost of rework**
- **Appraisal: cost of testing**
- **Prevention: cost of preventing nonconformance**

Change in average % project time by cost type

	Perform	Nonconf.	Appraise	Prevent
1988	34%	44%	15%	7%
1990	55%	18%	15%	12%

Source: Dion, IEEE *Software*, July 1992

Raytheon Experience: Savings on Cost of Nonconformance



Source: Dion, *IEEE Software*, July 1992

Raytheon Experience: Lessons Learned

Management commitment
- **program offices briefed on improvement plans**
- **buy-in from both business and engineering mgt.**

Interorganizational coordination
- **policy compliance review by each new project**
- **commitments on interfaces to other organizations**

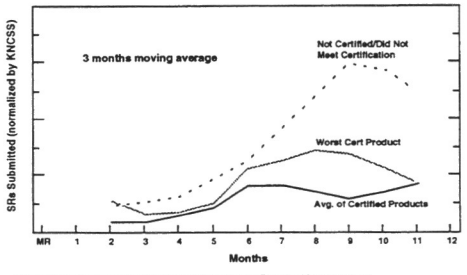
Source: Dion, *IEEE Software*, July 1992

Hewlett-Packard Experience: Quantitative Certification Process

Quantitative criteria for certifying software for release
- ***breadth:*** **test coverage of user-accessible and internal functions**
- ***depth*****: branch coverage of testing**
- ***reliability***
  - **continuous hours of operation under stress**
  - **rate of change in defect trends (stability)**
  - **ability to recover gracefully from error conditions**
- ***defect density:*** **remaining density at release**

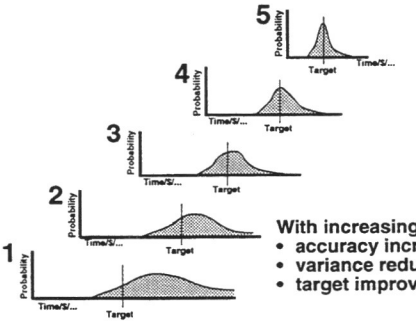
Source: Grady, *Practical Software Metrics for Project Management and Process Improvement*, 1992

Hewlett-Packard Experience: Service Request Results



Source: Grady, *Practical Software Metrics for Project Management and Process Improvement*, 1992

Process Capability Prediction



With increasing maturity
- **accuracy increases**
- **variance reduces**
- **target improves**

Difficulties with ROI Arguments

Level 1 organizations have too little data to develop ROI arguments.

Sound management and engineering practices should not have to be ROI issues.

Hard to quantify criteria
- competitive survival
- customer confidence and satisfaction
- product liability

By the time data are available few need convincing.

2 groups who don't ask for ROI arguments
- level 2 and 3 organizations
- the Japanese

Executive Cues for Success

Increasing visibility into project performance

Increasing predictability of results

Increasing staff morale

Increasing product quality

Increasing ability to manage complexity

Increasing project performance

What Should An Executive Do Now

Take up the gauntlet and lead

Don't forget to manage
- participate in planning improvements
- assign responsibility to execute the program
- provide visible support
- indicate serious intent to monitor progress
- involve everyone in your organization

Familiarize yourself with TQM and CMM principles

Institute change management and technology transfer techniques to accompany process improvements

Conclusions

Continuous process improvement
- increases software capability
- constitutes a long-term business strategy
- requires management commitment
- is no silver bullet

Software acquisition groups should evaluate a vendor's
- process capability
- commitment to improve

Process maturity will differentiate which software vendors survive this decade.

"Applying Metrics"

TUTORIAL "H"

Ms. Alka Shah-Jarvis
Charles Schwab

Ms. Alka Shah-Jarvis has 18 years experience in software development, 10 of which have been in quality assurance. Her background consists of management of large scale systems development projects, quality control, total quality management, specification reviews, management of testing process for large companies and teaching. She has been a frequent speaker on quality assurance issues at international as well as U.S.A. events and has worked in the quality management field in a variety of capacities for Bank of America, Pacific Bell and Apple Computers, Inc. Currently she is working as a Quality Assurance consultant for Charles Schwab in San Francisco and is an instructor for U.C. Berkeley, Extension as well as U.C Santa Cruz, Extension in the field of Quality.

Ms. Shah-Jarvis is the President of Bay Area Quality Assurance Association (BAQAA), an Applied Total Quality Advisory Board Member for the University of California-Berkeley-Extension, a Certification Board Member for the Quality Assurance Institute (QAI) and an Advisory Board Member to the Certificate Program in Continuos Improvement and Quality Management – U.C. Santa Cruz Extension. She is a Certified Quality Analyst.

WHY MEASURE?

- Olympic
- Traffic Violations
- Customer Dissatisfaction
  - Initial Project Meetings
  - Staff Meetings
- Common Complaints
  - Inadequate testing
  - Bad fixes/changes
- Increased Maintenance Cost
  - Management curiosity
- Low Employee Morale
- Defensive orientation
  - to justify existence

DEFINITIONS OF SOFTWARE QUALITY

- **Barry Boehm**

  Achieving high levels of user satisfaction, portability, maintainability, robustness, and fitness for use

- **Phil Crosby**

  Conformance to user requirements

- **W. Edward Deming**

  Striving for excellence in reliability and functions by continuous improvement in the process of development, supported by statistical analysis of the causes of failures

- **Capers Jones**

  The absence of defects that would make software either stop or produce unacceptable results. Defects can be traced to requirements, design, code, documents or bad fixes and can range from minor to serious in severity

POPULAR SAYINGS

- **To Measure Is To Know**
  James Clark Maxwell

- **You cannot Control What You Cannot Measure**
  Tom De Marco

- **Anything Can Be Measured**
  Tom Gilb

- **Invisible Targets Are Usually Hard To Hit**
  Tom Gilb

- **Plan, Do, Check, Act"**
  Deming

BASIC QUESTIONS

- How much time do we have to accomplish a good system of measurements?

- Interested in quality, quantity or both?

- Are there any measures already in place that can be used?

  - Automatic recording of problems(in place)
  - User survey (already developed)

- What are the levels of organizational resistance and what would be the best way to overcome?

SOFT VS. HARD MEASURES

- **Soft Measures**
  - **Focus on Attitude or Perception concerning the quality of the service or product a customer receives**
  - **Emphasizes on customer's requirements and expectations**
  - **User Satisfaction**
- **Hard Measures**
  - **Number of Changes**
  - **Number of Defects/Type**
  - **Bad Fixes, Times Regression Tested**
  - **etc...etc...**

CAUSES OF MEASUREMENTS

- Problems with requirements
- Environment problems
- Design errors
- Inadequate testing
- Response time problems
- Stress problems
- Bad fixes/changes
- Inaccurate/unfriendly documentation

BENEFITS OF METRICS

- Increase customer satisfaction
- Increase revenue
- Improve quality of processes
- Provide constructive means to make decisions about quality of the product
- Decrease development costs

Cost of Metrics

- **Training: required for the quality assurance group or other administrative group in the use of established metrics and interpreting the data collected**

- **SDLC: based on the metrics evaluation, there may be changes required in the SDLC**

- **Tools: new equipment may be required to count metrics, i.e. function points, errors**

- **Organization change: based on metrics, some organization change may be required**

- **Evaluation costs: collection, evaluation and reporting costs.**

DEFINITIONS OF SYSTEM QUALITY/COMMON DEFECTS

- The following definitions you may want to measure:

- Correctness: extent to which a program covers the requirements

- Reliability: extent to which a program can be expected to perform the intended function with accuracy

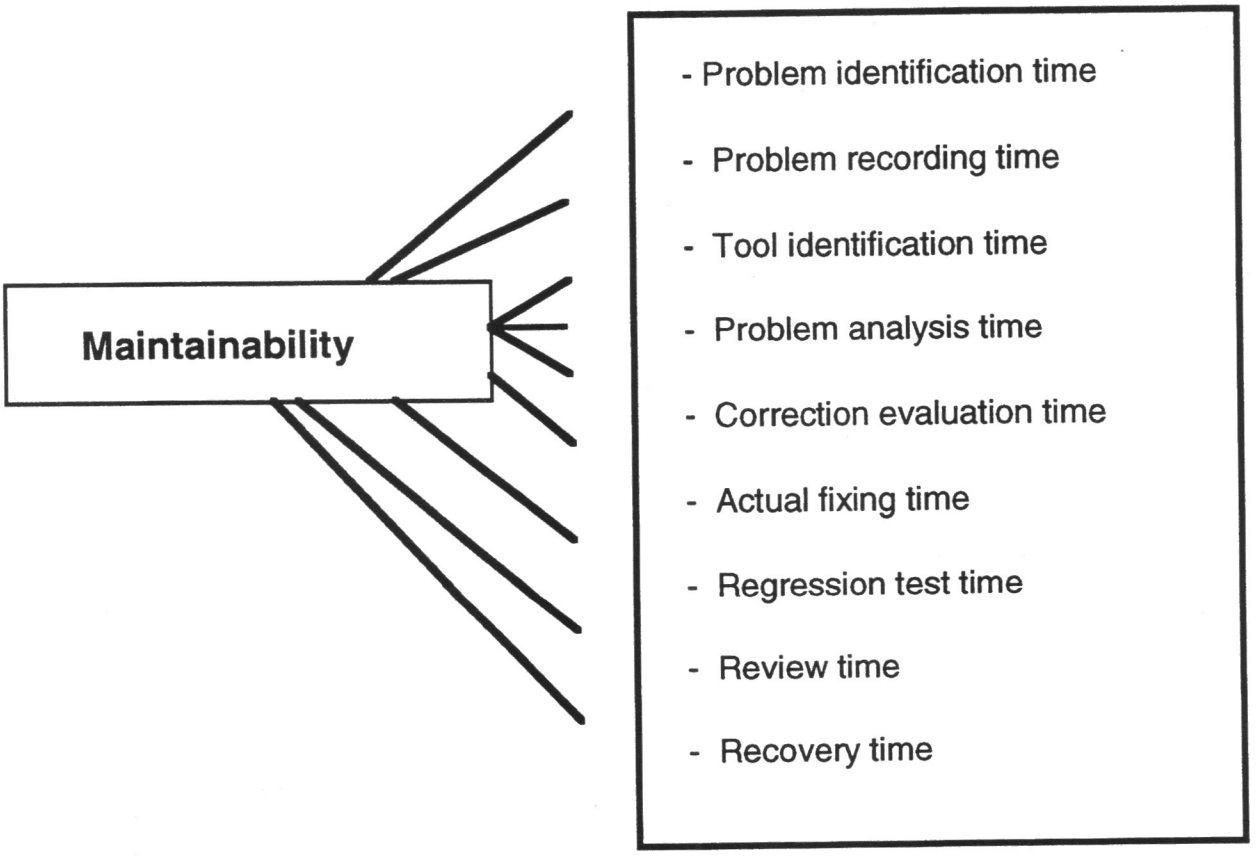
- Efficiency: the amount of resources and lines of code required to perform a function

- Integrity: ability to secure the data so that it cannot be accessed by just anyone

- Usability: how difficult is it to learn

- Maintainability: how hard is it to find and fix the problem

Sub-division of maintainability



- **Testability: efforts required to test a program**

- **Flexibility: the ease of updating or modifying a program**

- **Portability: efforts required to transfer a program from one hardware configuration to another or from one software system to another**

- **Reusability: what extend of code could be reused for another application?**

MEASURE USAGE

- **Set quantifiable targets for all the activities and outputs**

- **These targets will permit the amount of work required in each stage to be estimated for scheduling or financial reasons**

- **Measure the actual values**

- **Compare the actual values with the desired goal**

Types of Measures

- **Measures connected to requirements and design documents:**
  - **Size**
  - **Number of words**
  - **Number of pages**
  - **Number of functions, categorize small functions vs. large**
  - **Number of changes to the system at a later date**

- **Measures connected to the programming modules**

  - **Reliability of the modules**
  - **Consistency of the modules**
  - **Number of modules specified at a later phase of the system and were not specified in the original design**
  - **Number of modules which calls a given modules**
  - **Number of external data items from which a module reads**
  - **Number if external data items to which a module calls**

- **Measures connected to the testing and quality control activities**

  - **Test Plan: number of planned test cases that ran successfully**

  - **Success/effectiveness of test cases against its original test plan**

  - **Number of new, unplanned test cases which had to be developed at a later time**

  - **Debugging efficiency**

COMPLEXITY METRICS

- Measures of program characteristics that make the logic paths of a program difficult for a programmer to follow:
  - number of files
  - program size
  - number of verbs
  - number of "if" statements
  - number of paragraphs
  - total line of code
  - number of loops (go to based loops)

ARCHITECTURE METRICS

- Objective measures of the method in which a program is structured or how it is developed, eg:

  - go to
  - entry point
  - termination verbs
  - an overall architecture metric

DIAGNOSTIC METRIC

- Measure of violations of the programming language which may not cause compile errors or warnings but which are confusing, eg.

  - dead code
  - unresolved procedure exits

POPULAR MEASURES

1. NUMBER OF DEFECTS

- During Testing Phase:
  - Total # of Defects Found
    - Type of Severity
      - . Show Stopper: abends, function are unusable, require adjustment of hardware or software architecture
      - . Serious: problem affects many users, problem causes the system to do wrong transaction, i.e. post to the wrong account
      - . Moderate: system fails to satisfy the requirement but there is a workaround

- **Time Taken to Fix**

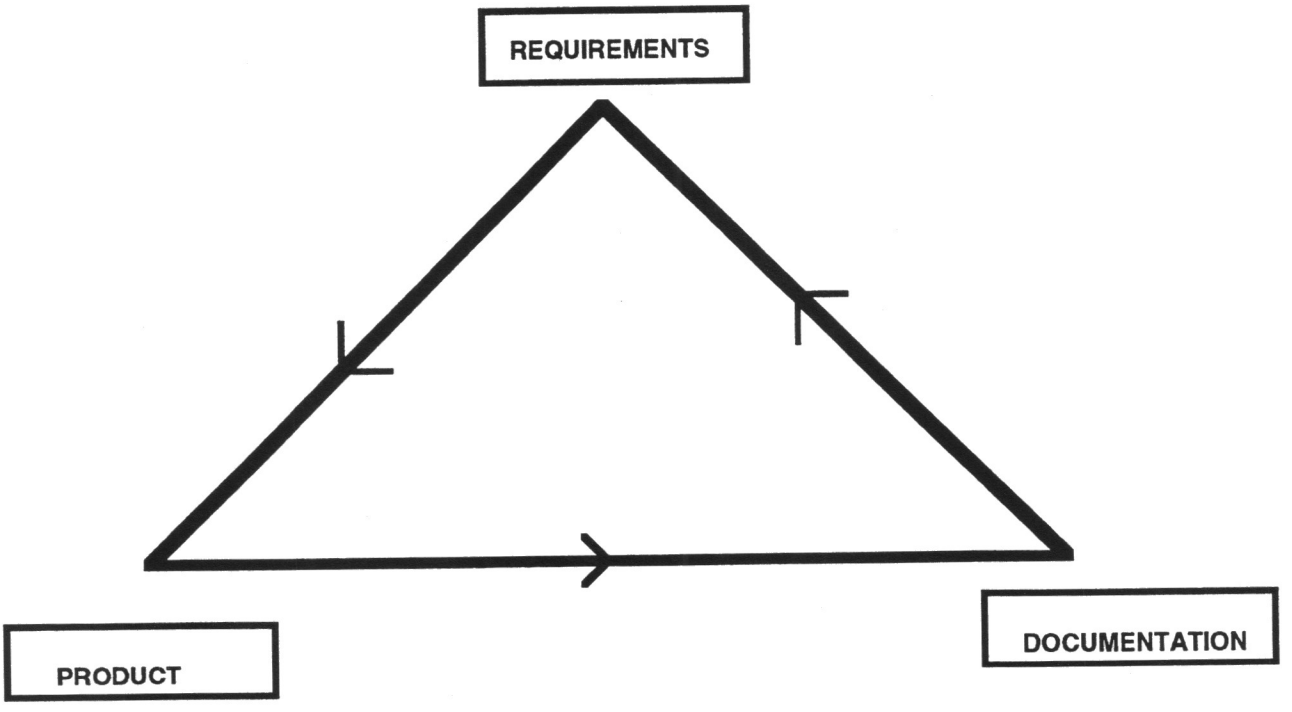
- **# Open Problems At The Turnover Time**

- **Total Test Time For Future Reference**

2. TEST OMISSIONS & NO/POOR COVERAGE

- Thorough test plan
- Cross reference between requirement & test cases
- Test inspections
- Test case reviews
- Proof of Correctness triangle

PROOF OF CORRECTNESS



3. BAD FIXES, # OF TIMES REGRESSION TESTED

- Determine the defect origin, i.e. Req, Design etc.
- Determine the human source
- Determine the defect severity
- Determine the time for fixing problems and the retesting time

4. NUMBER OF CHANGES

- Define the type of change, i.e. functionality, design, response time, environment

- Determine "mean time" from the requirement signoff

5. SIZE OF PRODUCT/EFFORTS/DEFECTS

- LINE OF CODE MEASURE

  - Empty lines
  - Comments/statements
  - Source lines
  - Reused lines
  - Lines used from other programs

- FUNCTION POINTS

  - **Logical Internal files**
    Data stored & maintained by system

  - **External Interface Files**
    Data passed or shared with other systems

  - **External Inputs**
    Transactions, additions, changes

  - **External Outputs**
    Reports

  - **External Inquiries**
    No updates

6. DOCUMENTATION ERRORS

- **Type of Documentation**

- **Nature of error:**
  - **missing functionality**
  - **unclear explanation**
  - **spelling**
  - **user-friendly**

7. USER SATISFACTION

- **Written surveys**

  - **when to send?**

- **Phone calls**

- **One/One feedback meetings**

EASY TO IMPLEMENT MEASURES

- Entries/Exits per module
- Defect Analysis
- Defect Rate
- Defect Density
- Mean Time To Failure
- Function Test Coverage
- Failure Rate
- Man-hours per defect detected

EASY MEASURES

- Entries/Exits per module: can be accomplished through FP
- Defect Analysis: determination of the phase of lifecycle in which the defect was encountered
- Defect Rate: number of defects found
- Defect Density: defects counted per XX lines of code
- Mean Time To Failure: is it better than past, how long before the system failed?
- Function Test Coverage: what is the percentage of functional tests covered?
- Failure Rate: failures vs. execution time
- Man-hours per defect detected: eg. less than two defects detected per 200 hours of testing

MEASURE USAGE

- **Entries/Exits per module:**
  - **Used to predict the completion time of the system**
  - **Determine quality of the system based on historical data (# of problems discovered in the past in a similar system)**

- **Defect Analysis**
  - **To measure the type of defects, i.e. environment, testing, poor requirement etc.**

- **Defect Rate**
  - **Identify programs that may need further testing**
  - **Gain confidence in the software prior to release**
  - **Predict remaining fault and when the product may be available**
- **Defect Density**
  - **Evaluate defects per number of executable source code lines**
  - **Part of Inspection procedure**
  - **Used for release criteria**
- **Mean Time To Failure**
  - **Gives a barometer for judging the quality of the software**

- **Function Test Coverage**
  - **Used for release criteria**
  - **Indicates the functions not tested**
- **Failure Rate:**
  - **Used to evaluate number of failures encountered to the actual operating time**
  - **Gives a barometer for judging the quality of software**
- **Man-hours per defect detected**
  - **Considered for releases decisions**
  - **Part of Inspection procedures**

OTHER METRICS

- **Management:**
  - schedule: budgeted cost of work scheduled
  - cost: actual cost of work for the finished project
  - productivity: resource utilization vs. time
  - computer usage: time and memory used on the computer

COMMON DEFECTS

- **Requirements**

  - **Ambiguous**
  - **Incorrect**
  - **Missing**
  - **Incomplete**

  Ambiguous Algorithm: divide the number of ambiguous words by the number of words and multiply by 100, you will get % of ambiguous words

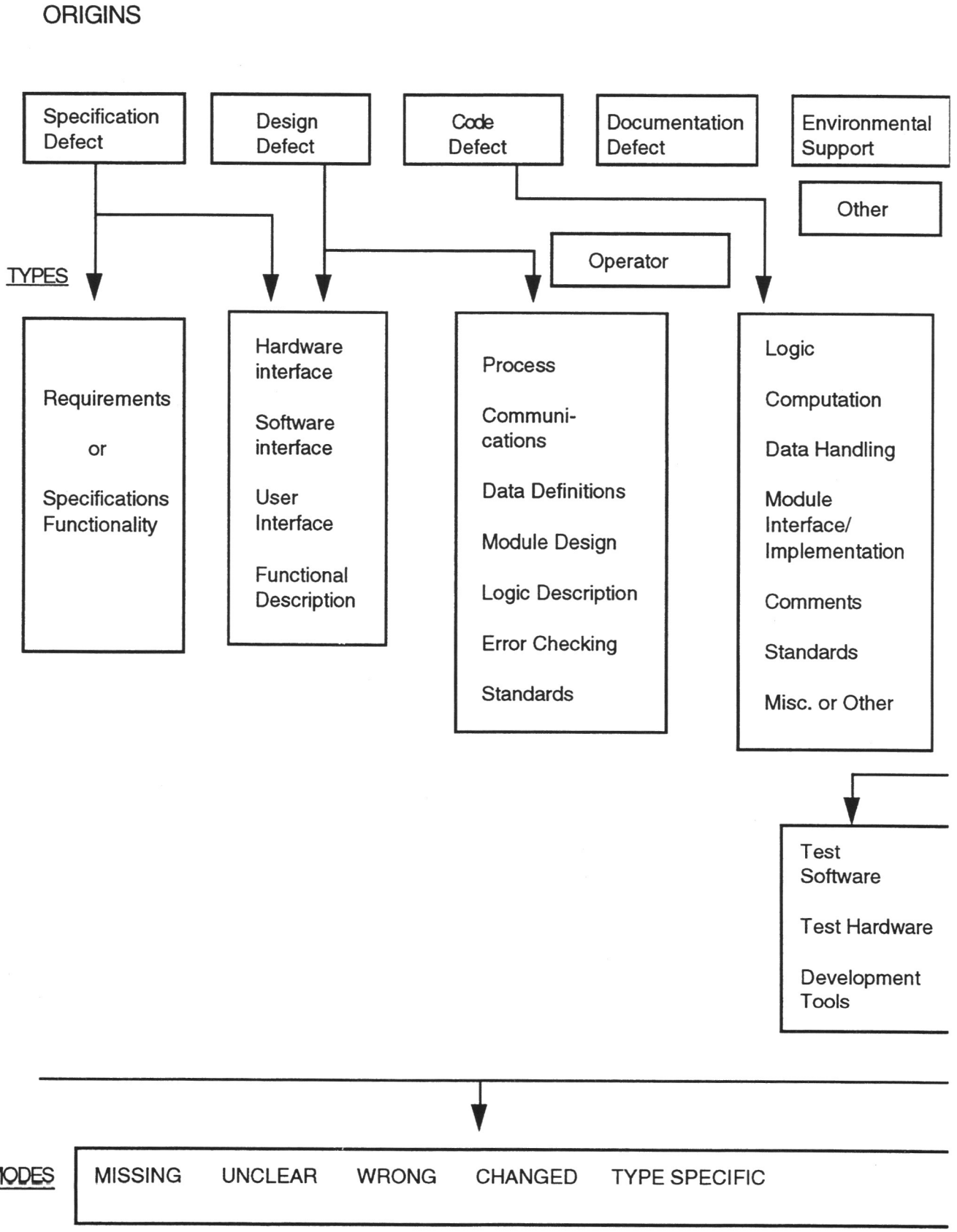
- **Design**

  - **Functionality**
  - **Data Flow**

- **Implementation**

  - **Coding**
  - **Testing**
  - **Response Time**

CATEGORIZATION OF SOURCES OF SOFTWARE DEFECTS



MEASURING REQUIREMENTS

- **Consistency: consistent with interfaces from other systems**

- **Traceability: requirements are written in a fashion that they can be traced in design as well as testing**

- **Completeness: the requirement phase is completed and no changes are required**

DESIGN

- **Size: measures the complexity of the system**

- **Modularity: measures how well a design is decomposed into small, manageable modules**

- **Coupling: which measures the interfaces between the units**

CODE

- Standards compliance
- Report Analysis
- Fault Density

TEST

- **Failure Rate**

- **Mean Time To Failure**

- **Coverage**

TIME

- To complete the system
- Number of Resources

BASED ON FUNCTION POINT

- Number of Defects
- Completion time estimates
- Resource utilization

LINES OF CODE vs. FP

- Dependent on the complexity of code
- The system must be completely coded
- Can be used for any type of software development
- Used to measure productivity and number of defects

FUNCTION POINT

- Independent of languages
- Used to measure productivity and # of defects
- Available early during functional specification timeframe
- Applications can be counted in short time

MAJOR OBSTACLES

- Definition
  - Software quality is not well defined
  - Is it satisfying product requirements?
  - Product response time is not identified accurately
  - .....
- Goal Definition
  - The purpose of measurement is vague: is it cost, schedule, identifying problem areas.....

- **Complex Measurement Criteria**
  - The software measurement programs are not too practical and are cumbersome to pursue
  - Requires high data collection
  - Feedback from measurement is slow
  - .....

- **Vague Evaluation**
  - Interpretation of the measures are confusing or conflicting
  - Lack of standards for data collection could result in this category
  - .....

- **Far Too Many Metrics**
  - Collection of too many measures
  - People do not have time to evaluate all the measures
- **Resistance**
  - High when a system lacks quality
  - Employees must be periodically retrained, lack of knowledge results in resistance
  - Lack of employee involvement and employee empowerment

NEGATIVE METRICS

Examples:

- System Availability: the system may be up 99% of the time and the users can log on 99% of the time, but what good is it if the users are not happy with what the system does?

- Lines of Code: the value is subjective and significantly altered by individual programming styles. If the individual is rewarded based on performance, this can bring negative results

BASIC QUESTIONS

- How much time do we have to accomplish a good system of measurements?

- Interested in quality, quantity or both?

- Are there any measures already in place that can be used?

  - Manual reporting of problems(in place)
  - User survey (already developed)

- What are the levels of organizational resistance and what would be the best way to overcome?

PLAN MEASUREMENT ACTIVITIES

- Establish history of development costs for each system: type, time, size etc.
- Establish history of expenses for maintenance for each system: categorize by type, time, size
- Establish history of operational expenses: cost of training new people, computer processing time, etc. (history can be used for annual budget purpose also)
    - Annual internal management survey: evaluate upward without causing dramatic grief!
    - Focus on success
- Measurement organization take a long time (years) to establish, start now!

ESTABLISH MEASUREMENT REQUIREMENTS

- Direct metric value: a numerical target for a factor to be met in the final product

  - System response time/4.2 sec
  - System availability/99%
  - User satisfaction/95%
  - Network availability/98%

- Predictive metric value: a numerical target related to a factor to be met during system development. This is an intermediate requirement that is an early indicator of final system performance. For example, design or code errors may be early predictors of final system error rates.

- Identify unacceptable quality

(IEEE, Standards for A Software Quality Metrics Methodology)

IMPLEMENTATION ISSUES

- **Develop sound basis for collection of data, determine the data to be collected, identify which department will be involved**

  - Benefits:

    - monitoring progress
    - improving productivity
    - assisting predictability

- **Explore what are the true concerns of the users about the system & start measuring**

- **Consider how predictive metrics cab be developed to take account of changes/fixes**

- Consider users needs of metrics with each application as it may differ

- Don't staff a measurement group until the basic charter is established and approved

- Should not staff with all "excellent" employees. This may cause hostility

PROTOTYPE THE MEASUREMENT PROCESS

- Continuous metrics validation
- Adjust the metrics plan
- Obtain commitment for the metrics plan
- Prototype the Measurement process: test the plan on a small project
- Performance and compliance of the measurement project must be externally audited, even though monitored by QA

SELECTION OF TOOLS

- Criteria for selecting tools:
  - Ease of use
  - Accuracy/reliability
  - Availability
  - Ability to handle large volumes of data
  - Cost

TOOLS OF QUALITY

Pareto Chart

- Define the characteristics for comparing problems, i.e. number, %, cost etc.
- Gather data
- Arrange the data groups by size with the largest group on the left.

Questions

- Which improvement opportunity should we select first?
- Which problem to select first?
- Which problem has a greater contribution towards cost and customer dissatisfaction?
- How do we communicate to the rest of employees why a certain improvement opportunity was selected?

Process Flow Chart

- Identify the reason for your Flow Chart and give it a title
- Be clear in stating the beginning and the end of the process
- Identify the steps between the start and stop points and describe each step as wither an action, decision or document transmittal.

Questions:

- How does the business run?
- Who does what and when?
- Where does the problem exist?
- Which areas will the recommended process change affect?
- Where else can we make process change?

Cause & Effect Diagram (Fishbone Diagram)

- Define the problem and place it at the head of the fish

- Define the major causes and place them on each bone:
  - PPPP= people, policy, procedures, plant
  MMMM= machines, methods, manpower, material

- When the entire diagram is completed, check to see if the root causes of the problems have been identified

QUESTIONS:

- What is causing the problem?

- How do we efficiently collect the ideas to define root causes of the problem?

- How do we show the findings about the root causes? What would be the potential effects of a solution?

Histogram

- Establish the range of your data by subtracting the smallest data point from the largest (range = high value-low value)
- Decide on the number of bars you would like to have on the histogram
- Calculate the size of each bar (one bar = range/number of bars)
- Tally the number of data points in each bar in order
- Give explanation for variation displayed in your chart

Questions:

- How often does errors, complaints, incorrect outcomes occur?
- How frequently does something happen in a time period?

Brainstorming

- State the purpose of the exercise, i.e. identify problem, resolve problem etc.
- Giver equal amount of time to each person
- Record the ideas
- After recording each idea, let the group expand or elaborate on the ideas
- Reduce the list by eliminating duplicate ideas

Questions

- Is your team in conflict about:
  - Products or services produced by any one group?
  - Potential reason(s) of a problem?
  - Potential solution(s) of a problem?
  - Which problem to tackle first?

Multi-Vote

- Count the number of ideas on the list; divide the number by 50% and add one, i.e. for 20 ideas, you will come up with 11

- Each individual votes the number of times decided in the above step

- If you have 11 ideas, keep the top 5 vote getters. This way you will reduce the number

Questions:

- Are the team members in conflict?

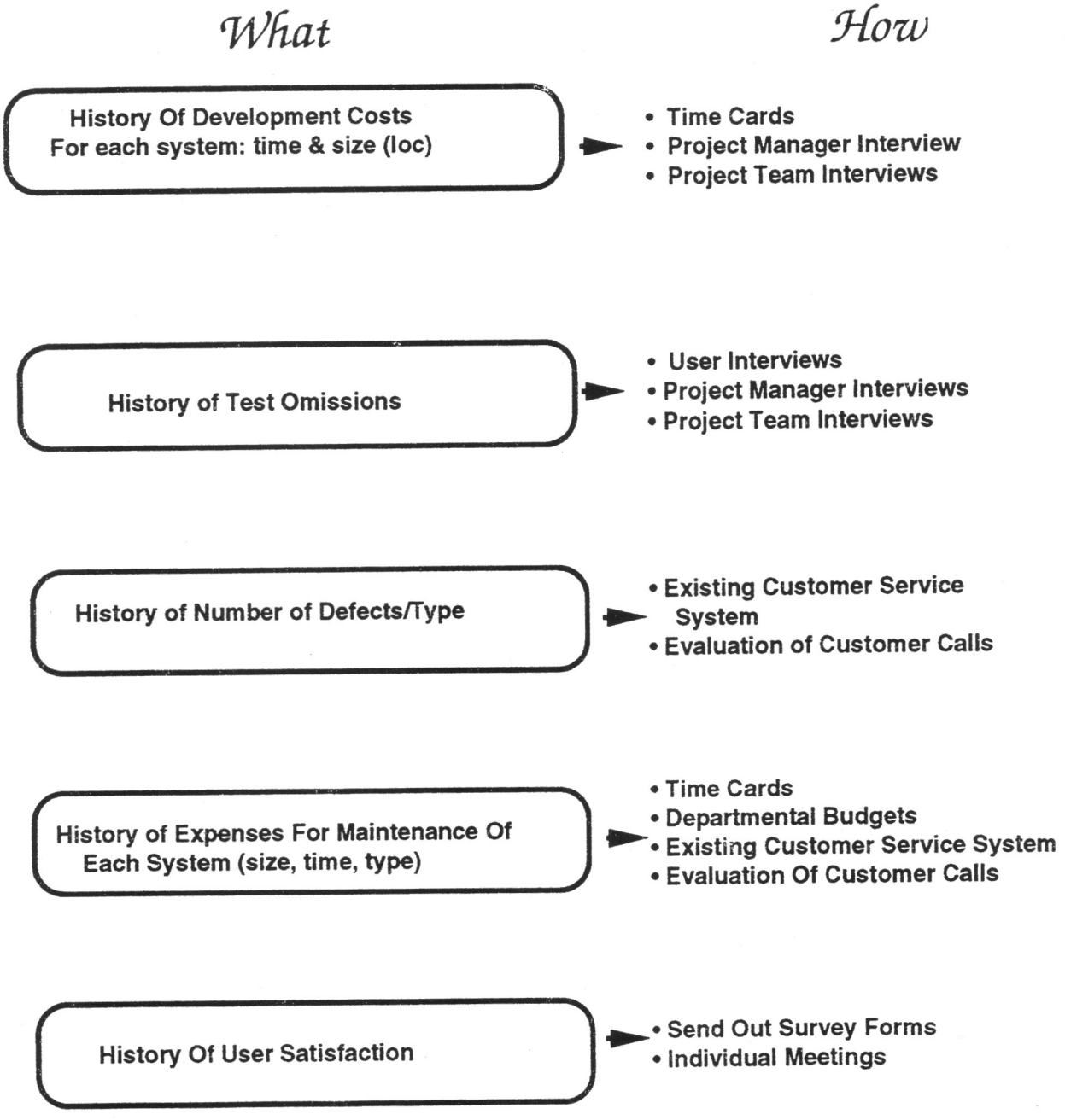
- Do you want the team members to select the priority items for improvements?

- Do you have limited time to select and resolve problems?

BASED ON MEASUREMENTS

- Review historical problem reports of similar systems to identify problem areas
- Identify causes of the problem areas (interpret the results)
- During inspections of new products, look for potential problem areas
- Review and test all potential changes
- Conduct vigorous testing for complex modules
- Obtain necessary signoffs

PLAN FOR ESTABLISHING HISTORICAL DATA



IMPLEMENTATION OF STRATEGIES TO PREVENT FUTURE PROBLEMS

(RECURRENCES)

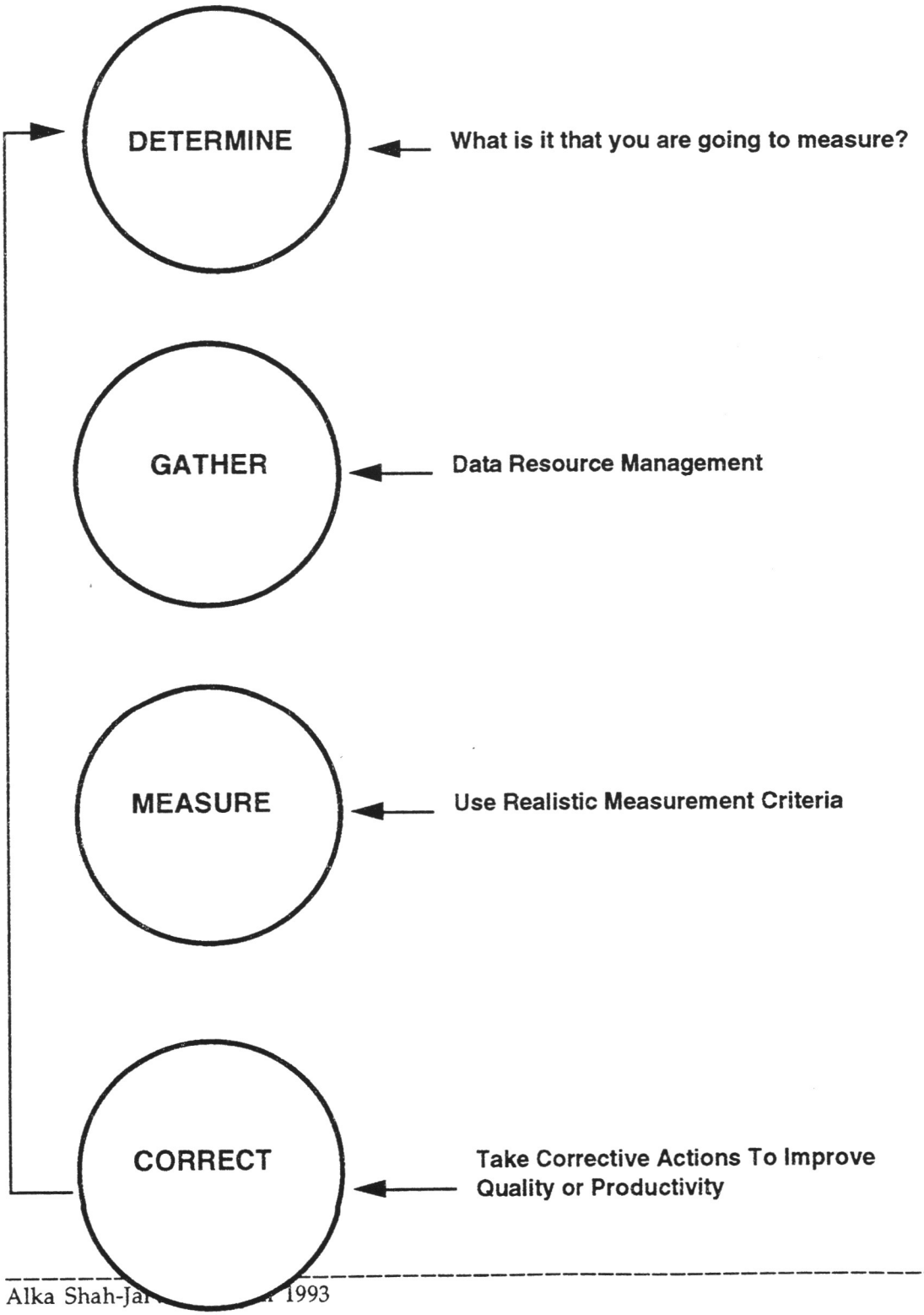
Issue	Recommendation
• Coding started too early	Define Code Phase Initiation Criteria
• Inconsistent specification	Standard structured specification
• New Product poorly focused	Review charter marketing/product res.
• No Corporate quality policy	Create corporate quality policy
• Uncoordinated develop. effort	Standard SDLC/proj. mgmt. tools
• Inadequate resources	Standard estimation/resources
• No concern for product robustness	Standard product specification
• Fuzzy specification/scope	Joint Application Design
• Poor Documentation	Walkthrough, Docu. Impl. standards
• Standard Test Env. needed	Define std. product & test bed
• Central Software Library needed	Implement
• Too much rework client site	Formal alpha accept. test/exit criteria
• No database control	Implement
• Quality irrelevant to Mgmt.	Install
• No code or version control	Implement
•	
•	
•	

SEI PROCESS MATURITY MEASUREMENT PARADIGM

Level	Measurement
5. Optimizing	Process improvement Effectiveness of Tools, Techniques and Methods
4. Managed	Defect distribution Productivity, Resource utilization Plans vs. actuals
3. Defined	Design, Design methods, Code complexity Inspection criteria
2. Repeatable	Requirement size, # people, Experience, Time & Cost
1. Initial	Product size, staff effort

(By: Watts Humphrey, SEI. In his book Managing the Software Process specifies measure that should be collected at each of the five process maturity levels)

PROCESS MODEL FOR MEASUREMENT



KELVIN'S PRINCIPLE

" I often say that when you can measure

what you

are speaking about,

and

express it in numbers,

you know something about it,

when you cannot express it in numbers,

your knowledge

is of a meager and unsatisfactory kind"

Lord Kelvin - 1983

"Software Change – A Guide for Process Improvement"

TUTORIAL "J"

Mr. Bob Wachtel
GPSA

Mr. Bob Wachtel is an independent consultant from Occidental, California who works with organizations in the US and several other countries to improve software analysis techniques and system change management. He has over twenty-five years of experience in system development, procurement, research, and instruction.

SOFTWARE CHANGE - Guide for Improvement

© 1992, 1993 by

Robert Wachtel
Box 38
Occidental CA 95465
707-874-3195
rwachtel@attmail.com

All rights reserved.

CONTENTS

Changing Systems

The Life Cycle and the Change Model

The "Missing Link" - The Analysis Database

Analysis Phases

Phase 1

Phase 2

Phase 3

Phase 4

Phase 5

Phase 6

System Test Organization

Regression Test Organization

Costs

Systems are like airports – obsolete or under construction.

System change requests occur because of changes in:

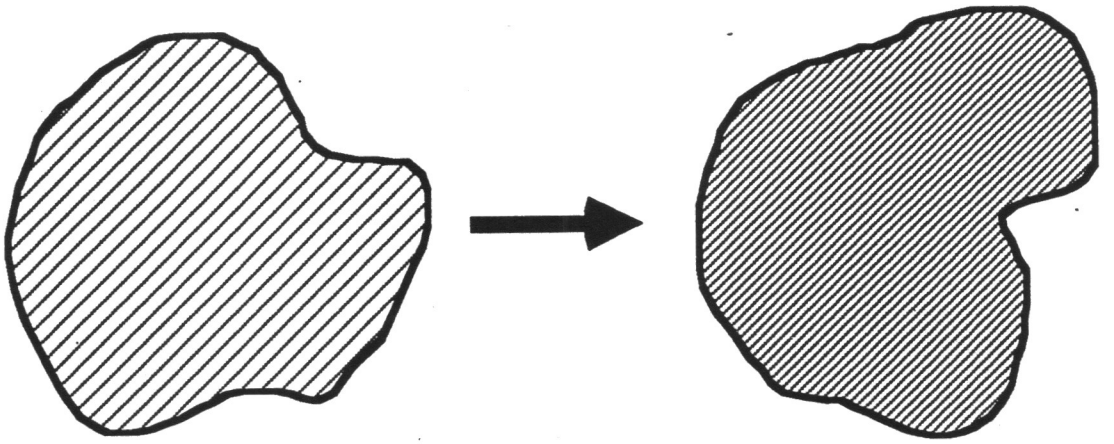
- **markets** – For example, the Vice President of Marketing and the Vice President of Product Development go to lunch and return with an idea for a new product line, which, of course, must be handled by the order processing system.

- **regulations** – For example, the government of Italy decides to revalue the lire by dividing it by 100 and calling it the nuovo lire. No problem for school children – just move the decimal point two places to the left. But what about all those financial programs and databases?

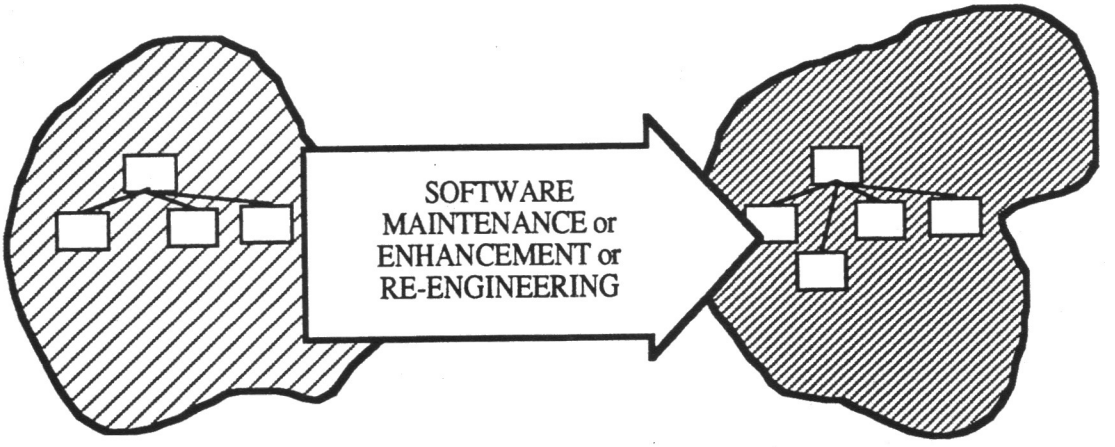
- **changes in technology** – For example, upgrading an application program to handle new operating system features described as "transparent" by the vendor.

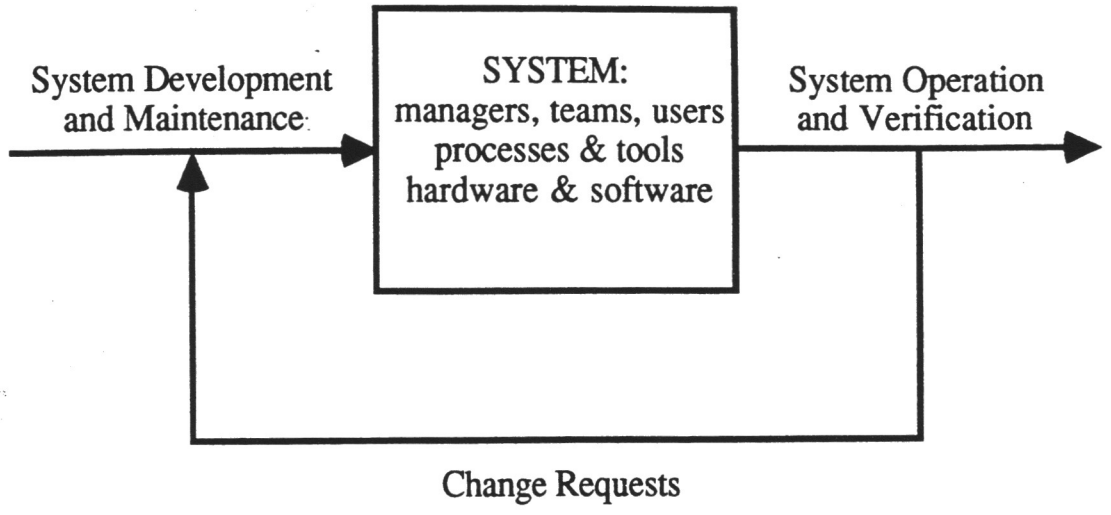
- **management policy** – For example, management now requires summary reports on follow-up sales calls. Management policy is often aptly described by one of Parkinson's Laws which states, "Any organization of a thousand people or more can create enough internal work to make external contacts unnecessary."

Systems change!

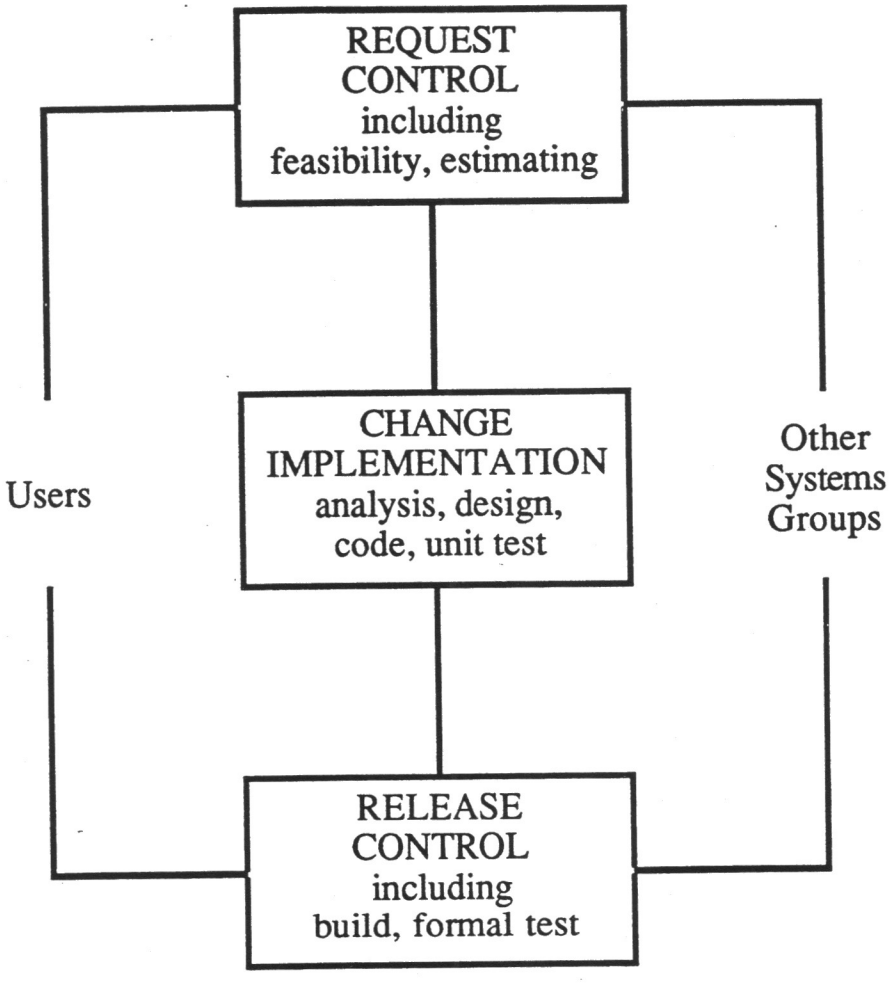


Thus, sometimes software changes.



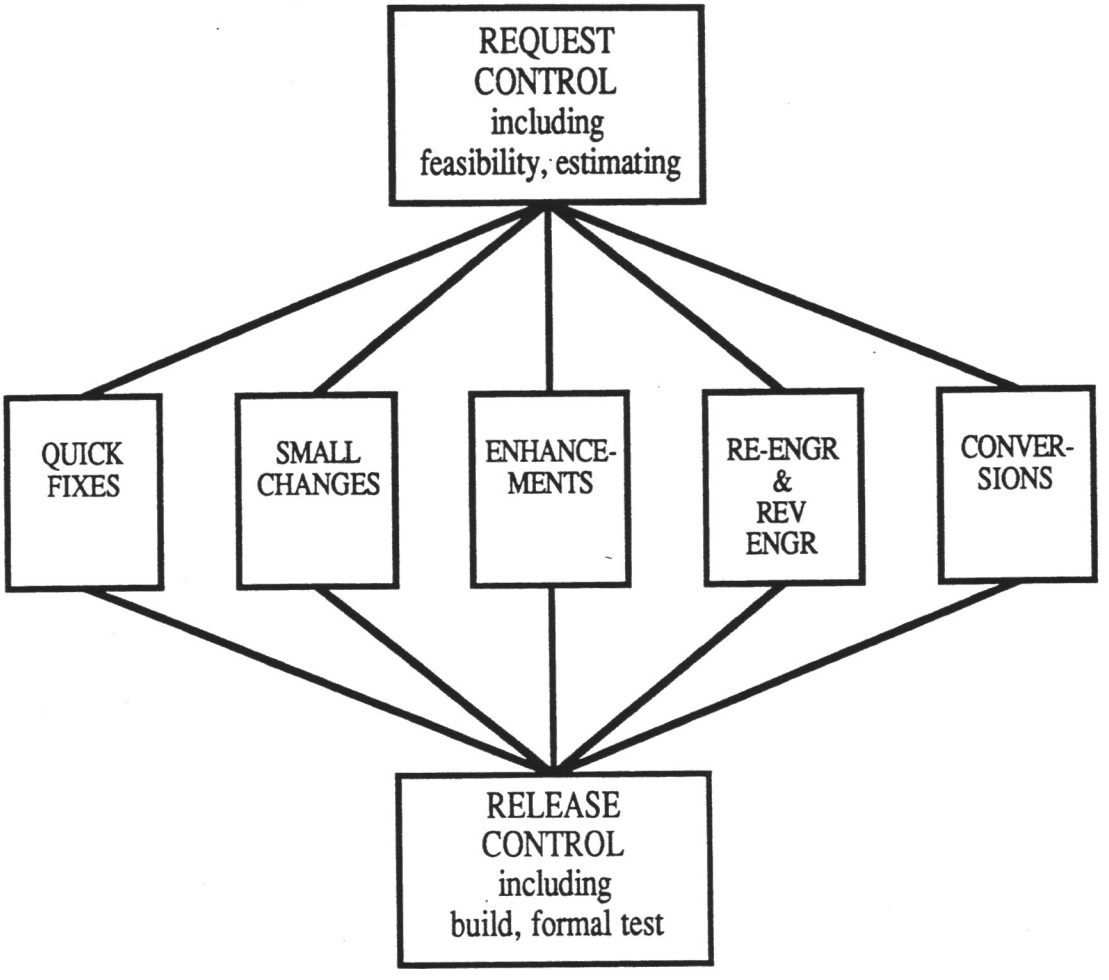

System Development
and Maintenance
SYSTEM:
managers, teams, users
processes & tools
hardware & software
System Operation
and Verification
Change Requests

A General Model of System Change



Model used by
Centre for Software Maintenance (CSM) Ltd, Durham, England
and John Foster, Software Maintenance Research, British Telecom

A Broader View of System Changes



The Existing System Change Process

- How thorough is each part of the system change process?
- On what assumptions is each part of the process based?
- For example
  - History – oral tradition
  - Low likelihood of high impact error
  - Adequate verification techniques
  - Low impact of common errors – cost effective recovery
  - "That's the way we've always done it."

Extent of requests

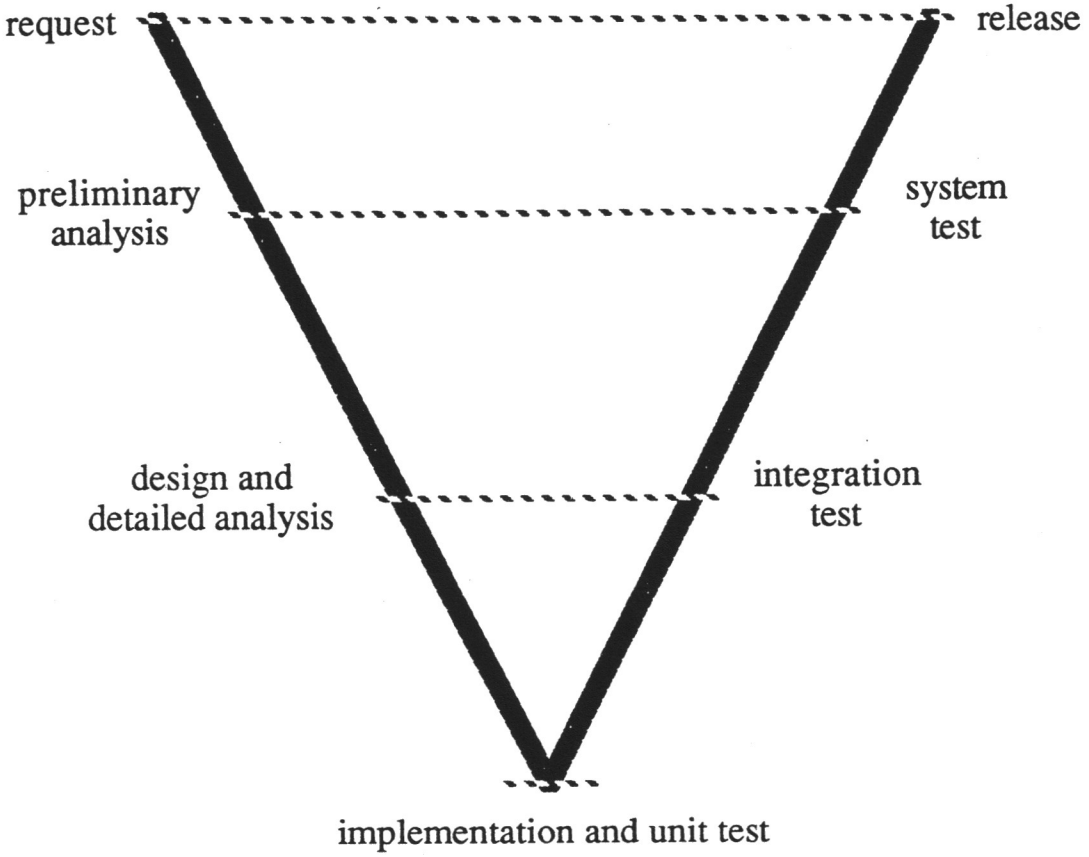
no change	moderate change	extensive change
TESTING	MAINTENANCE	CONVERSION
VALIDATION	ENHANCEMENT	RE-ENGINEERING
VERIFICATION		REVERSE ENGINEERING
QUALITY ASSURANCE		RESTRUCTURING
AUDITING		REWRITING

Testing or auditing can be thought of as requests for information about a system. At the other extreme, re-engineering and conversion imply extensive changes. Most requests imply moderate change, which is usually called maintenance or enhancement. There are no universally accepted definitions for these terms, but they all involve extensive analysis both of the requests themselves and the software that they affect.

The Software Life Cycle - a guide for analysis phases

- An engineering problem-solving approach to systems.
- Useful for any application, large or small.
- Useful for any language or system.
- A scheme that defines intermediate results.
- Life Cycles can become too bureaucratic or more complex than the systems they are intended to guide, but the general idea of the software life cycle is sound and widely used.
- Adaptation of the Software Life Cycle to define analysis phases to:
  - Organize software in any state of order or chaos
  - Work under the usual time pressures
  - Provide a framework for saving analysis

The V Model


request
release
preliminary analysis
system test
design and detailed analysis
integration test
implementation and unit test

Typical Intermediate Results in Life Cycle Development

request
FEASIBLE REQUESTS

preliminary analysis
REQUIREMENTS

design and detailed analysis
SPECIFICATIONS

implementation and unit test
CODE & TEST RESULTS

integration test
TEST RESULTS

system test
TEST RESULTS

release
INSTALLED SYSTEM

These general phases are necessary for both development and maintenance, regardless of the scale of the job.

Typical Intermediate Results for Testing

request
TEST POLICY & RESPONSIBILITIES

preliminary analysis
TEST OBJECTIVES &
ACCEPTANCE CRITERIA

design and detailed analysis
TEST CONDITIONS

implementation and unit test
CODE & TEST RESULTS

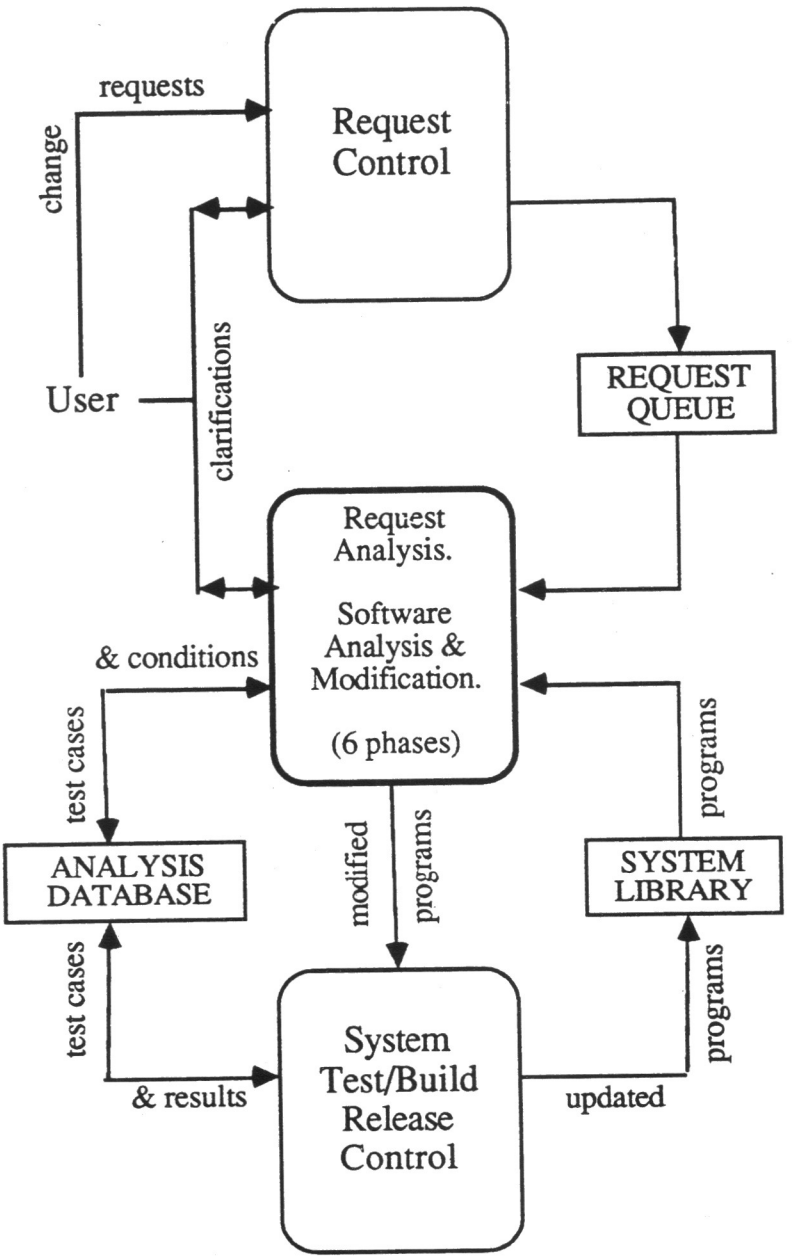
integration test
TEST RESULTS

system test
TEST RESULTS

release
INSTALLATION TEST RESULTS

The testing life cycle is parallel to both development and maintenance life cycles.

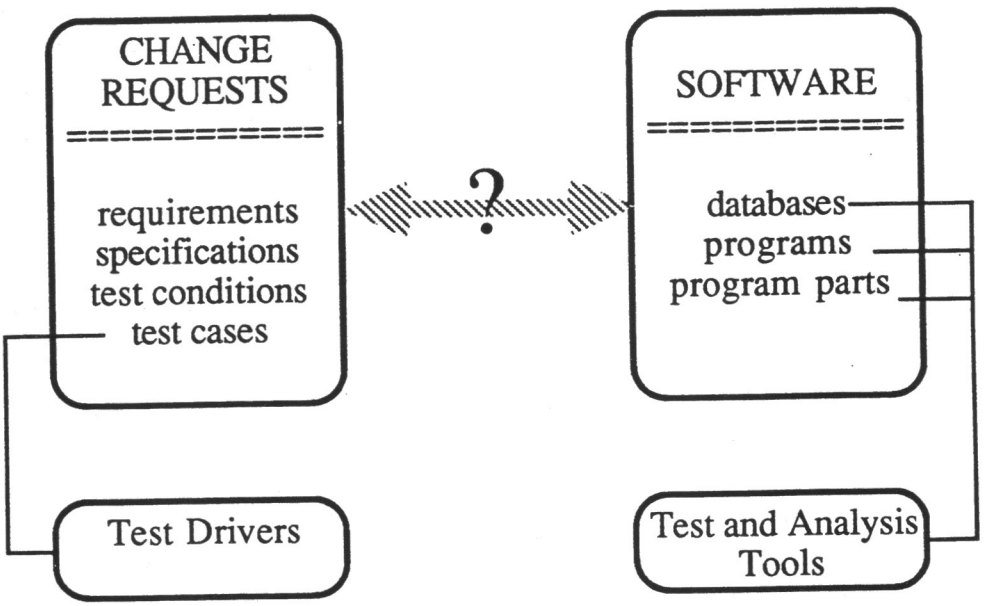
The System Modification Process



Team Responsibilities for the Change Process

- Defining it and its component parts
- Defining the information required to conduct each part
- Defining the results of each part
- Developing checklists for reviewing each part
- Adapting and applying tools and techniques to make each part work
- Walking through / inspecting intermediate and final results when appropriate
- Determining what "appropriate" means
- Conducting the work simultaneously for various change requests (i.e. handling interruptions)

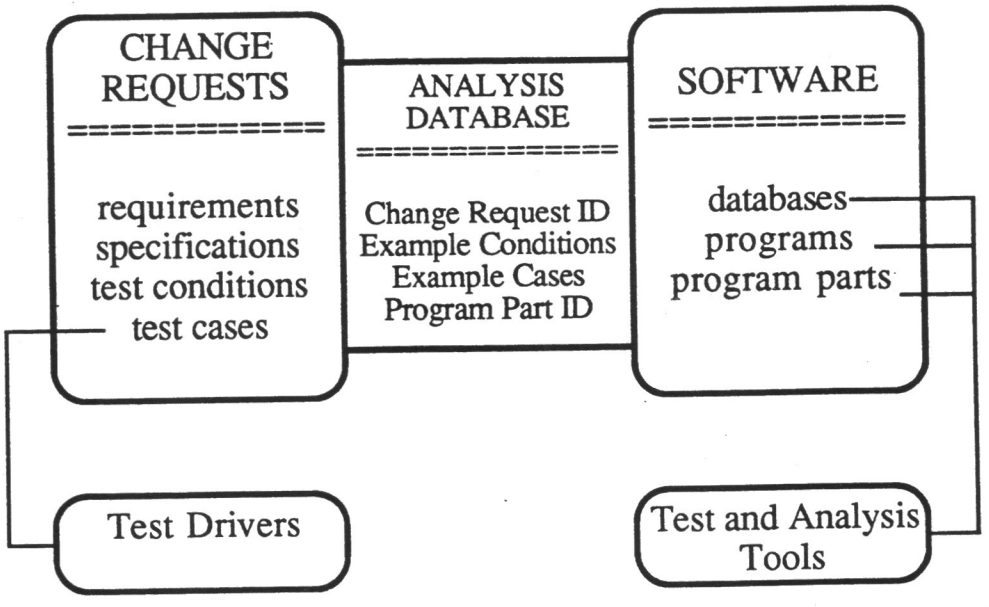
"Missing Links" in Change Management



"The Missing Links"

- Some of them are in people's memories
- Some of them are "documented"
- Some of them are forgotten once the system change is made
- So, to manage (including estimating) and carry out the work
  - People are assigned to programs for long periods
  - This restricts assignment of resources
  - Limits flexibility
  - Leads to stagnation
  - And the "world renown" solution to relieve the slow, tedious way that programs are learned is to "document more"

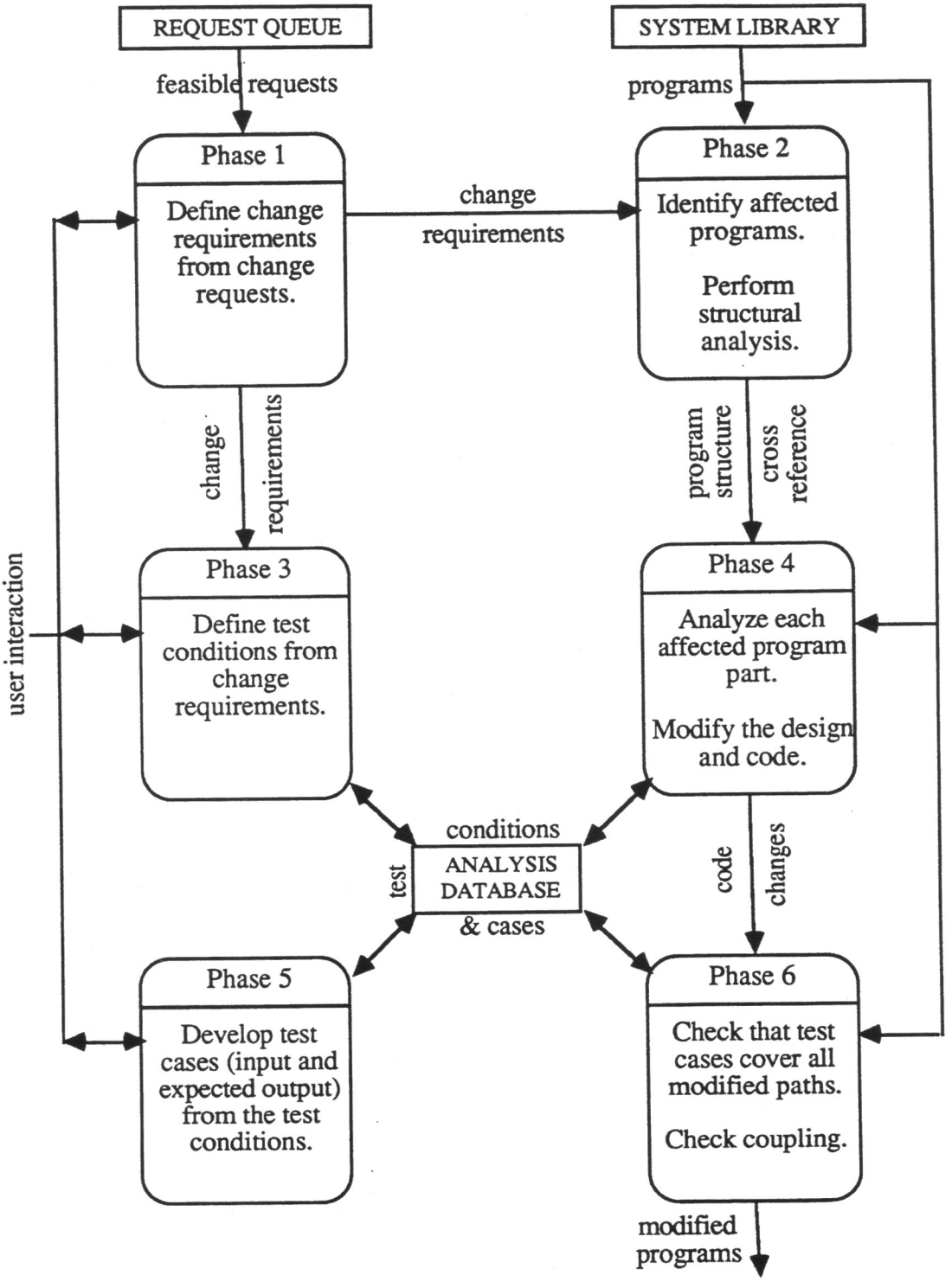
ANALYSIS DATABASE -- Supplying the "Missing Link"



Defining and Implementing an ANALYSIS DATABASE

- Ordinary database package on a PC or mainframe
- Possibly as extra (comment) fields in a test driver
- Build it incrementally
- One entry is enough to begin
- Usefulness grows with number of entries
- Useful without needing to be "complete"
- Completeness - even if it could be defined
  - impractical and probably too costly
- Applications
  - change analysis
  - software analysis
  - planning and estimating
  - test preparation
- Incremental cost
- Not for volume tests

Six phases of analysis



Two tracks of analysis.

Functional: PHASES 1, 3 , 5	**Structural: PHASES 2, 4, 6**
Black box	**Glass box**
External view	**Internal view**
User view	**Programmer view**
Emphasis on data transformation	**Emphasis on control, logic, algorithm, and coupling**

Three levels of analysis.

Preliminary, estimating: PHASES 1&2

Requirements	**Programs, structures**

Central, functional: PHASES 3&4

Specifications, test conditions	**Program parts, functions**

Detailed, checking: PHASES 5&6

Test cases	**Control paths and coupling**

PHASES 1, 3, 5

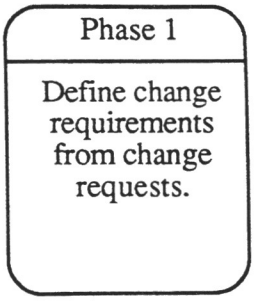
External point of view – a **function** is an operation that

- **transforms data** (sometimes expressed in the form of actions or objects) from input into output, and
- **transfers control** from entry to exit.

PHASES 2, 4, 6

Internal point of view – a **function** can usually be decomposed into smaller functions.

- **structure** – the **control** relationships among functions.
- **coupling** – the **data** relationships among functions.



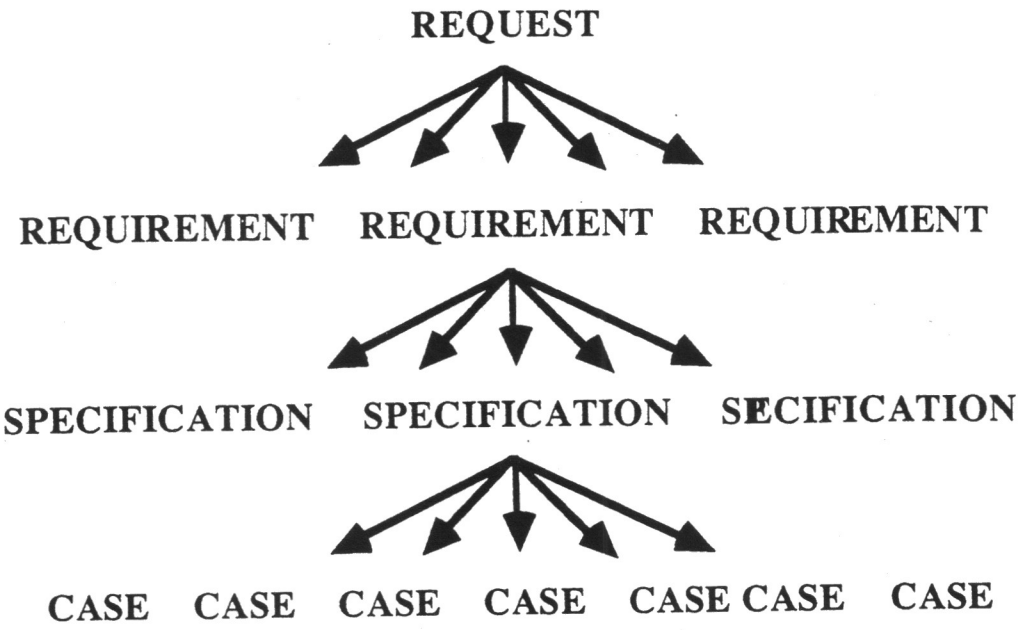
Information required for Phase 1

- Feasible request
- Change history

Results of Phase 1

- Definitions of change requirements

Decomposition of requests.



Functional change requests imply change requirements, changes in the action of the system as perceived by users. Change requirements imply change specifications, changes in design as perceived by programmer-analysts. Change specifications imply changes in particular cases, inputs and outputs.

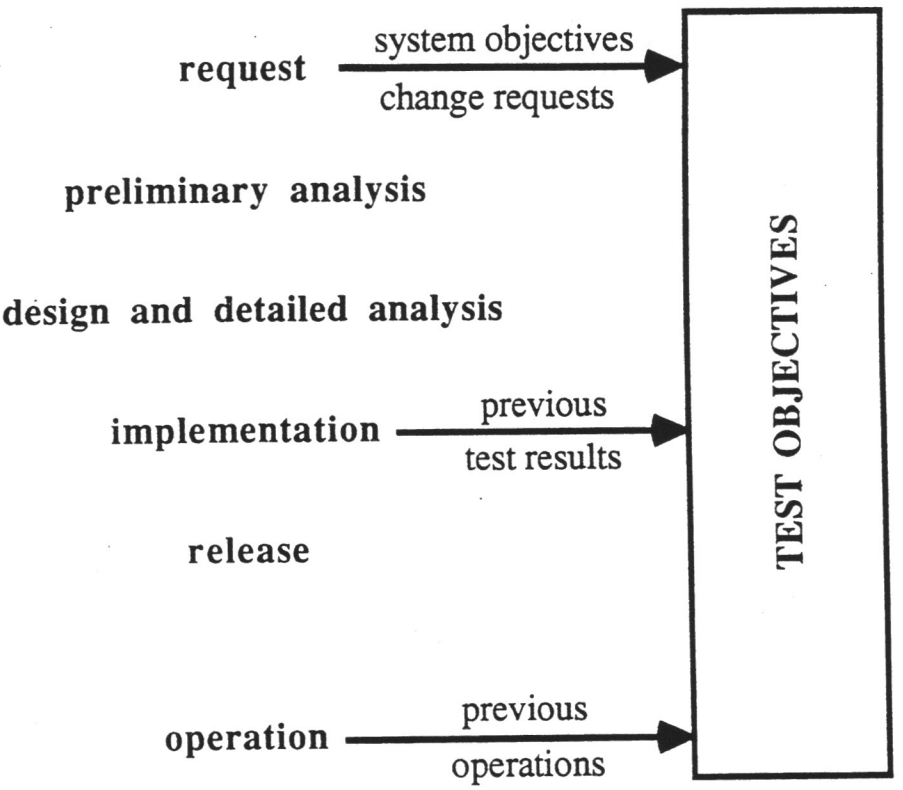
Design change requests often do not affect requirements, but imply change specifications, for example, a change to a program to use a new internal communications scheme. Particular cases must be analyzed and tested.

Error correction is a form of request that might only affect particular cases.

Information requests (for example, an audit request) might require information about requirements, specifications, and particular cases.

Test objectives that correspond to system objectives and change requests.

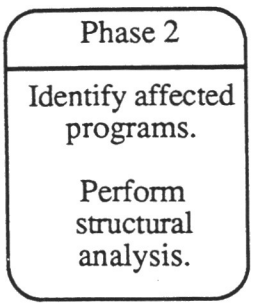
Sources of test objectives.



Checklist for Phase 1

1 Are the change requirements clear enough to being Phase 2?

2 Are the change requirements clear enough to begin to formulate specifications and test conditions in Phase 3?



Information required for Phase 2

- Change requirements

Results of Phase 2

- Programs and databases affected by the changes
- Structure of programs affected by the changes

Tools for Phase 2

- Environmental analyzers
- Static analyzers
- Restructurers
- Cross reference programs
- On-line debuggers

Checklist for Phase 2

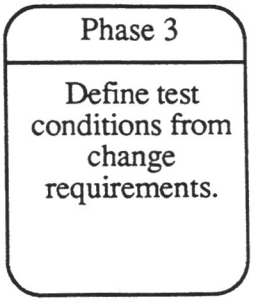
1 What programs are affected by the changes?

2 What databases?

3 What is the extent of the proposed changes?

4 What is the condition of each program affected by the changes?
(e.g. poorly structured, well structured, adequately documented, etc)

5 What modules or program parts are affected by the changes?



Information required for Phase 3

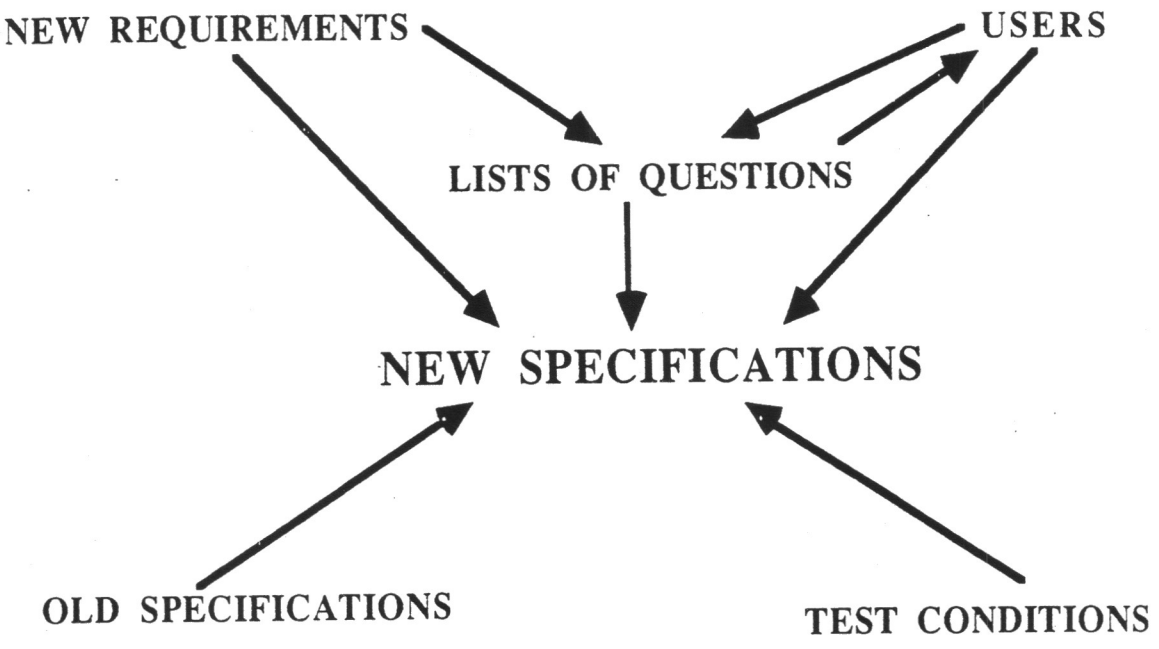
- Change requirements
- Programs and parts of programs affected by the changes
- History (related changes)

Results of Phase 3

- Functional change specifications
- Test conditions

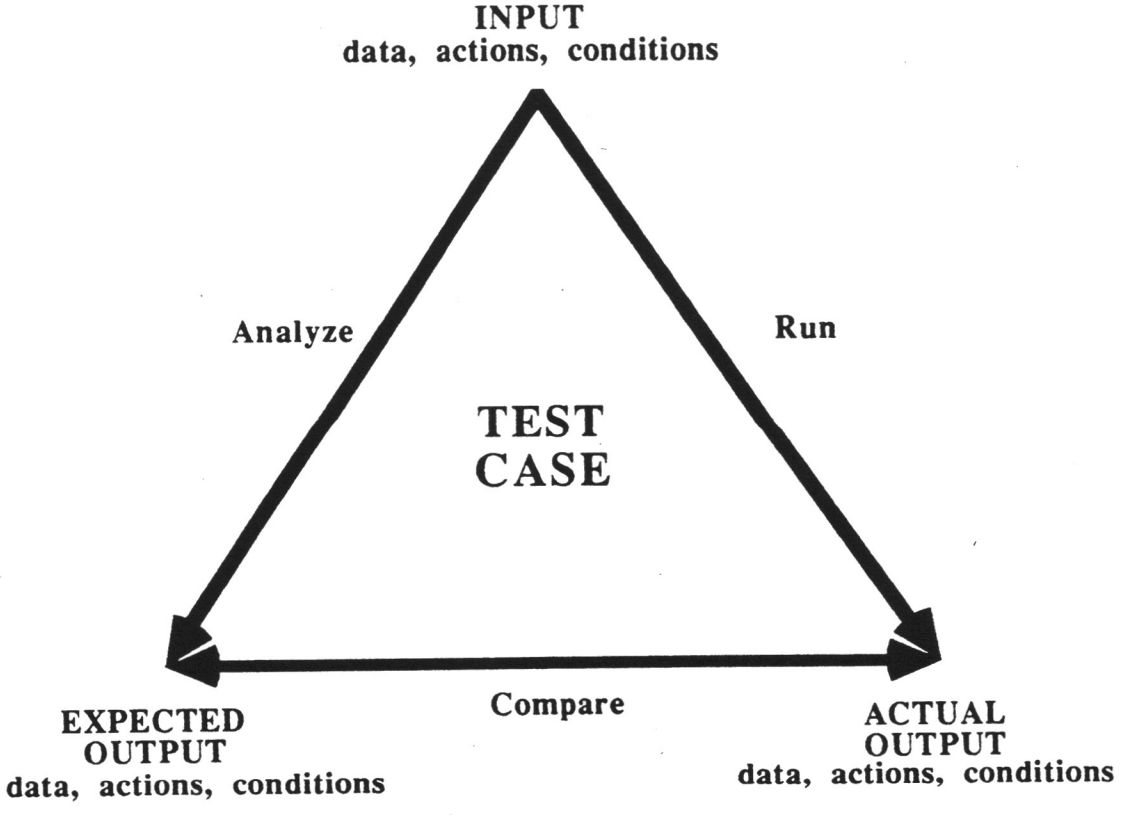
Definition of new specifications

Sources of new specifications.



Requests imply **test cases.**

- **Input** data and actions under certain **conditions.**
- A test that is run to produce **actual output** data and resulting actions and **conditions.**
- A comparison of **actual output** with **expected output** data, actions and **conditions.**
- Therefore, **conditions** must be identified and analyzed to produce **expected output.** Since this must be done anyway, the work can be used for immediate analysis.



Functional test conditions that correspond to requirements and specifications

Function: What the system does.

Data Structures: Input or output databases, files, data communication interfaces, etc.
Form: Format of data structures, e.g. report or screen layouts.
Number: Number of data structures.
Volume: Volume of the data to be input or output. Empty files.

Data Elements: Input or output without yet considering combinations, order or syntax.
Type: For example, numeric, non-numeric, real, etc.
Number: Number of input or output elements.
Value: Ranges of data values, including boundary values.

Data Relationships: E.g. combinations, order, or syntax of data elements. (Consider these after data structures and elements.)

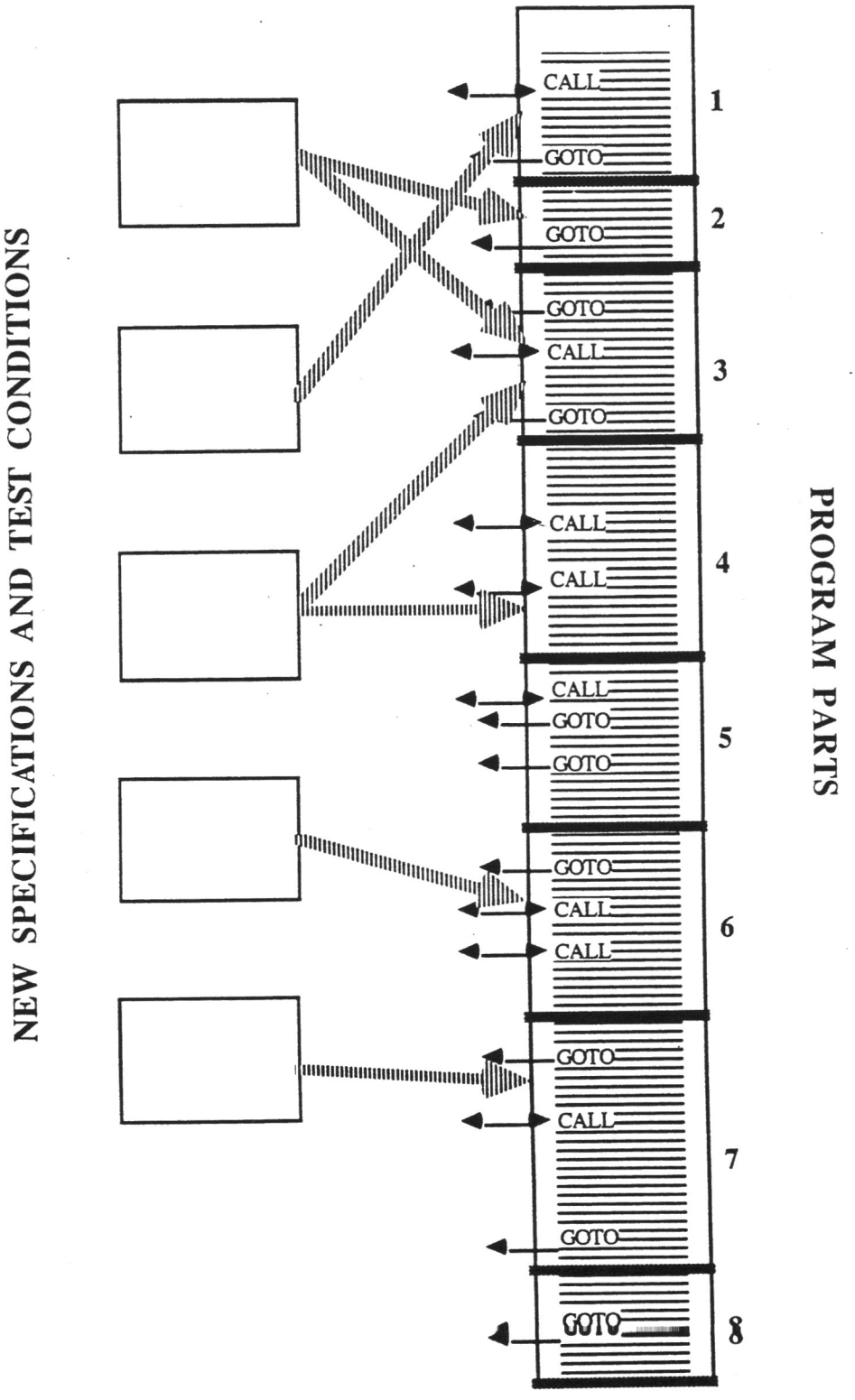
Control Interfaces: External system calls, job streams, etc.

Performance: How well the system works.

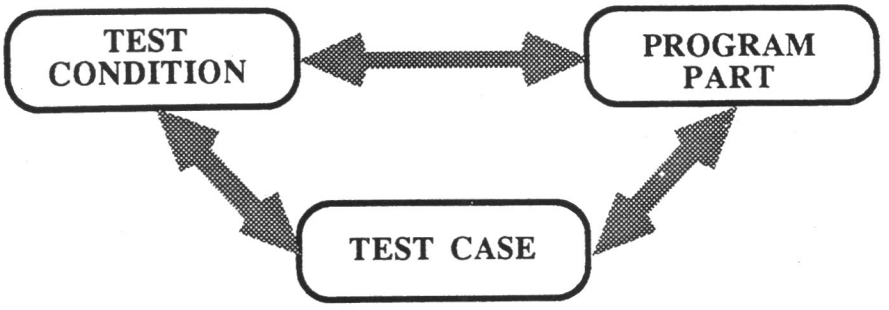
Space: Storage requirements.
Time: Execution time.
Other: For example, response time, output repetition frequency, etc.

Definition of new specifications

Identifying new specifications and test conditions with program parts.



Using an ANALYSIS DATABASE



TEST CONDITION:

- A concise description of what to test. For example, no more than three input values, or the valid range of B is from 101 to 999.
- Essential references: the program parts that handle the test condition, and the test cases that cover the test condition.
- Useful references: the change request or requirement that corresponds to the condition.

PROGRAM PART:

- Identification of the program part in code by line numbers, module name, procedure, etc.
- Essential references: the test conditions that the program part handles, the test cases that invoke the program part.
- Useful references: a functional specification of the program part in the form of a comment.

TEST CASE:

- Input and expected or actual output.
- Essential references: the test conditions that the test case covers, and the program parts that the test case invokes.

Using an ANALYSIS DATABASE (continued)

For new test condtions:

- Add them to the Analysis Database.
- Link them to the proper program parts and test cases when they become known.

For obsolete test conditions:

- Remove them and references to them from the Analysis Database.

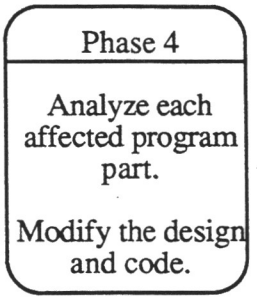
For modified test conditions:

- Use the existing test case to quickly locate affected program parts and test cases.
- Modifiy the test conditions in the Analysis Database and update the references if necessary.

Checklist for Phase 3

1 Are there specs and test conditions for all changes?

2 Are they saved - in the ANALYSIS DATABASE or somewhere?



Information required for Phase 4

- Specs and test conditions for the changes
- Program parts and structures affected by the changes

Results of Phase 4

- Functional specifications of modified program parts (e.g. program comments)
- Modified code of the program parts

Extracting precise specifications from code

- Code tells how it works.
- Specifications should tell what it does.
- Determining a functional specification of a program part "after the fact", i.e. after it has been written.

This can be accomplished in four interrelated steps:

1. Locate and check all data references in the program part.
2. Locate all the entrances and exits of the program part, and trace the flow of control from the entrances to the exits.
3. Formulate a functional specification of the program part.

Then, afterwards,

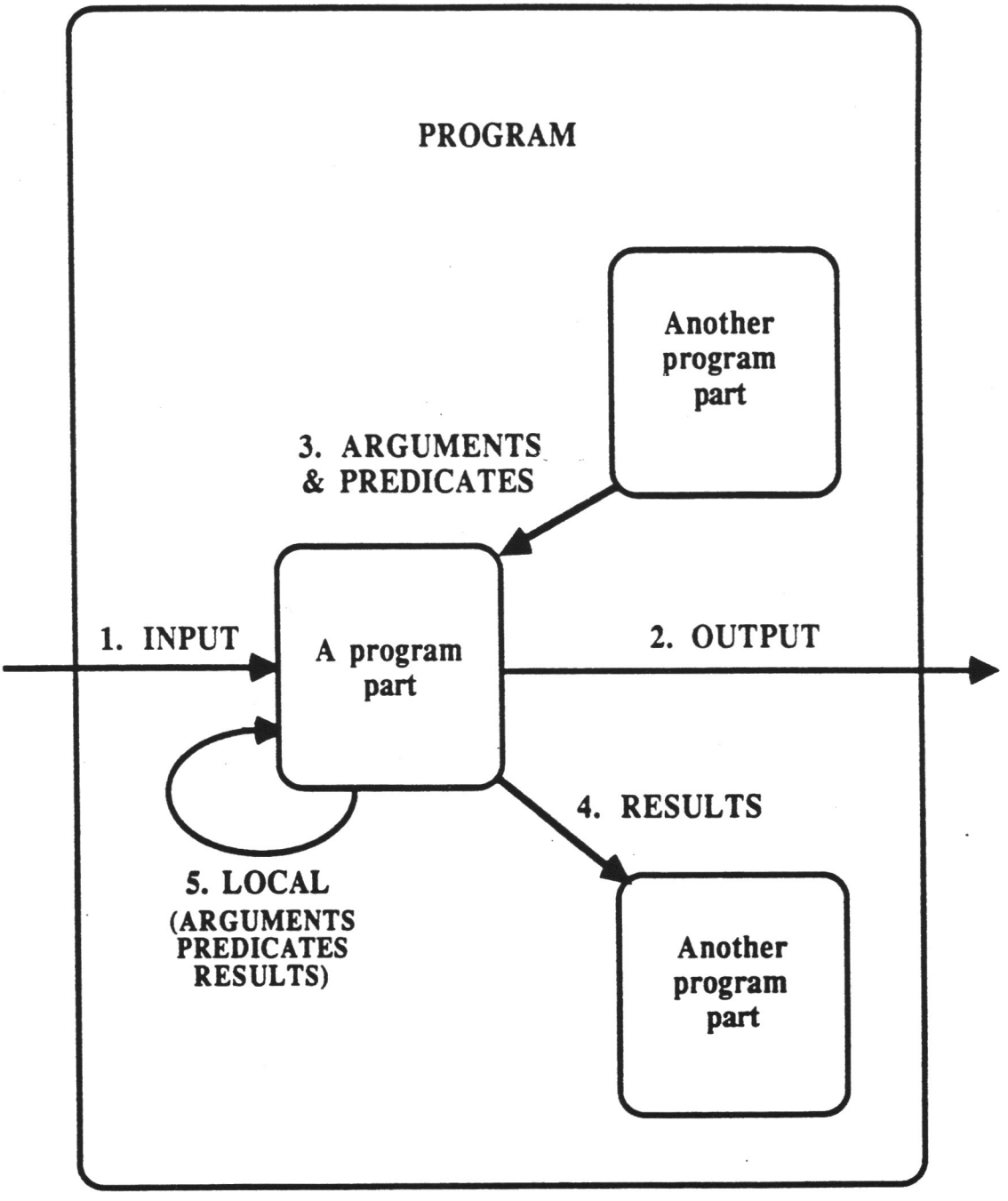
4. Formulate a new functional specification for the program part. This can be followed by design and code modification.

Conditions to deal with in Phase 4.

- Ill-defined program parts: Redefine them to form functions that are easier to define.

Data analysis.

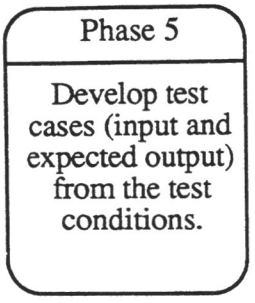
Coupling.



Checklist for Phase 4

1 Specifications of modified program parts

2 Modified code



Information required for Phase 5

- Functional specifications
- Test conditions

Results of Phase 5

- Functional test cases

Test cases from functional test conditions

Guidelines.

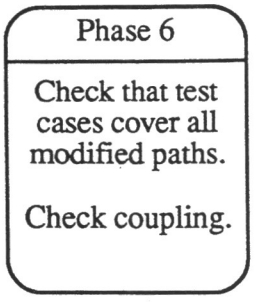
- The objective is to check new or modified functions, program part by program part.
- This can be done with small sets of test inputs and expected outputs. This might not be suitable for volume and stress testing, but this is not volume or stress testing.

Checklist.

- Most likely "normal" cases.
- Most likely error cases.
- Boundary cases.
- Cases indicated by the expression, "Oh, that will never happen".
- Examples suggested by the user or in the user guide.
- Naive user reactions, especially to recover or back-out of a situation.
- Bad data where good data is expected.

Checklist for Phase 5

1 Functional test cases for the changes



Information required for Phase 6

- Modified code
- Functional test cases

Results

- Structural (path) test cases
- Verification of coupling affected by the modified program parts

Structural (path) test cases.

Uses:

- **Detection of unreachable code.**
- **Detection of functions that should not be in the code.**
- **Elimination of redundant cases.** For example, eliminating a case that follows the identical path of another case.
- **Verification of the completeness of functional test cases.**

Checklist for minimal test coverage:

SEQUENCE: Make sure there is a test case that forces action through every modified sequence of code at least once.

SELECTION: Make sure there is a test case that forces action down each branch of a modified selection (e.g. if-then-else, case) at least once.

LOOPS: Make sure there is a test case that exercises each sequence or each test in a modified loop at least once. Loops take on many forms (e.g. perform-until, do-while, a sequence followed by a goto start of loop), but all loops have five aspects that must be checked.

(1) **Initial values.**

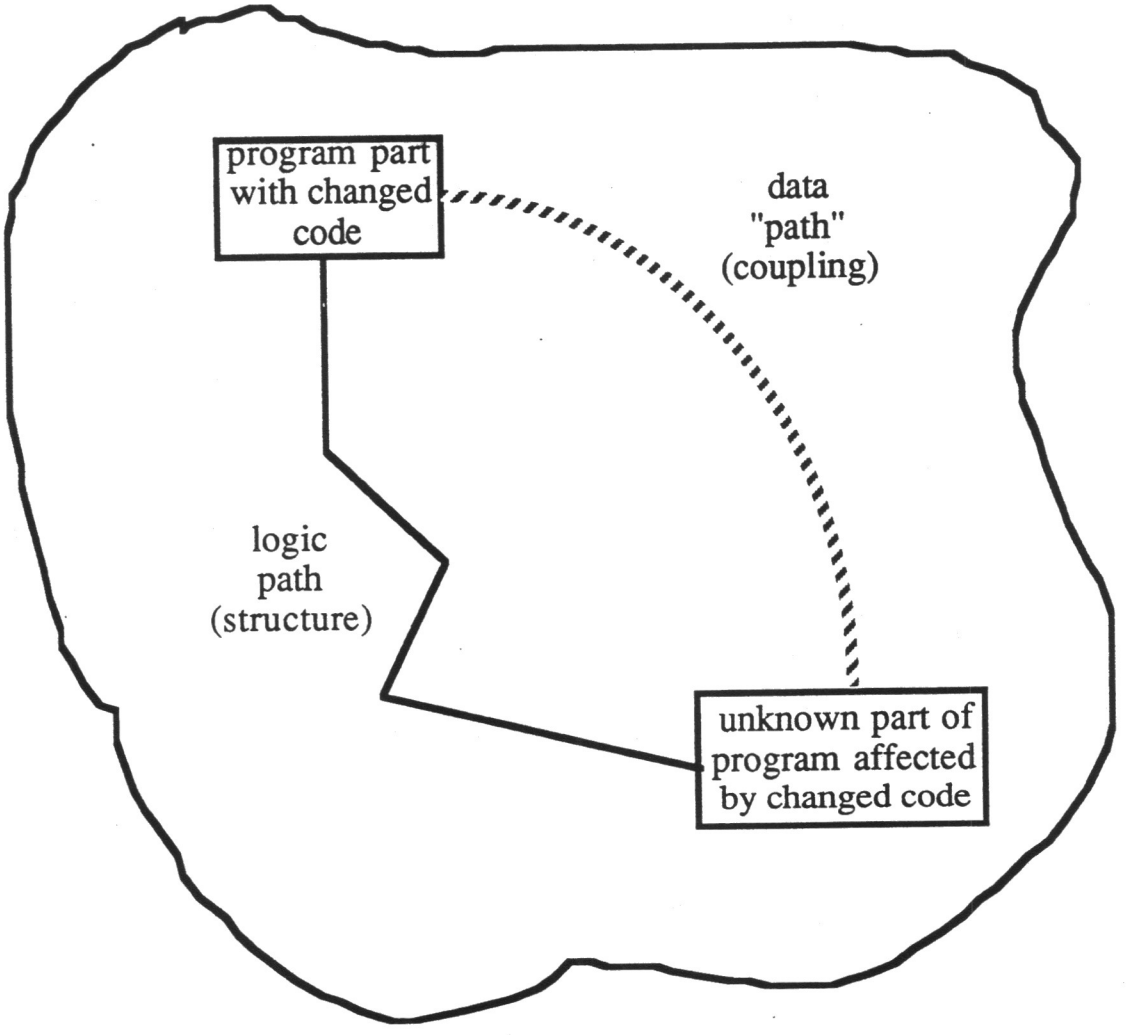
(2) **Entry conditions, all combinations.**

(3) **Loop body itself.**

(4) **Exit conditions, all combinations.**

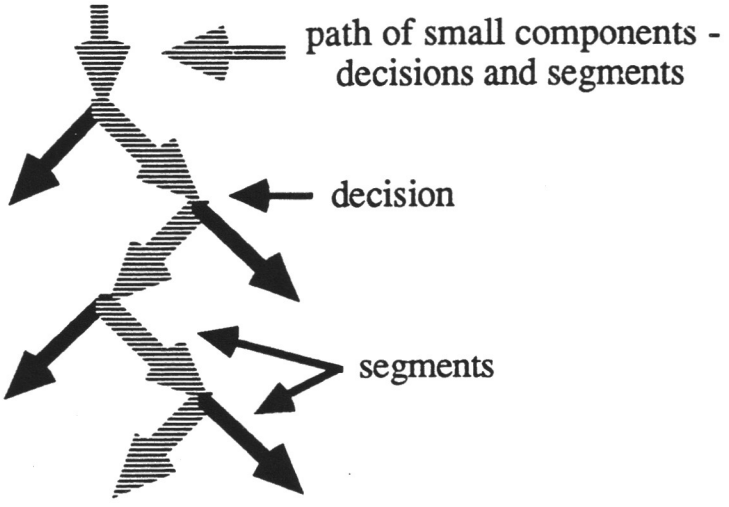
(5) **Final values.**

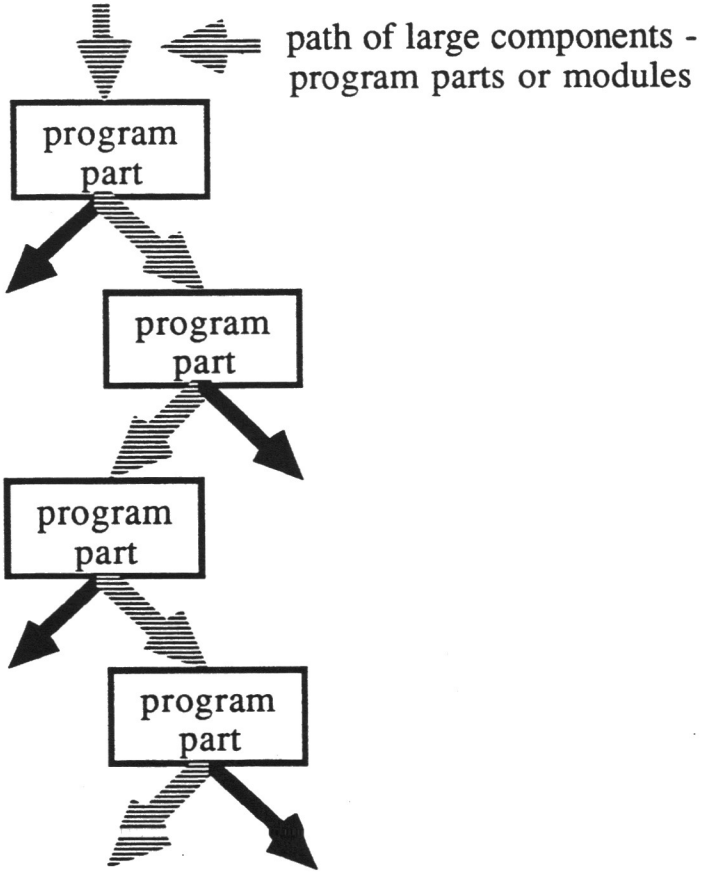
Propagation of side effects



Also - propagation of side effects by combinations of logic and data

Logic paths (structure)





Checklist for Phase 6

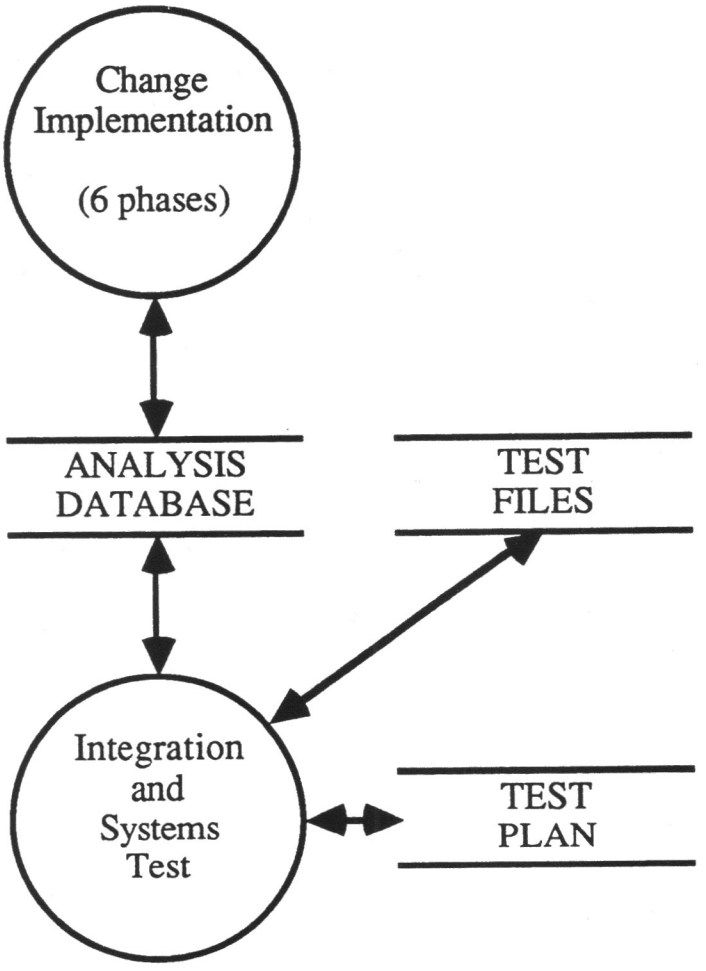
1 Acceptable level of test coverage by the test cases from Phases 5 & 6

2 Correct coupling with the modified program parts

Testing system changes is a team activity

- How does the system behave now compared to prior to the changes?
- Change testing - correct or incorrect according to the change request
- Regression testing - correct or incorrect but not defined by the change request - system functions that should not change
- Test plan
  - Test planning begins in Request Control or Phase 1
  - Test planning begins with objectives and responsibilities
  - Most non-procedural information comes from the six phases (e.g. test conditions, test cases)
  - Unit test "naturally" becomes an extension of Phase 6
  - Procedural aspects of a test plan come from information learned in the six phases, as well as considerations based on schedules, tools, system constraints, type of tests

Making use of previous analysis for testing



Summary of Test Plan contents for system changes

- Objectives
  - Coverage
  - Functional completeness
  - Non-functional attributes (e.g. response time)
- Approach
  - Test sequences
  - Integration sequences
  - Use of intermediate results
- Information from previous work
  - From development (if available):
    structure charts, data dictionaries, module specifications
  - From change analysis, design, implementation (Phases 1-6):
    change requirements
    structural analysis
    cross references
    change specifications
    test conditions
    functional analysis of program parts
    modified code
    test cases

Regression Testing - General Approach

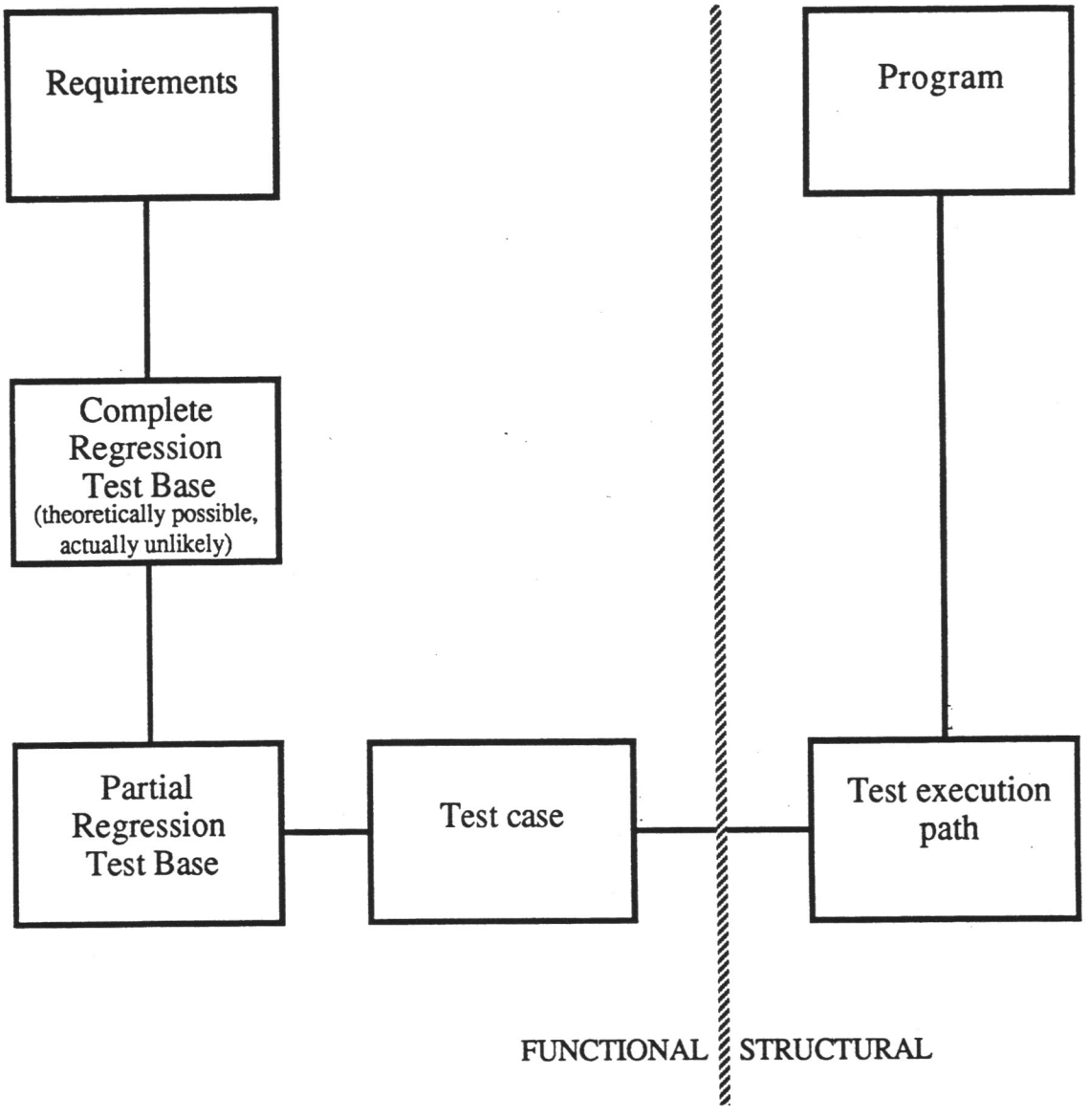
- Test changes first

- If rework is necessary, thoroughly check and test
  - Modified parts (unit level)
  - Interfaces (integration level)
  - Saves time!

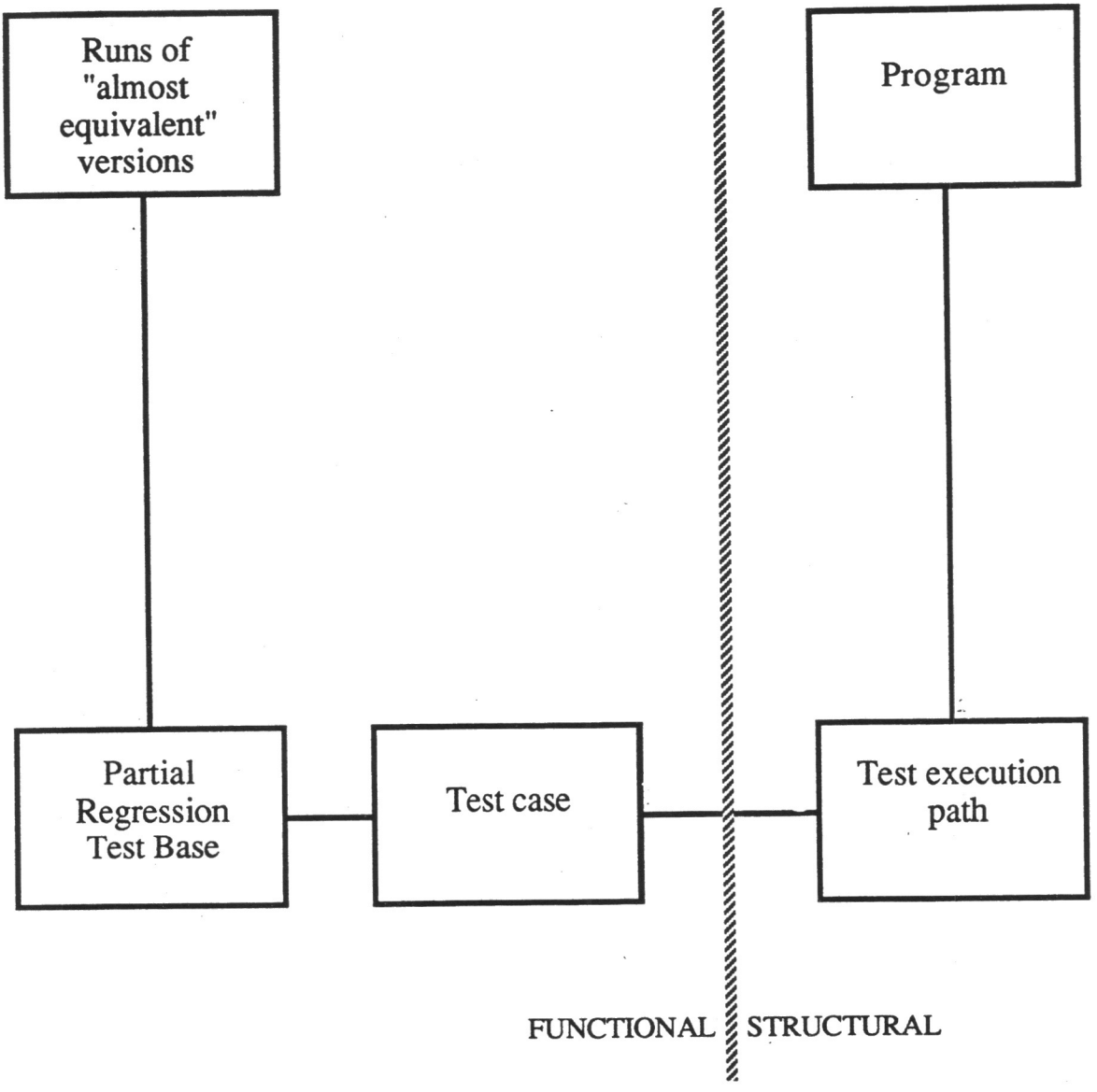
- Define an ACCEPTABLE Regression Test Base
  - Complete Regression Test Base unlikely
  - Existing test files might be acceptable
  - Runs of "almost equivalent" versions might be acceptable
  - Acceptability depends on risk!
  - Running regression tests can reveal unacceptable errors or omissions in the Regression Test Base
  - Update the Regression Test Base
  - This process continues for the life of the application

- Run regression tests and compare results with the Regression Test Base

Regression Test Base relationships



Regression Test Base relationships



Analysis of regression test results

		Comparison of Test Run with Regression Test Base	
		Same	Different
Regression Test Base	Acceptable	Changes cause no apparent side effects.	Changes cause apparent side effects.
	Unacceptable	No apparent side effects. Possible errors or omissions in the Regression Test Base.	Apparent side effects. Possible errors and omissions in the Regression Test Base.

Which of the testing situations, described in the table above, provides the most information about the changed program?

Maintaining the Regression Test Base

- The ideal state of a Regression Test Base
  - Comprehensive, at least for high risk cases
  - Low redundancy
  - Current

- The approach
  - Incremental: add, modify, and delete cases as the function of the program changes and more is learned about the Regression Test Base itself.
  - Use techniques from Phases 1-6, particularly Phase 6 and the ANALYSIS DATABASE, in analyzing errors and omissions.
  - Use capture-replay tools for the Regression Test Base.
  - Consider the Regression Test Base and comparison with regression test runs of the latest version as a more automated way to document actual program behavior.

- Costs of this approach: more time and more effort
- Benefits of this approach: quicker and more accurate changes (less rework)
- Alternatives? - Earlier more thorough analysis
(see Strategies on next page)

Strategies for getting time

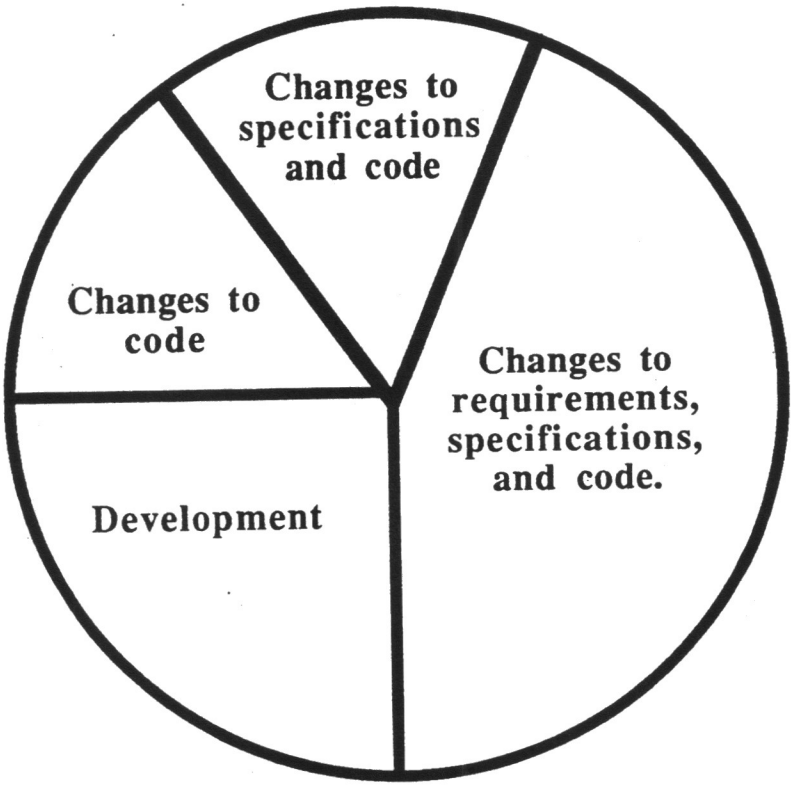
- Keep the Regression Test Base in good order.
- Testing "good" parts of the system often goes well. This, unfortunately, lowers expectations for the time it will take to complete testing.
- Test worst parts and interfaces of the system first!
- Candidates for the worst parts and interfaces

  - Interfaces with new or modified utilities, programs, operating systems,
    communications software, packages, etc.

  - System functions and their interfaces produced
    under internal dissension
    under disorganized project management
    in isolation
    in personal conflicts
    unaccustomed to the working environment
    under severe time pressure
    under management threat

- Perform early, thorough analysis (in the 6 Phases) and test planning.

Cost comparison of software development and maintenance

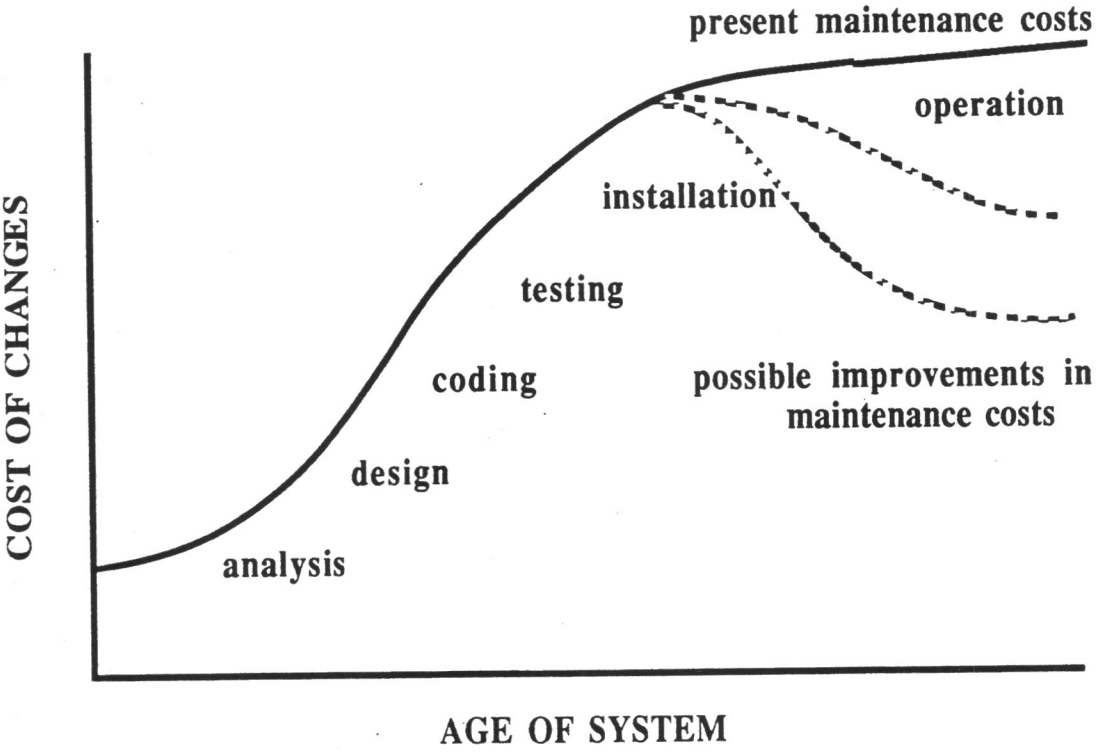


What percentage of maintenance is analysis?

- Reading, interpreting, and discussing requests.
- Examining current inputs, outputs, and problem reports.
- Developing examples (cases) to clarify new requirements.
- Studying programs and other system documentation.
- Discussing features of programs and databases.
- Learning the overall system purposes and specific parts to be changed.
- Examining and discussing why changes are needed.

This could account for at least 80% of maintenance. If maintenance is 75% of all software work, then perhaps 60% of all software work is analysis.

Cost of Changes.



Cost of changes depends on speed and accuracy of analysis and modification and the management of it.

The Second Law of Thermodynamics, which describes a condition of increasing disorder or entropy, has often been used to describe software. This law can be stated in a less abstract way that is self-evident.

Second Law:
Without attention everything becomes a bigger mess.

We use tools, procedures, and our own thought processes to direct our attention in the on-going work of analysis and management of changing systems. Of course, all this requires time and energy which is already covered by the well known First Law of Thermodynamics.

First Law:
You don't get something for nothing.

There are costs and benefits for maintaining and upgrading software processes and products. This is based on the little known Zeroth Law of Thermodynamics.

Zeroth Law:
The only thing that happens is that stuff moves around.