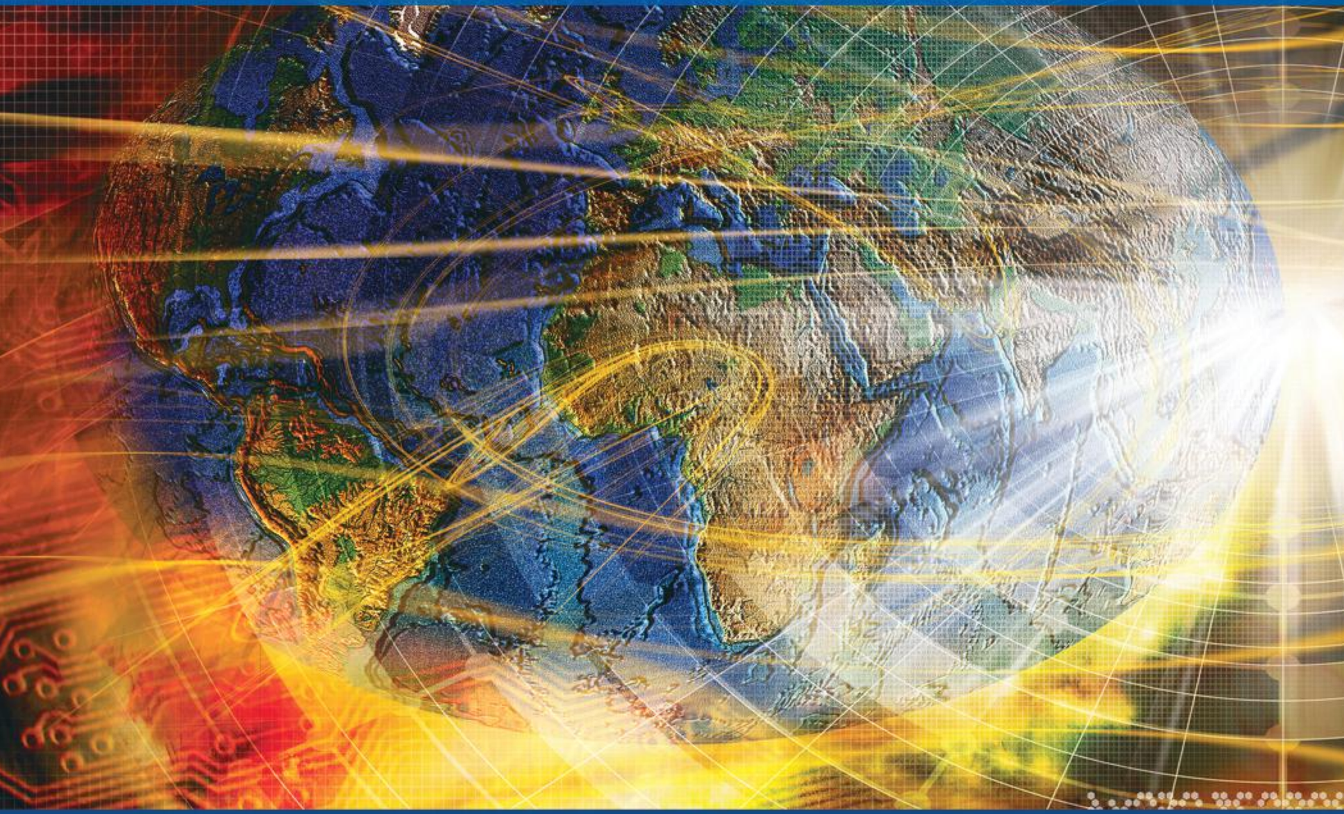
Software Testing

A Craftsman's Approach

Fourth Edition



Paul C. Jorgensen

CRC Press
Taylor & Francis Group
AN AUERBACH BOOK

Software Testing

A Craftsman's Approach

Fourth Edition

Software Testing

A Craftsman's Approach

Fourth Edition

Paul C. Jorgensen

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business
AN AUERBACH BOOK

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

CRC Press is an imprint of Taylor & Francis Group, an Informa business

No claim to original U.S. Government works
Version Date: 20130815

International Standard Book Number-13: 978-1-4665-6069-7 (eBook - PDF)

This book contains information obtained from authentic and highly regarded sources. Reasonable efforts have been made to publish reliable data and information, but the author and publisher cannot assume responsibility for the validity of all materials or the consequences of their use. The authors and publishers have attempted to trace the copyright holders of all material reproduced in this publication and apologize to copyright holders if permission to publish in this form has not been obtained. If any copyright material has not been acknowledged please write and let us know so we may rectify in any future reprint.

Except as permitted under U.S. Copyright Law, no part of this book may be reprinted, reproduced, transmitted, or utilized in any form by any electronic, mechanical, or other means, now known or hereafter invented, including photocopying, microfilming, and recording, or in any information storage or retrieval system, without written permission from the publishers.

For permission to photocopy or use material electronically from this work, please access www.copyright.com (http://www.copyright.com/) or contact the Copyright Clearance Center, Inc. (CCC), 222 Rosewood Drive, Danvers, MA 01923, 978-750-8400. CCC is a not-for-profit organization that provides licenses and registration for a variety of users. For organizations that have been granted a photocopy license by the CCC, a separate system of payment has been arranged.

Trademark Notice: Product or corporate names may be trademarks or registered trademarks, and are used only for identification and explanation without intent to infringe.

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

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

To Carol, Kirsten, and Katia

Contents

Preface to the Fourth Edition .... **xix**
Preface to the Third Edition .... **xxi**
Preface to the Second Edition .... **xxiii**
Preface to the First Edition .... **xxv**
Author .... **xxvii**
Abstract .... **xxix**

PART I A MATHEMATICAL CONTEXT

1 A Perspective on Testing **3**
- 1.1 Basic Definitions 3
- 1.2 Test Cases 4
- 1.3 Insights from a Venn Diagram 5
- 1.4 Identifying Test Cases 6
  - 1.4.1 Specification-Based Testing 7
  - 1.4.2 Code-Based Testing 8
  - 1.4.3 Specification-Based versus Code-Based Debate 8
- 1.5 Fault Taxonomies 9
- 1.6 Levels of Testing 12
- References 13

2 Examples **15**
- 2.1 Generalized Pseudocode 15
- 2.2 The Triangle Problem 17
  - 2.2.1 Problem Statement 17
  - 2.2.2 Discussion 18
  - 2.2.3 Traditional Implementation 18
  - 2.2.4 Structured Implementations 21
- 2.3 The NextDate Function 23
  - 2.3.1 Problem Statement 23
  - 2.3.2 Discussion 23
  - 2.3.3 Implementations 24

2.4 The Commission Problem 26
  2.4.1 Problem Statement 26
  2.4.2 Discussion 27
  2.4.3 Implementation 27
2.5 The SATM System 28
  2.5.1 Problem Statement 29
  2.5.2 Discussion 30
2.6 The Currency Converter 30
2.7 Saturn Windshield Wiper Controller 31
2.8 Garage Door Opener 31
References 33

3 Discrete Math for Testers 35
3.1 Set Theory 35
  3.1.1 Set Membership 36
  3.1.2 Set Definition 36
  3.1.3 The Empty Set 37
  3.1.4 Venn Diagrams 37
  3.1.5 Set Operations 38
  3.1.6 Set Relations 40
  3.1.7 Set Partitions 40
  3.1.8 Set Identities 41
3.2 Functions 42
  3.2.1 Domain and Range 42
  3.2.2 Function Types 43
  3.2.3 Function Composition 44
3.3 Relations 45
  3.3.1 Relations among Sets 45
  3.3.2 Relations on a Single Set 46
3.4 Propositional Logic 47
  3.4.1 Logical Operators 48
  3.4.2 Logical Expressions 49
  3.4.3 Logical Equivalence 49
3.5 Probability Theory 50
Reference 52

4 Graph Theory for Testers 53
4.1 Graphs 53
  4.1.1 Degree of a Node 54
  4.1.2 Incidence Matrices 55
  4.1.3 Adjacency Matrices 56
  4.1.4 Paths 56
  4.1.5 Connectedness 57
  4.1.6 Condensation Graphs 58
  4.1.7 Cyclomatic Number 58
4.2 Directed Graphs 59

4.2.1 Indegrees and Outdegrees 60
4.2.2 Types of Nodes 60
4.2.3 Adjacency Matrix of a Directed Graph 61
4.2.4 Paths and Semipaths 62
4.2.5 Reachability Matrix 62
4.2.6 *n*-Connectedness 63
4.2.7 Strong Components 64
4.3 Graphs for Testing 65
4.3.1 Program Graphs 65
4.3.2 Finite State Machines 66
4.3.3 Petri Nets 68
4.3.4 Event-Driven Petri Nets 70
4.3.5 StateCharts 73
References 75

PART II UNIT TESTING

5 Boundary Value Testing 79
5.1 Normal Boundary Value Testing 80
5.1.1 Generalizing Boundary Value Analysis 81
5.1.2 Limitations of Boundary Value Analysis 82
5.2 Robust Boundary Value Testing 82
5.3 Worst-Case Boundary Value Testing 83
5.4 Special Value Testing 84
5.5 Examples 85
5.5.1 Test Cases for the Triangle Problem 85
5.5.2 Test Cases for the NextDate Function 86
5.5.3 Test Cases for the Commission Problem 91
5.6 Random Testing 93
5.7 Guidelines for Boundary Value Testing 94

6 Equivalence Class Testing 99
6.1 Equivalence Classes 99
6.2 Traditional Equivalence Class Testing 100
6.3 Improved Equivalence Class Testing 101
6.3.1 Weak Normal Equivalence Class Testing 102
6.3.2 Strong Normal Equivalence Class Testing 102
6.3.3 Weak Robust Equivalence Class Testing 103
6.3.4 Strong Robust Equivalence Class Testing 104
6.4 Equivalence Class Test Cases for the Triangle Problem 105
6.5 Equivalence Class Test Cases for the NextDate Function 107
6.5.1 Equivalence Class Test Cases 109
6.6 Equivalence Class Test Cases for the Commission Problem 111
6.7 Edge Testing 113
6.8 Guidelines and Observations 113
References 115

7 Decision Table–Based Testing....117
7.1 Decision Tables....117
7.2 Decision Table Techniques....118
7.3 Test Cases for the Triangle Problem....122
7.4 Test Cases for the NextDate Function....123
  7.4.1 First Try....123
  7.4.2 Second Try....124
  7.4.3 Third Try....126
7.5 Test Cases for the Commission Problem....127
7.6 Cause-and-Effect Graphing....128
7.7 Guidelines and Observations....130
References....131

8 Path Testing....133
8.1 Program Graphs....133
  8.1.1 Style Choices for Program Graphs....133
8.2 DD-Paths....136
8.3 Test Coverage Metrics....138
  8.3.1 Program Graph–Based Coverage Metrics....138
  8.3.2 E.F. Miller's Coverage Metrics....139
    8.3.2.1 Statement Testing....139
    8.3.2.2 DD-Path Testing....140
    8.3.2.3 Simple Loop Coverage....140
    8.3.2.4 Predicate Outcome Testing....140
    8.3.2.5 Dependent Pairs of DD-Paths....141
    8.3.2.6 Complex Loop Coverage....141
    8.3.2.7 Multiple Condition Coverage....142
    8.3.2.8 "Statistically Significant" Coverage....142
    8.3.2.9 All Possible Paths Coverage....142
  8.3.3 A Closer Look at Compound Conditions....142
    8.3.3.1 Boolean Expression (per Chilenski)....142
    8.3.3.2 Condition (per Chilenski)....143
    8.3.3.3 Coupled Conditions (per Chilenski)....143
    8.3.3.4 Masking Conditions (per Chilenski)....144
    8.3.3.5 Modified Condition Decision Coverage....144
  8.3.4 Examples....145
    8.3.4.1 Condition with Two Simple Conditions....145
    8.3.4.2 Compound Condition from NextDate....146
    8.3.4.3 Compound Condition from the Triangle Program....147
  8.3.5 Test Coverage Analyzers....149
8.4 Basis Path Testing....149
  8.4.1 McCabe's Basis Path Method....150
  8.4.2 Observations on McCabe's Basis Path Method....152
  8.4.3 Essential Complexity....154
8.5 Guidelines and Observations....156
References....158

9 Data Flow Testing 159
9.1 Define/Use Testing 160
9.1.1 Example 161
9.1.2 Du-paths for Stocks 164
9.1.3 Du-paths for Locks 164
9.1.4 Du-paths for totalLocks 168
9.1.5 Du-paths for Sales 169
9.1.6 Du-paths for Commission 170
9.1.7 Define/Use Test Coverage Metrics 170
9.1.8 Define/Use Testing for Object-Oriented Code 172
9.2 Slice-Based Testing 172
9.2.1 Example 175
9.2.2 Style and Technique 179
9.2.3 Slice Splicing 181
9.3 Program Slicing Tools 182
References 183

10 Retrospective on Unit Testing 185
10.1 The Test Method Pendulum 186
10.2 Traversing the Pendulum 188
10.3 Evaluating Test Methods 193
10.4 Insurance Premium Case Study 195
10.4.1 Specification-Based Testing 195
10.4.2 Code-Based Testing 199
10.4.2.1 Path-Based Testing 199
10.4.2.2 Data Flow Testing 200
10.4.2.3 Slice Testing 201
10.5 Guidelines 202
References 203

PART III BEYOND UNIT TESTING

11 Life Cycle–Based Testing 207
11.1 Traditional Waterfall Testing 207
11.1.1 Waterfall Testing 209
11.1.2 Pros and Cons of the Waterfall Model 209
11.2 Testing in Iterative Life Cycles 210
11.2.1 Waterfall Spin-Offs 210
11.2.2 Specification-Based Life Cycle Models 212
11.3 Agile Testing 214
11.3.1 Extreme Programming 215
11.3.2 Test-Driven Development 215
11.3.3 Scrum 216
11.4 Agile Model–Driven Development 218
11.4.1 Agile Model–Driven Development 218
11.4.2 Model–Driven Agile Development 218
References 219

12 Model-Based Testing 221
12.1 Testing Based on Models 221
12.2 Appropriate Models 222
12.2.1 Peterson's Lattice 222
12.2.2 Expressive Capabilities of Mainline Models 224
12.2.3 Modeling Issues 224
12.2.4 Making Appropriate Choices 225
12.3 Commercial Tool Support for Model-Based Testing 226
References 227

13 Integration Testing 229
13.1 Decomposition-Based Integration 229
13.1.1 Top–Down Integration 232
13.1.2 Bottom–Up Integration 234
13.1.3 Sandwich Integration 235
13.1.4 Pros and Cons 235
13.2 Call Graph–Based Integration 236
13.2.1 Pairwise Integration 237
13.2.2 Neighborhood Integration 237
13.2.3 Pros and Cons 240
13.3 Path-Based Integration 241
13.3.1 New and Extended Concepts 241
13.3.2 MM-Path Complexity 243
13.3.3 Pros and Cons 244
13.4 Example: integrationNextDate 244
13.4.1 Decomposition-Based Integration 245
13.4.2 Call Graph–Based Integration 245
13.4.3 MM-Path-Based Integration 250
13.5 Conclusions and Recommendations 250
References 251

14 System Testing 253
14.1 Threads 253
14.1.1 Thread Possibilities 254
14.1.2 Thread Definitions 255
14.2 Basis Concepts for Requirements Specification 256
14.2.1 Data 256
14.2.2 Actions 257
14.2.3 Devices 257
14.2.4 Events 258
14.2.5 Threads 259
14.2.6 Relationships among Basis Concepts 259
14.3 Model-Based Threads 259
14.4 Use Case–Based Threads 264
14.4.1 Levels of Use Cases 264
14.4.2 An Industrial Test Execution System 265
14.4.3 System-Level Test Cases 268

14.4.4 Converting Use Cases to Event-Driven Petri Nets 269
14.4.5 Converting Finite State Machines to Event-Driven Petri Nets 270
14.4.6 Which View Best Serves System Testing? 271
14.5 Long versus Short Use Cases 271
14.6 How Many Use Cases? 274
14.6.1 Incidence with Input Events 274
14.6.2 Incidence with Output Events 275
14.6.3 Incidence with All Port Events 277
14.6.4 Incidence with Classes 277
14.7 Coverage Metrics for System Testing 277
14.7.1 Model-Based System Test Coverage 277
14.7.2 Specification-Based System Test Coverage 278
14.7.2.1 Event-Based Thread Testing 278
14.7.2.2 Port-Based Thread Testing 279
14.8 Supplemental Approaches to System Testing 279
14.8.1 Operational Profiles 279
14.8.2 Risk-Based Testing 282
14.9 Nonfunctional System Testing 284
14.9.1 Stress Testing Strategies 284
14.9.1.1 Compression 284
14.9.1.2 Replication 285
14.9.2 Mathematical Approaches 286
14.9.2.1 Queuing Theory 286
14.9.2.2 Reliability Models 286
14.9.2.3 Monte Carlo Testing 286
14.10 Atomic System Function Testing Example 287
14.10.1 Identifying Input and Output Events 289
14.10.2 Identifying Atomic System Functions 290
14.10.3 Revised Atomic System Functions 291
References 292

15 Object-Oriented Testing 295
15.1 Issues in Testing Object-Oriented Software 295
15.1.1 Units for Object-Oriented Testing 295
15.1.2 Implications of Composition and Encapsulation 296
15.1.3 Implications of Inheritance 297
15.1.4 Implications of Polymorphism 299
15.1.5 Levels of Object-Oriented Testing 299
15.1.6 Data Flow Testing for Object-Oriented Software 299
15.2 Example: ooNextDate 300
15.2.1 Class: CalendarUnit 301
15.2.2 Class: testIt 302
15.2.3 Class: Date 302
15.2.4 Class: Day 303
15.2.5 Class: Month 303
15.2.6 Class: Year 304
15.3 Object-Oriented Unit Testing 304

15.3.1 Methods as Units 305
15.3.2 Classes as Units 305
15.3.2.1 Pseudocode for Windshield Wiper Class 306
15.3.2.2 Unit Testing for Windshield Wiper Class 306
15.4 Object-Oriented Integration Testing 311
15.4.1 UML Support for Integration Testing 311
15.4.2 MM-Paths for Object-Oriented Software 313
15.4.3 A Framework for Object-Oriented Data Flow Testing 318
15.4.3.1 Event-/Message-Driven Petri Nets 318
15.4.3.2 Inheritance-Induced Data Flow 320
15.4.3.3 Message-Induced Data Flow 320
15.4.3.4 Slices? 321
15.5 Object-Oriented System Testing 321
15.5.1 Currency Converter UML Description 321
15.5.1.1 Problem Statement 321
15.5.1.2 System Functions 321
15.5.1.3 Presentation Layer 322
15.5.1.4 High-Level Use Cases 322
15.5.1.5 Essential Use Cases 323
15.5.1.6 Detailed GUI Definition 325
15.5.1.7 Expanded Essential Use Cases 326
15.5.1.8 Real Use Cases 328
15.5.2 UML-Based System Testing 328
15.5.3 StateChart-Based System Testing 329
References 330

16 Software Complexity 331
16.1 Unit-Level Complexity 331
16.1.1 Cyclomatic Complexity 332
16.1.1.1 “Cattle Pens” and Cyclomatic Complexity 332
16.1.1.2 Node Outdegrees and Cyclomatic Complexity 332
16.1.1.3 Decisional Complexity 334
16.1.2 Computational Complexity 335
16.1.2.1 Halstead’s Metrics 335
16.1.2.2 Example: Day of Week with Zeller’s Congruence 336
16.2 Integration-Level Complexity 338
16.2.1 Integration-Level Cyclomatic Complexity 339
16.2.2 Message Traffic Complexity 340
16.3 Software Complexity Example 341
16.3.1 Unit-Level Cyclomatic Complexity 344
16.3.2 Message Integration-Level Cyclomatic Complexity 344
16.4 Object-Oriented Complexity 344
16.4.1 WMC—Weighted Methods per Class 344
16.4.2 DIT—Depth of Inheritance Tree 345
16.4.3 NOC—Number of Child Classes 345
16.4.4 CBO—Coupling between Classes 345
16.4.5 RFC—Response for Class 345

        16.4.6 LCOM—Lack of Cohesion on Methods 345
    16.5 System-Level Complexity 345
    Reference 348

17 Model-Based Testing for Systems of Systems 349
    17.1 Characteristics of Systems of Systems 350
    17.2 Sample Systems of Systems 351
        17.2.1 The Garage Door Controller (Directed) 351
        17.2.2 Air Traffic Management System (Acknowledged) 352
        17.2.3 The GVSU Snow Emergency System (Collaborative) 353
        17.2.4 The Rock Solid Federal Credit Union (Virtual) 354
    17.3 Software Engineering for Systems of Systems 354
        17.3.1 Requirements Elicitation 355
        17.3.2 Specification with a Dialect of UML: SysML 355
            17.3.2.1 Air Traffic Management System Classes 355
            17.3.2.2 Air Traffic Management System Use Cases and Sequence Diagrams 356
        17.3.3 Testing 359
    17.4 Communication Primitives for Systems of Systems 359
        17.4.1 ESML Prompts as Petri Nets 359
            17.4.1.1 Petri Net Conflict 360
            17.4.1.2 Petri Net Interlock 360
            17.4.1.3 Enable, Disable, and Activate 361
            17.4.1.4 Trigger 361
            17.4.1.5 Suspend and Resume 361
        17.4.2 New Prompts as Swim Lane Petri Nets 363
            17.4.2.1 Request 363
            17.4.2.2 Accept 363
            17.4.2.3 Reject 364
            17.4.2.4 Postpone 364
            17.4.2.5 Swim Lane Description of the November 1993 Incident 364
    17.5 Effect of Systems of Systems Levels on Prompts 365
        17.5.1 Directed and Acknowledged Systems of Systems 366
        17.5.2 Collaborative and Virtual Systems of Systems 367
    References 367

18 Exploratory Testing 369
    18.1 Exploratory Testing Explored 369
    18.2 Exploring a Familiar Example 371
    18.3 Observations and Conclusions 373
    References 374

19 Test-Driven Development 375
    19.1 Test-Then-Code Cycles 375
    19.2 Automated Test Execution (Testing Frameworks) 384
    19.3 Java and JUnit Example 385
        19.3.1 Java Source Code 385
        19.3.2 JUnit Test Code 387

19.4 Remaining Questions ... 388
19.4.1 Specification or Code Based? ... 388
19.4.2 Configuration Management? ... 388
19.4.3 Granularity? ... 388
19.5 Pros, Cons, and Open Questions of TDD ... 390
19.6 Retrospective on MDD versus TDD ... 390

20 A Closer Look at All Pairs Testing ... 395
20.1 The All Pairs Technique ... 395
20.1.1 Program Inputs ... 396
20.1.2 Independent Variables ... 398
20.1.3 Input Order ... 399
20.1.4 Failures Due Only to Pairs of Inputs ... 403
20.2 A Closer Look at the NIST Study ... 404
20.3 Appropriate Applications for All Pairs Testing ... 404
20.4 Recommendations for All Pairs Testing ... 405
References ... 406

21 Evaluating Test Cases ... 407
21.1 Mutation Testing ... 407
21.1.1 Formalizing Program Mutation ... 408
21.1.2 Mutation Operators ... 409
21.1.2.1 isLeap Mutation Testing ... 410
21.1.2.2 isTriangle Mutation Testing ... 411
21.1.2.3 Commission Mutation Testing ... 412
21.2 Fuzzing ... 414
21.3 Fishing Creel Counts and Fault Insertion ... 414
References ... 415

22 Software Technical Reviews ... 417
22.1 Economics of Software Reviews ... 417
22.2 Roles in a Review ... 419
22.2.1 Producer ... 419
22.2.2 Review Leader ... 419
22.2.3 Recorder ... 420
22.2.4 Reviewer ... 420
22.2.5 Role Duplication ... 420
22.3 Types of Reviews ... 420
22.3.1 Walkthroughs ... 421
22.3.2 Technical Inspections ... 421
22.3.3 Audits ... 421
22.3.4 Comparison of Review Types ... 422
22.4 Contents of an Inspection Packet ... 422
22.4.1 Work Product Requirements ... 422
22.4.2 Frozen Work Product ... 422
22.4.3 Standards and Checklists ... 423
22.4.4 Review Issues Spreadsheet ... 423
22.4.5 Review Reporting Forms ... 424

22.4.6 Fault Severity Levels 425
22.4.7 Review Report Outline 425
22.5 An Industrial-Strength Inspection Process 426
22.5.1 Commitment Planning 426
22.5.2 Reviewer Introduction 427
22.5.3 Preparation 427
22.5.4 Review Meeting 428
22.5.5 Report Preparation 428
22.5.6 Disposition 428
22.6 Effective Review Culture 429
22.6.1 Etiquette 429
22.6.2 Management Participation in Review Meetings 429
22.6.3 A Tale of Two Reviews 430
22.6.3.1 A Pointy-Haired Supervisor Review 430
22.6.3.2 An Ideal Review 430
22.7 Inspection Case Study 431
Reference 432

23 Epilogue: Software Testing Excellence 433
23.1 Craftsmanship 433
23.2 Best Practices of Software Testing 434
23.3 My Top 10 Best Practices for Software Testing Excellence 435
23.3.1 Model-Driven Agile Development 435
23.3.2 Careful Definition and Identification of Levels of Testing 435
23.3.3 System-Level Model-Based Testing 436
23.3.4 System Testing Extensions 436
23.3.5 Incidence Matrices to Guide Regression Testing 436
23.3.6 Use of MM-Paths for Integration Testing 436
23.3.7 Intelligent Combination of Specification-Based and Code-Based Unit-Level Testing 436
23.3.8 Code Coverage Metrics Based on the Nature of Individual Units 437
23.3.9 Exploratory Testing during Maintenance 437
23.3.10 Test-Driven Development 437
23.4 Mapping Best Practices to Diverse Projects 437
23.4.1 A Mission-Critical Project 438
23.4.2 A Time-Critical Project 438
23.4.3 Corrective Maintenance of Legacy Code 438
References 438

Appendix: Complete Technical Inspection Packet 439

Preface to the Fourth Edition

Software Testing: A Craftsman's Approach, Fourth Edition

Software Testing: A Craftsman's Approach extends an 18-year emphasis on model-based testing with deeper coverage of path testing and four new chapters. The book has evolved over three editions and 18 years of classroom and industrial use. It presents a strong combination of theory and practice, with well chosen, but easily understood, examples. In addition, much of the material from the Third Edition has been merged, reorganized, and made more concise. Much of the material on object-oriented software testing has been unified with procedural software testing into a coherent whole. In addition, the chapter on path testing contains new material on complex condition testing and Modified Condition Decision Coverage as mandated by Federal Aviation Authority and US Department of Defense standards.

Here is a brief summary of the new chapters:

- Software reviews, especially technical inspections (Chapter 22). This is really recognized as "static testing," whereas the first three editions focused on "dynamic testing," in which program code executes carefully selected test cases. The material in this chapter is derived from 20 years of industrial/practical experience in a development organization that had an extremely mature review process.
- An Appendix that contains a full set of documents that are appropriate for an industrial-strength technical inspection of a set of Use Cases (from UML) that respond to a typical customer specification. The Appendix includes a Use Case Standard, a definition of Use Case Fault severities, a technical inspection checklist of potential problems, and typical reviewer and final report forms.
- Testing Systems of Systems (Chapter 17). Systems of systems is a relatively new (since 1999) topic. The practitioner community is just now following the lead of a few university researchers, mostly in the area of how to specify a system of systems. This chapter introduces "Swim Lane Event-Driven Petri Nets," which have an expressive power very similar to the world-famous Statecharts. This makes model-based testing possible for systems of systems.
- Software complexity (Chapter 16). Most texts only consider cyclomatic (aka McCabe) complexity at the unit level. This chapter extends unit level complexity in two ways and then adds two views of integration level complexity. There is a short treatment of the complexities due to object-oriented programming and also to system level testing. At all levels, complexity is

an important way to improve designs, coding, testing, and maintenance. Having a coherent presentation of software complexity enhances each of these activities.

- Evaluating test cases (Chapter 21). This new chapter considers a difficult question: how can a set of test cases be evaluated? Test coverage metrics are a long-accepted answer, but there will always be a level of uncertainty. The old Roman question of who guards the guards/custodians is extended to who tests the tests. Mutation testing has been an answer for decades, and its contribution is covered in this chapter. Two other approaches are offered: fuzzing and fault insertion.

After 47 years as a software developer and university professor, I have knowledge of software testing that is both deep and extensive. My university education was in mathematics and computer science, and that, together with 20 years of industrial software development and management experience, puts me in a strong position to codify and improve the teaching and understanding of software testing. I keep gaining new insights, often when I teach out of the book. I see this book as my contribution to the field of software engineering. Finally, I thank three of my colleagues—Dr. Roger Ferguson, Dr. Jagadeesh Nandigam, and Dr. Christian Trefftz—for their help in the chapters on object-oriented testing. *Wopila tanka!*

Paul C. Jorgensen
Rockford, Michigan

Preface to the Third Edition

Software Testing: A Craftsman's Approach, Third Edition

Five years have passed since the Second Edition appeared, and software testing has seen a *renaissance* of renewed interest and technology. The biggest change is the growing prominence and acceptance of agile programming. The various flavors of agile programming have interesting, and serious, implications for software testing. Almost as a reaction to agile programming, the model-based approaches to both development and testing have also gained adherents. Part VI of the Third Edition analyzes the testing methods that are gaining acceptance in the new millennium. Except for correction of errors, the first five parts are generally unchanged.

Over the years, several readers have been enormously helpful in pointing out defects—both typos and more serious faults. Particular thanks go to Neil Bitzenhofer (St. Paul, Minnesota), Han Ke (Beijing), Jacob Minidor (Illinois), and Jim Davenport (Ohio). In addition, many of my graduate students have made helpful suggestions in the past fifteen years. Special mention goes to two valued colleagues, Dr. Roger Ferguson and Dr. Christian Trefftz, for their help.

This book is now used as a text for formal courses in software testing in dozens of countries. To support both instructors and students, I have added more exercises, and faculty adopters can get a package of support materials from CRC Press.

I, too, have changed in the past five years. I now count several Lakota people among my friends. There is an interesting bit of their influence in Chapter 23, and just one word here—Hecatuyelo!

Paul C. Jorgensen
Rockford, Michigan

Preface to the Second Edition

Software Testing: A Craftsman's Approach, Second Edition

Seven years have passed since I wrote the preface to the First Edition. Much has happened in that time, hence this new edition. The most significant change is the dominance of the unified modeling language (UML) as a standard for the specification and design of object-oriented software. This is reflected in the five chapters in Part V that deal with testing object-oriented software. Nearly all of the material in Part V is UML-based.

The second major change is that the Pascal examples of the First Edition are replaced by a language neutral pseudo-code. The examples have been elaborated; Visual Basic executable modules available on the CRC Press website support them. Several new examples illustrate some of the issues of testing object-oriented software. There are dozens of other changes: an improved description of equivalence class testing, a continuing case study, and more details about integration testing are the most important additions.

I am flattered that the First Edition is one of the primary references on software testing in the trial-use standard "Software Engineering Body of Knowledge" jointly produced by the ACM and IEEE Computer Society (www.swebok.org). This recognition makes the problem of uncorrected mistakes more of a burden. Prof. Gi H. Kwon of Kyonggi University in South Korea sent me a list of 38 errors in the First Edition, and students in my graduate class on software testing have gleefully contributed others. There is a nice analogy with testing here: I have fixed all the known errors, and my editor tells me it is time to stop looking for others. If you find any, please let me know—they are my responsibility. My email address is jorgensp@gvsu.edu.

I need to thank Jerry Papule and Helena Redcap at CRC Press for their patience. I also want to thank my friend and colleague, Prof. Roger Ferguson, for his continued help with the new material in Part V, especially the continuing object-oriented calendar example. In a sense, Roger has been a tester of numerous drafts of chapters sixteen through twenty.

Paul C. Jorgensen
Rockford, Michigan

Preface to the First Edition

Software Testing: A Craftsman's Approach

We huddled around the door to the conference room, each taking a turn looking through the small window. Inside, a recently hired software designer had spread out source listings on the conference table, and carefully passed a crystal hanging from a long chain over the source code. Every so often, the designer marked a circle in red on the listing. Later, one of my colleagues asked the designer what he had been doing in the conference room. The nonchalant reply: "Finding the bugs in my program." This is a true story, it happened in the mid-1980s when people had high hopes for hidden powers in crystals.

In a sense, the goal of this book is to provide you with a better set of crystals. As the title suggests, I believe that software (and system) testing is a craft, and I think I have some mastery of that craft. Out of a score of years developing telephone switching systems, I spent about a third of that time on testing: defining testing methodologies and standards, coordinating system testing for a major international telephone toll switch, specifying and helping build two test execution tools (now we would call them CASE tools), and a fair amount of plain, hands-on testing. For the past seven years, I have been teaching software engineering at the university graduate level. My academic research centers on specification and testing. Adherents to the Oxford Method claim that you never really learn something until you have to teach it—I think they're right. The students in my graduate course on testing are all full-time employees in local industries. Believe me, they keep you honest. This book is an outgrowth of my lectures and projects in that class.

I think of myself as a software engineer, but when I compare the level of precision and depth of knowledge prevalent in my field to those of more traditional engineering disciplines, I am uncomfortable with the term. A colleague and I were returning to our project in Italy when Myers' book, *The Art of Software Testing*, first came out. On the way to the airport, we stopped by the MIT bookstore and bought one of the early copies. In the intervening 15 years, I believe we have moved from an art to a craft. I had originally planned to title this book *The Craft of Software Testing*, but as I neared the final chapters, another book with that title appeared. Maybe that's confirmation that software testing is becoming a craft. There's still a way to go before it is a science.

Part of any craft is knowing the capabilities and limitations of both the tools and the medium. A good woodworker has a variety of tools and, depending on the item being made and the wood being used, knows which tool is the most appropriate. Of all the phases of the traditional Waterfall Model of the software development life cycle, testing is the most amenable to precise analysis.

Elevating software testing to a craft requires that the testing craftsperson know the basic tools. To this end, Chapters 3 and 4 provide mathematical background that is used freely in the remainder of the text.

Mathematics is a descriptive device that helps us better understand software to be tested. Precise notation, by itself, is not enough. We must also have good technique and judgment to identify appropriate testing methods and to apply them well. These are the goals of Parts II and III, which deal with fundamental functional and structural testing techniques. These techniques are applied to the continuing examples, which are described in Chapter 2. In Part IV, we apply these techniques to the integration and system levels of testing, and to object-oriented testing. At these levels, we are more concerned with what to test than how to test it, so the discussion moves toward requirements specification. Part IV concludes with an examination of testing interactions in a software controlled system, with a short discussion of client–server systems.

It is ironic that a book on testing contains faults. Despite the conscientious efforts of reviewers and editors, I am confident that faults persist in the text. Those that remain are my responsibility.

In 1977, I attended a testing seminar given by Edward Miller, who has since become one of the luminaries in software testing circles. In that seminar, Miller went to great lengths to convince us that testing need not be bothersome drudgery, but can be a very creative, interesting part of software development. My goal for you, the reader of this book, is that you will become a testing craftsperson, and that you will be able to derive the sense of pride and pleasure that a true craftsperson realizes from a job well done.

Paul C. Jorgensen
Rockford, Michigan

Author

Paul Jorgensen, PhD, spent 20 years of his first career in all phases of software development for telephone switching systems. He began his university career in 1986 teaching graduate courses in software engineering at Arizona State University and since 1988 at Grand Valley State University where he is a full professor. His consulting business, Software Paradigms, hibernates during the academic year and emerges for a short time in the warmer months. He has served on major CODASYL, ACM, and IEEE standards committees, and in 2012, his university recognized his lifetime accomplishments with its “Distinguished Contribution to a Discipline Award.”

In addition to the fourth edition of his software testing book, he is also the author of *Modeling Software Behavior: A Craftsman’s Approach*. He is a coauthor of *Mathematics for Data Processing* (McGraw-Hill, 1970) and *Structured Methods—Merging Models*, Techniques, and CASE (McGraw-Hill, 1993). More recently, Dr. Jorgensen has been involved with the International Software Testing Certification Board (ISTQB) where he is a coauthor of the Advanced Level Syllabi and served as the vice-chair of the ISTQB Glossary Working Group.

Living and working in Italy for three years made him a confirmed “Italophile.” He, his wife Carol, and daughters Kirsten and Katia have visited friends there several times. In the Michigan summer, he sails his Rebel when he can. Paul and Carol have volunteered at the Porcupine School on the Pine Ridge Reservation in South Dakota every summer since 2000. His email address is jorgensp@gvsu.edu.

Abstract

Since the last publication of this international bestseller, there has been a renewed interest in model-based testing. This Fourth Edition of *Software Testing: A Craftsman's Approach* continues and solidifies this emphasis with its careful blend of theory and well-chosen examples. The book has evolved over three editions and 18 years of classroom and industrial use. Much of the material from the Third Edition has been reorganized and consolidated, making room for important emerging topics. The chapter on path testing contains new material on complex condition testing and modified condition decision coverage as mandated by Federal Aviation Authority and US Department of Defense standards.

There are four chapters of new material in the Fourth Edition. The chapter on software reviews emphasizes technical inspections and is supplemented by an appendix with a full package of documents required for a sample use case technical inspection. It contains lessons learned from 15 years of industrial practice. The treatment of systems of systems introduces an innovative approach that merges the event-driven Petri nets from the earlier editions with the "swim lane" concept from the unified modeling language (UML) that permits model-based testing for four levels of interaction among constituents in a system of systems. Swim lane Petri nets exactly represent the issues of concurrency that previously needed the orthogonal regions of StateCharts for correct description. The chapter on software complexity presents a coherent, graph theory–based view of complexity across the traditional three levels of testing, unit, integration, and system. Finally, the last new chapter considers how a set of test cases might be evaluated and presents mutation testing and two alternatives.

Thoroughly revised, updated, and extended, *Software Testing: A Craftsman's Approach, Fourth Edition* is sure to become a standard reference for those who need to stay up-to-date with the most recent ideas in software testing. It also serves as a strong textbook for university courses in software testing. It continues to be a valuable reference for software testers, developers, engineers, and researchers. A full set of support materials is available to faculty who adopt the Fourth Edition as a textbook.

A MATHEMATICAL CONTEXT

I

Chapter 1

A Perspective on Testing

Why do we test? The two main reasons are to make a judgment about quality or acceptability and to discover problems. We test because we know that we are fallible—this is especially true in the domain of software and software-controlled systems. The goal of this chapter is to create a framework within which we can examine software testing.

1.1 Basic Definitions

Much of testing literature is mired in confusing (and sometimes inconsistent) terminology, probably because testing technology has evolved over decades and via scores of writers. The International Software Testing Qualification Board (ISTQB) has an extensive glossary of testing terms (see the website http://www.istqb.org/downloads/glossary.html). The terminology here (and throughout this book) is compatible with the ISTQB definitions, and they, in turn, are compatible with the standards developed by the Institute of Electronics and Electrical Engineers (IEEE) Computer Society (IEEE, 1983). To get started, here is a useful progression of terms.

Error—People make errors. A good synonym is *mistake*. When people make mistakes while coding, we call these mistakes *bugs*. Errors tend to propagate; a requirements error may be magnified during design and amplified still more during coding.

Fault—A fault is the result of an error. It is more precise to say that a fault is the representation of an error, where representation is the mode of expression, such as narrative text, Unified Modeling Language diagrams, hierarchy charts, and source code. *Defect* (see the ISTQB Glossary) is a good synonym for fault, as is *bug*. Faults can be elusive. An error of omission results in a fault in which something is missing that should be present in the representation. This suggests a useful refinement; we might speak of faults of commission and faults of omission. A fault of commission occurs when we enter something into a representation that is incorrect. Faults of omission occur when we fail to enter correct information. Of these two types, faults of omission are more difficult to detect and resolve.

Failure—A failure occurs when the code corresponding to a fault executes. Two subtleties arise here: one is that failures only occur in an executable representation, which is usually taken to be source code, or more precisely, loaded object code; the second subtlety is that this definition

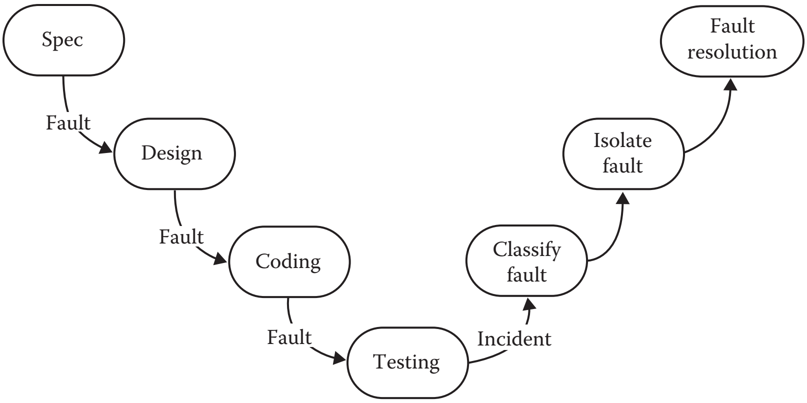


Figure 1.1 A testing life cycle.

relates failures only to faults of commission. How can we deal with failures that correspond to faults of omission? We can push this still further: what about faults that never happen to execute, or perhaps do not execute for a long time? Reviews (see Chapter 22) prevent many failures by finding faults; in fact, well-done reviews can find faults of omission.

Incident—When a failure occurs, it may or may not be readily apparent to the user (or customer or tester). An incident is the symptom associated with a failure that alerts the user to the occurrence of a failure.

Test—Testing is obviously concerned with errors, faults, failures, and incidents. A test is the act of exercising software with test cases. A test has two distinct goals: to find failures or to demonstrate correct execution.

Test case—A test case has an identity and is associated with a program behavior. It also has a set of inputs and expected outputs.

Figure 1.1 portrays a life cycle model for testing. Notice that, in the development phases, three opportunities arise for errors to be made, resulting in faults that may propagate through the remainder of the development process. The fault resolution step is another opportunity for errors (and new faults). When a fix causes formerly correct software to misbehave, the fix is deficient. We will revisit this when we discuss regression testing.

From this sequence of terms, we see that test cases occupy a central position in testing. The process of testing can be subdivided into separate steps: test planning, test case development, running test cases, and evaluating test results. The focus of this book is how to identify useful sets of test cases.

1.2 Test Cases

The essence of software testing is to determine a set of test cases for the item to be tested. A test case is (or should be) a recognized work product. A complete test case will contain a test case identifier, a brief statement of purpose (e.g., a business rule), a description of preconditions, the actual test case inputs, the expected outputs, a description of expected postconditions, and an execution history. The execution history is primarily for test management use—it may contain the date when the test was run, the person who ran it, the version on which it was run, and the pass/fail result.

The output portion of a test case is frequently overlooked, which is unfortunate because this is often the hard part. Suppose, for example, you were testing software that determines an optimal route for an aircraft, given certain Federal Aviation Administration air corridor constraints and the weather data for a flight day. How would you know what the optimal route really is? Various responses can address this problem. The academic response is to postulate the existence of an oracle who "knows all the answers." One industrial response to this problem is known as reference testing, where the system is tested in the presence of expert users. These experts make judgments as to whether outputs of an executed set of test case inputs are acceptable.

Test case execution entails establishing the necessary preconditions, providing the test case inputs, observing the outputs, comparing these with the expected outputs, and then ensuring that the expected postconditions exist to determine whether the test passed. From all of this, it becomes clear that test cases are valuable—at least as valuable as source code. Test cases need to be developed, reviewed, used, managed, and saved.

1.3 Insights from a Venn Diagram

Testing is fundamentally concerned with behavior, and behavior is orthogonal to the code-based view common to software (and system) developers. A quick distinction is that the code-based view focuses on what it *is* and the behavioral view considers what it *does*. One of the continuing sources of difficulty for testers is that the base documents are usually written by and for developers; the emphasis is therefore on code-based, instead of behavioral, information. In this section, we develop a simple Venn diagram that clarifies several nagging questions about testing.

Consider a universe of program behaviors. (Notice that we are forcing attention on the essence of testing.) Given a program and its specification, consider the set S of specified behaviors and the set P of programmed behaviors. Figure 1.2 shows the relationship between the specified and programmed behaviors. Of all the possible program behaviors, the specified ones are in the circle labeled S and all those behaviors actually programmed are in P. With this diagram, we can see more clearly the problems that confront a tester. What if certain specified behaviors have not been programmed? In our earlier terminology, these are faults of omission. Similarly, what if certain programmed (implemented) behaviors have not been specified? These correspond to faults of commission and to errors that occurred after the specification was complete. The intersection of S and P (the football-shaped region) is the "correct" portion, that is, behaviors that are both specified and implemented. A very good view of testing is that it is the determination of the extent of program behavior that is both specified and implemented. (As an aside, note that "correctness" only has meaning with respect to a specification and an implementation. It is a relative term, not an absolute.)

The new circle in Figure 1.3 is for test cases. Notice the slight discrepancy with our universe of discourse and the set of program behaviors. Because a test case causes a program behavior, the mathematicians might forgive us. Now, consider the relationships among sets S, P, and T. There may be specified behaviors that are not tested (regions 2 and 5), specified behaviors that are tested (regions 1 and 4), and test cases that correspond to unspecified behaviors (regions 3 and 7).

Similarly, there may be programmed behaviors that are not tested (regions 2 and 6), programmed behaviors that are tested (regions 1 and 3), and test cases that correspond to behaviors that were not implemented (regions 4 and 7).

Each of these regions is important. If specified behaviors exist for which no test cases are available, the testing is necessarily incomplete. If certain test cases correspond to unspecified behaviors,

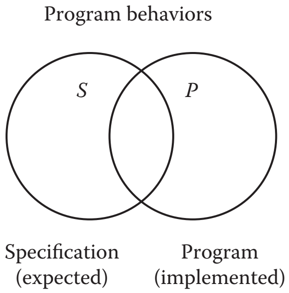

Program behaviors
S
P
Specification
(expected)
Program
(implemented)

Figure 1.2 Specified and implemented program behaviors.

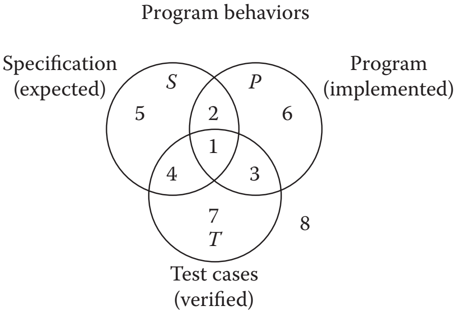

Program behaviors
Specification
(expected)
S
P
Program
(implemented)
5
2
6
1
4
3
7
T
8
Test cases
(verified)

Figure 1.3 Specified, implemented, and tested behaviors.

some possibilities arise: either such a test case is unwarranted, the specification is deficient, or the tester wishes to determine that specified non-behavior does not occur. (In my experience, good testers often postulate test cases of this latter type. This is a fine reason to have good testers participate in specification and design reviews.)

We are already at a point where we can see some possibilities for testing as a craft: what can a tester do to make the region where these sets all intersect (region 1) as large as possible? Another approach is to ask how the test cases in set *T* are identified. The short answer is that test cases are identified by a testing method. This framework gives us a way to compare the effectiveness of diverse testing methods, as we shall see in Chapter 10.

1.4 Identifying Test Cases

Two fundamental approaches are used to identify test cases; traditionally, these have been called functional and structural testing. Specification-based and code-based are more descriptive names, and they will be used here. Both approaches have several distinct test case identification methods; they are generally just called testing methods. They are methodical in the sense that two testers following the same "method" will devise very similar (equivalent?) test cases.

1.4.1 Specification-Based Testing

The reason that specification-based testing was originally called "functional testing" is that any program can be considered to be a function that maps values from its input domain to values in its output range. (Function, domain, and range are defined in Chapter 3.) This notion is commonly used in engineering, when systems are considered to be black boxes. This led to another synonymous term—black box testing, in which the content (implementation) of the black box is not known, and the function of the black box is understood completely in terms of its inputs and outputs (see Figure 1.4). In *Zen and the Art of Motorcycle Maintenance*, Robert Pirsig refers to this as "romantic" comprehension (Pirsig, 1973). Many times, we operate very effectively with black box knowledge; in fact, this is central to object orientation. As an example, most people successfully operate automobiles with only black box knowledge.

With the specification-based approach to test case identification, the only information used is the specification of the software. Therefore, the test cases have two distinct advantages: (1) they are independent of how the software is implemented, so if the implementation changes, the test cases are still useful; and (2) test case development can occur in parallel with the implementation, thereby reducing the overall project development interval. On the negative side, specification-based test cases frequently suffer from two problems: significant redundancies may exist among test cases, compounded by the possibility of gaps of untested software.

Figure 1.5 shows the results of test cases identified by two specification-based methods. Method A identifies a larger set of test cases than does method B. Notice that, for both methods, the set of test cases is completely contained within the set of specified behavior. Because specification-based methods are based on the specified behavior, it is hard to imagine these methods identifying behaviors that are not specified. In Chapter 8, we will see direct comparisons of test cases generated by various specification-based methods for the examples defined in Chapter 2.

In Chapters 5 through 7, we will examine the mainline approaches to specification-based testing, including boundary value analysis, robustness testing, worst-case analysis, special value testing, input (domain) equivalence classes, output (range) equivalence classes, and decision table-based testing. The common thread running through these techniques is that all are based on

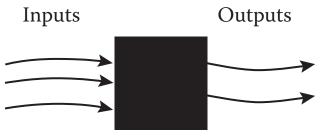


Figure 1.4 Engineer's black box.

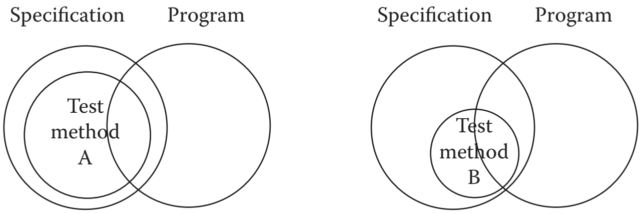


Figure 1.5 Comparing specification-based test case identification methods.

definitional information of the item tested. Some of the mathematical background presented in Chapter 3 applies primarily to specification-based approaches.

1.4.2 Code-Based Testing

Code-based testing is the other fundamental approach to test case identification. To contrast it with black box testing, it is sometimes called white box (or even clear box) testing. The clear box metaphor is probably more appropriate because the essential difference is that the implementation (of the black box) is known and used to identify test cases. The ability to “see inside” the black box allows the tester to identify test cases on the basis of how the function is actually implemented.

Code-based testing has been the subject of some fairly strong theories. To really understand code-based testing, familiarity with the concepts of linear graph theory (Chapter 4) is essential. With these concepts, the tester can rigorously describe exactly what is tested. Because of its strong theoretical basis, code-based testing lends itself to the definition and use of test coverage metrics. Test coverage metrics provide a way to explicitly state the extent to which a software item has been tested, and this in turn makes testing management more meaningful.

Figure 1.6 shows the results of test cases identified by two code-based methods. As before, method A identifies a larger set of test cases than does method B. Is a larger set of test cases necessarily better? This is an excellent question, and code-based testing provides important ways to develop an answer. Notice that, for both methods, the set of test cases is completely contained within the set of programmed behavior. Because code-based methods are based on the program, it is hard to imagine these methods identifying behaviors that are not programmed. It is easy to imagine, however, that a set of code-based test cases is relatively small with respect to the full set of programmed behaviors. In Chapter 10, we will see direct comparisons of test cases generated by various code-based methods.

1.4.3 Specification-Based versus Code-Based Debate

Given the two fundamentally different approaches to test case identification, it is natural to question which is better. If you read much of the literature, you will find strong adherents to either choice.

The Venn diagrams presented earlier yield a strong resolution to this debate. Recall that the goal of both approaches is to identify test cases (Figure 1.7). Specification-based testing uses only the specification to identify test cases, while code-based testing uses the program source code (implementation) as the basis of test case identification. Later chapters will establish that

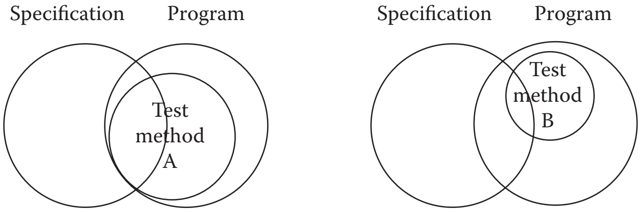


Figure 1.6 Comparing code-based test case identification methods.

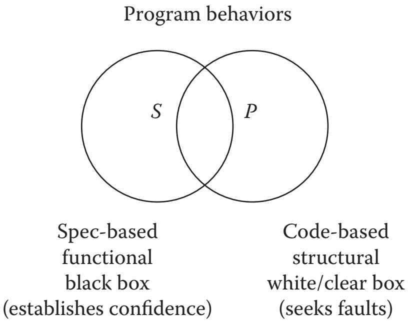


Figure 1.7 Sources of test cases.

neither approach by itself is sufficient. Consider program behaviors: if all specified behaviors have not been implemented, code-based test cases will never be able to recognize this. Conversely, if the program implements behaviors that have not been specified, this will never be revealed by specification-based test cases. (A Trojan horse is a good example of such unspecified behavior.) The quick answer is that both approaches are needed; the testing craftsperson's answer is that a judicious combination will provide the confidence of specification-based testing and the measurement of code-based testing. Earlier, we asserted that specification-based testing often suffers from twin problems of redundancies and gaps. When specification-based test cases are executed in combination with code-based test coverage metrics, both of these problems can be recognized and resolved.

The Venn diagram view of testing provides one final insight. What is the relationship between set T of test cases and sets S and P of specified and implemented behaviors? Clearly, the test cases in set T are determined by the test case identification method used. A very good question to ask is how appropriate (or effective) is this method? To close a loop from an earlier discussion, recall the causal trail from error to fault, failure, and incident. If we know what kind of errors we are prone to make, and if we know what kinds of faults are likely to reside in the software to be tested, we can use this to employ more appropriate test case identification methods. This is the point at which testing really becomes a craft.

1.5 Fault Taxonomies

Our definitions of error and fault hinge on the distinction between process and product: process refers to how we do something, and product is the end result of a process. The point at which testing and Software Quality Assurance (SQA) meet is that SQA typically tries to improve the product by improving the process. In that sense, testing is clearly more product oriented. SQA is more concerned with reducing errors endemic in the development process, whereas testing is more concerned with discovering faults in a product. Both disciplines benefit from a clearer definition of types of faults. Faults can be classified in several ways: the development phase in which the corresponding error occurred, the consequences of corresponding failures, difficulty to resolve, risk of no resolution, and so on. My favorite is based on anomaly (fault) occurrence: one time only, intermittent, recurring, or repeatable.

For a comprehensive treatment of types of faults, see the IEEE Standard Classification for Software Anomalies (IEEE, 1993). (A software anomaly is defined in that document as "a departure from the expected," which is pretty close to our definition.) The IEEE standard defines a detailed anomaly resolution process built around four phases (another life cycle): recognition, investigation, action, and disposition. Some of the more useful anomalies are given in Tables 1.1 through 1.5; most of these are from the IEEE standard but I have added some of my favorites.

Since the primary purpose of a software review is to find faults, review checklists (see Chapter 22) are another good source of fault classifications. Karl Wiegers has an excellent set of checklists on his website: http://www.processimpact.com/pr_goodies.shtml.

Table 1.1 Input/Output Faults

Type	*Instances*
Input	Correct input not accepted
	Incorrect input accepted
	Description wrong or missing
	Parameters wrong or missing
Output	Wrong format
	Wrong result
	Correct result at wrong time (too early, too late)
	Incomplete or missing result
	Spurious result
	Spelling/grammar
	Cosmetic

Table 1.2 Logic Faults

Missing case(s)
Duplicate case(s)
Extreme condition neglected
Misinterpretation
Missing condition
Extraneous condition(s)
Test of wrong variable
Incorrect loop iteration
Wrong operator (e.g., $<$ instead of $\leq$)

Table 1.3 Computation Faults

Incorrect algorithm
Missing computation
Incorrect operand
Incorrect operation
Parenthesis error
Insufficient precision (round-off, truncation)
Wrong built-in function

Table 1.4 Interface Faults

Incorrect interrupt handling
I/O timing
Call to wrong procedure
Call to nonexistent procedure
Parameter mismatch (type, number)
Incompatible types
Superfluous inclusion

Table 1.5 Data Faults

Incorrect initialization
Incorrect storage/access
Wrong flag/index value
Incorrect packing/unpacking
Wrong variable used
Wrong data reference
Scaling or units error
Incorrect data dimension
Incorrect subscript
Incorrect type
Incorrect data scope
Sensor data out of limits
Off by one
Inconsistent data

1.6 Levels of Testing

Thus far, we have said nothing about one of the key concepts of testing—levels of abstraction. Levels of testing echo the levels of abstraction found in the waterfall model of the software development life cycle. Although this model has its drawbacks, it is useful for testing as a means of identifying distinct levels of testing and for clarifying the objectives that pertain to each level. A diagrammatic variation of the waterfall model, known as the V-Model in ISTQB parlance, is given in Figure 1.8; this variation emphasizes the correspondence between testing and design levels. Notice that, especially in terms of specification-based testing, the three levels of definition (specification, preliminary design, and detailed design) correspond directly to three levels of testing—system, integration, and unit testing.

A practical relationship exists between levels of testing versus specification-based and code-based testing. Most practitioners agree that code-based testing is most appropriate at the unit level, whereas specification-based testing is most appropriate at the system level. This is generally true; however, it is also a likely consequence of the base information produced during the requirements specification, preliminary design, and detailed design phases. The constructs defined for code-based testing make the most sense at the unit level, and similar constructs are only now becoming available for the integration and system levels of testing. We develop such structures in Chapters 11 through 17 to support code-based testing at the integration and system levels for both traditional and object-oriented software.

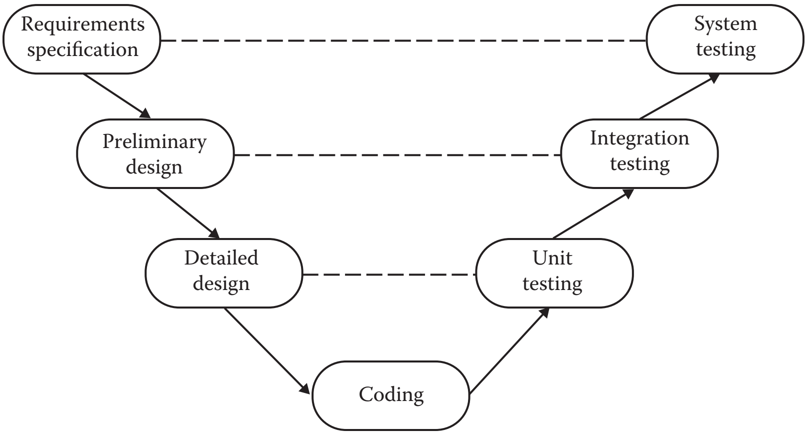


Figure 1.8 Levels of abstraction and testing in waterfall model.

EXERCISES

1. Make a Venn diagram that reflects a part of the following statement: "… we have left undone that which we ought to have done, and we have done that which we ought not to have done …"
2. Make a Venn diagram that reflects the essence of Reinhold Niebuhr's "Serenity Prayer":
   God, grant me the serenity to accept the things I cannot change,
   Courage to change the things I can,
   And wisdom to know the difference.
3. Describe each of the eight regions in Figure 1.3. Can you recall examples of these in software you have written?
4. One of the tales of software lore describes a disgruntled employee who writes a payroll program that contains logic that checks for the employee's identification number before producing paychecks. If the employee is ever terminated, the program creates havoc. Discuss this situation in terms of the error, fault, and failure pattern, and decide which form of testing would be appropriate.

References

IEEE Computer Society, *IEEE Standard Glossary of Software Engineering Terminology*, 1983, ANSI/IEEE Std 729-1983.

IEEE Computer Society, *IEEE Standard Classification for Software Anomalies*, 1993, IEEE Std 1044-1993.

Pirsig, R. M., *Zen and the Art of Motorcycle Maintenance*, Bantam Books, New York, 1973.

Chapter 2

Examples

Three examples will be used throughout in Chapters 5 through 9 to illustrate the various unit testing methods: the triangle problem (a venerable example in testing circles); a logically complex function, NextDate; and an example that typifies MIS applications, known here as the commission problem. Taken together, these examples raise most of the issues that testing craftspersons will encounter at the unit level. The discussion of higher levels of testing in Chapters 11 through 17 uses four other examples: a simplified version of an automated teller machine (ATM), known here as the simple ATM system (SATM); the currency converter, an event-driven application typical of graphical user interface (GUI) applications; and the windshield wiper control device from the Saturn™ automobile. The last example, a garage door controller, illustrates some of the issues of "systems of systems."

For the purposes of code-based testing, pseudocode implementations of the three unit-level examples are given in this chapter. System-level descriptions of the SATM system, the currency converter, the Saturn windshield wiper system, and the garage door controller are given in Chapters 11 through 17. These applications are modeled with finite-state machines, variations of event-driven petri nets, selected StateCharts, and with the Universal Modeling Language (UML).

2.1 Generalized Pseudocode

Pseudocode provides a language-neutral way to express program source code. This version is loosely based on Visual Basic and has constructs at two levels: unit and program components. Units can be interpreted either as traditional components (procedures and functions) or as object-oriented components (classes and objects). This definition is somewhat informal; terms such as expression, variable list, and field description are used with no formal definition. Items in angle brackets indicate language elements that can be used at the identified positions. Part of the value of any pseudocode is the suppression of unwanted detail; here, we illustrate this by allowing natural language phrases in place of more formal, complex conditions (see Table 2.1).

Table 2.1 Generalized Pseudocode

Language Element	*Generalized Pseudocode Construct*
Comment	‘ <text>
Data structure declaration	Type <type name> <list of field descriptions> End <type name>
Data declaration	Dim <variable> As <type>
Assignment statement	<variable> = <expression>
Input	Input (<variable list>)
Output	Output (<variable list>)
Condition	<expression> <relational operator> <expression>
Compound condition	<Condition> <logical connective> <Condition>
Sequence	statements in sequential order

Simple selection	If <condition> Then
	    <then clause>
	EndIf

Selection	If <condition>
	    Then <then clause>
	    Else <else clause>
	EndIf

Multiple selection	Case <variable> Of Case 1: <predicate> <Case clause> … Case *n*: <predicate> <Case clause> EndCase
Counter-controlled repetition	For <counter> = <start> To <end> <loop body> EndFor
Pretest repetition	While <condition> <loop body> EndWhile

(*continued*)

Table 2.1 Generalized Pseudocode (Continued)

Language Element	*Generalized Pseudocode Construct*
Posttest repetition	Do <loop body> Until <condition>
Procedure definition (similarly for functions and o–o methods)	<procedure name> (Input: <list of variables>; Output: <list of variables>) <body> End <procedure name>
Interunit communication	Call <procedure name> (<list of variables>; <list of variables>)
Class/Object definition	<name> (<attribute list>; <method list>, <body>) End <name>
Interunit communication	msg <destination object name>.<method name> (<list of variables>)
Object creation	Instantiate <class name>.<object name> (list of attribute values)
Object destruction	Delete <class name>.<object name>
Program	Program <program name> <unit list> End<program name>

2.2 The Triangle Problem

The triangle problem is the most widely used example in software testing literature. Some of the more notable entries in three decades of testing literature are Gruenberger (1973), Brown and Lipov (1975), Myers (1979), Pressman (1982) and subsequent editions, Clarke and Richardson (1983, 1984), Chellappa (1987), and Hetzel (1988). There are others, but this list makes the point.

2.2.1 Problem Statement

Simple version: The triangle program accepts three integers, a, b, and c, as input. These are taken to be sides of a triangle. The output of the program is the type of triangle determined by the three sides: Equilateral, Isosceles, Scalene, or NotATriangle. Sometimes, this problem is extended to include right triangles as a fifth type; we will use this extension in some of the exercises.

Improved version: The triangle program accepts three integers, a, b, and c, as input. These are taken to be sides of a triangle. The integers a, b, and c must satisfy the following conditions:

c1. $1 \leq a \leq 200$
c2. $1 \leq b \leq 200$
c3. $1 \leq c \leq 200$
c4. $a < b + c$
c5. $b < a + c$
c6. $c < a + b$

The output of the program is the type of triangle determined by the three sides: Equilateral, Isosceles, Scalene, or NotATriangle. If an input value fails any of conditions c1, c2, or c3, the program notes this with an output message, for example, "Value of b is not in the range of permitted values." If values of a, b, and c satisfy conditions c4, c5, and c6, one of four mutually exclusive outputs is given:

1. If all three sides are equal, the program output is Equilateral.
2. If exactly one pair of sides is equal, the program output is Isosceles.
3. If no pair of sides is equal, the program output is Scalene.
4. If any of conditions c4, c5, and c6 is not met, the program output is NotATriangle.

2.2.2 Discussion

Perhaps one of the reasons for the longevity of this example is that it contains clear but complex logic. It also typifies some of the incomplete definitions that impair communication among customers, developers, and testers. The first specification presumes the developers know some details about triangles, particularly the triangle inequality: the sum of any pair of sides must be strictly greater than the third side. The upper limit of 200 is both arbitrary and convenient; it will be used when we develop boundary value test cases in Chapter 5.

2.2.3 Traditional Implementation

The traditional implementation of this grandfather of all examples has a rather FORTRAN-like style. The flowchart for this implementation appears in Figure 2.1. Figure 2.2 is a flowchart for the improved version. The flowchart box numbers correspond to comment numbers in the (FORTRAN-like) pseudocode program given next. (These numbers correspond exactly to those in Pressman [1982].) This implementation shows its age; a better implementation is given in Section 2.2.4.

The variable "match" is used to record equality among pairs of the sides. A classic intricacy of the FORTRAN style is connected with the variable "match": notice that all three tests for the triangle inequality do not occur. If two sides are equal, say a and c, it is only necessary to compare a + c with b. (Because b must be greater than zero, a + b must be greater than c because c equals a.) This observation clearly reduces the number of comparisons that must be made. The efficiency of this version is obtained at the expense of clarity (and ease of testing). We will find this version useful later when we discuss infeasible program execution paths. That is the best reason for perpetuating this version. Notice that six ways are used to reach the NotATriangle box (12.1–12.6), and three ways are used to reach the Isosceles box (15.1–15.3).

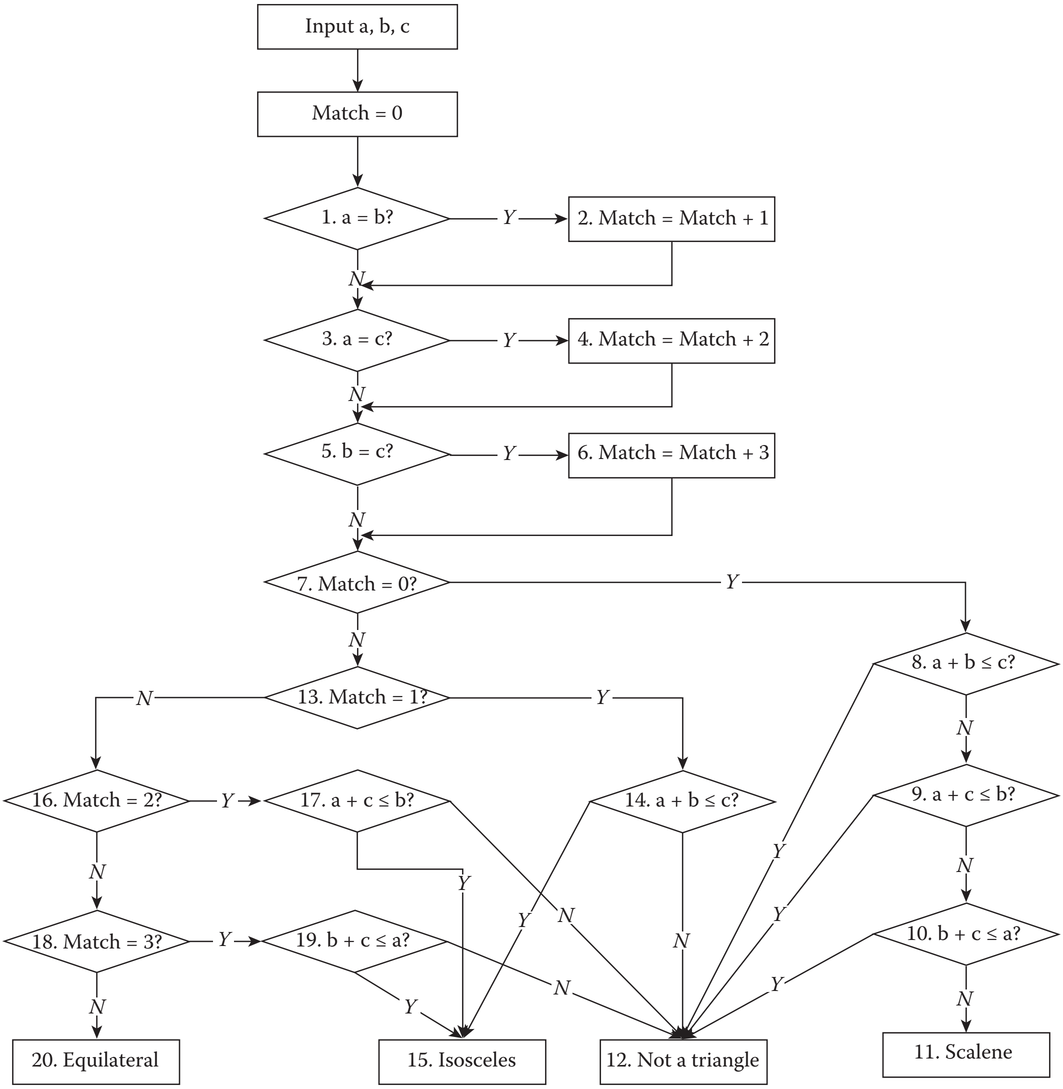

Input a, b, c
Match = 0
1. a = b?
2. Match = Match + 1
3. a = c?
4. Match = Match + 2
5. b = c?
6. Match = Match + 3
7. Match = 0?
8. a + b ≤ c?
9. a + c ≤ b?
10. b + c ≤ a?
11. Scalene
12. Not a triangle
13. Match = 1?
14. a + b ≤ c?
15. Isosceles
16. Match = 2?
17. a + c ≤ b?
18. Match = 3?
19. b + c ≤ a?
20. Equilateral
Y
N

Figure 2.1 Flowchart for traditional triangle program implementation.

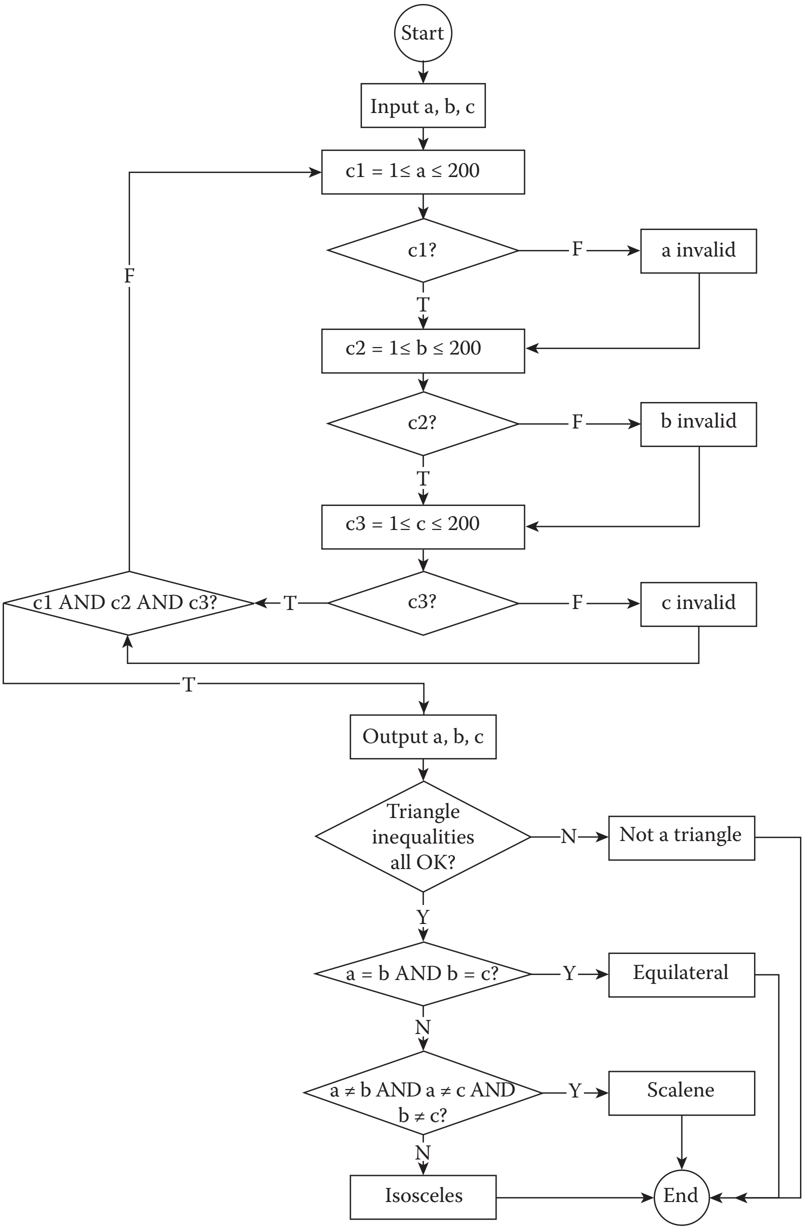

Start
Input a, b, c
c1 = 1≤ a ≤ 200
c1?
F
a invalid
T
c2 = 1≤ b ≤ 200
c2?
F
b invalid
T
c3 = 1≤ c ≤ 200
c3?
F
c invalid
T
c1 AND c2 AND c3?
F
T
Output a, b, c
Triangle inequalities all OK?
N
Not a triangle
Y
a = b AND b = c?
Y
Equilateral
N
a ≠ b AND a ≠ c AND b ≠ c?
Y
Scalene
N
Isosceles
End

Figure 2.2 Flowchart for improved triangle program implementation.

The pseudocode for this is given next.

```
Program triangle1 'Fortran-like version
'
Dim a, b, c, match As INTEGER
'
Output("Enter 3 integers which are sides of a triangle")
Input(a, b, c)
```

```
Output("Side A is",a)
Output("Side B is",b)
Output("Side C is",c)
match = 0
If a = b                                                    '(1)
   Then match = match + 1                                   '(2)
EndIf
If a = c                                                    '(3)
   Then match = match + 2                                   '(4)
EndIf
If b = c                                                    '(5)
   Then match = match + 3                                   '(6)
EndIf
If match = 0                                                '(7)
   Then If (a + b)≤ c                                       '(8)
            Then Output("NotATriangle")                     '(12.1)
            Else If (b + c) ≤ a                             '(9)
                     Then Output("NotATriangle")            '(12.2)
                     Else If (a + c) ≤ b                    '(10)
                              Then Output("NotATriangle")   '(12.3)
                              Else Output ("Scalene")       '(11)
                          EndIf
                 EndIf
          EndIf
   Else If match = 1                                        '(13)
            Then If (a + c) ≤ b                             '(14)
                     Then Output("NotATriangle")            '(12.4)
                     Else Output ("Isosceles")              '(15.1)
                 EndIf
            Else If match=2                                 '(16)
                     Then If (a + c) ≤ b
                              Then Output("NotATriangle")   (12.5)
                              Else Output ("Isosceles")     '(15.2)
                          EndIf
                     Else If match = 3                      '(18)
                          Then If (b + c) ≤ a  '(19)
                              Then Output("NotATriangle")   '(12.6)
                              Else Output ("Isosceles")     '(15.3)
                          EndIf
                     Else Output ("Equilateral")            '(20)
                 EndIf
            EndIf
      EndIf
EndIf
'
End Triangle1
```

2.2.4 Structured Implementations

Program triangle2 'Structured programming version of simpler specification

```
Dim a,b,c As Integer
Dim IsATriangle As Boolean
```

```
'Step 1: Get Input
Output("Enter 3 integers which are sides of a triangle")
Input(a,b,c)
Output("Side A is",a)
Output("Side B is",b)
Output("Side C is",c)
'Step 2: Is A Triangle?'
If (a < b + c) AND (b < a + c) AND (c < a + b)
    Then IsATriangle = True
    Else IsATriangle = False
EndIf
`
'Step 3: Determine Triangle Type
If IsATriangle
    Then If (a = b) AND (b = c)
         Then Output ("Equilateral")
         Else If (a ≠ b) AND (a ≠ c) AND (b ≠ c)
                 Then Output ("Scalene")
                 Else Output ("Isosceles")
              EndIf
         EndIf
    Else Output("Not a Triangle")
EndIf
End triangle2
```

Third version

```
Program triangle3'
Dim a, b, c As Integer
Dim c1, c2, c3, IsATriangle As Boolean
'Step 1: Get Input
Do
    Output("Enter 3 integers which are sides of a triangle")
    Input(a, b, c)
    c1 = (1 ≤ a) AND (a ≤ 300)
    c2 = (1 ≤ b) AND (b ≤ 300)
    c3 = (1 ≤ c) AND (c ≤ 300)
    If NOT(c1)
         Then Output("Value of a is not in the range of permitted values")
    EndIf
    If NOT(c2)
         Then Output("Value of b is not in the range of permitted values")
    EndIf
    If NOT(c3)
         ThenOutput("Value of c is not in the range of permitted values")
    EndIf
Until c1 AND c2 AND c3
Output("Side A is",a)
Output("Side B is",b)
Output("Side C is",c)
'Step 2: Is A Triangle?
If (a < b + c) AND (b < a + c) AND (c < a + b)
    Then IsATriangle = True
    Else IsATriangle = False
```

```
EndIf
'Step 3: Determine Triangle Type
If IsATriangle
   Then If (a = b) AND (b = c)
          Then Output ("Equilateral")
          Else If (a ≠ b) AND (a ≠ c) AND (b ≠ c)
                 Then Output ("Scalene")
                 Else Output ("Isosceles")
              EndIf
       EndIf
   Else Output("Not a Triangle")
EndIf
End triangle3
```

2.3 The NextDate Function

The complexity in the triangle program is due to the relationships between inputs and correct outputs. We will use the NextDate function to illustrate a different kind of complexity—logical relationships among the input variables.

2.3.1 Problem Statement

NextDate is a function of three variables: month, date, and year. It returns the date of the day after the input date. The month, date, and year variables have integer values subject to these conditions (the year range ending in 2012 is arbitrary, and is from the first edition):

c1. $1 \le \text{month} \le 12$
c2. $1 \le \text{day} \le 31$
c3. $1812 \le \text{year} \le 2012$

As we did with the triangle program, we can make our problem statement more specific. This entails defining responses for invalid values of the input values for the day, month, and year. We can also define responses for invalid combinations of inputs, such as June 31 of any year. If any of conditions c1, c2, or c3 fails, NextDate produces an output indicating the corresponding variable has an out-of-range value—for example, "Value of month not in the range 1...12." Because numerous invalid day–month–year combinations exist, NextDate collapses these into one message: "Invalid Input Date."

2.3.2 Discussion

Two sources of complexity exist in the NextDate function: the complexity of the input domain discussed previously, and the rule that determines when a year is a leap year. A year is 365.2422 days long; therefore, leap years are used for the "extra day" problem. If we declared a leap year every fourth year, a slight error would occur. The Gregorian calendar (after Pope Gregory) resolves this by adjusting leap years on century years. Thus, a year is a leap year if it is divisible by 4, unless it is a century year. Century years are leap years only if they are multiples of 400 (Inglis, 1961); thus, 1992, 1996, and 2000 are leap years, while the year 1900 is not a leap year. The NextDate function also illustrates a sidelight of software testing. Many times, we find examples of Zipf's law, which states that 80% of the activity occurs in 20% of the space. Notice how much of the source code is devoted to leap year considerations. In the second implementation, notice how much code is devoted to input value validation.

2.3.3 Implementations

```
Program NextDate1 'Simple version
Dim tomorrowDay,tomorrowMonth,tomorrowYear As Integer
Dim day,month,year As Integer
Output ("Enter today's date in the form MM DD YYYY")
Input (month, day, year)
Case month Of
Case 1: month Is 1,3,5,7,8, Or 10: '31 day months (except Dec.)
 If day < 31
   Then tomorrowDay = day + 1
   Else
    tomorrowDay = 1
    tomorrowMonth = month + 1
 EndIf
Case 2: month Is 4,6,9, Or 11 '30 day months
 If day < 30
   Then tomorrowDay = day + 1
   Else
    tomorrowDay = 1
    tomorrowMonth = month + 1
 EndIf
Case 3: month Is 12: 'December
 If day < 31
   Then tomorrowDay = day + 1
   Else
    tomorrowDay = 1
    tomorrowMonth = 1
    If year = 2012
      Then Output ("2012 is over")
      Else tomorrow.year = year + 1
 EndIf
Case 4: month is 2: 'February
 If day < 28
   Then tomorrowDay = day + 1
   Else
    If day = 28
      Then If ((year is a leap year)
             Then tomorrowDay = 29 'leap year
             Else 'not a leap year
              tomorrowDay = 1
              tomorrowMonth = 3
           EndIf
    Else If day = 29
           Then If ((year is a leap year)
                  Then tomorrowDay = 1
```

```
                        tomorrowMonth = 3
                   Else ‘not a leap year
                      Output(“Cannot have Feb.”, day)
                 EndIf
          EndIf
     EndIf
 EndIf
EndCase
Output (“Tomorrow’s date is”, tomorrowMonth, tomorrowDay, tomorrowYear)
End NextDate

Program NextDate2   Improved version
‘
Dim tomorrowDay,tomorrowMonth,tomorrowYear As Integer
Dim day,month,year As Integer
Dim c1, c2, c3 As Boolean
‘
Do
 Output (“Enter today’s date in the form MM DD YYYY”)
 Input (month, day, year)
 c1 = (1 ≤ day) AND (day ≤ 31)
 c2 = (1 ≤ month) AND (month ≤ 12)
 c3 = (1812 ≤ year) AND (year ≤ 2012)
 If NOT(c1)
  Then Output(“Value of day not in the range 1..31”)
 EndIf
 If NOT(c2)
  Then Output(“Value of month not in the range 1..12”)
 EndIf
 If NOT(c3)
  Then Output(“Value of year not in the range 1812..2012”)
 EndIf
Until c1 AND c2 AND c2

Case month Of
Case 1: month Is 1,3,5,7,8, Or 10: ‘31 day months (except Dec.)
 If day < 31
    Then tomorrowDay = day + 1
    Else
      tomorrowDay = 1
      tomorrowMonth = month + 1
 EndIf
Case 2: month Is 4,6,9, Or 11 ‘30 day months
 If day < 30
    Then tomorrowDay = day + 1
    Else
      If day = 30
        Then tomorrowDay = 1
             tomorrowMonth = month + 1
        Else Output(“Invalid Input Date”)
      EndIf
 EndIf
Case 3: month Is 12: ‘December
```

```
 If day < 31
    Then tomorrowDay = day + 1
    Else
      tomorrowDay = 1
      tomorrowMonth = 1
      If year = 2012
        Then Output (“Invalid Input Date”)
        Else tomorrow.year = year + 1
      EndIf
 EndIf
Case 4: month is 2: ‘February
 If day < 28
    Then tomorrowDay = day + 1
    Else
      If day = 28
        Then
         If (year is a leap year)
            Then tomorrowDay = 29 ‘leap day
            Else ‘not a leap year
              tomorrowDay = 1
              tomorrowMonth = 3
         EndIf
        Else
         If day = 29
            Then
              If (year is a leap year)
                 Then tomorrowDay = 1
                      tomorrowMonth = 3
                 Else
                  If day > 29
                     Then Output(“Invalid Input Date”)
                  EndIf
              EndIf
         EndIf
      EndIf
 EndIf
EndCase
Output (“Tomorrow’s date is”, tomorrowMonth, tomorrowDay, tomorrowYear)
‘
End NextDate2
```

2.4 The Commission Problem

Our third example is more typical of commercial computing. It contains a mix of computation and decision making, so it leads to interesting testing questions. Our main use of this example will be in our discussion of data flow and slice-based testing.

2.4.1 Problem Statement

A rifle salesperson in the former Arizona Territory sold rifle locks, stocks, and barrels made by a gunsmith in Missouri. Locks cost $45, stocks cost $30, and barrels cost $25. The salesperson had to sell at least one lock, one stock, and one barrel (but not necessarily one complete rifle) per month, and production limits were such that the most the salesperson could sell in a month was 70 locks, 80 stocks, and 90 barrels. After each town visit, the salesperson sent a telegram to the Missouri gunsmith with the number of locks, stocks, and barrels sold in that town. At the end of a month, the salesperson sent a very short telegram showing –1 lock sold. The gunsmith then knew the sales for the month were complete and computed the salesperson's commission as follows: 10% on sales up to (and including) $1000, 15% on the next $800, and 20% on any sales in excess of $1800.

2.4.2 Discussion

This example is somewhat contrived to make the arithmetic quickly visible to the reader. It might be more realistic to consider some other additive function of several variables, such as various calculations found in filling out a US 1040 income tax form. (We will stay with rifles.) This problem separates into three distinct pieces: the input data portion, in which we could deal with input data validation (as we did for the triangle and NextDate programs), the sales calculation, and the commission calculation portion. This time, we will omit the input data validation portion. We will replicate the telegram convention with a sentinel-controlled while loop that is typical of MIS data gathering applications.

2.4.3 Implementation

```
Program Commission (INPUT,OUTPUT)
'
Dim locks, stocks, barrels As Integer
Dim lockPrice, stockPrice, barrelPrice As Real
Dim totalLocks,totalStocks,totalBarrels As Integer
Dim lockSales, stockSales, barrelSales As Real
Dim sales,commission : REAL
'
lockPrice = 45.0
stockPrice = 30.0
barrelPrice = 25.0
totalLocks = 0
totalStocks = 0
totalBarrels = 0
'
Input(locks)
While NOT(locks = -1)      'Input device uses -1 to indicate end of data
    Input(stocks, barrels)
    totalLocks = totalLocks + locks
    totalStocks = totalStocks + stocks
    totalBarrels = totalBarrels + barrels
    Input(locks)
EndWhile
'
Output("Locks sold:", totalLocks)
Output("Stocks sold:", totalStocks)
Output("Barrels sold:", totalBarrels)
'
lockSales = lockPrice * totalLocks
stockSales = stockPrice * totalStocks
```

```
barrelSales = barrelPrice * totalBarrels
sales = lockSales + stockSales + barrelSales
Output(“Total sales:”, sales)
`
If (sales > 1800.0)
    Then
       commission = 0.10 * 1000.0
       commission = commission + 0.15 * 800.0
       commission = commission + 0.20 * (sales-1800.0)
    Else If (sales > 1000.0)
            Then
               commission = 0.10 * 1000.0
               commission = commission + 0.15*(sales-1000.0)
            Else commission = 0.10 * sales
         EndIf
EndIf
Output(“Commission is $”,commission)
End Commission
```

2.5 The SATM System

To better discuss the issues of integration and system testing, we need an example with larger scope (Figure 2.3).

The ATM described here is minimal, yet it contains an interesting variety of functionality and interactions that typify the client side of client–server systems.

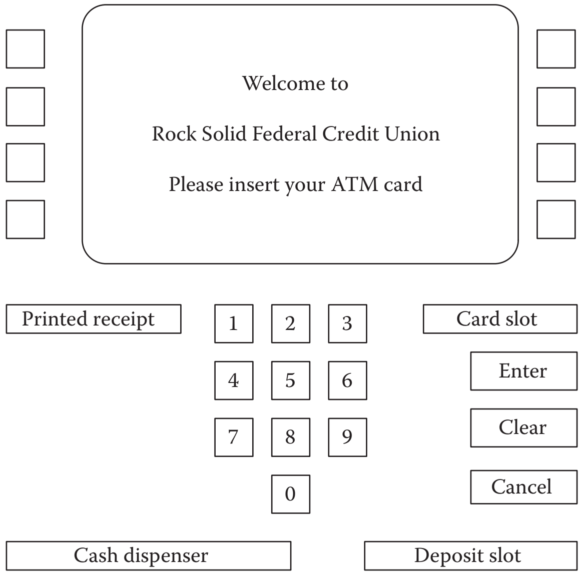

Welcome to
Rock Solid Federal Credit Union
Please insert your ATM card
Printed receipt
1
2
3
4
5
6
7
8
9
0
Card slot
Enter
Clear
Cancel
Cash dispenser
Deposit slot

Figure 2.3 SATM terminal.

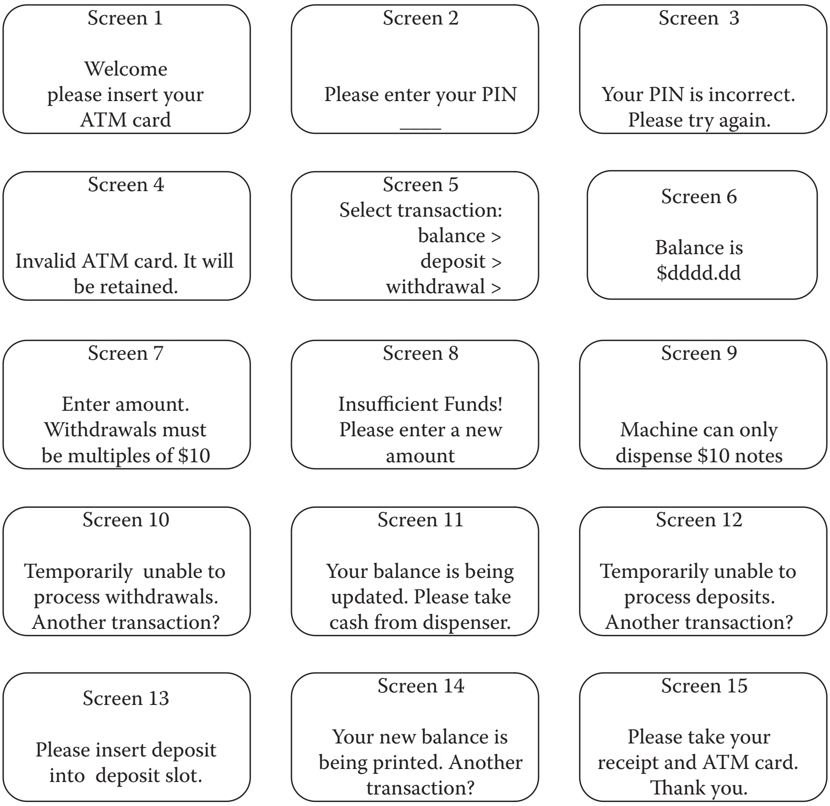
2.5.1 Problem Statement

The SATM system communicates with bank customers via the 15 screens shown in Figure 2.4. Using a terminal with features as shown in Figure 2.3, SATM customers can select any of three transaction types: deposits, withdrawals, and balance inquiries. For simplicity, these transactions can only be done on a checking account.

When a bank customer arrives at an SATM station, screen 1 is displayed. The bank customer accesses the SATM system with a plastic card encoded with a personal account number (PAN), which is a key to an internal customer account file, containing, among other things, the customer's name and account information. If the customer's PAN matches the information in the customer account file, the system presents screen 2 to the customer. If the customer's PAN is not found, screen 4 is displayed, and the card is kept.

At screen 2, the customer is prompted to enter his or her personal identification number (PIN). If the PIN is correct (i.e., matches the information in the customer account file), the system displays screen 5; otherwise, screen 3 is displayed. The customer has three chances to get the PIN correct; after three failures, screen 4 is displayed, and the card is kept.

On entry to screen 5, the customer selects the desired transaction from the options shown on screen. If balance is requested, screen 14 is then displayed. If a deposit is requested, the status of the deposit envelope slot is determined from a field in the terminal control file. If no problem is known, the system displays screen 7 to get the transaction amount. If a problem occurs with the deposit envelope slot, the



Screen 1
Welcome
please insert your
ATM card

Screen 2
Please enter your PIN
____

Screen 3
Your PIN is incorrect.
Please try again.

Screen 4
Invalid ATM card. It will
be retained.

Screen 5
Select transaction:
balance >
deposit >
withdrawal >

Screen 6
Balance is
$dddd.dd

Screen 7
Enter amount.
Withdrawals must
be multiples of $10

Screen 8
Insufficient Funds!
Please enter a new
amount

Screen 9
Machine can only
dispense $10 notes

Screen 10
Temporarily unable to
process withdrawals.
Another transaction?

Screen 11
Your balance is being
updated. Please take
cash from dispenser.

Screen 12
Temporarily unable to
process deposits.
Another transaction?

Screen 13
Please insert deposit
into deposit slot.

Screen 14
Your new balance is
being printed. Another
transaction?

Screen 15
Please take your
receipt and ATM card.
Thank you.

Figure 2.4 SATM screens.

system displays screen 12. Once the deposit amount has been entered, the system displays screen 13, accepts the deposit envelope, and processes the deposit. The system then displays screen 14.

If a withdrawal is requested, the system checks the status (jammed or free) of the withdrawal chute in the terminal control file. If jammed, screen 10 is displayed; otherwise, screen 7 is displayed so the customer can enter the withdrawal amount. Once the withdrawal amount is entered, the system checks the terminal status file to see if it has enough currency to dispense. If it does not, screen 9 is displayed; otherwise, the withdrawal is processed. The system checks the customer balance (as described in the balance request transaction); if the funds in the account are insufficient, screen 8 is displayed. If the account balance is sufficient, screen 11 is displayed and the money is dispensed. The balance is printed on the transaction receipt as it is for a balance request transaction. After the cash has been removed, the system displays screen 14.

When the "No" button is pressed in screens 10, 12, or 14, the system presents screen 15 and returns the customer's ATM card. Once the card is removed from the card slot, screen 1 is displayed. When the "Yes" button is pressed in screens 10, 12, or 14, the system presents screen 5 so the customer can select additional transactions.

2.5.2 Discussion

A surprising amount of information is "buried" in the system description just given. For instance, if you read it closely, you can infer that the terminal only contains $10 bills (see screen 7). This textual definition is probably more precise than what is usually encountered in practice. The example is deliberately simple (hence the name).

A plethora of questions could be resolved by a list of assumptions. For example, is there a borrowing limit? What keeps customers from taking out more than their actual balance if they go to several ATM terminals? A lot of start-up questions are used: how much cash is initially in the machine? How are new customers added to the system? These and other real-world refinements are eliminated to maintain simplicity.

2.6 The Currency Converter

The currency conversion program is another event-driven program that emphasizes code associated with a GUI. A sample GUI is shown in Figure 2.5.

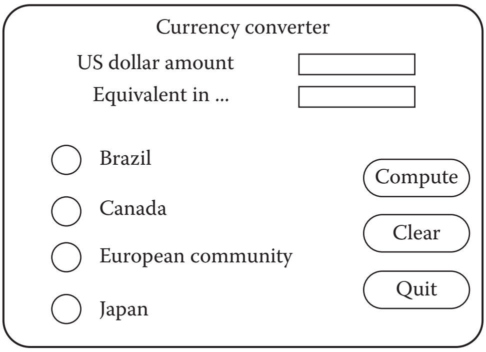


Figure 2.5 Currency converter graphical user interface.

The application converts US dollars to any of four currencies: Brazilian reals, Canadian dollars, European Union euros, and Japanese yen. Currency selection is governed by the radio buttons (option buttons), which are mutually exclusive. When a country is selected, the system responds by completing the label; for example, "Equivalent in …" becomes "Equivalent in Canadian dollars" if the Canada button is clicked. Also, a small Canadian flag appears next to the output position for the equivalent currency amount. Either before or after currency selection, the user inputs an amount in US dollars. Once both tasks are accomplished, the user can click on the Compute button, the Clear button, or the Quit button. Clicking on the Compute button results in the conversion of the US dollar amount to the equivalent amount in the selected currency. Clicking on the Clear button resets the currency selection, the US dollar amount, and the equivalent currency amount and the associated label. Clicking on the Quit button ends the application. This example nicely illustrates a description with UML and an object-oriented implementation, which we will use in Chapter 15.

2.7 Saturn Windshield Wiper Controller

The windshield wiper on some Saturn automobiles is controlled by a lever with a dial. The lever has four positions: OFF, INT (for intermittent), LOW, and HIGH; and the dial has three positions, numbered simply 1, 2, and 3. The dial positions indicate three intermittent speeds, and the dial position is relevant only when the lever is at the INT position. The decision table below shows the windshield wiper speeds (in wipes per minute) for the lever and dial positions.

c1. Lever	OFF	INT	INT	INT	LOW	HIGH
c2. Dial	n/a	1	2	3	n/a	n/a
a1. Wiper	0	4	6	12	30	60

2.8 Garage Door Opener

A system to open a garage door is composed of several components: a drive motor, a drive chain, the garage door wheel tracks, a lamp, and an electronic controller. This much of the system is powered by commercial 110 V electricity. Several devices communicate with the garage door controller—a wireless keypad (usually in an automobile), a digit keypad on the outside of the garage door, and a wall-mounted button. In addition, there are two safety features, a laser beam near the floor and an obstacle sensor. These latter two devices operate only when the garage door is closing. If the light beam is interrupted (possibly by a pet), the door immediately stops, and then reverses direction until the door is fully open. If the door encounters an obstacle while it is closing (say a child's tricycle left in the path of the door), the door stops and reverses direction until it is fully open. There is a third way to stop a door in motion, either when it is closing or opening. A signal from any of the three devices (wireless keypad, digit keypad, or wall-mounted control button). The response to any of these signals is different—the door stops in place. A subsequent signal from any of the devices starts the door in the same direction as when it was stopped. Finally, there are sensors that detect when the door has moved to one of the extreme positions, either fully open

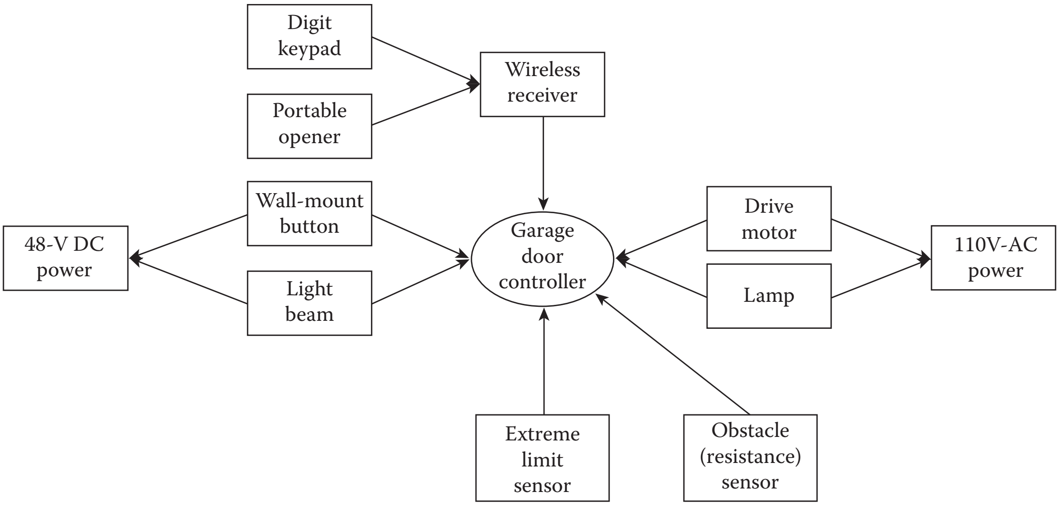


Figure 2.6 SysML diagram of garage door controller.

or fully closed. When the door is in motion, the lamp is lit, and remains lit for approximately 30 seconds after the door reaches one of the extreme positions.

The three signaling devices and the safety features are optional additions to the basic garage door opener. This example will be used in Chapter 17 in the discussion of systems of systems. For now, a SysML context diagram of the garage door opener is given in Figure 2.6.

EXERCISES

1. Revisit the traditional triangle program flowchart in Figure 2.1. Can the variable match ever have the value of 4? Of 5? Is it ever possible to "execute" the following sequence of numbered boxes: 1, 2, 5, 6?
2. Recall the discussion from Chapter 1 about the relationship between the specification and the implementation of a program. If you study the implementation of NextDate carefully, you will see a problem. Look at the CASE clause for 30-day months (4, 6, 9, 11). There is no special action for day = 31. Discuss whether this implementation is correct. Repeat this discussion for the treatment of values of day 29 in the CASE clause for February.
3. In Chapter 1, we mentioned that part of a test case is the expected output. What would you use as the expected output for a NextDate test case of June 31, 1812? Why?
4. One common addition to the triangle problem is to check for right triangles. Three sides constitute a right triangle if the Pythagorean relationship is satisfied: $c^2 = a^2 + b^2$. This change makes it convenient to require that the sides be presented in increasing order, that is, $a \leq b \leq c$. Extend the Triangle3 program to include the right triangle feature. We will use this extension in later exercises.
5. What will the Triangle2 program do for the sides –3, –3, 5? Discuss this in terms of the considerations we made in Chapter 1.
6. The function YesterDate is the inverse of NextDate. Given a month, day, year, YesterDate returns the date of the day before. Develop a program in your favorite language (or our generalized pseudocode) for YesterDate. We will also use this as a continuing exercise.

7. Part of the art of GUI design is to prevent user input errors. Event-driven applications are particularly vulnerable to input errors because events can occur in any order. As the given definition stands, a user could enter a US dollar amount and then click on the compute button without selecting a country. Similarly, a user could select a country and then click on the compute button without inputting a dollar amount. GUI designers use the concept of "forced navigation" to avoid such situations. In Visual Basic, this can be done using the visibility properties of various controls. Discuss how you could do this.
8. The CRC Press website (http://www.crcpress.com/product/isbn/9781466560680) contains some software supplements for this book. There is a series of exercises that I use in my graduate class in software testing; the first part of a continuing exercise is to use the naive.xls (runs in most versions of Microsoft Excel) program to test the triangle, NextDate, and commission problems. The spreadsheet lets you postulate test cases and then run them simply by clicking on the "Run Test Cases" button. As a start to becoming a testing craftsperson, use naive.xls to test our three examples in an intuitive (hence "naive") way. There are faults inserted into each program. If (when) you find failures, try to hypothesize the underlying fault. Keep your results for comparison to ideas in Chapters 5, 6, and 9.

References

Brown, J.R. and Lipov, M., Testing for software reliability, *Proceedings of the International Symposium on Reliable Software,* Los Angeles, April 1975, pp. 518–527.

Chellappa, M., Nontraversible paths in a program, *IEEE Transactions on Software Engineering,* Vol. SE-13, No. 6, June 1987, pp. 751–756.

Clarke, L.A. and Richardson, D.J., The application of error sensitive strategies to debugging, *ACM SIGSOFT Software Engineering Notes,* Vol. 8, No. 4, August 1983.

Clarke, L.A. and Richardson, D.J., A reply to Foster's comment on "The Application of Error Sensitive Strategies to Debugging," *ACM SIGSOFT Software Engineering Notes,* Vol. 9, No. 1, January 1984.

Gruenberger, F., Program testing, the historical perspective, in *Program Test Methods,* William C. Hetzel, Ed., Prentice-Hall, New York, 1973, pp. 11–14.

Hetzel, Bill, *The Complete Guide to Software Testing,* 2nd ed., QED Information Sciences, Inc., Wellesley, MA, 1988.

Inglis, Stuart J., *Planets, Stars, and Galaxies*, 4th Ed., John Wiley & Sons, New York, 1961.

Myers, G.J., *The Art of Software Testing,* Wiley Interscience, New York, 1979.

Pressman, R.S., *Software Engineering: A Practitioner's Approach*, McGraw-Hill, New York, 1982.

Chapter 3

Discrete Math for Testers

More than any other life cycle activity, testing lends itself to mathematical description and analysis. In this chapter and in the next, testers will find the mathematics they need. Following the craftsperson metaphor, the mathematical topics presented here are tools; a testing craftsperson should know how to use them well. With these tools, a tester gains rigor, precision, and efficiency—all of which improve testing. The "for testers" part of the chapter title is important: this chapter is written for testers who either have a sketchy math background or who have forgotten some of the basics. Serious mathematicians (or maybe just those who take themselves seriously) will likely be annoyed by the informal discussion here. Readers who are already comfortable with the topics in this chapter should skip to the next chapter and start right in on graph theory.

In general, discrete mathematics is more applicable to functional testing, while graph theory pertains more to structural testing. "Discrete" raises a question: What might be indiscrete about mathematics? The mathematical antonym is continuous, as in calculus, which software developers (and testers) seldom use. Discrete math includes set theory, functions, relations, propositional logic, and probability theory, each of which is discussed here.

3.1 Set Theory

How embarrassing to admit, after all the lofty expiation of rigor and precision, that no explicit definition of a set exists. This is really a nuisance because set theory is central to these two chapters on math. At this point, mathematicians make an important distinction: naive versus axiomatic set theory. In naive set theory, a set is recognized as a primitive term, much like point and line are primitive concepts in geometry. Here are some synonyms for "set": collection, group, and bunch—you get the idea. The important thing about a set is that it lets us refer to several things as a group, or a whole. For example, we might wish to refer to the set of months that have exactly 30 days (we need this set when we test the NextDate function from Chapter 2). In set theory notation, we write

$$M1 = \{\text{April, June, September, November}\}$$

and we read this notation as "$M1$ is the set whose elements are the months April, June, September, November."

3.1.1 Set Membership

The items in a set are called elements or members of the set, and this relationship is denoted by the symbol ∈. Thus, we could write April $\in M1$. When something is not a member of a set, we use the symbol ∉, so we might write December $\notin M1$.

3.1.2 Set Definition

A set is defined in three ways: by simply listing its elements, by giving a decision rule, or by constructing a set from other sets. The listing option works well for sets with only a few elements as well as for sets in which the elements obey an obvious pattern. We used this method in defining $M1$ above. We might define the set of allowable years in the NextDate program as follows:

$$Y = \{1812, 1813, 1814, \ldots, 2011, 2012\}$$

When we define a set by listing its elements, the order of the elements is irrelevant. We will see why when we discuss set equality. The decision rule approach is more complicated, and this complexity carries both advantages and penalties. We could define the years for NextDate as

$$Y = \{\text{year}: 1812 \le \text{year} \le 2012\}$$

which reads "Y is the set of all years such that (the colon is 'such that') the years are between 1812 and 2012 inclusive." When a decision rule is used to define a set, the rule must be unambiguous. Given any possible value of year, we can therefore determine whether or not that year is in our set Y.

The advantage of defining sets with decision rules is that the unambiguity requirement forces clarity. Experienced testers have encountered "untestable requirements." Many times, the reason that such requirements cannot be tested boils down to an ambiguous decision rule. In our triangle program, for example, suppose we defined a set

$$N = \{t: t \text{ is a nearly equilateral triangle}\}$$

We might say that the triangle with sides (500, 500, 501) is an element of N, but how would we treat the triangles with sides (50, 50, 51) or (5, 5, 6)?

A second advantage of defining sets with decision rules is that we might be interested in sets where the elements are difficult to list. In the commission problem, for example, we might be interested in the set

$$S = \{\text{sales: the 15\% commission rate applies to the total sale}\}$$

We cannot easily write down the elements of this set; however, given a particular value for sale, we can easily apply the decision rule.

The main disadvantage of decision rules is that they can become logically complex, particularly when they are expressed with the predicate calculus quantifiers ∃ ("there exists") and ∀ ("for all”). If everyone understands this notation, the precision is helpful. Too often customers are overwhelmed by statements with these quantifiers. A second problem with decision rules has to do with self-reference. This is interesting, but it really has very little application for testers. The problem arises when a decision rule refers to itself, which is a circularity. As an example, the Barber of Seville “is the man who shaves everyone who does not shave himself.”

3.1.3 The Empty Set

The empty set, denoted by the symbol ∅, occupies a special place in set theory. The empty set contains no elements. At this point, mathematicians will digress to prove a lot of facts about empty sets:

1. The empty set is unique; that is, there cannot be two empty sets (we will take their word for it).
2. ∅, {∅}, and {{∅}} are all different sets (we will not need this).

It is useful to note that, when a set is defined by a decision rule that is always false, the set is empty. For instance, $\emptyset = \{year: 2012 \leq year \leq 1812\}$.

3.1.4 Venn Diagrams

There are two traditional techniques to diagram relationships among sets: Venn diagrams and Euler diagrams. Both help visualize concepts that have already been expressed textually. The chair of my college mathematics department maintained that, in her words, “Mathematics is not a function of its diagrams.” Maybe not, but diagrams are certainly expressive, and they promote easy communication and understanding. Today, sets are commonly pictured by Venn diagrams—as in Chapter 1, when we discussed sets of specified and programmed behaviors. In a Venn diagram, a set is depicted as a circle; points in the interior of the circle correspond to elements of the set. Then, we might draw our set $M1$ of 30-day months as in Figure 3.1.

Venn diagrams were originally devised by John Venn, a British logician, in 1881. Most Venn diagrams show two or three overlapping circles. (It is impossible to show a Venn diagram of five sets showing all the possible intersections.) Shading is used in two opposite ways—most often, shaded regions are subsets of interest, but occasionally, shading is used to indicate an empty region. It is therefore important to include a legend explicitly stating the meaning of shading. Also, Venn diagrams should be placed within a rectangle that represents the universe of discourse. Figures 1.3 and 1.4 in Chapter 1 show examples of two- and three-set Venn diagrams. When the circles

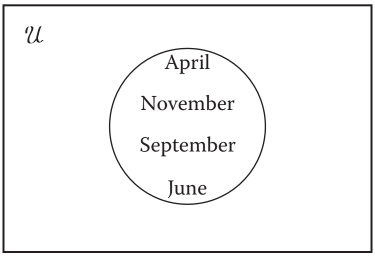


Figure 3.1 Venn diagram of set of 30-day months.

overlap, there is no presumption of relationships among the sets; at the same time, the overlapping describes all the potential intersections. Finally, there is no way to diagram the empty set.

Venn diagrams communicate various set relationships in an intuitive way, but some picky questions arise. What about finite versus infinite sets? Both can be drawn as Venn diagrams; in the case of finite sets, we cannot assume that every interior point corresponds to a set element. We do not need to worry about this, but it is helpful to know the limitations. Sometimes, we will find it helpful to label specific elements.

Another sticking point has to do with the empty set. How do we show that a set, or maybe a portion of a set, is empty? The common answer is to shade empty regions; however, this is often contradicted by other uses in which shading is used to highlight regions of interest. The best practice is to provide a legend that clarifies the intended meaning of shaded areas.

It is often helpful to think of all the sets in a discussion as being subsets of some larger set, known as the universe of discourse. We did this in Chapter 1 when we chose the set of all program behaviors as our universe of discourse. The universe of discourse can usually be guessed from given sets. In Figure 3.1, most people would take the universe of discourse to be the set of all months in a year. Testers should be aware that assumed universes of discourse are often sources of confusion. As such, they constitute a subtle point of miscommunication between customers and developers.

3.1.5 Set Operations

Much of the expressive power of set theory comes from basic operations on sets: union, intersection, and complement. Other handy operations are used: relative complement, symmetric difference, and Cartesian product. Each of these is defined next. In each of these definitions, we begin with two sets, A and B, contained in some universe of discourse U. The definitions use logical connectives from the propositional calculus: and ($\wedge$), or ($\vee$), exclusive–or ($\oplus$), and not ($\sim$).

Definition

Given sets A and B

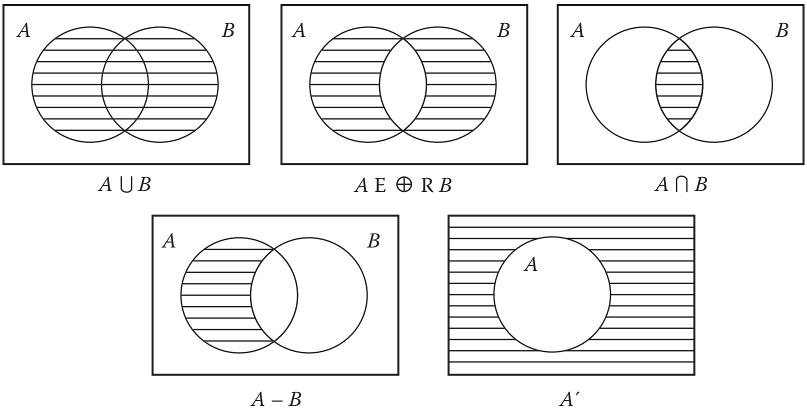
Their *union* is the set $A \cup B = \{x: x \in A \vee x \in B\}$.
Their *intersection* is the set $A \cap B = \{x: x \in A \wedge x \in B\}$.
The *complement* of A is the set $A' = \{x: x \notin A\}$.
The *relative complement of B with respect to A* is the set $A - B = \{x: x \in A \wedge x \notin B\}$.
The *symmetric difference of A and B* is the set $A \oplus B = \{x: x \in A \oplus x \in B\}$.

Venn diagrams for these sets are shown in Figure 3.2.

The intuitive expressive power of Venn diagrams is very useful for describing relationships among test cases and among items to be tested. Looking at the Venn diagrams in Figure 3.2, we might guess that

$$A \oplus B = (A \cup B) - (A \cap B)$$

This is the case, and we could prove it with propositional logic.



$A \cup B$ A E $\oplus$ R B $A \cap B$

$A - B$ A'

Figure 3.2 Venn diagrams of basic sets.

Venn diagrams are used elsewhere in software development: together with directed graphs, they are the basis of the StateCharts notations, which are among the most rigorous specification techniques supported by computer-aided software engineering (CASE) technology. StateCharts are also the control notation chosen for the UML, the Unified Modeling Language from the IBM Corp. and the Object Management Group.

The Cartesian product (also known as the cross product) of two sets is more complex; it depends on the notion of ordered pairs, which are two element sets in which the order of the elements is important. The usual notation for unordered and ordered pairs is

Unordered pair: (a, b)
Ordered pair: $<a, b>$

The difference is that, for $a \neq b$, $(a, b) = (b, a)$, but $<a, b> \neq <b, a>$. This distinction is important to the material in Chapter 4; as we shall see, the fundamental difference between ordinary and directed graphs is exactly the difference between unordered and ordered pairs.

Definition

The *Cartesian product of two sets A and B* is the set

$$A \times B = \{<x, y>: x \in A \wedge y \in B\}$$

Venn diagrams do not show Cartesian products, so we will look at a short example. The Cartesian product of the sets $A = \{1, 2, 3\}$ and $B = \{w, x, y, z\}$ is the set

$$A \times B = \{<1, w>, <1, x>, <1, y>, <1, z>, <2, w>, <2, x>, \\ <2, y>, <2, z>, <3, w>, <3, x>, <3, y>, <3, z>\}$$

The Cartesian product has an intuitive connection with arithmetic. The cardinality of a set A is the number of elements in A and is denoted by $|A|$. (Some authors prefer Card(A).) For sets A and

B, $|A \times B| = |A| \times |B|$. When we study functional testing in Chapter 5, we will use the Cartesian product to describe test cases for programs with several input variables. The multiplicative property of the Cartesian product means that this form of testing generates a large number of test cases.

3.1.6 Set Relations

We use set operations to construct interesting new sets from existing sets. When we do, we often would like to know something about the way the new and the old sets are related. Given two sets, A and B, we define three fundamental set relationships:

Definition

A is a subset of B, written $A \subseteq B$, if and only if (iff) $a \in A \Rightarrow a \in B$.
A is a proper subset of B, written $A \subset B$, iff $A \subseteq B \wedge B - A \neq \varnothing$.
A and B are equal sets, written $A = B$, iff $A \subseteq B \wedge B \subseteq A$.

In plain English, set A is a subset of set B if every element of A is also an element of B. To be a proper subset of B, A must be a subset of B and there must be some element in B that is not an element of A. Finally, the sets A and B are equal if each is a subset of the other.

3.1.7 Set Partitions

A partition of a set is a very special situation that is extremely important for testers. Partitions have several analogs in everyday life: we might put up partitions to separate an office area into individual offices; we also encounter political partitions when a state is divided up into legislative districts. In both of these, notice that the sense of "partition" is to divide up a whole into pieces such that everything is in some piece and nothing is left out. More formally:

Definition

Given a set A, and a set of subsets $A_1, A_2, \ldots, A_n$ of A, the subsets are a *partition of A* iff

$$A_1 \cup A_2 \cup \ldots \cup A_n = A, \text{ and } i \neq j \Rightarrow A_i \cap A_j = \varnothing.$$

Because a partition is a set of subsets, we frequently refer to individual subsets as elements of the partition.

The two parts of this definition are important for testers. The first part guarantees that every element of A is in some subset, while the second part guarantees that no element of A is in two of the subsets.

This corresponds well with the legislative district example: everyone is represented by some legislator, and nobody is represented by two legislators. A jigsaw puzzle is another good example of a partition; in fact, Venn diagrams of partitions are often drawn like puzzles, as in Figure 3.3.

Partitions are helpful to testers because the two definitional properties yield important assurances: completeness (everything is somewhere) and nonredundancy. When we study functional

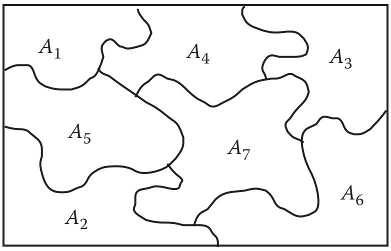


Figure 3.3 Venn diagram of a partition.

testing, we shall see that its inherent weakness is the vulnerability to both gaps and redundancies: some things may remain untested, while others are tested repeatedly. One of the difficulties of functional testing centers is finding an appropriate partition. In the triangle program, for example, the universe of discourse is the set of all triplets of positive integers. (Note that this is actually a Cartesian product of the set of positive integers with itself three times.) We might partition this universe three ways:

1. Into triangles and nontriangles
2. Into equilateral, isosceles, scalene, and nontriangles
3. Into equilateral, isosceles, scalene, right, and nontriangles

At first these partitions seem okay, but there is a problem with the last partition. The sets of scalene and right triangles are not disjoint (the triangle with sides 3, 4, 5 is a right triangle that is scalene).

3.1.8 Set Identities

Set operations and relations, when taken together, yield an important class of set identities that can be used to algebraically simplify complex set expressions. Math students usually have to derive all these; we will just list them and (occasionally) use them.

Name	*Expression*
Identity laws	$A \cup \varnothing = A$ $A \cap U = A$
Domination laws	$A \cup U = U$ $A \cap \varnothing = \varnothing$
Idempotent laws	$A \cup A = A$ $A \cap A = A$
Complementation laws	$(A')' = A$
Commutative laws	$A \cup B = B \cup A$ $A \cap B = B \cap A$

Associative laws	$A \cup (B \cup C) = (A \cup B) \cup C$ $A \cap (B \cap C) = (A \cap B) \cap C$
Distributive laws	$A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$ $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$
DeMorgan's laws	$(A \cup B)' = A' \cap B'$ $(A \cap B)' = A' \cup B'$

3.2 Functions

Functions are a central notion to software development and testing. The whole functional decomposition paradigm, for example, implicitly uses the mathematical notion of a function. Informally, a function associates elements of sets. In the NextDate program, for example, the function of a given date is the date of the following day, and in the triangle problem, the function of three input integers is the kind of triangle formed by sides with those lengths. In the commission problem, the salesperson's commission is a function of sales, which in turn is a function of the number of locks, stocks, and barrels sold. Functions in the SATM system are much more complex; not surprisingly, this will add complexity to the testing.

Any program can be thought of as a function that associates its outputs with its inputs. In the mathematical formulation of a function, the inputs are the domain and the outputs are the range of the function.

Definition

Given sets A and B, a function f is a subset of $A \times B$ such that for $a_i, a_j \in A$, $b_i, b_j \in B$, and $f(a_i) = b_i$, $f(a_j) = b_j$, $b_i \neq b_j \Rightarrow a_i \neq a_j$.

Formal definitions like this one are notoriously terse, so let us take a closer look. The inputs to the function f are elements of the set A, and the outputs of f are elements of B. What the definition says is that the function f is "well behaved" in the sense that an element in A is never associated with more than one element of B. (If this could happen, how would we ever test such a function? This would be an example of nondeterminism.)

3.2.1 Domain and Range

In the definition just given, the set A is the domain of the function f and the set B is the range. Because input and output have a "natural" order, it is an easy step to say that a function f is really a set of ordered pairs in which the first element is from the domain and the second element is from the range. Here are two common notations for function:

$$f: A \rightarrow B$$

$$f \subseteq A \times B$$

We have not put any restrictions on the sets A and B in this definition. We could have $A = B$, and either A or B could be a Cartesian product of other sets.

3.2.2 Function Types

Functions are further described by particulars of the mapping. In the definition below, we start with a function $f: A \rightarrow B$, and we define the set

$$f(A) = \{b_i \in B: b_i = f(a_i) \text{ for some } a_i \in A\}$$

This set is sometimes called the image of A under f.

Definition

f is a *function* from A *onto* B iff $f(A) = B$
f is a *function* from A *into* B iff $f(A) \subset B$ (note the proper subset here!)
f is a *one-to-one function* from A to B iff, for all $a_i, a_j \in A$, $a_i \neq a_j \Rightarrow f(a_i) \neq f(a_j)$
f is a *many-to-one function* from A to B iff, there exists $a_i, a_j \in A$, $a_i \neq a_j$ such that $f(a_i) = f(a_j)$.

Back to plain English, if f is a function from A onto B, we know that every element of B is associated with some element of A. If f is a function from A into B, we know that there is at least one element of B that is not associated with an element of A. One-to-one functions guarantee a form of uniqueness: distinct domain elements are never mapped to the same range element. (Notice this is the inverse of the "well-behaved" attribute described earlier.) If a function is not one-to-one, it is many-to-one; that is, more than one domain element can be mapped to the same range element. In these terms, the well-behaved requirement prohibits functions from being one-to-many. Testers familiar with relational databases will recognize that all these possibilities (one-to-one, one-to-many, many-to-one, and many-to-many) are allowed for relations.

Referring again to our testing examples, suppose we take A, B, and C to be sets of dates for the NextDate program, where

$$A = \{\text{date: 1 January 1812} \leq \text{date} \leq \text{31 December 2012}\}$$

$$B = \{\text{date: 2 January 1812} \leq \text{date} \leq \text{1 January 2013}\}$$

$$C = A \cup B$$

Now, NextDate: $A \rightarrow B$ is a one-to-one, onto function, and NextDate: $A \rightarrow C$ is a one-to-one, into function.

It makes no sense for NextDate to be many-to-one, but it is easy to see how the triangle problem can be many-to-one. When a function is one-to-one and onto, such as NextDate: $A \rightarrow B$ previously, each element of the domain corresponds to exactly one element of the range; conversely, each element of the range corresponds to exactly one element of the domain. When this happens, it is always possible to find an inverse function (see the YesterDate exercise in Chapter 2) that is one-to-one from the range back to the domain.

All this is important for testing. The into versus onto distinction has implications for domain- and range-based functional testing, and one-to-one functions may require much more testing than many-to-one functions.

3.2.3 Function Composition

Suppose we have sets and functions such that the range of one is the domain of the next:

$$f\colon A \to B$$

$$g\colon B \to C$$

$$h\colon C \to D$$

When this occurs, we can compose the functions. To do this, let us refer to specific elements of the domain and range sets $a \in A$, $b \in B$, $c \in C$, $d \in D$, and suppose that $f(a) = b$, $g(b) = c$, and $h(c) = d$. Now the composition of functions h, g, and f is

$$\begin{aligned} h \circ g \circ f(a) &= h(g(f(a))) \\ &= h(g(b)) \\ &= h(c) \\ &= d \end{aligned}$$

Function composition is a very common practice in software development; it is inherent in the process of defining procedures and subroutines. We have an example of it in the commission program, in which

$$f_1\ (\text{locks, stocks, barrels}) = \text{sales}$$

$$f_2\ (\text{sales}) = \text{commission}$$

$$\text{So } f_2\ (f_1(\text{locks, stocks, barrels}) = \text{commission}$$

Composed chains of functions can be problematic for testers, particularly when the range of one function is a proper subset of the domain of the "next" function in the chain. A special case of composition can be used, which helps testers in a curious way. Recall we discussed how one-to-one onto functions always have an inverse function. It turns out that this inverse function is unique and is guaranteed to exist (again, the math folks would prove this). If f is a one-to-one function from A onto B, we denote its unique inverse by f^{-1}. It turns out that for $a \in A$ and $b \in B$, $f^{-1} \cdot f(a) = a$ and $f \cdot f^{-1}(b) = b$. The NextDate and YesterDate programs are such inverses. The way this helps testers is that, for a given function, its inverse acts as a "cross-check," and this can often expedite the identification of functional test cases.

3.3 Relations

Functions are a special case of a relation: both are subsets of some Cartesian product; however, in the case of functions, we have the well-behaved requirement that says that a domain element cannot be associated with more than one range element. This is borne out in everyday usage: when we say something "is a function" of something else, our intent is that there is a deterministic relationship present. Not all relationships are strictly functional. Consider the mapping between a set of patients and a set of physicians. One patient may be treated by several physicians, and one physician may treat several patients—a many-to-many mapping.

3.3.1 Relations among Sets

Definition

Given two sets A and B, a relation R is a subset of the Cartesian product $A \times B$.

Two notations are popular; when we wish to speak about the entire relation, we usually just write $R \subseteq A \times B$; for specific elements $a_i \in A$, $b_i \in B$, we write $a_i\ R\ b_i$. Most math texts omit treatment of relations; we are interested in them because they are essential to both data modeling and object-oriented analysis.

Next, we have to explain an overloaded term—cardinality. Recall that, as it applies to sets, cardinality refers to the number of elements in a set. Because a relation is also a set, we might expect that the cardinality of a relation refers to how many ordered pairs are in the set $R \subseteq A \times B$. Unfortunately, this is not the case.

Definition

Given two sets A and B, a relation $R \subseteq A \times B$, the cardinality of relation R is

One-to-one iff R is a one-to-one function from A to B
Many-to-one iff R is a many-to-one function from A to B
One-to-many iff at least one element $a \in A$ is in two ordered pairs in R, that is $<a, b_i> \in R$ and $<a, b_j> \in R$
Many-to-many iff at least one element $a \in A$ is in two ordered pairs in R, that is $<a, b_i> \in R$ and $<a, b_j> \in R$ and at least one element $b \in B$ is in two ordered pairs in R, that is $<a_i, b> \in R$ and $<a_j, b> \in R$

The distinction between functions into and onto their range has an analog in relations—the notion of participation.

Definition

Given two sets A and B, a relation $R \subseteq A \times B$, the participation of relation R is

Total iff every element of A is in some ordered pair in R
Partial iff some element of A is not in some ordered pair in R
Onto iff every element of B is in some ordered pair in R
Into iff some element of B is not in some ordered pair in R

In plain English, a relation is total if it applies to every element of A and partial if it does not apply to every element. Another term for this distinction is mandatory versus optional participation. Similarly, a relation is onto if it applies to every element of B and into if it does not. The parallelism between total/partial and onto/into is curious and deserves special mention here. From the standpoint of relational database theory, no reason exists for this; in fact, a compelling reason exists to avoid this distinction. Data modeling is essentially declarative, while process modeling is essentially imperative. The parallel sets of terms force a direction on relations, when in fact no need exists for the directionality. Part of this is a likely holdover from the fact that Cartesian products consist of ordered pairs, which clearly have a first and second element.

Thus far, we have only considered relations between two sets. Extending relations to three or more sets is more complicated than simply the Cartesian product. Suppose, for example, we had three sets, A, B, and C, and a relation $R \subseteq A \times B \times C$. Do we intend the relation to be strictly among three elements, or is it between one element and an ordered pair (there would be three possibilities here)? This line of thinking also needs to be applied to the definitions of cardinality and participation. It is straightforward for participation, but cardinality is essentially a binary property. (Suppose, for example, the relation is one-to-one from A to B and is many-to-one from A to C.) We discussed a three-way relation in Chapter 1, when we examined the relationships among specified, implemented, and tested program behaviors. We would like to have some form of totality between test cases and specification–implementation pairs; we will revisit this when we study functional and structural testing.

Testers need to be concerned with the definitions of relations because they bear directly on software properties to be tested. The onto/into distinction, for example, bears directly on what we will call output-based functional testing. The mandatory–optional distinction is the essence of exception handling, which also has implications for testers.

3.3.2 Relations on a Single Set

Two important mathematical relations are used, both of which are defined on a single set: ordering relations and equivalence relations. Both are defined with respect to specific properties of relations.

Let A be a set, and let $R \subseteq A \times A$ be a relation defined on A, with $<a, a>$, $<a, b>$, $<b, a>$, $<b, c>$, $<a, c> \in R$. Relations have four special attributes:

Definition

A relation $R \subseteq A \times A$ is

Reflexive iff for all $a \in A$, $<a, a> \in R$
Symmetric iff $<a, b> \in R \Rightarrow <b, a> \in R$
Antisymmetric $<a, b>, <b, a> \in R \Rightarrow a = b$
Transitive iff $<a, b>, <b, c> \in R \Rightarrow <a, c> \in R$

Family relationships are nice examples of these properties. You might want to think about the following relationships and decide for yourself which attributes apply: brother of, sibling of, and ancestor of. Now we can define the two important relations.

Definition

A relation $R \subseteq A \times A$ is an *ordering relation* if R is reflexive, antisymmetric, and transitive.

Ordering relations have a sense of direction; some common ordering relations are *older than*, ≥, ⇒, and *ancestor of.* (The reflexive part usually requires some fudging—we really should say *not younger than* and *not a descendant of.*) Ordering relations are a common occurrence in software: data access techniques, hashing codes, tree structures, and arrays are all situations in which ordering relations are used.

The power set of a given set is the set of all subsets of the given set. The power set of the set A is denoted $P(A)$. The subset relation $\subseteq$ is an ordering relation on $P(A)$ because it is reflexive (any set is trivially a subset of itself), it is antisymmetric (the definition of set equality), and it is transitive.

Definition

A relation $R \subseteq A \times A$ is an *equivalence relation* if R is reflexive, symmetric, and transitive.

Mathematics is full of equivalence relations: equality and congruence are two quick examples. A very important connection exists between equivalence relations and partitions of a set. Suppose we have some partition $A_1, A_2, \ldots, A_n$ of a set B, and we say that two elements, b_1 and b_2 of B, are related (i.e., $b_1 \; R \; b_2$) if b_1 and b_2 are in the same partition element. This relation is reflexive (any element is in its own partition), it is symmetric (if b_1 and b_2 are in a partition element, then b_2 and b_1 are), and it is transitive (if b_1 and b_2 are in the same set, and if b_2 and b_3 are in the same set, then b_1 and b_3 are in the same set). The relation defined from the partition is called the equivalence relation induced by the partition. The converse process works in the same way. If we start with an equivalence relation defined on a set, we can define subsets according to elements that are related to each other. This turns out to be a partition, and is called the partition induced by the equivalence relation. The sets in this partition are known as equivalence classes. The end result is that partitions and equivalence relations are interchangeable, and this becomes a powerful concept for testers. Recall that the two properties of a partition are notions of completeness and nonredundancy. When translated into testing situations, these notions allow testers to make powerful, absolute statements about the extent to which a software item has been tested. In addition, great efficiency follows from testing just one element of an equivalence class and assuming that the remaining elements will behave similarly.

3.4 Propositional Logic

We have already been using propositional logic notation; if you were perplexed by this usage definition before, you are not alone. Set theory and propositional logic have a chicken-and-egg relationship—it is hard to decide which should be discussed first. Just as sets are taken as primitive terms and are therefore not defined, we take propositions to be primitive terms. A proposition is a sentence that is either true or false, and we call these the truth values of the proposition. Furthermore, propositions are unambiguous: given a proposition, it is always possible to tell whether it is true or false. The sentence "Mathematics is difficult" would not qualify as a proposition because of the ambiguity. There are also temporal and spatial aspects of propositions. For example, "It is raining" may be true at some times at false at others. In addition, it may be true for one person and false for another at the same time but different locations.

We usually denote propositions with lower-case letters, p, q, and r. Propositional logic has operations, expressions, and identities that are very similar (in fact, they are isomorphic) to set theory.

3.4.1 Logical Operators

Logical operators (also known as logical connectives or operations) are defined in terms of their effect on the truth values of the propositions to which they are applied. This is easy; only two values are used: *T* (for true) and *F* (for false). Arithmetic operators could also be defined this way (in fact, that is how they are taught to children), but the tables become too large. The three basic logical operators are *and* (∧), *or* (∨), and *not* (~); these are sometimes called conjunction, disjunction, and negation. Negation is the only unary (one operand) logical operator; the others are all binary. These, and other logical operators, are defined by "truth tables."

p	q	$p \wedge q$	$p \vee q$	$\sim p$
T	T	T	T	F
T	F	F	T	F
F	T	F	T	T
F	F	F	F	T

Conjunction and disjunction are familiar in everyday life: a conjunction is true only when all components are true, and a disjunction is true if at least one component is true. Negations also behave as we expect. Two other common connectives are used: exclusive–or (⊕) and IF–THEN (→). They are defined as follows:

p	q	$p \oplus q$	$p \rightarrow q$
T	T	F	T
T	F	T	F
F	T	T	T
F	F	F	T

An exclusive–or is true only when exactly one of the propositions is true, while a disjunction (or inclusive–or) is true also when both propositions are true. The IF–THEN connective usually causes the most difficulty. The easy view is that this is just a definition; however, because the other connectives all transfer nicely to natural language, we have similar expectations for IF–THEN. The quick answer is that the IF–THEN connective is closely related to the process of deduction: in a valid deductive syllogism, we can say "if premises, then conclusion" and the IF–THEN statement will be a tautology.

3.4.2 Logical Expressions

We use logical operators to build logical expressions in exactly the same way that we use arithmetic operators to build algebraic expressions. We can specify the order in which operators are applied with the usual conventions on parentheses, or we can employ a precedence order (negation first, then conjunction followed by disjunction). Given a logical expression, we can always find its truth table by "building up" to it following the order determined by the parentheses. For example, the expression $\sim((p \to q) \wedge (q \to p))$ has the following truth table:

p	q	$p \to q$	$q \to p$	$(p \to q) \wedge (q \to p)$	$\sim((p \to q) \wedge (q \to p))$
T	T	T	T	T	F
T	F	F	T	F	T
F	T	T	F	F	T
F	F	T	T	T	F

3.4.3 Logical Equivalence

The notions of arithmetic equality and identical sets have analogs in propositional logic. Notice that the expressions $\sim((p \to q) \wedge (q \to p))$ and $p \oplus q$ have identical truth tables. This means that, no matter what truth values are given to the base propositions p and q, these expressions will always have the same truth value. This property can be defined in several ways; we use the simplest.

Definition

Two propositions p and q are *logically equivalent* (denoted $p \Leftrightarrow q$) iff their truth tables are identical.

By the way, the curious "iff" abbreviation we have been using for "if and only if" is sometimes called the bi-conditional, so the proposition p iff q is really $(p \to q) \wedge (q \to p)$, which is denoted $p \Leftrightarrow q$.

Definition

A proposition that is always true is a *tautology*; a proposition that is always false is a *contradiction*.

To be a tautology or a contradiction, a proposition must contain at least one connective and two or more primitive propositions. We sometimes denote a tautology as a proposition T, and a contradiction as a proposition F. We can now state several laws that are direct analogs of the ones we had for sets.

Law	*Expression*
Identity	$p \wedge T \Leftrightarrow p$ $p \vee F \Leftrightarrow p$
Domination	$p \vee T \Leftrightarrow T$ $p \wedge F \Leftrightarrow F$

Law	*Expression*
Idempotent	$p \wedge p \Leftrightarrow p$ $p \vee p \Leftrightarrow p$
Complementation	$\sim(\sim p) \Leftrightarrow p$
Commutative	$p \wedge q \Leftrightarrow q \wedge p$ $p \vee q \Leftrightarrow q \vee p$
Associative	$p \wedge (q \wedge r) \Leftrightarrow (p \wedge q) \wedge r$ $p \vee (q \vee r) \Leftrightarrow (p \vee q) \vee r$
Distributive	$p \wedge (q \vee r) \Leftrightarrow (p \wedge q) \vee (p \wedge r)$ $p \vee (q \wedge r) \Leftrightarrow (p \vee q) \wedge (p \vee r)$
DeMorgan's laws	$\sim(p \wedge q) \Leftrightarrow \sim p \vee \sim q$ $\sim(p \vee q) \Leftrightarrow \sim p \wedge \sim q$

3.5 Probability Theory

We will have two occasions to use the probability theory in our study of software testing: one deals with the probability that a particular path of statements executes, and the other generalizes this to a popular industrial concept called an operational profile (see Chapter 14). Because of this limited use, we will only cover the rudiments here.

As with both set theory and propositional logic, we start out with a primitive concept—the probability of an event. Here is the definition provided by a classic textbook (Rosen, 1991):

> The probability of an event E, which is a subset of a finite sample space S of equally likely outcomes, is $p(E) = |E|/|S|$.

This definition hinges on the idea of an experiment that results in an outcome, the sample space is the set of all possible outcomes, and an event is a subset of outcomes. This definition is circular: What are "equally likely" outcomes? We assume these have equal probabilities, but then probability is defined in terms of itself.

The French mathematician Laplace had a reasonable working definition of probability two centuries ago. To paraphrase it, the probability that something occurs is the number of favorable ways it can occur divided by the total number of ways (favorable and unfavorable). Laplace's definition works well when we are concerned with drawing colored marbles out of a bag (probability folks are unusually concerned with their marbles; maybe there is a lesson here), but it does not extend well to situations in which it is hard to enumerate the various possibilities.

We will use our (refurbished) capabilities in set theory and propositional logic to arrive at a more cohesive formulation. As testers, we will be concerned with things that happen; we will call these events and say that the set of all events is our universe of discourse. Next, we will devise propositions about events, such that the propositions refer to elements in the universe of discourse. Now, for some universe U and some proposition p about elements of U, we make a definition:

Definition

The truth set T of a proposition p, written $T(p)$, is the set of all elements in the universe U for which p is true.

Propositions are either true or false; therefore, a proposition p divides the universe of discourse into two sets, $T(p)$ and $(T(p))'$, where $T(p) \cup (T(p))' = U$. Notice that $(T(p))'$ is the same as $T(\sim p)$. Truth sets facilitate a clear mapping among set theory, propositional logic, and probability theory.

Definition

The probability that a proposition p is true, denoted $\Pr(p)$, is $|T(p)|/|U|$.

With this definition, Laplace's "number of favorable ways" becomes the cardinality of the truth set $T(p)$, and the total number of ways becomes the cardinality of the universe of discourse. This forces one more connection: because the truth set of a tautology is the universe of discourse, and the truth set of a contradiction is the empty set, the probabilities of $\varnothing$ and U are 0 and 1, respectively.

The NextDate problem is a good source of examples. Consider the month variable and the proposition

$$p(m)\text{: } m \text{ is a 30-day month}$$

The universe of discourse is the set U = {Jan., Feb., …, Dec.}, and the truth set of $p(m)$ is the set

$$T(p(m)) = \{\text{Apr., June, Sept., Nov.}\}$$

Now, the probability that a given month is a 30-day month is

$$\Pr(p(m)) = |T(p(m))|/|U| = 4/12$$

A subtlety exists in the role of the universe of discourse; this is part of the craft of using probability theory in testing—choosing the right universe. Suppose we want to know the probability that a month is February. The quick answer: 1/12. Now, suppose we want the probability of a month with exactly 29 days. Less easy—we need a universe that includes both leap years and common years. We could use congruence arithmetic and choose a universe that consists of months in a period of four consecutive years—say 1991, 1992, 1993, and 1994. This universe would contain 48 "months," and in this universe the probability of a 29-day month is 1/48. Another possibility would be to use the 200-year range of the NextDate program, in which the year 1900 is not a leap year. This would slightly reduce the probability of a 29-day month. One conclusion: getting the right universe is important. A bigger conclusion: it is even more important to avoid "shifting universes."

Here are some facts about probabilities that we will use without proof. They refer to a given universe, propositions p and q, with truth sets $T(p)$ and $T(q)$:

$$\Pr(\sim p) = 1 - \Pr(p)$$

$$\Pr(p \wedge q) = \Pr(p) \times \Pr(q)$$

$$\Pr(p \vee q) = \Pr(p) + \Pr(q) - \Pr(p \wedge q)$$

These facts, together with the tables of set theory and propositional identities, provide a strong algebraic capability to manipulate probability expressions.

EXERCISES

1. A very deep connection (an isomorphism) exists between set operations and the logical connectives in the propositional logic.

Operation	*Propositional Logic*	*Set Theory*
Disjunction	Or	Union
Conjunction	And	Intersection
Negation	Not	Complement
Implication	If, Then	Subset
	Exclusive or	Symmetric difference

   a. Express $A \oplus B$ in words.
   b. Express $(A \approx B) - (A \cap B)$ in words.
   c. Convince yourself that $A \oplus B$ and $(A \approx B) - (A \cap B)$ are the same set.
   d. Is it true that $A \oplus B = (A - B) \approx (B - A)$?
   e. What name would you give to the blank entry in the previous table?
2. In many parts of the United States, real estate taxes are levied by different taxing bodies, for example, a school district, a fire protection district, a township, and so on. Discuss whether these taxing bodies form a partition of a state. Do the 50 states form a partition of the United States of America? (What about the District of Columbia?)
3. Is brotherOf an equivalence relation on the set of all people? How about siblingOf?

Reference

Rosen, K.H., *Discrete Mathematics and Its Applications*, McGraw-Hill, New York, 1991.

Chapter 4

Graph Theory for Testers

Graph theory is a branch of topology that is sometimes referred to as "rubber sheet geometry." Curious, because the rubber sheet parts of topology have little to do with graph theory; furthermore, the graphs in graph theory do not involve axes, scales, points, and curves as you might expect. Whatever the origin of the term, graph theory is probably the most useful part of mathematics for computer science—far more useful than calculus—yet it is not commonly taught. Our excursion into graph theory will follow a "pure math" spirit in which definitions are as devoid of specific interpretations as possible. Postponing interpretations results in maximum latitude in interpretations later, much like well-defined abstract data types promote reuse.

Two basic kinds of graphs are used: undirected and directed. Because the latter are a special case of the former, we begin with undirected graphs. This will allow us to inherit many concepts when we get to directed graphs.

4.1 Graphs

A graph (also known as a linear graph) is an abstract mathematical structure defined from two sets—a set of nodes and a set of edges that form connections between nodes. A computer network is a fine example of a graph. More formally:

Definition

A *graph* $G = (V, E)$ is composed of a finite (and nonempty) set V of nodes and a set E of unordered pairs of nodes.

$$V = \{n_1, n_2, \ldots, n_m\}$$

and

$$E = \{e_1, e_2, \ldots, e_p\}$$

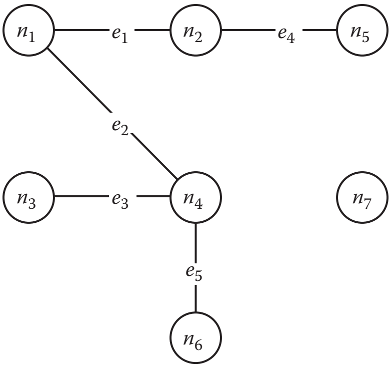


Figure 4.1 Graph with seven nodes and five edges.

where each edge $e_k = \{n_i, n_j\}$ for some nodes $n_i, n_j \in V$. Recall from Chapter 3 that the set $\{n_i, n_j\}$ is an unordered pair, which we sometimes write as (n_i, n_j).

Nodes are sometimes called vertices; edges are sometimes called arcs; and we sometimes call nodes the endpoints of an arc. The common visual form of a graph shows nodes as circles and edges as lines connecting pairs of nodes, as in Figure 4.1. We will use this figure as a continuing example, so take a minute to become familiar with it.

In the graph in Figure 4.1, the node and edge sets are

$$V = \{n_1, n_2, n_3, n_4, n_5, n_6, n_7\}$$

$$E = \{e_1, e_2, e_3, e_4, e_5\}$$

$$= \{(n_1, n_2), (n_1, n_4), (n_3, n_4),(n_2, n_5), (n_4, n_6)\}$$

To define a particular graph, we must first define a set of nodes and then define a set of edges between pairs of nodes. We usually think of nodes as program statements, and we have various kinds of edges, representing, for instance, flow of control or define/use relationships.

4.1.1 Degree of a Node

Definition

The *degree of a node* in a graph is the number of edges that have that node as an endpoint. We write deg(n) for the degree of node n.

We might say that the degree of a node indicates its "popularity" in a graph. In fact, social scientists use graphs to describe social interactions, in which nodes are people, edges often refer to things like "friendship," "communicates with," and so on. If we make a graph in which objects are nodes and edges are messages, the degree of a node (object) indicates the extent of integration testing that is appropriate for the object.

The degrees of the nodes in Figure 4.1 are

$$\deg(n_1) = 2$$
$$\deg(n_2) = 2$$
$$\deg(n_3) = 1$$
$$\deg(n_4) = 3$$
$$\deg(n_5) = 1$$
$$\deg(n_6) = 1$$
$$\deg(n_7) = 0$$

4.1.2 Incidence Matrices

Graphs need not be represented pictorially—they can be fully represented in an incidence matrix. This concept becomes very useful for testers, so we will formalize it here. When graphs are given a specific interpretation, the incidence matrix always provides useful information for the new interpretation.

Definition

The *incidence matrix* of a graph $G = (V, E)$ with m nodes and n edges is an $m \times n$ matrix, where the element in row i, column j is a 1 if and only if node i is an endpoint of edge j; otherwise, the element is 0.

The incidence matrix of the graph in Figure 4.1 is as follows:

	e_1	e_2	e_3	e_4	e_5
n_1	1	1	0	0	0
n_2	1	0	0	1	0
n_3	0	0	1	0	0
n_4	0	1	1	0	1
n_5	0	0	0	1	0
n_6	0	0	0	0	1
n_7	0	0	0	0	0

We can make some observations about a graph by examining its incidence matrix. First, notice that the sum of the entries in any column is 2. That is because every edge has exactly two endpoints. If a column sum in an incidence matrix is ever something other than 2, there is a mistake somewhere. Thus, forming column sums is a form of integrity checking similar in spirit to that of parity checks. Next, we see that the row sum is the degree of the node. When the degree of a node is zero, as it is for node n_7, we say the node is isolated. (This might correspond to unreachable code or to objects that are included but never used.)

4.1.3 Adjacency Matrices

The adjacency matrix of a graph is a useful supplement to the incidence matrix. Because adjacency matrices deal with connections, they are the basis of many later graph theory concepts.

Definition

The *adjacency matrix* of a graph $G = (V, E)$ with m nodes is an $m \times m$ matrix, where the element in row i, column j is a 1 if and only if an edge exists between node i and node j; otherwise, the element is 0.

The adjacency matrix is symmetric (element i,j always equals element j,i), and a row sum is the degree of the node (as it was in the incidence matrix).

The adjacency matrix of the graph in Figure 4.1 is as follows:

	n_1	n_2	n_3	n_4	n_5	n_6	n_7
n_1	0	1	0	1	0	0	0
n_2	1	0	0	0	1	0	0
n_3	0	0	0	1	0	0	0
n_4	1	0	1	0	0	1	0
n_5	0	1	0	0	0	0	0
n_6	0	0	0	1	0	0	0
n_7	0	0	0	0	0	0	0

4.1.4 Paths

As a preview of how we will use graph theory, the code-based approaches to testing (see Chapters 8 and 9) all center on types of paths in a program. Here, we define (interpretation-free) paths in a graph.

Definition

A *path* is a sequence of edges such that for any adjacent pair of edges e_i, e_j in the sequence, the edges share a common (node) endpoint.

Paths can be described either as sequences of edges or as sequences of nodes; the node sequence choice is more common.

Some paths in the graph in Figure 4.1:

Path	*Node Sequence*	*Edge Sequence*
Between n_1 and n_5	n_1, n_2, n_5	e_1, e_4
Between n_6 and n_5	n_6, n_4, n_1, n_2, n_5	e_5, e_2, e_1, e_4
Between n_3 and n_2	n_3, n_4, n_1, n_2	e_3, e_2, e_1

Paths can be generated directly from the adjacency matrix of a graph using a binary form of matrix multiplication and addition. In our continuing example, edge e_1 is between nodes n_1 and n_2, and edge e_4 is between nodes n_2 and n_5. In the product of the adjacency matrix with itself, the element in position (1,2) forms a product with the element in position (2,5), yielding an element in position (1,5), which corresponds to the two-edge path between n_1 and n_5. If we multiplied the product matrix by the original adjacency matrix again, we would get all three edge paths, and so on. At this point, the pure math folks go into a long digression to determine the length of the longest path in a graph; we will not bother. Instead, we focus our interest on the fact that paths connect "distant" nodes in a graph.

The graph in Figure 4.1 predisposes a problem. It is not completely general because it does not show all the situations that might occur in a graph. In particular, no paths exist in which a node occurs twice in the path. If it did, the path would be a loop (or circuit). We could create a circuit by adding an edge between nodes n_3 and n_6.

4.1.5 Connectedness

Paths let us speak about nodes that are connected; this leads to a powerful simplification device that is very important for testers.

Definition

Two *nodes are connected* if and only if they are in the same path.

"Connectedness" is an equivalence relation (see Chapter 3) on the node set of a graph. To see this, we can check the three defining properties of equivalence relations:

1. Connectedness is reflexive because every node is, by default, in a path of length 0 with itself. (Sometimes, for emphasis, an edge is shown that begins and ends on the same node.)
2. Connectedness is symmetric because if nodes n_i and n_j are in a path, then nodes n_j and n_i are in the same path.
3. Connectedness is transitive (see the discussion of adjacency matrix multiplication for paths of length 2).

Equivalence relations induce a partition (see Chapter 3 if you need a reminder); therefore, we are guaranteed that connectedness defines a partition on the node set of a graph. This permits the definition of components of a graph.

Definition

A *component of a graph* is a maximal set of connected nodes.

Nodes in the equivalence classes are components of the graph. The classes are maximal due to the transitivity part of the equivalence relation. The graph in Figure 4.1 has two components: $\{n_1, n_2, n_3, n_4, n_5, n_6\}$ and $\{n_7\}$.

4.1.6 Condensation Graphs

We are finally in a position to formalize an important simplification mechanism for testers.

Definition

Given a graph $G = (V, E)$, its *condensation graph* is formed by replacing each component by a condensing node.

Developing the condensation graph of a given graph is an unambiguous (i.e., algorithmic) process. We use the adjacency matrix to identify path connectivity, and then use the equivalence relation to identify components. The absolute nature of this process is important: the condensation graph of a given graph is unique. This implies that the resulting simplification represents an important aspect of the original graph.

The components in our continuing example are $S_1 = \{n_1, n_2, n_3, n_4, n_5, n_6\}$ and $S_2 = \{n_7\}$.

No edges can be present in a condensation graph of an ordinary (undirected) graph. Two reasons are

1. Edges have individual nodes as endpoints, not sets of nodes. (Here, we can finally use the distinction between n_7 and $\{n_7\}$.)
2. Even if we fudge the definition of edge to ignore this distinction, a possible edge would mean that nodes from two different components were connected, thus in a path, thus in the same (maximal!) component.

The implication for testing is that components are independent in an important way; thus, they can be tested separately.

4.1.7 Cyclomatic Number

The cyclomatic complexity property of graphs has deep implications for testing.

Definition

The *cyclomatic number* of a graph G is given by $V(G) = e - n + p$, where

e is the number of edges in G.
n is the number of nodes in G.
p is the number of components in G.

$V(G)$ is the number of distinct regions in a strongly connected directed graph. In Chapter 8, we will examine a formulation of code-based testing that considers all the paths in a program graph to be a vector space. There are $V(G)$ elements in the set of basis vectors for this space. The cyclomatic number of our example graph is $V(G) = 5 - 7 + 2 = 0$. This is not a very good example for cyclomatic complexity. When we use cyclomatic complexity in Chapter 8, and expand on it in Chapter 16, we will (usually) have strongly connected graphs, which will have a larger cyclomatic complexity than this small example.

4.2 Directed Graphs

Directed graphs are a slight refinement to ordinary graphs: edges acquire a sense of direction. Symbolically, the unordered pairs (n_i, n_j) become ordered pairs $<n_i, n_j>$, and we speak of a directed edge going from node n_i to n_j, instead of being between the nodes.

Definition

A *directed graph* (or digraph) $D = (V, E)$ consists of a finite set $V = \{n_1, n_2, \ldots, n_m\}$ of nodes, and a set $E = \{e_1, e_2, \ldots, e_p\}$ of edges, where each edge $e_k = <n_i, n_j>$ is an ordered pair of nodes $n_i, n_j \in V$.

In the directed edge $e_k = <n_i, n_j>$, n_i is the initial (or start) node and n_j is the terminal (or finish) node. Edges in directed graphs fit naturally with many software concepts: sequential behavior, imperative programming languages, time-ordered events, define/reference pairings, messages, function and procedure calls, and so on. Given this, you might ask why we spent (wasted?) so much time on ordinary graphs. The difference between ordinary and directed graphs is very analogous to the difference between declarative and imperative programming languages. In imperative languages (e.g., COBOL, FORTRAN, Pascal, C, Java, Ada®), the sequential order of source language statements determines the execution time order of compiled code. This is not true for declarative languages (such as Prolog). The most common declarative situation for most software developers is entity/relationship (E/R) modeling. In an E/R model, we choose entities as nodes and identify relationships as edges. (If a relationship involves three or more entities, we need the notion of a "hyper-edge" that has three or more endpoints.) The resulting graph of an E/R model is more properly interpreted as an ordinary graph. Good E/R modeling practice suppresses the sequential thinking that directed graphs promote.

When testing a program written in a declarative language, the only concepts available to the tester are those that follow from ordinary graphs. Fortunately, most software is developed in imperative languages; so testers usually have the full power of directed graphs at their disposal.

The next series of definitions roughly parallels the ones for ordinary graphs. We modify our now familiar continuing example to the one shown in Figure 4.2.

We have the same node set $V = \{n_1, n_2, n_3, n_4, n_5, n_6, n_7\}$, and the edge set appears to be the same: $E = \{e_1, e_2, e_3, e_4, e_5\}$. The difference is that the edges are now ordered pairs of nodes in V:

$$E = \{<n_1, n_2>, <n_1, n_4>, <n_3, n_4>, <n_2, n_5>, <n_4, n_6>\}$$

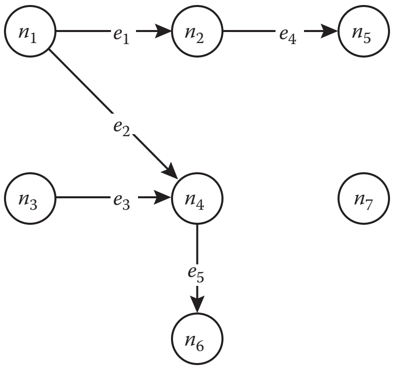


Figure 4.2 Directed graph.

4.2.1 Indegrees and Outdegrees

The degree of a node in an ordinary graph is refined to reflect direction, as follows:

Definition

The *indegree of a node* in a directed graph is the number of distinct edges that have the node as a terminal node. We write indeg(n) for the indegree of node n.

The *outdegree of a node* in a directed graph is the number of distinct edges that have the node as a start point. We write outdeg(n) for the outdegree of node n.

The nodes in the digraph in Figure 4.2 have the following indegrees and outdegrees:

$$\begin{array}{ll} \text{indeg}(n_1) = 0 & \text{outdeg}(n_1) = 2 \\ \text{indeg}(n_2) = 1 & \text{outdeg}(n_2) = 1 \\ \text{indeg}(n_3) = 0 & \text{outdeg}(n_3) = 1 \\ \text{indeg}(n_4) = 2 & \text{outdeg}(n_4) = 1 \\ \text{indeg}(n_5) = 1 & \text{outdeg}(n_5) = 0 \\ \text{indeg}(n_6) = 1 & \text{outdeg}(n_6) = 0 \\ \text{indeg}(n_7) = 0 & \text{outdeg}(n_7) = 0 \end{array}$$

Ordinary and directed graphs meet through definitions that relate obvious correspondences, such as deg(n) = indeg(n) + outdeg(n).

4.2.2 Types of Nodes

The added descriptive power of directed graphs lets us define different kinds of nodes:

Definition

A node with indegree = 0 is a *source node*.
A node with outdegree = 0 is a *sink node*.
A node with indegree ≠ 0 and outdegree ≠ 0 is a *transfer node*.

Source and sink nodes constitute the external boundary of a graph. If we made a directed graph of a context diagram (from a set of data flow diagrams produced by structured analysis), the external entities would be source and sink nodes.

In our continuing example, n_1, n_3, and n_7 are source nodes; n_5, n_6, and n_7 are sink nodes; and n_2 and n_4 are transfer (also known as interior) nodes. A node that is both a source and a sink node is an isolated node.

4.2.3 Adjacency Matrix of a Directed Graph

As we might expect, the addition of direction to edges changes the definition of the adjacency matrix of a directed graph. (It also changes the incidence matrix, but this matrix is seldom used in conjunction with digraphs.)

Definition

The *adjacency matrix of a directed graph* $D = (V, E)$ with m nodes is an $m \times m$ matrix: $A = (a(i, j))$ where $a(i, j)$ is a 1 if and only if there is an edge from node i to node j; otherwise, the element is 0.

The adjacency matrix of a directed graph is not necessarily symmetric. A row sum is the outdegree of the node; a column sum is the indegree of a node. The adjacency matrix of our continuing example is as follows:

	n_1	n_2	n_3	n_4	n_5	n_6	n_7
n_1	0	1	0	1	0	0	0
n_2	0	0	0	0	1	0	0
n_3	0	0	0	1	0	0	0
n_4	0	0	0	0	0	1	0
n_5	0	0	0	0	0	0	0
n_6	0	0	0	0	0	0	0
n_7	0	0	0	0	0	0	0

One common use of directed graphs is to record family relationships, in which siblings, cousins, and so on are connected by an ancestor; and parents, grandparents, and so on are connected by a descendant. Entries in powers of the adjacency matrix now show existence of directed paths.

4.2.4 Paths and Semipaths

Direction permits a more precise meaning to paths that connect nodes in a directed graph. As a handy analogy, you may think in terms of one-way and two-way streets.

Definition

A *(directed) path* is a sequence of edges such that, for any adjacent pair of edges e_i, e_j in the sequence, the terminal node of the first edge is the initial node of the second edge.

A *cycle* is a directed path that begins and ends at the same node.

A *chain* is a sequence of nodes such that each interior node has indegree = 1 and outdegree = 1. The initial node may have indegree = 0 or indegree > 1. The terminal node may have outdegree = 0 or outdegree > 1 (we will use this concept in Chapter 8).

A *(directed) semipath* is a sequence of edges such that for at least one adjacent pair of edges e_i, e_j in the sequence, the initial node of the first edge is the initial node of the second edge or the terminal node of the first edge is the terminal node of the second edge.

Our continuing example contains the following paths and semipaths (not all are listed):

A path from n_1 to n_6
A semipath between n_1 and n_3
A semipath between n_2 and n_4
A semipath between n_5 and n_6

4.2.5 Reachability Matrix

When we model an application with a digraph, we often ask questions that deal with paths that let us reach (or "get to") certain nodes. This is an extremely useful capability and is made possible by the reachability matrix of a digraph.

Definition

The *reachability matrix* of a directed graph $D = (V, E)$ with m nodes is an $m \times m$ matrix $R = (r(i, j))$, where $r(i, j)$ is a 1 if and only if there is a path from node i to node j; otherwise, the element is 0.

The reachability matrix of a directed graph D can be calculated from the adjacency matrix A as

$$R = I + A + A^2 + A^3 + \ldots + A^k$$

where k is the length of the longest path in D, and I is the identity matrix. The reachability matrix for our continuing example is as follows:

	n_1	n_2	n_3	n_4	n_5	n_6	n_7
n_1	1	1	0	1	1	1	0
n_2	0	1	0	0	1	0	0
n_3	0	0	1	1	0	1	0
n_4	0	0	0	1	0	1	0
n_5	0	0	0	0	1	0	0
n_6	0	0	0	0	0	1	0
n_7	0	0	0	0	0	0	1

The reachability matrix tells us that nodes n_2, n_4, n_5, and n_6 can be reached from n_1; node n_5 can be reached from n_2; and so on.

4.2.6 n-*Connectedness*

Connectedness of ordinary graphs extends to a rich, highly explanatory concept for digraphs.

Definition

Two nodes n_i and n_j in a directed graph are

0-connected iff no path exists between n_i and n_j
1-connected iff a semipath but no path exists between n_i and n_j
2-connected iff a path exists between n_i and n_j
3-connected iff a path goes from n_i to n_j and a path goes from n_j to n_i

No other degrees of connectedness exist.

We need to modify our continuing example to show 3-connectedness. The change is the addition of a new edge e_6 from n_6 to n_3, so the graph contains a cycle.

With this change, we have the following instances of *n*-connectivity in Figure 4.3 (not all are listed):

n_1 and n_7 are 0-connected
n_2 and n_6 are 1-connected
n_1 and n_6 are 2-connected
n_3 and n_6 are 3-connected

In terms of one-way streets, you cannot get from n_2 to n_6.

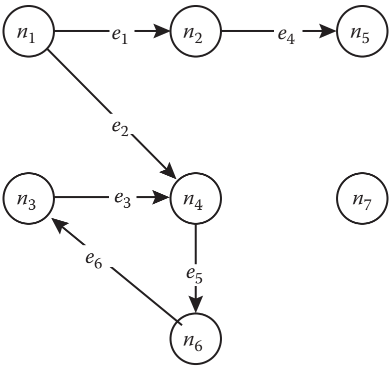


Figure 4.3 Directed graph with a cycle.

4.2.7 Strong Components

The analogy continues. We get two equivalence relations from *n*-connectedness: 1-connectedness yields what we might call "weak connection," and this in turn yields weak components. (These turn out to be the same as we had for ordinary graphs, which is what should happen, because 1-connectedness effectively ignores direction.) The second equivalence relation, based on 3-connectedness, is more interesting. As before, the equivalence relation induces a partition on the node set of a digraph; however, the condensation graph is quite different. Nodes that previously were 0-, 1-, or 2-connected remain so. The 3-connected nodes become the strong components.

Definition

A *strong component of a directed graph* is a maximal set of 3-connected nodes.

In our amended example, the strong components are the sets $\{n_3, n_4, n_6\}$ and $\{n_7\}$. The condensation graph for our amended example is shown in Figure 4.4.

Strong components let us simplify by removing loops and isolated nodes. Although this is not as dramatic as the simplification we had in ordinary graphs, it does solve a major testing problem. Notice that the condensation graph of a digraph will never contain a loop. (If it did, the loop would have been condensed by the maximal aspect of the partition.) These graphs have a special name: directed acyclic graphs, sometimes written as DAG.

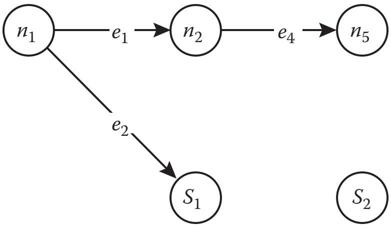


Figure 4.4 Condensation graph of digraph in Figure 4.3.

Many papers on structured testing make quite a point of showing how relatively simple programs can have millions of distinct execution paths. The intent of these discussions is to convince us that exhaustive testing is exactly that—exhaustive. The large number of execution paths comes from nested loops. Condensation graphs eliminate loops (or at least condense them down to a single node); therefore, we can use this as a strategy to simplify situations that otherwise are computationally untenable.

4.3 Graphs for Testing

We conclude this chapter with four special graphs that are widely used for testing. The first of these, the program graph, is used primarily at the unit testing level. The other three, finite state machines, state charts, and Petri nets, are best used to describe system-level behavior, although they can be used at lower levels of testing.

4.3.1 Program Graphs

At the beginning of this chapter, we made a point of avoiding interpretations on the graph theory definitions to preserve latitude in later applications. Here, we give the most common use of graph theory in software testing—the program graph. To better connect with existing testing literature, the traditional definition is given, followed by an improved definition.

Definition

Given a program written in an imperative programming language, its *program graph* is a directed graph in which

1. (Traditional definition)
   Nodes are program statements, and edges represent flow of control (there is an edge from node *i* to node *j* iff the statement corresponding to node *j* can be executed immediately after the statement corresponding to node *i*).
2. (Improved definition)
   Nodes are either entire statements or fragments of a statement, and edges represent flow of control (there is an edge from node *i* to node *j* iff the statement or statement fragment corresponding to node *j* can be executed immediately after the statement or statement fragment corresponding to node *i*).

It is cumbersome to always say "statement or statement fragment," so we adopt the convention that a statement fragment can be an entire statement. The directed graph formulation of a program enables a very precise description of testing aspects of the program. For one thing, a very satisfying connection exists between this formulation and the precepts of structured programming. The basic structured programming constructs (sequence, selection, and repetition) all have the directed graphs as shown in Figure 4.5.

When these constructs are used in a structured program, the corresponding graphs are either nested or concatenated. The single entrance and single exit criteria result in unique source and sink nodes in the program graph. In fact, the old (nonstructured) "spaghetti code" resulted in very

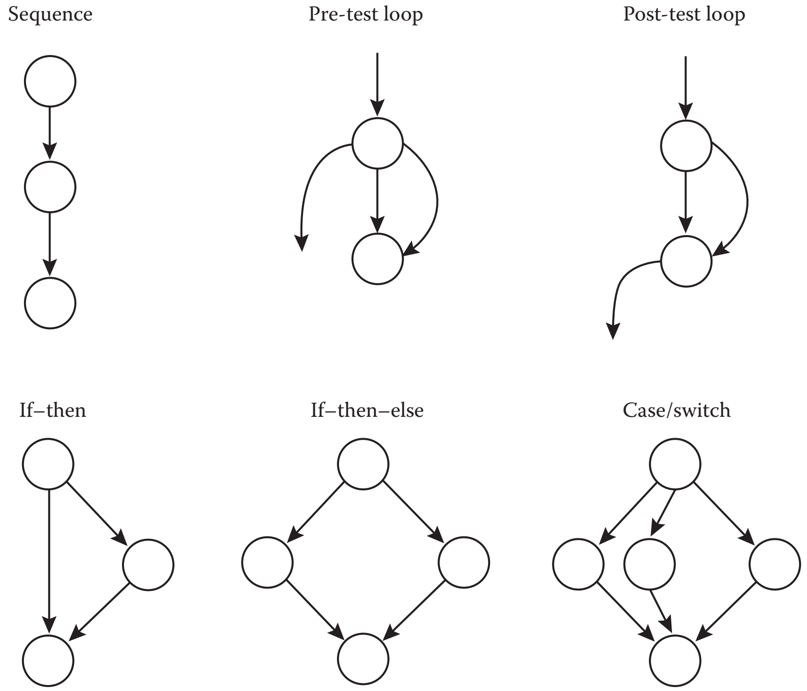


Figure 4.5 Digraphs of structured programming constructs.

complex program graphs. GOTO statements, for example, introduce edges; and when these are used to branch into or out of loops, the resulting program graphs become even more complex. One of the pioneering analysts of this is Thomas McCabe, who popularized the cyclomatic number of a graph as an indicator of program complexity (McCabe, 1976). When a program executes, the statements that execute comprise a path in the program graph. Loops and decisions greatly increase the number of possible paths and therefore similarly increase the need for testing.

One of the problems with program graphs is how to treat nonexecutable statements such as comments and data declaration statements. The simplest answer is to ignore them. A second problem has to do with the difference between topologically possible and semantically feasible paths. We will discuss this in more detail in Chapter 8.

4.3.2 Finite State Machines

Finite state machines have become a fairly standard notation for requirements specification. All the real-time extensions of structured analysis use some form of finite state machine, and nearly all forms of object-oriented analyses require them. A finite state machine is a directed graph in which states are nodes and transitions are edges. Source and sink states become initial and terminal nodes, sequences of transitions are modeled as paths, and so on. Most finite state machine notations add information to the edges (transitions) to indicate the cause of the transition and actions that occur as a result of the transition.

Figure 4.6 is a finite state machine for the garage door controller described in Chapter 2. (We will revisit this finite state machine in Chapters 14 and 17.) The labels on the transitions

Input events
e_1: depress controller button
e_2: end of down track hit
e_3: end of up track hit
e_4: obstacle hit
e_5: laser beam crossed

Output events (actions)
a_1: start drive motor down
a_2: start drive motor up
a_3: stop drive motor
a_4: door stops part way
a_5: door continues opening
a_6: door continues closing

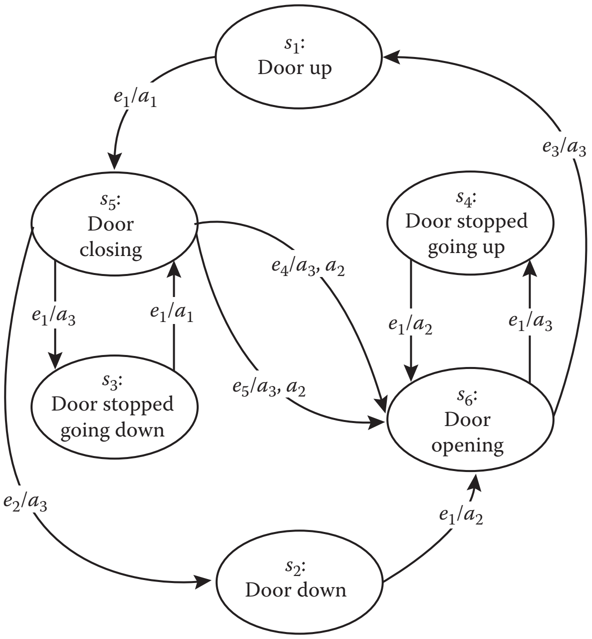


Figure 4.6 Finite state machine for garage door controller.

follow a convention that the “numerator” is the event that causes the transition, and the “denominator” is the action that is associated with the transition. The events are mandatory—transitions do not just happen, but the actions are optional. Finite state machines are simple ways to represent situations in which a variety of events may occur, and their occurrences have different consequences.

Finite state machines can be executed; however, a few conventions are needed first. One is the notion of the active state. We speak of a system being “in” a certain state; when the system is modeled as a finite state machine, the active state refers to the state “we are in.” Another convention is that finite state machines may have an initial state, which is the state that is active when a finite state machine is first entered. (Initial and final states are recognized by the absence of incoming and outgoing transitions, respectively.) Exactly one state can be active at any time. We also think of transitions as instantaneous occurrences, and the events that cause transitions also occur one at a time. To execute a finite state machine, we start with an initial state and provide a sequence of events that causes state transitions. As each event occurs, the transition changes the active state and a new event occurs. In this way, a sequence of events selects a path of states (or equivalently, of a sequence of transitions) through the machine.

4.3.3 Petri Nets

Petri nets were the topic of Carl Adam Petri's PhD dissertation in 1963; today, they are the accepted model for protocols and other applications involving concurrency and distributed processing. Petri nets are a special form of directed graph: a bipartite directed graph. (A bipartite graph has two sets of nodes, V_1 and V_2, and a set of edges E, with the restriction that every edge has its initial node in one of the sets V_1, V_2, and its terminal node in the other set.) In a Petri net, one of the sets is referred to as "places," and the other is referred to as "transitions." These sets are usually denoted as P and T, respectively. Places are inputs to and outputs of transitions; the input and output relationships are functions, and they are usually denoted as In and Out, as in the following definition.

Definition

A *Petri net* is a bipartite directed graph (P, T, In, Out), in which P and T are disjoint sets of nodes, and In and Out are sets of edges, where In $\subseteq P \times T$, and Out $\subseteq T \times P$.

For the sample Petri net in Figure 4.7, the sets P, T, In, and Out are

$$P = \{p_1, p_2, p_3, p_4, p_5\}$$

$$T = \{t_1, t_2, t_3\}$$

$$\text{In} = \{<p_1, t_1>, <p_5, t_1>, <p_5, t_3>, <p_2, t_3>, <p_3, t_2>\}$$

$$\text{Out} = \{<t_1, p_3>, <t_2, p_4>, <t_3, p_4>\}$$

Petri nets are executable in more interesting ways than finite state machines. The next few definitions lead us to Petri net execution.

Definition

A *marked Petri net* is a 5-tuple (P, T, In, Out, M) in which (P, T, In, Out) is a Petri net and M is a set of mappings of places to positive integers.

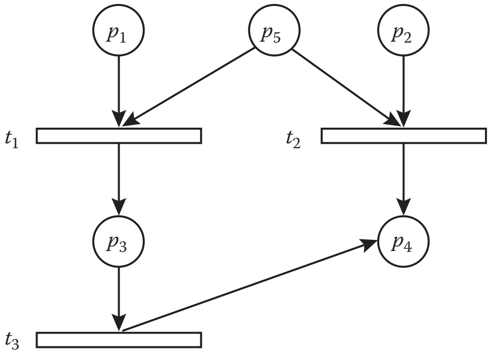


Figure 4.7 **Petri net.**

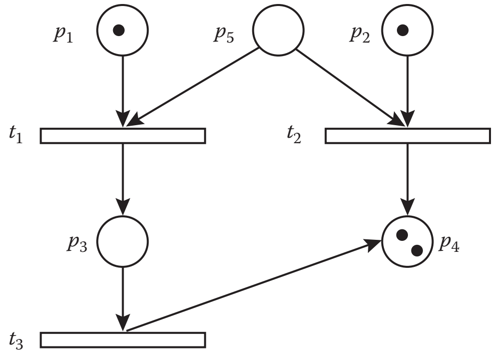


Figure 4.8 Marked Petri net.

The set M is called the marking set of the Petri net. Elements of M are n-tuples, where n is the number of places in the set P. For the Petri net in Figure 4.7, the set M contains elements of the form $<n_1, n_2, n_3, n_4, n_5>$, where the n's are the integers associated with the respective places. The number associated with a place refers to the number of tokens that are said to be "in" the place. Tokens are abstractions that can be interpreted in modeling situations. For example, tokens might refer to the number of times a place has been used, or the number of things in a place, or whether the place is true. Figure 4.8 shows a marked Petri net.

Definition

A transition in a Petri net is *enabled* if at least one token is in each of its input places.

The marking tuple for the marked Petri net in Figure 4.8 is <1, 1, 0, 2, 0>. We need the concept of tokens to make two essential definitions. No enabled transitions are in the marked Petri net in Figure 4.8. If we put a token in place p_3, then transition t_2 would be enabled.

Definition

When an enabled Petri net *transition fires*, one token is removed from each of its input places and one token is added to each of its output places.

In Figure 4.9, transition t_2 is enabled in the left net and has been fired in the right net. The marking sequence for the net in Figure 4.9 contains two tuples—the first shows the net when t_2 is enabled, and the second shows the net after t_2 has fired:

$$M = \{<1, 1, 0, 2, 1>, <1, 0, 0, 3, 0>\}$$

Tokens may be created or destroyed by transition firings. Under special conditions, the total number of tokens in a net never changes; such nets are called conservative. We usually do not worry about token conservation. Markings let us execute Petri nets in much the same way that we execute finite state machines. (It turns out that finite state machines are a special case of Petri nets.)

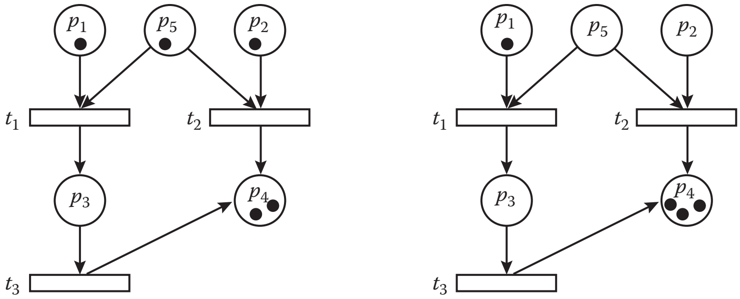


Figure 4.9 Before and after firing t_2.

Look again at the net in Figure 4.9; in the left net (before firing any transition), places p_1, p_2, and p_5 are all marked. With such a marking, transitions t_1 and t_2 are both enabled. We chose to fire transition t_2, the token in place p_5 is removed and t_1 is no longer enabled. Similarly, if we choose to fire t_1, we disable t_2. This pattern is known as Petri net conflict. More specifically, we say that transitions t_1 and t_2 are in conflict with respect to place p_5. Petri net conflict exhibits an interesting form of interaction between two transitions; we will revisit this (and other interactions) in Chapter 17.

4.3.4 Event-Driven Petri Nets

Basic Petri nets need two slight enhancements to become Event-Driven Petri Nets (EDPNs). The first enables them to express more closely event-driven systems, and the second deals with Petri net markings that express event quiescence, an important notion in object-oriented applications. Taken together, these extensions result in an effective, operational view of software requirements.

Definition

An *Event-Driven Petri Net* is a tripartite-directed graph (*P*, *D*, *S*, In, Out) composed of three sets of nodes, *P*, *D*, and *S*, and two mappings, In and Out, where

P is a set of port events
D is a set of data places
S is a set of transitions

In is a set of ordered pairs from $(P \cup D) \times S$
Out is a set of ordered pairs from $S \times (P \cup D)$

EDPNs express four of the five basic system constructs defined in Chapter 14; only devices are missing. The set *S* of transitions corresponds to ordinary Petri net transitions, which are interpreted as actions.

Two kinds of places, port events and data places, are inputs to or outputs of transitions in *S* as defined by the input and output functions In and Out. A thread is a sequence of transitions in *S*, so we can always construct the inputs and outputs of a thread from the inputs and outputs of the transitions in the thread. EDPNs are graphically represented in much the same way as ordinary Petri nets; the only difference is the use of triangles for port event places. The EDPN in Figure 4.10 has four transitions, s_7, s_8, s_9, and s_{10}; two port input events, p_3 and p_4; and three data places, d_5, d_6, and d_7. It does not have port output events.

This is the EDPN that corresponds to the finite state machine developed for the dial portion of the Saturn windshield wiper system in Chapter 15 (see Figure 15.1). The components of this net are described in Table 4.1.

Markings for an EDPN are more complicated because we want to be able to deal with event quiescence.

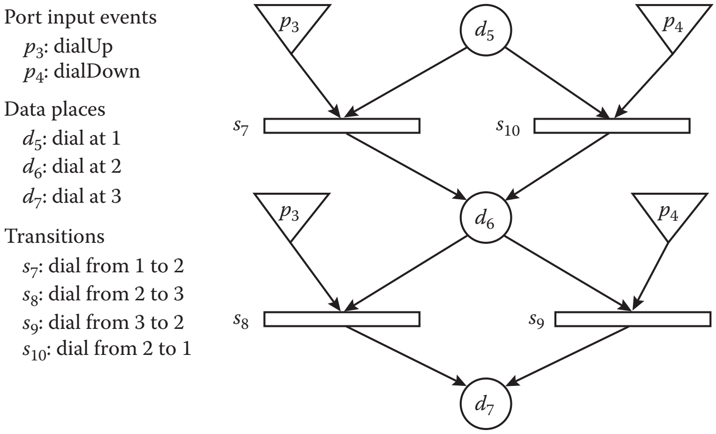

Port input events
p_3: dialUp
p_4: dialDown
Data places
d_5: dial at 1
d_6: dial at 2
d_7: dial at 3
Transitions
s_7: dial from 1 to 2
s_8: dial from 2 to 3
s_9: dial from 3 to 2
s_{10}: dial from 2 to 1
p_3
d_5
p_4
s_7
s_{10}
p_3
d_6
p_4
s_8
s_9
d_7

Figure 4.10 Event-driven Petri net.

Table 4.1 EDPN Elements in Figure 4.10

Element	*Type*	*Description*
p_3	Port input event	Rotate dial clockwise
p_4	Port input event	Rotate dial counterclockwise
d_5	Data place	Dial at position 1
d_6	Data place	Dial at position 2
d_7	Data place	Dial at position 3
s_7	Transition	State transition: d_5 to d_6
s_8	Transition	State transition: d_6 to d_7
s_9	Transition	State transition: d_7 to d_6
s_{10}	Transition	State transition: d_6 to d_5

Definition

A *marking* M of an EDPN (P, D, S, In, Out) is a sequence $M = <m_1, m_2, \ldots>$ of p-tuples, where $p = k + n$, and k and n are the number of elements in the sets P and D, and individual entries in a p-tuple indicate the number of tokens in the event or data place.

By convention, we will put the data places first, followed by the input event places, and then the output event places. An EDPN may have any number of markings; each corresponds to an execution of the net. Table 4.2 shows a sample marking of the EDPN in Figure 4.10.

The rules for transition enabling and firing in an EDPN are exact analogs of those for traditional Petri nets; a transition is enabled if there is at least one token in each input place; and when an enabled transition fires, one token is removed from each of its input places, and one token is placed in each of its output places. Table 4.3 follows the marking sequence given in Table 4.2, showing which transitions are enabled and fired.

The important difference between EDPNs and traditional Petri nets is that event quiescence can be broken by creating a token in a port input event place. In traditional Petri nets, when no transition is enabled, we say that the net is deadlocked. In EDPNs, when no transition is enabled, the net is at a point of event quiescence. (Of course, if no event occurs, this is the same as deadlock.) Event quiescence occurs four times in the thread in Table 4.3: at m_1, m_3, m_5, and m_7.

Table 4.2 Marking of EDPN in Figure 4.10

Tuple	*(p_3, p_4, d_5, d_6, d_7)*	*Description*
m_1	(0, 0, 1, 0, 0)	Initial condition, in state d_5
m_2	(1, 0, 1, 0, 0)	p_3 occurs
m_3	(0, 0, 0, 1, 0)	In state d_6
m_4	(1, 0, 0, 1, 0)	p_3 occurs
m_5	(0, 0, 0, 0, 1)	In state d_7
m_6	(0, 1, 0, 0, 1)	p_4 occurs
m_7	(0, 0, 0, 1, 0)	In state d_6

Table 4.3 Enabled and Fired Transitions in Table 4.2

Tuple	*(p_3, p_4, d_5, d_6, d_7)*	*Description*
m_1	(0, 0, 1, 0, 0)	Nothing enabled
m_2	(1, 0, 1, 0, 0)	s_7 enabled; s_7 fired
m_3	(0, 0, 0, 1, 0)	Nothing enabled
m_4	(1, 0, 0, 1, 0)	s_8 enabled; s_8 fired
m_5	(0, 0, 0, 0, 1)	Nothing enabled
m_6	(0, 1, 0, 0, 1)	s_9 enabled; s_9 fired
m_7	(0, 0, 0, 1, 0)	Nothing enabled

The individual members in a marking can be thought of as snapshots of the executing EDPN at discrete points in time; these members are alternatively referred to as time steps, *p*-tuples, or marking vectors. This lets us think of time as an ordering that allows us to recognize "before" and "after." If we attach instantaneous time as an attribute of port events, data places, and transitions, we obtain a much clearer picture of thread behavior. One awkward part to this is how to treat tokens in a port output event place. Port output places always have outdegree = 0; in an ordinary Petri net, tokens cannot be removed from a place with a zero outdegree. If the tokens in a port output event place persist, this suggests that the event occurs indefinitely. Here again, the time attributes resolve the confusion; this time we need a duration of the marked output event. (Another possibility is to remove tokens from a marked output event place after one time step; this works reasonably well.)

4.3.5 StateCharts

David Harel had two goals when he developed the StateChart notation: he wanted to devise a visual notation that combined the ability of Venn diagrams to express hierarchy and the ability of directed graphs to express connectedness (Harel, 1988). Taken together, these capabilities provide an elegant answer to the "state explosion" problem of ordinary finite state machines. The result is a highly sophisticated and very precise notation that is supported by commercially available CASE tools, notably the StateMate system. StateCharts are now the control model of choice for the Unified Modeling Language (UML) from IBM. (See http://www-306.ibm.com/software/rational/uml/ for more details.)

Harel uses the methodology neutral term "blob" to describe the basic building block of a StateChart. Blobs can contain other blobs in the same way that Venn diagrams show set containment. Blobs can also be connected to other blobs with edges in the same way that nodes in a directed graph are connected. In Figure 4.11, blob A contains two blobs (B and C), and they are connected by edges. Blob A is also connected to blob D by an edge.

As Harel intends, we can interpret blobs as states and edges as transitions. The full StateChart system supports an elaborate language that defines how and when transitions occur (their training course runs for a full week, so this section is a highly simplified introduction). StateCharts are executable in a much more elaborate way than ordinary finite state machines. Executing a StateChart requires a notion similar to that of Petri net markings. The "initial state" of a StateChart is indicated by an edge that has no source state.

When states are nested within other states, the same indication is used to show the lower-level initial state. In Figure 4.12, state A is the initial state; and when it is entered, state B is also entered at the lower level. When a state is entered, we can think of it as active in a way analogous to a marked place in a Petri net. (The StateChart tool uses colors to show which states are active, and this is equivalent to marking places in a Petri net.) A subtlety exists in Figure 4.12, the transition

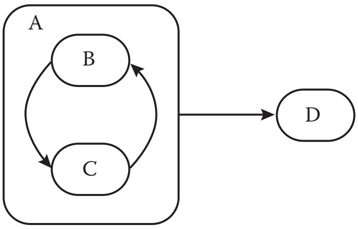


Figure 4.11 Blobs in a StateChart.

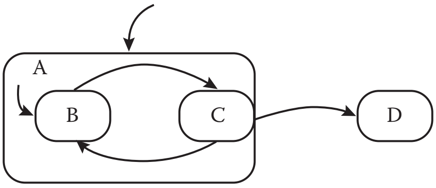


Figure 4.12 Initial states in a StateChart.

from state A to state D seems ambiguous at first because it has no apparent recognition of states B and C. The convention is that edges must start and end on the outline (Harel uses the term "contour") of a state. If a state contains substates, as state A does, the edge "refers" to all substates. Thus, the edge from A to D means that the transition can occur either from state B or from state C. If we had an edge from state D to state A, as in Figure 4.13, the fact that state B is indicated as the initial state means that the transition is really from state D to state B. This convention greatly reduces the tendency of finite state machines to look like "spaghetti code."

The last aspect of StateCharts we will discuss is the notion of concurrent StateCharts. The dotted line in state D (see Figure 4.14) is used to show that state D really refers to two concurrent states, E and F. (Harel's convention is to move the state label of D to a rectangular tag on the perimeter of the state.) Although not shown here, we can think of E and F as separate devices that execute concurrently. Because the edge from state A terminates on the perimeter of state D, when that transition occurs, both devices E and F are active (or marked, in the Petri net sense).

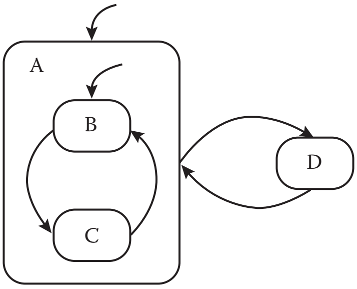


Figure 4.13 Default entry into substates.

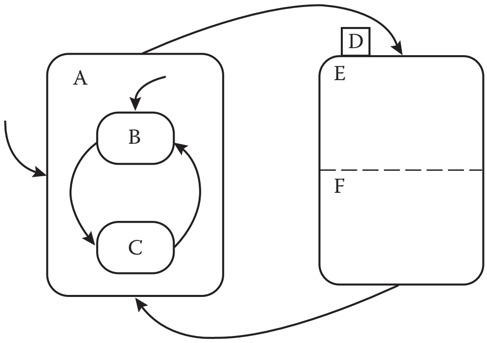


Figure 4.14 Concurrent states.

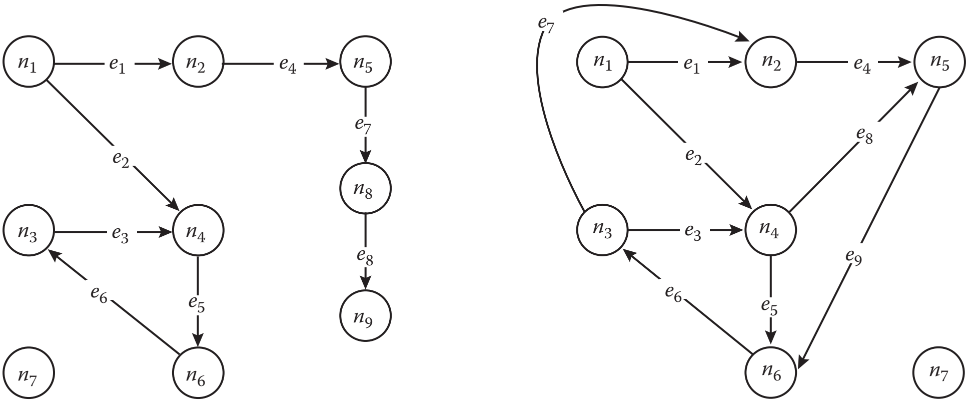


Figure 4.15 Directed graphs for exercise 5.

EXERCISES

1. Propose a definition for the length of a path in a graph.
2. What loop(s) is/are created if an edge is added between nodes n_5 and n_6 in the graph in Figure 4.1?
3. Convince yourself that 3-connectedness is an equivalence relation on the nodes of a digraph.
4. Compute the cyclomatic complexity for each of the structured programming constructs in Figure 4.5.
5. The digraphs in Figure 4.15 were obtained by adding nodes and edges to the digraph in Figure 4.3. Compute the cyclomatic complexity of each new digraph, and explain how the changes affected the complexity.
6. Suppose we make a graph in which nodes are people and edges correspond to some form of social interaction, such as "talks to" or "socializes with." Find graph theory concepts that correspond to social concepts such as popularity, cliques, and hermits.

References

Harel, D., On visual formalisms, *Communications of the ACM,* Vol. 31, No. 5, May 1988, pp. 514–530.

McCabe, T. J., A complexity measure, *IEEE Transactions on Software Engineering*, Vol. 2, No. 4, December 1976, pp. 308–320.

UNIT TESTING

II

The term “unit” needs explanation. There are several interpretations about exactly what constitutes a unit. In a procedural programming language, a unit can be

- A single procedure
- A function
- A body of code that implements a single function
- Source code that fits on one page
- A body of code that represents work done in 4 to 40 hours (as in a work breakdown structure)
- The smallest body of code that can be compiled and executed by itself

In an object-oriented programming language, there is general agreement that a class is a unit. However, methods of a class might be limited by any of the “definitions” of a unit for procedural code.

The bottom line is that “unit” is probably best defined by organizations implementing code. In my telephony career, the “standard unit” for planning purposes was 300 lines of source code. The main reason for this was that telephone switching system software is very large, so bigger units were appropriate. My personal definition of a unit is a body of software that is designed, coded, and tested by either one person or possibly a programmer pair. Chapters 5 through 10 cover unit-level testing, and since methods in object-oriented programming are so similar to procedural units, the material applies to both forms of programming language.

Chapter 5

Boundary Value Testing

In Chapter 3, we saw that a function maps values from one set (its domain) to values in another set (its range) and that the domain and range can be cross products of other sets. Any program can be considered to be a function in the sense that program inputs form its domain and program outputs form its range. In this and the next two chapters, we examine how to use knowledge of the functional nature of a program to identify test cases for the program. Input domain testing (also called "boundary value testing") is the best-known specification-based testing technique. Historically, this form of testing has focused on the input domain; however, it is often a good supplement to apply many of these techniques to develop range-based test cases.

There are two independent considerations that apply to input domain testing. The first asks whether or not we are concerned with invalid values of variables. Normal boundary value testing is concerned only with valid values of the input variables. Robust boundary value testing considers invalid and valid variable values. The second consideration is whether we make the "single fault" assumption common to reliability theory. This assumes that faults are due to incorrect values of a single variable. If this is not warranted, meaning that we are concerned with interaction among two or more variables, we need to take the cross product of the individual variables. Taken together, the two considerations yield four variations of boundary value testing:

- Normal boundary value testing
- Robust boundary value testing
- Worst-case boundary value testing
- Robust worst-case boundary value testing

For the sake of comprehensible drawings, the discussion in this chapter refers to a function, F, of two variables x_1 and x_2. When the function F is implemented as a program, the input variables x_1 and x_2 will have some (possibly unstated) boundaries:

$$a \le x_1 \le b$$

$$c \le x_2 \le d$$

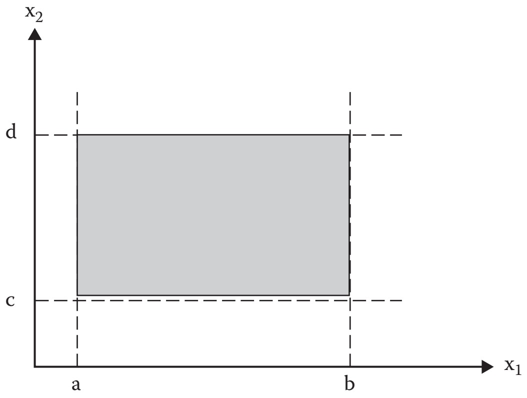


Figure 5.1 Input domain of a function of two variables.

Unfortunately, the intervals [a, b] and [c, d] are referred to as the ranges of x_1 and x_2, so right away we have an overloaded term. The intended meaning will always be clear from its context. Strongly typed languages (such as Ada® and Pascal) permit explicit definition of such variable ranges. In fact, part of the historical reason for strong typing was to prevent programmers from making the kinds of errors that result in faults that are easily revealed by boundary value testing. Other languages (such as COBOL, FORTRAN, and C) are not strongly typed, so boundary value testing is more appropriate for programs coded in these languages. The input space (domain) of our function F is shown in Figure 5.1. Any point within the shaded rectangle and including the boundaries is a legitimate input to the function F.

5.1 Normal Boundary Value Testing

All four forms of boundary value testing focus on the boundary of the input space to identify test cases. The rationale behind boundary value testing is that errors tend to occur near the extreme values of an input variable. Loop conditions, for example, may test for < when they should test for ≤, and counters often are "off by one." (Does counting begin at zero or at one?) The basic idea of boundary value analysis is to use input variable values at their minimum, just above the minimum, a nominal value, just below their maximum, and at their maximum. A commercially available testing tool (originally named T) generates such test cases for a properly specified program. This tool has been successfully integrated with two popular front-end CASE tools (Teamwork from Cadre Systems, and Software through Pictures from Aonix [part of Atego]; for more information, see http://www.aonix.com/pdf/2140-AON.pdf). The T tool refers to these values as min, min+, nom, max–, and max. The robust forms add two values, min– and max+.

The next part of boundary value analysis is based on a critical assumption; it is known as the "single fault" assumption in reliability theory. This says that failures are only rarely the result of the simultaneous occurrence of two (or more) faults. The All Pairs testing approach (described in Chapter 20) contradicts this, with the observation that, in software-controlled medical systems, almost all faults are the result of interaction between a pair of variables. Thus, the normal and robust variations cases are obtained by holding the values of all but one variable at their nominal

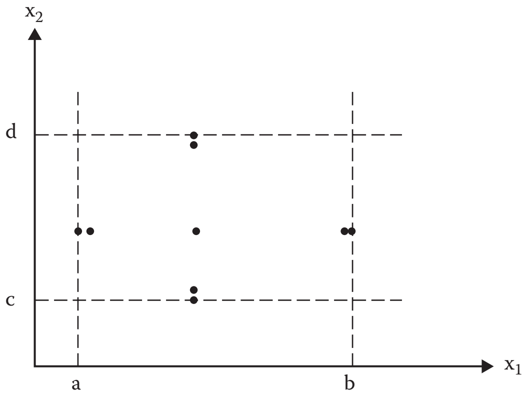


Figure 5.2 Boundary value analysis test cases for a function of two variables.

values, and letting that variable assume its full set of test values. The normal boundary value analysis test cases for our function F of two variables (illustrated in Figure 5.2) are

{<x_{1nom}, x_{2min}>, <x_{1nom}, x_{2min+}>, <x_{1nom}, x_{2nom}>, <x_{1nom}, x_{2max-}>, <x_{1nom}, x_{2max}>, <x_{1min}, x_{2nom}>, <x_{1min+}, x_{2nom}>, <x_{1max-}, x_{2nom}>, <x_{1max}, x_{2nom}>}

5.1.1 Generalizing Boundary Value Analysis

The basic boundary value analysis technique can be generalized in two ways: by the number of variables and by the kinds of ranges. Generalizing the number of variables is easy: if we have a function of n variables, we hold all but one at the nominal values and let the remaining variable assume the min, min+, nom, max–, and max values, repeating this for each variable. Thus, for a function of n variables, boundary value analysis yields 4n + 1 unique test cases.

Generalizing ranges depends on the nature (or more precisely, the type) of the variables themselves. In the NextDate function, for example, we have variables for the month, the day, and the year. In a FORTRAN-like language, we would most likely encode these, so that January would correspond to 1, February to 2, and so on. In a language that supports user-defined types (like Pascal or Ada), we could define the variable month as an enumerated type {Jan., Feb., …, Dec.}. Either way, the values for min, min+, nom, max–, and max are clear from the context. When a variable has discrete, bounded values, as the variables in the commission problem have, the min, min+, nom, max–, and max are also easily determined. When no explicit bounds are present, as in the triangle problem, we usually have to create "artificial" bounds. The lower bound of side lengths is clearly 1 (a negative side length is silly); but what might we do for an upper bound? By default, the largest representable integer (called MAXINT in some languages) is one possibility; or we might impose an arbitrary upper limit such as 200 or 2000. For other data types, as long as a variable supports an ordering relation (see Chapter 3 for a definition), we can usually infer the min, min+, nominal, max–, and max values. Test values for alphabet characters, for example, would be {a, b, m, y, and z}.

Boundary value analysis does not make much sense for Boolean variables; the extreme values are TRUE and FALSE, but no clear choice is available for the remaining three. We will see in

Chapter 7 that Boolean variables lend themselves to decision table-based testing. Logical variables also present a problem for boundary value analysis. In the ATM example, a customer's PIN is a logical variable, as is the transaction type (deposit, withdrawal, or inquiry). We could go through the motions of boundary value analysis testing for such variables, but the exercise is not very satisfying to the tester's intuition.

5.1.2 Limitations of Boundary Value Analysis

Boundary value analysis works well when the program to be tested is a function of several independent variables that represent bounded physical quantities. Mathematically, the variables need to be described by a true ordering relation, in which, for every pair <a, b> of values of a variable, it is possible to say that a ≤ b or b ≤ a. (See Chapter 3 for a detailed definition of ordering relations.) Sets of car colors, for example, or football teams, do not support an ordering relation; thus, no form of boundary value testing is appropriate for such variables. The key words here are independent and physical quantities. A quick look at the boundary value analysis test cases for NextDate (in Section 5.5) shows them to be inadequate. Very little stress occurs on February and on leap years. The real problem here is that interesting dependencies exist among the month, day, and year variables. Boundary value analysis presumes the variables to be truly independent. Even so, boundary value analysis happens to catch end-of-month and end-of-year faults. Boundary value analysis test cases are derived from the extrema of bounded, independent variables that refer to physical quantities, with no consideration of the nature of the function, nor of the semantic meaning of the variables. We see boundary value analysis test cases to be rudimentary because they are obtained with very little insight and imagination. As with so many things, you get what you pay for.

The physical quantity criterion is equally important. When a variable refers to a physical quantity, such as temperature, pressure, air speed, angle of attack, load, and so forth, physical boundaries can be extremely important. (In an interesting example of this, Sky Harbor International Airport in Phoenix had to close on June 26, 1992, because the air temperature was 122°F. Aircraft pilots were unable to make certain instrument settings before takeoff: the instruments could only accept a maximum air temperature of 120°F.) In another case, a medical analysis system uses stepper motors to position a carousel of samples to be analyzed. It turns out that the mechanics of moving the carousel back to the starting cell often causes the robot arm to miss the first cell.

As an example of logical (vs. physical) variables, we might look at PINs or telephone numbers. It is hard to imagine what faults might be revealed by testing PIN values of 0000, 0001, 5000, 9998, and 9999.

5.2 Robust Boundary Value Testing

Robust boundary value testing is a simple extension of normal boundary value testing: in addition to the five boundary value analysis values of a variable, we see what happens when the extrema are exceeded with a value slightly greater than the maximum (max+) and a value slightly less than the minimum (min–). Robust boundary value test cases for our continuing example are shown in Figure 5.3.

Most of the discussion of boundary value analysis applies directly to robustness testing, especially the generalizations and limitations. The most interesting part of robustness testing is not with the inputs but with the expected outputs. What happens when a physical quantity exceeds its

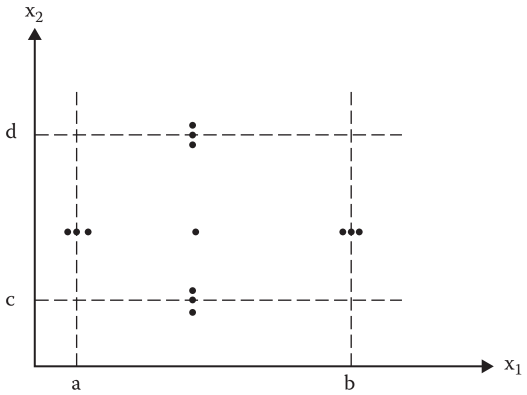


Figure 5.3 Robustness test cases for a function of two variables.

maximum? If it is the angle of attack of an airplane wing, the aircraft might stall. If it is the load capacity of a public elevator, we hope nothing special would happen. If it is a date, like May 32, we would expect an error message. The main value of robustness testing is that it forces attention on exception handling. With strongly typed languages, robustness testing may be very awkward. In Pascal, for example, if a variable is defined to be within a certain range, values outside that range result in run-time errors that abort normal execution. This raises an interesting question of implementation philosophy: is it better to perform explicit range checking and use exception handling to deal with "robust values," or is it better to stay with strong typing? The exception handling choice mandates robustness testing.

5.3 Worst-Case Boundary Value Testing

Both forms of boundary value testing, as we said earlier, make the single fault assumption of reliability theory. Owing to their similarity, we treat both normal worst-case boundary testing and robust worst-case boundary testing in this subsection. Rejecting single-fault assumption means that we are interested in what happens when more than one variable has an extreme value. In electronic circuit analysis, this is called "worst-case analysis"; we use that idea here to generate worst-case test cases. For each variable, we start with the five-element set that contains the min, min+, nom, max–, and max values. We then take the Cartesian product (see Chapter 3) of these sets to generate test cases. The result of the two-variable version of this is shown in Figure 5.4.

Worst-case boundary value testing is clearly more thorough in the sense that boundary value analysis test cases are a proper subset of worst-case test cases. It also represents much more effort: worst-case testing for a function of n variables generates 5^n test cases, as opposed to $4n + 1$ test cases for boundary value analysis.

Worst-case testing follows the generalization pattern we saw for boundary value analysis. It also has the same limitations, particularly those related to independence. Probably the best application for worst-case testing is where physical variables have numerous interactions, and where failure of the function is extremely costly. For really paranoid testing, we could go to robust worst-case testing. This involves the Cartesian product of the seven-element sets we used in robustness testing resulting in 7^n test cases. Figure 5.5 shows the robust worst-case test cases for our two-variable function.

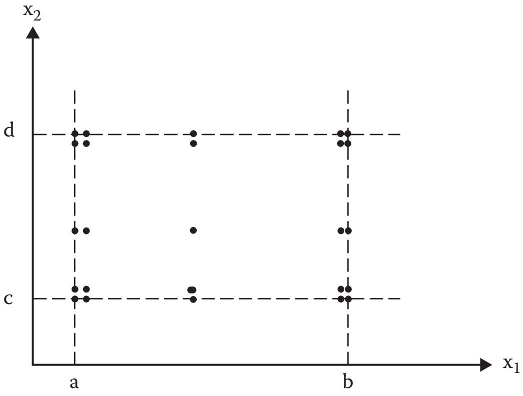


Figure 5.4 Worst-case test cases for a function of two variables.

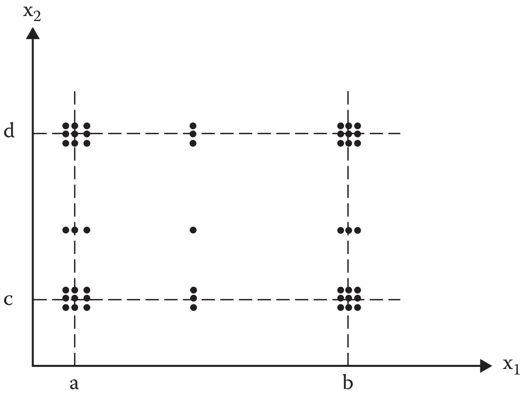


Figure 5.5 Robust worst-case test cases for a function of two variables.

5.4 Special Value Testing

Special value testing is probably the most widely practiced form of functional testing. It also is the most intuitive and the least uniform. Special value testing occurs when a tester uses domain knowledge, experience with similar programs, and information about "soft spots" to devise test cases. We might also call this *ad hoc* testing. No guidelines are used other than "best engineering judgment." As a result, special value testing is very dependent on the abilities of the tester.

Despite all the apparent negatives, special value testing can be very useful. In the next section, you will find test cases generated by the methods we just discussed for three of our examples. If you look carefully at these, especially for the NextDate function, you find that none is very satisfactory. Special value test cases for NextDate will include several test cases involving February 28, February 29, and leap years. Even though special value testing is highly subjective, it often results in a set of test cases that is more effective in revealing faults than the test sets generated by boundary value methods—testimony to the craft of software testing.

5.5 Examples

Each of the three continuing examples is a function of three variables. Printing all the test cases from all the methods for each problem is very space consuming, so we just have selected examples for worst-case boundary value and robust worst-case boundary value testing.

5.5.1 Test Cases for the Triangle Problem

In the problem statement, no conditions are specified on the triangle sides, other than being integers. Obviously, the lower bounds of the ranges are all 1. We arbitrarily take 200 as an upper bound. For each side, the test values are {1, 2, 100, 199, 200}. Robust boundary value test cases will add {0, 201}. Table 5.1 contains boundary value test cases using these ranges. Notice that test cases 3, 8, and 13 are identical; two should be deleted. Further, there is no test case for scalene triangles.

The cross-product of test values will have 125 test cases (some of which will be repeated)—too many to list here. The full set is available as a spreadsheet in the set of student exercises. Table 5.2 only lists the first 25 worst-case boundary value test cases for the triangle problem. You can picture them as a plane slice through the cube (actually it is a rectangular parallelepiped) in which a = 1 and the other two variables take on their full set of cross-product values.

Table 5.1 Normal Boundary Value Test Cases

Case	a	b	c	*Expected Output*
1	100	100	1	Isosceles
2	100	100	2	Isosceles
3	100	100	100	Equilateral
4	100	100	199	Isosceles
5	100	100	200	Not a triangle
6	100	1	100	Isosceles
7	100	2	100	Isosceles
8	100	100	100	Equilateral
9	100	199	100	Isosceles
10	100	200	100	Not a triangle
11	1	100	100	Isosceles
12	2	100	100	Isosceles
13	100	100	100	Equilateral
14	199	100	100	Isosceles
15	200	100	100	Not a triangle

Table 5.2 (Selected) Worst-Case Boundary Value Test Cases

Case	a	b	c	*Expected Output*
1	1	1	1	Equilateral
2	1	1	2	Not a triangle
3	1	1	100	Not a triangle
4	1	1	199	Not a triangle
5	1	1	200	Not a triangle
6	1	2	1	Not a triangle
7	1	2	2	Isosceles
8	1	2	100	Not a triangle
9	1	2	199	Not a triangle
10	1	2	200	Not a triangle
11	1	100	1	Not a triangle
12	1	100	2	Not a triangle
13	1	100	100	Isosceles
14	1	100	199	Not a triangle
15	1	100	200	Not a triangle
16	1	199	1	Not a triangle
17	1	199	2	Not a triangle
18	1	199	100	Not a triangle
19	1	199	199	Isosceles
20	1	199	200	Not a triangle
21	1	200	1	Not a triangle
22	1	200	2	Not a triangle
23	1	200	100	Not a triangle
24	1	200	199	Not a triangle
25	1	200	200	Isosceles

5.5.2 Test Cases for the NextDate Function

All 125 worst-case test cases for NextDate are listed in Table 5.3. Take some time to examine it for gaps of untested functionality and for redundant testing. For example, would anyone actually want to test January 1 in five different years? Is the end of February tested sufficiently?

Table 5.3 Worst-Case Test Cases

Case	*Month*	*Day*	*Year*	*Expected Output*
1	1	1	1812	1, 2, 1812
2	1	1	1813	1, 2, 1813
3	1	1	1912	1, 2, 1912
4	1	1	2011	1, 2, 2011
5	1	1	2012	1, 2, 2012
6	1	2	1812	1, 3, 1812
7	1	2	1813	1, 3, 1813
8	1	2	1912	1, 3, 1912
9	1	2	2011	1, 3, 2011
10	1	2	2012	1, 3, 2012
11	1	15	1812	1, 16, 1812
12	1	15	1813	1, 16, 1813
13	1	15	1912	1, 16, 1912
14	1	15	2011	1, 16, 2011
15	1	15	2012	1, 16, 2012
16	1	30	1812	1, 31, 1812
17	1	30	1813	1, 31, 1813
18	1	30	1912	1, 31, 1912
19	1	30	2011	1, 31, 2011
20	1	30	2012	1, 31, 2012
21	1	31	1812	2, 1, 1812
22	1	31	1813	2, 1, 1813
23	1	31	1912	2, 1, 1912
24	1	31	2011	2, 1, 2011
25	1	31	2012	2, 1, 2012
26	2	1	1812	2, 2, 1812
27	2	1	1813	2, 2, 1813
28	2	1	1912	2, 2, 1912

(*continued*)

Table 5.3 Worst-Case Test Cases (Continued)

Case	*Month*	*Day*	*Year*	*Expected Output*
29	2	1	2011	2, 2, 2011
30	2	1	2012	2, 2, 2012
31	2	2	1812	2, 3, 1812
32	2	2	1813	2, 3, 1813
33	2	2	1912	2, 3, 1912
34	2	2	2011	2, 3, 2011
35	2	2	2012	2, 3, 2012
36	2	15	1812	2, 16, 1812
37	2	15	1813	2, 16, 1813
38	2	15	1912	2, 16, 1912
39	2	15	2011	2, 16, 2011
40	2	15	2012	2, 16, 2012
41	2	30	1812	Invalid date
42	2	30	1813	Invalid date
43	2	30	1912	Invalid date
44	2	30	2011	Invalid date
45	2	30	2012	Invalid date
46	2	31	1812	Invalid date
47	2	31	1813	Invalid date
48	2	31	1912	Invalid date
49	2	31	2011	Invalid date
50	2	31	2012	Invalid date
51	6	1	1812	6, 2, 1812
52	6	1	1813	6, 2, 1813
53	6	1	1912	6, 2, 1912
54	6	1	2011	6, 2, 2011
55	6	1	2012	6, 2, 2012
56	6	2	1812	6, 3, 1812
57	6	2	1813	6, 3, 1813

(*continued*)

Table 5.3 Worst-Case Test Cases (Continued)

Case	*Month*	*Day*	*Year*	*Expected Output*
58	6	2	1912	6, 3, 1912
59	6	2	2011	6, 3, 2011
60	6	2	2012	6, 3, 2012
61	6	15	1812	6, 16, 1812
62	6	15	1813	6, 16, 1813
63	6	15	1912	6, 16, 1912
64	6	15	2011	6, 16, 2011
65	6	15	2012	6, 16, 2012
66	6	30	1812	7, 1, 1812
67	6	30	1813	7, 1, 1813
68	6	30	1912	7, 1, 1912
69	6	30	2011	7, 1, 2011
70	6	30	2012	7, 1, 2012
71	6	31	1812	Invalid date
72	6	31	1813	Invalid date
73	6	31	1912	Invalid date
74	6	31	2011	Invalid date
75	6	31	2012	Invalid date
76	11	1	1812	11, 2, 1812
77	11	1	1813	11, 2, 1813
78	11	1	1912	11, 2, 1912
79	11	1	2011	11, 2, 2011
80	11	1	2012	11, 2, 2012
81	11	2	1812	11, 3, 1812
82	11	2	1813	11, 3, 1813
83	11	2	1912	11, 3, 1912
84	11	2	2011	11, 3, 2011
85	11	2	2012	11, 3, 2012
86	11	15	1812	11, 16, 1812

(*continued*)

Table 5.3 Worst-Case Test Cases (Continued)

Case	*Month*	*Day*	*Year*	*Expected Output*
87	11	15	1813	11, 16, 1813
88	11	15	1912	11, 16, 1912
89	11	15	2011	11, 16, 2011
90	11	15	2012	11, 16, 2012
91	11	30	1812	12, 1, 1812
92	11	30	1813	12, 1, 1813
93	11	30	1912	12, 1, 1912
94	11	30	2011	12, 1, 2011
95	11	30	2012	12, 1, 2012
96	11	31	1812	Invalid date
97	11	31	1813	Invalid date
98	11	31	1912	Invalid date
99	11	31	2011	Invalid date
100	11	31	2012	Invalid date
101	12	1	1812	12, 2, 1812
102	12	1	1813	12, 2, 1813
103	12	1	1912	12, 2, 1912
104	12	1	2011	12, 2, 2011
105	12	1	2012	12, 2, 2012
106	12	2	1812	12, 3, 1812
107	12	2	1813	12, 3, 1813
108	12	2	1912	12, 3, 1912
109	12	2	2011	12, 3, 2011
110	12	2	2012	12, 3, 2012
111	12	15	1812	12, 16, 1812
112	12	15	1813	12, 16, 1813
113	12	15	1912	12, 16, 1912
114	12	15	2011	12, 16, 2011
115	12	15	2012	12, 16, 2012

(*continued*)

Table 5.3 Worst-Case Test Cases (Continued)

Case	*Month*	*Day*	*Year*	*Expected Output*
116	12	30	1812	12, 31, 1812
117	12	30	1813	12, 31, 1813
118	12	30	1912	12, 31, 1912
119	12	30	2011	12, 31, 2011
120	12	30	2012	12, 31, 2012
121	12	31	1812	1, 1, 1813
122	12	31	1813	1, 1, 1814
123	12	31	1912	1, 1, 1913
124	12	31	2011	1, 1, 2012
125	12	31	2012	1, 1, 2013

5.5.3 Test Cases for the Commission Problem

Instead of going through 125 boring test cases again, we will look at some more interesting test cases for the commission problem. This time, we will look at boundary values derived from the output range, especially near the threshold points of $1000 and $1800 where the commission percentage changes. The output space of the commission is shown in Figure 5.6. The intercepts of these threshold planes with the axes are shown.

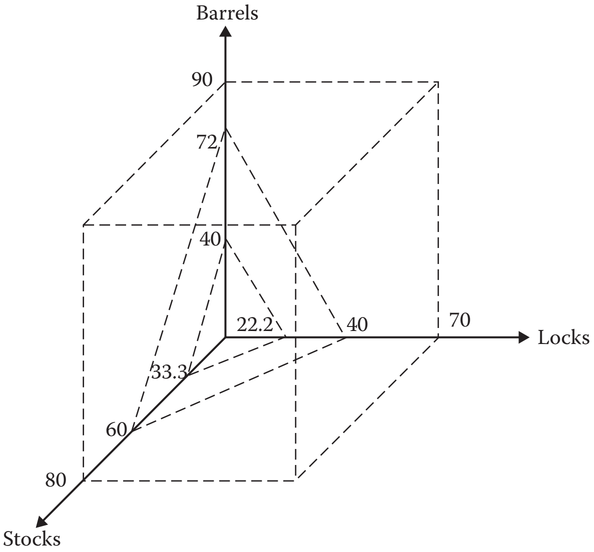


Figure 5.6 Input space of the commission problem.

Table 5.4 Output Boundary Value Analysis Test Cases

Case	*Locks*	*Stocks*	*Barrels*	*Sales*	*Comm*	*Comment*
1	1	1	1	100	10	Output minimum
2	1	1	2	125	12.5	Output minimum +
3	1	2	1	130	13	Output minimum +
4	2	1	1	145	14.5	Output minimum +
5	5	5	5	500	50	Midpoint
6	10	10	9	975	97.5	Border point –
7	10	9	10	970	97	Border point –
8	9	10	10	955	95.5	Border point –
9	10	10	10	1000	100	Border point
10	10	10	11	1025	103.75	Border point +
11	10	11	10	1030	104.5	Border point +
12	11	10	10	1045	106.75	Border point +
13	14	14	14	1400	160	Midpoint
14	18	18	17	1775	216.25	Border point –
15	18	17	18	1770	215.5	Border point –
16	17	18	18	1755	213.25	Border point –
17	18	18	18	1800	220	Border point
18	18	18	19	1825	225	Border point +
19	18	19	18	1830	226	Border point +
20	19	18	18	1845	229	Border point +
21	48	48	48	4800	820	Midpoint
22	70	80	89	7775	1415	Output maximum –
23	70	79	90	7770	1414	Output maximum –
24	69	80	90	7755	1411	Output maximum –
25	70	80	90	7800	1420	Output maximum

The volume between the origin and the lower plane corresponds to sales below the $1000 threshold. The volume between the two planes is the 15% commission range. Part of the reason for using the output range to determine test cases is that cases from the input range are almost all in the 20% zone. We want to find input variable combinations that stress the sales/commission boundary values: $100, $1000, $1800, and $7800. The minimum and maximum were easy, and

Table 5.5 Output Special Value Test Cases

Case	*Locks*	*Stocks*	*Barrels*	*Sales*	*Comm*	*Comment*
1	10	11	9	1005	100.75	Border point +
2	18	17	19	1795	219.25	Border point –
3	18	19	17	1805	221	Border point +

the numbers happen to work out so that the border points are easy to generate. Here is where it gets interesting: test case 9 is the $1000 border point. If we tweak the input variables, we get values just below and just above the border (cases 6–8 and 10–12). If we wanted to, we could pick values near the borders such as (22, 1, 1). As we continue in this way, we have a sense that we are "exercising" interesting parts of the code. We might claim that this is really a form of special value testing because we used our mathematical insight to generate test cases.

Table 5.4 contains test cases derived from boundary values on the output side of the commission function. Table 5.5 contains special value test cases.

5.6 Random Testing

At least two decades of discussion of random testing are included in the literature. Most of this interest is among academics, and in a statistical sense, it is interesting. Our three sample problems lend themselves nicely to random testing. The basic idea is that, rather than always choose the min, min+, nom, max–, and max values of a bounded variable, use a random number generator to pick test case values. This avoids any form of bias in testing. It also raises a serious question: how many random test cases are sufficient? Later, when we discuss structural test coverage metrics, we will have an elegant answer. For now, Tables 5.6 through 5.8 show the results of randomly generated test cases. They are derived from a Visual Basic application that picks values for a bounded variable $a \le x \le b$ as follows:

Table 5.6 Random Test Cases for Triangle Program

Test Cases	*Nontriangles*	*Scalene*	*Isosceles*	*Equilateral*
1289	663	593	32	1
15,436	7696	7372	367	1
17,091	8556	8164	367	1
2603	1284	1252	66	1
6475	3197	3122	155	1
5978	2998	2850	129	1
9008	4447	4353	207	1
Percentage	49.83%	47.87%	2.29%	0.01%

Table 5.7 Random Test Cases for Commission Program

Test Cases	*10%*	*15%*	*20%*
91	1	6	84
27	1	1	25
72	1	1	70
176	1	6	169
48	1	1	46
152	1	6	145
125	1	4	120
Percentage	1.01%	3.62%	95.37%

$$x = \text{Int}((b - a + 1) * \text{Rnd} + a)$$

where the function Int returns the integer part of a floating point number, and the function Rnd generates random numbers in the interval [0, 1]. The program keeps generating random test cases until at least one of each output occurs. In each table, the program went through seven "cycles" that ended with the "hard-to-generate" test case. In Tables 5.6 and 5.7, the last line shows what percentage of the random test cases was generated for each column. In the table for NextDate, the percentages are very close to the computed probability given in the last line of Table 5.8.

5.7 Guidelines for Boundary Value Testing

With the exception of special value testing, the test methods based on the input domain of a function (program) are the most rudimentary of all specification-based testing methods. They share the common assumption that the input variables are truly independent; and when this assumption is not warranted, the methods generate unsatisfactory test cases (such as June 31, 1912, for NextDate). Each of these methods can be applied to the output range of a program, as we did for the commission problem.

Another useful form of output-based test cases is for systems that generate error messages. The tester should devise test cases to check that error messages are generated when they are appropriate, and are not falsely generated. Boundary value analysis can also be used for internal variables, such as loop control variables, indices, and pointers. Strictly speaking, these are not input variables; however, errors in the use of these variables are quite common. Robustness testing is a good choice for testing internal variables.

There is a discussion in Chapter 10 about "the testing pendulum"—it refers to the problem of syntactic versus semantic approaches to developing test cases. Here is a short example given both ways. Consider a function F of three variables, a, b, and c. The boundaries are $0 \leq a < 10,000$, $0 \leq b < 10,000$, and $0 \leq c < 18.8$. The function F is $F = (a - b)/c$; Table 5.9 shows the normal boundary value test cases. Absent semantic knowledge, the first four test cases in Table 5.9 are what a boundary value testing tool would generate (a tool would not generate the expected output values). Even just the syntactic version is problematic—it does not avoid the division by zero possibility in test case 11.

Table 5.8 Random Test Cases for NextDate Program

Test Cases	*Days 1–30 of 31-Day Months*	*Day 31 of 31-Day Months*	*Days 1–29 of 30-Day Months*	*Day 30 of 30-Day Months*
913	542	17	274	10
1101	621	9	358	8
4201	2448	64	1242	46
1097	600	21	350	9
5853	3342	100	1804	82
3959	2195	73	1252	42
1436	786	22	456	13
Percentage	56.76%	1.65%	30.91%	1.13%
Probability	56.45%	1.88%	31.18%	1.88%

Days 1–27 of Feb.	*Feb. 28 of a Leap Year*	*Feb. 28 of a Non-Leap Year*	*Feb. 29 of a Leap Year*	*Impossible Days*
45	1	1	1	22
83	1	1	1	19
312	1	8	3	77
92	1	4	1	19
417	1	11	2	94
310	1	6	5	75
126	1	5	1	26
7.46%	0.04%	0.19%	0.08%	1.79%
7.26%	0.07%	0.20%	0.07%	1.01%

When we add the semantic information that F calculates the miles per gallon of an automobile, where a and b are end and start trip odometer values, and c is the gas tank capacity, we see more severe problems:

1. We must always have a ≥ b. This will avoid the negative values of F (test cases 1, 2, 9, and 10).
2. Test cases 3, 8, and 12–15 all refer to trips of length 0, so they could be collapsed into one test case, probably test case 8.
3. Division by zero is an obvious problem, thereby eliminating test case 11. Applying the semantic knowledge will result in the better set of case cases in Table 5.10.
4. Table 5.10 is still problematic—we never see the effect of boundary values on the tank capacity.

Table 5.9 Normal Boundary Value Test Cases for F = (a – b)/c

Test Case	a	b	c	F
1	0	5000	9.4	–531.9
2	1	5000	9.4	–531.8
3	5000	5000	9.4	0.0
4	9998	5000	9.4	531.7
5	9999	5000	9.4	531.8
6	5000	0	9.4	531.9
7	5000	1	9.4	531.8
8	5000	5000	9.4	0.0
9	5000	9998	9.4	–531.7
10	5000	9999	9.4	–531.8
11	5000	5000	0	Undefined
12	5000	5000	1	0.0
13	5000	5000	9.4	0.0
14	5000	5000	18.7	0.0
15	5000	5000	18.8	0.0

Table 5.10 Semantic Boundary Value Test Cases for F = (a – b)/c

Test Case	*End Odometer*	*Start Odometer*	*Tank Capacity*	*Miles per Gallon*
4	9998	5000	9.4	531.7
5	9999	5000	9.4	531.8
6	5000	0	9.4	531.9
7	5000	1	9.4	531.8
8	5000	5000	9.4	0.0

EXERCISES

1. Develop a formula for the number of robustness test cases for a function of n variables.
2. Develop a formula for the number of robust worst-case test cases for a function of n variables.
3. Make a Venn diagram showing the relationships among test cases from boundary value analysis, robustness testing, worst-case testing, and robust worst-case testing.
4. What happens if we try to do output range robustness testing? Use the commission problem as an example.

5. If you did exercise 8 in Chapter 2, you are already familiar with the CRC Press website for downloads (http://www.crcpress.com/product/isbn/9781466560680). There you will find an Excel spreadsheet named specBasedTesting.xls. (It is an extended version of Naive.xls, and it contains the same inserted faults.) Different sheets contain worst-case boundary value test cases for the triangle, NextDate, and commission problems, respectively. Run these sets of test cases and compare the results with your naive testing from Chapter 2.
6. Apply special value testing to the miles per gallon example in Tables 5.9 and 5.10. Provide reasons for your chosen test cases.

Chapter 6

Equivalence Class Testing

The use of equivalence classes as the basis for functional testing has two motivations: we would like to have a sense of complete testing, and, at the same time, we would hope to avoid redundancy. Neither of these hopes is realized by boundary value testing—looking at the tables of test cases, it is easy to see massive redundancy, and looking more closely, serious gaps exist. Equivalence class testing echoes the two deciding factors of boundary value testing, robustness and the single/multiple fault assumption. This chapter presents the traditional view of equivalence class testing, followed by a coherent treatment of four distinct forms based on the two assumptions. The single versus multiple fault assumption yields the weak/strong distinction and the focus on invalid data yields a second distinction: normal versus robust. Taken together, these two assumptions result in Weak Normal, Strong Normal, Weak Robust, and Strong Robust Equivalence Class testing.

Two problems occur with robust forms. The first is that, very often, the specification does not define what the expected output for an invalid input should be. (We could argue that this is a deficiency of the specification, but that does not get us anywhere.) Thus, testers spend a lot of time defining expected outputs for these cases. The second problem is that strongly typed languages eliminate the need for the consideration of invalid inputs. Traditional equivalence testing is a product of the time when languages such as FORTRAN and COBOL were dominant; thus, this type of error was common. In fact, it was the high incidence of such errors that led to the implementation of strongly typed languages.

6.1 Equivalence Classes

In Chapter 3, we noted that the important aspect of equivalence classes is that they form a partition of a set, where partition refers to a collection of mutually disjoint subsets, the union of which is the entire set. This has two important implications for testing—the fact that the entire set is represented provides a form of completeness, and the disjointedness ensures a form of nonredundancy. Because the subsets are determined by an equivalence relation, the elements of a subset have something in common. The idea of equivalence class testing is to identify test cases by using one element from each equivalence class. If the equivalence classes are chosen wisely, this greatly reduces the potential redundancy among test cases. In the triangle problem, for example, we would certainly have a test case for an equilateral triangle, and we might pick the triple (5, 5, 5) as inputs for a test case. If we did this, we would not expect to learn much from test cases such as (6, 6, 6) and (100, 100, 100). Our intuition tells us that these would be "treated the same" as the first test case; thus, they would be redundant. When we consider code-based testing in Chapters 8 and 9, we shall see that "treated the same" maps onto "traversing the same execution path." The four forms of equivalence class testing all address the problems of gaps and redundancies that are common to the four forms of boundary value testing. Since the assumptions align, the four forms of boundary value testing also align with the four forms of equivalence class testing. There will be one point of overlap—this occurs when equivalence classes are defined by bounded variables. In such cases, a hybrid of boundary value and equivalence class testing is appropriate. The International Software Testing Qualifications Board (ISTQB) syllabi refer to this as "edge testing." We will see this in the discussion in Section 6.3.

6.2 Traditional Equivalence Class Testing

Most of the standard testing texts (e.g., Myers, 1979; Mosley, 1993) discuss equivalence classes based on valid and invalid variable values. Traditional equivalence class testing is nearly identical to weak robust equivalence class testing (see Section 6.3.3). This traditional form focuses on invalid data values, and it is/was a consequence of the dominant style of programming in the 1960s and 1970s. Input data validation was an important issue at the time, and "Garbage In, Garbage Out" was the programmer's watchword. In the early years, it was the program user's responsibility to provide valid data. There was no guarantee about results based on invalid data. The term soon became known as GIGO. The usual response to GIGO was extensive input validation sections of a program. Authors and seminar leaders frequently commented that, in the classic afferent/central/efferent architecture of structured programming, the afferent portion often represented 80% of the total source code. In this context, it is natural to emphasize input data validation. Clearly, the defense against GIGO was to have extensive testing to assure data validity. The gradual shift to modern programming languages, especially those that feature strong data typing, and then to graphical user interfaces (GUIs) obviated much of the need for input data validation. Indeed, good use of user interface devices such as drop-down lists and slider bars reduces the likelihood of bad input data.

Traditional equivalence class testing echoes the process of boundary value testing. Figure 6.1 shows test cases for a function F of two variables x_1 and x_2, as we had in Chapter 5. The extension to more realistic cases of n variables proceeds as follows:

1. Test F for valid values of all variables.
2. If step 1 is successful, then test F for invalid values of x_1 with valid values of the remaining variables. Any failure will be due to a problem with an invalid value of x_1.
3. Repeat step 2 for the remaining variables.

One clear advantage of this process is that it focuses on finding faults due to invalid data. Since the GIGO concern was on invalid data, the kinds of combinations that we saw in the worst-case variations of boundary value testing were ignored. Figure 6.1 shows the five test cases for this process for our continuing function F of two variables.

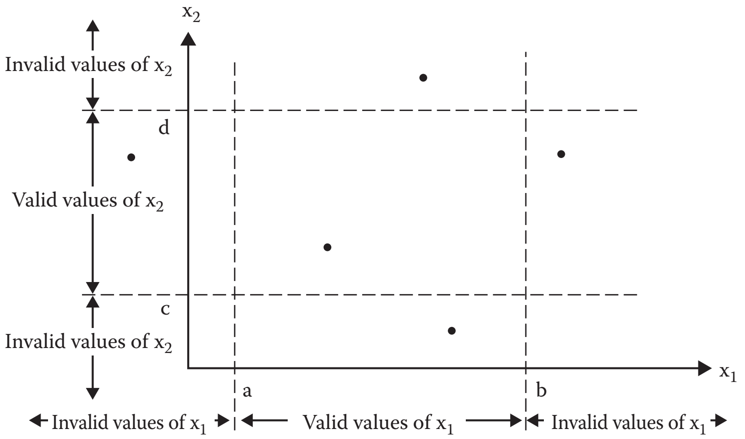


Figure 6.1 Traditional equivalence class test cases.

6.3 Improved Equivalence Class Testing

The key (and the craft!) of equivalence class testing is the choice of the equivalence relation that determines the classes. Very often, we make this choice by second-guessing the likely implementation and thinking about the functional manipulations that must somehow be present in the implementation. We will illustrate this with our continuing examples. We need to enrich the function we used in boundary value testing. Again, for the sake of comprehensible drawings, the discussion relates to a function, F, of two variables x_1 and x_2. When F is implemented as a program, the input variables x_1 and x_2 will have the following boundaries, and intervals within the boundaries:

$$a \le x_1 \le d, \text{ with intervals } [a, b), [b, c), [c, d]$$
$$e \le x_2 \le g, \text{ with intervals } [e, f), [f, g]$$

where square brackets and parentheses denote, respectively, closed and open interval endpoints. The intervals presumably correspond to some distinction in the program being tested, for example, the commission ranges in the commission problem. These ranges are equivalence classes. Invalid values of x_1 and x_2 are $x_1 < a$, $x_1 > d$, and $x_2 < e$, $x_2 > g$. The equivalence classes of valid values are

$$V1 = \{x_1: a \le x_1 < b\}, V2 = \{x_1: b \le x_1 < c\}, V3 = \{x_1: c \le x_1 \le d\}, V4 = \{x_2: e \le x_2 < f\}, V5 = \{x_2: f \le x_2 \le g\}$$

The equivalence classes of invalid values are

$$NV1 = \{x_1: x_1 < a\}, NV2 = \{x_1: d < x_1\}, NV3 = \{x_2: x_2 < e\}, NV4 = \{x_2: g < x_2\}$$

The equivalence classes V1, V2, V3, V4, V5, NV1, NV2, NV3, and NV4 are disjoint, and their union is the entire plane. In the succeeding discussions, we will just use the interval notation rather than the full formal set definition.

6.3.1 Weak Normal Equivalence Class Testing

With the notation as given previously, weak normal equivalence class testing is accomplished by using one variable from each equivalence class (interval) in a test case. (Note the effect of the single fault assumption.) For the running example, we would end up with the three weak equivalence class test cases shown in Figure 6.2. This figure will be repeated for the remaining forms of equivalence class testing, but, for clarity, without the indication of valid and invalid ranges. These three test cases use one value from each equivalence class. The test case in the lower left rectangle corresponds to a value of x_1 in the class [a, b), and to a value of x_2 in the class [e, f). The test case in the upper center rectangle corresponds to a value of x_1 in the class [b, c) and to a value of x_2 in the class [f, g]. The third test case could be in either rectangle on the right side of the valid values. We identified these in a systematic way, thus the apparent pattern. In fact, we will always have the same number of weak equivalence class test cases as classes in the partition with the largest number of subsets.

What can we learn from a weak normal equivalence class test case that fails, that is, one for which the expected and actual outputs are inconsistent? There could be a problem with x_1, or a problem with x_2, or maybe an interaction between the two. This ambiguity is the reason for the "weak" designation. If the expectation of failure is low, as it is for regression testing, this can be an acceptable choice. When more fault isolation is required, the stronger forms, discussed next, are indicated.

6.3.2 Strong Normal Equivalence Class Testing

Strong equivalence class testing is based on the multiple fault assumption, so we need test cases from each element of the Cartesian product of the equivalence classes, as shown in Figure 6.3. Notice the similarity between the pattern of these test cases and the construction of a truth table in propositional logic. The Cartesian product guarantees that we have a notion of "completeness" in two senses: we cover all the equivalence classes, and we have one of each possible combination of inputs. As we shall see from our continuing examples, the key to "good" equivalence class testing is the selection of the equivalence relation. Watch for the notion of inputs being "treated the same." Most of the time, equivalence class testing defines classes of the input domain. There is no reason why we could not define equivalence relations on the output range of the program function being tested; in fact, this is the simplest approach for the triangle problem.

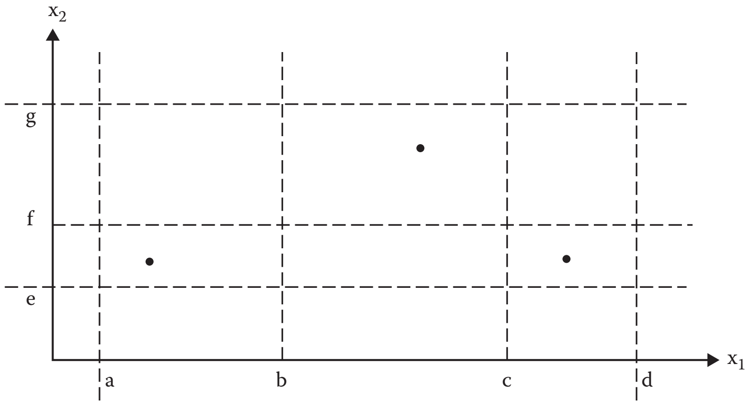


Figure 6.2 Weak normal equivalence class test cases.

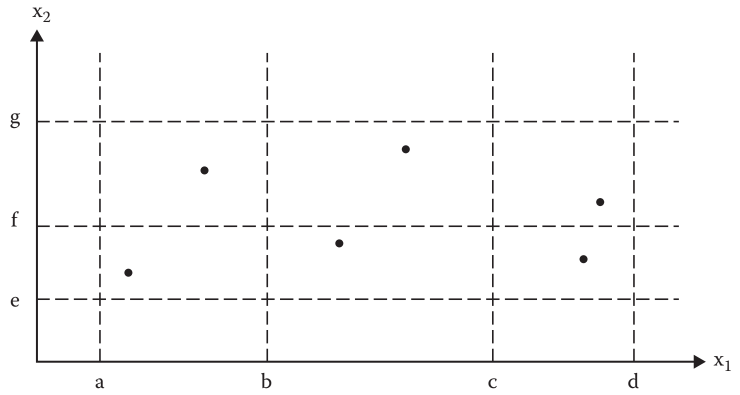


Figure 6.3 Strong normal equivalence class test cases.

6.3.3 Weak Robust Equivalence Class Testing

The name for this form is admittedly counterintuitive and oxymoronic. How can something be both weak and robust? The robust part comes from consideration of invalid values, and the weak part refers to the single fault assumption. The process of weak robust equivalence class testing is a simple extension of that for weak normal equivalence class testing—pick test cases such that each equivalence class is represented. In Figure 6.4, the test cases for valid classes are as those in Figure 6.2. The two additional test cases cover all four classes of invalid values. The process is similar to that for boundary value testing:

1. For valid inputs, use one value from each valid class (as in what we have called weak normal equivalence class testing). (Note that each input in these test cases will be valid.)
2. For invalid inputs, a test case will have one invalid value and the remaining values will all be valid. (Thus, a "single failure" should cause the test case to fail.)

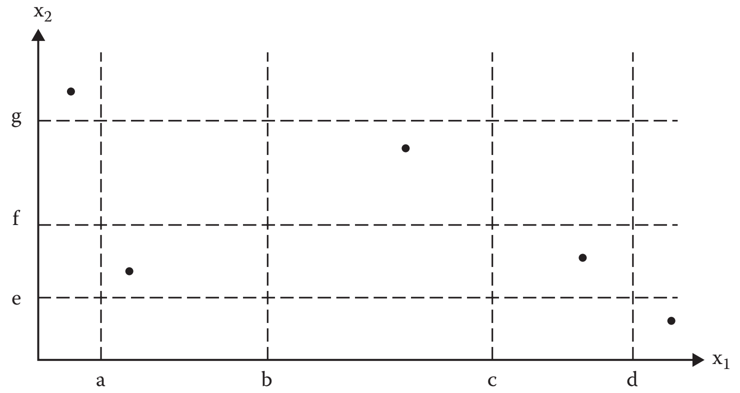


Figure 6.4 Weak robust equivalence class test cases.

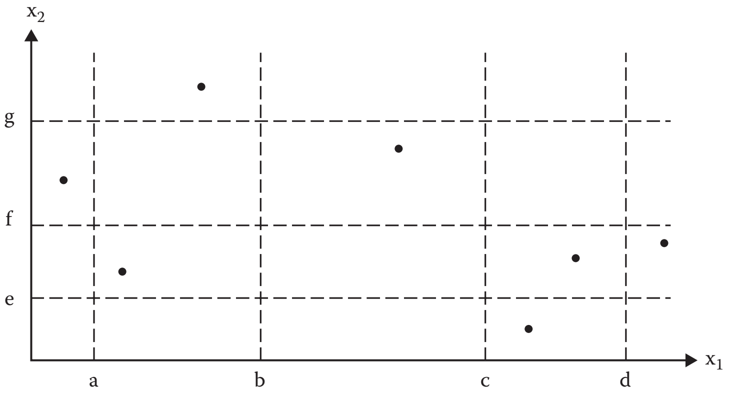


Figure 6.5 Revised weak robust equivalence class test cases.

The test cases resulting from this strategy are shown in Figure 6.4. There is a potential problem with these test cases. Consider the test cases in the upper left and lower right corners. Each of the test cases represents values from two invalid equivalence classes. Failure of either of these could be due to the interaction of two variables. Figure 6.5 presents a compromise between "pure" weak normal equivalence class testing and its robust extension.

6.3.4 Strong Robust Equivalence Class Testing

At least the name for this form is neither counterintuitive nor oxymoronic, just redundant. As before, the robust part comes from consideration of invalid values, and the strong part refers to the multiple fault assumption. We obtain test cases from each element of the Cartesian product of all the equivalence classes, both valid and invalid, as shown in Figure 6.6.

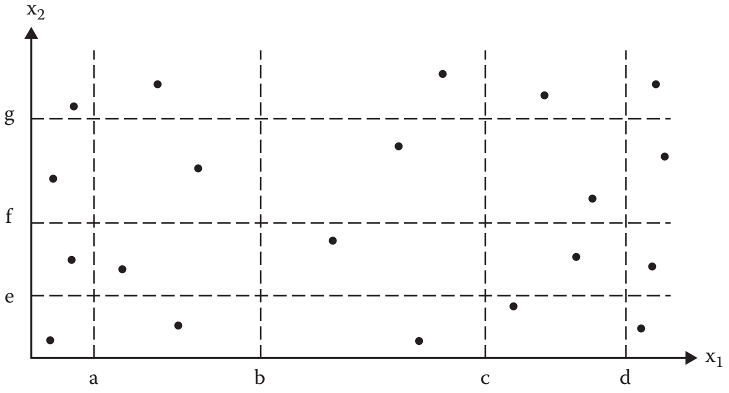


Figure 6.6 Strong robust equivalence class test cases.

6.4 Equivalence Class Test Cases for the Triangle Problem

In the problem statement, we note that four possible outputs can occur: NotATriangle, Scalene, Isosceles, and Equilateral. We can use these to identify output (range) equivalence classes as follows.

R1 = {<a, b, c>: the triangle with sides a, b, and c is equilateral}
R2 = {<a, b, c>: the triangle with sides a, b, and c is isosceles}
R3 = {<a, b, c>: the triangle with sides a, b, and c is scalene}
R4 = {<a, b, c>: sides a, b, and c do not form a triangle}

Four weak normal equivalence class test cases, chosen arbitrarily from each class are as follows:

Test Case	a	b	c	*Expected Output*
WN1	5	5	5	Equilateral
WN2	2	2	3	Isosceles
WN3	3	4	5	Scalene
WN4	4	1	2	Not a triangle

Because no valid subintervals of variables a, b, and c exist, the strong normal equivalence class test cases are identical to the weak normal equivalence class test cases.

Considering the invalid values for a, b, and c yields the following additional weak robust equivalence class test cases. (The invalid values could be zero, any negative number, or any number greater than 200.)

Test Case	a	b	c	*Expected Output*
WR1	–1	5	5	Value of a is not in the range of permitted values
WR2	5	–1	5	Value of b is not in the range of permitted values
WR3	5	5	–1	Value of c is not in the range of permitted values
WR4	201	5	5	Value of a is not in the range of permitted values
WR5	5	201	5	Value of b is not in the range of permitted values
WR6	5	5	201	Value of c is not in the range of permitted values

Here is one "corner" of the cube in three-space of the additional strong robust equivalence class test cases:

Test Case	a	b	c	*Expected Output*
SR1	–1	5	5	Value of a is not in the range of permitted values
SR2	5	–1	5	Value of b is not in the range of permitted values
SR3	5	5	–1	Value of c is not in the range of permitted values
SR4	–1	–1	5	Values of a, b are not in the range of permitted values
SR5	5	–1	–1	Values of b, c are not in the range of permitted values
SR6	–1	5	–1	Values of a, c are not in the range of permitted values
SR7	–1	–1	–1	Values of a, b, c are not in the range of permitted values

Notice how thoroughly the expected outputs describe the invalid input values.

Equivalence class testing is clearly sensitive to the equivalence relation used to define classes. Here is another instance of craftsmanship. If we base equivalence classes on the output domain, we obtain a richer set of test cases. What are some of the possibilities for the three integers, a, b, and c? They can all be equal, exactly one pair can be equal (this can happen in three ways), or none can be equal.

$D1 = \{<a, b, c>: a = b = c\}$
$D2 = \{<a, b, c>: a = b, a \neq c\}$
$D3 = \{<a, b, c>: a = c, a \neq b\}$
$D4 = \{<a, b, c>: b = c, a \neq b\}$
$D5 = \{<a, b, c>: a \neq b, a \neq c, b \neq c\}$

As a separate question, we can apply the triangle property to see if they even constitute a triangle. (For example, the triplet <1, 4, 1> has exactly one pair of equal sides, but these sides do not form a triangle.)

$D6 = \{<a, b, c>: a \geq b + c\}$
$D7 = \{<a, b, c>: b \geq a + c\}$
$D8 = \{<a, b, c>: c \geq a + b\}$

If we wanted to be still more thorough, we could separate the "greater than or equal to" into the two distinct cases; thus, the set D6 would become

$D6' = \{<a, b, c>: a = b + c\}$
$D6'' = \{<a, b, c>: a > b + c\}$

and similarly for D7 and D8.

6.5 Equivalence Class Test Cases for the NextDate Function

The NextDate function illustrates very well the craft of choosing the underlying equivalence relation. Recall that NextDate is a function of three variables: month, day, and year, and these have intervals of valid values defined as follows:

M1 = {month: $1 \le$ month ≤ 12}
D1 = {day: $1 \le$ day ≤ 31}
Y1 = {year: $1812 \le$ year ≤ 2012}

The invalid equivalence classes are

M2 = {month: month < 1}
M3 = {month: month > 12}
D2 = {day: day < 1}
D3 = {day: day > 31}
Y2 = {year: year < 1812}
Y3 = {year: year > 2012}

Because the number of valid classes equals the number of independent variables, only one weak normal equivalence class test case occurs, and it is identical to the strong normal equivalence class test case:

Case ID	*Month*	*Day*	*Year*	*Expected Output*
WN1, SN1	6	15	1912	6/16/1912

Here is the full set of weak robust test cases:

Case ID	*Month*	*Day*	*Year*	*Expected Output*
WR1	6	15	1912	6/16/1912
WR2	–1	15	1912	Value of month not in the range 1 ... 12
WR3	13	15	1912	Value of month not in the range 1 ... 12
WR4	6	–1	1912	Value of day not in the range 1 ... 31
WR5	6	32	1912	Value of day not in the range 1 ... 31
WR6	6	15	1811	Value of year not in the range 1812 ... 2012
WR7	6	15	2013	Value of year not in the range 1812 ... 2012

As with the triangle problem, here is one "corner" of the cube in three-space of the additional strong robust equivalence class test cases:

Case ID	*Month*	*Day*	*Year*	*Expected Output*
SR1	–1	15	1912	Value of month not in the range 1 ... 12
SR2	6	–1	1912	Value of day not in the range 1 ... 31
SR3	6	15	1811	Value of year not in the range 1812 ... 2012
SR4	–1	–1	1912	Value of month not in the range 1 ... 12 Value of day not in the range 1 ... 31
SR5	6	–1	1811	Value of day not in the range 1 ... 31 Value of year not in the range 1812 ... 2012
SR6	–1	15	1811	Value of month not in the range 1 ... 12 Value of year not in the range 1812 ... 2012
SR7	–1	–1	1811	Value of month not in the range 1 ... 12 Value of day not in the range 1 ... 31 Value of year not in the range 1812 ... 2012

If we more carefully choose the equivalence relation, the resulting equivalence classes will be more useful. Recall that earlier we said that the gist of the equivalence relation is that elements in a class are "treated the same way." One way to see the deficiency of the traditional approach is that the "treatment" is at the valid/invalid level. We next reduce the granularity by focusing on more specific treatment.

What must be done to an input date? If it is not the last day of a month, the NextDate function will simply increment the day value. At the end of a month, the next day is 1 and the month is incremented. At the end of a year, both the day and the month are reset to 1, and the year is incremented. Finally, the problem of leap year makes determining the last day of a month interesting. With all this in mind, we might postulate the following equivalence classes:

M1 = {month: month has 30 days}
M2 = {month: month has 31 days}
M3 = {month: month is February}
D1 = {day: $1 \le \text{day} \le 28$}
D2 = {day: day = 29}
D3 = {day: day = 30}
D4 = {day: day = 31}
Y1 = {year: year = 2000}
Y2 = {year: year is a non-century leap year}
Y3 = {year: year is a common year}

By choosing separate classes for 30- and 31-day months, we simplify the question of the last day of the month. By taking February as a separate class, we can give more attention to leap year questions. We also give special attention to day values: days in D1 are (nearly) always incremented, while days in D4 only have meaning for months in M2. Finally, we have three classes of years, the special case of the year 2000, leap years, and non-leap years. This is not a perfect set of equivalence classes, but its use will reveal many potential errors.

6.5.1 Equivalence Class Test Cases

These classes yield the following weak normal equivalence class test cases. As before, the inputs are mechanically selected from the approximate middle of the corresponding class:

Case ID	*Month*	*Day*	*Year*	*Expected Output*
WN1	6	14	2000	6/15/2000
WN2	7	29	1996	7/30/1996
WN3	2	30	2002	Invalid input date
WN4	6	31	2000	Invalid input date

Mechanical selection of input values makes no consideration of our domain knowledge, thus the two impossible dates. This will always be a problem with "automatic" test case generation, because all of our domain knowledge is not captured in the choice of equivalence classes. The strong normal equivalence class test cases for the revised classes are as follows:

Case ID	*Month*	*Day*	*Year*	*Expected Output*
SN1	6	14	2000	6/15/2000
SN2	6	14	1996	6/15/1996
SN3	6	14	2002	6/15/2002
SN4	6	29	2000	6/30/2000
SN5	6	29	1996	6/30/1996
SN6	6	29	2002	6/30/2002
SN7	6	30	2000	Invalid input date
SN8	6	30	1996	Invalid input date
SN9	6	30	2002	Invalid input date
SN10	6	31	2000	Invalid input date
SN11	6	31	1996	Invalid input date
SN12	6	31	2002	Invalid input date
SN13	7	14	2000	7/15/2000
SN14	7	14	1996	7/15/1996

Case ID	Month	Day	Year	Expected Output
SN15	7	14	2002	7/15/2002
SN16	7	29	2000	7/30/2000
SN17	7	29	1996	7/30/1996
SN18	7	29	2002	7/30/2002
SN19	7	30	2000	7/31/2000
SN20	7	30	1996	7/31/1996
SN21	7	30	2002	7/31/2002
SN22	7	31	2000	8/1/2000
SN23	7	31	1996	8/1/1996
SN24	7	31	2002	8/1/2002
SN25	2	14	2000	2/15/2000
SN26	2	14	1996	2/15/1996
SN27	2	14	2002	2/15/2002
SN28	2	29	2000	3/1/2000
SN29	2	29	1996	3/1/1996
SN30	2	29	2002	Invalid input date
SN31	2	30	2000	Invalid input date
SN32	2	30	1996	Invalid input date
SN33	2	30	2002	Invalid input date
SN34	2	31	2000	Invalid input date
SN35	2	31	1996	Invalid input date
SN36	2	31	2002	Invalid input date

Moving from weak to strong normal testing raises some of the issues of redundancy that we saw with boundary value testing. The move from weak to strong, whether with normal or robust classes, always makes the presumption of independence, and this is reflected in the cross product of the equivalence classes. Three month classes times four day classes times three year classes results in 36 strong normal equivalence class test cases. Adding two invalid classes for each variable will result in 150 strong robust equivalence class test cases (too many to show here!).

We could also streamline our set of test cases by taking a closer look at the year classes. If we merge Y1 and Y2, and call the result the set of leap years, our 36 test cases would drop down to 24. This change suppresses special attention to considerations in the year 2000, and it also adds some complexity to the determination of which years are leap years. Balance this against how much might be learned from the present test cases.

6.6 Equivalence Class Test Cases for the Commission Problem

The input domain of the commission problem is "naturally" partitioned by the limits on locks, stocks, and barrels. These equivalence classes are exactly those that would also be identified by traditional equivalence class testing. The first class is the valid input; the other two are invalid. The input domain equivalence classes lead to very unsatisfactory sets of test cases. Equivalence classes defined on the output range of the commission function will be an improvement.

The valid classes of the input variables are

L1 = {locks: 1 ≤ locks ≤ 70}
L2 = {locks = –1} (occurs if locks = –1 is used to control input iteration)
S1 = {stocks: 1 ≤ stocks ≤ 80}
B1 = {barrels: 1 ≤ barrels ≤ 90}

The corresponding invalid classes of the input variables are

L3 = {locks: locks = 0 OR locks < –1}
L4 = {locks: locks > 70}
S2 = {stocks: stocks < 1}
S3 = {stocks: stocks > 80}
B2 = {barrels: barrels < 1}
B3 = {barrels: barrels > 90}

One problem occurs, however. The variable "locks" is also used as a sentinel to indicate no more telegrams. When a value of –1 is given for locks, the while loop terminates, and the values of totalLocks, totalStocks, and totalBarrels are used to compute sales, and then commission.

Except for the names of the variables and the interval endpoint values, this is identical to our first version of the NextDate function. Therefore, we will have exactly one weak normal equivalence class test case—and again, it is identical to the strong normal equivalence class test case. Note that the case for locks = –1 just terminates the iteration. We will have eight weak robust test cases.

Case ID	*Locks*	*Stocks*	*Barrels*	*Expected Output*
WR1	10	10	10	$100
WR2	–1	40	45	Program terminates
WR3	–2	40	45	Value of locks not in the range 1 ... 70
WR4	71	40	45	Value of locks not in the range 1 ... 70
WR5	35	–1	45	Value of stocks not in the range 1 ... 80
WR6	35	81	45	Value of stocks not in the range 1 ... 80
WR7	35	40	–1	Value of barrels not in the range 1 ... 90
WR8	35	40	91	Value of barrels not in the range 1 ... 90

Here is one “corner” of the cube in 3-space of the additional strong robust equivalence class test cases:

Case ID	*Locks*	*Stocks*	*Barrels*	*Expected Output*
SR1	–2	40	45	Value of locks not in the range 1 ... 70
SR2	35	–1	45	Value of stocks not in the range 1 ... 80
SR3	35	40	–2	Value of barrels not in the range 1 ... 90
SR4	–2	–1	45	Value of locks not in the range 1 ... 70 Value of stocks not in the range 1 ... 80
SR5	–2	40	–1	Value of locks not in the range 1 ... 70 Value of barrels not in the range 1 ... 90
SR6	35	–1	–1	Value of stocks not in the range 1 ... 80 Value of barrels not in the range 1 ... 90
SR7	–2	–1	–1	Value of locks not in the range 1 ... 70 Value of stocks not in the range 1 ... 80 Value of barrels not in the range 1 ... 90

Notice that, of strong test cases—whether normal or robust—only one is a legitimate input. If we were really worried about error cases, this might be a good set of test cases. It can hardly give us a sense of confidence about the calculation portion of the problem, however. We can get some help by considering equivalence classes defined on the output range. Recall that sales is a function of the number of locks, stocks, and barrels sold:

Sales = 45 <mathMultiply> locks + 30 <mathMultiply> stocks + 25 <mathMultiply> barrels

We could define equivalence classes of three variables by commission ranges:

S1 = {<locks, stocks, barrels>: sales ≤ 1000}
S2 = {<locks, stocks, barrels>: 1000 < sales ≤ 1800}
S3 = {<locks, stocks, barrels>: sales > 1800}

Figure 5.6 helps us get a better feel for the input space. Elements of S1 are points with integer coordinates in the pyramid near the origin. Elements of S2 are points in the “triangular slice” between the pyramid and the rest of the input space. Finally, elements of S3 are all those points in the rectangular volume that are not in S1 or in S2. All the error cases found by the strong equivalence classes of the input domain are outside of the rectangular space shown in Figure 5.6.

As was the case with the triangle problem, the fact that our input is a triplet means that we no longer take test cases from a Cartesian product.

Test Case	*Locks*	*Stocks*	*Barrels*	*Sales*	*Commission*
OR1	5	5	5	500	50
OR2	15	15	15	1500	175
OR3	25	25	25	2500	360

These test cases give us some sense that we are exercising important parts of the problem. Together with the weak robust test cases, we would have a pretty good test of the commission problem. We might want to add some boundary checking, just to make sure the transitions at sales of $1000 and $1800 are correct. This is not particularly easy because we can only choose values of locks, stocks, and barrels. It happens that the constants in this example are contrived so that there are "nice" triplets.

6.7 Edge Testing

The *ISTQB Advanced Level Syllabus* (ISTQB, 2012) describes a hybrid of boundary value analysis and equivalence class testing and gives it the name "edge testing." The need for this occurs when contiguous ranges of a particular variable constitute equivalence classes. Figure 6.2 shows three equivalence classes of valid values for x_1 and two classes for x_2. Presumably, these classes refer to variables that are "treated the same" in some application. This suggests that there may be faults near the boundaries of the classes, and edge testing will exercise these potential faults. For the example in Figure 6.2, a full set of edge testing test values are as follows:

Normal test values for x_1: {a, a+, b–, b, b+, c–, c, c+, d–, d}
Robust test values for x_1: {a–, a, a+, b–, b, b+, c–, c, c+, d–, d, d+}
Normal test values for x_2: {e, e+, f–, f, f+, g–, g}
Robust test values for x_2: {e–, e, e+, f–, f, f+, g–, g, g+}

One subtle difference is that edge test values do not include the nominal values that we had with boundary value testing. Once the sets of edge values are determined, edge testing can follow any of the four forms of equivalence class testing. The numbers of test cases obviously increase as with the variations of boundary value and equivalence class testing.

6.8 Guidelines and Observations

Now that we have gone through three examples, we conclude with some observations about, and guidelines for, equivalence class testing.

1. Obviously, the weak forms of equivalence class testing (normal or robust) are not as comprehensive as the corresponding strong forms.
2. If the implementation language is strongly typed (and invalid values cause run-time errors), it makes no sense to use the robust forms.

3. If error conditions are a high priority, the robust forms are appropriate.
4. Equivalence class testing is appropriate when input data is defined in terms of intervals and sets of discrete values. This is certainly the case when system malfunctions can occur for out-of-limit variable values.
5. Equivalence class testing is strengthened by a hybrid approach with boundary value testing. (We can "reuse" the effort made in defining the equivalence classes.)
6. Equivalence class testing is indicated when the program function is complex. In such cases, the complexity of the function can help identify useful equivalence classes, as in the NextDate function.
7. Strong equivalence class testing makes a presumption that the variables are independent, and the corresponding multiplication of test cases raises issues of redundancy. If any dependencies occur, they will often generate "error" test cases, as they did in the NextDate function. (The decision table technique in Chapter 7 resolves this problem.)
8. Several tries may be needed before the "right" equivalence relation is discovered, as we saw in the NextDate example. In other cases, there is an "obvious" or "natural" equivalence relation. When in doubt, the best bet is to try to second-guess aspects of any reasonable implementation. This is sometimes known as the "competent programmer hypothesis."
9. The difference between the strong and weak forms of equivalence class testing is helpful in the distinction between progression and regression testing.

EXERCISES

1. Starting with the 36 strong normal equivalence class test cases for the NextDate function, revise the day classes as discussed, and then find the other nine test cases.
2. If you use a compiler for a strongly typed language, discuss how it would react to robust equivalence class test cases.
3. Revise the set of weak normal equivalence classes for the extended triangle problem that considers right triangles.
4. Compare and contrast the single/multiple fault assumption with boundary value and equivalence class testing.
5. The spring and fall changes between standard and daylight savings time create an interesting problem for telephone bills. In the spring, this switch occurs at 2:00 a.m. on a Sunday morning (late March, early April) when clocks are reset to 3:00 a.m. The symmetric change takes place usually on the last Sunday in October, when the clock changes from 2:59:59 back to 2:00:00.

   Develop equivalence classes for a long-distance telephone service function that bills calls using the following rate structure:

   Call duration ≤20 minutes charged at \$0.05 per minute or fraction of a minute
   Call duration >20 minutes charged at \$1.00 plus \$0.10 per minute or fraction of a minute in excess of 20 minutes.

   Make these assumptions:
   - Chargeable time of a call begins when the called party answers, and ends when the calling party disconnects.
   - Call durations of seconds are rounded up to the next larger minute.
   - No call lasts more than 30 hours.
6. If you did exercise 8 in Chapter 2, and exercise 5 in Chapter 5, you are already familiar with the CRC Press website for downloads (http://www.crcpress.com/product/isbn/97818466560680). There you will find an Excel spreadsheet named specBasedTesting.xls.

(It is an extended version of Naive.xls, and it contains the same inserted faults.) Different sheets contain strong, normal equivalence class test cases for the triangle, NextDate, and commission problems, respectively. Run these sets of test cases and compare the results with your naive testing from Chapter 2 and your boundary value testing from Chapter 5.

References

ISTQB Advanced Level Working Party, *ISTQB Advanced Level Syllabus*, 2012.

Mosley, D.J., *The Handbook of MIS Application Software Testing,* Yourdon Press, Prentice Hall, Englewood Cliffs, NJ, 1993.

Myers, G.J., *The Art of Software Testing,* Wiley Interscience, New York, 1979.

Chapter 7

Decision Table–Based Testing

Of all the functional testing methods, those based on decision tables are the most rigorous because of their strong logical basis. Two closely related methods are used: cause-and-effect graphing (Elmendorf, 1973; Myers, 1979) and the decision tableau method (Mosley, 1993). These are more cumbersome to use and are fully redundant with decision tables; both are covered in Mosley (1993). For the curious, or for the sake of completeness, Section 7.5 offers a short discussion of cause-and-effect graphing.

7.1 Decision Tables

Decision tables have been used to represent and analyze complex logical relationships since the early 1960s. They are ideal for describing situations in which a number of combinations of actions are taken under varying sets of conditions. Some of the basic decision table terms are illustrated in Table 7.1.

A decision table has four portions: the part to the left of the bold vertical line is the stub portion; to the right is the entry portion. The part above the bold horizontal line is the condition portion, and below is the action portion. Thus, we can refer to the condition stub, the condition entries, the action stub, and the action entries. A column in the entry portion is a rule. Rules indicate which actions, if any, are taken for the circumstances indicated in the condition portion of the rule. In the decision table in Table 7.1, when conditions c1, c2, and c3 are all true, actions a1 and a2 occur. When c1 and c2 are both true and c3 is false, then actions a1 and a3 occur. The entry for c3 in the rule where c1 is true and c2 is false is called a "don't care" entry. The don't care entry has two major interpretations: the condition is irrelevant, or the condition does not apply. Sometimes people will enter the "n/a" symbol for this latter interpretation.

When we have binary conditions (true/false, yes/no, 0/1), the condition portion of a decision table is a truth table (from propositional logic) that has been rotated 90°. This structure guarantees that we consider every possible combination of condition values. When we use decision tables for test case identification, this completeness property of a decision table guarantees a form of complete testing. Decision tables in which all the conditions are binary are called Limited Entry Decision Tables (LETDs). If conditions are allowed to have several values, the resulting tables

Table 7.1 Portions of a Decision Table

Stub	*Rule 1*	*Rule 2*	*Rules 3, 4*	*Rule 5*	*Rule 6*	*Rules 7, 8*
c1	T	T	T	F	F	F
c2	T	T	F	T	T	F
c3	T	F	—	T	F	—
a1	X	X		X		
a2	X				X	
a3		X		X		
a4			X			X

are called Extended Entry Decision Tables (EEDTs). We will see examples of both types for the NextDate problem. Decision tables are deliberately declarative (as opposed to imperative); no particular order is implied by the conditions, and selected actions do not occur in any particular order.

7.2 Decision Table Techniques

To identify test cases with decision tables, we interpret conditions as inputs and actions as outputs. Sometimes conditions end up referring to equivalence classes of inputs, and actions refer to major functional processing portions of the item tested. The rules are then interpreted as test cases. Because the decision table can mechanically be forced to be complete, we have some assurance that we will have a comprehensive set of test cases. Several techniques that produce decision tables are more useful to testers. One helpful style is to add an action to show when a rule is logically impossible. In the decision table in Table 7.2, we see examples of don't care entries and impossible rule usage. If the integers a, b, and c do not constitute a triangle, we do not even care about possible

Table 7.2 Decision Table for Triangle Problem

c1: a, b, c form a triangle?	F	T	T	T	T	T	T	T	T
c2: a = b?	—	T	T	T	T	F	F	F	F
c3: a = c?	—	T	T	F	F	T	T	F	F
c4: b = c?	—	T	F	T	F	T	F	T	F
a1: Not a triangle	X								
a2: Scalene									X
a3: Isosceles					X		X	X	
a4: Equilateral		X							
a5: Impossible			X	X		X			

equalities, as indicated in the first rule. In rules 3, 4, and 6, if two pairs of integers are equal, by transitivity, the third pair must be equal; thus, the negative entry makes these rules impossible.

The decision table in Table 7.3 illustrates another consideration: the choice of conditions can greatly expand the size of a decision table. Here, we have expanded the old condition (c1: a, b, c form a triangle?) to a more detailed view of the three inequalities of the triangle property. If any one of these fails, the three integers do not constitute sides of a triangle.

We could expand this still further because there are two ways an inequality could fail: one side could equal the sum of the other two, or it could be strictly greater.

When conditions refer to equivalence classes, decision tables have a characteristic appearance. Conditions in the decision table in Table 7.4 are from the NextDate problem; they refer to the mutually exclusive possibilities for the month variable. Because a month is in exactly one equivalence class, we cannot ever have a rule in which two entries are true. The don't care entries (—) really mean "must be false." Some decision table aficionados use the notation *F*! to make this point.

Use of don't care entries has a subtle effect on the way in which complete decision tables are recognized. For a limited entry decision table with n conditions, there must be 2^n independent

Table 7.3 Refined Decision Table for Triangle Problem

c1: a < b + c?	F	T	T	T	T	T	T	T	T	T	T
c2: b < a + c?	—	F	T	T	T	T	T	T	T	T	T
c3: c < a + b?	—	—	F	T	T	T	T	T	T	T	T
c4: a = b?	—	—	—	T	T	T	T	F	F	F	F
c5: a = c?	—	—	—	T	T	F	F	T	T	F	F
c6: b = c?	—	—	—	T	F	T	F	T	F	T	F
a1: Not a triangle	X	X	X								
a2: Scalene											X
a3: Isosceles							X		X	X	
a4: Equilateral				X							
a5: Impossible					X	X		X			

Table 7.4 Decision Table with Mutually Exclusive Conditions

Conditions	*R1*	*R2*	*R3*
c1: Month in M1?	T	—	—
c2: Month in M2?	—	T	—
c3: Month in M3?	—	—	T
a1			
a2			
a3			

rules. When don't care entries really indicate that the condition is irrelevant, we can develop a rule count as follows: rules in which no don't care entries occur count as one rule, and each don't care entry in a rule doubles the count of that rule. The rule counts for the decision table in Table 7.3 are shown in Table 7.5. Notice that the sum of the rule counts is 64 (as it should be).

If we applied this simplistic algorithm to the decision table in Table 7.4, we get the rule counts shown in Table 7.6. We should only have eight rules, so we clearly have a problem. To see where the problem lies, we expand each of the three rules, replacing the "—" entries with the T and F possibilities, as shown in Table 7.7.

Notice that we have three rules in which all entries are T: rules 1.1, 2.1, and 3.1. We also have two rules with T, T, F entries: rules 1.2 and 2.2. Similarly, rules 1.3 and 3.2 are identical; so are rules 2.3 and 3.3. If we delete the repetitions, we end up with seven rules; the missing rule is the one in which all conditions are false. The result of this process is shown in Table 7.8. The impossible rules are also shown.

Table 7.5 Decision Table for Table 7.3 with Rule Counts

c1: a < b + c?	F	T	T	T	T	T	T	T	T	T	T
c2: b < a + c?	—	F	T	T	T	T	T	T	T	T	T
c3: c < a + b?	—	—	F	T	T	T	T	T	T	T	T
c4: a = b?	—	—	—	T	T	T	T	F	F	F	F
c5: a = c?	—	—	—	T	T	F	F	T	T	F	F
c6: b = c?	—	—	—	T	F	T	F	T	F	T	F
Rule count	32	16	8	1	1	1	1	1	1	1	1
a1: Not a triangle	X	X	X								
a2: Scalene											X
a3: Isosceles							X		X	X	
a4: Equilateral				X							
a5: Impossible					X	X		X			

Table 7.6 Rule Counts for a Decision Table with Mutually Exclusive Conditions

Conditions	*R1*	*R2*	*R3*
c1: Month in M1	T	—	—
c2: Month in M2	—	T	—
c3: Month in M3	—	—	T
Rule count	4	4	4
a1			

Table 7.7 Impossible Rules in Table 7.7

Conditions	*1.1*	*1.2*	*1.3*	*1.4*	*2.1*	*2.2*	*2.3*	*2.4*	*3.1*	*3.2*	*3.3*	*3.4*
c1: Month in M1	T	T	T	T	T	T	F	F	T	T	F	F
c2: Month in M2	T	T	F	F	T	T	T	T	T	F	T	F
c3: Month in M3	T	F	T	F	T	F	T	F	T	T	T	T
Rule count	1	1	1	1	1	1	1	1	1	1	1	1
a1: Impossible	X	X	X	—	X	X	X	—	X	X	—	

Table 7.8 Mutually Exclusive Conditions with Impossible Rules

	1.1	*1.2*	*1.3*	*1.4*	*2.3*	*2.4*	*3.4*	
c1: Month in M1	T	T	T	T	F	F	F	F
c2: Month in M2	T	T	F	F	T	T	F	F
c3: Month in M3	T	F	T	F	T	F	T	F
Rule count	1	1	1	1	1	1	1	1
a1: Impossible	X	X	X		X			X

Table 7.9 A Redundant Decision Table

Conditions	*1–4*	*5*	*6*	*7*	*8*	*9*
c1	T	F	F	F	F	T
c2	—	T	T	F	F	F
c3	—	T	F	T	F	F
a1	X	X	X	—	—	X
a2	—	X	X	X	—	—
a3	X	—	X	X	X	X

The ability to recognize (and develop) complete decision tables puts us in a powerful position with respect to redundancy and inconsistency. The decision table in Table 7.9 is redundant—three conditions and nine rules exist. (Rule 9 is identical to rule 4.) Notice that the action entries in rule 9 are identical to those in rules 1–4. As long as the actions in a redundant rule are identical to the corresponding part of the decision table, we do not have much of a problem. If the action entries are different, as in Table 7.10, we have a bigger problem.

If the decision table in Table 7.10 were to process a transaction in which c1 is true and both c2 and c3 are false, both rules 4 and 9 apply. We can make two observations:

1. Rules 4 and 9 are inconsistent.
2. The decision table is nondeterministic.

Table 7.10 An Inconsistent Decision Table

Conditions	*1–4*	*5*	*6*	*7*	*8*	*9*
c1	T	F	F	F	F	T
c2	—	T	T	F	F	F
c3	—	T	F	T	F	F
a1	X	X	X	—	—	—
a2	—	X	X	X	—	X
a3	X	—	X	X	X	—

Rules 4 and 9 are inconsistent because the action sets are different. The whole table is nondeterministic because there is no way to decide whether to apply rule 4 or rule 9. The bottom line for testers is that care should be taken when don't care entries are used in a decision table.

7.3 Test Cases for the Triangle Problem

Using the decision table in Table 7.3, we obtain 11 functional test cases: three impossible cases, three ways to fail the triangle property, one way to get an equilateral triangle, one way to get a scalene triangle, and three ways to get an isosceles triangle (see Table 7.11). We still need to provide

Table 7.11 Test Cases from Table 7.3

Case ID	a	b	c	*Expected Output*
DT1	4	1	2	Not a triangle
DT2	1	4	2	Not a triangle
DT3	1	2	4	Not a triangle
DT4	5	5	5	Equilateral
DT5	?	?	?	Impossible
DT6	?	?	?	Impossible
DT7	2	2	3	Isosceles
DT8	?	?	?	Impossible
DT9	2	3	2	Isosceles
DT10	3	2	2	Isosceles
DT11	3	4	5	Scalene

actual values for the variables in the conditions, but we cannot do this for the impossible rules. If we extended the decision table to show both ways to fail an inequality, we would pick up three more test cases (where one side is exactly the sum of the other two). Some judgment is required in this because of the exponential growth of rules. In this case, we would end up with many more don't care entries and more impossible rules.

7.4 Test Cases for the NextDate Function

The NextDate function was chosen because it illustrates the problem of dependencies in the input domain. This makes it a perfect example for decision table–based testing, because decision tables can highlight such dependencies. Recall that, in Chapter 6, we identified equivalence classes in the input domain of the NextDate function. One of the limitations we found in Chapter 6 was that indiscriminate selection of input values from the equivalence classes resulted in "strange" test cases, such as finding the next date to June 31, 1812. The problem stems from the presumption that the variables are independent. If they are, a Cartesian product of the classes makes sense. When logical dependencies exist among variables in the input domain, these dependencies are lost (suppressed is better) in a Cartesian product. The decision table format lets us emphasize such dependencies using the notion of the "impossible" action to denote impossible combinations of conditions (which are actually impossible rules). In this section, we will make three tries at a decision table formulation of the NextDate function.

7.4.1 First Try

Identifying appropriate conditions and actions presents an opportunity for craftsmanship. Suppose we start with a set of equivalence classes close to the one we used in Chapter 6.

M1 = {month: month has 30 days}
M2 = {month: month has 31 days}
M3 = {month: month is February}
D1 = {day: $1 \le$ day ≤ 28}
D2 = {day: day = 29}
D3 = {day: day = 30}
D4 = {day: day = 31}
Y1 = {year: year is a leap year}
Y2 = {year: year is not a leap year}

If we wish to highlight impossible combinations, we could make a limited entry decision table with the following conditions and actions. (Note that the equivalence classes for the year variable collapse into one condition in Table 7.12.)

This decision table will have 256 rules, many of which will be impossible. If we wanted to show why these rules were impossible, we might revise our actions to the following:

a1: Day invalid for this month
a2: Cannot happen in a non-leap year
a3: Compute the next date

Table 7.12 First Try Decision Table with 256 Rules

Conditions											
c1: Month in M1?	T										
c2: Month in M2?		T									
c3: Month in M3?			T								
c4: Day in D1?											
c5: Day in D2?											
c6: Day in D3?											
c7: Day in D4?											
c8: Year in Y1?											
a1: Impossible											
a2: Next date											

7.4.2 Second Try

If we focus on the leap year aspect of the NextDate function, we could use the set of equivalence classes as they were in Chapter 6. These classes have a Cartesian product that contains 36 triples, with several that are impossible.

To illustrate another decision table technique, this time we will develop an extended entry decision table, and we will take a closer look at the action stub. In making an extended entry decision table, we must ensure that the equivalence classes form a true partition of the input domain. (Recall from Chapter 3 that a partition is a set of disjoint subsets where the union is the entire set.) If there were any “overlaps” among the rule entries, we would have a redundant case in which more than one rule could be satisfied. Here, Y2 is the set of years between 1812 and 2012, evenly divisible by four excluding the year 2000.

M1 = {month: month has 30 days}
M2 = {month: month has 31 days}
M3 = {month: month is February}
D1 = {day: $1 \le$ day ≤ 28}
D2 = {day: day = 29}
D3 = {day: day = 30}
D4 = {day: day = 31}
Y1 = {year: year = 2000}
Y2 = {year: year is a non-century leap year}
Y3 = {year: year is a common year}

In a sense, we could argue that we have a “gray box” technique, because we take a closer look at the NextDate problem statement. To produce the next date of a given date, only five possible actions are needed: incrementing and resetting the day and month, and incrementing the year.

(We will not let time go backward by resetting the year.) To follow the metaphor, we still cannot see inside the implementation box—the implementation could be a table look-up.

These conditions would result in a decision table with 36 rules that correspond to the Cartesian product of the equivalence classes. Combining rules with don't care entries yields the decision table in Table 7.13, which has 16 rules. We still have the problem with logically impossible rules, but this formulation helps us identify the expected outputs of a test case. If you complete the action entries in this table, you will find some cumbersome problems with December (in rule 8) and other problems with Feb. 28 in rules 9, 11, and 12. We fix these next.

Table 7.13 Second Try Decision Table with 36 Rules

	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*
c1: Month in	M1	M1	M1	M1	M2	M2	M2	M2
c2: Day in	D1	D2	D3	D4	D1	D2	D3	D4
c3: Year in	—	—	—	—	—	—	—	—
Rule count	3	3	3	3	3	3	3	3
Actions								
a1: Impossible				X				
a2: Increment day	X	X			X	X	X	
a3: Reset day			X					X
a4: Increment month			X					?
a5: Reset month								?
a6: Increment year								?
	9	*10*	*11*	*12*	*13*	*14*	*15*	*16*
c1: Month in	M3	M3	M3	M3	M3	M3	M3	M3
c2: Day in	D1	D1	D1	D2	D2	D2	D3	D4
c3: Year in	Y1	Y2	Y3	Y1	Y2	Y3	—	—
Rule count	1	1	1	1	1	1	3	3
Actions								
a1: Impossible						X	X	X
a2: Increment day	X	X	?					
a3: Reset day			?	X	X			
a4: Increment month	X		X	X	X			
a5: Reset month								
a6: Increment year								

7.4.3 Third Try

We can clear up the end-of-year considerations with a third set of equivalence classes. This time, we are very specific about days and months, and we revert to the simpler leap year or non-leap year condition of the first try—so the year 2000 gets no special attention. (We could do a fourth try, showing year equivalence classes as in the second try, but by now you get the point.)

M1 = {month: month has 30 days}
M2 = {month: month has 31 days except December}
M3 = {month: month is December}
M4 = {month: month is February}
D1 = {day: 1 ≤ day ≤ 27}
D2 = {day: day = 28}
D3 = {day: day = 29}
D4 = {day: day = 30}
D5 = {day: day = 31}
Y1 = {year: year is a leap year}
Y2 = {year: year is a common year}

The Cartesian product of these contains 40 elements. The result of combining rules with don't care entries is given in Table 7.14; it has 22 rules, compared with the 36 of the second try. Recall from Chapter 1 the question of whether a large set of test cases is necessarily better than a smaller set. Here, we have a 22-rule decision table that gives a clearer picture of the NextDate function than does the 36-rule decision table. The first five rules deal with 30-day months; notice that the leap year considerations are irrelevant. The next two sets of rules (6–15) deal with 31-day months, where rules 6–10 deal with months other than December and rules 11–15 deal with December. No impossible rules are listed in this portion of the decision table, although there is some redundancy that an efficient tester might question. Eight of the 10 rules simply increment the day. Would we really require eight separate test cases for this subfunction? Probably not; but note the insights we can get from the decision table. Finally, the last seven rules focus on February in common and leap years.

The decision table in Table 7.14 is the basis for the source code for the NextDate function in Chapter 2. As an aside, this example shows how good testing can improve programming. All the decision table analysis could have been done during the detailed design of the NextDate function.

We can use the algebra of decision tables to further simplify these 22 test cases. If the action sets of two rules in a limited entry decision table are identical, there must be at least one condition that allows two rules to be combined with a don't care entry. This is the decision table equivalent of the "treated the same" guideline that we used to identify equivalence classes. In a sense, we are identifying equivalence classes of rules. For example, rules 1, 2, and 3 involve day classes D1, D2, and D3 for 30-day months. These can be combined similarly for day classes D1, D2, D3, and D4 in the 31-day month rules, and D4 and D5 for February. The result is in Table 7.15.

The corresponding test cases are shown in Table 7.16.

Table 7.14 Decision Table for NextDate Function

	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*		
c1: Month in	M1	M1	M1	M1	M1	M2	M2	M2	M2	M2		
c2: Day in	D1	D2	D3	D4	D5	D1	D2	D3	D4	D5		
c3: Year in	—	—	—	—	—	—	—	—	—	—		
Actions												
a1: Impossible					X							
a2: Increment day	X	X	X			X	X	X	X			
a3: Reset day				X						X		
a4: Increment month				X						X		
a5: Reset month												
a6: Increment year												
	11	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*	*20*	*21*	*22*
c1: Month in	M3	M3	M3	M3	M3	M4	M4	M4	M4	M4	M4	M4
c2: Day in	D1	D2	D3	D4	D5	D1	D2	D2	D3	D3	D4	D5
c3: Year in	—	—	—	—	—	—	Y1	Y2	Y1	Y2	—	—
Actions												
a1: Impossible										X	X	X
a2: Increment day	X	X	X	X		X	X					
a3: Reset day					X			X	X			
a4: Increment month								X	X			
a5: Reset month					X							
a6: Increment year					X							

7.5 Test Cases for the Commission Problem

The commission problem is not well served by a decision table analysis. This is not surprising because very little decisional logic is used in the problem. Because the variables in the equivalence classes are truly independent, no impossible rules will occur in a decision table in which conditions correspond to the equivalence classes. Thus, we will have the same test cases as we did for equivalence class testing.

Table 7.15 Reduced Decision Table for NextDate Function

	1–3	*4*	*5*	*6–9*	*10*	*11–14*	*15*	*16*	*17*	*18*	*19*	*20*	*21, 22*
c1: Month in	M1	M1	M1	M2	M2	M3	M3	M4	M4	M4	M4	M4	M4
c2: Day in	D1, D2, D3	D4	D5	D1, D2, D3, D4	D5	D1, D2, D3, D4	D5	D1	D2	D2	D3	D3	D4, D5
c3: Year in	—	—	—	—	—	—	—	—	Y1	Y2	Y1	Y2	—
Actions													
a1: Impossible			X									X	X
a2: Increment day	X			X		X		X	X				
a3: Reset day		X			X		X			X	X		
a4: Increment month		X			X					X	X		
a5: Reset month							X						
a6: Increment year							X						

7.6 Cause-and-Effect Graphing

In the early years of computing, the software community borrowed many ideas from the hardware community. In some cases this worked well, but in others, the problems of software just did not fit well with established hardware techniques. Cause-and-effect graphing is a good example of this. The base hardware concept was the practice of describing circuits composed of discrete components with AND, OR, and NOT gates. There was usually an input side of a circuit diagram, and

Table 7.16 Decision Table Test Cases for NextDate

Case ID	*Month*	*Day*	*Year*	*Expected Output*
1–3	4	15	2001	4/16/2001
4	4	30	2001	5/1/2001
5	4	31	2001	Invalid input date
6–9	1	15	2001	1/16/2001
10	1	31	2001	2/1/2001
11–14	12	15	2001	12/16/2001
15	12	31	2001	1/1/2002
16	2	15	2001	2/16/2001
17	2	28	2004	2/29/2004
18	2	28	2001	3/1/2001
19	2	29	2004	3/1/2004
20	2	29	2001	Invalid input date
21, 22	2	30	2001	Invalid input date

the flow of inputs through the various components could be generally traced from left to right. With this, the effects of hardware faults such as stuck-at-one/zero could be traced to the output side. This greatly facilitated circuit testing.

Cause-and-effect graphs attempt to follow this pattern, by showing unit inputs on the left side of a drawing, and using AND, OR, and NOT “gates” to express the flow of data across stages of a unit. Figure 7.1 shows the basic cause-and-effect graph structures. The basic structures can be augmented by less used operations: Identity, Masks, Requires, and Only One.

The most that can be learned from a cause-and-effect graph is that, if there is a problem at an output, the path(s) back to the inputs that affected the output can be retraced. There is little support for actually identifying test cases. Figure 7.2 shows a cause-and-effect graph for the commission problem.

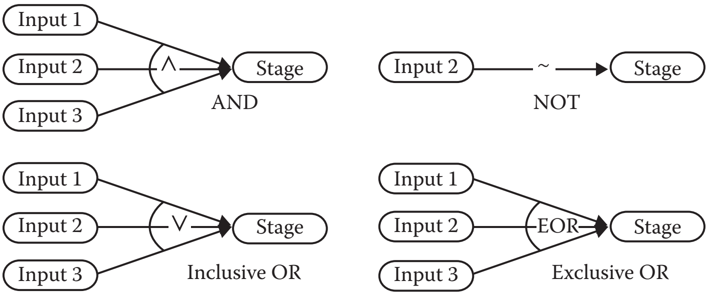


Figure 7.1 Cause-and-effect graphing operations.

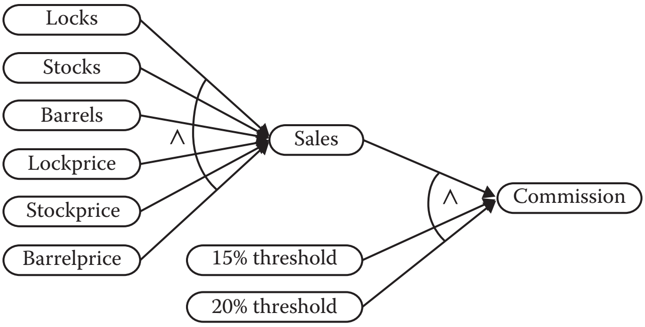


Figure 7.2 Cause-and-effect graph for commission problem.

7.7 Guidelines and Observations

As with the other testing techniques, decision table–based testing works well for some applications (such as NextDate) and is not worth the trouble for others (such as the commission problem). Not surprisingly, the situations in which it works well are those in which a lot of decision making takes place (such as the triangle problem), and those in which important logical relationships exist among input variables (the NextDate function).

1. The decision table technique is indicated for applications characterized by any of the following:
   a. Prominent if–then–else logic
   b. Logical relationships among input variables
   c. Calculations involving subsets of the input variables
   d. Cause-and-effect relationships between inputs and outputs
   e. High cyclomatic complexity (see Chapter 9)
2. Decision tables do not scale up very well (a limited entry table with n conditions has 2^n rules). There are several ways to deal with this—use extended entry decision tables, algebraically simplify tables, “factor” large tables into smaller ones, and look for repeating patterns of condition entries. Try factoring the extended entry table for NextDate (Table 7.14).
3. As with other techniques, iteration helps. The first set of conditions and actions you identify may be unsatisfactory. Use it as a stepping stone and gradually improve on it until you are satisfied with a decision table.

EXERCISES

1. Develop a decision table and additional test cases for the right triangle addition to the triangle problem (see Chapter 2 exercises). Note that there can be isosceles right triangles, but not with integer sides.
2. Develop a decision table for the “second try” at the NextDate function. At the end of a 31-day month, the day is always reset to 1. For all non-December months, the month is incremented; and for December, the month is reset to January, and the year is incremented.
3. Develop a decision table for the YesterDate function (see Chapter 2 exercises).
4. Expand the commission problem to consider “violations” of the sales limits. Develop the corresponding decision tables and test cases for a “company friendly” version and a “salesperson friendly” version.

5. Discuss how well decision table testing deals with the multiple fault assumption.
6. Develop decision table test cases for the time change problem (Chapter 6, problem 5).
7. If you did exercise 8 in Chapter 2, exercise 5 in Chapter 5, and exercise 6 in Chapter 6, you are already familiar with the CRC Press website for downloads (http://www.crcpress.com/product/isbn/9781466560680). There you will find an Excel spreadsheet named specBasedTesting.xls. (It is an extended version of Naive.xls, and it contains the same inserted faults.) Different sheets contain decision table–based test cases for the triangle, NextDate, and commission problems, respectively. Run these sets of test cases and compare the results with your naive testing from Chapter 2, your boundary value testing from Chapter 5, and your equivalence class testing from Chapter 6.
8. The retirement pension salary of a Michigan public school teacher is a percentage of the average of their last 3 years of teaching. Normally, the number of years of teaching service is the percentage multiplier. To encourage senior teachers to retire early, the Michigan legislature enacted the following incentive in May of 2010:

    Teachers must apply for the incentive before June 11, 2010. Teachers who are currently eligible to retire (age ≥ 63 years) shall have a multiplier of 1.6% on their salary up to, and including, $90,000, and 1.5% on compensation in excess of $90,000. Teachers who meet the 80 total years of age plus years of teaching shall have a multiplier of 1.55% on their salary up to, and including, $90,000 and 1.5% on compensation in excess of $90,000.

    Make a decision table to describe the retirement pension policy; be sure to consider the retirement eligibility criteria carefully. What are the compensation multipliers for a person who is currently 64 with 20 years of teaching whose salary is $95,000?

References

Elmendorf, W.R., *Cause–Effect Graphs in Functional Testing,* IBM System Development Division, Poughkeepsie, NY, TR-00.2487, 1973.

Mosley, D.J., *The Handbook of MIS Application Software Testing,* Yourdon Press, Prentice Hall, Englewood Cliffs, NJ, 1993.

Myers, G.J., *The Art of Software Testing,* Wiley Interscience, New York, 1979.

Chapter 8

Path Testing

The distinguishing characteristic of code-based testing methods is that, as the name implies, they are all based on the source code of the program tested, and not on the specification. Because of this absolute basis, code-based testing methods are very amenable to rigorous definitions, mathematical analysis, and useful measurement. In this chapter, we examine the two most common forms of path testing. The technology behind these has been available since the mid-1970s, and the originators of these methods now have companies that market very successful tools that implement the techniques. Both techniques start with the program graph; we repeat the improved definition from Chapter 4 here.

8.1 Program Graphs

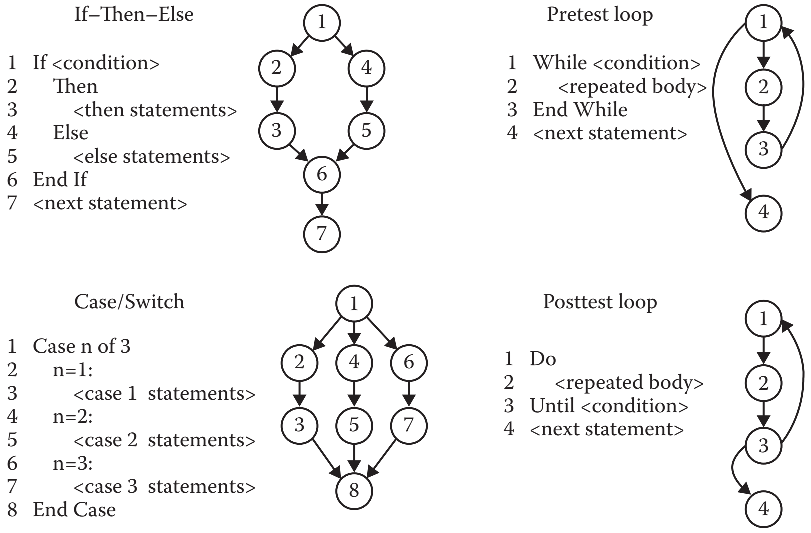
Definition

Given a program written in an imperative programming language, its *program graph* is a directed graph in which nodes are statement fragments, and edges represent flow of control. (A complete statement is a "default" statement fragment.)

If i and j are nodes in the program graph, an edge exists from node i to node j if and only if the statement fragment corresponding to node j can be executed immediately after the statement fragment corresponding to node i.

8.1.1 Style Choices for Program Graphs

Deriving a program graph from a given program is an easy process. It is illustrated here with four of the basic structured programming constructs (Figure 8.1), and also with our pseudocode implementation of the triangle program from Chapter 2. Line numbers refer to statements and statement fragments. An element of judgment can be used here: sometimes it is convenient to keep



If–Then–Else

```
1  If <condition>
2     Then
3        <then statements>
4     Else
5        <else statements>
6  End If
7  <next statement>
```

Pretest loop

```
1  While <condition>
2        <repeated body>
3  End While
4  <next statement>
```

Case/Switch

```
1  Case n of 3
2     n=1:
3        <case 1  statements>
4     n=2:
5        <case 2  statements>
6     n=3:
7        <case 3  statements>
8  End Case
```

Posttest loop

```
1  Do
2        <repeated body>
3  Until <condition>
4  <next statement>
```

Figure 8.1 Program graphs of four structured programming constructs.

a fragment as a separate node; other times it seems better to include this with another portion of a statement. For example, in Figure 8.2, line 14 could be split into two lines:

```
14      Then If (a = b) AND (b = c)
14a         Then
14b            If (a = b) AND (b = c)
```

This latitude collapses onto a unique DD-path graph, so the differences introduced by differing judgments are moot. (A mathematician would make the point that, for a given program, several distinct program graphs might be used, all of which reduce to a unique DD-path graph.) We also need to decide whether to associate nodes with nonexecutable statements such as variable and type declarations; here we do not. A program graph of the second version of the triangle problem (see Chapter 2) is given in Figure 8.2.

Nodes 4 through 8 are a sequence, nodes 9 through 12 are an if–then–else construct, and nodes 13 through 22 are nested if–then–else constructs. Nodes 4 and 23 are the program source and sink nodes, corresponding to the single entry, single-exit criteria. No loops exist, so this is a directed acyclic graph. The importance of the program graph is that program executions correspond to paths from the source to the sink nodes. Because test cases force the execution of some such program path, we now have a very explicit description of the relationship between a test case and the part of the program it exercises. We also have an elegant, theoretically respectable way to deal with the potentially large number of execution paths in a program.

There are detractors of path-based testing. Figure 8.3 is a graph of a simple (but unstructured!) program; it is typical of the kind of example detractors use to show the (practical) impossibility of completely testing even simple programs. (This example first appeared in Schach [1993].) In this program, five paths lead from node B to node F in the interior of the loop. If the loop may have up to 18 repetitions, some 4.77 trillion distinct program execution paths exist. (Actually, it

```
1  Program triangle2
2  Dim a,b,c As Integer
3  Dim IsATrinagle As Boolean
4  Output("Enter 3 integers which are sides of a triangle")
5  Input(a,b,c)
6  Output("Side A is", a)
7  Output("Side B is", b)
8  Output("Side C is", c)
9  If (a < b + c) AND (b < a + c) AND (c < a + b)
10   Then IsATriangle = True
11   Else IsATriangle = False
12  EndIf
13  If IsATriangle
14   Then  If (a = b) AND (b = c)
15           Then Output ("Equilateral")
16           Else  If (a≠b) AND (a≠c) AND (b≠c)
17                   Then Output ("Scalene")
18                   Else Output ("Isosceles")
19                EndIf
20           EndIf
21   Else  Output("Nota a Triangle")
22  EndIf
23  End triangle2
```

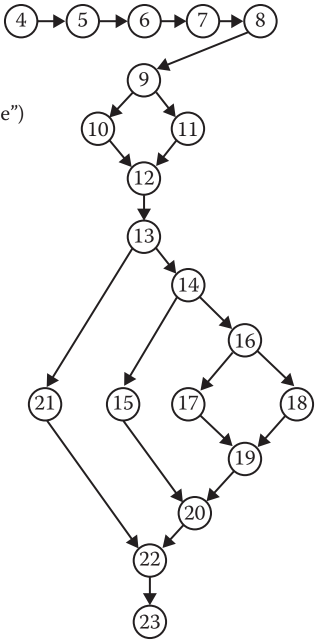


Figure 8.2 Program graph of triangle program.

is 4,768,371,582,030 paths.) The detractor's argument is a good example of the logical fallacy of extension—take a situation, extend it to an extreme, show that the extreme supports your point, and then apply it back to the original question. The detractors miss the point of code-based testing—later in this chapter, we will see how this enormous number can be reduced, with good reasons, to a more manageable size.

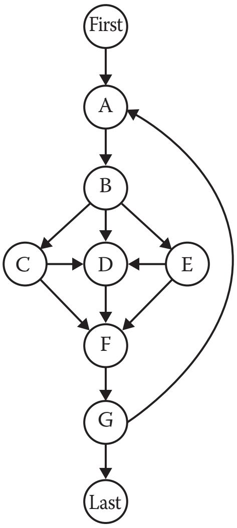


Figure 8.3 Trillions of paths.

8.2 DD-Paths

The best-known form of code-based testing is based on a construct known as a decision-to-decision path (DD-path) (Miller, 1977). The name refers to a sequence of statements that, in Miller's words, begins with the "outway" of a decision statement and ends with the "inway" of the next decision statement. No internal branches occur in such a sequence, so the corresponding code is like a row of dominoes lined up so that when the first falls, all the rest in the sequence fall. Miller's original definition works well for second-generation languages like FORTRAN II, because decision-making statements (such as arithmetic IFs and DO loops) use statement labels to refer to target statements. With modern languages (e.g., Pascal, Ada®, C, Visual Basic, Java), the notion of statement fragments resolves the difficulty of applying Miller's original definition. Otherwise, we end up with program graphs in which some statements are members of more than one DD-path. In the ISTQB literature, and also in Great Britain, the DD-path concept is known as a "linear code sequence and jump" and is abbreviated by the acronym LCSAJ. Same idea, longer name.

We will define DD-paths in terms of paths of nodes in a program graph. In graph theory, these paths are called chains, where a chain is a path in which the initial and terminal nodes are distinct, and every interior node has indegree = 1 and outdegree = 1. (See Chapter 4 for a formal definition.) Notice that the initial node is 2-connected to every other node in the chain, and no instances of 1- or 3-connected nodes occur, as shown in Figure 8.4. The length (number of edges) of the chain in Figure 8.4 is 6.

Definition

A *DD-path* is a sequence of nodes in a program graph such that

Case 1: It consists of a single node with indeg = 0.
Case 2: It consists of a single node with outdeg = 0.
Case 3: It consists of a single node with indeg ≥ 2 or outdeg ≥ 2.
Case 4: It consists of a single node with indeg = 1 and outdeg = 1.
Case 5: It is a maximal chain of length ≥ 1.

Cases 1 and 2 establish the unique source and sink nodes of the program graph of a structured program as initial and final DD-paths. Case 3 deals with complex nodes; it assures that no node is contained in more than one DD-path. Case 4 is needed for "short branches"; it also preserves the one-fragment, one DD-path principle. Case 5 is the "normal case," in which a DD-path is a single entry, single-exit sequence of nodes (a chain). The "maximal" part of the case 5 definition is used to determine the final node of a normal (nontrivial) chain.

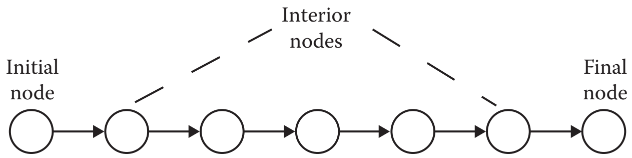


Figure 8.4 Chain of nodes in a directed graph.

Definition

Given a program written in an imperative language, its *DD-path graph* is the directed graph in which nodes are DD-paths of its program graph, and edges represent control flow between successor DD-paths.

This is a complex definition, so we will apply it to the program graph in Figure 8.2. Node 4 is a case 1 DD-path; we will call it "first." Similarly, node 23 is a case 2 DD-path, and we will call it "last." Nodes 5 through 8 are case 5 DD-paths. We know that node 8 is the last node in this DD-path because it is the last node that preserves the 2-connectedness property of the chain. If we go beyond node 8 to include node 9, we violate the indegree = outdegree = 1 criterion of a chain. If we stop at node 7, we violate the "maximal" criterion. Nodes 10, 11, 15, 17, 18, and 21 are case 4 DD-paths. Nodes 9, 12, 13, 14, 16, 19, 20, and 22 are case 3 DD-paths. Finally, node 23 is a case 2 DD-path. All this is summarized in Figure 8.5.

In effect, the DD-path graph is a form of condensation graph (see Chapter 4); in this condensation, 2-connected components are collapsed into individual nodes that correspond to case 5 DD-paths. The single-node DD-paths (corresponding to cases 1–4) are required to preserve the convention that a statement (or statement fragment) is in exactly one DD-path. Without this convention, we end up with rather clumsy DD-path graphs, in which some statement fragments are in several DD-paths.

This process should not intimidate testers—high-quality commercial tools are available, which generate the DD-path graph of a given program. The vendors make sure that their products work for a wide variety of programming languages. In practice, it is reasonable to manually create

Figure 8.2 Nodes	DD-Path	Case of definition
4	First	1
5-8	A	5
9	B	3
10	C	4
11	D	4
12	E	3
13	F	3
14	H	3
15	I	4
16	J	3
17	K	4
18	L	4
19	M	3
20	N	3
21	G	4
22	O	3
23	Last	2

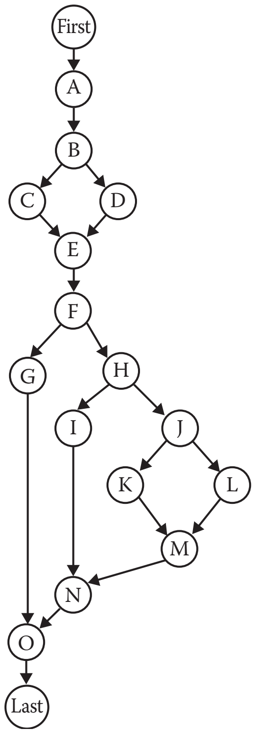


Figure 8.5 DD-path graph for triangle program.

DD-path graphs for programs up to about 100 source lines. Beyond that, most testers look for a tool.

Part of the confusion with this example is that the triangle problem is logic intensive and computationally sparse. This combination yields many short DD-paths. If the THEN and ELSE clauses contained blocks of computational statements, we would have longer chains, as we will see in the commission problem.

8.3 Test Coverage Metrics

The *raison d'être* of DD-paths is that they enable very precise descriptions of test coverage. Recall (from Chapters 5 through 7) that one of the fundamental limitations of specification-based testing is that it is impossible to know either the extent of redundancy or the possibility of gaps corresponding to the way a set of functional test cases exercises a program. Back in Chapter 1, we had a Venn diagram showing relationships among specified, programmed, and tested behaviors. Test coverage metrics are a device to measure the extent to which a set of test cases covers (or exercises) a program.

8.3.1 Program Graph–Based Coverage Metrics

Given a program graph, we can define the following set of test coverage metrics. We will use them to relate to other published sets of coverage metrics.

Definition

Given a set of test cases for a program, they constitute *node coverage* if, when executed on the program, every node in the program graph is traversed. Denote this level of coverage as G_{node}, where the G stands for program graph.

Since nodes correspond to statement fragments, this guarantees that every statement fragment is executed by some test case. If we are careful about defining statement fragment nodes, this also guarantees that statement fragments that are outcomes of a decision-making statement are executed.

Definition

Given a set of test cases for a program, they constitute *edge coverage* if, when executed on the program, every edge in the program graph is traversed. Denote this level of coverage as G_{edge}.

The difference between G_{node} and G_{edge} is that, in the latter, we are assured that all outcomes of a decision-making statement are executed. In our triangle problem (see Figure 8.2), nodes 9, 10, 11, and 12 are a complete if–then–else statement. If we required nodes to correspond to full statements, we could execute just one of the decision alternatives and satisfy the statement coverage criterion. Because we allow statement fragments, it is natural to divide such a statement into separate nodes (the condition test, the true outcome, and the false outcome). Doing so results in predicate outcome coverage. Whether or not our convention is followed, these coverage metrics require that we find a set of test cases such that, when executed, every node of the program graph is traversed at least once.

Definition

Given a set of test cases for a program, they constitute *chain coverage* if, when executed on the program, every chain of length greater than or equal to 2 in the program graph is traversed. Denote this level of coverage as G_{chain}.

The G_{chain} coverage is the same as node coverage in the DD-path graph that corresponds to the given program graph. Since DD-paths are important in E.F. Miller's original formulation of test covers (defined in Section 8.3.2), we now have a clear connection between purely program graph constructs and Miller's test covers.

Definition

Given a set of test cases for a program, they constitute *path coverage* if, when executed on the program, every path from the source node to the sink node in the program graph is traversed. Denote this level of coverage as G_{path}.

This coverage is open to severe limitations when there are loops in a program (as in Figure 8.3). E.F. Miller partially anticipated this when he postulated the C_2 metric for loop coverage. Referring back to Chapter 4, observe that every loop in a program graph represents a set of strongly (3-connected) nodes. To deal with the size implications of loops, we simply exercise every loop, and then form the condensation graph of the original program graph, which must be a directed acyclic graph.

8.3.2 E.F. Miller's Coverage Metrics

Several widely accepted test coverage metrics are used; most of those in Table 8.1 are due to the early work of Miller (1977). Having an organized view of the extent to which a program is tested makes it possible to sensibly manage the testing process. Most quality organizations now expect the C_1 metric (DD-path coverage) as the minimum acceptable level of test coverage.

These coverage metrics form a lattice (see Chapter 9 for a lattice of data flow coverage metrics) in which some are equivalent and some are implied by others. The importance of the lattice is that there are always fault types that can be revealed at one level and can escape detection by inferior levels of testing. Miller (1991) observes that when DD-path coverage is attained by a set of test cases, roughly 85% of all faults are revealed. The test coverage metrics in Table 8.1 tell us what to test but not how to test it. In this section, we take a closer look at techniques that exercise source code. We must keep an important distinction in mind: Miller's test coverage metrics are based on program graphs in which nodes are full statements, whereas our formulation allows statement fragments (which can be entire statements) to be nodes.

8.3.2.1 Statement Testing

Because our formulation of program graphs allows statement fragments to be individual nodes, Miller's C_0 metric is subsumed by our G_{node} metric.

Table 8.1 Miller's Test Coverage Metrics

Metric	*Description of Coverage*
C_0	Every statement
C_1	Every DD-path
C_{1p}	Every predicate to each outcome
C_2	C_1 coverage + loop coverage
C_d	C_1 coverage + every dependent pair of DD-paths
C_{MCC}	Multiple condition coverage
C_{ik}	Every program path that contains up to k repetitions of a loop (usually $k = 2$)
C_{stat}	"Statistically significant" fraction of paths
C_∞	All possible execution paths

Statement coverage is generally viewed as the bare minimum. If some statements have not been executed by the set of test cases, there is clearly a severe gap in the test coverage. Although less adequate than DD-path coverage, the statement coverage metric (C_0) is still widely accepted: it is mandated by ANSI (American National Standards Institute) Standard 187B and has been used successfully throughout IBM since the mid-1970s.

8.3.2.2 DD-Path Testing

When every DD-path is traversed (the C_1 metric), we know that each predicate outcome has been executed; this amounts to traversing every edge in the DD-path graph (or program graph). Therefore, the C_1 metric is exactly our G_{chain} metric.

For if–then and if–then–else statements, this means that both the true and the false branches are covered (C_{1p} coverage). For CASE statements, each clause is covered. Beyond this, it is useful to ask how we might test a DD-path. Longer DD-paths generally represent complex computations, which we can rightly consider as individual functions. For such DD-paths, it may be appropriate to apply a number of functional tests, especially those for boundary and special values.

8.3.2.3 Simple Loop Coverage

The C_2 metric requires DD-path coverage (the C_1 metric) plus loop testing.

The simple view of loop testing is that every loop involves a decision, and we need to test both outcomes of the decision: one is to traverse the loop, and the other is to exit (or not enter) the loop. This is carefully proved in Huang (1979). Notice that this is equivalent to the G_{edge} test coverage.

8.3.2.4 Predicate Outcome Testing

This level of testing requires that every outcome of a decision (predicate) must be exercised. Because our formulation of program graphs allows statement fragments to be individual nodes, Miller's C_{1p} metric is subsumed by our G_{edge} metric. Neither E.F. Miller's test covers nor the graph-based covers deal with decisions that are made on compound conditions. They are the subjects of Section 8.3.3.

8.3.2.5 Dependent Pairs of DD-Paths

Identification of dependencies must be made at the code level. This cannot be done just by considering program graphs. The C_d metric foreshadows the topic of Chapter 9—data flow testing. The most common dependency among pairs of DD-paths is the define/reference relationship, in which a variable is defined (receives a value) in one DD-path and is referenced in another DD-path. The importance of these dependencies is that they are closely related to the problem of infeasible paths. We have good examples of dependent pairs of DD-paths: in Figure 8.5, C and H are such a pair, as are DD-paths D and H. The variable IsATriangle is set to TRUE at node C, and FALSE at node D. Node H is the branch taken when IsATriangle is TRUE win the condition at node F. Any path containing nodes D and H is infeasible. Simple DD-path coverage might not exercise these dependencies; thus, a deeper class of faults would not be revealed.

8.3.2.6 Complex Loop Coverage

Miller's C_{ik} metric extends the loop coverage metric to include full paths from source to sink nodes that contain loops.

The condensation graphs we studied in Chapter 4 provide us with an elegant resolution to the problems of testing loops. Loop testing has been studied extensively, and with good reason—loops are a highly fault-prone portion of source code. To start, an amusing taxonomy of loops occurs (Beizer, 1984): concatenated, nested, and horrible, shown in Figure 8.6.

Concatenated loops are simply a sequence of disjoint loops, while nested loops are such that one is contained inside another. Knotted (Beizer calls them "horrible") loops cannot occur when the structured programming precepts are followed, but they can occur in languages like Java with try/catch. When it is possible to branch into (or out from) the middle of a loop, and these branches

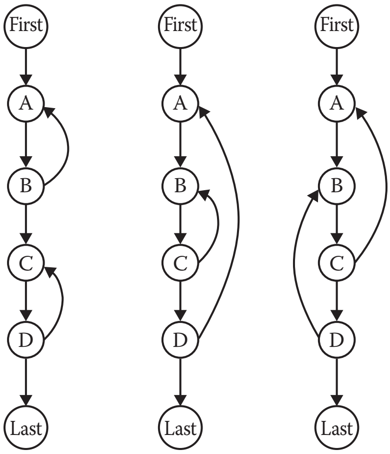


Figure 8.6 Concatenated, nested, and knotted loops.

are internal to other loops, the result is Beizer's knotted loop. We can also take a modified boundary value approach, where the loop index is given its minimum, nominal, and maximum values (see Chapter 5). We can push this further to full boundary value testing and even robustness testing. If the body of a simple loop is a DD-path that performs a complex calculation, this should also be tested, as discussed previously. Once a loop has been tested, the tester condenses it into a single node. If loops are nested, this process is repeated starting with the innermost loop and working outward. This results in the same multiplicity of test cases we found with boundary value analysis, which makes sense, because each loop index variable acts like an input variable. If loops are knotted, it will be necessary to carefully analyze them in terms of the data flow methods discussed in Chapter 9. As a preview, consider the infinite loop that could occur if one loop tampers with the value of the other loop's index.

8.3.2.7 Multiple Condition Coverage

Miller's C_{MCC} metric addresses the question of testing decisions made by compound conditions. Look closely at the compound conditions in DD-paths B and H. Instead of simply traversing such predicates to their true and false outcomes, we should investigate the different ways that each outcome can occur. One possibility is to make a decision table; a compound condition of three simple conditions will have eight rules (see Table 8.2), yielding eight test cases. Another possibility is to reprogram compound predicates into nested simple if–then–else logic, which will result in more DD-paths to cover. We see an interesting tradeoff: statement complexity versus path complexity. Multiple condition coverage assures that this complexity is not swept under the DD-path coverage rug. This metric has been refined to Modified Condition Decision Coverage (MCDC), defined in Section 8.3.3.

8.3.2.8 "Statistically Significant" Coverage

The C_{stat} metric is awkward—what constitutes a statistically significant set of full program paths? Maybe this refers to a comfort level on the part of the customer/user.

8.3.2.9 All Possible Paths Coverage

The subscript in Miller's C_∞ metric says it all—this can be enormous for programs with loops, *a la* Figure 8.3. This can make sense for programs without loops, and also for programs for which loop testing reduces the program graph to its condensation graph.

8.3.3 A Closer Look at Compound Conditions

There is an excellent reference (Chilenski, 2001) that is 214 pages long and is available on the Web. The definitions in this subsection are derived from this reference. They will be related to the definitions in Sections 8.3.1 and 8.3.2.

8.3.3.1 Boolean Expression (per Chilenski)

"A *Boolean expression* evaluates to one of two possible (Boolean) outcomes traditionally known as False and True."

A Boolean expression may be a simple Boolean variable, or a compound expression containing one or more Boolean operators. Chilenski clarifies Boolean operators into four categories:

Operator Type	*Boolean Operators*
Unary (single operand)	NOT(~),
Binary (two operands)	AND(∧), OR(∨), XOR(⊕)
Short circuit operators	AND (AND–THEN), OR (OR–ELSE)
Relational operators	=, ≠, <, ≤, >, ≥

In mathematical logic, Boolean expressions are known as *logical expressions*, where a logical expression can be

1. A simple proposition that contains no logical connective
2. A compound proposition that contains at least one logical connective

Synonyms: *predicate, proposition, condition*.

In programming languages, Chilenski's Boolean expressions appear as conditions in decision making statements: If–Then, If–Then–Else, If–ElseIf, Case/Switch, For, While, and Until loops. This subsection is concerned with the testing needed for compound conditions. Compound conditions are shown as single nodes in a program graph; hence, the complexity they introduce is obscured.

8.3.3.2 Condition (per Chilenski)

"A *condition* is an operand of a Boolean operator (Boolean functions, objects and operators).

Generally this refers to the lowest level conditions (i.e., those operands that are not Boolean operators themselves), which are normally the leaves of an expression tree. Note that a condition is a Boolean (sub)expression."

In mathematical logic, Chilenski's conditions are known as simple, or atomic, propositions. Propositions can be simple or compound, where a compound proposition contains at least one logical connective. Propositions are also called predicates, the term that E.F. Miller uses.

8.3.3.3 Coupled Conditions (per Chilenski)

Two (or more) conditions are *coupled* if changing one also changes the other(s).

When conditions are coupled, it may not be possible to vary individual conditions, because the coupled condition(s) might also change. Chelinski notes that conditions can be strongly or weakly coupled. In a strongly coupled pair, changing one condition always changes the other. In a weakly coupled triplet, changing one condition may change one other coupled condition, but not the third one. Chelinski offers these examples:

In ((($x = 0$) AND A) OR (($x \neq 0$) AND B)), the conditions ($x = 0$) and ($x \neq 0$) are strongly coupled.

In (($x = 1$) OR ($x = 2$) OR ($x = 3$)), the three conditions are weakly coupled.

8.3.3.4 Masking Conditions (per Chilenski)

"The process *masking conditions* involves of setting the one operand of an operator to a value such that changing the other operand of that operator does not change the value of the operator.

Referring to Chapter 3.4.3, masking uses the Domination Laws. For an AND operator, masking of one operand can be achieved by holding the other operand False.

(X AND False = False AND X = False no matter what the value of X is.)

For an OR operator, masking of one operand can be achieved by holding the other operand True.

(X OR True = True OR X = True no matter what the value of X is.)."

8.3.3.5 Modified Condition Decision Coverage

MCDC is required for "Level A" software by testing standard DO-178B. MCDC has three variations: Masking MCDC, Unique-Cause MCDC, and Unique-Cause + Masking MCDC. These are explained in exhaustive detail in Chilenski (2001), which concludes that Masking MCDC, while demonstrably the weakest form of the three, is recommended for compliance with DO-178B. The definitions below are quoted from Chilenski.

Definition

MCDC requires

1. Every statement must be executed at least once.
2. Every program entry point and exit point must be invoked at least once.
3. All possible outcomes of every control statement are taken at least once.
4. Every nonconstant Boolean expression has been evaluated to both true and false outcomes.
5. Every nonconstant condition in a Boolean expression has been evaluated to both true and false outcomes.
6. Every nonconstant condition in a Boolean expression has been shown to independently affect the outcomes (of the expression).

The basic definition of MCDC needs some explanation. Control statements are those that make decisions, such as If statements, Case/Switch statements, and looping statements. In a program graph, control statements have an outdegree greater than 1. Constant Boolean expressions are those that always evaluate to the same end value. For example, the Boolean expression $(p \vee \sim p)$ always evaluates to True, as does the condition $(a = a)$. Similarly, $(p \wedge \sim p)$ and $(a \neq a)$ are constant expressions (that evaluate to False). In terms of program graphs, MCDC requirements 1 and 2 translate to node coverage, and MCDC requirements 3 and 4 translate to edge coverage. MCDC requirements 5 and 6 get to the complex part of MCDC testing. In the following, the three variations discussed by Chilenski are intended to clarify the meaning of point 6 of the general definition, namely, the exact meaning of "independence."

Definition (per Chilenski)

"*Unique-Cause MCDC* [requires] a unique cause (toggle a single condition and change the expression result) for all possible (uncoupled) conditions."

Definition (per Chilenski)

"*Unique-Cause + Masking MCDC* [requires] a unique cause (toggle a single condition and change the expression result) for all possible (uncoupled) conditions. In the case of strongly coupled conditions, masking [is allowed] for that condition only, i.e., all other (uncoupled) conditions will remain fixed."

Definition (per Chilenski)

"*Masking MCDC* allows masking for all conditions, coupled and uncoupled (toggle a single condition and change the expression result) for all possible (uncoupled) conditions. In the case of strongly coupled conditions, masking [is allowed] for that condition only (i.e., all other (uncoupled) conditions will remain fixed)."

Chilenski comments: "In the case of strongly coupled conditions, no coverage set is possible as DO-178B provides no guidance on how such conditions should be covered."

8.3.4 Examples

The examples in this section are directed at the variations of testing code with compound conditions.

8.3.4.1 Condition with Two Simple Conditions

Consider the program fragment in Figure 8.7. It is deceptively simple, with a cyclomatic complexity of 2.

The decision table (see Chapter 7) for the condition (a AND (b OR c)) is in Table 8.2. Decision coverage is attained by exercising any pair of rules such that each action is executed at least once. Test cases corresponding to rules 3 and 4 provide decision coverage, as do rules 1 and 8. Condition coverage is attained by exercising a set of rules such that each condition is evaluated to both true and false. Test cases corresponding to rules 1 and 8 provide decision coverage, as do rules 4 and 5.

```
1. If (a AND (b OR c))
2.     Then  y = 1
3.     Else   y = 2
4. EndIf
```

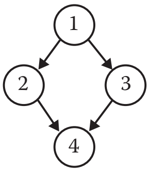


Figure 8.7 Compound condition and its program graph.

Table 8.2 Decision Table for Example Program Fragment

Conditions	*Rule 1*	*Rule 2*	*Rule 3*	*Rule 4*	*Rule 5*	*Rule 6*	*Rule 7*	*Rule 8*
a	T	T	T	T	F	F	F	F
b	T	T	F	F	T	T	F	F
c	T	F	T	F	T	F	T	F
a AND (b OR c)	True	True	True	False	False	False	False	False
Actions								
$y = 1$	x	x	x	—	—	—	—	—
$y = 2$	—	—	—	x	x	x	x	x

To attain MCDC, each condition must be evaluated to both true and false while holding the other conditions constant, and the change must be visible at the outcome. Rules 1 and 5 toggle condition a; rules 2 and 4 toggle condition b; and rules 3 and 4 toggle condition c.

In the Chelinski (2001) paper (p. 9), it happens that the Boolean expression used is

(a AND (b OR c))

In its expanded form, (a AND b) OR (a AND c), the Boolean variable *a* cannot be subjected to unique cause MCDC testing because it appears in both AND expressions.

Given all the complexities here (see Chelinski [2001] for much, much more) the best practical solution is to just make a decision table of the actual code, and look for impossible rules. Any dependencies will typically generate an impossible rule.

8.3.4.2 Compound Condition from NextDate

In our continuing NextDate problem, suppose we have some code checking for valid inputs of the day, month, and year variables. A code fragment for this and its program graph are in Figure 8.8. Table 8.3 is a decision table for the NextDate code fragment. Since the day, month, and year variables are all independent, each can be either true or false. The cyclomatic complexity of the program graph in Figure 8.8 is 5.

Decision coverage is attained by exercising any pair of rules such that each action is executed at least once. Test cases corresponding to rule 1 and any one of rules 2–8 provide decision coverage.

Multiple condition coverage requires exercising a set of rules such that each condition is evaluated to both True and False. The eight test cases corresponding to all eight rules are necessary to provide decision coverage.

To attain MCDC, each condition must be evaluated to both true and false while holding the other conditions constant, and the change must be visible at the outcome. Rules 1 and 2 toggle condition yearOK; rules 1 and 3 toggle condition monthOK, and rules 1 and 5 toggle condition dayOK.

Since the three variables are truly independent, multiple condition coverage will be needed.

```
1  NextDate Fragment
2  Dim day, month, year As Integer
3  Dim dayOK, monthOK, yearOK As Boolean
4  Do
5      Input(day, month, year)
6      If 0 < day < 32
7          Then dayOK = True
8          Else dayOK = False
9      EndIf
10     If 0 < month < 13
11         Then monthOK = True
12         Else  monthOK = False
13     EndIf
14     If 1811 < year < 2013
15         Then yearOK = True
16         Else  yearOK = False
17     EndIf
18 Until (dayOK AND monthOK AND yearOK)
19 End Fragment
```

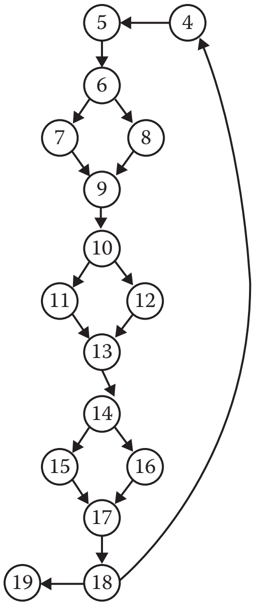


Figure 8.8 NextDate fragment and its program graph.

Table 8.3 Decision Table for NextDate Fragment

Conditions	*Rule 1*	*Rule 2*	*Rule 3*	*Rule 4*	*Rule 5*	*Rule 6*	*Rule 7*	*Rule 8*
dayOK	T	T	T	T	F	F	F	F
monthOK	T	T	F	F	T	T	F	F
YearOK	T	F	T	F	T	F	T	F
The Until condition	True	False	False	False	False	False	False	False
Actions								
Leave the loop	x	—	—	—	—	—	—	—
Repeat the loop	—	x	x	x	x	x	x	x

8.3.4.3 Compound Condition from the Triangle Program

This example is included to show important differences between it and the first two examples. The code fragment in Figure 8.9 is the part of the triangle program that checks to see if the values of sides a, b, and c constitute a triangle. The test incorporates the definition that each side must be strictly less than the sum of the other two sides. Notice that the program graphs in Figures 8.7 and 8.9 are identical. The NextDate fragment and the triangle program fragment are both functions of three variables. The second difference is that a, b, and c in the triangle program are dependent, whereas dayOK, monthOK, and yearOK in the NextDate fragment are truly independent variables.

```
1. If (a < b + c) AND (a < b + c) AND (a < b + c)
2.     Then  IsA Triangle = True
3.     Else   IsA Triangle = False
4. EndIf
```

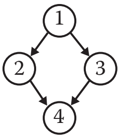


Figure 8.9 Triangle program fragment and its program graph.

The dependence among a, b, and c is the cause of the four impossible rules in the decision table for the fragment in Table 8.4; this is proved next.

Fact: It is numerically impossible to have two of the conditions false.

Proof (by contradiction): Assume any pair of conditions can both be true. Arbitrarily choosing the first two conditions that could both be true, we can write the two inequalities

$$a >= (b + c)$$

$$b >= (a + c)$$

Adding them together, we have

$$(a + b) >= (b + c) + (a + c)$$

and rearranging the right side, we have

$$(a + b) >= (a + b) + 2c$$

But a, b, and c are all > 0, so we have a contradiction. QED.

Decision coverage is attained by exercising any pair of rules such that each action is executed at least once. Test cases corresponding to rules 1 and 2 provide decision coverage, as do rules 1 and 3, and rules 1 and 5. Rules, 4, 6, 7, and 8 cannot be used owing to their numerical impossibility.

Condition coverage is attained by exercising a set of rules such that each condition is evaluated to both true and false. Test cases corresponding to rules 1 and 2 toggle the $(c < a + b)$ condition, rules 1 and 3 toggle the $(b < a + c)$ condition, and 1 and 5 toggle the $(a < b + c)$ condition.

MCDC is complicated by the numerical (and hence logical) impossibilities among the three conditions. The three pairs (rules 1 and 2, rules 1 and 3, and rules 1 and 5) constitute MCDC.

Table 8.4 Decision Table for Triangle Program Fragment

Conditions	*Rule 1*	*Rule 2*	*Rule 3*	*Rule 4*	*Rule 5*	*Rule 6*	*Rule 7*	*Rule 8*
(a < b + c)	T	T	T	T	F	F	F	F
(b < a + c)	T	T	F	F	T	T	F	F
(c < a + b)	T	F	T	F	T	F	T	F
IsATriangle = True	x	—	—	—	—	—	—	—
IsATriangle = False	—	x	x	—	x	—	—	—
Impossible	—	—	—	x		x	x	x

In complex situations such as these examples, falling back on decision tables is an answer that will always work. Rewriting the compound condition with nested If logic, we will have (preserving the original statement numbers)

```
1.1    If (a < b + c)
1.2          Then If (b < a + c)
1.3                Then If (c < a + b)
2                        Then IsATriangle = True
3.1                      Else IsATriangle = False
3.2                     End If
3.3                Else IsATriangle = False
3.4               End If
3.5          Else IsATriangle = False
4      EndIf
```

This code fragment avoids the numerically impossible combinations of a, b, and c. There are four distinct paths through its program graph, and these correspond to rules 1, 2, 3, and 5 in the decision table.

8.3.5 Test Coverage Analyzers

Coverage analyzers are a class of test tools that offer automated support for this approach to testing management. With a coverage analyzer, the tester runs a set of test cases on a program that has been "instrumented" by the coverage analyzer. The analyzer then uses information produced by the instrumentation code to generate a coverage report. In the common case of DD-path coverage, for example, the instrumentation identifies and labels all DD-paths in an original program. When the instrumented program is executed with test cases, the analyzer tabulates the DD-paths traversed by each test case. In this way, the tester can experiment with different sets of test cases to determine the coverage of each set. Mr. Tilo Linz maintains a website with excellent test tool information at www.testtoolreview.com.

8.4 Basis Path Testing

The mathematical notion of a "basis" has attractive possibilities for structural testing. Certain sets can have a basis; and when they do, the basis has very important properties with respect to the entire set. Mathematicians usually define a basis in terms of a structure called a "vector space," which is a set of elements (called vectors) as well as operations that correspond to multiplication and addition defined for the vectors. If a half dozen other criteria apply, the structure is said to be a vector space, and all vector spaces have a basis (in fact they may have several bases). The basis of a vector space is a set of vectors that are independent of each other and "span" the entire vector space in the sense that any other vector in the space can be expressed in terms of the basis vectors. Thus, a set of basis vectors somehow represents "the essence" of the full vector space: everything else in the space can be expressed in terms of the basis, and if one basis element is deleted, this spanning property is lost. The potential application of this theory for testing is that, if we can view a program as a vector space, then the basis for such a space would be a very interesting set of elements to test. If the basis is okay, we could hope that everything that can be expressed in terms of the basis is also okay. In this section, we examine the early work of Thomas McCabe, who recognized this possibility in the mid-1970s.

8.4.1 McCabe's Basis Path Method

Figure 8.10 is taken from McCabe (1982). It is a directed graph that we might take to be the program graph (or the DD-path graph) of some program. For the convenience of readers who have encountered this example elsewhere (McCabe, 1987; Perry, 1987), the original notation for nodes and edges is repeated here. (Notice that this is not a graph derived from a structured program: nodes B and C are a loop with two exits, and the edge from B to E is a branch into the if–then statement in nodes D, E, and F.) The program does have a single entry (A) and a single exit (G). McCabe based his view of testing on a major result from graph theory, which states that the cyclomatic number (see Chapter 4) of a strongly connected graph is the number of linearly independent circuits in the graph. (A circuit is similar to a chain: no internal loops or decisions occur, but the initial node is the terminal node. A circuit is a set of 3-connected nodes.)

We can always create a strongly connected graph by adding an edge from the (every) sink node to the (every) source node. (Notice that, if the single-entry, single-exit precept is violated, we greatly increase the cyclomatic number because we need to add edges from each sink node to each source node.) The right side of Figure 8.10 shows the result of doing this; it also contains edge labels that are used in the discussion that follows.

Some confusion exists in the literature about the correct formula for cyclomatic complexity. Some sources give the formula as $V(G) = e - n + p$, while others use the formula $V(G) = e - n + 2p$; everyone agrees that e is the number of edges, n is the number of nodes, and p is the number of connected regions. The confusion apparently comes from the transformation of an arbitrary directed graph (such as the one in Figure 8.10, left side) to a strongly connected, directed graph obtained by adding one edge from the sink to the source node (as in Figure 8.10, right side). Adding an edge clearly affects value computed by the formula, but it should not affect the number of circuits. Counting or not counting the added edge accounts for the change to the coefficient of p, the number of connected regions. Since p is usually 1, adding the extra edge means we move from $2p$ to p. Here is a way to resolve the apparent inconsistency. The number of linearly independent paths from the source node to the sink node of the graph on the left side of Figure 8.10 is

$$\begin{aligned} V(G) &= e - n + 2p \\ &= 10 - 7 + 2(1) = 5 \end{aligned}$$

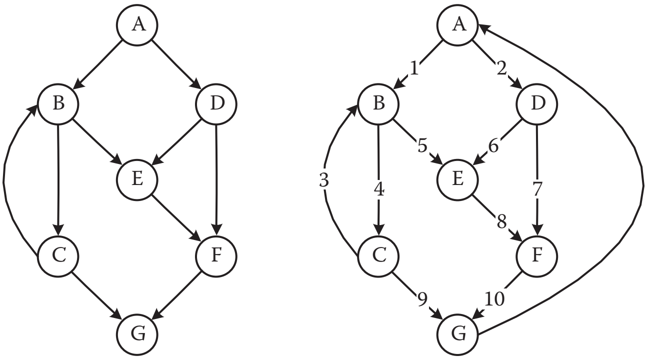


Figure 8.10 McCabe's control graph and derived strongly connected graph.

The number of linearly independent circuits of the graph on the right side of the graph in Figure 8.10 is

$$\begin{aligned} V(G) &= e - n + p \\ &= 11 - 7 + 1 = 5 \end{aligned}$$

The cyclomatic complexity of the strongly connected graph in Figure 8.10 is 5; thus, there are five linearly independent circuits. If we now delete the added edge from node G to node A, these five circuits become five linearly independent paths from node A to node G. In small graphs, we can visually identify independent paths. Here, we identify paths as sequences of nodes:

p1: A, B, C, G
p2: A, B, C, B, C, G
p3: A, B, E, F, G
p4: A, D, E, F, G
p5: A, D, F, G

Table 8.5 shows the edges traversed by each path, and also the number of times an edge is traversed. We can force this to begin to look like a vector space by defining notions of addition and scalar multiplication: path addition is simply one path followed by another path, and multiplication corresponds to repetitions of a path. With this formulation, McCabe arrives at a vector space of program paths. His illustration of the basis part of this framework is that the path A, B, C, B, E, F, G is the basis sum p2 + p3 – p1, and the path A, B, C, B, C, B, C, G is the linear combination 2p2 – p1. It is easier to see this addition with an incidence matrix (see Chapter 4) in which rows correspond to paths, and columns correspond to edges, as in Table 8.5. The entries in this table are obtained by following a path and noting which edges are traversed. Path p1, for example, traverses edges 1, 4, and 9, while path p2 traverses the following edge sequence: 1, 4, 3, 4, 9. Because edge 4 is traversed twice by path p2, that is the entry for the edge 4 column.

We can check the independence of paths p1 – p5 by examining the first five rows of this incidence matrix. The bold entries show edges that appear in exactly one path, so paths p2 – p5 must

Table 8.5 Path/Edge Traversal

Path/Edges Traversed	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
p1: A, B, C, G	1	0	0	1	0	0	0	0	1	0
p2: A, B, C, B, C, G	1	0	**1**	2	0	0	0	0	1	0
p3: A, B, E, F, G	1	0	0	0	**1**	0	0	1	0	1
p4: A, D, E, F, G	0	1	0	0	0	**1**	0	1	0	1
p5: A, D, F, G	0	1	0	0	0	0	**1**	0	0	1
ex1: A, B, C, B, E, F, G	1	0	1	1	1	0	0	1	0	1
ex2: A, B, C, B, C, B, C, G	1	0	2	3	0	0	0	0	1	0

Table 8.6 Basis Paths in Figure 8.5

Original	p1: A–B–C–E–F–H–J–K–M–N–O–Last	Scalene
Flip p1 at B	p2: A–B–D–E–F–H–J–K–M–N–O–Last	Infeasible
Flip p1 at F	p3: A–B–C–E–F–G–O–Last	Infeasible
Flip p1 at H	p4: A–B–C–E–F–H–I–N–O–Last	Equilateral
Flip p1 at J	p5: A–B–C–E–F–H–J–L–M–N–O–Last	Isosceles

be independent. Path p1 is independent of all of these, because any attempt to express p1 in terms of the others introduces unwanted edges. None can be deleted, and these five paths span the set of all paths from node A to node G. At this point, you might check the linear combinations of the two example paths. (The addition and multiplication are performed on the column entries.)

McCabe next develops an algorithmic procedure (called the baseline method) to determine a set of basis paths. The method begins with the selection of a baseline path, which should correspond to some "normal case" program execution. This can be somewhat arbitrary; McCabe advises choosing a path with as many decision nodes as possible. Next, the baseline path is retraced, and in turn each decision is "flipped"; that is, when a node of outdegree ≥ 2 is reached, a different edge must be taken. Here we follow McCabe's example, in which he first postulates the path through nodes A, B, C, B, E, F, G as the baseline. (This was expressed in terms of paths p1 – p5 earlier.) The first decision node (outdegree ≥ 2) in this path is node A; thus, for the next basis path, we traverse edge 2 instead of edge 1. We get the path A, D, E, F, G, where we retrace nodes E, F, G in path 1 to be as minimally different as possible. For the next path, we can follow the second path, and take the other decision outcome of node D, which gives us the path A, D, F, G. Now, only decision nodes B and C have not been flipped; doing so yields the last two basis paths, A, B, E, F, G and A, B, C, G. Notice that this set of basis paths is distinct from the one in Table 8.6: this is not problematic because a unique basis is not required.

8.4.2 Observations on McCabe's Basis Path Method

If you had trouble following some of the discussion on basis paths and sums and products of these, you may have felt a haunting skepticism—something along the lines of, "Here's another academic oversimplification of a real-world problem." Rightly so, because two major soft spots occur in the McCabe view: one is that testing the set of basis paths is sufficient (it is not), and the other has to do with the yoga-like contortions we went through to make program paths look like a vector space. McCabe's example that the path A, B, C, B, C, B, C, G is the linear combination 2p2 – p1 is very unsatisfactory. What does the 2p2 part mean? Execute path p2 twice? (Yes, according to the math.) Even worse, what does the –p1 part mean? Execute path p1 backward? Undo the most recent execution of p1? Do not do p1 next time? Mathematical sophistries like this are a real turnoff to practitioners looking for solutions to their very real problems. To get a better understanding of these problems, we will go back to the triangle program example.

Start with the DD-path graph of the triangle program in Figure 8.5. We begin with a baseline path that corresponds to a scalene triangle, for example, with sides 3, 4, 5. This test case will traverse the path p1 (see Table 8.5). Now, if we flip the decision at node B, we get path p2. Continuing the procedure, we flip the decision at node F, which yields the path p3. Now, we continue to flip decision nodes in the baseline path p1; the next node with outdegree = 2 is node H. When we flip node H, we get the path p4. Next, we flip node J to get p5. We know we are done because there are only five basis paths; they are shown in Table 8.5.

Time for a reality check: if you follow paths p2 and p3, you find that they are both infeasible. Path p2 is infeasible because passing through node D means the sides are not a triangle; so the outcome of the decision at node F must be node G. Similarly, in p3, passing through node C means the sides do form a triangle; so node G cannot be traversed. Paths p4 and p5 are both feasible and correspond to equilateral and isosceles triangles, respectively. Notice that we do not have a basis path for the NotATriangle case.

Recall that dependencies in the input data domain caused difficulties for boundary value testing and that we resolved these by going to decision table-based specification-based testing, where we addressed data dependencies in the decision table. Here, we are dealing with code-level dependencies, which are absolutely incompatible with the latent assumption that basis paths are independent. McCabe's procedure successfully identifies basis paths that are topologically independent; however, when these contradict semantic dependencies, topologically possible paths are seen to be logically infeasible. One solution to this problem is to always require that flipping a decision results in a semantically feasible path. Another is to reason about logical dependencies. If we think about this problem, we can identify two rules:

If node C is traversed, then we must traverse node H.
If node D is traversed, then we must traverse node G.

Taken together, these rules, in conjunction with McCabe's baseline method, will yield the following feasible basis path set. Notice that logical dependencies reduce the size of a basis set when basis paths must be feasible.

p1: A–B–C–E–F–H–J–K–M–N–O–Last	Scalene
p6: A–B–D–E–F–G–O–Last	Not a triangle
p4: A–B–C–E–F–H–I–N–O–Last	Equilateral
p5: A–B–C–E–F–H–J–L–M–N–O–Last	Isosceles

The triangle problem is atypical in that no loops occur. The program has only eight topologically possible paths; and of these, only the four basis paths listed above are feasible. Thus, for this special case, we arrive at the same test cases as we did with special value testing and output range testing.

For a more positive observation, basis path coverage guarantees DD-path coverage: the process of flipping decisions guarantees that every decision outcome is traversed, which is the same as DD-path coverage. We see this by example from the incidence matrix description of basis paths and in our triangle program feasible basis paths. We could push this a step further and observe that the set of DD-paths acts like a basis because any program path can be expressed as a linear combination of DD-paths.

8.4.3 Essential Complexity

Part of McCabe's work on cyclomatic complexity does more to improve programming than testing. In this section, we take a quick look at this elegant blend of graph theory, structured programming, and the implications these have for testing. This whole package centers on the notion of essential complexity (McCabe, 1982), which is only the cyclomatic complexity of yet another form of condensation graph. Recall that condensation graphs are a way of simplifying an existing graph; thus far, our simplifications have been based on removing either strong components or DD-paths. Here, we condense around the structured programming constructs, which are repeated as Figure 8.11.

The basic idea is to look for the graph of one of the structured programming constructs, collapse it into a single node, and repeat until no more structured programming constructs can be found. This process is followed in Figure 8.12, which starts with the DD-path graph of the pseudocode triangle program. The if–then–else construct involving nodes B, C, D, and E is condensed into node a, and then the three if–then constructs are condensed onto nodes b, c, and d. The remaining if–then–else (which corresponds to the IF IsATriangle statement) is condensed into node e, resulting in a condensed graph with cyclomatic complexity $V(G) = 1$. In general, when a program is well structured (i.e., is composed solely of the structured programming constructs), it can always be reduced to a graph with one path.

The graph in Figure 8.10 cannot be reduced in this way (try it!). The loop with nodes B and C cannot be condensed because of the edge from B to E. Similarly, nodes D, E, and F look like an if–then construct, but the edge from B to E violates the structure. McCabe (1976) went on to find elemental "unstructures" that violate the precepts of structured programming. These are shown in Figure 8.13. Each of these violations contains three distinct paths, as opposed to the two paths present in the corresponding structured programming constructs;

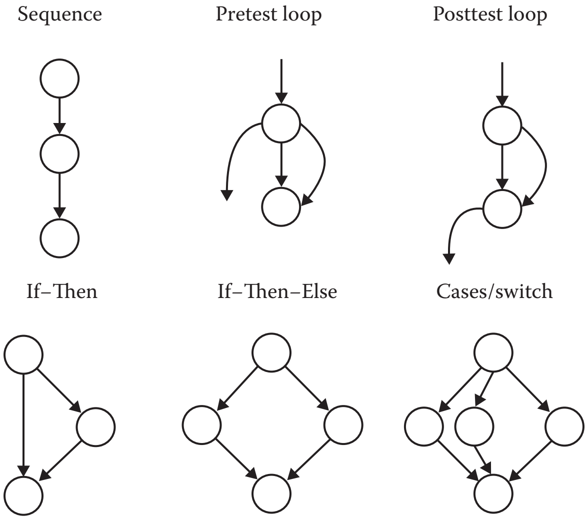


Figure 8.11 Structured programming constructs.

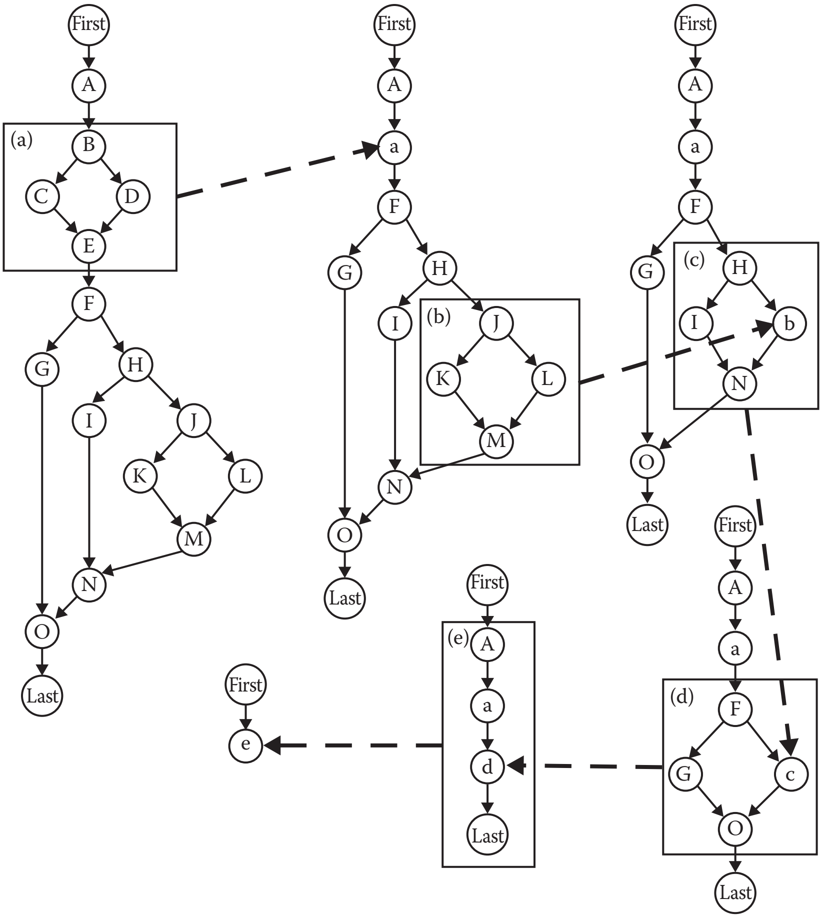


Figure 8.12 Condensing with respect to structured programming constructs.

so one conclusion is that such violations increase cyclomatic complexity. The *pièce de resistance* of McCabe's analysis is that these violations cannot occur by themselves: if one occurs in a program, there must be at least one more, so a program cannot be only slightly unstructured. Because these increase cyclomatic complexity, the minimum number of test cases is thereby increased. In the next chapter, we will see that the violations have interesting implications for data flow testing.

The bottom line for testers is this: programs with high cyclomatic complexity require more testing. Of the organizations that use the cyclomatic complexity metric, most set some guideline for maximum acceptable complexity; $V(G) = 10$ is a common choice. What happens if a unit has a higher complexity? Two possibilities: either simplify the unit or plan to do more testing. If the unit is well structured, its essential complexity is 1; so it can be simplified easily. If the unit has an essential complexity greater than 1, often the best choice is to eliminate the violations.

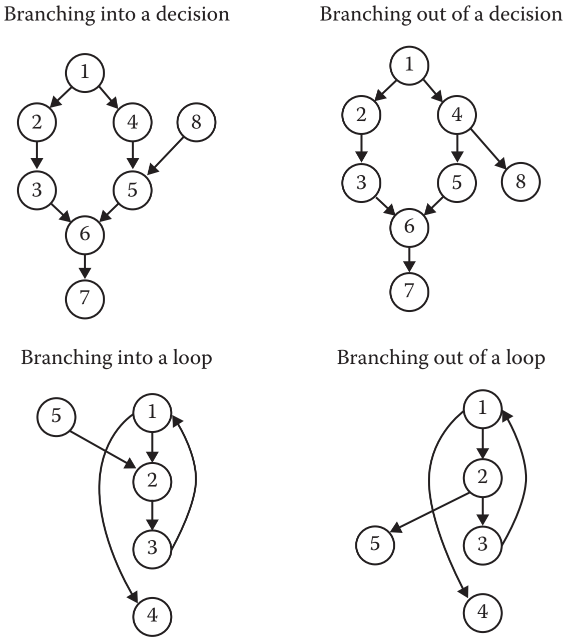


Figure 8.13 Violations of structured programming constructs.

8.5 Guidelines and Observations

In our study of specification-based testing, we observed that gaps and redundancies can both exist and, at the same time, cannot be recognized. The problem was that specification-based testing removes us too far from the code. The path testing approaches to code-based testing represent the case where the pendulum has swung too far the other way: moving from code to directed graph representations and program path formulations obscures important information that is present in the code, in particular the distinction between feasible and infeasible paths. Also, no form of code-based testing can reveal missing functionality that is specified in the requirements. In the next chapter, we look at data flow-based testing. These techniques move closer to the code, so the pendulum will swing back from the path analysis extreme.

McCabe (1982) was partly right when he observed, “It is important to understand that these are purely criteria that measure the quality of testing, and not a procedure to identify test cases.” He was referring to the DD-path coverage metric and his basis path heuristic based on cyclomatic complexity metric. Basis path testing therefore gives us a lower boundary on how much testing is necessary.

Path-based testing also provides us with a set of metrics that act as crosschecks on specification-based testing. We can use these metrics to resolve the gaps and redundancies question. When we find that the same program path is traversed by several functional test cases, we suspect that this redundancy is not revealing new faults. When we fail to attain DD-path coverage, we know that there are gaps in the functional test cases. As an example, suppose we have a program that contains extensive error handling, and we test it with boundary value test cases (min, min+, nom, max–, and max). Because these are all permissible values, DD-paths corresponding to the error-handling code will not be traversed. If we add test cases derived from robustness testing or traditional equivalence class testing, the DD-path coverage will improve. Beyond this rather obvious use of coverage metrics, an opportunity exists for real testing craftsmanship. Any of the coverage metrics in Section 8.3 can operate in two ways: either as a blanket-mandated standard (e.g., all units shall be tested to attain full DD-path coverage) or as a mechanism to selectively test portions of code more rigorously than others. We might choose multiple-condition coverage for modules with complex logic, while those with extensive iteration might be tested in terms of the loop coverage techniques. This is probably the best view of structural testing: use the properties of the source code to identify appropriate coverage metrics, and then use these as a crosscheck on functional test cases. When the desired coverage is not attained, follow interesting paths to identify additional (special value) test cases.

EXERCISES

1. Find the cyclomatic complexity of the graph in Figure 8.3.
2. Identify a set of basis paths for the graph in Figure 8.3.
3. Discuss McCabe's concept of "flipping" for nodes with outdegree ≥ 3.
4. Suppose we take Figure 8.3 as the DD-path graph of some program. Develop sets of paths (which would be test cases) for the C_0, C_1, and C_2 metrics.
5. Develop multiple-condition coverage test cases for the pseudocode triangle program. (Pay attention to the dependency between statement fragments 14 and 16 with the expression (a = b) AND (b = c).)
6. Rewrite the program segment 14–20 such that the compound conditions are replaced by nested if–then–else statements. Compare the cyclomatic complexity of your program with that of the existing version.

```
14. If (a = b) AND (b = c)
15.  Then Output ("Equilateral")
16.  Else If (a ≠ b) AND (a ≠ c) AND (b ≠ c)
17.      Then Output ("Scalene")
18.      Else Output ("Isosceles")
19.  EndIf
20. EndIf
```

7. Look carefully at the original statement fragments 14–20. What happens with a test case (e.g., a = 3, b = 4, c = 3) in which a = c? The condition in line 14 uses the transitivity of equality to eliminate the a = c condition. Is this a problem?
8. The codeBasedTesting.xls Excel spreadsheet at the CRC website (www.crcpress.com/product/isbn/9781466560680) contains instrumented VBA implementations of the triangle, NextDate, and commission problems that you may have analyzed with the specBasedTesting.xls spreadsheet. The output shows the DD-path coverage of individual test cases and an indication of any faults revealed by a failing test case. Experiment with various sets of test cases to see if you can devise a set of test cases that has full DD-path coverage yet does not reveal the known faults.
9. (For mathematicians only.) For a set V to be a vector space, two operations (addition and scalar multiplication) must be defined for elements in the set. In addition, the following criteria must hold for all vectors x, y, and $z \in V$, and for all scalars k, l, 0, and 1:
    a. If $x, y \in V$, the vector $x + y \in V$.
    b. $x + y = y + x$.
    c. $(x + y) + z = x + (y + z)$.
    d. There is a vector $0 \in V$ such that $x + 0 = x$.

e. For any $x \in V$, there is a vector $-x \in V$ such that $x + (-x) = 0$.
f. For any $x \in V$, the vector $kx \in V$, where k is a scalar constant.
g. $k(x + y) = kx + ky$.
h. $(k + l)x = kx + lx$.
i. $k(lx) = (kl)x$.
j. $1x = x$.

How many of these 10 criteria hold for the “vector space” of paths in a program?

References

Beizer, B., *Software Testing Techniques*, Van Nostrand, New York, 1984.
Chilenski, J.J., *An Investigation of Three Forms of the Modified Condition Decision Coverage (MCDC) Criterion*, DOT/FAA/AR-01/18, April 2001.
http://www.faa.gov/about/office_org/headquarters_offices/ang/offices/tc/library/ (see actlibrary.tc.faa.gov).
Huang, J.C., Detection of dataflow anomaly through program instrumentation, *IEEE Transactions on Software Engineering*, Vol. SE-5, 1979, pp. 226–236.
Miller, E.F. Jr., *Tutorial: Program Testing Techniques*, COMPSAC ’77, IEEE Computer Society, 1977.
Miller, E.F. Jr., Automated software testing: a technical perspective, *American Programmer*, Vol. 4, No. 4, April 1991, pp. 38–43.
McCabe, T. J., A complexity metric, *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 4, December 1976, pp. 308–320.
McCabe, T.J., *Structural Testing: A Software Testing Methodology Using the Cyclomatic Complexity Metric, National Bureau of Standards (Now NIST)*, Special Publication 500-99, Washington, DC, 1982.
McCabe, T.J., *Structural Testing: A Software Testing Methodology Using the Cyclomatic Complexity Metric*, McCabe and Associates, Baltimore, 1987.
Perry, W.E., *A Structured Approach to Systems Testing*, QED Information Systems, Inc., Wellesley, MA, 1987.
Schach, S.R., *Software Engineering*, 2nd ed., Richard D. Irwin, Inc. and Aksen Associates, Inc., Homewood, IL, 1993.

Chapter 9

Data Flow Testing

Data flow testing is an unfortunate term because it suggests some connection with data flow diagrams; no connection exists. Data flow testing refers to forms of structural testing that focus on the points at which variables receive values and the points at which these values are used (or referenced). We will see that data flow testing serves as a “reality check” on path testing; indeed, many of the data flow testing proponents (and researchers) see this approach as a form of path testing. While dataflow and slice-based testing are cumbersome at the unit level; they are well suited for object-oriented code. We will look at two mainline forms of data flow testing: one provides a set of basic definitions and a unifying structure of test coverage metrics, while the other is based on a concept called a “program slice.” Both of these formalize intuitive behaviors (and analyses) of testers; and although they both start with a program graph, both move back in the direction of functional testing. Also, both of these methods are difficult to perform manually, and unfortunately, few commercial tools exist to make life easier for the data flow and slicing testers. On the positive side, both techniques are helpful for coding and debugging.

Most programs deliver functionality in terms of data. Variables that represent data somehow receive values, and these values are used to compute values for other variables. Since the early 1960s, programmers have analyzed source code in terms of the points (statements and statement fragments) at which variables receive values and points at which these values are used. Many times, their analyses were based on concordances that list statement numbers in which variable names occur. Concordances were popular features of second-generation language compilers (they are still popular with COBOL programmers). Early data flow analyses often centered on a set of faults that are now known as define/reference anomalies:

A variable that is defined but never used (referenced)
A variable that is used before it is defined
A variable that is defined twice before it is used

Each of these anomalies can be recognized from the concordance of a program. Because the concordance information is compiler generated, these anomalies can be discovered by what is known as static analysis: finding faults in source code without executing it.

9.1 Define/Use Testing

Much of the formalization of define/use testing was done in the early 1980s (Rapps and Weyuker, 1985); the definitions in this section are compatible with those in Clarke et al. (1989), which summarizes most define/use testing theory. This body of research is very compatible with the formulation we developed in Chapters 4 and 8. It presumes a program graph in which nodes are statement fragments (a fragment may be an entire statement) and programs that follow the structured programming precepts.

The following definitions refer to a program P that has a program graph $G(P)$ and a set of program variables V. The program graph $G(P)$ is constructed as in Chapter 4, with statement fragments as nodes and edges that represent node sequences. $G(P)$ has a single-entry node and a single-exit node. We also disallow edges from a node to itself. Paths, subpaths, and cycles are as they were in Chapter 4. The set of all paths in P is PATHS(P).

Definition

Node $n \in G(P)$ is a *defining node* of the variable $v \in V$, written as DEF(v, n), if and only if the value of variable v is defined as the statement fragment corresponding to node n.

Input statements, assignment statements, loop control statements, and procedure calls are all examples of statements that are defining nodes. When the code corresponding to such statements executes, the contents of the memory location(s) associated with the variables are changed.

Definition

Node $n \in G(P)$ is a *usage node* of the variable $v \in V$, written as USE(v, n), if and only if the value of the variable v is used as the statement fragment corresponding to node n.

Output statements, assignment statements, conditional statements, loop control statements, and procedure calls are all examples of statements that are usage nodes. When the code corresponding to such statements executes, the contents of the memory location(s) associated with the variables remain unchanged.

Definition

A usage node USE(v, n) is a *predicate use* (denoted as P-use) if and only if the statement n is a predicate statement; otherwise, USE(v, n) is a computation use (denoted C-use).

The nodes corresponding to predicate uses always have an outdegree ≥ 2, and nodes corresponding to computation uses always have an outdegree ≤ 1.

Definition

A *definition/use path* with respect to a variable v (denoted du-path) is a path in PATHS(P) such that, for some $v \in V$, there are define and usage nodes DEF(v, m) and USE(v, n) such that m and n are the initial and final nodes of the path.

Definition

A *definition-clear path* with respect to a variable *v* (denoted dc-path) is a definition/use path in PATHS(*P*) with initial and final nodes DEF(*v*, *m*) and USE(*v*, *n*) such that no other node in the path is a defining node of *v*.

Testers should notice how these definitions capture the essence of computing with stored data values. Du-paths and dc-paths describe the flow of data across source statements from points at which the values are defined to points at which the values are used. Du-paths that are not definition clear are potential trouble spots. One of the main values of du-paths is they identify points for variable "watches" and breakpoints when code is developed in an Integrated Development Environment. Figure 9.3 illustrates this very well later in the chapter.

9.1.1 Example

We will use the commission problem and its program graph to illustrate these definitions. The numbered pseudocode and its corresponding program graph are shown in Figure 9.1. This program computes the commission on the sales of the total numbers of locks, stocks, and barrels sold. The while loop is a classic sentinel controlled loop in which a value of –1 for locks signifies the end of the sales data. The totals are accumulated as the data values are read in the while loop. After printing this preliminary information, the sales value is computed, using the constant item prices defined at the beginning of the program. The sales value is then used to compute the commission in the conditional portion of the program.

Figure 9.2 shows the decision-to-decision path (DD-path) graph of the program graph in Figure 9.1. More compression exists in this DD-path graph because of the increased computation in the commission problem. Table 9.1 details the statement fragments associated with DD-paths. Some DD-paths (per the definition in Chapter 8) are combined to simplify the graph. We will need this figure later to help visualize the differences among DD-paths, du-paths, and program slices.

Table 9.2 lists the define and usage nodes for the variables in the commission problem. We use this information in conjunction with the program graph in Figure 9.1 to identify various definition/use and definition-clear paths. It is a judgment call whether nonexecutable statements such as constant and variable declaration statements should be considered as defining nodes. Such nodes are not very interesting when we follow what happens along their du-paths; but if something is wrong, it can be helpful to include them. Take your pick. We will refer to the various paths as sequences of node numbers.

Tables 9.3 and 9.4 present some of the du-paths in the commission problem; they are named by their beginning and ending nodes (from Figure 9.1). The third column in Table 9.3 indicates whether the du-paths are definition clear. Some of the du-paths are trivial—for example, those for lockPrice, stockPrice, and barrelPrice. Others are more complex: the while loop (node sequence <14, 15, 16, 17, 18, 19, 20>) inputs and accumulated values for totalLocks, totalStocks, and totalBarrels. Table 9.3 only shows the details for the totalStocks variable. The initial value definition for totalStocks occurs at node 11, and it is first used at node 17. Thus, the path (11, 17), which consists of the node sequence <11, 12, 13, 14, 15, 16, 17>, is definition clear. The path (11, 22), which consists of the node sequence <11, 12, 13, (14, 15, 16, 17, 18, 19, 20)*, 21, 22>, is not definition clear because values of totalStocks are defined at node 11 and (possibly several times at) node 17. (The asterisk after the while loop is the Kleene Star notation used both in formal logic and regular expressions to denote zero or more repetitions.)

```
1  Program Commission (INPUT,OUTPUT)
2  Dim locks, stocks, barrels As Integer
3  Dim lockPrice, stockPrice, barrelPrice As Real
4  Dim totalLocks, totalStocks, totalBarrels As Integer
5  Dim lockSales, stockSales, barrelSales As Real
6  Dim sales, commission As Real
7  lockPrice = 45.0
8  stockPrice = 30.0
9  barrelPrice = 25.0
10 totalBarrels = 0
11 totalStocks = 0
12 totalBarrels = 0
13 Input(locks)
14 While NOT(locks = –1)  “locks = –1 signals end of data
15   Input(stocks, barrels)
16   totalLocks = totalLocks + locks
17   totalStocks = totalStocks + stocks
18   totalBarrels = totalBarrels + barrels
19   Input(locks)
20 EndWhile
21 Output(“Locks sold:,” totalLocks)
22 Output(“Stocks sold:,” totalStocks)
23 Output(“Barrels sold:,” totalBarrels)
24 lockSales = lockPrice*totalLocks
25 stockSales = stockPrice*totalStocks
26 barrelsSales = barrelPrice * totalBarrels
27 sales = lockSales + stockSales + barrelSales
28 Output(“Total sales: ”, sales)
29 If (sales > 1800.0)
30   Then
31     commission = 0.10 * 1000.0
32     commission = commission + 0.15 * 800.0
33     commission = commission + 0.20*(sales–1800.0)
34   Else If (sales > 1000.0)
35     Then
36     commission = 0.10 * 1000.0
37     commission = commission + 0.15*(sales–1000.0)
38     Else
39     commission = 0.10 * sales
40   EndIf
41 EndIf
42 Output(“Commission is $”, commission)
43 End Commission
```

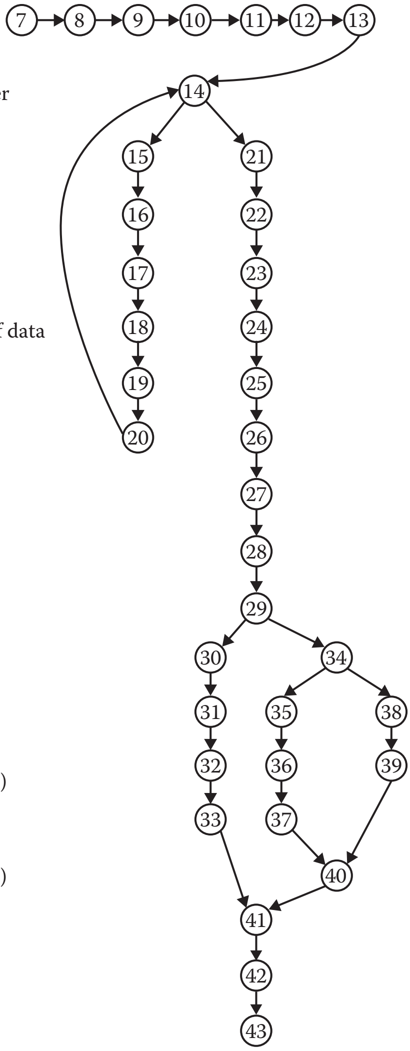


Figure 9.1 Commission problem and its program graph.

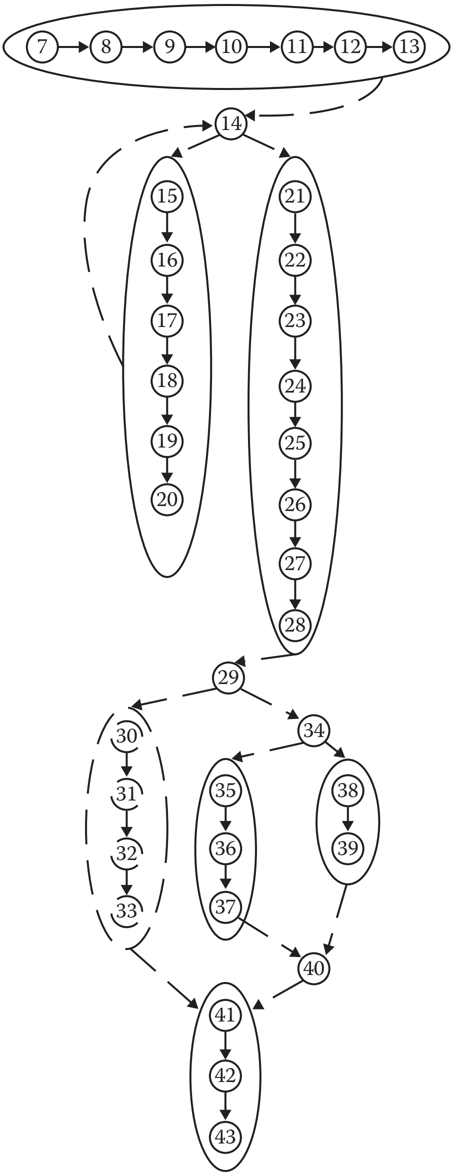

7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43

Figure 9.2 **DD-path graph of commission problem pseudocode (in Figure 9.1).**

9.1.2 Du-paths for Stocks

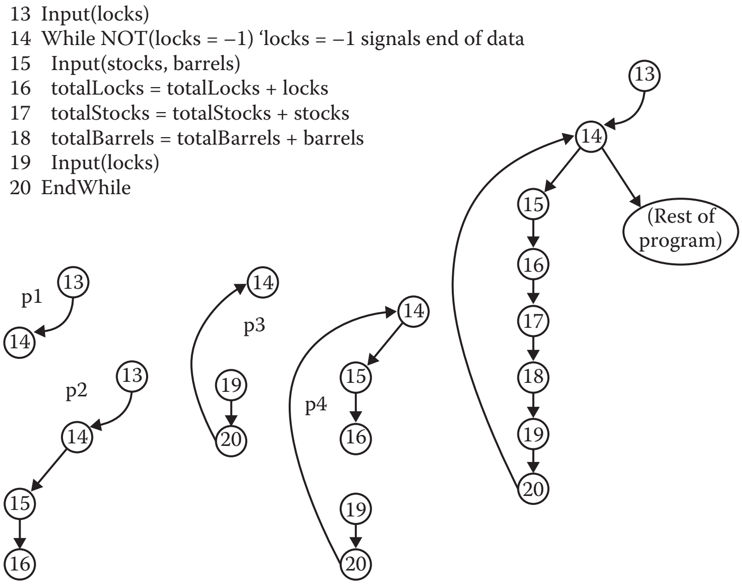
First, let us look at a simple path: the du-path for the variable stocks. We have DEF(stocks, 15) and USE(stocks, 17), so the path <15, 17> is a du-path with respect to stocks. No other defining nodes are used for stocks; therefore, this path is also definition clear.

9.1.3 Du-paths for Locks

Two defining and two usage nodes make the locks variable more interesting: we have DEF(locks, 13), DEF(locks, 19), USE(locks, 14), and USE(locks, 16). These yield four du-paths; they are shown in Figure 9.3.

p1 = <13, 14>
p2 = <13, 14, 15, 16>
p3 = <19, 20, 14>
p4 = <19, 20, 14, 15, 16>

Note: du-paths p1 and p2 refer to the priming value of locks, which is read at node 13. The locks variable has a predicate use in the while statement (node 14), and if the condition is true (as in path p2), a computation use at statement 16. The other two du-paths start near the end of the while loop and occur when the loop repeats. These paths provide the loop coverage discussed in Chapter 8—bypass the loop, begin the loop, repeat the loop, and exit the loop. All these du-paths are definition clear.



```
13  Input(locks)
14  While NOT(locks = −1) 'locks = −1 signals end of data
15    Input(stocks, barrels)
16    totalLocks = totalLocks + locks
17    totalStocks = totalStocks + stocks
18    totalBarrels = totalBarrels + barrels
19    Input(locks)
20  EndWhile
```

Figure 9.3 Du-paths for locks.

Table 9.1 DD-paths in Figure 9.1

DD-path	*Nodes*
A	7, 8, 9, 10, 11, 12, 13
B	14
C	15, 16, 17, 18, 19, 20
D	21, 22, 23, 24, 25, 26, 27, 28
E	29
F	30, 31, 32, 33
G	34
H	35, 36, 37
I	38
J	39
K	40
L	41, 42

Table 9.2 Define/Use Nodes for Variables in Commission Problem

Variable	*Defined at Node*	*Used at Node*
lockPrice	7	24
stockPrice	8	25
barrelPrice	9	26
totalLocks	10, 16	16, 21, 24
totalStocks	11, 17	17, 22, 25
totalBarrels	12, 18	18, 23, 26
Locks	13, 19	14, 16
Stocks	15	17
Barrels	15	18
lockSales	24	27
stockSales	25	27
barrelSales	26	27
Sales	27	28, 29, 33, 34, 37, 38
Commission	31, 32, 33, 36, 37, 38	32, 33, 37, 41

Table 9.3 Selected Define/Use Paths

Variable	*Path (Beginning, End) Nodes*	*Definition Clear?*
lockPrice	7, 24	Yes
stockPrice	8, 25	Yes
barrelPrice	9, 26	Yes
totalStocks	11, 17	Yes
totalStocks	11, 22	No
totalStocks	11, 25	No
totalStocks	17, 17	Yes
totalStocks	17, 22	No
totalStocks	17, 25	No
Locks	13, 14	Yes
Locks	13, 16	Yes
Locks	19, 14	Yes
Locks	19, 16	Yes
Sales	27, 28	Yes
Sales	27, 29	Yes
Sales	27, 33	Yes
Sales	27, 34	Yes
Sales	27, 37	Yes
Sales	27, 38	Yes

Table 9.4 Define/Use Paths for Commission

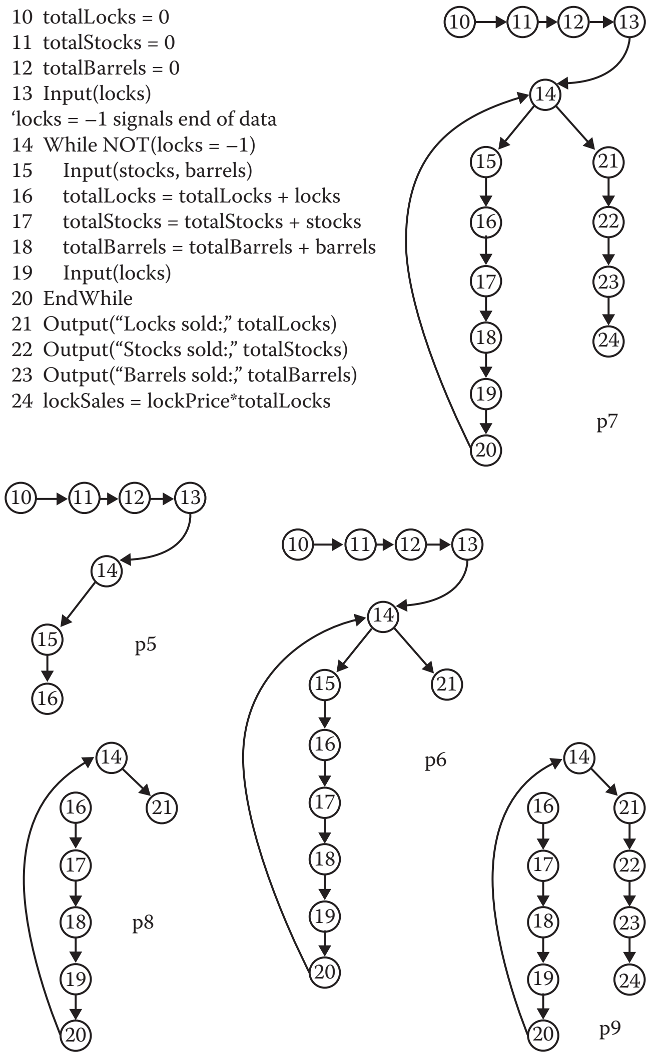
Variable	*Path (Beginning, End) Nodes*	*Feasible?*	*Definition Clear?*
Commission	31, 32	Yes	Yes
Commission	31, 33	Yes	No
Commission	31, 37	No	N/A
Commission	31, 41	Yes	No
Commission	32, 32	Yes	Yes
Commission	32, 33	Yes	Yes
Commission	32, 37	No	N/A
Commission	32, 41	Yes	No
Commission	33, 32	No	N/A
Commission	33, 33	Yes	Yes
Commission	33, 37	No	N/A
Commission	33, 41	Yes	Yes
Commission	36, 32	No	N/A
Commission	36, 33	No	N/A
Commission	36, 37	Yes	Yes
Commission	36, 41	Yes	No
Commission	37, 32	No	N/A
Commission	37, 33	No	N/A
Commission	37, 37	Yes	Yes
Commission	37, 41	Yes	Yes
Commission	38, 32	No	N/A
Commission	38, 33	No	N/A
Commission	38, 37	No	N/A
Commission	38, 41	Yes	Yes

9.1.4 Du-paths for totalLocks

The du-paths for totalLocks will lead us to typical test cases for computations. With two defining nodes (DEF(totalLocks, 10) and DEF(totalLocks, 16)) and three usage nodes (USE(totalLocks, 16), USE(totalLocks, 21), USE(totalLocks, 24)), we might expect six du-paths. Let us take a closer look.

Path p5 = <10, 11, 12, 13, 14, 15, 16> is a du-path in which the initial value of totalLocks (0) has a computation use. This path is definition clear. The next path is problematic:

p6 = <10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 14, 21>



```
10 totalLocks = 0
11 totalStocks = 0
12 totalBarrels = 0
13 Input(locks)
'locks = –1 signals end of data
14 While NOT(locks = –1)
15    Input(stocks, barrels)
16    totalLocks = totalLocks + locks
17    totalStocks = totalStocks + stocks
18    totalBarrels = totalBarrels + barrels
19    Input(locks)
20 EndWhile
21 Output("Locks sold:," totalLocks)
22 Output("Stocks sold:," totalStocks)
23 Output("Barrels sold:," totalBarrels)
24 lockSales = lockPrice*totalLocks
```

Figure 9.4 Du-paths for totalLocks.

Path p6 ignores the possible repetition of the while loop. We could highlight this by noting that the subpath <16, 17, 18, 19, 20, 14, 15> might be traversed several times. Ignoring this for now, we still have a du-path that fails to be definition clear. If a problem occurs with the value of totalLocks at node 21 (the Output statement), we should look at the intervening DEF(totalLocks, 16) node.

The next path contains p6; we can show this by using a path name in place of its corresponding node sequence:

p7 = <10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 14, 21, 22, 23, 24>
p7 = <p6, 22, 23, 24>

Du-path p7 is not definition clear because it includes node 16. Subpaths that begin with node 16 (an assignment statement) are interesting. The first, <16, 16>, seems degenerate. If we "expanded" it into machine code, we would be able to separate the define and usage portions. We will disallow these as du-paths. Technically, the usage on the right-hand side of the assignment refers to a value defined at node 10 (see path p5). The remaining two du-paths are both subpaths of p7:

p8 = <16, 17, 18, 19, 20, 14, 21>
p9 = <16, 17, 18, 19, 20, 14, 21, 22, 23, 24>

Both are definition clear, and both have the loop iteration problem we discussed before. The du-paths for totalLocks are shown in Figure 9.4.

9.1.5 Du-paths for Sales

There is one defining node for sales; therefore, all the du-paths with respect to sales must be definition clear. They are interesting because they illustrate predicate and computation uses. The first three du-paths are easy:

p10 = <27, 28>
p11 = <27, 28, 29>
p12 = <27, 28, 29, 30, 31, 32, 33>

Notice that p12 is a definition-clear path with three usage nodes; it also contains paths p10 and p11. If we were testing with p12, we know we would also have covered the other two paths. We will revisit this toward the end of the chapter.

The IF, ELSE IF logic in statements 29 through 40 highlights an ambiguity in the original research. Two choices for du-paths begin with path p11: one choice is the path <27, 28, 29, 30, 31, 32, 33>, and the other is the path <27, 28, 29, 34>. The remaining du-paths for sales are

p13 = <27, 28, 29, 34>
p14 = <27, 28, 29, 34, 35, 36, 37>
p15 = <27, 28, 29, 34, 38>

Note that the dynamic view is very compatible with the kind of thinking we used for DD-paths in Chapter 8.

9.1.6 Du-paths for Commission

If you have followed this discussion carefully, you are probably dreading the analysis of du-paths with respect to commission. You are right—it is time for a change of pace. In statements 29 through 41, the calculation of commission is controlled by ranges of the variable sales. Statements 31 to 33 build up the value of commission by using the memory location to hold intermediate values. This is a common programming practice, and it is desirable because it shows how the final value is computed. (We could replace these lines with the statement "commission: = 220 + 0.20 * (sales –1800)," where 220 is the value of 0.10 * 1000 + 0.15 * 800, but this would be hard for a maintainer to understand.) The "built-up" version uses intermediate values, and these will appear as define and usage nodes in the du-path analysis. We decided to disallow du-paths from assignment statements like 31 and 32, so we will just consider du-paths that begin with the three "real" defining nodes: DEF(commission, 33), DEF(commission, 37), and DEF(commission, 39). Only one usage node is used: USE(commission, 41).

9.1.7 Define/Use Test Coverage Metrics

The whole point of analyzing a program with definition/use paths is to define a set of test coverage metrics known as the Rapps–Weyuker data flow metrics (Rapps and Weyuker, 1985). The first three of these are equivalent to three of E.F. Miller's metrics in Chapter 8: All-Paths, All-Edges, and All-Nodes. The others presume that define and usage nodes have been identified for all program variables, and that du-paths have been identified with respect to each variable. In the following definitions, T is a set of paths in the program graph $G(P)$ of a program P, with the set V of variables. It is not enough to take the cross product of the set of DEF nodes with the set of USE nodes for a variable to define du-paths. This mechanical approach can result in infeasible paths. In the next definitions, we assume that the define/use paths are all feasible.

Definition

The set T satisfies the *All-Defs criterion* for the program P if and only if for every variable $v \in V$, T contains definition-clear paths from every defining node of v to a use of v.

Definition

The set T satisfies the *All-Uses criterion* for the program P if and only if for every variable $v \in V$, T contains definition-clear paths from every defining node of v to every use of v, and to the successor node of each USE(v, n).

Definition

The set T satisfies the *All-P-Uses/Some C-Uses criterion* for the program P if and only if for every variable $v \in V$, T contains definition-clear paths from every defining node of v to every predicate use of v; and if a definition of v has no P-uses, a definition-clear path leads to at least one computation use.

Definition

The set T satisfies the *All-C-Uses/Some P-Uses criterion* for the program P if and only if for every variable $v \in V$, T contains definition clear paths from every defining node of v to every computation use of v; and if a definition of v has no C-uses, a definition-clear path leads to at least one predicate use.

Definition

The set T satisfies the *All-DU-paths criterion* for the program P if and only if for every variable $v \in V$, T contains definition-clear paths from every defining node of v to every use of v and to the successor node of each USE(v, n), and that these paths are either single loop traversals or they are cycle free.

These test coverage metrics have several set-theory-based relationships, which are referred to as "subsumption" in Rapps and Weyuker (1985). These relationships are shown in Figure 9.5. We now have a more refined view of structural testing possibilities between the extremes of the (typically unattainable) All-Paths metric and the generally accepted minimum, All-Edges. What good is all this? Define/use testing provides a rigorous, systematic way to examine points at which faults may occur.

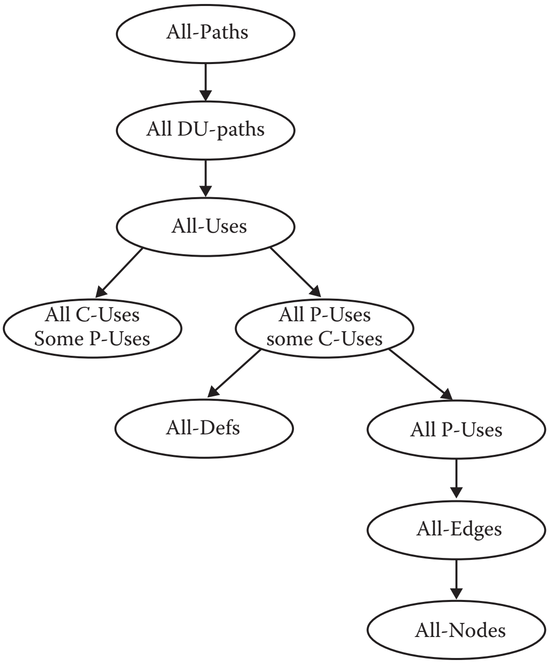


Figure 9.5 Rapps–Weyuker hierarchy of data flow coverage metrics.

9.1.8 Define/Use Testing for Object-Oriented Code

All of the define/use definitions thus far make no mention of where the variable is defined and where it is used. In a procedural code, this is usually assumed to be within a unit, but it can involve procedure calls to improperly coupled units. We might make this distinction by referring to these definitions as "context free"; that is, the places where variables are defined and used are independent. The object-oriented paradigm changes this—we must now consider the define and use locations with respect to class aggregation, inheritance, dynamic binding, and polymorphism. The bottom line is that data flow testing for object-oriented code moves from the unit level to the integration level.

9.2 Slice-Based Testing

Program slices have surfaced and submerged in software engineering literature since the early 1980s. They were proposed in Mark Weiser's dissertation in 1979 (Weiser, 1979), made more generally available in Weiser (1985), used as an approach to software maintenance in Gallagher and Lyle (1991), and more recently used to quantify functional cohesion in Bieman (1994). During the early 1990s, there was a flurry of published activity on slices, including a paper (Ball and Eick, 1994) describing a program to visualize program slices. This latter paper describes a tool used in industry. (Note that it took about 20 years to move a seminal idea into industrial practice.)

Part of the utility and versatility of program slices is due to the natural, intuitively clear intent of the concept. Informally, a program slice is a set of program statements that contributes to, or affects the value of, a variable at some point in a program. This notion of slice corresponds to other disciplines as well. We might study history in terms of slices—US history, European history, Russian history, Far East history, Roman history, and so on. The way such historical slices interact turns out to be very analogous to the way program slices interact.

We will start by growing our working definition of a program slice. We continue with the notation we used for define/use paths: a program P that has a program graph $G(P)$ and a set of program variables V. The first try refines the definition in Gallagher and Lyle (1991) to allow nodes in $P(G)$ to refer to statement fragments.

Definition

Given a program P and a set V of variables in P, a *slice on the variable set V at statement n*, written $S(V, n)$, is the set of all statement fragments in P that contribute to the values of variables in V at node n.

One simplifying notion—in our discussion, the set V of variables consists of a single variable, v. Extending this to sets of more than one variable is both obvious and cumbersome. For sets V with more than one variable, we just take the union of all the slices on the individual variables of V. There are two basic questions about program slices, whether they are backward or forward slices, and whether they are static or dynamic. Backward slices refer to statement fragments that contribute to the value of v at statement n. Forward slices refer to all the program statements that are affected by the value of v and statement n. This is one place where the define/use notions are helpful. In a backward slice $S(v, n)$, statement n is nicely understood as a Use node of the variable v, that is, Use(v, n). Forward slices are not as easily described, but they certainly depend on predicate uses and computation uses of the variable v.

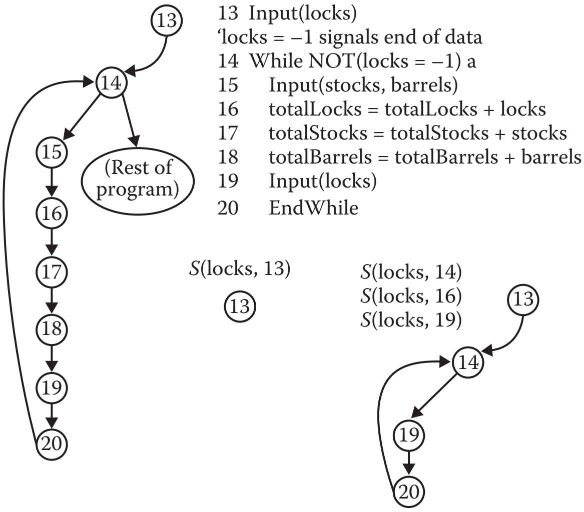
The static/dynamic dichotomy is more complex. We borrow two terms from database technology to help explain the difference. In database parlance, we can refer to the intension and extensions of a database. The intension (it is unique) is the fundamental database structure, presumably expressed in a data modeling language. Populating a database creates an extension, and changes to a populated database all result in new extensions. With this in mind, a static backward slice $S(v, n)$ consists of all the statements in a program that determine the value of variable v at statement n, independent of values used in the statements. Dynamic slices refer to execution–time execution of portions of a static slice with specific values of all variables in $S(v, n)$. This is illustrated in Figures 9.6 and 9.7.

Listing elements of a slice $S(V, n)$ will be cumbersome because, technically, the elements are program statement fragments. It is much simpler to list the statement fragment numbers in $P(G)$, so we make the following trivial change.

Definition

Given a program P and a program graph $G(P)$ in which statements and statement fragments are numbered, and a set V of variables in P, *the static, backward slice on the variable set V at statement fragment n*, written $S(V, n)$, is the set of node numbers of all statement fragments in P that contribute to the values of variables in V at statement fragment n.

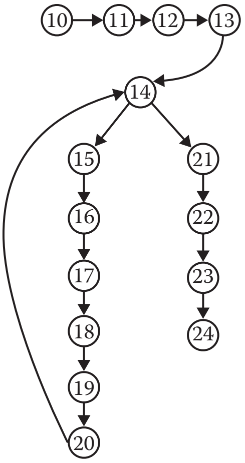
The idea of program slicing is to separate a program into components that have some useful (functional) meaning. Another refinement is whether or not a program slice is executable. Adding all the data declaration statements and other syntactically necessary statements clearly increases the size of a slice, but the full version can be compiled and separately executed and tested. Further, such compilable slices can be "spliced" together (Gallagher and Lyle, 1991) as a bottom–up way to develop a program. As a test of clear diction, Gallagher and Lyle suggest the term "slice splicing." In a sense, this is a precursor to agile programming. The alternative is to just consider program fragments, which we do here for space and clarity considerations. Eventually, we will develop a



```
13  Input(locks)
'locks = –1 signals end of data
14  While NOT(locks = –1) a
15      Input(stocks, barrels)
16      totalLocks = totalLocks + locks
17      totalStocks = totalStocks + stocks
18      totalBarrels = totalBarrels + barrels
19      Input(locks)
20      EndWhile
```

Figure 9.6 Selected slices on locks.

```
10 totalLocks = 0
11 totalStocks = 0
12 totalBarrels = 0
13 Input(locks)
'locks = –1 signals end of data
14 While NOT(locks = –1)
15    Input(stocks, barrels)
16    totalLocks = totalStocks + stocks
17    totalStocks = totalStocks + stocks
18    totalBarrels = totalBarrels + barrels
19    Input(locks)
20 EndWhile
21 Output("Locks sold:," totalLocks)
22 Output("Stocks sold:," totalStocks)
23 Output("Barrels sold:," totalBarrels)
24 lockSales = lockPrice*totalLocks
```



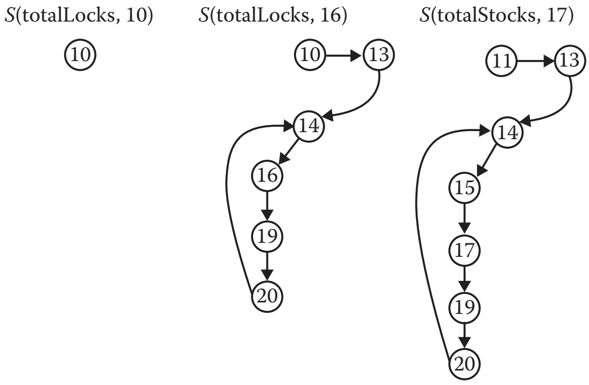


Figure 9.7 Selected slices in a loop.

lattice (a directed, acyclic graph) of static slices, in which nodes are slices and edges correspond to the subset relationship.

The "contribute" part is more complex. In a sense, data declaration statements have an effect on the value of a variable. For now, we only include all executable statements. The notion of contribution is partially clarified by the predicate (P-use) and computation (C-use) usage distinction of Rapps and Weyuker (1985), but we need to refine these forms of variable usage. Specifically, the USE relationship pertains to five forms of usage:

P-use	used in a predicate (decision)
C-use	used in computation
O-use	used for output
L-use	used for location (pointers, subscripts)
I-use	iteration (internal counters, loop indices)

Most of the literature on program slices just uses P-uses and C-uses. While we are at it, we identify two forms of definition nodes:

I-def	defined by input
A-def	defined by assignment

Recall our simplification that the slice *S*(*V*, *n*) is a slice on one variable; that is, the set *V* consists of a single variable, *v*. If statement fragment *n* is a defining node for *v*, then *n* is included in the slice. If statement fragment *n* is a usage node for *v*, then *n* is not included in the slice. If a statement is both a defining and a usage node, then it is included in the slice. In a static slice, P-uses and C-uses of other variables (not the *v* in the slice set *V*) are included to the extent that their execution affects the value of the variable *v*. As a guideline, if the value of *v* is the same whether a statement fragment is included or excluded, exclude the statement fragment.

L-use and I-use variables are typically invisible outside their units, but this hardly precludes the problems such variables often create. Another judgment call: here (with some peril), we choose to exclude these from the intent of "contribute." Thus, O-use, L-use, and I-use nodes are excluded from slices.

9.2.1 Example

The commission problem is used in this book because it contains interesting data flow properties, and these are not present in the triangle problem (nor in NextDate). In the following, except where specifically noted, we are speaking of static backward slices and we only include nodes corresponding to executable statement fragments. The examples refer to the source code for the commission problem in Figure 9.1. There are 42 "interesting" static backward slices in our example. They are named in Table 9.5. We will take a selective look at some interesting slices.

The first six slices are the simplest—they are the nodes where variables are initialized.

Table 9.5 Slices in Commission Problem

S_1: *S*(lockPrice, 7)	S_{15}: *S*(barrels, 18)	S_{29}: *S*(barrelSales, 26)
S_2: *S*(stockPrice, 8)	S_{16}: *S*(totalBarrels, 18)	S_{30}: *S*(sales, 27)
S_3: *S*(barrelPrice, 9)	S_{17}: *S*(locks, 19)	S_{31}: *S*(sales, 28)
S_4: *S*(totalLocks, 10)	S_{18}: *S*(totalLocks, 21)	S_{32}: *S*(sales, 29)
S_5: *S*(totalStocks, 11)	S_{19}: *S*(totalStocks, 22)	S_{33}: *S*(sales, 33)
S_6: *S*(totalBarrels, 12)	S_{20}: *S*(totalBarrels, 23)	S_{34}: *S*(sales, 34)
S_7: *S*(locks, 13)	S_{21}: *S*(lockPrice, 24)	S_{35}: *S*(sales, 37)
S_8: *S*(locks, 14)	S_{22}: *S*(totalLocks, 24)	S_{36}: *S*(sales, 39)
S_9: *S*(stocks, 15)	S_{23}: *S*(lockSales, 24)	S_{37}: *S*(commission, 31)
S_{10}: *S*(barrels, 15)	S_{24}: *S*(stockPrice, 25)	S_{38}: *S*(commission, 32)
S_{11}: *S*(locks, 16)	S_{25}: *S*(totalStocks, 25)	S_{39}: *S*(commission, 33)
S_{12}: *S*(totalLocks, 16)	S_{26}: *S*(stockSales, 25)	S_{40}: *S*(commission, 36)
S_{13}: *S*(stocks, 17)	S_{27}: *S*(barrelPrice, 26)	S_{41}: *S*(commission, 37)
S_{14}: *S*(totalStocks, 17)	S_{28}: *S*(totalBarrels, 26)	S_{42}: *S*(commission, 39)

S_1: S(lockPrice, 7) = {7}
S_2: S(stockPrice, 8) = {8}
S_3: S(barrelPrice, 9) = {9}
S_4: S(totalLocks, 10) = {10}
S_5: S(totalStocks, 11) = {11}
S_6: S(totalBarrels, 12) = {12}

Slices 7 through 17 focus on the sentinel controlled while loop in which the totals for locks, stocks, and barrels are accumulated. The locks variable has two uses in this loop: a P-use at fragment 14 and C-use at statement 16. It also has two defining nodes, at statements 13 and 19. The stocks and barrels variables have a defining node at 15, and computation uses at nodes 17 and 18, respectively. Notice the presence of all relevant statement fragments in slice 8. The slices on locks are shown in Figure 9.6.

S_7: S(locks, 13) = {13}
S_8: S(locks, 14) = {13, 14, 19, 20}
S_9: S(stocks, 15) = {13, 14, 15, 19, 20}
S_{10}: S(barrels, 15) = {13, 14, 15, 19, 20}
S_{11}: S(locks, 16) = {13, 14, 19, 20}
S_{12}: S(totalLocks, 16) = {10, 13, 14, 16, 19, 20}
S_{13}: S(stocks, 17) = {13, 14, 15, 19, 20}
S_{14}: S(totalStocks, 17) = {11, 13, 14, 15, 17, 19, 20}
S_{15}: S(barrels, 18) = {12, 13, 14, 15, 19, 20}
S_{16}: S(totalBarrels, 18) = {12, 13, 14, 15, 18, 19, 20}
S_{17}: S(locks, 19) = {13, 14, 19, 20}

Slices 18, 19, and 20 are output statements, and none of the variables is defined; hence, the corresponding statements are not included in these slices.

S_{18}: S(totalLocks, 21) = {10, 13, 14, 16, 19, 20}
S_{19}: S(totalStocks, 22) = {11, 13, 14, 15, 17, 19, 20}
S_{20}: S(totalBarrels, 23) = {12, 13, 14, 15, 18, 19, 20}

Slices 21 through 30 deal with the calculation of the variable sales. As an aside, we could simply write S_{30}: S(sales, 27) = $S_{23} \cup S_{26} \cup S_{29} \cup \{27\}$. This is more like the form that Weiser (1979) refers to in his dissertation—a natural way to think about program fragments. Gallagher and Lyle (1991) echo this as a thought pattern among maintenance programmers. This also leads to Gallager's "slice splicing" concept. Slice S_{23} computes the total lock sales, S_{25} the total stock sales, and S_{28} the total barrel sales. In a bottom–up way, these slices could be separately coded and tested, and later spliced together. "Splicing" is actually an apt metaphor—anyone who has ever spliced a twisted rope line knows that splicing involves carefully merging individual strands at just the right places. (See Figure 9.7 for the effect of looping on a slice.)

S_{21}: S(lockPrice, 24) = {7}
S_{22}: S(totalLocks, 24) = {10, 13, 14, 16, 19, 20}
S_{23}: S(lockSales, 24) = {7, 10, 13, 14, 16, 19, 20, 24}
S_{24}: S(stockPrice, 25) = {8}

S_{25}: S(totalStocks, 25) = {11, 13, 14, 15, 17, 19, 20}
S_{26}: S(stockSales, 25) = {8, 11, 13, 14, 15, 17, 19, 20, 25}
S_{27}: S(barrelPrice, 26) = {9}
S_{28}: S(totalBarrels, 26) = {12, 13, 14, 15, 18, 19, 20}
S_{29}: S(barrelSales, 26) = {9, 12, 13, 14, 15, 18, 19, 20, 26}
S_{30}: S(sales, 27) = {7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 24, 25, 26, 27}

Slices 31 through 36 are identical. Slice S_{31} is an O-use of sales; the others are all C-uses. Since none of these changes the value of sales defined at S_{30}, we only show one set of statement fragment numbers here.

S_{31}: S(sales, 28) = {7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 24, 25, 26, 27}

The last seven slices deal with the calculation of commission from the value of sales. This is literally where it all comes together.

S_{37}: S(commission, 31) = {7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 24, 25, 26, 27, 29, 30, 31}
S_{38}: S(commission, 32) = {7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 24, 25, 26, 27, 29, 30, 31, 32}
S_{39}: S(commission, 33) = {7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 24, 25, 26, 27, 29, 30, 31, 32, 33}
S_{40}: S(commission, 36) = {7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 24, 25, 26, 27, 29, 34, 35, 36}
S_{41}: S(commission, 37) = {7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 24, 25, 26, 27, 29, 34, 35, 36, 37}
S_{42}: S(commission, 39) = {7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 24, 25, 26, 27, 29, 34, 38, 39}
S_{43}: S(commission, 41) = {7, 8, 9 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 24, 25, 26, 27, 29, 30, 31, 32, 33, 34, 35, 36, 37, 39}

Looking at slices as sets of fragment numbers (Figure 9.8) is correct in terms of our definition, but it is also helpful to see how slices are composed of sets of previous slices. We do this next, and show the final lattice in Figure 9.9.

S_1: S(lockPrice, 7) = {7}
S_2: S(stockPrice, 8) = {8}
S_3: S(barrelPrice, 9) = {9}
S_4: S(totalLocks, 10) = {10}
S_5: S(totalStocks, 11) = {11}
S_6: S(totalBarrels, 12) = {12}
S_7: S(locks, 13) = {13}
S_8: S(locks, 14) = $S_7 \cup \{14, 19, 20\}$
S_9: S(stocks, 15) = $S_8 \cup \{15\}$
S_{10}: S(barrels, 15) = S_8
S_{11}: S(locks, 16) = S_8
S_{12}: S(totalLocks, 16) = $S_4 \cup S_{11} \cup \{16\}$
S_{13}: S(stocks, 17) = $S_9 = \{13, 14, 19, 20\}$

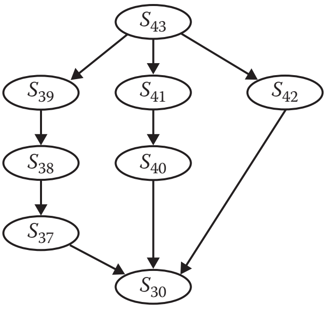

S43
S39
S41
S42
S38
S40
S37
S30

Figure 9.8 Partial lattice of slices on commission.

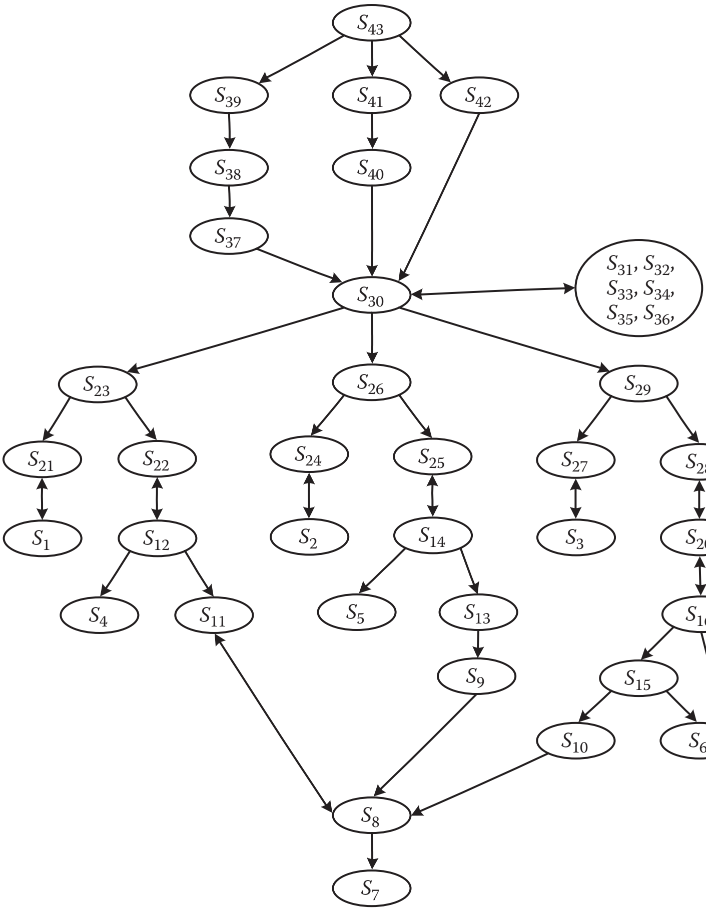

S43
S39
S41
S42
S38
S40
S37
S30
S31, S32, S33, S34, S35, S36,
S23
S26
S29
S21
S22
S24
S25
S27
S28
S1
S12
S2
S14
S3
S20
S4
S11
S5
S13
S16
S9
S15
S10
S6
S8
S7

Figure 9.9 Full lattice on commission.

S_{14}: S(totalStocks, 17) $= S_5 \cup S_{13} \cup \{17\}$
S_{15}: S(barrels, 18) $= S_6 \cup S_{10}$
S_{16}: S(totalBarrels, 18) $= S_6 \cup S_{15} \cup \{18\}$
S_{18}: S(totalLocks, 21) $= S_{12}$

$S_{19}: S(\text{totalStocks}, 22) = S_{14}$
$S_{20}: S(\text{totalBarrels}, 23) = S_{16}$
$S_{21}: S(\text{lockPrice}, 24) = S_1$
$S_{22}: S(\text{totalLocks}, 24) = S_{12}$
$S_{23}: S(\text{lockSales}, 24) = S_{21} \cup S_{22} \cup \{24\}$
$S_{24}: S(\text{stockPrice}, 25) = S_2$
$S_{25}: S(\text{totalStocks}, 25) = S_{14}$
$S_{26}: S(\text{stockSales}, 25) = S_{24} \cup S_{25} \cup \{25\}$
$S_{27}: S(\text{barrelPrice}, 26) = S_3$
$S_{28}: S(\text{totalBarrels}, 26) = S_{20}$
$S_{29}: S(\text{barrelSales}, 26) = S_{27} \cup S_{28} \cup \{26\}$
$S_{30}: S(\text{sales}, 27) = S_{23} \cup S_{26} \cup S_{29} \cup \{27\}$
$S_{31}: S(\text{sales}, 28) = S_{30}$
$S_{32}: S(\text{sales}, 29) = S_{30}$
$S_{33}: S(\text{sales}, 33) = S_{30}$
$S_{34}: S(\text{sales}, 34) = S_{30}$
$S_{35}: S(\text{sales}, 37) = S_{30}$
$S_{36}: S(\text{sales}, 39) = S_{30}$
$S_{37}: S(\text{commission}, 31) = S_{30} \cup \{29, 30, 31\}$
$S_{38}: S(\text{commission}, 32) = S_{37} \cup \{32\}$
$S_{39}: S(\text{commission}, 33) = S_{38} \cup \{33\}$
$S_{40}: S(\text{commission}, 36) = S_{30} \cup \{29, 34, 35, 36\}$
$S_{41}: S(\text{commission}, 37) = S_{40} \cup \{37\}$
$S_{42}: S(\text{commission}, 39) = S_{30} \cup \{29, 34, 38, 39\}$
$S_{43}: S(\text{commission}, 41) = S_{39} \cup S_{41} \cup S_{42}$

Several of the connections in Figure 9.9 are double-headed arrows indicating set equivalence. (Recall from Chapter 3 that if $A \subseteq B$ and $B \subseteq A$, then $A = B$.) We can clean up Figure 9.9 by removing these, and thereby get a better lattice. The result of doing this is in Figure 9.10.

9.2.2 Style and Technique

When we analyze a program in terms of interesting slices, we can focus on parts of interest while disregarding unrelated parts. We could not do this with du-paths—they are sequences that include statements and variables that may not be of interest. Before discussing some analytic techniques, we will first look at "good style." We could have built these stylistic precepts into the definitions, but then the definitions are more restrictive than necessary.

1. Never make a slice $S(V, n)$ for which variables v of V do not appear in statement fragment n. This possibility is permitted by the definition of a slice, but it is bad practice. As an example, suppose we defined a slice on the locks variable at node 27. Defining such slices necessitates tracking the values of all variables at all points in the program.
2. Make slices on one variable. The set V in slice $S(V, n)$ can contain several variables, and sometimes such slices are useful. The slice $S(V, 27)$ where

$$V = \{\text{lockSales}, \text{stockSales}, \text{barrelSales}\}$$

contains all the elements of the slice S_{30}: $S(\text{sales}, 27)$ except statement 27.

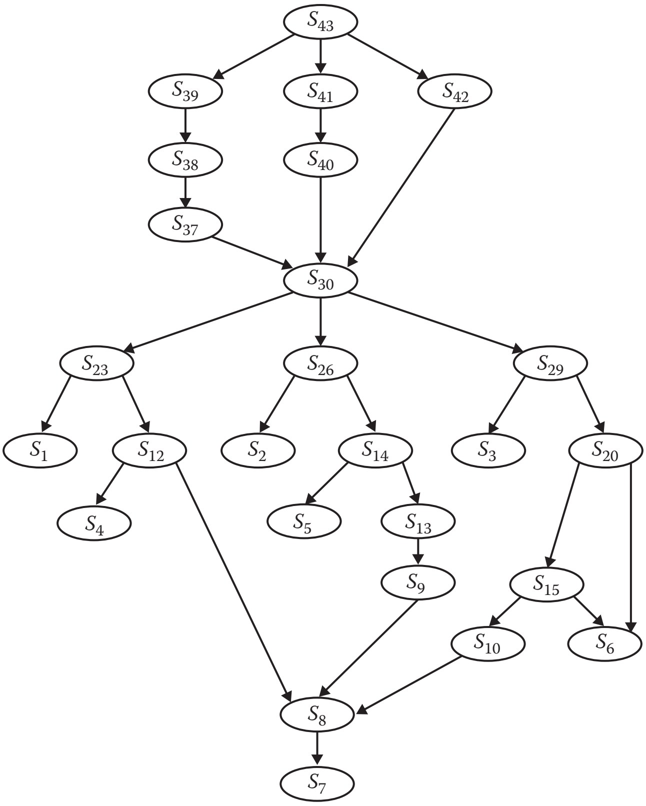


Figure 9.10 Simplified lattice on commission.

3. Make slices for all A-def nodes. When a variable is computed by an assignment statement, a slice on the variable at that statement will include (portions of) all du-paths of the variables used in the computation. Slice S_{30}: S(sales, 27) is a good example of an A-def slice. Similarly for variables defined by input statements (I-def nodes), such as S_{10}: S(barrels, 15).
4. There is not much reason to make slices on variables that occur in output statements. Slices on O-use variables can always be expressed as unions of slices on all the A-defs (and I-defs) of the O-use variable.
5. Make slices for P-use nodes. When a variable is used in a predicate, the slice on that variable at the decision statement shows how the predicate variable got its value. This is very useful in decision-intensive programs such as the triangle program and NextDate.
6. Consider making slices compilable. Nothing in the definition of a slice requires that the set of statements is compilable; however, if we make this choice, it means that a set of compiler directive and data declaration statements is a subset of every slice. If we added this same set of statements to all the slices we made for the commission program, our lattices remain undisturbed; however, each slice is separately compilable (and therefore executable). In Chapter 1, we suggested that good testing practices lead to better programming practices. Here, we have a good example. Think about developing programs in terms of compilable slices. If we did this, we could code a slice and immediately test it. We can then code and test other slices and merge them (sometimes called "slice splicing") into a fairly solid program. This is done in Section 9.2.3.

9.2.3 Slice Splicing

The commission program is deliberately small, yet it suffices to illustrate the idea of “slice splicing.” In Figures 9.11 through 9.14, the commission program is split into four slices. Statement fragment numbers and the program graphs are as they were in Figure 9.1. Slice 1 contains the input while loop controlled by the locks variable. This is a good starting point because both Slice 2 and Slice 3 use the loop to get input values for stocks and barrels, respectively. Slices 1, 2, and 3 each culminate in a value of sales, which is the starting point for Slice 4, which computes the commission bases on the value of sales.

This is overkill for this small example; however, the idea extends perfectly to larger programs. It also illustrates the basis for program comprehension needed in software maintenance. Slices allow the maintenance programmer to focus on the issues at hand and avoid the extraneous information that would be in du-paths.

```
 1  Program Slice1 (INPUT,OUTPUT)
 2  Dim locks As Integer
 3  Dim lockPrice As Real
 4  Dim totalLocks As Integer
 5  Dim lockSales As Real
 6  Dim sales As Real
 7  lockPrice = 45.0
13  Input(locks)
'locks = –1 signals end of data
14  While NOT(locks = –1)
16  totalLocks = totalLocks + locks
19  Input(locks)
20  EndWhile
21  Output(“Locks sold:”, totalLocks)
24  lockSales = lockPrice*totalLocks
27  sales = lockSales
28  Output(“Total sales: ”, sales)
```

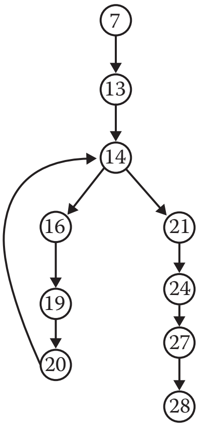


Figure 9.11 Slice 1.

```
 1  Program Slice2 (INPUT,OUTPUT)
 2  Dim locks, stocks As Integer
 3  Dim stockPrice As Real
 4  Dim totalStocks As Integer
 5  Dim stockSales As Real
 6  Dim sales As Real
 8  stockPrice = 30.0
11  totalStocks = 0
13  Input(locks)
'locks = –1 signals end of data
14  While NOT(locks = –1)
15    Input(stocks)
17    totalStocks = totalStocks + stocks
19    Input(locks)
20  EndWhile
22  Output(“Stocks sold:,” totalStocks)
25  stockSales = stockPrice*totalStockd
27  sales = stockSales
28  Output(“Total sales:,” sales)
```

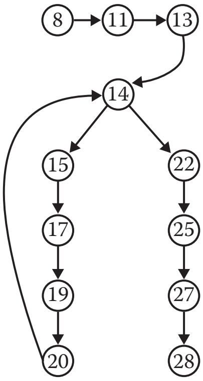


Figure 9.12 Slice 2.

```
 1  Program Slice3 (INPUT,OUTPUT)
 2  Dim locks, barrels As integer
 3  Dim barrelPrice As Real
 4  Dim totalBarrels As Integer
 5  Dim barrelSales As Real
 6  Dim sales As Real
 9  barrelPrice = 25.0
12  totalBarrels = 0
13  Input(locks)
'locks = –1 signals end of data
14  While NOT(locks = –1)
15    Input(barrels)
18    totalBarrels = totalBarrels + barrels
19    Input(locks)
20  EndWhile
23  Output("Barrels sold:," totalBarrels)
26  barrelSales = barrelsPrice * totalBarrles
27  sales = barrelSales
28  Output("Total sales:," sales)
```

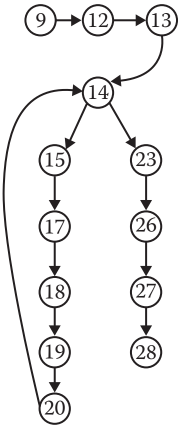


Figure 9.13 Slice 3.

```
 1  Program Slice4 (INPUT,OUTPUT)
 6  Dim sales, commission As Real
29  If (sales>1800.0)
30    Then
31     commission = 0.10 * 1000.0
32     commission = commission + 0.15*800.0
33     commission = commission + 0.20*(sales–1800.0)
34    Else If (sales > 1000.0)
35       Then
36     commission = 0.10 * 1000.0
37     commission = commission + 0.15*(sales–1000.0)
38     Else
39     commission = 0.10 * sales
40    EndIf
41  EndIf
42  Output("Commission is $," commission)
43  End Commission
```

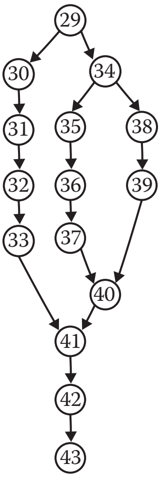


Figure 9.14 Slice 4.

9.3 Program Slicing Tools

Any reader who has gone carefully through the preceding section will agree that program slicing is not a viable manual approach. I hesitate assigning a slicing exercise to my university students because the actual learning is only marginal in terms of the time spent with good tools; however, program slicing has its place. There are a few program slicing tools; most are academic or experimental, but there are a very few commercial tools. (See Hoffner [1995] for a dated comparison.)

The more elaborate tools feature interprocedural slicing, something clearly useful for large systems. Much of the market uses program slicing to improve the program comprehension that maintenance programmers need. One, JSlice, will be appropriate for object-oriented software. Table 9.6 summarizes a few program slicing tools.

Table 9.6 Selected Program Slicing Tools

Tool/Product	*Language*	*Static/Dynamic?*
Kamkar	Pascal	Dynamic
Spyder	ANSI C	Dynamic
Unravel	ANSI C	Static
CodeSonar®	C, C++	Static
Indus/Kaveri	Java	Static
JSlice	Java	Dynamic
SeeSlice	C	Dynamic

EXERCISES

1. Think about the static versus dynamic ambiguity of du-paths in terms of DD-paths. As a start, what DD-paths are found in the du-paths p12, p13, and p14 for sales?
2. Try to merge some of the DD-path-based test coverage metrics into the Rapps–Weyuker hierarchy shown in Figure 9.5.
3. List the du-paths for the commission variable.
4. Our discussion of slices in this chapter has actually been about "backward slices" in the sense that we are always concerned with parts of a program that contribute to the value of a variable at a certain point in the program. We could also consider "forward slices" that refer to parts of the program where the variable is used. Compare and contrast forward slices with du-paths.

References

Bieman, J.M. and Ott, L.M., Measuring functional cohesion, *IEEE Transactions on Software Engineering*, Vol. SE-20, No. 8, August 1994, pp. 644–657.

Ball, T. and Eick, S.G., Visualizing program slices, *Proceedings of the 1994 IEEE Symposium on Visual Languages*, St. Louis, MO, October 1994, pp. 288–295.

Clarke, L.A. et al., A formal evaluation of dataflow path selection criteria, *IEEE Transactions on Software Engineering*, Vol. SE-15, No. 11, November 1989, pp. 1318–1332.

Gallagher, K.B. and Lyle, J.R., Using program slicing in software maintenance, *IEEE Transactions on Software Engineering*, Vol. SE-17, No. 8, August 1991, pp. 751–761.

Hoffner, T., *Evaluation and Comparison of Program Slicing Tools,* Technical Report, Dept. of Computer and Information Science, Linkoping University, Sweden, 1995.

Rapps, S. and Weyuker, E.J., Selecting software test data using dataflow information, *IEEE Transactions on Software Engineering,* Vol. SE-11, No. 4, April 1985, pp. 367–375.

Weiser, M., *Program Slices: Formal Psychological and Practical Investigations of an Automatic Program Abstraction Method.* PhD thesis, University of Michigan, Ann Arbor, MI. 1979.

Weiser, M.D., Program slicing, *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 4, April 1988, pp. 352–357.

Chapter 10

Retrospective on Unit Testing

When should unit testing stop? Here are some possible answers:

1. When you run out of time
2. When continued testing causes no new failures
3. When continued testing reveals no new faults
4. When you cannot think of any new test cases
5. When you reach a point of diminishing returns
6. When mandated coverage has been attained
7. When all faults have been removed

Unfortunately, the first answer is all too common, and the seventh cannot be guaranteed. This leaves the testing craftsperson somewhere in the middle. Software reliability models provide answers that support the second and third choices; both of these have been used with success in industry. The fourth choice is curious: if you have followed the precepts and guidelines we have been discussing, this is probably a good answer. On the other hand, if the reason is due to a lack of motivation, this choice is as unfortunate as the first. The point of diminishing returns choice has some appeal: it suggests that serious testing has continued, and the discovery of new faults has slowed dramatically. Continued testing becomes very expensive and may reveal no new faults. If the cost (or risk) of remaining faults can be determined, the trade-off is clear. (This is a big IF.) We are left with the coverage answer, and it is a pretty good one. In this chapter, we will see how using structural testing as a cross-check on functional testing yields powerful results. First, we take a broad brush look at the unit testing methods we studied. Metaphorically, this is pictured as a pendulum that swings between extremes. Next, we follow one swing of the pendulum from the most abstract form of code-based testing through strongly semantic-based methods, and then back toward the very abstract shades of specification-based testing. We do this tour with the triangle program. After that, some recommendations for both forms of unit testing, followed by another case study—this time of an automobile insurance example.

10.1 The Test Method Pendulum

As with many things in life, there is a pendulum that swings between two extremes. The test method pendulum swings between two extremes of low semantic content—from strictly topological to purely functional. As testing methods move away from the extremes and toward the center, they become, at once, both more effective and more difficult (see Figure 10.1).

On the code-based side, path-based testing relies on the connectivity of a program graph—the semantic meaning of the nodes is lost. A program graph is a purely topological abstraction of the code, and is nearly devoid of code meaning—only the control flow remains. This gives rise to program paths that can never be recognized as infeasible by automated means. Moving to data flow testing, the kinds of dependencies that typically create infeasible paths can often be detected. Finally, when viewed in terms of slices, we arrive as close as we can to the semantic meaning of the code.

On the specification-based side, testing based only on boundary values of the variables is vulnerable to severe gaps and redundancies, neither of which can be known in purely specification-based testing. Equivalence class testing uses the "similar treatment" idea to identify classes, and in doing so, uses more of the semantic meaning of the specification. Finally, decision table testing uses both necessary and impossible combinations of conditions, derived from the specification, to deal with complex logical considerations.

On both sides of the testing pendulum, test case identification becomes easier as we move toward the extremes. It also becomes less effective. As testing techniques move toward higher semantic meaning, they become more difficult to automate—and more effective. Hmmm … could it be that, when he wrote "The Pit and the Pendulum," Edgar Allan Poe was actually thinking about testing as a pit, and methods as a pendulum? You decide. Meanwhile, these ideas are approximated in Figure 10.2.

These graphs need some elaboration. Starting with program graph testing, notice that the nodes contain absolutely no semantic information about the statement fragments—and the edges just describe whether one fragment can be executed after a predecessor fragment. Paths in a

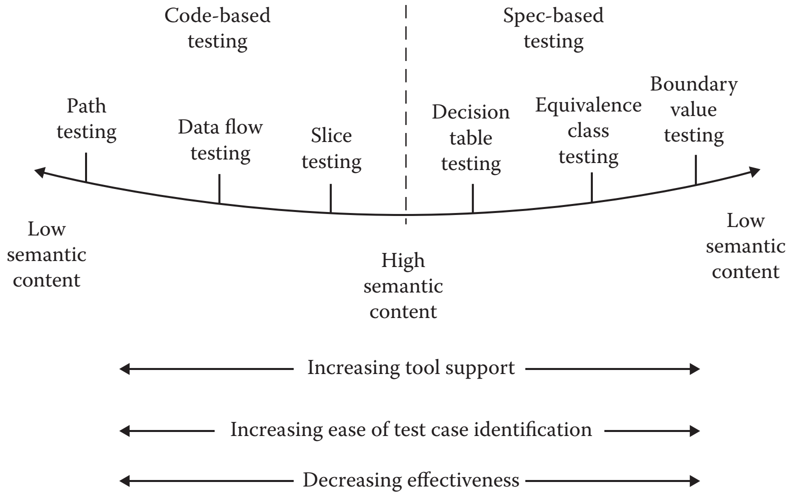

Code-based testing
Spec-based testing
Path testing
Data flow testing
Slice testing
Decision table testing
Equivalence class testing
Boundary value testing
Low semantic content
High semantic content
Low semantic content
Increasing tool support
Increasing ease of test case identification
Decreasing effectiveness

Figure 10.1 Test method pendulum.

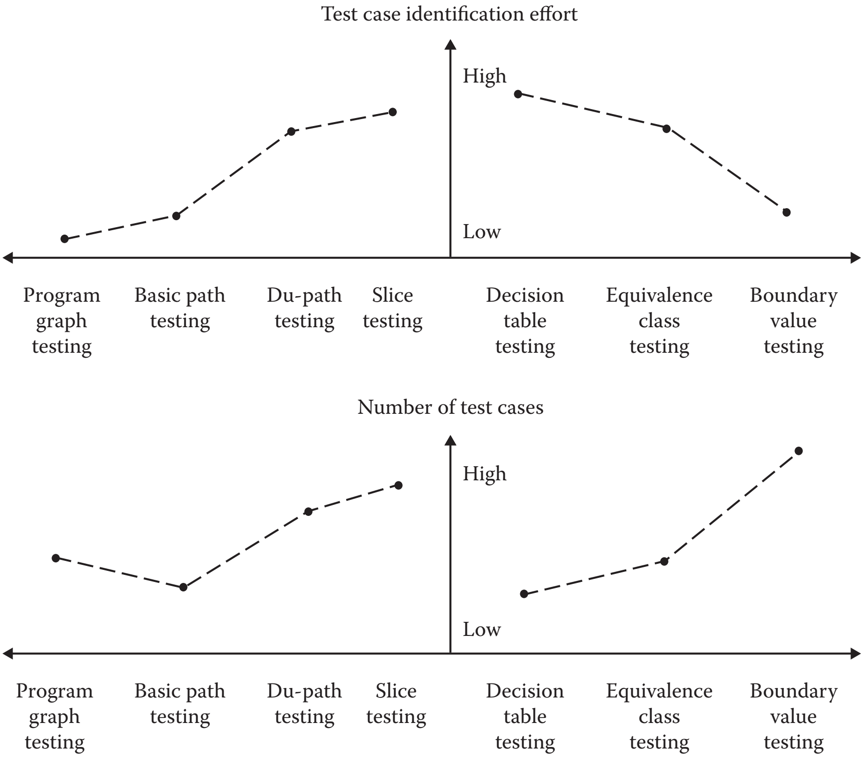


Figure 10.2 Effort and efficacy of unit test methods.

program graph are all topologically possible—in fact, they can be generated mathematically with Warshall's algorithm. The problem is that the set of topologically possible paths includes both feasible and infeasible paths, as we discussed in Chapter 8. Moving in the direction of McCabe's basis path testing adds a little semantic content. The recommended starting point is a mainline path that represents common unit functionality. The basis path method runs into trouble after that due to the heuristic of simply "flipping" decisions as they are encountered on the starting point path. This also leads to the possibility of infeasible paths. When testing moves to the define/use domain, we use more semantic meaning. We follow where values of variables are defined and later used. The distinction between du-paths and definition-clear du-paths gives the tester even more semantic information. Finally, backward slices do two things, they eliminate unwanted detail, and thereby focus attention exactly where it is needed—all the statements affecting the value of a variable at a given point in a program. The program slicing literature contains extensive discussions of automatic slicing algorithms that are beyond the scope of this book.

On the specification-based side, the various forms of boundary value testing are shown as the most abstract. All test cases are derived from properties of the input space with absolutely no consideration about how the values are used in the unit code. When we move to equivalence class testing, the prime factor that determines a class is the "similar treatment" principle. Clearly, this moves in the direction of semantic meaning. Moving from equivalence class testing to decision table testing is usually done for two reasons: the presence of dependencies among the variables and the possibility of impossible combinations.

The lower half of Figure 10.2 shows that, for specification-based testing, there is a true tradeoff between test case creation effort and test case execution time. If the testing is automated, as in a jUnit environment, this is not a penalty. On the code-based side, as methods get more sophisticated, they concurrently generate more test cases.

The bottom line to this discussion is that the combination of specification-based and code-based methods depends on the nature of the unit being tested, and this is where testers can exhibit craftsmanship.

10.2 Traversing the Pendulum

We will use the triangle program to explore some of the lessons of the testing pendulum. We begin with the FORTRAN-like version so popular in the early literature. We use this implementation here, mostly because it is the most frequently used in testing literature (Brown and Lipov, 1975; Pressman, 1982). The flowchart from Chapter 2 is repeated here in Figure 10.3, and transformed to a directed graph in Figure 10.4. The numbers of the flowchart symbols are preserved as node numbers in the corresponding directed acyclic graph in Figure 10.4.

We can begin to see some of the difficulties when we base testing on a program graph. There are 80 topologically possible paths in Figure 10.4 (and also in Figure 10.3), but only 11 of these are feasible; they are listed in Table 10.1. Since this is at one abstract end of the testing pendulum, we cannot expect any automated help to separate feasible from infeasible paths. Much of the infeasibility is due to the Match variable. Its intent was to reduce the number of decisions. The boxes incrementing the Match variable depend on tests of equality among the three pairs of sides. Of the eight paths from box 1 to box 7, the logically possible values of Match are 0, 1, 2, 3, and 6. The three impossible paths correspond to exactly two pairs of sides being equal, which, by transitivity, is impossible. There are 13 decisions in the flowchart, so the goal of reducing decisions was missed anyway.

What suggests that this is a FORTRAN-like implementation is that, in the early days of FORTRAN programming, memory was expensive and computers were relatively slow. On the basis of the flowchart, a good FORTRAN programmer would compute the sums of pairs (a + b, a + c, and b + c) only once and use these again in the decisions checking the triangle inequality (decisions 8, 9, 10, 14, 17, and 19).

Moving on to basis path testing, we have another problem. The program graph in Figure 10.4 has a cyclomatic complexity of 14. McCabe's basis path method would ask us to find 14 test cases, but there are only 11 feasible paths. Again, this view is too far removed from the semantic meaning of the code to help.

Data flow testing will give us some valuable insights. Consider du-paths on the Match variable. It has four definition nodes, three computation uses, and four predicate uses, so there are 28 possible du-paths. Looking at the definition-clear paths will be a good start to data flow testing. The sides, a, b, and c, have one definition node and nine use nodes. All nine du-paths on these variables will be definition clear. That means very little can happen to these variables unless there is an input problem.

Testing using backward static slices would be a good idea. Although no variable for this appears in the original flowchart, we can postulate a variable, triangleType, that has the four string values shown in boxes 11, 12, 15, and 20. The first slice to test would be *S*(triangleType, 11), which represents the only way to have a scalene triangle. We could test it with three test cases: (a, b, c) = (3, 4, 5), = (4, 5, 3), and = (5, 3, 4). These triplets let each variable take on all three possibilities,

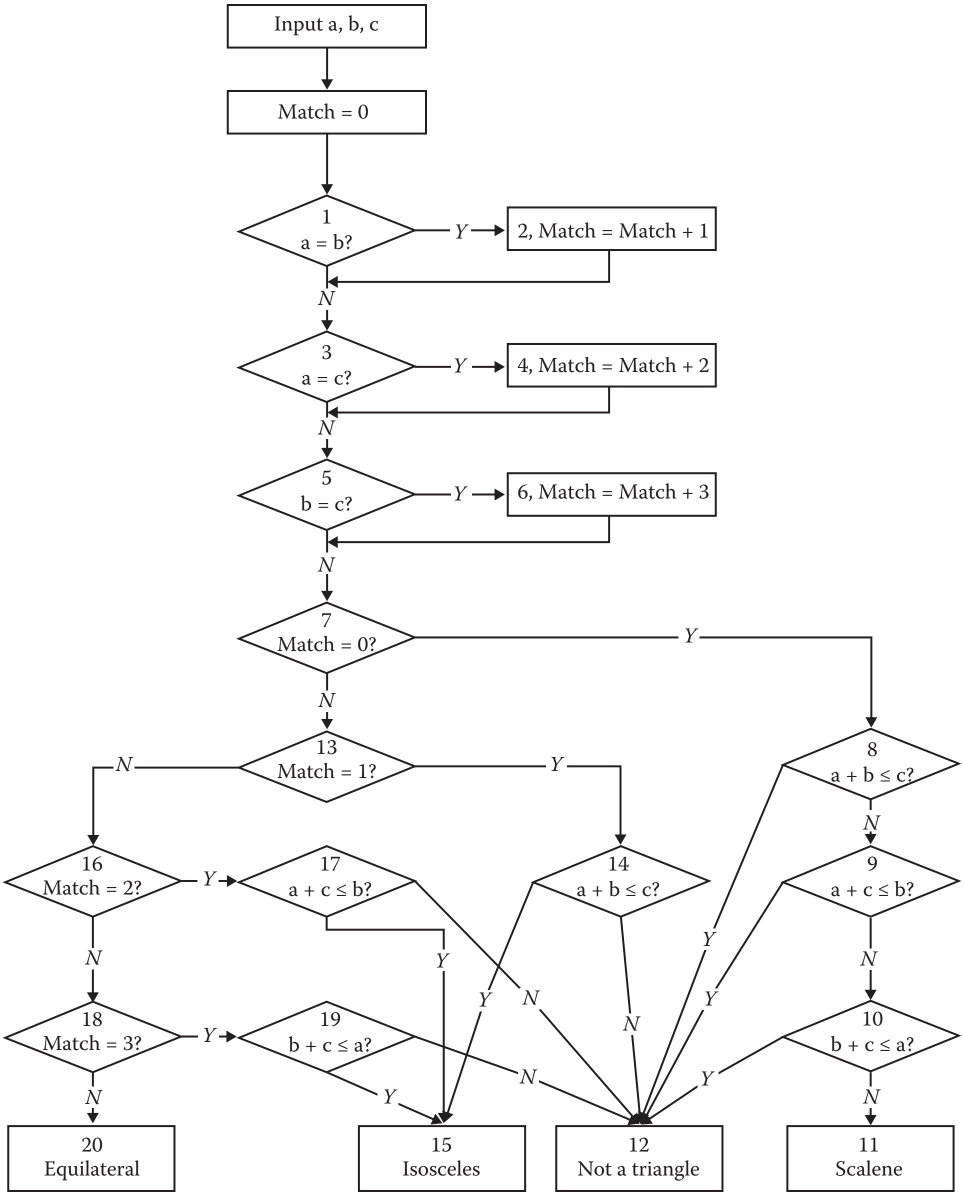

Input a, b, c
Match = 0
1
a = b?
Y
2, Match = Match + 1
N
3
a = c?
Y
4, Match = Match + 2
N
5
b = c?
Y
6, Match = Match + 3
N
7
Match = 0?
Y
N
13
Match = 1?
N
Y
8
a + b ≤ c?
N
16
Match = 2?
Y
17
a + c ≤ b?
14
a + b ≤ c?
9
a + c ≤ b?
N
Y
Y
Y
N
N
Y
N
18
Match = 3?
Y
19
b + c ≤ a?
N
Y
Y
10
b + c ≤ a?
N
N
20
Equilateral
15
Isosceles
12
Not a triangle
11
Scalene

Figure 10.3 Flowchart of FORTRAN-like triangle program.

a little like a Sudoku puzzle. Similar comments apply to the slice *S*(triangleType, 20) where the expected value of triangleType is "Equilateral." Here we would only need one test case, maybe foreshadowing equivalence class testing. The last slices to test would be for isosceles triangles, and then six ways for a, b, and c to fail to constitute sides of a triangle.

Notice that, in the pendulum swing from the very abstract program graphs to the semantically rich slice-based testing, the testing is improved. We can expect the same on the specification-based side.

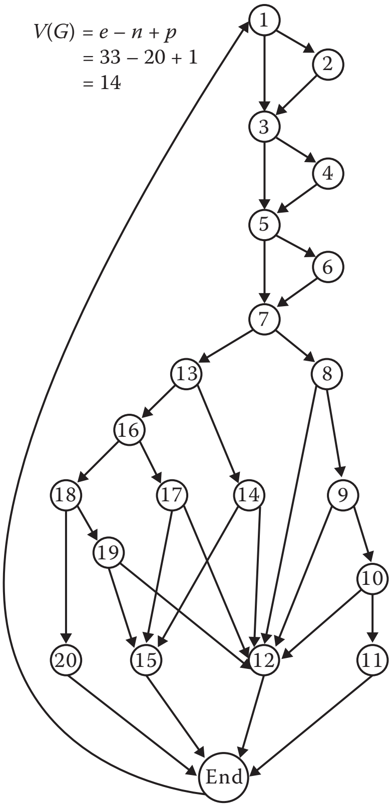


Figure 10.4 Directed graph of the FORTRAN-like triangle program.

Table 10.1 Feasible Paths in FORTRAN-Like Triangle Program

Path	*Node Sequence*	*Description*
p1	1–2–3–4–5–6–7–13–16–18–20	Equilateral
p2	1–3–5–6–7–13–16–18–19–15	Isosceles (b = c)
p3	1–3–5–6–7–13–16–18–19–12	Not a triangle (b = c)
p4	1–3–4–5–7–13–16–17–15	Isosceles (a = c)
p5	1–3–4–5–7–13–16–17–12	Not a triangle (a = c)
p6	1–2–3–5–7–13–14–15	Isosceles (a = b)
p7	1–2–3–5–7–13–14–12	Not a triangle (a = b)
p8	1–3–5–7–8–12	Not a triangle (a + b ≤ c)
p9	1–3–5–7–8–9–12	Not a triangle (b + c ≤ a)
p10	1–3–5–7–8–9–10–12	Not a triangle (a + c ≤ b)
p11	1–3–5–7–8–9–10–11	Scalene

Suppose we use boundary value testing to define test cases. We will do this for both the basic and worst-case formulations. Table 10.2 shows the test cases generated using the nominal boundary value form of functional testing. The last column shows the path (from Table 10.1) taken by the test case.

The following paths are covered: p1, p2, p3, p4, p5, p6, p7; paths p8, p9, p10, p11 are missed. Now suppose we use a more powerful functional testing technique, worst-case boundary value testing. We saw, in Chapter 5, that this yields 125 test cases; they are summarized here in Table 10.3 so you can see the extent of the redundant path coverage.

Taken together, the 125 test cases provide full path coverage, but the redundancy is onerous.

The next step in the pendulum progression is equivalence class testing. For the triangle problem, equivalence classes on the individual variables are pointless. Instead, we can make equivalence

Table 10.2 Path Coverage of Nominal Boundary Values

Case	*a*	*b*	*c*	*Expected Output*	*Path*
1	100	100	1	Isosceles	p6
2	100	100	2	Isosceles	p6
3	100	100	100	Equilateral	p1
4	100	100	199	Isosceles	p6
5	100	100	200	Not a triangle	p7
6	100	1	100	Isosceles	p4
7	100	2	100	Isosceles	p4
8	100	100	100	Equilateral	p1
9	100	199	100	Isosceles	p4
10	100	200	100	Not a triangle	p5
11	1	100	100	Isosceles	p2
12	2	100	100	Isosceles	p2
13	100	100	100	Equilateral	p1
14	199	100	100	Isosceles	p2
15	200	100	100	Not a triangle	p3

Table 10.3 Path Coverage of Worst-Case Values

	p1	*p2*	*p3*	*p4*	*p5*	*p6*	*p7*	*p8*	*p9*	*p10*	*p11*
Nominal	3	3	1	3	1	3	1	0	0	0	0
Worst-case	5	12	6	11	6	12	7	17	18	19	12

Table 10.4 Decision Table for FORTRAN-Like Triangle Program

c1. *Match* =	*0*				*1*				*2*			
c2. a + b < c?	T	F!	F!	F	T	F!	F!	F	T	F!	F!	F
c3. a + c < b?	F!	T	F!	F	F!	T	F!	F	F!	T	F!	F
c4. b + c < a?	F!	F!	T	F	F!	F!	T	F	F!	F!	T	F
a1. Scalene				×								
a2. Not a triangle	×	×	×		×	×	×		×	×	×	
a3. Isosceles							×					×
a4. Equilateral												
a5. Impossible												

c1. *Match* =	*3*				*4*	*5*	*6*			
c2. a + b < c?	T	F!	F!	F	—	—	T	F!	F!	F
c3. a + c < b?	F!	T	F!	F	—	—	F!	T	F!	F
c4. b + c < a?	F!	F!	T	F	—	—	F!	F!	T	F
a1. Scalene										
a2. Not a triangle	×	×	×				×	×	×	
a3. Isosceles				×						
a4. Equilateral										×
a5. Impossible					×	×				

classes on the types of triangles, and the six ways that the variables a, b, and c can fail to be sides of a triangle. In Chapter 6 (Section 6.4), we ended up with these equivalence classes:

$D1 = \{<a, b, c>: a = b = c\}$
$D2 = \{<a, b, c>: a = b, a \neq c\}$
$D3 = \{<a, b, c>: a = c, a \neq b\}$
$D4 = \{<a, b, c>: b = c, a \neq b\}$
$D5 = \{<a, b, c>: a \neq b, a \neq c, b \neq c\}$
$D6 = \{<a, b, c>: a > b + c\}$
$D7 = \{<a, b, c>: b > a + c\}$
$D8 = \{<a, b, c>: c > a + b\}$
$D9 = \{<a, b, c>: a = b + c\}$
$D10 = \{<a, b, c>: b = a + c\}$
$D11 = \{<a, b, c>: c = a + b\}$

Since these are equivalence classes, we will have just 11 test cases, and we know we will have full coverage of the 11 feasible paths in Figure 10.3.

The last step is to see if decision tables will add anything to the equivalence class test cases. They do not, but they can provide some insight into the decisions in the FORTRAN-like flowchart. In the decision table in Table 10.4, first notice that the condition on Match is an extended entry. Although it is topologically possible to have Match = 4 and Match = 5, these values are logically impossible. Conditions c2, c3, and c4 are exactly those used in the flowchart. We use the *F!* (must be false) notation to denote the impossibility of more than one of these conditions to be true. Also, note that there is no point in developing conditions on the individual variables a, b, and c. To conclude the traversal of the pendulum, decision table–based testing did not add much, but it did highlight why some cases are impossible.

10.3 Evaluating Test Methods

Evaluating a test method reduces to ways to evaluate how effective is a set of test cases generated by a test method, but we need to clarify what "effective" means. The easy choice is to be dogmatic: mandate a method, use it to generate test cases, and then run the test cases. This is absolute, and conformity is measurable; so it can be used as a basis for contractual compliance. We can improve on this by relaxing a dogmatic mandate and require that testers choose "appropriate methods," using the guidelines given at the ends of various chapters here. We can gain another incremental improvement by devising appropriate hybrid methods; we will have an example of this in Section 10.4.

Structured testing techniques yield a second choice for test effectiveness. We can use the notion of program execution paths, which provide a good formulation of test effectiveness. We will be able to examine a set of test cases in terms of the execution paths traversed. When a particular path is traversed more than once, we might question the redundancy. Mutation testing, the subject of Chapter 21, is an interesting way to assess the utility of a set of test cases.

The best interpretation for testing effectiveness is (no great surprise) the most difficult. We would really like to know how effective a set of test cases is for finding faults present in a program. This is problematic for two reasons: first, it presumes we know all the faults in a program. Quite a circularity—if we did, we would take care of them. Because we do not know all the faults in a program, we could never know if the test cases from a given method revealed them. The second reason is more theoretical: proving that a program is fault-free is equivalent to the famous halting problem of computer science, which is known to be impossible. The best we can do is to work backward from fault types. Given a particular kind of fault, we can choose testing methods (specification-based and code-based) that are likely to reveal faults of that type. If we couple this with knowledge of the most likely kinds of faults, we end up with a pragmatic approach to testing effectiveness. This is improved if we track the kinds (and frequencies) of faults in the software we develop.

By now, we have convinced ourselves that the specification-based methods are indeed open to the twin problems of gaps and redundancies; we can develop some metrics that relate the effectiveness of a specification-based technique with the achievement of a code-based metric. Specification-based testing techniques always result in a set of test cases, and a code-based metric is always expressed in terms of something countable, such as the number of program paths, the number of decision-to-decision paths (DD-paths), or the number of slices.

In the following definitions, we assume that a specification-based testing technique M generates m test cases, and that these test cases are tracked with respect to a code-based metric S that identifies *s* elements in the unit under test. When the *m* test cases are executed, they traverse *n* of the *s* structural elements.

Definition

The *coverage of a methodology M with respect to a metric S* is the ratio of *n* to *s*. We denote it as *C*(*M*,*S*).

Definition

The *redundancy of a methodology M with respect to a metric S* is the ratio of *m* to *s*. We denote it as *R*(*M*,*S*).

Definition

The *net redundancy of a methodology M with respect to a metric S* is the ratio of *m* to *n*. We denote it as NR(*M*,*S*).

We interpret these metrics as follows: the coverage metric, *C*(*M*,*S*), deals with gaps. When this value is less than 1, there are gaps in the coverage with respect to the metric. Notice that, when *C*(*M*,*S*) = 1, algebra forces *R*(*M*,*S*) = NR(*M*,*S*). The redundancy metric is obvious—the bigger it is, the greater the redundancy. Net redundancy is more useful—it refers to things actually traversed, not to the total space of things to be traversed. Taken together, these three metrics give a quantitative way to evaluate the effectiveness of any specification-based testing method (except special value testing) with respect to a code-based metric. This is only half the battle, however. What we really would like is to know how effective test cases are with respect to kinds of faults. Unfortunately, information such as this simply is not available. We can come close by selecting code-based metrics with respect to the kinds of faults we anticipate (or maybe faults we most fear). See the guidelines near the end of this chapter for specific advice.

In general, the more sophisticated code-based metrics result in more elements (the quantity *s*); hence, a given functional methodology will tend to become less effective when evaluated in terms of more rigorous code-based metrics. This is intuitively appealing, and it is borne out by our examples. These metrics are devised such that the best possible value is 1. Table 10.5 uses test case data from our earlier chapters to apply these metrics to the triangle program.
Table 10.6 repeats this analysis for the commission problem.

Table 10.5 Metrics for Triangle Program

Method	*m*	*n*	*s*	*C(M,S) = n/s*	*R(M,S) = m/s*	*NR(M,S) = m/n*
Nominal	15	7	11	0.64	1.36	2.14
Worst-case	125	11	11	1.00	11.36	11.36
Goal	s	s	s	1.00	1.00	1.00

Table 10.6 Metrics for Commission Problem

Method	*m*	*n*	*s*	*C(M,S) = n/s*	*R(M,S) = m/s*
Output bva	25	11	11	1	2.27
Decision table	3	11	11	1	0.27
DD-path	25	11	11	1	2.27
du-Path	25	33	33	1	0.76
Slice	25	40	40	1	0.63

10.4 Insurance Premium Case Study

Here is an example that lets us compare both specification-based and code-based testing methods and apply the guidelines. A hypothetical insurance premium program computes the semiannual car insurance premium based on two parameters: the policyholder's age and driving record:

Premium = BaseRate * ageMultiplier – safeDrivingReduction

The ageMultiplier is a function of the policyholder's age, and the safe driving reduction is given when the current points (assigned by traffic courts for moving violations) on the policyholder's driver's license are below an age-related cutoff. Policies are written for drivers in the age range of 16 to 100. Once a policyholder exceeds 12 points, the driver's license is suspended (thus, no insurance is needed). The BaseRate changes from time to time; for this example, it is $500 for a semiannual premium. The data for the insurance premium program are in Table 10.7.

10.4.1 Specification-Based Testing

Worst-case boundary value testing, based on the input variables, age, and points, yields the following extreme values of the age and points variables (Table 10.8). The corresponding 25 test cases are shown in Figure 10.5.

Table 10.7 Data for Insurance Premium Problem

Age Range	*Age Multiplier*	*Points Cutoff*	*Safe Driving Reduction*
16 ≤ Age < 25	2.8	1	50
25 ≤ Age < 35	1.8	3	50
35 ≤ Age < 45	1.0	5	100
45 ≤ Age < 60	0.8	7	150
60 ≤ Age < 100	1.5	5	200

Table 10.8 Data Boundaries for Insurance Premium Problem

Variable	*Min*	*Min+*	*Nom.*	*Max–*	*Max*
Age	16	17	54	99	100
Points	0	1	6	11	12

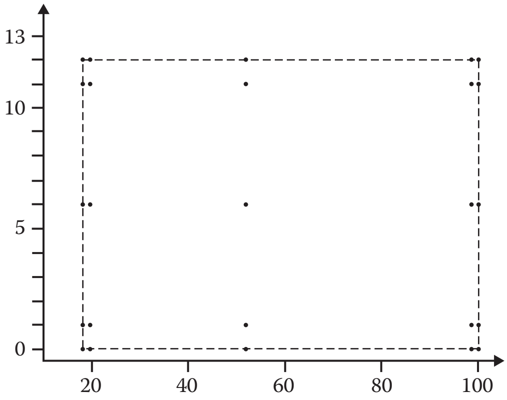


Figure 10.5 Worst-case boundary value test cases for insurance premium problem.

Nobody should be content with these test cases. There is too much of the problem statement missing. The various age cutoffs are not tested, nor are the point cutoffs. We could refine this by taking a closer look at classes based on the age ranges.

A1 = {age: 16 ≤ age < 25}
A2 = {age: 25 ≤ age < 35}
A3 = {age: 35 ≤ age < 45}
A4 = {age: 45 ≤ age < 60}
A5 = {age: 60 ≤ age < 100}

Here are the age-dependent classes on license points.

P1(A1) = {points = 0, 1}, {points = 2, 3, …, 12}
P2(A2) = {points = 0, 1, 2, 3}, {points = 4, 5, …, 12}
P3(A3) = {points = 0, 1, 2, 3, 4, 5}, {points = 6, 7, …, 12}
P4(A4) = {points = 0, 1, 2, 3, 4, 5, 6, 7}, {points = 8, 9, 10, 11, 12}
P5(A5) = {points = 0, 1, 2, 3, 4, 5}, {points = 6, 7, …, 12}

One added complexity is that the point ranges are dependent on the age of the policyholder and also overlap. Both of these constraints are shown in Figure 10.6. The dashed lines show the age-dependent equivalence classes. A set of worst-case boundary value test cases is shown only for class A4 and its two related point classes are given in Figure 10.6. Because these ranges meet at "endpoints," we would have the worst-case test values shown in Table 10.9. Notice that the discrete

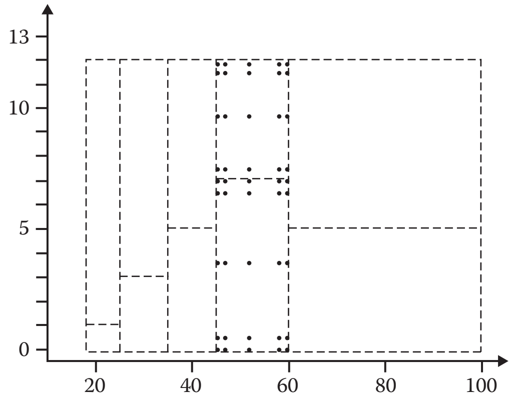


Figure 10.6 Detailed worst-case boundary value test cases for one age class.

Table 10.9 Detailed Worst-Case Values

Variable	*Min*	*Min+*	*Nom.*	*Max–*	*Max*
Age	16	17	20	24	
Age	25	26	30	34	
Age	35	36	40	44	
Age	45	46	53	59	
Age	60	61	75	99	100
Points(A1)	0	n/a	n/a	n/a	1
Points(A1)	2	3	7	11	12
Points(A2)	0	1	n/a	2	3
Points(A2)	4	5	8	11	12
Points(A3)	0	1	3	4	5
Points(A3)	6	7	9	11	12
Points(A4)	0	1	4	6	7
Points(A4)	8	9	10	11	12
Points(A5)	0	1	3	4	5
Points(A5)	6	7	9	11	12

values of the point variable do not lend themselves to the min+ and max– convention in some cases. These are the variable values that lead to 103 test cases.

We are clearly at a point of severe redundancy; time to move on to equivalence class testing. The age sets A1–A5, and the points sets P1–P5 are natural choices for equivalence classes. The corresponding weak normal equivalence class test cases are shown in Figure 10.7. Since the point

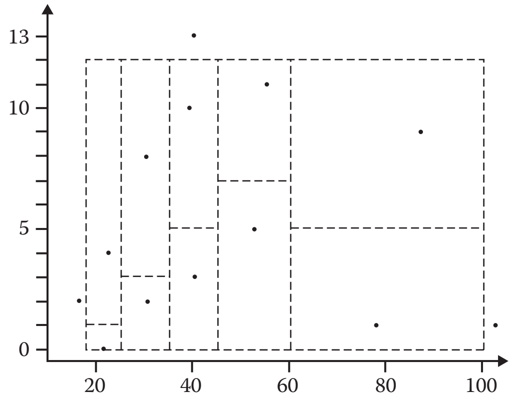


Figure 10.7 Weak and robust normal equivalence class test cases for insurance premium program.

classes are not independent, we cannot do the usual cross product. Weak robust cases are of some value because we would expect different outputs for drivers with age less than 16, and points in excess of 12. The additional weak robust test cases are shown as open circles in Figure 10.7.

The next step is to see if a decision table approach might help. Table 10.10 is a decision table based on the age equivalence classes. The decision table test cases are almost the same as those shown in Figure 10.7; the only weak robust test case missing in the decision table is that for points exceeding 12.

What are the error-prone aspects of the insurance premium program? The endpoints of the age ranges appear to be a good place to start, and this puts us back in boundary value mode. We can imagine many complaints from policyholders whose premium did not reflect a recent borderline birthday. Incidentally, this would be a good example of risk-based testing. Dealing with such complaints would be costly. Also, we should consider ages under 16 and over 100. Finally, we should probably check the values at which the safe driving reduction is lost, and maybe values

Table 10.10 Insurance Premium Decision Table

	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
Age is	<16	16–24		25–34		35–44		45–59		60–100		>100
Points below cutoff?	—	T	F	T	F	T	F	T	F	T	F	—
ageMultiplier = 2.8		×	×									
ageMultiplier = 1.8				×	×							
ageMultiplier = 1.0						×	×					
ageMultiplier = 0.8								×	×			
ageMultiplier = 1.5										×	×	
Safe driving discount		×		×		×		×		×		
No policy allowed	×											×

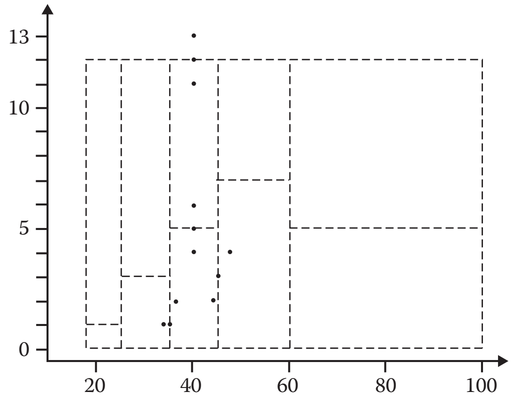


Figure 10.8 Hybrid test cases for the 35 to 45 age class.

of points over 12, when all insurance is lost. All of this is shown in Figure 10.7. (Notice that the responses to these were not in the problem statement, but our testing analysis provokes us to think about them.) Maybe this should be called hybrid functional testing: it uses the advantages of all three forms in a blend that is determined by the nature of the application (shades of special value testing). Hybrid appears appropriate because such selection is usually done to improve the stock.

To blend boundary value testing with weak robust equivalence class testing, note that the age class borders are helpful. Testing the max–, max, and max+ values of one age class automatically moves us into the next age class, so there is a slight economy. Figure 10.8 shows the hybrid test cases for the age range 35–45 in the insurance premium problem.

10.4.2 Code-Based Testing

Our analysis thus far has been entirely specification based. To be complete, we really need the code. It will answer questions such as whether or not the age variable is an integer (our assumption thus far) or not. There is no question that the points variable is an integer. The pseudocode implementation is minimal in the sense that it does very little error checking. The pseudocode and its program graph are in Figure 10.9. Because the program graph is acyclic, only a finite number of paths exist—in this case, 11. The best choice is simply to have test cases that exercise each path. This automatically constitutes both statement and DD-path coverage. The compound case predicates indicate multiple-condition coverage; this is accomplished only with the worst-case boundary test cases and the hybrid test cases. The remaining path-based coverage metrics are not applicable.

10.4.2.1 Path-Based Testing

The cyclomatic complexity of the program graph of the insurance premium program is $V(G)$ = 11, and exactly 11 feasible program execution paths exist. They are listed in Table 10.11. If you follow the pseudocode for the various sets of functional test cases in Chapter 5, you will find the results shown in Table 10.12. We can see some of the insights gained from structural testing. For one thing, the problem of gaps and redundancies is obvious. Only the test cases from the hybrid approach yield complete path coverage. It is instructive to compare the results of these 25 test cases with the other two methods yielding the same number of test cases. The 25 boundary value test

```
Program insurance premium
Dim driverAge, points As Real
Dim baseRate, premium As Real
1  Input(baseRate, driverAge, points)
2  premium = 0
3  safeDrivingReduction = 0
4  Select Case driverAge
5    Case 1:  16<= driverAge  < 25
6      ageMultiplier = 2.8
7      If points < 1 Then
8        safeDrivingReduction = 50
9      EndIf
10   Case 2:  25<= driverAge  < 35
11     ageMultiplier = 1.8
12     If points < 3 Then
13       safeDrivingReduction = 50
14     EndIf
15   Case 3:  35<= driverAge  < 45
16     ageMultiplier = 1.0
17     If points < 5 Then
18       safeDrivingReduction = 100
19     EndIf
20   Case 4:  45<= driverAge  < 60
21     ageMultiplier = 0.8
22     If points < 7 Then
23       safeDrivingReduction = 150
24     EndIf
25   Case 5:  60<= driverAge  <= 100
26     ageMultiplier = 1.5
27     If points < 5 Then
28       safeDrivingReduction = 200
29     EndIf
30   Case 6:  Else
31       Output("Driver age out of range")
32 End Select
33 premium = baseRate * ageMultiplier – safeDrivingReduction
34 Output(premium)
35 End
```

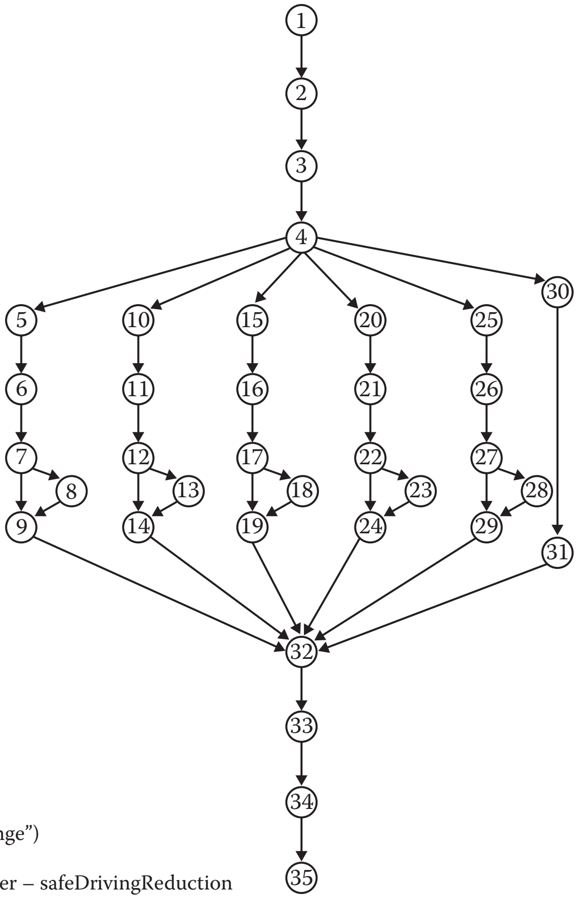


Figure 10.9 Insurance premium pseudocode and program graph.

cases only cover six of the feasible execution paths, while the 25 weak normal equivalence classes test cases cover 10 of the feasible execution paths. The next difference is in the coverage of the conditions in the case statement. Each predicate is a compound condition of the form $a \leq x < b$. The only methods that yield test cases that exercise these extreme values are the worst-case boundary value (103) test cases and the hybrid (32) test cases. Incidentally, the McCabe baseline method will yield 11 of the 12 decision table test cases.

10.4.2.2 Data Flow Testing

Data flow testing for this problem is boring. The driverAge, points, and safeDrivingReduction variables all occur in six definition-clear du-paths. The "uses" for driverAge and points are both predicate uses. Recall from Chapter 9 that the All-Paths criterion implies all the lower data flow covers.

Table 10.11 Paths in Insurance Premium Program

Path	*Node Sequence*
p1	1–2–3–4–5–6–7–9–32–33–34–35
p2	1–2–3–4–5–6–7–8–9–32–33–34–35
p3	1–2–3–4–10–11–12–14–32–33–34–35
p4	1–2–3–4–10–11–12–13–14–32–33–34–35
p5	1–2–3–4–15–16–17–19–32–33–34–35
p6	1–2–3–4–15–16–17–18–19–32–33–34–35
p7	1–2–3–4–20–21–22–24–32–33–34–35
p8	1–2–3–4–20–21–22–23–24–32–33–34–35
p9	1–2–3–4–25–26–27–29–32–33–34–35
p10	1–2–3–4–25–26–27–28–29–32–33–34–35
p11	1–2–3–4–30–31–32–33–34–35

Table 10.12 Path Coverage of Functional Methods in Insurance Premium Program

Figure	*Method*	*Test Cases*	*Paths Covered*
10.5	Boundary value	25	p1, p2, p7, p8, p9, p10
10.6	Worst-case boundary value	103	p1, p2, p3, p4, p5, p6, p7, p8, p9, p10
10.7	Weak normal equivalence class	10	P1, p2, p3, p4, p5, p6, p7, p8, p9
10.7	Robust normal equivalence class	12	p1, p2, p3, p4, p5, p6, p7, p8, p9, p10, p11
10.7	Decision table	12	p1, p2, p3, p4, p5, p6, p7, p8, p9, p10, p11
10.8	Hybrid specification-based (extended to all age classes)	32	p1, p2, p3, p4, p5, p6, p7, p8, p9, p10, p11

10.4.2.3 Slice Testing

Slice testing does not provide much insight either. Four slices are of interest:

S(safeDrivingReduction, 33) = {1, 2, 3, 4, 5, 7, 8, 9, 10, 12, 13, 14, 15, 17, 18, 19, 20, 22, 23, 24, 25, 27, 28, 29, 32}
S(ageMultiplier, 33) = {1, 2, 3, 4, 5, 6, 10, 11, 15, 16, 20, 21, 25, 26, 32}
S(baseRate, 33) = {1}
S(Premium, 33) = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 32}

The union of these slices is the whole program. The only insight we might get from slice-based testing is that, if a failure occurred at line 33, the slices on safeDrivingReduction and ageMultiplier separate the program into two disjoint pieces, and that would simplify fault isolation.

10.5 Guidelines

Here is one of my favorite testing stories. An inebriated man was crawling around on the sidewalk beneath a streetlight. When a policeman asked him what he was doing, he replied that he was looking for his car keys. "Did you lose them here?" the policeman asked. "No, I lost them in the parking lot, but the light is better here."

This little story contains an important message for testers: testing for faults that are not likely to be present is pointless. It is far more effective to have a good idea of the kinds of faults that are most likely (or most damaging) and then to select testing methods that are likely to reveal these faults.

Many times, we do not even have a feeling for the kinds of faults that may be prevalent. What then? The best we can do is use known attributes of the program to select methods that deal with the attributes—sort of a "punishment fits the crime" view. The attributes that are most helpful in choosing specification-based testing methods are

Whether the variables represent physical or logical quantities
Whether dependencies exist among the variables
Whether single or multiple faults are assumed
Whether exception handling is prominent

Here is the beginning of an "expert system" to help with this:

1. If the variables refer to physical quantities, boundary value testing and equivalence class testing are indicated.
2. If the variables are independent, boundary value testing and equivalence class testing are indicated.
3. If the variables are dependent, decision table testing is indicated.
4. If the single-fault assumption is warranted, boundary value analysis and robustness testing are indicated.
5. If the multiple-fault assumption is warranted, worst-case testing, robust worst-case testing, and decision table testing are indicated.
6. If the program contains significant exception handling, robustness testing and decision table testing are indicated.
7. If the variables refer to logical quantities, equivalence class testing and decision table testing are indicated.

Combinations of these may occur; therefore, the guidelines are summarized as a decision table in Table 10.13.

Table 10.13 Appropriate Choices for Functional Testing

c1	Variables (P, physical; L, logical)	P	P	P	P	P	L	L	L	L	L
c2	Independent variables?	Y	Y	Y	Y	N	Y	Y	Y	Y	N
c3	Single-fault assumption?	Y	Y	N	N	—	Y	Y	N	N	—
c4	Exception handling?	Y	N	Y	N	—	Y	N	Y	N	—
a1	Boundary value analysis		×								
a2	Robustness testing	×									
a3	Worst-case testing				×						
a4	Robust worst case			×							
a5	Weak robust equivalence class	×		×			×		×		
a6	Weak normal equivalence class	×	×				×	×			
a7	Strong normal equivalence class			×	×	×			×	×	×
a8	Decision table					×					×

EXERCISES

1. Repeat the gaps and redundancies analysis for the triangle problem using the structured implementation in Chapter 2 and its DD-path graph in Chapter 8.
2. Compute the coverage, redundancy, and net redundancy metrics for your study in exercise 1.
3. The pseudocode for the insurance premium program does not check for driver ages under 16 or (unlikely) over 100. The Else clause (case 6) will catch these, but the output message is not very specific. Also, the output statement (33) is not affected by the driver age checks. Which functional testing techniques will reveal this fault? Which structural testing coverage, if not met, will reveal this fault?

References

Brown, J.R. and Lipov, M., Testing for software reliability, *Proceedings of the International Symposium on Reliable Software,* Los Angeles, April 1975, pp. 518–527.

Pressman, R.S., *Software Engineering: A Practitioner's Approach*, McGraw-Hill, New York, 1982.

BEYOND UNIT TESTING III

In Part III, we build on the basic ideas of unit testing covered in Part II, with one major change. Now we are more concerned with knowing what to test. To that end, the discussion in this part begins with the whole idea of model-based testing. Chapter 11 examines testing based on models of software development life cycles, and models of software/system behavior are discussed in Chapter 12. Chapter 13 presents model-based strategies for integration testing, and these are extended to system testing in Chapter 14. We return to the question of testing object-oriented software in Chapter 15, with the main focus on points at which the object-oriented paradigm differs from the traditional paradigm. Having completed this much, we are in a position to finally take a serious look at software complexity in Chapter 16. We apply much of this to a relatively recent question, testing systems of systems, in Chapter 17.

Chapter 11

Life Cycle–Based Testing

In this chapter, we begin with various models of the software development life cycle in terms of the implications these life cycles have for testing. We took a general view in Chapter 1, where we identified three levels (unit, integration, and system) in terms of symmetries in the waterfall model of software development. This view has been relatively successful for decades, and these levels persist; however, the advent of alternative life cycle models mandates a deeper look at these views of testing. We begin with the traditional waterfall model, mostly because it is widely understood and is a reference framework for the more recent models. Then we look at derivatives of the waterfall model, and finally some mainline agile variations.

We also make a major shift in our thinking. We are more concerned with how to represent the item tested because the representation may limit our ability to identify test cases. Take a look at the papers presented at the leading conferences (professional or academic) on software testing—you will find nearly as many presentations on specification models and techniques as on testing techniques. Model-Based Testing (MBT) is the meeting place of software modeling and testing at all levels.

11.1 Traditional Waterfall Testing

The traditional model of software development is the waterfall model, which is illustrated in Figure 11.1. It is sometimes drawn as a V as in Figure 11.2 to emphasize how the basic levels of testing reflect the early waterfall phases. (In ISTQB circles, this is known as the "V-Model.") In this view, information produced in one of the development phases constitutes the basis for test case identification at that level. Nothing controversial here: we certainly would hope that system test cases are clearly correlated with the requirements specification, and that unit test cases are derived from the detailed design of the unit. On the upper left side of the waterfall, the tight what/how cycles are important. They underscore the fact that the predecessor phase defines what is to be done in the successor phase. When complete, the successor phase states how it accomplishes "what" was to be done. These are also ideal points at which to conduct software reviews (see Chapter 22). Some humorists assert that these phases are the fault creation phases, and those on the right are the fault detection phases.

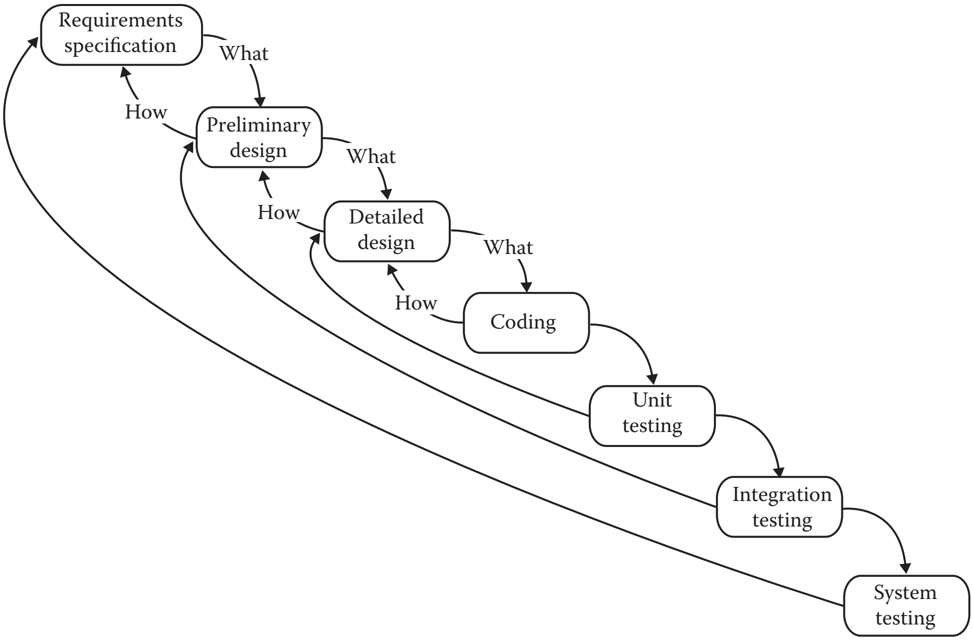

Requirements specification
What
How
Preliminary design
What
How
Detailed design
What
How
Coding
Unit testing
Integration testing
System testing

Figure 11.1 The waterfall life cycle.

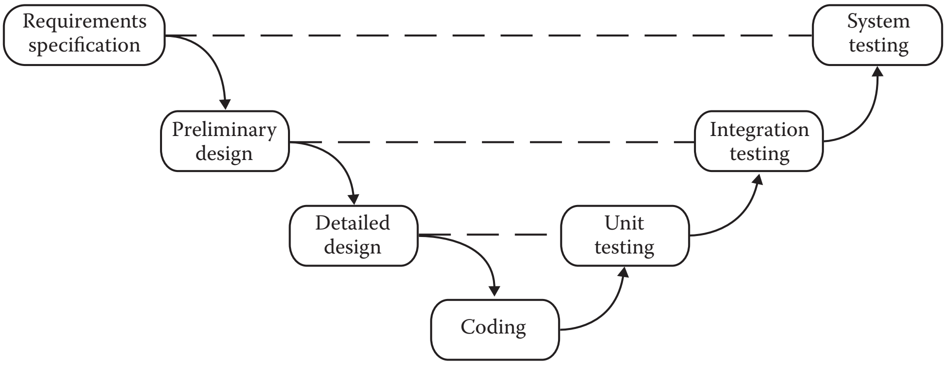

Requirements specification
System testing
Preliminary design
Integration testing
Detailed design
Unit testing
Coding

Figure 11.2 The waterfall life cycle as the V-Model.

Two observations: a clear presumption of functional testing is used here, and an implied bottom–up testing order is used. Here, "bottom–up" refers to levels of abstraction—unit first, then integration, and finally, system testing. In Chapter 13, bottom–up also refers to a choice of orders in which units are integrated (and tested).

Of the three main levels of testing (unit, integration, and system), unit testing is best understood. Chapters 5 through 10 are directed at the testing theory and techniques applicable to unit testing. System testing is understood better than integration testing, but both need clarification. The bottom–up approach sheds some insight: test the individual components, and then integrate these into subsystems until the entire system is tested. System testing should be something that the customer (or user) understands, and it often borders on customer acceptance testing. Generally, system testing is functional instead of structural; this is mostly due to the lack of higher-level structural notations.

11.1.1 Waterfall Testing

The waterfall model is closely associated with top–down development and design by functional decomposition. The end result of preliminary design is a functional decomposition of the entire system into a tree-like structure of functional components. With such a decomposition, top–down integration would begin with the main program, checking the calls to the next-level units, and so on until the leaves of the decomposition tree are reached. At each point, lower-level units are replaced by stubs—throwaway code that replicates what the lower-level units would do when called. Bottom–up integration is the opposite sequence, starting with the leaf units and working up toward the main program. In bottom–up integration, units at higher levels are replaced by drivers (another form of throwaway code) that emulate the procedure calls. The "big bang" approach simply puts all the units together at once, with no stubs or drivers. Whichever approach is taken, the goal of traditional integration testing is to integrate previously tested units with respect to the functional decomposition tree. Although this describes integration testing as a process, discussions of this type offer little information about the methods or techniques. We return to this in Chapter 13.

11.1.2 Pros and Cons of the Waterfall Model

In its history since the first publication in 1968, the waterfall model has been analyzed and critiqued repeatedly. The earliest compendium was by Agresti (1986), which stands as a good source. Agresti observes that

- The framework fits well with hierarchical management structures.
- The phases have clearly defined end products (exit criteria), which in turn are convenient for project management.
- The detailed design phase marks the starting point where individuals responsible for units can work in parallel, thereby shortening the overall project development interval.

More importantly, Agresti highlights major limitations of the waterfall model. We shall see that these limitations are answered by the derived life cycle models. He observes that

- There is a very long feedback cycle between requirements specification and system testing, in which the customer is absent.
- The model emphasizes analysis to the near exclusion of synthesis, which first occurs at the point of integration testing.
- Massive parallel development at the unit level may not be sustainable with staffing limitations.
- Most important, "perfect foresight" is required because any faults or omissions at the requirements level will penetrate through the remaining life cycle phases.

The "omission" part was particularly troubling to the early waterfall developers. As a result, nearly all of the papers of requirements specification demanded consistency, completeness, and clarity. Consistency is impossible to demonstrate for most requirements specification techniques (decision tables are an exception), and the need for clarity is obvious. The interesting part is completeness—all of the successor life cycles assume incompleteness, and depend on some form of iteration to gradually arrive at "completeness."

11.2 Testing in Iterative Life Cycles

Since the early 1980s, practitioners have devised alternatives in response to shortcomings of the traditional waterfall model just mentioned. Common to all of these alternatives is the shift away from the functional decomposition to an emphasis on iteration and composition. Decomposition is a perfect fit both to the top–down progression of the waterfall model and to the bottom–up testing order, but it relies on one of the major weaknesses of waterfall development cited by Agresti (1986)—the need for "perfect foresight." Functional decomposition can only be well done when the system is completely understood, and it promotes analysis to the near exclusion of synthesis. The result is a very long separation between requirements specification and a completed system, and during this interval, no opportunity is available for feedback from the customer. Composition, on the other hand, is closer to the way people work: start with something known and understood, then add to it gradually, and maybe remove undesired portions.

A very nice analogy can be applied to positive and negative sculpture. In negative sculpture, work proceeds by removing unwanted material, as in the mathematician's view of sculpting Michelangelo's *David*: start with a piece of marble, and simply chip away all non-*David*. Positive sculpture is often done with a pliable medium, such as wax. The central shape is approximated, and then wax is either added or removed until the desired shape is attained. The wax original is then cast in plaster. Once the plaster hardens, the wax is melted out, and the plaster "negative" is used as a mold for molten bronze. Think about the consequences of a mistake: with negative sculpture, the whole work must be thrown away and restarted. (A museum in Florence, Italy, contains half a dozen such false starts to the *David*.) With positive sculpture, the erroneous part is simply removed and replaced. We will see this is the defining essence of the agile life cycle models. The centrality of composition in the alternative models has a major implication for integration testing.

11.2.1 Waterfall Spin-Offs

There are three mainline derivatives of the waterfall model: incremental development, evolutionary development, and the spiral model (Boehm, 1988). Each of these involves a series of increments or builds as shown in Figure 11.3. It is important to keep preliminary design as an integral phase rather than to try to amortize such high-level design across a series of builds. (To do so usually results in unfortunate consequences of design choices made during the early builds that are regrettable in later builds.) This single design step cannot be done in the evolutionary and spiral models. This is also a major limitation of the bottom–up agile methods.

Within a build, the normal waterfall phases from detailed design through testing occur with one important difference: system testing is split into two steps—regression and progression testing. The main impact of the series of builds is that regression testing becomes necessary. The goal of regression testing is to ensure that things that worked correctly in the previous build still work with the newly added code. Regression testing can either precede or follow integration testing, or possibly occur in both places. Progression testing assumes that regression testing was successful and that the new functionality can be tested. (We like to think that the addition of new code represents progress, not a regression.) Regression testing is an absolute necessity in a series of builds

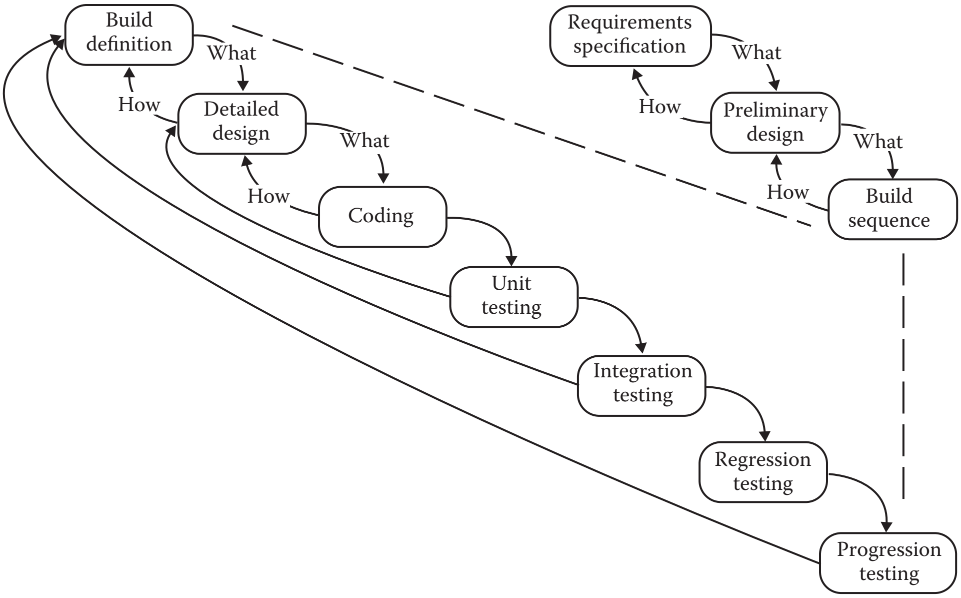


Figure 11.3 Iterative development.

because of the well-known ripple effect of changes to an existing system. (The industrial average is that one change in five introduces a new fault.)

Evolutionary development is best summarized as client-based iteration. In this spin-off, a small initial version of a product is given to users who then suggest additional features. This is particularly helpful in applications for which time-to-market is a priority. The initial version might capture a segment of the target market, and then that segment is "locked in" to future evolutionary versions. When these customers have a sense that they are "being heard," they tend to be more invested in the evolving product.

Barry Boehm's spiral model has some of the flavor of the evolutionary model. The biggest difference is that the increments are determined more on the basis of risk rather than on client suggestions. The spiral is superimposed on an x–y coordinate plane, with the upper left quadrant referring to determining objectives, the upper right to risk analysis, the lower right refers to development (and test), and the lower left is for planning the next iteration. These four phases—determine objectives, analyze risk, develop and test, and next iteration planning—are repeated in an evolutionary way. At each evolutionary step, the spiral enlarges.

There are two views of regression testing: one is to simply repeat the tests from the previous iteration; the other is to devise a smaller set of test cases specifically focused on finding affected faults. Repeating a full set of previous integration tests is fine in an automated testing environment, but is undesirable in a more manual environment. The expectation of test case failure is (or should be) lower for regression testing compared to that for progression testing. As a guideline, regression tests might fail in only 5% of the repeated progression tests. This may increase to 20% for progression tests. If regression tests are performed manually, there is an interesting term for special regression test cases: Soap Opera Tests. The idea is to have long, complex regression tests, akin to the complicated plot lines in television soap operas. A soap opera test case could fail in many ways, whereas a progression test case should fail for only a very few reasons. If a soap opera test case fails, clearly more focused testing is required to localize the fault. We will see this again in Chapter 20 on all-pairs testing.

The differences among the three spin-off models are due to how the builds are identified. In incremental development, the motivation for separate builds is usually to flatten the staff profile. With pure waterfall development, there can be a huge bulge of personnel for the phases from detailed design through unit testing. Many organizations cannot support such rapid staff fluctuations, so the system is divided into builds that can be supported by existing personnel. In evolutionary development, the presumption of a build sequence is still made, but only the first build is defined. On the basis of that, later builds are identified, usually in response to priorities set by the customer/user, so the system evolves to meet the changing needs of the user. This foreshadows the customer-driven tenet of the agile methods. The spiral model is a combination of rapid prototyping and evolutionary development, in which a build is defined first in terms of rapid prototyping and then is subjected to a go/no-go decision based on technology-related risk factors. From this, we see that keeping preliminary design as an integral step is difficult for the evolutionary and spiral models. To the extent that this cannot be maintained as an integral activity, integration testing is negatively affected. System testing is not affected.

Because a build is a set of deliverable end-user functionality, one advantage common to all these spin-off models is that they provide earlier synthesis. This also results in earlier customer feedback, so two of the deficiencies of waterfall development are mitigated. The next section describes two approaches to deal with the "perfect foresight" problem.

11.2.2 Specification-Based Life Cycle Models

When systems are not fully understood (by either the customer or the developer), functional decomposition is perilous at best. Barry Boehm jokes when he describes the customer who says "I don't know what I want, but I'll recognize it when I see it." The rapid prototyping life cycle (Figure 11.4)

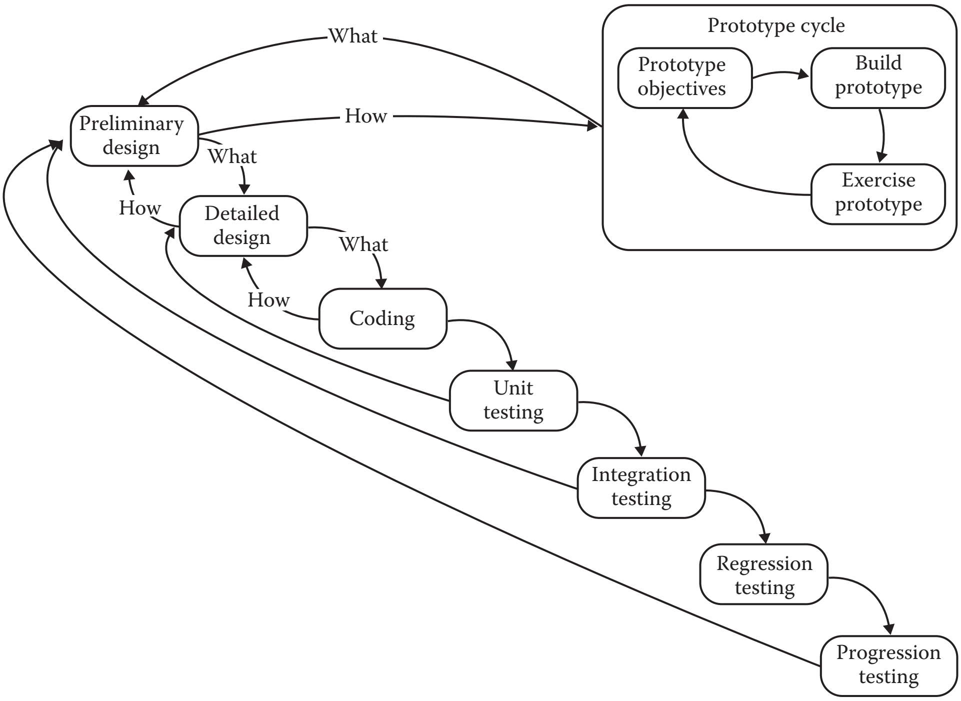


Figure 11.4 Rapid prototyping life cycle.

deals with this by providing the "look and feel" of a system. Thus, in a sense, customers can recognize what they "see." In turn, this drastically reduces the specification-to-customer feedback loop by producing very early synthesis. Rather than build a final system, a "quick and dirty" prototype is built and then used to elicit customer feedback. Depending on the feedback, more prototyping cycles may occur. Once the developer and the customer agree that a prototype represents the desired system, the developer goes ahead and builds to a correct specification. At this point, any of the waterfall spin-offs might also be used. The agile life cycles are the extreme of this pattern.

Rapid prototyping has no new implications for integration testing; however, it has very interesting implications for system testing. Where are the requirements? Is the last prototype the specification? How are system test cases traced back to the prototype? One good answer to questions such as these is to use the prototyping cycles as information-gathering activities and then produce a requirements specification in a more traditional manner. Another possibility is to capture what the customer does with the prototypes, define these as scenarios that are important to the customer, and then use these as system test cases. These could be precursors to the user stories of the agile life cycles. The main contribution of rapid prototyping is that it brings the operational (or behavioral) viewpoint to the requirements specification phase. Usually, requirements specification techniques emphasize the structure of a system, not its behavior. This is unfortunate because most customers do not care about the structure, and they do care about the behavior.

Executable specifications (Figure 11.5) are an extension of the rapid prototyping concept. With this approach, the requirements are specified in an executable format (such as finite state machines, StateCharts, or Petri nets). The customer then executes the specification to observe the intended system behavior and provides feedback as in the rapid prototyping model. The executable models are, or can be, quite complex. This is an understatement for the full-blown version of StateCharts. Building an executable model requires expertise, and executing it requires an engine. Executable

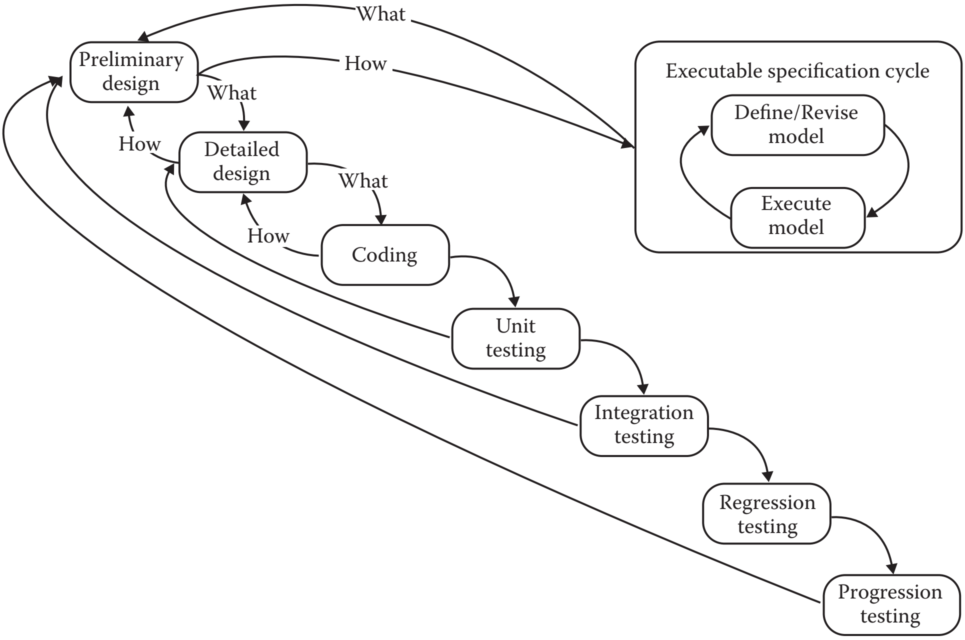


Figure 11.5 Executable specification.

specification is best applied to event-driven systems, particularly when the events can arrive in different orders. David Harel, the creator of StateCharts, refers to such systems as “reactive” (Harel, 1988) because they react to external events. As with rapid prototyping, the purpose of an executable specification is to let the customer experience scenarios of intended behavior. Another similarity is that executable models might have to be revised on the basis of customer feedback. One side benefit is that a good engine for an executable model will support the capture of “interesting” system transactions, and it is often a nearly mechanical process to convert these into true system test cases. If this is done carefully, system testing can be traced directly back to the requirements.

Once again, this life cycle has no implications for integration testing. One big difference is that the requirements specification document is explicit, as opposed to a prototype. More important, it is often a mechanical process to derive system test cases from an executable specification. We will see this in Chapter 14. Although more work is required to develop an executable specification, this is partially offset by the reduced effort to generate system test cases. Here is another important distinction: when system testing is based on an executable specification, we have an interesting form of structural testing at the system level. Finally, as we saw with rapid prototyping, the executable specification step can be combined with any of the iterative life cycle models.

11.3 Agile Testing

The *Agile Manifesto* (http://agilemanifesto.org/) was written by 17 consultants, the Agile Alliance, in February 2001. It has been translated into 42 languages and has drastically changed the software development world. The underlying characteristics of all agile life cycles are

- Customer-driven
- Bottom–up development
- Flexibility with respect to changing requirements
- Early delivery of fully functional components

These are sketched in Figure 11.6. Customers express their expectations in terms of user stories, which are taken as the requirements for very short iterations of design–code–test. When does an

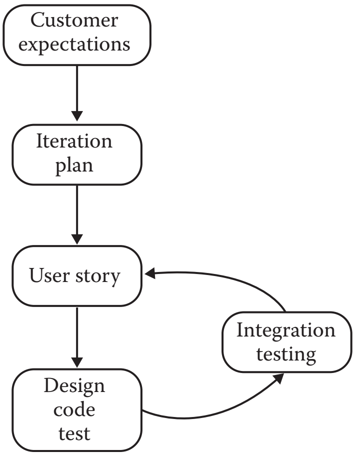


Figure 11.6 Generic agile life cycle.

agile project end? When the customer has no more user stories. Looking back at the iterative models, we see the progenitors of agility, especially in Barry Boehm's spiral model. Various websites will list as few as 3 to as many as 40 variations of agile software development. Here we look at three major ones, and focus on how they deal with testing.

11.3.1 Extreme Programming

Extreme Programming (XP) was first applied to a project (in a documented way) in 1996 by Kent Beck (http://www.extremeprogramming.org/) while he was at Chrysler Corporation. The clear success of the project, even though it was a revision of an earlier version, led to his book (Beck, 2004). The main aspects of XP are captured in Figure 11.7. It is clearly customer-driven, as shown by the position of user stories driving both a release plan and system testing. The release plan defines a sequence of iterations, each of which delivers a small working component. One distinction of XP is the emphasis on paired programming, in which a pair of developers work closely together, often sharing a single development computer and keyboard. One person works at the code level, while the other takes a slightly higher view. In a sense, the pair is conducting a continuous review. In Chapter 22, we will see that this is better described as a continuous code walk-through. There are many similarities to the basic iterative life cycle shown in Figure 11.3. One important difference is that there is no overall preliminary design phase. Why? Because this is a bottom–up process. If XP were truly driven by a sequence of user stories, it is hard to imagine what can occur in the release plan phase.

11.3.2 Test-Driven Development

Test-driven development (TDD) is the extreme case of agility. It is driven by a sequence of user stories, as shown in Figure 11.8. A user story can be decomposed into several tasks, and this is where the big difference occurs. Before any code is written for a task, the developer decides how it will be tested. The tests become the specification. The next step is curious—the tests are run on nonexistent code. Naturally, they fail, but this leads to the best feature of TDD—greatly simplified

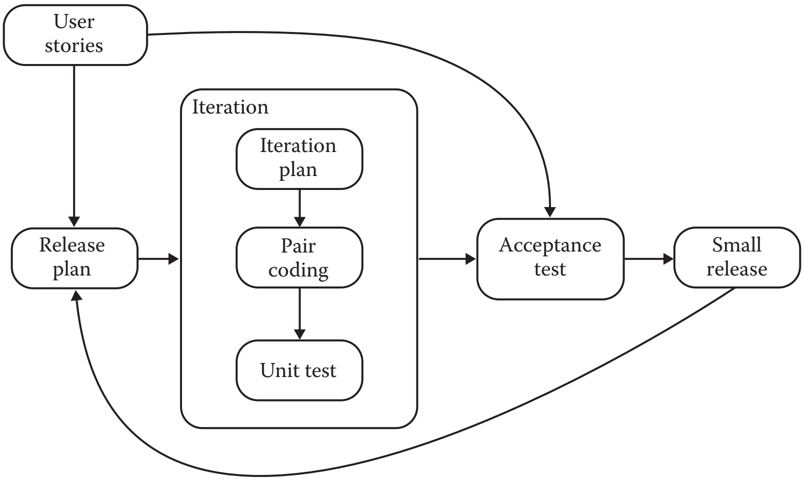


Figure 11.7 The Extreme Programming life cycle.

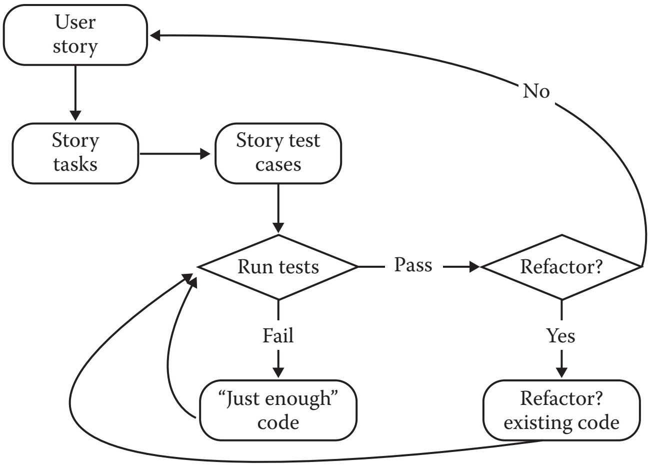


Figure 11.8 Test-driven development life cycle.

fault isolation. Once the tests have been run (and failed), the developer writes just enough code to make the tests pass, and the tests are rerun. If any test fails, the developer goes back to the code and makes a necessary change. Once all the tests pass, the next user story is implemented. Occasionally, the developer may decide to refactor the existing code. The cleaned–up code is then subjected to the full set of existing test cases, which is very close to the idea of regression testing. For TDD to be practical, it must be done in an environment that supports automated testing, typically with a member of the nUnit family of automated test environments. (We will have an example of this in Chapter 19.)

Testing in TDD is interesting. Since the story-level test cases drive the coding, they ARE the specification, so in a sense, TDD uses specification-based testing. But since the code is deliberately as close as possible to the test cases, we could argue that it is also code-based testing. There are two problems with TDD. The first is common to all agile flavors—the bottom–up approach prohibits a single, high-level design step. User stories that arrive late in the sequence may obviate earlier design choices. Then refactoring would have to also occur at the design level, rather than just at the code level. The agile community is very passionate about the claim that repeated refactoring results in an elegant design. Given one of the premises of agile development, namely that the customer is not sure of what is needed, or equivalently, rapidly changing requirements, refactoring at both the code and design levels seems the only way to end up with an elegant design. This is an inevitable constraint on bottom–up development.

The second problem is that all developers make mistakes—that is much of the reason we test in the first place. But consider: what makes us think that the TDD developer is perfect at devising the test cases that drive the development? Even worse: what if late user stories are inconsistent with earlier ones? A final limitation of TDD is there is no place in the life cycle for a cross-check at the user story level.

11.3.3 Scrum

Scrum is probably the most frequently used of all the agile life cycles. There is a pervading emphasis on the team members and teamwork. The name comes from the rugby maneuver in which the opposing teams are locked together and try to hook the football back to their respective sides. A rugby scrum requires organized teamwork—hence, the name for the software process.

The quick view of Scrum (the development life cycle) is that it is mostly new names for old ideas. This is particularly true about the accepted Scrum vocabulary. Three examples: roles, ceremonies, and artifacts. In common parlance, Scrum roles refer to project participants; the ceremonies are just meetings, the artifacts are work products. Scrum projects have Scrum masters (who act like traditional supervisors with less administrative power). Product owners are the customers of old, and the Scrum team is a development team. Figure 11.9 is adapted from the "official" Scrum literature, the Scrum Alliance (http://www.scrumalliance.org/learn_about_scrum). Think about the activities in terms of the iterative life cycle in Figure 11.3. The traditional iterations become "sprints," which last from 2 to 4 weeks. In a sprint, there is a daily stand-up meeting of the Scrum team to focus on what happened the preceding day and what needs to be done in the new day. Then there is a short burst of design–code–test followed by an integration of the team's work at the end of the day. This is the agile part—a daily build that contributes to a sprint-level work product in a short interval. The biggest differences between Scrum and the traditional view of iterative development are the special vocabulary and the duration of the iterations.

Testing in the Scrum life cycle occurs at two levels—the unit level at each day's end, and the integration level of the small release at the end of a sprint. Selection of the Sprint backlog from the product backlog is done by the product owner (the customer), which corresponds roughly to a requirements step. Sprint definition looks a lot like preliminary design because this is the point where the Scrum team identifies the sequence and contents of individual sprints. The bottom line? Scrum has two distinct levels of testing—unit and integration/system. Why "integration/system?" The small release is a deliverable product usable by the product owner, so it is clearly a system-level work product. But this is the point where all of the development work is integrated for the first time.

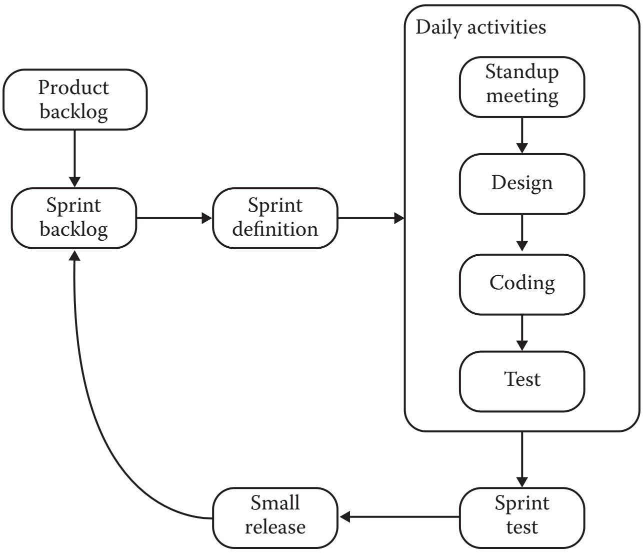


Figure 11.9 The Scrum life cycle.

11.4 Agile Model–Driven Development

My German friend Georg is a PhD mathematician, a software developer, and a Go player. For several months, we had an e-mail-based discussion about agile development. At one point, Georg asked if I play the oriental game Go. I do not, but he replied that, to be a successful Go player, one needs both strategy and tactics. A deficiency in either one puts a Go player at a disadvantage. In the software development realm, he equates strategy with an overall design, and tactics as unit-level development. His take on the flavors of agile development is that the strategy part is missing, and this leads us to a compromise between the agile world and the traditional views of software development. We first look at Agile Model–Driven Development (AMDD) popularized by Scott Ambler. This is followed by my mild reorganization of Ambler's work, named here as Model–Driven Agile Development (MDAD).

11.4.1 Agile Model–Driven Development

The agile part of AMDD is the modeling step. Ambler's advice is to model just enough for the current user story, and then implement it with TDD. The big difference between AMDD and any of the agile life cycles is that there is a distinct design step. (The agilists usually express their distaste/disdain for modeling by calling it the "Big Design Up Front" and abbreviate it as simply the BDUF.) See Figure 11.10.

Ambler's contribution is the recognition that design does indeed have a place in agile development. As this was being written, there was a protracted discussion on LinkedIn started by the question "Is there any room for design in agile software development?" Most of the thread affirms the need for design in any agile life cycle. Despite all this, there seems to be no room in AMDD for integration/system testing.

11.4.2 Model–Driven Agile Development

Model–driven agile development (MDAD) is my proposal for a compromise between the traditional and the agile worlds. It is stimulated by Georg's view of the need for both strategy and tactics, hence the compromise. How does MDAD differ from iterative development? MDAD recommends test-driven development as the tactic and it uses Ambler's view of short iterations.

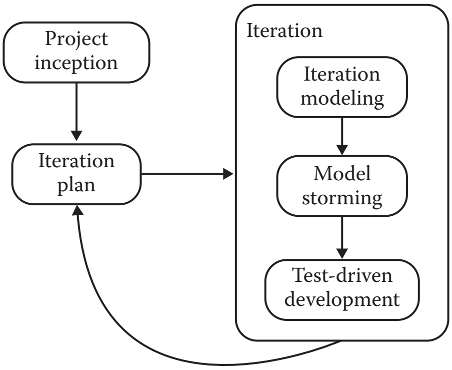


Figure 11.10 The agile model–driven development life cycle.

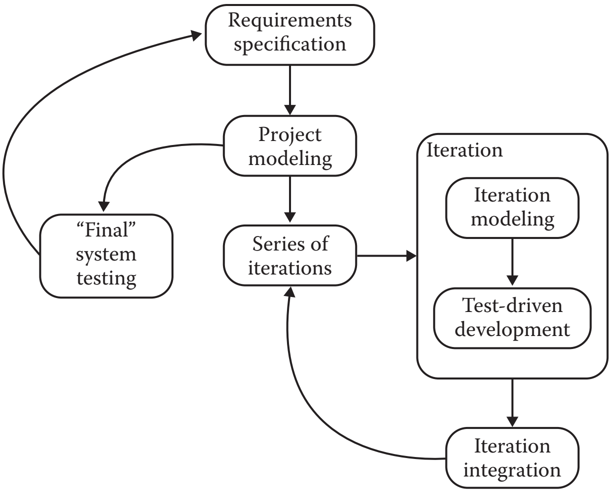

Requirements specification
Project modeling
"Final" system testing
Series of iterations
Iteration
Iteration modeling
Test-driven development
Iteration integration

Figure 11.11 The model–driven agile development life cycle.

The strategy part is the emphasis on an overall model, which in turn, supports MBT. In MDAD, the three levels of testing, unit, integration, and system, are present (Figure 11.11).

References

Agresti, W.W., *New Paradigms for Software Development,* IEEE Computer Society Press, Washington, DC, 1986.

Beck, K., *Extreme Programming Explained: Embrace Change*, 2nd ed., Addison Wesley, Boston, 2004.

Boehm, B.W., A spiral model for software development and enhancement, *IEEE Computer*, Vol. 21, No. 6, May 1988, pp. 61–72.

Harel, D., On visual formalisms, *Communications of the ACM*, Vol. 31, No. 5, May 1988, pp. 514–530.

Chapter 12

Model-Based Testing

> "By my faith! For more than forty years I have been speaking prose without knowing anything about it.…"
>
> **Monsieur Jourdain in *Le Bourgeois Gentilhomme***

I share the sentiment of Moliere's Monsieur Jourdain; since the first edition, this book has advocated what we now call Model-Based Testing (MBT). In this chapter, we describe the basic mechanism, discuss how to choose appropriate models, consider the pros and cons of MBT, and provide a short discussion of available tools. Actual examples of MBT are (and have been in the earlier editions) scattered throughout this book.

12.1 Testing Based on Models

The main advantage of modeling system behavior is that the process of creating a model usually results in deeper insights and understanding of the system being modeled/tested. This is particularly true of executable models such as finite state machines, Petri nets, and StateCharts. In Chapter 14, we will see that threads of system behavior, which are easily transformed into system level test cases, are readily derived from many behavioral models. Given this, the adequacy of MBT will always depend on the accuracy of the model. The essence of MBT is this sequence of steps:

1. Model the system.
2. Identify threads of system behavior in the model.
3. Transform these threads into test cases.
4. Execute the test cases (on the actual system) and record the results.
5. Revise the model(s) as needed and repeat the process.

12.2 Appropriate Models

Avvinare is one of my favorite Italian words. It refers to a process that many Italian families perform in autumn when they bottle wine. After buying a demijohn of bulk wine, they rinse out the empty bottles that they have saved during the year. There are always small droplets of water clinging to the sides of a bottle, but it is really difficult to remove them. Instead, they fill a bottle about half full of the wine, and shake it up to dissolve the water into the wine. Next, the wine is funneled into the next bottle, shaken, and poured into another bottle. This continues until all the bottles have been rinsed with wine, and they are ready for bottling. *Avvinare* is the verb that refers to this entire process. How would you translate this word into English? I really don't know, but it won't be easy. Languages evolve to meet the expressive needs of their speakers, and this activity is not very common in the English-speaking world. To wax esoteric, this is where software engineering meets epistemology. Since MBT begins with modeling, the choice of an appropriate model determines the ultimate success of the associated testing. Making an appropriate choice depends on several things: the expressive power of various models, the essential nature of the system being modeled, and the analyst's ability to use various models. We consider the first two of these next.

12.2.1 Peterson's Lattice

James Peterson (1981) developed an elegant lattice of models of computation, which is summarized in Figure 12.1. The arrows in the lattice signify a "more expressive than" relationship in which the model at the origin of an arrow is more expressive than that at the end of an arrow. In his text, Peterson carefully develops examples for each edge in the lattice. For example, he shows a semaphore system that cannot be expressed as a finite state machine. Four models in his lattice are fairly obscure: vector replacement systems, vector addition systems, UCLA graphs, and message

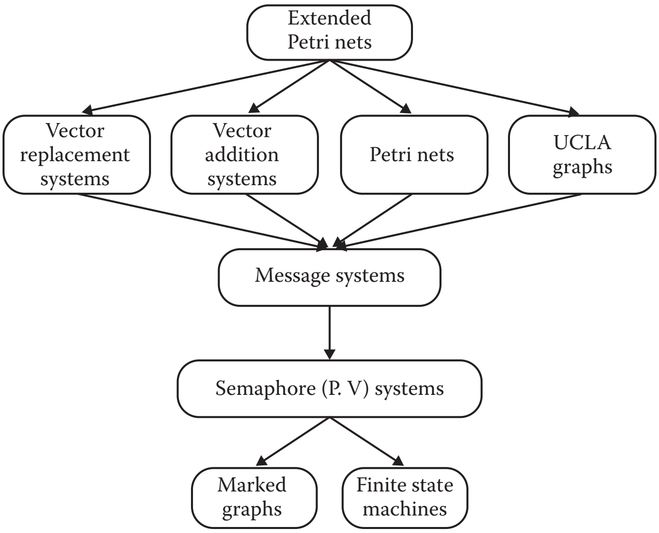


Figure 12.1 Peterson's lattice.

systems. There are scores of extensions to Petri nets; Peterson grouped these together for simplicity. Marked graphs are a formalization of data flow diagrams, and Peterson shows them to be formal duals of finite state machines.

Peterson's lattice is a good starting point for MBT. Given an application, good practice dictates choosing a model that is both necessary and sufficient—neither too weak nor too strong. If a model is too weak, important aspects of the application will not be modeled, and hence not tested. If a model is too strong, the extra effort to develop the model may be unnecessary.

Peterson's lattice predates the invention of StateCharts by David Harel, which raises the question of where they fit in Peterson's lattice. They are at least equivalent, and probably more expressive than most extensions of Petri nets. Several graduate students at Grand Valley State University have explored this question, with a variety of approaches. Their work is persuasive, but as yet, I have no formal proof of this potential equivalence. However, given a relatively complex StateChart, it can always be expressed as an event-driven Petri net (as defined in Chapter 4). The rich language associated with StateChart transitions will probably be difficult to express in most Petri net extensions. One promising approach offered by DeVries (2013) is that of "Swim Lane Petri Nets."

Figure 12.2 shows the anticipated placement of StateCharts in Peterson's lattice. The one-way arrow reflects the fact that a given StateChart can express concurrency (by the concurrent regions), and true concurrency cannot be expressed in a Petri net, nor in most extensions. Part of the work by DeVries describes Swim Lane Petri Nets. These use the UML notion of "swim lanes" to express parallel activities. We will revisit this concept in Chapter 17 when we use it to describe interactions among constituent systems in systems of systems. There we will use some of the prompts of the Extended Systems Modeling Language to show cross–swim lane communication of Event-Driven Petri Nets. Figure 12.3 shows the anticipated lattice among Event-Driven Petri Nets, Swim Lane Event-Driven Petri Nets, and a subclass of StateCharts.

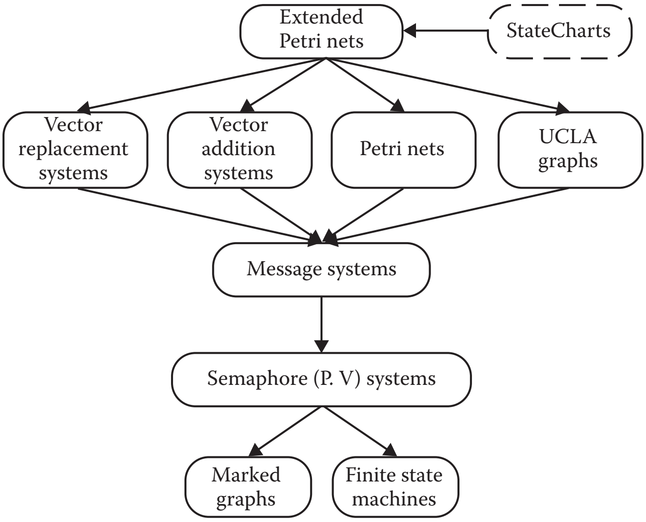


Figure 12.2 Placement of StateCharts in Peterson's lattice.

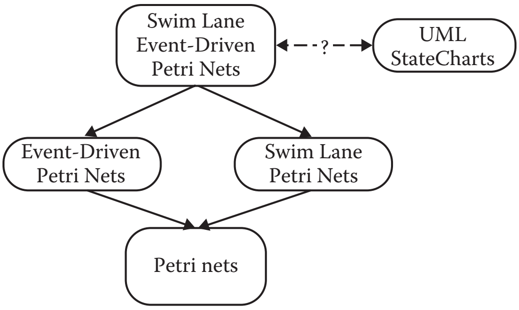

Swim Lane Event-Driven Petri Nets
?
UML StateCharts
Event-Driven Petri Nets
Swim Lane Petri Nets
Petri nets

Figure 12.3 Lattice with swim lane models.

12.2.2 Expressive Capabilities of Mainline Models

Peterson looked at four mainline models in terms of the kinds of behavioral issues that they can represent. The Venn diagram in Figure 12.4 shows his summary.

12.2.3 Modeling Issues

Much of the information in this subsection is taken from Jorgensen (2009). There are two fundamental types of requirements specification models: those that describe structure and those that describe behavior. These correspond to two fundamental views of a system: what a system *is* and what a system *does*. Data flow diagrams, entity/relation models, hierarchy charts, class diagrams, and object diagrams all focus on what a system is—the components, their functionality, and interfaces among them. They emphasize structure. The second type, including decision tables, finite state machines, StateCharts, and Petri nets, describes system behavior—what a system does. Models of system behavior have varying degrees of expressive capability, the technical equivalent of being able to express *avvinare* in another language.

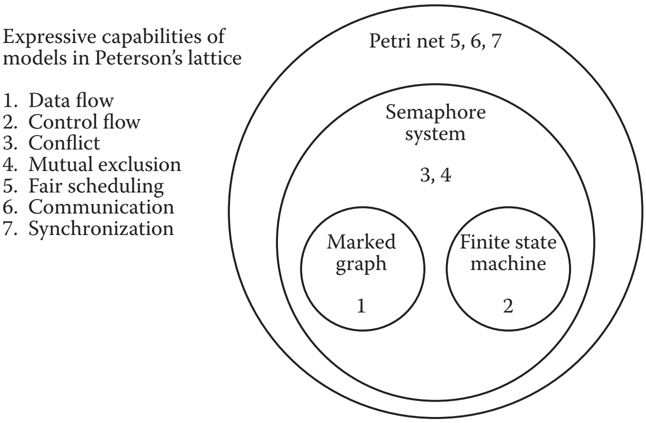

Expressive capabilities of models in Peterson's lattice
1. Data flow
2. Control flow
3. Conflict
4. Mutual exclusion
5. Fair scheduling
6. Communication
7. Synchronization
Petri net 5, 6, 7
Semaphore system
3, 4
Marked graph
1
Finite state machine
2

Figure 12.4 Expressive capabilities in Peterson's lattice.

Table 12.1 Expressive Capabilities of Selected Behavioral Models

Behavioral Issue	*Source of Issue*
Sequence	Structured Programming
Selection	
Repetition	
Enable	Extended Systems Modeling Language
Disable	
Trigger	
Activate	
Suspend	
Resume	
Pause	
Conflict	Task Management
Priority	
Mutual exclusion	
Concurrent execution	
Deadlock	
Context-sensitive input events	Events
Multiple context output events	
Asynchronous events	
Event quiescence	

The Jorgensen (2009) reference identifies 19 behavioral modeling issues, subdivided into the three groups described in Table 12.1. The first three are the code structuring precepts of Structured Programming. The next group is from the Extended Systems Modeling Language group (Bruyn et al., 1988). These prompts will be used in our modeling of systems of systems using Swim Lane Event-Driven Petri Nets in Chapter 17. The task management category consists of the basic Petri net mechanisms, and the last category deals with issues in event-driven systems.

Table 12.2 maps the 19 behavioral issues to five executable models, each of which is suitable for MBT.

12.2.4 Making Appropriate Choices

Choosing an appropriate model begins with understanding the essential nature of the system to be modeled (and tested). Once these aspects are understood, they must be related to the various capabilities just discussed, and then the appropriate choice is simplified. The ultimate choice will

Table 12.2 Expressive Capability of Five Executable Models

Behavioral Issue	*Decision Tables*	*FSMs*	*Petri Nets*	*EDPNs*	*StateCharts*
Sequence	No	Yes	Yes	Yes	Yes
Selection	Yes	Yes	Yes	Yes	Yes
Repetition	Yes	Yes	Yes	Yes	Yes
Enable	No	No	Yes	Yes	Yes
Disable	No	No	Yes	Yes	Yes
Trigger	No	No	Yes	Yes	Yes
Activate	No	No	Yes	Yes	Yes
Suspend	No	No	Yes	Yes	Yes
Resume	No	No	Yes	Yes	Yes
Pause	No	No	Yes	Yes	Yes
Conflict	No	No	Yes	Yes	Yes
Priority	No	No	Yes	Yes	Yes
Mutual exclusion	Yes	No	Yes	Yes	Yes
Concurrent execution	No	No	Yes	Yes	Yes
Deadlock	No	No	Yes	Yes	Yes
Context-sensitive input events	Yes	Yes	Indirectly	Yes	Yes
Multiple context output events	Yes	Yes	Indirectly	Yes	Yes
Asynchronous events	No	No	Indirectly	Yes	Yes
Event quiescence	No	No	Indirectly	Yes	Yes

always depend on other realities, such as company policy, relevant standards, analyst capability, and available tools. Always choosing the most powerful model is a simple-minded choice; a better choice might be to choose the simplest model that can express all the important aspects of the system being modeled.

12.3 Commercial Tool Support for Model-Based Testing

Alan Hartman (2003) separates commercial tools for MBT into three groups:

- Modeling tools
- Model-based test input generators
- Model-based test generators

According to Hartman, modeling tools such as IBM's Rational Rose and Telelogic's Rhapsody and Stalemate provide inputs to true model-based test generators, but by themselves, do not generate test cases. Model-based test input generators are a step up—they generate the input portion of test cases, but cannot generate the expected output portion. Full model-based test generators require some form of oracle to identify expected outputs. This is the sticking point for full test case generation. There are some existing university and company proprietary test generation systems, and a very few companies claim to have commercial tools available. Until this technology becomes commercially viable, note that use case–based testing provides the expected output portion of a test case.

Hartman's generalization is too sweeping—we know that full test cases, with expected outcomes, can be derived even from simple finite state machines—provided that the model shows the expected outputs. When the modeler is the oracle, and provides expected outputs in any model, the model can serve as a full test generator.

Mark Utting and Bruno Legeard address Model-Based Testing in their excellent book (Utting and Legeard, 2006).

References

Bruyn, W., Jensen, R., Keskar, D. and Ward, P. An extended systems modeling language (ESML), Association for Computing Machinery, ACM SIGSOFT Software Engineering Notes, Vol. 13, No. 1, January 1988, pp. 58–67.

DeVries, B., Mapping of UML Diagrams to Extended Petri Nets for Formal Verification, Master's thesis, Grand Valley State University, Allendale, MI, April 2013.

Hartman, A., Model Based Test Generation Tools, 2003, available at www.agedis.de/documents/ModelBasedTestGenerationTools_cs.pdf.

Jorgensen, P.C., *Modeling Software Behavior: A Craftsman's Approach*, CRC Press, New York, 2009.

Peterson, J.L, *Petri Net Theory and the Modeling of Systems*, Prentice Hall, Englewood Cliffs, NJ, 1981.

Utting, M. and Legeard, B., *Practical Model-Based Testing,* Morgan Kaufman Elsevier, San Francisco, 2006.

Chapter 13

Integration Testing

In September 1999, the Mars Climate Orbiter mission failed after successfully traveling 416 million miles in 41 weeks. It disappeared just as it was to begin orbiting Mars. The fault should have been revealed by integration testing: Lockheed Martin Astronautics used acceleration data in English units (pounds), while the Jet Propulsion Laboratory did its calculations with metric units (newtons). NASA announced a $50,000 project to discover how this could have happened (Fordahl, 1999). They should have read this chapter.

Of the three distinct levels of software testing—unit, integration, and system—integration testing is the least well understood of these; hence in practice, it is the phase most poorly done. This chapter examines two mainline and one less well-known integration testing strategies. They are illustrated with a continuing procedural example, discussed in some detail, and then critiqued with respect to their advantages and disadvantages.

Craftspersons are recognized by two essential characteristics: they have a deep knowledge of the tools of their trade, and they have a similar knowledge of the medium in which they work so that they understand their tools in terms of how they work with the medium. In Chapters 5 through 10, we focused on the tools (techniques) available to the testing craftsperson at the unit level. Our goal there was to understand testing techniques in terms of their advantages and limitations with respect to particular types of software. Here, we continue our emphasis on model-based testing, with the goal of improving the testing craftsperson's judgment through a better understanding of three underlying models. Integration testing for object-oriented software is integrated into this chapter.

13.1 Decomposition-Based Integration

Mainline introductory software engineering texts, for example, Pressman (2005) and Schach (2002), typically present four integration strategies based on the functional decomposition tree of the procedural software: top–down, bottom–up, sandwich, and the vividly named "big bang." Many classic software testing texts echo this approach, Deutsch (1982), Hetzel (1988), Kaner et al. (1993), and Mosley (1993), to name a few. Each of these strategies (except big bang) describes the order in which units are to be integrated. We can dispense with the big bang approach most easily: in this view of integration, all the units are compiled together and tested at once. The drawback to this is that when (not if!) a failure is observed, few clues are available to help isolate the location(s) of the fault. (Recall the distinction we made in Chapter 1 between faults and failures.)

The functional decomposition tree is the basis for integration testing because it is the main representation, usually derived from final source code, which shows the structural relationship of the system with respect to its units. All these integration orders presume that the units have been separately tested; thus, the goal of decomposition-based integration is to test the interfaces among separately tested units. A functional decomposition tree reflects the lexicological inclusion of units, in terms of the order in which they need to be compiled, to assure the correct referential scope of variables and unit names. In this chapter, our familiar NextDate unit is extended to a main program, Calendar, with procedures and functions. Figure 13.1 contains the functional decomposition tree for the Calendar program. The pseudocode is given in next.

The Calendar program sketched here in pseudocode acquires a date in the form mm, dd, yyyy, and provides the following functional capabilities:

- The date of the next day (our old friend, NextDate)
- The day of the week corresponding to the date (i.e., Monday, Tuesday, …)
- The zodiac sign of the date
- The most recent year in which Memorial Day was celebrated on May 27
- The most recent Friday the 13th

A sketch of the Calendar program is given next, followed by a condensed "skeleton," which is the basis for the functional decomposition in Figure 13.1.

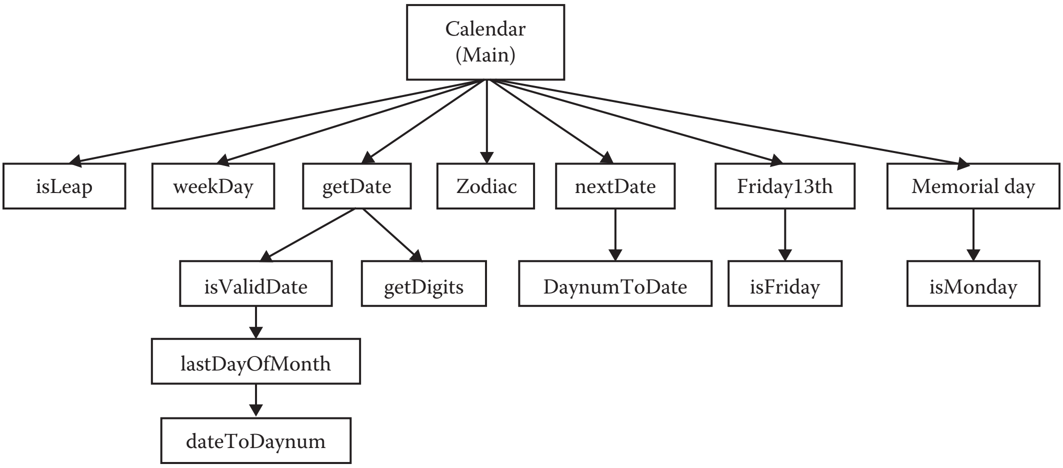


Figure 13.1 Functional decomposition of Calendar program.

Pseudocode for the Calendar Program

```
Main   Calendar
Data Declarations
    mm, dd, yyyy, dayNumber, dayName, zodiacSign
Function isLeap (input yyyy, returns T/F)
    (isLeap is self-contained)
End Function isLeap

Procedure getDate (returns mm, dd, yyyy, dayNumber)
    Function isValidDate (inputs mm, dd, yyyy; returns T/F)
       Function lastDayOfMonth (inputs mm, yyyy, returns 28, 29, 30, or 31)
          lastDayOfMonth body
               (uses isLeap)
          end lastDayOfMonth body
       End Function lastDayOfMonth

       isValidDate body
          (uses lastDayOfMonth)
       end isValidDate body
    End Function isValidDate

    Procedure getDigits(returns mm, dd, yyyy)
       (uses Function isValidDate)
    End Procedure getDigits

    Procedure memorialDay (inputs mm, dd, yyyy; returns yyyy)
       Function isMonday (inputs mm, dd, yyyy; returns T/F)
          (uses weekDay)
       End Function isMonday

       memorialDaybody
          isMonday
       end memorialDay
    End Procedure memorialDay

Procedure friday13th (inputs mm, dd, yyyy; returns mm1, dd1, yyyy1)
       Function isFriday (inputs mm, dd, yyyy; returns T/F)
          (uses weekDay)
       End Function isFriday

    friday13th body
       (uses isFriday)
    end friday13th
End Procedure friday13th

getDate body
    getDigits
    isValidDate
    dateToDayNumber
end getDate body
End Procedure getDate
Procedure nextDate (input daynum, output mm1, dd1, yyyy1)
    Procedure dayNumToDate
    dayNumToDate body
       (uses isLeap)
    end dayNumToDate body
nextDate body
    dayNumToDate
end nextDate body
End Procedure nextDate

Procedure weekDay (input mm, dd, yyyy; output dayName)
    (uses Zeller's Congruence)
End Procedure weekDay

Procedure zodiac (input dayNumber; output dayName)
    (uses dayNumbers of zodiac cusp dates)
End Procedure zodiac

Main program body
    getDate
    nextDate
    weekDay
    zodiac
    memorialDay
    friday13th
End Main program body
```

Lexicological Inclusion of the Calendar Program

```
Main   Calendar
    Function isLeap
    Procedure weekDay
    Procedure getDate
       Function isValidDate
          Function lastDayOfMonth
       Procedure getDigits
    Procedure memorialDay
          Function isMonday
    Procedure friday13th
       Function isFriday
    Procedure nextDate
       Procedure dayNumToDate
    Procedure zodiac
```

13.1.1 Top–Down Integration

Top–down integration begins with the main program (the root of the tree). Any lower-level unit that is called by the main program appears as a “stub,” where stubs are pieces of throwaway code that emulate a called unit. If we performed top–down integration testing for the Calendar program, the first step would be to develop stubs for all the units called by the main program—isLeap, weekDay, getDate, zodiac, nextDate, friday13th, and memorialDay. In a stub for any unit, the tester hard codes in a correct response to the request from the calling/invoking unit. In the stub for zodiac, for example, if the main program calls zodiac with 05, 27, 2012, zodiacStub would return “Gemini.” In extreme practice, the response might be “pretend zodiac returned Gemini.”

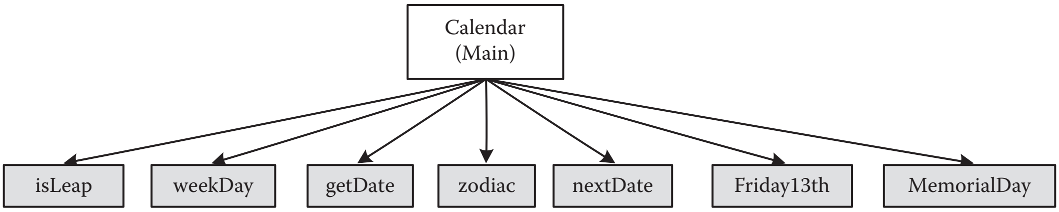

Calendar
(Main)
isLeap
weekDay
getDate
zodiac
nextDate
Friday13th
MemorialDay

Figure 13.2 First step in top–down integration.

The use of the pretend prefix emphasizes that it is not a real response. In practice, the effort to develop stubs is usually quite significant. There is good reason to consider stub code as part of the software project and maintain it under configuration management. In Figure 13.2, the first step in top–down integration is shown. The gray-shaded units are all stubs. The goal of the first step is to check that the main program functionality is correct.

Once the main program has been tested, we replace one stub at a time, leaving the others as stubs. Figure 13.3 shows the first three steps in the gradual replacement of stubs by actual code. The stub replacement process proceeds in a breadth-first traversal of the decomposition tree until all the stubs have been replaced. (In Figures 13.2 and 13.3, the units below the first level are not shown because they are not needed.)

The "theory" of top–down integration is that, as stubs are replaced one at a time, if there is a problem, it must be with the interface to the most recently replaced stub. (Note that the fault

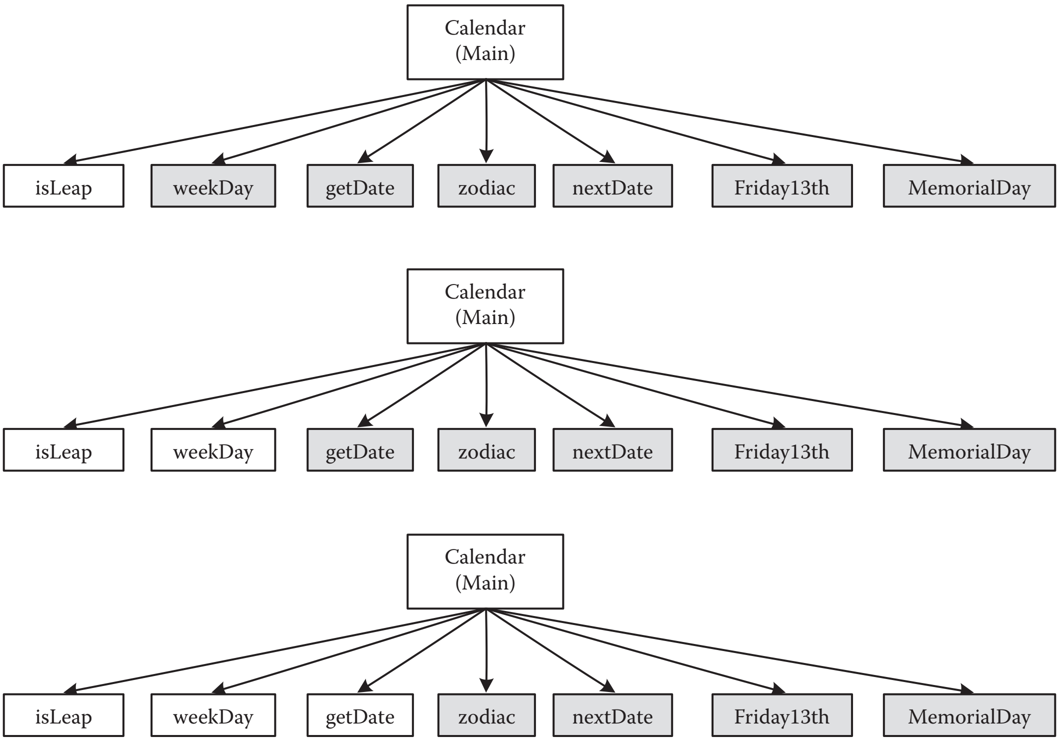

Calendar
(Main)
isLeap
weekDay
getDate
zodiac
nextDate
Friday13th
MemorialDay
Calendar
(Main)
isLeap
weekDay
getDate
zodiac
nextDate
Friday13th
MemorialDay
Calendar
(Main)
isLeap
weekDay
getDate
zodiac
nextDate
Friday13th
MemorialDay

Figure 13.3 Next three steps in top–down integration.

isolation is similar to that of test-driven development.) The problem is that a functional decomposition is deceptive. Because it is derived from the lexicological inclusion required by most compilers, the process generates impossible interfaces. Calendar main never directly refers to either isLeap or weekDay, so those test sessions could not occur.

13.1.2 Bottom–Up Integration

Bottom–up integration is a "mirror image" to the top–down order, with the difference that stubs are replaced by driver modules that emulate units at the next level up in the tree. (In Figure 13.4, the gray units are drivers.) Bottom–up integration begins with the leaves of the decomposition tree, and use a driver version of the unit that would normally call it to provide it with test cases. (Note the similarity to test driver units at the unit level. As units are tested, the drivers are gradually replaced, until the full decomposition tree has been traversed. Less throwaway code exists in bottom–up integration, but the problem of impossible interfaces persists.

Figure 13.5 shows one case where a unit (zodiac) can be tested with a driver. In this case, the Calendar driver would probably call zodiac with 36 test dates that are the day before a cusp date, the cusp date, and the day after the cusp date. The cusp date for Gemini is May 21, so the driver would call zodiac three times, with May 20, May 21, and May 22. The expected responses would be "Taurus," "Gemini," and "Gemini," respectively. Note how similar this is to the assert mechanism in the jUnit (and related) test environments.

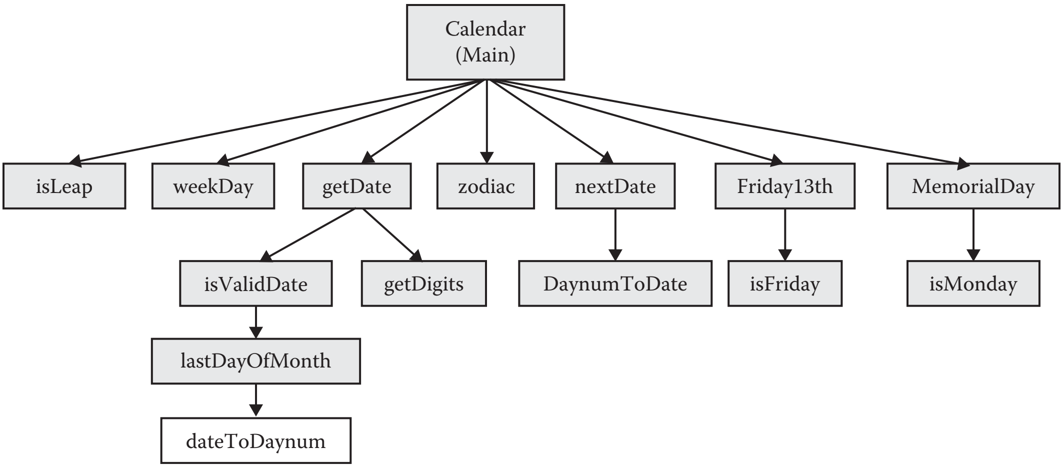

Calendar (Main)
isLeap
weekDay
getDate
zodiac
nextDate
Friday13th
MemorialDay
isValidDate
getDigits
DaynumToDate
isFriday
isMonday
lastDayOfMonth
dateToDaynum

Figure 13.4 First steps in bottom–up integration.

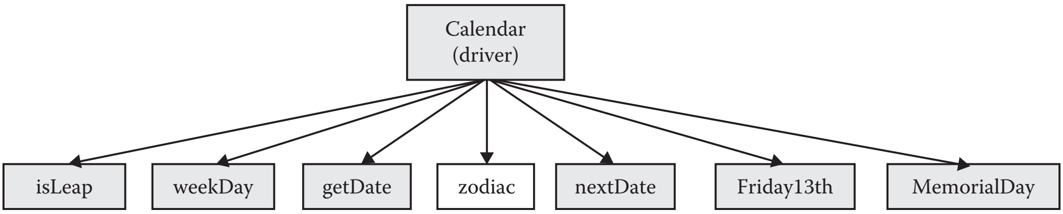

Calendar (driver)
isLeap
weekDay
getDate
zodiac
nextDate
Friday13th
MemorialDay

Figure 13.5 Bottom–up integration for zodiac.

13.1.3 Sandwich Integration

Sandwich integration is a combination of top–down and bottom–up integration. If we think about it in terms of the decomposition tree, we are really only doing big bang integration on a subtree (see Figure 13.6). There will be less stub and driver development effort, but this will be offset to some extent by the added difficulty of fault isolation that is a consequence of big bang integration. (We could probably discuss the size of a sandwich, from dainty finger sandwiches to Dagwood-style sandwiches, but not now.)

A sandwich is a full path from the root to leaves of the functional decomposition tree. In Figure 13.6, the set of units is almost semantically coherent, except that isLeap is missing. This set of units could be meaningfully integrated, but test cases at the end of February would not be covered. Also note that the fault isolation capability of the top–down and bottom–up approaches is sacrificed. No stubs nor drivers are needed in sandwich integration.

13.1.4 Pros and Cons

With the exception of big bang integration, the decomposition-based approaches are all intuitively clear. Build with tested components. Whenever a failure is observed, the most recently added unit is suspected. Integration testing progress is easily tracked against the decomposition tree. (If the tree is small, it is a nice touch to shade in nodes as they are successfully integrated.) The top–down and bottom–up terms suggest breadth-first traversals of the decomposition tree, but this is not mandatory. (We could use full-height sandwiches to test the tree in a depth-first manner.)

One of the most frequent objections to functional decomposition and waterfall development is that both are artificial, and both serve the needs of project management more than the needs of software developers. This holds true also for decomposition-based testing. The whole mechanism is that units are integrated with respect to structure; this presumes that correct behavior follows from individually correct units and correct interfaces. (Practitioners know better.) The development effort for stubs or drivers is another drawback to these approaches, and this is compounded by the retesting effort.

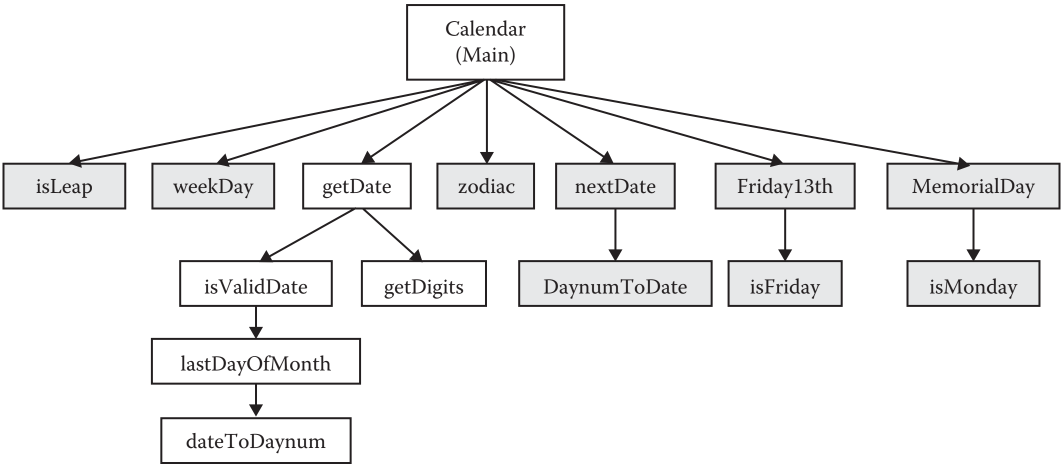


Figure 13.6 Sample sandwich integration.

13.2 Call Graph–Based Integration

One of the drawbacks of decomposition-based integration is that the basis is the functional decomposition tree. We saw that this leads to impossible test pairs. If we use the call graph instead, we resolve this deficiency; we also move in the direction of structural testing. The call graph is developed by considering units to be nodes, and if unit A calls (or uses) unit B, there is an edge from node A to node B. The call graph for the Calendar program is shown in Figure 13.7.

Since edges in the call graph refer to actual execution–time connections, the call graph avoids all the problems we saw in the decomposition tree–based versions of integration. In fact, we could repeat the discussion of Section 13.1 based on stubs and drivers in the units in Figure 13.7. This will work well, and it preserves the fault isolation feature of the decomposition-based approaches. Figure 13.8 shows the first step in call graph–based top–down integration.

The stubs in the first session could operate as follows. When the Calendar main program calls getDateStub, the stub might return May 27, 2013. The zodiacStub would return "Gemini," and so on. Once the main program logic is tested, the stubs would be replaced, as we discussed in Section 13.1. The three strategies of Section 13.1 will all work well when stubs and drivers are based on the call graph rather than the functional decomposition.

We are in a position to enjoy the investment we made in the discussion of graph theory. Because the call graph is a directed graph, why not use it the way we used program graphs? This leads us to two new approaches to integration testing: we will refer to them as pairwise integration and neighborhood integration.

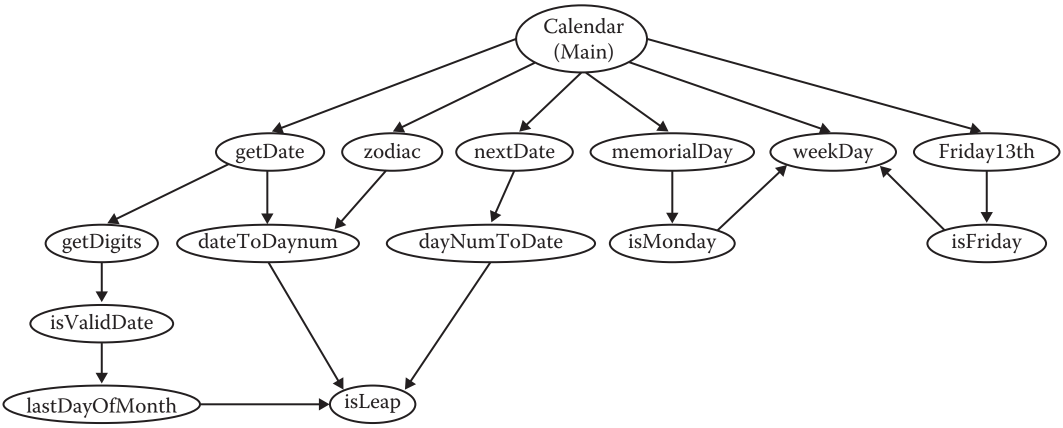

Calendar (Main)
getDate
zodiac
nextDate
memorialDay
weekDay
Friday13th
getDigits
dateToDaynum
dayNumToDate
isMonday
isFriday
isValidDate
lastDayOfMonth
isLeap

Figure 13.7 Call graph of Calendar program.

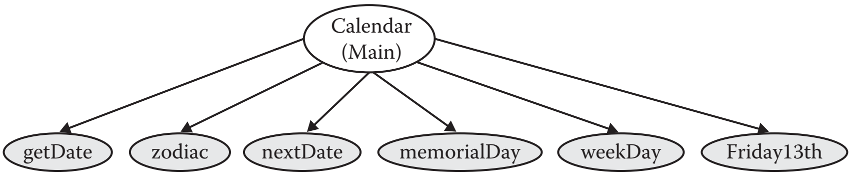

Calendar (Main)
getDate
zodiac
nextDate
memorialDay
weekDay
Friday13th

Figure 13.8 Call graph–based top–down integration of Calendar program.

13.2.1 Pairwise Integration

The idea behind pairwise integration is to eliminate the stub/driver development effort. Instead of developing stubs and/or drivers, why not use the actual code? At first, this sounds like big bang integration, but we restrict a session to only a pair of units in the call graph. The end result is that we have one integration test session for each edge in the call graph. Pairwise integration results in an increased number of integration sessions when a node (unit) is used by two or more other units. In the Calendar example, there would be 15 separate sessions for top–down integration (one for each stub replacement); this increases to 19 sessions for pairwise integration (one for each edge in the call graph). This is offset by a reduction in stub/driver development. Three pairwise integration sessions are shown in Figure 13.9.

The main advantage of pairwise integration is the high degree of fault isolation. If a test fails, the fault must be in one of the two units. The biggest drawback is that, for units involved on several pairs, a fix that works in one pair may not work in another pair. This is yet another example of the testing pendulum discussed in Chapter 10. Call graph integration is slightly better than the decomposition tree–based approach, but both can be removed from the reality of the code being tested.

13.2.2 Neighborhood Integration

We can let the mathematics carry us still further by borrowing the notion of a neighborhood from topology. (This is not too much of a stretch—graph theory is a branch of topology.) The neighborhood of a node in a graph is the set of nodes that are one edge away from the given node. (Technically, this is a neighborhood of radius 1; in larger systems, it makes sense to increase the neighborhood radius.) In a directed graph, this includes all the immediate predecessor nodes and all the immediate successor nodes (notice that these correspond to the set of stubs and drivers of the node). The neighborhoods of getDate, nextDate, Friday13th, and weekDay are shown in Figure 13.10.

The 15 neighborhoods for the Calendar example (based on the call graph in Figure 13.7) are listed in Table 13.1. To make the table simpler, the original unit names are replaced by node numbers (in Figure 13.11), where the numbering is generally breadth first. The juxtaposition and connectivity is preserved in Figure 13.11.

The information in Table 13.1 is given in Table 13.2 as the adjacency matrix for the call graph. The column sums show the indegrees of each node, and the row sums show the outdegrees.

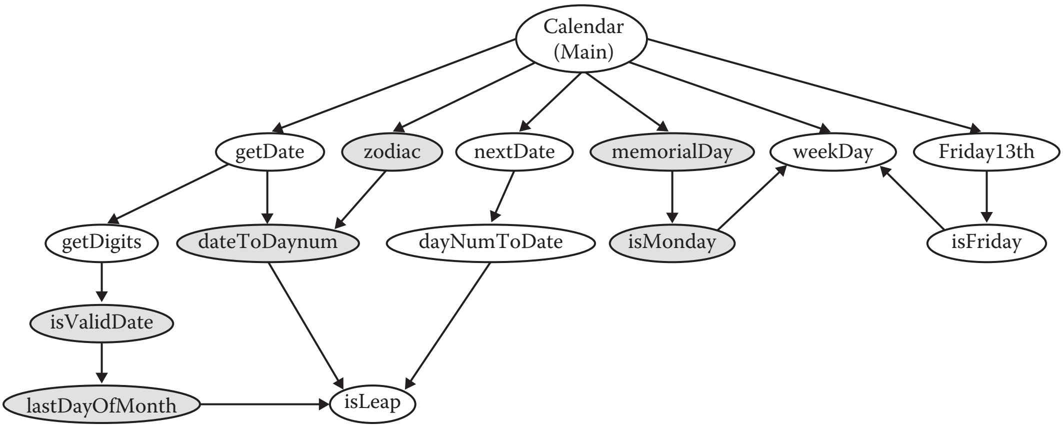


Figure 13.9 Three pairs for pairwise integration.

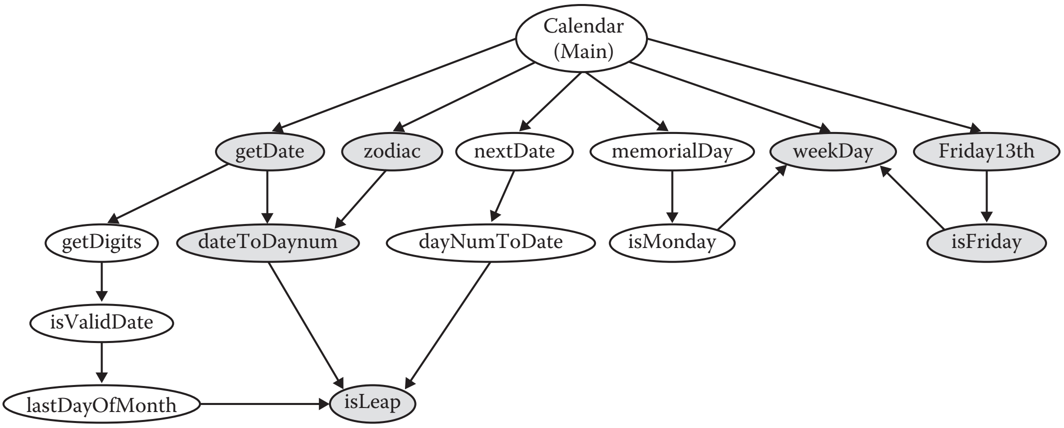

Calendar (Main)
getDate
zodiac
nextDate
memorialDay
weekDay
Friday13th
getDigits
dateToDaynum
dayNumToDate
isMonday
isFriday
isValidDate
lastDayOfMonth
isLeap

Figure 13.10 Three neighborhoods (of radius 1) for neighborhood integration.

Table 13.1 Neighborhoods of Radius 1 in Calendar Call Graph

Neighborhoods in Calendar Program Call Graph			
Node	*Unit Name*	*Predecessors*	*Successors*
1	Calendar (Main)	(None)	2, 3, 4, 5, 6, 7
2	getDate	1	8, 9
3	zodiac	1	9
4	nextDate	1	10
5	memorialDay	1	11
6	weekday	1, 11, 12	(None)
7	Friday13th	1	12
8	getDigits	2	13
9	dateToDayNum	3	15
10	dayNumToDate	4	15
11	isMonday	5	6
12	isFriday	7	6
13	isValidDate	8	14
14	lastDayOfMonth	13	15
15	isLeap	9, 10, 14	(None)

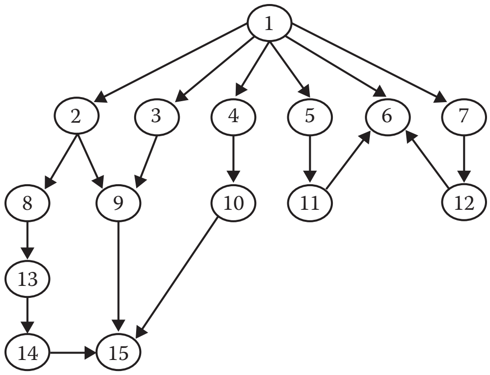


Figure 13.11 **Calendar call graph with units replaced by numbers.**

Table 13.2 **Adjacency Matrix of Calendar Call Graph**

	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*Row Sum*
1		1	1	1	1	1	1									6
2								1	1							2
3									1							1
4										1						1
5											1					1
6																0
7												1				1
8													1			1
9															1	1
10															1	1
11							1									1
12							1									1
13														1		1
14															1	1
15																0
Column sum	0	1	1	1	1	1	3	1	2	1	1	1	1	1	3	

We can always compute the number of neighborhoods for a given call graph. Each interior node will have one neighborhood, plus one extra in case leaf nodes are connected directly to the root node. (An interior node has a nonzero indegree and a nonzero outdegree.) We have

$$\text{Interior nodes} = \text{nodes} - (\text{source nodes} + \text{sink nodes})$$

$$\text{Neighborhoods} = \text{interior nodes} + \text{source nodes}$$

which combine to

$$\text{Neighborhoods} = \text{nodes} - \text{sink nodes}$$

Neighborhood integration usually yields a reduction in the number of integration test sessions, and it reduces stub and driver development. The end result is that neighborhoods are essentially the sandwiches that we slipped past in the previous section. (It is slightly different because the base information for neighborhoods is the call graph, not the decomposition tree.) What they share with sandwich integration is more significant—neighborhood integration testing has the fault isolation difficulties of "medium bang" integration.

13.2.3 Pros and Cons

The call graph–based integration techniques move away from a purely structural basis toward a behavioral basis; thus, the underlying assumption is an improvement. (See the Testing Pendulum in Chapter 10.) The neighborhood-based techniques also reduce the stub/driver development effort. In addition to these advantages, call graph–based integration matches well with developments characterized by builds and composition. For example, sequences of neighborhoods can be used to define builds. Alternatively, we could allow adjacent neighborhoods to merge (into villages?) and provide an orderly, composition-based growth path. All this supports the use of neighborhood-based integration for systems developed by life cycles in which composition dominates.

The biggest drawback to call graph–based integration testing is the fault isolation problem, especially for large neighborhoods. A more subtle but closely related problem occurs. What happens if (when) a fault is found in a node (unit) that appears in several neighborhoods? The adjacency matrix highlights this immediately—nodes with either a high row sum or a high column sum will be in several neighborhoods. Obviously, we resolve the fault in one neighborhood; but this means changing the unit's code in some way, which in turn means that all the previously tested neighborhoods that contain the changed node need to be retested.

Finally, a fundamental uncertainty exists in any structural form of testing: the presumption that units integrated with respect to structural information will exhibit correct behavior. We know where we are going: we want system-level threads of behavior to be correct. When integration testing based on call graph information is complete, we still have quite a leap to get to system-level threads. We resolve this by changing the basis from call graph information to special forms of paths.

13.3 Path-Based Integration

Much of the progress in the development of mathematics comes from an elegant pattern: have a clear idea of where you want to go, and then define the concepts that take you there. We do this here for path-based integration testing, but first we need to motivate the definitions.

We already know that the combination of structural and functional testing is highly desirable at the unit level; it would be nice to have a similar capability for integration (and system) testing. We also know that we want to express system testing in terms of behavioral threads. Lastly, we revise our goal for integration testing: instead of testing interfaces among separately developed and tested units, we focus on interactions among these units. ("Co-functioning" might be a good term.) Interfaces are structural; interaction is behavioral.

When a unit executes, some path of source statements is traversed. Suppose that a call goes to another unit along such a path. At that point, control is passed from the calling unit to the called unit, where some other path of source statements is traversed. We deliberately ignored this situation in Chapter 8, because this is a better place to address the question. Two possibilities are available: abandon the single-entry, single-exit precept and treat such calls as an exit followed by an entry, or suppress the call statement because control eventually returns to the calling unit anyway. The suppression choice works well for unit testing, but it is antithetical to integration testing.

13.3.1 New and Extended Concepts

To get where we need to go, we need to refine some of the program graph concepts. As before, these refer to programs written in an imperative language. We allow statement fragments to be a complete statement, and statement fragments are nodes in the program graph.

Definition

A *source node* in a program is a statement fragment at which program execution begins or resumes.

The first executable statement in a unit is clearly a source node. Source nodes also occur immediately after nodes that transfer control to other units.

Definition

A *sink node* in a unit is a statement fragment at which program execution terminates.

The final executable statement in a program is clearly a sink node; so are statements that transfer control to other units.

Definition

A *module execution path* is a sequence of statements that begins with a source node and ends with a sink node, with no intervening sink nodes.

The effect of the definitions thus far is that program graphs now have multiple source and sink nodes. This would greatly increase the complexity of unit testing, but integration testing presumes unit testing is complete.

Definition

A *message* is a programming language mechanism by which one unit transfers control to another unit, and acquires a response from the other unit.

Depending on the programming language, messages can be interpreted as subroutine invocations, procedure calls, function references, and the usual messages in an object-oriented programming language. We follow the convention that the unit that receives a message (the message destination) always eventually returns control to the message source. Messages can pass data to other units. We can finally make the definitions for path-based integration testing. Our goal is to have an integration testing analog of DD-paths.

Definition

An *MM-path* is an interleaved sequence of module execution paths and messages.

The basic idea of an MM-path (Jorgensen 1985; Jorgensen and Erickson 1994) is that we can now describe sequences of module execution paths that include transfers of control among separate units. In traditional software, "MM" is nicely understood as module–message; in object-oriented software, it is clearer to interpret "MM" as method–message. These transfers are by messages, therefore, MM-paths always represent feasible execution paths, and these paths cross unit boundaries. The hypothetical example in Figure 13.12 shows an MM-path (the solid edges) in which module A calls module B, which in turn calls module C. Notice that, for traditional (procedural) software, MM-paths will always begin (and end) in the main program.

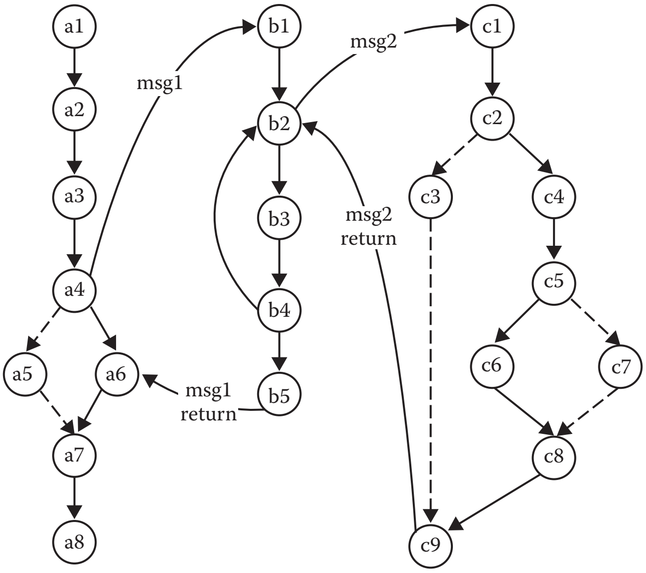


Figure 13.12 Hypothetical MM-path across three units.

In unit A, nodes a1 and a6 are source nodes (a5 and a6 are outcomes of the decision at node a5), and nodes a4 (a decision) and a8 are sink nodes. Similarly, in unit B, nodes b1 and b3 are source nodes, and nodes b2 and b5 are sink nodes. Node b2 is a sink node because control leaves unit B at that point. It could also be a source node, because unit C returns a value used at node b2. Unit C has a single source node, c1, and a single sink node, c9. Unit A contains three module execution paths: <a1, a2, a3, a4>, <a4, a5, a7, a8>, and <a4, a6, a7, a8>. The solid edges are edges actually traversed in this hypothetical example. The dashed edges are in the program graphs of the units as stand-alone units, but they did not "execute" in the hypothetical MM-path. We can now define an integration testing analog of the DD-path graph that serves unit testing so effectively.

Definition

Given a set of units, their *MM-path graph* is the directed graph in which nodes are module execution paths and edges correspond to messages and returns from one unit to another.

Notice that MM-path graphs are defined with respect to a set of units. This directly supports composition of units and composition-based integration testing. We can even compose down to the level of individual module execution paths, but that is probably more detailed than necessary.

We should consider the relationships among module execution paths, program paths, DD-paths, and MM-paths. A program path is a sequence of DD-paths, and an MM-path is a sequence of module execution paths. Unfortunately, there is no simple relationship between DD-paths and module execution paths. Either might be contained in the other, but more likely, they partially overlap. Because MM-paths implement a function that transcends unit boundaries, we do have one relationship: consider the intersection of an MM-path with a unit. The module execution paths in such an intersection are an analog of a slice with respect to the (MM-path) function. Stated another way, the module execution paths in such an intersection are the restriction of the function to the unit in which they occur.

The MM-path definition needs some practical guidelines. How long ("deep" might be better) is an MM-path? The notion of message quiescence helps here. Message quiescence occurs when a unit that sends no messages is reached (like module C in Figure 13.12). In a sense, this could be taken as a "midpoint" of an MM-path—the remaining execution consists of message returns. This is only mildly helpful. What if there are two points of message quiescence? Maybe a better answer is to take the longer of the two, or, if they are of equal depth, the latter of the two. Points of message quiescence are natural endpoints for an MM-path.

13.3.2 MM-Path Complexity

If you compare the MM-paths in Figures 13.12 and 13.17, it seems intuitively clear that the latter is more complex than the former. Because these are strongly connected directed graphs, we can "blindly" compute their cyclomatic complexities; recall the formula is $V(G) = e - n + 2p$, where p is the number of strongly connected regions. Since messages return to the sending unit, we will always have $p = 1$, so the formula reduces to $V(G) = e - n + 2$. Surprisingly, both graphs have $V(G) = 7$. Clearly, MM-path complexity needs some notion of size in addition to cyclomatic complexity (Figure 13.13).

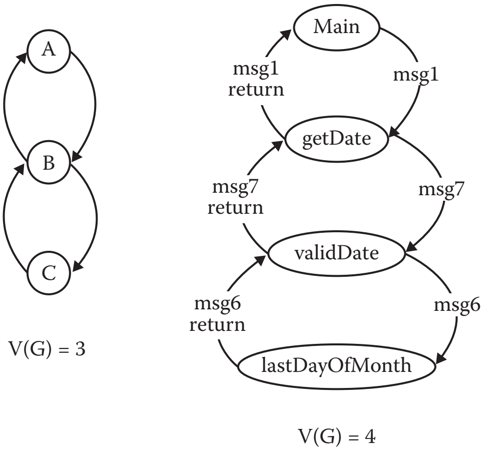


Figure 13.13 Cyclomatic complexities of two MM-paths.

13.3.3 Pros and Cons

MM-paths are a hybrid of functional and structural testing. They are functional in the sense that they represent actions with inputs and outputs. As such, all the functional testing techniques are potentially applicable. The net result is that the cross-check of the functional and structural approaches is consolidated into the constructs for path-based integration testing. We therefore avoid the pitfall of structural testing, and, at the same time, integration testing gains a fairly seamless junction with system testing. Path-based integration testing works equally well for software developed in the traditional waterfall process or with one of the composition-based alternative life cycle models. Finally, the MM-path concept applies directly to object-oriented software.

The most important advantage of path-based integration testing is that it is closely coupled with actual system behavior, instead of the structural motivations of decomposition and call graph–based integration. However, the advantages of path-based integration come at a price—more effort is needed to identify the MM-paths. This effort is probably offset by the elimination of stub and driver development.

13.4 Example: integrationNextDate

Our now familiar NextDate is rewritten here as a main program with a functional decomposition into procedures and functions. This integrationNextDate is a slight extension: there is added validity checking for months, days, and years, so the pseudocode, which follows Figures 13.14 and 13.15, grows from 50 statements to 81. Figures 13.14 and 13.15 show the functional decomposition and the call graph, respectively. Figure 13.16 shows the program graphs of the units in integrationNextDate. Figure 13.17 shows the MM-path for the input date May 27, 2012.

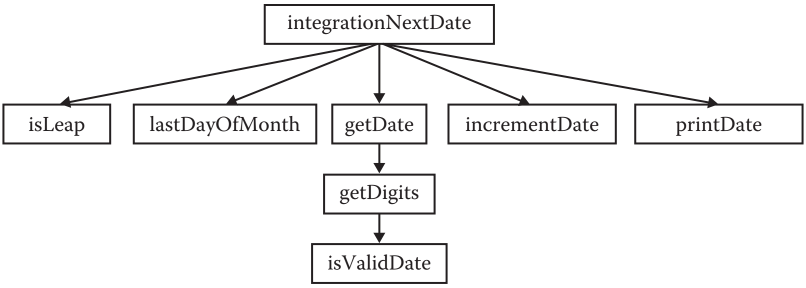

integrationNextDate
isLeap
lastDayOfMonth
getDate
incrementDate
printDate
getDigits
isValidDate

Figure 13.14 Functional decomposition of integrationNextDate.

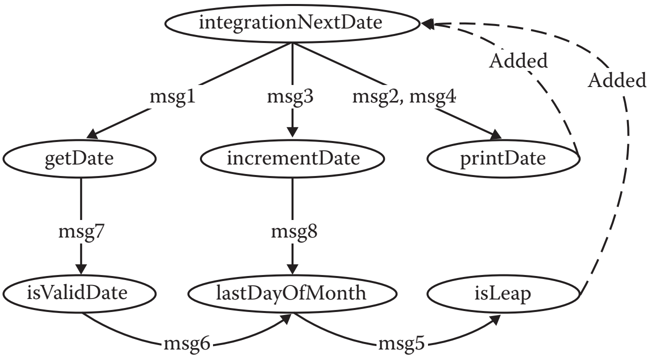

integrationNextDate
Added
Added
msg1
msg3
msg2, msg4
getDate
incrementDate
printDate
msg7
msg8
isValidDate
lastDayOfMonth
isLeap
msg6
msg5

Figure 13.15 Call graph of integrationNextDate.

13.4.1 Decomposition-Based Integration

The isLeap and lastDayOfMonth functions are in the first level of decomposition because they must be available to both GetDate and IncrementDate. (We could move isLeap to be contained within the scope of lastDayOfMonth.) Pairwise integration based on the decomposition in Figure 13.14 is problematic; the isLeap and lastDayOfMonth functions are never directly called by the Main program, so these integration sessions would be empty. Bottom–up pairwise integration starting with isLeap, then lastDayOfMonth, ValidDate, and GetDate would be useful. The pairs involving Main and GetDate, IncrementDate, and PrintDate are all useful (but short) sessions. Building stubs for ValidDate and lastDayOfMonth would be easy.

13.4.2 Call Graph–Based Integration

Pairwise integration based on the call graph in Figure 13.15 is an improvement over that for the decomposition-based pairwise integration. Obviously, there are no empty integration sessions because edges refer to actual unit references. There is still the problem of stubs. Sandwich integration is appropriate because this example is so small. In fact, it lends itself to a build sequence. Build 1 could contain Main and PrintDate. Build 2 could contain Main, IncrementDate, lastDayOfMonth, and IncrementDate in addition to the already present PrintDate. Finally, build 3 would add the remaining units, GetDate and ValidDate.

Neighborhood integration based on the call graph would likely proceed with the neighborhoods of ValidDate and lastDayOfMonth. Next, we could integrate the neighborhoods of GetDate and IncrementDate. Finally, we would integrate the neighborhood of Main. Notice that these neighborhoods form a build sequence.

integrationNextDate pseudocode

```
Main integrationNextDate ‘start program event occurs here
   Type         Date
                Month As Integer
                Day As Integer
                Year As Integer
   EndType
   Dim today, tomorrow As Date
   Output(“Welcome to NextDate!”)
   GetDate(today)                                                    ‘msg1
   PrintDate(today)                                                  ‘msg2
   tomorrow = IncrementDate(today)                                   ‘msg3
   PrintDate(tomorrow)                                               ‘msg4
End Main
Function isLeap(year) Boolean
   If (year divisible by 4)
         Then
            If (year is NOT divisible by 100)
                  Then isLeap = True
                  Else
                      If (year is divisible by 400)
                         Then isLeap = True
                         Else isLeap = False
                  EndIf
            EndIf
Else isLeap = False
EndIf
End (Function isLeap)

Function lastDayOfMonth(month, year)   Integer
   Case month Of
         Case 1: 1, 3, 5, 7, 8, 10, 12
            lastDayOfMonth = 31
         Case 2: 4, 6, 9, 11
            lastDayOfMonth = 30
         Case 3: 2
            If (isLeap(year))                                        ‘msg5
                  Then lastDayOfMonth = 29
                  Else lastDayOfMonth = 28
            EndIf
   EndCase
End (Function lastDayOfMonth)
Function GetDate(aDate)  Date
   dim aDate As Date

   Function ValidDate(aDate)    Boolean ‘within scope of GetDate
         dim aDate As Date
         dim dayOK, monthOK, yearOK As Boolean
         If ((aDate.Month > 0) AND (aDate.Month <=12)
            Then      monthOK = True
                      Output(“Month OK”)
            Else  monthOK = False
                      Output(“Month out of range”)
         EndIf
         If (monthOK)
            Then
            If ((aDate.Day > 0) AND (aDate.Day <=
            lastDayOfMonth(aDate.Month, aDate.Year))                   ‘msg6
                      Then      dayOK = True
                                Output(“Day OK”)
                      Else      dayOK = False
                                Output(“Day out of range”)
                  EndIf
            EndIf
         If ((aDate.Year > 1811) AND (aDate.Year <=2012)
            Then      yearOK = True
                      Output(“Year OK”)
            Else      yearOK = False
                      Output(“Year out of range”)
         EndIf
         If (monthOK AND dayOK AND yearOK)
            Then ValidDate = True
                      Output(“Date OK”)
            Else ValidDate = False
                  Output(“Please enter a valid date”)
         EndIf
   End (Function ValidDate)

‘ GetDate body begins here
   Do
         Output(“enter a month”)
         Input(aDate.Month)
         Output(“enter a day”)
         Input(aDate.Day)
         Output(“enter a year”)
         Input(aDate.Year)
         GetDate.Month = aDate.Month
         GetDate.Day = aDate.Day
         GetDate.Year = aDate.Year
   Until (ValidDate(aDate))                                            ‘msg7
   End (Function GetDate)
   Function IncrementDate(aDate)Date
         If (aDate.Day < lastDayOfMonth(aDate.Month))                  ‘msg8
            Then aDate.Day = aDate.Day + 1
            Else aDate.Day = 1
                  If (aDate.Month = 12)
                      Then      aDate.Month = 1
                                aDate.Year = aDate.Year + 1
                      Else      aDate.Month = aDate.Month + 1
                  EndIf
         EndIf
End (IncrementDate)

Procedure PrintDate(aDate)
   Output(“Day is “, aDate.Month, “/”, aDate.Day, “/”, aDate.Year)
End (PrintDate)
```

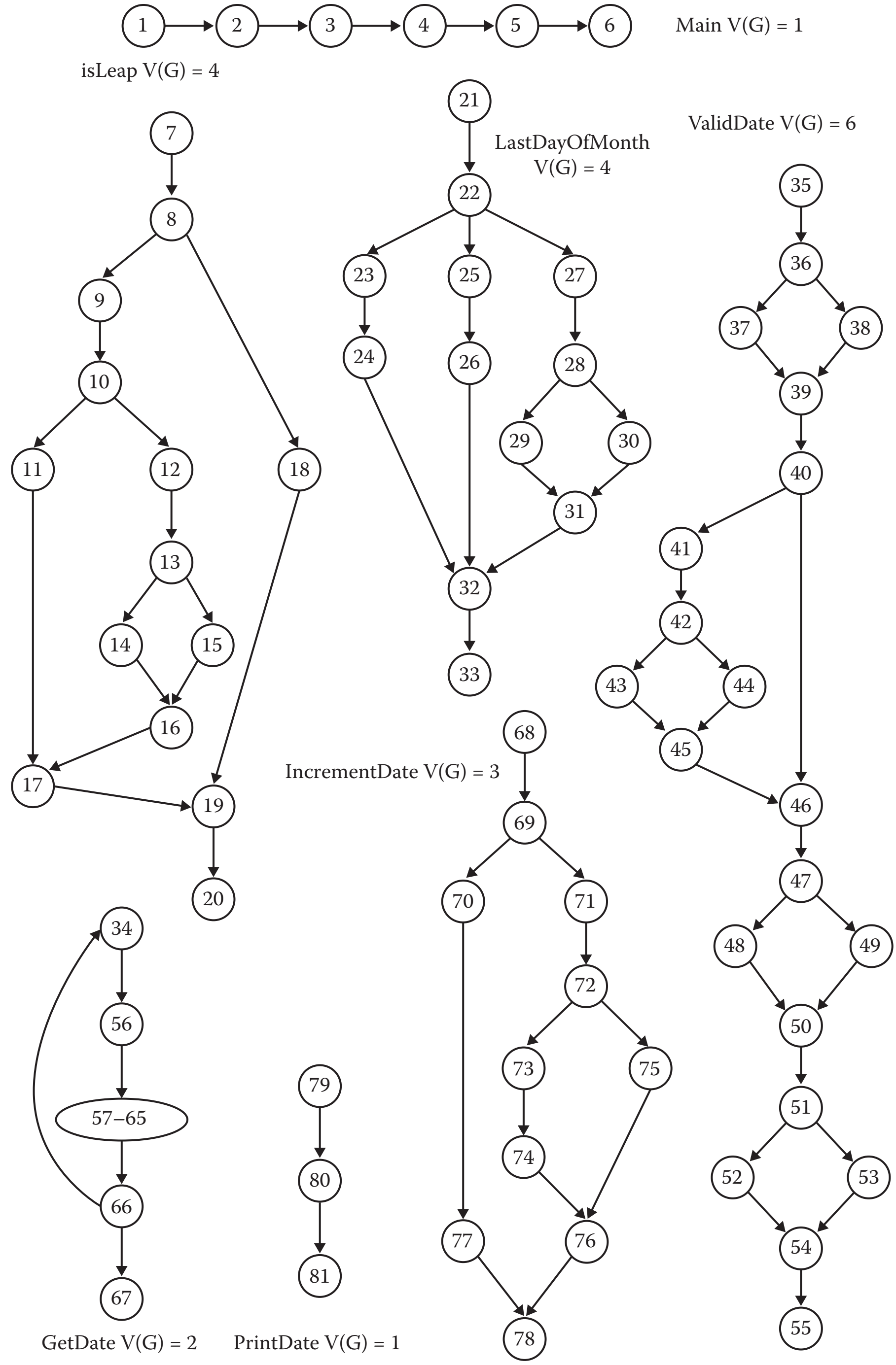

Main V(G) = 1
1
2
3
4
5
6
isLeap V(G) = 4
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
LastDayOfMonth
V(G) = 4
22
23
24
25
26
27
28
29
30
31
32
33
ValidDate V(G) = 6
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
IncrementDate V(G) = 3
68
69
70
71
72
73
74
75
76
77
78
34
56
57–65
66
67
79
80
81
GetDate V(G) = 2
PrintDate V(G) = 1

Figure 13.16 Program graphs of units in integrationNextDate.

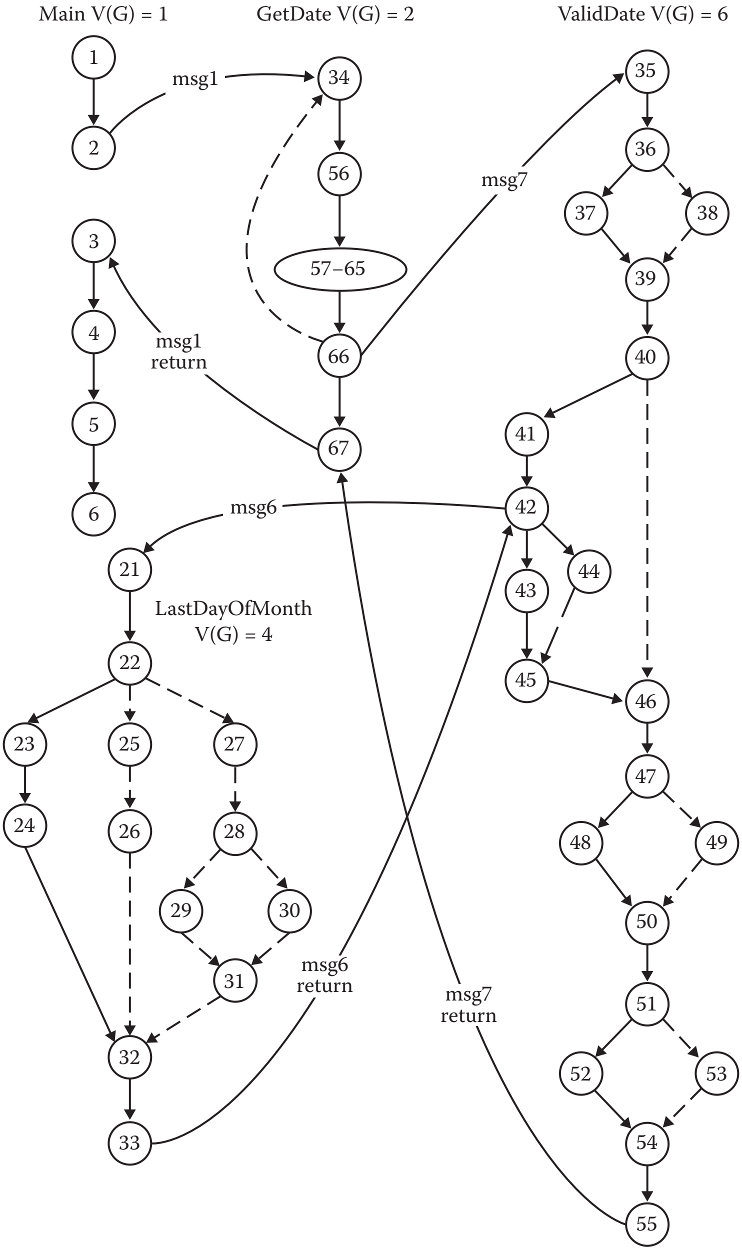

Main V(G) = 1
GetDate V(G) = 2
ValidDate V(G) = 6
LastDayOfMonth V(G) = 4
msg1
msg1 return
msg6
msg6 return
msg7
msg7 return
57–65
1
2
3
4
5
6
21
22
23
24
25
26
27
28
29
30
31
32
33
34
56
66
67
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55

Figure 13.17 MM-path for May 27, 2012.

13.4.3 MM-Path-Based Integration

Because the program is data-driven, all MM-paths begin in and return to the main program. Here is the first MM-path for May 27, 2012 (there are others when the Main program calls PrintDate and IncrementDate). It is shown in Figure 13.17.

```
Main (1, 2)
     msg1
     GetDate (34, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66)
           msg7
           validDate (35, 36, 37, 39, 40, 41, 42))
                 msg6
                 lastDayOfMonth (21, 22, 23, 24, 32, 33)
                     'point of message quiescence
           ValidDate (43, 45, 46, 47, 48, 50, 51, 52, 54, 55)
     GetDate (67)
Main (3)
```

We are now in a strong position to describe how many MM-paths are sufficient: the set of MM-paths should cover all source-to-sink paths in the set of units. This is subtly present in Figure 13.17. The solid edges are in the MM-path, but the dashed edges are not. When loops are present, condensation graphs will result in directed acyclic graphs, thereby resolving the problem of potentially infinite (or excessively large) number of paths.

13.5 Conclusions and Recommendations

Table 13.3 summarizes the observations made in the preceding discussion. The significant improvement of MM-paths as a basis for integration testing is due to their exact representation of dynamic software behavior. MM-paths are also the basis for present research in data flow (define/use) approaches to integration testing. Integration testing with MM-paths requires extra effort. As a fallback position, perform integration testing based on call graphs.

Table 13.3 Comparison of Integration Testing Strategies

Strategy Basis	*Ability to Test Interfaces*	*Ability to Test Co-Functionality*	*Fault Isolation Resolution*
Functional decomposition	Acceptable but can be deceptive	Limited to pairs of units	Good, to faulty unit
Call graph	Acceptable	Limited to pairs of units	Good, to faulty unit
MM-path	Excellent	Complete	Excellent, to faulty unit execution path

EXERCISES

1. Find the source and sink nodes in isValidateDate and in getDate.
2. Write driver modules for isValidateDate and in getDate.
3. Write stubs for isValidateDate and in getDate.
4. Here are some other possible complexity metrics for MM-paths:
   $V(G) = e - n$
   $V(G) = 0.5e - n + 2$
   sum of the outdegrees of the nodes
   sum of the nodes plus the sum of the edges

   Make up some examples, try these out, and see if they have any explanatory value.
5. Make up a few test cases, interpret them as MM-paths, and then see what portions of the unit program graphs in Figure 13.16 are traversed by your MM-paths. Try to devise a "coverage metric" for MM-path-based integration testing.
6. One of the goals of integration testing is to be able to isolate faults when a test case causes a failure. Consider integration testing for a program written in a procedural programming language. Rate the relative fault isolation capabilities of the following integration strategies:
   A = Decomposition based top–down integration
   B = Decomposition based bottom–up integration
   C = Decomposition based sandwich integration
   D = Decomposition based "big bang" integration
   E = Call graph–based pairwise integration
   F = Call graph–based neighborhood integration (radius = 2)
   G = Call graph–based neighborhood integration (radius = 1)

   Show your ratings graphically by placing the letters corresponding to a strategy on the continuum below. As an example, suppose Strategies X and Y are about equal and not very effective, and Strategy Z is very effective.

```
     Y
     X                                                   Z
Low                                                   High
```

References

Deutsch, M.S., *Software Verification and Validation-Realistic Project Approaches*, Prentice-Hall, Englewood Cliffs, NJ, 1982.

Fordahl, M., *Elementary Mistake Doomed Mars Probe*, The Associated Press, available at http://mars.jpl.nasa.gov/msp98/news/mco990930.html, October 1, 1999.

Hetzel, B., *The Complete Guide to Software Testing*, 2nd ed., QED Information Sciences, Inc., Wellesley, MA, 1988.

Jorgensen, P.C., *The Use of MM-Paths in Constructive Software Development*, Ph.D. dissertation, Arizona State University, Tempe, AZ, 1985.

Jorgensen, P.C. and Erickson, C., Object-oriented integration testing, *Communications of the ACM*, September 1994.

Kaner, C., Falk, J. and Nguyen, H.Q., *Testing Computer Software*, 2nd ed., Van Nostrand Reinhold, New York, 1993.

Mosley, D.J., *The Handbook of MIS Application Software Testing*, Yourdon Press, Prentice-Hall, Englewood Cliffs, NJ, 1993.

Pressman, R.S., *Software Engineering: A Practitioner's Approach*, 6th ed., McGraw-Hill, New York, 2005.

Schach, S.R., *Object-Oriented and Classical Software Engineering*, 5th ed., McGraw-Hill, New York, 2002.

Chapter 14

System Testing

Of the three levels of testing, the system level is closest to everyday experience. We test many things: a used car before we buy it, an online network service before we subscribe, and so on. A common pattern in these familiar forms is that we evaluate a product in terms of our expectations—not with respect to a specification or a standard. Consequently, the goal is not to find faults but to demonstrate correct behavior. Because of this, we tend to approach system testing from a specification-based standpoint instead of from a code-based one. Because it is so intuitively familiar, system testing in practice tends to be less formal than it might be, and this is compounded by the reduced testing interval that usually remains before a delivery deadline.

The craftsperson metaphor continues to serve us. We need a better understanding of the medium; we will view system testing in terms of threads of system-level behavior. We begin with a new construct—an Atomic System Function (ASF)—and develop the thread concept, highlighting some of the practical problems of thread-based system testing. System testing is closely coupled with requirements specification; therefore, we shall use appropriate system-level models to enjoy the benefits of model-based testing. Common to all of these is the idea of "threads," so we shall see how to identify system-level threads in a variety of common models. All this leads to an orderly thread-based system testing strategy that exploits the symbiosis between specification-based and code-based testing. We will apply the strategy to our simple automated teller machine (SATM) system, first described in Chapter 2.

14.1 Threads

Threads are hard to define; in fact, some published definitions are counterproductive, misleading, or wrong. It is possible to simply treat threads as a primitive concept that needs no formal definition. For now, we will use examples to develop a "shared vision." Here are several views of a thread:

A scenario of normal usage
A system-level test case
A stimulus/response pair
Behavior that results from a sequence of system-level inputs

An interleaved sequence of port input and output events
A sequence of transitions in a state machine description of the system
An interleaved sequence of object messages and method executions
A sequence of machine instructions
A sequence of source instructions
A sequence of MM-paths
A sequence of ASFs (to be defined in this chapter)

Threads have distinct levels. A unit-level thread is usefully understood as an execution–time path of source instructions or, alternatively, as a sequence of DD-paths. An integration-level thread is an MM-path—that is, an alternating sequence of module execution paths and messages. If we continue this pattern, a system-level thread is a sequence of ASFs. Because ASFs have port events as their inputs and outputs, a sequence of ASFs implies an interleaved sequence of port input and output events. The end result is that threads provide a unifying view of our three levels of testing. Unit testing tests individual functions; integration testing examines interactions among units; and system testing examines interactions among ASFs. In this chapter, we focus on system-level threads and answer some fundamental questions, such as, “How big is a thread? Where do we find them? How do we test them?”

14.1.1 Thread Possibilities

Defining the endpoints of a system-level thread is a bit awkward. We motivate a tidy, graph theory–based definition by working backward from where we want to go with threads. Here are four candidate threads in our SATM system:

Entry of a digit
Entry of a personal identification number (PIN)
A simple transaction: ATM card entry, PIN entry, select transaction type (deposit, withdraw), present account details (checking or savings, amount), conduct the operation, and report the results
An ATM session containing two or more simple transactions

Digit entry is a good example of a minimal ASF. It begins with a port input event (the digit keystroke) and ends with a port output event (the screen digit echo), so it qualifies as a stimulus/response pair. This level of granularity is too fine for the purposes of system testing.

The second candidate, PIN entry, is a good example of an upper limit to integration testing and, at the same time, a starting point of system testing. PIN entry is a good example of an ASF. It is also a good example of a family of stimulus/response pairs (system-level behavior that is initiated by a port input event, traverses some programmed logic, and terminates in one of several possible responses [port output events]). PIN entry entails a sequence of system-level inputs and outputs.

1. A screen requesting PIN digits.
2. An interleaved sequence of digit keystrokes and screen responses.
3. The possibility of cancellation by the customer before the full PIN is entered.
4. A system disposition: a customer has three chances to enter the correct PIN. Once a correct PIN has been entered, the user sees a screen requesting the transaction type; otherwise, a screen advises the customer that the ATM card will not be returned, and no access to ATM functions is provided.

Several stimulus/response pairs are evident, putting ASFs clearly in the domain of system-level testing. Other examples of ASFs include card entry, transaction selection, provision of transaction details, transaction reporting, and session termination. Each of these is maximal in an integration testing sense and minimal in a system testing sense. That is, we would not want to integration test something larger than an ASF; at the same time, we would not want to test anything smaller as part of system testing.

The third candidate, the simple transaction, has a sense of "end-to-end" completion. A customer could never execute PIN entry alone (a card entry is needed), but the simple transaction is commonly executed. This is a good example of a system-level thread; note that it involves the interaction of several ASFs.

The last possibility (the session) is actually a sequence of threads. This is also properly a part of system testing; at this level, we are interested in the interactions among threads. Unfortunately, most system testing efforts never reach the level of thread interaction.

14.1.2 Thread Definitions

We simplify our discussion by defining a new term that helps us get to our desired goal.

Definition

An *Atomic System Function (ASF)* is an action that is observable at the system level in terms of port input and output events.

In an event-driven system, ASFs are separated by points of event quiescence; these occur when a system is (nearly) idle, waiting for a port input event to trigger further processing. Event quiescence has an interesting Petri net insight. In a traditional Petri net, deadlock occurs when no transition is enabled. In an Event-Driven Petri Net (defined in Chapter 4), event quiescence is similar to deadlock; but an input event can bring new life to the net. The SATM system exhibits event quiescence in several places: one is the tight loop at the beginning of an ATM session, where the system has displayed the welcome screen and is waiting for a card to be entered into the card slot. Event quiescence is a system-level property; it is a direct analog of message quiescence at the integration level.

The notion of event quiescence does for ASFs what message quiescence does for MM-paths—it provides a natural endpoint. An ASF begins with a port input event and terminates with a port output event. When viewed from the system level, no compelling reason exists to decompose an ASF into lower levels of detail (hence, the atomicity). In the SATM system, digit entry is a good example of an ASF—so are card entry, cash dispensing, and session closing. PIN entry is probably too big; perhaps we should call it a molecular system function.

Atomic system functions represent the seam between integration and system testing. They are the largest item to be tested by integration testing and the smallest item for system testing. We can test an ASF at both levels. We will revisit the integrationNextDate program to find ASFs in Section 14.10.

Definition

Given a system defined in terms of ASFs, the *ASF graph* of the system is the directed graph in which nodes are ASFs and edges represent sequential flow.

Definition

A *source ASF* is an Atomic System Function that appears as a source node in the ASF graph of a system; similarly, a *sink ASF* is an Atomic System Function that appears as a sink node in the ASF graph.

In the SATM system, the card entry ASF is a source ASF, and the session termination ASF is a sink ASF. Notice that intermediary ASFs could never be tested at the system level by themselves—they need the predecessor ASFs to "get there."

Definition

A *system thread* is a path from a source ASF to a sink ASF in the ASF graph of a system.

These definitions provide a coherent set of increasingly broader views of threads, starting with very short threads (within a unit) and ending with interactions among system-level threads. We can use these views much like the ocular on a microscope, switching among them to see different levels of granularity. Having these concepts is only part of the problem; supporting them is another. We next take a tester's view of requirements specification to see how to identify threads.

14.2 Basis Concepts for Requirements Specification

Recall the notion of a basis of a vector space: a set of independent elements from which all the elements in the space can be generated (see problem 9, Chapter 8). Instead of anticipating all the variations in scores of requirements specification methods, notations, and techniques, we will discuss system testing with respect to a basis set of requirements specification constructs: data, actions, devices, events, and threads. Every system can be modeled in terms of these five fundamental concepts (and every requirements specification model uses some combination of these). We examine these fundamental concepts here to see how they support the tester's process of thread identification.

14.2.1 Data

When a system is described in terms of its data, the focus is on the information used and created by the system. We describe data in terms of variables, data structures, fields, records, data stores, and files. Entity/relationship (E/R) models are the most common choice at the highest level, and some form of a regular expression (e.g., Jackson diagrams or data structure diagrams) is used at a more detailed level. The data-centered view is also the starting point for several flavors of object-oriented analysis. Data refers to information that is either initialized, stored, updated, or (possibly) destroyed. In the SATM system, initial data describes the various accounts (each with its Personal Account Number, or PAN) and their PINs, and each account has a data structure with information such as the account balance. As ATM transactions occur, the results are kept as created data and used in the daily posting of terminal data to the central bank. For many systems, the data-centered view dominates. These systems are often developed in terms of CRUD actions (Create, Retrieve, Update, Delete). We could describe the transaction portion of the SATM system in this way, but it would not work well for the user interface portion.

Sometimes threads can be identified directly from the data model. Relationships among data entities can be one-to-one, one-to-many, many-to-one, or many-to-many; these distinctions all have implications for threads that process the data. For example, if bank customers can have several accounts, each account needs a unique PIN. If several people can access the same account, they need

ATM cards with identical PANs. We can also find initial data (such as PAN, ExpectedPIN pairs) that are read but never written. Such read-only data must be part of the system initialization process. If not, there must be threads that create such data. Read-only data is therefore an indicator of source ASFs.

14.2.2 Actions

Action-centered modeling is still a common requirements specification form. This is a historical outgrowth of the action-centered nature of imperative programming languages. Actions have inputs and outputs, and these can be either data or port events. Here are some methodology-specific synonyms for actions: transform, data transform, control transform, process, activity, task, method, and service. Actions can also be decomposed into lower-level actions, most notably in the data flow diagrams of Structured Analysis. The input/output view of actions is exactly the basis of specification-based testing, and the decomposition (and eventual implementation) of actions is the basis of code-based testing.

14.2.3 Devices

Every system has port devices; these are the sources and destinations of system-level inputs and outputs (port events). The slight distinction between ports and port devices is sometimes helpful to testers. Technically, a port is the point at which an I/O device is attached to a system, as in serial and parallel ports, network ports, and telephone ports. Physical actions (keystrokes and light emissions from a screen) occur on port devices, and these are translated from physical to logical (or logical to physical) forms. In the absence of actual port devices, much of system testing can be accomplished by "moving the port boundary inward" to the logical instances of port events. From now on, we will just use the term "port" to refer to port devices. The ports in the SATM system include the digit and cancel keys, the function keys, the display screen, the deposit and withdrawal doors, the card and receipt slots, and several less obvious devices, such as the rollers that move cards and deposit envelopes into the machine, the cash dispenser, the receipt printer, and so on. (See Figure 14.1 for our working example.)

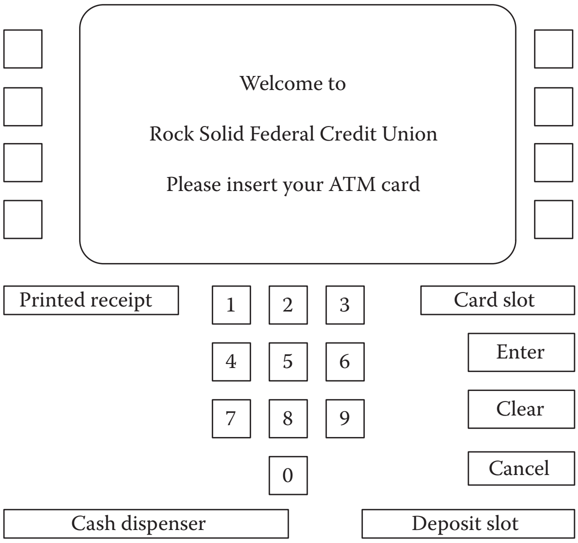
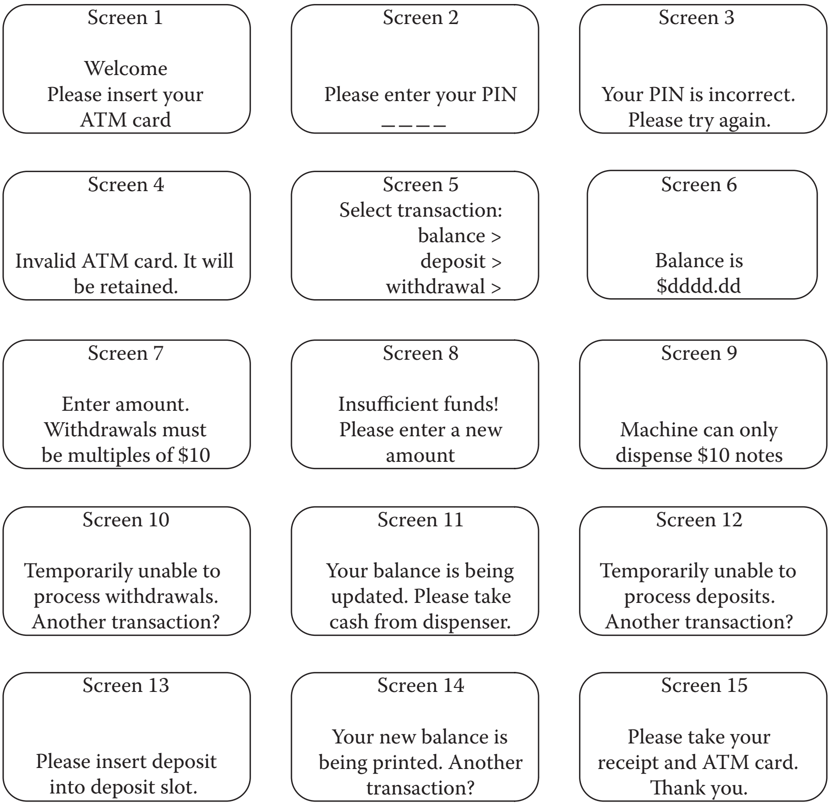


Figure 14.1 The Simple ATM (SATM) terminal.



Screen 1
Welcome
Please insert your
ATM card

Screen 2
Please enter your PIN
_ _ _ _

Screen 3
Your PIN is incorrect.
Please try again.

Screen 4
Invalid ATM card. It will
be retained.

Screen 5
Select transaction:
balance >
deposit >
withdrawal >

Screen 6
Balance is
$dddd.dd

Screen 7
Enter amount.
Withdrawals must
be multiples of $10

Screen 8
Insufficient funds!
Please enter a new
amount

Screen 9
Machine can only
dispense $10 notes

Screen 10
Temporarily unable to
process withdrawals.
Another transaction?

Screen 11
Your balance is being
updated. Please take
cash from dispenser.

Screen 12
Temporarily unable to
process deposits.
Another transaction?

Screen 13
Please insert deposit
into deposit slot.

Screen 14
Your new balance is
being printed. Another
transaction?

Screen 15
Please take your
receipt and ATM card.
Thank you.

Figure 14.2 SATM screens.

Thinking about the ports helps the tester define both the input space that specification-based system testing needs; similarly, the output devices provide output-based test information. For example, we would like to have enough threads to generate all 15 SATM screens in Figure 14.2.

14.2.4 Events

Events are somewhat schizophrenic: they have some characteristics of data and some of actions. An event is a system-level input (or output) that occurs on a port device. Similar to data, events can be inputs to or outputs of actions. Events can be discrete (such as SATM keystrokes) or they can be continuous (such as temperature, altitude, or pressure). Discrete events necessarily have a time duration, and this can be a critical factor in real-time systems. We might picture input events as destructive read-out data, but it is a stretch to imagine output events as destructive write operations.

Events are like actions in the sense that they are the translation point between real-world physical events and internal logical manifestations of these. Port input events are physical-to-logical translations, and, symmetrically, port output events are logical-to-physical translations. System testers should focus on the physical side of events, not the logical side (the focus of integration testers). Situations occur where the context of present data values changes the logical meaning of physical events. In the SATM system, for example, the port input event of depressing button B1 means “balance” when screen 5 is displayed, “checking” when screen 6 is displayed, and “yes” when screens 10, 11, and 14 are displayed. We refer to these situations as “context-sensitive port events,” and we would expect to test such events in each context.

14.2.5 Threads

Unfortunately for testers, threads are the least frequently used of the five fundamental constructs. Because we test threads, it usually falls to the tester to find them in the interactions among the data, events, and actions. About the only place that threads appear per se in a requirements specification is when rapid prototyping is used in conjunction with a scenario recorder. It is easy to find threads in control models, as we will soon see. The problem with this is that control models are just that—they are models, not the reality of a system.

14.2.6 Relationships among Basis Concepts

Figure 14.3 is an E/R model of our basis concepts. Notice that all relationships are many-to-many: Data and Events are inputs to or outputs of the Action entity. The same event can occur on several ports, and typically many events occur on a single port. Finally, an action can occur in several threads, and a thread is composed of several actions. This diagram demonstrates some of the difficulty of system testing. Testers must use events and threads to ensure that all the many-to-many relationships among the five basis concepts are correct.

14.3 Model-Based Threads

In this section, we will use the SATM system (defined in Chapter 2) to illustrate how threads can be identified from models. Figure 14.2 shows the 15 screens needed for SATM. (This is really a bare bones, economy ATM system!)

Finite state machine models of the SATM system are the best place to look for system testing threads. We will start with a hierarchy of state machines; the upper level is shown in Figure 14.4. At this level, states correspond to stages of processing, and transitions are caused by abstract logical (instead of port) events. The card entry “state,” for example, would be decomposed into lower levels that deal with details such as jammed cards, cards that are upside down, stuck card rollers,

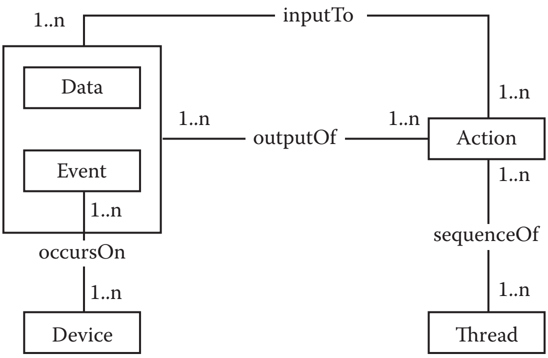


Figure 14.3 E/R model of basis concepts.

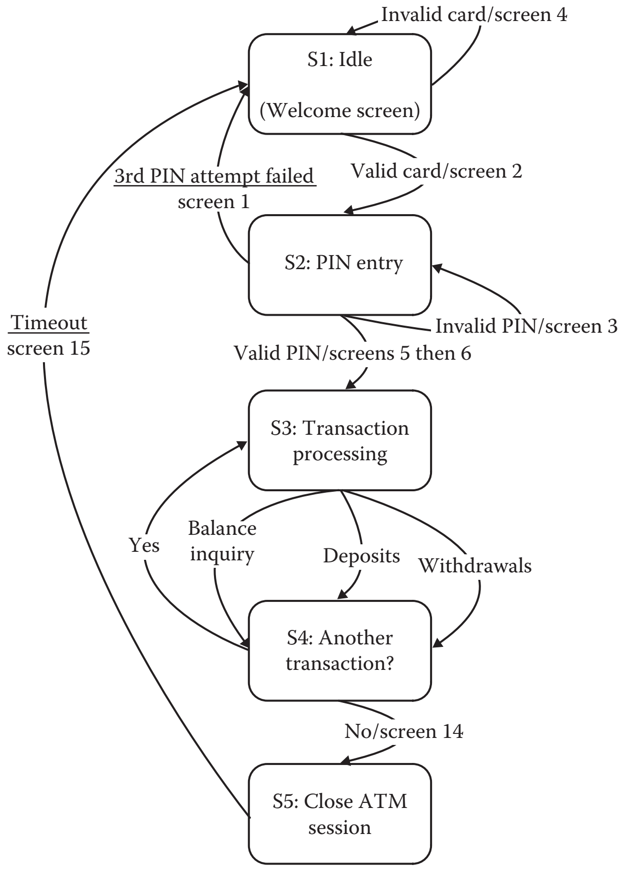

Invalid card/screen 4
S1: Idle
(Welcome screen)
3rd PIN attempt failed
screen 1
Valid card/screen 2
S2: PIN entry
Invalid PIN/screen 3
Timeout
screen 15
Valid PIN/screens 5 then 6
S3: Transaction processing
Yes
Balance inquiry
Deposits
Withdrawals
S4: Another transaction?
No/screen 14
S5: Close ATM session

Figure 14.4 Uppermost level SATM finite state machine.

and checking the card against the list of cards for which service is offered. Once the details of a macro-state are tested, we use a simple thread to get to the next macro-state.

The PIN entry state S2 is decomposed into the more detailed view in Figure 14.5. The adjacent states are shown because they are sources and destinations of transitions from the PIN entry state at the upper level. (This approach to decomposition is reminiscent of the old data flow diagramming idea of balanced decomposition.) At the S2 decomposition, we focus on the PIN retry mechanism; all of the output events are true port events, but the input events are still logical events.

The transaction processing state S3 is decomposed into a more detailed view in Figure 14.6. In that finite state machine, we still have abstract input events, but the output events are actual port events. State 3.1 requires added information. Two steps are combined into this state: choice of the account type and selection of the transaction type. The little "<" and ">" symbols are supposed to point to the function buttons adjacent to the screen, as shown in Figure 14.1. As a side note, if this were split into two states, the system would have to "remember" the account type choice in the first state. However, there can be no memory in a finite state machine, hence the combined state. Once again, we have abstract input events and true port output events.

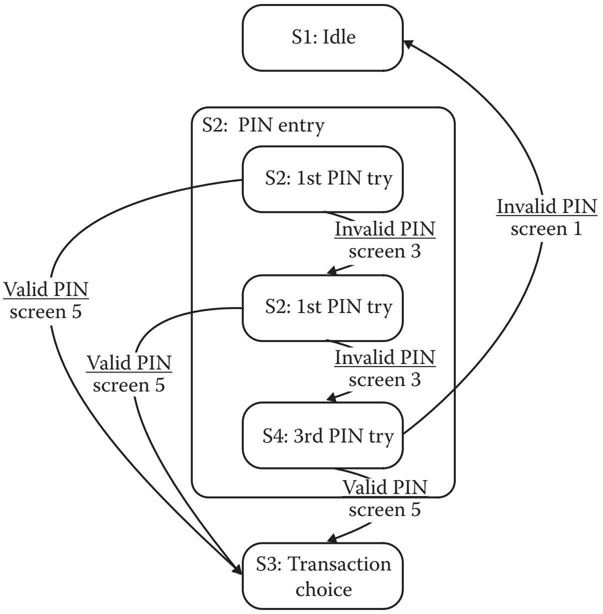

S1: Idle
S2: PIN entry
S2: 1st PIN try
Invalid PIN
screen 3
S2: 1st PIN try
Invalid PIN
screen 3
S4: 3rd PIN try
Valid PIN
screen 5
S3: Transaction choice
Valid PIN
screen 5
Valid PIN
screen 5
Invalid PIN
screen 1

Figure 14.5 Decomposition of PIN entry state.

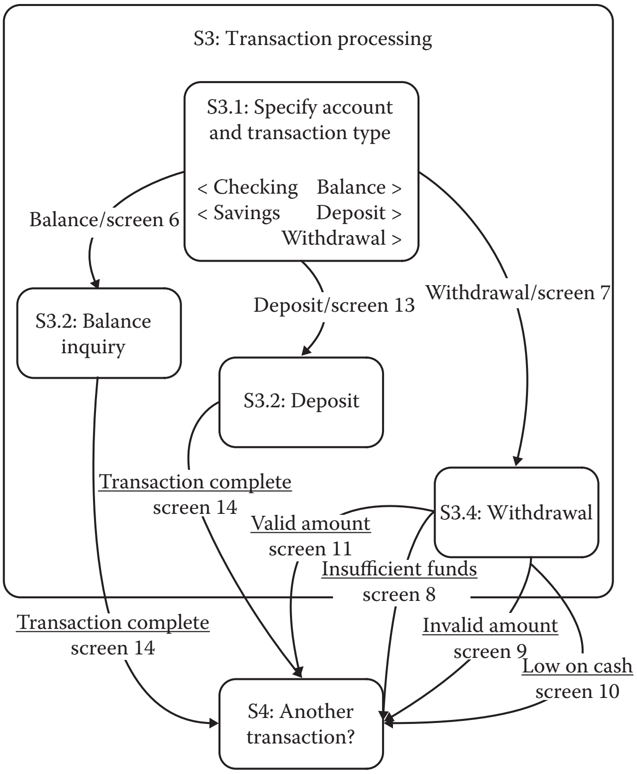

S3: Transaction processing
S3.1: Specify account and transaction type
< Checking
< Savings
Balance >
Deposit >
Withdrawal >
Balance/screen 6
S3.2: Balance inquiry
Deposit/screen 13
S3.2: Deposit
Withdrawal/screen 7
S3.4: Withdrawal
Transaction complete
screen 14
Valid amount
screen 11
Insufficient funds
screen 8
Transaction complete
screen 14
Invalid amount
screen 9
Low on cash
screen 10
S4: Another transaction?

Figure 14.6 Decomposition of transaction processing state.

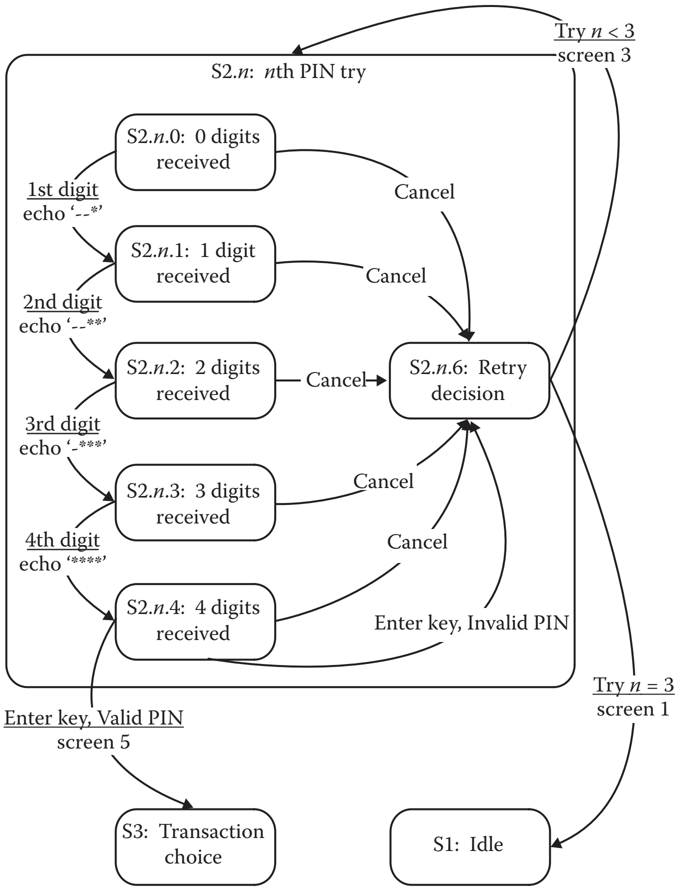

Try n < 3
screen 3
S2.n: nth PIN try
S2.n.0: 0 digits received
1st digit
echo '--*'
Cancel
S2.n.1: 1 digit received
Cancel
2nd digit
echo '--**'
S2.n.2: 2 digits received
Cancel
S2.n.6: Retry decision
3rd digit
echo '-***'
S2.n.3: 3 digits received
Cancel
Cancel
4th digit
echo '****'
S2.n.4: 4 digits received
Enter key, Invalid PIN
Try n = 3
screen 1
Enter key, Valid PIN
screen 5
S3: Transaction choice
S1: Idle

Figure 14.7 Decomposition of PIN try states.

Our final state decomposition is applied to see the details of PIN entry tries S2.1, S2.2, and S2.3 (see Figure 14.7). Each PIN try is identical, so the lower-level states are numbered S2.*n*, where *n* signifies the PIN try. We almost have true input events. If we knew that the expected PIN was "2468" and if we replaced the digit entries, for example, "first digit," with "2," then we would finally have true port input events. A few abstract inputs remain—those referring to valid and invalid PINs and conditions on the number of tries.

It is good form to reach a state machine in which transitions are caused by actual port input events, and the actions on transitions are port output events. If we have such a finite state machine, generating system test cases for these threads is a mechanical process—simply follow a path of transitions and note the port inputs and outputs as they occur along the path. Table 14.1 traces one such path through the PIN try finite state machine in Figure 14.7. This path corresponds to a thread in which a PIN is correctly entered on the first try. To make the test case explicit, we assume a precondition that the expected PIN is "2468." The event in parentheses in the last row of Table 14.1 is the logical event that "bumps up" to the parent state machine and causes a transition there to the Await Transaction Choice state.

Table 14.1 Port Event Sequence for Correct PIN on First Try

Port Input Event	*Port Output Event*
	Screen 2 displayed with '- - - -'
2 Pressed	
	Screen 2 displayed with '- - - *'
4 Pressed	
	Screen 2 displayed with '- - * *'
6 Pressed	
	Screen 2 displayed with '- * * *'
8 Pressed	
	Screen 2 displayed with '* * * *'
(Valid PIN)	Screen 5 displayed

The most common products for model-based testing (Jorgensen, 2009) start with a finite state machine description of the system to be tested and then generate all paths through the graph. If there are loops, these are (or should be) replaced by two paths, as we did at the program graph level in Chapter 8. Given such a path, the port inputs that cause transitions are events in a system test case; similarly for port outputs that occur as actions on transitions.

Here is a hard lesson from industrial experience. A telephone switching system laboratory tried defining a small telephone system with finite state machines. The system, a Private Automatic Branch Exchange (PABX), was chosen because, as switching systems go, it is quite simple. There was a grizzled veteran system tester, Casimir, assigned to help with the development of the model. He was well named. According to Wikipedia, his name means "someone who destroys opponent's prestige/glory during battle" (http://en.wikipedia.org/wiki/Casimir). Throughout the process, Casimir was very suspicious, even untrusting. The team reassured him that, once the project was finished, a tool would generate literally several thousand system test cases. Even better, this provided a mechanism to trace system testing directly back to the requirements specification model. The actual finite state machine had more than 200 states, and the tool generated more than 3000 test cases. Finally, Casimir was impressed, until one day when he discovered an automatically generated test case that was logically impossible. On further, very detailed analysis, the invalid test case was derived from a pair of states that had a subtle dependency (and finite state machines must have independent states). Out of 200-plus states, recognizing such dependencies is extremely difficult. The team explained to Casimir that the tool could analyze any thread that traversed the pair of dependent states, thereby identifying any other impossible threads. This technical triumph was short-lived, however, when Casimir asked if the tool could identify any other pairs of dependent states. No tool can do this because this would be equivalent to the famous Halting Problem. The lesson: generating threads from finite state machines is attractive, and can be quite effective; however, care must be taken to avoid both memory and dependence issues.

14.4 Use Case–Based Threads

Use Cases are a central part of the Unified Modeling Language (UML). Their main advantage is that they are easily understood by both customers/users and developers. They capture the *does view* that emphasizes behavior, rather than the *is view* that emphasizes structure. Customers and testers both tend to naturally think of a system in terms of the *does view*, so use cases are a natural choice.

14.4.1 Levels of Use Cases

One author (Larman, 2001) defines a hierarchy of use cases in which each level adds information to the predecessor level. Larman names these levels as follows:

- High level (very similar to an agile user story)
- Essential
- Expanded essential
- Real

The information content of these variations is shown in Venn diagram form in Figure 14.8.

Tables 14.2 through 14.4 show the gradual increase in Larman's use case hierarchy for the example in Table 14.1. High-level use cases are at the level of the user stories used in agile development. A set of high-level use cases gives a quick overview of the *does view* of a system. Essential use cases add the sequence of port input and output events. At this stage, the port boundary begins to become clear to both the customer/user and the developer.

Expanded essential use cases add pre- and postconditions. We shall see that these are key to linking use cases when they are expressed as system test cases.

Real use cases are at the actual system test case level. Abstract names for port events, such as "invalid PIN," are replaced by an actual invalid PIN character string. This presumes that some

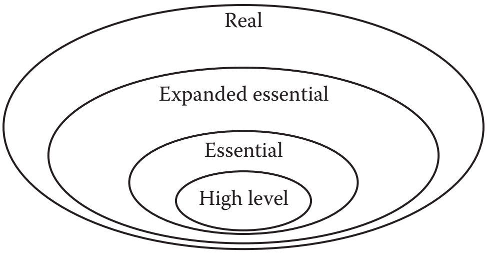


Figure 14.8 Larman's levels of use cases.

Table 14.2 High-Level Use Case for Correct PIN on First Try

Use case name	Correct PIN entry on first try
Use case ID	HLUC-1
Description	A customer enters the PIN number correctly on the first attempt.

Table 14.3 Essential Use Case for Correct PIN on First Try

Use case name	Correct PIN entry on first try
Use case ID	EUC-1
Description	A customer enters the PIN number correctly on the first attempt.
Event Sequence	
Input events	*Output events*
	1. Screen 2 shows '- - - - '
2. Customer touches 1st digit	
	3. Screen 2 shows '- - - * '
4. Customer touches 2nd digit	
	5. Screen 2 shows '- - * * '
6. Customer touches 3rd digit	
	7. Screen 2 shows '- * * * '
8. Customer touches 4th digit	
	9. Screen 2 shows '* * * * '
10. Customer touches Enter	
	11. Screen 5 is displayed

form of testing database has been assembled. In our SATM system, this would likely include several accounts with associated PINs and account balances (Table 14.5).

14.4.2 An Industrial Test Execution System

This section describes a system for automatic test execution that I was responsible for in the early 1980s. Since it was intended for executing regression test cases (a very boring manual assignment), it was named the Automatic Regression Testing System (ARTS). This is as close as I ever came to the art world. The ARTS system had a human readable system test case language that was interpretively executed on a personal computer. In the ARTS language, there were two verbs: CAUSE would cause a port input event to occur, and VERIFY would observe a port output event. In addition, a tester could refer to a limited number of devices and to a limited number of input events associated with those devices. Here is a small paraphrased excerpt of a typical ARTS test case.

```
CAUSE Go-Offhook On Line 4
VERIFY Dialtone On Line 4
CAUSE TouchDigit '3' On Line 4
VERIFY NoDialtone On Line 4
```

Table 14.4 Expanded Essential Use Case for Correct PIN on First Try

Use case name	Correct PIN entry on first try
Use case ID	EEUC-1
Description	A customer enters the PIN number correctly on the first attempt.
Preconditions	1. The expected PIN is known
	2. Screen 2 is displayed
Event Sequence	
Input events	*Output events*
	1. Screen 2 shows '- - - - '
2. Customer touches 1st digit	
	3. Screen 2 shows '- - - * '
4. Customer touches 2nd digit	
	5. Screen 2 shows '- - * * '
6. Customer touches 3rd digit	
	7. Screen 2 shows '- * * * '
8. Customer touches 4th digit	
	9. Screen 2 shows '* * * * '
10. Customer touches Enter	
	11. Screen 5 is displayed
Cross reference to functions	
Postconditions	Select Transaction screen is active

The physical connection to a telephone prototype required a harness that connected the personal computer with actual prototype ports. The test case language consisted of the CAUSE and VERIFY verbs, names for port input and output events, and names for available devices that were connected to the harness. On the input side, the harness accomplished a logical-to-physical transformation, with the symmetric physical-to-logical transformation on the output side. The basic architecture is shown in Figure 14.9.

We learned a lesson in human factors engineering. The test case language was actually free form, and the interpreter eliminated noise words. The freedom to add noise words was intended to give test case designers a place to put additional notes that would not be executed, but would be kept in the test execution report. The result was test cases like this (so much for test designer freedom):

> *As long as it is not raining, see if you can CAUSE a Go-Offhook event right away On Line 4, and then, see if you can VERIFY that some variation of Dialtone happened to occur*

Table 14.5 Real Use Case for Correct PIN on First Try

Use case name	Correct PIN entry on first try
Use case ID	RUC-1
Description	A customer enters the PIN number correctly on the first attempt.
Preconditions	1. The expected PIN is "2468"
	2. Screen 2 is displayed
Event Sequence	
Input events	*Output events*
	1. Screen 2 shows '- - - - '
2. Customer touches digit 2	
	3. Screen 2 shows '- - - * '
4. Customer touches digit 4	
	5. Screen 2 shows '- - * * '
6. Customer touches digit 6	
	7. Screen 2 shows '- * * * '
8. Customer touches digit 8	
	9. Screen 2 shows '* * * * '
10. Customer touches Enter	
	11. Screen 5 is displayed
Cross reference to functions	(normally done at this point)
Postconditions	Select Transaction is active

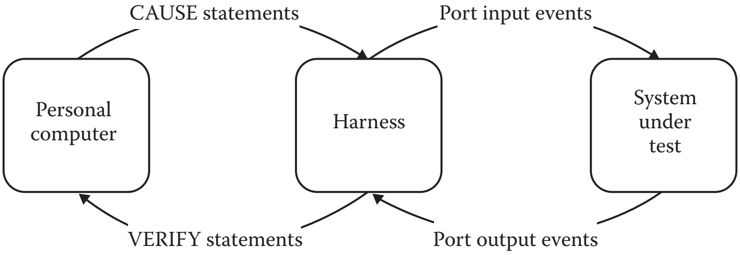

CAUSE statements
Port input events
Personal computer
Harness
System under test
VERIFY statements
Port output events

Figure 14.9 Automated test execution system architecture.

> *On Line 4. Then, if you are in a good mood, why not CAUSE a TouchDigit '3' action On Line 4. Finally, (at last!), see if you can VERIFY that NoDialtone is present On Line 4.*

In retrospect, the ARTS system predated the advent of use cases. Notice how the event sequence portion of a real use case is dangerously close to an ARTS test case. I learned later that the system evolved into a commercial product that had a 15-year lifetime.

14.4.3 System-Level Test Cases

A system-level test case, whether executed manually or automatically, has essentially the same information as a real use case (Table 14.6).

Table 14.6 System Test Case for Correct PIN on First Try

Test case name	Correct PIN entry on first try
Test case ID	TC-1
Description	A customer enters the PIN number correctly on the first attempt.
Preconditions needed to run this test case	1. The expected PIN is "2468"
	2. Screen 2 is displayed
Event Sequence	
Input events (performed by tester)	*Output events (observed by system tester)*
	1. Screen 2 shows '- - - - '
2. Touch digit 2	
	3. Screen 2 shows '- - - * '
4. Customer touches digit 4	
	5. Screen 2 shows '- - * * '
6. Customer touches digit 6	
	7. Screen 2 shows '- * * * '
8. Customer touches digit 8	
	9. Screen 2 shows '- - - * '
10. Customer touches Enter	
	11. Screen 5 is displayed
Cross reference to functions	
Postconditions	Select Transaction is active
Test execution result?	Pass/Fail
Test run by	<tester's name> date

14.4.4 Converting Use Cases to Event-Driven Petri Nets

Event-Driven Petri Nets (EDPNs) were defined in Chapter 4. They were originally developed for use in telephone switching systems. As the name implies, they are appropriate for any event-driven system, particularly those characterized by context-sensitive port input events. In an EDPN drawing, port events are shown as triangles, data places are circles, transitions are narrow rectangles, and the input and output connections are arrows. In an attempt at human factors design,

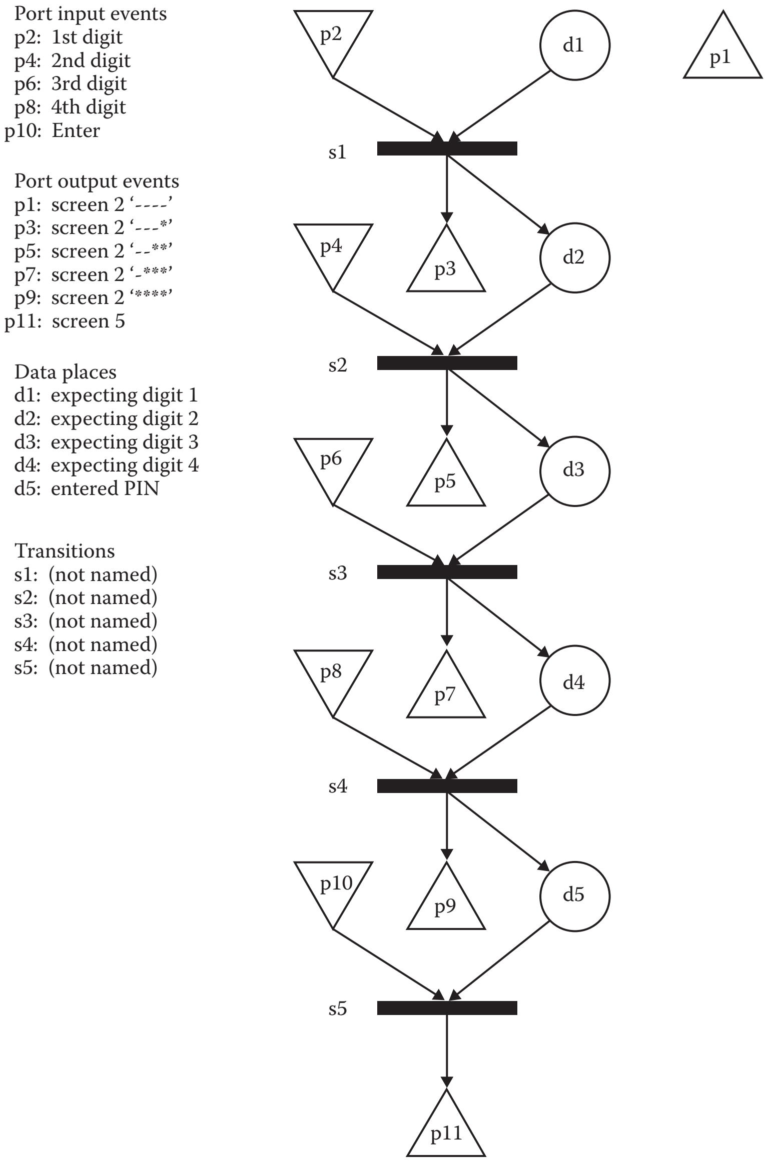

Port input events
p2: 1st digit
p4: 2nd digit
p6: 3rd digit
p8: 4th digit
p10: Enter
Port output events
p1: screen 2 '----'
p3: screen 2 '---*'
p5: screen 2 '--**'
p7: screen 2 '-***'
p9: screen 2 '****'
p11: screen 5
Data places
d1: expecting digit 1
d2: expecting digit 2
d3: expecting digit 3
d4: expecting digit 4
d5: entered PIN
Transitions
s1: (not named)
s2: (not named)
s3: (not named)
s4: (not named)
s5: (not named)
p2
d1
p1
s1
p4
p3
d2
s2
p6
p5
d3
s3
p8
p7
d4
s4
p10
p9
d5
s5
p11

Figure 14.10 Event-Driven Petri Net for correct PIN on first try.

EDPN diagrams show input port events as a downward pointing triangle as if they were a funnel. Similarly, output port events are upward pointing, as if they were megaphones. Figure 14.10 is the EDPN for our continuing example, Correct PIN on First Try.

Automatic derivation of an EDPN from a use case is only partly successful—port input events can be derived from the input portion of the event sequence, similarly for port output events. Also, the interleaved order of input and output events can be preserved. Finally, the pre- and postconditions are mapped to data places. There are a few problems, however.

1. The most obvious is the port output event p1 that refers to screen 2 being displayed with four blank positions for the PIN to be entered. It is an orphan, in the sense that it is not created by a transition.
2. Transitions are not named. If a person were to develop the EDPN, there would likely be descriptive names for the transitions, for example, s1: accept first digit.
3. An immediate cause-and-effect connection is presumed. This would fail if two out-of-sequence input events were required to produce an output event.
4. There is no provision for intermediate data that may be produced.
5. The data places d1–d5 do not appear in the use case. (They could be derived from the finite state machine, however.)

One answer to this is to follow the lead of formal systems and define the information content of a "well-formed use case." At a minimum, a well-formed use case should conform to these requirements.

1. The event sequence cannot begin with an output event. This could just be considered as a precondition.
2. The event sequence cannot end with an input event. This could just be considered as a postcondition.
3. Preconditions must be both necessary and sufficient to the use case. There are no superfluous preconditions, and every precondition must be used or needed by the use case. Similarly for postconditions.
4. There must be at least one precondition and at least one postcondition.

The value in deriving EDPNs from use cases is that, because they are special cases of Petri Nets, they inherit a wealth of analytical possibilities. Here are some analyses that are easy with Petri Nets, and all but impossible with use cases:

1. Interactions among use cases, such as one use case being a prerequisite for another
2. Use cases that are in conflict with others
3. Context-sensitive input events
4. Inverse use cases, where one "undoes" the other

14.4.5 Converting Finite State Machines to Event-Driven Petri Nets

Mathematically speaking, finite state machines are a special case of ordinary Petri Nets in which every Petri Net transition has one input place and one output place. Since EDPNs are an extension of ordinary Petri Nets, the conversion of finite state machines to EDPNs is guaranteed. Figure 14.11 shows a portion of the finite state machine in Figure 14.7 converted to an EDPN.

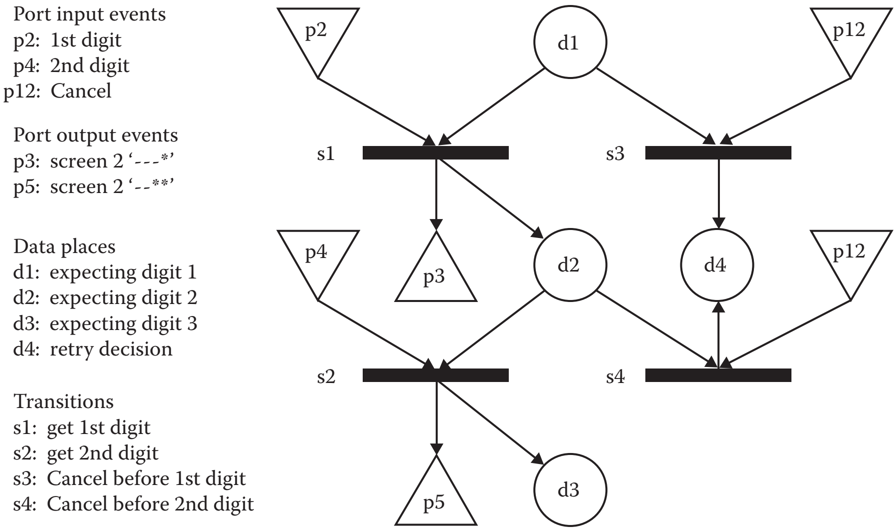

Port input events
p2: 1st digit
p4: 2nd digit
p12: Cancel
Port output events
p3: screen 2 '---*'
p5: screen 2 '--**'
Data places
d1: expecting digit 1
d2: expecting digit 2
d3: expecting digit 3
d4: retry decision
Transitions
s1: get 1st digit
s2: get 2nd digit
s3: Cancel before 1st digit
s4: Cancel before 2nd digit
p2
d1
p12
s1
s3
p4
p3
d2
d4
p12
s2
s4
p5
d3

Figure 14.11 Event-Driven Petri Net from part of finite state machine in Figure 14.12.

In Figure 14.11, input events p2 and p12 can both occur when the ATM is awaiting the first PIN digit. Similarly, input events p4 and p12 can both occur when the ATM is awaiting the second PIN digit. Close examination shows three distinct paths. There are two main ways to describe these paths—as a sequence of port input events, or as a sequence of EDPN transitions. Using the latter, the three paths in Figure 14.11 are <s1, s2>, <s1, s4>, and <s3>. There is an interesting connection between EDPNs and obscure database terminology. The *intention* of a database is the underlying data model. Different populations of the intention are known as *extensions* of the database. The intention of a given database is unique, but there can be myriad possible extensions. The same is true for unmarked versus marked EDPNs—an unmarked EDPN can have many possible marking sequence executions.

14.4.6 Which View Best Serves System Testing?

Of the three views in this section, use cases are the best for communication between customers/users and developers; however, they do not support much in the way of analysis. Finite state machines are in common use, but composing finite state machines inevitably leads to the well-known "finite state machine explosion." Finite state machine–based support tools are available, but the explosion part is problematic. Both of these notations can be converted to EDPNs, although some information must be added to EDPNs derived from use cases. The big advantage of EDPNs is that they are easily composed with other EDPNs, and the intension/extension relationship between an unmarked EDPN and various markings (execution sequences) makes them the preferred choice for system testing. We did not discuss deriving system test cases from an EDPN, but the process is obvious.

14.5 Long versus Short Use Cases

There is an element of foreshadowing in the preceding material. Early on, we spoke of various thread candidates. In that discussion, we saw a range of very short to very long threads. Each of these choices translates directly to our three models, use cases, finite state machines, and EDPNs.

In the use case domain, the usual view is that a use case is a full, end-to-end, transaction. For the SATM system, full use cases would start and end with screen 1, the Welcome screen. In between, some path would occur, either as a specific use case, a path in the finite state machine, or as a marking in the full EDPN. The problem with this is that there is a large number of paths in either model, and a large number of individual use cases. The end-to-end use case is a "long use case." The use cases in Chapter 22 (Software Technical Reviews) are short use cases. As a quick example of a long use case, consider a story with the following sequence:

> *A customer enters a valid card, followed by a valid PIN entry on the first try. The customer selects the Withdraw option, and enters $20 for the withdrawal amount. The SATM system dispenses two $10 notes and offers the customer a chance to request another transaction. The customer declines, the SATM system updates the customer account, returns the customer's ATM card, prints a transaction receipt, and returns to the Welcome screen.*

Here we suggest "short use cases," which begin with a port input event and end with a port output event. Short use cases must be at the Expanded Essential Use Case level, so the pre- and postconditions are known. We can then develop sequences of short use cases with the connections based on the pre- and postconditions. Short use case B can follow short use case A if the postconditions of A are consistent with the preconditions of B. The long use case above might be expressed in terms of four short use cases:

1. Valid card
2. Correct PIN on first try
3. Withdrawal of $20
4. Select no more transitions

Table 14.7 Short Use Cases for Successful SATM Transactions

Short Use Case	*Description*
SUC1	Valid ATM card swipe
SUC2	Invalid ATM card swipe
SUC3	Correct PIN attempt
SUC4	Failed PIN attempt
SUC5	Choose Balance
SUC6	Choose Deposit
SUC7	Choose Withdrawal: valid withdrawal amount
SUC8	Choose Withdrawal: amount not a multiple of $20
SUC9	Choose Withdrawal: amount greater than account balance
SUC10	Choose Withdrawal: amount greater than daily limit
SUC11	Choose no other transaction
SUC12	Choose another transaction

The motivation for short use cases is that there are 1909 possible paths through the SATM finite state machine in Figure 14.5 considering all four state decompositions. The great majority of these are due to failed PIN entry attempts (six ways to fail, one way to be successful). Table 14.7 lists a useful set of short use cases for successful SATM transactions. If we added short use cases for all the ways that PIN entry can fail, we would have good coverage of the SATM system.

Figure 14.12 shows the short use cases from Table 14.7 linked with respect to a slightly different finite state machine model of the SATM system.

What about the ways PIN entry can fail? One could argue that this is really a unit-level question; hence, we do not need short use cases for these possibilities. On the other hand, there are only 13 transitions in the detailed view of PIN entry in Figure 14.7, and we have port inputs and outputs for the digit entry transitions. (The transitions for the five possible points of cancellation and the invalid four-digit PIN all have an intermediate state to simplify the figure.) Table 14.8 lists the short use cases for complete PIN entry coverage.

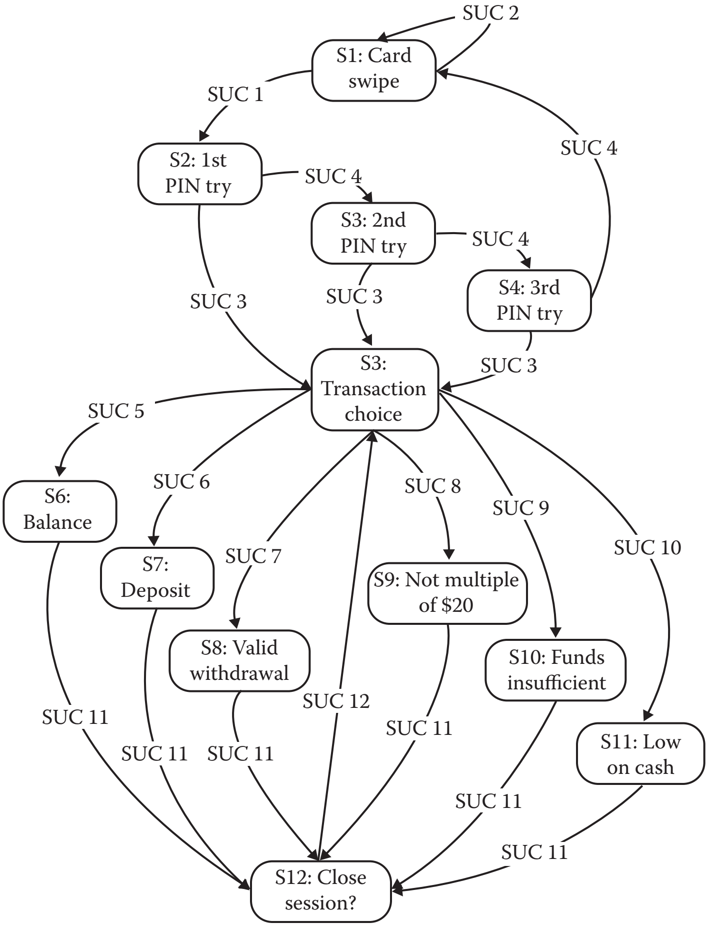

SUC 2
S1: Card swipe
SUC 1
SUC 4
S2: 1st PIN try
SUC 4
S3: 2nd PIN try
SUC 4
S4: 3rd PIN try
SUC 3
SUC 3
SUC 3
S3: Transaction choice
SUC 5
S6: Balance
SUC 6
S7: Deposit
SUC 7
S8: Valid withdrawal
SUC 8
S9: Not multiple of $20
SUC 9
S10: Funds insufficient
SUC 10
S11: Low on cash
SUC 12
SUC 11
SUC 11
SUC 11
SUC 11
SUC 11
SUC 11
S12: Close session?

Figure 14.12 SATM finite state machine with short use cases causing transitions.

Table 14.8 Short Use Cases for Failed PIN Entry Attempts

Short Use Case	*Description*
SUC13	Digit 1 entered
SUC14	Digit 2 entered
SUC15	Digit 3 entered
SUC16	Digit 4 entered
SUC17	Enter with valid PIN
SUC18	Cancel before digit 1
SUC19	Cancel after digit 1
SUC20	Cancel after digit 2
SUC21	Cancel after digit 3
SUC22	Cancel after digit 4
SUC23	Enter with invalid PIN
SUC24	Next PIN try
SUC25	Last PIN try

Now we see the advantage of short use cases—1909 long use cases are covered by the 25 short use cases. The advantage of model-based testing becomes yet clearer with this compression. Note the similarity to node and edge test coverage at the unit level that we saw in Chapter 8.

14.6 How Many Use Cases?

When a project is driven by use cases, there is the inevitable question as to how many use cases are needed. Use case–driven development is inherently a bottom–up process. In the agile world, the answer is easy—the customer/user decides how many use cases are needed. But what happens in a non-agile project? Use case–driven development is still (or can be) an attractive option. In this section, we examine four strategies to help decide how many bottom–up use cases are needed. Each strategy employs an incidence matrix (see Chapter 4). We could have a fifth strategy if bottom–up use case development is done in conjunction with a gradually developed model, as we just saw in Section 14.5.

14.6.1 Incidence with Input Events

As use cases are identified jointly between the customer/user and developers, both parties gradually identify port-level input events. This very likely is an iterative process, in which use cases provoke the recognition of port input events, and they, it turn, suggest additional use cases. These are kept in an incidence showing which use cases require which port input events. As the process continues, both parties reach a point where the existing set of input events is sufficient for any new use case. Once this point is reached, it is clear that a minimal set of use cases covers all the

Table 14.9 SATM Port Input Events

Port Input Event	*Description*
e1	Valid ATM card swipe
e2	Invalid ATM card swipe
e3	Correct PIN attempt
e4	Touch Enter
e5	Failed PIN
e6	Touch Cancel
e7	Touch button corresponding to Checking
e8	Touch button corresponding to Savings
e9	Choose Balance
e10	Choose Deposit
e11	Enter deposit amount
e12	Choose Withdrawal
e13	Enter withdrawal amount
e14	Valid withdrawal amount
e15	Withdrawal amount not a multiple of $20
e16	Withdrawal amount greater than account balance
e17	Withdrawal amount greater than cash in SATM
e18	Touch button corresponding to Yes
e19	Touch button corresponding to No

port level input events. Table 14.9 lists (most of) the port input events for the SATM system, and Table 14.10 shows their incidence with the first 12 short use cases. Exercise 5 asks you to develop a similar table for the PIN entry use cases.

Both Tables 14.9 and 14.10 should be understood as the result of an iterative process, which is inevitable in a bottom–up approach. If port inputs are identified that are not used anywhere, we know we need at least one more short use case. Similarly, if we have a short use case that does not involve any of the existing port input events, we know we need at least one more event.

14.6.2 Incidence with Output Events

The matrix showing the incidence of short use cases with port output events is developed in the same iterative way as that for input events. Table 14.11 lists the port output events for the SATM system. As you develop this incidence matrix (see Exercise 5), you should note if any screen is never used in the finite state machines in Figures 14.4 through 14.7. This is also revisited in Chapter 22 on Software Technical Reviews.

Table 14.10 Short Use Case Incidence Matrix with Port Input Events

SUC	*Port Input Events*																		
	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*
1	X																		
2		X																	
3			X	X															
4				X	X	X													
5							X	X	X										
6							X	X		X	X								
7							X	X				X	X	X					
8							X	X				X	X		X				
9							X	X				X	X			X			
10							X	X				X	X				X		
11																			X
12																		X	

Table 14.11 SATM Port Output Events

Port Input Event	*Description*
Screen 1	Welcome. Please insert your ATM card.
Screen 2	Please enter your PIN.
Screen 3	Your PIN is incorrect. Please try again.
Screen 4	Invalid ATM card. It will be retained.
Screen 5	Select transaction: balance, deposit, or withdrawal.
Screen 6	Your account balance is $___.__.
Screen 7	Enter withdrawal amount. Must be a multiple of $10.
Screen 8	Insufficient funds. Please enter a new withdrawal amount.
Screen 9	Sorry. Machine can only dispense $10 notes.
Screen 10	Temporarily unable to process withdrawals. Another transaction?
Screen 11	Your balance is updated. Please remove cash from dispenser.
Screen 12	Temporarily unable to process deposits. Another transaction?
Screen 13	Please insert deposit envelope into slot.
Screen 14	Your new balance is being printed. Another transaction?
Screen 15	Please take your receipt and ATM card.

14.6.3 Incidence with All Port Events

In practice, the incidence approach needs to be done for all port-level events. This is just a combination of the discussions in Sections 16.2.1 and 16.2.2. One advantage of the full port event incidence matrix is that it can be reordered in useful ways. For example, the short use cases could be placed together, or possibly listed in a sensible transaction-based order. The port events can also be permuted into cohesive subgroups, for example, input events related to PIN entry, or output events related to deposits.

14.6.4 Incidence with Classes

There is a perennial debate among object-oriented developers as to how to begin—use cases first, or classes first. One of my colleagues (a very classy person) insists on the class-first approach, while others are more comfortable with the use case first view. A good compromise is to develop an incidence matrix showing which classes are needed to support which use cases. Often, it is easier to identify classes for a use case, rather than for a full system. As with the other incidence matrices, this approach provides a good answer to when a sufficient set of classes has been identified.

14.7 Coverage Metrics for System Testing

In Chapter 10, we saw the advantage of combining specification-based and code-based testing techniques, because they are complementary. We are in the same position now with system testing. The model–based approaches of Section 14.3 can be combined with the use case–based approaches of Section 14.4. Further, the incidence matrices of Section 14.6 can be used as a basis for specification-based system test coverage metrics.

14.7.1 Model-Based System Test Coverage

We can use model-based metrics as a cross-check on use case–based threads in much the same way that we used DD-paths at the unit level to identify gaps and redundancies in specification-based test cases. We really have pseudostructural testing (Jorgensen, 1994) because the node and edge coverage metrics are defined in terms of a model of a system, not derived directly from the system implementation. In general, behavioral models are only approximations of a system's reality, which is why we could decompose our models down to several levels of detail. If we made a true code-based model, its size and complexity would make it too cumbersome to use. The big weakness of model-based metrics is that the underlying model may be a poor choice. The three most common behavioral models (decision tables, finite state machines, and Petri nets) are appropriate to transformational, interactive, and concurrent systems, respectively.

Decision tables and finite state machines are good choices for ASF testing. If an ASF is described using a decision table, conditions typically include port input events, and actions are port output events. We can then devise test cases that cover every condition, every action, or, most completely, every rule. For finite state machine models, test cases can cover every state, every transition, or every path.

Thread testing based on decision tables is cumbersome. We might describe threads as sequences of rules from different decision tables, but this becomes very messy to track in terms of coverage. We need finite state machines as a minimum, and if any form of interaction occurs, Petri nets are a better choice. There, we can devise thread tests that cover every place, every transition, and every sequence of transitions.

14.7.2 Specification-Based System Test Coverage

The model-based approaches to thread identification are clearly useful, but what if no behavioral model exists for a system to be tested? The testing craftsperson has two choices: develop a behavioral model or resort to the system-level analogs of specification-based testing. Recall that when specification-based test cases are identified, we use information from the input and output spaces as well as the function itself. We describe system testing threads here in terms of coverage metrics that are derived from three of the basis concepts (events, ports, and data).

14.7.2.1 Event-Based Thread Testing

Consider the space of port input events. Five port input thread coverage metrics are easily defined. Attaining these levels of system test coverage requires a set of threads such that

Port Input 1: each port input event occurs
Port Input 2: common sequences of port input events occur
Port Input 3: each port input event occurs in every "relevant" data context
Port Input 4: for a given context, all "inappropriate" input events occur
Port Input 5: for a given context, all possible input events occur

The Port Input 1 metric is a bare minimum and is inadequate for most systems. Port Input 2 coverage is the most common, and it corresponds to the intuitive view of system testing because it deals with "normal use." It is difficult to quantify, however. What is a common sequence of input events? What is an uncommon one?

The last three metrics are defined in terms of a "context." The best view of a context is that it is a point of event quiescence. In the SATM system, screen displays occur at the points of event quiescence. The Port Input 3 metric deals with context-sensitive port input events. These are physical input events that have logical meanings determined by the context within which they occur. In the SATM system, for example, a keystroke on the B1 function button occurs in five separate contexts (screens displayed) and has three different meanings. The key to this metric is that it is driven by an event in all of its contexts. The Port Input 4 and Port Input 5 metrics are converses: they start with a context and seek a variety of events. The Port Input 4 metric is often used on an informal basis by testers who try to break a system. At a given context, they want to supply unanticipated input events just to see what happens. In the SATM system, for example, what happens if a function button is depressed during the PIN entry stage? The appropriate events are the digit and cancel keystrokes. The inappropriate input events are the keystrokes on the B1, B2, and B3 buttons.

This is partially a specification problem: we are discussing the difference between prescribed behavior (things that should happen) and proscribed behavior (things that should not happen). Most requirements specifications have a hard time only describing prescribed behavior; it is usually testers who find proscribed behavior. The designer who maintains my local ATM system told me that once someone inserted a fish sandwich in the deposit envelope slot. (Apparently they thought it was a waste receptacle.) At any rate, no one at the bank ever anticipated insertion of a fish sandwich as a port input event. The Port Input 4 and Port Input 5 metrics are usually very effective, but they raise one curious difficulty. How does the tester know what the expected response should be to a proscribed input? Are they simply ignored? Should there be an output warning message? Usually, this is left to the tester's intuition. If time permits, this is a powerful point of feedback to requirements specification. It is also a highly desirable focus for either rapid prototyping or executable specifications.

We can also define two coverage metrics based on port output events:

Port Output 1: each port output event occurs
Port Output 2: each port output event occurs for each cause

Port Output 1 coverage is an acceptable minimum. It is particularly effective when a system has a rich variety of output messages for error conditions. (The SATM system does not.) Port Output 2 coverage is a good goal, but it is hard to quantify. For now, note that Port Output 2 coverage refers to threads that interact with respect to a port output event. Usually, a given output event only has a small number of causes. In the SATM system, screen 10 might be displayed for three reasons: the terminal might be out of cash, it may be impossible to make a connection with the central bank to get the account balance, or the withdrawal door might be jammed. In practice, some of the most difficult faults found in field trouble reports are those in which an output occurs for an unsuspected cause. Here is one example: My local ATM system (not the SATM) has a screen that informs me that "Your daily withdrawal limit has been reached." This screen should occur when I attempt to withdraw more than $300 in one day. When I see this screen, I used to assume that my wife has made a major withdrawal (thread interaction), so I request a lesser amount. I found out that the ATM also produces this screen when the amount of cash in the dispenser is low. Instead of providing a lot of cash to the first users, the central bank prefers to provide less cash to more users.

14.7.2.2 Port-Based Thread Testing

Port-based testing is a useful complement to event-based testing. With port-based testing, we ask, for each port, what events can occur at that port. We then seek threads that exercise input ports and output ports with respect to the event lists for each port. (This presumes such event lists have been specified; some requirements specification techniques mandate such lists.) Port-based testing is particularly useful for systems in which the port devices come from external suppliers. The main reason for port-based testing can be seen in the E/R model of the basis constructs (Figure 14.3). The many-to-many relationship between devices and events should be exercised in both directions. Event-based testing covers the one-to-many relationship from events to ports, and, conversely, port-based testing covers the one-to-many relationship from ports to events. The SATM system fails us at this point—no SATM event occurs at more than one port.

14.8 Supplemental Approaches to System Testing

All model-based testing approaches have been open to the criticism that the testing is only as good as the underlying model. There is no escaping this. In response, some authorities recommend various "random" supplements. Two such techniques, mutation testing and fuzzing, are discussed in Chapter 21. In this section, we consider two fallback strategies, each of which has thread execution probability as a starting point. Both operational profiling and risk-based testing are responses to the "squeeze" on available system testing time.

14.8.1 Operational Profiles

In its most general form, Zipf's law holds that 80% of the activities occur in 20% of the space. Activities and space can be interpreted in numerous ways: people with messy desks hardly ever use most of their desktop clutter; programmers seldom use more than 20% of the features of their favorite programming language, and Shakespeare (whose writings contain an enormous vocabulary) uses a small fraction of his vocabulary most of the time. Zipf's law applies to software (and testing) in several ways. The most useful interpretation for testers is that the space consists of all possible threads, and activities are thread executions (or traversals). Thus, for a system with many threads, 80% of the execution traverses only 20% of the threads.

Recall that a failure occurs when a fault is executed. The whole idea of testing is to execute test cases such that, when a failure occurs, the presence of a fault is revealed. We can make an important distinction: the distribution of faults in a system is only indirectly related to the reliability of the system. The simplest view of system reliability is the probability that no failure occurs during a specific time interval. (Notice that no mention is even made of faults, the number of faults, or fault density.) If the only faults are "in the corners" on threads that are seldom traversed, the overall reliability is higher than if the same number of faults were on "high-traffic" threads. The idea of operational profiles is to determine the execution frequencies of various threads and to use this information to select threads for system testing. Particularly when test time is limited (usually), operational profiles maximize the probability of finding faults by inducing failures in the most frequently traversed threads. Here we use our SATM system. In Figure 14.13, the short use case labels on the transitions in Figure 14.12 are replaced by estimated transition probabilities.

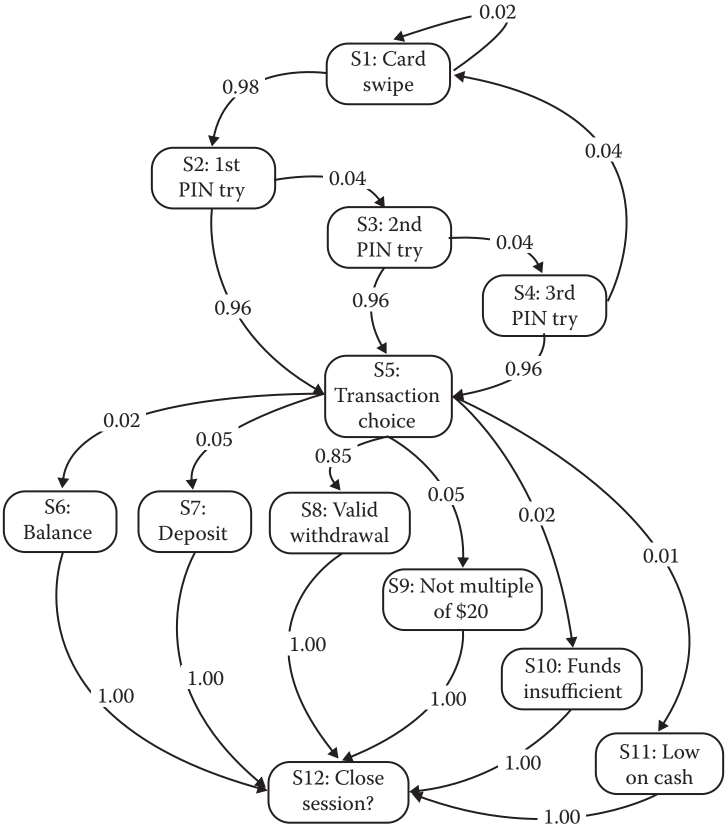

0.02
S1: Card swipe
0.98
0.04
S2: 1st PIN try
0.04
S3: 2nd PIN try
0.04
S4: 3rd PIN try
0.96
0.96
0.96
S5: Transaction choice
0.02
0.05
0.85
0.05
0.02
0.01
S6: Balance
S7: Deposit
S8: Valid withdrawal
S9: Not multiple of $20
S10: Funds insufficient
S11: Low on cash
1.00
1.00
1.00
1.00
1.00
1.00
S12: Close session?

Figure 14.13 Transition probabilities in SATM finite state machine of Figure 14.12.

Finite state machines are the preferred model for identifying thread execution probabilities. The mathematics behind this is that the transition probabilities can be expressed is a "transition matrix" where the element in row *i*, column *j* is the probability of the transition from state *i* to state *j*. Powers of the transition matrix are analogous to the powers of the adjacency matrix when we discussed reachability in Chapter 4. For small systems, it is usually easier to show the transition probabilities in a spreadsheet, as in Table 14.12. Once the thread probabilities are known, they sorted according to execution probability, most to least probable. This is done in Table 14.13.

Just as the quality of model-based testing is limited by the correctness of the underlying model, the analysis of operational profiles is limited by the validity of the transition probability estimates. There are strategies to develop these estimates. One is to use historical data from similar systems. Another is to use customer-supplied estimates. Still another is to use a Delphi approach in which a group of experts give their guesses, and some average is determined. This might be based on convergence of a series of estimates, or possibly by having seven experts, and eliminating the high and low estimates.

Table 14.12 Spreadsheet of SATM Transition Probabilities

	Path	*Transition Probabilities*					*Path Probability*
First try	S1, S2, S5, S6, S12	0.999	0.96	0.02	1	1	0.019181
	S1, S2, S5, S7, S12	0.999	0.96	0.05	1	1	0.047952
	S1, S2, S5, S8, S12	0.999	0.96	0.85	1	1	0.815184
	S1, S2, S5, S9, S12	0.999	0.96	0.05	1	1	0.047952
	S1, S2, S5, S10, S12	0.999	0.96	0.02	1	1	0.019181
	S1, S2, S5, S11, S12	0.999	0.96	0.01	1	1	0.009590
Second try	S1, S2, S3, S5, S6, S12	0.999	0.04	0.96	0.02	1	0.000767
	S1, S2, S3, S5, S7, S12	0.999	0.04	0.96	0.05	1	0.001918
	S1, S2, S3, S5, S8, S12	0.999	0.04	0.96	0.85	1	0.032607
	S1, S2, S3, S5, S9, S12	0.999	0.04	0.96	0.05	1	0.001918
	S1, S2, S3, S5, S10, S12	0.999	0.04	0.96	0.02	1	0.000767
	S1, S2, S3, S5, S11, S12	0.999	0.04	0.96	0.01	1	0.000384
Third try	S1, S2, S3, S4, S5, S6, S12	0.999	0.04	0.04	0.96	0.02	0.000031
	S1, S2, S3, S4, S5, S7, S12	0.999	0.04	0.04	0.96	0.05	0.000077
	S1, S2, S3, S4, S5, S8, S12	0.999	0.04	0.04	0.96	0.85	0.001304
	S1, S2, S3, S4, S5, S9, S12	0.999	0.04	0.04	0.96	0.05	0.000077
	S1, S2, S3, S4, S5, S10, S12	0.999	0.04	0.04	0.96	0.02	0.000031
	S1, S2, S3, S4, S5, S11, S12	0.999	0.04	0.04	0.96	0.01	0.000015
Bad card	S1, S1	0.001	1	1	1	1	0.001000
PIN failed	S1, S2, S3, S1	0.999	0.04	0.04	0.04	1	0.000064

Table 14.13 SATM Operational Profile

	Path	*Transition Probabilities*					*Path Probability*
First try	S1, S2, S5, S8, S12	0.999	0.96	0.85	1	1	81.5184%
First try	S1, S2, S5, S7, S12	0.999	0.96	0.05	1	1	4.7952%
First try	S1, S2, S5, S9, S12	0.999	0.96	0.05	1	1	4.7952%
Second try	S1, S2, S3, S5, S8, S12	0.999	0.04	0.96	0.85	1	3.2607%
First try	S1, S2, S5, S6, S12	0.999	0.96	0.02	1	1	1.9181%
First try	S1, S2, S5, S10, S12	0.999	0.96	0.02	1	1	1.9181%
First try	S1, S2, S5, S11, S12	0.999	0.96	0.01	1	1	0.9590%
Second try	S1, S2, S3, S5, S7, S12	0.999	0.04	0.96	0.05	1	0.1918%
Second try	S1, S2, S3, S5, S9, S12	0.999	0.04	0.96	0.05	1	0.1918%
Third try	S1, S2, S3, S4, S5, S8, S12	0.999	0.04	0.04	0.96	0.85	0.1304%
Bad card	S1, S1	0.001	1	1	1	1	0.1000%
Second try	S1, S2, S3, S5, S6, S12	0.999	0.04	0.96	0.02	1	0.0767%
Second try	S1, S2, S3, S5, S10, S12	0.999	0.04	0.96	0.02	1	0.0767%
Second try	S1, S2, S3, S5, S11, S12	0.999	0.04	0.96	0.01	1	0.0384%
Third try	S1, S2, S3, S4, S5, S7, S12	0.999	0.04	0.04	0.96	0.05	0.0077%
Third try	S1, S2, S3, S4, S5, S9, S12	0.999	0.04	0.04	0.96	0.05	0.0077%
PIN failed	S1, S2, S3, S1	0.999	0.04	0.04	0.04	1	0.0064%
Third try	S1, S2, S3, S4, S5, S6, S12	0.999	0.04	0.04	0.96	0.02	0.0031%
Third try	S1, S2, S3, S4, S5, S10, S12	0.999	0.04	0.04	0.96	0.02	0.0031%
Third try	S1, S2, S3, S4, S5, S11, S12	0.999	0.04	0.04	0.96	0.01	0.0015%

Whatever approach is used, the final transition probabilities are still estimates. On the positive side, we could do a sensitivity analysis. In this situation, the overall ordering of probabilities is not particularly sensitive to small variations in the individual transition probabilities.

Operational profiles provide a feeling for the traffic mix of a delivered system. This is helpful for reasons other than only optimizing system testing. These profiles can also be used in conjunction with simulators to get an early indication of execution time performance and system transaction capacity.

14.8.2 Risk-Based Testing

Risk-based testing is a refinement of operational profiles. Just knowing which threads are most likely to execute might not be enough. What if a malfunction of a somewhat obscure thread were to be extremely costly? The cost might be in terms of legal penalties, loss of revenue, or difficulty of repair. The basic definition of risk is

Risk = cost * (probability of occurrence)

Since operational profiles give the (estimate of) probability of occurrence, we only need to make an estimate of the cost factor.

Hans Schaefer, a consultant who specializes in risk-based testing, advises that the first step is to group the system into risk categories. He advises four risk categories: Catastrophic, Damaging, Hindering, and Annoying (Schaefer, 2005). Next, the cost weighting is assessed. He suggests a logarithmic weighting: 1 for low cost of failure, 3 for medium, and 10 for high. Why logarithmic? Psychologists are moving in this direction because subjects who are asked to rank factors on linear scales, for example, 1 for low and 5 for high, do not make enough of a distinction in what is usually a subjective assessment. Table 14.14 is the result of this process for our SATM use cases in Table 14.13. In this assessment, failure of a deposit was the most severe.

Table 14.14 SATM Risk Assessment

Use Case Description	*Use Case Probability*	*Cost of Failure*	*Risk*
First try, normal withdrawal	81.5184%	3	2.4456
First try, deposit	4.7952%	10	0.4795
First try, withdrawal but insufficient funds	1.9181%	10	0.1918
First try, withdrawal not multiple of $20	4.7952%	3	0.1439
Second try, normal withdrawal	3.2607%	3	0.0978
First try, withdrawal, ATM low on cash	0.9590%	10	0.0959
First try, balance inquiry	1.9181%	1	0.0192
Second try, deposit	0.1918%	10	0.0192
Insertion of invalid ATM card	0.1000%	10	0.0100
Second try, withdrawal insufficient funds	0.0767%	10	0.0077
Second try, withdrawal not multiple of $20	0.1918%	3	0.0058
Third try, normal withdrawal	0.1304%	3	0.0039
Second try, withdrawal, ATM low on cash	0.0384%	10	0.0038
Second try, balance inquiry	0.0767%	1	0.0008
Third try, deposit	0.0077%	10	0.0008
Third try, withdrawal but insufficient funds	0.0031%	10	0.0003
Third try, withdrawal not multiple of $20	0.0077%	3	0.0002
PIN entry failed after third attempt	0.0064%	3	0.0002
Third try, withdrawal, ATM low on cash	0.0015%	10	0.0002
Third try, balance inquiry	0.0031%	1	0.0000

Schaefer's risk categories applied to the SATM use cases are given below. Deposit failures are seen as most severe because a customer may depend on a deposit being made to cover other checks. Balance inquiries are least severe because a malfunction is only inconvenient.

Catastrophic:	deposits, invalid withdrawals
Damaging:	normal withdrawals
Hindering:	invalid ATM card, PIN entry failure
Annoying:	balance inquiries

The risk-ordered SATM use cases in Table 4.14 differ slightly from their operational profile in Table 14.13. The normal withdrawal after a successful PIN entry on the first try still heads the list, mostly due to its high probability.

As a refinement, Schaefer suggests assigning several attributes to a use case and giving these attributes weighting values. For our SATM system, we might consider factors such as customer convenience, bank security, and identity theft.

14.9 Nonfunctional System Testing

The system testing ideas thus far discussed have been based on specification-based, or behavioral, requirements. Functional requirements are absolutely in the *does view*, as they describe what a system does (or should do). To generalize, nonfunctional testing refers to how well a system performs its functional requirements. Many nonfunctional requirements are categorized onto "-abilities": reliability, maintainability, scalability, usability, compatibility, and so on. While many practitioners have clear ideas on the meaning of the -abilities in their product domains, there is not much standardization of either the terms or the techniques. Here we consider the most common form of nonfunctional testing—stress testing.

14.9.1 Stress Testing Strategies

Synonymously called performance testing, capacity testing, or load testing, this is the most common, and maybe the most important form of nonfunctional testing. Because stress testing is so closely related to the nature of the system being tested, stress testing techniques are also application dependent. Here we describe three common strategies, and illustrate them with examples.

14.9.1.1 Compression

Consider the performance of a system in the presence of extreme loads. A web-based application may be very popular, and its server might not have the capacity. Telephone switching systems use the term Busy Hour Call Attempts (BHCAs) to refer to such offered traffic loads. The strategy in those systems is best understood as compression.

A local switching system must recognize when a subscriber originates a call. Other than sensing a change in subscriber line status from idle to active, the main indicator of a call attempt is the entry of digits. Although some dial telephones still exist, most subscribers use digit keys. The technical term is Dual Tone Multifrequency (DTMF) tones, as the usual 3 × 4 array of digit keys has three frequencies for the columns and four frequencies for the rows of digit keypads. Each digit is therefore represented by two frequency tones, hence the name. The local switching system must convert the tones to a digital form, and this is done with a DTMF receiver.

Here is a hypothetical example, with simplified numbers, to help understand the compression strategy. Suppose a local switching system must support 50,000 BHCAs. To do so, the system might have 5000 DTMF receivers. To test this traffic load, somehow 50,000 call originations must be generated in 60 minutes. The whole idea of compression strategies it to reduce these numbers to more manageable sizes. If a prototype only has 50 DTMF receivers, the load testing would need to generate 500 call attempts.

This pattern of compressing some form of traffic and associated devices to handle the offered traffic occurs in many application domains, hence the general term, traffic engineering.

14.9.1.2 Replication

Some nonfunctional requirements may be unusually difficult to actually perform. Many times, actual performance would destroy the system being tested (destructive vs. nondestructive testing). There was a Calvin and Hobbes comic strip that succinctly explained this form of testing. In the first frame, Calvin sees a sign on a bridge that says “Maximum weight 5 tons.” He asks his father how this is determined. The father answers that successively heavier trucks are driven over the bridge until the bridge collapses. In the last frame, Calvin has his standard shock/horror expression. Rather than destroy a system, some form of replication can be tried. Two examples follow.

One of the nonfunctional requirements for an army field telephone switching center was that it had to be operational after a parachute drop. Actually doing this was both very expensive and logistically complex. None of the system testers knew how to replicate this; however, in consultation with a former paratrooper, the testers learned that the impact of a parachute drop is similar to jumping off a ten-foot (three meter) wall. The testers put a prototype on a fork lift skid, lifted it to a height of ten feet, and tilted it forward until it fell off the skid. After hitting the ground, the prototype was still operational, and the test passed.

One of the most dangerous incidents for aircraft is a mid-air collision with a bird. Here is an excerpt of a nonfunctional requirement for the F 35 jet aircraft built by Lockheed Martin (Owens et al., 2009).

> *The Canopy System Must Withstand Impact of a 4 lb Bird at 480 Knots on the Reinforced Windscreen and 350 Knots on the Canopy Crown Without:*
>
> - *Breaking or Deflecting so as to Strike the Pilot When Seated in the Design Eye “High” Position,*
> - *Damage To The Canopy That Would Cause Incapacitating Injury To The Pilot, or*
> - *Damage That Would Preclude Safe Operation of, or Emergency Egress from the Aircraft*

Clearly it would be impossible to arrange a mid-air bird collision, so the Lockheed Martin testers replicated the problem with an elaborate cannon that would shoot a dead chicken at the windscreen and at the canopy. The tests passed.

There is an urban legend, debunked on Snopes.com, about follow-up on a British (or French, or fill-in-your-favorite-country) firm that used the same idea for canopy testing, but their tests all failed. When they asked the US testers why the failures were so consistent, they received a wry answer: “you need to thaw the chicken first.” Why mention this? If nonfunctional testing is done with a replication strategy, it is important to replicate, as closely as possible, the actual test scenario. (But it is funny.)

14.9.2 Mathematical Approaches

In some cases, nonfunctional testing cannot be done either directly, indirectly, or with commercial tools. There are three forms of analysis that might help—queuing theory, reliability models, and simulation.

14.9.2.1 Queuing Theory

Queuing theory deals with servers and queues of tasks that use the service. The mathematics behind queuing theory deals with task arrival rates, and service times, as well as the number of queues and the number of servers. In everyday life, we see examples of queuing situations: checkout lines in a grocery store, lines to buy tickets at a movie theater, or lift lines at a ski area. Some settings, for example, a local post office, uses a single queue of patrons waiting for service at one of several clerk positions. This happens to be the most efficient queuing discipline—single queue, multiple server. Service times represent some form of system capacity, and queues represent traffic (transactions) offered to the system.

14.9.2.2 Reliability Models

Reliability models are somewhat related to queuing theory. Reliability deals with failure rates of components and computes characteristics such as likelihood of system failure, mean time to failure (MTTF), mean time between failures (MTBF), and mean time to repair (MTTR). Given actual or assumed failure rates of system components, these quantities can be computed.

A telephone switching system has a reliability requirement of not more than two hours of downtime in 40 years of continuous operation. This is an availability of 0.99999429, or stated negatively, failure rate of 5.7×10^{-6}, (0.0000057). How can this be guaranteed? Reliability models are the first choice. They can be expressed as tree diagrams or as directed graphs, very similar to the approach used to compute an operational profile. These models are based on failure rates of individual system components that are linked together physically, and abstractly in the reliability model.

A digital end office intended for the rural U.S. market had to be certified by an agency of the U.S. government, the Rural Electric Administration (REA). That body followed a compression strategy, and required an on-site test for six months. If the system functioned with less than 30 minutes of downtime, it was certified. A few months into the test interval, the system had less than two minutes of downtime. Then a tornado hit the town and destroyed the building that contained the system. The REA declared the test to be a failure. Only extreme pleading resulted in a retest. The second time, there was less than 30 seconds of downtime in the six-month interval.

Reliability models have a solid history of applicability to physical systems, but can they be applied to software? Physical components can age, and therefore deteriorate. This is usually shown in the Weibull distribution, in which failures drop to nearly zero rapidly. Some forms show an increase after an interval that represents the useful life of a component. The problem is that software, once well tested, does not deteriorate. The main difference between reliability models applied to software versus hardware comes down to the arrival rate of failures. Testing based on operational profiles, and the extension to risk-based testing is a good start; however, no amount of testing can guarantee the absence of software faults.

14.9.2.3 Monte Carlo Testing

Monte Carlo testing might be considered a last resort in the system tester's arsenal. The basic idea of Monte Carlo testing is to randomly generate a large number of threads (transactions) and then see if anything unexpected happens. The Monte Carlo part comes from the use of pseudorandom numbers, not from the fact that the whole approach is a gamble. Monte Carlo testing has been successful in applications where computations involving physical (as opposed to logical, see Chapter 6) variables are used. The major drawback to Monte Carlo testing is that the large number of random transactions requires a similarly large number of expected outputs in order to determine whether a random test case passes or fails.

14.10 Atomic System Function Testing Example

We can illustrate ASF testing on our integrationNextDate pseudocode. This version differs slightly from that in Chapter 13. A few output statements were added to make ASF identification more visible.

```
1   Main integrationNextDate      ‘start program event occurs here
       Type   Date
              Month As Integer
              Day As Integer
              Year As Integer
       EndType
       Dim today As Date
       Dim tomorrow As Date
2      Output(“Welcome to NextDate!”)
3      GetDate(today)                                ‘msg1
4      PrintDate(today)                              ‘msg2
5      tomorrow = IncrementDate(today)               ‘msg3
6      PrintDate(tomorrow)                           ‘msg4
7   End Main

8   Function isLeap(year) Boolean
9      If (year divisible by 4)
10          Then
11             If (year is NOT divisible by 100)
12                   Then isLeap = True
13                   Else
14                       If (year is divisible by 400)
15                            Then isLeap = True
16                            Else isLeap = False
17                       EndIf
18             EndIf
19     Else isLeap = False
20     EndIf
21  End (Function isLeap)

22  Function lastDayOfMonth(month, year) Integer
23     Case month Of
24         Case 1: 1, 3, 5, 7, 8, 10, 12
25             lastDayOfMonth = 31
26         Case 2: 4, 6, 9, 11
27             lastDayOfMonth = 30
28         Case 3: 2
```

```
            If (isLeap(year))                     ‘msg5
                  Then lastDayOfMonth = 29
                  Else lastDayOfMonth = 28
            EndIf
   EndCase
End (Function lastDayOfMonth)

Function GetDate(aDate) Date
   dim aDate As Date

   Function ValidDate(aDate)   Boolean ‘within scope of GetDate
       dim aDate As Date
       dim dayOK, monthOK, yearOK As Boolean
       If ((aDate.Month > 0) AND (aDate.Month < = 12)
            Then   monthOK = True
                   Output(“Month OK”)
            Else   monthOK = False
                   Output(“Month out of range”)
       EndIf
       If (monthOK)
            Then
            If ((aDate.Day > 0) AND (aDate.Day < =
                  lastDayOfMonth(aDate.Month, aDate.Year))  ‘msg6
                      Then   dayOK = True
                             Output(“Day OK”)
                      Else   dayOK = False
                             Output(“Day out of range”)
            EndIf
       EndIf
       If ((aDate.Year > 1811) AND (aDate.Year < = 2012)
            Then   yearOK = True
                   Output(“Year OK”)
            Else   yearOK = False
                   Output(“Year out of range”)
       EndIf
       If (monthOK AND dayOK AND yearOK)
            Then ValidDate = True
                    Output(“Date OK”)
            Else ValidDate = False
                    Output(“Please enter a valid date”)
       EndIf
   End (Function ValidDate)

‘GetDate body begins here
       Do
            Output(“enter a month”)
            Input(aDate.Month)
            Output(“enter a day”)
            Input(aDate.Day)
            Output(“enter a year”)
            Input(aDate.Year)
            GetDate.Month = aDate.Month
            GetDate.Day = aDate.Day
            GetDate.Year = aDate.Year
       Until (ValidDate(aDate))                                    ‘msg7
End (Function GetDate)
```

```
77  Function IncrementDate(aDate) Date
78              If (aDate.Day < lastDayOfMonth(aDate.Month))    ‘msg8
79                   Then aDate.Day = aDate.Day + 1
80                   Else aDate.Day = 1
81                        If (aDate.Month = 12)
82                             Then aDate.Month = 1
83                                  aDate.Year = aDate.Year + 1
84                             Else aDate.Month = aDate.Month + 1
85                        EndIf
86              EndIf
87  End (IncrementDate)

88  Procedure PrintDate(aDate)
89        Output(“Day is “, aDate.Month, “/”, aDate.Day, “/”, aDate.Year)
90  End (PrintDate)
```

14.10.1 Identifying Input and Output Events

Recall that an ASF begins with a port input event, conducts some processing, and, depending on the project-chosen granularity, ends with one or more port output events. We can identify ASFs from source code by locating the nodes at which port inputs and outputs occur. Table 14.15 lists the port input and output events in integrationNextDate and the corresponding node numbers.

Table 14.15 Location of Port Events in integrationNextDate

Input Events	*Node*	*Output Event Description*	*Node*
e0: Start program event	1	e7: Welcome message	2
e1: Center a valid month	67	e8: Print today’s date	4
e2: Enter an invalid month	67	e9: Print tomorrow’s date	6
e3: Enter a valid day	69	e10: “Month OK”	39
e4: Enter an invalid day	69	e11: “Month out of range”	41
e5: Enter a valid year	71	e12: “Day OK”	47
e6: Enter an invalid year	71	e13: “Day out of range”	49
		e14: “Year OK”	54
		e15: “Year out of range”	56
		e16: “Date OK”	60
		e17: “Please enter a valid date”	62
		e18: “Enter a month”	66
		e19: “Enter a day”	68
		e20: “Enter a year”	70
		e21: “Day is month, day, year”	89

14.10.2 Identifying Atomic System Functions

The next step is to identify the ASFs in terms of the port input and output events. Table 14.16 contains the first attempt at ASF identification. There is a subtle problem with this first set of ASFs. They presume that the states in the ASF graph in Figure 14.14 are independent, but they are not. As we have seen, dependent nodes often lead to impossible paths.

The ASF graph shows all the ways both valid months, days, and years can be entered, but the transition to ASF-8 (or to ASF-9) depends on the history of previous ASFs. Since FSMs have no memory, these transitions are necessarily undefined.

Table 14.16 First Attempt at ASFs

Atomic System Function	*Inputs*	*Outputs*
ASF-1: start program	e0	e7
ASF-2: enter a valid month	e1	e10
ASF-3: enter an invalid month	e2	e11
ASF-4: enter a valid day	e3	e12
ASF-5: enter an invalid day	e4	e13
ASF-6: enter a valid year	e5	e14
ASF-7: enter an invalid year	e6	e15
ASF-8: print for valid input		
ASF-9: print for invalid input		

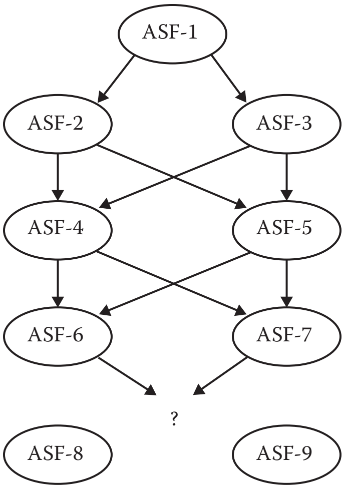


Figure 14.14 Directed graph of Atomic System Functions for integrationNextDate.

14.10.3 Revised Atomic System Functions

The Do Until loop from nodes 65 to 75 allows for many mistakes. The checking for valid month, day, and year values is linear, but all three must be correct to terminate the Do Until loop. Since we can make any number of input variable mistakes, in any order, there is no way to represent in an ASF graph when the Do Until loop will terminate. Incidentally, this is case of the Anna Karenina principle (see Diamond, 1997). The principle refers to situations where a conjunction of criteria must all be true; any one becoming false negates the whole situation. It comes from the first sentence in Leo Tolstoy's famous Russian novel, *Anna Karenina*: "Happy families are all alike; every unhappy family is unhappy in its own way."

The second attempt (see Table 14.17) postulates larger ASFs (maybe they are "molecular"?) that have pluralities of port inputs and port outputs.

Now each ASF is the entry of a triple (month, day, year). ASFs 2, 3, and 4 make one mistake at a time, and ASF-5 gets all the values right. Would we really need the last four (two or three mistakes at a time)? Probably not. That kind of testing should have been done at the unit level. Figure 14.15 is the new ASF graph for the first five of the "revised" ASFs.

Table 14.17 Second Attempt at ASFs

Atomic System Function	*Inputs*	*Outputs*
ASF-1: start program	e0	e7
ASF-2: enter a date with an invalid month, valid day, and valid year	e2, e3, e5	e11, e12, e14, e17
ASF-3: enter a date with an invalid day, valid month, and valid year	e1, e4, e5	e10, e13, e14, e17
ASF-4: enter a date with an invalid year, valid day, and valid month	e1, e3, e6	e10, e12, e15, e17
ASF-5: enter a date with valid month, day, and year	e1, e3, e5	e10, e12, e14, e16, e21
ASF-6: enter a date with valid month, invalid day, and invalid year	e1, e4, e6	e10, e13, e15, e17
ASF-7: enter a date with valid day, invalid month, and invalid year	e2, e3, e6	e11, e12, e15, e17
ASF-8: enter a date with valid year, invalid day, and invalid month	e5, e4, 46	e14, e13, e15, e17
ASF-9: enter a date with invalid month, invalid day, and invalid year	e2, e4, e6	e11, e13, e15, e17

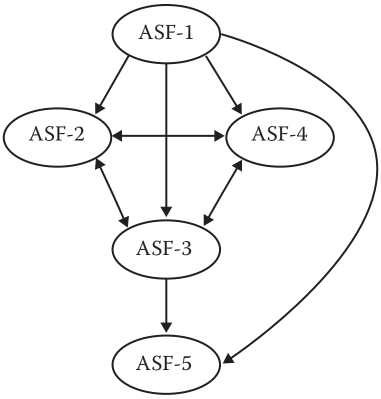


Figure 14.15 Directed graph of five Atomic System Functions for integrationNextDate.

EXERCISES

1. One of the problems of system testing, particularly with interactive systems, is to anticipate all the strange things the user might do. What happens in the SATM system if a customer enters three digits of a PIN and then leaves?
2. To remain "in control" of abnormal user behavior (the behavior is abnormal, not the user), the SATM system might introduce a timer with a 30-second time-out. When no port input event occurs for 30 seconds, the SATM system could ask if the user needs more time. The user can answer yes or no. Devise a new screen and identify port events that would implement such a time-out event.
3. Suppose you add the time-out feature described in Exercise 2 to the SATM system. What regression testing would you perform?
4. Make an additional refinement to the PIN try finite state machine (Figure 14.6) to implement your time-out mechanism from Exercise 2, then revise the thread test case in Table 14.3.
5. Develop an incidence matrix similar to Table 14.10 for the PIN entry short use cases in Table 14.8.
6. Does it make sense to use test coverage metrics in conjunction with operational profiles? Same question for risk-based testing. Discuss this.

For questions 7 through 9, revisit the description of the Garage Door Controller in Chapter 2 and the corresponding finite state machine in Chapter 4.

7. Develop extended essential use cases for the Garage Door Controller.
8. The input and output events for the Garage Door Controller are shown in Figure 4.6. Use these as starting points for incidence matrices of your use cases with respect to input events, output events, and all events.
9. Develop Event-Driven Petri Nets for the Garage Door Controller.

References

Diamond, J., *Guns, Germs, and Steel,* W. W. Norton, New York, 1997.

Jorgensen, P.C., System testing with pseudo-structures, *American Programmer*, Vol. 7, No. 4, April 1994, pp. 29–34.

Jorgensen, P.C., *Modeling Software Behavior: A Craftsman's Approach*, CRC Press, New York, 2009.

Larman, C., *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design*, 2nd ed., Prentice-Hall, Upper Saddle River, NJ, 2001.

Schaefer, H., Risk based testing, strategies for prioritizing tests against deadlines, *Software Test Consulting*, http://home.c2i.net/schaefer/testing.html, 2005.

Owens, S.D., Caldwell, E.O. and Woodward, M.R., Birdstrike certification tests of F-35 canopy and airframe structure, *2009 Aircraft Structural Integrity Program (ASIP) Conference*, Jacksonville, FL, December 2009, also can be found at Trimble, S., July 28, 2010, http://www.flightglobal.com/blogs/the-dewline/2010/07/video-f-35-birdstrike-test-via.html and http://www.flightglobal.com/blogs/the-dewline/Birdstrike%20Impact%20Studies.pdf.

Chapter 15

Object-Oriented Testing

Both theoretical and practical work on the testing of object-oriented software has flourished since the second half of the 1990s, leading to the clear dominance of the paradigm in 2013. One of the original hopes for object-oriented software was that objects could be reused without modification or additional testing. This was based on the assumption that well-conceived objects encapsulate functions and data "that belong together," and once such objects are developed and tested, they become reusable components. The new consensus is that there is little reason for this optimism—object-oriented software has potentially more severe testing problems than those for traditional software. On the positive side, the Unified Modeling Language (UML) is clearly the *de facto* framework for object-oriented software development.

15.1 Issues in Testing Object-Oriented Software

Our goal in this section is to identify the testing issues raised by object-oriented software. First, we have the question of levels of testing; this, in turn, requires clarification of object-oriented units. Next, we consider some of the implications of the strategy of composition (as opposed to functional decomposition). Object-oriented software is characterized by inheritance, encapsulation, and polymorphism; therefore, we look at ways that traditional testing can be extended to address the implications of these issues. In the remaining sections, we examine class testing, integration, system testing, UML-based testing, and the application of data flow testing to object-oriented software. We will do this using the object-oriented calendar version of the integrationNextDate procedural example and the windshield wiper controller.

15.1.1 Units for Object-Oriented Testing

Traditional software has a variety of definitions for "unit." Two that pertain to object-oriented software are

> A unit is the smallest software component that can be compiled and executed.
> A unit is a software component that would never be assigned to more than one designer to develop.

These guidelines can be contradictory. Certain industrial applications have huge classes; these clearly violate the one-designer, one-class definition. In such applications, it seems better to define an object-oriented unit as the work of one person, which likely ends up as a subset of the class operations. In an extreme case, an object-oriented unit might be a subclass of a class that contains only the attributes needed by a single operation or method. (In this chapter, we will use "operation" to refer to the definition of a class function and "method" to refer to its implementation.) For such units, object-oriented unit testing reduces to traditional testing. This is a nice simplification but somewhat problematic because it shifts much of the object-oriented testing burden onto integration testing. Also, it does not exploit the gains made by encapsulation.

The class-as-unit choice has several advantages. In a UML context, a class has an associated StateChart that describes its behavior. Later, we shall see that this is extremely helpful in test case identification. A second advantage is that object-oriented integration testing has clearer goals, namely, to check the cooperation of separately tested classes, which echoes traditional software testing.

15.1.2 Implications of Composition and Encapsulation

Composition (as opposed to decomposition) is the central design strategy in object-oriented software development. Together with the goal of reuse, composition creates the need for very strong unit testing. Because a unit (class) may be composed with previously unknown other units, the traditional notions of coupling and cohesion are applicable. Encapsulation has the potential to resolve this concern, but only if the units (classes) are highly cohesive and very loosely coupled. The main implication of composition is that, even presuming very good unit-level testing, the real burden is at the integration testing level.

Some of this is clarified by example. Suppose we revisit the Saturn windshield wiper system from an object-oriented viewpoint. We would most likely identify three classes: lever, wiper, and dial; their behavior is shown by the finite state machines (which are special cases of StateCharts) in Figure 15.1. The pseudocode for one choice of the interfaces of these classes is as follows:

```
Class lever(leverPosition;
      private senseLeverUp(),
      private senseLeverDown())
Class dial(dialPosition;
      private senseDialUp(),
      private senseDialDown())
Class wiper(wiperSpeed;
      setWiperSpeed(newSpeed))
```

The lever and dial classes have operations that sense physical events on their respective devices. When these methods (corresponding to the operations) execute, they report their respective device positions to the wiper class. The interesting part of the windshield wiper example is that the lever and the dial are independent devices, and they interact when the lever is in the INT (intermittent) position. The question raised by encapsulation is, where should this interaction be controlled?

The precept of encapsulation requires that classes only know about themselves and operate on their own. Thus, the lever does not know the dial position, and the dial does not know the lever position. The problem is that the wiper needs to know both positions. One possibility, as shown in the previous interface, is that the lever and dial always report their positions, and the wiper figures

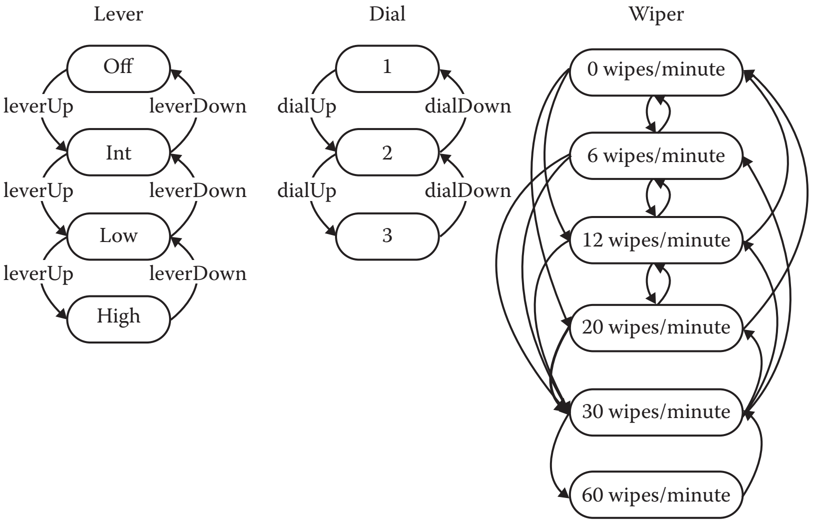


Figure 15.1 Behavior of windshield wiper classes.

out what it must do. With this choice, the wiper class becomes the "main program" and contains the basic logic of the whole system.

Another choice might be to make the lever class the "smart" object because it knows when it is in the INT state. With this choice, when the response to a lever event puts the lever in the INT state, a method gets the dial status (with a getDialPosition message) and simply tells the wiper what speed is needed. With this choice, the three classes are more tightly coupled, and, as a result, less reusable. Another problem occurs with this choice. What happens if the lever is in the INT position and a subsequent dial event occurs? There would be no reason for the lever to get the new dial position, and no message would be sent to the wiper class.

A third choice might be to make the wiper the main program (as in the first choice), but use a Has relation to the lever and dial classes. With this choice, the wiper class uses the sense operations of the lever and dial classes to detect physical events. This forces the wiper class to be continuously active, in a polling sense, so that asynchronous events at the lever and dial can be observed.

Consider these three choices from the standpoint of composition and encapsulation. The first choice (cleverly named because it is the best) has very little coupling among the classes. This maximizes the potential of the classes to be reused (i.e., composed in unforeseen ways). For example, a cheaper windshield wiper might omit the dial altogether, and an expensive windshield wiper might replace the three-position dial with a "continuous" dial. Similar changes might be made to the lever. In the other two choices, the increased coupling among the classes reduces their ability to be composed. Our conclusion: good encapsulation results in classes that can more easily be composed (and thus reused) and tested.

15.1.3 Implications of Inheritance

Although the choice of classes as units seems natural, the role of inheritance complicates this choice. If a given class inherits attributes and/or operations from super classes, the stand-alone compilation criterion of a unit is sacrificed. Binder (1996) suggests "flattened classes" as an answer.

A flattened class is an original class expanded to include all the attributes and operations it inherits. Flattened classes are mildly analogous to the fully flattened data flow diagrams of Structured Analysis. (Notice that flattened classes are complicated by multiple inheritance, and really complicated by selective and multiple selective inheritance.) Unit testing on a flattened class solves the inheritance problem, but it raises another. A flattened class will not be part of a final system, so some uncertainty remains. Also, the methods in a flattened class might not be sufficient to test the class. The next work-around is to add special-purpose test methods. This facilitates class-as-unit testing but raises a final problem: a class with test methods is not (or should not be) part of the delivered system. This is perfectly analogous to the question of testing original or instrumented code in traditional software. Some ambiguity is also introduced: the test methods can also be faulty. What if a test method falsely reports a fault, or worse, incorrectly reports success? Test methods are subject to the same false-positive and false-negative outcomes as medical experiments. This leads to an unending chain of methods testing other methods, very much like the attempt to provide external proofs of consistency of a formal system.

Figure 15.2 shows a UML inheritance diagram of a part of our earlier Simple Automated Teller Machine (SATM) system; some functionality has been added to make this a better example. Both checking and savings accounts have account numbers and balances, and these can be accessed and changed. Checking accounts have a per-check processing charge that must be deducted from the account balance. Savings accounts draw interest that must be calculated and posted on some periodic basis.

If we did not "flatten" the checkingAccount and savingsAccount classes, we would not have access to the balance attributes, and we would not be able to access or change the balances. This is clearly unacceptable for unit testing. Figure 15.3 shows the "flattened" checkingAccount and savingsAccount classes. These are clearly stand-alone units that are sensible to test. Solving one problem raises another: with this formulation, we would test the getBalance and setBalance operations twice, thereby losing some of the hoped-for economies of object orientation.

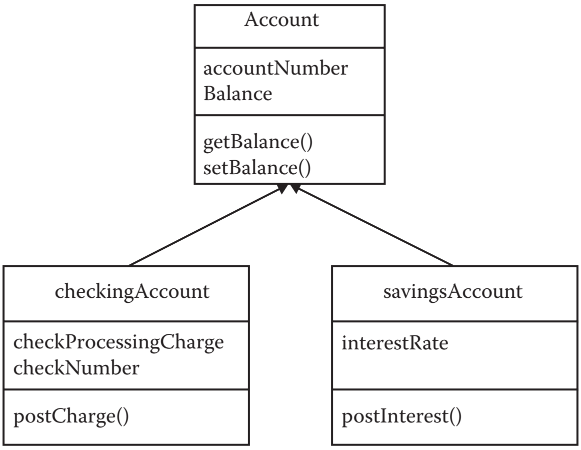


Figure 15.2 UML inheritance.

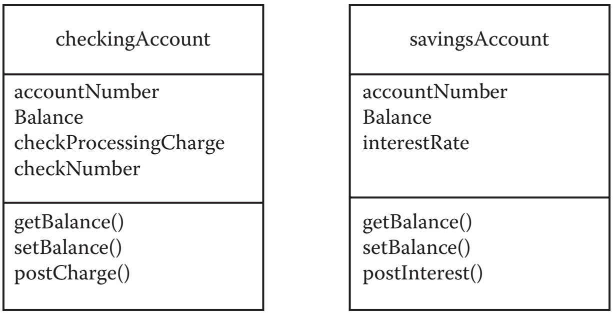


Figure 15.3 Flattened checkingAccount and savingsAccount classes.

15.1.4 Implications of Polymorphism

The essence of polymorphism is that the same method applies to different objects. Considering classes as units implies that any issues of polymorphism will be covered by the class/unit testing. Again, the redundancy of testing polymorphic operations sacrifices hoped-for economies.

15.1.5 Levels of Object-Oriented Testing

Three or four levels of object-oriented testing are used, depending on the choice of what constitutes a unit. If individual operations or methods are considered to be units, we have four levels: operation/method, class, integration, and system testing. With this choice, operation/method testing is identical to unit testing of procedural software. Class and integration testing can be well renamed as intraclass and interclass testing. The second level, then, consists of testing interactions among previously tested operations/methods. Integration testing, which we will see is the major issue of object-oriented testing, must be concerned with testing interactions among previously tested classes. Finally, system testing is conducted at the port event level, and is (or should be) identical to system testing of traditional software. The only difference is where system-level test cases originate.

15.1.6 Data Flow Testing for Object-Oriented Software

When we considered data flow testing in Chapter 9, it was restricted to a single unit. The issues of inheritance and composition require a deeper view. The emerging consensus in the object-oriented testing community is that some extension of data flow "should" address these special needs. In Chapter 9, we saw that data flow testing is based on identifying define and use nodes in the program graph of a unit and then considering various define/use paths. Procedure calls in traditional software complicate this formulation; one common work-around is to embed called procedures into the unit tested (very much like fully flattened classes). Later in this chapter (Section 15.4), we develop a revision of Event-Driven Petri Nets (EDPNs) that exactly describes data flow among object-oriented operations. Within this formulation, we can express the object-oriented extension of data flow testing.

15.2 Example: ooNextDate

Very little documentation is required by UML at the unit/class level. Here, we add the class responsibility collaboration (CRC) cards for each class, followed by the class pseudocode, and then the program graphs for the class operations (see Figures 15.4 through 15.7). CRC cards are not formally part of UML.

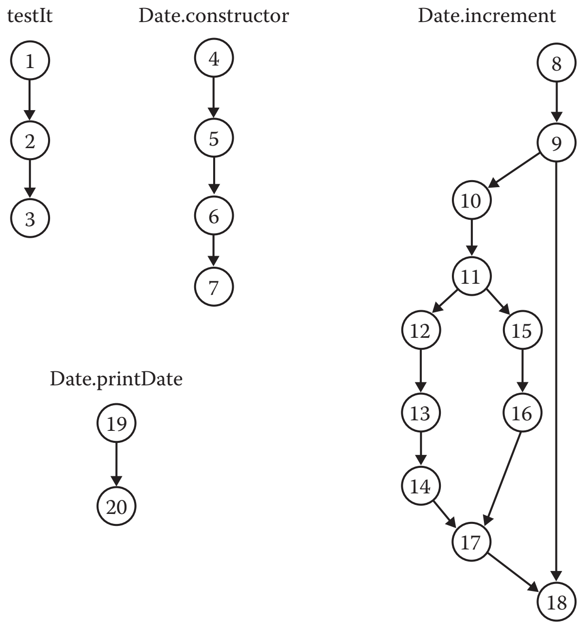

testIt
1
2
3
Date.constructor
4
5
6
7
Date.increment
8
9
10
11
12
15
13
16
14
17
18
Date.printDate
19
20

Figure 15.4 Program graphs for testIt and Date classes.

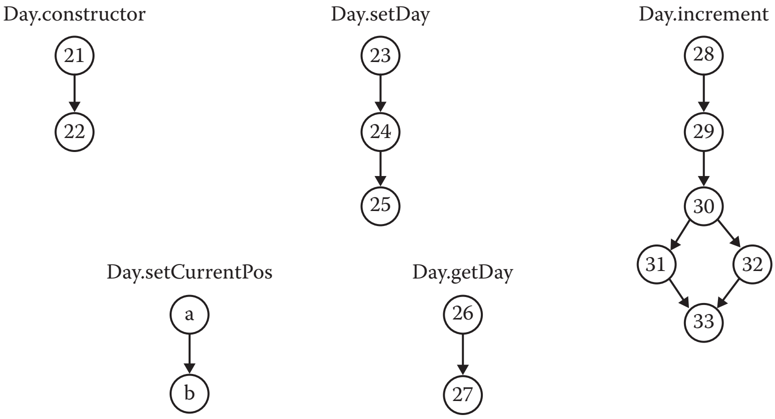

Day.constructor
21
22
Day.setDay
23
24
25
Day.increment
28
29
30
31
32
33
Day.setCurrentPos
a
b
Day.getDay
26
27

Figure 15.5 Program graphs for Day class.

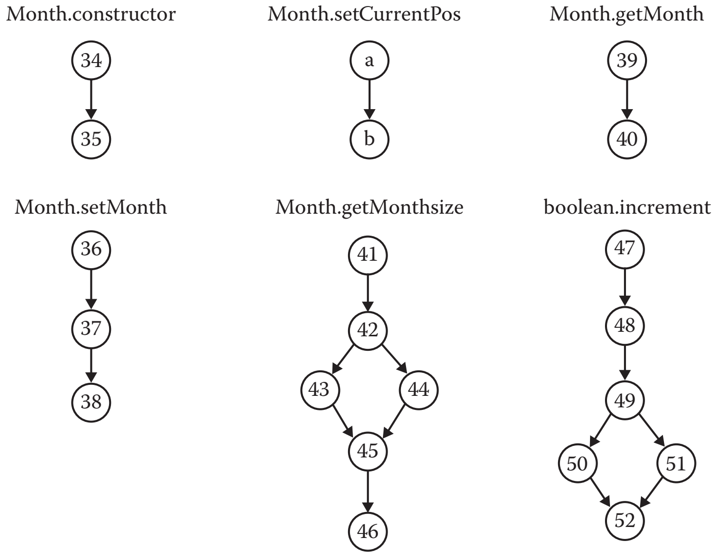

Month.constructor
34
35
Month.setCurrentPos
a
b
Month.getMonth
39
40
Month.setMonth
36
37
38
Month.getMonthsize
41
42
43
44
45
46
boolean.increment
47
48
49
50
51
52

Figure 15.6 Program graphs for Month class.

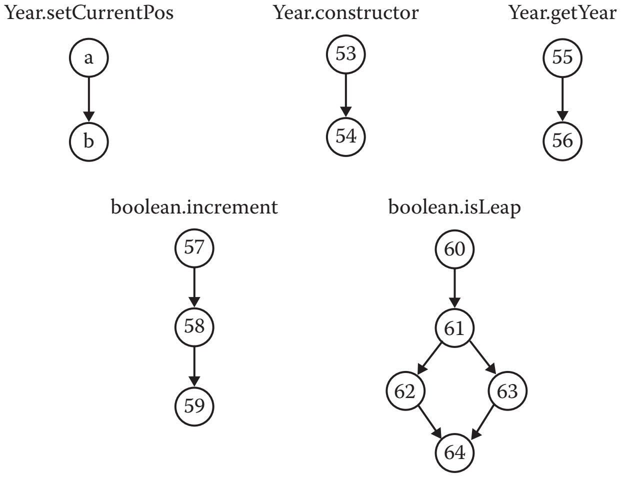

Year.setCurrentPos
a
b
Year.constructor
53
54
Year.getYear
55
56
boolean.increment
57
58
59
boolean.isLeap
60
61
62
63
64

Figure 15.7 Program graphs for Year class.

15.2.1 Class: CalendarUnit

Responsibility: Provides an operation to set its value in inherited classes and provides a Boolean operation that tells whether an attribute in an inherited class can be incremented.

```
class CalendarUnit  'abstract class
       currentPos As Integer
       CalendarUnit(pCurrentPos)
              currentPos = pCurrentPos
       End    'CalendarUnit
```

```
a      setCurrentPos(pCurrentPos)
b            currentPos = pCurrentPos
       End          'setCurrentPos
       abstract protected Boolean increment()
```

15.2.2 Class: testIt

Responsibility: Serves as a test driver by creating a test date object, then requesting the object to increment itself, and finally, to print its new value.

```
class testIt
   main()
1      testdate = instantiate Date(testMonth, testDay, testYear)     msg1
2      testdate.increment()                                          msg2
3      testdate.printDate()                                          msg3
   End      'testIt
```

15.2.3 Class: Date

Responsibility: A Date object is composed of day, month, and year objects. A Date object increments itself using the inherited Boolean increment methods in Day and Month objects. If the day and month objects cannot be incremented (e.g., last day of the month or year), Date's increment method resets day and month as needed. In the case of December 31, it also increments the year. The printDate operation uses the get() methods in Day, Month, and Year objects and prints a date value in mm/dd/yyyy format.

```
class Date
   private Day d
   private Month m
   private Year y
4  Date(pMonth, pDay, pYear)
5      y = instantiate Year(pYear)                                   msg4
6      m = instantiate Month(pMonth, y)                              msg5
7      d = instantiate Day(pDay, m)                                  msg6
   End      'Date constructor
8  increment ()
9      if (NOT(d.increment()))                                       msg7
10         Then
11             if (NOT(m.increment()))                               msg8
12                 Then
13                    y.increment()                                  msg9
14                    m.setMonth(1,y)                               msg10
15                 Else
16                    d.setDay(1, m)                                msg11
17             EndIf
18     EndIf
   End      'increment

19 printDate ()
20     Output (m.getMonth()+”/”+                                    msg12
                            d.getDay()+”/”+                         msg13
                            y.getYear())                            msg14
   End      'printDate
```

15.2.4 Class: Day

Responsibility: A Day object has a private month attribute that the increment method uses to see if a day value can be incremented or reset to 1. Day objects also provide get() and set() methods.

```
class Day isA CalendarUnit
   private Month m
21 Day(pDay, Month pMonth)
22     setDay(pDay, pMonth)                                            msg15
   End     'Day constructor

23 setDay(pDay, Month pMonth)
24     setCurrentPos(pDay)                                             msg16
25     m = pMonth
   End     'setDay

26 getDay()
27     return currentPos
   End     'getDay

28 boolean increment()
29     currentPos = currentPos + 1
30     if (currentPos < = m.getMonthSize())                            msg17
31         Then     return 'True'
32         Else     return 'False'
33     EndIf
   End     'increment
```

15.2.5 Class: Month

Responsibility: Month objects have a value attribute that is used as a subscript to an array of values of last month days (e.g., the last day of January is 31, the last day of February is 28, and so on). Month objects provide get() and set() services, and the inherited Boolean increment method. The possibility of February 29 is determined with the isleap message to a Year object.

```
   class Month isA CalendarUnit
       private Year y
       private sizeIndex = <31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31>
34 Month(pcur, Year pYear)
35     setMonth(pCurrentPos, Year pyear)                               msg18
   End      'Month constructor

36 setMonth(pcur, Year pYear)
37     setCurrentPos(pcur)                                             msg19
38     y = pYear
   End      'setMonth

39 getMonth()
40     return currentPos
   End      'getMonth

41 getMonthSize()
42     if (y.isleap())                                                 msg20
```

```
         Then     sizeIndex[1] = 29
         Else     sizeIndex[1] = 28
    EndIf
    return sizeIndex[currentPos -1]
End      ‘getMonthSize

boolean increment()
    currentPos = currentPos + 1
    if (currentPos > 12)
         Then     return ‘False’
         Else     return ‘True’
    EndIf
End ‘increment
```

15.2.6 Class: Year

Responsibility: In addition to the usual get() and set() methods, a Year object increments itself when the test date is December 31 of any year. Year objects provide a Boolean service that tells whether the current value corresponds to a leap year.

```
class Year isA CalendarUnit
     Year(pYear)
setCurrentPos(pYear)                                               msg21
End        ‘Year constructor

getYear()
     return currentPos
End        ‘getYear

boolean increment()
     currentPos = currentPos + 1
     return ‘True’
End        ‘increment

boolean isleap()
     if (((currentPos MOD 4 = 0) AND NOT(currentPos MOD 100 = 0))
                                      OR (currentPos MOD 400 = 0))
          Then     return ‘True’
          Else     return ‘False’
     EndIf
End        ‘isleap
```

15.3 Object-Oriented Unit Testing

In this section, we revisit the question of whether a class or a method is a unit. Most of the object-oriented literature leans toward the class-as-unit side, but this definition has problems. The guidelines mentioned in Section 15.1.1 also make sense for object-oriented software, but they do not resolve whether classes or methods should be considered as units. A method implements a single function, and it would not be assigned to more than one person, so methods might legitimately be considered as units. The smallest compilation requirement is problematic. Technically, we could compile a single-method class by ignoring the other methods in the class (probably by commenting them out), but this creates an organizational mess. We will present both views of object-oriented unit testing; you can let particular circumstances decide which is more appropriate.

15.3.1 Methods as Units

Superficially, this choice reduces object-oriented unit testing to traditional (procedural) unit testing. A method is nearly equivalent to a procedure, so all the traditional specification-based and code-based testing techniques apply. Unit testing of procedural code requires stubs and a driver test program to supply test cases and record results. Mock objects are the object-oriented analogue of this practice. Since instances of the nUnit framework are available for most object-oriented languages, the assert mechanism in those frameworks is the most convenient choice.

When we look more closely at individual methods, we see the happy consequence of encapsulation: they are generally simple. The pseudocode and corresponding program graphs for the classes that make up the ooCalendar application are in Section 15.2. Notice that the cyclomatic complexities of the various operations are uniformly low. Date.increment has the highest cyclomatic complexity, and it is only $V(G) = 3$. To be fair, this implementation is intentionally simple—no checking is necessary for valid inputs. With validity checking, the cyclomatic complexities would increase. As we saw in Chapter 6, equivalence class testing is a good choice for logic-intensive units. The Date.increment operation treats the three equivalence classes of days:

D1 = {day: 1 < = day < last day of the month}
D2 = {day: day is the last day of a non-December month}
D3 = {day: day is December 31}

At first, these equivalence classes appear to be loosely defined, especially D1, with its reference to the unspecified last day of the month and no reference to which month. Thanks to encapsulation, we can ignore these questions. (Actually, the questions are transferred to the testing of the Month.increment operation.)

Even though the cyclomatic complexity is low, the interface complexity is quite high. Looking at Date.increment again, notice the intense messaging: messages are sent to two operations in the Day class, to one operation in the Year class, and to two operations in the Month class. This means that nearly as much effort will be made to create the proper stubs as in identifying test cases. Another more important consequence is that much of the burden is shifted to integration testing. In fact, we can identify two levels of integration testing: intraclass and interclass integration.

15.3.2 Classes as Units

Treating a class as a unit solves the intraclass integration problem, but it creates other problems. One has to do with various views of a class. In the static view, a class exists as source code. This is fine if all we do is code reading. The problem with the static view is that inheritance is ignored, but we can fix this by using fully flattened classes. We might call the second view the compile-time view because this is when the inheritance actually “occurs.” The third view is the execution-time view, when objects of classes are instantiated. Testing really occurs with the third view, but we still have some problems. For example, we cannot test abstract classes because they cannot be instantiated. Also, if we are using fully flattened classes, we will need to “unflatten” them to their original form when our unit testing is complete. If we do not use fully flattened classes, in order to compile a class, we will need all the other classes above it in the inheritance tree. One can imagine the software configuration management implications of this requirement.

The class-as-unit choice makes the most sense when little inheritance occurs, and classes have what we might call internal control complexity. The class itself should have an "interesting" (mildly complex, nontrivial) StateChart, and there should be a fair amount of internal messaging. To explore class-as-unit testing, we will revisit the windshield wiper example with a more complex version.

15.3.2.1 Pseudocode for Windshield Wiper Class

The three classes discussed in Section 15.1.2 are merged into one class here. With this formulation, operations sense lever and dial events and maintain the state of the lever and dial in the leverPosition and dialPosition state variables. When a dial or lever event occurs, the corresponding sense method sends an (internal) message to the setWiperSpeed method, which, in turn, sets its corresponding state variable wiperSpeed. Our revised windshieldWiper class has three attributes, get and set operations for each variable, and methods that sense the four physical events on the lever and dial devices.

```
class windshieldWiper
      private wiperSpeed
      private leverPosition
      private dialPosition
      windshieldWiper(wiperSpeed, leverPosition, dialPosition)
      getWiperSpeed()
      setWiperSpeed()
      getLeverPosition()
      setLeverPosition()
      getDialPosition()
      setDialPosition()
      senseLeverUp()
      senseLeverDown()
      senseDialUp(),
      senseDialDown()
End class windshieldWiper
```

The class behavior is shown in the StateChart in Figure 15.8, where the three devices appear in the orthogonal components. The StateChart notation treats orthogonal regions as truly concurrent regions. Another way to look at this property is that orthogonal regions can represent communicating finite state machines. In the Dial and Lever components, transitions are caused by events, whereas the transitions in the wiper component are all caused by propositions that refer to what state is "active" in the Dial or Lever orthogonal components. (Such propositions are part of the rich syntax of transition annotations permitted in the StateMate product.)

15.3.2.2 Unit Testing for Windshield Wiper Class

Part of the difficulty with the class-as-unit choice is that there are levels of unit testing. In our example, it makes sense to proceed in a bottom–up order beginning with the get/set methods for the state variables (these are only present in case another class needs them). The dial and lever sense methods are all quite similar; pseudocode for the senseLeverUp method is given next.

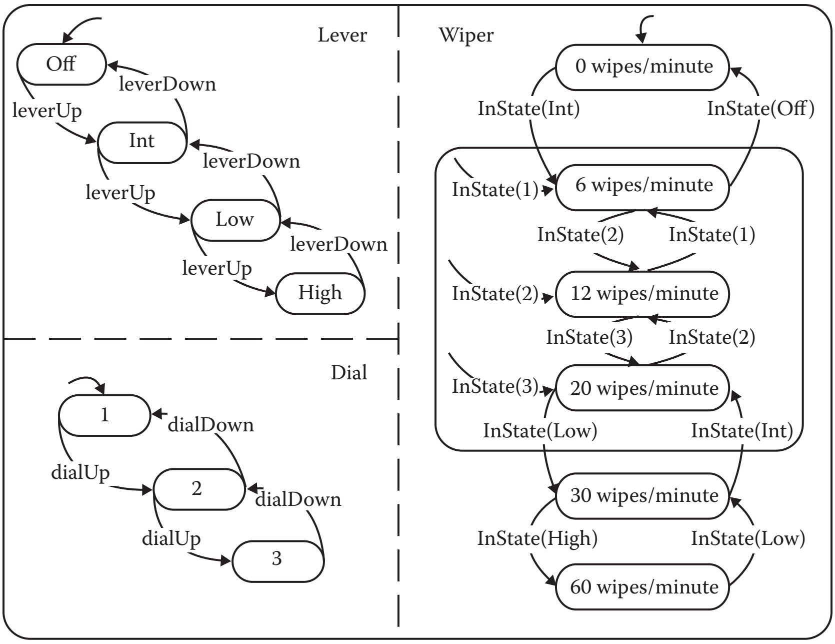

Lever
Off
leverDown
leverUp
Int
leverDown
leverUp
Low
leverDown
leverUp
High
Dial
1
dialDown
dialUp
2
dialDown
dialUp
3
Wiper
0 wipes/minute
InState(Int)
InState(Off)
InState(1)
6 wipes/minute
InState(2)
InState(1)
InState(2)
12 wipes/minute
InState(3)
InState(2)
InState(3)
20 wipes/minute
InState(Low)
InState(Int)
30 wipes/minute
InState(High)
InState(Low)
60 wipes/minute

Figure 15.8 Windshield wiper StateChart.

```
senseLeverUp()
      Case leverPosition Of
            Case 1: Off
                  leverPosition = Int
                  Case dialPosition Of
                        Case 1:1
                              wiperSpeed = 6
                        Case 2:2
                              wiperSpeed = 12
                        Case 3:3
                              wiperSpeed = 20
                  EndCase 'dialPosition
            Case 2:Int
                  leverPosition = Low
                  wiperSpeed = 30
            Case 3: Low
                  leverPosition = High
                  wiperSpeed = 60
            Case 4: High
                  (impossible; error condition)
      EndCase 'leverPosition
End enseLeverUp
```

Testing the senseLeverUp method will require checking each of the alternatives in the Case and nested Case statements. The tests for the “outer” Case statement cover the corresponding leverUp transitions in the StateChart. In a similar way, we must test the leverDown, dialUp, and dialDown methods. Once we know that the Dial and Lever components are correct, we can test the wiper component. Pseudocode for the test driver class will look something like this:

```
class testSenseLeverUp
      wiperSpeed
      leverPos
      dialPos
      testResult 'boolean
main()
      testCase = instantiate windshieldWiper(0, Off, 1)
      windshieldWiper.senseLeverUp()
      leverPos = windshieldWiper.getLeverPosition()
      If leverPos = Int
            Then testResult = Pass
            Else testResult = Fail
      EndIf
End 'main
```

There would be two other test cases, testing the transitions from INT to LOW, and LOW to HIGH. Next, we test the rest of the windshieldWiper class with the following pseudocode.

```
class test WindshieldWiper
      wiperSpeed
      leverPos
      dialPos
      testResult 'boolean
main()
      testCase = instantiate windshieldWiper(0, Off, 1)
      windshieldWiper.senseLeverUp()
      wiperSpeed = windshieldWiper.getWiperSpeed()
      If wiperSpeed = 6
            Then testResult = Pass
            Else testResult = Fail
      EndIf
End 'main
```

Two subtleties occur here. The instantiate windshieldWiper statement sets the preconditions of the test case. The test case in the pseudocode happens to correspond to the default entry states of the Dial and Lever components of the StateChart in Figure 15.8. Notice it is easy to force other preconditions. The second subtlety is more obscure. The wiper component of the StateChart has what we might call the tester's (or external) view of the class. In this view, the wiper default entries and transitions are caused by various "inState" propositions. The implementation, however, causes these transitions by using the set methods to change the values of the state variables.

The use case in Table 15.1 describes a typical scenario of windshield Lever and Dial events. The corresponding test cases are given in Table 15.2. This example represents the cusp between integration and system testing for object-oriented code.

We have the StateChart definition of the class behavior; therefore, we can use it to define our test cases in much the same way that we used finite state machines to identify system-level test cases. StateChart-based class testing supports reasonable test coverage metrics. Some obvious ones are

Table 15.1 Lever and Dial Use Case

Use case name	Normal usage
Use case ID	UC-1
Description	The windshield wiper is in the OFF position, and the Dial is at the 1 position; the user moves the lever to INT, and then moves the dial first to 2 and then to 3; the user then moves the lever to LOW; the user moves the lever to INT, and then to OFF.
Preconditions	1. The Lever is in the OFF position.
	2. The Dial is at the 1 position.
	3. The wiper speed is 0.
Event Sequence	
Input Events	Output Events
1. Move lever to INT	2. Wiper speed is 6
3. Move dial to 2	4. Wiper speed is 12
5. Move dial to 3	6. Wiper speed is 20
7. Move lever to LOW	8. Wiper speed is 30
9. Move lever to INT	10. Wiper speed is 20
11. Move lever to OFF	12. Wiper speed is 0
Postconditions	1. The Lever is in the OFF position.
	2. The Dial is at the 3 position.
	3. The wiper speed is 0.

Every event
Every state in a component
Every transition in a component
All pairs of interacting states (in different components)
Scenarios corresponding to customer-defined use cases

Table 15.2 contains test cases with the instantiate statements (to establish preconditions) and expected outputs for the "every transition in a component" coverage level for the Lever component.

Notice that the higher levels of coverage actually imply an intraclass integration of methods, which seem to contradict the idea of class-as-unit. The scenario coverage criterion is nearly identical to system-level testing. Here is a use case and the corresponding message sequences needed in a test class.

```
class testScenario
      wiperSpeed
      leverPos
```

```
        dialPos
        step1OK 'boolean
        step2OK 'boolean
        step3OK 'boolean
        step4OK 'boolean
        step5OK 'boolean
        step6OK 'boolean
main()
        testCase = instantiate windshieldWiper(0, Off, 1)
        windshieldWiper.senseLeverUp()
        wiperSpeed = windshieldWiper.getWiperSpeed()
        If wiperSpeed = 4
                Then step1OK = Pass
                Else step1OK = Fail
        EndIf

        windshieldWiper.senseDialUp()
        wiperSpeed = windshieldWiper.getWiperSpeed()
        If wiperSpeed = 6
                Then step2OK = Pass
                Else step2OK = Fail
        EndIf

        windshieldWiper.senseDialUp()
        wiperSpeed = windshieldWiper.getWiperSpeed()
        If wiperSpeed = 12
                Then step3OK = Pass
                Else step3OK = Fail
        EndIf

        windshieldWiper.senseLeverUp()
        wiperSpeed = windshieldWiper.getWiperSpeed()
        If wiperSpeed = 20
                Then step4OK = Pass
                Else step4OK = Fail
        EndIf

        windshieldWiper.senseLeverDown()
        wiperSpeed = windshieldWiper.getWiperSpeed()
        If wiperSpeed = 12
                Then step5OK = Pass
                Else step5OK = Fail
        EndIf

        windshieldWiper.senseLeverDown()
        wiperSpeed = windshieldWiper.getWiperSpeed()
        If wiperSpeed = 0
                Then step6OK = Pass
                Else step6OK = Fail
        EndIf
End 'main
```

Table 15.2 Test Cases for Lever Component

Test Case	*Preconditions (Instantiate Statement)*	*windshieldWiper Event*	*Expected leverPos*
1	windshieldWiper(0,Off,1)	senseLeverUp()	INT
2	windshieldWiper(0,Int,1)	senseLeverUp()	LOW
3	windshieldWiper(0,Low,1)	senseLeverUp()	HIGH
4	windshieldWiper(0,High,1)	senseLeverDown()	LOW
5	windshieldWiper(0,Low,1)	senseLeverDown()	INT
6	windshieldWiper(0,Int,1)	senseLeverDown()	OFF

15.4 Object-Oriented Integration Testing

Of the three main levels of software testing, integration testing is the least understood; this is true for both traditional and object-oriented software. As with traditional procedural software, object-oriented integration testing presumes complete unit-level testing. Both unit choices have implications for object-oriented integration testing. If the operation/method choice is taken, two levels of integration are required: one to integrate operations into a full class, and one to integrate the class with other classes. This should not be dismissed. The whole reason for the operation-as-unit choice is that the classes are very large, and several designers were involved.

Turning to the more common class-as-unit choice, once the unit testing is complete, two steps must occur: (1) if flattened classes were used, the original class hierarchy must be restored, and (2) if test methods were added, they must be removed.

Once we have our "integration test bed," we need to identify what needs to be tested. As we saw with traditional software integration, static and dynamic choices can be made. We can address the complexities introduced by polymorphism in a purely static way: test messages with respect to each polymorphic context. The dynamic view of object-oriented integration testing is more interesting.

15.4.1 UML Support for Integration Testing

In UML-defined, object-oriented software, collaboration and sequence diagrams are the basis for integration testing. Once this level is defined, integration-level details are added. A collaboration diagram shows (some of) the message traffic among classes. Figure 15.9 is a collaboration diagram for the ooCalendar application. A collaboration diagram is very analogous to the Call Graph we used in Chapter 13. As such, a collaboration diagram supports both the pairwise and neighborhood approaches to integration testing.

With pairwise integration, a unit (class) is tested in terms of separate "adjacent" classes that either send messages to or receive messages from the class being integrated. To the extent that the class sends/receives messages from other classes, the other classes must be expressed as stubs. All this extra effort makes pairwise integration of classes as undesirable as we saw pairwise integration of procedural units to be. On the basis of the collaboration diagram in Figure 15.9, we would have the following pairs of classes to integrate:

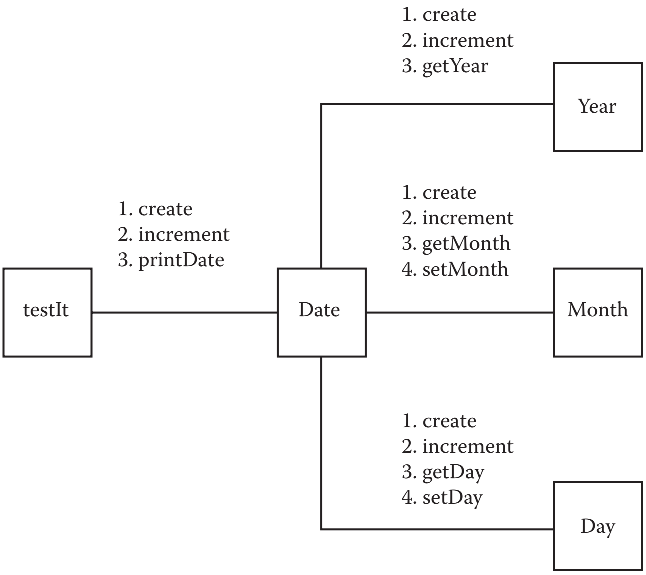


Figure 15.9 Collaboration diagram for ooCalendar.

testIt and Date, with stubs for Year, Month, and Day
Date and Year, with stubs for testIt, Month, and Day
Date and Month, with stubs for testIt, Year, and Day
Date and Day, with stubs for testIt, Month, and Year
Year and Month, with stubs for Date and Day
Month and Day, with stubs for Date and Year

One drawback to basing object-oriented integration testing on collaboration diagrams is that, at the class level, the behavior model of choice in UML is the StateChart. For class-level behavior, StateCharts are an excellent basis of test cases, and this is particularly appropriate for the class-as-unit choice. The problem, however, is that in general it is difficult to compose StateCharts to see behavior at a higher level (Regmi, 1999).

Neighborhood integration raises some very interesting questions from graph theory. Using the (undirected) graph in Figure 15.9, the neighborhood of Date is the entire graph, while the neighborhood of testIt is just Date. Mathematicians have identified various “centers” of a linear graph. One of them, for example, is the ultracenter, which minimizes the maximum distances to the other nodes in the graph. In terms of an integration order, we might picture the circular ripples caused by tossing a stone in calm water. We start with the ultracenter and the neighborhood of nodes one edge away, then add the nodes two edges away, and so on. Neighborhood integration of classes will certainly reduce the stub effort, but this will be at the expense of diagnostic precision. If a test case fails, we will have to look at more classes to find the fault.

A sequence diagram traces an execution-time path through a collaboration diagram. (In UML, a sequence diagram has two levels: at the system/use case level and at the class interaction level.) Thick, vertical lines represent either a class or an instance of a class, and the arrows are labeled with the messages sent by (instances of) the classes in their time order. The portion of the ooCalendar application that prints out the new date is shown as a sequence diagram in Figure 15.10.

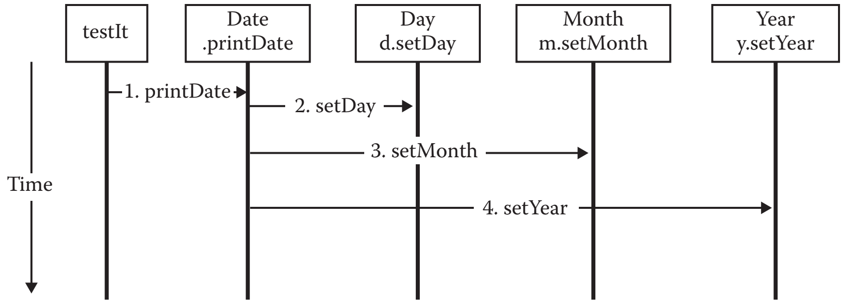


Figure 15.10 Sequence diagram for printDate.

To the extent that sequence diagrams are created, they are a reasonable basis for object-oriented integration testing. They are nearly equivalent to the object-oriented version of MM-paths (which we define in the next subsection). An actual test for this sequence diagram would have pseudocode similar to this:

```
1.      testDate
2.              d.setDay(27)
3.              m.setMonth(5)
4.              y.setYear(2013)
5.              Output (“Expected value is 5/27/2013”)
6.              testIt.printDate
7.              Output (“Actual output is...”)
8.      End testDate
```

Statements 2, 3, and 4 use the previously unit-tested methods to set the expected output in the classes to which messages are sent. As it stands, this test driver depends on a person to make a pass/fail judgment based on the printed output. We could put comparison logic into the testDriver class to make an internal comparison. This might be problematic in the sense that, if we made a mistake in the code tested, we might make the same mistake in the comparison logic.

Using collaboration diagrams or sequence diagrams as a basis for object-oriented integration testing is suboptimal. Collaboration diagrams force a pairwise approach to integration testing. Sequence diagrams are a little better, but somehow the integration tester needs all the sequence diagrams that pertain to a particular set of units to be integrated. A third, non-UML strategy is to use the call graph that we discussed in Chapter 13. Recall that nodes in a call graph can be either procedural units or object-oriented methods. For object-oriented integration, edges represent messages from one method to another. The remainder of the call graph–based integration strategies apply to object-oriented integration testing.

15.4.2 MM-Paths for Object-Oriented Software

Definition

An *object-oriented MM-path* is a sequence of method executions linked by messages.

When we spoke of MM-paths in traditional software, we used “message” to refer to the invocation among separate units (modules), and we spoke of module execution paths (module-level threads) instead of full modules. Here, we use the same acronym to refer to an alternating sequence of method executions separated by messages—thus, Method/Message Path. Just as in traditional software, methods may have several internal execution paths. We choose not to operate at that level of detail for object-oriented integration testing. An MM-path starts with a method and ends when it reaches a method that does not issue any messages of its own; this is the point of message quiescence. Figure 15.11 shows the extension of call graphs to object-oriented software. The classes, methods, and messages all refer to the ooNextDate pseudocode in Section 15.2. In

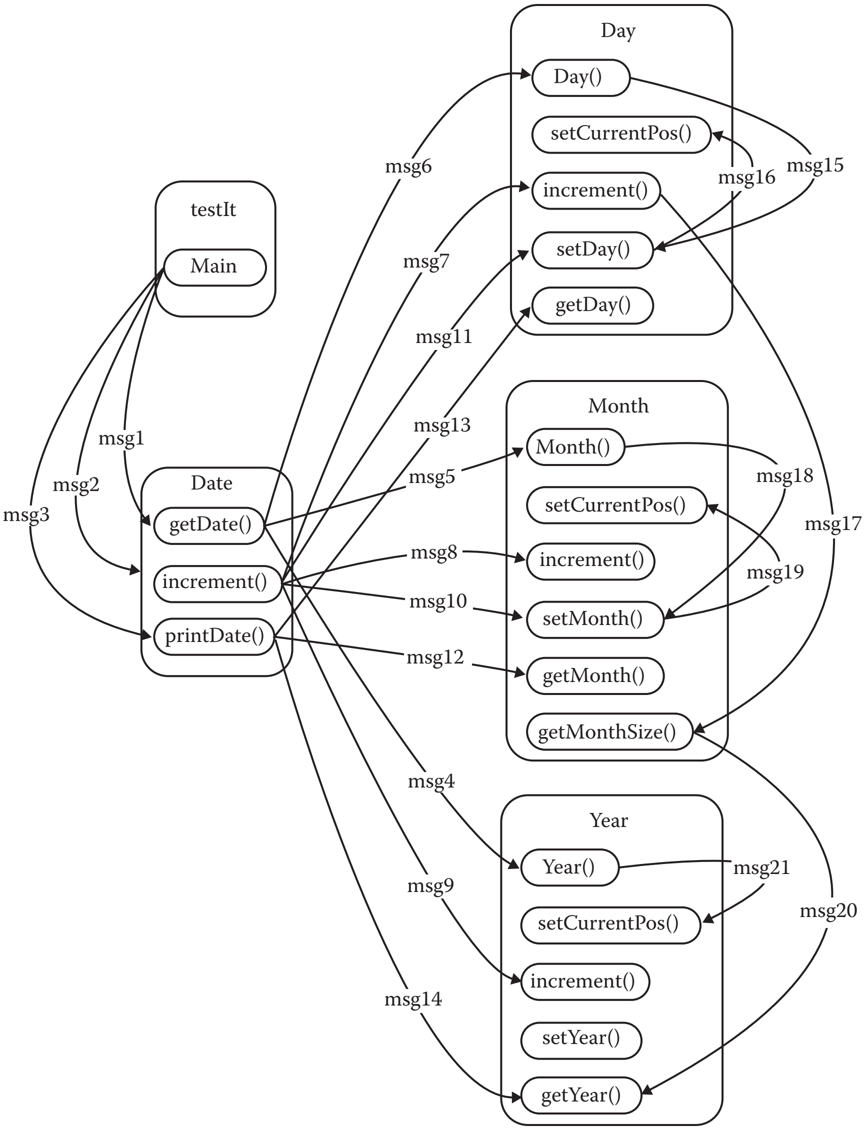

Day
Day()
setCurrentPos()
increment()
setDay()
getDay()
testIt
Main
Date
getDate()
increment()
printDate()
Month
Month()
setCurrentPos()
increment()
setMonth()
getMonth()
getMonthSize()
Year
Year()
setCurrentPos()
increment()
setYear()
getYear()
msg1
msg2
msg3
msg4
msg5
msg6
msg7
msg8
msg9
msg10
msg11
msg12
msg13
msg14
msg15
msg16
msg17
msg18
msg19
msg20
msg21

Figure 15.11 Potential message flow in ooNextDate.

one sense, this figure is a very detailed view of the collaboration graph; it shows all the potential message flows, very much like the discussion of intension versus extension we had for EDPNs in Chapter 14. With this view, we can understand object-oriented integration testing independently of the choices of units as operations or classes.

Here is a partial MM-path for the instantiate Date with the value January 3, 2013. The statement and message numbers are from the pseudocode in Section 15.2. This MM-path is shown in Figure 15.12.

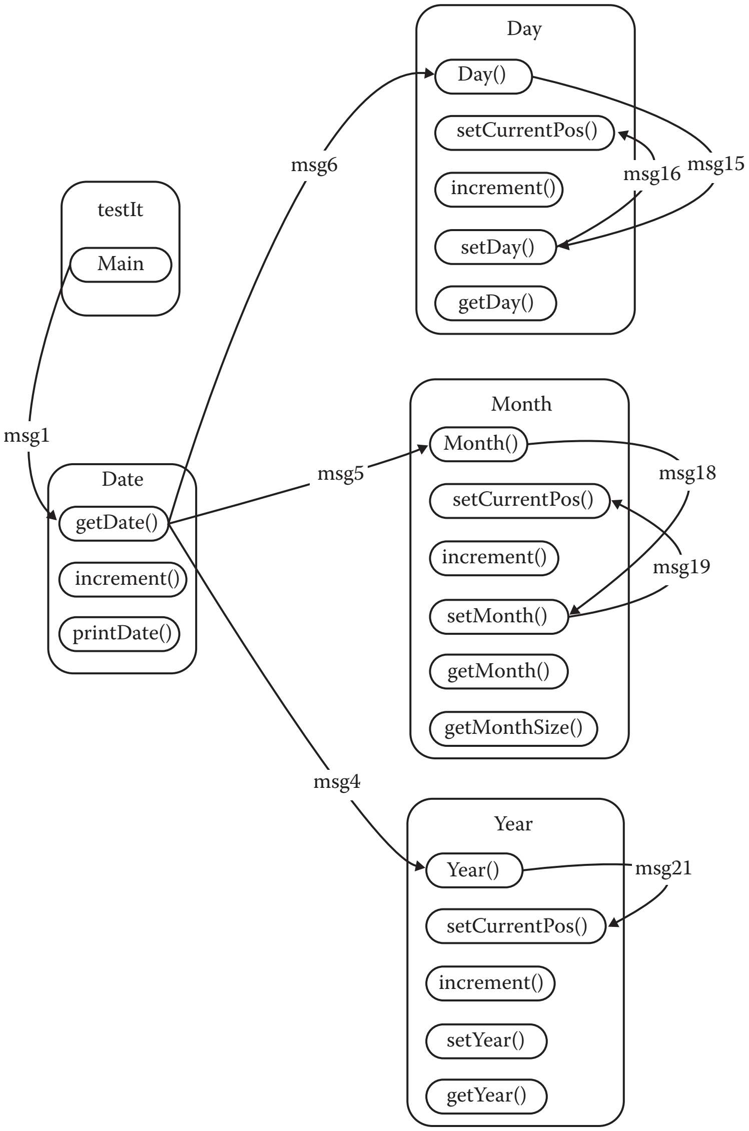

Day
Day()
setCurrentPos()
increment()
setDay()
getDay()
msg6
msg16
msg15
testIt
Main
msg1
Date
getDate()
increment()
printDate()
msg5
Month
Month()
setCurrentPos()
increment()
setMonth()
getMonth()
getMonthSize()
msg18
msg19
msg4
Year
Year()
setCurrentPos()
increment()
setYear()
getYear()
msg21

Figure 15.12 MM-path for January 3, 2013.

```
testIt<1>
       msg1
Date:testdate<4, 5>
       msg4
Year:y<53, 54>
       msg21
Year:y.setCurrentPos<a, b>
       (return to Year.y)
       (return to Date:testdate)
Date:testdate<6>
       msg5
Month:m<34, 35>
       msg18
Month:m.setMonth<36, 37>
       msg19
Month:m.setCurrentPos<a, b>
       (return to Month:m.setMonth)
       (return to Month:m)
       (return to Date:testdate)
Date:testdate<7>
       msg6
Day:d<21, 22>
       msg15
Day:d.setDay<23, 24>
       msg16
Day:d.setCurrentPos<a, b>
       (return to Day:d.setDay)
Day:d.setDay<25)
       (return to Day:d)
       (return to Date:testdate)
```

Here is a more interesting MM-path, for the instantiated date April 30, 2013 (Figure 15.13).

```
testIt<2>
       msg2
Date:testdate.increment<8,9>
       msg7
Day:d.increment<28, 29>                     'now Day.d.currentPos = 31
       msg17
Month:m.getMonthSize<41, 42>
       msg20
Year:y.isleap<60, 61, 63, 64>                         'not a leap year
       (return to Month:m.getMonthSize)
Month:m.getMonthSize<44, 45, 46>               'returns month size = 30
       (return to Day:d.increment)
Day:d.increment<32, 33>                                 'returns false
       (return to Date:testdate.increment)
Date:testdate.increment<10, 11>
       msg8
Month:m.increment<47, 48, 49, 51, 52>                    'returns true
       (return to Date:testdate.increment)
Date:testdate.increment<15, 16>
       msg11
```

```
Day:d.setDay<23, 24, 25>                        'now day is 1, month is 5
       (return to Date:testdate.increment)
Date:testdate.increment<17, 18>
       (return to testIt)
```

Having a directed graph formulation puts us in a position to be analytical about choosing MM-path-based integration test cases. First, we might ask how many test cases will be needed. The directed graph in Figure 15.11 has a cyclomatic complexity of 23 (the return edges of each message are not shown, but they must be counted). While we could certainly find that many

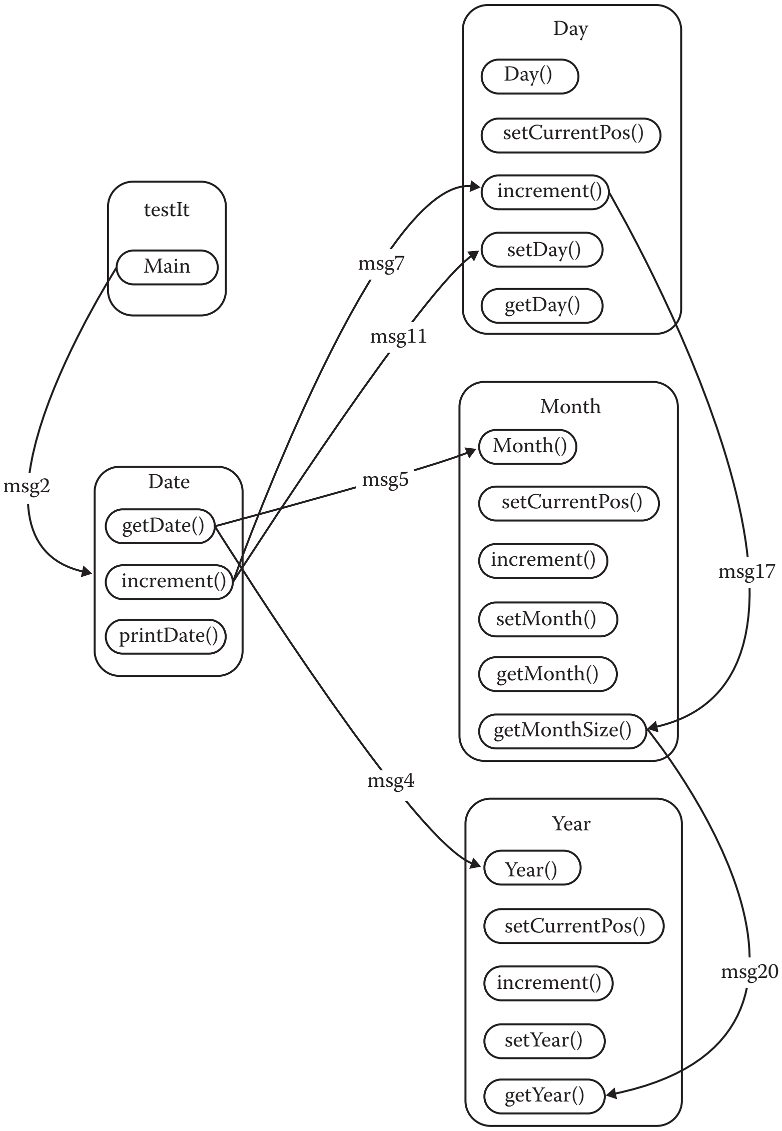

Day
Day()
setCurrentPos()
increment()
setDay()
getDay()
testIt
Main
msg7
msg11
Month
Month()
setCurrentPos()
increment()
setMonth()
getMonth()
getMonthSize()
msg2
Date
getDate()
increment()
printDate()
msg5
msg17
msg4
Year
Year()
setCurrentPos()
increment()
setYear()
getYear()
msg20

Figure 15.13 MM-path for April 30, 2013.

basis paths, there is no need to do this because a single MM-path will cover many of these paths, and many more are logically infeasible. A lower limit would be three test cases; the MM-paths beginning with statements 1, 2, and 3 in the testIt pseudocode. This might not be sufficient because, for example, if we choose an "easy" date (like January 3, 2013), the messages involving isLeap and setMonth will not occur. Just as we saw with unit-level testing of procedural code, as a minimum, we need a set of MM-paths that covers every message. The 13 decision table–based test cases we identified for NextDate in Chapter 8 constitute a thorough set of integration test cases for the ooCalendar application. This is a point where the code-based view of object-oriented integration testing gives insights that we cannot get from the specification-based view. We can look for MM-paths to make sure that every message (edge) in the graph of Figure 15.11 is traversed.

15.4.3 A Framework for Object-Oriented Data Flow Testing

MM-paths were defined to serve as integration testing analogs of DD-paths. As we saw for procedural software, DD-path testing is often insufficient; and in such cases, data flow testing is more appropriate. The same holds for integration testing of object-oriented software; if anything, the need is greater for two reasons: (1) data can get values from inheritance tree and (2) data can be defined at various stages of message passing.

Program graphs formed the basis for describing and analyzing data flow testing for procedural code. The complexities of object-oriented software exceed the expressive capabilities of directed (program) graphs. This subsection presents an expressive framework within which data flow testing questions for object-oriented software can be described and analyzed.

15.4.3.1 Event-/Message-Driven Petri Nets

Event-driven Petri nets were defined in Chapter 4; we extend them here to express the message communication among objects. Figure 15.14 shows the notational symbols used in an event- and message-driven Petri net (EMDPN). The fused triangle shape for messages is intended to convey that a message is an output of the sending method and an input to the destination method.

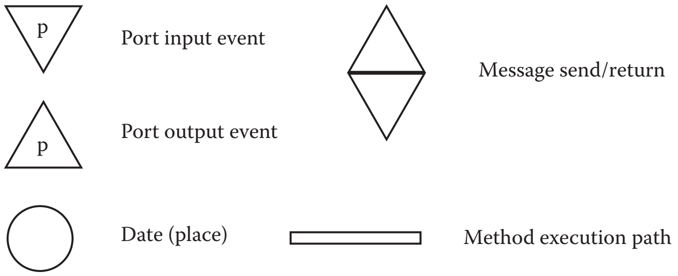


Figure 15.14 Symbols for event-/message-driven Petri nets (E/MDPN).

Definition

An *event- and message-driven Petri net (EMDPN)* is a quadripartite directed graph (P, D, M, S, In, Out) composed of four sets of nodes, P, D, M, and S, and two mappings, In and Out, where

P is a set of port events
D is a set of data places
M is a set of message places
S is a set of transitions
In is a set of ordered pairs from $(P \cup D \cup M) \times S$
Out is a set of ordered pairs from $S \times (P \cup D \cup M)$

We retain the port input and output events because these will certainly occur in event-driven, object-oriented applications. Obviously, we still need data places, and we will interpret Petri net transitions as method execution paths. The new symbol is intended to capture the essence of interobject messages:

They are an output of a method execution path in the sending object.
They are an input to a method execution path in the receiving object.
The return is a very subtle output of a method execution path in the receiving object.
The return is an input to a method execution path in the sending object.

Figure 15.15 shows the only way that the new message place can appear in an EMDPN.

The EMDPN structure, because it is a directed graph, provides the needed framework for data flow analysis of object-oriented software. Recall that data flow analysis for procedural code centers on nodes where values are defined and used. In the EMDPN framework, data is represented by a data place, and values are defined and used in method execution paths. A data place can be either an input to or an output of a method execution path; therefore, we can now represent the define/use paths (du-path) in a way very similar to that for procedural code. Even though four types of nodes exist, we still have paths among them; so we simply ignore the types of nodes in a du-path and focus only on the connectivity.

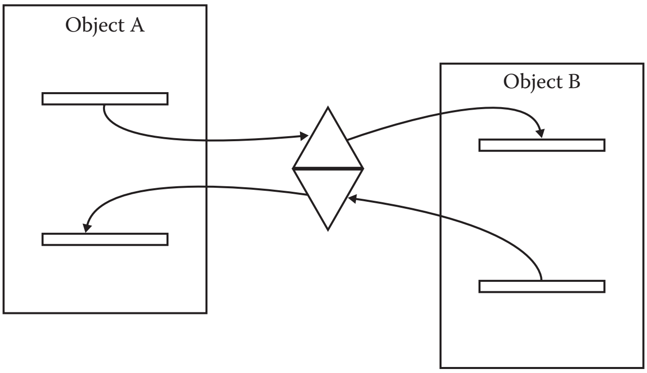


Figure 15.15 Message connection between objects.

15.4.3.2 Inheritance-Induced Data Flow

Consider an inheritance tree in which the value of a data item is defined; in that tree, consider a chain that begins with a data place where the value is defined, and ends at the "bottom" of the tree. That chain will be an alternating sequence of data places and degenerate method execution paths, in which the method execution paths implement the inheritance mechanism of the object-oriented language. This framework therefore supports several forms of inheritance: single, multiple, and selective multiple. The EMDPN that expresses inheritance is composed only of data places and method execution paths, as shown in Figure 15.16.

15.4.3.3 Message-Induced Data Flow

The EMDPN in Figure 15.17 shows the message communication among three objects. As an example of a define/use path, suppose mep3 is a Define node for a data item that is passed on to mep5, modified in mep6, and finally used in the Use node mep2. We can identify these two du-paths:

du1 = <mep3, msg2, mep5, d6, mep6, return(msg2), mep4, return(msg1), mep2>
du2 = <mep6, return(msg2), mep4, return(msg1), mep2>

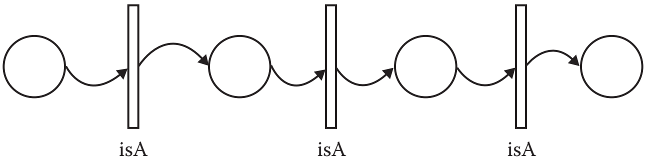


Figure 15.16 Data flow from inheritance.

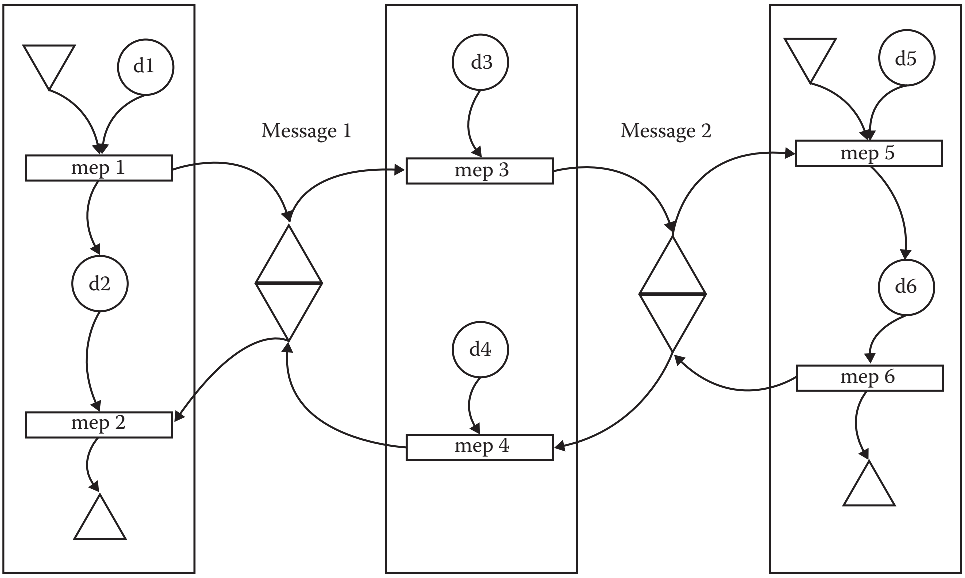


Figure 15.17 Data flow from message passing.

In this example, du2 is definition clear; du1 is not. Although we do not develop data flow testing for object-oriented software here, this formulation will support that endeavor.

15.4.3.4 Slices?

It is tempting to assert that this formulation also supports slices in object-oriented software. The fundamentals are there, so we could go through the graph theory motions. Recall that the more desirable form of a slice is one that is executable. This appears to be a real stretch, and without it, such slices are interesting only as a desk-checking approach to fault location.

15.5 Object-Oriented System Testing

System testing is (or should be) independent of system implementation. A system tester does not really need to know if the implementation is in procedural or object-oriented code. As we saw in Chapter 14, the primitives of system testing are port input and output events, and we know how to express system-level threads as EDPNs. The issue is how to identify threads to be used as test cases. In Chapter 14, we used the requirements specification models, particularly the behavioral models, as the basis of thread test case identification. We also discussed pseudostructural coverage metrics in terms of the underlying behavioral models. In a sense, this chapter is very object oriented: we inherit many ideas from system testing of traditional software. The only real difference in this chapter is that we presume the system has been defined and refined with the UML. One emphasis, then, is finding system-level thread test cases from standard UML models. At the system level, as we saw in Chapter 14, a UML description is composed of various levels of use cases, a use case diagram, class definitions, and class diagrams.

15.5.1 Currency Converter UML Description

We will use the currency converter application as an example for system testing. Because the UML from the Object Management Group is now widely accepted, we will use a rather complete UML description, in the style of Larman (1997). The terminology and UML content generally follow the Larman UML style, with the addition of pre- and postconditions in expanded essential use cases (EEUCs).

15.5.1.1 Problem Statement

The currency converter application converts U.S. dollars to any of four currencies: Brazilian reals, Canadian dollars, European Community euros, and Japanese yen. The user can revise inputs and perform repeated currency conversion.

15.5.1.2 System Functions

In the first step, sometimes called project inception, the customer/user describes the application in very general terms. This might take the form of "user stories," which are precursors to use cases. From these, three types of system functions are identified: evident, hidden, and frill. Evident functions are the obvious ones. Hidden functions might not be discovered immediately, and frills are the "bells and whistles" that so often occur. Table 15.3 lists the system functions for the currency converter application.

Table 15.3 System Functions for Currency Converter Application

Reference No.	*Function*	*Category*
R1	Start application	Evident
R2	End application	Evident
R3	Input US dollar amount	Evident
R4	Select country	Evident
R5	Perform conversion calculation	Evident
R6	Clear user inputs and program outputs	Evident
R7	Maintain exclusive–or relationship among countries	Hidden
R8	Display country flag images	Frill

15.5.1.3 Presentation Layer

Pictures are still worth a thousand words. The third step in Larman's approach is to sketch the user interface; our version is in Figure 15.18. This much information can support a customer walk-through to demonstrate that the system functions identified can be supported by the interface.

15.5.1.4 High-Level Use Cases

The use case development begins with a very high-level view. Notice, as the succeeding levels of use cases are elaborated, much of the early information is retained. It is convenient to have a short, structured naming convention for the various levels of use cases. Here, for example, HLUC refers to high-level use case (where would we be without acronyms?). Very few details are provided in a high-level use case; they are insufficient for test case identification. The main point of high-level use cases is that they capture a narrative description of something that happens in the system to be built.

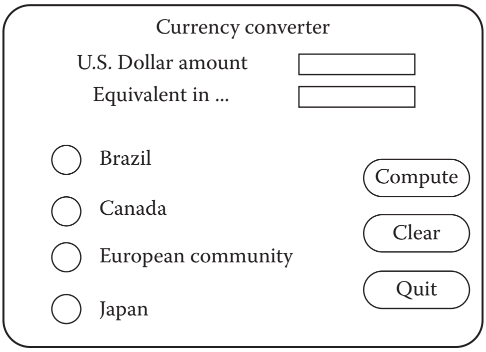


Figure 15.18 Currency converter GUI.

HLUC 1	Start application.
Description	The user starts the currency conversion application in Windows®.

HLUC 2	End application.
Description	The user ends the currency conversion application in Windows.

HLUC 3	Convert dollars.
Description	The user inputs a US dollar amount and selects a country; the application computes and displays the equivalent in the currency of the selected country.

HLUC 4	Revise inputs.
Description	The user resets inputs to begin a new transaction.

HLUC 5	Repeated conversions, same dollar amount.
Description	The user inputs a US dollar amount and selects a country; the application computes and displays the equivalent in the currency of the selected country.

HLUC 6	Revise inputs.
Description	A US dollar amount has been entered OR a country has been selected.

HLUC 7	Abnormal case: no country selected.
Description	User enters a dollar amount and clicks on the Compute button without selecting a country.

HLUC 8	Abnormal case: no dollar amount entered.
Description	User selects a country and clicks on the Compute button without entering a dollar amount.

HLUC 9	Abnormal case: no dollar amount entered and no country selected.
Description	User clicks on the Compute button without entering a dollar amount and without selecting a country.

15.5.1.5 Essential Use Cases

Essential use cases add "actor" and "system" events to a high-level use case. Actors in UML are sources of system-level inputs (i.e., port input events). Actors can be people, devices, adjacent systems, or abstractions such as time. Since the only actor is the User, that part of an essential use case is omitted. The numbering of actor actions (port input events) and system responses (port output events) shows their approximate sequences in time. In EUC3, for example, human observers cannot detect the sequence of system responses 4 and 5; they would appear to be simultaneous. Also, because some of the essential use cases are obvious, they are deleted; however, the numbering still refers to the high-level use cases.

EUC-1	Start application
Description	The user starts the currency conversion application in Windows.
Event Sequence	
Input Events	Output Events
1. The user starts the application, either with a Run … command or by double clicking the application icon.	
	2. The currency conversion application GUI appears on the monitor and is ready for user input.

EUC-3	Convert dollars
Description	The user inputs a US dollar amount and selects a country; the application computes and displays the equivalent in the currency of the selected country.
Event Sequence	
Input Events	Output Events
1. The user enters a dollar amount.	
	2. The dollar amount is displayed on the GUI.
3. The user selects a country.	
	4. The name of the country's currency is displayed.
	5. The flag of the country is displayed.
6. The user requests a conversion calculation.	
	7. The equivalent currency amount is displayed.

EUC-4	Revise inputs
Description	The user resets inputs to begin a new transaction.
Event Sequence	
Input Events	Output Events
1. The user enters a dollar amount.	
	2. The dollar amount is displayed on the GUI.
3. The user selects a country.	
	4. The name of the country's currency is displayed.
	5. The flag of the country is displayed.
6. The user cancels the inputs.	
	7. The name of the country's currency is removed.
	8. The flag of the country is no longer visible.

EUC-7	Abnormal case: no country selected
Description	The user enters a dollar amount and clicks on the cmdCompute button without selecting a country.
Event Sequence	
Input Events	Output Events
1. The user enters a dollar amount.	2. The dollar amount is displayed on the GUI.
3. User clicks the Compute button.	4. A message box appears with the caption "must select a country."
5. The user closes the message box.	6. The message box is no longer visible.
	7. The flag of the country is no longer visible.

15.5.1.6 Detailed GUI Definition

Once a set of essential use cases has been identified, the graphical user interface (GUI) is fleshed out with design-level detail. Here, we implement the currency converter in Visual Basic and follow a recommended naming convention for Visual Basic controls as shown in Figure 15.19. (For readers not familiar with Visual Basic, this design uses four types of controls: text boxes for input, labels for output, option buttons to indicate choices, and command buttons to control the execution of the application.) These controls will be referred to in the EEUCs. Again for space reasons, only selected EEUCs will be included.

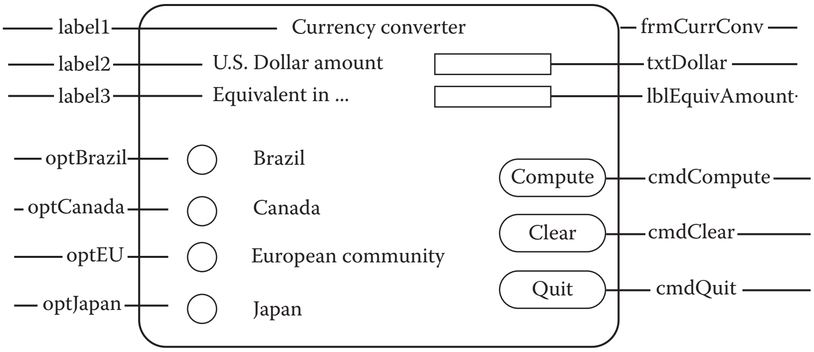

label1
Currency converter
frmCurrConv
label2
U.S. Dollar amount
txtDollar
label3
Equivalent in ...
lblEquivAmount
optBrazil
Brazil
optCanada
Canada
optEU
European community
optJapan
Japan
Compute
cmdCompute
Clear
cmdClear
Quit
cmdQuit

Figure 15.19 Detailed GUI definition.

15.5.1.7 Expanded Essential Use Cases

The EEUCs are the penultimate refinement of the high-level use cases. Here, we add pre- and postcondition information (not part of the Larman flavor), information about alternative sequences of events, and a cross-reference to the system functions identified very early in the process. The other expansion is that more use cases are identified and added at this point. This is a normal part of any specification and design process: more detailed views provide more detailed insights. Note that the numeric tracing across levels of use cases is lost at this point.

The pre- and postconditions warrant some additional comment. We are only interested in conditions that directly pertain to the EEUC defined, as in the Chapter 14 discussion of well-formed use cases. We could always add preconditions such as "power is on," and "computer is running under Windows"; however, if these are not used, they are not added to the preconditions. Similar comments apply to postconditions.

EEUC-1	Start application
Description	The user starts the currency conversion application in Windows.
Preconditions	Currency conversion application in (disk) storage
Event Sequence	
Input Events	Output Events
1. User double-clicks currency conversion application icon	2. frmCurrConv appears on screen
Postconditions	1. Currency conversion application is in memory 2. txtDollar has focus

EEUC-3	Normal usage (dollar amount entered first)
Description	The user inputs a US dollar amount and selects a country; the application computes and displays the equivalent in the currency of the selected country.
Preconditions	txtDollar has focus
Event Sequence	
Input Events	Output Events
1. User enters US dollar amount on keyboard	2. Dollar amount appears in txtDollar
3. User clicks on a country button	4. Country currency name appears in lblEquiv
5. User clicks cmdCompute button	6. Computed equivalent amount appears in lblEqAmount
Postconditions	cmdClear has focus

EEUC-4	Repeated conversions, same country
Description	The user inputs a US dollar amount and selects a country; the application computes and displays the equivalent in the currency of the selected country.
Preconditions	txtDollar has focus
Event Sequence	
Input Events	Output Events
1. User enters US dollar amount on keyboard	2. Dollar amount appears in txtDollar
3. User clicks on a country button	4. Country currency name appears in lblEquiv
5. User clicks cmdCompute button	6. Computed equivalent amount appears in lblEqAmount
7. User clicks on txtDollar	8. txtDollar has focus
9. User enters different US dollar amount on keyboard	10. Dollar amount appears in txtDollar
11. User clicks cmdCompute button	12. Computed equivalent amount appears in lblEqAmount
Postconditions	cmdClear has focus

15.5.1.8 Real Use Cases

In Larman's terms, real use cases are only slightly different from the EEUCs. Phrases such as "enter a US dollar amount" must be replaced by the more specific "enter 125 in txtDollar." Similarly, "select a country" would be replaced by "click on the optBrazil button." In the interest of space (and reduced reader boredom), real use cases are omitted. Note that system-level test cases could be mechanically derived from real use cases.

15.5.2 UML-Based System Testing

Our formulation lets us be very specific about system-level testing; there are at least four identifiable levels with corresponding coverage metrics for GUI applications. The first level is to test the system functions given as the first step in Larman's UML approach (see Table 15.3). These are cross-referenced in the extended essential use cases, so we can easily build an incidence matrix such as Table 15.4.

Examining the incidence matrix, we can see several possible ways to cover the seven system functions. One way would be to derive test cases from real use cases that correspond to extended essential use cases 1, 2, 5, and 6. These will need to be real use cases as opposed to the EEUCs. The difference is that specific countries and dollar values are used, instead of the higher-level statements such as "click on a country button" and enter a dollar amount. Deriving system test cases from real use cases is mechanical: the use case preconditions are the test case preconditions, and the sequences of actor actions and system responses map directly into sequences of user input events and system output events. The set of extended essential use cases 1, 2, 5, and 6 is a nice example of a set of regression test cases; taken together, they cover all seven system functions.

The second level is to develop test cases from all of the real use cases. Assuming that the customer approved of the original EEUCs, this is the minimally acceptable level of system test

Table 15.4 Incidence Matrix of Use Cases with System Functions

EEUC	*R1*	*R2*	*R3*	*R4*	*R5*	*R6*	*R7*
1	×	—	—	—	—	—	—
2	—	×	—	—	—	—	—
3	—	—	×	×	×	—	—
4	—	—	×	×	×	—	—
5	—	—	×	×	×	—	×
6	—	—	×	×	—	×	×
7	—	—	×	—	×	—	—
8	—	—	×	—	×	—	—
9	—	—	—	—	×	—	—

coverage. Here is a sample system-level test case derived from the real use case based on extended essential use case EEUC3. (Assume the exchange rate for 1 euro is US$1.31.)

System test case 3	Normal usage (dollar amount entered first)
Test performed by	Paul Jorgensen
Preconditions	txtDollar has focus
Event Sequence	
Input Events	Output Events
1. Enters 10 on the keyboard	
	2. Observe 10 appears in txtDollar
3. Click on optEU button	
	4. Observe "euros" appears in label3
5. Clicks cmdCompute button	
	6. Observe "7.60" appears in lblEquivAmount
Postconditions	cmdClear has focus
Test result	Pass (on first attempt)
Date run	May 27, 2013

The third level is to derive test cases from the finite state machines derived from a finite state machine description of the external appearance of the GUI, as we did in Chapter 14 for the SATM system. The fourth level is to derive test cases from state-based event tables; this makes sense for states with a high outdegree of transitions. This is an "exhaustive" level because it exercises every possible event for each state. It is not truly exhaustive, however, because we have not tested all sequences of events across states. The other problem is that it is an extremely detailed view of system testing that is likely very redundant with integration- and even unit-level test cases.

15.5.3 StateChart-Based System Testing

A caveat is required here. StateCharts are a fine basis for system testing. The problem is that StateCharts are prescribed to be at the class level in UML. There is no easy way to compose StateCharts of several classes to get a system-level StateChart (Regmi, 1999). A possible workaround is to translate each class-level StateChart into a set of EDPNs, and then compose the EDPNs.

EXERCISE

1. The ooCalendar problem can be extended in several ways. One extension is to add an astrological content: each zodiac sign has a name and a beginning date (usually the 21st of a month). Add attributes and methods to the month class so that testIt can find the zodiac sign for a given date.

References

Binder, R.V., The free approach for testing use cases, threads, and relations, *Object*, Vol. 6, No. 2, February 1996, pp. 73–75.

Jorgensen, P.C., *Modeling Software Behavior: A Craftsman's Approach*, CRC Press, New York, 2009.

Larman, C., *Applying UML and Patterns*, Prentice-Hall, Upper Saddle River, New Jersey, 1997.

Regmi, D.R., *Object-Oriented Software Construction Based on State Charts*, Grand Valley State University Master's project, Allendale, MI, 1999.

Chapter 16

Software Complexity

Most discussions of software complexity focus on two main models—cyclomatic (or decisional) complexity, and textual complexity as measured by the Halstead metrics. Both approaches are commonly used at the unit level; however, both can also be used at the integration and system levels. This chapter takes a closer look at software complexity at all three levels—unit, integration, and system. At the unit level, the basic cyclomatic complexity model (also known as McCabe complexity) is extended in two ways. Integration-level complexity applies cyclomatic complexity to a directed graph in which units are nodes and edges represent either object-oriented messages or procedural calls. After discussing the complexities due to object-oriented practice, system-level complexity is expressed in terms of an incidence matrix that relates the *is* and *does* views of a software system.

Software complexity is usually analyzed as a static (i.e., compile-time) property of source code, not an execution-time property. The approaches discussed here are derived either directly from source code, or possibly from design- and specification-level models. Why worry about software complexity? It has the most direct bearing on the extent of required software testing, but also, it is an indicator of difficulty in software maintenance, particularly program comprehension. As software complexity increases, development effort also increases, although this is a little deceptive since much of the analysis is based on existing code (too late!). Finally, an awareness of software complexity may lead to improved programming practices, and even better design techniques.

16.1 Unit-Level Complexity

The description of unit-level complexity begins with the notion of a program graph from Chapter 8 (Path Testing). Recall that for a program written in an imperative programming language, its program graph is a directed graph in which nodes are either entire statements or fragments of a statement, and edges represent flow of control (there is an edge from node *i* to node *j* if and only if the statement [or statement fragment] corresponding to node *j* can be executed immediately after the statement or statement fragment corresponding to node *i*). The program graph represents the control flow structure of the source code, and this leads to the usual definition of cyclomatic complexity.

16.1.1 Cyclomatic Complexity

Definition: In a strongly connected directed graph G, its *cyclomatic complexity*, denoted by $V(G)$, is given by $V(G) = e - n + p$, where

e is the number of edges in G
n is the number of nodes in G
p is the number of connected regions in G

In code that conforms to structured programming (single entry, single exit), we always have $p = 1$. There is some confusion in the literature about the formula for $V(G)$. There are two formulas commonly seen:

(1) $V(G) = e - n + p$
(2) $V(G) = e - n + 2p$

Equation (1) refers to a directed graph G that is strongly connected, that is, for any two nodes n_j and n_k of G, there is a path from n_j to n_k, and a path from n_k to n_j. Since the program graph of a structured program has a single-entry node and a single-exit node, the graph is not quite strongly connected because there is no path from the sink node to the source node. The usual way to apply the formula is to add an edge from the sink node to the source node. If an edge is added, Equation (1) applies; otherwise, equation (2) applies. With this definition, and given a program graph, the cyclomatic complexity is determined by simply counting the nodes and edges, and applying equation (2). This is fine for small programs, but what about a program graph such as the one in Figure 16.1? Even for program graphs of this size, counting nodes and edges is tedious. For that matter, drawing the program graph is also tedious. Fortunately, there is a more elegant way, based on an insight from directed graph theory. We next develop two shortcuts.

16.1.1.1 "Cattle Pens" and Cyclomatic Complexity

Cyclomatic complexity refers to the number of independent cycles in a strongly connected directed graph. When drawn in the usual way (as in Figure 16.1), these cycles are easily identified visually, and this can be done for simple programs. Rather than count all the nodes and edges in a larger graph, we can imagine nodes to be fence posts, and edges to be fencing used in a cattle pen. Then the number of "cattle pens" can be counted visually. (The more esoteric term is "enclosed regions," which the topologists prefer.) In the program graph in Figure 16.1, there are 37 edges and 31 nodes. Since the graph is not strongly connected, equation (2) applies, and $V(G) = 37 - 31 + 2 = 8$. The eight "cattle pens" are also numbered (notice that one pen is "outside" all the others). Drawing the directed graph to identify cattle pens is still tedious. Again, there is a more elegant way, based on more definitions from graph theory.

16.1.1.2 Node Outdegrees and Cyclomatic Complexity

As we saw in Chapter 4, the indegree of a node in a directed graph is the number of edges that terminate on the node. Similarly, the outdegree of a node in a directed graph is the number of edges that originate at the node. These are commonly denoted for node n as inDeg(n) and outDeg(n). We need another definition to replace the thinking that went into the cattle pen approach.

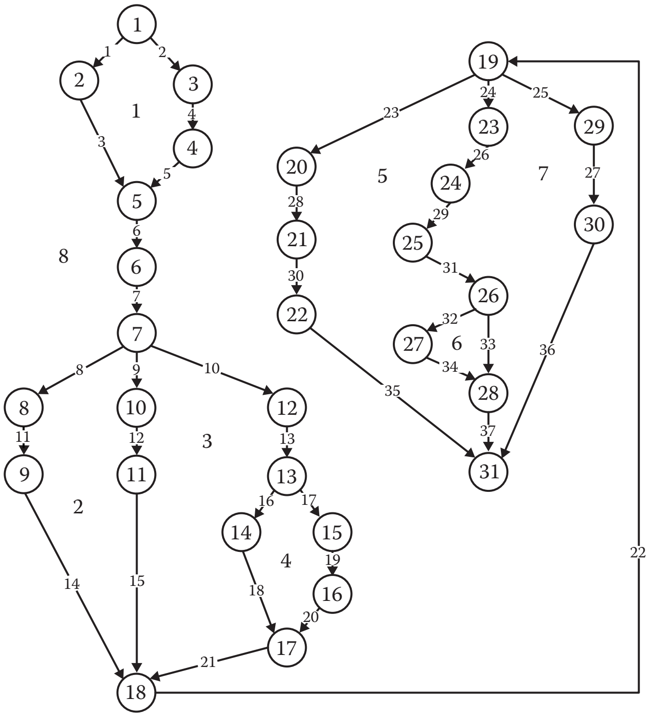


Figure 16.1 Mildly complex program graph.

Definition

The *reduced outdegree of node n in a directed graph* is one less than the outdegree of *n*.

Denote the reduced outdegree of node *n* as reducedOut(*n*); then we can write

$$\text{reducedOut}(n) = \text{outDeg}(n) - 1$$

We use the reduced outdegree of nodes in a program graph to compute its cyclomatic complexity. Notice that a cattle pen "begins" with a node with outDeg > = 2. Table 16.1 shows the nodes in Figure 16.1 that satisfy this observation.

Table 16.1 Reduced Outdegrees in Figure 16.1

Node	*outDeg*	*reducedOut*
1	2	1
7	3	2
13	2	1
19	3	2
26	2	1
	Total =	7

The sum of the reduced outdegrees is the number of cattle pens, but this does not count the "outside" cattle pen, which makes 8—the cyclomatic complexity of the directed graph. The outdegrees can be determined from the source code, eliminating the need to draw the directed graph, and perform the other tedious steps. As a guideline, a simple loop determines a cattle pen, as do the If, Then and If, Then, Else statements. Switch (Case) statements with *k* alternatives determine *k* – 1 cattle pens. So now, finding cyclomatic complexity is reduced to determining the reduced outdegrees of all decision-making statements in the source code. We can state this as a formal theorem (without proof).

Theorem: Given a directed graph *G* of *n* nodes, the cyclomatic complexity *V*(*G*) of *G* is given by the sum of the reduced outdegrees of the nodes of *G* plus 1, that is

$$V(G) = 1 + \left(\sum_{i=1}^{n} \ldots \text{reducedOut}(i) \right)$$

16.1.1.3 Decisional Complexity

Cyclomatic complexity is a start, but it is an oversimplification. Why? Because all decision-making statements are not equal—compound conditions add complexity. Consider the following code fragment from a program that computes the type of triangle formed by three integers, `a`, `b`, and `c`. The fragment applies the triangle inequality that requires that each side of a triangle is strictly less than the sum of the other two sides.

```
If (a < b + c) AND (b < a + c) AND (c < a + b)
    Then IsATriangle = True
    Else IsATriangle = False
Endif
```

The program graph of this fragment is very simple—it has a cyclomatic complexity of 2. From a software testing standpoint, we would apply multiple condition testing, or we could rewrite the fragment as follows, with the resulting cyclomatic complexity of 4:

```
If (a < b + c)
    Then If (b < a + c)
          Then   If (c < a + b)
                        Then IsATriangle = True
                        Else IsATriangle = False
                 EndIf '(c < a + b)
          Else IsATriangle = False
         EndIf '(b < a + c)
    Else IsATriangle = False
EndIf '(a < b + c)
```

The program graphs of these fragments are in Figure 16.2. The added complexity of compound conditions cannot be determined from a program graph—it must be derived from the source code. Doing a full multiple conditional testing analysis for a compound condition entails making a truth table in which the simple conditions are considered as individual propositions, and then finding the truth table of the compound expression. For now, we choose to simplify this and just define the added complexity of compound conditions to be one less than the number of simple conditions in the expression. Why one less? The compound condition creates a unit of cyclomatic complexity, so this avoids "double counting."

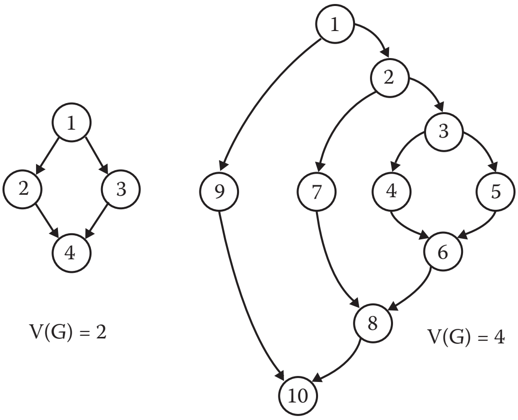


Figure 16.2 Program graphs of two equivalent program fragments.

16.1.2 Computational Complexity

Thus far, we have focused on what might be called control complexity, or maybe decisional complexity—basically looking at the edges leaving nodes in a program graph. But what about the nodes themselves? Just as with decisions, all nodes are not "created equal." To explore this, we use the definitions of DD-path and DD-path graph made in Chapter 8.

Recall that DD-path execution is like a sequence of dominoes: once the first statement executes, every statement in the DD-path executes, until the next decision point is reached. At this point, we can begin to think about the length of a DD-path. Since a DD-path contains no internal decision-making statements for any program *P*, the cyclomatic complexity of *P* equals the cyclomatic complexity of the DD-path graph of *P*. Our problem is now reduced to considering the computational complexity of a DD-path, and this is where the Halstead metrics are useful.

16.1.2.1 Halstead's Metrics

For a given program (DD-path), consider the operators and operands in the program code. Operators include the usual arithmetic and logical operators, as well as built-in functions such as square root. Operands are identifiers. The Halstead metrics are based on the following quantities, derived from the source code of the program (DD-path):

- The number of distinct operators, n_1
- The number of distinct operands, n_2
- The total number of operators, N_1
- The total number of operands, N_2

On the basis of these, Halstead defines

- Program length as $N = N_1 + N_2$
- Program vocabulary as $n = n_1 + n_2$
- Program volume as $V = N\log_2(n)$
- Program difficulty as $D = (n_1 N_2)/2n_2$

Of these, the formula for program volume seems to make the most sense, but we could choose to use program difficulty, as this seems to be linguistically related to our goal of describing software complexity.

16.1.2.2 Example: Day of Week with Zeller's Congruence

Here we compare two slightly different implementations of Zeller's congruence, which determines the day of the week of a given date. The inputs d, m, y are day, month, and year, respectively. Tables 16.2 and 16.3 show the values of the inputs to Halstead's metrics.

First implementation

```
if (m < 3) {
       m += 12;
              y -= 1;
            }
int k = y% 100;
int j = y/100;
int dayOfWeek = ((d + (((m + 1) * 26)/10) + k + (k/4) + (j/4)) + (5 * j))%7;
```

Table 16.2 Halstead's Metrics for First Implementation

Operator	*Number of Occurrences*	*Operand*	*Number of Occurrences*
If	1	*m*	3
<	1	*y*	3
+=	1	*k*	3
–=	1	*j*	3
=	3	dayOfWeek	1
%	2	*d*	1
/	4	3	1
+	6	12	1
*	2	1	1
$n_1 = 9$	$N_1 = 21$	100	2
		26	1
		10	1
		4	2
		5	1
		7	1
		$n_2 = 15$	$N_2 = 25$

Table 16.3 Halstead's Metrics for Second Implementation

Operator	*Number of Occurrences*	*Operand*	*Number of Occurrences*
If	1	Month	3
<	1	Year	5
+=	1	Dayray	1
–	1	Day	1
Return	1	3	1
+	6	12	1
*	2	1	1
/	2	26	1
%	1	10	1
$n_1 = 9$	$N_1 = 16$	4	1
		6	1
		100	1
		400	1
		7	1
		$n_2 = 14$	$N_2 = 20$

Second implementation

```
if (month < 3)
       {
             month += 12;
             -year;
       }
       return dayray[(int)(day + (month + 1) * 26/10 + year +
                           year/4 + 6 * (year/100) + year/400)% 7];
```

Table 16.4 shows the Halstead metrics for the two implementations. Look at the two versions, and decide if you think these metrics are helpful. Remember that these are very small fragments.

The calculations in Table 16.4 are rounded to a reasonable precision. Both versions have nearly equal totals of distinct operators and operands. The big difference is in the number of occurrences (21 vs. 16 and 25 vs. 20, yielding program lengths of 46 and 36). However, the Microsoft Word editor provides the text statistics in Table 16.5, which show that the first version is longer in two senses. Does sheer length add complexity? It depends on what is being done with the code. Size, the number of operators, and the number of operands have clear implications for program comprehension and software maintenance. The testing for the two versions is identical.

Table 16.4 Halstead Metrics for Two Implementations

Halstead's Metric	*Version 1*	*Version 2*
Program length, $N = N_1 + N_2$	21 + 25 = 46	16 + 20 = 36
Program vocabulary, $n = n_1 + n_2$	9 + 15 = 24	9 + 14 = 23
Program volume, $V = N\log_2(n)$	46 ($\log_2$(24)) = 46*4.58 = 210.68	36 ($\log_2$(23)) = 36*4.52 = 162.72
Program difficulty, $D = (n_1N_2)/2n_2$	(9*25)/2*15 = 225/30 = 7.500	(9*20)/2*14 = 180/28 = 6.428

Table 16.5 Character Counts in Two Versions

Size Attribute	*Version 1*	*Version 2*
Characters (no spaces)	99	107
Characters (with spaces)	147	157
Lines	7	7

16.2 Integration-Level Complexity

The entire discussion in Section 16.1 on unit-level complexity applies to both procedural code and to object-oriented methods. The differences in these two paradigms are first noticed at the integration level—in fact they are restricted to that level. At the integration testing level, the concern shifts from correctness of individual units to correct function across units. One presumption of integration-level testing is that the units have been thoroughly tested "in isolation." Therefore, the attention shifts to interfaces among units and what we might call "communication traffic." As with unit-level complexity, we use directed graphs to help our discussion and analysis. The starting point is the call graph from Chapter 13.

Definition

Given a program written in an imperative programming language, its *call graph* is a directed graph in which nodes correspond to units, and edges correspond to messages.

For object-oriented code, if method A sends a message to method B, there is an edge from node A to node B. For procedural code, if unit A refers to unit B, there is an edge from node A to node B. As a general rule, the integration-level call graphs of procedural code are less complex than those of functionally equivalent object-oriented code. At the same time, the unit-level complexity of methods is typically less than that of procedures. This almost suggests a "law of conservation of complexity," in which complexity does not disappear from object-oriented code; it just relocates to the integration level. (This is beyond the scope of this chapter, so it remains an observation.)

16.2.1 Integration-Level Cyclomatic Complexity

Cyclomatic complexity at the integration level echoes the approach we took at the unit level, only now we use a call graph instead of a program graph. As before, we need to distinguish between strongly connected call graphs and call graphs that are "almost" strongly connected. Recall we had two equations for this distinction:

(1) $V(G) = e - n + p$, for strongly connected call graphs
(2) $V(G) = e - n + 2p$ for call graphs that have a single source node and multiple sink nodes

Notice that the next definitions apply to both object-oriented and procedural code. We repeat two definitions from Chapter 4 next.

Definition

Given the call graph of a program (regardless of language paradigm), the *integration-level cyclomatic complexity* is the cyclomatic complexity of the call graph.

Definition

Given a directed graph G with n nodes, its *adjacency matrix* is the $n \times n$ matrix $A = (a_{i,j})$, where $a_{i,j} = 1$ if there is an edge from node i to node j, 0 otherwise.

As we saw in Chapter 4, all of the information in a directed graph can be derived from its (unique!) adjacency matrix, except for the geometric placement of nodes and edges. For example, the sum of elements in row n is the outdegree of node n; similarly, the sum of elements in column n is the indegree of node n. The sum of the indegrees and outdegrees a node is the degree of the node. Since every edge contributes to the outdegree of some node, this, in turn, together with the number of nodes yields the cyclomatic complexity $V(G)$ = edges – nodes + $2p$.

Given this, many times it is simpler to provide an adjacency matrix rather than a drawn call graph. Section 16.3 develops a full example of unit- and integration-level complexity for a rewritten version of NextDate. The call graph of that version is in Figure 16.3, followed by its adjacency matrix (Table 16.6).

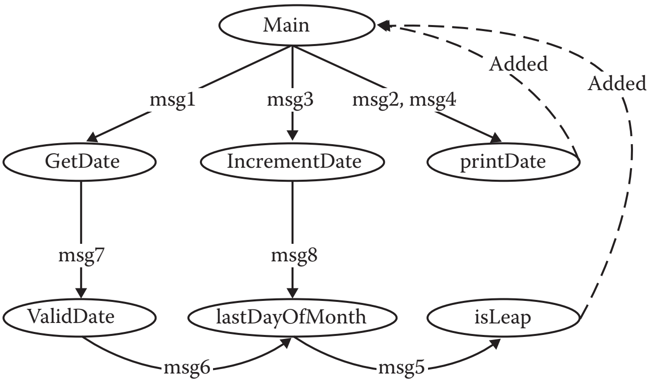


Figure 16.3 Call graph of integration version.

Table 16.6 Adjacency Matrix of Call Graph in Figure 16.3

	Main	*GetDate*	*Increment Date*	*printDate*	*ValidDate*	*lastDayOf Month*	*isleap*	*row sum (outdegree)*
Main		1	1	1				3
GetDate					1			1
IncrementDate						1		1
printDate								0
ValidDate						1		1
lastDayOfMonth							1	1
isleap								0
Column sum	0	1	1	1	1	2	1	7

Integration-level call graphs are seldom (never?) strongly connected, but we can still derive everything we need from the adjacency matrix of a call graph. The sum of the row sums (or column sums) is 7. Nodes with outdegree = 0 must be sink nodes; thus, for each sink node, we would add an edge to make the call graph strongly connected. There are two such nodes in Figure 16.3, so the calculation of integration-level cyclomatic complexity is

$$V(G) = \text{edges} - \text{nodes} + 1 = 9 - 7 + 1 = 3$$

16.2.2 Message Traffic Complexity

As we saw with unit-level complexity, only considering cyclomatic complexity is an oversimplification. Just as not all decisions are equal, neither are all interfaces. Suppose, for example, that we find one method repeatedly sending messages to the same destination—clearly this adds to the overall complexity, and we would like to consider this in our integration testing. To do this, we use an extended adjacency matrix of the call graph. In the extended version, rather than just 1's and 0's, an element shows the number of times a method (or a unit) refers to another method (unit). For the Figure 16.3 example, this only happens once, when the Main unit calls printDate twice. We would have this extended adjacency matrix.

16.3 Software Complexity Example

The NextDate program (usually expressed as a function) is rewritten here as a main program with a functional decomposition into procedures and functions (Figure 16.4). The pseudocode grows from 50 statements to 81. Figures 16.5 through 16.7 show the program graphs of units in yet another integration version of NextDate. The functional decomposition is shown in Figure 16.4, and the call graph is shown in Figure 16.3. The points of added decisional complexity are noted as comments in bold font.

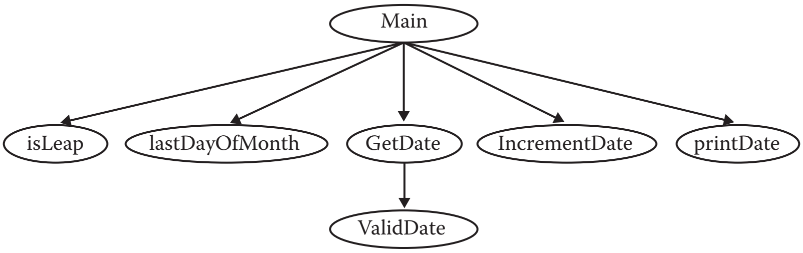

Main
isLeap
lastDayOfMonth
GetDate
IncrementDate
printDate
ValidDate

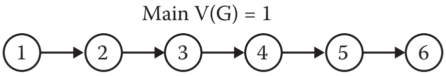
Figure 16.4 Functional decomposition on integration version of NextDate.

```
1  Main integrationNextDate

       Type Date
          Month As Integer
          Day As Integer
          Year As Integer
       End Type

       Dim today As Date
       Dim tomorrow As Date

   2  GetDate(today)                           ‘msg1
   3  PrintDate(today)                         ’msg2
   4  tomorrow = IncrementDate(today)          ‘msg3
   5  PrintDate(tomorrow)                      ’msg4
   6  End Main
```


Main V(G) = 1
1
2
3
4
5
6

```
   7  Function isLeap(year)  Boolean
   8    If (year divisible by 4)
   9      Then
   10       If (year is NOT divisible by 100)
   11         Then isLeap = True
   12         Else
   13          If (year is divisible by 400)
   14             Then isLeap = True
   15             Else isLeap = False
   16          EndIf
   17       Endif
   18     Else isLeap = False
   19   Endif
   20  End (Function isLeap)
```

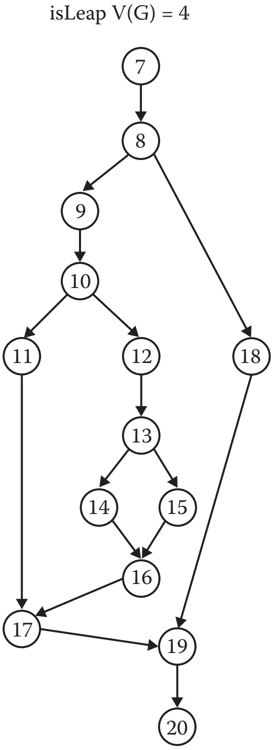
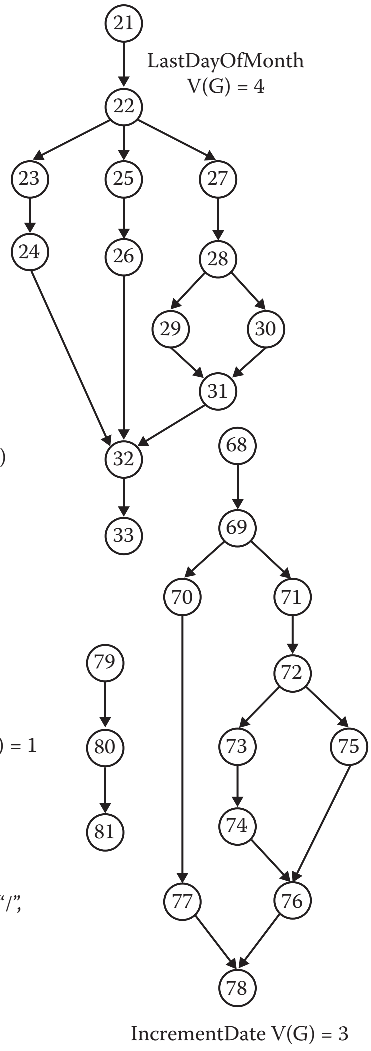

isLeap V(G) = 4
7
8
9
10
11
12
18
13
14
15
16
17
19
20

Figure 16.5 Program graph of integrationNextDate, part 1.

```
21 Function lastDayOfMonth(month, year)  Integer
22   Case month Of
23     Case 1: 1, 3, 5, 7, 8, 10, 12
24       lastDayOfMonth = 31
25     Case 2: 4, 6, 9, 11
26       lastDayOfMonth = 30
27     Case 3: 2
28       If (isLeap(year))                    'msg5
29         Then lastDayOfMonth = 29
30         Else lastDayOfMonth = 28
31       EndIf
32   EndCase
33 End (Function lastDayOfMonth)
```

```
68 Function IncrementDate(aDate)  Date
69   If (aDate.Day < lastDayOfMonth(aDate.Month))
'msg8
70     Then aDate.Day = aDate.Day + 1
71     Else aDate.Day = 1
72       If (aDate.Month = 12)
73         Then aDate>month = 1
74           aDate.Year = aDate.Year + 1
75         Else aDate.Month = aDate.Year + 1
76       EndIf
77   EndIf
78 End (IncrementDate)
```


LastDayOfMonth
V(G) = 4
21
22
23
25
27
24
26
28
29
30
31
32
33
68
69
70
71
72
79
80
81
73
75
74
77
76
78
PrintDate V(G) = 1
IncrementDate V(G) = 3

```
79 Procedure PrintDate(aDate)
80   Output( "Day is *, aDate.Month, "/" aDate.Day, "/",
aDate.Year)
81 End (PrintDate)
```

Figure 16.6 Program graph of integrationNextDate, part 2.

```
34  Function GetDate(aDate  Date
    dim aDate As Date
35   Function ValidDate(aDate)  Boolean  'within scope of GetDate
    dim aDate As Date
    dim dayOK, monthOk, yearOK As Boolean
36     If ((aDate.Month > 0) AND (aDate.Month <=12)
                'added decisional complexity = +1
37       Then monthOK = True
38       Else monthOK = False
39     EndIf
40     If (monthOK)
41       Then
42         If ((aDate.Day > 0) AND             'msg6
       (aDate.Day <= lastDayOfMonth(aDate.Month, aDate.Year))
                'added decisional complexity = +1
43          Then dayOK = True
44          Else dayOK = False
45         EndIf
46       EndIf
47   If ((aDate.Year > 1811) AND (aDate.Year <= 2012)
                'added decisional complexity = +1
48       Then yearOK = True
49       Else yearOK = False
50       EndIf
51   If (monthOK AND dayOK AND yearOK)
                'added decisional complexity = +2
52       Then ValidDate = True
53       Else ValidDate = False
54   EndIf
55  End (Function ValidDate)

' GetDate body begins here
56   Do
57     Output("enter a month")
58     Input(aDate.Month)
59     Output("enter a day")
60     Input(aDate.Day)
61     Output("enter a year")
62     Input(aDate.Year)
63     GetDat.Month = aDate.Month
64     GetDate.Day = aDate.Day
65     GetDate.Year = aDate.Year
66   Until (ValidDate(aDate))            'msg7
67   End (Function GetDate)
```

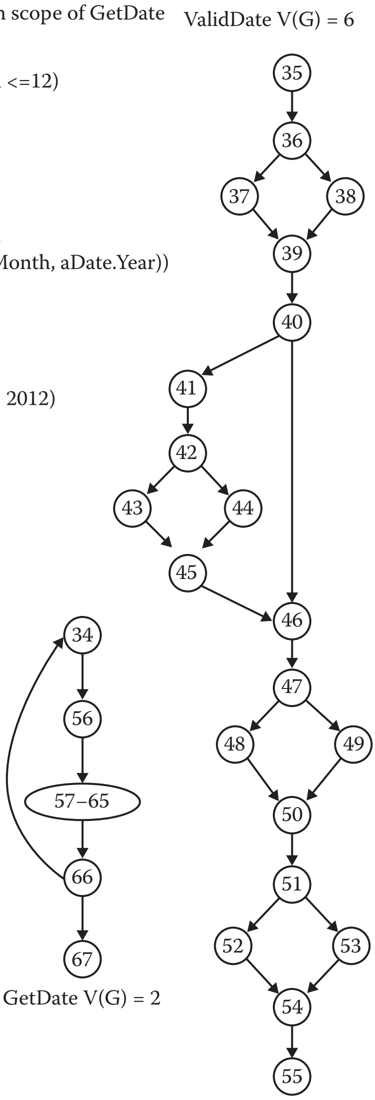


Figure 16.7 Program graph of integrationNextDate, part 3.

16.3.1 Unit-Level Cyclomatic Complexity

Unit	*Cyclomatic Complexity*	*Added Decisional Complexity*	*Total Unit Complexity*
Main integrationNextDate	1	0	1
GetDate	2	0	2
IncrementDate	3	0	3
PrintDate	1	0	1
ValidDate	6	5	11
lastDayOfMonth	4	0	4
isLeap	4	0	4
Sum of total unit complexities			26

16.3.2 Message Integration-Level Cyclomatic Complexity

In Figure 16.3, edges are added from printDate to Main and from isLeap to Main to make the graph a strongly connected directed graph. (So now the $V(G) = e - + p$ formula applies.)

$$V(G(\text{call graph})) = 9 - 7 + 1 = 3$$

The Extended Adjacency Message Traffic Increment = 1, so the total integration-level complexity is 4, added to the unit-level complexity (25) gives a total complexity of 29.

16.4 Object-Oriented Complexity

The Chidamber/Kemerer (CK) metrics are the best-known metrics for object-oriented software (Chidamber and Kemerer, 1994). The names for the six CK metrics are almost self-explanatory; some can be derived from a call graph; others use the unit-level complexity discussed in Section 16.2.

- WMC—Weighted Methods per Class
- DIT—Depth of Inheritance Tree
- NOC—Number of Child Classes
- CBO—Coupling between Classes
- RFC—Response for Class
- LCOM—Lack of Cohesion on Methods

16.4.1 WMC—Weighted Methods per Class

The WMC metric counts the number of methods in a class and weights them by their cyclomatic complexity. This weighting can easily be extended to include the notion of decisional complexity in Section 16.2. As with procedural code, this metric is a good predictor of implementation and testing effort.

16.4.2 DIT—Depth of Inheritance Tree

The name says it all. If we made another call graph to show inheritance, this is the length of the longest inheritance path from root to leaf node. This is directly derivable from a standard UML Class Inheritance Diagram. While comparatively large values of the DIT metric imply good reuse, this also increases testing difficulty. One strategy is to "flatten" the inheritance classes such that all inherited methods are in one class for testing purposes. Current guidelines recommend a limit of DIT = 3.

16.4.3 NOC—Number of Child Classes

The Number of Child Classes of a class in the inheritance diagram for the DIT metric is simply the outdegree of each node. This is very analogous to the cyclomatic complexity of the call graph.

16.4.4 CBO—Coupling between Classes

This metric is a carryover from the procedural coupling metrics central to the Structured Analysis design technique, in which coupling is increased when one unit refers to variables in another unit. In the procedural version, several levels of coupling were identified. These can all be applied to object-oriented methods. As is the case with procedural code, greater coupling implies both greater testing and greater maintenance difficulty. Well-designed classes should reduce the CBO values.

16.4.5 RFC—Response for Class

The RFC method refers to the length of the message sequence that results from an initial message. In Chapter 13, we saw that this is also the "length" of the integration-level testing construct, the MM-path.

16.4.6 LCOM—Lack of Cohesion on Methods

The LCOM metric is another direct extension of the cohesion metric for procedural code. LCOM is the number of methods that use a given instance variable in the class, and is computed for each instance variable.

16.5 System-Level Complexity

While it is conceptually possible to consider cyclomatic complexity at the system level, the sizes make this unwieldy—it can be done, and there are commercial tools that support this, but the results are not particularly helpful. In the words of R.J. Hamming: "The purpose of computing is insight, not numbers."

Part of system-level complexity stems from how closely intertwined are the software units. This is nicely shown in an incidence matrix that relates use cases to classes (or even to methods) as we noted in Chapter 14. Rows correspond to use cases, and columns to classes (or methods). Then an "×" in row *i* column *j* means that class (method) *j* is used to support the execution of use case *i*. Note that for procedural code, the incidence is between features and procedures/functions. Now consider whether this matrix is sparse or dense—a sparse incidence indicates that much of the software is only loosely interwoven, making maintenance and testing relatively easy. Conversely, a dense incidence means that the units are tightly coupled, and therefore highly interdependent. With dense incidence, we can expect more ripple effect of simple changes, and a greater need for rigorous regression testing after a change is made. As an aside, the incidence matrix serves as a handy way to control the items to be regression tested.

EXERCISES

1. Compare the cyclomatic complexity of the procedural implementation of NextDate with the total complexity of the integration version. Then do the same for the object-oriented version in Chapter 15.
2. Consider a calendar function that finds the zodiac sign for a given date. Compare the total complexities of the pseudocode descriptions of zodiac1, zodiac2, and zodiac3.

Zodiac1 uses a procedure validEntry to check the valid ranges of month, day, and year.

```
public zodiac1(month, day, year)
{
  if (validEntry(month, day, year))
      {
     if (month = 3 AND day > = 21 OR month = 4 AND day < = 19) {
          return(“Aries”);
     } else if (month = 4 OR month = 5 AND day < = 20) {
          return(“Taurus”);
     } else if (month = 5 OR month = 6 AND day < = 20) {
          return(“Gemini”);
     } else if (month = 6 OR month = 7 AND day < = 22) {
          return(“Cancer”);
     } else if (month = 7 OR month = 8 AND day < = 22) {
          return(“Leo”);
     } else if (month = 8 OR month = 9 AND day < = 22) {
          return(“Virgo”);
     } else if (month = 9 OR month = 10 AND day < = 22) {
          return(“Libra”);
     } else if (month = 10 OR month = 11 AND day < = 21) {
          return(“Scorpio”);
     } else if (month = 11 OR month = 12 AND day < = 21) {
          return(“Sagittarius”);
     } else if (month = 12 OR month = 1 AND day < = 19) {
          return(“Capricorn”);
     } else if (month = 1 OR month = 2 AND day < = 18) {
          return(“Aquarius”);
     } else {
          return(“Pisces”);
     }
  } else {
     return(“Invalid Date”);
  }
}
```

Zodiac2 presumes that the values of month, day, and year are valid. The zodiac signs are assumed to be in an array zodiac(*i*).

```
public Zodiac2(month, day, year)
{
    switch(month
    {
    case 1: {if(day() > = 20)
                 return zodiac[0];
        else
                 return zodiac[3];    }
    case 2: {if(day() > = 19)
                 return zodiac[7];
        else
                 return zodiac[0];    }
    case 3: {if(day() > = 21)
                 return zodiac[1];
        else
                 return zodiac[7];    }
    case 4: {if(day() > = 20)
                 return zodiac[10];
        else
                 return zodiac[1];    }
    case 5: {if(day() > = 21)
                 return zodiac[4];
        else
                 return zodiac[10];   }
    case 6: {if(day() > = 21)
                 return zodiac[2];
        else
                 return zodiac[4];    }
    case 7: {if(day() > = 23)
                 return zodiac[5];
        else
                 return zodiac[2];    }
    case 8: {if(day() > = 23)
                 return zodiac[11];
        else
                 return zodiac[5];    }
    case 9: {if(day() > = 23)
                 return zodiac[6];
        else
                 return zodiac[11];   }
    case 10: {if(day() > = 23)
                 return zodiac[9];
        else
                 return zodiac[6];    }
    case 11: {if(day() > = 22)
                 return zodiac[8];
        else
                 return zodiac[9];    }
    case 12: {if(day() > = 20)
                 return zodiac[3];
        else
             return zodiac[8];    }
    default:
        return zodiac[12]; }
    }
```

This design choice uses the "ordinal day of the year." Feb. 1 is ordinal day 32. It presumes a function that converts a date to the ordinal day of the year. This version only works for common years. A one-day correction would be needed for leap years.

```
public String zodiac3()
 { switch(month)
  { case 1:  if(ordinalDay<20)
             return "Capricorn";
    case 2:  if(ordinalDay<50)
             return "Aquarius";
    case 3:  if(ordinalDay<79)
             return "Pisces";
    case 4:  if(ordinalDay<109)
             return "Aries";
    case 5:  if(ordinalDay< = 140)
             return "Taurus";
    case 6:  if(ordinalDay<171)
             return "Gemini";
    case 7:  if(ordinalDay<203)
             return "Cancer";
    case 8:  if(ordinalDay<234)
             return "Leo";
    case 9:  if(ordinalDay<265)
             return "Virgo";
    case 10: if(ordinalDay<295)
             return "Libra";
    case 11: if(ordinalDay<325)
             return "Scorpio";
    case 12: if(ordinalDay<355)
              return "Sagittarius";
              else
              return "Capricorn";
  }
}
```

Reference

Chidamber, S.R. and Kemerer, C.F., A metrics suite for object-oriented design, *IEEE Transactions of Software Engineering*, Vol. 20, No. 6, 1994, pp. 476–493.

Chapter 17

Model-Based Testing for Systems of Systems

On March 2, 2012, a class EF-4 tornado struck the town of Henryville, Indiana (USA). The tornado had winds of 170 mph and left a path of destruction 50 miles long. My wife and I were driving south on Interstate 65; when we were about 50 miles north of Henryville, we saw an Indiana state police car with a sign directing motorists to move to the left lane of the highway. This was the beginning of a direct experience with a "system of systems." Soon, traffic came to a halt, and then impatient drivers started using the right lane anyway, quickly bringing that lane also to a stop. Then we saw emergency vehicles and heavy equipment heading south using the shoulder of the road. We learned from a truck driver that a tornado had hit Henryville about an hour earlier, and that the emergency vehicles and heavy equipment were attempting to reach the devastated area. We noticed that there was very little northbound traffic on Interstate 65, so clearly, northbound traffic south of Henryville was also stopped. The next day, we saw that a highway rest area had been converted to a command center for the Indiana National Guard to coordinate the disaster relief effort. This effort involved:

- The Indiana state police
- Local and county police departments
- Regional fire departments
- Regional ambulance services
- Heavy (tree moving) equipment from the public utility companies
- The Indiana National Guard
- Traffic helicopters from Indianapolis television stations
- The US Weather Bureau
- (And probably many others)

Consider how this all happened. How did these disparate groups come together for an emergency? How did they communicate? Was there any central coordination?

Systems of systems have become an increasingly important topic in several areas of software engineering. In this chapter, we look at some of the early definitions (Maier, 1999), some SysML techniques to specify requirements of these systems, and finally, we develop a new model to describe systems of systems and their model-based testing.

17.1 Characteristics of Systems of Systems

We all experience complex systems every day, but what distinguishes a complex system from a system of systems? Some early attempts to clarify this distinction are

- A "super system"
- A collection of cooperating systems
- A collection of autonomous systems
- A set of component systems

These early attempts all get at the central idea, but they would also apply to systems such as an automobile, an integrated MIS system in a company, and even the human body. There is a growing clarity of definitions for the underlying nature of systems of systems. Maier begins his distinction by noting two fundamental differences—systems of systems are either directed or collaborative. Initially, he used "collaborative systems" as a synonym for "systems of systems," with the defining characteristic that systems of systems are "built from components which are large scale systems in their own right." He offers air defense networks, the Internet, and emergency response teams as better examples. Maier then provides some more specific attributes:

- They are built from components that are (or can be) independent systems.
- They have managerial/administrative independence.
- They are usually developed in an evolutionary way.
- They exhibit emergent (as opposed to preplanned) behaviors.

In addition, he observes that the components may not be co-located, and this imposes a constraint on information sharing. The generally accepted term for the components is "constituent system," and a general architecture is shown in Figure 17.1. Notice that constituent systems

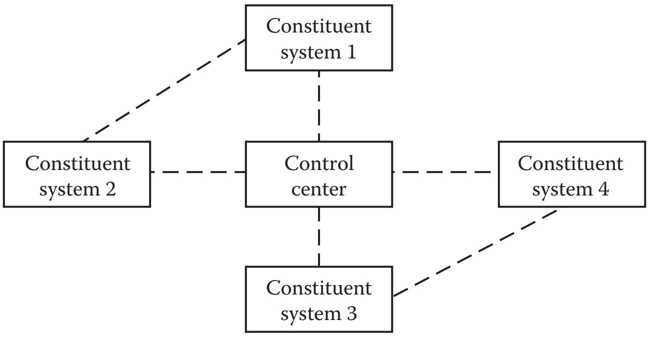


Figure 17.1 Generic view of a system of systems.

may have links other than to the central control point. The control center portion leads to three important distinctions that Maier makes regarding the nature of cooperation among the constituent systems.

Definition (Maier, 1999)

A *directed system of systems* is designed, built, and managed for a specific purpose.
A *collaborative system of systems* has limited centralized management and control.
A *virtual system of systems* has no centralized management and control.

The dominant characteristic that distinguishes these categories is the manner in which they communicate and control/cooperate. Maier further asserts that there are two essential requirements that a potential system of systems must satisfy:

1. The constituent systems must be stand-alone systems in their own right.
2. Each constituent has administrative independence from the other constituents.

Maier's three categories were extended (Lane, 2012) to include a fourth category: acknowledged. In order from most to least controlling, we have directed, acknowledged, collaborative, and virtual systems of systems.

Systems of systems (abbreviated as SoS) can evolve. The Henryville tornado incident began as a virtual system of systems—there was no centralized control point. When the Indiana state police arrived, it evolved into a collaborative system of systems. By the next morning, the Indiana National Guard had turned a rest area into a command center, and it was then an acknowledged system of systems. Why is this not a directed system of systems? The constituents are all independent systems that can function in their own right, and each has separate administrative control; however, as a system of systems, it was never created with that purpose in mind.

17.2 Sample Systems of Systems

To gain some insight into Maier's categories of systems of systems, we consider one example of each type. The emphasis in this section is how the constituent systems communicate, and how they are, or might be, controlled.

17.2.1 The Garage Door Controller (Directed)

A nearly complete garage door controller system (see Chapter 2) is shown as a system of systems in Figure 17.2. Some elements must be present, namely, the drive motor, a wall-mount button, and the extreme limit sensors. The other constituents are optional but common. The portable opener is usually kept in a car, and there may be two or more of these. Sometimes a digit keypad is mounted on the outside of a garage, possibly so children can enter after school. The openers and digit keypads send weak radio signals to the wireless receiver, which in turn, controls the drive motor. A possible Internet-based controller is not shown, but could be added. Finally, the

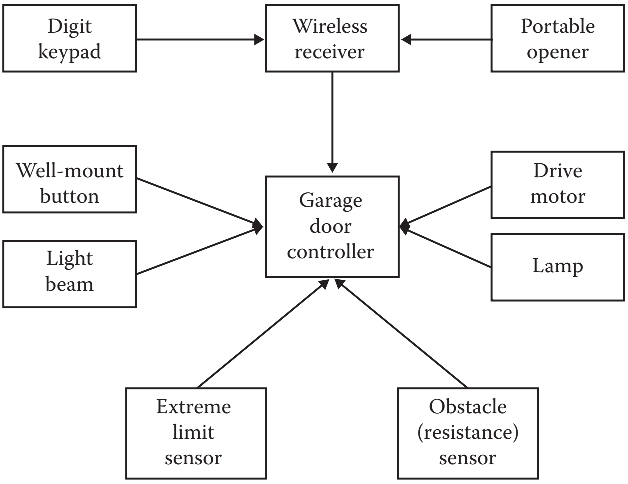

Digit keypad
Wireless receiver
Portable opener
Well-mount button
Garage door controller
Drive motor
Light beam
Lamp
Extreme limit sensor
Obstacle (resistance) sensor

Figure 17.2 Garage door controller constituents.

light beam and resistance sensors are added as optional safety devices. Many of the constituent systems are made by separate manufacturers and are integrated into a commercial garage door opener system.

The garage door controller satisfies most of Maier's definitional criteria—there is a true central controller, and commercial versions of the full system of systems can evolve with the addition of some constituent systems (e.g., the digit keypad).

17.2.2 Air Traffic Management System (Acknowledged)

At a commercial airport (or at any controlled airfield), the air traffic controllers use an air traffic management system (yet another ATM) to manage takeoffs and landings. Figure 17.3 shows the major constituent systems for an air traffic control system. The first decision an air

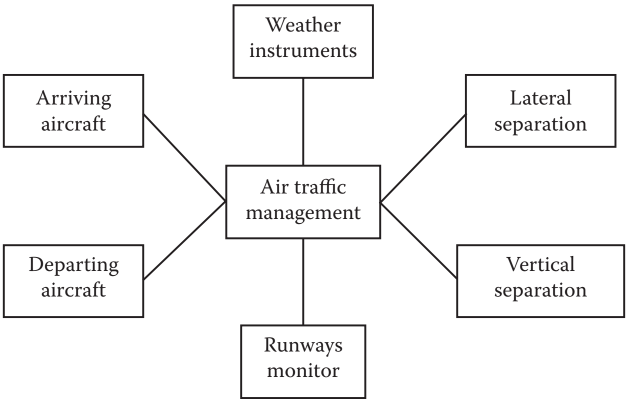

Weather instruments
Arriving aircraft
Lateral separation
Air traffic management
Departing aircraft
Vertical separation
Runways monitor

Figure 17.3 Air traffic management system constituents.

traffic controller must make is runway allocation. This depends mostly on the wind direction, but it may also consider local noise restrictions. Arriving aircraft generally have preference over departing aircraft, because an aircraft on the ground can just stay out of the way of landing aircraft. Airborne aircraft are subject to three forms of separation, each of which must be maintained—vertical separation, lateral separation, and time separation. The only exception to these protocols is that, in an emergency situation, the pilot of an arriving aircraft can request emergency landing priority.

Why is this "acknowledged" and not "directed"? In general, the air traffic controllers, as the name implies, control everything involved with runway use, separation, landing, and departing aircraft. However, emergencies can occur, as we shall see later, making this an acknowledged systems of systems.

17.2.3 The GVSU Snow Emergency System (Collaborative)

Grand Valley State University (GVSU) is located in western Michigan where significant snowfall events can and do occur. In extreme events, the campus must be closed for the safety of students, faculty, and staff. In a snow emergency, it is important to prevent vehicles from coming to the campus, or if they are already there, to remove them safely. The Information Technology office maintains a "reverse 911" system to notify all parties (students, faculty, and staff) by both email and telephone if there is a snow emergency and the attendant campus closure. If a snow emergency is declared before 6:00 a.m., local television and radio stations are notified (Figure 17.4).

The Campus Safety Department (campus police) acts as the control point in a snow emergency. Campus safety cruisers are posted at the entrances to the campus to turn away any arriving traffic. When a snow emergency begins during a class day, the problem is more complex. All automobiles in campus parking lots must be removed so that the snow removal process can begin. This may entail just the university grounds equipment; however, in an extreme snowfall, supplemental snow plowing may be obtained from local snow removal companies and the county highway department. All the constituents have primary functions, but in a snow emergency, these responsibilities become secondary to the overall snow emergency procedures.

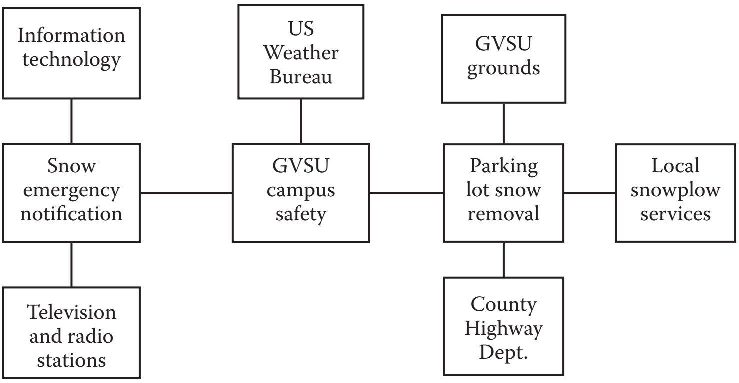


Figure 17.4 GVSU Snow Emergency System constituents.

17.2.4 The Rock Solid Federal Credit Union (Virtual)

The hypothetical Rock Solid Federal Credit Union (RSFCU) is a small credit union organized under the auspices of the U.S. government. All credit unions are non-profit organizations that exist to serve their members. Member services include

- Opening and closing accounts
- Making loans (mortgage, home equity, auto/boat, and signature)
- Credit counseling

In addition, there are administrative functions, including

- Staff management (salary, work assignments, hire/fire responsibilities)
- Interfacing with the board of directors
- Public relations

Because the RSFCU is federally chartered, there are levels of government control and responsibilities, including local, state, and federal interfaces. To conduct its business, the RSFCU works with several constituent systems. The major ones are shown in Figure 17.5.

The RSFCU is a virtual system of systems because each of the constituent systems can, in some cases, make demands on the credit union. In turn, the credit union can make demands on some of the other constituents. Each of the constituents clearly has administrative autonomy, but there are also established patterns of collaboration. To make a mortgage loan, for example, all of the parties except the Federal Reserve Bank are involved.

17.3 Software Engineering for Systems of Systems

Very little published work exists to apply software engineering principles and techniques to systems of systems. Some of the early work (Maier, 1999; Lane, 2012) is described here, as well as

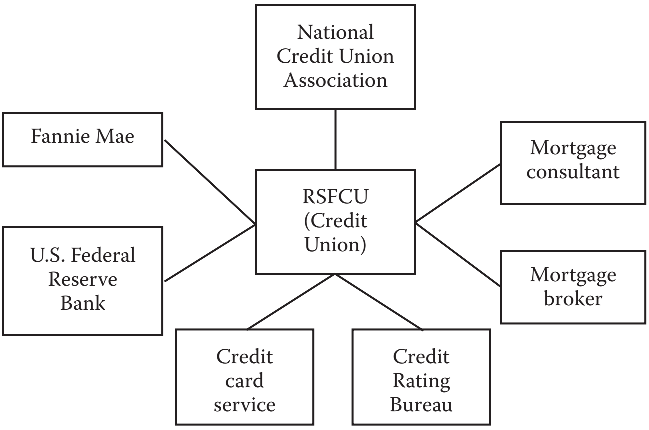


Figure 17.5 Constituents of a small federal credit union.

some original material. We will refer to all of this as a UML dialect. After illustrating how this UML dialect can represent systems of systems, we will turn to an approach that supports model-based testing of systems of systems.

17.3.1 Requirements Elicitation

In a webinar, Jo Ann Lane described an emergency response system of systems that dealt with grass fires in southern California (Lane, 2012). Lane offers a waterfall-like sequence of activities to describe general requirements of a given system of systems. The steps include

- Identifying resources—potential constituent systems, and modeling them with SysML
- Determining options—responsibilities and dependencies
- Assessing options—expressed as use cases
- Identifying a workable combination of constituent systems
- Allocating responsibilities to constituent systems

In the next few sections, we revise and extend the standard UML practices to make them work for systems of systems. These are illustrated with the examples in Section 17.2.

17.3.2 Specification with a Dialect of UML: SysML

There are three parts to the SysML dialect—class-like definitions of a constituent system in terms of its responsibilities to other constituents and the services it provides. Use cases show the flow across constituents for overall system of system functions, and traditional UML sequence diagrams to show the incidence of these use cases with constituent systems.

17.3.2.1 Air Traffic Management System Classes

The SysML dialect extends and revises some of the traditional UML models. In the SysML dialect, constituent systems are modeled as classes in which responsibilities with other classes occupy the position of attributes, and the services take the place of class methods. Two of the constituent systems from Figure 17.3 are described as “classes” in text format here.

Incoming aircraft

Responsibilities to other constituents

- Communicate with air traffic controller

Services

- Fly aircraft
- Land aircraft
- Remain prepared for emergency situations

Air traffic controller

Responsibilities to other constituents

- Incoming aircraft
- Departing aircraft
- Runway (status)
- Separation instruments
- Weather instruments

Services

- Assign runways on the basis of weather conditions
- Monitor separation instruments
- Assign landing clearance
- Assign takeoff clearance
- Maintain runway status

17.3.2.2 Air Traffic Management System Use Cases and Sequence Diagrams

In both the standard UML and in the dialect used here, classes constitute the *is view* that focuses on the structure and components of a system (and systems of systems).

The *is view* is most useful to developers, but less so for customer/users and testers, who utilize the *does view*, which focuses on behavior. Use cases are the earliest UML model that relate to the *does view*, and they are widely recognized as the preferred view of customer/users. The UML sequence diagram is the only place where the *does view* is related to the *is view*. Figure 17.6 is a sequence diagram of the Normal Landing use case. For our dialect, we added Actors (constituent systems) to the use case format. Also, the usual Event Sequence of a standard UML use case is replaced by the sequence of constituent system actions.

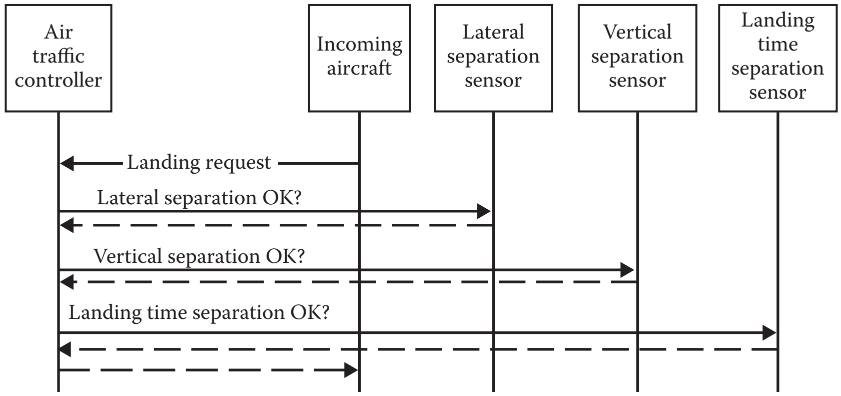

Air traffic controller
Incoming aircraft
Lateral separation sensor
Vertical separation sensor
Landing time separation sensor
Landing request
Lateral separation OK?
Vertical separation OK?
Landing time separation OK?

Figure 17.6 Sequence diagram of Normal Landing.

Normal Landing Use Case

ID: Name	SoS UC1: Normal aircraft landing
Description	The procedure that governs an arriving aircraft under normal conditions.
Actor(s)	1. Air traffic controller
	2. Incoming aircraft
	3. Separation sensors (vertical, lateral, and time)
Preconditions	1. Designated runway clear
	2. Incoming aircraft ready to land
Action Sequence	
Actor	Action
Incoming aircraft	1. Requests clearance to land
Air traffic controller	2. Checks all separation sensors
Lateral separation	3. OK
Vertical separation	4. OK
Time separation	5. OK
Air traffic controller	6. Landing clearance given
Incoming aircraft	7. Initiates landing procedures
Incoming aircraft	8. On assigned runway
Incoming aircraft	9. Taxi to assigned gate
Air traffic controller	10. Landing complete
Postconditions	1. Runway available to other aircraft

In November 1993, a commercial aircraft was on its final landing approach to a runway at Chicago's O'Hare International Airport. When the incoming aircraft was at an altitude of about 100 ft, a pilot waiting to take off saw that the landing aircraft had not lowered its landing gear. There is no direct communication between landing and departing aircraft, so the pilot contacted the O'Hare field control tower about the impending disaster. The control tower waved off the landing aircraft, and a disaster was avoided. This is the subject of our second use case and sequence diagram. In this use case, aircraft L is the landing aircraft, and aircraft G is the one on the ground. We can imagine that the second use case could be a continuation of the first one at action step 7. We can also imagine that everyone involved was very relieved once the postcondition was attained.

November 1993 Incident Use Case

ID: Name	SoS UC2: November 1993 Incident at O'Hare Field
Description	Aircraft on final approach had landing gear up. Pilot on taxiway saw this and notified control tower.
Actor(s)	1. Air traffic controller
	2. Incoming aircraft L
	3. Aircraft G waiting to take off
Preconditions	1. Aircraft L cleared to land
	2. Aircraft G waiting to take off
	3. Aircraft L has landing gear up
Action Sequence	
Actor	Action
Air traffic controller	1. Authorizes aircraft L to land
Aircraft L	2. Initiates landing preparation
Aircraft L	3. Fails to lower landing gear
Aircraft L	4. 100 ft above end of assigned runway
Aircraft G	5. Aircraft G pilot radios air traffic controller
Air traffic controller	6. Terminates landing permission
Aircraft L	7. Aircraft L aborts landing
Aircraft L	8. Aircraft L regains altitude over runway
Air traffic controller	9. Instructs aircraft L to circle and land
Air traffic controller	10. Thanks pilot of aircraft G
Air traffic controller	11. Authorizes aircraft L to land
Aircraft L	12. Landing complete
Postconditions	1. Runway available to other aircraft

This incident happened when I arrived in Chicago to make a presentation on software technical reviews. Oddly enough, the topic of the day was the importance of review checklists. Obviously, the pilot of the landing aircraft did not pay attention to the landing checklist. In a television news report later, a Federal Aviation Authority official commented that he was far more worried about routine flights than flights during extreme conditions. His reason—people are far more attentive in extreme situations. The sequence diagram for the November 1993 incident is in Figure 17.7. Notice that many of the "internal" actions (2, 3, and 4) are important to the use case, but they do not appear in the sequence diagram.

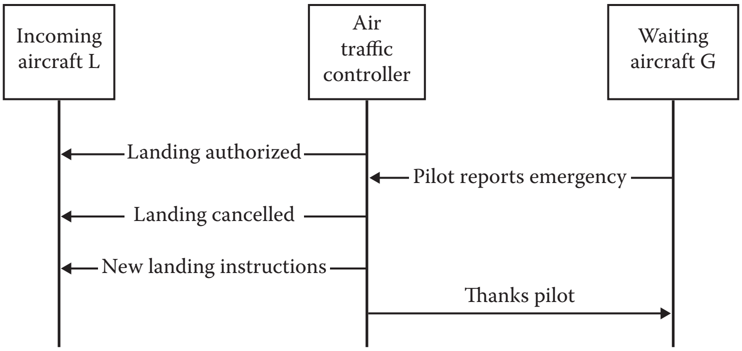


Figure 17.7 Sequence diagram of November 1993 incident.

17.3.3 Testing

Testing for systems of systems must focus on the ways in which constituent systems communicate. Just as integration testing presumes complete unit-level testing, the testing of systems of systems must presume that the constituent systems have been thoroughly tested as stand-alone components. The SysML dialect models are only general guidelines for system of systems testing. The primary goal of testing for systems of systems is to focus on the communication among constituents. In the next section, we develop a set of primitives that describe the types of communication among constituent systems. They will be presented as Petri nets, and we will use them to describe the control distinctions that are the essence of the four levels of cooperation (directed, acknowledged, voluntary, or virtual).

17.4 Communication Primitives for Systems of Systems

The distinctions among the four types of systems of systems reduce to the manner in which the constituents communicate with each other. In this section, we first map the prompts of the Extended Systems Modeling Language (ESML) into Swim Lane Petri nets. In Section 17.5, we use the Petri net forms of the ESML prompts in swim lanes to illustrate the communication mechanisms of the four types of systems of systems. We understand swim lanes to be device oriented, very similar to the orthogonal regions of StateCharts. More specifically, we will use swim lanes to represent constituent systems and the ESML prompts to represent the types of communication among constituents. Finally in Section 17.6, we illustrate the systems of systems communication using Swim Lane Event-Driven Petri Nets on the November 1993 incident.

The first candidate for a set of communication primitives is the set of ESML prompts. Most of these express the power of the central controlling constituent, so they are clearly applicable to directed systems of systems, and probably also to acknowledged systems of systems. We need similar primitives for the collaborative and virtual systems of systems. Here we propose four new primitives: Request, Accept, Reject, and Postpone.

17.4.1 ESML Prompts as Petri Nets

The ESML real-time extension to structured analysis (Bruyn et al., 1988) was developed as a way to describe how one activity in a data flow diagram can control another activity. There are five basic ESML prompts: Enable, Disable, Trigger, Suspend, and Resume, and they are most appropriate to directed and acknowledged systems of systems. Two others are pairs of the original five: Activate is an Enable followed by a Disable, and Pause is a Suspend followed by a Resume. The ESML prompts are represented as traditional Petri nets and briefly described in this section. The marking and firing of Petri nets are described in Chapter 4.

17.4.1.1 Petri Net Conflict

We describe the Petri net conflict first because it appears in some of the ESML prompts. Figure 17.8 shows the basic Petri net conflict pattern—the place p2 is an input to both the function 1 and function 2 transitions. All three places are marked, so both transitions are enabled, in the Petri net sense. ("Enabling" is an overloaded term here—the ESML sense refers to a prompt, and the Petri net transition sense refers to a property of a transition.) If we choose to fire the function 1 transition, the tokens in places p1 and p2 are consumed, and this disables the function 2 transition, hence the conflict.

In the air traffic management and control example, two constituents, arriving and departing aircraft, both use the same runway, putting them in contention for the limited resource—a good example of Petri net conflict. Since arriving aircraft have preference over departing aircraft, we have an instance of the interlock mechanism described next.

17.4.1.2 Petri Net Interlock

An interlock is used to assure that one action precedes (or has priority over) another. In Petri nets, this is accomplished by an interlock place, labeled "i" in Figure 17.9 that is an output of the preferred transition and is an input to the secondary transition. The only way the interlock place can be marked is for the preferred transition to fire.

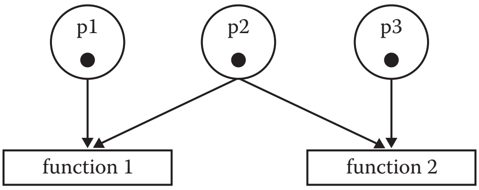


Figure 17.8 Petri net conflict.

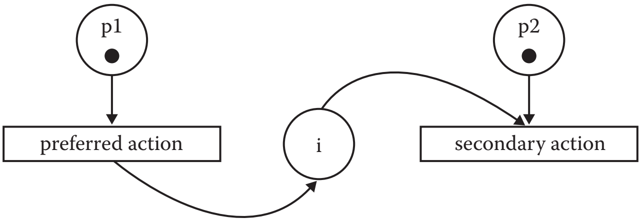


Figure 17.9 Petri net interlock.

17.4.1.3 Enable, Disable, and Activate

The Enable prompt expresses the interaction in which one action permits another action to occur. There is no requirement that the second action actually does occur, just that it may occur. In the Petri net in Figure 17.10, the transition labeled "controlled action" has two input places. For it to be an enabled transition, both its input places must be marked. But the place labeled "e/d" can only be marked if the enable transition is fired. The controlled action then has one of its prerequisites, but it still needs to wait for the other input place to be marked. When the controlled action transition fires, it marks the e/d place again, so that it remains enabled.

At a controlled airfield, the air traffic controller selects a runway, and then gives permission to arriving aircraft to land. We can model this with the Enable prompt. Owing to the interlock relationship between arriving and departing aircraft, this effectively causes a Disable prompt for the aircraft waiting to take off. Since aircraft landings clearly begin and end, this can also be interpreted as an Activate prompt. The air traffic controller "activates" the landing process.

The Disable prompt depends on the Petri net conflict pattern. The disable transition and the controlled action transitions in Figure 17.10 are in conflict with respect to the e/d place. If the disable transition fires, the controlled action transition cannot fire. Also, the e/d place acts as an interlock between the enable and disable transitions, so a controlled action can only be disabled after it has been enabled. The original ESML team found that the Enable, Disable sequence occurred so frequently, it acquired a name: Activate.

17.4.1.4 Trigger

The Trigger prompt (Figure 17.11) is a stronger version of the Enable prompt—it causes the controlled action to occur immediately. In ordinary language, we could say that the effect of Enable is "you may" and that of Trigger is "you must, now!" Notice that Trigger has the same renewal pattern that we saw with Enable. We could modify this if necessary so that Trigger is a one-time action. Just removing the output edge from the controlled action back to the "t" place suffices. The ESML committee never made this distinction.

17.4.1.5 Suspend and Resume

The ESML Suspend and Resume prompts are shown in Figure 17.12. When they occur together, their sequence is known as the ESML Pause prompt. Suspend has the same interrupting power as the Trigger prompt—it can interrupt an ongoing activity, and when the interrupting task is

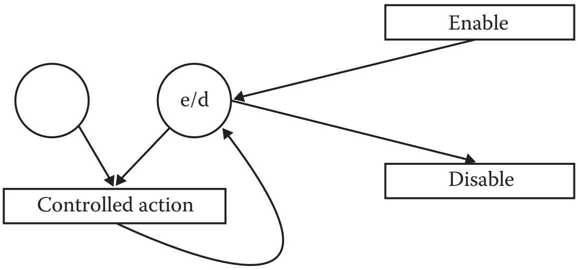


Figure 17.10 ESML Enable, Disable, and Activate.

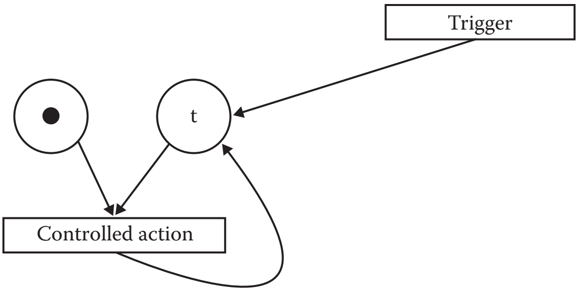


Figure 17.11 ESML Trigger.

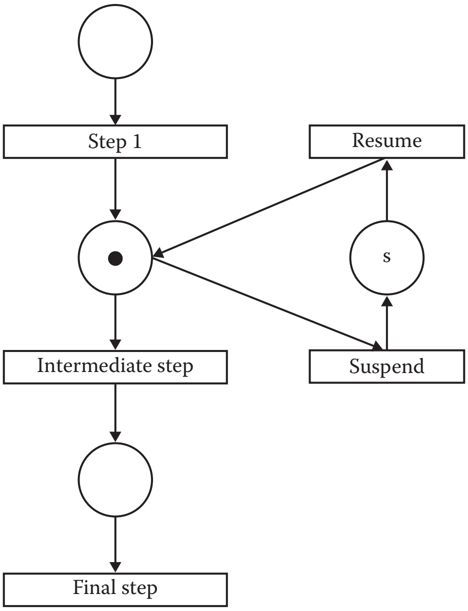


Figure 17.12 ESML Suspend, Resume, and Pause.

complete, the Resume prompt assures that the interrupted activity does not have to start over—it resumes where it left off. Conversationally, we could say “Stop what you are doing.”

As with the Enable/Disable pair, Suspend and Resume have an interlock place, noted as “s” in Figure 17.12. An activity can only be resumed after it has been suspended, so place “s” is an interlock between the Suspend and Resume actions. Also, the Suspend action and the intermediate step action are in Petri net conflict with respect to the marked input place of the intermediate step. Presumably, a Suspend is followed by a Trigger to another required action, which, when complete, leads to a Resume prompt.

The November 1993 incident described earlier is a good example of where the Suspend and Resume prompts could be used. There is no direct communication between landing and departing aircraft, so the pilot on the ground contacted the O’Hare field control tower about the impending disaster. The control tower waved off the landing aircraft (Suspend), and once the disaster was avoided, issued a Resume.

17.4.2 New Prompts as Swim Lane Petri Nets

The directed and acknowledged systems of systems are characterized by a strong, usually central, controlling constituent. Collaborative and virtual systems of systems do not have this strong position; the constituents are more autonomous. Can a constituent in one of these systems of systems control another? Certainly, but it is more likely that the communication is more collaborative than controlling. Four new primitives are proposed here to capture this more collaborative communication—Request, Accept, Reject, and Postpone. As with the ESML prompts, they can and should interact.

Parallel activities are shown in UML as "swim lanes" to connote that a swimmer in one lane is separated from a swimmer in an adjacent lane. In Section 17.4.1, we mapped each of the ESML prompts into Petri nets. We understand swim lanes to be device oriented, very similar to the orthogonal regions of StateCharts. More specifically, we will use swim lanes to represent constituent systems, and the communication prompts to represent the types of communication among constituents. Finally, we illustrate the systems of systems communication using Swim Lane Petri Nets on the November 1999 incident. In this subsection, the constituent systems are all members of either collaborative or virtual systems of systems.

17.4.2.1 Request

In Figure 17.13, constituent A requests a service from constituent B, and receives a response to the request. The figure only shows the interaction from the point of view of constituent A because the response choice of constituent B is not known. In general, a response is either Accepted, Rejected, or Postponed.

17.4.2.2 Accept

The Accept and Reject primitives are nearly identical, except for the nature of the response (see Figures 17.14 and 17.15).

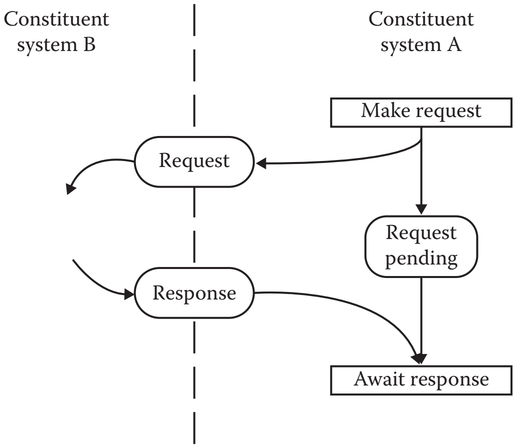


Figure 17.13 Request Petri net.

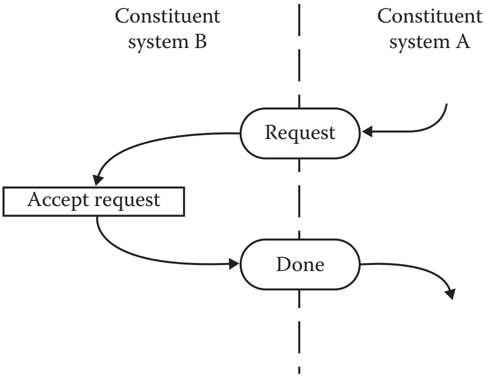

Constituent system B
Constituent system A
Request
Accept request
Done

Figure 17.14 Accept Petri net.

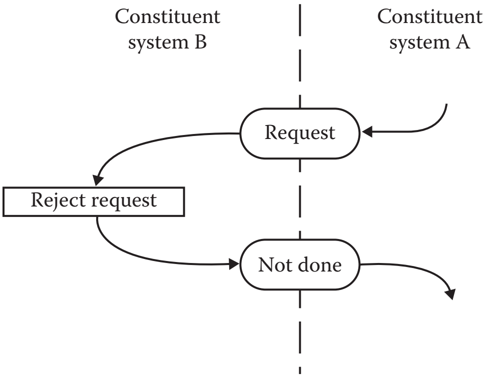

Constituent system B
Constituent system A
Request
Reject request
Not done

Figure 17.15 Reject Petri net.

17.4.2.3 Reject

The "not done" part of a Reject response could be problematic for testers. How can something that does not happen be tested? The Accept and Reject responses are often subject to a Petri net conflict in the receiving constituent, as in Figure 17.16.

Figure 17.17 shows a fairly complete picture of the Petri net conflicts in both constituents. Constituent A makes a request of constituent B. In turn, B either accepts or rejects the request, so either the "done" or the "not done" place is marked, and this resolves the Petri net conflict in constituent A.

17.4.2.4 Postpone

What happens if constituent B is busy with an internal priority, and receives a request from constituent A? The interlock pattern is how constituent B completes its preferred task before responding to the request from constituent A (see Figure 17.18).

17.4.2.5 Swim Lane Description of the November 1993 Incident

See Figure 17.19.

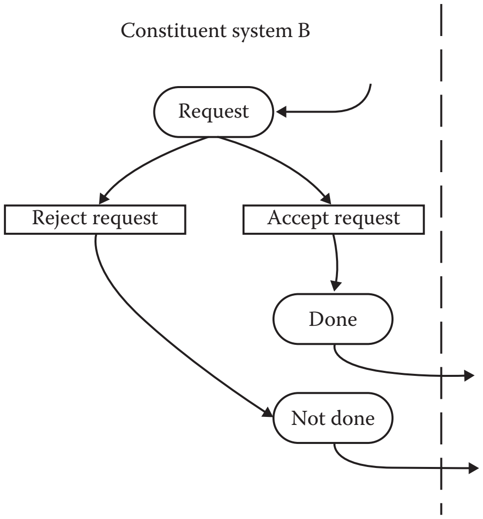

Constituent system B
Request
Reject request
Accept request
Done
Not done

Figure 17.16 Accept and Reject Petri net conflict.

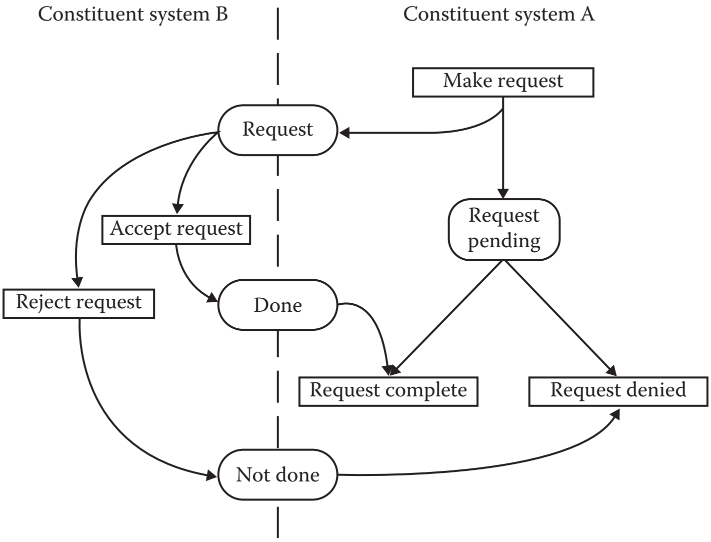

Constituent system B
Constituent system A
Make request
Request
Accept request
Request pending
Reject request
Done
Request complete
Request denied
Not done

Figure 17.17 Connections among Request, Accept, and Reject Petri nets.

17.5 Effect of Systems of Systems Levels on Prompts

When the ESML committee first defined the five prompts, there was some confusion about sequences of prompts. For example, could a Suspend take precedence over a Trigger? Part of the confusion was that the ESML committee was not thinking in terms of systems of systems. To some extent, the definition of the four levels of systems of systems resolves these questions.

One way to begin clarification of this is to postulate two types of communication—commands and requests. The four new prompts are already known as requests, but what about the

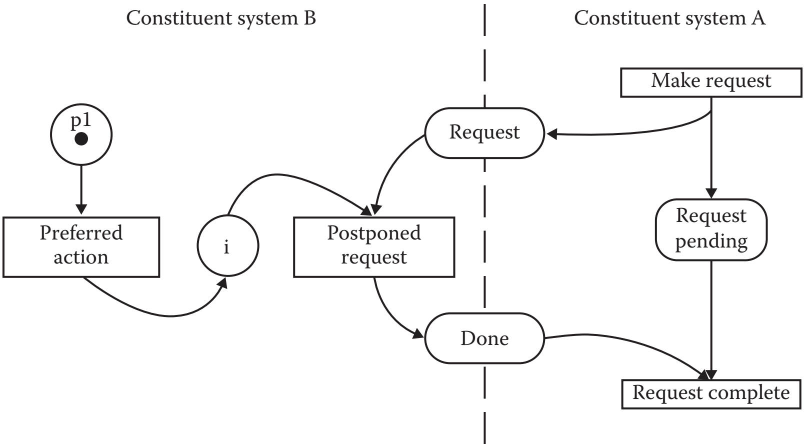

Constituent system B
Constituent system A
Make request
p1
Request
Request pending
Preferred action
i
Postponed request
Done
Request complete

Figure 17.18 Postpone Petri net.

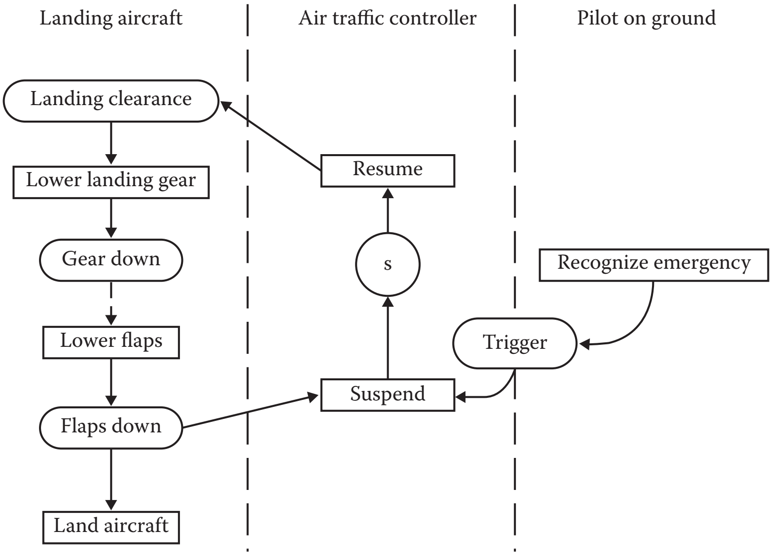

Landing aircraft
Air traffic controller
Pilot on ground
Landing clearance
Lower landing gear
Resume
Gear down
s
Recognize emergency
Lower flaps
Trigger
Suspend
Flaps down
Land aircraft

Figure 17.19 Swim lane Petri net for November 1993 incident.

original ESML prompts? The Trigger, Suspend, Disable, and Resume prompts are all commands, whereas Enable is more of a request.

17.5.1 Directed and Acknowledged Systems of Systems

The central controllers in directed and acknowledged systems of systems are clearly intended to have the “command” power of Trigger, Suspend, Disable, and Resume with respect to their constituents. What about the reverse? Does it make sense for a constituent to “control” the central controller? This seems appropriate when a constituent communicates with what would be an interrupt in software. Consider the safety features in the garage door controller—when an obstacle is encountered, or when the light beam is crossed. The motor immediately stop and reverses to open the garage door.

17.5.2 Collaborative and Virtual Systems of Systems

Because they lack the strong central controlling constituent, both of these types of systems of systems can use any of the prompts. In the RSFCU, for example, the U.S. government can (and does) make Trigger prompts. At the same time, the credit union can make immediate demands of some of its constituents. The same relationships occur in the GVSU Snow Emergency system of systems.

EXERCISES

1. Discuss whether the Disable prompt should have the same interrupting power as the Suspend prompt. Use examples if you wish.

   Questions 2 and 3 revisit the description of the garage door controller in Chapter 2 and the corresponding finite state machine in Chapter 4.
2. Decide which of the four types of systems of systems best describes the garage door controller.
3. Use swim lane Petri nets to show the interactions in the garage door controller.

References

Bruyn, W., Jensen, R., Keskar, D. and P. Ward., An extended systems modeling language based on the data flow diagram, *ACM Software Engineering Notes*, Vol. 13, No. 1, 1988, pp. 58–67.

Lane, J.A., *System of Systems Capability-to-Requirements Engineering*, Viterbi School of Engineering, University of Southern California, webinar given February 2012.

Maier, M., Architecting principles for systems-of-systems, *System Engineering*, Vol. 1, No. 4, 1999, pp. 251–315.

Chapter 18

Exploratory Testing

Consider a person admitted to a hospital emergency room (ER) who has trouble breathing. The ER physician is tasked with identifying the underlying problem and then devising a medical response. How does the ER physician proceed? First, a case history of relevant information about the patient is gathered. The next likely step is a few broad-spectrum tests that are intended to eliminate common causes of the breathing difficulty. Knowledge obtained from one test usually leads to follow-up, more specific tests. Throughout this process, the ER physician is guided by extensive experience and domain knowledge. This same pattern applies to software testing in the process known as exploratory testing.

18.1 Exploratory Testing Explored

Andy Tinkham and Cem Kaner present a concise summary of the work (and people) that defined exploratory testing (Tinkham and Kaner, 2003). They identify five essential characteristics of exploratory testing. It is interactive; it involves concurrent cognition and execution; it is highly creative; it intends to produce results quickly; and it reduces the traditional emphasis on formal test documents. The first two characteristics also describe a classic learning system. In testing terms, the exploratory tester learns about the system under test, and uses this new knowledge to explore the system more deeply with more focused tests—very much like the professor giving an oral examination. Because exploratory testing is also highly creative, it is difficult to describe the process precisely. It clearly depends on the attitude and motivation of the tester, but it also depends on the nature of the system under test and on the priorities of the system stakeholders. One quick example: imagine the differing priorities of a highly reliable telephone switching system versus the extreme time-to-market pressure of an e-commerce application.

On the surface, exploratory testing seems to be a sophisticated alias for the special value testing discussed in Chapter 5. Here, pun intended, we explore exploratory testing, but first, we need to explore the metaphor. One sense of "explore" is to investigate the unknown. This sense conjures up images of a scientist in a laboratory, or of a world explorer. Both images are relevant—think about the contributions famous world explorers have made to the world body of knowledge. The Lewis and Clark expedition into the Louisiana Purchase is a good example. Thomas Jefferson wanted to know more about the sizeable chunk of North America that he bought from France. Lewis and Clark assembled a well-rounded team for their expedition, including army personnel, hunters and trappers, craftsmen, naturalists, and a Shoshone woman, Sacajewea, who was familiar with much of the Missouri River basin. More importantly, she could communicate with the native people.

The expedition began with congressional approval and a detailed plan devised by Thomas Jefferson, then the United States President. In a letter dated June 20, 1803, Thomas Jefferson wrote (Jackson, 1978):

> *"The object of your mission is to explore the Missouri river, & such principal stream of it, as, by its course & communication with the waters of the Pacific Ocean, whether the Columbia, Oregan, Colorado or and other river may offer the most direct & practicable water communication across this continent, for the purposes of commerce.*
>
> *Beginning at the mouth of the Missouri, you will take careful observations of latitude & longitude, at all remarkeable points on the river, & especially at the mouths of rivers, at rapids, at islands, & other places & objects distinguished by such natural marks & characters of a durable kind, as that they may with certainty be recognised hereafter. The courses of the river between these points of observation may be supplied by the compass the log-line & by time, corrected by the observations themselves. The variations of the compass too, in different places, should be noticed.*
>
> *The interesting points of the portage between the heads of the Missouri, & of the water offering the best communication with the Pacific ocean, should also be fixed by observation, & the course of that water to the ocean, in the same manner as that of the Missouri.*
>
> *Your observations are to be taken with great pains & accuracy, to be entered distinctly & intelligibly for others as well as yourself, to comprehend all the elements necessary, with the aid of the usual tables, to fix the latitude and longitude of the places at which they were taken, and are to be rendered to the war-office, for the purpose of having the calculations made concurrently by proper persons within the U.S. Several copies of these as well as of your other notes should be made at leisure times, & put into the care of the most trustworthy of your attendants, to guard, by multiplying them, against the accidental losses to which they will be exposed. A further guard would be that one of these copies be on the paper of the birch, as less liable to injury from damp than common paper.*
>
> *The commerce which may be carried on with the people inhabiting the line you will pursue, renders a knowledge of those people important. You will therefore endeavor to make yourself acquainted, as far as a diligent pursuit of your journey shall admit, with the names of the nations & their numbers;*
>
> *the extent & limits of their possessions; their relations with other tribes of nations; their language, traditions, monuments;*
> *their ordinary occupations in agriculture, fishing, hunting, war, arts, & the implements for these;*
> *their food, clothing, & domestic accommodations;*
> *the diseases prevalent among them, & the remedies they use;*
> *moral & physical circumstances which distinguish them from the tribes we know; peculiarities in their laws, customs & dispositions;*
> *and articles of commerce they may need or furnish, & to what extent."*

The Lewis and Clark Expedition lasted 28 months and brought back much detailed information about the region. They arrived at the mouth of the Columbia River in what is now Astoria, Oregon. (Of course, they still had the return trip to complete!) Most people (with the exception of the Native Americans) considered the expedition to be an enormous success. It certainly opened the way for waves of new settlers. The key part, as far as exploratory testing is concerned, is that their implementation of Jefferson's instructions is almost perfectly analogous to the goals and techniques of exploratory testing:

- They knew what they were looking for (a route to the Pacific).
- They had appropriate staffing and other resources.
- They learned as they explored.
- They were given plenty of time.
- They were required to carefully document what they saw.

A second form of exploration illustrates the learning component of exploratory testing; it occurs when a professor gives an oral examination to a student. The first similarity is that the professor clearly has extensive domain knowledge. Second, the professor wishes to explore the extent to which the student has mastered the subject matter. The third, and most instructive, similarity is that, when the student shows a weakness, the professor asks follow-up questions to explore the extent of the weakness; thus, knowledge gained from the answer to one question provokes a related question. This pattern is called adaptive testing.

These forms of exploration help explain the difference between special value testing and exploratory testing. As noted in Chapter 5, special value testing depends on the skill, insight, domain knowledge, and experience of the tester. The tester postulates test cases that "seem to be important" in that they might reveal faults. This is exploration, in a sense, but there is no feedback (other than pass/fail results of test case execution). Most often, the past experience of the tester is the greatest asset, as in the professor who gives an oral examination. By contrast, the exploratory tester is more focused, and has better technology to help discover faults—much like the Lewis and Clark Expedition. The essence of exploratory testing, per James Bach (2003), one of the originators of the term, is "…simultaneous learning, test design, and test execution." Bach agrees that an exploratory tester is most like the professor giving an oral exam. When an exploratory tester encounters suspicious behavior in a program, he will design and execute more tests to isolate the problem. The fault may reside in a normal part of the program that would not be found by special value testing. Results of some tests determine the nature of additional tests. It is the learning aspect that separates special value testing from exploratory testing.

18.2 Exploring a Familiar Example

The commission problem offers a chance to replicate exploratory testing. Suppose we know that a given implementation is faulty; the objective of traditional software testing is to simply verify the presence of faults. Exploratory testing goes a step further, and tries to discover the nature of revealed faults. Recall that the commission problem deals with a salesperson who sells interchangeable rifle parts: locks, stocks, and barrels. The locks cost $45, the stocks cost $30, and the barrels cost $25; so a complete rifle costs $100. When our hypothetical salesperson reports monthly sales, the commission carries an incentive: 10% on the first $1000, 15% on sales between $1001 and $1800, and 20% on sales over $1800.

In the first few months, the neophyte salesperson never exceeds the \$1000 goal, and the commission is correct. When sales finally reach the 15% level, however, the salesperson's commission is less than expected. One month, when sales are nearly \$2000, the commission was slightly more than expected. The inquisitive salesperson began the exploration with four equations (locks, stocks, and barrels are, respectively, the numbers of each item sold):

(1) Sales = 45*locks + 30*stocks + 35*barrels
(2) Commission = 0.10*sales (for $\$0 \leq$ sales $\leq \$1000$)
(3) Commission = \$100 + 0.15*(sales – \$1000) (for $\$1000 <$ sales $\leq \$1800$)
(4) Commission = \$220 + 0.20*(sales – \$1800) (for sales $> \$1800$)

If our progressive salesperson had access to a spreadsheet, she would have seen something like Table 18.1.

What could go wrong with equation (1)? The sales column is correct, but maybe there were off-setting errors in the coefficients. So the first exploration is to see what happens when only one item is sold. The coefficients are clearly correct (Table 18.2).

To explore equation (3), our intrepid salesperson wanted to devise sales near the \$1000 commission incentive point. Conveniently enough (who said this is a contrived example?), the coefficients lend themselves well to this task. The results are in Table 18.3.

Table 18.1 First Exploration

Case No.	*Locks*	*Stocks*	*Barrels*	*Sales*	*Expected Commission*	*Computed Commission*	*Pass?*	*Expected Less Computed*
1	1	1	1	\$100.00	\$10.00	\$10.00	Pass	\$0.00
2	8	8	8	\$800.00	\$80.00	\$80.00	Pass	\$0.00
3	10	10	10	\$1000.00	\$100.00	\$100.00	Pass	\$0.00
4	11	11	11	\$1100.00	\$115.00	\$100.00	Fail	\$15.00
5	17	17	17	\$1700.00	\$205.00	\$190.00	Fail	\$15.00
6	18	18	18	\$1800.00	\$220.00	\$205.00	Fail	\$15.00
7	19	19	19	\$1900.00	\$240.00	\$260.00	Fail	–\$20.00

Table 18.2 Second Exploration

Case No.	*Locks*	*Stocks*	*Barrels*	*Sales*	*Expected Commission*	*Computed Commission*	*Pass?*	*Expected Less Computed*
1	10	0	0	\$450.00	\$45.00	\$45.00	Pass	\$0.00
2	0	10	0	\$300.00	\$30.00	\$30.00	Pass	\$0.00
3	0	0	10	\$250.00	\$25.00	\$25.00	Pass	\$0.00

Table 18.3 Third Exploration

Case No.	*Locks*	*Stocks*	*Barrels*	*Sales*	*Expected Commission*	*Computed Commission*	*Pass?*	*Expected Less Computed*
1	22	0	0	$990.00	$99.00	$99.00	Pass	$0.00
2	21	0	2	$995.00	$99.50	$99.50	Pass	$0.00
3	21	1	1	$1000.00	$100.00	$100.00	Pass	$0.00
4	21	2	0	$1005.00	$100.75	$85.75	Fail	$15.00

Aha!, reasoned our algebraic salesperson, the only thing wrong with equation (3) must be the amount subtracted from sales. Solving the two equations:

$$\text{(3 should be)}\quad \$100.75 = \$100 + 0.15(\$1005 - \$1000)$$

$$\text{(3 computed)}\quad \$85.25 = \$100 + 0.15(\$1005 - x),$$

$$x = \$1100$$

The salesperson reasoned that, in fact, the calculation was

$$\text{(3 faulty)}\quad \text{Commission} = \$100 + 0.15{*}(\text{sales} - \$1100)$$

Notice the exploratory process in this little, and yes, contrived, example. The salesperson had domain knowledge of the commission policy, and investigated what might have been wrong. After eliminating one possibility (incorrect coefficients), tried other tests. Finally, with some algebraic analysis, the salesperson found the problem.

As a footnote, we might imagine that our diligent salesperson reported this fault, and was told, yes, that was the old policy and the commission program had not been updated.

18.3 Observations and Conclusions

James Bach maintains that anyone who tests software does a certain amount of exploratory testing. It is more accurate to say that debugging one's own code is exploratory testing. Because the descriptions of these forms of testing are so general, it is difficult to draw precise conclusions about them. Here are mine:

1. Exploratory testing is appropriate, but difficult, in an agile programming environment. The whole idea of follow-up tests based on the outcome of previous tests presumes a completed application.
2. Exploratory testing is inherently dependent on the domain experience of the tester. By analogy, how effectively could a computer science professor conduct an oral examination for a history major?

3. To be successful, the exploratory tester must be highly motivated, curious, and creative. A dull, disinterested tester will not be able to devise interesting follow-up tests.
4. Exploratory testing defies predictive measurement. This is true of any essentially creative activity, not just software testing. Even a very effective exploratory tester cannot estimate how much additional testing is needed, and it is theoretically impossible to determine how many faults remain. A conscientious exploratory tester can tell when no more faults are being revealed. Imagine the difficulty Meriwether Lewis would have had predicting the date of completion of his expedition.
5. Management of exploratory testing is reduced to insistence on having a clear charter, and requiring documented tests and results.
6. The effectiveness of exploratory testing is inversely proportional to the size and complexity of the system under test. Since exploratory testing does not work well with a team approach, an individual tester will always have some threshold of system comprehension. An exploratory tester can certainly "explore" a large, complex system, but it will be difficult to keep track of all the follow-up tests.

EXERCISES

1. Here is a true story (but the name is changed to protect the guilty):

   Ralph was the project manager of a small telephone switching system development. He started out as an electrical engineer, specifically as a logic designer. As his career progressed, he acquired solid domain knowledge of telephone switching systems. When the project prototype was completed, and when the first increment of software was loaded, Ralph signed up for three hours of scarce system test time. At the end of his session, he called the whole project team together and announced that the system was "full of holes" and that much work remained. When asked for more details, all Ralph could say was that he tried "a bunch of things" and most of them did not work. There was no record of faults found, no indication of the tests that were executed, and nothing was repeatable to help isolate the faults.

   Discuss the ways in which Ralph's testing conforms to and differs from exploratory testing.
2. If you look carefully at Table 18.1, there is another fault, this time in favor of our adjective-laden salesperson. Did she report the second fault, as an honest salesperson, or did she say nothing, and become a greedy salesperson? You can use the exploreCommission.xls spreadsheet to detect the other fault.

References

Bach, J., *Exploratory Testing Explained*, Vol. 1.3, April 2003, available at www.satisfice.com/articles/et-article.pdf.

Jackson, D., ed. *Letters of the Lewis and Clark Expedition with Related Documents*, 2nd ed., University of Illinois Press, Urbana, IL, 1978 (2 volumes).

Kaner, C., *The Context-Driven Approach to Software Testing*, STAR East, Orlando, FL, 2002.

Tinkham, A. and Kaner, C., Exploring exploratory testing, 2003, available at www.testingeducation.org/a/explore.pdf.

Chapter 19

Test-Driven Development

> "Pick a little, talk a little, pick a little, talk a little, pick, pick, pick, talk a lot, pick a little more."
>
> **from Meredith Wilson's musical, *The Music Man!***

If we replace "pick" with "test" and "talk" with "code," the song captures the essence of Test-Driven Development: write tests and code in small, incremental, alternating steps. A test case is written first, and, in the absence of the corresponding code, the test case fails when executed. Immediately, just enough code is written so that the test case will pass. (*Nota bene:* "pass" means the observed outputs are consistent with the expected outputs.) As code size increases, refactoring is permitted, but all the original tests must still pass (otherwise the refactoring is faulty!). Test-Driven Development (or just TDD for short) has become very popular in the agile programming community. Here we take a closer look at the process, using our standby example, the NextDate program.

19.1 Test-Then-Code Cycles

Test-Driven Development has matured to the point where there are both commercial and free tools to support the process. Two of the tenets of Extreme Programming are clearly present in TDD: doing just enough (the "You Aren't Going to Need It" directive) and always having a working, albeit possibly incomplete, version of the program. Take time to follow the example carefully. The convention in the example is that source code in boldface font is what has been added to make the corresponding test case pass. Advocates of TDD are quick to claim that fault isolation is almost trivial in the TDD process. At any point in the process, all previous test cases must have passed. If a new test case fails, the fault can only be in the most recently added code. This is generally true, but it is not immediately obvious that it will always be true for "deeper" faults, such as those revealed only by data flow testing.

Test-Driven Development is guided by a sequence of user stories obtained from the customer/user. Table 19.1 contains the sequence of user stories developed in this section. One of the assumptions of all agile programming variants is that the customer may not know exactly what is desired, and that seeing an implemented (and tested!) part of the eventual application often leads to additional user stories. The whole user story-driven process is very dependent on the order in which stories are given/received. In this example, the stories appear in an extremely bottom–up order—this is ideal for TDD, but it may not happen in practice. There is a companion question of story granularity; this is deferred to Section 19.3.3.

Table 19.1 NextDate User Stories

User Stories
1. The program compiles.
2. A day can be input and displayed.
3. An input month can be displayed.
4. An input year can be displayed.
5. A day below minimum can be detected.
6. A day above maximum can be detected.
7. A month below minimum can be detected.
8. A month above maximum can be detected.
9. A year below minimum can be detected.
10. A year above maximum can be detected.
11. Day = 31 in a 30 day month.
12. Day = 30 in February (note: also need Day = 31 in February).
13. Day = 29 in February in a common year.
14. Day = 29 in February in a leap year.
15. A year divisible by 4 is a leap year (except for century years).
16. A year not divisible by 4 is a common year.
17. A century year not divisible by 400 is a common year.
18. A century year divisible by 400 is a leap year.
19. Increment a non-last day of a month.
20. Increment the last day of a 30-day month.
21. Increment the last day of a 31-day month.
22. Increment December 31.
23. Increment February 28 in a common year.
24. Increment February 28 in a leap year.
25. Increment February 29 in a leap year.

In this sequence of user stories, only the most recent test is shown (to conserve space). In practice, all tests are present and run automatically. Here, the new code added to make the new test pass is in boldface font.

```
User Story 1  The program compiles.
      Input: the NextDate source code.
      Expected Output: OK
  Pseudocode
  Program NextDate
  End NextDate
```

```
User Story 2  A day can be input and displayed
      Input: 15
      Expected Output: day = 15
Pseudocode
Program NextDate
  dim thisDay As Integer
  Input (thisDay)
  Output ("day =", thisDay)
End NextDate
```

```
User Story 3  An input month can be displayed
      Input: 15, 11
      Expected Output: day = 15
            month = 11
Pseudocode
Program NextDate
  dim thisDay As Integer
  dim thisMonth As Integer
  Input (thisDay)
  Output ("day = ", thisDay)
  Input (thisMonth)
  Output ("month = ", thisMonth)
End NextDate
```

```
User Story 4  An input year can be displayed
      Input: 15, 11, 2005
      Expected Output: day = 15
            month = 11
            year = 2005
Pseudocode
Program NextDate
  dim thisDay As Integer
  dim thisMonth As Integer
  dim thisYear As Integer
  Input (thisDay)
  Output ("day =", thisDay)
  Input (thisMonth)
  Output ("month =", thisMonth)
  Input (thisYear)
  Output ("year =", thisYear)
End NextDate
```

Refactor Code (this one is trivial, but it saves space)

```
Program NextDate
  dim thisDay, thisMonth, thisYear As Integer
  Input (thisDay, thisMonth, thisYear)
  Output (“day =”, thisDay, “month =”, thisMonth, “year =”, thisYear)
End NextDate
```

User Story 5 A day below minimum can be detected

```
      Input: 0, 11, 2005
      Expected Output:
      day = 0 is below minimum
      month = 11
      year = 2005
```

Pseudocode

```
Program NextDate
  dim thisDay, thisMonth, thisYear As Integer
  Input (thisDay, thisMonth, thisYear)
  If (thisDay < 1) Then
  Output (“day =”, thisDay, “is below minimum”)
  Output (“month =”, thisMonth, “year =”, thisYear)
  Else
  Output (“day =”, thisDay, “month =”, thisMonth, “year =”,
      thisYear)
  EndIf
End NextDate
```

User Story 6 A day above maximum can be detected

```
      Input: 32, 11, 2005
      Expected Output:
      day = 32 is above maximum
      month = 11
      year = 2005
```

Pseudocode

```
Program NextDate
  dim thisDay, thisMonth, thisYear As Integer
  Input (thisDay, thisMonth, thisYear)
  If (thisDay < 1) Then
  Output (“day =”, thisDay, “is below minimum”)
  Output (“month =”, thisMonth, “year =”, thisYear)
  Else
  Output (“day =”, thisDay, “month =”, thisMonth, “year =”,
      thisYear)
  EndIf
  If (thisDay > 31) Then
  Output (“day =”, thisDay, “is above maximum”)
  Output (“month = “, thisMonth, “year = “, thisYear)
  Else
  Output (“day = “, thisDay, “month = “, thisMonth, “year = “,
      thisYear)
  EndIf
End NextDate
```

Refactor Code (Refactor the sequential tests of thisDay into a nested IF statement.)

```
Program NextDate
  dim thisDay, thisMonth, thisYear As Integer
  Input (thisDay, thisMonth, thisYear)
  If (thisDay > = 1) AND (thisDay < = 31) Then
  Output ("day =", thisDay, "month =", thisMonth, "year =", thisYear)
  Else
      If (thisDay < 1) Then
      Output ("day =", thisDay, "is below minimum")
      Output ("month =", thisMonth, "year =", thisYear)
  EndIf
  If (thisDay > 31) Then
      Output ("day =", thisDay, "is above maximum")
      Output ("month =", thisMonth, "year =", thisYear)
  EndIf
  EndIf
End NextDate
```

User Story 7 A month below minimum can be detected

Input: 15, 0, 2005
Expected Output: day = 15
month = 0 is below minimum
year = 2005

User Story 8 A month above maximum can be detected

Input: 15, 13, 2005
Expected Output: day = 15
month = 13 is above maximum
year = 2005

User Story 9 A year below minimum can be detected

Input: 15, 11, 1811
Expected Output: day = 15
month = 11
year = 1811 is below minimum

User Story 10 A year above maximum can be detected

Input: 15, 11, 2013
Expected Output: day = 15
month = 11
year = 2013 is above maximum

Pseudocode (after adding code similar to that for day validity, and refactoring)

```
Program NextDate
  dim thisDay, thisMonth, thisYear As Integer
  Input (thisDay, thisMonth, thisYear)
  If (thisDay > = 1) AND (thisDay < = 31) Then
      Output (“day =”, thisDay)
  Else
    If (thisDay < 1) Then
      Output (“day =”, thisDay, “is below minimum”)
    Else
      If (thisDay > 31) Then
        Output (“day =”, thisDay, “is above maximum”)
    EndIf
  EndIf
  If (thisMonth > = 1) AND (thisMonth < = 12) Then
      Output (“month =”, thisMonth)
  Else
    If (thisMonth < 1) Then
      Output (“month =”, thisMonth, “is below minimum”)
    Else
      If (thisMonth > 12) Then
        Output (“month =”, thisMonth, “is above maximum”)
    EndIf
  EndIf
  If (thisYear > = 1812) AND (thisYear < = 2012) Then
      Output (“year =”, thisYear)
  Else
    If (thisYear < 1812) Then
      Output (“year =”, thisYear, “is below minimum”)
    Else
      If (thisYear > 2012) Then
          Output (“year =”, thisYear, “is above maximum”)
    EndIf
  EndIf
End NextDate
```

At this point, the input data value ranges have been checked. The next iterations deal with impossible days in a given month. To save space, the data validity checking code is deleted. In TDD practice, of course, it would still be present.

```
User Story 11 Day = 31 in a 30-day month
      Input: 31, 11, 2005
      day = 31 cannot happen when month is 11
      month = 11
      year = 2005
Pseudocode
Program NextDate
  dim thisDay, thisMonth, thisYear As Integer
  Input (thisDay, thisMonth, thisYear)
  ‘data validity checking code would normally be here
  If (thisDay = 31) AND thisMonth IN {2, 4, 6, 9, 11} Then
  Output(“day =”, thisDay, “cannot happen when month is”,
      thisMonth)
  EndIf
End NextDate
```

User Story 12 Day > = 29 in February

```
      Input: 30, 2, 2005
      Expected Output:
      day = 30 cannot happen when month is February
      month = 2
      year = 2005
```

Pseudocode

```
Program NextDate
  dim thisDay, thisMonth, thisYear As Integer
  Input (thisDay, thisMonth, thisYear)
  ‘data validity checking code would normally be here
  If (thisDay = 31) AND thisMonth IN {2, 4, 6, 9, 11} Then
  Output(“day =”, thisDay, “cannot happen when month is”, thisMonth)
  EndIf
  If (thisDay > = 29) AND thisMonth = 2 Then
      Output(“day =”, thisDay, “cannot happen in February”)
  EndIf
End NextDate
```

User Story 13 Day = 29 in February in a common year

```
  Input: 29, 2, 2005
  Expected Output:
  day = 29 cannot happen when month is February in a
  common year
  month = 2
  year = 2005
  day = 29
```

Pseudocode

```
Program NextDate
  dim thisDay, thisMonth, thisYear As Integer
  Input (thisDay, thisMonth, thisYear)
  ‘data validity checking code would normally be here
  If (thisDay = 31) AND thisMonth IN {2, 4, 6, 9, 11} Then
  Output(“day =”, thisDay, “cannot happen when month is”, thisMonth)
  EndIf
  If (thisDay > = 29) AND thisMonth = 2 Then
  Output(“day =”, thisDay, “cannot happen in February”)
  EndIf
  ‘Note: isLeap is a Boolean function that returns true when the
  ‘argument corresponds to a leap year. Cannot run this test case until
  ‘Function isLeap is tested.
  If (thisDay = 29) AND thisMonth = 2 AND NOT(isLeap(this year))
  Then Output(“day =”, thisDay, “cannot happen in February in a
      common year”)
  EndIf
End NextDate
```

An anomaly occurs here. The developer postulated a Boolean function, isLeap, that responds with True or False depending on whether the input year is a leap or a common year. This function would have to be tested before the test "current" case. Possibly, this would be another point of refactoring. Function isLeap is the subject of the next few user stories.

User Story 14 `A year divisible by 4 is a leap year (except for century years)`

```
  Input: 2004
  Expected Output: True
```

Pseudocode

```
Function isLeap(year) As Boolean
  dim year As Integer
  '1812 < = year < = 2012 is given, and tested in main program
  isLeap = False
  'MOD is the modulo arithmetic built-in operator in most languages
  If ((year MOD 4) = 0) Then
  IsLeap = True
  EndIf
End isLeap
```

User Story 15 `A year not divisible by 4 is a common year.`

```
  Input: 2005
  Expected Output: False
```

Pseudocode

```
Function isLeap(year) As Boolean
  dim year As Integer
  '1812 < = year < = 2012 is given, and tested in main program
  isLeap = False
  'MOD is the modulus built-in operator in most languages
  If ((year MOD 4) = 0) Then
  IsLeap = True
  Else
      IsLeap = False
  EndIf
End isLeap
```

User Story 16 `A century year not divisible by 400 is a common year.`

```
  Input: 1900
  Expected Output: False
```

Pseudocode

```
Function isLeap(year) As Boolean
  dim year As Integer
  '1812 < = year < = 2012 is given, and tested in main program
  isLeap = False
  'MOD is the modulus built-in operator in most languages
  If (((year MOD 4) = 0) AND NOT(year MOD 100 = 0)) Then
      IsLeap = True
  Else
      IsLeap = False
  EndIf
End isLeap
```

```
User Story 17 A century year divisible by 400 is a leap year.
  Input: 2000
  Expected Output: True
Pseudocode
Function isLeap(year) As Boolean
  dim year As Integer
  ‘1812 < = year < = 2012 is given, and tested in main program
  isLeap = False
  ‘MOD is the modulus built-in operator in most languages
  If (((year MOD 4) = 0) AND NOT(year MOD 100 = 0)) OR
           ((year MOD 400) = 0) Then
      IsLeap = True
  Else
      IsLeap = False
  EndIf
End isLeap
```

Comment: TDD shows a clear advantage in the development of Function isLeap. In a classroom experiment, only a few students were able to code the full condition (in user story 17) directly from the definition. The TDD build-up nicely simplifies this somewhat confusing condition. Now the user stories return to date validity tests.

```
User Story 18 Day = 29 in February in a leap year
  Input: 29, 2, 2004
  Expected Output: day = 1
      month = 3
      year = 2004
Pseudocode
Program NextDate
  dim thisDay, thisMonth, thisYear As Integer
  Input (thisDay, thisMonth, thisYear)
  ‘data validity checking code would normally be here
  If (thisDay = 31) AND thisMonth IN {2, 4, 6, 9, 11} Then
      Output(“day =”, thisDay, “cannot happen when month is”, thisMonth)
  EndIf
  If (thisDay = 30) AND thisMonth = 2 Then
      Output(“day =”, thisDay, “cannot happen in February”)
  EndIf
  If (thisDay = 29) AND thisMonth = 2 AND NOT(isLeap(this year))
      Then Output(“day =”, thisDay, “cannot happen in February in a
           common year”)
  Else
      Output(day = 1, month = 3, year = this year)
  EndIf
End NextDate
```

The first 10 user stories checked to see that values of day, month, and year are in the appropriate ranges. User stories 11 to 18 deal with valid and impossible dates. The remaining user stories deal with correct date increments. By now, the basic "test a little, code a little" principle should be clear. The remaining user stories are shown in Section 19.3.3 where the question of user story granularity is discussed.

19.2 Automated Test Execution (Testing Frameworks)

Test-Driven Development depends on an environment in which it is easy to postulate and run tests. To facilitate TDD, test execution frameworks have been written for most mainline programming languages. Most of these environments require the tester to write a test driver program that contains the actual test case data—both inputs and expected outputs. Section 19.3 contains a Java/JUnit example. Here is a partial list of TDD frameworks for various programming languages from Wikipedia (http://en.wikipedia.org/wiki/XUnit) that demonstrates the variety of languages for which TDD frameworks are available.

AUnit—a unit testing framework for Ada programming language
AsUnit for ActionScript
AS2Unit—a unit testing framework for ActionScript2.0
As2libUnitTest—a unit testing framework for ActionScript2.0
CUnit—a unit testing framework for C
CuTest—a cute unit testing framework for C
CFUnit—a unit testing framework for ColdFusion
CPPUnit—a unit testing framework for C++
csUnit—a unit testing framework for the .NET programming languages
DBUnit—a unit testing framework for databases as a JUnit extension
DUnit—a unit testing module for Delphi
FoxUnit—a unit testing framework for Visual FoxPro
FRUIT—FORTRAN Unit Testing Framework
fUnit—a unit testing framework for Fortran
FUTS—the Framework for Unit Testing SAS
GUnit—a unit testing framework for C with GNOME support
HttpUnit—testing framework for Web applications, typically used in combination with JUnit
jsUnit—a unit testing framework for client-side (in-browser) JavaScript
JUnit—a unit testing framework for Java
JUnitEE—a unit testing framework for JavaEE
MbUnit—a unit testing framework for Microsoft.NET
NUnit for Microsoft.NET
ObjcUnit—JUnit-like unit testing framework for Objective-C
OCUnit—a unit testing framework for Objective-C
OUnit—a unit testing framework for Ocaml
PHPUnit—a unit testing framework for PHP
PyUnit—a unit testing module for Python
RBUnit—a unit testing framework for REALbasic

SimpleTest for PHP
SUnit—a unit testing framework for Smalltalk (the original xUnit framework)
Test::Class—another unit testing module for Perl
Test::Unit—a unit testing module for Perl
Test::Unit—a unit testing module for Ruby
Testoob—an extended testing framework for use with PyUnit
TSQLUnit—a unit testing framework for Transact-SQL
VbaUnit—a unit testing framework for Visual Basic for Applications
VbUnit—a unit testing framework for Visual Basic

19.3 Java and JUnit Example

The JUnit program is typical of the TDD test frameworks. Here is the Java code that corresponds to most of the example in Section 19.2.

19.3.1 Java Source Code

```
//class ValidDate checks if a date is correct, by Dr. Christian Trefftz
public class ValidDate

{public static boolean isLeap(int year)
       {if (((year%4) ==0) && !((year%100) ==0) || ((year%400) ==0))
           return true;
       else
           return false;}
   //validRangeForDay will return true if the parameter thisDay is in the
valid range
   public static boolean validRangeForDay(int thisDay)
       {if ((thisDay > = 1) && (thisDay < = 31))
      {System.out.println(“Day = “+thisDay);
          return true;}
       else {if (thisDay < 1)
            {System.out.println(“Day = “+thisDay+” is below minimum.”);
                   return false;}
               else
                      if (thisDay > 31)
                {System.out.println(“Day = “+thisDay+” is above maximum.”);
                       return false;}
               }
       return false;}
//validRangeForMonth will return true if the parameter thisMonth is in
the valid range
  public static boolean validRangeForMonth(int thisMonth) {
       if ((thisMonth > = 1) && (thisMonth < = 12))
     {System.out.println(“Month = “+thisMonth);
          return true;}
       else
     {if (thisMonth < 1)
            {System.out.println(“Month = “+thisMonth+” is below minimum.”);
                    return false;}
          else
               if (thisMonth > 12)
            {System.out.println(“Month = “+thisMonth+” is above maximum.”);
                        return false;}
            }
       return false;}
  //validRangeForYear will return true if the parameter thisYear is in
the valid range
  public static boolean validRangeForYear(int thisYear) {
      if ((thisYear > = 1812) && (thisYear < = 2012))
     {System.out.println(“Year = “+thisYear);
      return true;}
      else
         {if (thisYear < 1812) {
              System.out.println(“Year = “+thisYear+” is below minimum.”);
             return false;}
         else
             if (thisYear > 2012)
                 {System.out.println(“Year = “+thisYear+” is above maximum.”);
                 return false;}
       }
       return false;}
  //validCombination will return true if the parameters are a valid combination
  public static boolean validCombination(int thisDay,int thisMonth,int
thisYear){
        if ((thisDay = = 31) && ((thisMonth = = 2) || (thisMonth = =4) ||
(thisMonth = = 6) || (thisMonth = =9) || (thisMonth = = 11)))
       {System.out.println(“Day = “+thisDay+” cannot happen when month is
“+thisMonth);
            return false;}
         if ((thisDay = = 30) && (thisMonth = = 2))
            {System.out.println(“Day = “+thisDay+” cannot happen in February”);
            return false;}
         if ((thisDay = = 29) && (thisMonth = = 2) && !(isLeap(thisYear)))
            {System.out.println(“Day = “+thisDay+” cannot happen in February.”);
            return false;}
       return true;}
//validDate will return true if the combination of the parameters is valid
public static boolean validDate(int thisDay,int thisMonth,int thisYear)
       {if (!validRangeForDay(thisDay))
           {return false;}
       if (!validRangeForMonth(thisMonth))
           {return false;}
       if (!validRangeForYear(thisYear))
           {return false;}
       if (!validCombination(thisDay,thisMonth,thisYear)) {
           return false;}
       //If this point is reached, the date is valid
       return true;}}
```

19.3.2 *JUnit Test Code*

To test a Java unit, the tester must first write a test program such as the one that follows. This establishes a connection between the Java unit to be tested and the JUnit framework. Actual test cases use the assertEquals method, where an assertion contains the pass/fail result of executing the called unit with test case values. For example, the assertion

```
assertEquals(true, ValidDate.validDate(29, 2, 2000));
```

asks JUnit to run the validDate method of ValidDate with the test case corresponding to February 29, 2000. This is a valid date, and the expected JUnit response is "true." Similarly, the assertion below tests an invalid February date.

```
assertEquals(false, ValidDate.validDate(29, 2, 2001));
```

Here is the actual JUnit test code.

```
//The test class ValidDateTest, by Dr. Christian Trefftz
public class ValidDateTest extends junit.framework.TestCase
{
    //Default constructor for test class ValidDateTest
    public ValidDateTest()
    {}
    //Sets up the test fixture. Called before every test case method.
    protected void setUp()
    {}
    //Tears down the test fixture. Called after every test case method.
    protected void tearDown()
    {}
     public void testIsLeap()
            {assertEquals(true, ValidDate.isLeap(2000));
            assertEquals(false, ValidDate.isLeap(1900));
            assertEquals(false, ValidDate.isLeap(1999));}
      public void testValidRangeForDay()
            {assertEquals(false, ValidDate.validRangeForDay(-1));
            assertEquals(false, ValidDate.validRangeForDay(32));
            assertEquals(true, ValidDate.validRangeForDay(20));}
      public void testValidRangeForMonth()
            {assertEquals(false, ValidDate.validRangeForMonth(0));
            assertEquals(false, ValidDate.validRangeForMonth(13));
            assertEquals(true, ValidDate.validRangeForMonth(6));}
      public void testValidRangeForYear()
            {assertEquals(false, ValidDate.validRangeForYear(1811));
            assertEquals(false, ValidDate.validRangeForYear(2013));
            assertEquals(true, ValidDate.validRangeForYear(1960));}
      public void testValidCombination()
            {assertEquals(false, ValidDate.validCombination(31, 4, 1960));
            assertEquals(false, ValidDate.validCombination(29, 2, 2001));
            assertEquals(true, ValidDate.validCombination(29, 2, 2000));
            assertEquals(true, ValidDate.validCombination(28, 2, 2001));}
      public void testValidDate()
            {assertEquals(true, ValidDate.validDate(29, 2, 2000));
```

```
          assertEquals(false, ValidDate.validDate(29, 2, 2001));
          assertEquals(true, ValidDate.validDate(11, 10, 2006));
          assertEquals(false, ValidDate.validDate(04, 30, 1960));
          assertEquals(true, ValidDate.validDate(30, 04, 1960));}
     }
```

19.4 Remaining Questions

19.4.1 Specification or Code Based?

Is TDD code based, or specification based? In a sense, a test case is a very low level specification, so TDD seems to be specification based. However, test cases are very closely associated with code, so it has the appearance of code-based testing. Certainly, code coverage, at least at the DD-path level, is unavoidable. Is it a stretch to claim that the set of all test cases constitutes a requirements specification? Imagine the reaction of a customer trying to understand a TDD program from the set of test cases. In the agile programming sense, however, the purpose of each test case can be considered to be a user story, and user stories are accepted by customers. It is really a question of level of detail, and this leads to a variant of TDD. Practitioners who object to tiny, incremental steps suggest that "larger" test cases, followed by larger chunks of code, are preferable. This has the advantage of introducing a small element of code design, and probably reduces the frequency of refactoring. Then the strictly bottom–up approach of "pure" TDD is complemented by top–down thinking.

19.4.2 Configuration Management?

Superficially, TDD appears to be a configuration management nightmare. Even a program as small as NextDate has dozens of versions in its growth from inception to completion. This is where refactoring comes in. TDD forces a bottom–up approach to code development. At certain points, the conscientious programmer will see that the code can be reorganized into something more elegant. There are no rules as to when refactoring should occur, but when it does, it is important to note that the original test cases are preserved. If the refactored code fails to pass all tests, there is a problem in the refactoring. Again, note the simple fault isolation. Refactoring points (once all test cases have passed) are good candidates for configuration management actions. These are points where a design object is, or can be, promoted to configuration item status. If later code causes earlier test cases to fail, this is another clear configuration management point. The configuration item should be demoted to a design object, which by definition, is subject to change.

19.4.3 Granularity?

The sequence of user stories in the example in Section 19.3.1 uses very fine-grained level of detail. As an alternative, consider the enlarged granularity of user stories in Table 19.2. With "larger" user stories, a particular user story is broken down to a series of finer tasks, and code is developed for

Table 19.2 User Story Granularity

Large-Grain User Stories	*Fine-Grain User Stories*
1. The program compiles.	1. The program compiles.
2. A date can be input and displayed.	2. A day can be input and displayed.
	3. An input month can be displayed.
	4. An input year can be displayed.
3. Invalid days can be recognized.	5. A day below minimum can be detected.
	6. A day above maximum can be detected.
4. Invalid months can be recognized.	7. A month below minimum can be detected.
	8. A month above maximum can be detected.
5. Invalid years can be recognized.	9. A year below minimum can be detected.
	10. A year above maximum can be detected.
6. Invalid dates can be recognized.	11. Day = 31 in a 30 day month.
	12. Day > = 29 in February.
	13. Day = 29 in February in a common year.
	14. Day = 29 in February in a leap year.
7. Leap years can be recognized.	15. A year divisible by 4 is a leap year.
	16. A year not divisible by 4 is a common year.
	17. A century year not divisible by 400 is a common year.
	18. A century year divisible by 400 is a leap year.
8. Valid dates can be incremented.	19. Increment a non-last day of a month.
	20. Increment the last day of a 30-day month.
	21. Increment the last day of a 31-day month.
	22. Increment December 31.
	23. Increment February 28 in a common year.
	24. Increment February 28 in a leap year.
	25. Increment February 29 in a leap year.

each task. In this way, the fault isolation is preserved. To distinguish between these granularity choices, sometimes the larger version is named "story-driven development."

19.5 Pros, Cons, and Open Questions of TDD

As with most innovations, TDD has its advantages, disadvantages, claims, and unanswered questions. The advantages of TDD are very clear. Owing to the extremely tight test/code cycles, something always works. In turn, this means a TDD project can be turned over to someone else, likely a programming pair, for continued development. Probably the biggest advantage of TDD is the excellent fault isolation. If a test fails, the cause must be the most recently added code. Finally, TDD is supported by an extensive variety of test frameworks, including those listed in Section 19.2.

It is nearly impossible, or at best, very cumbersome, to perform TDD in the absence of test frameworks. There really is not much of an excuse for this because the frameworks are readily available for most programming languages. If a tester cannot find a test framework for the project language, TDD is a poor choice. (It is probably better to just change programming languages.) At a deeper level, TDD is inevitably dependent on the ingenuity of the tester. Good test cases are necessary but not sufficient for TDD to produce good code. Part of the reason is that the bottom–up nature of TDD provides little opportunity for elegant design. TDD advocates respond by claiming that a good design is eventually accomplished by a series of refactorings, each of which improves the code a little bit. A final disadvantage of TDD is that the bottom–up process makes it unlikely that "deeper faults," such as those only revealed by data flow testing, will be revealed by the incrementally created test cases. These faults require a more comprehensive understanding of the code, and this disadvantage is exacerbated by the possibility of the thread interaction faults discussed in Chapter 17.

Any new technology or technique has a set of open questions, and this is certainly true for TDD. The easiest question is that of scale-up to large applications. It would seem that there are practical limits as to how much an individual can "keep in mind" during a development. This is one of the early motivating factors for program modularity and information hiding, which are the foundations of the object-oriented paradigm. If size is a problem, complexity is even more serious. Can systems developed with TDD effectively deal with questions such as reliability and safety? Such questions usually require sophisticated models, but these are not produced in TDD. Finally, there is the question of support for long-term maintenance. The agile programming community and the TDD advocates maintain that there is no need for the documentation produced by the more traditional development approaches. The more extreme advocates even argue against comments in source code. Their view: the test cases *are* the specification, and well-written code, with meaningful variable and method names, is self-documenting. Time will tell.

19.6 Retrospective on MDD versus TDD

The Northern Cheyenne people of the North American plains have teaching stories based on what they observe in nature. When they speak of the Medicine Wheel, they associate animals with each of the four directions, and the animals have qualities that are seen in nature. One interesting pair is the Eagle and the Mouse. The Eagle sees the "big picture" and therefore understands the important relationships among things. The Mouse, on the other hand, sees only the ground where it scurries, and the grasses it encounters—a very detailed view. Living by the Medicine Wheel means that each view is honored—each view is needed to have better understanding.

It is unlikely that the Northern Cheyenne ever thought much about Model-Driven Development (MDD) and TDD, but the lessons are obvious: both are needed to have better understanding, in this case, of a program to be developed. This really is not too surprising. In the 1970s and 1980s, camps in the software community passionately debated the merits of specification-based versus code-based testing. Thoughtful people soon concluded that some blend of both approaches is necessary. To illustrate these two approaches, consider our Boolean function, isLeap, that determines whether a given year is a common or a leap year.

A model-driven approach to developing isLeap would likely begin with a decision table (Table 19.3) showing the relationships among the phrases of the definition.

The advantage of using a decision table for the model is that it is complete, consistent, and not redundant. Rule r1 refers to century years that are leap years, while rule r2 refers to century years that are common years. Rule r3 describes non-century leap years, and rule r8 describes non-century common years. The other rules are logically impossible. If we write isLeap from this decision table, we would get something like the Visual Basic function in Figure 19.1.

Notice that there are four paths from the source node to the sink node. The path through node 8 corresponds to rule r1, the one through nodes 9 and 10 to rule r2, and so on. Coding nested If logic three levels deep is probably not what the average developer would do, at least not on the first try. (And it is even less likely that a developer would get it correct on the first try. Score one for MDD.)

The test-driven approach results in a different form of complexity. Referring to the code for user stories 14 through 17 on Section 19.1, notice that the TDD code gradually developed a compound If statement, rather than the nested If logic in the MDD version (slightly refactored again in Figure 19.2).

Table 19.3 Leap Year Decision Table

Conditions	*r1*	*r2*	*r3*	*r4*	*r5*	*r6*	*r7*	*r8*
c1. year is a multiple of 4	T	T	T	T	F	F	F	F
c2. year is a century year	T	T	F	F	T	T	F	F
c3. year is a multiple of 400	T	F	T	F	T	F	T	F
Actions								
Logically impossible			×		×	×	×	
a1. year is a common year		×						×
a2. year is a leap year	×			×				
Test case: year =	2000	1900		2008				2011

```
Public function isLeap(year) As Boolean
   Dim c1, c2, c3 As Boolean
   Dim year As Integer
1    isLeap = False
2    c1 = (year Mod 4 = 0)         ' leap years are divisible by 4
3    c2 = (year Mod 100 = 0)       ' but century years are common years
4    c3 = (year Mod 400 = 0)       ' unless they are divisible by 400
5    If c1 Then
6          If c2 Then
7                If c3 Then
8                      isLeap = True          'rule r1
9                Else
10                     isLeap = False         'rule r2
11               End If
12         Else
13               isLeap = True                'rule r4
14         End If
15    Else
16         isLeap = False                     'rule r8
17    End If
End Function
```

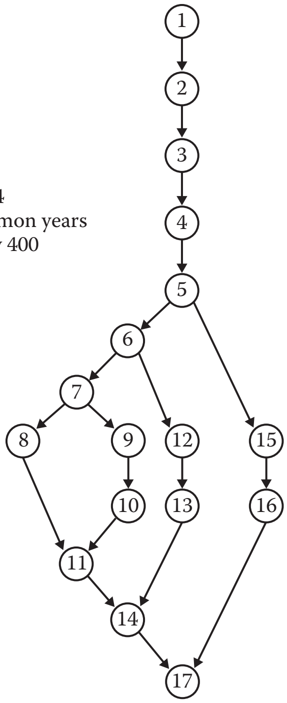


Figure 19.1 MDD version of isLeap.

```
Public function isLeap(year) As Boolean
   Dim c1, c2, c3 As Boolean
   Dim year As Integer

1  c1 = (year Mod 4 = 0)
2  c2 = (year Mod 100 = 0)
3  c3 = (year Mod 400 = 0)
4  isLeap = False
5  If ((c1 AND NOT(c2)) OR (c3)) Then
6           isLeap = True
7  End If
8  End Function
```

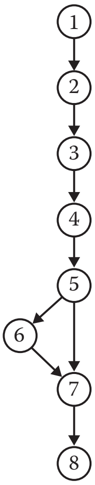


Figure 19.2 TDD version of isLeap.

As a cross check, here is the truth table for the compound condition.

```
(c1 AND NOT(c2)) OR (c3)
```

c1	*c2*	*c3*	*NOT(c2)*	*c1 AND NOT(c2)*	*(c1 AND NOT(c2)) OR c3*	*Year*
T	T	T	F	F	T	2000
T	T	F	F	F	F	1900
T	F	T	T	T	T	Imp
T	F	F	T	T	T	2008
F	T	T	F	F	T	Imp
F	T	F	F	F	F	Imp
F	F	T	T	F	T	Imp
F	F	F	T	F	F	2011

Notice that the same test cases and impossibilities (the “Imp” entries) occur in the rows of the truth table, and the columns of the decision table; therefore, the two versions of isLeap are logically equivalent. Looking at the program graphs of the two implementations, the MDD version seems to be more complex. In fact, the cyclomatic complexity of the MDD version is 4, while that of the TDD version is only 2. From a testing standpoint, however, the compound condition in the TDD version requires multiple condition coverage. Both versions end up with the same necessary (and sufficient) four test cases.

What, if any, conclusions can we draw from this? The MDD approach yields the Eagle’s view of the full picture. We know from the way decision tables work that the result is correct. We had to do a little more work to reach the same level of confidence with the TDD approach; however, in the end, the two implementations are logically equivalent. The apparent difference in cyclomatic complexity is negated by the need for multiple condition coverage testing. The nested If complexity is moved into condition complexity—it does not disappear.

Any weaknesses? The MDD approach ultimately depends on the modeling skill; similarly, the TDD approach depends on testing skill. No significant difference there. What about size? The MDD version is longer: 17 statement fragments versus 9, but the TDD process requires more keystrokes. No significant difference here either.

The biggest difference would seem to be maintenance. Presumably, the modeling would be more helpful to a maintainer—the Eagle again. But the test cases from the TDD approach will help the maintainer recreate and isolate a fault—the Mouse view.

Chapter 20

A Closer Look at All Pairs Testing

When it was first introduced, the All Pairs testing possibility was extremely popular. According to James Bach, more than 40 journal articles and conference papers have been written about the technique (Bach and Schroeder, 2003). It continues to be discussed in recent books on software testing, it is in the ISTQB Advanced Level syllabus, and the practitioner conferences continue to offer tutorials on All Pairs testing. It is tempting to say that more has been written about All Pairs testing than is known. In this chapter, as the title implies, we take a closer look at the All Pairs testing technique, answering these questions:

- What is the All Pairs technique?
- Why is it so popular?
- When does it work well?
- When is it not appropriate?

The chapter ends with recommendations for appropriate use.

20.1 The All Pairs Technique

The All Pairs testing technique has its origins in statistical design of experiments. There, orthogonal arrays are a means of generating all pairs of experimental variables such that each pair occurs with equal probability. Mathematically, the statistical technique derives from Latin Squares (Mandl, 1985). The NIST papers by Wallace and Kuhn (2000, 2001) captured the attention of the software development community, particularly the agile community. The papers concluded that 98% of the defects in software-controlled medical systems were due to the interaction of pairs of variables.

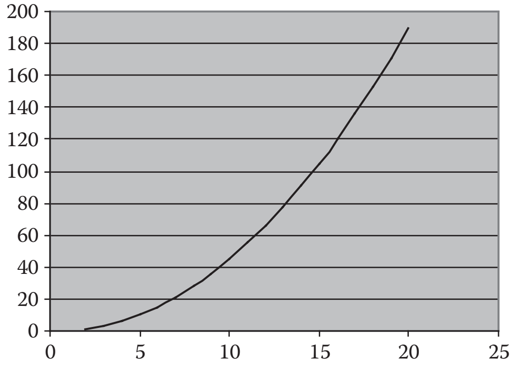


Figure 20.1 Combinatorial explosion.

Given a program with *n* input variables, the All Pairs technique is a way to identify each pair. Mathematically, this is commonly called the number of combinations of *n* things taken two at a time, and is computed by the formula

$$_nC_2 = (n!)/((2!)(n-2)!)$$

which is the basis for the well-known "combinatorial explosion." The first 20 values of $_nC_2$ are graphed in Figure 20.1. With the All Pairs technique, for example, the 66 pairs of interactions among 12 variables are exercised in a single test case.

Perhaps the most commonly cited example of All Pairs testing was developed by Bernie Berger and presented at the STAREast conference in 2003 (Berger, 2003). His paper contains a mortgage application example that has 12 input variables. (In a private e-mail, he said that "12" is a simplification.) Berger identified equivalence classes for the 12 variables, varying in number between seven classes for two variables to two classes for six variables. The cross product of the equivalence classes results in 725,760 test cases. Applying the All Pairs technique, this is reduced to 50 test cases—quite a reduction.

The All Pairs technique is supported by a commercial tool, the Automatic Efficient Test Generator (AETG) system (Cohen, 1994). It is also supported by a free program that is available from James Bach at his website (http://www.satisfice.com). The technique makes the following assumptions:

- Meaningful equivalence classes can be identified for each program input.
- Program inputs are independent.
- There is no order to program inputs.
- Faults are due only to the interaction of pairs of program inputs.

The necessity of each assumption is demonstrated (with counter-examples) next.

20.1.1 Program Inputs

As we have seen in earlier chapters, program inputs can be either events or data. The All Pairs technique refers only to data; that is, inputs are values of variables, not events. It is useful to distinguish between physical and logical variables, as in Chapter 10, Table 10.13. As a guideline, physical variables are usually associated with some unit of measure, such as velocity, altitude, temperature, or mass. Logical variables are seldom associated with units of measure; instead, they usually refer to some enumerated type, such as a telephone directory number or an employee identification number. It is usually easier to identify equivalence classes for logical variables.

As a counter-example, consider the triangle program. The three sides, a, b, and c, are all integers and are arbitrarily bounded by $1 \le \text{side} \le 200$. The sides are physical variables, measured in some unit of length. What equivalence classes apply to a, b, and c? Only the robust equivalence classes that deal with valid and invalid input values of a side:

EqClass1(side) = $\{x$: x is an integer and $x < 1\}$ (invalid values)
EqClass2(side) = $\{x$: x is an integer and $1 \le x \le 200\}$ (valid values)
EqClass3(side) = $\{x$: x is an integer and $x > 200\}$ (invalid values)

The actual Notepad input file to Bach's allpairs.exe program is

```
side a    side b    side c
a < 1    b < 1    c < 1
1 ≤ a ≤ 200    1 ≤ b ≤ 200    1 ≤ c ≤ 200
a > 200    b > 200    c > 200
```

An interested tester might postulate equivalence classes such as one in which exactly two sides are equal, but such classes are on triples of triangle program inputs, not on individual variables. Table 20.1 contains the allpairs.exe output generated for these equivalence classes; the actual test cases are in Table 20.2.

As expected from the allpairs.exe output, there is never an opportunity to choose values for the sides that correspond to an actual triangle. Because six of the nine equivalence classes deal with invalid values, this only exercises data validity, not correct function with valid values.

Table 20.1 Allpairs.exe Output

Case	*Side a*	*Side b*	*Side c*	*Pairings*
1	a < 1	b < 1	c < 1	3
2	a < 1	1 ≤ b ≤ 200	1 ≤ c ≤ 200	3
3	a < 1	b > 200	c > 200	3
4	1 ≤ a ≤ 200	b < 1	1 ≤ c ≤ 200	3
5	1 ≤ a ≤ 200	1 ≤ b ≤ 200	c < 1	3
6	1 ≤ a ≤ 200	b > 200	c < 1	2
7	a > 200	b < 1	c > 200	3
8	a > 200	1 ≤ b ≤ 200	c < 1	2
9	a > 200	b > 200	1 ≤ c ≤ 200	3
10	1 ≤ a ≤ 200	1 ≤ b ≤ 200	c > 200	2

Table 20.2 Triangle Program Test Cases Generated by Allpairs.exe

Case	*Side a*	*Side b*	*Side c*	*Expected Output*
1	–3	–2	–4	Not a triangle
2	–3	5	7	Not a triangle
3	–3	201	205	Not a triangle
4	6	–2	9	Not a triangle
5	6	5	–4	Not a triangle
6	6	201	–4	Not a triangle
7	208	–2	205	Not a triangle
8	208	5	–4	Not a triangle
9	208	201	7	Not a triangle
10	6	5	205	Not a triangle

20.1.2 Independent Variables

The NextDate function violates the independent variables assumption. There are dependencies between the day and month variables (a 30-day month cannot have day = 31) and between month and year (the last day of February depends on whether the year is leap or common). The day, month, and year variables are logical variables, and they are amenable to useful equivalence classes. In Chapter 6, we had the following equivalence classes and we used a decision table to deal with the dependencies. Table 20.3 is an extended entry decision table; it is the result of algebraically reducing the complete decision table in Chapter 6. It is "canonical" in the sense that it exactly

Table 20.3 Canonical Decision Table of Valid NextDate Variables

Rules	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
Day	D6	D4	D7	D5	D7	D5	D1	D2	D2	D3
Month	M1	M1	M2	M2	M3	M3	M4	M4	M4	M4
Year	—	—	—	—	—	—	—	Y1	Y2	Y2
Day = 1		×		×		×		×		×
Day++	×		×		×		×		×	
Month = 1						×				×
Month++		×		×				×		
Year++						×				

represents all the combinations of valid variable values. The dependencies among day, month, and year are all expressed in the canonical decision table for NextDate.

The base equivalence classes from Chapter 6 are repeated here:

For day:
  D1 = {1 ≤ day ≤ 27}
  D2 = {28}
  D3 = {29}
  D4 = {30}
  D5 = {31}
For month:
  M1 = {30-day months}
  M2 = {31-day months except December}
  M3 = {December}
  M4 = {February}
For year:
  Y1 = {common years}
  Y2 = {leap years}

Table 20.3 shows the result of combining rules from a complete extended entry decision table with the day equivalence classes

D6 = D1 ∪ D2 ∪ D3 = {1 ≤ day ≤ 29}
D7 = D1 ∪ D2 ∪ D3 ∪ D4 = {1 ≤ day ≤ 30}

The allpairs.exe test cases for NextDate are given in Table 20.4. Note that the 10 canonical test cases are only partly present in the 20 All Pairs test cases. Since the All Pairs algorithm does not merge decision table rules, some of the generated test cases correspond to a single rule in the canonical decision table. For example, All Pairs test cases 1, 3, and 15 all correspond to rule 1; cases 2, 4, 16, and 18 correspond to rule 3; and cases 6, 8, 12, and 14 correspond to rule 5. The redundancy is understandable. The more serious problems are the missing test case (for rule 8) and the invalid test cases (cases 7, 9, and 19). The missing test case consists of the interaction of all three variables, so the All Pairs algorithm cannot be expected to find this one. The invalid test cases are all due to dependencies among pairs of variables; these demonstrate the necessity of the independent variable assumption.

20.1.3 Input Order

Applications that use a Graphical User Interface (GUI) frequently allow inputs to be entered in any order. Figure 20.2 is a simple GUI for a simplified currency converter. A user can enter a whole US dollar amount up to $10,000, select one of three currencies, and then click on the Compute button to display the equivalent amount in the selected currency. The Clear All button can be clicked at any time; it resets the US dollar amount and resets any selected currency. Once a US dollar amount has been entered, a user may perform a series of currency conversions by first selecting a currency type, then clicking on Compute, then repeating this sequence for other currencies. The Quit button ends the application.

Table 20.4 All Pairs Test Cases for NextDate

Case	*Day*	*Month*	*Year*	*Pairings*	*Valid?*	*DT Rule*
1	1–27	30-day	Leap	3	Yes	1
2	1–27	31-day	Common	3	Yes	3
3	28	30-day	Common	3	Yes	1
4	28	31-day	Leap	3	Yes	3
5	29	February	Leap	3	Yes	10
6	29	December	Common	3	Yes	5
7	30	February	Common	3	No	
8	30	December	Leap	3	Yes	5
9	31	30-day	Leap	2	No	
10	31	31-day	Common	2	Yes	4
11	1–27	February	~Leap	1	Yes	7
12	1–27	December	~Common	1	Yes	5
13	28	February	~Common	1	Yes	9
14	28	December	~Leap	1	Yes	5
15	29	30-day	~Common	1	Yes	1
16	29	31-day	~Leap	1	Yes	3
17	30	30-day	~Leap	1	Yes	2
18	30	31-day	~Common	1	Yes	3
19	31	February	~Leap	1	No	
20	31	December	~Common	1	Yes	6

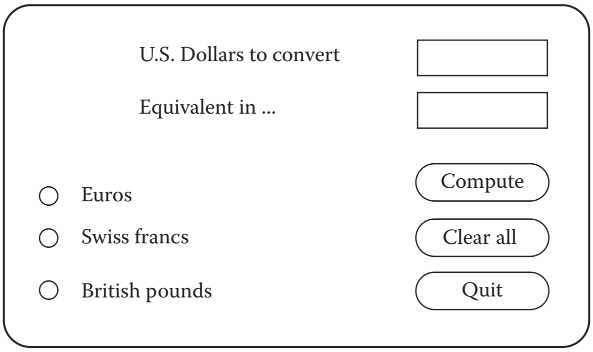

U.S. Dollars to convert
Equivalent in ...
○ Euros
○ Swiss francs
○ British pounds
Compute
Clear all
Quit

Figure 20.2 Currency Conversion GUI.

Because there is no control on the sequence of user input events, the Compute button must anticipate invalid user input sequences. It produces five error messages:

Error message 1: No US dollar amount entered
Error message 2: No currency selected
Error message 3: No US dollar amount entered and no currency selected
Error message 4: US dollar amount cannot be negative
Error message 5: US dollar amount cannot be greater than $10,000

Clicking on the Compute button is therefore a context-sensitive input event, with six contexts—the five that result in the error messages, and an input US dollar amount in the valid range. The data contexts of an input event are clearly pairs of interest to a tester, so the All Pairs technique should be appropriate.

At first glance, the Currency Conversion GUI seems to lend itself nicely to the All Pairs technique. The following equivalence classes are derived naturally from the description and are shown in Table 20.5:

USdollar1 = {No entry}
USdollar2 = {<$0}
USdollar3 = {$1–$10K}
USdollar4 = {>$10K}
Currency1 = {Euros}
Currency2 = {Pounds}
Currency3 = {Swiss francs}
Currency4 = {Nothing selected}
Operation1 = {Compute}
Operation2 = {Clear All}
Operation3 = {Quit}

The first four columns of Table 20.6 are the allpairs.exe program outputs. The (tester-provided) expected outputs are in the last column. The "~Compute" in test cases 15 and 16 is an allpairs.exe output that directs the tester to pick an operation other than Compute. (It is an extension of the "Don't Care" entry in decision tables.) Notice that only error messages 1, 4, and either 2 or 5 are generated. Test case 9 generates a fourth context, in which the equivalent currency in pounds is computed. This is the only actual computation—the All Pairs test cases never check the conversion of dollars to euros or to Swiss francs.

Table 20.5 Allpairs.exe Input for Currency Conversion GUI

US Dollar	*Currency*	*Operation*
No entry	Euros	Compute
<$0	Pounds	Clear All
$1–$10K	Swiss francs	Quit
>$10K	Nothing selected	

Table 20.6 Allpairs.exe Test Cases for Currency Conversion GUI

Case	*US Dollar*	*Currency*	*Operation*	*Expected Output*
1	No entry	Euros	Compute	Error message 1
2	No entry	Pounds	Clear All	Pounds reset
3	No entry	Swiss francs	Quit	Application ends
4	<$0	Euros	Clear All	US dollar amount reset, euros reset
5	<$0	Pounds	Compute	Error message 4
6	<$0	Swiss francs	Compute	Error message 4
7	<$0	Nothing selected	Quit	Application ends
8	$1–$10K	Euros	Quit	Application ends
9	$1–$10K	Pounds	Compute	Equivalent in Pounds
10	$1–$10K	Swiss francs	Clear All	US dollar amount and Swiss francs reset
11	>$10K	Pounds	Quit	Application ends
12	>$10K	Nothing selected	Compute	Error message 5 or Error message 2
13	>$10K	Euros	Clear All	US dollar amount reset, euros reset
14	No entry	Nothing selected	Clear All	No change in GUI
15	$1–$10K	Nothing selected	~Compute	?
16	>$10K	Swiss francs	~Compute	?

There is a more subtle problem with the All Pairs algorithm—the order of inputs can make a surprising difference, even though it should be irrelevant. Table 20.7 just changes the order of US dollar inputs, and the resulting test cases are in Table 20.8. With just this slight change, two currency conversions are performed (to British pounds and to Swiss francs), but only error messages 3, 4, and 5 are generated.

The change is caused by the way in which the algorithm picks pairs of variables. The early test cases contain the greatest number of pairs, and the later ones contain the fewest. This means

Table 20.7 Allpairs.exe Input in Different Order

US Dollar	*Currency*	*Operation*
<$0	Euros	Compute
$1–$10K	Pounds	Clear All
>$10K	Swiss francs	Quit
No entry	Nothing selected	

Table 20.8 Allpairs.exe Test Cases (Note Differences with Table 20.6)

Case	*US Dollar*	*Currency*	*Operation*	*Expected Output*
1	<$0	Euros	Compute	Error message 4
2	<$0	Pounds	Clear All	US dollar amount reset, pounds reset
3	<$0	Swiss francs	Quit	Application ends
4	$1–$10K	Euros	Clear All	US dollar amount reset, euros reset
5	$1–$10K	Pounds	Compute	Equivalent in pounds
6	$1–$10K	Swiss francs	Compute	Equivalent in Swiss francs
7	$1–$10K	Nothing selected	Quit	Application ends
8	>$10K	Euros	Quit	Application ends
9	>$10K	Pounds	Compute	Error message 5
10	>$10K	Swiss francs	Clear All	US dollar amount reset, Swiss francs reset
11	No entry	Pounds	Quit	Application ends
12	No entry	Nothing selected	Compute	Error message 3
13	No entry	Euros	Clear All	Euros reset
14	<$0	Nothing selected	Clear All	US dollar amount reset
15	>$10K	Nothing selected	~Compute	?
16	No entry	Swiss francs	~Compute	?

that a potential All Pairs tester needs to be clever about the order in which classes of a variable are presented to the algorithm.

20.1.4 Failures Due Only to Pairs of Inputs

By definition, the All Pairs technique only potentially reveals faults due to the interaction of two variables. The NextDate counter-example showed that faults due to interaction of three variables (e.g., February 28 in a common year) will not be detected. This cannot be an indictment of the All Pairs technique—the advocates are quite clear that the intent is to find faults due only to the interaction of pairs of values. Orthogonal arrays and the OATS technique can be used to find interactions among three or more variables. As long as the program being tested uses logical variables, there is not too much risk. If a program involves computations with physical variables, some insight will likely be needed. Suppose, for example, a ratio is computed, and the numerator and denominator are from different classes. There may be no problem with nominal values, but a very large numerator divided by a very small denominator might cause an overflow fault. Worst-case boundary value testing would be a more likely method to reveal such a fault.

20.2 A Closer Look at the NIST Study

Most introductory logic courses discuss a class of arguments known as informal fallacies. One of these, the Fallacy of Extension, occurs when an argument is extended from a simple to an extreme situation where it is easier to persuade the point to be made. The conclusion is then brought back to the simple case. The Fallacy of Extension most commonly occurs when someone is asking for special consideration, and the response is something like "What if we let EVERYONE have that exception?"

There is an element of the Fallacy of Extension in the myriad of papers that emphasize how the All Pairs algorithm compresses an enormous number of test cases into a smaller, more manageable set. While the popular papers cite the NIST study as the basis for the All Pairs technique, the NIST papers (Wallace and Kuhn, 2000, 2001) never stress this idea of compression; rather, they stress that faults due to more than two variables are relatively rare (2% in the examples they studied). Both papers are concerned with describing faults, identifying root causes, and suggesting fairly standard software engineering techniques to avoid similar faults in future systems.

The closest the NIST papers come to the dominant All Pairs emphasis on test case compression is when they discuss their analysis of 109 failure reports. They note that, "Only three of the 109 failure reports indicated that more than two conditions were required to cause the failure" (Wallace and Kuhn, 2000). Further, "The most complex of these [three failures] involved four conditions." The conclusion of that part of the report is that "… of the 109 reports that are detailed, 98% showed that the problem could have been detected by testing the device with all pairs of parameter settings." The report notes that most medical devices only have "a relatively small number of inputs variables, each with either a small discrete set of possible settings or a finite range of values." Then the Fallacy of Extension occurs. Quoting from Wallace and Kuhn (2000):

> *Medical devices vary among treatment areas, but in general have a relatively small number of input variables, each with either a small discrete set of possible settings, or a finite range of values. For example, consider a device that has 20 inputs, each with 10 settings, for a total of 10^{20} combinations of settings. The few hundred test cases that can be built under most development budgets will of course cover less than a tiny fraction of a percent of the possible combinations. The number of pairs of settings is in fact very small, and since each test case must have a value for each of the ten variables, more than one pair can be included in a single test case. Algorithms based on orthogonal Latin squares are available that can generate test data for all pairs (or higher order combinations) at a reasonable cost. One method makes it possible to cover all pairs of values for this example using only 180 test cases [8].*

What is really perplexing about this is they preface it with the note that most devices only have a few input settings, so the extension to 10^{20} cases makes little sense.

20.3 Appropriate Applications for All Pairs Testing

Table 20.9 presents two considerations that help determine whether All Pairs is appropriate for a given application. The first consideration is whether the application is static or dynamic. Static applications are those in which all inputs are available before calculation begins. David Harel refers to such applications as "transformational" because they transform their inputs into output

Table 20.9 Applications Appropriate for All Pairs Testing

	Single Processor	*Multiple Processors*
Static	All Pairs potentially OK	All Pairs cannot deal with input orders
Dynamic	All Pairs potentially problematic	All Pairs cannot deal with input orders

data (Harel, 1988). Classic COBOL programs with their Input, Processing, and Output divisions are good examples of static applications.

Dynamic applications are those in which not all of the inputs that determine the ultimate path through a program are available at the onset of calculation. Harel uses the term “reactive” to convey the fact that these applications react to inputs that occur in time sequence. The difference between static and dynamic applications is analogous to the difference between combinatorial and sequential circuits of discrete components. Because the order of inputs in important, dynamic applications are not very appropriate to the All Pairs technique. There is no way to guarantee that interesting pairs will occur in the necessary order. Also, dynamic applications frequently contain context-sensitive input events in which the logical meaning of a physical input is determined by the context in which it occurs. The currency conversion example in Section 20.1.3 contains context-sensitive input events.

The second consideration is whether the application executes on a single or on multiple processors. The All Pairs technique cannot guarantee appropriate pairs of input data across multiple processors. Race conditions, duration of events in real time, and asynchronous input orders are common in multiprocessing applications, and these needs will likely not be met by All Pairs. Therefore, applications on the dynamic side of the partition, whether in single or in multiple processors, are not appropriate for All Pairs.

The remaining quadrant, static applications in a multiple processing environment, is less clear. These applications are usually computation intensive (hence the need for parallel processing). If they are truly static, within a processor, All Pairs can be an appropriate choice.

20.4 Recommendations for All Pairs Testing

All Pairs testing is just another shortcut. When the time allocated for testing shrinks, as it frequently does, shortcuts are both attractive and risky. If the following questions can all be answered “yes,” then the risk of using All Pairs is reduced.

- Are the inputs exclusively data (rather than a mix of data and events)?
- Are the variables logical (rather than physical)?
- Are the variables independent?
- Do the variables have useful equivalence classes?
- Is the order of inputs irrelevant (i.e., is the application both static and single processor)?

Since the All Pairs algorithm only generates the input portion of a test case, one last question: can the expected outputs for All Pairs test cases be determined?

EXERCISE

1. Download the allpairs.exe program from James Bach’s website and experiment with your version of the YesterDate program.

References

Bach, J. and Schroeder, P.J., Pairwise testing: A best practice that isn't, *STARWest*, San Jose, CA, October 2003.

Berger, B., Efficient testing with all-pairs, *STAREast*, Orlando, FL, May 2003.

Cohen, D.M., Dalal, S.R., Kajla, A. and Patton, G.C., The Automatic Efficient Test Generator (AETG) system, *Proceedings of the 5th International Symposium on Software Reliability Engineering*, IEEE Computer Society Press, 1994, pp. 303–309.

Harel, D., On visual formalisms, *Communications of the ACM,* Vol. 31, No. 5, May 1988, pp. 514–530.

Mandl, R., Orthogonal Latin squares: An application of experiment design to compiler testing, *Communications of the ACM*, Vol. 28, No. 10, 1985 pp. 1054–1058.

Wallace, D.R. and Kuhn, D.R., *Converting System Failure Histories into Future Win Situations*, available at http://hissa.nist.gov/effProject/handbook/failure/hase99.pdf, 2000.

Wallace, D.R. and Kuhn, D.R., Failure modes in medical device software: An analysis of 15 years of recall data, *International Journal of Reliability, Quality, and Safety Engineering*, Vol. 8, No. 4, 2001, pp. 351–371.

Chapter 21

Evaluating Test Cases

> Quis custodiet ipsos custodes?
>
> **Juvenal (*Satire VI*, lines 347–8)**
> *ca. late 1st century, a.d.*

Just as the Roman satirist Juvenal asked who would guard the guards, no matter how carefully a set of tests is developed, software testers should ask how good are their test cases. Edsger Dykstra observed that testing can detect the presence of faults but can never assert their absence. Who tests the test cases? More precisely, how can a set of test cases be tested? One answer has been around for more than 30 years—mutation testing. A more recent addition, "fuzzing," is closer to random testing. The idea that fishing creel counts as an estimate of test case success is both novel and successful. We take a brief look at all three approaches in this chapter.

21.1 Mutation Testing

A story persists, now considered to be an urban legend, about an early space program error in a FORTRAN program in which a period was used instead of a comma (Myers, 1976). The statement was (supposedly) a statement like

$$\text{DO 20 } I = 0{,}100{,}2$$

that was entered as DO 20, $I = 0{,}100.2$, which should have defined a Do-loop terminating at statement number 20 in which the loop index, I, was to vary from 0 to 100 in steps of 2. Other versions have the blanks being eliminated, creating an assignment statement

$$\text{DO20I} = 100.2$$

With the typographical error, the loop used the default increment of 1. Supposedly, this error caused a Mariner I probe to Venus to fail. This kind of fault could have been caught by mutation testing.

The term *mutant* is borrowed from biology, where it refers to a small variation of an original organism. Mutation testing begins with a unit and a set of test cases for that unit, such that all the test cases pass when executed. Then a small change is made to the original unit, and the test cases are run on the mutant. If all the test cases pass, we know that the mutant was undetected. At this point there are two possibilities—either the small change resulted in a logically equivalent program, or the set of test cases was incapable of detecting the change. This raises one of the problems of mutation testing—identification of equivalent mutants. We need some definitions to frame this discussion.

21.1.1 Formalizing Program Mutation

Definition

A *mutant* P' of a program P is the result of making a change to the source code of the original program P.

Mutation testing uses small changes, usually only one, to the source code of P. Mutation testing begins when there is a set T of test cases of P such that, for every test case $t \in T$, the test passes; specifically, the expected output of P executing t matches the observed output. The point of mutation testing is to see if the set T detects the small changes in any mutant.

Definition

Given a program P, a mutant P', a set of test cases T such that every $t \in T$ passes for P, the mutant P' is *killed* if at least one test case $t \in T$ fails.

Definition

Given a program P, a mutant P', a set of test cases T such that every $t \in T$ passes both for P and for P', the mutant P' is considered to be a *live mutant*.

Live mutants are the hardest part of mutation. There are only two possibilities—either a live mutant P' is logically equivalent to P, or the test cases in T are not sufficient to reveal the difference. Why is this a problem? Determining if P and P' are logically equivalent is formally undecidable. At this point, the theory of mutation testing develops guideline quantities that involve ratios of the number of killed mutants to the total number of mutants (which includes an undecidable number of equivalent mutants).

Definition

Suppose a program P, and a set M of mutations of P, and a set of test cases T results in x killed mutants out of y total mutants in M. Then, the ratio x/y is the *mutation score of P with respect to M*.

Higher values of mutation score increase the confidence in the utility of the original test set T. At this point, it is clear that there is a large amount of computation associated with mutation. Until very recently, the computational size relegated mutation testing to the status of an academic curiosity. At this writing, there are a few tools available; most are freeware.

21.1.2 Mutation Operators

Ammann and Offut (2008) are two of the long-term mutation researchers. In their book, they develop 11 categories of mutation operators (Hmmm … could this be “mutation overkill”? Sorry, I just couldn’t resist the pun.) To get the flavor of these replacements, the most common mutations are the result of replacing a syntactic element with another member of the same set. For example, the set of arithmetic operators, *A*, is the set

$$A = \{+, -, *, /, \%\}.$$

(We could add exponentiation and other primitives if we wished.) Similarly, the set of relational operators, *R*, is the set

$$R = \{<, <=, ==, \neq, >, >=\}$$

The third common set of replacements, *L*, deals with logical connectives that we met in Chapter 3:

$$L = \{\wedge, \vee, \oplus, \sim, \rightarrow\}$$

At this point, a mutation tester’s creativeness can extend the basic idea to unit-specific choices. If a program uses trigonometric functions, they can all be replaced by other trig functions; similarly for statistical functions. In Chapter 16, we came close to this when we discussed Halstead’s metrics. There, all of these were just lumped together into the set of operations in a program. The next three subsections use a free, online mutation testing tool (PIT) to analyze three of our previously worked examples, isLeap (from NextDate), isTriangle (from the Triangle program), and a version of the commission problem. Mutation testers can let their imaginations run wild with the “mutation explosion.” Recall the Boolean function isLeap from our earlier examples.

```
Public Function isLeap(year) As Boolean
   Dim year As Integer
   Dim c1, c2, c3 As Boolean
1. c1 = (year% 4 == 0)
2. c2 = (year% 100 == 0)
3. c3 = (year% 400 == 0)
4. isLeap = False
5. If ((c1 AND NOT(c2)) OR (c3)) Then
6.    IsLeap = True
7. Else
8.    IsLeap = False
9. EndIf
End Function
```

The “mutation explosion” is only hinted at here. In a full mutation, there are four mutations of the % operation in statement 1, compounded by five replacements of the == connective. Since mutations are made singly, there would be nine mutations of statement 1, similarly nine each for statements 2 and 3. The compound condition in statement 5 contains three logical connectives, and each of these could be replaced by four other connectives. Then we can add slight changes to the constants: we might replace 4 any of {–4, 3, 0, 5} and 100 any of {101, 0, 99, –100}. Altogether, we can imagine dozens of mutants of the basic isLeap.

21.1.2.1 isLeap Mutation Testing

Here isLeap is coded as a Java method, isLeapYear, and a class test fixture, TestisLeap. My colleague and friend, Dr. Christian Trefftz, developed and ran the examples in this section.

```
public class IsLeap
{
    public static boolean isLeapYear(int value)
    {
        return (value % 4 == 0 && value % 100 != 0)||value % 400 == 0;
    }
}

import org.junit.Test;
import static org.junit.Assert.*;

public class TestIsLeap
{
    @Test
    public void testIsLeapYear()
     {
       assertTrue(IsLeap.isLeapYear(2012));
       assertTrue(IsLeap.isLeapYear(2000));
       assertTrue(!IsLeap.isLeapYear(2013));
       assertTrue(!IsLeap.isLeapYear(1900));
     }
}
```

PIT is a freeware mutation program that integrates well with Java. Here are the results reported by PIT version 0.29:

1. Replaced integer modulus with multiplication: KILLED -> TestIsLeap.testIsLeapYear(TestIsLeap)
2. Replaced integer modulus with multiplication: KILLED -> TestIsLeap.testIsLeapYear(TestIsLeap)
3. Replaced return of integer sized value with (x == 0 ? 1 : 0): KILLED -> TestIsLeap.testIsLeapYear(TestIsLeap)
4. Negated conditional: KILLED -> TestIsLeap.testIsLeapYear(TestIsLeap)
5. Negated conditional: KILLED -> TestIsLeap.testIsLeapYear(TestIsLeap)
6. Negated conditional: KILLED -> TestIsLeap.testIsLeapYear(TestIsLeap)
7. Replaced integer modulus with multiplication: KILLED -> TestIsLeap.testIsLeapYear(TestIsLeap)

PIT performs selected mutation by examining the source code and selecting likely mutator replacement mechanisms. In this example, the following mutators were used by PIT:

- INCREMENTS_MUTATOR
- CONDITIONALS_BOUNDARY_MUTATOR
- RETURN_VALS_MUTATOR

- VOID_METHOD_CALL_MUTATOR
- INVERT_NEGS_MUTATOR
- MATH_MUTATOR
- NEGATE_CONDITIONALS_MUTATOR

21.1.2.2 isTriangle Mutation Testing

```
public class IsTriangle
{
    public static boolean isATriangle(int a, int b, int c)
    {
       return ((a < (b + c)) && (b < (a + c)) && (c < (a + b)));
    }
}

import org.junit.Test;
import static org.junit.Assert.*;

public class TestIsTriangle
{
    @Test
    public void testIsTriangle()
      {
         assertTrue(IsTriangle.isATriangle (3,4,5));
         assertTrue(!IsTriangle.isATriangle (5,2,3));
         assertTrue(!IsTriangle.isATriangle (6,2,3));
         assertTrue(!IsTriangle.isATriangle (2,5,3));
         assertTrue(!IsTriangle.isATriangle (2,6,3));
         assertTrue(!IsTriangle.isATriangle (3,2,5));
         assertTrue(!IsTriangle.isATriangle (3,2,6));
      }
}
```

Excerpts from the PIT report

1. Negated conditional: KILLED -> TestIsTriangle.testIsTriangle(TestIsTriangle)
2. Changed conditional boundary: KILLED -> TestIsTriangle.testIsTriangle(TestIsTriangle)
3. Negated conditional: KILLED -> TestIsTriangle.testIsTriangle(TestIsTriangle)
4. Negated conditional: KILLED -> TestIsTriangle.testIsTriangle(TestIsTriangle)
5. Replaced integer addition with subtraction: KILLED -> TestIsTriangle.testIsTriangle(TestIsTriangle)
6. Replaced return of integer sized value with (x == 0 ? 1 : 0): KILLED -> TestIsTriangle.testIsTriangle(TestIsTriangle)
7. Replaced integer addition with subtraction: KILLED -> TestIsTriangle.testIsTriangle(TestIsTriangle)
8. Changed conditional boundary: KILLED -> TestIsTriangle.testIsTriangle(TestIsTriangle)
9. Replaced integer addition with subtraction: KILLED -> TestIsTriangle.testIsTriangle(TestIsTriangle)
10. Changed conditional boundary: KILLED -> TestIsTriangle.testIsTriangle(TestIsTriangle)

Active mutators

- INCREMENTS_MUTATOR
- CONDITIONALS_BOUNDARY_MUTATOR
- RETURN_VALS_MUTATOR
- VOID_METHOD_CALL_MUTATOR
- INVERT_NEGS_MUTATOR
- MATH_MUTATOR
- NEGATE_CONDITIONALS_MUTATOR

Tests examined

- TestIsTriangle.testIsTriangle(TestIsTriangle) (92 ms)

21.1.2.3 Commission Mutation Testing

This example is more interesting. In the PIT report, we see that two mutants (11 and 17) survived. Sadly, there is no further information about the surviving mutations. This run would give a mutation score of 19/21, or 0.905. The previous two examples had perfect mutation scores (1.0).

```
public class SalesCommission
{
   public static int calcSalesCommission (int locks, int stocks, int barrels)
    {
        int sales,commision;
        sales = 45*locks + 30*stocks + 25*barrels;
        if (sales <= 1000)
            commission = sales*0.10;
        if ((sales > 1000) && (sales <= 1800))
            commission = 100 + (sales - 1000)*0.15;
        if ((sales > 1800)
            commission = 100 + 800*0.15 + (sales - 1800)*0.20;

        return commission;
    }
}

import org.junit.Test;
import static org.junit.Assert.*;

public class TestSalesCommission
{
   @Test
   public void testCommission()
    {
        assertEquals(SalesCommission.calcSalesCommission (1,1,1), 10);
        assertEquals(SalesCommission.calcSalesCommission (8,8,8), 80);
        assertEquals(SalesCommission.calcSalesCommission (10,10,10), 100);
        assertEquals(SalesCommission.calcSalesCommission (11,11,11), 115);
        assertEquals(SalesCommission.calcSalesCommission (17,17,17), 205);
        assertEquals(SalesCommission.calcSalesCommission (18,18,18), 220);
        assertEquals(SalesCommission.calcSalesCommission (19,19,19), 240);
        assertEquals(SalesCommission.calcSalesCommission (10,0,0), 45);
        assertEquals(SalesCommission.calcSalesCommission (0,10,0), 30);
        assertEquals(SalesCommission.calcSalesCommission (0,0,10), 25);
    }
}
```

Excerpts from PIT report

1. Replaced integer addition with subtraction: KILLED -> TestSalesCommission.testCommission(TestSalesCommission)
2. Replaced integer multiplication with division: KILLED -> TestSalesCommission.testCommission(TestSalesCommission)
3. Replaced integer multiplication with division: KILLED -> TestSalesCommission.testCommission(TestSalesCommission)
4. Replaced integer addition with subtraction: KILLED -> TestSalesCommission.testCommission(TestSalesCommission)
5. Replaced integer multiplication with division: KILLED -> TestSalesCommission.testCommission(TestSalesCommission)
6. Changed conditional boundary: KILLED -> TestSalesCommission.testCommission(TestSalesCommission)
7. Negated conditional: KILLED -> TestSalesCommission.testCommission(TestSalesCommission)
8. Replaced double multiplication with division: KILLED -> TestSalesCommission.testCommission(TestSalesCommission)
9. Changed conditional boundary: KILLED -> TestSalesCommission.testCommission(TestSalesCommission)
10. Negated conditional: KILLED -> TestSalesCommission.testCommission(TestSalesCommission)
11. Changed conditional boundary: SURVIVED
12. Negated conditional: KILLED -> TestSalesCommission.testCommission(TestSalesCommission)
13. Replaced double multiplication with division: KILLED -> TestSalesCommission.testCommission(TestSalesCommission)
14. Replaced integer subtraction with addition: KILLED -> TestSalesCommission.testCommission(TestSalesCommission)
15. Replaced double addition with subtraction: KILLED -> TestSalesCommission.testCommission(TestSalesCommission)
16. Negated conditional: KILLED -> TestSalesCommission.testCommission(TestSalesCommission)
17. Changed conditional boundary: SURVIVED
18. Replaced double addition with subtraction: KILLED -> TestSalesCommission.testCommission(TestSalesCommission)
19. Replaced integer subtraction with addition: KILLED -> TestSalesCommission.testCommission(TestSalesCommission)
20. Replaced double multiplication with division: KILLED -> TestSalesCommission.testCommission(TestSalesCommission)

21. Replaced return of integer sized value with ($x == 0 ? 1 : 0$): KILLED -> TestSalesCommission.testCommission(TestSalesCommission)

Active mutators

- INCREMENTS_MUTATOR
- CONDITIONALS_BOUNDARY_MUTATOR
- RETURN_VALS_MUTATOR
- VOID_METHOD_CALL_MUTATOR
- INVERT_NEGS_MUTATOR
- MATH_MUTATOR
- NEGATE_CONDITIONALS_MUTATOR

Tests examined

- TestSalesCommission.testCommission(TestSalesCommission) (112 ms)

21.2 Fuzzing

Fuzzing is an academic curiosity that began at the University of Wisconsin (Miller et al., 1989). In true romance novel form, the seminal report notes that, "on a dark and stormy night…" the idea was accidentally discovered. Barton Miller and two graduate students, Lars Fredriksen and Bryan So, were using a dial-up Internet connection during a storm. Electronic noise on the line generated garbled character strings that caused failures on several UNIX utilities. This piqued their curiosity, and evolved into a protracted study. Their study examined 88 utilities running on seven UNIX versions, and the attendant faults revealed by random character strings.

Since then, the idea of fuzzing has been extended to several operating systems. "Fuzzers" are programs that present random strings of characters as inputs to both command line and interactive applications. The random strings are an advantage in that they can reveal situations a tester would never think of. The downside is that the expected output part of a test case cannot be defined. This is not too much of a problem because usually the response to faulty input is an error message.

This is very similar to the "automatic dialers" used in telephone switching system prototypes (but they generate multifrequency digits, not dial pulse digits). These devices generate large numbers of calls to random prototype directory numbers, with the objective of determining a lost call ratio. These systems also double as traffic generators.

21.3 Fishing Creel Counts and Fault Insertion

Fish and wildlife management offices in states with strong fishing programs use a method based on "creel counts" to assess the success of fish stocking policies. As an example, the Rogue River near Rockford, Michigan, is a designated trout stream. Every year, it is stocked with hatchery fish. The adipose fin (near the tail) is removed, and during the season, anglers are asked to participate in creel counts in which they voluntarily report their catches. On the basis of this data, stream management can make an estimate of the relative populations of wild (with adipose fin) and hatchery trout. Suppose, for example, that the Rogue River is stocked with 1000 marked trout, and that during the fishing season, the creel report totals are 500 fish caught, of which 300 are hatchery trout. The stream management team would conclude that 60% of the trout in the Rogue River are hatchery (planted) fish. The total populations could also be estimated; in this case, the 500 fish caught represent 30% of the total population. This approach to estimation assumes that the hatchery and wild fish are mixed in a uniform way.

The same idea can be applied to estimate the success of test cases and the number of remaining faults not caught by an existing set of test cases. Suppose an organization keeps demographic information on the numbers and types of faults found in developed programs. Assuming new programs are roughly similar, a set of known faults is added to the "wild code" and then the existing set of test cases is run on the "stocked" code. If the set of test cases reveals all the inserted faults, the organization can be pretty confident that the set of test cases is acceptable. If only half the inserted faults are revealed, the number of "wild faults" probably represents only half the total number of faults.

The efficacy of a fault insertion approach clearly assumes that the demographic profile of inserted faults is reasonably representative of the "wild fault" population.

EXERCISE

1. Discuss whether or not the creel count approach to fault detection might be used to optimize the starting set of test cases. Once a set of test cases finds all the inserted faults, would it be possible to reduce the number of test cases?

References

Ammann, P. and Offutt, J., *Introduction to Software Testing*, Cambridge University Press, New York, 2008.

Juvenal (*Satire VI*, lines 347–8), ca. late 1st century, a.d.

Miller, B. et al., *An Empirical Study of the Reliability of UNIX Utilities*, 1989, ftp://ftp.cs.wisc.edu/paradyn/technical_papers/fuzz.pdf

Myers, G.J., *Software Reliability: Principles & Practice,* Wiley, New York, 1976, p. 25.

Chapter 22

Software Technical Reviews

“A stitch in time saves nine.”

Francis Baily, 1797

Conventional wisdom echoes the English astronomer Baily in many aspects of daily life, from oil changes in a car to preventative software maintenance. In so many ways, we all depend on forms of reviews—surgical second opinions, movie and restaurant reviews, home safety inspections, Federal Aviation Authority aircraft inspections, and so on (add your favorites).

Are software technical reviews a form of testing? The generally accepted answer is a considered “yes.” This is amplified by the chapters on software reviews in the International Software Testing Certification Board (ISTQB) Foundation and Advanced Level Syllabi (ISTQB 2007, 2012). Software testing seeks to identify faults by causing failures, as discussed in Chapter 1. Software reviews try to identify faults (not software failures), but an identified fault typically morphs into faulty code, which, when executed, causes a failure.

Much of the material in this chapter is based on experience in the development of telephone switching system software. Those applications could have a 30-year serviceable lifetime; hence, software maintenance could last that long. In self-defense, that organization refined its review process over an interval of 15 years, resulting in “industrial-strength technical inspections.” The industrial strength part refers to a process that was gradually refined, and which contains several subtle checks and balances.

It is helpful to understand software reviews as a critical evaluation of a work product, performed by people who are technically competent. A software review is, or should be, a scheduled, budgeted development activity with formal entry and exit criteria.

22.1 Economics of Software Reviews

Many development organizations are reluctant to institute software reviews, mostly because of a shortsighted view of cost of preparation. As far back as 1981, Barry Boehm dispelled this notion with his graph of fault resolution costs as a function of when they are discovered (see Figure 22.1)

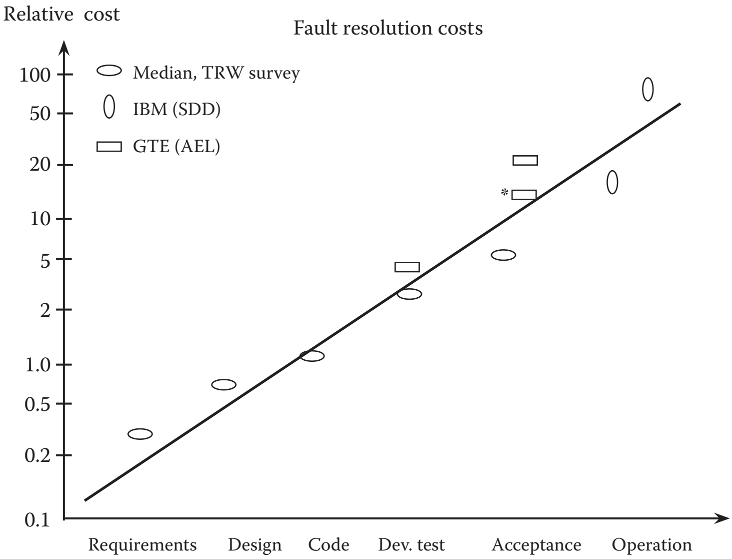


Figure 22.1 Relative costs of defect resolution. (From Boehm, B., *Software Engineering Economics*, Englewood Cliffs, NJ: Prentice-Hall, 1981.)

(Boehm, 1981). This is a remarkable comparison as it relates data from three diverse organizations. (As a curious footnote, the GTE AEL project closest to the line in the Acceptance phase is the project for which I prepared data on Dr. Boehm's request.) The cost axis is a logarithmic scale, and the straight line of best fit means that correction costs increase exponentially with time.

As early as 1982, Roger Pressman (Pressman, 1982) used a defect amplification model that was presented in a course at the IBM System Sciences Institute. The model describes how defects from one waterfall phase might be amplified in a following phase. Some defects might be simply passed through, and others might be amplified by work done in the successor phase. The defects then form their own waterfall, something probably not intended by the waterfall model. The report continues by postulating a defect detection step in which technical inspections detect a percentage of defects before they can be passed on to successor phases. Pressman presents a hypothetical example showing two versions of a waterfall-based software development—one with technical inspections and one without. The result: 12 defects without reviews were reduced to 3 after reviews at three development phases. This is a hypothetical example, but it illustrates a widely agreed-on fact—reviews reduce faults, and consequently, the overall development cost.

More recently, Karl Wiegers (1995) reports that, in an unnamed German company, correcting defects found by testing was 14.5 times the cost to find the problem in an inspection, and this grew to 68 times the inspection cost if the defect was reported by a customer. Wiegers continues with an updated IBM statistic: correcting defects found in a released product was 45 times the cost if the defect was fixed at design time. He asserts that, while technical inspections may constitute 5%–15% of total project cost, "Jet Propulsion Laboratory estimated a net savings of $7.5 million from 300 inspections performed on software they produced for NASA" and "another company reports annual savings of $2.5 million." One last Wiegers statistic: in another unnamed company, the cost to fix a defect found by inspection was $146 compared with the cost to fix a defect found by customer: $2900, resulting in a cost/benefit ratio of 0.0503.

The bottom line? People in development organizations make mistakes, and the earlier these are caught, the cheaper they are to resolve. To be effective, reviews need both process and reviewer credibility, and they must consider human factors. In the next sections, we describe the roles in a review; we then look at and compare three types of reviews, the materials needed to conduct a thorough review, a time-tested review process, and review etiquette. The chapter concludes with a rather surprising study done at Grand Valley State University.

22.2 Roles in a Review

In all three types of reviews, there are similar roles. A review team consists of the person who developed the work product, reviewers, a review leader, and a recorder. These roles may involve some duplication, and in some cases, some may be missing. Reviews are an interesting point in a software project because the technical and management roles intersect there. The outcome of each type of review is a technical recommendation to the responsible administrator, and this is a crucial point at which responsibility transfers from developers to management.

22.2.1 Producer

As the name implies, this is the person who created the work product being examined. The producer is present in the review meeting but may not contribute much as one of the reviewers. Why? We all know it is much easier to proofread someone else's work rather than one's own. The same holds true for technical reviews. In Section 22.3, the roles that a producer may have in the different types of reviews are discussed. At the end of all types of technical reviews, the producer will be the person who resolves the action items identified during the review meeting.

22.2.2 Review Leader

Review leaders are responsible for the overall success of the review. They have the following duties:

- Schedule the actual review meeting
- Ensure that all members of the review team have the appropriate review materials
- Conduct the actual review meeting
- Write the review report

To do all of this, a review leader must be technically competent, be well organized, have leadership ability, and must be able to prioritize. Above all, a review leader must be able to conduct an orderly, well-paced business meeting. There are lessons to be learned from a poorly run business meeting. Such meetings are characterized by some or all of the following:

- Participants see them as a waste of time.
- The wrong people are at the meeting.
- There is no agenda, or if there is, it is not followed.
- There is no prior preparation.
- No issues are identified.
- The discussion is easily side-tracked.
- Time is spent fixing problems rather than just identifying them.

Any one of these will doom a review meeting, and it is the responsibility of the review leader to assure that they do not occur.

22.2.3 Recorder

Because of connotations associated with "secretary," the preferred term for this role is review recorder. As the title implies, the recorder takes notes during the review meeting. To do this, recorders must be able to track conversations and write notes in parallel—quite a skill, and not all of us have that ability. It is helpful if recorders can write clearly and succinctly because the recorded notes will be the basis for the formal review report. Often the recorder helps the review leader write the review report. It is a good practice for the recorder to have a "mini-review" in the last few minutes of the review meeting to go over the notes to see if anything was missed.

22.2.4 Reviewer

The individual reviewers are responsible for objectively reviewing the work product. To do this, they must be technically competent, and should not have any biases or irrelevant personal agendas. The reviewers identify issues and assign a severity level to each item. During the review meeting, these issues are discussed, and the severity level may be changed by consensus. Before the review meeting, each reviewer submits a review ballot that contains the following information:

- Reviewer name
- Review preparation time
- A list of issues with severity
- An overall review disposition recommendation (OK as is, accept with minor rework, major rework with a new review needed)

22.2.5 Role Duplication

In smaller organizations, it may be necessary for one person to fill two review roles. Here are some common pairings, and a short comment on each possibility:

- Review leader is the producer—this happens in a walkthrough. It is usually a poor idea, particularly if the producer is technically insecure.
- Review leader is the recorder—this can work, but it is difficult.
- Review leader is a reviewer—this works reasonably well, but is very time consuming.

22.3 Types of Reviews

There are three fundamental types of software reviews: walkthroughs, technical inspections, and audits. Each of these is described here, and then compared with the others. Before doing that, we clarify reasons to conduct a review. Here is a list of frequently given reasons:

- Communication among developers
- Training, especially for new personnel, or for personnel recently added to a project
- Management progress reporting
- Defect discovery
- Performance evaluation (of the work product producer)
- Team morale
- Customer (re)assurance

All of these can happen with a software review; however, the best (some say only!) reason to have reviews is to discover defects. With this focus, all of the other "reasons" turn out to be diversions, and each diminishes the defect discovery goal.

22.3.1 Walkthroughs

Walkthroughs are the most common form of review, and they are the least formal. They often involve just two people, the producer and a colleague. There is generally no preparation ahead of the walkthrough, and usually little or no documentation is produced. The producer is the review leader; therefore, the utility of a walkthrough depends on the real goal of the producer. It is easy for a producer/review leader to direct the walkthrough to the "safe" parts of the work product, and avoid the portions where the producer is unsure. This is clearly a degenerate case, but it happens, particularly when technical people resent the review process. Walkthroughs are most effective at the source code level, and on other small work products.

22.3.2 Technical Inspections

Pioneered by Michael Fagan while he was at IBM in the 1970s, technical inspections are the most effective form of software reviews. They are a highly formal process, and more details of technical inspections are given in Sections 22.4 and 22.5. The effectiveness of technical inspections is a result of several success factors, including

- A documented inspection process
- Formal review training
- Budgeted review preparation time
- Sufficient lead time
- Thoughtful identification of the inspection team
- A refined review checklist
- Technically competent participants
- "Buy in" by both technical and management personnel

22.3.3 Audits

Audits are usually performed by some external group, rather than the development team. Audits may be conducted by a software quality assurance group, a project group, an outside agency, or possibly a government standards agency. Audits are not primarily concerned with finding defects—the main concern is conformance to some expectations, either internal or external. This is not to diminish the importance of audits—they can be very expensive because they require significant preparation time. Whereas a technical inspection meeting may last 60 to 90 minutes, an audit may last a full day or more. Audits may be required by contract, and an unsatisfactory audit usually results in expensive corrective actions.

22.3.4 Comparison of Review Types

The main characteristics of the three review types are summarized in Table 22.1.

Because technical inspections are the most effective at finding defects early, they are the focus of the remainder of this chapter.

22.4 Contents of an Inspection Packet

One of the success factors of a technical inspection is the packet of materials that the inspection team uses in its preparation. Each inspection packet item is described in the succeeding subsections. The Appendix contains a sample inspection packet for a use case inspection.

22.4.1 Work Product Requirements

As mentioned earlier, technical inspections are valuable because they find faults early in a development process. In the waterfall life cycle, and in many of its derivatives, the early phases are characterized by tight what/how cycles, in which one phase describes what must be done in the next phase, and the successor phase describes "how" it responds to the "what" definition. These tight what/how cycles are ideally suited for technical inspections; therefore, one important element in the inspection packet is the work product requirements. Without this, the review team will not be able to determine if the "how" part has actually been accomplished.

22.4.2 Frozen Work Product

Once an inspection team has been identified, each member receives the full inspection packet. This is a point at which three software project disciplines converge: development, management, and configuration management. In the configuration management view, a work product is called a "design item." Once a design item has been reviewed and approved, it becomes a "configuration item." Design items can be changed by the responsible designers (producers), but configuration items are frozen, meaning that they cannot be changed by anyone unless they are first demoted to

Table 22.1 Comparison of Review Types

Aspect	*Walkthrough*	*Inspection*	*Audit*
Coverage	Broad, sketchy	Deep	Varies with auditor(s)
Driver	Producer	Checklist	Standard
Preparation time	Low	High	Could be very high
Formality	Low	High	Rigid
Effectiveness	Low	High	Low

design item status. Once a design item enters the inspection process, the producer may no longer make changes to it. This ensures that the full inspection team is literally on the same page.

22.4.3 Standards and Checklists

When given a work product to inspect, how does a reviewer know what to do? What to look for? In a mature inspection process, the organization has checklists appropriate to the various work products subject to inspections. A checklist identifies the kinds of problems that a reviewer should look for. Checklists are refined over time, and many companies consider their inspection checklists to be proprietary information. (Who would like to share with the world what their product weak points and concerns are?)

A good checklist is modified as it is used. In fact, one inspection meeting agenda item can be to ask whether any changes in the checklist are needed. Checklists should be public in a development organization. One side benefit is that checklists can improve the development process. This is very similar to the use of grading rubrics in the academic world. If students know the grading criteria, they are much more likely to submit a better assignment. When developers consult a checklist, they know what historical situations have been fault-prone, and therefore they can proactively deal with these potential problems.

There is a wealth of online material to get started with developing checklists. One paper (http://portal.acm.org/citation.cfm?id=308798) surveys 117 checklists from 24 sources. Different categories of checklist items are discussed and examples are provided of good checklist items as well as those that should be avoided. Karl Weigers' website is another good source for checklists (http://www.processimpact.com/pr_goodies.shtml).

Applicable standards play a role similar to checklists. Development organizations may have code-naming standards, for example, or required templates for test case definition. Conformance to applicable standards is usually required, and is therefore and easy item on an inspection checklist. As with checklists, standards may be subject to change, albeit more slowly.

22.4.4 Review Issues Spreadsheet

Individual reviewers identify issues and submit them to the review leader. A spreadsheet with columns as shown in Table 22.2 greatly facilitates the process that the review leader uses to merge the inputs from the full inspection team.

Table 22.2 Individual Reviewer Issues Spreadsheet

<Work product information>					
<Reviewer Name>					
<Preparation Date>					
<Reviewer Preparation Time>					
	Location		*Checklist*		
Issue #	*Page*	*Line*	*Item*	*Severity*	*Description*
1	1	18	Typo	1	Change "accound" to "account"

Information in the individual reviewer issues spreadsheets is merged into a master issues spreadsheet by the review leader (Table 22.3). The spreadsheet can then be sorted by location, by checklist item, by fault severity, or some combination of these. This enables the review leader to prioritize the issues, which then becomes the skeleton of the review meting agenda. This overview of the full set of identified issues can also be used to estimate the length of the review meeting time. In extreme cases, the faults might constitute a "showstopper"—faults so severe that the work product is not yet ready for a review and is returned to the producer. The producer can then use the combined issues list to guide revision work.

22.4.5 Review Reporting Forms

Once reviewers complete their examination of the work product, they submit an individual review report form to the review leader. This form should contain the following information:

- Reviewer name
- Work product reviewed
- Preparation hours spent
- Summary of the review issues spreadsheet showing the number of issues of each severity level

Table 22.3 Review Report Spreadsheet

<Work Product Information>						
Review Team Members		*Preparation Time*				
Leader						
Recorder						
Reviewer						
Reviewer						
Reviewer						
Reviewer						
	Total prep time					
Meeting date						
<Review Recommendation>						
		Location		Checklist		
Issue #	*Reviewer*	*Page*	*Line*	*Item*	*Severity*	Description
1		1	18	Typo	1	Change "accound" to "account"

- Description of any “showstopper” issue(s)
- The reviewers recommendation (OK as is, minor rework needed, or major rework with new review needed)

This information can be used to analyze the effectiveness of the review process. During my industrial career, the software quality assurance group made a study of the severity of found defects as a function of preparation hours. They proved the obvious, but the results are interesting: out of four severity levels, the only reviewers who found the really severe faults were those who spent six to eight hours of preparation time. At the other end of the severity spectrum, those who only found the lowest severity faults only spent one to two preparation hours.

There are other possible analyses, and they relate to the whole idea of openness and accountability. The underlying assumption is that all review documents are open, in the sense that they are available to everyone in the organization. Accountability is the desired consequence of this openness. Consider reviewers who report significant preparation time yet do not report the severe faults that other reviewers find. If there is a pattern of this, some supervisory intervention is appropriate. Conversely, reviewers who consistently find the severe faults can be recognized as effective review team members, and this can be a consideration in an annual performance review.

22.4.6 Fault Severity Levels

It is helpful if items in an inspection checklist are given severity levels. The Appendix contains a sample definition of severity levels for use cases. More recently, the IEEE Standard Classification for Software Anomalies Working Group has published (and sells) 1044-2009 IEEE Standard Classification for Software Anomalies (IEEE, 2009). While examples are nice, detailed fault severity levels are awkward in practice. Rather than have a debate about whether a discovered fault is severity level 7 or 8, it is more productive to have a simple three- or four-level severity classification (such as the one in the Appendix).

The order of severity levels is less interesting: usually the simplest faults are of severity 1 and the most complex are the high end of the scale (3 or 4). This avoids the confusion that sometimes occurs with priority levels. (Consider priority = 4 and priority = 1: does the 4 mean high priority, or does the 1 mean first priority?)

22.4.7 Review Report Outline

The review report is the point where technical responsibility ends and administrative responsibility begins, so the review report must serve the needs of both groups of people. It also becomes the basis for accountability because the management relies on the technical judgment of the review team.

Here is a sample outline of a review report:

1. Introduction
   a. Work product identification
   b. Review team members and roles
2. Preliminary issue list
   a. Potential fault
   b. Severity
3. Prioritized action item list
   a. Identified fault
   b. Severity

4. Summary of individual reports
5. Review statistics
    a. Total hours spent
    b. Faults sorted by severity
    c. Faults sorted by location
6. Review recommendation
7. Appendix with the full review packet

22.5 An Industrial-Strength Inspection Process

This section describes a process for technical reviews that gradually evolved over a period of 12 years in a research and development laboratory that developed telephone switching system hardware and software. Since the commercial lifetime of these systems could reach 30 years, the developing organization had to produce nearly fault-free systems as a matter of economic necessity. As they say, necessity is the mother of invention—certainly true in what is termed here an "industrial-strength inspection process." Some of the checks and balances will be highlighted, as well as some of the resolutions to hard questions.

Figure 22.2 shows the stages in the industrial-strength inspection process. Even these stages were carefully devised. As presented, it happens to resemble common depictions of the waterfall life cycle mode, but there are several important differences. The sequence of stages is important, and deviations from the sequence simply do not work. The activities of each stage, and some of the reasons for them, are described in the next subsections.

22.5.1 Commitment Planning

The technical inspection process begins with a meeting between the producer of the work product and his/her supervisor. Working together, they identify an appropriate review team and the review leader. In a degenerate case, this can be mildly adversarial—the producer may wish to "stack the

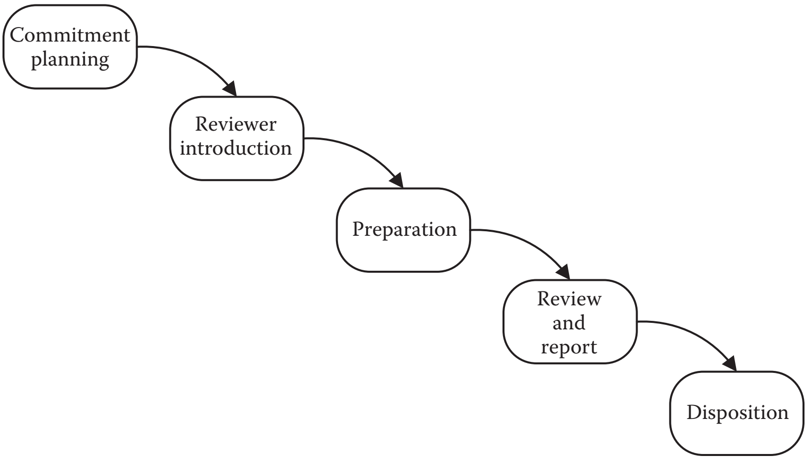


Figure 22.2 Stages in industrial-strength inspection process.

deck" with close friends while the supervisor may wish to "send a message" to the producer. Both possibilities are clearly regrettable, but they can happen. On the positive side, if the producer and the supervisor both agree on the value of inspections, they will both see it as a way to promote their own self-interests. After some negotiation, both the producer and the supervisor need to accept and approve the identified review team. In a truly formal process, both parties might even sign off on this agreement.

Once the review team is identified, the supervisor completes any necessary administrative approval. One curious question can arise at this point. What if a review team member is from another supervisory group? Even worse, what if the other supervisor feels that the requested reviewer is on a critical path and cannot be spared? This becomes a question of corporate culture. A good answer is that, if the organization is truly committed to technical inspections, everyone understands that such conflicts can occur. This should be discussed and agreed upon at the project initiation, thereby preventing future conflicts.

The supervisor should have a commitment meeting, with other supervisors if necessary, to obtain commitments for all review team members. Any task approvals are communicated at this meeting. Once all this is done, the results are given to the review leader. This is the point where administrative authority is handed over to the technical people. It is also the point at which management separates from the inspection process.

22.5.2 Reviewer Introduction

Once the review process is turned over to the review team, the review leader assembles the team for a brief meeting. In preparation for this meeting, the producer prepares the full review packet and freezes the work product to be examined. At the preliminary meeting, the review leader delivers the review packet and gives a brief overview of the work product. There may be a discussion of the work product, including any special concerns. Since the review team is accountable for the technical recommendation, the team should decide whether or not the review packet is complete. One item of business is to select the review recorder and to schedule the review meeting time. The meeting ends with all team members either committing to the process or possibly disqualifying themselves. In the latter case, the process may go back to the commitment planning stage (this is, or should be, rare).

22.5.3 Preparation

The review team members have approved preparation time—this is important. It is simply not realistic to rely on a team member's good will to spend personal time (i.e., unpaid) on review preparation. The preparation interval for a review of normal duration (60–90 minutes) should be five full working days, in which up to 8 hours of preparation time can be used by each review team member. Allowing a 5-day interval should be enough for reviewers to meet most of their other commitments.

As part of the preparation, reviewers examine the work product with respect to the review checklist and their own expertise. As issues are recognized, they are recorded into the individual reviewer issues spreadsheet (see Table 22.2). Reviewers should describe the issue, provide a short explanation or description, and then make a severity assessment. At least one full day before the review meeting, the reviewers send their individual spreadsheets to the review leader, along with their ballots showing actual time spent, and their preliminary recommendations.

Once all the individual reports are in, the review leader merges them into a single spreadsheet, and prioritizes the issues. This involves some insight, because often two reviewers may provide slightly different descriptions of the same underlying issue. The location information usually is enough to recognize this problem. Given a final issues list, the review leader makes a Go/No Go decision based on the number and severity of the issues. (Review cancellation should be rare, but it is wise to allow for the possibility.) Assuming the review will occur, the review leader prepares the final agenda by prioritizing the merged issues—a form of triage.

22.5.4 Review Meeting

The actual review meeting should be conducted as an effective business meeting. In Section 22.2.2, there is a list of characteristics of a poorly run business meeting. Some steps in the review process have already been taken to assure an effective review meeting:

- The review team was carefully selected, so the right people will be in the meeting.
- The agenda is based on the prioritized list of issues, so there should not be a sense that the meeting is a waste of time.
- The process calls for budgeted preparation time in which issues are identified before the meeting.

Normally, the first order of business is to decide if the meeting should be postponed. The main reasons are most likely absent or unprepared team members. Assuming that the review will proceed, the main job of the review leader is to follow the agenda, and make sure that issues are identified, and agreed upon, but not resolved. Once the agenda has been completed, the review leader asks for a consensus of the review recommendation. Recall that the options are Accept as is, Accept with minor rework but no additional review is needed, or Reject. The review meeting ends with a short wrap-up conducted by the recorder in which the issues list is finalized, the individual ballots are collected, and the team checks that nothing was forgotten.

22.5.5 Report Preparation

The review leader is primarily responsible for writing the review report, but assistance from the recorder is certainly in order. The report is a technical recommendation to management, and it ends the technical responsibility (but not the accountability). If there are any issues, they are noted as action items that require additional work from the producer. The review report, and all other materials, should be open to the entire organization, as this enhances accountability.

22.5.6 Disposition

Once the producer's supervisor receives the report, it becomes the basis of a management decision. There may be pressing reasons to ignore the technical findings, but if this happens, it is clearly a management choice. Assuming the recommendation is to accept the work product, it becomes subject to the configuration management function, and the work product is no longer a design object; it is a configuration item. As such, it can be used in the remainder of the project as a reliable component, not subject to change. If the review recommendation lists action items, the producer's supervisor and the producer make an estimate of the effort required to resolve the action items, and the work is done by the producer. Once all action items are resolved, the supervisor either closes the review or starts a re-review process.

22.6 Effective Review Culture

All forms of reviews are social processes; hence, they become corporate culture considerations. In addition, reviews can be quite stressful, and this also requires social considerations. Reviews are a group activity, so group size becomes a question. In general, technical inspection teams should have from four to six members. Fewer members might be necessary in small development organizations. More than six team members is usually counterproductive.

Part of an effective corporate culture is that reviews must be seen as valuable activities by both management and technical people. Reviews must have formally budgeted time for all the activities described in Section 22.5. Human factors are important. Long reviews are seldom effective—psychologists claim that the attention span of most adults is about 12 minutes. Consider what effect this can have on a 2-hour meeting. Most review meetings should be in the 60- to 90-minute range, with shorter meetings preferred. Furthermore, review meetings should be viewed as important, and interruptions should not be tolerated. (This includes cell phones!)

The best time to have a review meeting? About an hour after the normal start of the working day. This allows review team members to take care of little things that otherwise might be distractions. The worst time? Just after lunch, or maybe beginning at 3:00 on a Friday afternoon.

22.6.1 Etiquette

To reduce the stress that can accompany a review, the following points of review etiquette should be observed:

1. Be prepared. Otherwise, the review effectiveness will be diminished. In a sense, an unprepared team member is disrespecting the rest of the review team.
2. Be respectful. Review the product, not the producer.
3. Avoid discussions of style.
4. Provide minor comments (e.g., spelling corrections) to the producer at the end of the meeting.
5. Be constructive. Reviews are not the place for personal criticism, nor for praise.
6. Remain focused. Identify issues; do not try to resolve them.
7. Participate, but do not dominate the discussion. Careful thought went into selection of the review team.
8. Be open. All review information should be widely available to the full organization.

22.6.2 Management Participation in Review Meetings

Many organizations struggle with the question of management participation in reviews. Generally, this is a bad idea. Management presence in a review easily creates additional stress on all team members, but in particular, on the producer. If management participation is common, the whole process can easily degenerate into unspoken agreements among the technical staff (I won't make you look bad if you don't make me look bad). Another possible consequence is that management might not want negative results to be public—clearly a conflict of interest. How credible might a management person be as a reviewer? Willing to do the normal preparation? Capable of doing the normal preparation? Failing either of these questions, a management person becomes a drag on the review meeting. To be fair, there are managers who are technically competent, and they can be disciplined enough to respect the process. The admission ticket would be to do the normal review preparation and set aside any managerial objectives.

22.6.3 A Tale of Two Reviews

The "Dilbert" comic strip of Scott Adams usually contains poignant insights to software development situations. What follows are two possible reviews that would fit into an extended "Dilbert" scenario.

22.6.3.1 A Pointy-Haired Supervisor Review

1. The producer picks friendly reviewers.
2. There is little or no lead time.
3. There is no approved preparation time.
4. The work item is not frozen.
5. The review meeting is postponed twice.
6. Some reviewers are absent; others take cell phone calls.
7. Some designers never participate because they cannot be spared.
8. There is no checklist.
9. No action items are identified and reported.
10. The review leader proceeds in a page-by-page order (no triage).
11. Faults are resolved "while they are fresh in mind."
12. Coffee and lunch breaks are needed.
13. Reviewers float in and out of the meeting.
14. The producer's supervisor is the review leader.
15. Several people are invited as spectators.

Just imagine this as a review!

22.6.3.2 An Ideal Review

Here are some characteristics of a review in a desirable review culture.

1. Producers do not dread reviews.
2. Reviewers have approved preparation time.
3. A complete review packet is delivered with sufficient lead time.
4. All participants have had formal review training.
5. Technical people perceive reviews as productive.
6. Management people perceive reviews as productive.
7. Review meetings have high priority.
8. Checklists are actively maintained.
9. Top developers are frequent reviewers.
10. Reviewer effectiveness is recognized as part of performance evaluation.
11. Review materials are openly available and used.

22.7 Inspection Case Study

One of the few things that can be done in a university setting that cannot be done in industry is repetition. Industrial development groups cannot justify doing the same thing multiple times. This section reports results of a study done in a graduate course on software testing at Grand Valley State University. Five groups of graduate students each performed a Use Case Technical Inspection using the review packet of materials in the Appendix. (The use cases have been simplified in the Appendix.) The team members in the class are fairly representative of development groups in industry—a range of experience from new hires through people with two decades of software development. Table 22.4 summarizes the experience profiles of the five review teams.

Table 22.5 clarifies the experience levels in terms of years of industrial experience.

The class had 3 hours of instruction based on materials that were precursors to this chapter. The review teams were identified in a class meeting, and they used the review packet in the Appendix. The teams had a full week for review preparation, and communicated via e-mail. The following week, each team conducted a 50-minute technical inspection.

In Table 22.6, the last two columns need explanation. The total number of issues reported to the review leader is typically reduced during the review to a shorter list of action items that require additional work. In the case of group 3, for example, many of the low-severity issues were just simple corrections. Also, there will be duplication among the reported issues—something that the review leader must recognize and collapse into one agenda item.

It would be nice to have a Venn diagram showing the final action items of each review team. This is topologically impossible with five circles. Instead, Table 22.7 describes the overlap among groups. Of the 32 possible subsets of groups, only those with an overlap are listed. After the review meetings, the five groups found a total of 116 action items.

When all of these are aligned (by eliminating separate appearances of the same underlying fault), Table 22.7 is alarming. Consider the first few rows, in which 50 faults are found only by one

Table 22.4 Experience Levels of Review Teams

Group	*Experience*
1	1 very experienced, 3 with some experience
2	4 with significant experience
3	2 with significant experience, 2 with little experience
4	2 with significant experience, 2 with little experience
5	2 with little experience

Table 22.5 Experience Levels of Review Teams

Experience Level	*Years*
Little	0–2
Some	3–6
Significant	7–15
Very	>15

Table 22.6 Preparation Time and Fault Severity of Each Team

Group	*Total Preparation Time (Hours)*	*Low Severity*	*Medium Severity*	*High Severity*	*Total Issues Found*	*Review Action Items*
1	7		33		33	18
2	6	32	27		59	26
3	36	66	27		93	12
4	21	24	20	9	53	46
5	22	13	4	10	27	10

Table 22.7 Demographics of Faults Found by Inspection Teams

Groups	*Issues*	*Groups*	*Issues*
1 only	4	2 and 4 only	6
2 only	9	3 and 4 only	1
3 only	6	1, 2, and 4	3
4 only	27	1, 2, and 5	1
5 only	4	2, 4, and 5	1
1 and 2 only	2	1, 2, 4, and 5	1
1 and 3 only	1	1, 3, 4, and 5	1
1 and 4 only	3	2, 3, 4, and 5	1
2 and 3 only	1	All groups	1

group. Even worse, look at the last four entries, where only one fault was found by all five groups, and only four faults were found by four of the five groups.

The implications of this are enormous—companies simply cannot afford to have duplicate inspections of the same work product, so it behooves companies to provide review training, and inspection teams need to use their limited time as effectively as possible.

References

Boehm, B., *Software Engineering Economics*, Englewood Cliffs, NJ: Prentice-Hall, 1981.

Pressman, R.S., *Software Engineering: A Practitioner's Approach*, New York: McGraw-Hill, 1982.

Wiegers, K., Improving quality through software inspections, *Software Development*, Vol. 3, No. 4, April 1995, available at http://www.processimpact.com/articles/inspects.html.

Chapter 23

Epilogue: Software Testing Excellence

Finishing a book is almost as hard as beginning one. The ubiquitous temptation is to return to "finished" chapters and add a new idea, change something, or maybe delete a part. This is a pattern that writing shares with software development, and both activities endure small anxieties as deadlines near.

This book started as a response to Myers' *The Art of Software Testing*; in fact, the original working title was *The Craft of Software Testing*, but Brian Marrick's book with that title appeared first. In the years between 1978 (Myers' book) and 1995 (the first edition of this book), software testing tools and techniques had matured sufficiently to support the *craft* motif.

Imagine a continuum with Art at one end, leading to Craft, then to Science, and ending with Engineering. Where does software testing belong on this continuum? Tool vendors would put it all the way at the engineering end, claiming that their products remove the need for the kinds of thinking needed elsewhere on the continuum. The process community would consider it to be a science, arguing that it is sufficient to follow a well-defined testing process. The context-driven school would probably leave software testing as an art, owing to the need for creativity and individual talent. Personally, I still consider software testing to be a craft. Wherever it is placed on the continuum, software testing can also be understood in terms of *excellence*.

23.1 Craftsmanship

First, a disclaimer. The more politically correct *craftspersonship* word is too cumbersome. Here, *craftsman* uses the gender-neutral sense of the *-man* suffix. What makes someone a craftsman? One of my grandfathers was a Danish cabinet maker, and that level of woodworking is clearly a craft. My father was a tool and die maker—another craft with extremely stringent standards. What did they, and others recognized as craftsmen, have in common? Here is a pretty good list:

- Mastery of the subject matter
- Mastery of the associated tools

- Mastery of the associated techniques
- The ability to make appropriate choices about tools and techniques
- Extensive experience with the subject matter
- A significant history of high-quality work with the subject matter

Since the days of Juran and Deming, portions of the software development community have been focused on *quality*. Software quality is clearly desirable; however, it is hard to define, and harder still to measure. Simply listing quality attributes, such as simplicity, extensibility, reliability, testability, maintainability, etc., begs the question. The *-ability* attributes are all similarly hard to define and measure. The process community claims that a good process results in quality software, but this will be hard to prove. Can quality software be developed in an *ad hoc* process? Probably, and the agile community certainly believes this. Do standards guarantee software quality? This, too, seems problematic. I can imagine a program that conforms to some set of defined standards, yet is of poor quality. So where does this leave the person who seeks software quality? I believe craftsmanship is a pretty good answer, and this is where *excellence* comes in. A true craftsman takes pride in his work—he knows when he has done his best work, and this results in a sense of pride. Pride in one's work also defies definition, but everyone who is honest with himself knows when he has done a really good job. So we have craftsmanship, pride, and excellence tightly coupled, recognizable, yet difficult to define, and hence to measure, but all are associated with the concept of best practices.

23.2 Best Practices of Software Testing

Any list of claimed best practices is subjective and always open to criticism. Here is a reasonable list of characteristics of a best practice:

- They are usually defined by practitioners.
- They are "tried and true."
- They are very dependent on the subject matter.
- They have a significant history of success.

The software development community has a long history of proposed "solutions" to the difficulties of software development. In his famous 1986 paper, "No Silver Bullet," Fred Brooks suggested that the software community will never find a single technology that will kill the werewolf of software development difficulties (Brooks, 1986). Here is a partial list of "best practices," each of which was intended as a silver bullet. The list is in approximate chronological order.

- High-level programming languages (FORTRAN and COBOL)
- Structured programming
- Third-generation programming languages
- Software reviews and inspections
- The waterfall model of the software development life cycle
- Fourth-generation programming languages (domain specific)
- The object-oriented paradigm
- Various replacements for the waterfall model
- Rapid prototyping

- Software metrics
- CASE (computer-aided software engineering) tools
- Commercial tools for project, change, and configuration management
- Integrated development environments
- Software process maturity (and assessment)
- Software process improvement
- Executable specifications
- Automatic code generation
- UML (and its variants)
- Model-driven development
- Extreme Programming (with its odd acronym, XP)
- Agile programming
- Test-driven development
- Automated testing frameworks

Quite a list, isn't it? There are probably some missing entries, but the point is, software development remains a difficult activity, and dedicated practitioners will always seek new or improved best practices.

23.3 My Top 10 Best Practices for Software Testing Excellence

The underlying assumption about best testing practices is that software testing is performed by software testing craftsmen. Per the earlier discussion, this implies that the tester is very knowledgeable in the craft, and has both the tools and the time to perform the task with *excellence*. There is a perennial debate as to whether or not a tester should be a talented programmer. To me, the answer is an emphatic yes. As a craftsman, programming is clearly part of the subject matter. Other attributes include creativity, ingenuity, curiosity, discipline, and, somewhat cynically, a can-I-break-it mentality. My personal "top 10" best practices are only briefly described here; most of them are treated more completely in the indicated chapters.

23.3.1 Model-Driven Agile Development

Model-driven agile development (see Chapter 11) has emerged as a powerful blend of traditional model-driven development (MDD) and the bottom–up increments of the agile world. One of the main advantages of MDD is that models, if used well, can provoke recognition of details that otherwise might be ignored. Also, the models themselves will be extremely useful for maintenance changes during the applications useful lifetime. The agile part brings the recognized advantages of test-driven development (TDD) to an agile MDD project.

23.3.2 Careful Definition and Identification of Levels of Testing

Any application (unless it is quite small) should have at least two levels of testing—unit and system. Larger applications generally do well to add integration testing. Controlling the testing at these levels is critical. Each level has clearly defined objectives, and these should be observed. System-level test cases that exercise unit-level considerations are both absurd and a waste of precious test time.

23.3.3 System-Level Model-Based Testing

If an executable specification is used, a large number of system-level test cases can be automatically generated. This in itself greatly offsets the extra effort of creating an executable model. In addition, this enables direct tracing of system testing against a requirements model. Because executable specifications are provocative, the automatically generated system test cases include many possibilities that otherwise might not be created.

23.3.4 System Testing Extensions

For complex, mission-critical applications, simple thread testing is necessary but not sufficient. At a minimum, thread interaction testing is needed. Particularly in complex systems, thread interactions are both serious and difficult to identify. Stress testing is a brute force way of identifying thread interaction. Many times, just the sheer magnitude on interactions forced by stress testing reveals the presence of previously undiscovered faults (Hill, 2006). Hill notes that stress testing is focused on known (or suspected) weak spots in the software, and that pass/fail judgments are typically more subjective than those for conventional testing. Risk-based testing is a shortcut that may be necessary. Risk-based testing is an extension of the operational profiles approach discussed in Chapter 14. Rather than just test the most frequent (high probability) threads, risk-based testing multiplies the probability of a thread by the cost (or penalty) of failure. When test time is severely limited, threads are tested in terms of risk rather than simple probability.

23.3.5 Incidence Matrices to Guide Regression Testing

Both traditional and object-oriented software projects benefit from an incidence matrix. For procedural software, the incidence between mainline functions (sometimes called features) and the implementing procedures is recorded in the matrix. Thus, for a particular function, the set of procedures needed to support that function is readily identified. Similarly for object-oriented software, the incidence between use cases and classes is recorded. In either paradigm, this information can be used to

- Determine the order and contents of builds (or increments)
- Facilitate fault isolation when faults are revealed (or reported)
- Guide regression testing

23.3.6 Use of MM-Paths for Integration Testing

Given the three fundamental approaches to integration testing discussed in Chapter 13, MM-paths are demonstrably superior. They can also be used with incidence matrices in a way that parallels that for system-level testing.

23.3.7 Intelligent Combination of Specification-Based and Code-Based Unit-Level Testing

Neither specification-based nor code-based unit testing is sufficient by itself, but the combination is highly desirable. The best practice is to choose a specification-based technique based on the nature of the unit (see Chapter 10), run the test cases with a tool to show test coverage, and then use the coverage report to reduce redundant test cases and add additional test cases mandated by coverage.

23.3.8 Code Coverage Metrics Based on the Nature of Individual Units

There is no "one size fits all" test coverage metric. The best practice is to choose a coverage metric based on the properties of the source code.

23.3.9 Exploratory Testing during Maintenance

Exploratory testing is a powerful approach when testing code written by someone other than the tester. This is particularly true for maintenance on legacy code.

23.3.10 Test-Driven Development

The agile programming community has demonstrated success in using TDD in applications where an agile approach is appropriate. The main advantage of TDD is the excellent fault isolation capability.

23.4 Mapping Best Practices to Diverse Projects

Best practices are necessarily project dependent. The software controlling a NASA space mission is clearly distinct from a quick-and-dirty program to develop some information requested by someone's supervisor. Here are three distinct project types. After their description, the top 10 best practices are mapped to the projects in Table 23.1.

Table 23.1 Best Testing Practices for Diverse Projects

Best Practice	*Mission Critical*	*Time Critical*	*Legacy Code*
Model-driven development	×		
Careful definition and identification of levels of testing	×	×	×
System-level model-based testing	×		
System testing extensions	×		
Incidence matrices to guide regression testing	×		×
Use of MM-paths for integration testing	×		
Intelligent combination of specification-based and code-based unit-level testing	×		×
Code coverage metrics based on the nature of individual units	×		
Exploratory testing during maintenance			×
Test-driven development		×	

23.4.1 A Mission-Critical Project

Mission-critical projects have severe reliability and performance constraints, and are often characterized by highly complex software. They are usually large enough so that no single person can comprehend the full system with all its potential interactions.

23.4.2 A Time-Critical Project

While mission-critical projects may also be time-critical, this section refers to those projects that must be completed rapidly. Time-to-market and the associated loss of market share are the usual drivers of this project type.

23.4.3 Corrective Maintenance of Legacy Code

Corrective maintenance is the most common form of software maintenance. It is in response to a reported fault. Software maintenance typically represents three-fourths of the programming activity in most organizations, and this is exacerbated by the pattern that maintenance changes are usually done by someone who did not create the code being changed.

References

Brooks, F.P., No silver bullet—Essence and accident in software engineering, *Proceedings of the IFIP Tenth World Computing Conference*, 1986, pp. 1069–1076, also found at No silver bullet—Essence and accidents of software engineering, *IEEE Computer*, Vol. 20, No. 4, April 1987, pp. 10–19.

Hill, T.A., Importance of performing stress testing on embedded software applications, *Proceedings of QA & TEST Conference*, Bilbao, Spain, October 2006.

Software Engineering & Systems Development

This updated and reorganized fourth edition of ***Software Testing: A Craftsman's Approach*** applies the strong mathematics content of previous editions to a coherent treatment of Model-Based Testing for both code-based (structural) and specification-based (functional) testing. These techniques are extended from the usual unit testing discussions to full coverage of less understood levels integration and system testing.

The Fourth Edition:

- Emphasizes technical inspections and is supplemented by an appendix with a full package of documents required for a sample Use Case technical inspection
- Introduces an innovative approach that merges the Event-Driven Petri Nets from the earlier editions with the "Swim Lane" concept from the Unified Modeling Language (UML) that permits model-based testing for four levels of interaction among constituents in a System of Systems
- Introduces model-based development and provides an explanation of how to conduct testing within model-based development environments
- Presents a new section on methods for testing software in an Agile programming environment
- Explores test-driven development, reexamines all-pairs testing, and explains the four contexts of software testing

Thoroughly revised and updated, ***Software Testing: A Craftsman's Approach, Fourth Edition*** is sure to become a standard reference for those who need to stay up to date with evolving technologies in software testing. Carrying on the tradition of previous editions, it will continue to serve as a valuable reference for software testers, developers, and engineers.

CRC Press
Taylor & Francis Group
an **informa** business
www.crcpress.com

6000 Broken Sound Parkway, NW
Suite 300, Boca Raton, FL 33487
711 Third Avenue
New York, NY 10017
2 Park Square, Milton Park
Abingdon, Oxon OX14 4RN, UK

K15906
ISBN: 978-1-4665-6068-0
90000
9 781466 560680

www.auerbach-publications.com