Forewords

A decade ago *refactoring* was a word only known to a few people, mostly in the Smalltalk community. It's been wonderful to watch more and more people learn how to use refactoring to modify working code in a disciplined and effective manner. As a result many people now see code refactoring as an essential part of software development.

I live in the world of enterprise applications, and a big part of enterprise application development is working with databases. In my original book on refactoring, I picked out databases as a major problem area in refactoring because refactoring databases introduces a new set of problems. These problems are exacerbated by the sad division that's developed in the enterprise software world where database professionals and software developers are separated by a wall of mutual incomprehension and contempt.

One of the things I like about Scott and Pramod is that, in different ways, they have both worked hard to try and cross this division. Scott's writings on databases have been a consistent attempt to bridge the gap, and his work on object-relational mapping has been a great influence on my own writings on enterprise application architecture. Pramod may be less known, but his impact has been just as great on me. When he started work on a project with me at ThoughtWorks we were told that refactoring of databases was impossible. Pramod rejected that notion, taking some sketchy ideas and turning them into a disciplined program that kept the database schema in constant, but controlled, motion. This freed up the application developers to use evolutionary design in the code, too. Pramod has since taken these techniques to many of our clients, spreading them around our ThoughtWorks colleagues and, at least for us, forever banishing databases from the list of roadblocks to continual design.

This book assembles the lessons of two people who have lived in the no-mans land between applications and data, and presents a guide on how to use refactoring techniques for databases. If you're familiar with refactoring, you'll notice that the major change is that you have to manage continual migration of the data itself, not just change the program and data structures. This book tells you how to do that, backed by the project experience (and scars) that these two have accumulated.

Much though I'm delighted by the appearance of this book, I also hope it's only a first step. After my refactoring book appeared I was delighted to find sophisticated tools appear that automated many refactoring tasks. I hope the

same thing happens with databases, and we begin to see vendors offer tools that make continual migrations of schema and data easier for everyone. Before that happens, this book will help you build your own processes and tools to help; afterward this book will have lasting value as a foundation for using such tools successfully.
—Martin Fowler, series editor; chief scientist, ThoughtWorks

In the years since I first began my career in software development, many aspects of the industry and technology have changed dramatically. What hasn't changed, however, is the fundamental nature of software development. It has never been hard to create software—just get a computer and start churning out code. But it was hard to create good software, and exponentially harder to create great software. This situation hasn't changed today. Today it is easier to create larger and more complex software systems by cobbling together parts from a variety of sources, software development tools have advanced in bounds, and we know a lot more about what works and doesn't work for the process of creating software. Yet most software is still brittle and struggling to achieve acceptable quality levels. Perhaps this is because we are creating larger and more complex systems, or perhaps it is because there are fundamental gaps in the techniques still used. I believe that software development today remains as challenging as ever because of a combination of these two factors. Fortunately, from time to time new technologies and techniques appear that can help. Among these advances, a rare few are have the power to improve greatly our ability to realize the potential envisioned at the start of most projects. The techniques involved in refactoring, along with their associate Agile methodologies, were one of these rare advances. The work contained in this book extends this base in a very important direction.

Refactoring is a controlled technique for safely improving the design of code without changing its behavioral semantics. Anyone can take a chance at improving code, but refactoring brings a discipline of safely making changes (with tests) and leveraging the knowledge accumulated by the software development community (through refactorings). Since Fowler's seminal book on the subject, refactoring has been widely applied, and tools assisting with detection of refactoring candidates and application of refactorings to code have driven widespread adoption. At the data tier of applications, however, refactoring has proven much more difficult to apply. Part of this problem is no doubt cultural, as this book shows, but also there has not been a clear process and set of refactorings applicable to the data tier. This is really unfortunate, since poor design at the data level almost always translates into problems at the higher tiers, typically causing a chain of bad designs in a futile effort to stabilize the shaky foundation. Further, the inability to evolve the data tier, whether due to denial or fear of change, hampers the ability of all that rests on it to deliver the best software possible. These problems are exactly what make this work so important: we now have a process and catalog for enabling iterative design improvements on in this vital area.

I am very excited to see the publication of this book, and hope that it drives the creation of tools to support the techniques it describes. The software industry is currently in an interesting stage, with the rise of open-source software and the collaborative vehicles it brings. Projects such as the Eclipse Data Tools Platform are natural collection areas for those interested in bringing database refactoring to life in tools. I hope the open-source community will work hard to realize this vision, because the potential payoff is great. Software development will move to the next level of maturity when database refactoring is as common and widely applied as general refactoring itself.
—John Graham, Eclipse Data Tools Platform, Project Management, committee chair, senior staff engineer, Sybase, Inc.

In many ways the data community has missed the entire agile software development revolution. While application developers have embraced refactoring, test-driven development, and other such techniques that encourage iteration as a productive and advantageous approach to software development, data professionals have largely ignored and even insulated themselves from these trends.

This became clear to me early in my career as an application developer at a large financial services institution. At that time I had a cubicle situated right between the development and database teams. What I quickly learned was that although they were only a few feet apart, the culture, practices, and processes of each group were significantly different. A customer request to the development team meant some refactoring, a code check-in, and aggressive acceptance testing. A similar request to the database team meant a formal change request processed through many levels of approval before even the modification of a schema could begin. The burden of the process constantly led to frustrations for both developers and customers but persisted because the database team knew no other way.

But they must learn another way if their businesses are to thrive in today's ever-evolving competitive landscape. The data community must somehow adopt the agile techniques of their developer counterparts.

Refactoring Databases is an invaluable resource that shows data professionals just how they can leap ahead and confidently, safely embrace change. Scott and Pramod show how the improvement in design that results from small, iterative refactorings allow the agile DBA to avoid the mistake of big upfront design and evolve the schema along with the application as they gradually gain a better understanding of customer requirements.

Make no mistake, refactoring databases is hard. Even a simple change like renaming a column cascades throughout a schema, to its objects, persistence frameworks, and application tier, making it seem to the DBA like a very inaccessible technique.

Refactoring Databases outlines a set of prescriptive practices that show the professional DBA exactly how to bring this agile method into the design and development of databases. Scott's and Pramod's attention to the minute details of what it takes to actually implement every database refactoring technique proves that it can be done and paves the way for its widespread adoption.

Thus, I propose a call to action for all data professionals. Read on, embrace change, and spread the word. Database refactoring is key to improving the data community's agility.

—Sachin Rekhi, program manager, Microsoft Corporation

In the world of system development, there are two distinct cultures: the world dominated by object-oriented (OO) developers who live and breathe Java and agile software development, and the relational database world populated by people who appreciate careful engineering and solid relational database design. These two groups speak different languages, attend different conferences, and rarely seem to be on speaking terms with each other. This schism is reflected within IT departments in many organizations. OO developers complain that DBAs are stodgy conservatives, unable to keep up with the rapid pace of change. Database professionals bemoan the idiocy of Java developers who do not have a clue what to do with a database.

Scott Ambler and Pramod Sadalage belong to that rare group of people who straddle both worlds. *Refactoring Databases: Evolutionary Database Design* is about database design written from the perspective of an OO architect. As a result, the book provides value to both OO developers and relational database professionals. It will help OO developers to apply agile code refactoring techniques to the database arena as well as give relational database professionals insight into how OO architects think.

This book includes numerous tips and techniques for improving the quality of database design. It explicitly focuses on how to handle real-world situations where the database already exists but is poorly designed, or when the initial database design failed to produce a good model.

The book succeeds on a number of different levels. First, it can be used as a tactical guide for developers in the trenches. It is also a thought-provoking treatise about how to merge OO and relational thinking. I wish more system architects echoed the sentiments of Ambler and Sadalage in recognizing that a database is more than just a place to put persistent copies of classes.

—Dr. Paul Dorsey, president, Dulcian, Inc.; president, New York Oracle Users Group; chairperson, J2EE SIG