Test-Driven Development of Relational Databases

Scott W. Ambler, *IBM*

Implementing test-driven database design involves database refactoring, regression testing, and continuous integration. Although technically straightforward, TDDD involves cultural challenges that slow its adoption.

In test-first development, developers formulate and implement a detailed design iteratively, one test at a time.[1,2] Test-driven development (also called test-driven design) combines TFD with refactoring,[3] wherein developers make small changes (refactorings) to improve code design without changing the code's semantics. When developers decide to use TDD to implement a new feature, they must first ask whether the current design is the easiest possible design to enable the feature's addition. If so, they implement the feature via TFD. If not, they first refactor the code, then use TFD. In the behavior-driven development[4] approach to TDD, the naming conventions and terminology make the developer's primary goal explicit: to specify, rather than validate, the system.

Test-driven database design applies TDD to database development. In TDDD, developers implement critical behaviors (including business rules about data invariants, such as "The only valid colors are red, blue, and purple") and business functionality within relational databases. There's nothing special about relational databases—we specify database behaviors via database tests in the same way we implement application-code behaviors via developer tests. TDDD is not a stand-alone activity. It's best viewed as simply the database aspects of TDD. That is, you should develop your database schema in lock-step with your application code[5] because sometimes you'll write a developer test that specifies application code behavior, while other times it will specify database behavior.

TDDD is important for several reasons. First, all of application TDD's benefits apply to TDDD: you can take small, safe steps; refactoring lets you maintain high-quality design throughout the life cycle; regression testing lets you detect defects earlier in the life cycle; and, because TDDD gives you an executable system specification, you're motivated to keep it up-to-date (unlike with traditional design documentation). Moreover, with TDD, your database development efforts effectively dovetail into your overall application development efforts. The IT community has long suffered because of the cultural mismatch between application developers and data professionals.[5] This mismatch results largely from differing philosophies and ways of working. Modern methodologies, in-

0740-7459/07/$25.00 © 2007 IEEE

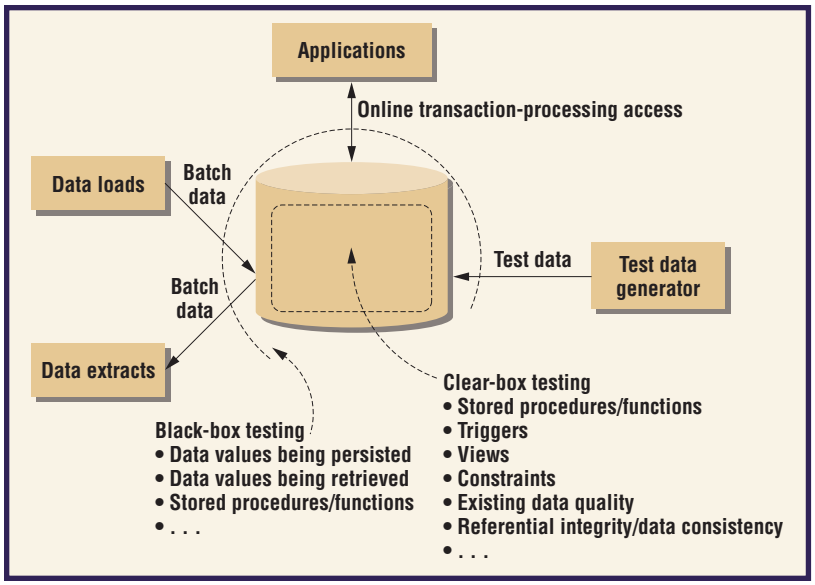

Figure 1. Testing a relational database. Developers must test the database both internally and at its interface—the equivalent of clear-box testing and black-box testing, respectively.

cluding Rational Unified Process, Extreme Programming, and Dynamic System Development Method, work in an evolutionary (iterative and incremental) manner. It therefore behooves us to find techniques that let data professionals work in an evolutionary manner with their application developer counterparts. TDDD is one such technique.

Extending TDD to database development

To extend TDD to database development, we need database equivalents of regression testing, refactoring, and continuous integration.

Database regression testing

In database regression testing,[6,7] you regularly validate the database by running a comprehensive test suite—ideally, whenever you change the database schema itself or access the database in a new way. As the threat boundaries[8] (dashed lines) in figure 1 show, you must test both the database interface and the database itself.

Interface testing. We use database interface tests to specify how systems will access the database. From the database's viewpoint, these are effectively black-box tests. Such a test might specify a portion of a hard-coded SQL statement that you might submit to the database. It might also specify that

- an SQL statement returns an expected result,
- a view produces an expected result,
- SQL statement calculations return the expected results,
- a particular user ID can't access specific information, or
- a particular user ID can access specific information.

Interface tests are fairly straightforward to implement because good tools exist for many platforms, especially the xUnit framework. Commercial tools are also viable options. My advice: use the same tools that you're using for application regression testing.

Internal testing. Internal database tests specify both behaviors that the database implements and data validity. There's nothing special about stored procedure code—if you can write tests for Java code, surely you can write tests for procedural language extensions to SQL (PLSQL) code. According to current rhetoric, data is a corporate asset—shouldn't you therefore specify what kind of data you intend to store, along with the data's invariants? From the database's viewpoint, these are effectively clear-box tests. Such tests might specify

- some of a stored procedure's behavior,
- a validity check for a parameter being passed to a stored procedure,
- a validity check for the value that a stored procedure returns,
- a stored procedure to return an expected error code,
- a column invariant,
- a referential integrity rule, and
- a column default.

Tools for internal database testing are a bit harder to come by, but that's changing. For example, tools such as Quest's Qute validate Oracle PLSQL code and Microsoft's Visual Studio Team System for Database Professionals includes testing tools for SQL Server.

Testing example. The following example illustrates regression testing. Assume that we're building a banking system and have two features to implement: Retrieve Account from Database and Save Account to Database. Let's also assume that this is the first time we've run into the Account concept and that we're implementing these two features. For account retrieval, we'd run several database tests (one at a

time), including shouldExistOneOrMoreAccountsForCustomer, shouldExistOneOrMoreCustomersForAccount, shouldHaveUniqueAccountID, and shouldHavePositiveAccountBalance. For saving an account into the database, we'd add two new tests: shouldBeAssignedUniqueAccountIDOnCreation and shouldBeAssignedZeroBalanceOnCreation. I'm using the BBD "should" naming conventions[4] here, but the "traditional" TDD naming convention (starting tests with "test") would also work fine.

With a TDDD approach, we'd write each of these tests one at a time, then evolve the database schema to fulfill the test-specified functionality. Consider the test shouldHavePositiveAccountBalance. If this is the first time we've considered the account-balance concept, we'd add the Account.Balance column. Depending on our database design standards, implementing such a data-oriented business rule could motivate us to also add a column constraint or an update trigger that checks the value of Account.Balance. Regardless of the implementation strategy, it's critical to validate that your database properly supports the business rule.

There's no magic to identifying a database test—the advice presented in this special issue's other articles, as well as in TDD books, is pertinent to TDDD.[1,2] The primary difference in TDDD is that we must recognize how important it is to validate both the functionality implemented by, and the data contained within, the database. A database is nothing special—like any other system component, it must be validated.

Database refactoring

Refactoring is a disciplined approach to restructuring code: you make small changes to improve the code's design, without changing its behavioral semantics—that is, you neither remove nor add behavior.[3,9] Refactoring lets you improve your code over time in a safe, evolutionary manner. Similarly, in database refactoring, you make a simple change to a database that improves its design, while retaining both its behavioral and informational semantics. Databases include structural aspects (such as table and view definitions), functional aspects (such as stored procedures and triggers), and informational aspects (the database data). There are many examples of database refactoring, including dropping a view that's no longer used, moving a column to a more appropriate table, renaming a column, splitting a multiple-use column, and parameterizing a database method to consolidate several similar ones.

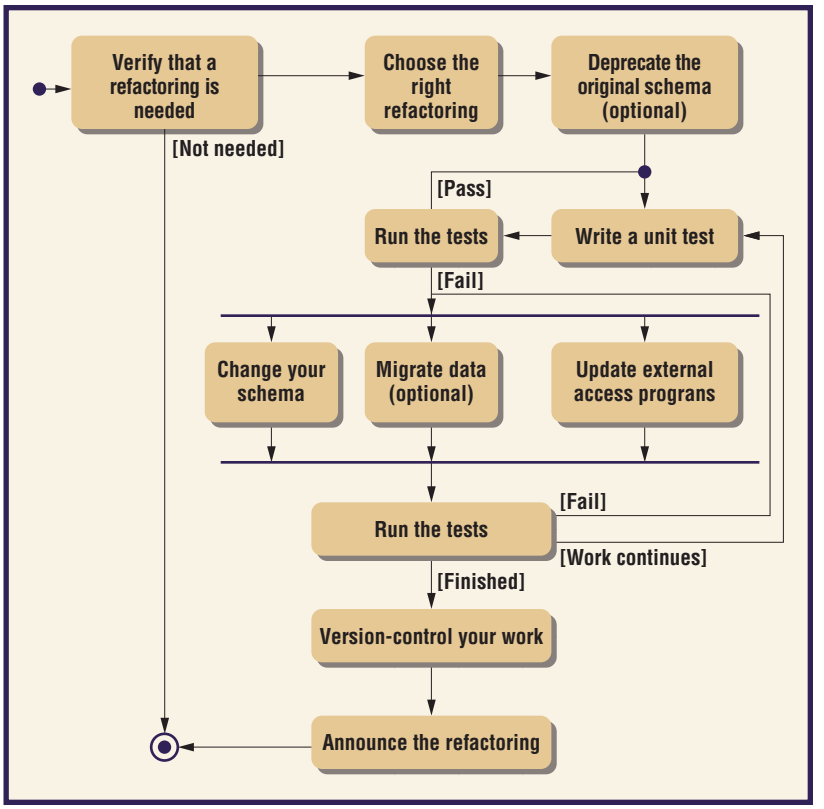

Figure 2. Database refactoring. Developers evolve the database schema in their own sandboxes before sharing the change with their teammates.

Conceptually, a database refactoring is more difficult than a code refactoring because you must preserve informational semantics along with behavioral semantics. Preserving informational semantics implies that if you change the values of a column's data, the change shouldn't materially affect that information's clients. For example, assume that a phone number is stored in character-based format. If you change the value from "(416) 555-1212" to "4165551212," you haven't materially changed the information's semantics, as long as all that column's clients can still work with the new format. In contrast, changing the value to "(416) 555-5555" clearly changes the informational semantics. Similarly, to preserve behavioral semantics, you must keep the same black-box functionality. The implication? You must rework any source code that works with changed aspects of your database schema to maintain the existing functionality.

Process overview. Figure 2 offers an overview of the database refactoring process using a UML activity diagram. The recommended ap-

Organizations must promote an effective means of communicating database changes among teams.

proach is to first implement the database refactoring in a separate developer's "sandbox" where you can perform the work in isolation before promoting it to shared environments, such as your team integration sandbox. Better yet: do this early work in pairs, where at least one person has agile database development skills and—more importantly—knowledge of the target database.[5]

The first step in refactoring is to verify that you need to refactor the schema and then, if so, to choose the right refactoring. For example, sometimes people suggest renaming a column, but don't realize that the existing name conforms to corporate naming conventions. Or, perhaps the real problem isn't the column's name, but rather that it's in the wrong table and therefore appears poorly named in that table's context. In other cases, refactoring is simply not cost effective.

A possible second step here is to deprecate the target schema's existing portion. This is an optional step; some production databases are accessed by dozens, if not hundreds, of systems running on different platforms, written in different languages, and released into production on different schedules. In this situation, you must support a transition window—which might last several years in some organizations—where the schema's existing and refactored versions run in parallel. In simpler situations, you can update and release both the database schema and accessing systems in parallel, so you don't need to support a transition window. The point is, you can do database refactoring in many situations, from small systems to highly complex ones.

In the next step, you write the code (or generate it using a database refactoring tool) to make the change to the database. For database refactorings that change the table structure, you'd create Data Definition Language (DDL) code to add the new schema and Data Manipulation Language (DML) code to manipulate the data. To keep the data synchronized, you need to add scaffolding code—typically a trigger, although in some instances an updatable view can work, too. If you're doing data-quality refactoring, you might only need to write DML code to address the quality issue. Similarly, if you're doing database-method refactorings, you might only have to rework stored procedure code. In parallel with this, you must test your work, taking a test-first approach to design and running your regression test suite whenever you integrate your system (more on these topics later). As you find bugs, you'll need to fix them, iterating back and forth between testing and coding as required.

Once you've completed the refactoring, you should put your work under configuration management (CM) control and announce the refactoring both to your team and the organization as a whole. Because databases are shared resources, database refactoring is often a challenge to organizations in that they must promote an effective means of communicating database changes among teams.

Structural refactoring example. To show how refactoring works, we'll work through a simple structural refactoring, moving a column from one table to another. Let's say our example environment has more than 100 systems accessing a mission-critical database that we can't afford to break. Figure 3a shows a portion of a bank database's original schema. The Customer table has a Balance column that belongs in the Account table. As a result of this design flaw, the company has been coding workarounds for years, making it difficult to support the business.

Figure 3b shows the transitional database schema, which we need because the situation is somewhat complex. We've selected a two-year transition window to ensure that the various development teams can refactor and test their code to work with the new schema. As the transitional image shows, we've added the new schema (the Account.Balance column) as well as scaffolding code—the SynchronizeCustomerBalance and SynchronizeAccountBalance triggers—to keep the two columns synchronized. We must assume that at any point during the transition, each of the systems accessing this database will access one but not the other column. In other words, the database must be responsible for keeping itself consistent.

As figure 3b shows, we've marked the scaffolding code and original schema (Customer.Balance) with their intended drop date. We're basically following the same strategy that Sun Microsystems uses to evolve the Java Development Kit (JDK)—add the new functionality and deprecate the old functionality to indicate that people should no longer use it. The difference here is that our drop date tells people how long they have to refactor their code. Naturally, we're not going to automati-

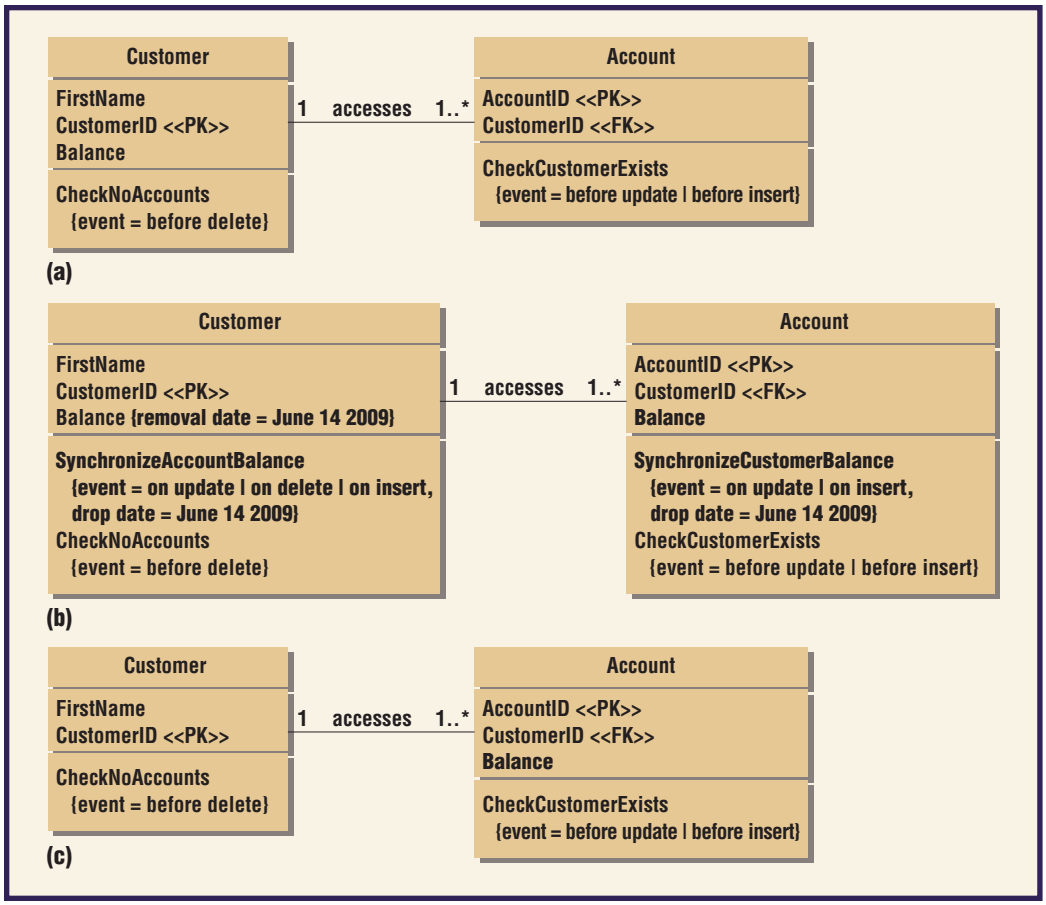

Figure 3. A simple structural refactoring. (a) The original database schema's Balance column is in the wrong table. (b) During the transition period, the database must support both the original and new schema versions. (c) Once developers successfully update and deploy the systems that accessed the original column, we remove the original schema and transition scaffolding.

cally drop the old schema and scaffolding code on that date; we'll ensure that developers have successfully updated and deployed the accessing applications first. Figure 3c shows the resulting database schema when the process is complete.

Continuous database integration

In continuous database integration, design team members regularly integrate changes to their own database instances, including structural, functional, and informational changes, as well as those related to automated system build and regression testing.[10] Ideally, they'd do this at least daily. In any case, whenever someone submits a database change, everyone else working with that database should download the change from the CM system and apply it to their database instance as soon as possible. Better yet, such changes should be automated—in the same way tools such as Build Forge or CruiseControl automatically download application code changes and rebuild your system on your machine.

There are several best practices for continuous database integration:

1. *Automate the build*. Your database build process should detect whether it needs to apply any changes to the schema, then apply those changes and run the test suite. The database build process is part of your overall system build process. Currently, database build tools are a relatively new concept, but open source developers are working on ActiveRecordMigration in Ruby on Rails and dbdeploy.
2. *Put everything under version control*. Data models, data scripts, test data, and similar artifacts are all important work products, and you should manage them appropriately. My philosophy is simple: if it's worth creating, it's worth putting under version control.
3. *Give developers their own database copies*. Developers should first work on application code and database schema changes in their own sandboxes.[11] Once they've fully tested changes and deemed them safe, they should apply the changes to the project integration environment. This reduces the chance that individual developers will break the build,

Developers embraced TDD years ago, yet it remains a relatively new concept for data professionals.

because they first do their work within their own independent environments.

4. *Automate database creation*. There are several reasons to create a database copy. You might need to rebuild it from scratch for comprehensive testing, new people might join your team and require proper environment setup, or you might need to install the database on a workstation to demo the system for someone. Creating a new database instance is something that you're likely to do often, so you should strive to automate the task. Similarly, you should be able to drop the database easily.
5. *Refactor each database individually*. Individual refactoring lets you apply schema changes one at a time, which in turn lets you start from any database schema version and evolve it forward to any other.
6. *Adopt a consistent identification strategy*. Common identification strategies include using an incremental integer number (1, 2, 3, 4, …), a date/time stamp, or a release number (v2.3.1.5). All three strategies are fine, but I prefer the integer strategy because it's simple. It's important that you be able to apply database changes in the proper order, particularly if you want to automate database creation and schema update.
7. *Bundle database changes when needed*. To implement a single feature, a developer might need to make several database schema changes. When deploying your system from your project integration environment to your preproduction testing environment, you'll likely need to bundle up several schema changes. For example, assume that the current database schema version is 1701, and you need to apply three new database refactorings (1702, 1703, and 1704). Once you've applied all three, in order, the database schema's new version would be 1704.
8. *Ensure that the database knows its version number*. Your database update scripts should help developers easily determine the current database version. For example, you should have an `UpdateSchema` script that takes as a parameter a database version—such as 1704 or 2007-06-14-23:45:00—and automatically runs the right change scripts to update to that version from the current one.

Continuous system integration includes the system's automated build and regression testing, and continuous database integration is one part of that overall effort. In other words, it isn't enough to just integrate your application code; you also need to integrate your database code.

Adopting TDDD

TDDD is an integrated part of the overall development process, not a standalone activity that data professionals perform in parallel with application TDD. Although from a technical viewpoint, TDDD is relatively straightforward, we must overcome several challenges to its wholesale adoption throughout the IT community.

Cultural divisions

The cultural divisions between the development and data communities are the most significant barrier to adoption.[5] The development community is clearly moving toward evolutionary development, while the data community seems firmly rooted in serial development. Developers embraced TDD years ago, yet it remains a relatively new concept for data professionals. A notable exception here is in the data warehousing/business intelligence community, where thought leaders promote evolutionary work processes (although practitioners, to their peril, often fail to heed this good advice). Unfortunately, this same community seems to believe that, because they work in an evolutionary manner, they don't need to test.[6] We saw this same attitude within the object development community in the early 1990s.

A dearth of tools

As I touched on earlier, our second challenge is the lack of tooling. We're starting to see interesting tools for database refactoring, testing, and continuous integration emerge in the marketplace. Because databases are shared resources, we also need tools that let us communicate schema changes among all interested teams.

We also need improvements to database modeling tools—we should, for example, be able to indicate a deprecated database element's drop date (as in figure 3b). Good database modeling tools should add value to design visualization. Better yet, we might one day see modeling tools that can define database tests. Finally, we need to rework databases themselves. For example, shouldn't the database throw a warning whenever someone accesses a deprecated database element, just as Sun's JDK does with deprecated Java code?

Lack of familiarity

In my experience, agile software developers seem to take to the TDDD idea rather quickly. Presumably, this is because they've already adopted TDD, so extending it to include database development makes sense to them. Traditional developers—and traditional data professionals, in particular—are often intrigued by the TDDD idea, but they struggle because it's so different from what they're used to. My hope is that we can overcome this challenge through training, education, mentoring, and a bit of patience.

Clearly, we can extend TDD into the database development realm, where implementing a more evolutionary methodology would better align the field with other modern methods and applications. All of the techniques required for TDDD adoption already exist and—more importantly—we're seeing nascent tool support for them. I expect that within the next two to three years, we'll see a surge in database development tools to support evolutionary, if not agile, database development techniques.

References

1. D. Astels, *Test Driven Development: A Practical Guide*, Prentice Hall, 2003.
2. K. Beck, *Test Driven Development: By Example*, Addison-Wesley, 2003.
3. M. Fowler, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley Longman, 1999.
4. *Behavior Driven Development*, Mar. 2007; www.behaviour-driven.org.
5. S.W. Ambler, *Agile Database Techniques: Effective Strategies for the Agile Software Developer*, John Wiley & Sons, 2003.
6. S.W. Ambler, *Database Regression Testing*, 2006; www.agiledata.org/essays/databaseTesting.html.
7. S.A. Becker and A. Berkemeyer, "The Application of a Software Testing Technique to Uncover Errors in Database Systems," *Proc. 20th Pacific Northwest Software Quality Conf.*, PNSQC/Pacific Agenda, 1999, pp. 173–183.
8. F. Swiderski and W. Snyder, *Threat Modeling*, Microsoft Press, 2004.
9. S.W. Ambler and P. Sadalage, *Refactoring Databases: Evolutionary Database Design*, Addison-Wesley, 2006.
10. M. Fowler and P. Sadalage, "Evolutionary Database Design," 2003; www.martinfowler.com/articles/evodb.html.
11. S.W. Ambler, "Development Sandboxes: An Agile Best Practice," 2002–2006; www.agiledata.org/essays/sandboxes.html.

About the Author

Scott W. Ambler is the practice leader for agile development with IBM Rational and a senior contributing editor with *Dr. Dobb's Journal*. His research interests include evolutionary database development and effective data management practices. He originated the Agile Modeling Method, the Agile Data Method, and the Enterprise Unified Process and is an active contributor to the Open Unified Process. He received his MIS in computer-supported cooperative work from the University of Toronto. He is coauthor of several books, including *Refactoring Databases* (Addison-Wesley, 2006). Contact him at scott_ambler@ca.ibm.com; www-306.ibm.com/software/rational/bios/ambler.html.