contents

foreword to the second edition xv
foreword to the first edition xvii
preface xix
acknowledgments xxi
about this book xxii
about the cover illustration xxvi

PART 1 GETTING STARTED 1

1 *The basics of unit testing 3*

1.1 Defining unit testing, step by step 4

The importance of writing good unit tests 5 ▪ We've all written unit tests (sort of) 5

1.2 Properties of a good unit test 6

1.3 Integration tests 7

Drawbacks of nonautomated integration tests compared to automated unit tests 9

1.4 What makes unit tests good 11

1.5 A simple unit test example 11

1.6 Test-driven development 14

Download from Wow! eBook <www.wowebook.com>

1.7 The three core skills of successful TDD 17
1.8 Summary 17

2 *A first unit test* 19

2.1 Frameworks for unit testing 20

What unit testing frameworks offer 20
The xUnit frameworks 22

2.2 Introducing the LogAn project 22
2.3 First steps with NUnit 23

Installing NUnit 23 ▪ Loading up the solution 25
Using the NUnit attributes in your code 27

2.4 Writing your first test 27

The Assert class 28 ▪ Running your first test with NUnit 29
Adding some positive tests 30 ▪ From red to green: passing the tests 31 ▪ Test code styling 31

2.5 Refactoring to parameterized tests 31
2.6 More NUnit attributes 33

Setup and teardown 34 ▪ Checking for expected exceptions 36
Ignoring tests 39 ▪ NUnit's fluent syntax 39
Setting test categories 40

2.7 Testing results that are system state changes instead of return values 40
2.8 Summary 44

PART 2 CORE TECHNIQUES 47

3 *Using stubs to break dependencies* 49

3.1 Introducing stubs 50
3.2 Identifying a filesystem dependency in LogAn 50
3.3 Determining how to easily test LogAnalyzer 51
3.4 Refactoring your design to be more testable 53

Extract an interface to allow replacing underlying implementation 55 ▪ Dependency injection: inject a fake implementation into a unit under test 57 ▪ Inject a fake at the constructor level (constructor injection) 57
Simulating exceptions from fakes 61 ▪ Injecting a fake as a property get or set 61 ▪ Injecting a fake just before a method call 63

Download from Wow! eBook <www.wowebook.com>

3.5 Variations on refactoring techniques 69

Using Extract and Override to create fake results 70

3.6 Overcoming the encapsulation problem 71

Using internal and [InternalsVisibleTo] 72 ▪ Using the [Conditional] attribute 72 ▪ Using #if and #endif with conditional compilation 73

3.7 Summary 73

4 *Interaction testing using mock objects 75*

4.1 Value-based vs. state-based vs. interaction testing 76

4.2 The difference between mocks and stubs 78

4.3 A simple handwritten mock example 79

4.4 Using a mock and a stub together 81

4.5 One mock per test 85

4.6 Fake chains: stubs that produce mocks or other stubs 86

4.7 The problems with handwritten mocks and stubs 87

4.8 Summary 88

5 *Isolation (mocking) frameworks 90*

5.1 Why use isolation frameworks? 91

5.2 Dynamically creating a fake object 93

Introducing NSubstitute into your tests 93 ▪ Replacing a handwritten fake object with a dynamic one 94

5.3 Simulating fake values 96

A mock, a stub, and a priest walk into a test 97

5.4 Testing for event-related activities 102

Testing an event listener 102 ▪ Testing whether an event was triggered 103

5.5 Current isolation frameworks for .NET 104

5.6 Advantages and traps of isolation frameworks 106

Traps to avoid when using isolation frameworks 106
Unreadable test code 106 ▪ Verifying the wrong things 106
Having more than one mock per test 107
Overspecifying the tests 107

5.7 Summary 107

Download from Wow! eBook <www.wowebook.com>

6 ***Digging deeper into isolation frameworks 109***

6.1 Constrained and unconstrained frameworks 110

Constrained frameworks 110 ▪ Unconstrained frameworks 110 How profiler-based unconstrained frameworks work 112

6.2 Values of good isolation frameworks 114

6.3 Features supporting future-proofing and usability 114

Recursive fakes 115 ▪ Ignored arguments by default 115 Wide faking 116 ▪ Nonstrict behavior of fakes 116 Nonstrict mocks 117

6.4 Isolation framework design antipatterns 117

Concept confusion 118 ▪ Record and replay 119 Sticky behavior 120 ▪ Complex syntax 120

6.5 Summary 121

PART 3 THE TEST CODE 123

7 ***Test hierarchies and organization 125***

7.1 Automated builds running automated tests 126

Anatomy of a build script 127 ▪ Triggering builds and integration 128

7.2 Mapping out tests based on speed and type 130

The human factor when separating unit from integration tests 130 The safe green zone 131

7.3 Ensuring tests are part of source control 131

7.4 Mapping test classes to code under test 132

Mapping tests to projects 132 ▪ Mapping tests to classes 132 Mapping tests to specific unit of work method entry points 133

7.5 Cross-cutting concerns injection 134

7.6 Building a test API for your application 136

Using test class inheritance patterns 136 ▪ Creating test utility classes and methods 148 ▪ Making your API known to developers 149

7.7 Summary 149

8 ***The pillars of good unit tests 151***

8.1 Writing trustworthy tests 152

Deciding when to remove or change tests 152 ▪ Avoiding logic in tests 156 ▪ Testing only one concern 158 ▪ Separate unit from integration tests 159 ▪ Assuring code review with code coverage 159

Download from Wow! eBook <www.wowebook.com>

8.2 Writing maintainable tests 161

Testing private or protected methods 161
Removing duplication 163 ▪ Using setup methods in a maintainable manner 166 ▪ Enforcing test isolation 169
Avoiding multiple asserts on different concerns 174
Comparing objects 176 ▪ Avoiding overspecification 178

8.3 Writing readable tests 180

Naming unit tests 181 ▪ Naming variables 181
Asserting yourself with meaning 182 ▪ Separating asserts from actions 183 ▪ Setting up and tearing down 184

8.4 Summary 184

PART 4 DESIGN AND PROCESS 187

9 ***Integrating unit testing into the organization 189***

9.1 Steps to becoming an agent of change 190

Be prepared for the tough questions 190 ▪ Convince insiders: champions and blockers 190 ▪ Identify possible entry points 191

9.2 Ways to succeed 193

Guerrilla implementation (bottom up) 193
Convincing management (top down) 193 ▪ Getting an outside champion 194 ▪ Making progress visible 194 ▪ Aiming for specific goals 196 ▪ Realizing that there will be hurdles 197

9.3 Ways to fail 197

Lack of a driving force 197 ▪ Lack of political support 198
Bad implementations and first impressions 198
Lack of team support 198

9.4 Influence factors 199

9.5 Tough questions and answers 200

How much time will unit testing add to the current process? 200
Will my QA job be at risk because of unit testing? 202
How do we know unit tests are actually working? 202
Is there proof that unit testing helps? 203 ▪ Why is the QA department still finding bugs? 203 ▪ We have lots of code without tests: where do we start? 204 ▪ We work in several languages: is unit testing feasible? 204 ▪ What if we develop a combination of software and hardware? 204 ▪ How can we know we don't have bugs in our tests? 205 ▪ My debugger shows that my code works; why do I need tests? 205 ▪ Must we do TDD-style coding? 205

9.6 Summary 205

Download from Wow! eBook <www.wowebook.com>

10 ***Working with legacy code* 207**

10.1 Where do you start adding tests? 208

10.2 Choosing a selection strategy 209

Pros and cons of the easy-first strategy 210 ▪ *Pros and cons of the hard-first strategy 210*

10.3 Writing integration tests before refactoring 211

10.4 Important tools for legacy code unit testing 212

Isolate dependencies easily with unconstrained isolation frameworks 212 ▪ *Use JMockit for Java legacy code 213 Use Vise while refactoring your Java code 215* ▪ *Use acceptance tests before you refactor 216* ▪ *Read Michael Feathers's book on legacy code 216* ▪ *Use NDepend to investigate your production code 216* ▪ *Use ReSharper to navigate and refactor production code 217* ▪ *Detect duplicate code (and bugs) with Simian and TeamCity 218*

10.5 Summary 218

11 ***Design and testability* 219**

11.1 Why should I care about testability in my design? 219

11.2 Design goals for testability 220

Make methods virtual by default 221 ▪ *Use interface-based designs 222* ▪ *Make classes nonsealed by default 222 Avoid instantiating concrete classes inside methods with logic 222 Avoid direct calls to static methods 222* ▪ *Avoid constructors and static constructors that do logic 223* ▪ *Separate singleton logic from singleton holders 223*

11.3 Pros and cons of designing for testability 224

Amount of work 225 ▪ *Complexity 225 Exposing sensitive IP 226* ▪ *Sometimes you can't 226*

11.4 Alternatives to designing for testability 226

Design arguments and dynamically typed languages 227

11.5 Example of a hard-to-test design 228

11.6 Summary 232

11.7 Additional resources 232

appendix Tools and frameworks 234

index 253

Download from Wow! eBook <www.wowebook.com>