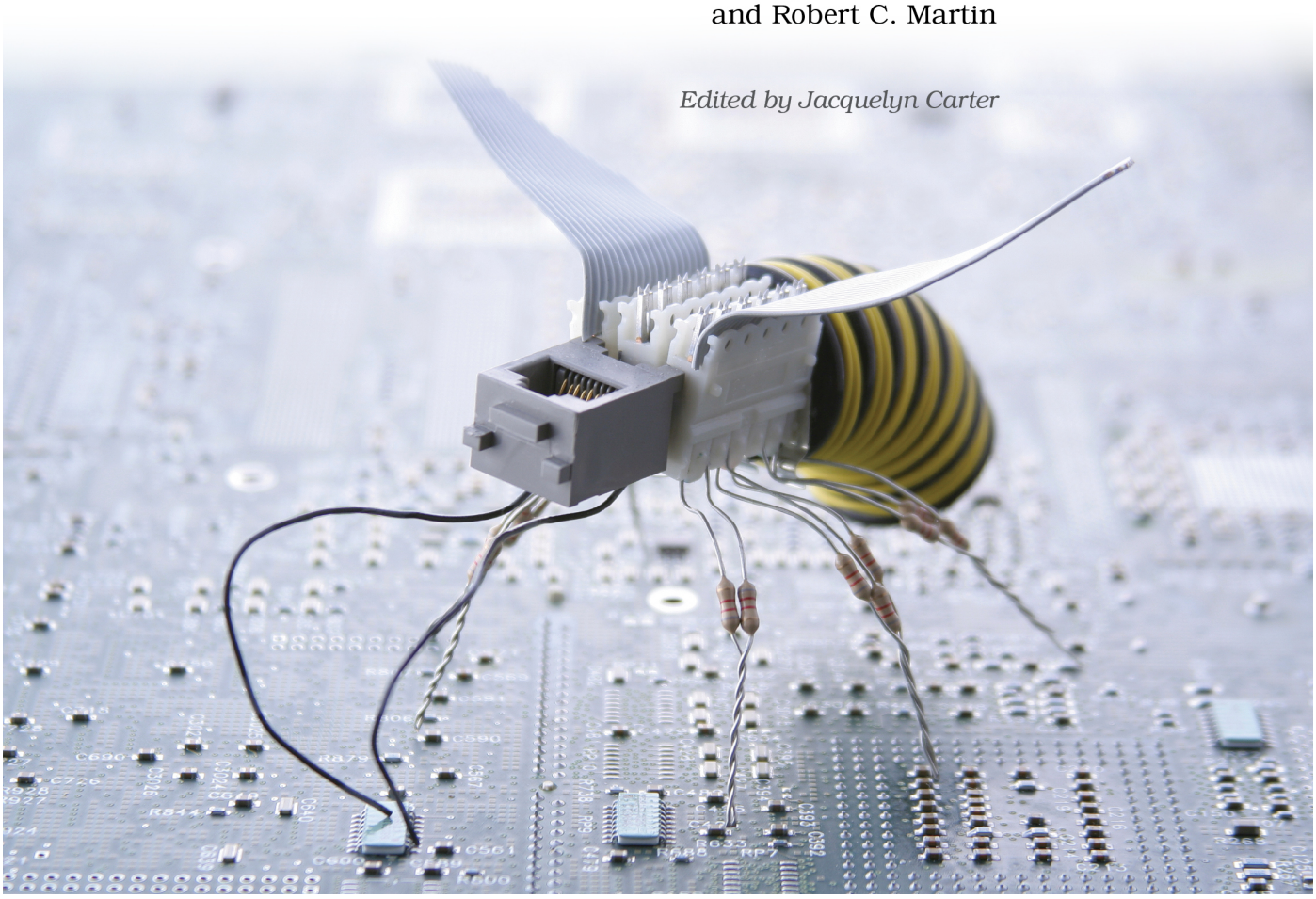
The Pragmatic Programmers

Test-Driven Development for Embedded C

James W. Grenning

Forewords by Jack Ganssle
and Robert C. Martin

Edited by Jacquelyn Carter



What People Are Saying About
Test-Driven Development for Embedded C

In this much-needed book, Agile methods expert James Grenning concisely demonstrates why and how to apply Test-Driven Development in embedded software development. Coming from a purely embedded background, I was myself skeptical about TDD initially. But with this book by my side, I'm ready to plunge right in and certain I can apply TDD even to device drivers and other challenging low-level code.

➤ **Michael Barr**
Author of *Programming Embedded Systems: With C and GNU Development Tools* and *Embedded C Coding Standard*, Netrino, Inc.

"Test-Driven Development cannot work for us! We work in C, and Test-Driven Development requires an object-oriented language such as Java!" I frequently hear statements such as these when coaching teams in TDD in C. I've always pointed them to the work of James Grenning, such as the article "Embedded TDD Cycle." James is a true pioneer in applying Agile development techniques to embedded product development. I was really excited when he told me he was going to write this book because I felt it would definitively help the embedded Agile community forward. It took James more than two years, but the result, this book, was worth waiting for. This is a good and useful book that every embedded developer should read.

➤ **Bas Vodde**
Author of *Scaling Lean and Agile Development* and *Practices for Scaling Lean and Agile Development*, Odd-e, Singapore

I have been preaching and teaching TDD in C for years, and finally there is a book I can recommend to fellow C programmers who want to learn more about modern programming techniques.

➤ **Olve Maudal**
C programmer, Cisco Systems

This book is a practical guide that sheds light on how to apply Agile development practices in the world of embedded software. You'll soon be writing tests that help you pinpoint problems early and avoid hours tearing your hair out trying to figure out what's going on. From my experience writing code for robotics, telemetry, and telecommunications products, I can heartily recommend reading this book; it's a great way to learn how you can apply Test-Driven Development for embedded C.

➤ **Rachel Davies**
Author of *Agile Coaching*, Agile Experience Limited

Test-Driven Development for Embedded C is the first book I would recommend to both C and C++ developers wanting to learn TDD, whether or not their target is an embedded platform. It's just that good.

➤ **C. Keith Ray**
Agile coach/trainer, Industrial Logic, Inc.

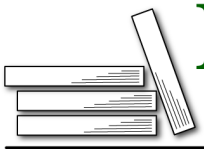
This book is targeting the embedded-programmer-on-the-street and hits its target. It is neither spoon-fed baby talk nor useless theory-spin. In clear and simple prose, James shows working geeks each of the TDD concepts and their C implementations. Any C programmer can benefit from working through this book.

➤ **Michael "GeePaw" Hill**
Senior TDD coach, Anarchy Creek Software

Test-Driven Development for Embedded C

James W. Grenning

The Pragmatic Bookshelf
Dallas, Texas • Raleigh, North Carolina



Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://pragprog.com*.

The team that produced this book includes:

Jacquelyn Carter (editor)
Potomac Indexing, LLC (indexer)
Kim Wimpsett (copyeditor)
David Kelly (typesetter)
Janet Furlow (producer)
Juliet Benda (rights)
Ellie Callahan (support)

Printed in the United States of America.
ISBN-13: 978-1-934356-62-3
Encoded using the finest acid-free high-entropy binary digits.
Book version: P3.0—September 2014

In dedication to my dad, for giving me a good compass, and my loving wife Marilee for helping me not lose it.

Contents

Foreword by Jack Ganssle xiii

Foreword by Robert C. Martin xv

Acknowledgments xix

Preface xxi

1. Test-Driven Development 1
   - 1.1 Why Do We Need TDD? 2
   - 1.2 What Is Test-Driven Development? 4
   - 1.3 Physics of TDD 5
   - 1.4 The TDD Microcycle 6
   - 1.5 TDD Benefits 8
   - 1.6 Benefits for Embedded 10

Part I — Getting Started

2. Test-Driving Tools and Conventions 13
   - 2.1 What Is a Unit Test Harness? 13
   - 2.2 Unity: A C-Only Test Harness 14
   - 2.3 CppUTest: A C++ Unit Test Harness 21
   - 2.4 Unit Tests Can Crash 24
   - 2.5 The Four-Phase Test Pattern 25
   - 2.6 Where Are We? 25

3. Starting a C Module 27
   - 3.1 Elements of a Testable C Module 27
   - 3.2 What Does an LED Driver Do? 29
   - 3.3 Write a Test List 29
   - 3.4 Writing the First Test 31
   - 3.5 Test-Drive the Interface Before the Internals 36

3.6 Incremental Progress 42
3.7 Test-Driven Developer State Machine 45
3.8 Tests Are FIRST 46
3.9 Where Are We? 47

4. Testing Your Way to Done 49
4.1 Grow the Solution from Simple Beginnings 49
4.2 Keep the Code Clean—Refactor as You Go 64
4.3 Repeat Until Done 67
4.4 Take a Step Back Before Claiming Done 73
4.5 Where Are We? 74

5. Embedded TDD Strategy 77
5.1 The Target Hardware Bottleneck 77
5.2 Benefits of Dual-Targeting 78
5.3 Risks of Dual-Target Testing 79
5.4 The Embedded TDD Cycle 80
5.5 Dual-Target Incompatibilities 83
5.6 Testing with Hardware 88
5.7 Slow Down to Go Fast 91
5.8 Where Are We? 91

6. Yeah, but... 93
6.1 We Don't Have Time 93
6.2 Why Not Write Tests After the Code? 97
6.3 We'll Have to Maintain the Tests 97
6.4 Unit Tests Don't Find All the Bugs 98
6.5 We Have a Long Build Time 98
6.6 We Have Existing Code 99
6.7 We Have Constrained Memory 99
6.8 We Have to Interact with Hardware 100
6.9 Why a C++ Test Harness for Testing C? 101
6.10 Where Are We? 102

Part II — Testing Modules with Collaborators

7. Introducing Test Doubles 107
7.1 Collaborators 107
7.2 Breaking Dependencies 108
7.3 When to Use a Test Double 112

7.4 Faking It in C, What's Next 113
7.5 Where Are We? 116

8. Spying on the Production Code 117
8.1 Light Scheduler Test List 118
8.2 Dependencies on Hardware and OS 118
8.3 Link-Time Substitution 119
8.4 Spying on the Code Under Test 120
8.5 Controlling the Clock 126
8.6 Make It Work for None, Then One 127
8.7 Make It Work for Many 140
8.8 Where Are We? 145

9. Runtime-Bound Test Doubles 147
9.1 Testing Randomness 147
9.2 Faking with a Function Pointer 149
9.3 Surgically Inserted Spy 152
9.4 Verifying Output with a Spy 156
9.5 Where Are We? 160

10. The Mock Object 163
10.1 Flash Driver 163
10.2 MockIO 171
10.3 Test-Driving the Driver 174
10.4 Simulating a Device Timeout 178
10.5 Is It Worth It? 180
10.6 Mocking with CppUMock 180
10.7 Generating Mocks 183
10.8 Where Are We? 185

Part III — Design and Continuous Improvement

11. SOLID, Flexible, and Testable Designs 189
11.1 SOLID Design Principles 190
11.2 SOLID C Design Models 193
11.3 Evolving Requirements and a Problem Design 195
11.4 Improving the Design with Dynamic Interface 203
11.5 More Flexibility with Per-Type Dynamic Interface 210
11.6 How Much Design Is Enough? 214
11.7 Where Are We? 216

12. Refactoring 219
12.1 Two Values of Software 219
12.2 Three Critical Skills 220
12.3 Code Smells and How to Improve Them 222
12.4 Transforming the Code 232
12.5 But What About Performance and Size? 249
12.6 Where Are We? 252

13. Adding Tests to Legacy Code 253
13.1 Legacy Code Change Policy 253
13.2 Boy Scout Principle 254
13.3 Legacy Change Algorithm 255
13.4 Test Points 257
13.5 Two-Stage struct Initialization 260
13.6 Crash to Pass 263
13.7 Characterization Tests 268
13.8 Learning Tests for Third-Party Code 271
13.9 Test-Driven Bug Fixes 274
13.10 Add Strategic Tests 274
13.11 Where Are We? 274

14. Test Patterns and Antipatterns 277
14.1 Ramble-on Test Antipattern 277
14.2 Copy-Paste-Tweak-Repeat Antipattern 279
14.3 Sore Thumb Test Cases Antipattern 280
14.4 Duplication Between Test Groups Antipattern 282
14.5 Test Disrespect Antipattern 283
14.6 Behavior-Driven Development Test Pattern 283
14.7 Where Are We? 284

15. Closing Thoughts 285

Part IV — Appendixes

A1. Development System Test Environment 291
A1.1 Development System Tool Chain 291
A1.2 Full Test Build makefile 293
A1.3 Smaller Test Builds 294

A2. Unity Quick Reference 297
A2.1 Unity Test File 297
A2.2 Unity Test main 299
A2.3 Unity TEST Condition Checks 299
A2.4 Command-Line Options 300
A2.5 Unity in Your Target 300

A3. CppUTest Quick Reference 303
A3.1 The CppUTest Test File 303
A3.2 Test Main 304
A3.3 TEST Condition Checks 304
A3.4 Test Execution Order 305
A3.5 Scripts to Create Starter Files 305
A3.6 CppUTest in Your Target 306
A3.7 Convert CppUTest Tests to Unity 307

A4. LedDriver After Getting Started 309
A4.1 LedDriver First Few Tests in Unity 309
A4.2 LedDriver First Few Tests in CppUTest 310
A4.3 LedDriver Early Interface 310
A4.4 LedDriver Skeletal Implementation 311

A5. Example OS Isolation Layer 313
A5.1 Test Cases to Assure Substitutable Behavior 314
A5.2 POSIX Implementation 315
A5.3 Micrium RTOS Implementation 317
A5.4 Win32 Implementation 319
A5.5 Burden the Layer, Not the Application 320

A6. Bibliography 321

Index 323

Foreword by Jack Ganssle

Test-Driven Development for Embedded C is hands-down the best book on the subject. This is an amiable, readable book with an easy style that is fairly code-centric, taking the reader from the essence of TDD through mastery using detailed examples. It's a welcome addition to the genre because the book is completely C-focused, unlike so many others, and is specifically for those of us writing firmware.

James skips no steps and leads one through the gritty details but always keeps the discussion grounded so one is not left confused by the particulars. The discussion is laced with homey advice and great insight. He's not reluctant to draw on the wisdom of others, which gives the book a sense of completeness.

The early phases of a TDD project are mundane to the point of seeming pointlessness. One writes tests to ensure that the most elemental of things work correctly. Why bother checking to see that what is essentially a simple write works correctly? I've tossed a couple of books on the floor in disgust at this seeming waste of time, but James warns the gentle reader to adopt patience, with a promise, later fulfilled, that he'll show how the process is a gestalt that yields great code.

TDD does mean one is buried in the details of a particular method or a particular test, and the path ahead can be obscured by the tests at hand. If you're a TDD cynic or novice, be sure to read the entire book before forming any judgments so you can see how the details morph into a complete system accompanied by a stable of tests.

Better than any book I've read on the subject, *Test-Driven Development for Embedded C* lays out the essential contrast between TDD and the more conventional write-a-lot-of-code-and-start-debugging style for working. With the latter technique, we're feeding chili dogs to our ulcers as the bugs stem from work we did long ago and are correspondingly hard to find. TDD, on the other hand, means today's bug is a result of work one did ten minutes ago. They're

exposed, like ecdysiast Gypsy Rose Lee's, uh, assets. A test fails? Well, the bug must be in the last thing you did.

One of TDD's core strengths is the testing of boundary conditions. My file of embedded disasters reeks of expensive failures caused by code that failed because of overflows, off-by-one errors, and the like. TDD—or, at least James' approach to it—means getting the "happy" path working and tested and then writing tests to ensure each and every boundary condition is also tested. Conventional unit testing is rarely so extensive and effective.

Embedded TDD revolves around creating a test harness, which is a software package that allows a programmer to express how production code should behave. James delves into both Unity and CppUTest in detail. (Despite its name, the latter supports both C++ and C). Each test invokes creation and teardown routines to set up and remove the proper environment, like, for instance, initializing a buffer and then checking for buffer overflows. I found that very cool.

Test-Driven Development for Embedded C is an active-voice work packed with practical advice and useful aphorisms, such as "refactor on green" (get the code working first, and when the tests pass, then you can improve the code if necessary). Above all, the book stresses having fun while doing development. And that's why most of us got into this field in the first place.

Jack Ganssle

Foreword by Robert C. Martin

You've picked up this book because you are an embedded software engineer. You don't live in the programmer's world of multicores, terabytes, and gigaflops. You live in the *engineer's* world of hard limits and physical constraint and of microseconds, milliwatts, and kilobytes. You probably use C more than C++ because you *know* the code the C compiler will generate. You probably write assembler when necessary because sometimes even the C compiler is too profligate.

So, what are you doing looking at a book about Test-Driven Development? You don't live in the kind of spendthrift environment where programmers piddle around with fads like that. Come on, TDD is for Java programmers and Ruby programmers. TDD code runs in interpreted languages and virtual machines. It's not for the kind of code that runs on *real metal*, is it?

James Grenning and I cut our teeth on embedded software in the late 70s and early 80s. We worked together programming 8085 assembler on telephone test systems that were installed in racks in telephone central offices. We spent many an evening in central offices sitting on concrete floors with oscilloscopes, logic analyzers, and prom burners. We had 32KB of RAM and 32KB of ROM in which to work our miracles. And boy, what miracles we worked!

James and I were the first to introduce C into the embedded systems at our company. We had to fight the battles against those hardware engineers who claimed "C is too slow." We wrote the drivers, the monitors, and the task switchers that allowed our systems run in a 16-bit address space split between RAM and ROM. It took several years, but in the end, we saw all the newer embedded systems at our company written in C.

After those heady days in the 70s and 80s, James and I parted company. I wandered off into the realms of IT and product-ware, where resources flow like wine at an Italian wedding. But James had a special love for the embedded world, so for the past thirty+ years James Grenning has been writing code in

embedded environments such as digital telephone switches, high-speed photocopiers, radio controllers, cell phones, and the like.

James and I joined forces again in the late 90s. He and I consulted at Xerox on the embedded C++ software running on 68000s in Xerox's high-end digital printers. James was also consulting at a well-known cell phone company on its communications subsystems.

As accomplished as James is as an embedded software engineer, he is also an accomplished software craftsman. He cares deeply about the code he writes and the products he produces. He also cares about his industry. His goal has always been to improve the state-of-the-art in embedded development.

When the first XP Immersion took place in 1999, James was there. When the Agile Manifesto was conceived in Snowbird in 2001, James was there and was one of the original signatories. James was determined to find a way to introduce the embedded industry to the values and techniques of Agile software development.

So, for the past decade, James has participated in the Agile community and worked to find a way to integrate the best ideas of Agile software development with embedded software development. He has introduced TDD to many embedded shops and helped their engineers write better, more reliable, embedded code.

This book is the result of all that hard work. This book is the integration of Agile and embedded. Actually, this book has the wrong title. It should be *Crafting Embedded Systems in C* because although this book talks a lot about TDD, it talks about an awful lot more than that! This book provides a very complete and highly professional approach to engineering high-quality embedded software in C, quickly and reliably. I think this book is destined to become the bible of embedded software engineering.

Yes, you *can* do TDD in the embedded world. Not only that, you should! In these pages, James will show you how to use TDD economically, efficiently, and profitably. He'll show you the tricks and techniques, the disciplines, and the processes. And, he'll show you the code!

Get ready to read a lot of code. This book is chock-full of code. And it's code written by a craftsman with a lot to teach. As you read through this book and all the code within it, James will teach you about testing, design principles, refactoring, code smells, legacy code management, design patterns, test patterns, and much more.

And, on top of that, the code is almost entirely written in C and is 100 percent applicable to the constrained development and execution environments of embedded systems.

So, if you are a pragmatic embedded engineer who lives in the real world and codes close to the metal, then, yes, this book is for you. You've picked it up and read this far. Now finish what you started and read the rest of it.

Robert C. Martin (Uncle Bob)
October 2010