TEST-DRIVEN DATABASE DEVELOPMENT

Unlocking Agility

MAX GUERNSEY, III

Foreword by SCOTT BAIN

Net Objectives

Lean-Agile Series

Test-Driven Database Development

Test-Driven Database Development

Unlocking Agility

Max Guernsey, III

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Cape Town • Sydney • Tokyo • Singapore • Mexico City

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States, please contact:

International Sales
international@pearsoned.com

Visit us on the Web: informit.com/aw

Library of Congress Cataloging-in-Publication Data

Guernsey, Max, 1978-
Test-driven database development : unlocking agility / Max Guernsey.
pages cm
Includes bibliographical references and index.
ISBN-13: 978-0-321-78412-4 (pbk. : alk. paper)
ISBN-10: 0-321-78412-X (pbk. : alk. paper)
1. Database design. 2. Agile software development. I. Title.
QA76.9.D26G84 2013
005.1--dc23

2012047608

ISBN-13: 978-0-32-178412-4
ISBN-10: 0-32-178412-X

Text printed in the United States on recycled paper at RR Donnelley and Sons in Crawfordsville, Indiana.

First printing, February 2013

This book is dedicated to my wife, Amy Bingham,
who is largely responsible for my being the man I am today.

This page intentionally left blank

Contents at a Glance

Contents

This page intentionally left blank

Foreword

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.

About the Author

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).

This page intentionally left blank

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:

```
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`.

```
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.

```
<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.

```
<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.

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

```
<Database>
  <Version Number="1">
    <Script>
      <![CDATA[
CREATE TABLE Users(ID INT PRIMARY KEY, Email NVARCHAR(4000));

ALTER TABLE Users ADD CONSTRAINT OnlyOneEmail UNIQUE (Email);

CREATE TABLE Messages(
  UserID INT FOREIGN KEY REFERENCES Users(ID),
```

```
    Title NVARCHAR(256),
    Body TEXT);
  ]]>
      </Script>
    </Version>
  </Database>
```

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:

```
[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:

```
[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.

```
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.

```
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).

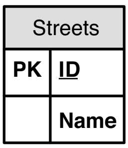

Figure 3.1 *Simple design*

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).

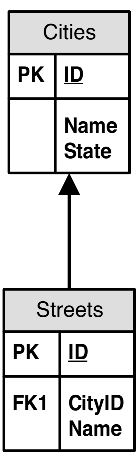

Figure 3.2 *Streets segregated by city*

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.

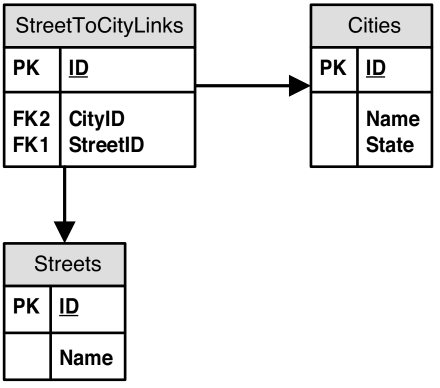

Figure 3.3 *A street going through multiple cities*

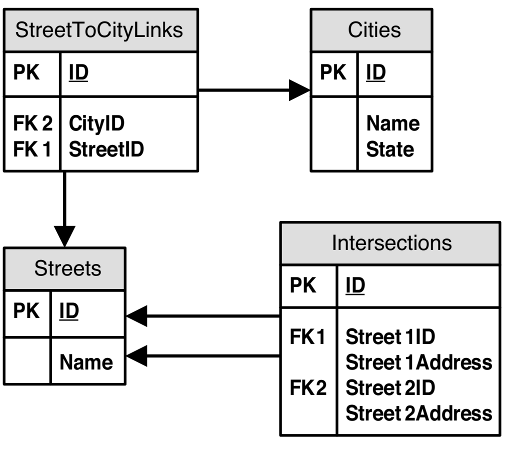

Figure 3.4 *Streets organized by city and cross-referenced by intersection*

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 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:

```
[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:

```
<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:

```
[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.')");
```

```
  connection.ExecuteSql("INSERT INTO Intersections VALUES(1)");
  connection.ExecuteSql("INSERT INTO IntersectionStreets VALUES(1, 1)");
  connection.ExecuteSql("INSERT INTO IntersectionStreets VALUES(1, 2)");

  connection.ExecuteSql("INSERT INTO Intersections VALUES(2)");
  connection.ExecuteSql("INSERT INTO IntersectionStreets VALUES(2, 1)");
  connection.ExecuteSql("INSERT INTO IntersectionStreets VALUES(2, 3)");

  connection.ExecuteSql("INSERT INTO Intersections VALUES(3)");
  connection.ExecuteSql("INSERT INTO IntersectionStreets VALUES(3, 3)");
  connection.ExecuteSql("INSERT INTO IntersectionStreets VALUES(3, 4)");
  var result = connection.ExecuteScalar(
    "SELECT Depth FROM Connections() WHERE StartID = 2 AND EndID = 4");

  Assert.That(result, Is.EqualTo(3));
}
```

That test would drive me to develop the design in the next snippet:

```
<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
```

```
    INSERT INTO @Temp SELECT ID AS StartID, ID AS EndID FROM Streets;

    WHILE EXISTS (SELECT TOP 1 * FROM @Temp)
    BEGIN
      INSERT INTO @Result SELECT @Depth, StartID, EndID FROM @Temp;
      DELETE @Temp;

      INSERT INTO @Temp SELECT r.StartID, ic.EndID FROM @Result AS r
        INNER JOIN ImmediateConnections AS ic ON r.EndID = ic.StartID

      DELETE @Temp FROM @Temp AS tc INNER JOIN @Result AS r
        ON tc.StartID = r.StartID AND tc.EndID = r.EndID

      SET @Depth = @Depth + 1
    END;
  RETURN
  END
        ]]>
      </Script>
    </Version>
  </Database>
```

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

C

D

E

F

G

H

I

J-K

L

M

N

O

P

Q-R

S

T

U

V-W

X-Y-Z