Forewords

Building enterprise software is rarely easy. Although we have a plethora of tools and frameworks to make it easier, we still have to figure out how to use these tools well. There are lots of approaches you can take, but the trick is knowing which one to use in specific situations—hardly ever does one approach work in all cases. Over the last few years there's grown up a community of people looking to capture approaches to design enterprise applications and document them in the form of patterns (I keep an overview with links at http://martinfowler.com/articles/enterprisePatterns.html). People involved in this effort, such as me, try to find common approaches and describe how to do them well and when they are applicable. The resulting work is pretty wide ranging, and that can lead to too much choice for the reader.

When I started writing *Patterns of Enterprise Application Architecture* (Addison-Wesley, 2002), I looked for this kind of design advice in the Microsoft world. I struggled to find much of anything, but one rare book that tackled the territory was Jimmy's earlier book. I liked his informal writing style and eagerness to dig into concepts that many others skimmed over. So it's fitting that Jimmy decided to take many of the ideas from me and the others in the enterprise patterns community and show how you can apply them in writing .NET applications.

The focus of this enterprise patterns community is documenting good designs, but another thread runs through us. We are also big fans of agile methods, embracing techniques such as Test-Driven Development (TDD) and refactoring. So Jimmy also brought these ideas into this book. Many people think that pattern-people's focus on design and TDD's focus on evolution are at odds. The huge overlap between pattern-people and TDDers shows this isn't true, and Jimmy has weaved both of these threads into this book.

The result is a book about design in the .NET world, driven in an agile manner and infused with the products of the enterprise patterns community. It's a book that shows you how to begin applying such things as TDD, object-relational mapping, and domain-driven design to .NET projects. If you haven't yet come across these concepts, you'll find that this book is an introduction to techniques that many developers think are the key for future software development. If you are familiar with these ideas, the book will help you pass those ideas on to your colleagues.

Many people feel the Microsoft community has not been as good as others in propagating good design advice for enterprise applications. As the technology becomes more capable and sophisticated, it becomes more important to understand how to use it well. This book is a valuable step in advancing that understanding.

—Martin Fowler
http://martinfowler.com

The best way to learn how to do Domain-Driven Design (DDD) is to sit down next to a friendly, patient, experienced practitioner and work through problems together, step-by-step. That is what reading this book is like.

This book does not push a new grand scheme. It unaffectedly reports on one expert practitioner's use of and combination of the current practices he has been drawn to.

Jimmy Nilsson reiterates what many of us have been saying: that several currently trendy topics—specifically, DDD, Patterns of Enterprise Application Architecture (PoEAA), and Test-Driven Development (TDD)—are not alternatives to each other, but are mutually reinforcing elements of successful development.

Furthermore, all three of these are harder than they look at first. They require extensive knowledge over a wide range. This book does spend some time advocating these approaches, but mostly it focuses on the details of how to make them work.

Effective design is not just a bunch of techniques to be learned by rote; it is a way of thinking. As Jimmy dives into an example he gives us a little window into his mind. He not only shows his solution and explains it, he lets us see how he got there.

When I am designing something, dozens of considerations flit through my mind. If they are factors I've dealt with often, they pass so quickly I am barely conscious of them. If they are in areas where I have less confidence, I dwell on them more. I presume this is typical of designers, but it is difficult to communicate to another person. As Jimmy walks through his examples, it is as if he were slowing this process down to an observable pace. At every little juncture, three or four alternatives present themselves and get weighed and rejected in favor of the one he eventually chooses.

For example, we want model objects that are implemented free of entanglement with the persistence technology. So what are eight ways (eight!) that a

persistence framework can force you to contaminate the implementation of a domain object? What considerations would lead you to compromise on some of these points? What do the currently popular frameworks, including the .NET platform, impose?

Jimmy thinks pragmatically. He draws on his experience to make a design choice that will likely take him toward the goal, adhering to the deeper design principle, rather than the choice that looks the most like a textbook example. And all of his decisions are provisional.

The first design principle Jimmy holds in front of himself is the fundamental goal of DDD: a design that reflects deep understanding of the business problem at hand in a form that allows adaptation to new wrinkles. So why so much discussion of technical framework and architecture?

It is a common misperception, perhaps a natural one, that such a priority on the domain demands less technical talent and skill. Would that this were true. It would not be quite so difficult to become a competent domain designer. Ironically, to render clear and useful domain concepts in software, to keep them from being suffocated under technical clutter, requires particularly deft use of technology. My observation is that those with the greatest mastery of technology and architectural principles often know how to keep technology in its place and are among the most effective domain modelers.

I do not refer to the knowledge of every quirk of complex tools, but to the mastery of the sort of knowledge laid out in Martin Fowler's PoEAA, because naïve application of technology paradoxically makes that technology more intrusive into the application.

For many people this book will fill in gaps of how to implement expressive object models in practice. I picked up a number of useful ways of thinking through the application of technical frameworks, and I especially firmed up my understanding of some particulars of doing DDD in a .NET setting.

In addition to technical architecture, Jimmy spends a great deal of time on how to write tests. TDD complements DDD in a different way. In the absence of a focus on refining an ever more useful model, TDD is prone to fragmented applications, where a single-minded attack on one feature at a time leads to an unextendable system. A comprehensive test suite actually allows such a team to continue making progress longer than would be possible without it, but this is just the basest value of TDD.

At its best, the test suite is the laboratory for the domain model and a technical expression of the ubiquitous language. Tests of a particular style drive the modeling process forward and keep it focused. This book steps us through examples of developing such tests.

Jimmy Nilsson has a rare combination of self-confidence and humility, which I have observed to be characteristic of the best designers. We get a glimpse of how he got to his current understanding as he tells us what he used to believe and why that opinion changed, which helps to take the reader past the specifics of the techniques to the underlying principles. This humility makes him open to a wide range of influences, which gives us this fusion of ideas from different sources. He has tried a lot of things and has let his results and experience be his guide. His conclusions are not presented as revealed truth, but as his best understanding so far with an implicit recognition that we never have complete knowledge. All this makes the advice more useful to the reader. And this attitude, in itself, illustrates an important element of successful software development leadership.

—Eric Evans