

Introduction to Software Testing

Extensively class tested, this text takes an innovative approach to software testing: it defines testing as the process of applying a few well-defined, general-purpose test criteria to a structure or model of the software. The structure of the text directly reflects the pedagogical approach and incorporates the latest innovations in testing, including modern types of software such as OO, Web applications, and embedded software. The book contains numerous examples throughout. An instructor's solution manual, PowerPoint slides, sample syllabi, additional examples and updates, testing tools for students, and example software programs in Java are available on an extensive Web site at www.introssoftwaretesting.com.

Paul Ammann, PhD, is an Associate Professor of software engineering at George Mason University. He received an outstanding teaching award in 2007 from the Volgenau School of Information Technology and Engineering. Dr. Ammann earned an AB degree in computer science from Dartmouth College and MS and PhD degrees in computer science from the University of Virginia.

Jeff Offutt, PhD, is a Professor of software engineering at George Mason University. He is editor-in-chief of the *Journal of Software Testing, Verification and Reliability*; chair of the steering committee for the IEEE International Conference on Software Testing, Verification, and Validation; and on the editorial boards for several journals. He received the outstanding teacher award from the Volgenau School of Information Technology and Engineering in 2003. Dr. Offutt earned a BS degree in mathematics and data processing from Morehead State University and MS and PhD degrees in computer science from the Georgia Institute of Technology.

Cambridge University Press
978-0-521-88038-1 - Introduction to Software Testing
Paul Ammann and Jeff Offutt
Frontmatter
[More information](#)

INTRODUCTION TO SOFTWARE TESTING

Paul Ammann

George Mason University

Jeff Offutt

George Mason University



CAMBRIDGE
UNIVERSITY PRESS

Cambridge University Press
978-0-521-88038-1 - Introduction to Software Testing
Paul Ammann and Jeff Offutt
Frontmatter
[More information](#)

CAMBRIDGE UNIVERSITY PRESS
Cambridge, New York, Melbourne, Madrid, Cape Town, Singapore, São Paulo, Delhi

Cambridge University Press
32 Avenue of the Americas, New York, NY 10013-2473, USA
www.cambridge.org
Information on this title: www.cambridge.org/9780521880381

© Paul Ammann and Jeff Offutt 2008

This publication is in copyright. Subject to statutory exception and to the provisions of relevant collective licensing agreements, no reproduction of any part may take place without the written permission of Cambridge University Press.

First published 2008

Printed in the United States of America

A catalog record for this publication is available from the British Library.

Library of Congress Cataloging in Publication Data

Ammann, Paul, 1961–

Introduction to software testing / Paul Ammann, Jeff Offutt.

p. cm.

Includes bibliographical references and index.

ISBN 978-0-521-88038-1 (hardback)

1. Computer software – Testing. I. Offutt, Jeff, 1961– II. Title.

QA76.76.T48A56 2008

004.2'4–dc22 2007035077

ISBN 978-0-521-88038-1 hardback

Cambridge University Press has no responsibility for the persistence or accuracy of URLs for external or third-party Internet Web sites referred to in this publication and does not guarantee that any content on such Web sites is, or will remain, accurate or appropriate.

Contents

<i>List of Figures</i>	<i>page ix</i>
<i>List of Tables</i>	xiii
<i>Preface</i>	xv
Part 1 Overview	1
1 Introduction	3
1.1 Activities of a Test Engineer	4
1.1.1 Testing Levels Based on Software Activity	5
1.1.2 Beizer's Testing Levels Based on Test Process Maturity	8
1.1.3 Automation of Test Activities	10
1.2 Software Testing Limitations and Terminology	11
1.3 Coverage Criteria for Testing	16
1.3.1 Infeasibility and Subsumption	20
1.3.2 Characteristics of a Good Coverage Criterion	20
1.4 Older Software Testing Terminology	21
1.5 Bibliographic Notes	22
Part 2 Coverage Criteria	25
2 Graph Coverage	27
2.1 Overview	27
2.2 Graph Coverage Criteria	32
2.2.1 Structural Coverage Criteria	33
2.2.2 Data Flow Criteria	44
2.2.3 Subsumption Relationships among Graph Coverage Criteria	50
2.3 Graph Coverage for Source Code	52

vi **Contents**

2.3.1	Structural Graph Coverage for Source Code	52
2.3.2	Data Flow Graph Coverage for Source Code	54
2.4	Graph Coverage for Design Elements	65
2.4.1	Structural Graph Coverage for Design Elements	65
2.4.2	Data Flow Graph Coverage for Design Elements	67
2.5	Graph Coverage for Specifications	75
2.5.1	Testing Sequencing Constraints	75
2.5.2	Testing State Behavior of Software	77
2.6	Graph Coverage for Use Cases	87
2.6.1	Use Case Scenarios	90
2.7	Representing Graphs Algebraically	91
2.7.1	Reducing Graphs to Path Expressions	94
2.7.2	Applications of Path Expressions	96
2.7.3	Deriving Test Inputs	96
2.7.4	Counting Paths in a Flow Graph and Determining Max Path Length	97
2.7.5	Minimum Number of Paths to Reach All Edges	98
2.7.6	Complementary Operations Analysis	98
2.8	Bibliographic Notes	100
3	Logic Coverage	104
3.1	Overview: Logic Predicates and Clauses	104
3.2	Logic Expression Coverage Criteria	106
3.2.1	Active Clause Coverage	107
3.2.2	Inactive Clause Coverage	111
3.2.3	Infeasibility and Subsumption	112
3.2.4	Making a Clause Determine a Predicate	113
3.2.5	Finding Satisfying Values	115
3.3	Structural Logic Coverage of Programs	120
3.3.1	Predicate Transformation Issues	127
3.4	Specification-Based Logic Coverage	131
3.5	Logic Coverage of Finite State Machines	134
3.6	Disjunctive Normal Form Criteria	138
3.7	Bibliographic Notes	147
4	Input Space Partitioning	150
4.1	Input Domain Modeling	152
4.1.1	Interface-Based Input Domain Modeling	153
4.1.2	Functionality-Based Input Domain Modeling	154
4.1.3	Identifying Characteristics	154
4.1.4	Choosing Blocks and Values	156
4.1.5	Using More than One Input Domain Model	158
4.1.6	Checking the Input Domain Model	158
4.2	Combination Strategies Criteria	160
4.3	Constraints among Partitions	165
4.4	Bibliographic Notes	166

	Contents	vii
5 Syntax-Based Testing	170	
5.1 Syntax-Based Coverage Criteria	170	
5.1.1 BNF Coverage Criteria	170	
5.1.2 Mutation Testing	173	
5.2 Program-Based Grammars	176	
5.2.1 BNF Grammars for Languages	176	
5.2.2 Program-Based Mutation	176	
5.3 Integration and Object-Oriented Testing	191	
5.3.1 BNF Integration Testing	192	
5.3.2 Integration Mutation	192	
5.4 Specification-Based Grammars	197	
5.4.1 BNF Grammars	198	
5.4.2 Specification-Based Mutation	198	
5.5 Input Space Grammars	201	
5.5.1 BNF Grammars	201	
5.5.2 Mutation for Input Grammars	204	
5.6 Bibliographic Notes	210	
Part 3 Applying Criteria in Practice	213	
6 Practical Considerations	215	
6.1 Regression Testing	215	
6.2 Integration and Testing	217	
6.2.1 Stubs and Drivers	218	
6.2.2 Class Integration Test Order	218	
6.3 Test Process	219	
6.3.1 Requirements Analysis and Specification	220	
6.3.2 System and Software Design	221	
6.3.3 Intermediate Design	222	
6.3.4 Detailed Design	223	
6.3.5 Implementation	223	
6.3.6 Integration	224	
6.3.7 System Deployment	224	
6.3.8 Operation and Maintenance	224	
6.3.9 Summary	225	
6.4 Test Plans	225	
6.5 Identifying Correct Outputs	230	
6.5.1 Direct Verification of Outputs	230	
6.5.2 Redundant Computations	231	
6.5.3 Consistency Checks	231	
6.5.4 Data Redundancy	232	
6.6 Bibliographic Notes	233	
7 Engineering Criteria for Technologies	235	
7.1 Testing Object-Oriented Software	236	
7.1.1 Unique Issues with Testing OO Software	237	

viii **Contents**

7.1.2	Types of Object-Oriented Faults	237
7.2	Testing Web Applications and Web Services	256
7.2.1	Testing Static Hyper Text Web Sites	257
7.2.2	Testing Dynamic Web Applications	257
7.2.3	Testing Web Services	260
7.3	Testing Graphical User Interfaces	260
7.3.1	Testing GUIs	261
7.4	Real-Time Software and Embedded Software	262
7.5	Bibliographic Notes	265
8	Building Testing Tools	268
8.1	Instrumentation for Graph and Logical Expression Criteria	268
8.1.1	Node and Edge Coverage	268
8.1.2	Data Flow Coverage	271
8.1.3	Logic Coverage	272
8.2	Building Mutation Testing Tools	272
8.2.1	The Interpretation Approach	274
8.2.2	The Separate Compilation Approach	274
8.2.3	The Schema-Based Approach	275
8.2.4	Using Java Reflection	276
8.2.5	Implementing a Modern Mutation System	277
8.3	Bibliographic Notes	277
9	Challenges in Testing Software	280
9.1	Testing for Emergent Properties: Safety and Security	280
9.1.1	Classes of Test Cases for Emergent Properties	283
9.2	Software Testability	284
9.2.1	Testability for Common Technologies	285
9.3	Test Criteria and the Future of Software Testing	286
9.3.1	Going Forward with Testing Research	288
9.4	Bibliographic Notes	290
	<i>List of Criteria</i>	293
	<i>Bibliography</i>	295
	<i>Index</i>	319

List of Figures

1.1	Activities of test engineers	<i>page</i> 4
1.2	Software development activities and testing levels – the “V Model”	6
2.1	Graph (a) has a single initial node, graph (b) multiple initial nodes, and graph (c) (rejected) with no initial nodes	28
2.2	Example of paths	29
2.3	A single entry single exit graph	30
2.4	Test case mappings to test paths	31
2.5	A set of test cases and corresponding test paths	32
2.6	A graph showing node coverage and edge coverage	34
2.7	Two graphs showing prime path coverage	37
2.8	Graph with a loop	37
2.9	Tours, sidetrips, and detours in graph coverage	38
2.10	An example for prime test paths	40
2.11	A graph showing variables, def sets and use sets	44
2.12	A graph showing an example of du-paths	46
2.13	Graph showing explicit def and use sets	47
2.14	Example of the differences among the three data flow coverage criteria	49
2.15	Subsumption relations among graph coverage criteria	50
2.16	CFG fragment for the if-else structure	52
2.17	CFG fragment for the if structure without an else	53
2.18	CFG fragment for the while loop structure	53
2.19	CFG fragment for the for loop structure	54
2.20	CFG fragment for the case structure	54
2.21	TestPat for data flow example	56
2.22	A simple call graph	65
2.23	A simple inheritance hierarchy	66
2.24	An inheritance hierarchy with objects instantiated	67
2.25	An example of parameter coupling	68
2.26	Coupling du-pairs	69
2.27	Last-defs and first-uses	69

x **List of Figures**

2.28	Quadratic root program	71
2.29	Def-use pairs under intra-procedural and inter-procedural data flow	72
2.30	Def-use pairs in object-oriented software	72
2.31	Def-use pairs in web applications and other distributed software	73
2.32	Control flow graph using the File ADT	76
2.33	Elevator door open transition	79
2.34	Stutter – Part A	80
2.35	Stutter – Part B	81
2.36	A FSM representing Stutter, based on control flow graphs of the methods	82
2.37	A FSM representing Stutter, based on the structure of the software	83
2.38	A FSM representing Stutter, based on modeling state variables	84
2.39	A FSM representing Stutter, based on the specifications	85
2.40	Class Queue for exercises.	86
2.41	ATM actor and use cases	88
2.42	Activity graph for ATM withdraw funds	90
2.43	Examples of path products	92
2.44	Null path that leads to additive identity ϕ	93
2.45	A or lambda	94
2.46	Example graph to show reduction to path expressions	94
2.47	After step 1 in path expression reduction	95
2.48	After step 2 in path expression reduction	95
2.49	After step 3 in path expression reduction	95
2.50	Removing arbitrary nodes	95
2.51	Eliminating node n_2	95
2.52	Removing sequential edges	95
2.53	Removing self-loop edges	96
2.54	Final graph with one path expression	96
2.55	Graph example for computing maximum number of paths	97
2.56	Graph example for complementary path analysis	99
3.1	Subsumption relations among logic coverage criteria	113
3.2	TriTyp – Part A	121
3.3	TriTyp – Part B	122
3.4	Calendar method	132
3.5	FSM for a memory car seat – Lexus 2003 ES300	135
3.6	Fault detection relationships	143
4.1	Partitioning of input domain D into three blocks	151
4.2	Subsumption relations among input space partitioning criteria	163
5.1	Method Min and six mutants	177
5.2	Mutation testing process	181
5.3	Partial truth table for $(a \wedge b)$	187
5.4	Finite state machine for SMV specification	199
5.5	Mutated finite state machine for SMV specification	200
5.6	Finite state machine for bank example	202
5.7	Finite state machine for bank example grammar	202
5.8	Simple XML message for books	204
5.9	XML schema for books	205

7.1 Example class hierarchy in UML	238
7.2 Data flow anomalies with polymorphism	238
7.3 Calls to $d()$ when object has various actual types	239
7.4 ITU: Descendant with no overriding methods	241
7.5 SDA, SDIH: State definition anomalies	243
7.6 IISD: Example of indirect inconsistent state definition	244
7.7 ACB1: Example of anomalous construction behavior	245
7.8 SVA: State visibility anomaly	247
7.9 Sample class hierarchy (a) and associated type families (b)	248
7.10 Control flow graph fragment (a) and associated definitions and uses (b)	249
7.11 Def-use pairs in object-oriented software	250
7.12 Control flow schematic for prototypical coupling sequence	251
7.13 Sample class hierarchy and <i>def-use</i> table	252
7.14 Coupling sequence: o of type A (a) bound to instance of A (b), B (c) or C (d)	253
8.2 Node coverage instrumentation	269
8.3 Edge coverage instrumentation	270
8.4 All uses coverage instrumentation	271
8.5 Correlated active clause coverage instrumentation	273

List of Tables

2.1	Defs and uses at each node in the CFG for TestPat	<i>page</i> 57
2.2	Defs and uses at each edge in the CFG for TestPat	57
2.3	Du-path sets for each variable in TestPat	58
2.4	Test paths to satisfy all du-paths coverage on TestPat	59
2.5	Test paths and du-paths covered on TestPat	59
3.1	Reachability for Triang predicates	123
3.2	Reachability for Triang predicates – reduced by solving for triOut	124
3.3	Predicate coverage for Triang	125
3.4	Clause coverage for Triang	126
3.5	Correlated active clause coverage for Triang	127
3.6	Correlated active clause coverage for cal() preconditions	133
3.7	Predicates from memory seat example	136
3.8	DNF fault classes	143
4.1	First partitioning of TriTyp's inputs (interface-based)	156
4.2	Second partitioning of TriTyp's inputs (interface-based)	157
4.3	Possible values for blocks in the second partitioning in Table 4.2	157
4.4	Geometric partitioning of TriTyp's inputs (functionality-based)	158
4.5	Correct geometric partitioning of TriTyp's inputs (functionality-based)	158
4.6	Possible values for blocks in geometric partitioning in Table 4.5	159
4.7	Examples of invalid block combinations	165
5.1	Java's access levels	193
6.1	Testing objectives and activities during requirements analysis and specification	221
6.2	Testing objectives and activities during system and software design	222
6.3	Testing objectives and activities during intermediate design	222
6.4	Testing objectives and activities during detailed design	223
6.5	Testing objectives and activities during implementation	223
6.6	Testing objectives and activities during integration	224
6.7	Testing objectives and activities during system deployment	224
6.8	Testing objectives and activities during operation and maintenance	225
7.1	Faults and anomalies due to inheritance and polymorphism	240

xiv **List of Tables**

7.2	ITU: Code example showing inconsistent type usage	242
7.3	IC: Incomplete construction of state variable <i>fd</i>	246
7.4	Summary of sample coupling paths	254
7.5	Binding triples for coupling sequence from class hierarchy in Figure 7.13	254

Preface

This book presents software testing as a practical engineering activity, essential to producing high-quality software. It is designed to be used as the primary textbook in either an undergraduate or graduate course on software testing, as a supplement to a general course on software engineering or data structures, and as a resource for software test engineers and developers. This book has a number of unique features:

- It organizes the complex and confusing landscape of test coverage criteria with a novel and extremely simple structure. At a technical level, software testing is based on satisfying coverage criteria. The book's central observation is that there are few truly different coverage criteria, each of which fits easily into one of four categories: graphs, logical expressions, input space, and syntax structures. This not only simplifies testing, but it also allows a convenient and direct theoretical treatment of each category. This approach contrasts strongly with the traditional view of testing, which treats testing at each phase in the development process differently.
- It is designed and written to be a textbook. The writing style is direct, it builds the concepts from the ground up with a minimum of required background, and it includes lots of examples, homework problems, and teaching materials. It provides a balance of theory and practical application, presenting testing as a collection of objective, quantitative activities that can be measured and repeated. The theoretical concepts are presented when needed to support the practical activities that test engineers follow.
- It assumes that testing is part of a mental discipline that helps all IT professionals develop higher-quality software. Testing is not an anti-engineering activity, and it is not an inherently destructive process. Neither is it only for testing specialists or domain experts who know little about programming or math.
- It is designed with modular, interconnecting pieces; thus it can be used in multiple courses. Most of the book requires only basic discrete math and introductory programming, and the parts that need more background are clearly marked. By

using the appropriate sections, this book can support several classes, as described later in the preface.

- It assumes the reader is learning to be an engineer whose goal is to produce the best possible software with the lowest possible cost. The concepts in this book are well grounded in theory, are practical, and most are currently in use.

WHY SHOULD THIS BOOK BE USED?

Not very long ago, software development companies could afford to employ programmers who could not test and testers who could not program. For most of the industry, it was not necessary for either group to know the technical principles behind software testing or even software development. Software testing in industry historically has been a nontechnical activity. Industry viewed testing primarily from the managerial and process perspective and had limited expectations of practitioners' technical training.

As the software engineering profession matures, and as software becomes more pervasive in everyday life, there are increasingly stringent requirements for software reliability, maintainability, and security. Industry must respond to these changes by, among other things, improving the way software is tested. This requires increased technical expertise on the part of test engineers, as well as increased emphasis on testing by software developers. The good news is that the knowledge and technology are available and based on over 30 years of research and practice. This book puts that knowledge into a form that students, test engineers, test managers, and developers can access.

At the same time, it is relatively rare to find courses that teach testing in universities. Only a few undergraduate courses exist, almost no masters degree programs in computer science or software engineering require a course in software testing, and only a few dozen have an elective course. Not only is testing not covered as an essential part of undergraduate computer science education, most computer science students either never gain any knowledge about testing, or see only a few lectures as part of a general course in software engineering.

The authors of this book have been teaching software testing to software engineering and computer science students for more than 15 years. Over that time we somewhat reluctantly came to the conclusion that no one was going to write the book we wanted to use. Rather, to get the book we wanted, we would have to write it.

Previous testing books have presented software testing as a relatively simple subject that relies more on process than technical understanding of how software is constructed, as a complicated and fractured subject that requires detailed understanding of numerous software development technologies, or as a completely theoretical subject that can be mastered only by mathematicians and theoretical computer scientists. Most books on software testing are organized around the phases in a typical software development lifecycle, an approach that has the unfortunate side effect of obscuring common testing themes. Finally, most testing books are written as reference books, not textbooks. As a result, only instructors with prior expertise in software testing can easily teach the subject. **This book is accessible to instructors who are not already testing experts.**

This book differs from other books on software testing in other important ways. Many books address managing the testing process. While this is important, it is equally important to give testers specific techniques grounded in basic theory. This book provides a balance of theory and practical application. This is important information that software companies must have; however, this book focuses specifically on the technical nuts-and-bolts issues of designing and creating tests. Other testing books currently on the market focus on specific techniques or activities, such as system testing or unit testing. This book is intended to be comprehensive over the entire software development process and to cover as many techniques as possible.

As stated previously, the motivation for this book is to support courses in software testing. Our first target was our own software testing course in our Software Engineering MS program at George Mason University. This popular elective is taught to about 30 computer science and software engineering students every semester. We also teach PhD seminars in software testing, industry short courses on specialized aspects, and lectures on software testing in various undergraduate courses. Although few undergraduate courses on software testing exist, we believe that they should exist, and we expect they will in the near future. Most testing books are not designed for classroom use. We specifically wrote this book to support our classroom activities, and it is no accident that the syllabus for our testing course, available on the book's Web site (www.introsoftwaretesting.com), closely follows the table of contents for this book.

This book includes numerous carefully worked examples to help students and teachers alike learn the sometimes complicated concepts. The instructor's resources include high-quality powerpoint slides, presentation hints, solutions to exercises, and working software. Our philosophy is that we are doing more than writing a book; we are offering our course to the community. One of our goals was to write material that is scholarly and true to the published research literature, but that is also accessible to nonresearchers. Although the presentation in the book is quite a bit different from the research papers that the material is derived from, the essential ideas are true to the literature. To make the text flow more smoothly, we have removed the references from the presentation. For those interested in the research genealogy, each chapter closes with a bibliographic notes section that summarizes where the concepts come from.

WHO SHOULD READ THIS BOOK?

Students who read and use this book will learn the fundamental principles behind software testing, and how to apply these principles to produce better software, faster. They will not only become better programmers, they will also be prepared to carry out high-quality testing activities for their future employers. *Instructors* will be able to use this book in the classroom, even without prior practical expertise in software testing. The numerous exercises and thought-provoking problems, classroom-ready and classroom-tested slides, and suggested outside activities make this material teachable by instructors who are not already experts in software testing. *Research students* such as beginning PhD students will find this book to be an invaluable resource as a starting point to the field. The theory is sound and clearly

xviii **Preface**

presented, the practical applications reveal what is useful and what is not, and the advanced reading and bibliographic notes provide pointers into the literature. Although the set of research students in software testing is a relatively small audience, we believe it is a key audience, because a common, easily achievable baseline would reduce the effort required for research students to join the community of testing researchers. *Researchers* who are already familiar with the field will find the criteria-approach to be novel and interesting. Some may disagree with the pedagogical approach, but we have found that the view that testing is an application of only a few criteria to a very few software structures to be very helpful to our research. We hope that testing research in the future will draw away from searches for more criteria to novel uses and evaluations of existing criteria.

Testers in the industry will find this book to be an invaluable collection of techniques that will help improve their testing, no matter what their current process is. The criteria presented here are intended to be used as a “toolbox” of tricks that can be used to find faults. *Developers* who read this book will find numerous ways to improve their own software. Their self-testing activities can become more efficient and effective, and the discussions of software faults that test engineers search for will help developers avoid them. To paraphrase a famous parable, if you want to teach a person to be a better fisherman, explain how and where the fish swim. Finally, *managers* will find this book to be a useful explanation of how clever test engineers do their job, and of how test tools work. They will be able to make more effective decisions regarding hiring, promotions, and purchasing tools.

HOW CAN THIS BOOK BE USED?

A major advantage of the structure of this book is that it can be easily used for several different courses. Most of the book depends on material that is taught very early in college and some high schools: basic concepts from data structures and discrete math. The sections are organized so that the early material in each chapter is accessible to less advanced students, and material that requires more advanced knowledge is clearly marked.

Specifically, the book defines six separate sets of chapter sections that form *streams* through the book:

1. A module within a CS II course
2. A sophomore-level course on software testing
3. A module in a general software engineering course
4. A senior-level course on software testing
5. A first-year MS level course on software testing
6. An advanced graduate research-oriented course on software testing
7. Industry practioner relevant sections

The stream approach is illustrated in the abbreviated table of contents in the figure shown on pp. xix–xx. Each chapter section is marked with which stream it belongs too. Of course, individual instructors, students, and readers may prefer to adapt the stream to their own interests or purposes. We suggest that the first two sections of Chapter 1 and the first two sections of Chapter 6 are appropriate reading for a module in a data structures (CS II) class, to be followed by a simple

- Stream 1: Module in a CS II course.
- Stream 2: Sophomore-level course on software testing.
- Stream 3: Module in a general software engineering course.
- Stream 4: Senior-level course on software testing.
- Stream 5: First-year MS course on software testing.
- Stream 6: Advanced graduate research-oriented course on software testing.
- Stream 7: Industry practitioner relevant sections

	STREAMS						
	1	2	3	4	5	6	7
Part I: Overview							
Chapter 1. Introduction	■	■	■	■	■	■	■
1.1 Activities of a Test Engineer	■	■	■	■	■	■	■
1.2 Software Testing Limitations and Terminology	■	■	■	■	■	■	■
1.3 Coverage Criteria for Testing		■	■	■	■	■	■
1.4 Older Software Testing Terminology					■	■	
1.5 Bibliographic Notes						■	
Part II: Coverage Criteria							
Chapter 2. Graph Coverage	■	■	■	■	■	■	■
2.1 Overview	■	■	■	■	■	■	■
2.2 Graph Coverage Criteria	■	■	■	■	■	■	■
2.3 Graph Coverage for Source Code	■	■	■	■	■	■	■
2.4 Graph Coverage for Design Elements				■	■	■	■
2.5 Graph Coverage for Specifications				■	■	■	■
2.6 Graph Coverage for Use Cases				■	■	■	■
2.7 Representing Graphs Algebraically					■	■	
2.8 Bibliographic Notes						■	
Chapter 3. Logic Coverage	■	■	■	■	■	■	■
3.1 Overview: Logic Predicates and Clauses	■	■	■	■	■	■	■
3.2 Logic Expression Coverage Criteria	■	■	■	■	■	■	■
3.3 Structural Logic Coverage of Programs	■			■	■	■	
3.4 Specification-Based Logic Coverage				■	■	■	
3.5 Logic Coverage of Finite State Machines			■	■	■	■	
3.6 Disjunctive Normal Form Criteria					■	■	
3.7 Bibliographic Notes						■	
Chapter 4. Input Space Partitioning			■	■	■	■	■
4.1 Input Domain Modeling			■	■	■	■	■
4.2 Combination Strategies Criteria			■	■	■	■	■
4.3 Constraints among Partitions					■	■	■
4.4 Bibliographic Notes						■	
Chapter 5. Syntax-Based Testing	■	■	■	■	■	■	■
5.1 Syntax-Based Coverage Criteria	■	■	■	■	■	■	■
5.2 Program-Based Grammars			■	■	■	■	■
5.3 Integration and Object-Oriented Testing				■	■	■	
5.4 Specification-Based Grammars					■	■	
5.5 Input Space Grammars	■	■	■	■	■	■	■
5.6 Bibliographic Notes						■	

xx **Preface**

- Stream 1: Module in a CS II course.
- Stream 2: Sophomore-level course on software testing.
- Stream 3: Module in a general software engineering course.
- Stream 4: Senior-level course on software testing.
- Stream 5: First-year MS course on software testing.
- Stream 6: Advanced graduate research-oriented course on software testing.
- Stream 7: Industry practitioner relevant sections

	STREAMS						
	1	2	3	4	5	6	7
Part III: Applying Criteria in Practice							
Chapter 6. Practical Considerations	■	■	■	■	□	□	□
6.1 Regression Testing	■	■	■	■	□	□	□
6.2 Integration and Testing	■	■	■	■	□	□	□
6.3 Test Process			■	■	□	□	□
6.4 Test Plans			■	■	□	□	□
6.5 Identifying Correct Outputs		■	■	■	□	□	
6.5 Bibliographic Notes						□	
Chapter 7. Engineering Criteria for Technologies				■	□	□	□
7.1 Testing Object-Oriented Software				■	□	□	□
7.2 Testing Web Applications and Web Services					□	□	□
7.3 Testing Graphical User Interfaces					□	□	□
7.4 Real-Time Software and Embedded Software					□	□	□
7.5 Bibliographic Notes						□	
Chapter 8. Building Testing Tools					□	□	□
8.1 Instrumentation for Graph and Logical Expression Criteria					□	□	□
8.2 Building Mutation Testing Tools						□	
8.3 Bibliographic Notes						□	
Chapter 9. Challenges in Testing Software					□	□	□
9.1 Testing for Emergent Properties: Safety and Security					□	□	□
9.2 Software Testability					□	□	□
9.3 Test Criteria and the Future of Software Testing					□	□	
9.4 Bibliographic Notes						□	

assignment. Our favorite is to ask the students to retrieve one of their previously graded programs and satisfy some simple test criterion like branch coverage. We offer points for every fault found, driving home two concepts: an “A” grade doesn’t mean the program always works, and finding faults is a good thing.

The sophomore-level course on software testing (stream 2) is designed to immediately follow a data structures course (CS II). The marked sections contain material that depends only on data structures and discrete math.

A module in a general software engineering course (stream 3) could augment the survey material typical in such courses. The sections marked provide basic literacy in software testing.

The senior-level course on software testing (stream 4) is the primary target for this text. It adds material that requires a little more sophistication in terms of

software development than the sophomore stream. This includes sections in Chapter 2 on data flow testing, sections that involve integration testing of multiple modules, and sections that rely on grammars or finite state machines. Most senior computer science students will have seen this material in their other courses. Most of the sections that appear in stream 4 but not stream 2 could be added to stream 2 with appropriate short introductions. It is important to note that a test engineer does not need to know all the theory of parsing to use data flow testing or all the theory on finite state machines to use statecharts for testing.

The graduate-level course on software testing (stream 5) adds some additional sections that rely on a broader context and that require more theoretical maturity. For example, these sections use knowledge of elementary formal methods, polymorphism, and some of the UML diagrams. Some of the more advanced topics and the entire chapter on building testing tools are also intended for a graduate audience. This chapter could form the basis for a good project, for example, to implement a simple coverage analyzer.

An advanced graduate course in software testing with a research emphasis such as a PhD seminar (stream 6) includes issues that are still unproven and research in nature. The bibliographic notes are recommended only for these students as indicators for future in-depth reading.

Finally, sections that are reasonably widely used in industry, especially those that have commercial tool support, are marked for stream 7. These sections have a minimum of theory and omit criteria that are still of questionable usefulness.

Extensive supplementary materials, including sample syllabuses, PowerPoint slides, presentation hints, solutions to exercises, working software, and errata are available on the book's companion Web site.

ACKNOWLEDGMENTS

Many people helped us write this book. Not only have the students in our Software Testing classes at George Mason been remarkably tolerant of using a work in progress, they have enthusiastically provided feedback on how to improve the text. We cannot acknowledge all by name (ten semesters worth of students have used it!), but the following have made especially large contributions: Aynur Abdu-razik, Muhammad Abdulla, Yuquin Ding, Jyothi Chinman, Blaine Donley, Patrick Emery, Brian Geary, Mark Hinkle, Justin Hollingsworth, John King, Yuelan Li, Xiaojuan Liu, Chris Magrin, Jyothi Reddy, Raimi Rufai, Jeremy Schneider, Bill Shelton, Frank Shukis, Quansheng Xiao, and Linzhen Xue. We especially appreciate those who generously provided extensive comments on the entire book: Guillermo Calderon-Meza, Becky Hartley, Gary Kaminski, and Andrew J. Offutt. We gratefully acknowledge the feedback of early adopters at other educational institutions: Roger Alexander, Jane Hayes, Ling Liu, Darko Marinov, Arthur Reyes, Michael Shin, and Tao Xie. We also want to acknowledge several people who provided material for the book: Roger Alexander, Mats Grindal, Hong Huang, Gary Kaminski, Robert Nilsson, Greg Williams, Wuzhi Xu. We were lucky to receive excellent suggestion from Lionel Briand, Renée Bryce, Kim King, Sharon Ritchey, Bo Sanden, and Steve Schach. We are grateful to our editor, Heather Bergman,

xxii **Preface**

for providing unwavering support and enforcing the occasional deadline to move the project along, as well as Kerry Cahill from Cambridge University Press for very strong support on this project.

We also acknowledge George Mason University for supporting both of us on sabbaticals and for providing GTA support at crucial times. Our department Chair, Hassan Gomaa, has enthusiastically supported this effort.

Finally, of course none of this is possible without the support of our families. Thanks to Becky, Jian, Steffi, Matt, Joyce, and Andrew for keeping us grounded in reality and helping keep us happy for the past five years.

Just as all programs contain faults, all texts contain errors. Our text is no different. And, as responsibility for software faults rests with the developers, responsibility for errors in this text rests with us, the authors. In particular, the bibliographic notes sections reflect our perspective of the testing field, a body of work we readily acknowledge as large and complex. We apologize in advance for omissions, and invite pointers to relevant citations.

Paul Ammann
Jeff Offutt