

Unit Testing: Principles, Practices, and Patterns

VLADIMIR KHORIKOV



MANNING
SHELTER ISLAND

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact


Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Email: orders@manning.com

©2020 by Manning Publications Co. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in the book, and Manning Publications was aware of a trademark claim, the designations have been printed in initial caps or all caps.

☺ Recognizing the importance of preserving what has been written, it is Manning's policy to have the books we publish printed on acid-free paper, and we exert our best efforts to that end. Recognizing also our responsibility to conserve the resources of our planet, Manning books are printed on paper that is at least 15 percent recycled and processed without the use of elemental chlorine.

 Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964

Acquisitions editor: Mike Stephens
Development editor: Marina Michaels
Technical development editor: Sam Zaydel
Review editor: Aleksandar Dragosavljević
Production editor: Anthony Calcara
Copy editor: Tiffany Taylor
ESL copyeditor: Frances Buran
Proofreader: Keri Hales
Technical proofreader: Alessandro Campeis
Typesetter: Dennis Dalinnik
Cover designer: Marija Tudor

ISBN: 9781617296277
Printed in the United States of America

brief contents

| | | |
|---------------|--------------------------------------------|------------|
| PART 1 | THE BIGGER PICTURE..... | 1 |
| 1 | ■ The goal of unit testing | 3 |
| 2 | ■ What is a unit test? | 20 |
| 3 | ■ The anatomy of a unit test | 41 |
| PART 2 | MAKING YOUR TESTS WORK FOR YOU..... | 65 |
| 4 | ■ The four pillars of a good unit test | 67 |
| 5 | ■ Mocks and test fragility | 92 |
| 6 | ■ Styles of unit testing | 119 |
| 7 | ■ Refactoring toward valuable unit tests | 151 |
| PART 3 | INTEGRATION TESTING..... | 183 |
| 8 | ■ Why integration testing? | 185 |
| 9 | ■ Mocking best practices | 216 |
| 10 | ■ Testing the database | 229 |
| PART 4 | UNIT TESTING ANTI-PATTERNS..... | 257 |
| 11 | ■ Unit testing anti-patterns | 259 |

contents

| | |
|-------------------------------------|------------|
| <i>preface</i> | <i>xiv</i> |
| <i>acknowledgments</i> | <i>xv</i> |
| <i>about this book</i> | <i>xvi</i> |
| <i>about the author</i> | <i>xix</i> |
| <i>about the cover illustration</i> | <i>xx</i> |

PART 1 THE BIGGER PICTURE1

| | | |
|----------|-------------------------------------------------------------------|----------|
| 1 | <i>The goal of unit testing</i> | 3 |
| 1.1 | The current state of unit testing | 4 |
| 1.2 | The goal of unit testing | 5 |
| | <i>What makes a good or bad test?</i> | 7 |
| 1.3 | Using coverage metrics to measure test suite quality | 8 |
| | <i>Understanding the code coverage metric</i> | 9 |
| | <i>Understanding the branch coverage metric</i> | 10 |
| | <i>Problems with coverage metrics</i> | 12 |
| | <i>Aiming at a particular coverage number</i> | 15 |
| 1.4 | What makes a successful test suite? | 15 |
| | <i>It's integrated into the development cycle</i> | 16 |
| | <i>It targets only the most important parts of your code base</i> | 16 |
| | <i>It provides maximum value with minimum maintenance costs</i> | 17 |
| 1.5 | What you will learn in this book | 17 |

2 What is a unit test? 20

- 2.1 The definition of “unit test” 21
 - The isolation issue: The London take* 21 ■ *The isolation issue: The classical take* 27
- 2.2 The classical and London schools of unit testing 30
 - How the classical and London schools handle dependencies* 30
- 2.3 Contrasting the classical and London schools of unit testing 34
 - Unit testing one class at a time* 34 ■ *Unit testing a large graph of interconnected classes* 35 ■ *Revealing the precise bug location* 36
 - Other differences between the classical and London schools* 36
- 2.4 Integration tests in the two schools 37
 - End-to-end tests are a subset of integration tests* 38

3 The anatomy of a unit test 41

- 3.1 How to structure a unit test 42
 - Using the AAA pattern* 42 ■ *Avoid multiple arrange, act, and assert sections* 43 ■ *Avoid if statements in tests* 44
 - How large should each section be?* 45 ■ *How many assertions should the assert section hold?* 47 ■ *What about the teardown phase?* 47 ■ *Differentiating the system under test* 47
 - Dropping the arrange, act, and assert comments from tests* 48
- 3.2 Exploring the xUnit testing framework 49
- 3.3 Reusing test fixtures between tests 50
 - High coupling between tests is an anti-pattern* 52 ■ *The use of constructors in tests diminishes test readability* 52 ■ *A better way to reuse test fixtures* 52
- 3.4 Naming a unit test 54
 - Unit test naming guidelines* 56 ■ *Example: Renaming a test toward the guidelines* 56
- 3.5 Refactoring to parameterized tests 58
 - Generating data for parameterized tests* 60
- 3.6 Using an assertion library to further improve test readability 62

PART 2 MAKING YOUR TESTS WORK FOR YOU65

4 *The four pillars of a good unit test* 67

- 4.1 Diving into the four pillars of a good unit test 68
 - The first pillar: Protection against regressions* 68 ■ *The second pillar: Resistance to refactoring* 69 ■ *What causes false positives?* 71 ■ *Aim at the end result instead of implementation details* 74
- 4.2 The intrinsic connection between the first two attributes 76
 - Maximizing test accuracy* 76 ■ *The importance of false positives and false negatives: The dynamics* 78
- 4.3 The third and fourth pillars: Fast feedback and maintainability 79
- 4.4 In search of an ideal test 80
 - Is it possible to create an ideal test?* 81 ■ *Extreme case #1: End-to-end tests* 81 ■ *Extreme case #2: Trivial tests* 82
 - Extreme case #3: Brittle tests* 83 ■ *In search of an ideal test: The results* 84
- 4.5 Exploring well-known test automation concepts 87
 - Breaking down the Test Pyramid* 87 ■ *Choosing between black-box and white-box testing* 89

5 *Mocks and test fragility* 92

- 5.1 Differentiating mocks from stubs 93
 - The types of test doubles* 93 ■ *Mock (the tool) vs. mock (the test double)* 94 ■ *Don't assert interactions with stubs* 96
 - Using mocks and stubs together* 97 ■ *How mocks and stubs relate to commands and queries* 97
- 5.2 Observable behavior vs. implementation details 99
 - Observable behavior is not the same as a public API* 99 ■ *Leaking implementation details: An example with an operation* 100
 - Well-designed API and encapsulation* 103 ■ *Leaking implementation details: An example with state* 104
- 5.3 The relationship between mocks and test fragility 106
 - Defining hexagonal architecture* 106 ■ *Intra-system vs. inter-system communications* 110 ■ *Intra-system vs. inter-system communications: An example* 111

- 5.4 The classical vs. London schools of unit testing, revisited 114
 - Not all out-of-process dependencies should be mocked out* 115
 - Using mocks to verify behavior* 116

6 *Styles of unit testing* 119

- 6.1 The three styles of unit testing 120
 - Defining the output-based style* 120 ■ *Defining the state-based style* 121 ■ *Defining the communication-based style* 122
- 6.2 Comparing the three styles of unit testing 123
 - Comparing the styles using the metrics of protection against regressions and feedback speed* 124 ■ *Comparing the styles using the metric of resistance to refactoring* 124 ■ *Comparing the styles using the metric of maintainability* 125 ■ *Comparing the styles: The results* 127
- 6.3 Understanding functional architecture 128
 - What is functional programming?* 128 ■ *What is functional architecture?* 132 ■ *Comparing functional and hexagonal architectures* 133
- 6.4 Transitioning to functional architecture and output-based testing 135
 - Introducing an audit system* 135 ■ *Using mocks to decouple tests from the filesystem* 137 ■ *Refactoring toward functional architecture* 140 ■ *Looking forward to further developments* 146
- 6.5 Understanding the drawbacks of functional architecture 146
 - Applicability of functional architecture* 147 ■ *Performance drawbacks* 148 ■ *Increase in the code base size* 149

7 *Refactoring toward valuable unit tests* 151

- 7.1 Identifying the code to refactor 152
 - The four types of code* 152 ■ *Using the Humble Object pattern to split overcomplicated code* 155
- 7.2 Refactoring toward valuable unit tests 158
 - Introducing a customer management system* 158 ■ *Take 1: Making implicit dependencies explicit* 160 ■ *Take 2: Introducing an application services layer* 160 ■ *Take 3: Removing complexity from the application service* 163 ■ *Take 4: Introducing a new Company class* 164

- 7.3 Analysis of optimal unit test coverage 167
 - Testing the domain layer and utility code* 167 ■ *Testing the code from the other three quadrants* 168 ■ *Should you test preconditions?* 169
- 7.4 Handling conditional logic in controllers 169
 - Using the CanExecute/Execute pattern* 172 ■ *Using domain events to track changes in the domain model* 175
- 7.5 Conclusion 178

PART 3 INTEGRATION TESTING.....183

8 Why integration testing? 185

- 8.1 What is an integration test? 186
 - The role of integration tests* 186 ■ *The Test Pyramid revisited* 187 ■ *Integration testing vs. failing fast* 188
- 8.2 Which out-of-process dependencies to test directly 190
 - The two types of out-of-process dependencies* 190 ■ *Working with both managed and unmanaged dependencies* 191 ■ *What if you can't use a real database in integration tests?* 192
- 8.3 Integration testing: An example 193
 - What scenarios to test?* 194 ■ *Categorizing the database and the message bus* 195 ■ *What about end-to-end testing?* 195
 - Integration testing: The first try* 196
- 8.4 Using interfaces to abstract dependencies 197
 - Interfaces and loose coupling* 198 ■ *Why use interfaces for out-of-process dependencies?* 199 ■ *Using interfaces for in-process dependencies* 199
- 8.5 Integration testing best practices 200
 - Making domain model boundaries explicit* 200 ■ *Reducing the number of layers* 200 ■ *Eliminating circular dependencies* 202
 - Using multiple act sections in a test* 204
- 8.6 How to test logging functionality 205
 - Should you test logging?* 205 ■ *How should you test logging?* 207 ■ *How much logging is enough?* 212
 - How do you pass around logger instances?* 212
- 8.7 Conclusion 213

9 *Mocking best practices* 216

9.1 Maximizing mocks' value 217

Verifying interactions at the system edges 219 ■ *Replacing mocks with spies* 222 ■ *What about IDomainLogger?* 224

9.2 Mocking best practices 225

Mocks are for integration tests only 225 ■ *Not just one mock per test* 225 ■ *Verifying the number of calls* 226 ■ *Only mock types that you own* 227

10 *Testing the database* 229

10.1 Prerequisites for testing the database 230

Keeping the database in the source control system 230 ■ *Reference data is part of the database schema* 231 ■ *Separate instance for every developer* 232 ■ *State-based vs. migration-based database delivery* 232

10.2 Database transaction management 234

Managing database transactions in production code 235 ■ *Managing database transactions in integration tests* 242

10.3 Test data life cycle 243

Parallel vs. sequential test execution 243 ■ *Clearing data between test runs* 244 ■ *Avoid in-memory databases* 246

10.4 Reusing code in test sections 246

Reusing code in arrange sections 246 ■ *Reusing code in act sections* 249 ■ *Reusing code in assert sections* 250
Does the test create too many database transactions? 251

10.5 Common database testing questions 252

Should you test reads? 252 ■ *Should you test repositories?* 253

10.6 Conclusion 254

PART 3 UNIT TESTING ANTI-PATTERNS.....257

11 *Unit testing anti-patterns* 259

11.1 Unit testing private methods 260

Private methods and test fragility 260 ■ *Private methods and insufficient coverage* 260 ■ *When testing private methods is acceptable* 261

11.2 Exposing private state 263

11.3 Leaking domain knowledge to tests 264

| | | |
|------|---------------------------------------|-----|
| 11.4 | Code pollution | 266 |
| 11.5 | Mocking concrete classes | 268 |
| 11.6 | Working with time | 271 |
| | <i>Time as an ambient context</i> | 271 |
| | <i>Time as an explicit dependency</i> | 272 |
| 11.7 | Conclusion | 273 |
| | <i>index</i> | 275 |

preface

I remember my first project where I tried out unit testing. It went relatively well; but after it was finished, I looked at the tests and thought that a lot of them were a pure waste of time. Most of my unit tests spent a great deal of time setting up expectations and wiring up a complicated web of dependencies—all that, just to check that the three lines of code in my controller were correct. I couldn't pinpoint what exactly was wrong with the tests, but my sense of proportion sent me unambiguous signals that something was off.

Luckily, I didn't abandon unit testing and continued applying it in subsequent projects. However, disagreement with common (at that time) unit testing practices has been growing in me ever since. Throughout the years, I've written a lot about unit testing. In those writings, I finally managed to crystallize what exactly was wrong with my first tests and generalized this knowledge to broader areas of unit testing. This book is a culmination of all my research, trial, and error during that period—compiled, refined, and distilled.

I come from a mathematical background and strongly believe that guidelines in programming, like theorems in math, should be derived from first principles. I've tried to structure this book in a similar way: start with a blank slate by not jumping to conclusions or throwing around unsubstantiated claims, and gradually build my case from the ground up. Interestingly enough, once you establish such first principles, guidelines and best practices often flow naturally as mere implications.

I believe that unit testing is becoming a de facto requirement for software projects, and this book will give you everything you need to create valuable, highly maintainable tests.

acknowledgments

This book was a lot of work. Even though I was prepared mentally, it was still much more work than I could ever have imagined.

A big “thank you” to Sam Zaydel, Alessandro Campeis, Frances Buran, Tiffany Taylor, and especially Marina Michaels, whose invaluable feedback helped shape the book and made me a better writer along the way. Thanks also to everyone else at Manning who worked on this book in production and behind the scenes.

I’d also like to thank the reviewers who took the time to read my manuscript at various stages during its development and who provided valuable feedback: Aaron Barton, Alessandro Campeis, Conor Redmond, Dror Helper, Greg Wright, Hemant Koneru, Jeremy Lange, Jorge Ezequiel Bo, Jort Rodenburg, Mark Nenadov, Marko Umek, Markus Matzker, Srihari Sridharan, Stephen John Warnett, Sumant Tambe, Tim van Deurzen, and Vladimir Kuptsov.

Above all, I would like to thank my wife Nina, who supported me during the whole process.