Test-Driven Database Development
The Net Objectives Lean-Agile Series provides fully integrated Lean-Agile training, consulting, and coaching solutions for businesses, management, teams, and individuals. Series editor Alan Shalloway and the Net Objectives team strongly believe that it is not the software, but rather the value that software contributes – to the business, to the consumer, to the user – that is most important.

The best – and perhaps only – way to achieve effective product development across an organization is a well-thought-out combination of Lean principles to guide the enterprise, agile practices to manage teams, and core technical skills. The goal of The Net Objectives Lean-Agile Series is to establish software development as a true profession while helping unite management and individuals in work efforts that "optimize the whole,” including

- The whole organization: Unifying enterprises, teams, and individuals to best work together
- The whole product: Not just its development, but also its maintenance and integration
- The whole of time: Not just now, but in the future – resulting in a sustainable return on investment

The titles included in this series are written by expert members of Net Objectives. These books are designed to help practitioners understand and implement the key concepts and principles that drive the development of valuable software.
Test-Driven Database Development
Unlocking Agility

Max Guernsey, III
This book is dedicated to my wife, Amy Bingham, who is largely responsible for my being the man I am today.
## Contents at a Glance

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preface</td>
<td></td>
<td>xix</td>
</tr>
<tr>
<td>Chapter 1</td>
<td>Why, Who, and What</td>
<td>1</td>
</tr>
<tr>
<td>Chapter 2</td>
<td>Establishing a Class of Databases</td>
<td>9</td>
</tr>
<tr>
<td>Chapter 3</td>
<td>A Little TDD</td>
<td>19</td>
</tr>
<tr>
<td>Chapter 4</td>
<td>Safely Changing Design</td>
<td>37</td>
</tr>
<tr>
<td>Chapter 5</td>
<td>Enforcing Interface</td>
<td>63</td>
</tr>
<tr>
<td>Chapter 6</td>
<td>Defining Behaviors</td>
<td>95</td>
</tr>
<tr>
<td>Chapter 7</td>
<td>Building for Maintainability</td>
<td>115</td>
</tr>
<tr>
<td>Chapter 8</td>
<td>Error and Remediation</td>
<td>137</td>
</tr>
<tr>
<td>Chapter 9</td>
<td>Design</td>
<td>159</td>
</tr>
<tr>
<td>Chapter 10</td>
<td>Mocking</td>
<td>191</td>
</tr>
<tr>
<td>Chapter 11</td>
<td>Refactoring</td>
<td>213</td>
</tr>
<tr>
<td>Chapter 12</td>
<td>Legacy Databases</td>
<td>227</td>
</tr>
<tr>
<td>Chapter 13</td>
<td>The Facade Pattern</td>
<td>249</td>
</tr>
<tr>
<td>Chapter 14</td>
<td>Variations</td>
<td>269</td>
</tr>
<tr>
<td>Chapter 15</td>
<td>Other Applications</td>
<td>283</td>
</tr>
<tr>
<td>Index</td>
<td></td>
<td>301</td>
</tr>
</tbody>
</table>
Contents

Foreword ....................................................... xvii
Preface ......................................................... xix
Acknowledgments ............................................. xxv
About the Author ............................................. xxvii

Chapter 1 Why, Who, and What ......................... 1

Why .......................................................... 1
Agility Progressively Invades Domains Every Day ...... 2
Agility Cannot Work Without TDD ...................... 2
TDD in the Database World Is a Challenge .......... 3

Who ............................................................ 3
TDD and OOP .................................................. 4
Applications and Databases ............................... 4

What .......................................................... 4
Databases Are Objects .................................... 5
TDD Works on Classes, Not Objects ................. 5
We Need Classes of Databases ....................... 6
Summary ....................................................... 7

Chapter 2 Establishing a Class of Databases .......... 9

The Class’s Role in TDD ..................................... 9
A Reliable Instantiation Process ....................... 10
Tests Check Objects ..................................... 10
Classes in Object-Oriented Programming Languages ... 11
Making Classes Is Easy: Just Make New Objects .... 11
One Path: Destroy If Necessary .................... 11

Classes of Databases ..................................... 12
Two Paths: Create or Change ....................... 12
The Hard Part: Unifying the Two Paths ............ 13
Real Database Growth ................................. 13
### Contents

How About Making Every Database Build Like Production Databases? ........................................ 14
All DBs Would Follow the Exact Same Path .................................................. 15
Incremental Build .................................................................................. 15
  Document Each Database Change ...................................................... 15
  Identify Current Version ................................................................. 16
  Apply Changes in Order as Needed ................................................... 16
Implementation ............................................................................... 16
  Requirements .............................................................................. 16
  Pseudocode Database Instantiation Mechanism ................................ 17
  Pseudocode Input ........................................................................ 17
Summary ......................................................................................... 18

Chapter 3  A Little TDD ................................................................. 19
The Test-First Technique ..................................................................... 19
  Write the Test ........................................................................... 20
  Stub Out Enough to See a Failure .................................................. 22
  See the Test Pass ....................................................................... 22
  Repeat ..................................................................................... 23
Tests as Specifications .................................................................. 24
  “Tests Aren’t Tests, They Are Specifications” .................................. 24
  “Tests Aren’t Specifications, They Are Tests” ............................... 25
  Tests Are Executable Specifications ............................................. 26
Incremental Design ........................................................................ 27
Building Good Specifications ........................................................... 28
  Specify Behavior, Not Structure ..................................................... 28
  Drive Design In from Without, Not the Other Way Around .................... 29
  Defining the Design Inside Out ..................................................... 30
  Defining the Design Outside In ...................................................... 32
Summary ......................................................................................... 34

Chapter 4  Safely Changing Design .............................................. 37
What Is Safe? ................................................................................ 38
  Breaking a Contract Is a Little Bad ................................................ 38
  Losing Data Will Probably Get You Fired ...................................... 39
  Not Changing Design Is Also Dangerous ....................................... 40
Solution: Transition Testing .................................................. 44
    Test-Driving Instantiation ........................................... 44
    Transition Testing Creation ....................................... 44
    Transition Testing Addition ....................................... 47
    Transition Testing Metamorphosis ............................... 51
    Why Not Use the Public Interface? ............................... 56

Transition Safeguards .................................................. 56
    Read/Read Transition Tests ...................................... 56
    Run by the Class of Databases on Every Upgrade .............. 60
    Backup and Rollback on Fail ..................................... 60
    Making Transition Tests Leverage Transition Safeguards .... 60

Summary ................................................................. 61

Chapter 5 Enforcing Interface ........................................ 63

Interface Strength ...................................................... 64
    Stronger Coupling Languages .................................... 64
    Weaker Coupling Languages ..................................... 65
    The Common Thread ................................................. 66
    Coupling to Database Classes .................................... 66
    The Problem Is Duplication ...................................... 66

Client-Object-Like Enforcement ...................................... 67
    Creating Demand for a DatabaseDesign Class ................. 67
    Specifying the DatabaseDesign Class ......................... 68
    Getting Rid of Duplication with Multiple Client Platforms 70
    What Happens When Coupling Goes Bad? ........................ 71
    Eliminating Duplication Between Database Build and Client Code ........................................... 71
    Decoupling Implementation from Design ....................... 72

Sticking Point: Change .................................................. 73
    Designs Change Over Time ......................................... 74
    Document All Versions of Design ............................... 75
    Couple to the Correct Version of the Design ................ 77

Sticking Point: Coupling ................................................. 78
    Various Clients Couple to Various Versions .................. 78
    Having to Change Everything All the Time Is Duplication, Too .................................................. 79
Introducing the Lens Concept ........................................ 83
Virtual Lenses ......................................................... 85
The “Current” Lens .................................................. 89
The “New” Lens ...................................................... 89
Summary ............................................................... 93

Chapter 6 Defining Behaviors ........................................ 95
A New Group of Problems ............................................ 96
No Encapsulation ....................................................... 96
Hide Everything ....................................................... 97
Business Logic in the Database ................................. 97
Knowledge, Information, and Behavior ...................... 98
Information ............................................................ 99
Knowledge ............................................................ 102
Behavior ............................................................... 102
Outside-In Development ............................................ 106
Defining the Test ..................................................... 106
Growing Interface .................................................... 108
Growing Behavior and Structures ............................. 109
Justification by Specification ...................................... 111
Work Against Present Requirements, Not Future ........ 111
Build in Increments .................................................. 112
Limit Access to What Is Specified ............................ 112
Summary ............................................................... 113

Chapter 7 Building for Maintainability .......................... 115
Never Worry About the Future ................................. 116
Look for Opportunities in the Now ......................... 116
Design to Information .............................................. 117
Translate Info and Knowledge with Behavior ............. 121
Guard Knowledge with Fervor and Zeal .................... 124
Not Changing Is the Most Dangerous Choice .......... 124
Keep Your Design Natural ....................................... 126
Deal with the Future When It Happens ................... 127
Define New Design .................................................. 128
Introduce Minimal Changes ..................................... 129
Chapter 8 Error and Remediation

Kinds of Errors
Axis: Is the Error Good or Bad?
Axis: Is the Error Released or Not?
Dealing with Good Errors
Just Fix It
Document Behavior Now
Trace Feature Back to Its Genesis
Dealing with Bad Errors
Unreleased Errors
Released Errors
Catastrophic Errors
Summary

Chapter 9 Design

Structures Versus Design
Structures: Execution Details
Tests and Class Information
What Is Design?
Buckets of Concepts
Mandatory Part of True TDD
Composition and Aggregation
Composition: One Thing with Multiple Parts
Aggregation: Connecting Distinct Things
Reuse
Avoid Developing the Same Thing Twice
Reuse by Composition or Aggregation
Abstraction
Identifying Opportunities for Abstraction
Encapsulating Behaviors
Finding Ways to Allow Variation in Dependencies
Dealing with the Time Problem
Summary
## Contents

### Chapter 10 Mocking

- Testing Individual Behaviors ............................................. 191
  - Why Encapsulate ..................................................... 192
  - Tests Test Everything Not Under Their Control .............. 193
  - Controlling Irrelevant Behaviors from Tests ............... 194
  - Mocking Controls Behaviors ...................................... 194
- Mocking in Object-Oriented Programming ....................... 195
  - Setup ........................................................................ 195
  - Decoupling ............................................................. 199
  - Isolation .................................................................... 202
  - Integration .............................................................. 202
- Mocking in Database Design ............................................... 203
  - Example Problem ...................................................... 204
  - Example Solution ...................................................... 205
  - Composition ............................................................. 208
  - Aggregation .............................................................. 210
  - Designing for Testability .......................................... 210
- Summary ......................................................................... 210

### Chapter 11 Refactoring

- What Refactoring Is .......................................................... 214
  - Changing Design Without Changing Behavior ............. 214
  - In the Context of Passing Tests .................................. 215
- Lower and Higher Risk Design Changes ......................... 222
  - Lower Risk: Changing Class-Level Design ..................... 222
  - Medium Risk: Rearranging Behavior Logic ..................... 223
  - Higher Risk: Altering Knowledge Containers ............... 225
  - This Is Not an Invitation to Skip Testing ..................... 226
- Summary ......................................................................... 226

### Chapter 12 Legacy Databases

- Promoting to a Class ....................................................... 228
  - Deducing Initial Version ............................................. 228
  - Pinning the Transition Behavior with Tests .................. 231
- Controlling Coupling ...................................................... 231
  - Identifying and Locking Down to Existing Uses ............ 232
  - Encapsulating on Demand .......................................... 234
Chapter 15  Other Applications ................................. 283
   XML .......................................................... 284
   Encapsulation ................................................. 284
   XSD Schemas .................................................. 284
   XSLT Transitions ............................................ 286
   Transition Test XSLT Changes ............................. 287
   File Systems and Other Object Directories .............. 288
      Transition Test File System Manipulations ............ 289
      Shell Script Transitions ................................. 291
   Data Objects .................................................. 292
      Class Definitions Are Schemas ......................... 292
      Transition Test the Ugrader Class ..................... 294
      Code Transitions ......................................... 296
   Summary and Send Off ...................................... 300

Index .................................................................... 301
This page intentionally left blank
I’ve been a Test-Driven Development practitioner for many years and have also been writing, lecturing, and teaching courses on the subject as part of my duties at Net Objectives. A question that often arises during classes or conference talks is this: TDD seems ideal for business logic and other “middle-tier” concerns, but what about the presentation (UI) tier and the persistence (database) tier?

My answer, typically, has been to point out that there are two issues in each case:

1. How to manage the dependencies from the middle tier to the other two, so as to make the middle tier behaviors more easily testable
2. How to test-drive the other layers themselves

For issue #1, this is a matter of interfaces and mock objects, design patterns, and good separation of concerns in general. It is a matter of technique, and the TDD community has a lot of mature, proven techniques for isolating business logic from its dependencies. A lot of time is spent in my courses on these “tricks of the trade.”

But when it comes to issue #2—test-driving the user interface and the database themselves—I’ve always said that these are largely unsolved problems. Not that we don’t know how to test these things, but rather that we don’t know how to test drive them, to write the kind of isolated, fast, granular tests that good TDD requires. We have to be satisfied with more traditional testing when it comes to these other layers of the system; this has been my traditional answer.

As far as I know, this is still true for the UI. But when it comes to databases, Max Guernsey has figured it out.

In my book Emergent Design, I talk a lot about the parallels between systems design and evolutionary processes in nature. If you think of your source code as the “DNA” of the system and the executable as the “individual organism,” it really fits. The DNA is used to generate the individual. To make a change to the species, the DNA changes first, and then the next individual generated is altered. Nature does not evolve individuals; it makes changes to the species generationally. This is much akin to the code/compile/run nature of software. We throw
away the .exe file, change the source code, and the compiler makes a new, different, hopefully better executable. Because of this, the source is king. It’s the one thing we cannot afford to lose.

Max’s insight began, at least in my conversations with him, with the notion that databases are not like this. If the schema is akin to the DNA, and thus an installed, running instance of the database (with all its critical enterprise data) is an individual, we cannot take nature’s cavalier attitude toward it. The schema is easy to re-create—you just run your DDL scripts or equivalent. But a given, installed, “living” database contains information and knowledge that must be preserved as its structure changes.

Because of this, the paradigm of evolution does not fit. In databases, the individual is paramount. We cannot simply throw it away and re-create it from altered DNA. Nature, again, does not evolve individuals. But there is a natural paradigm that works. It is morphing. It is the way an individual creature transitions from one life phase to another—tadpole to frog, for example. In examining this insight and all of its ramifications, Max has developed a truly revolutionary view about databases: how to create them and change them and, at long last, how to test-drive them. His approach gives the database practitioner the fundamental clarity, safety, and leverage that a TDD practitioner enjoys.

This book is a ground-breaking work. Max has discovered the Rosetta stone of database development here, and if you follow him carefully, you will leave with a far more powerful way of doing your job when it comes to the persistence tier of your system. You will have the knowledge, tools, and overall approach that make this possible.

—Scott Bain
Senior Consultant, Net Objectives
Preface

This book applies the concepts of test-driven development to database development.

Who Should Read This Book

The short answer is “anyone who wants to learn how to do test-driven development of a database and is willing to do the hard work to get there.” The long answer follows.

This book is aimed primarily at programmers who are in some way responsible for the development of at least one database design. A secondary target is people who think of themselves primarily as database developers who are interested in adding test-driven development to their process.

That is not meant to in any way diminish the value of the second group of people. The techniques in this book build on principals and methods that, at the time of this writing, are gaining widespread acceptance among the former group and still struggling to gain traction with the latter. That’s not to say that things won’t change—I hope they will—but this book would have spun out of control if I tried to take on all first principles from which it is derived.

The goal of this book is to help people apply the process of test-driven development to the new domain of database development, where the forces are different if only slightly.

If you read the book and are able to sustainably drive development of your database through test, it’s a win both for you and for me. If you start using the principles to port over other techniques, such as pattern-oriented development, then the win will be doubled. If you start porting what you learn there back into any other domain where long-lived data is involved, such as the development of your installers, then the win is greater still.
What Needs to Be Done

To serve this goal, I start by establishing why test-driven development works. I then look at why it has so much trouble gaining traction in the database world. Mind you, it’s not that I think database development is basically untested, but my experience has shown me that it is not sustainably tested nor is it test driven.

The main problem that makes database testing hard is the absence or misplacement of the concept of a class. Even the most “Wild-West-style” modern language still supports the idea of having classes and instances. Database engines either pay lip service to this by having classes of data structures or do nothing whatsoever to establish truly testable classes. The reason for this appears to me to be that people generally haven’t recognized what the true first-class object of the database world is: the database itself. So the first step is to establish a class of databases.

Change is central to the test-driven development process. You are constantly changing design to support new needs and to support the testability of an expanding feature set. One of the forces that makes test-driven development harder to adopt and to sustain is that change is seen as more dangerous in database designs than in other kinds of design.

If you mess up a design change in your middle tier, you might have to roll back. If you mess up a design change in the data tier, you could erase valuable knowledge stored therein. The solution is to test not only what your database does but also how it is changed.

Another problem that faces databases in regard to changing design is that the coupling between a database and its clients is weakly enforced. The possibility exists to make a change to a database’s interface and not discover that you’ve broken a downstream application for a very long time. That risk can be mitigated by using your class of databases to harden the relationship between a database design and its clients.

Creating a strong class of databases with a controlled way of changing things solves the basic problems in the database world that stand in the way of TDD. That is, it gets developers up to the early ’90s in terms of support for modern practices.

To get into the twenty-first century, you have to go a bit further. To that end, I help you understand that the scope of a test should be verification of a behavior. I support that by defining what a behavior is in the context of database development.
Enabling Emergent Design

I also show you how to maximize long-term maintainability by limiting a database’s scope to what you need right now and using the techniques in this book to make it easy to add more features later. That is, I’ll help you give up the fear you tend to associate with not planning a database’s design far in advance.

No process is perfect and, even if it were, none of the people executing it would be. Despite all your efforts to avoid them, mistakes will be made. If you are doing true test-driven database development, most of the time a mistake will come in the form of a behavior not expressed in your test suite. I’ll show you how best to correct an error.

Knowing how to develop and write tests for a class of databases while keeping their designs as simple and problem-appropriate as possible will bring database development into the twenty-first century—just one short era behind modern object-oriented development.

Modernizing Development

The final phase of modernizing database development to support a test-driven process involves adopting and adapting what I call the “advanced” object-oriented methodologies.

Getting a grip on design at the class/database level is the first phase. If you have a choice between one big database design and two smaller ones, you should choose the pair of smaller ones. If the database technology isn’t in a place to permit that, use composition to place two logical database instances inside one physical database instance.

Another important activity is refactoring. You need to keep a database’s design problem-appropriate for its entire life. That means the design must start out small, but it also means that it needs to change shape as you cover more and more of the problem space. I’ll show you how to refactor database designs in the context of a test-driven process.

That will be everything you need to do test-driven database development for a typical, new database design. The remainder of the book is dedicated to helping you “wrangle” databases that were born in an untested context, deviating from the process in a controlled and testable way, and adapting the process to non-database applications.
Doing test-driven database development will not be easy—not at first. If you’ve already obtained a firm grip on “regular” test-driven development, this should come as absolutely no surprise. No matter how quickly you pick up new skills, TDD will take longer than anything else.

It costs so much because it is worth so much. When you finish this book, you’ll have a theoretical understanding of test-driven database development. One, three, or even eighteen months later, you will have mastered it.

After mastering it, you’ll be able to frequently, rapidly, and safely change your database designs with confidence. You’ll be able to build just what you need, just when you need it. As a result, database improvement can become a fluid part of your software development process. In addition you’ll be able to keep the design of your database clean, simple, and fast.

Chapter-by-Chapter Breakdown

Here’s a chapter-by-chapter breakdown of what this book covers.

In Chapter 1, “Why, Who, and What,” I explain why I wrote this book, who should read it, and what the real roadblocks to TDD are in the context of database design. I wrote this book because true test-driven development hasn’t really gained any traction in the database world. I am targeting people who think of themselves as software developers and also must work with database designs. The biggest problem in the database development world is that there is no clear concept of a class, which is a central element of traditional TDD efforts.

To build a class of databases, you need to keep a permanent record of the exact set of scripts that are run on a database and to have a clean way of tracking which ones have already been run. A little infrastructure allows you to ensure that every instance of a class of databases is built exactly the same way. In Chapter 2, “Establishing a Class of Databases,” I show you how to do exactly that.

Many things go into having a sustainable TDD process for a database design. The first step is to define a basic TDD process to which you can later add deeper, more data-oriented activities. In Chapter 3, “A Little TDD,” I show you how to do some simple test-driven development against a class of databases.

In Chapter 4, “Safely Changing Design,” I show you how to overcome one of the big obstacles: the risk associated with change. Introducing change frequently frightens a lot of people. The root of that fear is that databases store a lot of valuable stuff, and losing some of the data on account of a hastily made change is unacceptable in most environments. The fear and the risk can both be vanquished by testing not only the behavior of your database, but also the scripts that build or modify it.
Databases are the most depended-upon things in the software industry and modifying one’s design can have unforeseen consequences. At the heart of this problem is massive, infective duplication that many simply accept as “natural.” In Chapter 5, “Enforcing Interface,” I show you how to control the cost of a rapidly evolving database design by eliminating that duplication.

As far as the TDD process is concerned, tests specify behaviors in objects. The question then becomes “What is a behavior in the context of a database?” In Chapter 6, “Defining Behaviors,” I set a pretty good scope for a test by answering that question.

Having the scope of a single test well defined gives you the freedom to explore the larger topic of what kinds of database designs are conducive to change and which ones are difficult to maintain. In Chapter 7, “Building for Maintainability,” I show you that keeping a database light, lean, and simple is a better path to supporting future needs than attempting to predict now what you will need months from now.

“Sure, this is all great if you never mess up,” one might say, “but what happens when we do?” In Chapter 8, “Error and Remediation,” I show you techniques that allow you to deal with any unplanned changes that might find their way into your database design.

In Chapter 9, “Design,” I make recommendations about how to design a class of databases for maximum testability. I then go a little further and show how to apply object-oriented design concepts to classes of databases.

Tests are frequently plagued by unwanted coupling. Dependencies between behaviors create ripple effects and single changes end up causing dozens of tests to fail. In Chapter 10, “Mocking,” I show you how to isolate behaviors from one another by building on the design techniques shown in Chapter 9.

The better your test coverage and the more rapidly you can introduce change, the more frequently you are going to modify design. In Chapter 11, “Refactoring,” I demonstrate how to alter the design of your database while preserving behavior.

No process is complete unless it includes a mechanism to ingest software developed before its introduction. In Chapter 12, “Legacy Databases,” I cover one of two ways to take a database that was developed with databases that weren’t developed using the practices in this book by gradually covering them in tests.

Chapter 13, “The Façade Pattern,” covers the other option for dealing with a legacy database. When employing the Façade pattern, you encapsulate a legacy design behind a new, well-tested one and gradually transfer behaviors from the old design to the new.
I would be crazy if I tried to sell this as a “one-size-fits-all” solution to the problem of bringing TDD to the database development world. The practices herein will work for a lot of people without modification. However, some people operate under conditions to which the first thirteen chapters of this book do not perfectly fit. In Chapter 14, “Variations,” I cover some of the adaptations I’ve seen people apply in the past.

Finally, in Chapter 15, “Other Applications,” I demonstrate a number of ways that the various techniques in this book can apply to data persisted by means other than a database. Some examples of these other storage mechanisms are file systems, XML documents, and the dreaded serialized middle-tier object.

Downloadable Code

You can download the code used in this book by going to http://maxthe3rd.com/test-driven-database-development/code.aspx
Acknowledgments

There are numerous influences behind this book ranging back over nearly a decade.

First thanks go to my wife, Amy. She was a constant source of motivation and validation in the course of developing this book and, in the 15 years we’ve been together, has done at least one proofread of everything I’ve written.

Bill Zietzke was around at the very beginning of the process. It was a conversation I had with him while we were both contracting at an insurance company in Bellevue, Washington, that got this whole thing started.

Beau Bender helped discover the mechanism I prescribe for controlling the coupling between databases and their clients. For that, I am grateful.

My good friend and mentor, Scott L. Bain, is also deserving of thanks. It was he who first pushed me to get published and it was his influence that helped make it happen. He has also played an instrumental role in developing this technique by contributing valuable questions, criticisms, and observations along the way.

Alan Shalloway has also been a friend and mentor to me throughout the majority of my career. He helped me decide that my first idea—teaching everything to everyone—was not the right one. I am certain that, without his very constructive criticism, you would be reading a completely different book, probably written by a completely different author.

Both Alan and Scott played pivotal roles in my recent development as a professional software developer. Each nurtured the skills he already saw and supplied me with skills that were missing in a format that circumvented the considerable defenses that protect me from new things and ideas.

Of equal, and possibly greater, value was their advice on how to deal with people. At the time of this writing, I’m not exactly a beloved consensus builder, but I am a lot better at persuasion than I was before I met Scott and Alan. Without that guidance, without showing me how important it is to share what we know in an accessible format, I probably wouldn’t even have cared to write a book in the first place.

Some of my recently acquired friends and colleagues have served as guinea pigs, reading early versions of various chapters of the book for me. This allowed me to acquire feedback soon enough to act and helped me decide to put this
book through its second transformation. Without feedback from Seth McCarthy and Michael Gordon Brown, I would have tried to release a much larger book about “agile” database development instead of the focused, technical book you are reading now.

It goes without saying that my parents are in part responsible because, without them, there would be no me. However, my father played a special role in my development as a young programmer. Without his influence, I would probably be something useless, like a mathematician or a Wall Street analyst.
Max Guernsey is currently a Managing Member at Hexagon Software LLC. He has 15 years of experience as a professional software developer. For nearly half that time, he has been blogging, writing, and delivering lectures on the topic of agile and test-driven database development.

For much of Max’s professional career, he has been a consultant, advising a variety of software companies in many different industries using multiple programming and database technologies. In most of these engagements, he spent months or even years helping teams implement cutting-edge techniques such as test-driven development, object-oriented design, acceptance-test-driven development, and agile planning.

Max has always been a “hands-on” consultant, working with teams for long periods of time to help them build both software and skills. This series of diverse, yet deep, engagements helped him gain a unique understanding of the database-related testing and design problems that impede most agile teams. Since 2005, he has been thinking, writing, blogging, lecturing, and creating developer-facing software dedicated to resolving these issues.

Max can be reached via email at max@hexsw.com. He also posts regularly on his Twitter account (@MaxGuernseyIII) and his blog (maxg3prog.blogspot.com).
Chapter 3

A Little TDD

This chapter gives you a crash course in test-driven development (TDD) in case you are not familiar with the discipline.

A staple of the TDD process is the test-first technique. Many people who are new to test-driven development actually confuse it with the test-first technique, but they are not the same thing. Test-first is one tool in the TDD toolbelt, and a very important one at that, but there is a lot more to TDD.

The chapter then covers a test’s proper role in your organization. Tests are best thought of as executable specifications. That is, they not only test something but they also document what that thing should do or how it should look.

One very powerful benefit of cyclically defining and satisfying executable specifications is that it forces your design to emerge incrementally. Each new test you write demands that you revisit and, if necessary, revise your design.

Following that discussion, I cover what you actually want to specify and, probably at least as important, what you do not want to specify. In a nutshell, the rule is “specify behaviors only.” Deciding what a database’s behavior should be can be a little difficult, and I cover that topic in Chapters 6, “Defining Behaviors,” and 7, “Building for Maintainability.” This chapter deals with the behaviors inherent in tables.

Finally, an important piece of test-driven development is to drive behaviors into a database from outside, not the other way around. Again, you can find a lot more advice on how a database should actually be structured later in the book. This chapter deals only with traditional design concepts.

The Test-First Technique

If I were in an elevator, traveling to the top of a building with a software developer I would never see again who had never heard of TDD or the test-first
technique, I would try to teach him the test-first technique. I would choose that because it is so easy to teach and it is so easy to get people to try. Also, if done blindly, it creates problems that will force someone to teach himself test-driven development.

The technique is simple, and the following is often enough to teach it:

1. Write a test.
2. See it fail.
3. Make it pass.
4. Repeat.

There’s nothing more to test-first. There’s a lot more to test-driven development, but test-first really is that simple.

Write the Test

The first step in the technique is to write your test. If you’ve never done this before it might be a little bit uncomfortable at first. You might be thinking “How do I know what to test if there’s nothing there?” That’s a pretty normal feeling that I want you to ball up really tightly and shove down into your gut while you do this a few times. Later you will discover that the best way to determine what should be tested is to write the test for it, but convincing you of that is hard; you’ll have to convince yourself by way of experience.

Anyway, start out by writing a test. Let’s say that I want a database that can store messages sent between users identified by email addresses. The first thing I would do is write a test that requires that ability to be there in order to pass. The test is going to need to create a database of the current version, connect to it, and insert a record. This test is shown in the following listing as one would write it using NUnit and .NET:

```
[TestFixture]
public class TestFirst {  
    private Instantiator instantiator;  
    private IDbConnection connection;

    [SetUp]
    public void EstablishConnectionAndRecycleDatabase() {  
        instantiator = Instantiator.GetInstance(  
            DatabaseDescriptor.LoadFromFile("TestFirstDatabase.xml"));  
        connection = DatabaseProvisioning.CreateFreshDatabaseAndConnect();
    }
```
[TearDown]
public void CloseConnection() {
    connection.Close();
}

[Test]
public void TestTables() {
    instantiator.UpgradeToLatestVersion(connection);
    connection.ExecuteSql("INSERT INTO USERS VALUES(1, 'foo@bar.com')");
    connection.ExecuteSql(@"INSERT INTO MESSAGES " +
                           "VALUES(1, 'Hey!', 'Just checking in to see how it's going.')");
}

That code, as is, won’t compile because I delegate to a little bit of infrastructure that has to be written. One such tool is the DatabaseProvisioning class, which is responsible for creating, tearing down, and connecting to test databases. This class is shown in the following example code, assuming I wanted to test against a SQL Server database:

```csharp
public class DatabaseProvisioning {
    public static IDbConnection CreateFreshDatabaseAndConnect() {
        var connection = new SqlConnection(@"Data Source=.;SqlExpress;" +
                                           "Initial Catalog=master;Integrated Security=True");
        connection.Open();
        connection.ExecuteSql("ALTER DATABASE TDDD_Examples SET " +
                              "SINGLE_USER WITH ROLLBACK IMMEDIATE");
        connection.ExecuteSql("DROP DATABASE TDDD_Examples");
        connection.ExecuteSql("CREATE DATABASE TDDD_Examples");
        connection.ExecuteSql("USE TDDD_Examples");
        return connection;
    }
}
```

The other piece of infrastructure (following) is a small extension class that makes executing SQL statements—something I’m going to be doing a lot in this book—a little easier. For those of you who aren’t C# programmers, what this does is make it look like there is an ExecuteSql method for all instances of IDbConnection.

```csharp
public static class CommandUtilities {
    public static void ExecuteSql(
        this IDbConnection connection, string toExecute) {
        using (var command = connection.CreateCommand()) {
            command.CommandText = toExecute;
```
command.ExecuteNonQuery();

The next step is to see a failure.

**Stub Out Enough to See a Failure**

I like my failures to be interesting. It’s not strictly required, but there’s not a really good reason to avoid it, so assume that making a failure meaningful is implied in “see the test fail.” The main reason you want to see a test fail is because you want to know that it isn’t giving you a false positive. A test that can’t fail for a good reason is about as useful as a test that cannot fail for any reason.

The test I have would fail because there is no database to make, which isn’t a very interesting reason to fail. So let’s create a database class and make it so that the database gets created.

```xml
<Database>
  <Version Number="1">
  </Version>
</Database>
```

With that change in place, my test would fail for an interesting reason: The table into which I was trying to insert doesn’t exist. That’s a meaningful enough failure for me.

**See the Test Pass**

Now that a test is giving me a worthwhile failure, it’s time to make it pass. I do that by changing the class of databases to create the required table. If I had committed the most recent version of the database class to production, I would create a new version to preserve the integrity of my database class. As it stands, because this new database class hasn’t ever been deployed in an irreversible way, I’ll just update the most recent version to do what I want it to do.

```xml
<Database>
  <Version Number="1">
  <Script>
    <! [CDATA[
    CREATE TABLE Users(ID INT PRIMARY KEY, Email NVARCHAR(4000));
    CREATE TABLE Messages(
      UserID INT FOREIGN KEY REFERENCES Users(ID),
      Title NVARCHAR(256),
      Body TEXT);
    ]]>]
  </Script>
  </Version>
</Database>
```
That update causes my database class to create the message table in version 1. When I rerun my test, the database gets rebuilt with the appropriate structures required to make the test pass. Now I’m done with a test-first programming cycle.

**Repeat**

After the cycle is complete, there is an opportunity to start another cycle or to do some other things, such as refactoring. I’m going to go through one cycle just to show you how a design can emerge incrementally. After thinking about the design I created, I decided I don’t like it. I don’t want the email addresses to be duplicated.

How should I handle that? I’ll start by adding a test.

```csharp
[Test]
public void UsersCannotBeDuplicated() {
    instantiator.UpgradeToLatestVersion(connection);
    try {
        connection.ExecuteSql("@" + connection.CreateCommand("INSERT INTO Users(Email) VALUES('foo@bar.com')")");
    } catch {
        return;
    }

    Assert.Fail("Multiple copies of same email were allowed");
}
```

After I get that compiling, I’ll watch it fail. It will fail because I can have as many records with a well-known email address as I want. That’s an interesting failure, so I can go on to the next step: adding the constraint to the new version of my database.

```xml
</Database>
</Version>
</Database>
```

That update causes my database class to create the message table in version 1. When I rerun my test, the database gets rebuilt with the appropriate structures required to make the test pass. Now I’m done with a test-first programming cycle.
Recompiling and rerunning my test shows me that it passes. Had that new behavior caused another test to fail, I would update that test to work with the new design constraint, rerun my tests, and see everything pass. After I’ve done that, I decide I’m done with this phase of updating my database class’s design and move on to other activities.

Tests as Specifications

Another important thing to understand about test-driven development is this simple fact: Tests are specifications. A lot of people make the argument that tests aren’t really tests, but are specifications. Others argue that they aren’t really specifications, but are tests as the name implies.

My position is that both sides of that argument are half right. Tests are specifications. Tests are also tests. The two are not contradictory or even complementary; they are synonymous. What really distinguishes an automated test from other kinds of specifications and other kinds of tests is that it is the automation itself.

“Tests Aren’t Tests, They Are Specifications”

A large group of people exists who frequently tell new developers that tests aren’t really tests, or at least that they don’t start off that way. Tests are specifications and the fact that they also do some testing is just a side effect.

You can get into all kinds of mental gymnastics to justify this argument, and a lot of them have to do with definitions of the words test and specification. The best one I’ve heard is that tests cannot be tests without something to test, so a test is a specification until it passes; then it “falls” into the role of a test later in its life.

In my opinion, terminological correctness is just a device working in service of another motivation. That motivation is that when people think of tests as specifications, they write better tests. Another motivation is to circumvent any preconceived notions a student might have attached to the word test. Both are noble.

The “shock and awe” school of andragogy pulls stunts like this all the time. “To teach, I must first dislodge my student from his mental resting place,”
teachers say. “Otherwise, hysteresis will drag him back to where he started,” they add.

Consider the following code:

```csharp
[Test]
public void BadSpecification() {
    var processor = new Processor();
    Assert.That(processor.Process(-2), Is.EqualTo(-1));
    Assert.That(processor.Process(-1), Is.EqualTo(0));
    Assert.That(processor.Process(0), Is.EqualTo(1));
    Assert.That(processor.Process(5), Is.EqualTo(216));
    Assert.That(processor.Process(25), Is.EqualTo(17576));
}
```

The people in this camp would argue that this test might be succeeding as a test but failing as a specification and, because being a specification is what a test should really do, it is a poorly written test. By contrast, they would argue that the following test is vastly superior because a human reading it could easily understand the rule:

```csharp
[Test]
public void GoodSpecification() {
    var anyInput = 4;
    var processedResult = new Processor().Process(anyInput);
    Assert.That(
        processedResult,
        Is.EqualTo((anyInput + 1) * (anyInput + 1) * (anyInput + 1));
}
```

Let’s hear from the other side of the argument.

“Tests Aren’t Specifications, They Are Tests”

When I first heard someone say that tests aren’t really tests, my knee jerked and I reacted quite badly. I can’t imagine how badly I would have reacted if I didn’t consider that person a friend, but we probably wouldn’t have become friends if that was his first impression of me.

Some people, when they hear something that they believe to be wrong, immediately throw what they already think they know at the problem to see whether it goes away—and that’s exactly what I did. “No way,” I thought. “Tests aren’t specifications. They are obviously tests. That’s why we call them tests. That’s why we see them fail.”

The old me would have looked at the test with the formula and said it was a bad test because making it pass without really implementing the right rule was
easy. Old me also would have said the test with many examples and no explanation of what the rule is was a good test because it forced my production code to do what it really should do.

**Tests Are Executable Specifications**

The problem was that I was doing exactly what my teacher didn’t want. I was clinging to a preconceived notion and not hearing what he was trying to tell me. It was a reaction to something I knew was not right but it was still holding me back.

Each camp is half right.

The kinds of tests you write in test-driven development are not distinct because they are specifications. Nor are they distinct because they are tests. Programmers have been creating both of those artifacts for, literally, generations. The interesting new bit about TDD is that it produces executable specifications.

The process produces specifications that, by definition, must be precise enough to be run frequently by a machine and, consequently, are forced to always stay up to date. That’s what makes TDD so powerful and that is why, when you have a suite of tests that hasn’t been run for any significant amount of time, an enormous amount of work typically has to be done to make it useful.

Keep in mind that a test is a specification and a test also provides guidance on how to make better tests. If each side of the argument said that one of the two tests I showed earlier was better than the other and each side is half right, then what’s the right answer in the contest between those tests?

The right answer is, “Both of those tests have good things about them but neither is the better test.” Instead of choosing one, make a test that clearly specifies the rule but also cannot easily be cheated. One option is to randomly select a number for the test that uses a formula.

```csharp
private int AnyInteger() {
    return new Random().Next(0, 1000);
}

[Test]
public void GoodSpecification() {
    var anyInput = AnyInteger();
    var processedResult = new Processor().Process(anyInput);

    Assert.That(
        processedResult,
        Is.EqualTo((anyInput + 1)*(anyInput + 1) * (anyInput + 1)));
}
```
Another option is to factor the formula test out into a method and then execute that method with several concrete values.

```csharp
private void RunSpecification(int anyInput) {
    var processedResult = new Processor().Process(anyInput);

    Assert.That(
        processedResult,
        Is.EqualTo((anyInput + 1) * (anyInput + 1) * (anyInput + 1))
    ;
}

[Test]
public void GoodSpecificationWithExamples() {
    RunSpecification(-2);
    RunSpecification(-1);
    RunSpecification(0);
    RunSpecification(1);
    RunSpecification(5);
    RunSpecification(25);
}
```

I tend toward the former option and, when I’m doing middle-tier development, I don’t much care which option a person chooses because they are both alright and they are both better than the two earlier options offered by thinking of tests exclusively as tests or as specifications.

### Incremental Design

A side-effect of test-driven development is that it enables you to work in an incremental fashion regardless of the kind of process you use to regulate work in your organization (for example, Scrum or Waterfall).

Every time you write a test, you extend the body of specifications defining what your software should do a little bit. To make that test pass, you have to change your product’s behavior or design slightly. Before you can start working on the next tiny piece of your product, you have to make all your tests pass.

As a result, test-driven development has the effect of focusing work, driving you to extend your software a little bit at a time, while keeping all the existing features working. In short: It imposes a little bit of agility on your process regardless of organizational constraints.
Advantages of TDD

Numerous other benefits and aspects of test-driven development exist that aren’t covered in this book. They are valuable and important for you to learn, but outside the scope of this book. Numerous resources already explore those advantages and, if you are interested, you should probably use them to research the topic on your own.

Building Good Specifications

You could specify many different kinds of things with tests in any given software development endeavor. You could specify structures, public interfaces or private constructs, or what’s in a class. In database terms, you could specify tables, views, and stored procedures.

A test should specify behavior, but should not specify structure. The more behavior-focused a test is, the better off you will be because structures tend to change a lot more quickly than behaviors. This is true even in the database world where, frankly, the pace of change is nigh unto glacial. If you object to my use of the word *quickly*, you can think of it this way: Structures change a lot less slowly than behaviors in a database design.

However, tests have to couple to something in order to invoke the behaviors they define. In fact, many structure decisions are involved in making a test pass. The key is to drive those design decisions into a class of databases from the outside, not the other way around.

Specify Behavior, Not Structure

The odds that you are not a software developer are extremely low. My suspicion is that many of the readers of this book are accomplished computer programmers who also do database work and want to learn how to do what they already know how to do in a database domain. You might also be someone who works only or primarily on databases.

The chance also exists that you are an extraterrestrial archeologist sifting through the intellectual ruins of a species long-since turned to dust. If so, I hope I just sent a shiver up whatever your equivalent of a spine is. Also: Hello, and sorry we didn’t survive long enough for our paths to cross—unless you exterminated us, in which case I’m sorry our paths crossed and I hope you caught some horrible disease from us in the process.
Database programmers and application programmers are both still programmers. Both groups are responsible for writing software, which itself is an act of prescribing behaviors and relationships. In the case of object-oriented programming, what those things mean is pretty clear. At least, it is pretty clear now; it might not have been decades ago.

In the case of database development, it’s a little less intuitive what the behaviors being defined are. People often want to think of databases as collections of tables and relationships. The good thing about that is the focus on a database’s primary responsibility: storing stuff. Yet, it’s still a structure-oriented way of considering design.

A table is a bundle of related features tied to a kind of data. The two basic behaviors a table supports are data manipulation and data querying. Other structures carry with them other behaviors and certain platforms offer extra behaviors with various structures.

Those are what you should specify in tests. Don’t specify that there is a table. Specify that you can store and retrieve data of an interesting sort. Don’t specify that there is a view; specify that you can perform useful searches across bodies of data. That decision might seem meaningless now, but as the book proceeds it will become more and more valuable to you.

**Drive Design In from Without, Not the Other Way Around**

In the procedural days, entities were just data—purely passive things subject to the whims of whatever function might have access to them. With the advent of object-oriented design, they became reactive things that told the world what they could do and then waited for instructions. Modern development practices make classes of objects into servants, told what they should be able to do by tests and then told to do it by other objects and, ultimately, by people.

When you write a test, you want it to specify the behaviors that live in a class of databases, but it’s going to have to talk to something to do that. An implication of specifying a behavior is that you must also specify the minimal amount of public interface required to invoke that behavior. The key to discovering that is to learn it from writing tests first.

Let’s consider a problem. Imagine I need to write an application that keeps a database of streets and cross references them with other intersecting streets. I could drive the requirements from tests, specifying behaviors and required interface, or I could define my design inside out—starting with capabilities, then building an interface around it. I’ll try the latter first.
Defining the Design Inside Out

Well, the obvious thing I need is a table in which to store streets. So let's start there (see Figure 3.1).

<table>
<thead>
<tr>
<th>Streets</th>
</tr>
</thead>
<tbody>
<tr>
<td>PK ID</td>
</tr>
<tr>
<td>Name</td>
</tr>
</tbody>
</table>

**Figure 3.1 Simple design**

Of course, streets exist in cities, so I need a cities table. Maybe later I'll need a states table, too, but for now, I can live without it. Let's add a cities table with a state column so I can track which street I am dealing with (see Figure 3.2).

<table>
<thead>
<tr>
<th>Streets</th>
</tr>
</thead>
<tbody>
<tr>
<td>PK ID</td>
</tr>
<tr>
<td>FK1 CityID</td>
</tr>
<tr>
<td>Name</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cities</th>
</tr>
</thead>
<tbody>
<tr>
<td>PK ID</td>
</tr>
<tr>
<td>Name</td>
</tr>
<tr>
<td>State</td>
</tr>
</tbody>
</table>

**Figure 3.2 Streets segregated by city**

Some streets span many cities, such as highways and interstate freeways. So I need to account for those, too (see Figure 3.3).

Now there's the fact that I need to track the intersections, so let's add that. It seems like it should be a cross-reference table with the address on each street at which the intersection takes place. Because streets sometimes cross in multiple places, I need a primary key that is distinct from the foreign keys on that table so I can support multiple links, as shown in Figure 3.4.
From there, I can start hypothesizing how the data might be used, adding views and stored procedures to support those needs. Then, I could write tests for all the behaviors I developed. Eventually, I’ll think I have enough to start writing an application.

Of course, I won’t.

For one thing, there is a distinct database for every city supported by the application. So, every application is encumbered by adding noise structures. The Cities and StreetToCityLinks tables are completely unnecessary as are the constraints surrounding them.

Also, the application doesn’t care where two streets connect, only that they connect. So, the Street1Address and Street2Address fields of the Intersections table...
table serve no purpose but to waste the time of everyone who touches them or reads about them.

**Defining the Design Outside In**

What if I try going the other direction? Suppose I want to start at the outside and work my way inward. In that event, by the time I’m defining a database design, I probably would have written the user interface and application logic already.

Having done those things would provide me with context and understanding as to what was really needed. If I work exclusively with the database, then someone else would provide the context for me and I would have a very clear idea of what the requirements are.

Either way, that understanding would be something that could be translated into tests as in the following:

```csharp
[Test]
public void CreateAndFindStreet() {
    connection.ExecuteSql("INSERT INTO Streets VALUES(5, 'Fun St.')");

    var id = connection.ExecuteScalar(
        "SELECT ID FROM Streets WHERE NAME LIKE '%Fun%'");

    Assert.That(id, Is.EqualTo(5));
}
```

That test would drive me to build a database class as follows:

```xml
<Database>
    <Version Number="1">
        <Script>
            <![CDATA[
                CREATE TABLE Streets(ID INT PRIMARY KEY, NAME NVARCHAR(4000))
            ]]></Script>
    </Version>
</Database>
```

Knowing that I also needed the capability to find related streets, I might write another test as follows:

```csharp
[Test]
public void CreateConnectedStreetsAndFindFewestIntersectionsConnected() {
    connection.ExecuteSql("INSERT INTO Streets VALUES(1, 'A St.')");
    connection.ExecuteSql("INSERT INTO Streets VALUES(2, 'B Dr.')");
    connection.ExecuteSql("INSERT INTO Streets VALUES(3, 'C Ave.')");
    connection.ExecuteSql("INSERT INTO Streets VALUES(4, 'D Ln.')");
    connection.ExecuteSql("INSERT INTO Streets VALUES(5, 'E Blvd.')");
```
That test would drive me to develop the design in the next snippet:

```xml
<Database>
  <Version Number="1">
    <Script>
      <![CDATA[
CREATE TABLE Streets(ID INT PRIMARY KEY, NAME NVARCHAR(4000))
CREATE TABLE Intersections([ID] INT PRIMARY KEY)
CREATE TABLE IntersectionStreets(
    [IntersectionID] INT FOREIGN KEY REFERENCES Intersections(ID),
    [StreetID] INT FOREIGN KEY REFERENCES Streets(ID))
]]>
    </Script>
    <Script>
      <![CDATA[
CREATE VIEW ImmediateConnections AS
SELECT s.StreetID AS StartID, e.StreetID AS EndID
FROM IntersectionStreets AS s
INNER JOIN IntersectionStreets AS e
ON s.IntersectionID = e.IntersectionID and s.StreetID <> e.StreetID
]]>
    </Script>
    <Script>
      <![CDATA[
CREATE FUNCTION Connections
(
)
RETURNS @Result TABLE (Depth INT, StartID INT, EndID INT)
AS
BEGIN
  DECLARE @Temp TABLE (StartID INT, EndID INT)
  DECLARE @Depth INT
  SET @Depth = 0
```
Note how narrow and focused the interface for the database that was designed outside-in is compared to the one that was designed inside-out. Yet, in certain areas such as the recursive view, the behavior is much deeper than with the inside-out design. The two side-effects of driving design into a system rather than designing a system and making clients find a way to use it are that you write something that can actually be used, and you spend more of your time developing worthwhile functionality.

**Summary**

A distinction exists between test-first programming and test-driven development. The former is an easy practice to convey whereas the latter is a hard discipline to learn. Test-first is, however, a stepping stone that helps you get to test-driven development.

TDD is more than just getting good specifications in place that happen to be tests. It is also more than just getting good tests in place that happen to be specifications. It is about building executable specifications. That is, it is about creating documents that are both tests and specifications to such a degree of quality that you don’t need any other documents to do either of those jobs.

Test-driven development has a lot more to it, but this chapter should give you the context you need to get started. Throughout the remainder of the book,
remember these things: Try to specify behaviors in tests before implementing them, and grow your designs inward from the point at which a test couples to what it tests.

The next step is to put in place structures that allow you to change your designs with great confidence, especially with regard to the safety of production data.
This page intentionally left blank
Index

A

abstraction, 160
  composition type relationships, 178-179
  database classes, linking, 178-179
  dependencies, allowing variation in, 185-186
  implementation and interface, synchronizing, 186-189
  low-risk refactoring operations, 222-223
  access to façade database, removing, 263-264
  advantages of TDD, 27
  aggregation, 160, 167, 172-174
    mocking, 203, 210
    reuse, 177
  allowing variation in dependencies, 185-186
  applications, coupling to database instances, 66
  applying
    changes to incremental builds, 16
    façade pattern to legacy databases, 254-261
      old interface, strangling, 262-264
    patches, 274-281
  linear growth pattern of database class, rejoining, 275-281
  resulting variation, limiting, 277
  transition testing, 277-281
  safeguards to upgrades, 60
  TDD
    to data objects, 292-300
    to databases, challenges in, 3
    to file systems, 288-291
    to XML, 284-288
  assembly language, suitability for TDD, 160-162
  auditing current uses of legacy databases, 232-233
  avoiding requirements forecasting, 111-112

B

backups, importance of, 156
bad errors
  released errors, 150-157
    documenting, 154-157
  unreleased errors, 147-150
behaviors, 102-106
  controlling through mocking, 194-195
defining for outside-in development, 109-110
desirable errors, testing for, 143-145
inside-out design, defining, 30-32
irrelevant behaviors, isolating from tests, 194
knowledge, protecting integrity of, 124-126
in legacy databases
legacy behaviors, exposing, 265-266
new behavior, implementing, 239-240
new behaviors, developing in façade database, 266
pinning, 237-239
testing, 235-236
transferring to façade database, 262-263
mapping by dependencies, 166-167
outside-in design, defining, 32-34
pinning, 49
specifying, 28-29, 193
versus structures, 28
table-supported, 29
Bain, Scott, xvii
beneficial defects, tracing history of, 145
benefits of TDD, 3
bottlenecks resulting from business logic, 98
building in increments, 112
business logic, 97-98
--- bottlenecks resulting from, 98
needs interface, 97-98
C
capabilities interface, 97
   converting to needs interface via façade pattern, 250-254
   hiding, 97
catastrophic errors, importance of backups in dealing with, 156
challenges in applying TDD to databases, 3
change-management, effect on maintainability, 124-126
changing existing data structures, 51-56
classes. See also linking through abstraction, 178-179
   in OOP, 11
   role in TDD, 9-11
   separating
      via aggregation, 172-174
      via composition, 168-172
code
   machine code, 159
   references, changing, 82
   reuse, 175-177
committed bugs, destroying, 154-157
comparing
   behaviors and structures, 28
   objects and classes, 10-11
   strong and weak coupling languages, 66
   TDD and object-oriented development, 2, 4
components
   encapsulating, 243-247
   locating, 242-243
composition, 160, 167-172
mocking, 203, 208-210
reuse, 177
concepts, dividing into buckets, 163
construction behaviors in legacy databases, pinning, 237-238
construction logic
  of data objects, testing, 294-296
  dividing into transitions, 163
constructors, 10
controlling
  behaviors through mocking, 194-195
legacy database coupling
  encapsulating on demand, 234
  existing uses of, auditing, 232-233
  permissions, locking down, 233-234
converting
  capabilities interface to needs interface via façade pattern, 250-254
legacy databases to a class, 229-230
coupling
  applications to database instances, 66
  enforcing, 68
error handling, 71
intermediate formats, creating for multiple platform clients, 70
to legacy databases, controlling, 231-234
  encapsulating on demand, 234
  existing uses of, auditing, 232-233
  permissions, locking down, 233-234
lens concept, 83-85
  virtual lenses, 85-89
logical versions, 93-94
to needs interface, 97
seams, 227
  locating, 240-243
validating, 63
creating intermediate formats for multiple platform clients, 70
CRUD (create, read, update, and delete), 97
current database version, identifying, 16

D

data objects, TDD applications, 292-300
  class definitions, 292-293
  transition testing, 294-296
  transitions, coding, 296-300
database classes
  comparing to objects, 10-11
coupling classes
  documenting, 74
  version-specific, 77-78
database coupling, 67-69
DatabaseDesign class
  creating demand for, 67-68
  specifying, 68-69
high-risk refactoring operations, 225
importance of, 270
legacy databases, promoting to, 228-231
low-risk refactoring operations, 222-223
medium-risk refactoring operations, 223-225
modification, 12-13
database development
incremental builds
changes, applying, 16
changes, documenting, 15
current version, identifying, 16
versus object-oriented
development, 2
DatabaseDesign class
creating demand for, 67-68
specifying, 68-69
databases
behaviors, 102-106. See also behaviors
business logic, 97-98
classes
implementing, 16-17
modification, 12-13
constructors, 44-56
coupling, 63
to applications, 66
enforcing, 68
design of
avoiding requirements
forecasting, 111-112
change-management effect on, 124-126
CRUD-based, 100
information-based, 117-120
internal structures, hiding, 112-113
new features, building support for, 128-129
without encapsulation, 96
IcecreamSales database, 41
information, 99-101
instantiation
pseudocode input, 17
pseudocode mechanism, 17
requirements for, 16-17
interfaces
capabilities interface, 97
duplication, 64
needs interface, 97
knowledge, 102
knowledge samples, validating, 59
legacy databases
components, encapsulating, 243-247
components, locating, 242-243
coupling, controlling, 231-234
existing uses of, auditing, 232-233
façade databases, testing, 254-261
façade pattern, 249
interface, creating, 234
permissions, locking down, 233-234
promoting to a class, 228-231
seams, locating, 240-243
as objects, 5
production databases, 13-14
proxies, testing, 6
tables, 29
TDD, difficulty in applying, 3
transition safeguards, 56-61
decoupling
implementation from design, 72-73
in OOP mocking, 199-202
defining
  external interface through information, 100
  inside-out design, 30-32
  outside-in design, 32-34
deleing unit tests, 154-157
delivering features incrementally, 116-117
dependencies
  allowing variation in, 185-186
  isolating from behaviors, 202
  mapping behaviors by, 166-167
design principles, 160
  abstraction, 178-189
    dependencies, allowing variation in, 185-186
  implementation and interface, synchronizing, 186-189
  aggregation, 167, 172-174
  composition, 167-172
  encapsulation, 163
    applying to XML documents, 284
    mapping behaviors by dependencies, 166-167
mocking, 203-210
  aggregation designs, 210
  composition designs, 208-210
  problem example, 204-205
  solution example, 205-208
refactoring, 213
  high-risk refactoring operations, 225
  low-risk refactoring operations, 222-223
medium-risk refactoring operations, 223-225
  role of tests in, 215-222
  reuse, 175-177
  risks in changing design, 213
desirable errors, documenting, 143-145
desirable errors, testing for, 145
desirable errors, testing for, 145
destroying committed bugs, 154-157
developing new behaviors in façade database, 266
development
  inside-out, 30-32
  outside-in, 32-34
    defining the test, 106-107
    interface, adding, 108-109
difficulty in applying TDD to databases, 3
dividing
  concepts into buckets, 163
  construction logic into transitions, 163
documenting
  changes in incremental builds, 15
  coupling classes for versions, 74
desirable errors, 143-145
  released errors, 154-157
duplication
  between database build and client code, eliminating, 71-72
  lens concept, 83-85
    virtual lenses, 85-89
  medium-risk refactoring operations, 223-225
Shalloway’s Law, 82
E

eliminating
duplication between database build and client code, 71-72
unit tests, 154-157
encapsulation
aggregation, 160, 167, 172-174
reuse, 177
applying to XML documents, 284
behaviors
hiding via abstraction, 179-185
mapping by dependencies, 166-167
composition, 160, 167-172
reuse, 177
in database design without encapsulation, 96
encapsulation of variation, 159
via abstraction, 160
façade pattern. See also façade pattern
capabilities interface, converting to needs interface, 250-254
old interface, strangling, 262-264
legacy database components, 243-247
needs interface, 97
encapsulation of variation, via abstraction, 160
enforcing
database coupling, 68
parity between first- and second-class designs, 270-273
error, types, 137
error handling
backups, importance of, 156
bad errors
released errors, 150-157
unreleased errors, 147-150
good errors
documenting, 143-145
tracing history of, 145
released errors, 150
documenting, 154-157
establishing maintainability of legacy databases, 231
evolution of related classes, synchronizing, 186-189
executable specifications as tests, 26-27
existing data structures, changing, 51-56
exposing behaviors
legacy behaviors in façade database, 265-266
with stored procedures, 121-124
external interface, defining in terms of information, 100

F

façade pattern, 249
applying to legacy databases
changing behaviors, transferring to façade database, 262-263
legacy behaviors, exposing, 265-266
new behaviors, developing, 266
old interface, strangling, 262-264
permissions, removing, 263-264
capabilities interface, converting to needs interface, 250-254
testing, 254-261
failure, observing
  pinning, 49
  test-first technique, 22
transition testing, 46-47
features
  adding to legacy databases, 234
  building support for, 128-133
delivering incrementally, 116-117
  including only current requirements, 111-112
  removing from façade database, 263-264
testing independently, 166
file systems, TDD applications, 288-291
  shell script transitions, 291
  transition testing, 289-291
finding seams, 240-243
first-class designs, enforcing parity
  with second-class designs, 270-273
fixing
  released errors, 150-157
  unreleased errors, 147-150
flexibility of design, achieving through abstraction, 179-185
Four Cs, 16-17
functionality of stored procedures, specifying, 131-132

G
Gang of Four, 117
good errors, 142
documenting, 143-145
  tracing history of, 145

H
handling coupling errors, 71
hiding
  behaviors through abstraction, 179-185
  tables through stored procedures, 119-120
hiding internal structures, 112-113
high-risk refactoring operations, 225
history of good errors, tracing, 145

I
IcecreamSales database, 41-43
identifying current database version, 16
implementation class, synchronizing with interface class, 186-189
implementing
  database instantiation, requirements, 16-17
  new behavior in legacy databases, 239-240
importance of classes, 270
incremental builds, 112
  changes, applying to, 16
  changes, documenting, 15
  current version, identifying, 16
Infinite Insights into Kenpo (Parker), 287
information and knowledge,
  translating between via behaviors, 102-106
information-based interface, 101
inside-out design, defining, 30-32
integration testing, 202-203
interfaces
  capabilities interface, 97
    converting to needs interface via façade pattern, 250-254
duplication, 64
external interface, defining through information, 100
inside-out design, 30-32
legacy database interface
  creating, 234
    old interface, strangling, 262-264
needs interface, 97
outside-in design, 32-34
for outside-in development, 108-109
synchronizing with implementation, 186-189
intermediate formats, creating, 70
internal structures
  hiding, 112-113
  testing, 46-47
irrelevant behaviors, isolating from tests, 194
isolating
  behaviors from dependencies, 202
  irrelevant behaviors from tests, 194

J-K
knowledge
  high-risk refactoring operations, 225
  versus information, 102
  protecting integrity of, 124-126
tables, 102

L
legacy databases
  behaviors
    legacy behaviors, exposing, 265-266
    new behavior, implementing, 239-240
    pinning, 237-239
    testing, 235-236
  components
    encapsulating, 243-247
    locating, 242-243
    coupling, controlling, 231-234
    decomposing into components, 227
    existing uses of, auditing, 232-233
    façade databases, 249
      features and permissions, removing, 263-264
      testing, 254-261
      transferring changing behaviors to, 262-263
    interface, creating, 234
      permissions, locking down, 233-234
      promoting to a class, 228-231
    seams, locating, 240-243
  lens concept, 83-85
    current version lens, 89
    new version lens, 89-93
    virtual lenses, 85-89
  leveraging transition safeguards, 60-61
  limiting variation from patches, 277
  line items, separating from transactions in tables, 41-43
  linear growth pattern of database class, rejoining patches to, 275-281
linking database classes through abstraction, 178-179
locating components, 242-243
seams, 240-243
locking down legacy database permissions, 233-234
logical databases, composition, 168-172
logical versions, coupling to, 93-94
low-risk refactoring operations, 222-223

M
machine code, 159
maintainability, 115
change-management effect on, 124-126
designing to information, 117-120
of legacy databases, establishing, 231
new features, building support for, 128-129
refactoring, 133-136
through minimal database design, 115
translating information and knowledge with behaviors, 121-124
mapping behaviors by dependencies, 166-167
between need and capability, 97
Martin, Robert, 214
medium-risk refactoring operations, 223-225

minimal database design, 115
mocking behaviors
controlling, 194-195
encapsulation, 192
irrelevant behaviors, isolating from tests, 194
specifying, 193
in database design
aggregation designs, 203, 210
composition designs, 203, 208-210
problem example, 204-205
solution example, 205-208
in OOP, 195-203
behaviors, isolating from their dependencies, 202
decoupling, 199-202
integration testing, 202-203
setup, 195-199
modification of database classes, 12-13
multiple client platforms, creating intermediate formats, 70

N
naming conventions, importance of, 133
natural versus overdesigned databases, 126-127
needs interface, 97
versus capabilities interface, 97
coupling to, 97
creating from capabilities interface via façade pattern, 250-254
O

objects
  comparing to classes, 10-11
databases as, 5
reuse, 175
observing test-first technique outcomes
  failure, 22
  success, 22-23
OOP (object-oriented programming)
  versus database development, 2
mocking, 195-203
  behaviors, isolating from their dependencies, 202
decoupling, 199-202
integration testing, 202-203
setup, 195-199
outside-in design
  behaviors, defining, 109-110
defining, 32-34
defining the test, 106-107
interface, adding, 108-109
structures, defining, 109-110
overdesigning databases, 126-127

permissions
  locking down, 233-234
  removing from façade database, 263-264
pinning, 49
  desirable errors, documenting, 143-145
  legacy database behaviors, 237-239
  released errors, fixing, 153-154
political barriers to effective design, 124-126
preserving database knowledge, transition safeguards, 56-61
production databases, 13-14
  as model for all databases, 14
“Production Promote,” 39-40
programming
  strong coupling languages, 64-65
  weak coupling languages, 65-66
programming languages
  assembly language, 160
  suitability for TDD, 159
promoting legacy databases to a class, 228-231
  script, creating, 229-230
  transition behavior, pinning with tests, 231
proxies, testing, 6
pseudocode input for database instantiation, 17
public interfaces
  duplication, 64
  separating from private implementation, 102
parity, enforcing between first- and second-class designs, 270-273
Parker, Ed, 287
patches
  linear growth pattern of database class, rejoining, 275-281
  resulting variation, limiting, 277
  transition testing, 277-281
Q-R

read/read transition testing, 56-59
refactoring, 133-136
  risks in changing database
deresign, 213
  high-risk refactoring
  operations, 225
  low-risk operations, 222-223
  medium-risk refactoring
  operations, 223-225
role of tests in, 215-222
relationships
  adding to transition tests, 50-51
  aggregation relationships, 173
  behaviors, mapping by
dependencies, 164-167
components
  encapsulating, 243-247
  locating, 242-243
  defining, suitability of programming
  languages in, 162
released errors, 150-157
  documenting, 154-157
removing
  permissions in façade database,
263-264
  unit tests, 154-157
repairing
  released errors, 150-157
  unreleased errors, 147-150
requirements
  for database instantiation, 16-17
  forecasting, avoiding, 111-112
reuse
  of design and instances, 175
  as primary benefit of object-
oriented development, 160
rewriting design updates, 75-77,
261-262
risk-free design, 38
risks
  of breaking contracts, 38-39
  in changing database design, 213
  low-risk refactoring operations,
222-223
  medium-risk refactoring
  operations, 223-225
  of changing structures, 37
role of classes in TDD, 9-11
role of tests in refactoring, 215-222
rollbacks, performing on fail, 60

S
safeguards
  adding to transition tests, 56-59
  applying to upgrades, 60
sampling, 58
  knowledge samples, validating, 59
scripts, transforming legacy databases
to a class, 229-230
seams, 227
  locating, 240-243
second-class designs, enforcing parity
with first-class designs, 270-273
security, locking down permissions,
233-234
separating
  behaviors by commonality, 165-167
database classes
  through aggregation, 172-174
  through composition, 168-172
line items from transactions in tables, 41-43
Shalloway’s Law, 82
shell script transitions, 291
software industry, testing in, 5-6
specifications
  behaviors
    specifying, 28-29
    versus structures, 28
incremental design, 27
inside-out design, defining, 30-32
outside-in design, defining, 32-34
tests as, 24-27
specifying
  behaviors, 193
DatabaseDesign class, 68-69
stored procedure functionality, 131-132
SQL Server
  promoting legacy databases to a class, 229-230
stored procedures
  behaviors, exposing, 121-124
  hiding tables through, 119-120
  needs interface, exposing, 97
  specifying functionality, 131-132
  updating for new versions, 134-135
strangling legacy database interface, 262-264
strong coupling languages, 64-65
structures
  versus behaviors, 28
  versus design, 160
as cause for released errors, 154-157
changing, risks of, 37
defining for outside-in development, 109-110
internal structures
  hiding, 112-113
  testing, 46-47
success, observing
  desirable errors, testing for, 144-145
  test-first technique, 22-23
suitability of programming languages
  in defining relationships, 162
  for TDD, 159
supporting new features, 128-129
synchronizing
  designs through reuse, 175-176
  implementation and interface, 186-189
T
tables, 29
  supported behavior, 29
  through stored procedures, 119-120
  transactions, separating from line items, 41-43
  as unit of knowledge, 102
target audience for this book, 3-4
TDD (test-driven development)
  applying
    to data objects, 292-300
    to file systems, 289-291
    to XML, 284-288
  benefits of, 3
classes, role in, 9-11
encapsulation
    aggregation, 160
    composition, 160
encapsulation of variation, 159
incremental design, 27
versus object-oriented
development, 2
suitability of programming
languages for, 159
test-first technique, 19-24
    failure, observing, 22
    repeating the process, 23-24
    success, observing, 22-23
    writing the test, 20-22
testing as executable specifications, 26-27
test-first technique, 19-24
    failure, observing, 22
    repeating the process, 23-24
    success, observing, 22-23
    writing the test, 20-22
testing behaviors
    irrelevant behaviors, isolating from tests, 194
    in legacy databases, 235-236
    specifying, 28-29
for desirable errors, 143-145
as executable specifications, 19, 26-27
façade databases, 254-261
integration testing, 202-203
internal structures, 46-47
in outside-in development
test, defining, 106-107
proxies, 6
role in refactoring, 215-222
in software industry, 5-6
as specifications, 24-27
transition safeguards, 56-61
    rollbacks, 60
transition testing, 37, 44-56
    data objects, 294-296
    desirable errors, testing for, 145
    file systems, 289-291
    read/read transition testing, 56-59
    relationships, adding, 50-51
    transition safeguards, leveraging, 60-61
    XSLT changes, 287-288
unit testing, 67-73
    DatabaseDesign class, 67-68
    decoupling implementation from design, 72-73
    deleting tests on purpose, 154-157
for unreleased errors, 147-150
thedailywtf.com, 39-40
tracing history of good errors, 145
transactions, separating from line items in tables, 41-43
transferring changing behaviors to façade database, 262-263
transition safeguards, 56-61
    leveraging, 60-61
    rollbacks, 60
    sampling, 58
transition testing, 37, 44-56. See also transition safeguards
    data objects, 294-296
    desirable errors, testing for, 145
    existing data structures, changing, 51-56
failure, observing, 46-47
file systems, 289-291
internal structures, testing, 46-47
for patches, 277-281
pinning, 49
released errors, fixing, 153-154
read/read transition testing, 56-59
relationships, adding, 50-51
sampling, 58
XSLT changes, 287-288
transitions, 214
translating information and knowledge
with behaviors, 121-124

U
database design class, creating
demand for, 67-68
decoupling implementation from
design, 72-73
deleting tests on purpose, 154-157
pinning, documenting desirable
errors, 143-145
unreleased errors, fixing, 147-150
unnecessary structures, removing,
154-157
unreleased errors, 147-150
updating
for new design versions, 75-77
new features, building support for,
128-133
with patches, 274-281
linear growth pattern of database
class, rejoining, 275-281
resulting variation, limiting, 277
transition testing, 277-281
rewrites, 261-262
upgrades
documenting, 75-77
safeguards, applying, 60

V-W
validating
coupling, 63
knowledge samples, 59
verifying refactors, 214
versions
coupling classes, documenting, 74
current version lens, 89
documenting, 75-77
in incremental builds, 15
enforcing parity between, 270-273
implementation and interface,
synchronizing, 186-189
logical versions, coupling to, 93-94
new version lens, 89-93
patches, applying to, 274-281
stored procedures, updating,
134-135
version-specific coupling classes, 77-78
virtual lenses, 85-89
weak coupling languages, 65-66
writing the test (test-first technique),
20-22
X-Y-Z

XML, TDD applications, 284-288
  encapsulation, 284
  XSD schemas, 284-286
  XSLT transformations, 286-288
XSD schemas, applying TDD to XML,
  284-286
XSLT transformations, 286-288