1

Why "J2EE Without EJB"?

COPYRIGHTED MATERIAL

The traditional approach to J2EE architecture has often produced disappointing results: applications that are more complex than their business requirements warrant, show disappointing performance, are hard to test, and cost too much to develop and maintain.

It doesn't need to be so hard. There is a better way for most applications. In this book, we'll describe a simpler, yet powerful architectural approach, building on experience with J2EE and newer technologies such as Inversion of Control and AOP. Replacing EJB with lighter-weight, more flexible, infrastructure typically produces significant benefits. We and many others have used this approach in many production applications, with better results than are usually produced from traditional architectures.

Let's begin with a quick tour of the topics we'll examine in more detail in later chapters.

EJB Under the Spotlight

Like most of my colleagues, I was excited by the promise of EJB when it first appeared. I believed it was the way forward for enterprise middleware. However, I've since revised my opinions, in the light of my experiences and those of many colleagues.

Much has changed since the EJB specification was conceived:

- ❑ Parts of the specification's design now seem dated. For example, dynamic proxies, introduced in J2SE 1.3, call into question the container code generation envisaged in the EJB specification and the multiple source files needed to implement every EJB.
- ❑ The traditional link between EJB and RMI remoting is looking dated, because of the emergence of web services and the recognition that EJBs sometimes need only local interfaces. EJB is a heavyweight model for objects that don't need to offer remote access.

- This is a special case of the fact that basing typical applications around distributed business objects—the architectural choice EJB implements best—has proved problematic.
- Usage of EJB indicates its strengths and weaknesses. Most developers and architects have restricted their use of EJB to stateless session beans and (if asynchronous calls are needed) message-driven beans. The relative simplicity of the services provided by the EJB container to support SLSBs means that the overhead of an EJB container is hard to justify in such applications.
- Although EJB has been around for five years, and its use is a given in many J2EE projects, it has become apparent that its complexity means that many developers still don't understand it. For example, many developer candidates I interview can't correctly describe how EJB containers handle exceptions and how this relates to transaction management.
- The EJB specification is becoming more and more complex in an attempt to address problems with EJB. It's now so long and complex that few developers or architects will have time to read and understand it. With specifications, as with applications, the need for continual workarounds and constantly growing complexity suggests fundamental problems.
- The complexity of EJB means that productivity in EJB applications is relatively poor. A number of tools try to address this, from "Enterprise" IDEs to XDoclet and other code generation tools, but the complexity still lurks under the surface and imposes ongoing costs.
- Rigorous unit testing and **test driven development** have become increasingly, and deservedly, popular. It's become clear that applications making heavy use of EJB are hard to test. Developing EJB applications **test first** requires a lot of fancy footwork; essentially, minimization of the dependence of application code on the EJB container.
- The emergence of **Aspect Oriented Programming** (AOP) points the way to more powerful—yet potentially simpler—approaches to the middleware problems addressed by EJB. AOP can be viewed in part as a more general application of the central EJB concepts, although of course it's much more than a potential replacement to EJB.
- Source level metadata attributes, as used in .NET, suggest a superior alternative in many cases to the verbose XML-based deployment descriptors used since EJB 1.1. EJB 3.0 looks like it's heading down that road as well, but it's a way off and will carry a lot of baggage.

Experience has also shown EJB to incur greater cost and deliver fewer benefits than were initially predicted. Developers have encountered intractable problems that weren't apparent when EJB first appeared. Experience has shown that EJB fails to deliver in several areas:

- It doesn't necessarily reduce complexity. It *introduces* a lot of complexity.
- The entity bean experiment for persistence has largely failed.
- Applications using EJB tend to be less portable between application servers than applications using other J2EE technologies, such as servlets.
- Despite the promises that EJB would prove the key to scalability, EJB systems often perform poorly and don't necessarily scale up well. Although statistics are hard to come by, anecdotal evidence suggests that the overhead of excessive use of EJB necessitates re-architecture or causes outright failure in a significant number of projects.
- EJB can make simple things hard. For example, the Singleton design pattern (or alternatives) is hard to implement in EJB.

All of these issues suggest that it's wise to analyze exactly what the value proposition is before using EJB. I hope to equip you with the tools to do this effectively and dispassionately.

In Chapter 5, we'll talk more about EJB and its problems. In the meantime, let's look at where J2EE is today, where I feel it's going, and how this book will help you deliver real solutions on time and budget.

What's Left of J2EE?

You may be wondering, "What's left of J2EE without EJB?"

The answer is: a great deal. J2EE is much more than EJB. Many J2EE developers believe otherwise, and will tell you so when they see this book on your desk, but a dispassionate analysis of what EJB does, and what J2EE does overall, shows that EJB is only a part of a much bigger and more important picture.

J2EE is essentially about standardizing a range of enterprise services, such as naming and directory services (JNDI), transaction management offering a standard API potentially spanning disparate transactional resources (JTS and JTA), connection to legacy systems (JCA), resource pooling, and thread management. The true power of J2EE lies in these services, and this standardization has done great service to the industry.

EJB, on the other hand, is merely one way of leveraging those valuable services, through a particular component model.

We can still access JNDI, JTA, JCA, resource pooling, and other core J2EE services without using EJB. We can do this by writing code that uses them directly (not as hair-raising as it may seem) or—better—using proven libraries and frameworks that abstract their use without imposing the complexity of EJB.

Only a few EJB container services are unique to EJB, and there are good alternatives to those. For example:

- ❑ **Entity beans** are the only dedicated data access components in J2EE. However, they're also the most questionable part of J2EE, and there are much better non-J2EE alternatives, such as Hibernate and JDO. In some applications, JDBC is a better option.
- ❑ **Container Managed Transactions (CMT):** EJBs are the only part of J2EE to enjoy declarative transaction management. This is a valuable service, but as we'll see in Chapters 8 and 9 we can also achieve declarative transaction management using AOP. CMT is a relatively thin layer over the underlying J2EE JTA service. It would be hard (and foolhardy to attempt) to replace an application server's global transaction management, but it's not so hard to access it to develop an alternative form of CMT.
- ❑ **Thread pooling for business objects:** We usually don't need this if we're supporting only web clients (or web services clients going through a servlet engine), because a web container provides thread pooling and there's no need to duplicate it in the business object tier. We do need thread pooling to support remote clients over RMI/IIOP, one case in which EJB remains a good, simple technology choice.

- (Related) **Thread management for business objects:** the ability to implement EJBs as though they are single-threaded. In my experience this is overrated for stateless service objects (the most useful kinds of EJB). EJB can't eliminate all threading complexity anyway, as problems can remain with objects used by EJB facades. There are good alternatives to EJB thread management, discussed in Chapter 12.

Only in the area of remoting is EJB the only way to implement such functionality in standard J2EE. As we'll see, only in RMI/IIOP remoting is EJB clearly an outstanding remoting technology; there are better alternatives for web services remoting.

There's a strong argument that EJB attempts to address a lot of issues it shouldn't. Take O/R mapping. This is a complex problem to which EJB provides a complex yet under-specified solution (entity beans) that simply ignores some of the central problems, such as mapping objects with an inheritance hierarchy to relational database tables. It would have been better for the designers of the EJB specification to leave this problem to those with much more experience of the issues around object persistence.

J2EE is much more than EJB. Using J2EE without EJB, we don't have to reinvent the wheel. We don't need to reimplement J2EE services, just consider alternative ways of tapping into them.

J2EE at a Crossroads

J2EE is at a fascinating point in its evolution. In many respects it's a great success. It has succeeded in bringing standardization where none existed; it has introduced a welcome openness into enterprise software. It has achieved fantastic industry and developer buy-in.

On the other hand, I feel it has come up short on a number of measures. J2EE applications are usually too expensive to develop. J2EE application projects are at least as prone to failure as pre-J2EE projects. (Which means that the failure rate is unacceptably high; developing software is far too hit-and-miss an affair.) In the areas where J2EE has failed, EJB has usually played a significant part.

J2EE has significant issues with ease of development. As I've said, J2EE applications tend to be unnecessarily complex. This is especially true of J2EE web applications, which, like the Sun Java Pet Store, are often absurdly over-engineered.

J2EE is still a relatively young technology. It's not surprising that it's imperfect. It's time to take stock of where it's worked, and where it hasn't worked so well, so that we can eliminate the negatives and enjoy the positives. Because J2EE contains a lot, this essentially means identifying the subset of J2EE that delivers most value, along with some supplementary infrastructure we need to harness it most effectively.

There is a growing movement in the J2EE community toward simpler solutions and less use of EJB. My previous book, *Expert One-on-One J2EE Design and Development* (2002), was a step in the growth of that movement, but was part of a broader trend. I believe this book represents the next step in defining and popularizing such solutions, but it's important to note that I'm by no means alone. Fellow pioneers include Rickard Oberg and Jon Tirsen (of Nanning Aspects), who have helped to demonstrate the power and simplicity of AOP-based solutions. The revisions in the second edition of *Core J2EE Patterns* suggest that even Sun is not immune; there is a new and welcome emphasis on use of plain Java objects.

Some of the problems with J2EE and EJB relate to its specification-driven origins. History shows that the most successful standards *evolve*, rather than are created by a committee. The danger of a "specification-first" approach is shown by the example of the OMG and CORBA. The OMG was founded to create a distributed object standard. Over 300 vendors signed up; the result was the slow production of complex specifications that never achieved widespread acceptance. As is often the case with committees, usability by developers was barely a consideration; the result was a horribly complex programming model.

J2EE is partly an evolution of existing middleware, because many of the problems it addresses were familiar when it was conceived in the late 1990s. For example, stateless session EJBs are merely an EJB take on a component type of proven value. Service objects with declarative transaction management existed in Microsoft Transaction Server, for example, before the EJB 1.0 specification. It's arguable that where J2EE has tried to innovate, through specifications being developed *before* any real applications using them, it has often failed. *Stateful* session beans, for example, were a new and unproven component type introduced in EJB. Five years on, they remain largely unproven. The tricky issue of state replication remains problematic, and most architects avoid stateful session beans if at all possible.

I suspect that the specification-driven nature of J2EE is going to change in practice, and that this is a good thing. I don't envisage J2EE descending into anarchy, but I do think that developers aren't automatically going to adopt each new feature of the J2EE specifications without considering alternative technologies, especially from the open source community.

This book represents part of that critical process: the recognition that end users of the technology—application developers, project managers responsible for development projects, and those who end up using applications—are the most important stakeholders, and that the reality at the coal face of application development isn't always obvious to those on specification committees.

The Way Forward

This book is not primarily about questioning EJB, but about mapping a path forward. This includes architectural principles, working code, and practical advice you can use in your projects today.

The way forward that this book proposes is to focus on core values—I'll call them *themes*—that help lead to project success, and to examine architectures and implementation choices that express them.

Themes

The central themes of this book are:

- ❑ Simplicity
- ❑ Productivity
- ❑ The fundamental importance of object orientation

- The primacy of business requirements
- The importance of empirical process
- The importance of testability

Let's briefly discuss these.

Simplicity

There *are* simple problems, and architecture and implementation should always be as simple as possible.

As I've already mentioned, J2EE projects are often over-engineered, partly because of an assumption that J2EE applications are necessarily complex. This isn't always true. Areas in which J2EE architects often assume requirements to be more complex than they are include:

- **Database distribution.** Many applications use only a single database. This means that they don't need JTA, two-phase commit, or XA transactions. All these high-end features incur cost in performance and complexity.
- **Assumption of multiple client types.** An application may have a requirement for a web interface. But J2EE architects are also likely to assume that it must also be able to support remote Swing clients. The assumption that all J2EE applications should be able to support multiple client types is deeply ingrained. (Indeed, I only realized that it's actually not that common when a reviewer on one of my previous books pointed it out, prompting me to reflect on real projects I'd been involved with.)

J2EE orthodoxy in both cases is that the user isn't allowed to have such simple requirements. We J2EE architects, in our wisdom, know that the client's business will get complicated enough to justify the complexity we're going to make him pay for up front.

There are two problems here. Firstly, including this complexity isn't our choice as architects and developers, as we don't write the checks. Secondly, even if the more complex requirements do ultimately emerge, how do we know it's cheaper to factor them in from day one? It may well be cheaper to wait until these requirements come up. It's quite likely they won't; if they do, we can cross that bridge when we come to it. For example, the eventual remote client choice might be C# or VB.NET on Windows; an EJB-based remoting architecture might not be the best choice to support this. One of the key lessons of XP is that it is often more cost-effective, and more likely to produce a quality implementation, not to try to solve all conceivable problems up front.

We should minimize complexity up front to what's necessary to support actual (and reasonably foreseeable) requirements. However, it is necessary to design a simple architecture that allows for architectural refactoring to scale up. Refactoring an architecture is not as simple as refactoring code; clearly we don't want to have to scrap much of our code to meet additional requirements.

It's important to have a simple architecture that can scale up. It's not so good to have a complex architecture, such as an EJB architecture, that can't scale down to meet simple requirements.

In my experience, the keys to enabling architectural refactoring in J2EE projects are:

- ❑ To follow good OO design practice and **program to interfaces rather than classes**. This is a fundamental teaching of the classic *Design Patterns* text, and too often neglected.
- ❑ To conceal technologies such as EJB behind plain Java interfaces.

The architectural approach and frameworks discussed in this book make it easy to practice these principles.

Productivity

Productivity is an immensely important consideration, too often ignored in J2EE.

J2EE has a poor productivity record. J2EE developers typically spend too much of their time wrestling with API and deployment complexity when they should really be concentrating on business logic. The relative proportions are better than in the CORBA days, but still not good enough. Much of this incidental time is associated with EJB.

The approaches advocated in this book are highly productive, partly because they're comparatively simple and dispense with a lot of unnecessary crud.

OO

Surely, since Java is a rather good OO language, object orientation is a given for J2EE applications? While it should be, in fact many J2EE applications are really "EJB" or "J2EE" applications more than OO applications. Many common J2EE practices and patterns sacrifice object orientation too easily.

OO design is more important than specific technologies, such as J2EE. We should try to avoid letting our technology choices, such as J2EE, constrain our ability to use true OO design.

Let's consider two examples of how many J2EE applications sacrifice OO:

- ❑ **The use of EJBs with remote interfaces to distribute business objects.** Designing an application in terms of distributed business objects with remote interfaces can deliver a fatal blow to OO. Components with remote interfaces must offer interfaces designed to avoid the need for "chatty" calling for performance reasons, and raise the tricky problem of marshaling input and output parameters in terms of **transfer** or **value** objects. There *are* applications that must offer distributed business objects, but most shouldn't and are much better off staying away from this particular minefield. The use of distributed objects is not unique to EJB: EJB wasn't the first distributed object technology and won't be the last. The reason that this problem is linked to EJB is that distributing components is the one thing that EJB makes easy: arguably, too easy.
- ❑ The assumption that persistent objects should contain no behavior. This has long been an article of faith among J2EE developers. I used to subscribe to it myself—this is one area in which my thinking has changed somewhat since I published *Expert One-on-One J2EE*. In fact, this assumption in a J2EE context owes more to the severe limitations of entity beans as a technology than sound design principles. Objects that expose only getters and setters (for example, to expose

persistent data) are not really objects. A true object should encapsulate behavior acting upon its state. The use of entity beans encouraged developers to accept this limitation as the norm, because business logic in entity beans couldn't easily be tested, and was fatally tied to a particular persistence strategy. (For example, if a significant amount of business logic is coded in entity beans, and it becomes apparent that the only way to achieve adequate performance is to perform relational data access using SQL and JDBC, a major refactoring exercise is required.) A better solution is to use a transparent persistence technology, such as JDO or Hibernate, which allows persistent objects to be true objects, with little or no dependence on how they're actually persisted.

Whenever you find yourself writing an object that's not really an object—such as an object that contains only methods exposing its data—think hard about why you're doing this, and whether there's a better alternative.

And there *is* real value in practicing OO. Applied effectively, OO can deliver very high code reuse and elegant design.

In this book we won't forget the value of OO. We'll try to show how J2EE applications can be object oriented.

Primacy of Requirements

It should be obvious, but application architecture should be driven by business requirements, not target technology.

Unfortunately this is not always the case in J2EE. Developers often assume phantom requirements, such as:

- Providing support for multiple databases in the one application, already discussed
- The ability to port to another application server at zero cost
- The ability to port to another database easily
- Supporting multiple client types

All of these are *potential* business requirements, but whether they are *actual* requirements should be evaluated for each application. Developers with other technologies, such as .NET, often don't need to worry about such phantom requirements, meaning that sometimes J2EE developers are expending effort only because of their technology choice, not because of client requirements.

It's a great bonus of J2EE that it makes a whole range of things possible that aren't possible with other technologies, but it's also a potential danger when it causes us to forget that all these things have a cost and that we shouldn't incur each cost without good reason.

Empirical Process

My wife is a doctor. In recent years, medical practice has been influenced by the rise of **Evidence-Based Medicine**: a model in which treatment decisions are strongly influenced by the evidence from medical research. Treatment decisions are typically made on the basis of the known outcomes of various choices for the patient's condition.

While the influence of EBM on medical practice may not be wholly positive, this kind of empirical approach definitely has lessons for software development.

The IT industry is strongly driven by fashion and emotion. I seldom pass a day without hearing someone repeat an orthodox opinion they can't justify (such as "EJB applications are inherently more scalable than applications without EJB"), or repeat a religious conviction that's backed up by no real evidence (such as ".NET is not a credible enterprise platform").

Too often this means that people architect enterprise systems based on their own particular bigotry, and without hard evidence backing up the approaches they propose. They just know which way is best.

One of the best comments I've heard about this situation is "Ask the computer" (from fellow author Randy Stafford). We should always "ask the computer" what it thinks of our proposed architecture before we spend too much money and effort on it.

This approach is formalized in iterative methodologies in the form of an **executable architecture** (RUP) or **vertical slice** or **spike solution** (XP). In each case the aim is to minimize risk by building an end-to-end executable as soon as possible. In the case of RUP this aims to tackle the trickiest architectural issues as the best means to mitigate risks; in XP it is driven by key *user stories*. In all cases, it must be relevant to actual requirements. An agile development process can naturally tend toward the creation of a vertical slice, making a distinct activity unnecessary. Once the vertical slice is built, it can be used as the basis for metrics that serve to validate the architectural choices. Important metrics include:

- **Performance.** Can non-functional requirements be met? This is the most likely sticking point in J2EE, where many projects that don't do an early vertical slice end up grappling with intractable performance problems at the last minute.
- **How hard was it to get here?** Were the development time and costs involved proportionate to the requirements implemented? Is the complexity of the result proportionate to the requirements? Was magic involved or is the process repeatable?
- **Maintainability.** How hard is it to add features to the vertical slice? How quickly can a new developer understand the application's architecture and implementation and become productive?

Unfortunately many projects don't do this kind of risk mitigation. Worse still, many proposed J2EE architectures are specification-driven or vendor-driven rather than proven in production, so we shouldn't be too trusting.

Don't trust us, or anyone else. Build a vertical slice of your application, and apply metrics based on your critical requirements. "Ask the computer" which architecture meets your requirements.

Your requirements just may be unique. However, the architecture described in this book has been used in many successful projects, and proven to deliver very good results.

Testability

Test first development has become much more popular in the last few years, and usually produces impressive results. Writing effective unit tests for an application isn't just a question of putting in the time and effort; it can be severely constrained by high-level architecture. In this book I'll stress architectures that make it easy to write effective unit tests. This is one of the biggest frustrations with EJB. Due to its heavy dependence on the EJB container, business logic coded in EJB is very hard to test.

Code that is hard to test is usually also hard to re-engineer, reuse in different contexts, and refactor. Testability is an essential characteristic of *agile* projects. The approach to J2EE architecture expounded in this book is ideal for agile projects, and we'll talk about agility throughout.

We'll discuss testability in detail in Chapter 14.

Lightweight Frameworks and Containers

Applications need infrastructure. Rejecting the use of EJB doesn't mean rejecting the need for an infrastructural solution to many of the concerns EJB addresses. We don't want to return to the pre-EJB era of the late 1990s when the first complex Java enterprise applications were developed, each with its own resource pooling, thread management, service lookup, data access layer, and other infrastructure.

This book will look at existing frameworks that provide such alternatives. We believe that such alternative infrastructure is essential to successful J2EE projects. Thus describing the capabilities and use of lightweight frameworks is a central part of this book.

In 2003 there seemed to be a flowering of such "lightweight" frameworks, which provide management of business objects and enterprise services without the heavyweight infrastructure of EJB. This reflects the movement toward simpler, lighter J2EE solutions that I've already mentioned. Many of these frameworks, such as Spring, PicoContainer, and Nanning Aspects, come out of Java's thriving open source community. I'll say more about this in Chapter 5.

There are also commercial products, such as The Mind Electric's GLUE web services product, that provide lighter-weight solutions than EJB in some of the areas that EJB addresses, such as remoting.

The Spring Framework

We couldn't have written this book without working code that illustrates the approaches we advocate, and that has been proved through use in production applications.

The Spring Framework (`www.springframework.org`) is a popular open source product dedicated to providing simple, effective solutions for common problems in J2EE. This project developed in early 2003 from the source code published with *Expert One-on-One J2EE Design and Development*, which was unusual in presenting an integrated and ambitious application framework. Spring has since acquired a thriving developer and user community and become much more full-featured and bulletproof than anything I could have developed alone. (My co-author, Juergen Hoeller, has made an invaluable contribution to making this happen as co-lead of Spring.) The basic framework design predated even *Expert One-on-One J2EE*, being the result of my experience over several commercial projects.

Spring wasn't conceived as an alternative to EJB, but it provides powerful, tested, implementations of features, such as declarative transaction management for plain Java objects, that enable users to dispense with EJB in many projects.

Spring is not the only project in this space. Little of what we say about design is unique to Spring. For example, we advocate the use of AOP to solve some common enterprise problems; there are several other open source AOP frameworks.

The use of a production-quality open source code base is a real differentiator of this book. Most J2EE books come with code examples. Unfortunately the value of the code is often limited, as it represents a simplification of the problems necessary for the pedagogical purpose, which rapidly becomes a problem when trying to apply the code in real applications. Thus illustrating points using a specific, proven framework—even if not all readers will work with that framework—is preferable to presenting unrealistically simple solutions that can be wholly contained in the text.

Spring has excellent support for architectural refactoring. For example, it's possible to add EJBs with local interfaces or AOP to business objects without modifying a line of calling code, so long as we follow the essential discipline of programming to interfaces rather than classes.

Should We Ever Use EJB?

There is a place for EJB. This book describes approaches that offer a simpler, more productive, alternative to EJB for the great majority of J2EE applications. However, we don't claim that this approach is the best solution to all problems.

In *Expert One-on-One J2EE Design and Development*, I repeatedly invoked the "Pareto Principle." Often referred to as the 80/20 (or 90/10) rule, this states that a small number of causes (10–20%) are responsible for most (80–90%) of the effect. In the architectural context, this emphasizes the value of finding good solutions to common problems, rather than always living with the complexity of solutions to rarer, more complex problems.

EJB stands on the wrong side of the Pareto Principle. It imposes undue complexity on the majority of cases to support the special requirements of the minority. For example, perhaps 10% of applications need distributed business objects; EJB is an infrastructure closely associated with distribution. EJB 2.1 and earlier entity beans are designed to be independent of the data store; the great majority of J2EE applications use relational databases, and gain no benefit from this. (While it offers a portability between data stores that's of more theoretical interest than practical value, it doesn't shine with object databases either. They are best accessed using their own rich APIs, or using a solution such as JDO.)

EJB remains the best choice for applications that genuinely need object distribution, especially if they are implemented wholly in Java or need to use IIOP as the communication protocol. This type of application is rarer than might be imagined.

EJB is also a pretty good solution for applications that are heavily based around messaging, as message driven beans are relatively simple and effective components.

One of the best examples of an application type to which EJB may add real value is financial middleware. Financial applications can involve processing that is so costly in time and computing power that the cost of remote invocation is (atypically) less than the cost of processing. For such applications, object distribution makes sense, and EJB is a good way of implementing it. Financial middleware is also heavily message-oriented, and suited to use of MDBs.

I believe that such applications are part of the 10%.

Of course there may be strong *political*, rather than technical, reasons that dictate the use of EJB. This is outside the scope of this book. In my experience, political battles can be much harder to win than technical battles, and you'll need Machiavelli rather than me as your guide.

I believe that EJB is a declining technology, and within three years it will be relegated to legacy status, despite attempts to make it more relevant with EJB 3.0. But this book focuses on what you can do right now to build enterprise applications. So if you have a requirement that's currently best addressed by EJB, I would advise you to use EJB—for now.

Summary

This chapter has provided a quick tour of the topics we'll discuss in the rest of this book.

Since the inception of J2EE, EJB has been promoted as the core of the J2EE platform. We believe that this is a misconception. EJB has a place, but most applications do better without it. J2EE is much more than EJB; EJB is just one way to tap into J2EE services. Thus, dispensing with EJB does not mean abandoning J2EE.

We believe that lightweight containers, such as the Spring Framework, provide a better way of structuring application code and a better model for leveraging J2EE and other services. We'll look at this architectural approach in detail throughout this book.

We believe that business requirements and basic OO values, not implementation technology, should drive projects and architectures.

In the next chapter, we'll look at the goals identified here in more detail, before moving on to application architecture and specific technologies in the remainder of the book.