

---

## Preface

In 2001, I was with Steve Daniel, a respected kayaker. We were at Bull Creek after torrential rains, staring at the rapid that we later named Bores. The left side of the rapid had water, but we wanted no part of it. We were here to run the V, a violent six-foot drop with undercut ledges on the right, a potential keeper hydraulic on the left, and a boiling tower of foam seven feet high in the middle. I didn't see a clean route. Steve favored staying right and cranking hard to the left after the drop to avoid the undercut ledge. I was leaning left, where I'd have a tricky setup, and where it would be tough to identify my line, but I felt that I could find it and jump over the hydraulic after making a dicey move at the top. We both dismissed the line in the middle. Neither of us thought we could keep our boats upright after running the drop and hitting the tower, which we called a haystack because of its shape. Neither of us was happy with our intended line, so we stood there and stared.

Then a funny thing happened. A little boy, maybe 11 years old, came over with a \$10 inflatable raft. He shoved it into the main current, and without paddle, life jacket, helmet, or any skill whatsoever, he jumped right in. He showed absolutely no fear. The stream predictably took him where most of the water was going, right into the "tower of power." The horizontal force of the water shot him through before the tower could budge him an inch. We both laughed hysterically. He should have been dead, but he made it—using an approach that more experienced kayakers would never have considered. We had our line.

In 2004, I went with 60 kids to Mexico to build houses for the poor. I'd done light construction of this kind before, and we'd always used portable cement mixers to do the foundation work. This group preferred another method. They'd pour all of the ingredients on the ground—cement, gravel, and sand. We'd mix up the piles with shovels, shape it like a volcano, and then pour water in the middle. The water would soak in, and we'd stir it up some more, and then shovel the fresh cement where we wanted it. The work was utterly exhausting. I later told the project director that he needed cement mixers; they would have saved a lot of backbreaking effort.



He asked me how to maintain the mixers. I didn't know. He asked where he might store them. I couldn't tell him. He then asked how he might transport them to the sites, because most groups tended to bring vans and not pickup trucks. I finally got the picture. He didn't use cement mixers because they were not the right tool for the job for remote sites in Mexico. They might save a half a day of construction effort, but they added just as much *or more* work to spare us that effort. The tradeoff, once fully understood, not only failed on a pure cost basis, but wouldn't work at all given the available resources.

In 2003, I worked with an IT department to simplify their design. They used a multi-layered EJB architecture because they believed that it would give them better scalability and protect their database integrity through sophisticated transactions. After much deliberation, we went from five logical tiers to two, completely removed the EJB session and entity beans, and deployed on Tomcat rather than Web Logic or JBoss. The new architecture was simpler, faster, and much more reliable.

It never ceases to amaze me how often the simplest answer turns out to be the best one. If you're like the average J2EE developer, you probably think you could use a little dose of simplicity about now. Java complexity is growing far beyond our capability to comprehend. XML is becoming much more sophisticated, and being pressed into service where simple parsed text would easily suffice. The EJB architecture is everywhere, whether it's warranted or not. Web services have grown from a simple idea and three major APIs to a mass of complex, overdone standards. I fear that they may also be forced into the mainstream. I call this tendency "the bloat."

Further, so many of us are trained to look for solutions that match our predetermined complicated notions that we don't recognize simple solutions unless they hit us in the face. As we stare down into the creek at the simple database problem, it *becomes* a blob of EJB. The interfaces *become* web services. This transformation happens to different developers at different times, but most enterprise developers eventually succumb. The solutions you see match the techniques you've learned, even if they're inappropriate; you've been trained to look beyond the simple solutions that are staring you in the face.

Java is in a dangerous place right now, because the real drivers, big vendors like Sun, BEA, Oracle, and IBM, are all motivated to build layer upon layer of sophisticated abstractions, to keep raising the bar and stay one step ahead of the competition. It's not enough to sell a plain servlet container anymore. Tomcat is already filling that niche. Many fear that JBoss will fill a similar role as a J2EE application server killer. So, the big boys innovate and build more complex, feature-rich servers. That's good—if the servers also deliver value that we, the customers, can leverage.

More and more, though, customers can't keep up. The new stuff is too hard. It forces us to know too much. A typical J2EE developer has to understand relational databases, the Java programming languages, EJB abstractions, JNDI for services, JTA for transactions, JCA and data sources for connection management, XML for data

representation, Struts for abstracting user interface MVC designs, and so on. Then, she's got to learn a whole set of design patterns to work around holes in the J2EE specification. To make things worse, she needs to keep an eye on the future and at least keep tabs on emerging technologies like Java Server Faces and web services that could explode at any moment.

To top it off, it appears that we are approaching an event horizon of sorts, where programmers are going to spend more time writing code to support their chosen frameworks than to solve their actual problems. It's just like with the cement mixers in Mexico: is it worth it to save yourself from spending time writing database transactions if you have to spend 50% of your time writing code supporting CMP?



Development processes as we know them are also growing out of control. No human with a traditional application budget can concentrate on delivering beautiful object interaction diagrams, class diagrams, and sophisticated use cases and still have enough time to create working code. We spend as much or more time on a project on artifacts that will never affect the program's performance, reliability, or stability. As requirements inevitably change due to increasing competitive pressures, these artifacts must also change, and we find that rather than aiding us, these artifacts turn into a ball, tied to a rope, with the other end forming an ever-tightening noose around our necks. There's a better way.

A few independent developers are trying to rethink enterprise development, and building tools that are more appropriate for the job. Gavin King, creator of Hibernate, is building a persistence framework that does its job with a minimal API and gets out of the way. Rod Johnson, creator of Spring, is building a container that's not invasive or heavy or complicated. They are not attempting to build on the increasingly precarious J2EE stack. They're digging through the muck to find a more solid foundation. In short, I'm not trying to start a revolution. It's already started.

That's the subject of this book. I recommend that we re-imagine what J2EE could and should be, and move back down to a base where we can apply real understanding and basic principles to build simpler applications. If you're staring at the rapids, looking at solutions you've been taught will work—but you still don't quite see how to get from point A to point B without real pain—it's time to rethink what you're doing. It's time to get beyond the orthodox approaches to software development and focus on making complex tasks simple. If you embrace the fundamental philosophies in this book, you'll spend more time on what's important. You'll build simpler solutions. When you're done, you'll find that your Java is better, faster, and lighter.

## Who Should Read This Book?

This book isn't for uber-programmers who already have all the answers. If you think that J2EE does everything that you need it to do and you can make it sing, this book is not for you. Believe me, there are already enough books out there for you.



If you've already cracked the code for simplicity and flexibility, I'm probably not going to teach you too much that's new. The frameworks I hold up as examples have been around for years—although incredibly, people are only now starting to write about them. The techniques I show will probably seem like common sense to you. I'll take your money, but you'll probably be left wanting when you're done.

This book is for the frustrated masses. It's intended for those intermediate-to-advanced developers with some real experience with Java who are looking for answers to the spiraling complexity. I'll introduce you to some ideas with power and bite. I know that you won't read a phone book. You haven't got time, so I'll keep it short. I'll try to show you techniques with real examples that will help you do things better than you did before.

## Organization of This Book

This book consists of 11 chapters and a Bibliography:

### Chapter 1, *The Inevitable Bloat*

This chapter highlights the problems inherent in the large-scale enterprise Java frameworks that most programmers work with today. I will cover not only what's wrong with these bloated frameworks, but how they got that way. Finally, I will lay out the core principles we'll cover in the rest of the book.

### Chapter 2, *Keep It Simple*

Many programmers fall into the same trap, believing that the more complicated their code, the better it must be. In fact, simplicity is the hallmark of a well-written application. This chapter defines the principle of simplicity, while drawing a distinction between simple and simplistic. I will also examine the tools and processes that help you achieve simplicity, like JUnit, Ant, and Agile development.

### Chapter 3, *Do One Thing, and Do It Well*

Programmers need to resist the urge to solve huge problems all at once. Code that tries to do too much is often too entangled to be readable, much less maintainable. This chapter traces the path from being presented with a problem, to truly understanding the problem and its requirements, to finally solving the problem through multiple, simple, and targeted layers. It finally describes how to design your layers to avoid unnecessary coupling.

### Chapter 4, *Strive for Transparency*

The programming community has tried for years to solve the problem of cross-cutting concerns. Generic services, like logging or database persistence, are necessary for most applications but have little to do with the actual problem domain. This chapter examines the methods for providing these kinds of services without unnecessarily affecting the code that solves your business problem—that is, how to solve them transparently. The two main methods we examine are reflection and code generation.







