Software Testing and Continuous Quality Improvement

William E. Lewis

AUERBACH

Software Testing and Continuous Quality Improvement

Software Testing and Continuous Quality Improvement

William E. Lewis

Software Testing Associates, Inc.
Plano, Texas

AUERBACH

Boca Raton London New York Washington, D.C.

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

Version Date: 20141217

International Standard Book Number-13: 978-1-4200-4812-4 (eBook - PDF)

Visit the Taylor & Francis Web site at
http://www.taylorandfrancis.com

and the CRC Press Web site at
http://www.crcpress.com

About the Author

William E. Lewis holds a B.A. in mathematics and an M.S. in operations research and has 34 years' experience in the computer industry. Currently, as a senior technology engineer for Technology Builders, Inc., Atlanta, Georgia, he trains and consults in the requirements-based testing area, focusing on leading-edge testing methods and tools.

He is a certified quality analyst (CQA) and certified software test engineer (CSTE) sponsored by the Quality Assurance Institute (QAI), Orlando, Florida. Over the years, he has presented several papers to conferences. In 1998 he presented a paper to QAI's Annual International Information Technology Quality Conference, entitled "Spiral Testing," an excerpt from Section III of this book. He also speaks at meetings of the American Society for Quality and the Association of Information Technology Practitioners.

Mr. Lewis was an assistant director with Ernst & Young, LLP, located in Las Colinas, Texas. He joined E & Y in 1994, authoring the company's software configuration management, software testing, and application evolutionary handbooks, and helping to develop the Navigator/Fusion Methodology application improvement routemaps. He was the quality assurance manager for several application development projects and has extensive experience in test planning, test design, execution, evaluation, reporting, and automated testing. He was also the director of the ISO initiative, which resulted in ISO9000 international certification for Ernst & Young.

Lewis also worked for the Saudi Arabian Oil Company (Aramco) in Jeddah, Saudi Arabia, on an overseas contract assignment as a quality assurance consultant. His duties included full integration and system testing, and he served on the automated tool selection committee and made recommendations to management. He also created software testing standards and procedures.

In 1998 Lewis retired from IBM after 28 years. His jobs included 12 years as a curriculum/course developer and instructor, and numerous years as a system programmer/analyst and performance analyst. An overseas assignment included service in Seoul, Korea, where he was the software engineering curriculum manager for the Korean Advanced Institute of Science and

Technology (KAIST), which is considered the MIT of higher education in Korea. Another assignment was in Toronto, Canada, at IBM Canada's headquarters, where he was responsible for upgrading the corporate education program. He has also traveled throughout the U.S., Rome, Amsterdam, Southampton, Hong Kong, and Sydney, teaching software development and quality assurance classes with a specialty in software testing.

He has also taught at the university level for five years as an adjunct professor. While so engaged he published a five-book series on computer problem solving.

For further information about the training and consulting services provided by Software Testing Associates, Inc., contact:

Software Testing Associates, Inc.
2713 Millington Drive
Plano, Texas 75093
(972) 985-7546

Acknowledgments

I would like to express my sincere gratitude to my loving wife, Carol, for her infinite patience with me in the preparation of this work, and to my mother, Joyce, whom I will never forget.

I thank John Wyzalek, Senior Acquisitions Editor for Auerbach Publishers, for recognizing the importance and potential of this work, and Mike Terkel, a colleague and quality assurance analyst, for reviewing the book, providing invaluable suggestions, and contributing to Section IV, "Modern Testing Tools" and Section VI, "Modern Maintenance Tools."

Finally, I would like to thank the following software testing tool vendors for providing descriptions of their modern testing tools (in alphabetical order): AutoTester, Inc., McCabe and Associates, Mercury Interactive, Rational Software, Sun Microsystems, Inc., and Technology Builders, Inc.

Contents

Introduction xxxiii

SECTION I SOFTWARE QUALITY IN PERSPECTIVE

Part 1 Quality Assurance Framework 3

What is Quality? 3
Prevention vs. Detection 4
Verification vs. Validation 5
Software Quality Assurance 6
Components of Quality Assurance 7
Software Testing 8
Quality Control 9
Software Configuration Management 10
Elements of Software Configuration Management 11
Component Identification 11
Version Control 12
Configuration Building 12
Change Control 13
Software Quality Assurance Plan 14
Steps to Develop and Implement a Software Quality Assurance Plan 14
ISO9000 Quality Standards 18

Part 2 Overview of Testing Techniques 19

Test Case Design 19
Black-Box Testing (functional) 19
White-Box Testing (structural) 20
Gray-Box Testing (functional and structural) 20
Manual vs. Automated Testing 21
Static vs. Dynamic Testing 21
Taxonomy of Software Testing Techniques 22

Part 3 Quality Through a Continuous Improvement Process 29

Contribution of Edward Deming 29
Role of Statistical Methods 30

Deming's 14 Quality Principles 31
Point 1: Create Constancy of Purpose 31
Point 2: Adopt the New Philosophy 31
Point 3: Cease Dependence on Mass Inspection 32
Point 4: End the Practice of Awarding Business on Price Tag Alone 32
Point 5: Improve Constantly and Forever the System of Production and Service 33
Point 6: Institute Training and Retraining 33
Point 7: Institute Leadership 33
Point 8: Drive Out Fear 34
Point 9: Break Down Barriers Between Staff Areas 34
Point 10: Eliminate Slogans, Exhortations, and Targets for the Workforce 35
Point 11: Eliminate Numerical Goals 35
Point 12: Remove Barriers to Pride of Workmanship 35
Point 13: Institute a Vigorous Program of Education and Retraining 36
Point 14: Take Action to Accomplish the Transformation 36
Continuous Improvement Through the Plan, Do, Check, Act Process 36
Going Around the PDCA Circle 38

SECTION II LIFE CYCLE TESTING REVIEW

Part 4 Overview 41

Waterfall Development Methodology. 41
Continuous Improvement "Phased" Approach 41
Psychology of Life Cycle Testing 41
Software Testing as a Continuous Improvement Process 43
The Testing Bible: Software Test Plan 46
Major Steps to Develop a Test Plan 47
Components of a Test Plan 49
Technical Reviews as a Continuous Improvement Process 50
Motivation for Technical Reviews 53
Types of Reviews 53
Structured Walkthroughs 54
Inspections 54
Participant Roles 57
Steps for an Effective Review 57

Part 5 Verifying the Requirements Phase 61

Testing the Requirements with Technical Reviews 62
Inspections and Walkthroughs 63

Checklists . . . 63
Methodology Checklist . . . 63
Requirements Traceability Matrix . . . 63
Building the System/Acceptance Test Plan . . . 64

Part 6 Verifying the Logical Design Phase . . . 67
Data Model, Process Model, and the Linkage . . . 67
Testing the Logical Design with Technical Reviews . . . 68
Refining the System/Acceptance Test Plan . . . 70

Part 7 Verifying the Physical Design Phase . . . 73
Testing the Physical Design with Technical Reviews . . . 73
Creating Integration Test Cases . . . 75
Methodology for Integration Testing . . . 75

Part 8 Verifying the Program Unit Design Phase . . . 77
Testing the Program Unit Design with Technical Reviews . . . 77
Sequence . . . 77
Selection . . . 77
Iteration . . . 78
Creating Unit Test Cases . . . 78

Part 9 Verifying the Coding Phase . . . 81
Testing Coding with Technical Reviews . . . 81
Executing the Test Plan . . . 82
Unit Testing . . . 82
Integration Testing . . . 83
System Testing . . . 84
Acceptance Testing . . . 84
Defect Recording . . . 85

SECTION III CLIENT/SERVER AND INTERNET TESTING METHODOLOGY

Part 10 Development Methodology Overview . . . 89
Limitations of Life Cycle Development . . . 89
The Client/Server Challenge . . . 90
Psychology of Client/Server Spiral Testing . . . 91
The New School of Thought . . . 91
Tester/Developer Perceptions . . . 92
Project Goal: Integrate QA and Development . . . 93
Iterative/Spiral Development Methodology . . . 94
Role of JADs . . . 96

Role of Prototyping 97
Methodology for Developing Prototypes 98
Continuous Improvement "Spiral" Testing Approach 102

Part 11 Information Gathering (Plan) 107

Step 1: Prepare for the Interview 107
Task 1: Identify the Participants 107
Task 2: Define the Agenda 108
Step 2: Conduct the Interview 108
Task 1: Understand the Project 110
Task 2: Understand the Project Objectives 111
Task 3: Understand the Project Status 112
Task 4: Understand the Project Plans 112
Task 5: Understand the Project Development Methodology 113
Task 6: Identify the High-Level Business Requirements 113
Task 7: Perform Risk Analysis 114
Computer Risk Analysis 115
Method 1 — Judgment and Instinct 115
Method 2 — Dollar Estimation 115
Method 3 — Identifying and Weighting Risk Attributes 116
Step 3: Summarize the Findings 117
Task 1: Summarize the Interview. 117
Task 2: Confirm the Interview Findings 117

Part 12 Test Planning (Plan) 119

Step 1: Build a Test Plan 121
Task 1: Prepare an Introduction 121
Task 2: Define the High-Level Functional Requirements (In Scope) 121
Task 3: Identify Manual/Automated Test Types 122
Task 4: Identify the Test Exit Criteria 123
Task 5: Establish Regression Test Strategy 124
Task 6: Define the Test Deliverables 125
Task 7: Organize the Test Team 127
Task 8: Establish a Test Environment 129
Task 9: Define the Dependencies 129
Task 10: Create a Test Schedule. 130
Task 11: Select the Test Tools 133
Task 12: Establish Defect Recording/Tracking Procedures 133
Task 13: Establish Change Request Procedures 134
Task 14: Establish Version Control Procedures 136
Task 15: Define Configuration Build Procedures 137
Task 16: Define Project Issue Resolution Procedures 138

Task 17: Establish Reporting Procedures 138
Task 18: Define Approval Procedures 139
Step 2: Define the Metric Objectives 139
Task 1: Define the Metrics 140
Task 2: Define the Metric Points 141
Step 3: Review/Approve the Plan 141
Task 1: Schedule/Conduct the Review 141
Task 2: Obtain Approvals 141

Part 13 Test Case Design (Do) 147

Step 1: Design Function Tests 147
Task 1: Refine the Functional Test Requirements 147
Task 2: Build a Function/Test Matrix 148
Step 2: Design GUI Tests 155
Task 1: Identify the Application GUI Components 155
Task 2: Define the GUI Tests 156
Step 3: Define the System/Acceptance Tests 157
Task 1: Identify Potential System Tests 157
Task 2: Design System Fragment Tests 159
Task 3: Identify Potential Acceptance Tests 159
Step 4: Review/Approve Design 161
Task 1: Schedule/Prepare for Review 161
Task 2: Obtain Approvals 161

Part 14 Test Development (Do) 163

Step 1: Develop Test Scripts 163
Task 1: Script the Manual/Automated GUI/Function Tests 163
Task 2: Script the Manual/Automated System Fragment Tests 163
Step 2: Review/Approve Test Development 164
Task 1: Schedule/Prepare for Review 164
Task 2: Obtain Approvals 165

Part 15 Test Execution/Evaluation (Do/Check) 167

Step 1: Setup and Testing 167
Task 1: Regression Test the Manual/Automated Spiral Fixes . . . 167
Task 2: Execute the Manual/Automated New Spiral Tests 169
Task 3: Document the Spiral Test Defects 169
Step 2: Evaluation 169
Task 1: Analyze the Metrics 169
Task 2: Refine the Test Schedule 170
Task 3: Identify Requirement Changes 171

Part 16 Prepare for the Next Spiral (Act) 173
Step 1: Refine the Tests 173
Task 1: Update the Function/GUI Tests 173
Task 2: Update the System Fragment Tests 175
Task 3: Update the Acceptance Tests 175
Step 2: Reassess the Team, Procedures, and Test Environment . . . 175
Task 1: Evaluate the Test Team 175
Task 2: Review the Test Control Procedures 176
Task 3: Update the Test Environment 177
Step 3: Publish Interim Test Report 177
Task 1: Publish the Metric Graphics 177

Part 17 Conduct the System Test 181
Step 1: Complete System Test Plan 181
Task 1: Finalize the System Test Types 181
Task 2: Finalize System Test Schedule 182
Task 3: Organize the System Test Team 182
Task 4: Establish the System Test Environment 182
Task 5: Install the System Test Tools 186
Step 2: Complete System Test Cases 187
Task 1: Design/Script the Performance Tests 187
Task 2: Design/Script the Security Tests 189
Task 3: Design/Script the Volume Tests 191
Task 4: Design/Script the Stress Tests 191
Task 5: Design/Script the Compatibility Tests 192
Task 6: Design/Script the Conversion Tests 193
Task 7: Design/Script the Usability Tests 194
Task 8: Design/Script the Documentation Tests 194
Task 9: Design/Script the Backup Tests 195
Task 10: Design/Script the Recovery Tests 195
Task 11: Design/Script the Installation Tests 196
Task 12: Design/Script Other System Test Types 197
Step 3: Review/Approve System Tests 198
Task 1: Schedule/Conduct the Review 198
Task 2: Obtain Approvals 199
Step 4: Execute the System Tests 199
Task 1: Regression Test the System Fixes 199
Task 2: Execute the New System Tests 199
Task 3: Document the System Defects 200

Part 18 Conduct Acceptance Testing 201
Step 1: Complete Acceptance Test Planning 201
Task 1: Finalize the Acceptance Test Types 201
Task 2: Finalize the Acceptance Test Schedule 201

Task 3: Organize the Acceptance Test Team 203
Task 4: Establish the Acceptance Test Environment 203
Task 5: Install Acceptance Test Tools 204
Step 2: Complete Acceptance Test Cases 204
Task 1: Subset the System-Level Test Cases. 205
Task 2: Design/Script Additional Acceptance Tests 205
Step 3: Review/Approve Acceptance Test Plan 205
Task 1: Schedule/Conduct the Review 205
Task 2: Obtain Approvals 206
Step 4: Execute the Acceptance Tests 206
Task 1: Regression Test the Acceptance Fixes. 206
Task 2: Execute the New Acceptance Tests 207
Task 3: Document the Acceptance Defects 207

Part 19 Summarize/Report Spiral Test Results 209

Step 1: Perform Data Reduction 209
Task 1: Ensure All Tests Were Executed/Resolved 209
Task 2: Consolidate Test Defects by Test Number 209
Task 3: Post Remaining Defects to a Matrix 209
Step 2: Prepare Final Test Report 209
Task 1: Prepare the Project Overview 210
Task 2: Summarize the Test Activities. 210
Task 3: Analyze/Create Metric Graphics 211
Task 4: Develop Findings/Recommendations 216
Step 3: Review/Approve the Final Test Report 219
Task 1: Schedule/Conduct the Review 219
Task 2: Obtain Approvals 220
Task 3: Publish the Final Test Report 221

SECTION IV MODERN TESTING TOOLS

Part 20 Introduction to Testing Tools 227

Justifying Testing Tools 227
When to Consider Using a Testing Tool 227
When to *Not* Consider Using a Testing Tool 228
Testing Tool Selection Checklist. 228
Types of Testing Tools 230
Year 2000 Tools 230
Web Site Management Tools 230
Requirements-Based Testing Tools 231
Test Management Tools 231
Regression Testing Tools 231
Coverage Analysis Tools 231
Dynamic Testing Tools 232
Static Testing Tools 232

Load Testing Tools 233
Comparators 233
Vendor Tool Descriptions 233
McCabe's Visual 2000 233
McCabe's Visual Testing ToolSet 238
McCabe's Visual Quality ToolSet 239
McCabe's Visual Reengineering ToolSet 240
Rational's SQA Suite (Version 6.1) 241
Rational's SQA SiteCheck Tool 242
Rational's SQA Robot (Version 6.1) 243
Rational's SQA Manager (Version 6.1) 244
Rational's SQA Load Test (Version 6.1) 246
Rational's Visual Test (Version 4.0r) 247
Rational's preVue Tool 248
Rational's PureCoverage Tool 250
Rational's Purify Tool 252
Rational's Quantify Tool 253
Technology Builders' Caliber-RBT 254
AutoTester's Test Library Manager 257
AutoTester's Test Station Tool 259
Mercury Interactive's TestDirector Test Management Tool 261
Mercury Interactive's WinRunner Functional Testing Tool for Windows 263
Mercury Interactive's XRunner Functional Testing Tool for UNIX 265
Mercury Interactive's LoadRunner Load/Stress Testing Tool 268
Mercury Interactive's Astra Site Manager Web Site Management Tool 270

Part 21 Methodology to Evaluate Testing Tools 273

Step 1: Define Test Goals 273
Step 2: Set Tool Objectives 273
Step 3A: Conduct Selection Activities for Informal Procurement 274
Task 1: Develop the Acquisition Plan 274
Task 2: Define Selection Criteria 274
Task 3: Identify Candidate Tools 274
Task 4: Conduct the Candidate Review 275
Task 5: Score the Candidates 275
Task 6: Select the Tool 275
Step 3B: Conduct Selection Activities for Formal Procurement 276
Task 1: Develop the Acquisition Plan 276

Task 2: Create the Technical Requirements Document 276
Task 3: Review Requirements. 276
Task 4: Generate the Request for Proposal 276
Task 5: Solicit Proposals 277
Task 6: Perform the Technical Evaluation 277
Task 7: Select a Tool Source 277
Step 4: Procure the Testing Tool. 277
Step 5: Create the Evaluation Plan 278
Step 6: Create the Tool Manager's Plan 278
Step 7: Create the Training Plan 278
Step 8: Receive the Tool 279
Step 9: Perform the Acceptance Test. 279
Step 10: Conduct Orientation 279
Step 11: Implement Modifications 279
Step 12: Train Tool Users. 280
Step 13: Use the Tool in the Operating Environment. 280
Step 14: Write the Evaluation Report. 280
Step 15: Determine Whether Goals Have Been Met 281

SECTION V TESTING IN THE MAINTENANCE ENVIRONMENT

Part 22 Overview of Software Maintenance 285

Balancing Three Conflicting Forces 285
Software Candidates for Redesign 286
Frequent System Failures 287
Code More than Five Years Old 287
Overly Complex Program Structure and Logic Flow. 288
Code Written for Previous Generation Hardware 288
Systems Running in Emulation Mode 288
Very Large Modules or Unit Subroutines 289
Excessive Resource Requirements 289
Hard-Coded Parameters that Are Subject to Change 289
Low-Level Language Support 289
Seriously Deficient Documentation. 290
Turnover of Personnel 290
Missing or Incomplete Design Specifications. 290
Obsolete Technologies 290
Extent of Testing vs. Size of Change 291
Inadequate Test Documentation 291
Types of Software Changes 291
Perfective Maintenance 291
Controlling Perfective Maintenance 292
Adaptive Maintenance 294
Controlling Adaptive Maintenance 295
Corrective Maintenance 295

Design Errors Resulting from Incomplete or Faulty Design 296
Logic Errors 296
Coding Errors 296
Computational Errors 296
Input/Output Errors 296
Data Handling Errors 296
Interface Errors 296
Data Definition Errors 296
Controlling Corrective Maintenance 296
Strategies for Managing Maintenance 297
Continue the Maintenance Development Test Strategies 297
Establish Automated Regression Testing Approach 298
Establish a Change Request System 298
Establish a Software Configuration Management System 299
Establish a Maintenance "Triage Approach" 299
Release Management Approach 299
Emergency and Intermediate Repair Approaches 300
Psychology of Maintenance Testing 301
Phased Life Cycle Maintenance 301
RAD Maintenance 302
A Continuous Improvement Maintenance Approach 304
Going Around the PDCA Circle 306
Maintenance Testing Methodology 307

Part 23 Enhancement/Defect Requirements Analysis (Plan) 309

Step 1: Prepare for the Maintenance Review 309
Task 1: Identify the Participants 309
Task 2: Define the Agenda 310
Step 2: Conduct the Maintenance Review 312
Task 1: Define Enhancement Opportunities 312
Task 2: Define High-Level Functional Changes 315
Task 3: Define High-Level Data Changes 316
Task 4: Define High-Level GUI Interface Changes 317
Task 5: Perform Impact Analysis 317
Step 3: Summarize the Findings 318
Task 1: Summarize the Maintenance Review 318
Task 2: Confirm the Maintenance Review Findings 319

Part 24 Preliminary Maintenance Test Planning (Plan) 321

Step 1: Build Preliminary Maintenance Test Plan 321
Task 1: Prepare Test Plan Outline 323
Task 2: Prepare an Introduction 324

Task 3: Define the High-Level Functional Changes 324
Task 4: Define the Maintenance Test Deliverables. 325
Task 5: Establish Defect Recording/Tracking Procedures 326
Task 6: Establish Change Request Procedures. 328
Task 7: Establish Version Control Procedures 330
Task 8: Define Configuration Build Procedures 330
Task 9: Define Project Issue Resolution Procedures 331
Task 10: Establish Reporting Procedures 331
Task 11: Establish Approval Procedures. 332
Step 2: Review the Preliminary Maintenance Test Plan. 333
Task 1: Schedule/Conduct the Review 333
Task 2: Obtain Approvals 333

Part 25 Enhancement Prototype Design (Plan) 335
Step 1: Prepare for the Design Session 335
Task 1: Identify the Design Participants 335
Task 2: Define the Agenda. 336
Step 2: Conduct the Design Session 337
Task 1: Design Functional Changes 337
Task 2: Design Data Changes 337
Task 3: Design GUI Interface Changes 338
Step 3: Review/Approve Maintenance Design. 339
Task 1: Schedule/Conduct Design Review 339
Task 2: Obtain Approvals 340

Part 26 Completed Maintenance Test Planning (Plan). 341
Step 1: Finalize the Maintenance Test Plan 341
Task 1: Identify Manual/Automated Test Types. 341
Task 2: Identify the Test Exit Criteria 343
Task 3: Establish Maintenance Regression Test Strategy 345
Task 4: Organize the Maintenance Test Team 347
Task 5: Establish the Maintenance Test Environment. 348
Task 6: Define the Dependencies 349
Task 7: Create a Maintenance Test Schedule 349
Task 8: Select the Maintenance Test Tools 352
Step 2: Define the Maintenance Metric Objectives. 353
Task 1: Define the Metrics. 353
Task 2: Define the Metric Points. 354
Step 3: Review/Approve the Maintenance Test Plan 354
Task 1: Schedule/Conduct the Review 354
Task 2: Obtain Approvals 360

Part 27 Maintenance Test Case Design (Do) 361
Step 1: Design Function Tests 361

Task 1: Refine the Functional Test Requirements 361
Task 2: Build/Modify Function/Test Matrix 361
Step 2: Design GUI Tests 369
Ten Guidelines for Good GUI Design 369
Task 1: Define the Application GUI Components 369
Windows 369
Menus 370
Forms 370
Icons 370
Controls 370
Task 2: Define the GUI Tests 370
Step 3: Define the System/Acceptance Tests 372
Task 1: Identify Potential System Tests 372
Task 2: Design System Fragment Tests 373
Task 3: Identify Potential Acceptance Tests 374
Step 4: Review/Approve Design 375
Task 1: Schedule/Prepare for Review 375
Task 2: Obtain Approvals 375

Part 28 Maintenance Test Development (Do) 377

Step 1: Develop Test Scripts 377
Task 1: Script the Manual/Automated GUI Function Tests 377
Task 2: Script the System Fragment Tests 377
Step 2: Review/Approve Test Maintenance Development 379
Task 1: Schedule/Prepare for Review 379
Task 2: Obtain Approvals 379

Part 29 Maintenance Test Execution/Evaluation (Do/Check) 381

Step 1: Setup and Testing 381
Task 1: Regression Test the Maintenance Fixes 381
Task 2: Execute the New Maintenance Tests 383
Task 3: Record the Maintenance Defects 383
Step 2: Evaluation 383
Task 1: Analyze the Metrics 383
Task 2: Refine the Test Schedule 384
Task 3: Identify Requirement Changes 384

Part 30 Prepare for the Next Test Cycle (Act) 387

Step 1: Refine the Tests 387
Task 1: Update the Function/GUI Tests 387
Task 2: Update the System Fragment Tests 387
Task 3: Update the Acceptance Tests 388
Step 2: Reassess the Team, Procedures, and Test Environment . . . 389
Task 1: Evaluate the Test Team 389

Task 2: Review the Test Control Procedures . . . 389
Task 3: Update the Test Environment . . . 390
Step 3: Publish Interim Test Report . . . 391
Task 1: Publish the Metric Graphics . . . 391
Test Case Execution Status . . . 391
Defect Gap Analysis . . . 392
Defect Severity Status . . . 392
Test Burnout Tracking . . . 392
Test Exit Tracking . . . 392

Part 31 Maintenance System Testing . . . 395

Part 32 Maintenance Acceptance Testing . . . 397

Part 33 Maintenance Summary Test Report . . . 399

Part 34 System Installation . . . 401

Step 1: Develop a Recovery Plan . . . 401
Step 2: Place Maintenance Changes into Production . . . 401
Step 3: Delete Old Production System . . . 402
Step 4: Operate and Monitor New System . . . 402
Step 5: Document System Problems . . . 403

SECTION VI MODERN MAINTENANCE TOOLS

Part 35 Introduction to Maintenance Tools . . . 407

Maintenance Testing Maintenance Tool Selection Checklist . . . 407
Types of Maintenance Tools . . . 410
Code Complexity Tools . . . 410
Regression Testing Tools . . . 410
Load and Performance Maintenance Tools . . . 410
Problem Management Tools . . . 413
Configuration/Process Management Tools . . . 413
Web Site Test Tools . . . 414
Code Coverage Tools . . . 414
Runtime Error Analysis Tools . . . 414
Cross-Reference Maintenance Tools . . . 416
Comparator Maintenance Tools . . . 418
Diagnostic Routines . . . 421
Application Utility Libraries . . . 429
Online Documentation Libraries . . . 429
Online/Interactive Change and Debugging Facilities . . . 430
Vendor Tool Descriptions . . . 430
McCabe's McCabe QA . . . 430
SunTest's JavaStar 1.2 . . . 431
Rational Robot . . . 433
Rational TestFactory . . . 434

AutoTester's Test Station Tool 435
Mercury's WinRunner Tool 5 437
Mercury's XRunner (Unix) Tool 4 439
SunTest's JavaLoad 442
Mercury's LoadRunner Tool 444
Rational's preVue Tool. 447
Rational's PerformanceStudio 449
Rational's Visual Quantify Tool 450
Rational's ClearDDTS. 451
Rational's ClearQuest 452
Rational's ClearCase 453
Rational's ClearCase MultiSite 453
Rational's SQA SiteCheck Tool 454
Rational Visual PureCoverage 6.0 455
Rational Purify 6.0 456
Other Maintenance Tools 457

APPENDIXES

Appendix A Spiral Testing Methodology. 459

Appendix B Software Quality Assurance Plan. 467

Appendix C Requirements Specification. 469

Appendix D Change Request Form 471

Appendix E Test Templates 473

E1: Unit Test Plan 473
E2: System/Acceptance Test Plan. 473
E3: Requirements Traceability Matrix 475
E4: Test Plan (Client/Server and Internet Spiral Testing) 475
E5: Function/Test Matrix 478
E6: GUI Component Test Matrix (Client/Server and Internet Spiral Testing) 478
E7: GUI-Based Functional Test Matrix (Client/Server and Internet Spiral Testing) 478
E8: Test Case 482
E9: Test Case Log 483
E10: Test Log Summary Report. 483
E11: System Summary Report. 483
E12: Defect Report 485
E13: Test Schedule 485
E14: Retest Matrix. 485
E15: Spiral Testing Summary Report (Client/Server and Internet Spiral Testing) 489

Appendix F Checklists 493
F1: Requirements Phase Defect Checklist 493
F2: Logical Design Phase Defect Checklist 493
F3: Physical Design Phase Defect Checklist 493
F4: Program Unit Design Phase Defect Checklist 497
F5: Coding Phase Defect Checklist 498
F6: Field Testing Checklist 500
F7: Record Testing Checklist 500
F8: File Test Checklist 503
F9: Error Testing Checklist 504
F10: Use Test Checklist 506
F11: Search Test Checklist 507
F12: Match/Merge Checklist 508
F13: Stress Test Checklist 509
F14: Attributes Testing Checklist 510
F15: States Testing Checklist 511
F16: Procedures Testing Checklist 512
F17: Control Testing Checklist 512
F18: Control Flow Testing Checklist 517
F19: Testing Tool Selection Checklist 517

Appendix G Integrating Testing into Development Methodology 519
Step 1: Organize the Test Team 520
Step 2: Identify Test Steps and Tasks to Integrate 520
Step 3: Customize Test Steps and Tasks 521
Step 4: Select Integration Points 521
Step 5: Modify the Development Methodology 522
Step 6: Incorporate Defect Recording 522
Step 7: Train in Use of the Test Methodology 522

Appendix H Software Testing Techniques 525
H1: Basis Path Testing 525
H2: Black Box Testing 526
H3: Bottom-Up Testing 527
H4: Boundary Value Testing 527
H5: Branch Coverage Testing 529
H6: Branch/Condition Coverage Testing 530
H7: Cause-Effect Graphing 530
H8: Condition Coverage 534
H9: CRUD Testing 535
H10: Database Testing 536

H11: Decision Tables 567
H12: Desk Checking 567
H13: Equivalence Partitioning. 567
H14: Exception Testing 569
H15: Free Form Testing 569
H16: Gray-Box Testing 570
H17: Histograms 571
H18: Inspections 571
H19: JADs. 572
H20: Orthogonal Array Testing. 573
H21: Pareto Analysis 574
H22: Positive and Negative Testing 576
H23: Prior Defect History Testing. 577
H24: Prototyping. 577
H25: Random Testing 582
H26: Range Testing. 583
H27: Regression Testing. 583
H28: Risk-Based Testing. 585
H29: Run Charts 585
H30: Sandwich Testing. 586
H31: Statement Coverage Testing. 586
H32: State Transition Testing 587
H33: Statistical Profile Testing 588
H34: Structured Walkthroughs 588
H35: Syntax Testing 589
H36: Table Testing 589
H37: Thread Testing. 590
H38: Top-Down Testing 590
H39: White-Box Testing 591

Glossary 593

Bibliography 599

Index. 605

EXHIBITS

Part 1

Exhibit 1 Quality Assurance Components. 8
Exhibit 2 Software Configuration Management 11
Exhibit 3 Companion ISO Standards 18

Part 2

Exhibit 1 Testing Technique Categories 23
Exhibit 2 Testing Technique Descriptions 25

Part 3

Exhibit 1 *The Deming Quality Circle* . . . 37
Exhibit 2 *The Ascending Spiral* . . . 38

Part 4

Exhibit 1 *Waterfall Development Methodology* . . . 42
Exhibit 2 *Development Phases vs. Testing Types* . . . 44
Exhibit 3 Test Case Form . . . 48
Exhibit 4 System/Acceptance Test Plan vs. Phase . . . 51
Exhibit 5 PDCA Process and Inspections . . . 55
Exhibit 6 *Phased Inspections as an Ascending Spiral* . . . 56

Part 5

Exhibit 1 *Requirements Phase and Acceptance Testing* . . . 62
Exhibit 2 Requirements Phase Defect Recording . . . 64
Exhibit 3 *Requirements/Test Matrix* . . . 66

Part 6

Exhibit 1 *CRUD Matrix* . . . 69
Exhibit 2 Logical Design Phase Defect Recording . . . 70

Part 7

Exhibit 1 Physical Design Phase Defect Recording . . . 74

Part 8

Exhibit 1 Program Unit Design Phase Defect Recording . . . 78

Part 9

Exhibit 1 Coding Phase Defect Recording . . . 82
Exhibit 2 *Executing the Tests* . . . 83

Part 10

Exhibit 1 *Spiral Testing Process* . . . 95
Exhibit 2 *Spiral Testing and Continuous Improvement* . . . 103
Exhibit 3 *Spiral Testing Methodology* . . . 105

Part 11

Exhibit 1 *Spiral Testing and Continuous Improvement* . . . 108
Exhibit 2 *Information Gathering (Steps/Tasks)* . . . 109
Exhibit 3 Interview Agenda . . . 110
Exhibit 4 Identifying and Weighting Risk Attributes . . . 117

Part 12

Exhibit 1 *Test Planning (Steps/Tasks)* 120
Exhibit 2 High-Level Business Functions 122
Exhibit 3 Functional Window Structure 122
Exhibit 4 Retest Matrix 126
Exhibit 5 Test Schedule. 130
Exhibit 6 Project Milestones 132
Exhibit 7 Defect States 134
Exhibit 8 Deliverable Approvals 139
Exhibit 9 Metric Points 142

Part 13

Exhibit 1 *Spiral Testing and Continuous Improvement* 148
Exhibit 2 *Test Design (Steps/Tasks)* 149
Exhibit 3 Functional Breakdown 150
Exhibit 4 Functional Window Structure 151
Exhibit 5 Functional/Test Matrix 154
Exhibit 6 GUI Component Test Matrix 156
Exhibit 7 GUI Component Checklist 157
Exhibit 8 Baseline Performance Measurements 160

Part 14

Exhibit 1 *Test Development (Steps/Tasks)* 164
Exhibit 2 Function/GUI Test Script 165

Part 15

Exhibit 1 *Spiral Testing and Continuous Improvement* 168
Exhibit 2 *Test Execution/Evaluation (Steps/Tasks)* 168

Part 16

Exhibit 1 *Spiral Testing and Continuous Improvement* 174
Exhibit 2 *Prepare for the Next Spiral (Steps/Tasks)* 174
Exhibit 3 *Test Execution Status* 178
Exhibit 4 *Defect Gap Analysis* 178
Exhibit 5 *Defect Severity Status* 179
Exhibit 6 *Test Burnout Tracking* 179

Part 17

Exhibit 1 *Conduct System Test (Steps/Tasks)* 183
Exhibit 2 Final System Test Schedule 184

Part 18

Exhibit 1 *Conduct Acceptance Testing (Steps/Tasks)* 202

Exhibit 2 Acceptance Test Schedule 203

Part 19

Exhibit 1 Summarize/Report Spiral Test Results 210
Exhibit 2 Defects Documented by Function. 212
Exhibit 3 Defects Documented by Tester 213
Exhibit 4 Defect Gap Analysis 214
Exhibit 5 Defect Severity Status 215
Exhibit 6 Test Burnout Tracking 216
Exhibit 7 Root Cause Analysis 217
Exhibit 8 Defects by How Found 217
Exhibit 9 Defects by Who Found 218
Exhibit 10 Functions Tested/Not Tested 219
Exhibit 11 System Testing by Root Cause 220
Exhibit 12 Acceptance Testing by Root Cause 221
Exhibit 13 Finding/Recommendations Matrix 222

Part 20

Exhibit 1 Testing Tool Selection Checklist 229
Exhibit 2 Vendor vs. Testing Tool Type 234
Exhibit 3 Vendor Information. 236

Part 21

No exhibits

Part 22

Exhibit 1 Legacy Trade-Off Matrix. 286
Exhibit 2 Perfective, Adaptive, and Corrective Maintenance 292
Exhibit 3 Release, Intermediate, and Emergency Repair 300
Exhibit 4 Maintenance Testing and Continuous
Improvement 304
Exhibit 5 Iterative Maintenance Improvements. 307
Exhibit 6 Maintenance Testing Methodology 308

Part 23

Exhibit 1 Maintenance Testing and Continuous Improvement . . . 310
Exhibit 2 Enhancement/Defect Requirement Analysis
(Steps/Tasks) 311
Exhibit 3 Maintenance Interview Agenda. 312
Exhibit 4 Defects by Priority. 313
Exhibit 5 Defects by Defect Type 314
Exhibit 6 Defects by Module/Program 314
Exhibit 7 Defects by Application Area 315

Exhibit 8 Impact/Consequence Matrix 319

Part 24

Exhibit 1 Preliminary Maintenance Test Planning (Steps/Tasks) 322
Exhibit 2 Preliminary Maintenance Test Plan Outline 323
Exhibit 3 High-Level Business Functions 325
Exhibit 4 Functional Window Structure 325
Exhibit 5 Maintenance Defect States. 327
Exhibit 6 Maintenance Deliverable Approvals. 332

Part 25

Exhibit 1 Enhancement Prototype Design (Plan). 336
Exhibit 2 Maintenance Design Session Agenda 337
Exhibit 3 CRUD Matrix 339

Part 26

Exhibit 1 Completed Maintenance Test Plan 342
Exhibit 2 Retest Matrix 346
Exhibit 3 Test Schedule. 350
Exhibit 4 Maintenance Milestones 352
Exhibit 5 Test Schedule. 355

Part 27

Exhibit 1 Maintenance Testing and Continuous Improvement . . . 362
Exhibit 2 Maintenance Testing and Continuous Improvement . . . 363
Exhibit 3 Functional Breakdown 364
Exhibit 4 Functional Window Structure 365
Exhibit 5 Functional/Test Matrix 368
Exhibit 6 GUI Component Test Matrix 370
Exhibit 7 GUI Component Checklist 371
Exhibit 8 Baseline Performance Measurements 374

Part 28

Exhibit 1 Maintenance Test Development (Steps/Tasks) 378
Exhibit 2 Function/GUI Test Script 378

Part 29

Exhibit 1 Maintenance Testing and Continuous Improvement . . . 382
Exhibit 2 Maintenance Test Execution/Evaluation (Tasks/Steps) 382

Part 30

Exhibit 1 Maintenance Testing and Continuous Improvement . . . 388
Exhibit 2 Prepare for the Next Test Cycle. 388
Exhibit 3 Test Execution Status 391
Exhibit 4 Defect Gap Analysis 392
Exhibit 5 Defect Severity Status 393
Exhibit 6 Test Burnout Tracking 393
Exhibit 7 Requirements/Test Cases vs. Number of Defects Uncovered 394

Part 31

Exhibit 1 Conduct Maintenance System Testing 395

Part 32

Exhibit 1 Conduct Maintenance Acceptance Testing 397

Part 33

Exhibit 1 Summarize/Report Maintenance Test Results. 399

Part 34

Exhibit 1 System Installation 402
Exhibit 2 System Installation Metrics 403

Part 35

Exhibit 1 Maintenance Tool Selection Checklist 408
Exhibit 2 Vendor vs. Maintenance Tool Type. 411
Exhibit 3 Java Testing Tools 415
Exhibit 4 Web Link Checking Tools 417
Exhibit 5 HTML Validation Tools 418
Exhibit 6 Web Regression Tools. 419
Exhibit 7 Web Security Tools 420
Exhibit 8 Web Management Tools 422

Appendices

Exhibit A-1 Continuous Improvement 459
Exhibit A-2 Information Gathering. 460
Exhibit A-3 Test Planning 461
Exhibit A-4 Test Case Design 462
Exhibit A-5 Test Development 462
Exhibit A-6 Test Execution/Evaluation 463
Exhibit A-7 Prepare for the Next Spiral 463

Exhibit A-8 *Conduct System Testing* 464
Exhibit A-9 *Conduct Acceptance Testing* 465
Exhibit A-10 *Summarize/Report Spiral Test Results* 466
Exhibit D-1 Change Request Form 471
Exhibit E-1 Requirements Traceability Matrix 476
Exhibit E-2 Function/Test Matrix 479
Exhibit E-3 GUI Component Test Matrix 479
Exhibit E-4 GUI-Based Function Test Matrix 480
Exhibit E-5 Test Case 481
Exhibit E-6 Test Case Log 482
Exhibit E-7 Test Log Summary Report 483
Exhibit E-8 Software Problem Report 486
Exhibit E-9 Test Schedule 488
Exhibit E-10 Retest Matrix 490
Exhibit F-1 Requirements Phase Checklist 494
Exhibit F-2 Logical Design Phase Checklist 495
Exhibit F-3 Physical Design Phase Checklist 496
Exhibit F-4 Program Unit Design Phase Checklist 498
Exhibit F-5 Coding Phase Checklist 499
Exhibit F-6 Field Testing Checklist 501
Exhibit F-7 Record Testing Checklist 502
Exhibit F-8 File Test Checklist 503
Exhibit F-9 Error Testing Checklist 504
Exhibit F-10 Use Text Checklist 506
Exhibit F-11 Search Test Checklist 507
Exhibit F-12 Match/Merge Checklist 508
Exhibit F-13 Stress Test Checklist 509
Exhibit F-14 Attributes Testing Checklist 510
Exhibit F-15 States Testing Checklist 511
Exhibit F-16 Procedures Testing Checklist 512
Exhibit F-17 Control Testing Checklist 513
Exhibit F-18 Control Flow Testing Checklist 517
Exhibit F-19 Testing Tool Selection Checklist 518
Exhibit H-1 *Cause-Effect Graph* 533
Exhibit H-2 Decision Table 534
Exhibit H-3 CRUD Testing 536
Exhibit H-4 Referential Integrity Test Cases 538
Exhibit H-5 Sample Table 541
Exhibit H-6 Employee Table 545
Exhibit H-7 Employee Table 546
Exhibit H-8 Employee Table 547
Exhibit H-9 Telephone Line Table 547
Exhibit H-10 Employee Table 548
Exhibit H-11 Project Table 548
Exhibit H-12 Employee/Project Table 548

Exhibit H-13	Employee Table	549
Exhibit H-14	Project Table	549
Exhibit H-15	Employee/Project Table	549
Exhibit H-16	Purchase Agreement Table	550
Exhibit H-17	Customer/Salesperson Table	551
Exhibit H-18	Employee Table	554
Exhibit H-19	Customer Table	556
Exhibit H-20	Customer Table	556
Exhibit H-21	Customer Address Table	556
Exhibit H-22	Product Model Table	557
Exhibit H-23	Product Table	558
Exhibit H-24	Product Model Table	558
Exhibit H-25	Order Table	559
Exhibit H-26	Order Table	560
Exhibit H-27	Product/Model/Contract Table	560
Exhibit H-28	Customer Table	561
Exhibit H-29	Product Table	561
Exhibit H-30	Order Table	561
Exhibit H-31	Order Table	564
Exhibit H-32	Line-Item Table	564
Exhibit H-33	Employee Table	565
Exhibit H-34	Employee Table	565
Exhibit H-35	Decision Table	568
Exhibit H-36	Income vs. Tax Percent	568
Exhibit H-37	Income/Tax Test Cases	569
Exhibit H-38	Test Case/Error Exception Test Matrix	570
Exhibit H-39	Response Time of 100 Samples (seconds)	571
Exhibit H-40	Response Time Histogram	571
Exhibit H-41	Parameter Combinations (with total enumeration)	574
Exhibit H-42	Parameter Combinations (OATS)	575
Exhibit H-43	*Pareto Chart*	576
Exhibit H-44	Range Testing Test Cases	584
Exhibit H-45	Sample Run Chart	585
Exhibit H-46	State Transition Table	588

Exhibits listed in italics are graphics.

Introduction

Numerous textbooks address software testing in a structured development environment. By "structured" is meant a well-defined development cycle in which discretely defined steps provide measurable outputs at each step. It is assumed that software testing activities are based on clearly defined requirements and software development standards, and that those standards are used to develop and implement a plan for testing. Unfortunately, this is often not the case. Typically, testing is performed against changing, or even wrong, requirements.

This text aims to provide a quality framework for the software testing process in the traditional structured as well as unstructured environments. The goal is to provide a continuous quality improvement approach to promote effective testing methods and provide tips, techniques, and alternatives from which the user can choose.

The basis of the continuous quality framework stems from Edward Deming's quality principles. Deming was the pioneer in quality improvement, which helped turn Japanese manufacturing around. Deming's principles are applied to software testing in the traditional "waterfall" and rapid application "spiral" development (RAD) environments. The waterfall approach is one in which predefined, sequential steps are followed with clearly defined requirements. In the spiral approach, these rigid sequential steps may, to varying degrees, be lacking or different.

Section I, "Software Quality in Perspective," reviews modern quality assurance principles and best practices. It provides the reader with a detailed overview of basic software testing techniques, and introduces Deming's concept of quality through a continuous improvement process. The Plan–Do–Check–Act (PDCA) quality wheel is applied to the software testing process.

The **Plan** step of the continuous improvement process starts with a definition of the test objectives, or what is to be accomplished as a result of testing. The elements of a test strategy and test plan are described. A test strategy is a concise statement of how to meet the goals of testing and precedes test plan development. The outline of a good test plan is provided, including an introduction, the overall plan, testing requirements, test procedures, and test plan details.

The **Do** step addresses how to design or execute the tests included in the test plan. A cookbook approach describes how to perform component, integration, and system acceptance testing in a spiral environment.

The **Check** step emphasizes the importance of metrics and test reporting. A test team must formally record the results of tests and relate them to the test plan and system objectives. A sample test report format is provided, along with several graphic techniques.

The **Act** step of the continuous improvement process provides guidelines for updating test cases and test scripts. In preparation for the next spiral, suggestions for improving the people, process, and technology dimensions are provided.

Section II, "Life Cycle Testing Review," reviews the waterfall development methodology and describes how continuous quality improvement can be applied to the phased approach through technical reviews and software testing. The requirements, logical design, physical design, program unit design, and coding phases are reviewed. The roles of technical reviews and software testing are applied to each. Finally, the psychology of software testing is discussed.

Section III, "Client/Server and Internet Testing Methodology," contrasts the waterfall development methodology to the rapid application (RAD) spiral environment from a technical and psychological point of view. A spiral testing approach is suggested when the requirements are rapidly changing. A spiral methodology is provided, broken down into parts, steps, and tasks, applying Deming's continuous quality improvement process in the context of the PDCA quality wheel.

Section IV, "Modern Testing Tools," provides an overview of tools and guidelines for when to consider a testing tool and when not to. It also provides a checklist for selecting testing tools, consisting of a series of questions and responses. Examples are given of some of the most popular products. Finally, a detailed methodology for evaluating testing tools is provided, ranging from the initial test goals through training and implementation.

Section V, "Testing in the Maintenance Environment," discusses the fundamental challenges of maintaining and improving existing systems. Software changes are described and contrasted. Strategies for managing the maintenance effort are presented, along with the psychology of the software maintenance activity. A maintenance testing methodology is then broken down into parts, steps, and tasks, applying Deming's continuous quality improvement process in the context of the PDCA quality wheel.

Section VI, "Modern Maintenance Tools," presents an overview of maintenance tools and provides guidelines for when to consider a maintenance testing tool and when not to. Using a question–response format, the section also provides a checklist for selecting maintenance testing tools. Samples of the most popular maintenance testing tools are included, ranging from code complexity tools to configuration/process management tools.

Section I
Software Quality in Perspective

Software quality is something everyone wants. Managers know that they want high quality; software developers know they want to produce a quality product; and users insist that software work consistently and be reliable.

Many organizations form software quality assurance groups to improve and evaluate their software applications. However, there is no commonly accepted practice for quality assurance. Thus the quality assurance groups in various organizations may perform different roles and may execute their planning using different procedures.

In some organizations, software testing is a responsibility of that group. In others, software testing is the responsibility of the development group or an independent organization.

Many software quality groups develop software quality assurance plans, which are similar to test plans. However, a software quality assurance plan may include a variety of activities beyond those included in a test plan.

The objectives of this section are to:

- Define quality and its cost
- Differentiate quality prevention from quality detection
- Differentiate verification from validation
- Outline the components of quality assurance
- Outline common testing techniques
- Describe how the continuous improvement process can be instrumental in achieving quality

Part 1
Quality Assurance Framework

WHAT IS QUALITY?

In *Webster's Dictionary,* quality is defined as "the essential character of something, an inherent or distinguishing character, degree or grade of excellence." If you look at the computer literature, you will see that there are two generally accepted meanings of quality. The first is that quality means "meeting requirements." With this definition, to have a quality product, the requirements must be measurable, and the product's requirements will either be met or not met. With this meaning, quality is a binary state, i.e., it is a quality product or it is not. The requirements may be very complete or they may be simple, but as long as they are measurable, it can be determined whether quality has or has not been met. This is the producer's view of quality as meeting the producer's requirements or specifications. Meeting the specifications becomes an end in itself.

Another definition of quality, the customer's, is the one we will use. With this definition, the customer defines quality as to whether or not the product or service does what the customer needs. Another way of wording it is "fit for use." There should also be a description of the purpose of the product, typically documented in a customer's "requirements specification" (see Appendix C, Requirements Specification, for more details). The requirements are the most important document, and the quality system revolves around it. In addition, quality *attributes* are described in the customer's requirements specification. Examples include usability, the relative ease with which a user communicates with the application, portability, the capability of the system to be executed across a diverse range of hardware architectures, and reusability, the ability to transfer software components constructed in one software system into another.

While everyone is committed to quality, the following are some confusions shared by many individuals, which inhibit achieving a quality commitment:

1. Quality requires a commitment, particularly from top management. Close cooperation of management and staff is required in order to make it happen.

2. Many individuals believe that defect-free products and services are impossible, and accept certain levels of defects are normal and acceptable.
3. Quality is frequently associated with cost, meaning that high quality equals high cost. This is a confusion between quality of design and quality of conformance.
4. Quality demands requirement specifications in enough detail that the products produced can be quantitatively measured against those specifications. Many organizations are not capable or willing to expend the effort to produce specifications at the level of detail required.
5. Technical personnel often believe that standards stifle their creativity, and thus do not abide by standards compliance. However, for quality to happen, well-defined standards and procedures must be followed.

PREVENTION VS. DETECTION

Quality cannot be achieved by assessing an already completed product. The aim, therefore, is to prevent quality defects or deficiencies in the first place, and to make the products assessable by quality assurance measures. Some quality assurance measures include: structuring the development process with a software development standard and supporting the development process with methods, techniques, and tools.

In addition to product assessments, process assessments are essential to a quality management program. Examples include documentation of coding standards, prescription and use of standards, methods, and tools, procedures for data backup, change management, defect documentation, and reconciliation.

Quality management decreases production costs because the sooner a defect is located and corrected, the less costly it will be in the long run. Although the initial investment can be substantial, the long-term result will be higher-quality products and reduced maintenance costs.

The total cost of effective quality management is the sum of four component costs — prevention, inspection, internal failure, and external failure. Prevention costs consist of actions taken to prevent defects from occurring in the first place. Inspection costs consist of measuring, evaluating, and auditing products or services for conformance to standards and specifications. Internal failure costs are those incurred in fixing defective products before they are delivered. External failure costs consist of the costs of defects discovered after the product has been released. The latter can be devastating because they may damage the organization's reputation or result in the loss of future sales.

The greatest payback is with prevention. Increasing the emphasis on prevention costs reduces the number of defects which go to the customer undetected, improves product quality, and reduces the cost of production and maintenance.

VERIFICATION VS. VALIDATION

Verification is proving that a product meets the requirements specified during previous activities carried out correctly throughout the development life cycle, while validation checks that the system meets the customer's requirements at the end of the life cycle. It is a proof that the product meets the expectations of the users, and it ensures that the executable system performs as specified. The creation of the test product is much more closely related to validation than to verification. Traditionally, software testing has been considered a validation process, i.e., a life cycle phase. After programming is completed, the system is validated or tested to determine its functional and operational performance.

When verification is incorporated into testing, testing occurs throughout the development life cycle. For best results, it is good practice to combine verification with validation in the testing process. Verification includes systematic procedures of review, analysis, and testing, employed throughout the software development life cycle, beginning with the software requirements phase and continuing through the coding phase. Verification ensures the quality of software production and maintenance. In addition, verification imposes such an organized, systematic development practice that the resulting program can be easily understood and evaluated by an independent party.

Verification emerged about 15 years ago as a result of the aerospace industry's need for extremely reliable software in systems in which an error in a program could cause mission failure and result in enormous time and financial setbacks, or even life-threatening situations. The concept of verification includes two fundamental criteria. First, the software must adequately and correctly perform all intended functions. Second, the software must not perform any function that either by itself or in combination with other functions can degrade the performance of the entire system. The overall goal of verification is to ensure that each software product developed throughout the software life cycle meets the customer's needs and objectives as specified in the software requirements document.

Verification also establishes tractability between the various sections of the software documentation and the associated parts of the requirements specification. A comprehensive verification effort ensures that all software performance and quality requirements in the specification are adequately tested and that the test results can be repeated after changes are installed. Verification is a "continuous improvement process" and has no definite termination. It should be used throughout the system life cycle to maintain configuration and operational integrity.

Verification ensures that the software functions as intended and has the required attributes, e.g., portability, and increases the chances that the software will contain few errors (i.e., an acceptable number in the final product). It provides a method for closely monitoring the software development project and provides management with a detailed status of the project at any point in time. When verification procedures are used, management can be assured that the developers follow a formal, sequential, traceable software development process, with a minimum set of activities to enhance the quality of the system.

One criticism of verification is that it increases software development costs considerably. When the cost of software throughout the total life cycle from inception to the final abandonment of the system is considered, however, verification actually reduces the overall cost of the software. With an effective verification program, there is typically a four-to-one reduction in defects in the installed system. Because error corrections can cost 20 to 100 times more during operations and maintenance than during design, overall savings far outweigh the initial extra expense.

SOFTWARE QUALITY ASSURANCE

A formal definition of software quality assurance is that it is "the systematic activities providing evidence of the fitness for use of the total software product." Software quality assurance is achieved through the use of established guidelines for quality control to ensure the integrity and prolonged life of software. The relationships between quality assurance, quality control, the auditing function, and software testing are often confused.

Quality assurance is the set of support activities needed to provide adequate confidence that processes are established and continuously improved in order to produce products that meet specifications and are fit for use. Quality control is the process by which product quality is compared with applicable standards and the action taken when nonconformance is detected. Auditing is the inspection/assessment activity that verifies compliance with plans, policies, and procedures.

Software quality assurance is a planned effort to ensure that a software product fulfills these criteria and has additional attributes specific to the project, e.g., portability, efficiency, reusability, and flexibility. It is the collection of activities and functions used to monitor and control a software project so that specific objectives are achieved with the desired level of confidence. It is not the sole responsibility of the software quality assurance group but is determined by the consensus of the project manager, project leader, project personnel, and users.

Quality assurance is the function responsible for managing quality. The word "assurance" means that if the processes are followed, management can be assured of product quality. Quality assurance is a catalytic function that should encourage quality attitudes and discipline on the part of management and workers. Successful quality assurance managers know how to make people quality conscious and to make them recognize the benefits of quality to themselves and to the organization.

The objectives of software quality are typically achieved by following a software quality assurance plan that states the methods the project will employ to ensure the documents or products produced and reviewed at each milestone are of high quality. Such an explicit approach ensures that all steps have been taken to achieve software quality and provides management with documentation of those actions. The plan states the criteria by which quality activities can be monitored rather than setting impossible goals, e.g., no software defects or 100% reliable software.

Software quality assurance is a strategy for risk management. It exists because software quality is typically costly and should be incorporated into the formal risk management of a project. Some examples of poor software quality include:

1. Delivered software frequently fails.
2. Unacceptable consequences of system failure, from financial to life-threatening scenarios.
3. Systems are often not available for their intended purpose.
4. System enhancements are often very costly.
5. Costs of detecting and removing defects are excessive.

Although most quality risks are related to defects, this only tells part of the story. A defect is a failure to comply with a requirement. If the requirements are inadequate or even incorrect, the risks of defects are more pervasive. The result is too many built-in defects and products that are not verifiable. Some risk management strategies and techniques include software testing, technical reviews, peer reviews, and compliance verification.

COMPONENTS OF QUALITY ASSURANCE

Most software quality assurance activities can be categorized into software testing, i.e., verification and validation, software configuration management, and quality control. But the success of a software quality assurance program also depends on a coherent collection of standards, practices, conventions, and specifications, as shown in Exhibit 1.

Software Testing

Software testing is a popular risk management strategy. It is used to verify that functional requirements were met. The limitation of this approach,

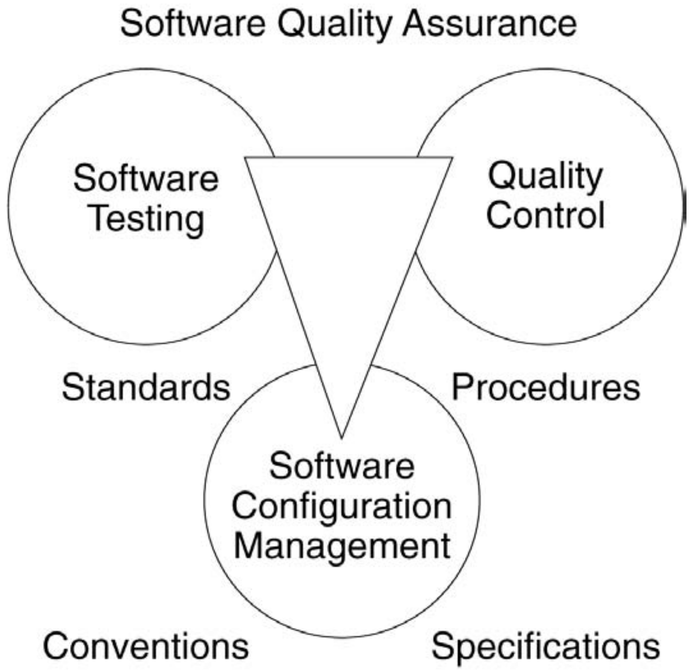

Exhibit 1. Quality Assurance Components

however, is that by the time testing occurs, it is too late to build quality into the product. Tests are only as good as the test cases, but they can be inspected to ensure that all the requirements are tested across all possible combinations of inputs and system states. However, not all defects are discovered during testing. Software testing includes the activities outlined in this text, including verification and validation activities. In many organizations, these activities, or their supervision, are included within the charter for the software quality assurance function. The extent to which personnel independent of design and coding should participate in software quality assurance activities is a matter of institutional, organizational, and project policy.

The major purpose of verification and validation activities is to ensure that software design, code, and documentation meet all the requirements imposed on them. Examples of requirements include user requirements, specifications derived from and designed to meet user requirements, code review and inspection criteria, test requirements at the modular, subsystem, and integrated software levels, and acceptance testing of the code after it has been fully integrated with hardware.

During software design and implementation, verification helps determine whether the products of one phase of the software development life cycle fulfill the requirements established during the previous phase. The verification effort takes less time and is less complex when conducted throughout the development process.

Quality Control

Quality control is defined as the processes and methods used to monitor work and observe whether requirements are met. It focuses on reviews and removal of defects before shipment of products. Quality control should be the responsibility of the organizational unit producing the product. It is possible to have the same group that builds the product perform the quality control function, or to establish a quality control group or department within the organizational unit that develops the product.

Quality control consists of well-defined checks on a product that are specified in the product quality assurance plan. For software products, quality control typically includes specification reviews, inspections of code and documents, and checks for user deliverables. Usually, document and product inspections are conducted at each life cycle milestone to demonstrate that the items produced are within the criteria specified by the software quality assurance plan. These criteria are normally provided in the requirements specifications, conceptual and detailed design documents, and test plans. The documents given to users are the requirement specifications, design documentation, results from the user acceptance test, the software code, user guide, and the operations and maintenance guide. Additional documents are specified in the software quality assurance plan.

Quality control can be provided by various sources. For small projects, the project personnel's peer group or the department's software quality coordinator can inspect the documents. On large projects, a configuration control board may be responsible for quality control. The board may include the users or a user representative, a member of the software quality assurance department, and the project leader.

Inspections are traditional functions of quality control, i.e., independent examinations to assess compliance with some stated criteria. Peers and subject matter experts review specifications and engineering work products to identify defects and suggest improvements. They are used to examine the software project for adherence to the written project rules at a project's milestones and at other times during the project's life cycle as deemed necessary by the project leader or the software quality assurance personnel. An inspection may be a detailed checklist for assessing compliance or a brief checklist to determine the existence of such deliverables as documentation. A report stating the purpose of the inspection and the deficiencies found goes to the project supervisor, project leader, and project personnel for action.

Responsibility for inspections is stated in the software quality assurance plan. For small projects, the project leader or the department's quality coordinator can perform the inspections. For large projects, a member of the software quality assurance group may lead an inspection performed by an audit team, which is similar to the configuration control board mentioned previously. Following the inspection, project personnel are assigned to correct the problems on a specific schedule.

Quality control is designed to detect and correct defects, while quality assurance is oriented toward preventing them. Detection implies flaws in the processes that are supposed to produce defect-free products and services. Quality assurance is a managerial function that prevents problems by heading them off, and by advising restraint and redirection.

Software Configuration Management

Software configuration management is concerned with labeling, tracking, and controlling changes in the software elements of a system. It controls the evolution of a software system by managing versions of its software components and their relationships.

The purpose of software configuration management is to identify all the interrelated components of software and to control their evolution throughout the various life cycle phases. Software configuration management is a discipline that can be applied to activities including software development, document control, problem tracking, change control, and maintenance. It can provide a high cost savings in software reusability because each software component and its relationship to other software components have been defined.

Software configuration management consists of activities that ensure that design and code are defined and cannot be changed without a review of the effect of the change itself and its documentation. The purpose of configuration management is to control code and its associated documentation so that final code and its description are consistent and represent those items that were actually reviewed and tested. Thus, spurious, last-minute software changes are eliminated.

For concurrent software development projects, software configuration management can have considerable benefits. It can organize the software under development and minimize the probability of inadvertent changes. Software configuration management has a stabilizing effect on all software when there is a great deal of change activity or a considerable risk of selecting the wrong software components.

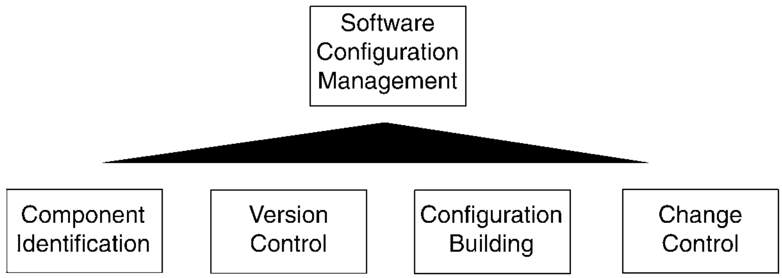

Exhibit 2. Software Configuration Management

ELEMENTS OF SOFTWARE CONFIGURATION MANAGEMENT

Software configuration management identifies a system configuration in order to systematically control changes, maintain integrity, and enforce tractability of the configuration throughout its life cycle. Components to be controlled include planning, analysis, and design documents, source code, executable code, utilities, job control language (JCL), test plans, test scripts, test cases, and development reports. The software configuration process typically consists of four elements — software component identification, software version control, configuration building, and software change control, as shown in Exhibit 2.

Component Identification

A basic software configuration management activity is the identification of the software components that make up a deliverable at each point of its development. Software configuration management provides guidelines to identify and name software baselines, software components, and software configurations.

Software components go through a series of changes. In order to manage the development process, one must establish methods and name standards for uniquely identifying each revision. A simple way to name component revisions is to use a series of discrete digits. The first integer could refer to a software component's external release number. The second integer could represent the internal software development release number. The transition from version number 2.9 to 3.1 would indicate that a new external release 3 has occurred. The software component version number is automatically incremented when the component is checked into the software library. Further levels of qualifiers could also be used as necessary, such as the date of a new version.

A software configuration is a collection of software elements that comprise a major business function. An example of a configuration is the set of program modules for an order system. Identifying a configuration is quite similar to identifying individual software components. Configurations can have a sequence of versions. Each configuration must be named in a way that distinguishes it from others. Each configuration version must be differentiated from other versions. The identification of a configuration must also include its approval status and a description of how the configuration was built.

A simple technique for identifying a configuration is to store all its software components in a single library or repository. The listing of all the components can also be documented.

Version Control

As an application evolves over time, many different versions of its software components are created, and there needs to be an organized process to manage changes in the software components and their relationships. In addition, there is usually the requirement to support parallel component development and maintenance.

Software is frequently changed as it evolves through a succession of temporary states called versions. A software configuration management facility for controlling versions is a software configuration management repository or library. Version control provides the tractability or history of each software change, including who did what, why, and when.

Within the software life cycle, software components evolve, and at a certain point each reaches a relatively stable state. But as defects are corrected and enhancement features are implemented, the changes result in new versions of the components. Maintaining control of these software component versions is called *versioning.*

A component is identified and labeled to differentiate it from all other software versions of the component. When a software component is modified, both the old and new versions should be separately identifiable. Therefore, each version, except for the initial one, has a predecessor. The succession of component versions is the component's history and tractability. Different versions also act as backups so that one can return to previous versions of the software.

Configuration Building

To build a software configuration one needs to identify the correct component versions and execute the component build procedures. This is often called *configuration building.*

A software configuration consists of a set of derived software components. An example is executable object programs derived from source programs. Derived software components are correctly associated with each source component to obtain an accurate derivation. The configuration build model defines how to control the way derived software components are put together.

The inputs and outputs required for a configuration build model include the primary inputs such as the source components, the version selection procedures, and the system model, which describes how the software components are related. The outputs are the target configuration and respectively derived software components.

Software configuration management environments use different approaches for selecting versions. The simplest approach to version selection is to maintain a list of component versions. Other approaches entail selecting the most recently tested component versions, or those modified on a particular date.

Change Control

Change control is the process by which a modification to a software component is proposed, evaluated, approved or rejected, scheduled, and tracked. Its basic foundation is a change control process, a component status reporting process, and an auditing process.

Software change control is a decision process used in controlling the changes made to software. Some proposed changes are accepted and implemented during this process. Others are rejected or postponed, and are not implemented. Change control also provides for impact analysis to determine the dependencies.

Modification of a configuration has at least four elements: a change request, an impact analysis of the change, a set of modifications and additions of new components, and a method for reliably installing the modifications as a new baseline (see Appendix D, Change Request Form, for more details).

A change often involves modifications to multiple software components. Therefore, a storage system that provides for multiple versions of a single file is usually not sufficient. A technique is required to identify the set of modifications as a single change. This is often called *delta storage*.

Every software component has a development life cycle. A life cycle consists of states and allowable transitions between those states. When a software component is changed, it should always be reviewed and frozen from further modifications until a new version is created. The reviewing authority must approve or reject the modified software component. A software library holds all software components as soon as they are frozen and also acts as a repository for approved components.

A derived component is linked to its source and has the same status as its source. In addition, a configuration cannot have a more complete status than any of its components, because it is meaningless to review a configuration when some of the associated components are not frozen.

All components controlled by software configuration management are stored in a software configuration library, including work products such as business data and process models, architecture groups, design units, tested application software, reusable software, and special test software. When a software component is to be modified, it is checked out of the repository into a private workspace. It evolves through many states which are temporarily out of the scope of configuration management control.

When a change is completed, the component is checked into the library and becomes a new software component version. The previous component version is also retained.

SOFTWARE QUALITY ASSURANCE PLAN

The software quality assurance (SQA) plan is an outline of quality measures to ensure quality levels within a software development effort. The plan is used as a baseline to compare the *actual* levels of quality during development with the *planned* levels of quality. If the levels of quality are not within the planned quality levels, management will respond appropriately as documented within the plan.

The plan provides the framework and guidelines for development of understandable and maintainable code. These ingredients help ensure the quality sought in a software project. A SQA plan also provides the procedures for ensuring that quality software will be produced or maintained inhouse or under contract. These procedures affect planning, designing, writing, testing, documenting, storing, and maintaining computer software. It should be organized in this way because the plan ensures the quality of the software rather than describing specific procedures for developing and maintaining the software.

Steps to Develop and Implement a Software Quality Assurance Plan

Step 1. Document the Plan. The software quality assurance plan should include the sections below (see Appendix B, Software Quality Assurance Plan, which contains a template for the plan).

1. *Purpose Section*
   This section delineates the specific purpose and scope of the particular SQA plan. It should list the name(s) of the software items covered by the SQA plan and the intended use of the software. It states the portion of the software life cycle covered by the SQA plan for each software item specified.
2. *Reference Document Section*
   This section provides a complete list of documents referenced elsewhere in the text of the SQA plan.
3. *Management Section*
   This section describes the project's organizational structure, tasks, and responsibilities.
4. *Documentation Section*
   This section identifies the documentation governing the development, verification and validation, use, and maintenance of the software. It also states how the documents are to be checked for adequacy. This includes the criteria and the identification of the review or audit by which the adequacy of each document will be confirmed.
5. *Standards, Practices, Conventions, and Metrics Section*
   This section identifies the standards, practices, conventions, and metrics to be applied, and also states how compliance with these items is to be monitored and assured.
6. *Reviews and Inspections Section*
   This section defines the technical and managerial reviews, walkthroughs, and inspections to be conducted. It also states how the reviews, walkthroughs, and inspections, are to be accomplished including follow-up activities and approvals.
7. *Software Configuration Management Section*
   This section is addressed in detail in the project's software configuration management plan.
8. *Problem Reporting and Corrective Action Section*
   This section is addressed in detail in the project's software configuration management plan.
9. *Tools, Techniques, and Methodologies Section*
   This section identifies the special software tools, techniques, and methodologies that support SQA, states their purposes, and describes their use.
10. *Code Control Section*

This section defines the methods and facilities used to maintain, store, secure, and document the controlled versions of the identified software during all phases of development. This may be implemented in conjunction with a computer program library and/or may be provided as a part of the software configuration management plan.

11. *Media Control Section*
    This section states the methods and facilities to be used to identify the media for each computer product and the documentation required to store the media, including the copy and restore process, and protects the computer program physical media from unauthorized access or inadvertent damage or degradation during all phases of development. This may be provided by the software configuration management plan.
12. *Supplier Control Section*
    This section states the provisions for assuring that software provided by suppliers meets established requirements. In addition, it should state the methods that will be used to assure that the software supplier receives adequate and complete requirements. For previously developed software, this section will state the methods to be used to assure the suitability of the product for use with the software items covered by the SQA plan. For software to be developed, the supplier will be required to prepare and implement an SQA plan in accordance with this standard. This section will also state the methods to be employed to assure that the developers comply with the requirements of this standard.
13. *Records Collection, Maintenance, and Retention Section*
    This section identifies the SQA documentation to be retained. It will state the methods and facilities to assemble, safeguard, and maintain this documentation, and will designate the retention period. The implementation of the SQA plan involves the necessary approvals for the plan as well as development of a plan for execution. The subsequent evaluation of the SQA plan will be performed as a result of its execution.
14. *Testing Methodology*
    This section defines the testing approach, techniques, and automated tools that will be used.

Step 2. Obtain Management Acceptance. Management participation is necessary for the successful implementation of an SQA plan. Management is responsible both for ensuring the quality of a software project and for providing the resources needed for software development.

The level of management commitment required for implementing an SQA plan depends on the scope of the project. If a project spans organizational boundaries, approval should be obtained from all affected areas. Once approval has been obtained, the SQA plan is placed under configuration control.

In the management approval process, management relinquishes tight control over software quality to the SQA plan administrator in exchange for improved software quality. Software quality is often left to software developers. Quality is desirable, but management may express concern as to the cost of a formal SQA plan. Staff should be aware that management views the program as a means of ensuring software quality, and not as an end in itself.

To address management concerns, software life cycle costs should be formally estimated for projects implemented both with and without a formal SQA plan. In general, implementing a formal SQA plan makes economic and management sense.

Step 3. Obtain Development Acceptance. Since the software development and maintenance personnel are the primary users of an SQA plan, their approval and cooperation in implementing the plan are essential. The software project team members must adhere to the project SQA plan; everyone must accept it and follow it.

No SQA plan is successfully implemented without the involvement of the software team members and their managers in the development of the plan. Because project teams generally have only a few members, all team members should actively participate in writing the SQA plan. When projects become much larger (i.e., encompassing entire divisions or departments), representatives of project subgroups should provide input. Constant feedback from representatives to team members helps gain acceptance of the plan.

Step 4. Plan for Implementation of the SQA Plan. The process of planning, formulating, and drafting an SQA plan requires staff and word processing resources. The individual responsible for implementing an SQA plan must have access to these resources. In addition, the commitment of resources requires management approval and, consequently, management support. To facilitate resource allocation, management should be made aware of any project risks that may impede the implementation process (e.g., limited availability of staff or equipment). A schedule for drafting, reviewing, and approving the SQA plan should be developed.

Step 5. Execute the SQA Plan. The actual process of executing an SQA plan by the software development and maintenance team involves determining necessary audit points for monitoring it. The auditing function must be scheduled during the implementation phase of the software product so that the SQA plan will not be hurt by improper monitoring of the software project. Audit points should occur either periodically during development or at specific project milestones (e.g., at major reviews or when part of the project is delivered).

ISO9000 QUALITY STANDARDS

ISO9000 is a quality series and comprises a set of five documents developed in 1987 by the International Standards Organization (ISO). ISO9000 standards and certification are usually associated with non-IS manufacturing processes. However, application development organizations can benefit from these standards and position themselves for certification, if necessary. All the ISO9000 standards are guidelines and interpretive because of their lack of stringency and rules. ISO certification is becoming more and more important throughout Europe and the U.S. for the manufacture of hardware. Software suppliers will increasingly be required to have certification. ISO9000 is a definitive set of quality standards, but it represents quality standards as part of a total quality management (TQM) program. It consists of ISO9001, ISO9002, or ISO9003, and it provides the guidelines for selecting and implementing a quality assurance standard.

ISO9001 is a very comprehensive standard and defines all the quality elements required to demonstrate the supplier's ability to design and deliver a quality product. ISO9002 covers quality considerations for the supplier to control the design and development activities. ISO9003 demonstrates the supplier's ability to detect and control product nonconformity during inspection and testing. ISO9004 describes the quality standards associated with ISO9001, ISO9002, and ISO9003 and provides a comprehensive quality checklist.

Exhibit 3 shows the ISO9000 and companion international standards.

Exhibit 3. Companion ISO Standards

International	U.S.	Europe	U.K.
ISO9000	ANSI/ASQA	EN29000	BS5750 (Part 0.1)
ISO9001	ANSI/ASQC	EN29001	BS5750 (Part 1)
ISO9002	ANSI/ASQC	EN29002	BS5750 (Part 2)
ISO9003	ANSI/ASQC	EN29003	BS5750 (Part 3)
ISO9004	ANSI/ASQC	EN29004	BS5750 (Part 4)

Part 2
Overview of Testing Techniques

TEST CASE DESIGN

Ad hoc testing or error guessing is an informal testing technique that relies on inspiration, creative thinking, and brainstorming to design tests. While this technique is important and often very useful, there is no substitute for formal test techniques. Formal design techniques provide a higher probability of assuring test coverage and reliability. Test case design requires a specification that includes a description of the functionality, inputs, and outputs. Test case design becomes a part of the system documentation. Each test case must have a clear definition of the test objectives. Risk management may help define the test objectives, especially in areas of high risk. Following is a discussion of the leading test design approaches.

Black-Box Testing (functional)

Black-box or functional testing is one in which test conditions are developed based on the program or system's functionality, i.e., the tester requires information about the input data and observed output, but does not know how the program or system works. Just as one does not have to know how a car works internally to drive it, it is not necessary to know the internal structure of a program to execute it. The tester focuses on testing the program's functionality against the specification. With black-box testing, the tester views the program as a black-box and is completely unconcerned with the internal structure of the program or system. Some examples in this category include: decision tables, equivalence partitioning, range testing, boundary value testing, database integrity testing, cause-effect graphing, orthogonal array testing, array and table testing, exception testing, limit testing, and random testing.

A major advantage of black-box testing is that the tests are geared to what the program or system is supposed to do, and it is natural and understood by everyone. This should be verified with techniques such as structured walkthroughs, inspections, and JADs. A limitation is that exhaustive input testing is not achievable, because this requires that every possible input condition or combination be tested. Additionally, since there is no knowledge of the internal structure or logic, there could be errors or deliberate mischief on the part of a programmer, which may not be detectable with black-box testing. For example, suppose a payroll programmer wants to insert some job security into a payroll application he or she is developing. By inserting the following extra code into the application, if the employee were to be terminated, i.e., his or her employee ID no longer exists in the system, justice would sooner or later prevail.

```
if my employee ID exists
   deposit regular pay check into my bank account
else
   deposit an enormous amount of money into my bank account
   erase any possible financial audit trails
   erase this code
```

White-Box Testing (structural)

In white-box or structural testing test conditions are designed by examining paths of logic. The tester examines the internal structure of the program or system. Test data are driven by examining the logic of the program or system, without concern for the program or system requirements. The tester knows the internal program structure and logic, just as a car mechanic knows the inner workings of an automobile. Specific examples in this category include basis path analysis, statement coverage, branch coverage, condition coverage, and branch/condition coverage.

An advantage of white-box testing is that it is thorough and focuses on the produced code. Since there is knowledge of the internal structure or logic, errors or deliberate mischief on the part of a programmer have a higher probability of being detected.

One disadvantage of white-box testing is that it does not verify that the specifications are correct, i.e., it focuses only on the internal logic and does not verify the logic to the specification. Another disadvantage is that there is no way to detect missing paths and data-sensitive errors. For example, if the statement in a program should be coded "if |a-b| < 10" but is coded "if (a-b) <1," this would not be detectable without specification details. A final disadvantage is that white-box testing cannot execute all possible logic paths through a program because this would entail an astronomically large number of tests.

Gray-Box Testing (functional and structural)

Black-box testing focuses on the program's functionality against the specification. White-box testing focuses on the paths of logic. Gray-box testing is a combination of black- and white-box testing. The tester studies the requirements specifications and communicates with the developer to understand the internal structure of the system. The motivation is to clear up ambiguous specifications and "read between the lines" to design implied tests. One example of the use of gray-box testing is when it appears to the tester that a certain functionality seems to be reused throughout an application. If the tester communicates with the developer and understands the internal design and architecture, many tests will be eliminated, because it may be possible to test the functionality only once. Another example is when the syntax of a command consists of seven possible parameters that can be entered in any order, as follows:

Command parm1, parm2, parm3, parm4, parm5, parm6, parm7

In theory, a tester would have to create 7! or 5,040 tests. The problem is compounded further if some of the parameters are optional. If the tester uses gray-box testing, by talking with the developer and understanding the parser algorithm, if each parameter is independent, only seven tests may be required.

Manual vs. Automated Testing

The basis of the manual testing categorization is that it is not typically carried out by people and is not implemented on the computer. Examples include, structured walkthroughs, inspections, joint application designs (JADs), and desk checking.

The basis of the automated testing categorization is that it is implemented on the computer. Examples include boundary value testing, branch coverage testing, prototyping, and syntax testing. Syntax testing is performed by a language compiler, and the compiler is a program that executes on a computer.

Static vs. Dynamic Testing

Static testing approaches are time independent and are classified in this way because they do not necessarily involve either manual or automated execution of the product being tested. Examples include syntax checking, structured walkthroughs, and inspections. An inspection of a program occurs against a source code listing in which each code line is read line by line and discussed. An example of static testing using the computer is a static flow analysis tool, which investigates another program for errors without executing the program. It analyzes the other program's control and data flow to discover problems such as references to a variable that has not been initialized and unreachable code.

Dynamic testing techniques are time dependent and involve executing a specific sequence of instructions on paper or by the computer. Examples include structured walkthroughs, in which the program logic is simulated by walking through the code and verbally describing it. Boundary testing is a dynamic testing technique that requires the execution of test cases on the computer with a specific focus on the boundary values associated with the inputs or outputs of the program.

TAXONOMY OF SOFTWARE TESTING TECHNIQUES

A testing technique is a set of interrelated procedures which, together, produce a test deliverable. There are many possible classification schemes for software testing and Exhibit 1 describes one way. The exhibit reviews formal popular testing techniques and also classifies each per the above discussion as manual, automated, static, dynamic, functional (black-box), or structural (white-box). Exhibit 2 provides a description of each technique.

Exhibit 1. Testing Technique Categories

Technique	Manual	Automated	Static	Dynamic	Functional	Structural
Basis Path Testing		x		x		x
Black-Box Testing		x		x	x	
Bottom-Up Testing		x		x		x
Boundary Value Testing		x		x	x	
Branch/Condition Coverage		x		x		x
Branch Coverage Testing		x		x		x
Cause-Effect Graphing		x		x	x	
Condition Coverage Testing		x		x		x
CRUD Testing		x		x	x	
Database Testing		x		x		x
Decision Tables		x		x	x	
Desk Checking	x			x		x
Equivalence Partitioning		x		x	x	
Exception Testing		x		x	x	
Free Form Testing		x		x	x	
Gray-Box Testing		x		x	x	x
Histograms	x				x	
Inspections	x		x		x	x
JADs	x				x	x
Orthogonal Array Testing	x		x		x	
Pareto Analysis	x				x	
Positive and Negative Testing		x		x	x	
Prior Defect History Testing	x		x		x	
Prototyping		x		x	x	
Random Testing		x		x	x	
Range Testing		x		x	x	
Regression Testing				x	x	
Risk-Based Testing	x		x		x	

Exhibit 1. (Continued) Testing Technique Categories

Technique	Manual	Automated	Static	Dynamic	Functional	Structural
Run Charts	x		x		x	
Sandwich Testing		x		x		x
Statement Coverage Testing		x		x		x
State Transition Testing		x		x	x	
Statistical Profile Testing	x		x		x	
Structured Walkthroughs	x			x	x	x
Syntax Testing		x	x	x	x	
Table Testing		x		x		x
Thread Testing		x		x		x
Top-Down Testing		x		x	x	x
White-Box Testing		x		x		x

Exhibit 2. Testing Technique Descriptions

Technique	Brief Description
Basis Path Testing	Identifying tests based on flow and paths of a program or system
Black-Box Testing	Test cases generated based on the system's functionality
Bottom-Up Testing	Integrating modules or programs starting from the bottom
Boundary Value Testing	Test cases generated from boundary values of equivalence classes
Branch/Condition Coverage Testing	Verify each condition in a decision takes on all possible outcomes at least once
Branch Coverage Testing	Verify each branch has true and false outcomes at least once
Cause-Effect Graphing	Mapping multiple simultaneous inputs which may affect others to identify their conditions to test
Condition Coverage Testing	Verify that each condition in a decision takes on all possible outcomes at least once
CRUD Testing	Build CRUD matrix and test all object creations, reads, updates, and deletions
Database Testing	Check the integrity of database field values
Decision Tables	Table showing the decision criteria and the respective actions
Desk Checking	Developer reviews code for accuracy
Equivalence Partitioning	Each input condition partitioned into two or more groups. Test cases are generated from representative valid and invalid classes
Exception Testing	Identify error messages and exception handling processes and conditions that trigger them
Free Form Testing	*Ad hoc* or brainstorming using intuition to define test cases

Exhibit 2. (Continued) Testing Technique Descriptions

Technique	Brief Description
Gray-Box Testing	A combination of black-box and white-box testing to take advantage of both
Histograms	A graphical representation of measured values organized according to the frequency of occurrence used to pinpoint hot spots
Inspections	Formal peer review that uses checklists, entry criteria, and exit criteria
JADs	Technique that brings users and developers together to jointly design systems in facilitated sessions
Orthogonal Array Testing	Mathematical technique to determine which variations of parameters need to be tested
Pareto Analysis	Analyze defect patterns to identify causes and sources
Positive and Negative Testing	Test the positive and negative values for all inputs
Prior Defect History Testing	Test cases are created or rerun for every defect found in prior tests of the system
Prototyping	General approach to gather data from users by building and demonstrating to them some part of a potential application
Random Testing	Technique involving random selection from a specific set of input values where any value is as likely as any other
Range Testing	For each input identifies the range over which the system behavior should be the same
Regression Testing	Tests a system in light of changes made during a development spiral, debugging, maintenance, or the development of a new release

Exhibit 2. (Continued) Testing Technique Descriptions

Technique	Brief Description
Risk-Based Testing	Measure the degree of business risk in a system to improve testing
Run Charts	A graphical representation of how a quality characteristic varies with time
Sandwich Testing	Integrating modules or programs from the top and bottom simultaneously
Statement Coverage Testing	Every statement in a program is executed at least once
State Transition Testing	Technique in which the states of a system are first identified and then test cases are written to test the triggers to cause a transition from one condition to another state
Statistical Profile Testing	Statistical techniques are used to develop a usage profile of the system that helps define transaction paths, conditions, functions, and data tables
Structured Walkthroughs	A technique for conducting a meeting at which project participants examine a work product for errors
Syntax Testing	Data-driven technique to test combinations of input syntax
Table Testing	Test access, security, and data integrity of table entries
Thread Testing	Combining individual units into threads of functionality which together accomplish a function or set of functions
Top-Down Testing	Integrating modules or programs starting from the top
White-Box Testing	Test cases are defined by examining the paths of logic of a system