Introduction to Software Testing

This extensively classroom-tested text takes an innovative approach to explaining software testing that defines it as the process of applying a few precise, general-purpose criteria to a structure or model of the software. The text incorporates cutting-edge developments, including techniques to test modern types of software such as OO, web applications, and embedded software. This revised second edition significantly expands coverage of the basics, thoroughly discussing test automaton frameworks, and adds new, improved examples and numerous exercises. Key features include:

- The theory of coverage criteria is carefully, cleanly explained to help students understand concepts before delving into practical applications.
- Extensive use of the JUnit test framework gives students practical experience in a test framework popular in industry.
- Exercises feature specifically tailored tools that allow students to check their own work.
- Instructor's manual, PowerPoint slides, testing tools for students, and example software programs in Java are available from the book's website.

Paul Ammann is Associate Professor of Software Engineering at George Mason University. He earned the Volgenau School's Outstanding Teaching Award in 2007. He led the development of the Applied Computer Science degree, and has served as Director of the MS Software Engineering program. He has taught courses in software testing, applied object-oriented theory, formal methods for software engineering, web software, and distributed software engineering. Ammann has published more than eighty papers in software engineering, with an emphasis on software testing, security, dependability, and software engineering education.

Jeff Offutt is Professor of Software Engineering at George Mason University. He leads the MS in Software Engineering program, teaches software engineering courses at all levels, and developed new courses on several software engineering subjects. He was awarded the George Mason University Teaching Excellence Award, Teaching with Technology, in 2013. Offutt has published more than 165 papers in areas such as model-based testing, criteria-based testing, test automaton, empirical software engineering, and software maintenance. He is Editor-in-Chief of the *Journal of Software Testing, Verification and Reliability*; helped found the IEEE International Conference on Software Testing; and is the founder of the μJava project.

INTRODUCTION TO SOFTWARE TESTING

Paul Ammann
George Mason University

Jeff Offutt
George Mason University

CAMBRIDGE
UNIVERSITY PRESS

Shaftesbury Road, Cambridge CB2 8EA, United Kingdom

One Liberty Plaza, 20th Floor, New York, NY 10006, USA

477 Williamstown Road, Port Melbourne, VIC 3207, Australia

314–321, 3rd Floor, Plot 3, Splendor Forum, Jasola District Centre, New Delhi – 110025, India

103 Penang Road, #05–06/07, Visioncrest Commercial, Singapore 238467

Cambridge University Press is part of Cambridge University Press & Assessment, a department of the University of Cambridge.

We share the University's mission to contribute to society through the pursuit of education, learning and research at the highest international levels of excellence.

www.cambridge.org
Information on this title: www.cambridge.org/9781107172012

DOI: 10.1017/9781316771273

First published 2017

A catalogue record for this publication is available from the British Library

Library of Congress Cataloging-in-Publication data
Names: Ammann, Paul, 1961– author. | Offutt, Jeff, 1961– author.
Title: Introduction to software testing / Paul Ammann, George Mason University, Jeff Offutt, George Mason University.
Description: Edition 2. | Cambridge, United Kingdom; New York, NY, USA: Cambridge University Press, [2016]
Identifiers: LCCN 2016032808 | ISBN 9781107172012 (hardback)
Subjects: LCSH: Computer software–Testing.
Classification: LCC QA76.76.T48 A56 2016 | DDC 005.3028/7–dc23
LC record available at https://lccn.loc.gov/2016032808

ISBN 978-1-107-17201-2 Hardback

Additional resources for this publication at https://cs.gmu.edu/~offutt/softwaretest/.

Contents

List of Figures *page* ix
List of Tables xii
Preface to the Second Edition xiv

Part 1 Foundations 1
1 Why Do We Test Software? 3
1.1 When Software Goes Bad 4
1.2 Goals of Testing Software 8
1.3 Bibliographic Notes 17
2 Model-Driven Test Design 19
2.1 Software Testing Foundations 20
2.2 Software Testing Activities 21
2.3 Testing Levels Based on Software Activity 22
2.4 Coverage Criteria 25
2.5 Model-Driven Test Design 27
2.5.1 Test Design 28
2.5.2 Test Automation 29
2.5.3 Test Execution 29
2.5.4 Test Evaluation 29
2.5.5 Test Personnel and Abstraction 29
2.6 Why MDTD Matters 31
2.7 Bibliographic Notes 33
3 Test Automation 35
3.1 Software Testability 36
3.2 Components of a Test Case 36

3.3 A Test Automation Framework 39
3.3.1 The JUnit Test Framework 40
3.3.2 Data-Driven Tests 44
3.3.3 Adding Parameters to Unit Tests 47
3.3.4 JUnit from the Command Line 50
3.4 Beyond Test Automation 50
3.5 Bibliographic Notes 53

4 Putting Testing First 54
4.1 Taming the Cost-of-Change Curve 54
4.1.1 Is the Curve Really Tamed? 56
4.2 The Test Harness as Guardian 57
4.2.1 Continuous Integration 58
4.2.2 System Tests in Agile Methods 59
4.2.3 Adding Tests to Legacy Systems 60
4.2.4 Weaknesses in Agile Methods for Testing 61
4.3 Bibliographic Notes 62

5 Criteria-Based Test Design 64
5.1 Coverage Criteria Defined 64
5.2 Infeasibility and Subsumption 68
5.3 Advantages of Using Coverage Criteria 68
5.4 Next Up 70
5.5 Bibliographic Notes 70

Part 2 Coverage Criteria 73

6 Input Space Partitioning 75
6.1 Input Domain Modeling 77
6.1.1 Interface-Based Input Domain Modeling 79
6.1.2 Functionality-Based Input Domain Modeling 79
6.1.3 Designing Characteristics 80
6.1.4 Choosing Blocks and Values 81
6.1.5 Checking the Input Domain Model 84
6.2 Combination Strategies Criteria 86
6.3 Handling Constraints Among Characteristics 92
6.4 Extended Example: Deriving an IDM from JavaDoc 93
6.4.1 Tasks in Designing IDM-Based Tests 93
6.4.2 Designing IDM-Based Tests for Iterator 94
6.5 Bibliographic Notes 102

7 Graph Coverage 106
7.1 Overview 106
7.2 Graph Coverage Criteria 111
7.2.1 Structural Coverage Criteria 112
7.2.2 Touring, Sidetrips, and Detours 116
7.2.3 Data Flow Criteria 123
7.2.4 Subsumption Relationships Among Graph Coverage Criteria 130

7.3 Graph Coverage for Source Code 131
7.3.1 Structural Graph Coverage for Source Code 132
7.3.2 Data Flow Graph Coverage for Source Code 136
7.4 Graph Coverage for Design Elements 146
7.4.1 Structural Graph Coverage for Design Elements 147
7.4.2 Data Flow Graph Coverage for Design Elements 148
7.5 Graph Coverage for Specifications 157
7.5.1 Testing Sequencing Constraints 157
7.5.2 Testing State Behavior of Software 160
7.6 Graph Coverage for Use Cases 169
7.6.1 Use Case Scenarios 171
7.7 Bibliographic Notes 173

8 Logic Coverage 177
8.1 Semantic Logic Coverage Criteria (Active) 178
8.1.1 Simple Logic Expression Coverage Criteria 179
8.1.2 Active Clause Coverage 181
8.1.3 Inactive Clause Coverage 185
8.1.4 Infeasibility and Subsumption 186
8.1.5 Making a Clause Determine a Predicate 187
8.1.6 Finding Satisfying Values 192
8.2 Syntactic Logic Coverage Criteria (DNF) 197
8.2.1 Implicant Coverage 198
8.2.2 Minimal DNF 199
8.2.3 The MUMCUT Coverage Criterion 200
8.2.4 Karnaugh Maps 205
8.3 Structural Logic Coverage of Programs 208
8.3.1 Satisfying Predicate Coverage 212
8.3.2 Satisfying Clause Coverage 213
8.3.3 Satisfying Active Clause Coverage 215
8.3.4 Predicate Transformation Issues 217
8.3.5 Side Effects in Predicates 220
8.4 Specification-Based Logic Coverage 223
8.5 Logic Coverage of Finite State Machines 226
8.6 Bibliographic Notes 231

9 Syntax-Based Testing 234
9.1 Syntax-Based Coverage Criteria 234
9.1.1 Grammar-Based Coverage Criteria 234
9.1.2 Mutation Testing 237
9.2 Program-Based Grammars 241
9.2.1 BNF Grammars for Compilers 241
9.2.2 Program-Based Mutation 242
9.3 Integration and Object-Oriented Testing 259
9.3.1 BNF Integration Testing 259
9.3.2 Integration Mutation 259
9.4 Specification-Based Grammars 266
9.4.1 BNF Grammars 266

9.4.2 Specification-Based Mutation 267
9.5 Input Space Grammars 271
9.5.1 BNF Grammars 271
9.5.2 Mutating Input Grammars 273
9.6 Bibliographic Notes 281

Part 3 Testing in Practice 283

10 Managing the Test Process 285

10.1 Overview 285
10.2 Requirements Analysis and Specification 286
10.3 System and Software Design 287
10.4 Intermediate Design 288
10.5 Detailed Design 288
10.6 Implementation 289
10.7 Integration 289
10.8 System Deployment 290
10.9 Operation and Maintenance 290
10.10 Implementing the Test Process 291
10.11 Bibliographic Notes 291

11 Writing Test Plans 292

11.1 Level Test Plan Example Template 293
11.2 Bibliographic Notes 295

12 Test Implementation 296

12.1 Integration Order 297
12.2 Test Doubles 298
12.2.1 Stubs and Mocks: Variations of Test Doubles 299
12.2.2 Using Test Doubles to Replace Components 300
12.3 Bibliographic Notes 303

13 Regression Testing for Evolving Software 304

13.1 Bibliographic Notes 306

14 Writing Effective Test Oracles 308

14.1 What Should Be Checked? 308
14.2 Determining Correct Values 310
14.2.1 Specification-Based Direct Verification of Outputs 310
14.2.2 Redundant Computations 311
14.2.3 Consistency Checks 312
14.2.4 Metamorphic Testing 312
14.3 Bibliographic Notes 314

List of Criteria 316

Bibliography 318

Index 337

Figures

1.1	Cost of late testing	12
2.1	Reachability, Infection, Propagation, Revealability (RIPR) model	21
2.2	Activities of test engineers	22
2.3	Software development activities and testing levels – the "V Model"	23
2.4	Model-driven test design	30
2.5	Example method, CFG, test requirements and test paths	31
3.1	`Calc` class example and JUnit test	41
3.2	Minimum element class	42
3.3	First three JUnit tests for `Min` class	43
3.4	Remaining JUnit test methods for `Min` class	45
3.5	Data-driven test class for `Calc`	46
3.6	JUnit Theory about sets	48
3.7	JUnit Theory data values	48
3.8	AllTests for the `Min` class example	49
4.1	Cost-of-change curve	55
4.2	The role of user stories in developing system (acceptance) tests	60
6.1	Partitioning of input domain D into three blocks	76
6.2	Subsumption relations among input space partitioning criteria	89
7.1	Graph (a) has a single initial node, graph (b) multiple initial nodes, and graph (c) (rejected) with no initial nodes	108
7.2	Example of paths	108
7.3	A Single-Entry Single-Exit graph	110
7.4	Test case mappings to test paths	110
7.5	A set of test cases and corresponding test paths	111
7.6	A graph showing Node Coverage and Edge Coverage	114
7.7	Two graphs showing prime path coverage	116
7.8	Graph with a loop	117
7.9	Tours, sidetrips, and detours in graph coverage	117
7.10	An example for prime test paths	119
7.11	A graph showing variables, def sets and use sets	124
7.12	A graph showing an example of du-paths	126

7.13 Graph showing explicit def and use sets 128
7.14 Example of the differences among the three data flow coverage criteria 129
7.15 Subsumption relations among graph coverage criteria 132
7.16 CFG fragment for the *if-else* structure 133
7.17 CFG fragment for the *if* structure without an *else* 133
7.18 CFG fragment for the *if* structure with a *return* 133
7.19 CFG fragment for the *while* loop structure 134
7.20 CFG fragment for the *for* loop structure 134
7.21 CFG fragment for the *do-while* structure 135
7.22 CFG fragment for the *while* loop with a *break* structure 136
7.23 CFG fragment for the *case* structure 136
7.24 CFG fragment for the *try-catch* structure 137
7.25 Method `patternIndex()` for data flow example 141
7.26 A simple call graph 147
7.27 A simple inheritance hierarchy 148
7.28 An inheritance hierarchy with objects instantiated 149
7.29 An example of parameter coupling 150
7.30 Coupling du-pairs 151
7.31 Last-defs and first-uses 151
7.32 Quadratic root program 153
7.33 Def-use pairs under intra-procedural and inter-procedural data flow 154
7.34 Def-use pairs in object-oriented software 155
7.35 Def-use pairs in web applications and other distributed software 155
7.36 Control flow graph using the File ADT 159
7.37 Elevator door open transition 161
7.38 `Watch`–Part A 163
7.39 `Watch`–Part B 164
7.40 An FSM representing `Watch`, based on control flow graphs of the methods 165
7.41 An FSM representing `Watch`, based on the structure of the software 165
7.42 An FSM representing `Watch`, based on modeling state variables 167
7.43 ATM actor and use cases 169
7.44 Activity graph for ATM withdraw funds 172
8.1 Subsumption relations among logic coverage criteria 187
8.2 Fault detection relationships 202
8.3 `Thermostat` class 210
8.4 PC true test for `Thermostat` class 213
8.5 CC test assignments for `Thermostat` class 214
8.6 `Calendar` method 226
8.7 FSM for a memory car seat–Nissan Maxima 2012 227
9.1 Method `Min` and six mutants 243
9.2 Mutation testing process 246
9.3 Partial truth table for $(a \wedge b)$ 253
9.4 Finite state machine for SMV specification 268
9.5 Mutated finite state machine for SMV specification 269
9.6 Finite state machine for bank example 271

9.7 Finite state machine for bank example grammar 272
9.8 Simple XML message for books 274
9.9 XML schema for books 275
12.1 Test double example: Replacing a component 300

Tables

6.1	First partitioning of triang()'s inputs (interface-based)	82
6.2	Second partitioning of triang()'s inputs (interface-based)	82
6.3	Possible values for blocks in the second partitioning in Table 6.2	83
6.4	Geometric partitioning of triang()'s inputs (functionality-based)	83
6.5	Correct geometric partitioning of triang()'s inputs (functionality-based)	84
6.6	Possible values for blocks in geometric partitioning in Table 6.5	84
6.7	Examples of invalid block combinations	93
6.8	Table A for Iterator example: Input parameters and characteristics	95
6.9	Table B for Iterator example: Partitions and base case	96
6.10	Table C for Iterator example: Refined test requirements	97
6.11	Table A for Iterator example: Input parameters and characteristics (revised)	100
6.12	Table C for Iterator example: Refined test requirements (revised)	101
7.1	Defs and uses at each node in the CFG for patternIndex()	139
7.2	Defs and uses at each edge in the CFG for patternIndex()	139
7.3	du-path sets for each variable in patternIndex()	140
7.4	Test paths to satisfy all du-paths coverage on patternIndex()	142
7.5	Test paths and du-paths covered in patternIndex()	143
8.1	DNF fault classes	201
8.2	Reachability for Thermostat predicates	211
8.3	Clauses in the Thermostat predicate on lines 28–30	212
8.4	Correlated active clause coverage for Thermostat	215
8.5	Correlated active clause coverage for cal() preconditions	225
8.6	Predicates from memory seat example	229
9.1	Java's access levels	261
10.1	Testing objectives and activities during requirements analysis and specification	287
10.2	Testing objectives and activities during system and software design	288
10.3	Testing objectives and activities during intermediate design	289
10.4	Testing objectives and activities during detailed design	289

10.5 Testing objectives and activities during implementation 290
10.6 Testing objectives and activities during integration 290
10.7 Testing objectives and activities during system deployment 291
10.8 Testing objectives and activities during operation and maintenance 291

Preface to the Second Edition

Much has changed in the field of testing in the eight years since the first edition was published. High-quality testing is now more common in industry. Test automation is now ubiquitous, and almost assumed in large segments of the industry. Agile processes and test-driven development are now widely known and used. Many more colleges offer courses on software testing, both at the undergraduate and graduate levels. The ACM curriculum guidelines for software engineering include software testing in several places, including as a strongly recommended course [Ardis et al., 2015].

The second edition of *Introduction to Software Testing* incorporates new features and material, yet retains the structure, philosophy, and online resources that have been so popular among the hundreds of teachers who have used the book.

What is new about the second edition?

The first thing any instructor has to do when presented with a new edition of a book is analyze what must be changed in the course. Since we have been in that situation many times, we want to make it as easy as possible for our audience. We start with a chapter-to-chapter mapping.

First Edition	Second Edition	Topic
Part I: Foundations		
Chapter 1	Chapter 01	Why do we test software? (motivation)
	Chapter 02	Model-driven test design (abstraction)
	Chapter 03	Test automation (JUnit)
	Chapter 04	Putting testing first (TDD)
	Chapter 05	Criteria-based test design (criteria)
Part II: Coverage Criteria		
Chapter 2	Chapter 07	Graph coverage
Chapter 3	Chapter 08	Logic coverage
Chapter 4	Chapter 09	Syntax-based testing
Chapter 5	Chapter 06	Input space partitioning

Part III: Testing in Practice		
Chapter 6	Chapter 10	Managing the test process
	Chapter 11	Writing test plans
	Chapter 12	Test implementation
	Chapter 13	Regression testing for evolving software
	Chapter 14	Writing effective test oracles
Chapter 7	*N/A*	Technologies
Chapter 8	*N/A*	Tools
Chapter 9	*N/A*	Challenges

The most obvious, and largest change, is that the introductory chapter 1 from the first edition has been expanded into five separate chapters. This is a significant expansion that we believe makes the book much better. The new part 1 grew out of our lectures. After the first edition came out, we started adding more foundational material to our testing courses. These new ideas were eventually reorganized into five new chapters. The new chapter 01[1] has much of the material from the first edition chapter 1, including motivation and basic definitions. It closes with a discussion of the cost of late testing, taken from the 2002 RTI report that is cited in every software testing research proposal. After completing the first edition, we realized that the key novel feature of the book, viewing test design as an abstract activity that is independent of the software artifact being used to design the tests, implied a completely different process. This led to chapter 02, which suggests how test criteria can fit into practice. Through our consulting, we have helped software companies modify their test processes to incorporate this model.

A flaw with the first edition was that it did not mention JUnit or other test automation frameworks. In 2016, JUnit is used very widely in industry, and is commonly used in CS1 and CS2 classes for automated grading. Chapter 03 rectifies this oversight by discussing test automation in general, the concepts that make test automation difficult, and explicitly teaches JUnit. Although the book is largely technology-neutral, having a consistent test framework throughout the book helps with examples and exercises. In our classes, we usually require tests to be automated and often ask students to try other "*-Unit" frameworks such as HttpUnit as homework exercises. We believe that test organizations cannot be ready to apply test criteria successfully before they have automated their tests.

Chapter 04 goes to the natural next step of test-driven development. Although TDD is a different take on testing than the rest of the book, it's an exciting topic for test educators and researchers precisely because it puts testing front and center—the tests become the requirements. Finally, chapter 05 introduces the concept of test criteria in an abstract way. The jelly bean example (which our students love, especially when we share), is still there, as are concepts such as subsumption.

Part 2, which is the heart of the book, has changed the least for the second edition. In 2014, Jeff asked Paul a very simple question: "Why are the four chapters in part 2 in that order?" The answer was stunned silence, as we realized that we had never asked which order they should appear in. It turns out that the RIPR model,

[1] To help reduce confusion, we developed the convention of using two digits for second edition chapters. Thus, in this preface, chapter 01 implies the second edition, whereas chapter 1 implies the first.

which is certainly central to software testing, dictates a logical order. Specifically, input space partitioning does not require reachability, infection, or propagation. Graph coverage criteria require execution to "get to" some location in the software artifact under test, that is, *reachability*, but not infection or propagation. Logic coverage criteria require that a predicate not only be reached, but be exercised in a particular way to affect the result of the predicate. That is, the predicate must be *infected*. Finally, syntax coverage not only requires that a location be reached, and that the program state of the "mutated" version be different from the original version, but that difference must be visible after execution finishes. That is, it must *propagate*. The second edition orders these four concepts based on the RIPR model, where each chapter now has successively stronger requirements. From a practical perspective, all we did was move the previous chapter 5 (now chapter 06) in front of the graph chapter (now chapter 07).

Another major structural change is that the second edition does **not** include chapters 7 through 9 from the first edition. The first edition material has become dated. Because it is used less than other material in the book, we decided not to delay this new edition of the book while we tried to find time to write this material. We plan to include better versions of these chapters in a third edition.

We also made hundreds of changes at a more detailed level. Recent research has found that in addition to an incorrect value propagating to the output, testing only succeeds if our automated test oracle looks at the right part of the software output. That is, the test oracle must *reveal* the failure. Thus, the old RIP model is now the RIPR model. Several places in the book have discussions that go beyond or into more depth than is strictly needed. The second edition now includes "meta discussions," which are ancillary discussions that can be interesting or insightful to some students, but unnecessarily complicated for others.

The new chapter 06 now has a fully worked out example of deriving an input domain model from a widely used Java library interface (in section 06.4). Our students have found this helps them understand how to use the input space partitioning techniques. The first edition included a section on "Representing graphs algebraically." Although one of us found this material to be fun, we both found it hard to motivate and unlikely to be used in practice. It also has some subtle technical flaws. Thus, we removed this section from the second edition. The new chapter 08 (logic) has a significant structural modification. The DNF criteria (formerly in section 3.6) properly belong at the front of the chapter. Chapter 08 now starts with semantic logic criteria (ACC and ICC) in 08.1, then proceeds to syntactic logic criteria (DNF) in 08.2. The syntactic logic criteria have also changed. One was dropped (UTPC), and CUTPNFP has been joined by MUTP and MNFP. Together, these three criteria comprise MUMCUT.

Throughout the book (especially part 2), we have improved the examples, simplified definitions, and included more exercises. When the first edition was published we had a partial solution manual, which somehow took five years to complete. We are proud to say that we learned from that mistake: we made (and stuck by!) a rule that we couldn't add an exercise without also adding a solution. The reader might think of this rule as testing for exercises. We are glad to say that the second edition book website **debuts** with a complete solution manual.

The second edition also has many dozens of corrections (starting with the errata list from the first edition book website), but including many more that we found while preparing the second edition. The second edition also has a better index. We put together the index for the first edition in about a day, and it showed. This time we have been indexing as we write, and committed time near the end of the process to specifically focus on the index. For future book writers, indexing is hard work and not easy to turn over to a non-author!

What is still the same in the second edition?

The things that have stayed the same are those that were successful in the first edition. The overall observation that test criteria are based on only four types of structures is still the key organizing principle of the second edition. The second edition is also written from an engineering viewpoint, assuming that users of the book are engineers who want to produce the highest quality software with the lowest possible cost. The concepts are well grounded in theory, yet presented in a practical manner. That is, the book tries to make theory meet practice; the theory is sound according to the research literature, but we also show how the theory applies in practice.

The book is also written as a text book, with clear explanations, simple but illustrative examples, and lots of exercises suitable for in-class or out-of-class work. Each chapter ends with bibliographic notes so that beginning research students can proceed to learning the deeper ideas involved in software testing. The book website (https://cs.gmu.edu/~offutt/softwaretest/) is rich in materials with solution manuals, listings of all example programs in the text, high quality PowerPoint slides, and software to help students with graph coverage, logic coverage, and mutation analysis. Some explanatory videos are also available and we hope more will follow. The solution manual comes in two flavors. The student solution manual, with solutions to about half the exercises, is available to everyone. The instructor solution manual has solutions to all exercises and is only available to those who convince the authors that they are using a book to teach a course.

Using the book in the classroom

The book chapters are built in a modular, component-based manner. Most chapters are independent, and although they are presented in the order that we use them, inter-chapter dependencies are few and they could be used in almost any order. Our primary target courses at our university are a fourth-year course (SWE 437) and a first-year graduate course (SWE 637). Interested readers can search on those courses ("mason swe 437" or "mason swe 637") to see our schedules and how we use the book. Both courses are required; SWE 437 is required in the software engineering concentration in our Applied Computer Science major, and SWE 637 is required in our MS program in software engineering[2]. Chapters 01 and 03 can be used in an early course such as CS2 in two ways. First, to sensitize early students to

[2] Our MS program is practical in nature, not research-oriented. The majority of students are part-time students with five to ten years of experience in the software industry. SWE 637 begat this book when we realized Beizer's classic text [Beizer, 1990] was out of print.

the importance of software quality, and second to get them started with test automation (we use JUnit at Mason). A second-year course in testing could cover all of part 1, chapter 06 from part 2, and all or part of part 3. The other chapters in part 2 are probably more than what such students need, but input space partitioning is a very accessible introduction to structured, high-end testing. A common course in north American computer science programs is a third-year course on general software engineering. Part 1 would be very appropriate for such a course. In 2016 we are introducing an advanced graduate course on software testing, which will span cutting-edge knowledge and current research. This course will use some of part 3, the material that we are currently developing for part 4, and selected research papers.

Teaching software testing

Both authors have become students of teaching over the past decade. In the early 2000s, we ran fairly traditional classrooms. We lectured for most of the available class time, kept organized with extensive PowerPoint slides, required homework assignments to be completed individually, and gave challenging, high-pressure exams. The PowerPoint slides and exercises in the first edition were designed for this model.

However, our teaching has evolved. We replaced our midterm exam with weekly quizzes, given in the first 15 minutes of class. This distributed a large component of the grade through the semester, relieved much of the stress of midterms, encouraged the students to keep up on a weekly basis instead of cramming right before the exam, and helped us identify students who were succeeding or struggling early in the term.

After learning about the "flipped classroom" model, we experimented with recorded lectures, viewed online, followed by doing the "homework" assignments in class with us available for immediate help. We found this particularly helpful with the more mathematically sophisticated material such as logic coverage, and especially beneficial to struggling students. As the educational research evidence against the benefits of lectures has mounted, we have been moving away from the "sage on a stage" model of talking for two hours straight. We now often talk for 10 to 20 minutes, then give in-class exercises[3] where the students immediately try to solve problems or answer questions. We confess that this is difficult for us, because we love to talk! Or, instead of showing an example during our lecture, we introduce the example, let the students work the next step in small groups, and then share the results. Sometimes our solutions are better, sometimes theirs are better, and sometimes solutions differ in interesting ways that spur discussion.

There is no doubt that this approach to teaching takes time and cannot acccomodate all of the PowerPoint slides we have developed. We believe that although we *cover* less material, we *uncover* more, a perception consistent with how our students perform on our final exams.

Most of the in-class exercises are done in small groups. We also encourage students to work out-of-class assignments collaboratively. Not only does evidence show

[3] These in-class exercises are not yet a formal part of the book website. But we often draw them from regular exercises in the text. Interested readers can extract recent versions from our course web pages with a search engine.

that students learn more when they work collaboratively ("peer-learning"), they enjoy it more, and it matches the industrial reality. Very few software engineers work alone.

Of course, you can use this book in your class as you see fit. We offer these insights simply as examples for things that work for us. We summarize our current philosophy of teaching simply: *Less talking, more teaching*.

Acknowledgments

It is our pleasure to acknowledge by name the many contributers to this text. We begin with students at George Mason who provided excellent feedback on early draft chapters from the second edition: Firass Almiski, Natalia Anpilova, Khalid Bargqdle, Mathew Fadoul, Mark Feghali, Angelica Garcia, Mahmoud Hammad, Husam Hilal, Carolyn Koerner, Han-Tsung Liu, Charon Lu, Brian Mitchell, Tuan Nguyen, Bill Shelton, Dzung Tran, Dzung Tray, Sam Tryon, Jing Wu, Zhonghua Xi, and Chris Yeung.

We are particularly grateful to colleagues who used draft chapters of the second edition. These early adopters provided valuable feedback that was extremely helpful in making the final document *classroom-ready*. Thanks to: Moataz Ahmed, King Fahd University of Petroleum & Minerals; Jeff Carver, University of Alabama; Richard Carver, George Mason University; Jens Hannemann, Kentucky State University; Jane Hayes, University of Kentucky; Kathleen Keogh, Federation University Australia; Robyn Lutz, Iowa State University; Upsorn Praphamontripong, George Mason University; Alper Sen, Bogazici University; Marjan Sirjani, Reykjavik University; Mary Lou Soffa, University of Virginia; Katie Stolee, North Carolina State University; and Xiaohong Wang, Salisbury University.

Several colleagues provided exceptional feedback from the first edition: Andy Brooks, Mark Hampton, Jian Zhou, Jeff (Yu) Lei, and six anonymous reviewers contacted by our publisher. The following individuals corrected, and in some cases developed, exercise solutions: Sana'a Alshdefat, Yasmine Badr, Jim Bowring, Steven Dastvan, Justin Donnelly, Martin Gebert, JingJing Gu, Jane Hayes, Rama Kesavan, Ignacio Martín, Maricel Medina-Mora, Xin Meng, Beth Paredes, Matt Rutherford, Farida Sabry, Aya Salah, Hooman Safaee, Preetham Vemasani, and Greg Williams. The following George Mason students found, and often corrected, errors in the first edition: Arif Al-Mashhadani, Yousuf Ashparie, Parag Bhagwat, Firdu Bati, Andrew Hollingsworth, Gary Kaminski, Rama Kesavan, Steve Kinder, John Krause, Jae Hyuk Kwak, Nan Li, Mohita Mathur, Maricel Medina Mora, Upsorn Praphamontripong, Rowland Pitts, Mark Pumphrey, Mark Shapiro, Bill Shelton, David Sracic, Jose Torres, Preetham Vemasani, Shuang Wang, Lance Witkowski, Leonard S. Woody III, and Yanyan Zhu. The following individuals from elsewhere found, and often corrected, errors in the first edition: Sana'a Alshdefat, Alexandre Bartel, Don Braffitt, Andrew Brooks, Josh Dehlinger, Gordon Fraser, Rob Fredericks, Weiyi Li, Hassan Mirian, Alan Moraes, Miika Nurminen, Thomas Reinbacher, Hooman Rafat Safaee, Hossein Saiedian, Aya Salah, and Markku Sakkinen. Lian Yu of Peking University translated the the first edition into Mandarin Chinese.

We also want to acknowledge those who implicitly contributed to the second edition by explicitly contributing to the first edition: Aynur Abdurazik, Muhammad Abdulla, Roger Alexander, Lionel Briand, Renee Bryce, George P. Burdell, Guillermo Calderon-Meza, Jyothi Chinman, Yuquin Ding, Blaine Donley, Patrick Emery, Brian Geary, Hassan Gomaa, Mats Grindal, Becky Hartley, Jane Hayes, Mark Hinkle, Justin Hollingsworth, Hong Huang, Gary Kaminski, John King, Yuelan Li, Ling Liu, Xiaojuan Liu, Chris Magrin, Darko Marinov, Robert Nilsson, Andrew J. Offutt, Buzz Pioso, Jyothi Reddy, Arthur Reyes, Raimi Rufai, Bo Sanden, Jeremy Schneider, Bill Shelton, Michael Shin, Frank Shukis, Greg Williams, Quansheng Xiao, Tao Xie, Wuzhi Xu, and Linzhen Xue.

While developing the second edition, our graduate teaching assistants at George Mason gave us fantastic feedback on early drafts of chapters: Lin Deng, Jingjing Gu, Nan Li, and Upsorn Praphamontripong. In particular, Nan Li and Lin Deng were instrumental in completing, evolving, and maintaining the software coverage tools available on the book website.

We are grateful to our editor, Lauren Cowles, for providing unwavering support and enforcing the occasional deadline to move the project along, as well as Heather Bergmann, our former editor, for her strong support on this long-running project.

Finally, of course none of this is possible without the support of our families. Thanks to Becky, Jian, Steffi, Matt, Joyce, and Andrew for helping us stay balanced.

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.