

the art of

UNIT TESTING

with Examples in .NET



 MANNING

ROY OSHEROVE

Contents

<i>foreword</i>	xv
<i>preface</i>	xvii
<i>acknowledgments</i>	xix
<i>about this book</i>	xx
<i>about the cover illustration</i>	xxiii

PART 1 GETTING STARTED 1

1 The basics of unit testing 3

1.1	<i>Unit testing—the classic definition</i>	4
	The importance of writing “good” unit tests	5
	◦ We’ve all written unit tests (sort of)	5
1.2	<i>Properties of a good unit test</i>	6
1.3	<i>Integration tests</i>	7
	Drawbacks of integration tests compared to automated unit tests	9
1.4	<i>Good unit test—a definition</i>	11
1.5	<i>A simple unit test example</i>	12
1.6	<i>Test-driven development</i>	16
1.7	<i>Summary</i>	19

2 A first unit test 21

2.1	<i>Frameworks for unit testing</i>	22
	What unit-testing frameworks offer	22
	◦ The xUnit frameworks	25

- 2.2 *Introducing the LogAn project* 25
- 2.3 *First steps with NUnit* 26
 - Installing NUnit 26
 - Loading up the solution 26
 - Using the NUnit attributes in your code 29
- 2.4 *Writing our first test* 30
 - The Assert class 31
 - Running our first test with NUnit 32
 - Fixing our code and passing the test 33
 - From red to green 33
- 2.5 *More NUnit attributes* 34
 - Setup and teardown 34
 - Checking for expected exceptions 36
 - Ignoring tests 38
 - Setting test categories 39
- 2.6 *Indirect testing of state* 40
- 2.7 *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* 51
- 3.3 *Determining how to easily test LogAnalyzer* 52
- 3.4 *Refactoring our design to be more testable* 55
 - Extract an interface to allow replacing underlying implementation 55
 - Inject stub implementation into a class under test 58
 - Receive an interface at the constructor level (constructor injection) 58
 - Receive an interface as a property get or set 64
 - Getting a stub just before a method call 66
- 3.5 *Variations on refactoring techniques* 74
 - Using Extract and Override to create stub results 75
- 3.6 *Overcoming the encapsulation problem* 77
 - Using internal and [InternalsVisibleTo] 78
 - Using the [Conditional] attribute 79
 - Using #if and #endif with conditional compilation 80
- 3.7 *Summary* 80

- 4 Interaction testing using mock objects 82**
 - 4.1 *State-based versus interaction testing* 83
 - 4.2 *The difference between mocks and stubs* 84
 - 4.3 *A simple manual mock example* 87
 - 4.4 *Using a mock and a stub together* 89
 - 4.5 *One mock per test* 94
 - 4.6 *Stub chains: stubs that produce mocks or other stubs* 95
 - 4.7 *The problems with handwritten mocks and stubs* 96
 - 4.8 *Summary* 97

- 5 Isolation (mock object) frameworks 99**
 - 5.1 *Why use isolation frameworks?* 100
 - 5.2 *Dynamically creating a fake object* 102
 - Introducing Rhino Mocks into your tests 102 ◦
 - Replacing a handwritten mock object with a dynamic one 103
 - 5.3 *Strict versus nonstrict mock objects* 106
 - Strict mocks 106 ◦ Nonstrict mocks 107
 - 5.4 *Returning values from fake objects* 108
 - 5.5 *Creating smart stubs with an isolation framework* 110
 - Creating a stub in Rhino Mocks 110 ◦ Combining dynamic stubs and mocks 112
 - 5.6 *Parameter constraints for mocks and stubs* 115
 - Checking parameters with string constraints 115 ◦
 - Checking parameter object properties with constraints 118 ◦ Executing callbacks for parameter verification 120
 - 5.7 *Testing for event-related activities* 121
 - Testing that an event has been subscribed to 122 ◦
 - Triggering events from mocks and stubs 123 ◦
 - Testing whether an event was triggered 124
 - 5.8 *Arrange-act-assert syntax for isolation* 126
 - 5.9 *Current isolation frameworks for .NET* 130
 - NUnit.Mocks 130 ◦ NMock 131 ◦ NMock2 131
 - Typemock Isolator 132 ◦ Rhino Mocks 132 ◦ Moq 134

- 5.10 *Advantages of isolation frameworks* 134
- 5.11 *Traps to avoid when using isolation frameworks* 135
 - Unreadable test code 135 ◦ Verifying the wrong things 136 ◦ Having more than one mock per test 136 ◦ Overspecifying the tests 136
- 5.12 *Summary* 137

PART 3 THE TEST CODE 139

6 Test hierarchies and organization 141

- 6.1 *Having automated builds run automated tests* 142
 - Anatomy of an automated build 142 ◦ Triggering builds and continuous integration 144 ◦ Automated build types 144
- 6.2 *Mapping out tests based on speed and type* 145
 - The human factor of separating unit from integration tests 146 ◦ The safe green zone 147
- 6.3 *Ensuring tests are part of source control* 148
- 6.4 *Mapping test classes to code under test* 148
 - Mapping tests to projects 148 ◦ Mapping tests to classes 149 ◦ Mapping tests to specific methods 150
- 6.5 *Building a test API for your application* 150
 - Using test class inheritance patterns 151 ◦ Creating test utility classes and methods 167 ◦ Making your API known to developers 168
- 6.6 *Summary* 169

7 The pillars of good tests 171

- 7.1 *Writing trustworthy tests* 172
 - Deciding when to remove or change tests 172 ◦ Avoiding logic in tests 178 ◦ Testing only one thing 179 ◦ Making tests easy to run 180 ◦ Assuring code coverage 180
- 7.2 *Writing maintainable tests* 181
 - Testing private or protected methods 182 ◦ Removing duplication 184 ◦ Using setup methods in a maintainable manner 188 ◦ Enforcing test isolation 191 ◦ Avoiding multiple asserts 198 ◦

Avoiding testing multiple aspects of the same object 202 ◦ Avoiding overspecification in tests 205

7.3 Writing readable tests 209

Naming unit tests 210 ◦ Naming variables 211 ◦ Asserting yourself with meaning 212 ◦ Separating asserts from actions 214 ◦ Setting up and tearing down 214

7.4 Summary 215

PART 4 DESIGN AND PROCESS 217

8 Integrating unit testing into the organization 219

8.1 Steps to becoming an agent of change 220

Be prepared for the tough questions 220 ◦ Convince insiders: champions and blockers 220 ◦ Identify possible entry points 222

8.2 Ways to succeed 223

Guerrilla implementation (bottom-up) 223 ◦ Convincing management (top-down) 224 ◦ Getting an outside champion 224 ◦ Making progress visible 225 ◦ Aiming for specific goals 227 ◦ Realizing that there will be hurdles 228

8.3 Ways to fail 229

Lack of a driving force 229 ◦ Lack of political support 229 ◦ Bad implementations and first impressions 230 ◦ Lack of team support 230

8.4 Tough questions and answers 231

How much time will this add to the current process? 231 ◦ Will my QA job be at risk because of this? 233 ◦ How do we know this is actually working? 234 ◦ Is there proof that unit testing helps? 234 ◦ Why is the QA department still finding bugs? 235 ◦ We have lots of code without tests: where do we start? 235 ◦ We work in several languages: is unit testing feasible? 236 ◦ What if we develop a combination of software and hardware? 236 ◦ How can we know we don't have bugs in our tests? 236 ◦ I see in my debugger that my code works fine: why do I need tests? 237 ◦ Must we do TDD-style coding? 237

8.5 Summary 238

9	Working with legacy code	239									
9.1	<i>Where do you start adding tests?</i>	240									
9.2	<i>Choosing a selection strategy</i>	242									
	Pros and cons of the easy-first strategy	242 ◦									
	Pros and cons of the hard-first strategy	243									
9.3	<i>Writing integration tests before refactoring</i>	244									
9.4	<i>Important tools for legacy code unit testing</i>	246									
	Isolate dependencies easily with Typemock Isolator	246 ◦ Find testability problems with Depender	248 ◦ Use JMockit for Java legacy code	248 ◦ Use Vise while refactoring your Java code	250 ◦ Use FitNesse for acceptance tests before you refactor	251 ◦ Read Michael Feathers' book on legacy code	253 ◦ Use NDepend to investigate your production code	253 ◦ Use ReSharper to navigate and refactor production code	253 ◦ Detect duplicate code (and bugs) with Simian	254 ◦ Detect threading issues with Typemock Racer	254
9.5	<i>Summary</i>	254									
Appendix A	Design and testability	256									
Appendix B	Extra tools and frameworks	268									
Index		284									